# Technical Charter (the “Charter”)
## for
# Accord Project a Series of LF Projects, LLC
# Adopted 14 May 2019

This charter (the “Charter”) sets forth the responsibilities and procedures for technical contribution to, and oversight of, the Accord Project community, which has been established as Accord Project a Series of LF Projects, LLC (the “Project”). LF Projects, LLC (“LF Projects”) is a Delaware series limited liability company. All contributors (including committers, maintainers, and other technical positions) and other participants in the Project (collectively, “Collaborators”) must comply with the terms of this Charter.

## 1. Mission and Scope of the Project
a. The mission of the Project is to develop open source software tools for legally enforceable smart contracts.

b. The scope of the Project includes collaborative development under the Project License (as defined herein) supporting the mission, including documentation, testing, integration and the creation of other artifacts that aid the development, deployment, operation or adoption of the open source project.

## 2. Technical Steering Committee
a. The Technical Steering Committee (the “TSC”) will be responsible for all technical oversight of the open source Project.

b. The TSC voting members are initially the Project’s Committers. At the inception of the project, the Committers of the Project will be as set forth within the “CONTRIBUTING” file within the Project’s code repository. The TSC may choose an alternative approach for determining the voting members of the TSC, and any such alternative approach will be documented in the CONTRIBUTING file. Any meetings of the Technical Steering Committee are intended to be open to the public, and can be conducted electronically, via teleconference, or in person.

c. TSC projects generally will involve Contributors and Committers. The TSC may adopt or modify roles so long as the roles are documented in the CONTRIBUTING file. Unless otherwise documented:

i. Contributors include anyone in the technical community that contributes code, documentation, or other technical artifacts to the Project;

ii. Committers are Contributors who have earned the ability to modify (“commit”) source code, documentation or other technical artifacts in a project’s repository; and

iii. A Contributor may become a Committer by a majority approval of the existing Committers. A Committer may be removed by a majority approval of the other existing Committers.

d. Participation in the Project through becoming a Contributor and Committer is open to anyone so long as they abide by the terms of this Charter.

e. The TSC may (1) establish work flow procedures for the submission, approval, and closure/archiving of projects, (2) set requirements for the promotion of Contributors to Committer status, as applicable, and (3) amend, adjust, refine and/or eliminate the roles of Contributors, and Committers, and create new roles, and publicly document any TSC roles, as it sees fit.

f. The TSC may elect a TSC Chair, who will preside over meetings of the TSC and will serve until their resignation or replacement by the TSC.

g. Responsibilities: The TSC will be responsible for all aspects of oversight relating to the Project, which may include:

i. coordinating the technical direction of the Project;

ii. approving, modifying and disbanding Working Groups and sub-projects;

iii. creating sub-committees to focus on cross-project issues and requirements;

iv. appointing representatives to work with other open source or open standards communities;

v. establishing meeting procedures, community norms, workflows, issuing releases, and security issue reporting policies;

vi. approving and implementing policies and processes for contributing (to be published in the CONTRIBUTING file) and coordinating with the series manager of the Project (as provided for in the Series Agreement, the "Series Manager") to resolve matters or concerns that may arise as set forth in Section 7 of this Charter;

vii. discussions, seeking consensus, and where necessary, voting on technical matters relating to the code base that affect multiple projects; and

viii.coordinating any marketing, events, or communications regarding the Project.

## 3. TSC Voting

a. While the Project aims to operate as a consensus-based community, if any TSC decision requires a vote to move the Project forward, the voting members of the TSC will vote on a one vote per voting member basis.

b. Quorum for TSC meetings requires at least fifty percent of all voting members of the TSC to be present. The TSC may continue to meet if quorum is not met but will be prevented from making any decisions at the meeting.

c. Except as provided in Section 7.c. and 8.a, decisions by vote at a meeting require a majority vote of those in attendance, provided quorum is met. Decisions made by electronic vote without a meeting require a majority vote of all voting members of the TSC.

d. In the event a vote cannot be resolved by the TSC, any voting member of the TSC may refer the matter to the Series Manager for assistance in reaching a resolution.

## 4. Compliance with Policies

a. This Charter is subject to the Series Agreement for the Project and the Operating Agreement of LF Projects. Contributors will comply with the policies of LF Projects as may be adopted and amended by LF Projects, including, without limitation the policies listed at https://lfprojects.org/policies/.

b. The TSC may adopt a code of conduct ("CoC") for the Project, which is subject to approval by the Series Manager. In the event that a Project-specific CoC has not been approved, the LF Projects Code of Conduct listed at https://lfprojects.org/policies will apply for all Collaborators in the Project.

c. When amending or adopting any policy applicable to the Project, LF Projects will publish such policy, as to be amended or adopted, on its web site at least 30 days prior to such policy taking effect; provided, however, that in the case of any amendment of the Trademark Policy or Terms of Use of LF Projects, any such amendment is effective upon publication on LF Project’s web site.

d. All Collaborators must allow open participation from any individual or organization meeting the requirements for contributing under this Charter and any policies adopted for all Collaborators by the TSC, regardless of competitive interests. Put another way, the Project community must not seek to exclude any participant based on any criteria, requirement, or reason other than those that are reasonable and applied on a non-discriminatory basis to all Collaborators in the Project community.

e. The Project will operate in a transparent, open, collaborative, and ethical manner at all times. The output of all Project discussions, proposals, timelines, decisions, and status should be made open and easily visible to all. Any potential violations of this requirement should be reported immediately to the Series Manager.

## 5. Community Assets
a. LF Projects will hold title to all trade or service marks used by the Project (“Project Trademarks”), whether based on common law or registered rights. Project Trademarks will be transferred and assigned to LF Projects to hold on behalf of the Project. Any use of any Project Trademarks by Collaborators in the Project will be in accordance with the license from LF Projects and inure to the benefit of LF Projects.

b. The Project will, as permitted and in accordance with such license from LF Projects, develop and own all Project GitHub and social media accounts, and domain name registrations created by the Project community.

c. Under no circumstances will LF Projects be expected or required to undertake any action on behalf of the Project that is inconsistent with the tax-exempt status or purpose, as applicable, of LFP, Inc. or LF Projects, LLC.

## 6. General Rules and Operations.
a. The Project will:

i. engage in the work of the Project in a professional manner consistent with maintaining a cohesive community, while also maintaining the goodwill and esteem of LF Projects, LFP, Inc. and other partner organizations in the open source community; and

ii. respect the rights of all trademark owners, including any branding and trademark usage guidelines.

## 7. Intellectual Property Policy
a. Collaborators acknowledge that the copyright in all new contributions will be retained by the copyright holder as independent works of authorship and that no contributor or copyright holder will be required to assign copyrights to the Project.

b. Except as described in Section 7.c., all contributions to the Project are subject to the following:

i. All new inbound code contributions to the Project must be made using the Apache License, Version 2.0, available at https://www.apache.org/licenses/LICENSE-2.0 (the “Project License”).

ii. All new inbound code contributions must also be accompanied by a Developer Certificate of Origin (http://developercertificate.org) sign-off in the source code system that is submitted through a TSC-approved contribution process which will bind the authorized contributor and, if not self-employed, their employer to the applicable license;

iii. All outbound code will be made available under the Project License.

iv. Documentation will be received and made available by the Project under the Creative Commons Attribution 4.0 International License (available at http://creativecommons.org/licenses/by/4.0/).

v. To the extent a contribution includes or consists of data, any rights in such data shall be made available under the CDLA-Permissive 1.0 License.

vi. The Project may seek to integrate and contribute back to other open source projects (“Upstream Projects”). In such cases, the Project will conform to all license requirements of the Upstream Projects, including dependencies, leveraged by the Project. Upstream Project code contributions not stored within the Project’s main code repository will comply with the contribution process and license terms for the applicable Upstream Project.

c. The TSC may approve the use of an alternative license or licenses for inbound or outbound contributions on an exception basis. To request an exception, please describe the contribution, the alternative open source license(s), and the justification for using an alternative open source license for the Project. License exceptions must be approved by a two-thirds vote of the entire TSC.

d. Contributed files should contain license information, such as SPDX short form identifiers, indicating the open source license or licenses pertaining to the file.

## 8. Amendments
a. This charter may be amended by a two-thirds vote of the entire TSC and is subject to approval by LF Projects.

# Accord Project Contribution Guide

We'd love for you to contribute to our source code and to make Accord Project technology even better than it is today! Here are the guidelines we'd like you to follow:

* [Code of Conduct][contribute.coc]
* [Questions and Problems][contribute.question]
* [Issues and Bugs][contribute.issue]
* [Feature Requests][contribute.feature]
* [Improving Documentation][contribute.docs]
* [Updating Documentation][contribute.updating]
* [Issue Submission Guidelines][contribute.submit]
* [Pull Request Submission Guidelines][contribute.submitpr]
* [Technical Documentation][contribute.techdocs]

## <a name="coc"></a> Code of Conduct

Help us keep the Accord Project open and inclusive. Please read and follow our [Code of Conduct][coc].

## <a name="requests"></a> Questions, Bugs, Features

### <a name="question"></a> Got a Question or Problem?

We want to keep GitHub Issues dedicated to bug reports and feature requests. Any general support questions are better directed towards a dedicated support platform, the best being the [Accord Project Discord Channel][apdiscord].

### <a name="issue"></a> Found an Issue or Bug?

If you find a bug in the source code or documentation, you can help us by submitting an issue to our respective GitHub repository. Even better, you can submit a Pull Request with a fix.

Please see the [Submission Guidelines][contribute.submit] below.

### <a name="feature"></a> Missing a Feature?

You can request a new feature by submitting an issue to the respective GitHub repository.

If you would like to implement a new feature then consider what kind of change it is:

* **Major Changes** that you wish to contribute to the project should be discussed first in a GitHub Issue that clearly outlines the changes and benefits of the feature.
* **Small Changes** can directly be crafted and submitted to the respective GitHub repository as a Pull Request. See the section about [Pull Request Submission Guidelines][contribute.submitpr], and for detailed information read the [core development documentation][developers].

### <a name="docs"></a> Want a Doc Fix?

Should you have a suggestion for the documentation, you can open an issue and outline the problem or improvement you have - however, creating the doc fix yourself is much better!

If you want to help improve the docs, it's a good idea to let others know what you're working on to minimize duplication of effort. Create a new issue (or comment on a related existing one) to let others know what you're working on.

For large fixes, please build and test the documentation before submitting the PR to be sure you haven't accidentally introduced any layout or formatting issues. You should also make sure that your commit message follows the **[Commit Message Guidelines][developers.commits]**.

## <a name="submit"></a> Issue Submission Guidelines
Before you submit your issue search the archive, maybe your question was already answered.

If your issue appears to be a bug, and hasn't been reported, open a new issue. Help us to maximize the effort we can spend fixing issues and adding new features, by not reporting duplicate issues.

The "new issue" form contains a number of prompts that you should fill out to make it easier to understand and categorize the issue.

If you get help, help others. Good karma rulez!

## <a name="submit-pr"></a> Pull Request Submission Guidelines

Before you submit your pull request consider the following guidelines:

~~~~markdown
* First check whether there is an open Issue for what you will be working on. If
there is not, open one up by going through [these guidelines][contribute.submit],
including links to _related_ Issues found for context.
* Search for an open or closed Pull Request that relates to your submission. You
don't want to duplicate effort, and you also want to include links to _related_
Pull Requests found for context.
* Create the [development environment][developers.setup]
* Make your changes in a new git branch: techdocs

  ```text
    git checkout -b name/issue-tracker/short-description master
  ```

  Name can be initials or GitHub username. An example of this could be:

  ```text
    git checkout -b irmerk/i75/readme-typos master
  ```

* Create your patch commit, **including appropriate test cases**.
* Follow our [Coding Rules][developers.rules].
* Ensure you provide a DCO sign-off for your commits using the `--signoff` option
of git commit. For more information see [how this works][dcohow].
* If the changes affect public APIs, change or add relevant [documentation]
[developers.documentation].
* Run the [unit test suite][developers.unit-tests], and ensure that all tests pass.

* Commit your changes using a descriptive commit message that follows our [commit
message conventions][developers.commits]. Adherence to the [commit message
conventions][developers.commits] is required, because release notes are
automatically generated from these messages.

  ```text
    git commit -a --signoff
  ```

  Note: the optional commit `-a` command line option will automatically "add" and
"rm" edited files.

* Before creating the Pull Request, ensure your branch sits on top of master (as
opposed to branch off a branch). This ensures the reviewer will need only minimal
effort to integrate your work by fast-fowarding master:

  ```text
    git rebase upstream/master
  ```

* Last step before creating the Pull Request, package and run all tests a last
time:

  ```text
    npm run test
  ```

* Push your branch to GitHub:

  ```text
    git push origin name/issue-tracker/short-description
  ```
~~~~

* In GitHub, send a pull request to `<REPOSITORY>:master` by following our [pull request conventions][developers.pullrequest]. This will trigger the check of the [Contributor License Agreement][contribute.cla] and the Travis integration.
* If you find that the Travis integration has failed, look into the logs on Travis to find out if your changes caused test failures, the commit message was malformed, etc. If you find that the tests failed or times out for unrelated reasons, you can ping a team member so that the build can be restarted.
* If we suggest changes, then:
  * Make the required updates.
  * Re-run the test suite to ensure tests are still passing.
  * Commit your changes to your branch (e.g. `name/issue-tracker/short-description`).
  * Push the changes to your GitHub repository (this will update your Pull Request).

    You can also amend the initial commits and force push them to the branch.

    ```text
    git rebase master -i
    git push origin name/issue-tracker/short-description -f
    ```

    This is generally easier to follow, but separate commits are useful if the Pull Request contains iterations that might be interesting to see side-by-side.

That's it! Thank you for your contribution!

#### After your pull request is merged

After your pull request is merged, you can safely delete your branch and pull the changes from the main (`upstream`) repository:

* Delete the remote branch on GitHub either through the GitHub web UI or your local shell as follows:

  ```text
    git push origin --delete name/issue-tracker/short-description
  ```

* Check out the master branch:

  ```text
    git checkout master -f
  ```

* Delete the local branch:

  ```text
    git branch -D name/issue-tracker/short-description
  ```

* Update your master with the latest upstream version:

  ```text
    git checkout master
    git fetch --all --prune
    git rebase upstream/master
    git push origin master
  ```

~~~~
  ```

## License <a name="license"></a>

Accord Project source code files are made available under the [Apache License,
Version 2.0][apache].

Accord Project documentation files are made available under the [Creative Commons
Attribution 4.0 International License][creativecommons] (CC-BY-4.0).

[coc]: https://lfprojects.org/policies/code-of-conduct/
[apdiscord]: https://discord.gg/Zm99SKhhtA

[contribute.coc]: CONTRIBUTING.md#coc
[contribute.cla]: CONTRIBUTING.md#cla
[contribute.question]: CONTRIBUTING.md#question
[contribute.issue]: CONTRIBUTING.md#issue
[contribute.feature]: CONTRIBUTING.md#feature
[contribute.docs]: CONTRIBUTING.md#docs
[contribute.updating]: CONTRIBUTING.md#updating
[contribute.submit]: CONTRIBUTING.md#submit
[contribute.submitpr]: CONTRIBUTING.md#submit-pr
[contribute.techdocs]: CONTRIBUTING.md#techdocs


[developers]: DEVELOPERS.md
[developers.setup]: DEVELOPERS.md#-development-setup
[developers.commits]: DEVELOPERS.md#commits
[developers.rules]: DEVELOPERS.md#rules
[developers.documentation]: DEVELOPERS.md#documentation
[developers.unit-tests]: DEVELOPERS.md#-running-the-unit-test-suite

[dcohow]: https://github.com/probot/dco#how-it-works

[apache]: https://github.com/accordproject/techdocs/blob/master/LICENSE
[creativecommons]: http://creativecommons.org/licenses/by/4.0/

--------------------------------------------------------------------------------
# Cicero Development Guide

##    Accord Project Development Guide
We'd love for you to help develop improvements to Cicero technology! Please refer
to the [Accord Project Development guidelines][apdev] we'd like you to follow.

## Installation

To build the documentation locally:
```
cd ./website
npm install
npm run start
```
If you want to re-generate the JSDoc API:
```
npm run build:api
```

## Creating a new version of the documentation
~~~~

~~~~
```
cd ./website
npm install
npm run version `0.23.0`
```
If you want to re-generate the JSDoc API:
```
npm run build:api
```

[apdev]: https://github.com/accordproject/techdocs/blob/master/DEVELOPERS.md

--------------------------------------------------------------------------------
<p align="center">
  <a href="https://www.accordproject.org/">
    <img src="assets/APLogo.png" alt="Accord Project Logo" width="400">
  </a>
</p>

<p align="center">
  <a href="./LICENSE">
    <img src="https://img.shields.io/github/license/accordproject/cicero?
color=bright-green" alt="GitHub license">
  </a>
</p>

## Introduction

Technical Documentation for all Accord Project code. This site uses [Docusaurus]
(https://docusaurus.io) to generate static HTML.

The site is hosted at: https://docs.accordproject.org

Accord Project is an open source, non-profit, initiative working to transform
contract management and contract automation by digitizing contracts. Accord Project
operates under the umbrella of the [Linux Foundation][linuxfound]. The technical
charter for the Accord Project can be found [here][charter].

## Learn More About Accord Project

### [Overview][apmain]

### [Documentation][apdoc]

## Contributing

The Accord Project technology is being developed as open source. All the software
packages are being actively maintained on GitHub and we encourage organizations and
individuals to contribute requirements, documentation, issues, new templates, and
code.

Find out what’s coming on our [blog][apblog].

Join the Accord Project Technology Working Group [Discord server][apslack] to get
involved!

For code contributions, read our [CONTRIBUTING guide][contributing] and information
for [DEVELOPERS][developers].
~~~~

### README Badge

Using Accord Project? Add a README badge to let everyone know: [![accord project](https://img.shields.io/badge/powered%20by-accord%20project-19C6C8.svg)](https://www.accordproject.org/)

```
[![accord project](https://img.shields.io/badge/powered%20by-accord%20project-19C6C8.svg)](https://www.accordproject.org/)
```

## License <a name="license"></a>

Accord Project source code files are made available under the [Apache License, Version 2.0][apache].
Accord Project documentation files are made available under the [Creative Commons Attribution 4.0 International License][creativecommons] (CC-BY-4.0).

[linuxfound]: https://www.linuxfoundation.org
[charter]: https://github.com/accordproject/techdocs/blob/master/CHARTER.md
[apmain]: https://accordproject.org/
[apblog]: https://medium.com/@accordhq
[apdoc]: https://docs.accordproject.org/
[apslack]: https://discord.gg/Zm99SKhhtA

[contributing]:
https://github.com/accordproject/techdocs/blob/master/CONTRIBUTING.md
[developers]: https://github.com/accordproject/techdocs/blob/master/DEVELOPERS.md

[apache]: https://github.com/accordproject/techdocs/blob/master/LICENSE
[creativecommons]: http://creativecommons.org/licenses/by/4.0/

<!--- Provide a formatted commit message describing this PR in the Title above -->
<!--- See our DEVELOPERS guide below: -->
<!--- https://github.com/accordproject/techdocs/blob/master/DEVELOPERS.md#commit-message-format -->
# Closes #<CORRESPONDING ISSUE NUMBER>
<!--- Provide an overall summary of the pull request -->

### Changes
<!--- More detailed and granular description of changes -->
<!--- These should likely be gathered from commit message summaries -->
- <ONE>
- <TWO>

### Flags
<!--- Provide context or concerns a reviewer should be aware of -->
- <ONE>
- <TWO>

### Screenshots or Video
<!--- Provide an easily accessible demonstration of the changes, if applicable -->

### Related Issues

```
- Issue #<NUMBER>
- Pull Request #<NUMBER>

### Author Checklist
- [ ] Ensure you provide a [DCO sign-off](https://github.com/probot/dco#how-it-
works) for your commits using the `--signoff` option of git commit.
- [ ] Vital features and changes captured in unit and/or integration tests
- [ ] Commits messages follow [AP
format](https://github.com/accordproject/techdocs/blob/master/DEVELOPERS.md#commit-
message-format)
- [ ] Extend the documentation, if necessary
- [ ] Merging to `master` from `fork:branchname`
- [ ] Manual accessibility test performed
    - [ ] Keyboard-only access, including forms
    - [ ] Contrast at least WCAG Level A
    - [ ] Appropriate labels, alt text, and instructions

--------------------------------------------------------------------------------
---
name: Bug Report
about: Create a report to help us improve
title: ''
labels: ''
assignees: ''

---

<!--- Provide a general summary of the issue in the Title above -->
# Bug Report
<!--- Provide an expanded summary of the issue -->

## Expected Behavior
<!--- Tell us what should happen -->

## Current Behavior
<!--- Tell us what happens instead of the expected behavior -->

## Possible Solution
<!--- Not obligatory, but suggest a fix/reason for the bug, -->

## Steps to Reproduce
<!--- Provide a link to a live example, or an unambiguous set of steps to -->
<!--- reproduce this bug. Include code to reproduce, if relevant -->
1.
2.
3.
4.

## Context (Environment)
<!--- How has this issue affected you? What are you trying to accomplish? -->
<!--- Providing context helps us come up with a solution that is most useful in the
real world -->
### Desktop
 - OS: [e.g. macOS]
 - Browser: [e.g. Chrome, Safari]
 - Version: [e.g. 0.22.15]

## Detailed Description
<!--- Provide a detailed description of the change or addition you are proposing --
```

```
>

## Possible Implementation
<!--- Not obligatory, but suggest an idea for implementing addition or change -->

--------------------------------------------------------------------------------
---
name: Custom Issue
about: For miscellaneous, such as questions, discussions, etc.
title: ''
labels: ''
assignees: ''

---

<!--- Provide a general summary of the issue in the Title above -->
# Discussion
<!--- Provide an expanded summary of the issue -->

## Context
<!--- Providing context helps us come to the discussion informed -->

## Detailed Description
<!--- Provide any further details -->

--------------------------------------------------------------------------------
---
name: Feature Request
about: Suggest an idea for this project
title: ''
labels: ''
assignees: ''

---

<!--- Provide a general summary of the feature in the Title above -->
# Feature Request
<!--- Provide an expanded summary of the feature -->

## Use Case
<!--- Tell us what feature we should support and what should happen -->

## Possible Solution
<!--- Not obligatory, but suggest an implementation -->

## Context
<!--- How has this issue affected you? What are you trying to accomplish? -->
<!--- Providing context helps us come up with a solution that is most useful in the
real world -->

## Detailed Description
<!--- Provide a detailed description of the change or addition you are proposing --
>

--------------------------------------------------------------------------------
---
id: accordproject-faq
title: FAQ
---
```

## Accord Project Frequently asked Questions

### What is a "Smart Contract" in the Accord Project?

A “smart” legal contract is a legally binding agreement that is digital and able to connect its terms and the performance of its obligations to external sources of data and software systems. The benefit is to enable a wide variety of efficiencies, automation, and real time visibility for lawyers, businesses, nonprofits, and government. The potential applications of smart legal contracts are limitless. Although the operation of smart legal contracts may be enhanced by using blockchain technology, a blockchain is not necessary, smart legal contracts can operate using traditional software systems without blockchain. A central goal of Accord Project technology is to be blockchain agnostic.

A smart legal contract consists of natural language text with certain parts (e.g. clauses, sections) of the agreement constructed as machine executable components. The libraries provided by the Accord Project enable a document to be:

* Structured as machine readable data objects; and
* Executed on, or integrated with, external systems (e.g. to initiate a payment or update an invoice)

While the Accord Project technology is targeted at the development of smart legal contracts, the open source codebase may also be used to develop other forms of machine-readable and executable documentation.

### How is an Accord Project "Smart Contract" different from "Smart Contracts" on the blockchain?

Accord Project Smart legal contracts should not be confused with so-called blockchain “smart contracts”, which are scripts that necesarily operate on a blockchain. On the blockchain a smart contract is often written in a specific language like solidity that executes and operates on the blockchain. It lives in a closed world. An Accord Project Smart Contract contains text based template that integrates with a data model and the Ergo language. The three components are integrated into a whole. Using Ergo an Accord Project Smart contract can communicate with other systems, it can send and receive data, it can perform calculations and it can interact with a blockchain.

### What benefits do Smart Legal Contracts provide?

Contractual agreements sit at the heart of any organization, governing relationships with employees, shareholders, customers, suppliers, financiers, and more. Yet contracts today are not capable of being efficiently managed as the valuable assets they are. Currently contracts exist as static text documents stored in cloud storage services, dated contract management systems, or even email inboxes. Often these documents are Word files or PDFs that can only be interpreted by humans. A smart legal contract, by contrast, can be interpreted by machines.

Smart Legal Contracts can be easily searched, analyzed, queried, and understood. By associating a data model to a contract, it is possible to extract a host of valuable data about a contract or draft a series of contracts from existing data points (i.e., variables and their values).

The data model is used to ensure that all of the necessary data is present in the contract, and that this data is valid. In addition, it provides the necessary structure to enable contracts to "come alive" by adding executable logic.

The result is a contract that is:

* Searchable
* Analyzable
* Real-time
* Integrated

Consequently, contracts are transformed from business liabilities in constant need of management to assets capable of providing real business intelligence and value. A Smart Contract contains a data model so that the data is part of the contract and not something held in an external system. The logical operations of the contract are also part of the contract. The contract can update itself and react to the outside world. Rather than being stored in filing cabinet it is a living breathing process.

### What is the Accord Project and what is its purpose?

The Accord Project is a non-profit, member-driven organization that builds open source code and documentation for smart legal contracts for use by transactional attorneys, business and finance professionals, and other contract users. Open source means that anyone can use and contribute to the code and documentation and use it in their own software applications and systems free of charge.

The purpose of the Accord Project is to establish and maintain a common and consistent legal and technical foundation for smart legal contracts. The Accord Project is organized into working groups focused on various use cases for Smart Contracts. The specific working groups are assisted by the Technology Working Group, which builds the underlying open source code and specifications to codify the knowledge of the transactional working groups. More details about the internal governance of the Accord Project are available [here](https://github.com/accordproject/governance).

### How can I get involved?

The Accord Project Community is developing several working groups focusing on different applications of smart contracts. The working groups have frequent calls and use the Accord Project’s Discord group chat application (join by clicking [here](https://discord.com/invite/Zm99SKhhtA)) for discussion. The dates, dial-in instructions, and agendas for the working groups are all listed in the Project’s public calendar and typically also in working group’s respective Discord channels.

A primary purpose of the working groups is to develop a universally accessible and widely used open source library of modular, smart legal contracts, smart templates and models that reflect input from the community. Smart legal contract templates are built according to the Project’s [Cicero Specification](https://github.com/accordproject/cicero).

Members can provide feedback into the templates and models relevant to a particular working group. You can immediately start contributing smart legal contract templates and models by using the Accord Project’s [Template Studio](https://studio.accordproject.org/).

The Accord Project has developed an easy-to-use programming language for building and executing smart legal contracts called Ergo. The goals of Ergo are to be accessible and usable by non-technical professionals, portable across, and compatible with, a variety of environments such as SaaS platforms and different blockchains, and meeting security, safety, and other requirements.

You can use the Accord Project’s [Template Studio](https://studio.accordproject.org/) to create and test your smart legal contracts.

---
id: accordproject-slc
title: Smart Legal Contracts

A Smart Legal Contract is a human-readable _and_ machine-readable agreement that is digital, consisting of natural language and computable components.

The human-readable nature of the document ensures that signatories, lawyers, contracting parties and others are able to understand the contract.

The machine-readable nature of the document enables it to be interpreted and executed by computers, making the document "smart".

Contracts drafted with Accord Project can contain both traditional and machine-readable clauses. For example, a Smart Legal Contract may include a smart payment clause while all of the other provisions of the contract (Definitions, Jurisdiction clause, Force Majeure clause, ...) are being documented solely in regular natural language text.

A Smart Legal Contract is a general term to refer to two compatible, architectural forms of contract:
- Machine-Readable Contracts, which tie legal text to data
- Machine-Executable Contracts, which tie legal text to data and executable code

### Machine-Readable Contracts

By combining Text and a data, a clause or contract becomes machine-readable.

For instance, the clause below for a [fixed rate loan](https://templates.accordproject.org/fixed-interests-static@0.2.0.html) includes natural language text coupled with variables. Together, these variables refer to some data for the clause and correspond to the 'deal points':

```tem
## Fixed rate loan

This is a *fixed interest* loan to the amount of {{loanAmount}}
at the yearly interest rate of {{rate}}%
with a loan term of {{loanDuration}},
and monthly payments of {{monthlyPayment}}.
```

To make sense of the data, a _Data Model_, expressed in the Concerto schema language, defines the variables for the template and their associated Data Types:

```ergo
  o Double loanAmount     // loanAmount is a floating-point number
  o Double rate           // rate is a floating-point number
  o Integer loanDuration  // loanDuration is an integer
  o Double monthlyPayment // monthlyPayment is a floating-point number
```

The Data Types allow a computer to validate values inserted into each of the `{{variable}}` placeholders (e.g., `2.5` is a valid `{{rate}}` but `January` isn't). In other words, the Data Model lets a computer make sense of the structure of (and data in) the clause. To learn more about Data Types see [Concerto Modeling](https://concerto.accordproject.org/docs/intro).

The clause data (the 'deal points') can then be capture as a machine-readable representation:

```js
{
  "$class": "org.accordproject.interests.TemplateModel",
  "clauseId": "cec0a194-cd45-42f7-ab3e-7a673978602a",
  "loanAmount": 100000.0,
  "rate": 2.5,
  "loanDuration": 15
  "monthlyPayment": 667.0
}
```

The values entered into the template text are associated with the name of the variable e.g. `{{rate}} = 2.5%`. This provides the structure for understanding the clause and its contents.

### Machine-Executable Contracts

By adding Logic to a machine-readable clause or contract in the form of expressions - much like in a spreadsheet - the contract is able to execute operations based upon data included in the contract.

For instance, the clause below is a variant of the earlier [fixed rate loan](https://templates.accordproject.org/fixed-interests@0.2.0.html). While it is consistent with the previous one, the `{{monthlyPayment}}` variable is replaced with an [Ergo](logic-ergo.md) expression `monthlyPaymentFormula(loanAmount,rate,loanDuration)` which calculates the monthly interest rate based upon the values of the other variables: `{{loanAmount}}`, `{{rate}}`, and `{{loanDuration}}`. To learn more about contract Logic see [Ergo Logic](logic-ergo.md).

```tem
## Fixed rate loan

This is a *fixed interest* loan to the amount of {{loanAmount}}
at a yearly interest rate of {{rate}}%
with a loan term of {{loanDuration}},
and monthly payments of {{% monthlyPaymentFormula(loanAmount,rate,loanDuration)
%}}.
```

This is a simple example of the benefits of Machine-Executable contract, here adding logic to ensure that the value of the `{{monthlyPayment}}` in the text is always consistent with the other variables in the clause. In this example, we display the contract text using the underlying [Markup](markup-preliminaries.md) format, instead of the rich-text output that would be found in [editor tools](started-resources.md#ecosystem-tools) and PDF outputs.

More complex examples, (e.g., how to add post-signature logic which responds to data sent to the contract or which triggers operations on external systems) can be found in the rest of this documentation.

---
id: accordproject-template
title: Accord Project Templates

An Accord Project template ties legal text to computer code. It is composed of three elements:

- **Template Text**: the natural language of the template
- **Template Model**: the data model that backs the template, acting as a bridge between the text and the logic
- **Template Logic**: the executable business logic for the template

![Template](assets/020/template.png)

The three components (Text - Model - Logic) can also be intuitively understood as a **progression**, from _human-readable_ legal text to _machine-readable_ and _machine-executable_. When combined these three elements allow templates to be edited, validated, and then executed on any computer platform (on your own machine, on a Cloud platform, on Blockchain, etc).

> We use the computing term 'executed' here, which means run by a computer. This is distinct from the legal term 'executed', which usually refers to the process of signing an agreement.

## Template Text

![Template Text](assets/020/template_text.png)

The template text is the natural language of the clause or contract. It can include markup to indicate [variables](ref-glossary.md#variable) for that template.

The following shows the text of an **Acceptance of Delivery** clause.

```tem
## Acceptance of Delivery.

{{shipper}} will be deemed to have completed its delivery obligations
if in {{receiver}}'s opinion, the {{deliverable}} satisfies the
Acceptance Criteria, and {{receiver}} notifies {{shipper}} in writing
that it is accepting the {{deliverable}}.

## Inspection and Notice.

{{receiver}} will have {{businessDays}} Business Days to inspect and
evaluate the {{deliverable}} on the delivery date before notifying
{{shipper}} that it is either accepting or rejecting the
{{deliverable}}.

## Acceptance Criteria.

The 'Acceptance Criteria' are the specifications the {{deliverable}}
must meet for the {{shipper}} to comply with its requirements and
obligations under this agreement, detailed in {{attachment}}, attached
to this agreement.
```

The text is written in plain English, with variables between `{{` and `}}`.
Variables allows template to be used in different agreements by replacing them with
different values.

For instance, the following show the same **Acceptance of Delivery** clause where
the `shipper` is `"Party A"`, the `receiver` is `"Party B"`, the `deliverable` is
`"Widgets"`, etc.

```md
## Acceptance of Delivery.

"Party A" will be deemed to have completed its delivery obligations
if in "Party B"'s opinion, the "Widgets" satisfies the
Acceptance Criteria, and "Party B" notifies "Party A" in writing
that it is accepting the "Widgets".

## Inspection and Notice.

"Party B" will have 10 Business Days to inspect and
evaluate the "Widgets" on the delivery date before notifying
"Party A" that it is either accepting or rejecting the
"Widgets".

## Acceptance Criteria.

The "Acceptance Criteria" are the specifications the "Widgets"
must meet for the "Party A" to comply with its requirements and
obligations under this agreement, detailed in "Attachment X", attached
to this agreement.
```

### TemplateMark

TemplateMark is the markup format in which the text for Accord Project templates is
written. It defines notations (such as the `{{` and `}}` notation for variables
used in the **Acceptance of Delivery** clause) which allows a computer to make
sense of your templates.

It also provides the ability to specify the document structure (e.g., headings,
lists), to highlight certain terms (e.g., in bold or italics), to indicate text
which is optional in the agreement, and more.

_More information about the Accord Project markup can be found in the [Markdown
Text](markup-templatemark.md) Section of this documentation._

## Template Model

![Template Model](assets/020/template_model.png)

Unlike a standard document template (e.g., in Word or pdf), Accord Project
templates associate a _model_ to the natural language text. The model acts as a
bridge between the text and logic; it gives the users an overview of the
components, as well as the types of different components.

The model categorizes variables (is it a number, a monetary amount, a date, a
reference to a business or organization, etc.). This is crucial as it allows the
computer to make sense of the information contained in the template.

The following shows the model for the **Acceptance of Delivery** clause.

```ergo
/* The template model */
asset AcceptanceOfDeliveryClause extends AccordClause {

  /* the shipper of the goods*/
  --> Organization shipper

  /* the receiver of the goods */
  --> Organization receiver

  /* what we are delivering */
  o String deliverable

  /* how long does the receiver have to inspect the goods */
  o Integer businessDays

  /* additional information */
  o String attachment
}
```

Thanks to that model, the computer knows that the `shipper` variable (`"Party A"` in the example) and the `receiver` variable (`"Party B"` in the example) are both `Organization` types. The computer also knows that variable `businessDays` (`10` in the example) is an `Integer` type; and that the variable `deliverable` (`"Widgets"` in the example) is a `String` type, and can contain any text description.

> If you are unfamiliar with the different types of variables, or want a more thorough explanation of what variables are, please refer to our [Glossary](ref-glossary.md#data-models) for a more detailed explanation.

### Concerto

Concerto is the language which is used to write models in Accord Project templates. Concerto offers modern modeling capabilities including support for primitive types (numbers, dates, etc), nested or optional data structures, enumerations, relationships, object-oriented style inheritance, and more.

More information about Concerto can be found in the [Concerto Model](https://concerto.accordproject.org/docs/intro) section of this documentation.

## Template Logic

![Template Logic](assets/020/template_logic.png)

The combination of text and model already makes templates _machine-readable_, while the logic makes it _machine-executable_.

### During Drafting

In the [Overview](accordproject.md) Section, we already saw how logic can be embedded in the text of the template itself to automatically calculate a monthly payment for a [fixed rate loan]():

```tem
## Fixed rate loan
```

~~~
This is a *fixed interest* loan to the amount of {{loanAmount}}
at a yearly interest rate of {{rate}}%
with a loan term of {{loanDuration}},
and monthly payments of {{% monthlyPaymentFormula(loanAmount,rate,loanDuration)
%}}.
```

This uses a `monthlyPaymentFormula` function which calculates the monthly payment
based on the other data points in the text:
```ergo
define function monthlyPaymentFormula(loanAmount: Double, rate: Double,
loanDuration: Integer) : Double {
  let term = longToDouble(loanDuration * 12);       // Term in months
  if (rate = 0.0) then return (loanAmount / term)   // If the rate is 0
  else
    let monthlyRate = (rate / 12.0) / 100.0;        // Rate in months
    let monthlyPayment =                            // Payment calculation
      (monthlyRate * loanAmount)
      / (1.0 - ((1.0 + monthlyRate) ^ (-term)));
    return roundn(monthlyPayment, 0)                // Rounding
}
```
Each logic function has a _name_ (e.g., `monthlyPayment`), a _signature_ indicating
the parameters with their types (e.g., `loanAmount:Double`), and a _body_ which
performs the appropriate computation based on the parameters. The main payment
calculation is here based on the [standardized calculation used in the United
States](https://en.wikipedia.org/wiki/Mortgage_calculator#Monthly_payment_formula)
with `*` standing for multiplication, `/` for division, and `^` for exponentiation.

### After Signature

The logic can also be used to associate behavior to the template _after_ the
contract has been signed. This can be used for instance to specify what happens
when a delivery is received late, to check conditions for payment, determine if
there has been a breach of contract, etc.

The following shows post-signature logic for the **Acceptance of Delivery** clause.

```ergo
contract SupplyAgreement over SupplyAgreementModel {
  clause acceptanceofdelivery(request : InspectDeliverable) : InspectionResponse {

    let received = request.deliverableReceivedAt;
    enforce isBefore(received,now()) else
      throw ErgoErrorResponse{ message : "Transaction time is before the
deliverable date." }
    ;

    let status =
      if isAfter(now(), addDuration(received, Duration{ amount:
contract.businessDays, unit: ~org.accordproject.time.TemporalUnit.days}))
      then OUTSIDE_INSPECTION_PERIOD
      else if request.inspectionPassed
      then PASSED_TESTING
      else FAILED_TESTING
    ;
    return InspectionResponse{
      status : status,
~~~

```
      shipper : contract.shipper,
      receiver : contract.receiver
    }
  }
}
```

This logic describes what conditions must be met for a delivery to be accepted. It checks whether the delivery has already been made; whether the acceptance is timely, within the specified inspection date; and whether the inspection has passed or not.

### Ergo

Ergo is the programming language which is used to express contractual logic in templates. Ergo is specifically designed for legal agreements, and is intended to be accessible for those creating the corresponding prose for those computable legal contracts. Ergo expressions can also be embedded in the text for a template.

More information about Ergo can be found in the [Ergo Logic](logic-ergo.md) Section of this documentation.

## Cicero

The implementation for the Accord Project templates is called [Cicero](https://github.com/accordproject/cicero). It defines and can read the structure of templates, with natural language bound to a data model and logic. By doing this, Cicero allows users to create, validate and execute software templates which embody all three components in the template triangle above.

More information about how to install Cicero and get started with Accord Project templates can be found in the [Installation](started-installation.md) Section of this documentation.

Let's look at each component of the template triangle, starting with the text.

### What next?

Build your first smart legal contract templates, either [online](tutorial-studio.md) with Template Studio, or by [installing Cicero](started-installation.md).

Explore [sample templates](started-resources.md) and other resources in the rest of this documentation.

If some of technical words are unfamiliar, please consult the [Glossary](ref-glossary.md) for more detailed explanations.

---
id: accordproject-tour
title: Online Tour

To get an better acquainted with Accord Project templates, the easiest way is through the online [Template Studio](https://studio.accordproject.org) editor.

:::tip

You can open template studio from anywhere in this documentation by clicking the
[Try Online!](https://studio.accordproject.org) button located in the top-right of
the page.
:::

The following video offers a tour of Template Studio and an introduction to the key
concepts behind the Accord Project technology.

<iframe src="https://player.vimeo.com/video/328933628" width="640" height="400"
frameborder="0" allow="autoplay; fullscreen" allowfullscreen></iframe>

Here is a timestamp of what is covered in the video:

- **00:08** : Introduction to template Studio & pointers the other parts of the
docs / AP website
- **03:51** : Helloworld template tutorial start
- **04:48** : Template Studio - Metadata
- **06:56** : Template Studio - Contract Text
  - **07:52** : The 'live' nature of the text; how to read errors
  - **08:39** : Changing the template text itself
  - **09:17** : Changing the variables in the template text
- **10:00** : Template Studio - Model update
- **11:28** : Template Studio - Logic update
  - **13:17** : Explanation about request types / response types
- **14:44** : Template Studio - Logic - Test Execution (request, response)
  - **15:57** : Test Execution - contract state
  - **16:49** : Test Execution - obligations
- **18:20** : Wrap-up

---
id: accordproject
title: Overview

## What is the Accord Project?

Accord Project is an open source, non-profit initiative aimed at transforming
contract management and contract automation by digitizing contracts. It provides an
open, standardized format for Smart Legal Contracts.

The Accord Project defines a notion of a legal template with associated computing
logic which is expressive, open-source, and portable. Accord Project templates are
similar to a clause or contract template in any document format, but they can be
read, interpreted, and run by a computer.

## Why is the Accord Project relevant?

The Accord Project provides a universal format for smart legal contracts, and this
format is embodied in a variety of open source projects that comprise the Accord
Project technology stack. Input from businesses, lawyers and developers is crucial
for the Accord Project.

### For Businesses

Contracting is undergoing a digital transformation driven by a need to deliver
customer-centric legal and business solutions faster, and at lower cost. This
imperative is fueling the adoption of a broad range of new technologies to improve

the efficiency of drafting, managing, and executing legal contracting operations; the Accord Project is proud to be part of that movement.

The Accord Project provides a Smart Contract that does not depend on a blockchain, that can integrate text
and data and that can continue operating over its lifespan. The Accord Project smart contract can integrate with your technology platforms and become part of you digital infrastructure.

In addition, contributions from businesses are crucial for the development of the Accord Project. The expertise of stakeholders, such as business professionals and attorneys, is invaluable in improving the functionality and content of the Accord Project's codebase and specifications, to ensure that the templates meet real-world business requirements.

If this interests you, please visit our [Lifecycle and Industry Working Groups](https://www.accordproject.org/liwg) page for more information.

### For Lawyers

The Legal world is changing and Legal Tech is growing into a billion dollar industry. The modern lawyer has to be at home in the digital world. Law Schools now teach courses in coding for lawyers, computational law, blockchain and artificial intelligence. Legal Hackers is a world wide movement uniting lawyers across the world in a shared passion for law and technology. Lawyers need to move beyond the the written word on paper.

The template in an Accord Project Contract is pure legal text that can be drafted by lawyers and interpreted by courts. An existing contract can easily be transformed into a template by adding data points between curly braces that represent the Concerto model and Ergo logic can be added as an integral part of the contract. The template language is subject to judicial interpretation and the Concerto model and Ergo logic can be interpreted by a computer creating a bridge between the two worlds.

As a lawyer, contributing to the Accord Project would be a great opportunity to learn about smart legal contracts. Through the Accord Project, you can understand the foundations of open source technologies and learn how to develop smart agreements.

If your organization wants to become a member of the Accord Project, please [join our community](https://discord.com/invite/Zm99SKhhtA).

### For Developers

The Accord Project provides a universal format for smart legal contracts, and this format is embodied in a variety of open source projects that comprise the Accord Project technology stack. The Accord Project is an open source project and welcomes contributions from anyone.

The Accord Project is developing tools including a [Visual Studio Code plugin](https://marketplace.visualstudio.com/items?itemName=accordproject.cicero-vscode-extension), [React based web components](https://github.com/accordproject/web-components) and a command line interface for working with Accord Project Contracts. You can integrate contracts into existing applications, create new applications or simply assist lawyers with developing applications with the Ergo language.

There is a welcoming community on Discord that is eager to help. [Join our Community](https://discord.com/invite/Zm99SKhhtA)

## About this documentation

If you are new to Accord Project, you may want to first read about the notion of
[Smart Legal Contracts](accordproject-slc.md) and about [Accord Project Templates]
(accordproject-template.md). We also recommend taking the [Online Tour]
(accordproject-tour.md).

To start using Accord Project templates, follow the [Install
Cicero](https://docs.accordproject.org/docs/next/started-installation.html)
instructions in the _Getting Started_ Section of the documentation.

You can find in-depth guides for the different components of a template in the
Template Guides part of the documentation:
- Learn how to write contract or template text in the [Markdown Text](markup-preliminaries.md) Guide
- Learn how to design your data model in the [Concerto
Model](https://concerto.accordproject.org/docs/intro) Guide
- Learn how to write smart contract logic in the [Ergo Logic](logic-ergo.md) Guide

Finally, the documentation includes several step by step [Tutorials](tutorial-templates.md) and some reference information (for APIs, command-line tools, etc.)
can be found in the [Reference Manual](ref-glossary.md).

---
id: ergo-tutorial
title: Ergo: A Tutorial

## Overview of Accord

Cicero is an Open Source implementation of the Accord Project Template
Specification. It defines the structure of natural language templates, bound to a
data model, that can be executed using Ergo and request/response JSON messages. You
can read the latest user documentation here: http://docs.accordproject.org.

In short, with the Accord Project you can take a classic contract, e.g. Word
document and use Cicero to define natural language contract and clause templates
that can be executed by an event driven computer program (aka Smart contract). For
the tutorial, Cicero will be used to define natural language contract and clause
templates. These clause templates handle the syllogistic language of contracts.

For example,
```md
 if the goods are more than [{DAYS}] late,
 then notify the supplier of the goods, with the message [{MESSAGE}].
```
DAYS and MESSAGE are variables

You can browse the library of Open Source Cicero contract and clause templates at:
https://templates.accordproject.org.

So how goes the contract get executed? That is where Ergo comes in Ergo is a
strongly-typed functional programming language designed to capture the legal intent
of legal contracts and clauses. We will use Ergo to create the contract logic
consisting of a contract class with executable embedded clauses. Note: prior to the

emergence of Ergo, the Cicero JavaScript component was primary to the execution of code.

Ergo obviates the Cicero JavaScript component for the execution phase with a new more comprehensive language which we explore in this tutorial.

## Cicero

The Open Source Cicero project defines the format of clause and contract templates based on to the Cicero Template Specification. The templates are the link between the natural language of contracts usually composed in a Word document and the specification of a machine executable transaction. Cicero templates define the API by specifying request and response elements for the logic associated with functional transaction executed by Ergo.

Cicero templates are composed of two elements:
* Template Grammar (the natural language text for the template),
* Template Model (the data model that includes the variables contained within the template).
* The Logic (the executable business logic for the template) will be handled by Ergo.

When combined these three elements allow templates to be edited, analyzed, queried and executed.

## Setup Ergo Development environment

Before you can build Ergo, you must install and configure the following dependencies on your machine:

### Git

* Git: The [Github Guide to Installing Git][git-setup.md] is a good source of information.

### Node.js

* Node.js (LTS): We use Node to generate the documentation, run a development web server, run tests, and generate distributable files. Depending on your system, you can install Node either from source or as a pre-packaged bundle.
> Tip: Use nvm (or nvm-windows) to manage and install Node.js, This facilitates a version change of Node.js per project.

* Lerna: This is a tool which helps when handling multiple npm packages in the Ergo repository. To install:

```
npm install -g lerna@^3.15.0
```

### Visual Studio Code

Visual Studio Code is a lightweight but powerful source code editor which runs on your desktop and is available for Windows, macOS and Linux. It comes with built-in support for JavaScript and Node.js and has a rich ecosystem of extensions for other languages (such Ergo).

Follow the platform specific guides below:
See, https://code.visualstudio.com/docs/setup/
* macOS
* Linux
* Windows

#### Install Ergo VisualStudio Plugin

### Validate Development Environment and Toolset

Clone https://github.com/accordproject/ergo to your local machine

### Getting started

Install Ergo

The easiest way to install Ergo is as a Node.js package. Once you have Node.js installed on your machine, you can get the Ergo compiler and command-line using the Node.js package manager by typing the following in a terminal:
$ npm install -g @accordproject/ergo-cli@0.20

This will install the compiler itself (ergoc) and a command-line tool (ergo) to execute Ergo code. You can check that both have been installed and print the version number by typing the following in a terminal:
```sh
$ ergoc --version
$ ergo --version
```
Then, to get command line help:
```
$ ergoc --help
$ ergo execute --help
```
Compiling your first contract
```ergo
namespace org.accordproject.volumediscount

contract VolumeDiscount over VolumeDiscountContract {
   // Clause for volume discount
   clause volumediscount(request : VolumeDiscountRequest) : VolumeDiscountResponse
{
        if request.netAnnualChargeVolume < contract.firstVolume
        then return VolumeDiscountResponse{ discountRate: contract.firstRate }
        else if request.netAnnualChargeVolume < contract.secondVolume
        then return VolumeDiscountResponse{ discountRate: contract.secondRate }
        else return VolumeDiscountResponse{ discountRate: contract.thirdRate }
  }
}
```

To compile your first Ergo contract to JavaScript , within Visual Studio code
* Open the folder where you cloned https://github.com/accordproject/ergo
* Use View/Terminal to run the Ergo compiler:
```sh
$ ergoc ./examples/volumediscount/model.cto ./examples/volumediscount/logic.ergo
Compiling Ergo './examples/volumediscount/logic.ergo' -- creating
'./examples/volumediscount/logic.js'
```

By default, Ergo compiles to JavaScript for execution. This may change in the future to support other languages. The compiled code for the result in stored as `./examples/volumediscount/logic.js`

### Execute a contract
To execute a contract, we pass the necessary parameters including the CTO, Ergo

files, the name of a contract and the json files containing request and contract state
ergorun [ctos] [ergos] --contractname [file] --contract [file] --state [file] --request [file]

So for example we use ergorun with :
```sh
$ ergorun ./examples/volumediscount/model.cto ./examples/volumediscount/logic.ergo
--contractname org.accordproject.volumediscount.VolumeDiscount
--contract ./examples/volumediscount/contract.json
--request ./examples/volumediscount/request.json
--state ./examples/volumediscount/state.json
```

Here contract.json contains the following values
```json
{
  "$class": "org.accordproject.volumediscount.VolumeDiscountContract",
  "parties": null,
  "contractId": "cr1",
  "firstVolume": 1,
  "secondVolume": 10,
  "firstRate": 3,
  "secondRate": 2.9,
  "thirdRate": 2.8
}
```

Request.json contains
```json
{
  "$class": "org.accordproject.volumediscount.VolumeDiscountRequest",
  "netAnnualChargeVolume": 10.4
}
```

logic.ergo contains:
```ergo
namespace org.accordproject.volumediscount

contract VolumeDiscount over VolumeDiscountContract {
  // Clause for volume discount
  clause volumediscount(request : VolumeDiscountRequest) : VolumeDiscountResponse {
    if request.netAnnualChargeVolume < contract.firstVolume
    then return VolumeDiscountResponse{ discountRate: contract.firstRate }
    else if request.netAnnualChargeVolume < contract.secondVolume
    then return VolumeDiscountResponse{ discountRate: contract.secondRate }
    else return VolumeDiscountResponse{ discountRate: contract.thirdRate }
  }
}
```

Here netAnnualCharge Volume equals 10.4 which is not less than firstVolume and secondVolume which are equal to 1 and 10 respectively so the logic for the volumediscount clause returns thirdRate which equals 2.8

```
7:31:58 PM - info: Logging initialized. 2018-09-27T23:31:58.623Z
7:31:59 PM - info: {"response":
```

```
{"discountRate":2.8,"$class":"org.accordproject.volumediscount.VolumeDiscountRespon
se"},"state":
{"$class":"org.accordproject.cicero.contract.AccordContractState","stateId":"1"},"e
mit":[]}
```

PS D:\Users\jbambara\Github\ergo>

## Ergo Development

Create Template
Start with basic agreement in natural language and locate the variables
Here in the example see the bold
Volume-Based Card Acceptance Agreement [Abbreviated]
This Agreement is by and between ………..you agree to be bound by the Agreement.
Discount means an amount that we charge you for accepting the Card, which amount is:
(i) a percentage (Discount Rate) of the face amount of the Charge that you submit, or a flat per-
Transaction fee, or a combination of both; and/or
(ii) a Monthly Flat Fee (if you meet our requirements).

Transaction Processing and Payments. ………………… less all applicable deductions, rejections, and withholdings, which include:
………………………….

SETTLEMENT
a) Settlement Amount. Our agent will pay you according to your payment plan, ……………………..which include:
        (i) the Discount,
……………………………………………..
b) Discount. The Discount is determined according to the following table:

Annual Dollar Volume	Discount
Less than $1 million	3.00%
$1 million to $10 million	2.90%
Greater than $10 million	2.80%

Identify the request variables and contract instance variables
Codify the variables with $[{request}] or [{contract instance}]

Annual Dollar Volume	Discount
Less than $[{firstVolume}] million	[{firstRate}]%
$[{firstVolume}] million to $[{secondVolume}] million	[{secondRate}]%
Greater than $[{secondVolume}] million	[{thirdRate}]%

Create Model
Define the model asset which contains the contract instance variables and the transaction request and response. Defines the data model for the VolumeDiscount template. This defines the structure that the parser for the template generates from input source text. See model.cto below:

```
 namespace org.accordproject.volumediscount
import org.accordproject.cicero.contract.* from
https://models.accordproject.org/cicero/contract.cto
import org.accordproject.cicero.runtime.* from
https://models.accordproject.org/cicero/runtime.cto
asset VolumeDiscountContract extends AccordContract {
  o Double firstVolume
  o Double secondVolume
  o Double firstRate
  o Double secondRate
  o Double thirdRate
}
transaction VolumeDiscountRequest {
  o Double netAnnualChargeVolume
}
transaction VolumeDiscountResponse {
        o Double discountRate
}
```

Create Logic
The contract logic is accomplished by coding ERGO statements and expressions to consume the request and use contract instance variables to produce the desired response. In our example, request.netAnnualChargeVolume is tested against contract rates to produce the result.

```
namespace org.accordproject.volumediscount

define the contract
contract VolumeDiscount over VolumeDiscountContract {

define the contract clause and request : response

   clause volumediscount(request : VolumeDiscountRequest) : VolumeDiscountResponse
{

define the logic ; here we use if /then /else statement to test request parameter
against contract instance variable
 and return

        if request.netAnnualChargeVolume < contract.firstVolume
        then return VolumeDiscountResponse{ discountRate: contract.firstRate }
        else if request.netAnnualChargeVolume < contract.secondVolume
        then return VolumeDiscountResponse{ discountRate: contract.secondRate }
        else return VolumeDiscountResponse{ discountRate: contract.thirdRate }
  }
```

Ergo Language
As you have seen in this tutorial, Ergo is a domain-specific language (DSL) that captures the execution logic of legal contracts. In this simple example, you see that Ergo aims to have contracts and clauses as first-class elements of the language. To accommodate the maturation of distributed ledger implementations, Ergo will be blockchain neutral, i.e., the same contract logic can be executed either on and off chain on distributed ledger technologies like HyperLedger Fabric. Most importantly, Ergo is consistent with the Accord Protocol Template Specification.
Follow the links below to learn more about
Introduction to Ergo
Ergo Language Guide
Ergo Reference Guide

October 12, 2018

```
---
id: example-eatapple
title: A Healthy Clause
---
```

## Eat Apples!

The healthy eating clause is inspired by the not so serious [terms of services contract](https://www.grahamcluley.com/page-46-apples-new-ios-agreement-funny-fake-makes-serious-point/).

For this example, let us look first at the template for that legal clause written in natural language:

```markdown
Eating healthy clause between [{employee}] (the Employee) and [{company}] (the Company).
The canteen only sells apple products. Apples, apple juice, apple flapjacks, toffee apples. Employee gets fired if caught eating anything without apples in it.
THE EMPLOYEE, IF ALLERGIC TO APPLES, SHALL ALWAYS BE HUNGRY.
Apple products at the canteen are subject to a [{tax}]% tax.
```

The text specifies the terms for the legal clause and includes a few variables such as `employee`, `company` and `tax`.

The second component of a smart legal template is the model, which is expressed using the [Concerto modeling language](https://github.com/accordproject/concerto).
The model describes the variables of the contract, as well as additional information required to execute the contract logic. In our example, this includes the input request for the clause (`Food`), the response to that request (`Outcome`) and possible events emitted during the clause execution (`Bill`).

```ergo
namespace org.accordproject.canteen

@AccordTemplateModel("eat-apples")
concept CanteenContract {
  o String employee
  o String company
  o Double tax
}

transaction Food {
  o String produce
  o Double price
}

transaction Outcome {
  o String notice
}

event Bill {
  o String billTo
  o Double amount
}
```

The last component of a smart legal template is the Ergo logic. In our example, it is a single clause `eathealthy` which can be used to process a `Food` request.

```ergo
namespace org.accordproject.canteen
```

```
contract EatApples over CanteenContract {
  clause eathealthy(request : Food) : Outcome {
    enforce request.produce = "apple"
    else return Outcome{ notice : "You're fired!" };

    emit Bill{
      billTo: contract.employee,
      amount: request.price * (1.0 + contract.tax / 100.0)
    };
    return Outcome{ notice : "Very healthy!" }
  }
}
```

As in the "Hello World!" example, this is a smart legal contract with a single clause, but it illustrate a few new ideas.

The `enforce` expression is used to check conditions that must be true for normal execution of the clause. In this example, the `enforce` makes sure the food is an apple and if not returns a new outcome indicating termination of employment.

If the condition is true, the contract proceeds by emitting a bill for the purchase of the apple. The employee to be billed is obtained from the contract (`contract.employee`). The total amount is calculated by adding the tax, which is obtained from the contract (`contract.tax`), to the purchase price, which is obtained from the request (`request.price`). The calculation is done using a simple arithmetic expression (`request.price * (1.0 + contract.tax / 100.0)`).

---
id: logic-advanced-expr
title: Advanced Expressions

## Match

### Match against Values

Match expressions allow to check an expression against multiple possible values:

```ergo
    match fruitcode
      with 1 then "Apple"
      with 2 then "Apricot"
      else "Strange Fruit"
```

Match expressions can also be used to match against enumerated values:
```ergo
    match state
      with NY then "Empire State"
      with NJ then "Garden State"
      else "Far from home state"
```

### Match against Types

Match expressions can be used to match a value against a class type:.

```
define constant products = [
    Product{ id : "Blender" },
    Car{ id : "Batmobile", range: "Infinite" },
    Product{ id : "Cup" }
  ]

foreach p in products
return
  match p
    with let x : Car then "Car (" ++ x.id ++ ") with range " ++ x.range
    with let x : Product then "Product (" ++ x.id ++ ")"
    else "Not a product"
```
Should return the array `["Product (Blender)", "Car (Batmobile) with range Infinite", "Product (Cup)"]`

## Foreach

Foreach expressions allow to apply an expression of every element in an input array of values and returns a new array:

```ergo
  foreach x in [1.0,-2.0,3.0] return x + 1.0
```

Foreach expressions can have an optional condition of the values being iterated over:

```ergo
  foreach x in [1.0,-2.0,3.0] where x > 0.0 return x + 1.0
```

Foreach expressions can iterate over multiple arrays. For example, the following foreach expression returns all all [Pythagorean triples](https://en.wikipedia.org/wiki/Pythagorean_triple):
```ergo
let nums = [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0];
foreach x in nums
foreach y in nums
foreach z in nums
where (x^2.0 + y^2.0 = z^2.0)
return {a: x, b: y, c: z}
```
and should return the array `[{a: 3.0, b: 4.0, c: 5.0}, {a: 4.0, b: 3.0, c: 5.0}, {a: 6.0, b: 8.0, c: 10.0}, {a: 8.0, b: 6.0, c: 10.0}]`.

## Template Literals

Template literals are similar to [String literals](logic-simple-expr.md#literal-values) but with the ability to embed Ergo expressions. They are written with between `` ` `` and may contains Ergo expressions inside `{{%` and `%}}`.

The following Ergo expressions illustrates the use of a template literal to

construct a String describing the content of a record.
```
let law101 = {
    name: "Law for developers",
    fee: 29.99
  };
`Course "{{% law101.name %}}" (Cost: {{% law101.fee %}})`
```
Should return the string literal `"Course \"Law for developers\" (Cost: 29.99)"`.

## Formatting

One can use template formatting using the `Expr as "FORMAT"` Ergo expression.
Supported formats are the same as those available in TemplateMark [Formatted Variables](markup-templatemark.md#formatted-variables).

For instance:
```
let payment = MonetaryAmount{ currencyCode: USD, doubleValue: 1129.99 };
payment as "K0,0.00"
```
Should return the string literal `"$1,129.99"`.

---
id: logic-complex-type
title: Complex Values & Types

So far we only considered atomic values and types, such as string values or integers, which are not sufficient for most contracts. In Ergo, values and types are based on the [Concerto Modeling](https://concerto.accordproject.org/docs/intro) (often referred to as CTO models after the `.cto` file extension). This provides a rich vocabulary to define the parameters of your contract, the information associated to contract participants, the structure of contract obligation, etc.

In Ergo, you can either import an existing CTO model or declare types directly within your code. Let us look at the different kinds of types you can define and how to create values with those types.

## Arrays

Array types lets you define collections of values and are denoted with `[]` after the type of elements in that collection:

```ergo
    String[]                         // a String array
    Double[]                         // a Double array
```

You can write arrays as follows:
```ergo
    ["pear","apple","strawberries"]  // an array of String values
    [3.14,2.72,1.62]                 // an array of Double values
```

You can construct arrays using other expressions:
```ergo
    let pi = 3.14;
```

    let e = 2.72;
    let golden = 1.62;
    [pi,e,golden]

Ergo also provides functions to manipulate arrays as parts of its [standard library](ref-ergo-stdlib.md#functions-on-arrays). The following example uses the `sum` function to calculate the sum of all the elements in the `prettynumbers` array.
```ergo
    let pi = 3.14;
    let e = 2.72;
    let golden = 1.62;
    let prettynumbers : Double[] = [pi,e,golden];
    sum(prettynumbers)
```

You can access the element at a given position inside the array using an index:
```ergo
    let fruits = ["pear","apple","strawberries"];
    fruits[0]          // Returns: some("pear")
     let fruits = ["pear","apple","strawberries"];
    fruits[2]          // Returns: some("strawberries")
     let fruits = ["pear","apple","strawberries"];
    fruits[4]          // Returns: none
```

 Note that the index starts at `0` for the first element and that indexed-based access returns an optional value, since Ergo compiler cannot statically determine whether there will be an element at the corresponding index. You can learn more about how to handle optional values and types in the [Optionals](logic-complex-type.md#optionals) Section below.

## Classes

You can declare classes in the Concerto Modeling Language (concepts, transactions, events, participants or assets) by importing them from a CTO file or directly within your Ergo program:

```ergo
   define concept Seminar {
     name : String,
     fee : Double
   }
   define asset Product {
     id : String
   }
   define asset Car extends Product {
     range : String
   }
   define transaction Response {
     rate : Double,
     penalty : Double
   }
  define event PaymentObligation{
    amount : Double,
    description : String
  }
```

Once a class type has been defined, you can create an instance of that type using
the class name along with the values for each fields:

```ergo
    Seminar{
      name: "Law for developers",
      fee: 29.99
    }
    Car{
      id: "Batmobile4156",
      range: "Infinite"
    }
```

> **TechNote:** When extending an existing class (e.g., `Car extends Product`), the
sub-class includes the fields from the super-class. So `Car` includes the field
`range` which is locally declared and the field `id` which is declared in
`Product`.

You can access fields for values of a class type by using the `.` operator:
```ergo
    Seminar{
      name: "Law for developers",
      fee: 29.99
    }.fee                        // Returns 29.99
```

## Records

Sometimes it is convenient to declare a structure without having to declare it
first. You can do that using a record, which is similar to a class but without a
name attached to it:

```ergo
   {
     name : String,  // A record with a name of type String
     fee : Double    // and a fee of type Double
   }
```

You do not need to declare that record, and can directly write an instance of that
record as follows:

```ergo
    {
      name: "Law for developers",
      fee: 29.99
    }
```

> Typing `return { name: "Law for developers", fee: 29.99 }` in the [Ergo REPL]
(https://ergorepl.netlify.com), should answer `Response. {name: "Law for
developers", fee: 29.99} : {fee: Double, name: String}`.

You can access the field of a record using the `.` operator:
```ergo
    {
      name: "Law for developers",
```

~~~
      fee: 29.99
    }.fee                        // Returns 29.99
```
## Enums

Here is how to declare an enumerated type:

```ergo
define enum ProductType {
   DAIRY,
   BEEF,
   VEGETABLES
}
```

To create an instance of that enum:
```ergo
DAIRY
BEEF
```

## Optionals

An optional type can contain a value or not and is indicated with a `?`.

```ergo
Integer?           // An optional integer
PaymentObligation? // An optional payment obligation
Double[]?          // An optional array of doubles
```

An optional value can be either present, written `some(v)`, or absent, written
`none`.

```ergo
let i1 : Integer? = some(1); i1
let i2 : Integer? = none; i2
```

To operate on an optional type, you need to say what to do when the value is
present and what to do when the value is not present. The most general way to do
that is with a match expression:

This example matches a value which is present:
```ergo
match some(1)
with let? x then "I found " ++ toString(x) ++ " :-)"
else "I found nothing :-("
```
and should return `"I found 1 :-)"`.

While this example matches against a value which is absent:
```
match none
with let? x then "I found " ++ toString(x) ++ " :-)"
else "I found nothing :-("
```
and should return `"I found nothing :-("`.
~~~

More details on match expressions can be found in [Advanced Expressions](logic-advanced-expr.md#match).

For conciseness, a few operators are also available on optional values. One can give a default value when the optional is `none` using the operator `??`. For instance:

```ergo
some(1) ?? 0       // Returns the integer 1
none ?? 0          // Returns the integer 0
```

You can also access the field inside an optional concept or an optional record using the operator `?.`. For instance:

```ergo
some({a:1})?.a     // Returns the optional value: some(1)
none?.a            // Returns the optional value: none
```

---
id: logic-decl
title: Declarations

Now that we have values, types, expressions and statements available, we can start writing more complex Ergo logic using by declaring functions, clauses and contracts.

## Constants and functions

It is possible to declare global constants and functions in Ergo:

```ergo
define constant pi = 3.1416
define function area(radius : Double) : Double {
   return pi * radius * radius
}
area(1.5)
```

Global variables can also be declared with a type:

```ergo
define constant pi : Double = 3.1416
```

The return type of functions can be omitted:

```ergo
define function area(radius : Double) {
   return pi * radius * radius
}
area(1.5)
```

## Clauses

In Ergo, a logical clause like the example clause noted below is represented as a “function” (akin to a “method” in languages like Java) that resides within its parent contract (akin to a “class” in a language like Java).

> Functions are "self contained" modules of code that accomplish a specific task. Functions usually "take in" data, process it, and "return" a result. Once a function is written, it is reusable , i.e., it can be used over and over and over again.
> Functions can be "called" from within other functions or from a clause.
> Functions have to be declared before they can be used. So functions "encapsulate" a task. They combine statements and expressions carried out as instructions which accomplish a specific task to allow their execution using a single line of code. Most programming languages provide libraries of built in functions (i.e., commonly used tasks like computing the square root of a number).
> Functions accelerate development and facilitate the reuse of code which performs common tasks.

The declaration of a Clause that contains the clause’s name, request type and return type collectively referred to as the ‘signature’ of the function.

### Example Prose

Additionally the Equipment should have proper devices on it to record any shock during transportation as any instance of acceleration outside the bounds of -0.5g and 0.5g. Each shock shall reduce the Contract Price by $5.00

### Syntax

```ergo
    clause fragileGoods(request : DeliveryUpdate) : ContractPrice {
        ... // A statement computing the clause response
    }
```

Inside a contract, the `contract` variable contains the instance of the template model for the current contract.

## Contract Declarations

The legal requirements for a valid contract at law vary by jurisdiction and contract type. The requisite elements that typically necessary for the formation of a legally binding contract are (1) _offer_; (2) _acceptance_; (3) _consideration_; (4) _mutuality of obligation_; (5) _competency and capacity_; and, in certain circumstances, (6) _a written instrument_.

Ergo contacts address consideration, mutuality of obligation, competency and capacity through statements that are described in this document.

Furthermore, an Ergo contract is an immutable written document which obviates a good deal of the issues and conflicts which emerge from existing contracts in use today. In Ergo, a contract:
- represents an agreement between parties creating mutual and enforceable obligations; and
- is a code module that uses conditionals and functions to describe execution by the parties with their obligations. Contracts accept input data either directly from the associated natural language text or through request _transactions_. The contract then uses _clause functions_ to process it, and _return_ a result.
Once a contract logic has been written within a template, it can be used over and

over and over again.

Instantiated contracts correspond to particular domain agreement. They combine functions and clauses to execute a specific agreement and to allow its automation. Many traditional contracts are “boilerplate” and as such are reusable in their specific legal domain, e.g., sale of goods.

You can declare a contract over a template model as follows. The `TemplateModel` is the data model for the parameters of the contract text.

```ergo
    contract ContractName over TemplateModel {
      clause C1(request : ReqType1) : RespType1 {
        // Statement
      }

      clause C2(request : ReqType2) : RespType2 {
        // Statement
      }
    }
```

## Contract State and Obligations

If your contract requires a state, or emits only certain kinds of obligations but not other, you can specify the corresponding types when declaring your contract:

```ergo
    contract ContractName over TemplateModel state MyState {
      clause C1(request : ReqType1) : RespType1 emits MyObligation {
        // Statement
      }

      clause C2(request : ReqType2) : RespType2 {
        // Statement
      }
    }
```

The state is always declared for the whole contract, while obligations can be declared individually for each clause.

---
id: logic-ergo
title: Ergo Overview

## Language Goals

Ergo aims to:
- have contracts and clauses as first-class elements of the language
- help legal-tech developers quickly and safely write computable legal contracts
- be modular, facilitating reuse of existing contract or clause logic
- ensure safe execution: the language should prevent run-time errors and non-terminating logic
- be blockchain neutral: the same contract logic can be executed either on and off chain on a variety of distributed ledger technologies

- be formally specified: the meaning of contracts should be well defined so it can be verified, and preserved during execution
- be consistent with the [Accord Project Templates](accordproject-template.md)

## Design Choices

To achieve those goals the design of Ergo is based on the following
principles:

- Ergo contracts have a class-like structure with clauses akin to methods
- Ergo can handle types (concepts, transactions, etc) defined with the [Concerto Modeling Language](https://github.com/accordproject/concerto) (so called CML models), as mandated by the Accord Project Template Specification
- Ergo borrows from strongly-typed functional programming languages: clauses have a well-defined type signature (input and output), they are functions without side effects
- The compiler guarantees error-free execution for well-typed Ergo programs
- Clauses and functions are written in an expression language with limited expressiveness (it allows conditional and bounded iteration)
- Most of the compiler is written in Coq as a stepping stone for formal specification and verification

## Status

- The current implementation is considered *in development*, we welcome contributions (be it bug reports, suggestions for new features or improvements, or pull requests)
- The current compiler targets JavaScript (either standalone or for use in Cicero Templates and Hyperledger Fabric) and Java (experimental)

## This Guide

Ergo provides a simple expression language to describe computation. From those expressions, one can write functions, clauses, and then whole contract logic. This guide explains most of the Ergo concepts starting from simple expressions all the way to contracts.

Ergo is a _strongly typed_ language, which means it checks that the expressions you use are consistent (e.g., you can take the square root of `3.14` but not of `"pi!"`). The type system is here to help you write better and safer contract logic, but it also takes a little getting used to. This page also introduces Ergo types and how to work with them.

---
id: logic-module
title: Modules

Finally, we can place multiple Ergo declarations (functions, contracts, etc) into a library so it can be shared with other developers.

## Namespaces

Each Ergo file starts with a namespace declaration which provides a way to identify it uniquely:
```ergo
namespace org.acme.mynamespace
```

## Libraries

A library is an Ergo file in a namespace which defines useful constants or
functions. For instance:

```ergo
namespace org.accordproject.ergo.money

define constant days_in_a_year = 365.0
define function compoundInterests(
  annualInterest : Double,
  numberOfDays : Double
) : Double {
    return (1.0 + annualInterest) ^ (numberOfDays / days_in_a_year)
}
```

## Import

You can then access this library in another Ergo file using import:
```ergo
namespace org.accordproject.promissorynote

contract PromissoryNote over PromissoryNoteContract {
  clause check(request : Payment) : Result {
        let interestRate = contract.interestRate ?? 3.4;
    enforce interestRate >= 0.0;
    enforce contract.amount.doubleValue >= 0.0;
    let outstanding = contract.amount.doubleValue - request.amountPaid.doubleValue;
    enforce outstanding >= 0.0;

    let numberOfDays =
      diffDurationAs(
        dateTime("17 May 2018 13:53:33 EST"),
        contract.date,
        ~org.accordproject.time.TemporalUnit.days).amount;

    enforce numberOfDays >= 0;

    let compounded =  outstanding
      * compoundInterestMultiple(interestRate, numberOfDays); // Defined in
ergo.money module

    return Result{
      outstandingBalance: compounded
    }
  }
}
```

> **TechNote:** the namespace and import handling in Ergo allows you to access
either existing CTO models or Ergo libraries in the same way.

---
id: logic-simple-expr

title: Simple Expressions

## Literal values

The simplest kind of expressions in Ergo are literal values.

```ergo
    "John Smith" // a String literal
    1            // an Integer literal
    3.0          // a Double literal
    3.5e-10      // another Double literal
    true         // the Boolean true
    false        // the Boolean false
```

Each line here is a separate expression. At the end of the line, the notation `// write something here` is a _comment_, which means it is a part of your Ergo program which is ignored by the Ergo compiler. It can be useful to document your code.

Every Ergo expression can be _evaluated_, which means it should compute some value. In the case of a literal value, the result of evaluation is itself (e.g., the expression `1` evaluates to the integer `1`).

> You can actually see the result of evaluating expressions by trying them out in the [Ergo REPL](https://ergorepl.netlify.com). You have to prefix them with `return`: for instance, to evaluate the String literal `"John Smith"` type: `return "John Smith"` (followed by clicking the button 'Evaluate') in the REPL. This should answer: `Response. "John Smith" : String`.

## Operators

You can apply operators to values. Those can be used for arithmetic operations, to compare two values, to concatenate two string values, etc.

```ergo
    1.0 + 2.0 * 3.0      // arithmetic operators on Double
    -1.0
    1 + 2 * 3            // arithmetic operators on Integer
    -1

    1.0 <= 3.0           // comparison operators on Double
    1.0 = 2.0
    2.0 > 1.0
    1 <= 3               // comparison operators on Integer
    1 = 2
    2 > 1.0

    true or false        // Boolean disjunction
    true and false       // Boolean conjunction
    !true                // Negation

    "Hello" ++ " World!" // String concatenation
```

> Again, you can try those in the [Ergo REPL](https://ergorepl.netlify.com). For instance, typing `return true and false` should answer `Response. false : Boolean`, and typing `return 1.0 + 2.0 * 3.0` should answer: `Response. 7.0 : Double`.

## Conditional expressions

Conditional expressions can be used to perform different computations depending on some condition:

```ergo
    if 1.0 < 0.0     // Condition
    then "negative"  // Expression if condition is true
    else "positive"  // Expression if condition is false
```

> Typing `return if 1.0 < 0.0 then "negative" else "positive"` in the [Ergo REPL](https://ergorepl.netlify.com), should answer `Response. "positive" : String`.

See also the [Conditional Expression Reference](ref-ergo-spec.html#condition-expressions)

## Let bindings

Local variables can be declared with `let`:

```ergo
    let x = 1;              // declares and initialize a variable
    x+2                     // rest of the expression, where x is in scope
```

Let bindings give a name to some intermediate result and allows you to reuse the corresponding value in multiple places:

```ergo
   let x = -1.0;            // bind x to the value -1.0
   if x < 0.0               // if x is negative
   then -x                  // then return the opposite of x
   else x                   // else return x
```

> **TechNote:** let bindings in Ergo are immutable, in a way similar to other functional languages. A nice explanation can be found e.g., in the documentation for let bindings in [ReasonML](https://reasonml.github.io/docs/en/let-binding).

---
id: logic-simple-type
title: Introducing Types

We have so far talked about types only informally. When we wrote earlier:
```ergo
    "John Smith" // a String literal
    1            // an Integer literal
    ...
```
the comments mention that `"John Smith"` is of type `String`, and that `1` is of type `Integer`.

In reality, the Ergo compiler understands which types your expressions have and can detect whether those expressions apply to the right type(s) or not.

Ergo types are based on the [Concerto
Modeling](https://concerto.accordproject.org/docs/intro) Language.

## Primitive types

The simplest of types are primitive types which describe the various kinds of
literal values we saw in the previous section. Those primitive types are:

```ergo
    Boolean
    String
    Double
    Integer
    Long
    DateTime
```

:::note
The two primitive types `Integer` and `Long` are currently treated as the same type
by the Ergo compiler.
:::

## Type errors

The Ergo compiler understand types and can detect type errors when you write
expressions. For instance, if you write: `1.0 + 2.0 * 3.0`, the Ergo compiler knows
that the expression is correct since all parameters for the operators `+` and `*`
are of type `Double`, and it knows the result of that expression will be a `Double`
as well.

If you write `1.0 + 2.0 * "some text"` the Ergo compiler will detect that `"some
text"` is of type `String`, which is not of the right type for the operator `*` and
return a type error.

> Typing `return 1.0 + 2.0 * "some text"` in the [Ergo
REPL](https://ergorepl.netlify.com), should answer a type error:
> ```text
> Type error (at line 1 col 13). This operator received
> unexpected arguments of type Double  and String.
> return 1.0 + 2.0 * "some text"
>              ^^^^^^^^^^^^^^^^^
> ```

## Type annotations

In a let bindings, you can also use a _type annotation_ to indicate which type you
expect it to have.

```ergo
    let name : String = "John"; // declares and initialize a string variable
    name ++ " Smith"            // rest of the expression
```
or
```ergo
    let x : Double = 3.1416     // declares and initialize a double variable
    sqrt(x)                     // rest of the expression
```

This can be useful to document your code, or to remember what type you expect from an expression.

The Ergo compiler will return a type error if the annotation is not consistent with
the expression that computes the value for that let binding. For instance, the
following will return a type error since `"pi!"` is not of type `Double`.

```ergo
    let x : Double = "pi!"; // TYPE ERROR: "pi!" is not a Double
    sqrt(x)
```

> Typing `return let x : Double = "pi!"; sqrt(x)` in the [Ergo
REPL](https://ergorepl.netlify.com), should answer a type error:
> ```text
> Type error (at line 1 col 7). The let type annotation Double for
> the name x does not match the actual type String.
> return let x : Double = "pi!"; sqrt(x)
>        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
> ```

This becomes particularly useful as your code becomes more complex. For instance
the following expression will also trigger a type error:

```ergo
    let rate = 3.5;
    let name : String =
      if rate > 0.0
      then 3.14         // TYPE ERROR: 3.14 is not a String
      else "John";
    name ++ " Smith"
```

Since not all the cases of the `if ... then ... else ...` expressions return a
value of type `String` which is the type annotation for the `name` variable.

---
id: logic-stmt
title: Statements

A clause's body is composed of statements. Statements are a special kind of
expression which can manipulate the contract state and emit obligations. Unlike
other expressions they may return a response or an error.

## Contract data

When inside a statement, data about the contract -- either the contract parameters,
clause parameters or contract state are available using the following Ergo
keywords:
```ergo
    contract   // The contract parameters (from a contract template)
    clause     // Local clause parameters (from a clause template)
    state      // The contract state
```

For instance, if your contract template parameters and state information have the
following types:

```ergo
    // Template parameters
    asset InstallmentSaleContract extends AccordContract {
      o AccordParty BUYER
      o AccordParty SELLER
      o Double INITIAL_DUE
      o Double INTEREST_RATE
      o Double TOTAL_DUE_BEFORE_CLOSING
      o Double MIN_PAYMENT
      o Double DUE_AT_CLOSING
      o Integer FIRST_MONTH
    }
    // Contract state
    enum ContractStatus {
      o WaitingForFirstDayOfNextMonth
      o Fulfilled
    }
    asset InstallmentSaleState extends AccordContractState {
      o ContractStatus status
      o Double balance_remaining
      o Integer next_payment_month
      o Double total_paid
   }
```

You can use the following expressions:
```ergo
    contract.BUYER
    state.balance_remaining
```

## Returning a response

Returning a response from a clause can be done by using a `return` statement:

```ergo
     return 1                       // Return the integer one
     return Payout{ amount: 39.99 } // Return a new Payout object
     return                         // Return nothing
```

> **TechNote:** the [Ergo REPL](https://ergorepl.netlify.com) takes statements as
input which is why we had to add `return` to expressions in previous examples.

## Returning a failure

Returning a failure from a clause can be done by using a `throw` statement:
```ergo
    throw ErgoErrorResponse{ message: "This is wrong" }
    define concept MyOwnError extends ErgoErrorResponse{ fee: Double }
    throw MyOwnError{ message: "This is wrong and costs a fee", fee: 29.99 }
```

For convenience, Ergo provides a `failure` function which takes a string as part of
its [standard library](ref-logic-stdlib#other-functions), so you can also write:
```ergo
    throw failure("This is wrong")
```

## Enforce statement

Before a contract is enforceable some preconditions must be satisfied:
- Competent parties who have the legal capacity to contract
- Lawful subject matter
- Mutuality of obligation
- Consideration

The constructs below will be used to determine if the preconditions have been met
and what actions to take if they are not

```test
Example Prose
    Do the parties have adequate funds to execute this contract?
```

One can check preconditions in a clause using enforce statements, as
follows:

```ergo
    enforce x >= 0.0                    // Condition
    else throw failure("not positive"); // Statement if condition is false
    return x+1.0                        // Statement if condition is true
```

The else part of the statement can be omitted in which case Ergo
returns an error by default.

```ergo
    enforce x >= 0.0;         // Condition
    return x+1.0              // Statement if condition is true
```

## Emitting obligations

When inside a clause or contract, you can emit (one or more) obligations as
follows:
```ergo
   emit PaymentObligation{ amount: 29.99, description: "12 red roses" };
   emit PaymentObligation{ amount: 19.99, description: "12 white tulips" };
   return
```

Note that `emit` is always terminated by a `;` followed by another statement.

To conditionally emit an obligation you must ensure that both the `then` and the
`else` branches
in your Ergo code have a `return` value. For example:

```ergo
  let response = VehiclePaymentResponse{ message: "A missed payment was declared
for " ++
      contract.buyer.partyId ++ ". There have been " ++ toString(newCounter) ++ "
missed payments." };

    if state.numMissedPayments > contract.numMissedPayments then
      emit DisableVehicle {
        vehicleId : contract.vehicleId,
        contract: contract,
        deadline: none,
        promisor: some(contract.buyer),
        promisee: some(contract.seller),
        numMissedPayments : newCounter
      };
      return response
    else
      return response
```

## Setting the contract state

When inside a clause or contract, you can create a contract state as follows:
```ergo
    set state InstallmentSaleState{
      stateId: "#1",
      status: "WaitingForFirstDayOfNextMonth",
      balance_remaining: contract.INITIAL_DUE,
      total_paid: 0.0,
      next_payment_month: contract.FIRST_MONTH
    };
    return
```

Note that `set state` is always terminated by a `;` followed by another statement.

Once the state is set, you can change its properties individually with the shorter:
```ergo
set state.total_paid = 100.0;
return
```

## Printing intermediate results

For debugging purposes a special `info` statement can be used in your contract logic. For instance, the following indicates that you would like the Ergo execution engine to print out the result of expression `state.status` on the standard output.

```ergo
    set state InstallmentSaleState{
      stateId: "#1",
      status: "WaitingForFirstDayOfNextMonth",
      balance_remaining: contract.INITIAL_DUE,
      total_paid: 0.0,
      next_payment_month: contract.FIRST_MONTH
    };
    info(state.status);     // Directive to print to standard output
    return
```

---
id: markup-ciceromark
title: CiceroMark

CiceroMark is an extension to CommonMark used to write the text in Accord Project contracts. The extension is limited in scope, and designed to help with parsing

~~~
clauses inside contracts and the result of formulas.

## Blocks

### Clause Blocks

Clause blocks can be used to identify a specific clause within a contract. Contract
blocks are written:
```
{{#clause clauseName}}
...Markdown of the clause...
{{/clause}}
```

### Example

For instance, the following is a valid contract text containing a payment clause:

```tem
## Copyright Notices.

Licensee shall ensure that its use of the Work is marked with the appropriate
copyright notices specified by Licensor in a reasonably prominent position in the
order and manner provided by Licensor. Licensee shall abide by the copyright laws
and what are considered to be sound practices for copyright notice provisions in
the Territory. Licensee shall not use any copyright notices that conflict with,
confuse, or negate the notices Licensor provides and requires hereunder.

{{#clause payment}}
Payment
-------
As consideration in full for the rights granted herein, Licensee shall pay Licensor
a one-time
fee in the amount of "one hundred US Dollars" (100.0 USD) upon execution of this
Agreement, payable as
follows: "bank transfer".
{{/clause}}

## General.

### Interpretation.

For purposes of this Agreement, (a) the words "include," "includes," and
"including" are deemed to be followed by the words "without limitation"; (b) the
word "or" is not exclusive; and (c) the words "herein," "hereof," "hereby,"
"hereto," and "hereunder" refer to this Agreement as a whole. This Agreement is
intended to be construed without regard to any presumption or rule requiring
construction or interpretation against the party drafting an instrument or causing
any instrument to be drafted.
```

## Ergo Formulas

Ergo formulas in template text are essentially similar to Excel formulas. They let
you create legal text dynamically based on the other variables in your contract.

If your contract contains the result of evaluating a formula, the corresponding
text should be written `{{% resultOfFormula %}}` where `resultOfFormula` is the
expected result of that formula.
~~~

~~~
### Example

For instance, the following is a valid contract text for the [fixed rate loan]
(https://templates.accordproject.org/fixed-interests@0.5.2.html) template:

```tem
## Fixed rate loan

This is a _fixed interest_ loan to the amount of £100,000.00
at the yearly interest rate of 2.5%
with a loan term of 15,
and monthly payments of {{%£667.00%}}
```

--------------------------------------------------------------------------------
---
id: markup-commonmark
title: CommonMark
---

The following CommonMark guide is non normative, but included for convenience. For
a more detailed introduction we refer the reader the [CommonMark
Webpage](https://commonmark.org/) and
[Specification](https://spec.commonmark.org/0.29/).

## Formatting

### Italics

To italicize text, add one asterisk `*` or underscore `_` both before and after the
relevant text.

##### Example

```md
_Donoghue v Stevenson_ is a landmark tort law case.
```
will be rendered as:

> _Donoghue v Stevenson_ is a landmark tort law case.

### Bold
To bold text, add two asterisks `**` or two underscores `__` both before and after
the relevant text.

##### Example

```md
**Price** is defined in the Appendix.
```

will be rendered as:

> **Price** is defined in the Appendix.


### Bold and Italic
To bold _and_ italicize text, add `***` both before and after the relevant text.
~~~

~~~~
##### Example

```md
***WARNING***: This product contains chemicals that may cause cancer.
```
will be rendered as:

> ***WARNING***: This product contains chemicals that may cause cancer.

## Paragraphs
To start a new paragraph, insert one or more blank lines. (In other words, all
paragraphs in markdown need to have one or more blank lines between them.)

##### Example

```md
This is the first paragraph.

This is the second paragraph.
This is not a third paragraph.
```
will be rendered as:

>This is the first paragraph.
>
>This is the second paragraph.
>This is not a third paragraph.


## Headings

### Using `#` (ATX Headings)

Level-1 through level-6 headings from are written with a `#` for each level.

#### Example

```md
# US Constitution
## Statutes enacted by Congress
### Rules promulgated by federal agencies
#### State constitution
##### Laws enacted by state legislature
###### Local laws and ordinances
```

will be rendered as:
> <h1>US Constitution</h1>
> <h2>Statutes enacted by Congress</h2>
> <h3>Rules promulgated by federal agencies</h3>
> <h4>State constitution</h4>
> <h5>Laws enacted by state legislature</h5>
> <h6>Local laws and ordinances</h6>

### Using `=` or `-` (Setext Headings)

Alternatively, headings with level 1 or 2 can be represented by using `=` and `-`
under the text of the heading.
~~~~

#### Example

```md
Linux Foundation
================

Accord Project
--------------
```

will be rendered as:
> <h1>Linux Foundation</h1>
> <h2>Accord Project</h2>

## Lists

### Unordered Lists
To create an unordered list, use asterisks `*`, plus `+`, or hyphens `-` in the beginning as list markers.

#### Example

```md
* Cicero
* Ergo
* Concerto
```

Will be rendered as:
>* Cicero
>* Ergo
>* Concerto

### Ordered Lists

To create an ordered list, use numbers followed by a period `.`.

#### Example

```md
1. One
2. Two
3. Three
```

will be rendered as:
>1. One
>2. Two
>3. Three

### Nested Lists

To create a list within another, indent each item in the sublist by four spaces.

#### Example
```md
1. Matters related to the business
    - enter into an agreement...
```

~~~
    - enter into any abnormal contracts...
2. Matters related to the assets
    - sell or otherwise dispose...
    - mortage, ...
```

will be rendered as:
>1. Matters related to the business
>    - enter into an agreement...
>    - enter into any abnormal contracts...
>2. Matters related to the assets
>    - sell or otherwise dispose...
>    - mortgage, ...

## Tables

To create a table, use pipes `|` to separate each column and use three or more
hyphens `---` for each column's header. For compatibility, you should not create a
table without a header and add also add a pipe on either end of a row.

#### Example

```md
| Header 1    | Header 2    |
| ----------- | ----------- |
| Column 1    | Column 2    |
```

will be rendered as

>| Header 1    | Header 2    |
>| ----------- | ----------- |
>| Column 1    | Column 2    |

It is not necessary to have identical cell widths for the whole table. The rendered
output will look the same irrespective of varying cell widths.

```md
| Header 1    | Header 2    |
| ---------| ---------|
| Column 1    | Column 2      |
```

will be rendered as

>| Header 1    | Header 2    |
>| ---------| ---------|
>| Column 1    | Column 2      |

### Formatting the Tables

A table can contain links, code (words or phrases in backticks (`) only) ,
formatted text (bold, italics) or images. However, adding lists, headings,
blockquotes, code blocks, horizontal rules or nested tables is not possible.

#### Example

```md
| Column1     | Column 2    |
~~~

| ----------- | ----------- |
text	![ap_logo](https://docs.accordproject.org/docs/assets/020/template.png "AP triangle")
\`\`\`code block\`\`\`	**Bold content**
[link](http://clause.io)	*Italics*
~~~
will be rendered as

>| Column1     | Column 2    |
>| ----------- | ----------- |
>| text | ![ap_logo](https://docs.accordproject.org/docs/assets/020/template.png "AP triangle")       |
>| \`\`\`code block\`\`\`   | **Bold content**     |
>| [link](http://clause.io) | *Italics* |

## Horizontal Rule

A horizontal rule may be used to create a "thematic break" between paragraph-level elements. In markdown, you can create a thematic break using either of the following:

* `___`: three consecutive underscores
* `---`: three consecutive dashes
* `***`: three consecutive asterisks

#### Example

```md
___
---
***
~~~

Will be rendered as:
>___
>
>---
>
>***

## Escaping

Any markdown character that is used for a special purpose may be _escaped_ by placing a backslash in front of it.

For instance avoid creating bold or italic when using `*` or `_` in a sentence, place a backslash `\` in the front, like: `\*` or `\_`.

#### Example

```md
This is \_not\_ italics but _this_ is!
```
Will be rendered as:
> This is _not_ italics but _this_ is!

<!--References:
Commonmark official page and tutorial: https://commonmark.org/help/

OpenLaw Beginner's Guide: https://docs.openlaw.io/beginners-guide/
Markdown cheatsheet: https://gist.github.com/jonschlinkert/5854601
Headings example:
http://www.nyc.gov/html/conflicts/downloads/pdf2/municipal_ethics_laws_ny_state/introduction_to_american_law.pdf
-->

---
id: markup-preliminaries
title: Preliminaries

## Markdown & CommonMark

The text for Accord Project templates is written using markdown. It builds on the [CommonMark](https://commonmark.org) standard so that any CommonMark document is valid text for a template or contract.

As with other markup languages, CommonMark can express the document structure (e.g., headings, paragraphs, lists) and formatting useful for readability (e.g., italics, bold, quotations).

The main reference is the [CommonMark Specification](https://spec.commonmark.org/0.29/) but you can find an overview of CommonMark main features in the [CommonMark](markup-commonmark.md) Section of this guide.

## Accord Project Extensions

Accord Project uses two extensions to CommonMark: CiceroMark for the contract text, and TemplateMark for the template grammar.

### Lexical Conventions

Accord Project contract or template text is a string of `UTF-8` characters.

:::note
By convention, CiceroMark files have the `.md` extensions, and TemplateMark files have the `.tem.md` extension.
:::

The two sequences of characters `{{` and `}}` are reserved and used for the CiceroMark and TemplateMark extensions to CommonMark. There are three kinds of extensions:
1. Variables (written `{{variableName}}`) which may include an optional formatting (written `{{variableName as "FORMAT"}}`).
2. Formulas (written `{{% expression %}}`).
3. Blocks which may contain additional text or markdown. Blocks come in two flavors:
   1. Blocks corresponding to [markdown inline elements](https://spec.commonmark.org/0.29/#inlines) which may contain only other markdown inline elements (e.g., text, emphasis, links). Those have to be written on a single line as follows:
      ```
      {{#blockName variableName}} ... text or markdown ... {{/blockName}}
      ```
   2. Blocks corresponding to [markdown container elements](https://spec.commonmark.org/0.29/#container-blocks) which may contain

arbitrary markdown elements (e.g., paragraphs, lists, headings). Those have to be
written with each opening and closing tags on their own line as follows:
      ```
      {{#blockName variableName}}
      ... text or markdown ...
      {{/blockName}}
      ```

### CiceroMark

CiceroMark is used to express the natural language text for legal clauses or
contracts. It uses two specific extensions to CommonMark to facilitate contract
parsing:
1. Clauses within a contract can be identified using a `clause` block:
   ```
   {{#clause clauseName}}
   text of the clause
   {{/clause}}
   ```
2. The result of formulas within a contract or clause can be identified using:
   ```
   {{% result_of_the_formula %}}
   ```

For instance, the following CiceroMark for a loan between `John Smith` and `Jane
Doe` includes a title (`Loan agreement`) followed by some text, followed by a fixed
rate interest clause. The clause contains the terms for the loan and the result of
calculating the monthly payment.
```tem
# Loan agreement

This is a loan agreement between "John Smith" and "Jane Doe", which shall be
entered into
by the parties on January 21, 2021 - 3 PM, except in the event of a force majeure.

{{#clause fixedRate}}
## Fixed rate loan

This is a _fixed interest_ loan to the amount of £100,000.00
at the yearly interest rate of 2.5%
with a loan term of 15,
and monthly payments of {{%£667.00%}}
{{/clause}}
```

More information and examples can be found in the [CiceroMark](markup-
ciceromark.md) part of this guide.

### TemplateMark

TemplateMark is used to describe families of contracts or clauses with some
variable parts. It is based on CommonMark with several extensions to indicate those
variables parts:
1. _Variables_: e.g., `{{loanAmount}}` indicates the amount for a loan.
2. _Template Blocks_: e.g., `{{#if forceMajeure}}, except in the event of a force
majeure{{/if}}` indicates some optional text in the contract.
3. _Formulas_: e.g., `{{% monthlyPaymentFormula(loanAmount,rate,loanDuration) %}}`
calculates a monthly payment based on the `loanAmount`, `rate`, and `loanDuration`
variables.

For instance, the following TemplateMark for a loan between a `borrower` and a `lender` includes a title (`Loan agreement`) followed by some text, followed by a fixed rate interest clause. This template allows for either taking force majeure into account or not, and calls into a formula to calculate the monthly payment.
```tem
# Loan agreement

This is a loan agreement between {{borrower}} and {{lender}}, which shall be entered into
by the parties on {{date as "MMMM DD, YYYY - h A"}}{{#if forceMajeure}}, except in the event of a force majeure{{/if}}.

{{#clause fixedRate}}
## Fixed rate loan

This is a _fixed interest_ loan to the amount of {{loanAmount as "K0,0.00"}}
at the yearly interest rate of {{rate}}%
with a loan term of {{loanDuration}},
and monthly payments of {{% monthlyPaymentFormula(loanAmount,rate,loanDuration) as "K0,0.00" %}}
{{/clause}}
```

More information and examples can be found in the [TemplateMark](markup-templatemark.md) part of this guide.

## Dingus

You can test your template or contract text using the [TemplateMark Dingus](https://templatemark-dingus.netlify.app), an online tool which lets you edit the markdown and see it rendered as HTML, or as a document object model.

![TemplateMark Dingus](assets/dingus1.png)

You can select whether to parse your text as pure CommonMark (i.e., according to the CommonMark specification), or with the CiceroMark or TemplateMark extensions.

![TemplateMark Dingus](assets/dingus2.png)

You can also inspect the HTML source, or the document object model (abstract syntax tree or AST), even see a pdf rendering for your template.

![TemplateMark Dingus](assets/dingus3.png)

For instance, you can open the TemplateMark from the loan example on this page by clicking [this link](https://templatemark-dingus.netlify.app/#md3=%7B%22source%22%3A%22%23%20Loan%20agreement%5Cn%5CnThis%20is%20a%20loan%20agreement%20between%20%7B%7Bborrower%7D%7D%20and%20%7B%7Blender%7D%7D%2C%20which%20shall%20be%20entered%20into%5Cnby%20the%20parties%20on%20%7B%7Bdate%20as%20%5C%22MMMMM%20DD%2C%20YYYY%20-%20hhA%5C%22%7D%7D%7B%7B%23if%20forceMajeure%7D%7D%2C%20except%20in%20the%20event%20of%20a%20force%20majeure%7B%7B%2Fif%7D%7D.%5Cn%5Cn%7B%7B%23clause%20fixedRate%7D%7D%5Cn%23%23%20Fixed%20rate%20loan%5Cn%5CnThis%20is%20a%20_fixed%20interest_%20loan%20to%20the%20amount%20of%20%7B%7BloanAmount%20as%20%5C%22K0%2C0.00%5C%22%7D%7D%5Cnat%20the%20yearly%20interest%20rate%20of%20%7B%7Brate%7D%7D%25%5Cnwith%20a%20loan%20term%20of%20%7B%7BloanDuration%7D%7D%2C%5Cnand%20monthly%20payments%20of%20%7B%7B%25%20monthlyPaymentFormula%28loanAmount%2Crate%2CloanDuration%29%20as%20%5C%22K0%2C0.00%5C%22%20%25%7D%7D%5Cn%7B%7B%2Fclause%7D%7D%5Cn%22%2C%22defaults%22%3A%7B%22templateMark%22%3Atrue%2C%22ciceroMark

%22%3Afalse%2C%22html%22%3Atrue%2C%22_highlight%22%3Atrue%2C%22_strict%22%3Afalse%2C%22_view%22%3A%22html%22%7D%7D).

![TemplateMark Dingus](assets/dingus4.png)

---
id: markup-templatemark
title: TemplateMark

TemplateMark is an extension to CommonMark used to write the text in Accord Project templates. The extension includes new markdown for variables, inline and container elements of the markdown and template formulas.

The kind of extension which can be used is based on the _type_ of the variable in the [Concerto Model](https://concerto.accordproject.org/docs/intro) for your template. For each type in your model differrent markdown elements apply: variable markdown for atomic types in the model, list blocks for array types in the model, optional blocks for optional types in the model, etc.

## Variables

Standard variables are written `{{variableName}}` where `variableName` is a variable declared in the model.

The following example shows a template text with three variables (`buyer`, `amount`, and `seller`):

```tem
Upon the signing of this Agreement, {{buyer}} shall pay {{amount}} to {{seller}}.
```

The way variables are handled (both during parsing and drafting) is based on their type.

### String Variable

#### Description

If the variable `variableName` has type `String` in the model:
```ergo
o String variableName
```
The corresponding instance should contain text between quotes (`"`).

#### Examples

For example, consider the following model:

```ergo
asset Template extends AccordClause {
  o String buyer
  o String supplier
}
```

the following instance text:

```md
This Supply Sales Agreement is made between "Steve Supplier" and "Betty Byer".
```

matches the template:
```tem
This Supply Sales Agreement is made between {{supplier}} and {{buyer}}.
```

while the following instance texts do not match:
```md
This Supply Sales Agreement is made between 2019 and 2020.
```
or
```md
This Supply Sales Agreement is made between Steve Supplier and Betty Byer.
```

### Numeric Variable

#### Description

If the variable `variableName` has type `Double`, `Integer` or `Long` in the model:
```ergo
o Double variableName
o Integer variableName2
o Long variableName3
```
The corresponding instance should contain the corresponding number.

#### Examples

For example, consider the following model:

```ergo
asset Template extends AccordClause {
  o Double penaltyPercentage
}
```

the following instance text:
```md
The penalty amount is 10.5% of the total value of the Equipment whose delivery has
been delayed.
```

matches the template:
```tem
The penalty amount is {{penaltyPercentage}}% of the total value of the Equipment
whose delivery has been delayed.
```

while the following instance texts do not match:
```md
The penalty amount is ten% of the total value of the Equipment whose delivery has
been delayed.
```
or
```md
```

~~~~
The penalty amount is "10.5"% of the total value of the Equipment whose delivery
has been delayed.
```

### Enum Variables

#### Description

If the variable `variableName` has an enumerated type:
```ergo
o EnumType variableName
```

The corresponding instance should contain a corresponding enumerated value without
quotes.

#### Examples

For example, consider the following model:
```ergo
import org.accordproject.money.CurrencyCode from
https://models.accordproject.org/money.cto
asset Template extends AccordClause {
  o CurrencyCode currency
}

```
the following instance text:
```md
Monetary amounts in this contract are denominated in USD.
```

matches the template:
```tem
Monetary amounts in this contract are denominated in {{currency}}.
```

while the following instance texts do not match:
```md
Monetary amounts in this contract are denominated in "USD".
```
or
```md
Monetary amounts in this contract are denominated in $.
```

## Formatted Variables

Formatted variables are written `{{variableName as "FORMAT"}}` where `variableName`
is a variable declared in the model and the `FORMAT` is a type-dependent
description for the syntax of the variables in the contract.

The following example shows a template text with one variable with a format
`DD/MM/YYYY`.

```tem
The contract was signed on {{contractDate as "DD/MM/YYYY"}}.
```
~~~~

### DateTime Variables

#### Description

If the variable `variableName` has type `DateTime`:
```ergo
o DateTime variableName
```
The corresponding instance should be a date and time, and can optionally be formatted. The default format is `MM/DD/YYYY`, commonly used in the US.

#### DateTime Formats

The textual representation of a DateTime can be customized by including an optional format string using the `as` keyword directly in a template grammar. The following formatting tokens are supported:

Tokens are case-sensitive.

Input	Example	Description
`YYYY`	`2014`	4 or 2 digit year
`M`	`12`	1 or 2 digit month number
`MM`	`04`	2 digit month number
`MMM`	`Feb.`	Short month name
`MMMM`	`December`	Long month name
`D`	`3`	1 or 2 digit day of month
`DD`	`04`	2 digit day of month
`H`	`3`	24 hours (1 or 2 digits)
`HH`	`04`	24 hours (2 digits)
`h`	`1`	12 hours (1 or 2 digits)
`hh`	`02`	12 hours (2 digits)
`a`	`am` or `pm`	morning/afternoon (lowercase)
`A`	`AM` or `PM`	morning/afternoon (uppercase)
`mm`	`59`	2 digit minutes
`ss`	`34`	2 digit seconds
`SSS`	`002`	3 digit milliseconds
`Z`	`+01:00`	UTC offset

:::note
If `Z` is specified, it must occur as the last token in the format string.
:::

#### Examples

The format of the `contractDate` variable of type `DateTime` can be specified with the `DD/MM/YYYY` format, as is commonly used in Europe.

```tem
The contract was signed on {{contractDate as "DD/MM/YYYY"}}.
The contract was signed on 26/04/2019.
```

Other examples:

```tem
dateTimeProperty: {{dateTimeProperty as "D MMM YYYY HH:mm:ss.SSSZ"}}
dateTimeProperty: 1 Jan 2018 05:15:20.123+01:02
```

```tem
dateTimeProperty: {{dateTimeProperty as "D MMMM YYYY HH:mm:ss.SSSZ"}}
dateTimeProperty: 1 January 2018 05:15:20.123+01:02
```

```tem
dateTimeProperty: {{dateTimeProperty as "D-M-YYYY H mm:ss.SSSZ"}}
dateTimeProperty: 31-12-2019 2 59:01.001+01:01
```

```tem
dateTimeProperty: {{dateTimeProperty as "DD/MM/YYYY"}}
dateTimeProperty: 01/12/2018
```

```tem
dateTimeProperty: {{dateTimeProperty as "DD-MMM-YYYY H mm:ss.SSSZ"}}
dateTimeProperty: 04-Jan-2019 2 59:01.001+01:01
```

### Amount Variables

#### Description

If the variable `variableName` is of type `Integer`, `Long`, `Double` or `MonetaryAmount`:
```ergo
o Integer integerVariable
o Long longVariable
o Double doubleVariable
o MonetaryAmount monetaryVariable
```

The corresponding instance should be a numeric value (with a currency code in the case of monetary amounts), and can optionally be formatted.

#### Amount Formats

The textual representation of an amount can be customized by including an optional format string using the `as` keyword directly in a template grammar. The following formatting tokens are supported:

Tokens are case-sensitive.

Input	Example	Description	Type Supported
`0,0`	`3,100,200`	integer part with `,` separator	Integer,Long,Double,MonetaryAmount
`0 0`	`3 100 200`	integer part with ` ` separator	Integer,Long,Double,MonetaryAmount
`0,0.00`	`3,100,200.95`	decimal with two digits precision	Double,MonetaryAmount
`0 0,00`	`3 100 200,95`	decimal with two digits precision	Double,MonetaryAmount
`0,0.0000`	`3,100,200.95`	decimal with four digits precision	Double,MonetaryAmount

`CCC`	`USD`	currency code	MonetaryAmount
`K`	`$`	currency symbol	MonetaryAmount

The general format for the amount is `0{sep}0({sep}0+)?` where `{sep}` is a single character (e.g., `,` or `.`). The first `{sep}` is used to separate every three digits of the integer part. The second `{sep}` is used as a decimal point. And the number of `0` after the second separator is used to indicate precision (number of digits after the decimal point).

#### Examples

The following examples show formating for `Integer` or `Long` values.

```
The manuscript shall be completed within {{days as "0,0"}} days.
The manuscript shall be completed within 1,001 days.
```

```
The manuscript shall contain at most {{words as "0 0"}} words.
The manuscript shall contain at most 1 500 001 words.
```

The following examples show formatting for `Double` values.

```
The effective range of the device should be at least {{distance as "0,0.00mm"}}.
The effective range of the device should be at least 1,250,400.99mm.
```

```
The effective range of the device should be at least {{distance as "0 0,0000mm"}}.
The effective range of the device should be at least 1 250 400,9900mm.
```

The following examples show formatting for `MonetaryAmount` values.

```
The loan principal is {{principal as "0,0.00 CCC"}}.
The loan principal is 2,000,500,000.00 GBP.
```

```
The loan principal is {{principal as "K0,0.00"}}.
The loan principal is £2,000,500,000.00.
```

```
The loan principal is {{principal as "0 0,00 K"}}.
The loan principal is 2 000 500 000,00 €.
```

## Complex Types Variables

### Duration Types

#### Description

If the variable `variableName` has type `Duration`:
```ergo
import org.accordproject.time.Duration
o Duration variableName
```

The corresponding instance should contain the corresponding duration written with
the amount as a number and the duration unit as literal text.

#### Examples

For example, consider the following model:
```ergo
asset Template extends AccordClause {
  o Duration termination
}
```

the following instance texts:
```md
If the delay is more than 15 days, the Buyer is entitled to terminate this
Contract.
```
and
```md
If the delay is more than 1 week, the Buyer is entitled to terminate this Contract.
```

both match the template:
```tem
If the delay is more than {{termination}}, the Buyer is entitled to terminate this
Contract.
```

while the following instance texts do not match:
```md
If the delay is more than a month, the Buyer is entitled to terminate this
Contract.
```
or
```md
If the delay is more than "two weeks", the Buyer is entitled to terminate this
Contract.
```

### Other Complex Types

#### Description

If the variable `variableName` has a complex type `ComplexType` (such as an
`asset`, a `concept`, etc.)
```ergo
o ComplexType variableName
```

The corresponding instance should contain all fields in the corresponding complex
type in the order they occur in the model, separated by a single white space

~~~
character.

#### Examples

For example, consider the following model:
```ergo
import org.accordproject.address.PostalAddress from
https://models.accordproject.org/address.cto
asset Template extends AccordClause {
  o PostalAddress address
}
```

the following instance text:
```md
Address of the supplier: "555 main street" "10290" "" "NY" "New York" "10001".
```

matches the template:
```tem
Address of the supplier: {{address}}.
```

Consider the following model:
```md
import org.accordproject.money.MonetaryAmount from
https://models.accordproject.org/money.cto
asset Template extends AccordClause {
  o MonetaryAmount amount
}
```

the following instance text:
```md
Total value of the goods: 50.0 USD.
```

matches the template:
```tem
Total value of the goods: {{amount}}.
```

## Inline Blocks

CiceroMark uses blocks to enable more advanced scenarios, to handle optional or
repeated text (e.g., lists), to change the variables in scope for a given section
of the text, etc. Inline blocks correspond to inline elements in the markdown.

Inline blocks always have the following syntactic structure:

```tem
{{#blockName variableName parameters}}...{{/blockName}}
```

where `blockName` indicates which kind of block it is (e.g., conditional block or
optional block), `variableName` indicates the template variable which is in scope
within the block. For certain blocks, additional `parameters` can be passed to
control the behavior of that block (e.g., the `join` block creates text from a list
with an optional separator).
~~~

### Conditional Blocks

Conditional blocks enables text which depends on a value of a `Boolean` variable in your model:

```tem
{{#if forceMajeure}}This is a force majeure{{/if}}
```

Conditional blocks can also include an `else` branch to indicate that some other text should be use when the value of the variable is `false`:

```tem
{{#if forceMajeure}}This is a force majeure{{else}}This is *not* a force
majeure{{/if}}
```

#### Examples

Drafting text with the first conditional block above using the following JSON data:
```json
{
  "$class": "org.accordproject.foo.Status",
  "forceMajeure": true
}
```

results in the following markdown text:

```md
This is a force majeure
```

Drafting text with this block using the following JSON data:
```json
{
  "$class": "org.accordproject.foo.Status",
  "forceMajeure": false
}
```

results in the following markdown text:

```md

```

### Optional Blocks

Optional blocks enable text which depends on the presence or absence of an `optional` variable in your model:

```tem
{{#optional forceMajeure}}This applies except for Force Majeure cases in a
{{miles}} miles radius.{{/optional}}
```

Optional blocks can also include an `else` branch to indicate that some other text should be use when the value of the variable is absent (`null` in the JSON data):

```tem
{{#optional forceMajeure}}This applies except for Force Majeure cases in a
{{miles}} miles radius.{{else}}This applies even in case a force
majeure.{{/optional}}
```

#### Examples

Drafting text with the second optional block above using the following JSON data:
```json
{
  "$class": "org.accordproject.foo.Status",
  "forceMajeure": {
    "$class": "org.accordproject.foo.Distance",
    "miles": 250
  }
}
```

results in the following markdown text:

```md
This applies except for Force Majeure cases in a 250 miles radius.
```

Drafting text with this block using the following JSON data:
```json
{
  "$class": "org.accordproject.foo.Status",
  "forceMajeure": null
}
```

results in the following markdown text:

```md
This applies even in case a force majeure.
```

### With Blocks

A `with` block can be used to change variables that are in scope in a specific part of a template grammar:

```tem
For the Tenant: {{#with tenant}}{{partyId}}, domiciled at {{address}}{{/with}}
For the Landlord: {{#with landlord}}{{partyId}}, domiciled at {{address}}{{/with}}
```

#### Example

Drafting text with this block using the following JSON data:
```json
{
  "$class": "org.accordproject.rentaldeposit.RentalDepositClause",
  "contractId": "31d817e2-d62a-4b70-b395-acd0d5da09f5",
  "tenant": {
```

~~~~
    "$class": "org.accordproject.rentaldeposit.RentalParty",
    "partyId": "Michael",
    "address": "111, main street"
  }
  ...
}
```

results in the following markdown text:

```md
For the Tenant: "Michael", domiciled at "111, main street"
For the Landlord: "Parsa", domiciled at "222, chestnut road"
```

### Join Blocks

A `join` block can be used to iterate over a variable containing an array of
values, and can use an (optional) separator.

```tem
Discount applies to the following items: {{#join items separator=", "}}{{name}}
({{id}}){{/join}}.
```

#### Example

Drafting text with this block using the following JSON data:
```json
{
  "$class": "org.accordproject.sale.Order",
  "contractId": "31d817e2-d62a-4b70-b395-acd0d5da09f5",
  "items": [{
      "$class": "org.accordproject.slate.Item",
      "id": "111",
      "name": "Pineapple"
    },{
      "$class": "org.accordproject.slate.Item",
      "id": "222",
      "name": "Strawberries"
    },{
      "$class": "org.accordproject.slate.Item",
      "id": "333",
      "name": "Pomegranate"
    }
  ]
}
```

results in the following markdown text:

```md
Discount applies to the following items: Pineapple (111), Strawberries (222),
Pomegranate (333).
```

## Container Blocks

CiceroMark uses block expressions to enable more advanced scenarios, to handle
~~~~

optional or repeated text (e.g., lists), to change the variables in scope for a given section of the text, etc.

Container blocks always have the following syntactic structure:

```tem
{{#blockName variableName parameters}}
...
{{/blockName}}
```

where `blockName` indicates which kind of block it is (e.g., conditional block or list block), `variableName` indicates the template variable which is in scope within the block. For certain blocks, additional `parameters` can be passed to control the behavior of that block (e.g., the `join` block creates text from a list with an optional separator).

### Unordered Lists

```tem
{{#ulist rates}}
{{volumeAbove}}$ M<= Volume < {{volumeUpTo}}$ M : {{rate}}%
{{/ulist}}
```

#### Example

Drafting text with this block using the following JSON data:
```json
{
  "$class": "org.accordproject.volumediscountlist.VolumeDiscountContract",
  "contractId": "19243313-adc2-4ff1-aa41-993816ed2cdc",
  "rates": [
    {
      "$class": "org.accordproject.volumediscountlist.RateRange",
      "volumeUpTo": 1,
      "volumeAbove": 0,
      "rate": 3.1
    },
    {
      "$class": "org.accordproject.volumediscountlist.RateRange",
      "volumeUpTo": 10,
      "volumeAbove": 1,
      "rate": 3.1
    },
    {
      "$class": "org.accordproject.volumediscountlist.RateRange",
      "volumeUpTo": 50,
      "volumeAbove": 10,
      "rate": 2.9
    }
  ]
}
```

results in the following markdown text:

```md
- 0.0$ M <= Volume < 1.0$ M : 3.1%
```

~~~
- 1.0$ M <= Volume < 10.0$ M : 3.1%
- 10.0$ M <= Volume < 50.0$ M : 2.9%
```

### Ordered Lists

```tem
{{#olist rates}}
{{volumeAbove}}$ M <= Volume < {{volumeUpTo}}$ M : {{rate}}%
{{/olist}}
```

#### Example

Drafting text with this block using the following JSON data:
```json
{
  "$class": "org.accordproject.volumediscountlist.VolumeDiscountContract",
  "contractId": "19243313-adc2-4ff1-aa41-993816ed2cdc",
  "rates": [
    {
      "$class": "org.accordproject.volumediscountlist.RateRange",
      "volumeUpTo": 1,
      "volumeAbove": 0,
      "rate": 3.1
    },
    {
      "$class": "org.accordproject.volumediscountlist.RateRange",
      "volumeUpTo": 10,
      "volumeAbove": 1,
      "rate": 3.1
    },
    {
      "$class": "org.accordproject.volumediscountlist.RateRange",
      "volumeUpTo": 50,
      "volumeAbove": 10,
      "rate": 2.9
    }
  ]
}
```

results in the following markdown text:
```md
1. 0.0$ M <= Volume < 1.0$ M : 3.1%
2. 1.0$ M <= Volume < 10.0$ M : 3.1%
3. 10.0$ M <= Volume < 50.0$ M : 2.9%
```

### Clause Blocks

Clause blocks can be used to include a clause template within a contract template:

```tem
Payment
-------
{{#clause payment}}
As consideration in full for the rights granted herein, Licensee shall pay Licensor
a one-time
~~~

~~~
fee in the amount of {{amountText}} ({{amount}}) upon execution of this Agreement,
payable as
follows: {{paymentProcedure}}.
{{/clause}}
```

#### Example

Drafting text with this block using the following JSON data:
```json
{
  "$class": "org.accordproject.copyrightlicense.CopyrightLicenseContract",
  "contractId": "944535e8-213c-4649-9e60-cc062cce24e8",
  ...
  "paymentClause": {
    "$class": "org.accordproject.copyrightlicense.PaymentClause",
    "clauseId": "6c7611dc-410c-4134-a9ec-17fb6aad5607",
    "amountText": "one hundred US Dollars",
    "amount": {
      "$class": "org.accordproject.money.MonetaryAmount",
      "doubleValue": 100,
      "currencyCode": "USD"
    },
    "paymentProcedure": "bank transfer"
  }
}
```

results in the following markdown text:

```md
Payment
----

As consideration in full for the rights granted herein, Licensee shall pay Licensor
a one-time
fee in the amount of "one hundred US Dollars" (100.0 USD) upon execution of this
Agreement, payable as
follows: "bank transfer".

```

## Ergo Formulas

Ergo formulas in template text are essentially similar to Excel formulas, and
enable to create legal text dynamically, based on the other variables in your
contract. They are written `{{% ergoExpression %}}` where `ergoExpression` is any
valid [Ergo Expression](logic-ergo).

::: note
Formulas allow the template developer to generate arbitrary contract text from
other contract and clause variables. They therefore cannot be used to set a
template model variable during parsing. In other words formulas are evaluated when
drafting a contract but are ignored when parsing the contract text.
:::

### Evaluation Context

The context in which expressions within templates text are evaluated includes:
~~~

- The contract variables, which can be accessed using the variable name (or `contract.variableName`)
- All constants or functions declared or imported in the main [Ergo module](logic-module) for your template.

#### Fixed Interests Clause

For instance, let us look one more time at [fixed rate loan](https://templates.accordproject.org/fixed-interests-static@0.2.0.html) clause that was used previously:

```tem
## Fixed rate loan

This is a *fixed interest* loan to the amount of {{loanAmount}}
at the yearly interest rate of {{rate}}%
with a loan term of {{loanDuration}},
and monthly payments of {{% monthlyPaymentFormula(loanAmount,rate,loanDuration) %}}
```

The [`logic` directory](https://github.com/accordproject/cicero-template-library/tree/master/src/fixed-interests/logic) for that template includes two Ergo modules:
```
./logic/interests.ergo  // Module containing the monthlyPaymentFormula function
./logic/logic.ergo      // Main module
```

A look inside the `logic.ergo` module shows the corresponding import, which ensures the `monthlyPaymentFormula` function is also in scope in the text for the template:
```
namespace org.accordproject.interests

import org.accordproject.loan.interests.*

contract Interests over TemplateModel {
  ...
}
```

### Examples

Ergo provides a wide range of capabilities which you can use to construct the text that should be included in the final clause or contract. Below are a few examples for illustrations, but we encourage you to consult the [Ergo Logic](logic-ergo) guide for a more comprehensive overview of Ergo.

#### Path expressions

The contents of complex values can be accessed using the `.` notation.

For instance the following template uses the `.` notation to access the first name and last name of the contract author.

```tem
This contract was drafted by {{% author.name.firstName %}} {{% author.name.lastName
%}}
```

#### Built-in Functions

Ergo offers a number of pre-defined functions for a variety of primitive types.
Please consult the [Ergo Standard Library](ref-logic-stdlib) reference for the
complete list of built-in functions.

For instance the following template uses the `addPeriod` function to automatically
include the date at which a lease expires in the text of the contract:

```tem
This lease was signed on {{signatureDate}}, and is valid for a {{leaseTerm}}
period.
This lease will expire on {{% addPeriod(signatureDate, leaseTerm) %}}`
```

#### Iterators

Ergo's `foreach` expressions lets you iterate over collections of values.

For instance the following template uses a `foreach` expression combined with the
`avg` built-in function to include the average product price in the text of the
contract:

```tem
The average price of the products included in this purchase
order is {{% avg(foreach p in products return p.price) %}}.
```

#### Conditionals

Conditional expressions lets you include alternative text based on arbitrary
conditions.

For instance, the following template uses a conditional expression to indicate the
governing jurisdiction:

```tem
Each party hereby irrevocably agrees that process may be served on it in
any manner authorized by the Laws of {{%
    if address.country = US and getYear(now()) > 1959
    then "the State of " ++ address.state
    else "the Country of " ++ address.country
%}}
```

---
id: ref-cicero-api
title: Cicero API

## Modules

<dl>
<dt><a href="#module_cicero-engine">cicero-engine</a></dt>
<dd><p>Clause Engine</p>
</dd>
<dt><a href="#module_cicero-core">cicero-core</a></dt>
<dd><p>Cicero Core - defines the core data types for Cicero.</p>

```
</dd>
</dl>

## Classes

<dl>
<dt><a href="#Clause">Clause</a></dt>
<dd><p>A Clause is executable business logic, linked to a natural language (legally
enforceable) template.
A Clause must be constructed with a template and then prior to execution the data
for the clause must be set.
Set the data for the clause (an instance of the template model) by either calling
the setData method or by
calling the parse method and passing in natural language text that conforms to the
template grammar.</p>
</dd>
<dt><a href="#Contract">Contract</a></dt>
<dd><p>A Contract is executable business logic, linked to a natural language
(legally enforceable) template.
A Clause must be constructed with a template and then prior to execution the data
for the clause must be set.
Set the data for the clause (an instance of the template model) by either calling
the setData method or by
calling the parse method and passing in natural language text that conforms to the
template grammar.</p>
</dd>
<dt><a href="#Metadata">Metadata</a></dt>
<dd><p>Defines the metadata for a Template, including the name, version, README
markdown.</p>
</dd>
<dt><a href="#Template">Template</a></dt>
<dd><p>A template for a legal clause or contract. A Template has a template model,
request/response transaction types,
a template grammar (natural language for the template) as well as Ergo code for the
business logic of the
template.</p>
</dd>
<dt><a href="#TemplateInstance">TemplateInstance</a></dt>
<dd><p>A TemplateInstance is an instance of a Clause or Contract template. It is
executable business logic, linked to
a natural language (legally enforceable) template.
A TemplateInstance must be constructed with a template and then prior to execution
the data for the clause must be set.
Set the data for the TemplateInstance by either calling the setData method or by
calling the parse method and passing in natural language text that conforms to the
template grammar.</p>
</dd>
<dt><a href="#CompositeArchiveLoader">CompositeArchiveLoader</a></dt>
<dd><p>Manages a set of archive loaders, delegating to the first archive
loader that accepts a URL.</p>
</dd>
</dl>

## Functions

<dl>
<dt><a href="#isPNG">isPNG(buffer)</a> ⇒ <code>Boolean</code></dt>
<dd><p>Checks whether the file is PNG</p>
</dd>
```

```
<dt><a href="#getMimeType">getMimeType(buffer)</a> ⇒ <code>Object</code></dt>
<dd><p>Returns the mime-type of the file</p>
</dd>
</dl>

<a name="module_cicero-engine"></a>

## cicero-engine
Clause Engine


* [cicero-engine](#module_cicero-engine)
    * [.Engine](#module_cicero-engine.Engine)
        * [new Engine()](#new_module_cicero-engine.Engine_new)
        * [.trigger(clause, request, state, [currentTime], [utcOffset])]
(#module_cicero-engine.Engine+trigger) ⇒ <code>Promise</code>
        * [.invoke(clause, clauseName, params, state, [currentTime], [utcOffset])]
(#module_cicero-engine.Engine+invoke) ⇒ <code>Promise</code>
        * [.init(clause, [currentTime], [utcOffset], params)](#module_cicero-
engine.Engine+init) ⇒ <code>Promise</code>
        * [.getErgoEngine()](#module_cicero-engine.Engine+getErgoEngine) ⇒
<code>ErgoEngine</code>

<a name="module_cicero-engine.Engine"></a>

### cicero-engine.Engine
<p>
Engine class. Stateless execution of clauses against a request object, returning a
response to the caller.
</p>

**Kind**: static class of [<code>cicero-engine</code>](#module_cicero-engine)
**Access**: public

* [.Engine](#module_cicero-engine.Engine)
    * [new Engine()](#new_module_cicero-engine.Engine_new)
    * [.trigger(clause, request, state, [currentTime], [utcOffset])]
(#module_cicero-engine.Engine+trigger) ⇒ <code>Promise</code>
    * [.invoke(clause, clauseName, params, state, [currentTime], [utcOffset])]
(#module_cicero-engine.Engine+invoke) ⇒ <code>Promise</code>
    * [.init(clause, [currentTime], [utcOffset], params)](#module_cicero-
engine.Engine+init) ⇒ <code>Promise</code>
    * [.getErgoEngine()](#module_cicero-engine.Engine+getErgoEngine) ⇒
<code>ErgoEngine</code>

<a name="new_module_cicero-engine.Engine_new"></a>

#### new Engine()
Create the Engine.

<a name="module_cicero-engine.Engine+trigger"></a>

#### engine.trigger(clause, request, state, [currentTime], [utcOffset]) ⇒
<code>Promise</code>
Send a request to a clause for execution

**Kind**: instance method of [<code>Engine</code>](#module_cicero-engine.Engine)
**Returns**: <code>Promise</code> - a promise that resolves to a result for the
clause
```

```
| Param | Type | Description |
| --- | --- | --- |
| clause | [<code>Clause</code>](#Clause) | the clause |
| request | <code>object</code> | the request, a JS object that can be deserialized
using the Composer serializer. |
| state | <code>object</code> | the contract state, a JS object that can be
deserialized using the Composer serializer. |
| [currentTime] | <code>string</code> | the definition of 'now', defaults to
current time |
| [utcOffset] | <code>number</code> | UTC Offset for this execution, defaults to
local offset |

<a name="module_cicero-engine.Engine+invoke"></a>

#### engine.invoke(clause, clauseName, params, state, [currentTime], [utcOffset]) ⇒
<code>Promise</code>
Invoke a specific clause for execution

**Kind**: instance method of [<code>Engine</code>](#module_cicero-engine.Engine)
**Returns**: <code>Promise</code> - a promise that resolves to a result for the
clause

| Param | Type | Description |
| --- | --- | --- |
| clause | [<code>Clause</code>](#Clause) | the clause |
| clauseName | <code>string</code> | the clause name |
| params | <code>object</code> | the clause parameters, a JS object whose fields
that can be deserialized using the Composer serializer. |
| state | <code>object</code> | the contract state, a JS object that can be
deserialized using the Composer serializer. |
| [currentTime] | <code>string</code> | the definition of 'now', defaults to
current time |
| [utcOffset] | <code>number</code> | UTC Offset for this execution, defaults to
local offset |

<a name="module_cicero-engine.Engine+init"></a>

#### engine.init(clause, [currentTime], [utcOffset], params) ⇒ <code>Promise</code>
Initialize a clause

**Kind**: instance method of [<code>Engine</code>](#module_cicero-engine.Engine)
**Returns**: <code>Promise</code> - a promise that resolves to a result for the
clause initialization

| Param | Type | Description |
| --- | --- | --- |
| clause | [<code>Clause</code>](#Clause) | the clause |
| [currentTime] | <code>string</code> | the definition of 'now', defaults to
current time |
| [utcOffset] | <code>number</code> | UTC Offset for this execution, defaults to
local offset |
| params | <code>object</code> | the clause parameters, a JS object whose fields
that can be deserialized using the Composer serializer. |

<a name="module_cicero-engine.Engine+getErgoEngine"></a>

#### engine.getErgoEngine() ⇒ <code>ErgoEngine</code>
Provides access to the underlying Ergo engine.
```

```
**Kind**: instance method of [<code>Engine</code>](#module_cicero-engine.Engine)
**Returns**: <code>ErgoEngine</code> - the Ergo Engine for this Engine
<a name="module_cicero-core"></a>

## cicero-core
Cicero Core - defines the core data types for Cicero.

<a name="Clause"></a>

## Clause
A Clause is executable business logic, linked to a natural language (legally
enforceable) template.
A Clause must be constructed with a template and then prior to execution the data
for the clause must be set.
Set the data for the clause (an instance of the template model) by either calling
the setData method or by
calling the parse method and passing in natural language text that conforms to the
template grammar.

**Kind**: global class
**Access**: public
<a name="Contract"></a>

## Contract
A Contract is executable business logic, linked to a natural language (legally
enforceable) template.
A Clause must be constructed with a template and then prior to execution the data
for the clause must be set.
Set the data for the clause (an instance of the template model) by either calling
the setData method or by
calling the parse method and passing in natural language text that conforms to the
template grammar.

**Kind**: global class
**Access**: public
<a name="Metadata"></a>

## Metadata
Defines the metadata for a Template, including the name, version, README markdown.

**Kind**: global class
**Access**: public

* [Metadata](#Metadata)
    * [new Metadata(packageJson, readme, samples, request, logo)]
(#new_Metadata_new)
    * _instance_
        * [.getTemplateType()](#Metadata+getTemplateType) ⇒ <code>number</code>
        * [.getLogo()](#Metadata+getLogo) ⇒ <code>Buffer</code>
        * [.getAuthor()](#Metadata+getAuthor) ⇒ <code>\*</code>
        * [.getRuntime()](#Metadata+getRuntime) ⇒ <code>string</code>
        * [.getCiceroVersion()](#Metadata+getCiceroVersion) ⇒ <code>string</code>
        * [.satisfiesCiceroVersion(version)](#Metadata+satisfiesCiceroVersion) ⇒
<code>string</code>
        * [.getSamples()](#Metadata+getSamples) ⇒ <code>object</code>
        * [.getRequest()](#Metadata+getRequest) ⇒ <code>object</code>
        * [.getSample(locale)](#Metadata+getSample) ⇒ <code>string</code>
        * [.getREADME()](#Metadata+getREADME) ⇒ <code>String</code>
        * [.getPackageJson()](#Metadata+getPackageJson) ⇒ <code>object</code>
        * [.getName()](#Metadata+getName) ⇒ <code>string</code>
        * [.getDisplayName()](#Metadata+getDisplayName) ⇒ <code>string</code>
        * [.getKeywords()](#Metadata+getKeywords) ⇒ <code>Array</code>
        * [.getDescription()](#Metadata+getDescription) ⇒ <code>string</code>
        * [.getVersion()](#Metadata+getVersion) ⇒ <code>string</code>
        * [.getIdentifier()](#Metadata+getIdentifier) ⇒ <code>string</code>
        * [.createTargetMetadata(runtimeName)](#Metadata+createTargetMetadata) ⇒
<code>object</code>
        * [.toJSON()](#Metadata+toJSON) ⇒ <code>object</code>
    * _static_
        * [.checkImage(buffer)](#Metadata.checkImage)
        * [.checkImageDimensions(buffer, mimeType)](#Metadata.checkImageDimensions)

<a name="new_Metadata_new"></a>

### new Metadata(packageJson, readme, samples, request, logo)
Create the Metadata.
<p>
<strong>Note: Only to be called by framework code. Applications should
retrieve instances from [Template](#Template)</strong>
</p>


| Param | Type | Description |
| --- | --- | --- |
| packageJson | <code>object</code> | the JS object for package.json (required) |
| readme | <code>String</code> | the README.md for the template (may be null) |
| samples | <code>object</code> | the sample markdown for the template in different
locales, |
| request | <code>object</code> | the JS object for the sample request |
| logo | <code>Buffer</code> | the bytes data for the image represented as an
object whose keys are the locales and whose values are the sample markdown. For
example:  {      default: 'default sample markdown',      en: 'sample text in
english',      fr: 'exemple de texte français'  } Locale keys (with the exception
of default) conform to the IETF Language Tag specification (BCP 47). THe `default`
key represents sample template text in a non-specified language, stored in a file
called `sample.md`. |

<a name="Metadata+getTemplateType"></a>

### metadata.getTemplateType() ⇒ <code>number</code>
Returns either a 0 (for a contract template), or 1 (for a clause template)

**Kind**: instance method of [<code>Metadata</code>](#Metadata)
**Returns**: <code>number</code> - the template type
<a name="Metadata+getLogo"></a>

### metadata.getLogo() ⇒ <code>Buffer</code>
Returns the logo at the root of the template

**Kind**: instance method of [<code>Metadata</code>](#Metadata)
**Returns**: <code>Buffer</code> - the bytes data of logo
<a name="Metadata+getAuthor"></a>

### metadata.getAuthor() ⇒ <code>\*</code>
Returns the author for this template.

**Kind**: instance method of [<code>Metadata</code>](#Metadata)
```

Returns: <code>*</code> - the author information
<a name="Metadata+getRuntime"></a>

### metadata.getRuntime() ⇒ <code>string</code>
Returns the name of the runtime target for this template, or null if this template
has not been compiled for a specific runtime.

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>string</code> - the name of the runtime
<a name="Metadata+getCiceroVersion"></a>

### metadata.getCiceroVersion() ⇒ <code>string</code>
Returns the version of Cicero that this template is compatible with.
i.e. which version of the runtime was this template built for?
The version string conforms to the semver definition

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>string</code> - the semantic version
<a name="Metadata+satisfiesCiceroVersion"></a>

### metadata.satisfiesCiceroVersion(version) ⇒ <code>string</code>
Only returns true if the current cicero version satisfies the target version of
this template

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>string</code> - the semantic version

Param	Type	Description
version	<code>string</code>	the cicero version to check against

<a name="Metadata+getSamples"></a>

### metadata.getSamples() ⇒ <code>object</code>
Returns the samples for this template.

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>object</code> - the sample files for the template
<a name="Metadata+getRequest"></a>

### metadata.getRequest() ⇒ <code>object</code>
Returns the sample request for this template.

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>object</code> - the sample request for the template
<a name="Metadata+getSample"></a>

### metadata.getSample(locale) ⇒ <code>string</code>
Returns the sample for this template in the given locale. This may be null.
If no locale is specified returns the default sample if it has been specified.

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>string</code> - the sample file for the template in the given
locale or null

Param	Type	Default	Description
locale	<code>string</code>	<code>null</code>	the IETF language code for the language.

<a name="Metadata+getREADME"></a>

### metadata.getREADME() ⇒ <code>String</code>
Returns the README.md for this template. This may be null if the template does not
have a README.md

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>String</code> - the README.md file for the template or null
<a name="Metadata+getPackageJson"></a>

### metadata.getPackageJson() ⇒ <code>object</code>
Returns the package.json for this template.

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>object</code> - the Javascript object for package.json
<a name="Metadata+getName"></a>

### metadata.getName() ⇒ <code>string</code>
Returns the name for this template.

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>string</code> - the name of the template
<a name="Metadata+getDisplayName"></a>

### metadata.getDisplayName() ⇒ <code>string</code>
Returns the display name for this template.

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>string</code> - the display name of the template
<a name="Metadata+getKeywords"></a>

### metadata.getKeywords() ⇒ <code>Array</code>
Returns the keywords for this template.

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>Array</code> - the keywords of the template
<a name="Metadata+getDescription"></a>

### metadata.getDescription() ⇒ <code>string</code>
Returns the description for this template.

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>string</code> - the description of the template
<a name="Metadata+getVersion"></a>

### metadata.getVersion() ⇒ <code>string</code>
Returns the version for this template.

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>string</code> - the description of the template
<a name="Metadata+getIdentifier"></a>

### metadata.getIdentifier() ⇒ <code>string</code>
Returns the identifier for this template, formed from name@version.

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>string</code> - the identifier of the template
<a name="Metadata+createTargetMetadata"></a>

```markdown
### metadata.createTargetMetadata(runtimeName) ⇒ <code>object</code>
Return new Metadata for a target runtime

**Kind**: instance method of [<code>Metadata</code>](#Metadata)
**Returns**: <code>object</code> - the new Metadata

| Param | Type | Description |
| --- | --- | --- |
| runtimeName | <code>string</code> | the target runtime name |

<a name="Metadata+toJSON"></a>

### metadata.toJSON() ⇒ <code>object</code>
Return the whole metadata content, for hashing

**Kind**: instance method of [<code>Metadata</code>](#Metadata)
**Returns**: <code>object</code> - the content of the metadata object
<a name="Metadata.checkImage"></a>

### Metadata.checkImage(buffer)
Check the buffer is a png file with the right size

**Kind**: static method of [<code>Metadata</code>](#Metadata)

| Param | Type | Description |
| --- | --- | --- |
| buffer | <code>Buffer</code> | the buffer object |

<a name="Metadata.checkImageDimensions"></a>

### Metadata.checkImageDimensions(buffer, mimeType)
Checks if dimensions for the image are correct.

**Kind**: static method of [<code>Metadata</code>](#Metadata)

| Param | Type | Description |
| --- | --- | --- |
| buffer | <code>Buffer</code> | the buffer object |
| mimeType | <code>string</code> | the mime type of the object |

<a name="Template"></a>

## *Template*
A template for a legal clause or contract. A Template has a template model,
request/response transaction types,
a template grammar (natural language for the template) as well as Ergo code for the
business logic of the
template.

**Kind**: global abstract class
**Access**: public

* *[Template](#Template)*
    * *[new Template(packageJson, readme, samples, request, logo, options,
authorSignature)](#new_Template_new)*
    * _instance_
        * *[.validate(options)](#Template+validate)*
        * *[.getTemplateModel()](#Template+getTemplateModel) ⇒
```

<code>ClassDeclaration</code>*
        * *[.getIdentifier()](#Template+getIdentifier) ⇒ <code>String</code>*
        * *[.getMetadata()](#Template+getMetadata) ⇒ [<code>Metadata</code>]
(#Metadata)*
        * *[.getName()](#Template+getName) ⇒ <code>String</code>*
        * *[.getDisplayName()](#Template+getDisplayName) ⇒ <code>string</code>*
        * *[.getVersion()](#Template+getVersion) ⇒ <code>String</code>*
        * *[.getDescription()](#Template+getDescription) ⇒ <code>String</code>*
        * *[.getHash()](#Template+getHash) ⇒ <code>string</code>*
        * *[.verifyTemplateSignature()](#Template+verifyTemplateSignature)*
        * *[.signTemplate(p12File, passphrase, timestamp)](#Template+signTemplate)*
        * *[.toArchive([language], [options])](#Template+toArchive) ⇒
<code>Promise.&lt;Buffer&gt;</code>*
        * *[.getParserManager()](#Template+getParserManager) ⇒
<code>ParserManager</code>*
        * *[.getLogicManager()](#Template+getLogicManager) ⇒
<code>LogicManager</code>*
        * *[.getIntrospector()](#Template+getIntrospector) ⇒
<code>Introspector</code>*
        * *[.getFactory()](#Template+getFactory) ⇒ <code>Factory</code>*
        * *[.getSerializer()](#Template+getSerializer) ⇒ <code>Serializer</code>*
        * *[.getRequestTypes()](#Template+getRequestTypes) ⇒ <code>Array</code>*
        * *[.getResponseTypes()](#Template+getResponseTypes) ⇒ <code>Array</code>*
        * *[.getEmitTypes()](#Template+getEmitTypes) ⇒ <code>Array</code>*
        * *[.getStateTypes()](#Template+getStateTypes) ⇒ <code>Array</code>*
        * *[.hasLogic()](#Template+hasLogic) ⇒ <code>boolean</code>*
    * _static_
        * *[.fromDirectory(path, [options])](#Template.fromDirectory) ⇒
[<code>Promise.&lt;Template&gt;</code>](#Template)*
        * *[.fromArchive(buffer, [options])](#Template.fromArchive) ⇒
[<code>Promise.&lt;Template&gt;</code>](#Template)*
        * *[.fromUrl(url, [options])](#Template.fromUrl) ⇒ <code>Promise</code>*
        * *[.instanceOf(classDeclaration, fqt)](#Template.instanceOf) ⇒
<code>boolean</code>*

<a name="new_Template_new"></a>

### *new Template(packageJson, readme, samples, request, logo, options,
authorSignature)*
Create the Template.
Note: Only to be called by framework code. Applications should
retrieve instances from [fromArchive](#Template.fromArchive) or [fromDirectory]
(#Template.fromDirectory).

Param	Type	Description
packageJson	<code>object</code>	the JS object for package.json
readme	<code>String</code>	the readme in markdown for the template (optional)
samples	<code>object</code>	the sample text for the template in different
locales
request	<code>object</code>	the JS object for the sample request
logo	<code>Buffer</code>	the bytes data of logo
options	<code>Object</code>	e.g., { warnings: true }
authorSignature	<code>Object</code>	object containing template hash,
timestamp, author's certificate, signature

<a name="Template+validate"></a>

### *template.validate(options)*
Verifies that the template is well formed.
Compiles the Ergo logic.
Throws an exception with the details of any validation errors.

Kind: instance method of [<code>Template</code>](#Template)

Param	Type	Description
options	<code>Object</code>	e.g., { verify: true }

<a name="Template+getTemplateModel"></a>

### *template.getTemplateModel() ⇒ <code>ClassDeclaration</code>*
Returns the template model for the template

Kind: instance method of [<code>Template</code>](#Template)
Returns: <code>ClassDeclaration</code> - the template model for the template
Throws:

- <code>Error</code> if no template model is found, or multiple template models are found

<a name="Template+getIdentifier"></a>

### *template.getIdentifier() ⇒ <code>String</code>*
Returns the identifier for this template

Kind: instance method of [<code>Template</code>](#Template)
Returns: <code>String</code> - the identifier of this template
<a name="Template+getMetadata"></a>

### *template.getMetadata() ⇒ [<code>Metadata</code>](#Metadata)*
Returns the metadata for this template

Kind: instance method of [<code>Template</code>](#Template)
Returns: [<code>Metadata</code>](#Metadata) - the metadata for this template
<a name="Template+getName"></a>

### *template.getName() ⇒ <code>String</code>*
Returns the name for this template

Kind: instance method of [<code>Template</code>](#Template)
Returns: <code>String</code> - the name of this template
<a name="Template+getDisplayName"></a>

### *template.getDisplayName() ⇒ <code>string</code>*
Returns the display name for this template.

Kind: instance method of [<code>Template</code>](#Template)
Returns: <code>string</code> - the display name of the template
<a name="Template+getVersion"></a>

### *template.getVersion() ⇒ <code>String</code>*
Returns the version for this template

Kind: instance method of [<code>Template</code>](#Template)
Returns: <code>String</code> - the version of this template. Use semver module

to parse.
<a name="Template+getDescription"></a>

### *template.getDescription() ⇒ <code>String</code>*
Returns the description for this template

Kind: instance method of [<code>Template</code>](#Template)
Returns: <code>String</code> - the description of this template
<a name="Template+getHash"></a>

### *template.getHash() ⇒ <code>string</code>*
Gets a content based SHA-256 hash for this template. Hash
is based on the metadata for the template plus the contents of
all the models and all the script files.

Kind: instance method of [<code>Template</code>](#Template)
Returns: <code>string</code> - the SHA-256 hash in hex format
<a name="Template+verifyTemplateSignature"></a>

### *template.verifyTemplateSignature()*
verifies the signature stored in the template object using the template hash and
timestamp

Kind: instance method of [<code>Template</code>](#Template)
<a name="Template+signTemplate"></a>

### *template.signTemplate(p12File, passphrase, timestamp)*
signs a string made up of template hash and time stamp using private key derived
from the keystore

Kind: instance method of [<code>Template</code>](#Template)

Param	Type	Description
p12File	<code>String</code>	encoded string of p12 keystore file
passphrase	<code>String</code>	passphrase for the keystore file
timestamp	<code>Number</code>	timestamp of the moment of signature is done

<a name="Template+toArchive"></a>

### *template.toArchive([language], [options]) ⇒
<code>Promise.&lt;Buffer&gt;</code>*
Persists this template to a Cicero Template Archive (cta) file.

Kind: instance method of [<code>Template</code>](#Template)
Returns: <code>Promise.&lt;Buffer&gt;</code> - the zlib buffer

Param	Type	Description
[language]	<code>string</code>	target language for the archive (should be 'ergo')
[options]	<code>Object</code>	JSZip options and keystore object containing path and passphrase for the keystore

<a name="Template+getParserManager"></a>

### *template.getParserManager() ⇒ <code>ParserManager</code>*
Provides access to the parser manager for this template.
The parser manager can convert template data to and from

natural language text.

Kind: instance method of [<code>Template</code>](#Template)
Returns: <code>ParserManager</code> - the ParserManager for this template
<a name="Template+getLogicManager"></a>

### *template.getLogicManager() ⇒ <code>LogicManager</code>*
Provides access to the template logic for this template.
The template logic encapsulate the code necessary to
execute the clause or contract.

Kind: instance method of [<code>Template</code>](#Template)
Returns: <code>LogicManager</code> - the LogicManager for this template
<a name="Template+getIntrospector"></a>

### *template.getIntrospector() ⇒ <code>Introspector</code>*
Provides access to the Introspector for this template. The Introspector
is used to reflect on the types defined within this template.

Kind: instance method of [<code>Template</code>](#Template)
Returns: <code>Introspector</code> - the Introspector for this template
<a name="Template+getFactory"></a>

### *template.getFactory() ⇒ <code>Factory</code>*
Provides access to the Factory for this template. The Factory
is used to create the types defined in this template.

Kind: instance method of [<code>Template</code>](#Template)
Returns: <code>Factory</code> - the Factory for this template
<a name="Template+getSerializer"></a>

### *template.getSerializer() ⇒ <code>Serializer</code>*
Provides access to the Serializer for this template. The Serializer
is used to serialize instances of the types defined within this template.

Kind: instance method of [<code>Template</code>](#Template)
Returns: <code>Serializer</code> - the Serializer for this template
<a name="Template+getRequestTypes"></a>

### *template.getRequestTypes() ⇒ <code>Array</code>*
Provides a list of the input types that are accepted by this Template. Types use
the fully-qualified form.

Kind: instance method of [<code>Template</code>](#Template)
Returns: <code>Array</code> - a list of the request types
<a name="Template+getResponseTypes"></a>

### *template.getResponseTypes() ⇒ <code>Array</code>*
Provides a list of the response types that are returned by this Template. Types use
the fully-qualified form.

Kind: instance method of [<code>Template</code>](#Template)
Returns: <code>Array</code> - a list of the response types
<a name="Template+getEmitTypes"></a>

### *template.getEmitTypes() ⇒ <code>Array</code>*
Provides a list of the emit types that are emitted by this Template. Types use the
fully-qualified form.

```
**Kind**: instance method of [<code>Template</code>](#Template)
**Returns**: <code>Array</code> - a list of the emit types
<a name="Template+getStateTypes"></a>

### *template.getStateTypes() ⇒ <code>Array</code>*
Provides a list of the state types that are expected by this Template. Types use
the fully-qualified form.

**Kind**: instance method of [<code>Template</code>](#Template)
**Returns**: <code>Array</code> - a list of the state types
<a name="Template+hasLogic"></a>

### *template.hasLogic() ⇒ <code>boolean</code>*
Returns true if the template has logic, i.e. has more than one script file.

**Kind**: instance method of [<code>Template</code>](#Template)
**Returns**: <code>boolean</code> - true if the template has logic
<a name="Template.fromDirectory"></a>

### *Template.fromDirectory(path, [options]) ⇒
[<code>Promise.&lt;Template&gt;</code>](#Template)*
Builds a Template from the contents of a directory.
The directory must include a package.json in the root (used to specify
the name, version and description of the template).

**Kind**: static method of [<code>Template</code>](#Template)
**Returns**: [<code>Promise.&lt;Template&gt;</code>](#Template) - a Promise to the
instantiated template

| Param | Type | Default | Description |
| --- | --- | --- | --- |
| path | <code>String</code> |  | to a local directory |
| [options] | <code>Object</code> | <code></code> | an optional set of options to
configure the instance. |

<a name="Template.fromArchive"></a>

### *Template.fromArchive(buffer, [options]) ⇒
[<code>Promise.&lt;Template&gt;</code>](#Template)*
Create a template from an archive.

**Kind**: static method of [<code>Template</code>](#Template)
**Returns**: [<code>Promise.&lt;Template&gt;</code>](#Template) - a Promise to the
template

| Param | Type | Default | Description |
| --- | --- | --- | --- |
| buffer | <code>Buffer</code> |  | the buffer to a Cicero Template Archive (cta)
file |
| [options] | <code>Object</code> | <code></code> | an optional set of options to
configure the instance. |

<a name="Template.fromUrl"></a>

### *Template.fromUrl(url, [options]) ⇒ <code>Promise</code>*
Create a template from an URL.

**Kind**: static method of [<code>Template</code>](#Template)
**Returns**: <code>Promise</code> - a Promise to the template
```

Param	Type	Default	Description
url	<code>String</code>		the URL to a Cicero Template Archive (cta) file
[options]	<code>Object</code>	<code></code>	an optional set of options to configure the instance.

<a name="Template.instanceOf"></a>

### *Template.instanceOf(classDeclaration, fqt) ⇒ <code>boolean</code>*
Check to see if a ClassDeclaration is an instance of the specified fully qualified
type name.

Kind: static method of [<code>Template</code>](#Template)
Returns: <code>boolean</code> - True if classDeclaration an instance of the specified fully
qualified type name, false otherwise.
Internal:

Param	Type	Description
classDeclaration	<code>ClassDeclaration</code>	The class to test
fqt	<code>String</code>	The fully qualified type name.

<a name="TemplateInstance"></a>

## *TemplateInstance*
A TemplateInstance is an instance of a Clause or Contract template. It is executable business logic, linked to
a natural language (legally enforceable) template.
A TemplateInstance must be constructed with a template and then prior to execution the data for the clause must be set.
Set the data for the TemplateInstance by either calling the setData method or by calling the parse method and passing in natural language text that conforms to the template grammar.

Kind: global abstract class
Access: public

* *[TemplateInstance](#TemplateInstance)*
    * *[new TemplateInstance(template)](#new_TemplateInstance_new)*
    * _instance_
        * *[.setData(data)](#TemplateInstance+setData)*
        * *[.getData()](#TemplateInstance+getData) ⇒ <code>object</code>*
        * *[.getEngine()](#TemplateInstance+getEngine) ⇒ <code>object</code>*
        * *[.getDataAsConcertoObject()](#TemplateInstance+getDataAsConcertoObject) ⇒ <code>object</code>*
        * *[.parse(input, [currentTime], [utcOffset], [fileName])](#TemplateInstance+parse)*
        * *[.draft([options], [currentTime], [utcOffset])](#TemplateInstance+draft) ⇒ <code>string</code>*
        * *[.formatCiceroMark(ciceroMarkParsed, options, format)](#TemplateInstance+formatCiceroMark) ⇒ <code>string</code>*
        * *[.getIdentifier()](#TemplateInstance+getIdentifier) ⇒ <code>String</code>*
        * *[.getTemplate()](#TemplateInstance+getTemplate) ⇒ [<code>Template</code>](#Template)*
        * *[.getLogicManager()](#TemplateInstance+getLogicManager) ⇒ <code>LogicManager</code>*

        * *[.toJSON()](#TemplateInstance+toJSON) ⇒ <code>object</code>*
    * _static_
        * *[.ciceroFormulaEval(logicManager, clauseId, ergoEngine, name)](#TemplateInstance.ciceroFormulaEval) ⇒ <code>*</code>*
        * *[.rebuildParser(parserManager, logicManager, ergoEngine, templateName, grammar)](#TemplateInstance.rebuildParser)*

<a name="new_TemplateInstance_new"></a>

### *new TemplateInstance(template)*
Create the Clause and link it to a Template.

Param	Type	Description
template	[<code>Template</code>](#Template)	the template for the clause

<a name="TemplateInstance+setData"></a>

### *templateInstance.setData(data)*
Set the data for the clause

Kind: instance method of [<code>TemplateInstance</code>](#TemplateInstance)

Param	Type	Description
data	<code>object</code>	the data for the clause, must be an instance of the template model for the clause's template. This should be a plain JS object and will be deserialized and validated into the Concerto object before assignment.

<a name="TemplateInstance+getData"></a>

### *templateInstance.getData() ⇒ <code>object</code>*
Get the data for the clause. This is a plain JS object. To retrieve the Concerto object call getConcertoData().

Kind: instance method of [<code>TemplateInstance</code>](#TemplateInstance)
Returns: <code>object</code> - - the data for the clause, or null if it has not been set
<a name="TemplateInstance+getEngine"></a>

### *templateInstance.getEngine() ⇒ <code>object</code>*
Get the current Ergo engine

Kind: instance method of [<code>TemplateInstance</code>](#TemplateInstance)
Returns: <code>object</code> - - the data for the clause, or null if it has not been set
<a name="TemplateInstance+getDataAsConcertoObject"></a>

### *templateInstance.getDataAsConcertoObject() ⇒ <code>object</code>*
Get the data for the clause. This is a Concerto object. To retrieve the
plain JS object suitable for serialization call toJSON() and retrieve the `data`
property.

Kind: instance method of [<code>TemplateInstance</code>](#TemplateInstance)
Returns: <code>object</code> - - the data for the clause, or null if it has not been set
<a name="TemplateInstance+parse"></a>

```
### *templateInstance.parse(input, [currentTime], [utcOffset], [fileName])*
Set the data for the clause by parsing natural language text.

**Kind**: instance method of [<code>TemplateInstance</code>](#TemplateInstance)

| Param | Type | Description |
| --- | --- | --- |
| input | <code>string</code> | the text for the clause |
| [currentTime] | <code>string</code> | the definition of 'now', defaults to
current time |
| [utcOffset] | <code>number</code> | UTC Offset for this execution, defaults to
local offset |
| [fileName] | <code>string</code> | the fileName for the text (optional) |

<a name="TemplateInstance+draft"></a>

### *templateInstance.draft([options], [currentTime], [utcOffset]) ⇒
<code>string</code>*
Generates the natural language text for a contract or clause clause; combining the
text from the template
and the instance data.

**Kind**: instance method of [<code>TemplateInstance</code>](#TemplateInstance)
**Returns**: <code>string</code> - the natural language text for the contract or
clause; created by combining the structure of
the template with the JSON data for the clause.

| Param | Type | Description |
| --- | --- | --- |
| [options] | <code>\*</code> | text generation options. |
| [currentTime] | <code>string</code> | the definition of 'now', defaults to
current time |
| [utcOffset] | <code>number</code> | UTC Offset for this execution, defaults to
local offset |

<a name="TemplateInstance+formatCiceroMark"></a>

### *templateInstance.formatCiceroMark(ciceroMarkParsed, options, format) ⇒
<code>string</code>*
Format CiceroMark

**Kind**: instance method of [<code>TemplateInstance</code>](#TemplateInstance)
**Returns**: <code>string</code> - the result of parsing and printing back the text

| Param | Type | Description |
| --- | --- | --- |
| ciceroMarkParsed | <code>object</code> | the parsed CiceroMark DOM |
| options | <code>object</code> | parameters to the formatting |
| format | <code>string</code> | to the text generation |

<a name="TemplateInstance+getIdentifier"></a>

### *templateInstance.getIdentifier() ⇒ <code>String</code>*
Returns the identifier for this clause. The identifier is the identifier of
the template plus '-' plus a hash of the data for the clause (if set).

**Kind**: instance method of [<code>TemplateInstance</code>](#TemplateInstance)
**Returns**: <code>String</code> - the identifier of this clause
<a name="TemplateInstance+getTemplate"></a>
```

### *templateInstance.getTemplate() ⇒ [<code>Template</code>](#Template)*
Returns the template for this clause

Kind: instance method of [<code>TemplateInstance</code>](#TemplateInstance)
Returns: [<code>Template</code>](#Template) - the template for this clause
<a name="TemplateInstance+getLogicManager"></a>

### *templateInstance.getLogicManager() ⇒ <code>LogicManager</code>*
Returns the template logic for this clause

Kind: instance method of [<code>TemplateInstance</code>](#TemplateInstance)
Returns: <code>LogicManager</code> - the template for this clause
<a name="TemplateInstance+toJSON"></a>

### *templateInstance.toJSON() ⇒ <code>object</code>*
Returns a JSON representation of the clause

Kind: instance method of [<code>TemplateInstance</code>](#TemplateInstance)
Returns: <code>object</code> - the JS object for serialization
<a name="TemplateInstance.ciceroFormulaEval"></a>

### *TemplateInstance.ciceroFormulaEval(logicManager, clauseId, ergoEngine, name) ⇒ <code>*</code>*
Constructs a function for formula evaluation based for this template instance

Kind: static method of [<code>TemplateInstance</code>](#TemplateInstance)
Returns: <code>*</code> - A function from formula code + input data to result

Param	Type	Description
logicManager	<code>*</code>	the logic manager
clauseId	<code>string</code>	this instance identifier
ergoEngine	<code>*</code>	the evaluation engine
name	<code>string</code>	the name of the formula

<a name="TemplateInstance.rebuildParser"></a>

### *TemplateInstance.rebuildParser(parserManager, logicManager, ergoEngine, templateName, grammar)*
Utility to rebuild a parser when the grammar changes

Kind: static method of [<code>TemplateInstance</code>](#TemplateInstance)

Param	Type	Description
parserManager	<code>*</code>	the parser manager
logicManager	<code>*</code>	the logic manager
ergoEngine	<code>*</code>	the evaluation engine
templateName	<code>string</code>	this template name
grammar	<code>string</code>	the new grammar

<a name="CompositeArchiveLoader"></a>

## CompositeArchiveLoader
Manages a set of archive loaders, delegating to the first archive
loader that accepts a URL.

Kind: global class

* [CompositeArchiveLoader](#CompositeArchiveLoader)
    * [new CompositeArchiveLoader()](#new_CompositeArchiveLoader_new)
    * [.addArchiveLoader(archiveLoader)](#CompositeArchiveLoader+addArchiveLoader)
    * [.clearArchiveLoaders()](#CompositeArchiveLoader+clearArchiveLoaders)
    * *[.accepts(url)](#CompositeArchiveLoader+accepts) ⇒ <code>boolean</code>*
    * [.load(url, options)](#CompositeArchiveLoader+load) ⇒ <code>Promise</code>

<a name="new_CompositeArchiveLoader_new"></a>

### new CompositeArchiveLoader()
Create the CompositeArchiveLoader. Used to delegate to a set of ArchiveLoaders.

<a name="CompositeArchiveLoader+addArchiveLoader"></a>

### compositeArchiveLoader.addArchiveLoader(archiveLoader)
Adds a ArchiveLoader implemenetation to the ArchiveLoader

Kind: instance method of [<code>CompositeArchiveLoader</code>]
(#CompositeArchiveLoader)

Param	Type	Description
archiveLoader	<code>ArchiveLoader</code>	The archive to add to the CompositeArchiveLoader

<a name="CompositeArchiveLoader+clearArchiveLoaders"></a>

### compositeArchiveLoader.clearArchiveLoaders()
Remove all registered ArchiveLoaders

Kind: instance method of [<code>CompositeArchiveLoader</code>]
(#CompositeArchiveLoader)
<a name="CompositeArchiveLoader+accepts"></a>

### *compositeArchiveLoader.accepts(url) ⇒ <code>boolean</code>*
Returns true if this ArchiveLoader can process the URL

Kind: instance abstract method of [<code>CompositeArchiveLoader</code>]
(#CompositeArchiveLoader)
Returns: <code>boolean</code> - true if this ArchiveLoader accepts the URL

Param	Type	Description
url	<code>string</code>	the URL

<a name="CompositeArchiveLoader+load"></a>

### compositeArchiveLoader.load(url, options) ⇒ <code>Promise</code>
Load a Archive from a URL and return it

Kind: instance method of [<code>CompositeArchiveLoader</code>]
(#CompositeArchiveLoader)
Returns: <code>Promise</code> - a promise to the Archive

Param	Type	Description
url	<code>string</code>	the url to get
options	<code>object</code>	additional options

~~~~
<a name="isPNG"></a>

## isPNG(buffer) ⇒ <code>Boolean</code>
Checks whether the file is PNG

**Kind**: global function
**Returns**: <code>Boolean</code> - whether the file in PNG

| Param | Type | Description |
| --- | --- | --- |
| buffer | <code>Buffer</code> | buffer of the file |

<a name="getMimeType"></a>

## getMimeType(buffer) ⇒ <code>Object</code>
Returns the mime-type of the file

**Kind**: global function
**Returns**: <code>Object</code> - the mime-type of the file

| Param | Type | Description |
| --- | --- | --- |
| buffer | <code>Buffer</code> | buffer of the file |


--------------------------------------------------------------------------------
---
id: ref-cicero-cli
title: Command Line
---

Install the `@accordproject/cicero-cli` npm package to access the Cicero command
line interface (CLI). After installation you can use the `cicero` command and its
sub-commands as described below.

To install the Cicero CLI:
```
npm install -g @accordproject/cicero-cli
```

## Usage

```md
cicero <cmd> [args]

Commands:
  cicero parse       parse a contract text
  cicero draft       create contract text from data
  cicero normalize   normalize markdown (parse & redraft)
  cicero trigger     send a request to the contract
  cicero invoke      invoke a clause of the contract
  cicero initialize  initialize a clause
  cicero archive     create a template archive
  cicero compile     generate code for a target platform
  cicero get         save local copies of external dependencies

Options:
  --version      Show version number                                   [boolean]
~~~~

~~~
  --verbose, -v                                                 [default: false]
  --help         Show help                                             [boolean]
```

## cicero parse

`cicero parse` loads a template from a directory on disk and then parses input
clause (or contract) text using the template. If successful, the template model is
printed to console. If there are syntax errors, the line and column and error
information are printed.

```md
cicero parse

parse a contract text

Options:
  --version      Show version number                                   [boolean]
  --verbose, -v                                                 [default: false]
  --help         Show help                                             [boolean]
  --template     path to the template                                   [string]
  --sample       path to the contract text                              [string]
  --output       path to the output file                                [string]
  --currentTime  set current time                       [string] [default: null]
  --utcOffset    set UTC offset                         [number] [default: null]
  --offline      do not resolve external models       [boolean] [default: false]
  --warnings     print warnings                       [boolean] [default: false]
```

## cicero draft

`cicero draft` creates contract text from data.

```md
cicero draft

create contract text from data

Options:
  --version           Show version number                              [boolean]
  --verbose, -v                                                 [default: false]
  --help              Show help                                        [boolean]
  --template          path to the template                              [string]
  --data              path to the contract data                         [string]
  --output            path to the output file                           [string]
  --currentTime       set current time                  [string] [default: null]
  --utcOffset         set UTC offset                    [number] [default: null]
  --offline           do not resolve external models  [boolean] [default: false]
  --format            target format                                     [string]
  --unquoteVariables  remove variables quoting        [boolean] [default: false]
  --warnings          print warnings                  [boolean] [default: false]
```

## cicero normalize

`cicero normalize` normalizes markdown text by parsing and redrafting the text.

```md
cicero normalize
~~~

~~~
normalize markdown (parse & redraft)

Options:
  --version           Show version number                              [boolean]
  --verbose, -v                                                 [default: false]
  --help              Show help                                        [boolean]
  --template          path to the template                              [string]
  --sample            path to the contract text                         [string]
  --overwrite         overwrite the contract text     [boolean] [default: false]
  --output            path to the output file                           [string]
  --currentTime       set current time                  [string] [default: null]
  --utcOffset         set UTC offset                    [number] [default: null]
  --offline           do not resolve external models  [boolean] [default: false]
  --warnings          print warnings                  [boolean] [default: false]
  --format            target format                                     [string]
  --unquoteVariables  remove variables quoting        [boolean] [default: false]
```

## cicero trigger

`cicero trigger` sends a request to the contract.

```md
cicero trigger

send a request to the contract

Options:
  --version      Show version number                                   [boolean]
  --verbose, -v                                                 [default: false]
  --help         Show help                                             [boolean]
  --template     path to the template                                   [string]
  --sample       path to the contract text                              [string]
  --request      path to the JSON request                                [array]
  --state        path to the JSON state                                 [string]
  --currentTime  set current time                       [string] [default: null]
  --utcOffset    set UTC offset                         [number] [default: null]
  --offline      do not resolve external models       [boolean] [default: false]
  --warnings     print warnings                       [boolean] [default: false]
```

## cicero invoke

`cicero invoke` invokes a specific clause (`--clauseName`) of the contract.

```md
cicero invoke

invoke a clause of the contract

Options:
  --version      Show version number                                   [boolean]
  --verbose, -v                                                 [default: false]
  --help         Show help                                             [boolean]
  --template     path to the template                                   [string]
  --sample       path to the contract text                              [string]
  --clauseName   the name of the clause to invoke                       [string]
  --params       path to the parameters                                 [string]
~~~

~~~
  --state        path to the JSON state                                 [string]
  --currentTime  set current time                       [string] [default: null]
  --utcOffset    set UTC offset                         [number] [default: null]
  --offline      do not resolve external models       [boolean] [default: false]
  --warnings     print warnings                       [boolean] [default: false]
```

## cicero initialize

`cicero initialize` initializes a clause.

```md
cicero initialize

initialize a clause

Options:
  --version      Show version number                                   [boolean]
  --verbose, -v                                                 [default: false]
  --help         Show help                                             [boolean]
  --template     path to the template                                   [string]
  --sample       path to the contract text                              [string]
  --params       path to the parameters                                 [string]
  --currentTime  initialize with this current time      [string] [default: null]
  --utcOffset    set UTC offset                         [number] [default: null]
  --offline      do not resolve external models       [boolean] [default: false]
  --warnings     print warnings                       [boolean] [default: false]
```

## cicero archive

`cicero archive` creates a Cicero Template Archive (`.cta`) file from a template
stored in a local directory.

```md
cicero archive

create a template archive

Options:
  --version      Show version number                                   [boolean]
  --verbose, -v                                                 [default: false]
  --help         Show help                                             [boolean]
  --template     path to the template                                   [string]
  --target       the target language of the archive   [string] [default: "ergo"]
  --output       file name for new archive              [string] [default: null]
  --warnings     print warnings                       [boolean] [default: false]
```

## cicero compile

`cicero compile` generates code for a target platform. It loads a template from a
directory on disk and then attempts to generate versions of the template model in
the specified format. The available formats include: `Go`, `PlantUML`,
`Typescript`, `Java`, and `JSONSchema`.

```md
cicero compile
~~~

~~~text
generate code for a target platform

Options:
  --version      Show version number                                   [boolean]
  --verbose, -v                                                 [default: false]
  --help         Show help                                             [boolean]
  --template     path to the template                                   [string]
  --target       target of the code generation  [string] [default: "JSONSchema"]
  --output       path to the output directory    [string] [default: "./output/"]
  --warnings     print warnings                       [boolean] [default: false]
```

## cicero get

`cicero get` saves local copies of external dependencies.

```md
cicero get

save local copies of external dependencies

Options:
  --version      Show version number                                   [boolean]
  --verbose, -v                                                 [default: false]
  --help         Show help                                             [boolean]
  --template     path to the template                                   [string]
  --output       output directory path                                  [string]
```

--------------------------------------------------------------------------------
---
id: ref-cicero-testing
title: Template Testing
---

Cicero uses [Cucumber](https://cucumber.io/docs) for writing template tests, which
provides a human readable syntax.

This documents the syntax available to write Cicero tests.

## Test Structure

Tests are located in the `./test/` directory for each template, which contains
files with the `.feature` extension.

Each file has the following structure:

```gherkin
Feature: Name of the template being tested
  Description for the test

  Background:
    Given that the contract says
"""
Text of the contract instance.
"""

  Scenario: Description for scenario 1
    [[First Scenario Sequence]]
~~~

```
  Scenario: Description for scenario 2
    [[Second Scenario Sequence]]

etc.
```

Each scenario can be thought of as a description for the behavior of the clause or contract template for the contract given as background.

Each scenario corresponds to one call to the contract. I.e., for a given current time, request and contract state, it says what the expected result of executing the contract should be. This can be either:
- A response, a new contract state, and a list of emitted obligations
- An error

## Scenarios

A complete scenario is described in the [Gherkin Syntax](https://cucumber.io/docs/gherkin/reference/) through a sequence of **Step**.

Each step starts with a keyword, either **Given**, **When**, **And**, or **Then**:

- **Given**, **When** and **And** are used to specify the input for a call to the contract;
- **Then** and **And** are used to specify the expected result.

### Request and Response

The simplest kind of scenario specifies the response expected for a given request.

For instance, the following scenario describes the expected response for a given request to the [helloworld template](https://templates.accordproject.org/helloworld@0.10.1.html):

```gherkin
  Scenario: The contract should say Hello to Betty Buyer, from the ACME Corporation
    When it receives the request
"""
{
    "$class": "org.accordproject.helloworld.MyRequest",
    "input": "ACME Corporation"
}
"""
    Then it should respond with
"""
{
    "$class": "org.accordproject.helloworld.MyResponse",
    "output": "Hello Betty Buyer ACME Corporation"
}
"""
```

Both the request and the response are written inside triple quotes `"""` using JSON. If the request or response is not valid wrt. to the data model, this will result in a failing test.

:::warning

While the syntax for each scenario uses _pseudo_ natural language (e.g., `When it
receives the request`), the tests use very specific sentences as illustrated in
this guide.
:::

### Defaults

You can use the sample contract `sample.txt` and request `request.json` provided
with a template by using specific steps.

For instance, the following scenario describes the expected response for the
default contract text when sending the default request to the [helloworld template]
(https://templates.accordproject.org/helloworld@0.10.1.html):
```gherkin
Feature: HelloWorld
  This describe the expected behavior for the Accord Project's "Hello World!"
contract

  Background:
    Given the default contract

  Scenario: The contract should say Hello to Fred Blogs, from the Accord Project,
for the default request
    When it receives the default request
    Then it should respond with
"""
{
    "$class": "org.accordproject.helloworld.MyResponse",
    "output": "Hello Fred Blogs Accord Project"
}
"""
```

### Errors

Whenever appropriate, it is good practice to include both successful executions, as
well as scenarios for cases when a call to a template might fail. This can be
written using a **Then** step that describes the error.

For instance, the following scenario describes an expected error for a given
request to the [Interest Rate Swap](https://templates.accordproject.org/interest-
rate-swap@0.4.1.html) template:
```gherkin
Feature: Interest Rate Swap
  This describes the expected behavior for the Accord Project's interest rate swap
contract

  Background:
    Given that the contract says
"""
INTEREST RATE SWAP TRANSACTION LETTER AGREEMENT
"Deutsche Bank"

Date: 06/30/2005
To: "MagnaChip Semiconductor S.A."
Attention: Swaps Documentation Department
Our Reference: "Global No. N397355N"
Re: Interest Rate Swap Transaction
```

Ladies and Gentlemen:

The purpose of this letter agreement is to set forth the terms and conditions of the Transaction entered into between "Deutsche Bank" and "MagnaChip Semiconductor S.A." (“Counterparty”) on the Trade Date specified below (the “Transaction”). This letter agreement constitutes a “Confirmation” as referred to in the Agreement specified below.

The definitions and provisions contained in the 2000 ISDA Definitions (the “Definitions”) as published by the International Swaps and Derivatives Association, Inc. are incorporated by reference herein. In the event of any inconsistency between the Definitions and this Confirmation, this Confirmation will govern.

For the purpose of this Confirmation, all references in the Definitions or the Agreement to a “Swap Transaction” shall be deemed to be references to this Transaction.

1. This Confirmation evidences a complete and binding agreement between "Deutsche Bank" (“Party A”) and Counterparty (“Party B”) as to the terms of the Transaction to which this Confirmation relates. In addition, Party A and Party B agree to use all reasonable efforts to negotiate, execute and deliver an agreement in the form of the ISDA 2002 Master Agreement with such modifications as Party A and Party B will in good faith agree (the “ISDA Form” or the “Agreement”). Upon execution by the parties of such Agreement, this Confirmation will supplement, form a part of and be subject to the Agreement. All provisions contained or incorporated by reference in such Agreement upon its execution shall govern this Confirmation except as expressly modified below. Until Party A and Party B execute and deliver the Agreement, this Confirmation, together with all other documents referring to the ISDA Form (each a “Confirmation”) confirming Transactions (each a “Transaction”) entered into between us (notwithstanding anything to the contrary in a Confirmation) shall supplement, form a part of, and be subject to an agreement in the form of the ISDA Form as if Party A and Party B had executed an agreement on the Trade Date of the first such Transaction between us in such form, with the Schedule thereto (i) specifying only that (a) the governing law is English law, provided, that such choice of law shall be superseded by any choice of law provision specified in the Agreement upon its execution, and (b) the Termination Currency is U.S. Dollars and (ii) incorporating the addition to the definition of “Indemnifiable Tax” contained in (page 49 of) the ISDA “User’s Guide to the 2002 ISDA Master Agreements”.
2. The terms of the particular Transaction to which this Confirmation relates are as follows:

Notional Amount: 300000000.00 USD
Trade Date: 06/23/2005
Effective Date: 06/27/2005
Termination Date: 06/18/2008

Fixed Amounts:
Fixed Rate Payer: "Counterparty"
Fixed Rate Payer Period End Dates: "The 15th day of March, June, September and December of each year, commencing September 15, 2005, through and including the Termination Date with No Adjustment"
Fixed Rate Payer Payment Dates: "The 15th day of March, June, September and December of each year, commencing September 15, 2005, through and including the Termination Date"
Fixed Rate: -4.09%
Fixed Rate Day Count Fraction: "30" "360"
Fixed Rate Payer Business Days:"New York"
Fixed Rate Payer Business Day Convention: "Modified Following"

```
Floating Amounts:
Floating Rate Payer: "DBAG"
Floating Rate Payer Period End Dates: "The 15th day of March, June, September and
December of each year, commencing September 15, 2005, through and including the
Termination Date with No Adjustment"
Floating Rate Payer Payment Dates: "The 15th day of March, June, September and
December of each year, commencing September 15, 2005, through and including the
Termination Date"
Floating Rate for initial Calculation Period: 3.41%
Floating Rate Option: "USD-LIBOR-BBA"
Designated Maturity: "Three months"
Spread: "None"
Floating Rate Day Count Fraction: "30" "360"
Reset Dates: "The first Floating Rate Payer Business Day of each Calculation Period
or Compounding Period, if Compounding is applicable."
Compounding: "Inapplicable"
Floating Rate Payer Business Days: "New York"
Floating Rate Payer Business Day Convention: "Modified Following"
"""

  Scenario: The fixed rate is negative
    When it receives the request
"""
{
    "$class": "org.accordproject.isda.irs.RateObservation"
}
"""
    Then it should reject the request with the error "[Ergo] Fixed rate cannot be
negative"
```

The reason for the error is that the contract has been defined with a negative interest rate (the line: `Fixed Rate: -4.09%` in the contract given as **Background** for the scenario).

### State Change

For templates which assume and can modify the contract state, the scenario should also include pre- and post- conditions for that state. In addition, some steps are available to define scenarios that specify the expected initial step for the contract.

For instance, the following scenario for the [Installment Sale](https://templates.accordproject.org/installment-sale@0.12.1.html) template describes the expected initial state and execution of one installment:
```gherkin
Feature: Installment Sale
  This describe the expected behavior for the Accord Project's installment sale
contract

  Background:
    Given that the contract says
"""
"Dan" agrees to pay to "Ned" the total sum e10000, in the manner following:

E500 is to be paid at closing, and the remaining balance of E9500 shall be paid as
follows:
```

```
E500 or more per month on the first day of each and every month, and continuing
until the entire balance, including both principal and interest, shall be paid in
full -- provided, however, that the entire balance due plus accrued interest and
any other amounts due here-under shall be paid in full on or before 24 months.

Monthly payments, which shall start on month 3, include both principal and interest
with interest at the rate of 1.5%, computed monthly on the remaining balance from
time to time unpaid.

"""

  Scenario: The contract should be in the correct initial state
    Then the initial state of the contract should be
"""
{
  "$class": "org.accordproject.installmentsale.InstallmentSaleState",
  "status" : "WaitingForFirstDayOfNextMonth",
  "balance_remaining" : 10000.00,
  "total_paid" : 0.00,
  "next_payment_month" : 3,
  "stateId": "#1"
}
"""

  Scenario: The contract accepts a first payment, and maintain the remaining
balance
    Given the state
"""
{
  "$class": "org.accordproject.installmentsale.InstallmentSaleState",
  "status" : "WaitingForFirstDayOfNextMonth",
  "balance_remaining" : 10000.00,
  "total_paid" : 0.00,
  "next_payment_month" : 3,
  "stateId": "#1"
}
"""
    When it receives the request
"""
{
    "$class": "org.accordproject.installmentsale.Installment",
    "amount": 2500.00
}
"""
    Then it should respond with
"""
{
  "total_paid": 2500,
  "balance": 7612.499999999999,
  "$class": "org.accordproject.installmentsale.Balance"
}
"""
    And the new state of the contract should be
"""
{
  "$class": "org.accordproject.installmentsale.InstallmentSaleState",
  "status" : "WaitingForFirstDayOfNextMonth",
  "balance_remaining" : 7612.499999999999,
  "total_paid" : 2500,
```

~~~~
  "next_payment_month" : 4,
  "stateId": "#1"
}
"""

```

### Current Time

The logic for some clause or contract templates is time-dependent. It can be useful
to specify multiple scenarios for the behavior under different date and time
assumptions. This can be described with an additional **When** step to set the
current time to a specific value.

For instance, the following shows two scenarios for the [IP
Payment](https://templates.accordproject.org/ip-payment@0.10.1.html) template,
which describe its expected behavior for two distinct current times:

```gherkin
Feature: IP Payment Contract
  This describes the expected behavior for the Accord Project’s IP Payment Contract
contract

  Background:
    Given the default contract

  Scenario: Payment of a specified amount should be made
    When the current time is "2019-03-04T16:34:00-05:00"
    And it receives the request
"""
{
    "$class": "org.accordproject.ippayment.PaymentRequest",
    "netSaleRevenue": 1200,
    "sublicensingRevenue": 450,
    "permissionGrantedBy": "2018-04-05T00:00:00-05:00"
}
"""
    Then it should respond with
"""
{
    "$class": "org.accordproject.ippayment.PayOut",
    "totalAmount": 77.4,
    "dueBy": "2018-04-12T00:00:00.000-05:00"
}
"""

  Scenario: Payment of a specified amount should be made
    When the current time is "2019-03-01T16:34:00-02:00"
    And it receives the request
"""
{
    "$class": "org.accordproject.ippayment.PaymentRequest",
    "netSaleRevenue": 1550,
    "sublicensingRevenue": 225,
    "permissionGrantedBy": "2018-04-05T00:00:00-05:00"
}
"""
    Then it should respond with
"""
~~~~

~~~~
{
    "$class": "org.accordproject.ippayment.PayOut",
    "totalAmount": 81.45,
    "dueBy": "2018-04-12T03:00:00.000-02:00"
}
"""
```

### Emitting Obligations

If the template execution emits obligations, those can also be specified in the
scenario as one of the **Then** steps.

For instance, the following shows a scenario for the [Rental
Deposit](https://templates.accordproject.org/ip-payment@0.10.1.html) template,
which describes the expected list of obligations that should be emitted for a given
request:
```gherkin
Feature: Rental Deposit
  This describe the expected behavior for the Accord Project's rental deposit
contract

  Background:
    Given the default contract

  Scenario: The property was inspected and there was damage
    When the current time is "2018-01-02T16:34:00Z"
    And it receives the default request
    Then it should respond with
"""
{
   "$class": "org.accordproject.rentaldeposit.PropertyInspectionResponse",
    "balance": {
      "$class": "org.accordproject.money.MonetaryAmount",
      "currencyCode" : "USD",
      "doubleValue" : 1550
    }
}
"""
    And the following obligations should have been emitted
"""
[
    {
        "$class": "org.accordproject.cicero.runtime.PaymentObligation",
        "amount": {
            "$class": "org.accordproject.money.MonetaryAmount",
            "doubleValue": 1550,
            "currencyCode": "USD"
        }
    }
]
"""
```
--------------------------------------------------------------------------------
---
id: ref-concerto-api
title: Concerto API
---
~~~~

---
id: ref-concerto-cli
title: Command Line

Install the `@accordproject/concerto-cli` npm package to access the Concerto
command line interface (CLI). After installation you can use the `concerto` command
and its sub-commands as described below.

To install the Concerto CLI:
```
npm install -g @accordproject/concerto-cli
```

## Usage

```md
concerto <cmd> [args]

Commands:
  concerto validate            validate JSON against model files
  concerto compile             generate code for a target platform
  concerto get                 save local copies of external model dependencies
  concerto parse               parse a cto string to a JSON syntax tree
  concerto print               print a JSON syntax tree to a cto string
  concerto version <release>  modify the version of one or more model files
  concerto compare             compare two Concerto model files
  concerto infer               generate a concerto model from a source schema
  concerto generate <mode>    generate a sample JSON object for a concept

Options:
      --version  Show version number                                   [boolean]
  -v, --verbose                                                 [default: false]
      --help     Show help                                             [boolean]
```

## concerto validate
`concerto validate` lets you check whether a JSON sample is a valid instance of the
given model.

```md
concerto validate

validate JSON against model files

Options:
      --version     Show version number                                [boolean]
  -v, --verbose                                                 [default: false]
      --help        Show help                                          [boolean]
      --input       JSON to validate                                    [string]
      --model       array of concerto model files                        [array]
      --utcOffset   set UTC offset                                      [number]
      --offline     do not resolve external models    [boolean] [default: false]
      --functional  new validation API                [boolean] [default: false]
      --ergo        validation and emit for Ergo      [boolean] [default: false]
```

### Example
For example, using the `validate` command to check the sample `request.json` file from a [Late Delivery and Penalty](https://github.com/accordproject/cicero-template-library/tree/master/src/latedeliveryandpenalty) clause:

```
concerto validate --input request.json --model model/clause.cto
```

returns:

```json
{
  "$class": 
"org.accordproject.latedeliveryandpenalty.LateDeliveryAndPenaltyRequest",
  "forceMajeure": false,
  "agreedDelivery": "2017-12-17T04:24:00.000-04:00",
  "goodsValue": 200,
  "$timestamp": "2021-06-17T09:41:54.207-04:00"
}
```

## concerto compile
`concerto compile` takes an array of local CTO files, downloads any external dependencies (imports) and then converts all the model to the target format.

```md
concerto compile

generate code for a target platform

Options:
      --version            Show version number                          [boolean]
  -v, --verbose                                                  [default: false]
      --help               Show help                                    [boolean]
      --model              array of concerto model files    [array] [default: []]
      --offline            do not resolve external models
                                                       [boolean] [default: false]
      --target             target of the code generation
                                                 [string] [default: "JSONSchema"]
      --output             output directory path [string] [default: "./output/"]
      --metamodel          Include the Concerto Metamodel in the output
                                                       [boolean] [default: false]
      --strict             Require versioned namespaces and imports
                                                       [boolean] [default: false]
      --useSystemTextJson  Compile for System.Text.Json library (`csharp` target
                           only)                       [boolean] [default: false]
      --useNewtonsoftJson  Compile for Newtonsoft.Json library (`csharp` target
                           only)                       [boolean] [default: false]
      --namespacePrefix    A prefix to add to all namespaces (`csharp` target
                           only)                                         [string]
      --pascalCase         Use PascalCase for generated identifier names
                                                        [boolean] [default: true]
```

At the moment, the available target formats are as follows:
- Go Lang: `concerto compile --model modelfile.cto --target Golang`
- JSONSchema: `concerto compile --model modelfile.cto --target JSONSchema`
- XMLSchema: `concerto compile --model modelfile.cto --target XMLSchema`

- Plant UML: `concerto compile --model modelfile.cto --target PlantUML`
- Typescript: `concerto compile --model modelfile.cto --target Typescript`
- Java: `concerto compile --model modelfile.cto --target Java`
- GraphQL: `concerto compile --model modelfile.cto --target GraphQL`
- CSharp: `concerto compile --model modelfile.cto --target CSharp`
- OData: `concerto compile --model modelfile.cto --target OData`
- Mermaid: `concerto compile --model modelfile.cto --target Mermaid`
- Markdown: `concerto compile --model modelfile.cto --target Markdown`

### Example
For example, using the `compile` command to export the `clause.cto` file from a [Late Delivery and Penalty](https://github.com/accordproject/cicero-template-library/tree/master/src/latedeliveryandpenalty) clause into `Go Lang` format:

```md
cd ./model
concerto compile --model clause.cto --target Golang
```

returns:
```md
info: Compiled to Go in './output/'.
```

## concerto get
`concerto get` allows you to resolve and download external models from a set of local CTO files.

```md
concerto get

save local copies of external model dependencies

Options:
      --version  Show version number                                 [boolean]
  -v, --verbose                                               [default: false]
      --help     Show help                                           [boolean]
      --model    array of concerto (cto) model files        [array] [required]
      --output   output directory path                 [string] [default: "./"]
```

### Example
For example, using the `get` command to get the external models in the `clause.cto` file from a [Late Delivery and Penalty](https://github.com/accordproject/cicero-template-library/tree/master/src/latedeliveryandpenalty) clause:

```md
concerto get --model clause.cto
```

returns:
```md
info: Loaded external models in './'.
```

## concerto parse
`concerto parse` allows you to parse a set of CTO models to their JSON representation (metamodel).

```md
parse a cto string to a JSON syntax tree

Options:
      --version               Show version number                      [boolean]
  -v, --verbose                                                 [default: false]
      --help                  Show help                                [boolean]
      --model                 array of concerto model files   [array] [required]
      --resolve               resolve names to fully qualified names
                                              [boolean] [default: false]
      --all                   import all models       [boolean] [default: false]
      --output                path to the output file                   [string]
      --excludeLineLocations  Exclude file line location metadata from metamodel
                              instance                [boolean] [default: false]
```

## concerto print
`concerto print` allows you to convert a model in JSON metamodel format to a CTO
string.

```md
concerto print

print a JSON syntax tree to a cto string

Options:
      --version  Show version number                                   [boolean]
  -v, --verbose                                                 [default: false]
      --help     Show help                                             [boolean]
      --input    the metamodel to export                   [string] [required]
      --output   path to the output file                                [string]
```

## concerto version
`concerto version` allows you to modify the version of one or more model files

```md
concerto version <release>

modify the version of one or more model files

Positionals:
  release  the new version, or a release to use when incrementing the existing
           version
    [string] [required] [choices: "keep", "major", "minor", "patch", "premajor",
                                           "preminor", "prepatch", "prerelease"]

Options:
      --version          Show version number                           [boolean]
  -v, --verbose                                                 [default: false]
      --help             Show help                                     [boolean]
      --model, --models  array of concerto model files        [array] [required]
      --prerelease       set the specified pre-release version          [string]
```

## concerto compare
`concerto compare` allows you to compare two model files

```md
```

~~~text
concerto compare

compare two Concerto model files

Options:
      --version  Show version number                                   [boolean]
  -v, --verbose                                                 [default: false]
      --help     Show help                                             [boolean]
      --old      the old Concerto model file                [string] [required]
      --new      the new Concerto model file                [string] [required]
```

## concerto infer
`concerto infer` allows you to generate a Concerto model from a source schema such
as JSON Schema or an OpenAPI definition.

```md
concerto infer

generate a concerto model from a source schema

Options:
      --version          Show version number                       [boolean]
  -v, --verbose                                             [default: false]
      --help             Show help                                 [boolean]
      --input            path to the input file          [string] [required]
      --output           path to the output file                    [string]
      --format           either `openapi` or `jsonSchema`
                                            [string] [default: "jsonSchema"]
      --namespace        The namespace for the output model
                                                         [string] [required]
      --typeName         The name of the root type[string] [default: "Root"]
      --capitalizeFirst  Capitalize the first character of type names
                                                  [boolean] [default: false]
```

### Example
```console
concerto infer --namespace com.example.restapi --format openapi --input
example.swagger.json --output example.cto
```

## concerto generate
`concerto generate` allows you to generate a sample instance for a type in a model

```md
concerto generate <mode>

generate a sample JSON object for a concept

Positionals:
  mode  Generation mode. `empty` will generate a minimal example, `sample` will
        generate random values  [string] [required] [choices: "sample", "empty"]

Options:
      --version                Show version number                     [boolean]
  -v, --verbose                                                 [default: false]
      --help                   Show help                               [boolean]
      --model                  The file location of the source models
~~~

```
                                                         [array] [required]
      --concept               The fully qualified name of the Concept type to
                              generate                     [string] [required]
      --includeOptionalFields  Include optional fields will be included in the
                              output                 [boolean] [default: false]
      --metamodel             Include the Concerto Metamodel in the output
                                                     [boolean] [default: false]
      --strict                Require versioned namespaces and imports
                                                     [boolean] [default: false]
```

---
id: ref-concerto-decorators
title: Decorators

Decorators are used to add metadata to Concerto model elements, typically to control how variables are edited, printed or transformed.

## Pdf

The `@Pdf` decorator is used to control how a variable is rendered by the `markdown-pdf` transformation, which is used to convert CiceroMark rich text to PDF.

### Attributes

style : specifies the style name used to render the variable. Default styles are [defined in the code](https://github.com/accordproject/markdown-transform/blob/master/packages/markdown-pdf/src/PdfTransformer.js#L278) and may be overridden or supplemented via the `options.styles` parameter.

### Example

The example below renders the `title` variable using the PDF background style, which is defined to have the color `white`.

```
asset ExampleClause extends AccordClause {
   @Pdf("style", "background")
   o String title
}
```

## ContractEditor

The `@ContractEditor` decorator is used to control how a variable is edited using the `ContractEditor` React [web-components](https://github.com/accordproject/web-components).

### Attributes

readOnly : when set to true the variable value cannot be edited

fontFamily : the name of the HTML font-family to use when rendering the variable

backgroundColor : the HTML background color to use when rendering the variable

border : the HTML border color to use when rendering the variable

### Example

The example below renders the `title` variable using custom font, background color and border color. The variable is read-only and cannot be edited.

```
asset ExampleClause extends AccordClause {
   @ContractEditor("readOnly", true,
    "fontFamily", "Lucida Console, Courier, monospace",
    "backgroundColor", "#FAE094", "border", '#CCA855' )
   o String title
}
```

## FormEditor

The `@FormEditor` decorator is used to control whether the `ConcertoForm` React [web-components](https://github.com/accordproject/web-components) creates an input field for the variable.

### Attributes

hide : when set to true an input field for the variable is not created

### Example

The example specifies that an input field for the `title` variable should not be created by the Concerto Form component.

```
asset ExampleClause extends AccordClause {
   @FormEditor("hide", true)
   o String title
}
```

## DocuSignTab

The `@DocuSignTab` decorator is used to specify how a variable is mapped to a DocuSign tab. This decorator is not currently supported by existing Accord Project transformations but is reserved for future use, and may be used by upstream consumers.

### Attributes

type : the type of the DocuSign tab. See the documentation for DocuSign [EnvelopeRecipientTabs](https://developers.docusign.com/docs/esign-rest-api/reference/Envelopes/EnvelopeRecipientTabs/#tab-types) for the list of supported tab types.

optional : whether the tab is optional or required

### Example

The example below maps the `title` variable to the DocuSign tab type `Title` and marks it as optional.

```
asset ExampleClause extends AccordClause {
  @DocuSignTab("type", "Title", "optional", true)
   o String title
}
```

---
id: ref-ergo-api
title: Ergo API

## Classes

<dl>
<dt><a href="#Commands">Commands</a></dt>
<dd><p>Utility class that implements the commands exposed by the Ergo CLI.</p>
</dd>
</dl>

## Functions

<dl>
<dt><a href="#getJson">getJson(input)</a> ⇒ <code>object</code></dt>
<dd><p>Load a file or JSON string</p>
</dd>
<dt><a href="#loadTemplate">loadTemplate(template, files)</a> ⇒
<code>Promise.&lt;LogicManager&gt;</code></dt>
<dd><p>Load a template from directory or files</p>
</dd>
<dt><a href="#fromDirectory">fromDirectory(path, [options])</a> ⇒
<code>Promise.&lt;LogicManager&gt;</code></dt>
<dd><p>Builds a LogicManager from a directory.</p>
</dd>
<dt><a href="#fromZip">fromZip(buffer, [options])</a> ⇒
<code>Promise.&lt;LogicManager&gt;</code></dt>
<dd><p>Builds a LogicManager from a Zip.</p>
</dd>
<dt><a href="#fromFiles">fromFiles(files, [options])</a> ⇒
<code>Promise.&lt;LogicManager&gt;</code></dt>
<dd><p>Builds a LogicManager from files.</p>
</dd>
<dt><a href="#validateContract">validateContract(modelManager, contract, utcOffset, options)</a> ⇒ <code>object</code></dt>
<dd><p>Validate contract JSON</p>
</dd>
<dt><a href="#validateInput">validateInput(modelManager, input, utcOffset)</a> ⇒
<code>object</code></dt>
<dd><p>Validate input JSON</p>
</dd>
<dt><a href="#validateStandard">validateStandard(modelManager, input, utcOffset)</a> ⇒ <code>object</code></dt>
<dd><p>Validate standard</p>
</dd>
<dt><a href="#validateInputRecord">validateInputRecord(modelManager, input,

utcOffset)</a> ⇒ <code>object</code></dt>
<dd><p>Validate input JSON record</p>
</dd>
<dt><a href="#validateOutput">validateOutput(modelManager, output, utcOffset)</a> ⇒
<code>object</code></dt>
<dd><p>Validate output JSON</p>
</dd>
<dt><a href="#validateOutputArray">validateOutputArray(modelManager, output,
utcOffset)</a> ⇒ <code>Array.&lt;object&gt;</code></dt>
<dd><p>Validate output JSON array</p>
</dd>
<dt><a href="#init">init(engine, logicManager, contractJson, currentTime,
utcOffset)</a> ⇒ <code>object</code></dt>
<dd><p>Invoke Ergo contract initialization</p>
</dd>
<dt><a href="#trigger">trigger(engine, logicManager, contractJson, stateJson,
currentTime, utcOffset, requestJson)</a> ⇒ <code>object</code></dt>
<dd><p>Trigger the Ergo contract with a request</p>
</dd>
<dt><a href="#resolveRootDir">resolveRootDir(parameters)</a> ⇒
<code>string</code></dt>
<dd><p>Resolve the root directory</p>
</dd>
<dt><a href="#compareComponent">compareComponent(expected, actual)</a></dt>
<dd><p>Compare actual and expected result components</p>
</dd>
<dt><a href="#compareSuccess">compareSuccess(expected, actual)</a></dt>
<dd><p>Compare actual result and expected result</p>
</dd>
</dl>

<a name="Commands"></a>

## Commands
Utility class that implements the commands exposed by the Ergo CLI.

Kind: global class

* [Commands](#Commands)
    * [.trigger(template, files, contractInput, stateInput, [currentTime],
[utcOffset], requestsInput, warnings)](#Commands.trigger) ⇒ <code>object</code>
    * [.invoke(template, files, clauseName, contractInput, stateInput,
[currentTime], [utcOffset], paramsInput, warnings)](#Commands.invoke) ⇒
<code>object</code>
    * [.initialize(template, files, contractInput, [currentTime], [utcOffset],
paramsInput, warnings)](#Commands.initialize) ⇒ <code>object</code>
    * [.parseCTOtoFileSync(ctoPath)](#Commands.parseCTOtoFileSync) ⇒
<code>string</code>
    * [.parseCTOtoFile(ctoPath)](#Commands.parseCTOtoFile) ⇒ <code>string</code>

<a name="Commands.trigger"></a>

### Commands.trigger(template, files, contractInput, stateInput, [currentTime],
[utcOffset], requestsInput, warnings) ⇒ <code>object</code>
Send a request an Ergo contract

Kind: static method of [<code>Commands</code>](#Commands)
Returns: <code>object</code> - Promise to the result of execution

Param	Type	Description
template	<code>string</code>	template directory
files	<code>Array.&lt;string&gt;</code>	input files
contractInput	<code>string</code>	the contract data
stateInput	<code>string</code>	the contract state
[currentTime]	<code>string</code>	the definition of 'now', defaults to current time
[utcOffset]	<code>number</code>	UTC Offset for this execution, defaults to local offset
requestsInput	<code>Array.&lt;string&gt;</code>	the requests
warnings	<code>boolean</code>	whether to print warnings

<a name="Commands.invoke"></a>

### Commands.invoke(template, files, clauseName, contractInput, stateInput, [currentTime], [utcOffset], paramsInput, warnings) ⇒ <code>object</code>
Invoke an Ergo contract's clause

Kind: static method of [<code>Commands</code>](#Commands)
Returns: <code>object</code> - Promise to the result of invocation

Param	Type	Description
template	<code>string</code>	template directory
files	<code>Array.&lt;string&gt;</code>	input files
clauseName	<code>string</code>	the name of the clause to invoke
contractInput	<code>string</code>	the contract data
stateInput	<code>string</code>	the contract state
[currentTime]	<code>string</code>	the definition of 'now', defaults to current time
[utcOffset]	<code>number</code>	UTC Offset for this execution, defaults to local offset
paramsInput	<code>object</code>	the parameters for the clause
warnings	<code>boolean</code>	whether to print warnings

<a name="Commands.initialize"></a>

### Commands.initialize(template, files, contractInput, [currentTime], [utcOffset], paramsInput, warnings) ⇒ <code>object</code>
Invoke init for an Ergo contract

Kind: static method of [<code>Commands</code>](#Commands)
Returns: <code>object</code> - Promise to the result of execution

Param	Type	Description
template	<code>string</code>	template directory
files	<code>Array.&lt;string&gt;</code>	input files
contractInput	<code>string</code>	the contract data
[currentTime]	<code>string</code>	the definition of 'now', defaults to current time
[utcOffset]	<code>number</code>	UTC Offset for this execution, defaults to local offset
paramsInput	<code>object</code>	the parameters for the clause
warnings	<code>boolean</code>	whether to print warnings

<a name="Commands.parseCTOtoFileSync"></a>

```markdown
### Commands.parseCTOtoFileSync(ctoPath) ⇒ <code>string</code>
Parse CTO to JSON File

**Kind**: static method of [<code>Commands</code>](#Commands)
**Returns**: <code>string</code> - The name of the generated CTOJ model file

| Param | Type | Description |
| --- | --- | --- |
| ctoPath | <code>string</code> | path to CTO model file |

<a name="Commands.parseCTOtoFile"></a>

### Commands.parseCTOtoFile(ctoPath) ⇒ <code>string</code>
Parse CTO to JSON File

**Kind**: static method of [<code>Commands</code>](#Commands)
**Returns**: <code>string</code> - The name of the generated CTOJ model file

| Param | Type | Description |
| --- | --- | --- |
| ctoPath | <code>string</code> | path to CTO model file |

<a name="getJson"></a>

## getJson(input) ⇒ <code>object</code>
Load a file or JSON string

**Kind**: global function
**Returns**: <code>object</code> - JSON object

| Param | Type | Description |
| --- | --- | --- |
| input | <code>object</code> | either a file name or a json string |

<a name="loadTemplate"></a>

## loadTemplate(template, files) ⇒ <code>Promise.&lt;LogicManager&gt;</code>
Load a template from directory or files

**Kind**: global function
**Returns**: <code>Promise.&lt;LogicManager&gt;</code> - a Promise to the
instantiated logicmanager

| Param | Type | Description |
| --- | --- | --- |
| template | <code>string</code> | template directory |
| files | <code>Array.&lt;string&gt;</code> | input files |

<a name="fromDirectory"></a>

## fromDirectory(path, [options]) ⇒ <code>Promise.&lt;LogicManager&gt;</code>
Builds a LogicManager from a directory.

**Kind**: global function
**Returns**: <code>Promise.&lt;LogicManager&gt;</code> - a Promise to the
instantiated logicmanager

| Param | Type | Description |
| --- | --- | --- |
| path | <code>String</code> | to a local directory |
| [options] | <code>Object</code> | an optional set of options to configure the
instance. |

<a name="fromZip"></a>

## fromZip(buffer, [options]) ⇒ <code>Promise.&lt;LogicManager&gt;</code>
Builds a LogicManager from a Zip.

**Kind**: global function
**Returns**: <code>Promise.&lt;LogicManager&gt;</code> - a Promise to the
instantiated logicmanager

| Param | Type | Description |
| --- | --- | --- |
| buffer | <code>Buffer</code> | the buffer to a Zip (zip) file |
| [options] | <code>Object</code> | an optional set of options to configure the
instance. |

<a name="fromFiles"></a>

## fromFiles(files, [options]) ⇒ <code>Promise.&lt;LogicManager&gt;</code>
Builds a LogicManager from files.

**Kind**: global function
**Returns**: <code>Promise.&lt;LogicManager&gt;</code> - a Promise to the
instantiated logicmanager

| Param | Type | Description |
| --- | --- | --- |
| files | <code>Array.&lt;String&gt;</code> | file names |
| [options] | <code>Object</code> | an optional set of options to configure the
instance. |

<a name="validateContract"></a>

## validateContract(modelManager, contract, utcOffset, options) ⇒
<code>object</code>
Validate contract JSON

**Kind**: global function
**Returns**: <code>object</code> - the validated contract

| Param | Type | Description |
| --- | --- | --- |
| modelManager | <code>object</code> | the Concerto model manager |
| contract | <code>object</code> | the contract JSON |
| utcOffset | <code>number</code> | UTC Offset for DateTime values |
| options | <code>object</code> | parameters for contract variables validation |

<a name="validateInput"></a>

## validateInput(modelManager, input, utcOffset) ⇒ <code>object</code>
Validate input JSON

**Kind**: global function
**Returns**: <code>object</code> - the validated input

| Param | Type | Description |
```

modelManager	<code>object</code>	the Concerto model manager
input	<code>object</code>	the input JSON
utcOffset	<code>number</code>	UTC Offset for DateTime values

<a name="validateStandard"></a>

## validateStandard(modelManager, input, utcOffset) ⇒ <code>object</code>
Validate standard

Kind: global function
Returns: <code>object</code> - the validated input

Param	Type	Description
modelManager	<code>object</code>	the Concerto model manager
input	<code>object</code>	the input JSON
utcOffset	<code>number</code>	UTC Offset for DateTime values

<a name="validateInputRecord"></a>

## validateInputRecord(modelManager, input, utcOffset) ⇒ <code>object</code>
Validate input JSON record

Kind: global function
Returns: <code>object</code> - the validated input

Param	Type	Description
modelManager	<code>object</code>	the Concerto model manager
input	<code>object</code>	the input JSON record
utcOffset	<code>number</code>	UTC Offset for DateTime values

<a name="validateOutput"></a>

## validateOutput(modelManager, output, utcOffset) ⇒ <code>object</code>
Validate output JSON

Kind: global function
Returns: <code>object</code> - the validated output

Param	Type	Description
modelManager	<code>object</code>	the Concerto model manager
output	<code>object</code>	the output JSON
utcOffset	<code>number</code>	UTC Offset for DateTime values

<a name="validateOutputArray"></a>

## validateOutputArray(modelManager, output, utcOffset) ⇒ <code>Array.&lt;object&gt;</code>
Validate output JSON array

Kind: global function
Returns: <code>Array.&lt;object&gt;</code> - the validated output array

Param	Type	Description
modelManager	<code>object</code>	the Concerto model manager

| output | <code>*</code> | the output JSON array |
| utcOffset | <code>number</code> | UTC Offset for DateTime values |

<a name="init"></a>

## init(engine, logicManager, contractJson, currentTime, utcOffset) ⇒ <code>object</code>
Invoke Ergo contract initialization

Kind: global function
Returns: <code>object</code> - Promise to the initial state of the contract

Param	Type	Description
engine	<code>object</code>	the execution engine
logicManager	<code>object</code>	the Template Logic
contractJson	<code>object</code>	contract data in JSON
currentTime	<code>string</code>	the definition of 'now'
utcOffset	<code>utcOffset</code>	UTC Offset for this execution

<a name="trigger"></a>

## trigger(engine, logicManager, contractJson, stateJson, currentTime, utcOffset, requestJson) ⇒ <code>object</code>
Trigger the Ergo contract with a request

Kind: global function
Returns: <code>object</code> - Promise to the response

Param	Type	Description
engine	<code>object</code>	the execution engine
logicManager	<code>object</code>	the Template Logic
contractJson	<code>object</code>	contract data in JSON
stateJson	<code>object</code>	state data in JSON
currentTime	<code>string</code>	the definition of 'now'
utcOffset	<code>utcOffset</code>	UTC Offset for this execution
requestJson	<code>object</code>	state data in JSON

<a name="resolveRootDir"></a>

## resolveRootDir(parameters) ⇒ <code>string</code>
Resolve the root directory

Kind: global function
Returns: <code>string</code> - root directory used to resolve file names

Param	Type	Description
parameters	<code>string</code>	Cucumber's World parameters

<a name="compareComponent"></a>

## compareComponent(expected, actual)
Compare actual and expected result components

Kind: global function

| Param | Type | Description |

| --- | --- | --- |
| expected | <code>string</code> | the expected component as specified in the test workload |
| actual | <code>string</code> | the actual component as returned by the engine |

<a name="compareSuccess"></a>

## compareSuccess(expected, actual)
Compare actual result and expected result

Kind: global function

Param	Type	Description
expected	<code>string</code>	the expected successful result as specified in the test workload
actual	<code>string</code>	the successful result as returned by the engine

----
---
id: ref-ergo-cli
title: Command Line

Install the `@accordproject/ergo-cli` npm package to access the Ergo command line interface (CLI). After installation you can use the ergo command and its sub-commands as described below.

To install the Ergo CLI:
```
npm install -g @accordproject/ergo-cli
```

This will install `ergo`, to compile and run contracts locally on your machine, and `ergotop`, which is a _read-eval-print-loop_ utility to write Ergo interactively.

## Ergo

### Usage

```md
ergo <command>

Commands:
  ergo trigger     send a request to the contract
  ergo invoke      invoke a clause of the contract
  ergo initialize  initialize the state for a contract
  ergo compile     compile a contract

Options:
  --help         Show help                                             [boolean]
  --version      Show version number                                   [boolean]
  --verbose, -v                                                 [default: false]
```

## ergo trigger

`ergo trigger` allows you to send a request to the contract.

```md
Usage: ergo trigger --data [file] --state [file] --request [file] [cto files] [ergo
files]

Options:
  --help         Show help                                            [boolean]
  --version      Show version number                                  [boolean]
  --verbose, -v                                                [default: false]
  --data         path to the contract data                           [required]
  --state        path to the state data                [string] [default: null]
  --currentTime  set current time                      [string] [default: null]
  --utcOffset    set UTC offset                        [number] [default: null]
  --request      path to the request data                  [array] [required]
  --template     path to the template directory        [string] [default: null]
  --warnings     print warnings                      [boolean] [default: false]
```

### Example

For example, using the `trigger` command for the [Volume Discount example]
(https://github.com/accordproject/ergo/tree/master/tests/volumediscount) in the
[Ergo Directory](https://github.com/accordproject/ergo):

```md
ergo trigger --template ./tests/volumediscount --data
./tests/volumediscount/data.json --request ./tests/volumediscount/request.json --
state ./tests/volumediscount/state.json
```

returns:

```json
{
  "clause": "orgXaccordprojectXvolumediscountXVolumeDiscount",
  "request": {
    "$class": "org.accordproject.volumediscount.VolumeDiscountRequest",
    "netAnnualChargeVolume": 10.4
  },
  "response": {
    "$class": "org.accordproject.volumediscount.VolumeDiscountResponse",
    "discountRate": 2.8,
    "$timestamp": "2021-06-17T09:36:53.847-04:00"
  },
  "state": {
    "$class": "org.accordproject.runtime.State",
    "$identifier": "7c19d1e3-1f70-4b30-8c3d-086dc45b1dd1"
  },
  "emit": []
}
```

As the `request` was sent for an annual charge volume of 10.4, which falls into the
third discount rate category (as specified in the `data.json` file), the `response`
returns with a discount rate of 2.8%.

## ergo invoke

`ergo invoke` allows you to invoke a specific clause of the contract. The main difference between `ergo invoke` and `ergo trigger` is that `ergo invoke` sends data to a specific clause, whereas `ergo trigger` lets the contract choose which clause to invoke. This is why `--clauseName` (the name of the contract you want to execute) is a required field for `ergo invoke`.

You need to pass the CTO and Ergo files (`--template`), the name of the contract that you want to execute (`--clauseName`), and JSON files for: the contract data (`--data`), the contract parameters (`--params`), the current state of the contract (`--state`), and the request.

If contract invocation is successful, `ergorun` will print out the response, the new contract state and any emitted events.

```md
Usage: ergo invoke --data [file] --state [file] --params [file] [cto files] [ergo
files]

Options:
  --help         Show help                                               [boolean]
  --version      Show version number                                     [boolean]
  --verbose, -v                                                   [default: false]
  --clauseName   the name of the clause to invoke                       [required]
  --data         path to the contract data                              [required]
  --state        path to the state data                        [string] [required]
  --currentTime  set current time                         [string] [default: null]
  --utcOffset    set UTC offset                           [number] [default: null]
  --params       path to the parameters        [string] [required] [default: {}]
  --template     path to the template directory           [string] [default: null]
  --warnings     print warnings                         [boolean] [default: false]
```

### Example

For example, using the `invoke` command for the [Volume Discount example](https://github.com/accordproject/ergo/tree/master/tests/volumediscount) in the [Ergo Directory](https://github.com/accordproject/ergo):

```md
ergo invoke --template ./tests/volumediscount --clauseName volumediscount --
data ./tests/volumediscount/data.json --params ./tests/volumediscount/params.json
--state ./tests/volumediscount/state.json
```

returns:

```json
{
  "clause": "orgXaccordprojectXvolumediscountXVolumeDiscount",
  "params": {
    "request": {
      "$class": "org.accordproject.volumediscount.VolumeDiscountRequest",
      "netAnnualChargeVolume": 10.4
    }
  },
  "response": {
    "$class": "org.accordproject.volumediscount.VolumeDiscountResponse",
    "discountRate": 2.8,
    "$timestamp": "2021-06-17T09:38:03.189-04:00"
  },
  "state": {
    "$class": "org.accordproject.runtime.State",
    "$identifier": "b757ad1f-e011-4fda-9b37-e7157512300f"
  },
  "emit": []
}
```

Although this looks very similar to what `ergo trigger` returns, it is important to note that `--clauseName volumediscount` was specifically invoked.

## ergo initialize
`ergo initialize` allows you to obtain the initial state of the contract. This is the state of the contract without requests or responses.

```md
Usage: ergo intialize --data [file] --params [file] [cto files] [ergo files]

Options:
  --help          Show help                                          [boolean]
  --version       Show version number                                [boolean]
  --verbose, -v                                               [default: false]
  --data          path to the contract data                         [required]
  --currentTime   set current time                     [string] [default: null]
  --utcOffset     set UTC offset                       [number] [default: null]
  --params        path to the parameters               [string] [default: null]
  --template      path to the template directory       [string] [default: null]
  --warnings      print warnings                     [boolean] [default: false]
```

### Example

For example, using the `initialize` command for the [Volume Discount example](https://github.com/accordproject/ergo/tree/master/tests/volumediscount) in the [Ergo Directory](https://github.com/accordproject/ergo):

```md
ergo initialize --template ./tests/volumediscount --data
./tests/volumediscount/data.json
```

returns:

```json
{
  "clause": "orgXaccordprojectXvolumediscountXVolumeDiscount",
  "params": {
  },
  "response": null,
  "state": {
    "$class": "org.accordproject.runtime.State",
    "$identifier": "af4f0f49-2658-4465-87f4-780e7d2e38a8"
  },
  "emit": []
}
```

## ergo compile
`ergo compile` takes your input models (`.cto` files) and input contracts (`.ergo`

files) and allows you to compile a contract into a target platform. By default, Ergo compiles to JavaScript (ES6 compliant) for execution.

```md
Usage: ergo compile --target [lang] --link --monitor --warnings [cto files] [ergo
files]

Options:
  --help         Show help                                             [boolean]
  --version      Show version number                                   [boolean]
  --verbose, -v                                                 [default: false]
  --target       Target platform (available: es5,es6,cicero,java)
                                                       [string] [default: "es6"]
  --link         Link the Ergo runtime with the target code (es5,es6,cicero
                 only)                                [boolean] [default: false]
  --monitor      Produce compilation time information [boolean] [default: false]
  --warnings     print warnings                       [boolean] [default: false]
```

### Example
For example, using the `compile` command on the [Volume Discount example](https://github.com/accordproject/ergo/tree/master/tests/volumediscount) in the [Ergo Directory](https://github.com/accordproject/ergo):

```md
ergo compile ./tests/volumediscount/model/model.cto
./tests/volumediscount/logic/logic.ergo
```

returns:

```md
Compiling Ergo './tests/volumediscount/logic/logic.ergo' --
'./tests/volumediscount/logic/logic.js'
```

Which means a new `logic.js` file is located in the `./tests/volumediscount/logic` directory.

To compile the contract to Javascript and **link the Ergo runtime for execution**:
```md
ergo compile ./tests/volumediscount/model/model.cto
./tests/volumediscount/logic/logic.ergo --link
```

returns:

```md
Compiling Ergo './tests/volumediscount/logic/logic.ergo' --
'./tests/volumediscount/logic/logic.js'
```

---
id: ref-ergo-repl
title: Read-Eval-Print Loop

`ergotop` is a convenient tool to try-out Ergo contracts in an interactive way. You can write commands, or expressions and see the result. It is often called the Ergo REPL, for _read-eval-print-loop_, since it literally: reads your input Ergo from the command-line, evaluates it, prints the result and loops back to read your next input.

## Starting the REPL

To start the REPL:

```
$ ergotop
Welcome to ERGOTOP version 0.20.0
ergo$
```

It should print the prompt `ergo$` which indicates it is ready to read your command. For instance:

```ergo
    ergo$ return 42
    Response. 42 : Integer
```

`ergotop` prints back both the resulting value and its type. You can then keep typing commands:

```ergo
    ergo$ return "hello " ++ "world!"
    Response. "hello world!" : String
    ergo$ define constant pi = 3.14
    ergo$ return pi ^ 2.0
    Response. 9.8596 : Double
```

If your expression is not valid, or not well-typed, it will return an error:

```ergo
    ergo$ return if true else "hello"
    Parse error (at line 1 col 15).
    return if true else "hello"
                   ^^^^
    ergo$ return if "hello" then 1 else 2
    Type error (at line 1 col 10). 'if' condition not boolean.
    return if "hello" then 1 else 2
              ^^^^^^^
```

If what you are trying to write is too long to fit on one line, you can use `\` to go to a new line:

```ergo
    ergo$ define function squares(l:Double[]) : Double[] { \
      ...   return \
      ...     foreach x in l return x * x \
      ... }
    ergo$ return squares([2.4,4.5,6.7])
    Response. [5.76, 20.25, 44.89] : Double[]
```

## Loading files

You can load CTO and Ergo files to use in your REPL session. Once the REPL is launched you will have to import the corresponding namespace. For instance, if you want to use the `compoundInterestMultiple` function defined in the `./examples/promissory-note/money.ergo` file, you can do it as follows:

```ergo
$ ergotop ./examples/promissory-note/logic/money.ergo
Welcome to ERGOTOP version 0.20.0

ergo$ import org.accordproject.ergo.money.*
ergo$ return compoundInterestMultiple(0.035, 100)
Response. 1.00946960405 : Double
```

## Calling contracts

To call a contract, you first needs to _instantiate_ it, which means setting its parameters and initializing its state. You can do this by using the `set contract` and `call init` commands respectively. Here is an example using the `volumediscount` template:

```ergo
$ ergotop ./examples/volumediscount/model/model.cto
./examples/volumediscount/logic/logic.ergo
ergo$ import org.accordproject.cicero.contract.*
ergo$ import org.accordproject.volumediscount.*
ergo$ set contract VolumeDiscount over VolumeDiscountContract {firstVolume: 1.0,
secondVolume: 10.0, firstRate: 3.0, secondRate: 2.9, thirdRate: 2.8, contractId:
"0", parties: none }
ergo$ call init()
Response. unit : Unit
State. AccordContractState{stateId:
"org.accordproject.cicero.contract.AccordContractState#1"} : AccordContractState
```

You can then invoke clauses of the contract:

```ergo
ergo$ call volumediscount(VolumeDiscountRequest{ netAnnualChargeVolume : 0.1 })
Response. VolumeDiscountResponse{discountRate: 3.0} : VolumeDiscountResponse
ergo$ call volumediscount(VolumeDiscountRequest{ netAnnualChargeVolume : 10.5 })
Response. VolumeDiscountResponse{discountRate: 2.8} : VolumeDiscountResponse
```

You can also invoke the contract without explicitly naming the clause by sending a request. The Ergo engine dispatches that request to the first clause which can handle it:
```ergo
ergo$ send VolumeDiscountRequest{ netAnnualChargeVolume : 0.1 }
Response. VolumeDiscountResponse{discountRate: 3.0} : VolumeDiscountResponse
ergo$ send VolumeDiscountRequest{ netAnnualChargeVolume : 10.5 }
Response. VolumeDiscountResponse{discountRate: 2.8} : VolumeDiscountResponse
```

---
id: ref-ergo-spec
title: Specification

## Lexical Conventions

### File Extension

Ergo files have the ``.ergo`` extension.

### Blanks

Blank characters (such as space, tabulation, carriage return) are
ignored but they are used to separate identifiers.

### Comments

Comments come in two forms. Single line comments are introduced by the
two characters `//` and are terminated by the end of the current
line. Multi-line comments start with the two characters `/*` and are
terminated by the two characters `*/`. Multi-line comments can be
nested.

Here are examples of comments:

```ergo
    // This is a single line comment
    /* This comment spans multiple lines
       and it can also be /* nested */ */
```

### Reserved Words

The following are reserved as keywords in Ergo. They cannot be used as identifiers.

```text
namespace, import, define, function, transaction, concept, event, asset,
participant, enum, extends, contract, over, clause, throws, emits, state, call,
enforce, if, then, else, let, foreach, return, in, where, throw,
constant, match, set, emit, with, or, and, true, false, unit, none
```

## Condition Expressions

Conditional statements, conditional expressions and conditional constructs are
features of a programming language which perform different computations or actions
depending on whether a programmer-specified boolean condition evaluates to true or
false.

Conditional expressions (also known as `if` statements) allow us to conditionally
execute Ergo code depending on the value of a test condition. If the test condition
evaluates to `true` then the code on the `then` branch is evaluated. Otherwise,
when the test condition evaluates to `false` then the `else` branch is evaluated.

### Example

```ergo
if delayInDays > 15.0 then
```

~~~
  BuyerMayTerminateResponse{};
else
  BuyerMayNotTerminateResponse{}
```

### Legal Prose

For example, this corresponds to a conditional logic statement in legal
prose.

    If the delay is more than 15 days, the Buyer is entitled to terminate this
Contract.

### Syntax

```ergo
    if expression1 then     // Condition
      expression2           // Expression if condition is true
    else
      expression3           // Expression if condition is false
```

Where `expression1` is an Ergo expression that evaluates to a Boolean
value (i.e. `true` or `false`), and `expression2` and `expression3` are
Ergo expressions.

> Note that as with all Ergo expressions, new lines and indentation
> don't change the meaning of your code. However it is good practice to
> standardise the way that you using whitespace in your code to make it
> easier to read.

### Usage

If statements can be chained , i.e., `if ... then .... else if ... then ...
else ...` to build more compound conditionals.

```ergo
if request.netAnnualChargeVolume < contract.firstVolume then
  return VolumeDiscountResponse{ discountRate: contract.firstRate }
else if request.netAnnualChargeVolume < contract.secondVolume then
  return VolumeDiscountResponse{ discountRate: contract.secondRate }
else
  return VolumeDiscountResponse{ discountRate: contract.thirdRate }
```

Conditional expressions can also be used as expressions, e.g., inside a constant
declaration:

```ergo
define constant price = 42;
define constant message =  if price >i 100 then "High price" else "Low Price";
message;
```

The value of message after running this code will be `"Low Price"`.

### Related

-   [Match expression](ref-logic#match-expressions) - where many
~~~

    alternative conditions check the same variable

## Match Expressions

Match expressions allow us to check an expression against multiple
possible values or patterns. If a match is found, then Ergo will
evaluate the corresponding expression.

> Match expressions are similar to `switch` statements in other
> programming languages

### Example

```ergo
match request.status
  with "CREATED" then
    new PayOut{ amount : contract.deliveryPrice }
  with "ARRIVED" then
    new PayOut{ amount : contract.deliveryPrice - shockPenalty }
  else
    new PayOut{ amount : 0.0 }
```

### Legal Prose

> Example needed.

### Syntax

```ergo
match expression0
  with pattern1 then       // Repeat this line
    expression1            //    and this line
  else
    expression2
```

### Usage

You can use a match expression to look for patterns based on the type of
an expression.

```ergo
match response
    with let b1 : BuyerMayTerminateResponse then
        // Do something with b1
    with let b2 : BuyerMayNotTerminateResponse then
        // Do something with b2
    else
        // Do a default action
```

You can use it to match against an optional value.

```ergo
match maybe_response
    with let? b1 : BuyerMayTerminateResponse then
        // Do something when there is a response
    else
```

~~~~
        // Do something else when there is no response
```

Often a match expression is a more concise way to represent a
conditional expression with a repeating, regular condition. For example:

```ergo
    if x = 1 then
      ...
    else if x = 2 then
      ...
    else if x = 3 then
      ...
    else if x = 4 then
      ...
    else
      ...
```

This is equivalent to the match expression:

```ergo
    match x
      with 1 then
        ...
      with 2 then
        ...
      with 3 then
        ...
      with 4 then
        ...
      else
        ...
```

## Operator Precedence

Precedence determines the order of operations in expressions with operators of
different priority. In the case of the same precedence, it is based on the
associativity of operators.

### Example

`a = b * c ^ d + e` is the same as `(a = (b * (c ^ d)) + e)`

`a = b * c * d / e` is the same as `(a = (((b * c) * d) / e)`

`a.b.c.d.e ^ f` is the same as `(((((a.b).c).d).e) ^ f)`

### Table of precedence

Table of operators in Ergo with their associativity and precedence from highest to
lowest:

**Order** | **Operator(s)** | **Description** | **Associativity**
--- | --- | --- | ---
1 | . <br> ?. | field access <br> field access of optional type | left to right
2 | [] | array index access | right to left
3 | ! | logical not | right to left
~~~~

4 | \- | arithmetic negation | right to left
5 | ++ | string concatenation | left to right
6 | ^ | floating point number power | left to right
7 | * <br> / <br> % | multiplication <br> division <br> remainder | left to right
8 | \+ <br> - | addition <br> subtraction | left to right
9 | ?? | default value of optional type | left to right
10 | and | logical conjunction | left to right
11 | or | logical disjunction | left to right
12 | < <br> > <br> <= <br> >= <br> = <br> != | less than <br> greater than <br> less or equal <br> greater or equal <br> equal <br> not equal | left to right

----
---
id: ref-ergo-stdlib
title: Standard Library

The following libraries are provided with the Ergo compiler.

## Stdlib

The following functions are in the `org.accordproject.ergo.stdlib` namespace and available by default.

### Functions on Integer

Name	Signature	Description
`integerAbs`	`(x:Integer) : Integer`	Absolute value
`integerLog2`	`(x:Integer) : Integer`	Base 2 integer logarithm
`integerSqrt`	`(x:Integer) : Integer`	Integer square root
`integerToDouble`	`(x:Integer) : Double`	Cast to a Double
`integerModulo`	`(x:Integer, y:Integer) : Integer`	Integer remainder
`integerMin`	`(x:Integer, y:Integer) : Integer`	Smallest of `x` and `y`
`integerMax`	`(x:Integer, y:Integer) : Integer`	Largest of `x` and `y`

### Functions on Long

Name	Signature	Description
`longAbs`	`(x:Long) : Long`	Absolute value
`longLog2`	`(x:Long) : Long`	Base 2 long logarithm
`longSqrt`	`(x:Long) : Long`	Long square root
`longToDouble`	`(x:Long) : Double`	Cast to a Double
`longModulo`	`(x:Long, y:Long) : Long`	Long remainder
`longMin`	`(x:Long, y:Long) : Long`	Smallest of `x` and `y`
`longMax`	`(x:Long, y:Long) : Long`	Largest of `x` and `y`

### Functions on Double

Name	Signature	Description
`abs`	`(x:Double) : Double`	Absolute value
`sqrt`	`(x:Double) : Double`	Square root
`exp`	`(x:Double) : Double`	Exponential
`log`	`(x:Double) : Double`	Natural logarithm
`log10`	`(x:Double) : Double`	Base 10 logarithm
`ceil`	`(x:Double) : Double`	Round to closest integer above
`floor`	`(x:Double) : Double`	Round to closest integer below

`truncate`	`(x:Double) : Integer`	Cast to an Integer
`doubleToInteger`	`(x:Double) : Integer`	Same as `truncate`
`doubleToLong`	`(x:Double) : Long`	Cast to a Long
`minPair`	`(x:Double, y:Double) : Double`	Smallest of `x` and `y`
`maxPair`	`(x:Double, y:Double) : Double`	Largest of `x` and `y`

### Functions on String

Name	Signature	Description
`length`	`(x:String) : Integer`	Prints length of a string
`encode`	`(x:String) : String`	Encode as URI component
`decode`	`(x:String) : String`	Decode as URI component
`doubleOpt`	`(x:String) : Double?`	Cast to a Double
`double`	`(x:String) : Double`	Cast to a Double or NaN
`integerOpt`	`(x:String) : Integer?`	Cast to an Integer
`integer`	`(x:String) : Integer`	Cast to a Integer or 0
`longOpt`	`(x:String) : Long?`	Cast to a Long
`long`	`(x:String) : Long`	Cast to a Long or 0

### Functions on Arrays

Name	Signature	Description
`count`	(x:Any[]) : Integer	Number of elements
`flatten`	(x:Any[][]) : Any[]	Flattens a nested array
`arrayAdd`	`(x:Any[],y:Any[]) : Any[]`	Array concatenation
`arraySubtract`	`(x:Any[],y:Any[]) : Any[]`	Removes elements of `y` in `x`
`inArray`	`(x:Any,y:Any[]) : Boolean`	Whether `x` is in `y`
`containsAll`	`(x:Any[],y:Any[]) : Boolean`	Whether all elements of `y` are in `x`
`distinct`	`(x:Any[]) : Any[]`	Duplicates elimination
`singleton`	`(x:Any[]) : Any?`	Single value from singleton array

Note: For most of these functions, the type-checker infers more precise types than indicated here. For instance `concat([1,2],[3,4])` will return `[1,2,3,4]` and have the type `Integer[]`.

### Log functions

Name	Signature	Description
`logString`	(x:String) : Unit	Adds string to the log

### Aggregate functions

Name	Signature	Description
`max`	(x:Double[]) : Double	The largest element in `x`
`min`	(x:Double[]) : Double	The smallest element in `x`
`sum`	(x:Double[]) : Double	Sum of the elements in `x`
`average`	(x:Double[]) : Double	Arithmetic mean

### Math functions

Name	Signature	Description
`acos`	(x:Double) : Double	The inverse cosine of x
`asin`	(x:Double) : Double	The inverse sine of x

`atan`	(x:Double) : Double	The inverse tangent of x
`atan2`	(x:Double, y:Double) : Double	The inverse tangent of `x / y`
`cos`	(x:Double) : Double	The cosine of x
`cosh`	(x:Double) : Double	The hyperbolic cosine of x
`sin`	(x:Double) : Double	The sine of x
`sinh`	(x:Double) : Double	The hyperbolic sine of x
`tan`	(x:Double) : Double	The tangent of x
`tanh`	(x:Double) : Double	The hyperbolic tangent of x

### Other functions

Name	Signature	Description
`failure`	`(x:String) : ErgoErrorResponse`	Ergo error from a string
`toString`	`(x:Any) : String`	Prints any value to a string
`toText`	`(x:Any) : String`	Template variant of `toString` (internal)

## Time

The following functions are in the `org.accordproject.time` namespace and are available by importing that namespace.
They rely on the [time.cto](https://models.accordproject.org/v2.0/time.html) types from the Accord Project models.

### Functions on DateTime

Name	Signature	Description
`now`	`() : DateTime`	Returns the time when execution started
`dateTime`	`(x:String) : DateTime`	Parse a date and time
`getSecond`	`(x:DateTime) : Long`	Second component of a DateTime
`getMinute`	`(x:DateTime) : Long`	Minute component of a DateTime
`getHour`	`(x:DateTime) : Long`	Hour component of a DateTime
`getDay`	`(x:DateTime) : Long`	Day of the month component of a DateTime
`getWeek`	`(x:DateTime) : Long`	Week of the year component of a DateTime
`getMonth`	`(x:DateTime) : Long`	Month component in a DateTime
`getYear`	`(x:DateTime) : Long`	Year component in a DateTime
`isAfter`	`(x:DateTime, y:DateTime) : Boolean`	Whether `x` if after `y`
`isBefore`	`(x:DateTime, y:DateTime) : Boolean`	Whether `x` is before `y`
`isSame`	`(x:DateTime, y:DateTime) : Boolean`	Whether `x` is the same DateTime as `y`
`dateTimeMin`	`(x:DateTime[]) : DateTime`	The earliest in an array of DateTime
`dateTimeMax`	`(x:DateTime[]) : DateTime`	The latest in an array of DateTime
`format`	`(x:DateTime,f:String) : String`	Prints date `x` according to [format](markup-variables#datetime-formats) `f`

### Functions on Duration

Name	Signature	Description
`durationAs`	`(x:Duration, y:TemporalUnit) : Duration`	Change the unit for duration `x` to `y`
`diffDurationAs`	`(x:DateTime, y:DateTime, z:TemporalUnit) : Duration`	Duration between `x` and `y` in unit `z`
`diffDuration`	`(x:DateTime, y:DateTime) : Duration`	Duration between `x` and `y` in seconds
`addDuration`	`(x:DateTime, y:Duration) : DateTime`	Add duration `y` to `x`

| `subtractDuration` | `(x:DateTime, y:Duration) : DateTime` | Subtract duration `y` to `x` |
| `divideDuration` | `(x:Duration, y:Duration) : Double` | Ratio between durations `x` and `y` |

### Functions on Period

Name	Signature	Description
`diffPeriodAs`	`(x:DateTime, y:DateTime, z:PeriodUnit) : Period`	Time period between `x` and `y` in unit `z`
`addPeriod`	`(x:DateTime, y:Period) : DateTime`	Add time period `y` to `x`
`subtractPeriod`	`(x:DateTime, y:Period) : DateTime`	Subtract time period `y` to `x`
`startOf`	`(x:DateTime, y:PeriodUnit) : DateTime`	Start of period `y` nearest to DateTime `x`
`endOf`	`(x:DateTime, y:PeriodUnit) : DateTime`	End of period `y` nearest to DateTime `x`

---
id: ref-errors
title: Errors

As much as possible, errors returned by all projects (notably Cicero and the Ergo compiler) are normalized and categorized in order to facilitate handling of those error by the application code. Those errors are raised as JavaScript _exceptions_.

## Errors Hierarchy

The hierarchy of errors (or exceptions) is shown on the following diagram:

![Error Hierarchy](assets/exceptions.png)

## Errors Model

For reference, those can also be described using the following Concerto model:

```ergo
namespace org.accordproject.errors
 /** Common */
concept LocationPoint {
  o Integer line
  o Integer column
  o Integer offset optional
}
concept FileLocation {
  o LocationPoint start
  o LocationPoint end
}
 concept BaseException {
      o String component // Node component the error originates from
  o String name      // name of the class
  o String message
}
concept BaseFileException extends BaseException {
      o FileLocation fileLocation
```

~~~~
      o String shortMessage
      o String fileName
}
concept ParseException extends BaseFileException {
}
 /* Model errors */
concept ValidationException extends BaseException {
}
concept TypeNotFoundException extends BaseException {
      o String typeName
}
concept IllegalModelException extends BaseFileException {
      o String modelFile
}
 /* Ergo errors */
concept CompilerException extends BaseFileException {
}
concept TypeException extends BaseFileException {
}
concept SystemException extends BaseFileException {
}
 /* Cicero errors */
concept TemplateException extends ParseException {
}
```
--------------------------------------------------------------------------------
---
id: ref-glossary
title: Glossary
---

## Variable

Variables function as ‘placeholders’ for information to be added when a template is
used. Variables are the information that will change between usages of a Template.
Here as an example of the text of a contract with three Variables defined:

```
Upon the signing of this Agreement, {{buyer}} shall pay {{amount}} to {{seller}}.
```

## Data Model

A Data Model is used to express the variables that are contained within a Template
in a structured way. A Data Model provides us with the structure of the ‘fields’ in
the contract which are completed when the templated is used for an agreement. For
example, ‘Price’ would be a variable name against which a value, say $100, is
entered. This enables us to create a contract document that has a definite
structure underlying it rather than simply lines of text.

The Data Model is used to create a machine-readable representation of the text of
the contract. It does this by creating a link with the text of the contract by
defining the Variables that should exist within the contract along with an
associated data type. The linkage between the text and the data model is created by
referencing the variable in both the model and the text.


For example, if the text is:
```
~~~~

Upon the signing of this Agreement, {{buyer}} shall pay {{amount}} to {{seller}}.

The model will include `buyer`, `amount`, and `seller`. A data type and a value is assigned against each of these variables.

### Components of Data Models

Data models consist of two core components:

- Variable Name: The name of the ‘placeholder’ for data to be added into a template to create an instance of the contract. By naming variables, we are able to specify what data should be entered into that placeholder in the contract.
- Data Type: The data type defines what type, or format, of data should be inserted in the ‘placeholder’.

The **value** is the actual data that is input into the ‘placeholder’. The value is displayed instead of the name of the variable in the Text. For example:

```md
Upon the signing of this Agreement, "Steve" shall pay 100.0 USD to "Dan".
```

In the model, this is represented as:

Variable Name	Data type	Value
buyer	`String`	Steve
amount	`MonetaryAmount`	100.0 USD
seller	`String`	Dan

Here, `amount` should be a combination of a decimalized value (a double) and a currency code, as opposed to an alphanumeric value, and `buyer` and `seller` should be a combination of alphanumeric characters. This ensures that invalid data cannot be added into the contract, much like letters cannot be added into a credit card section of a web form.

---
id: ref-markus-cli
title: Command Line

Install the `@accordproject/markdown-cli` npm package to access the Markdown Transform command line interface (CLI). After installation you can use the `markus` command and its sub-commands as described below.

To install the Markdown CLI:
```bash
npm install -g @accordproject/markdown-cli
```

## Usage

`markus` is a command line tool to debug and use markdown transformations.

```md
markus <cmd> [args]
```

~~~
Commands:
  markus transform  transform between two formats

Options:
  --version      Show version number                                   [boolean]
  --verbose, -v                                                 [default: false]
  --help         Show help                                             [boolean]
```

## markus transform

The `markus transform` command lets you transform between any two of the supported
formats

```md
markus transform

transform between two formats

Options:
  --version      Show version number                                   [boolean]
  --verbose, -v  verbose output                       [boolean] [default: false]
  --help         Show help                                             [boolean]
  --input        path to the input                                      [string]
  --from         source format                    [string] [default: "markdown"]
  --to           target format                  [string] [default: "commonmark"]
  --via          intermediate formats                      [array] [default: []]
  --roundtrip    roundtrip transform                  [boolean] [default: false]
  --output       path to the output file                                [string]
  --model        array of concerto model files                           [array]
  --template     template grammar                                       [string]
  --contract     contract template                    [boolean] [default: false]
  --currentTime  set current time                        [string] [default: null]
  --plugin       path to a parser plugin                                [string]
  --sourcePos    enable source position               [boolean] [default: false]
  --offline      do not resolve external models       [boolean] [default: false]
```

### Example

For example, you can use the `transform` command on the `README.md` file from the
[Hello World](https://github.com/accordproject/cicero-template-library/blob/
master/src/helloworld) template:

```bash
markus transform --input README.md
```

returns:
```json
{
  "$class": "org.accordproject.commonmark.Document",
  "xmlns": "http://commonmark.org/xml/1.0",
  "nodes": [
    {
      "$class": "org.accordproject.commonmark.Heading",
      "level": "1",
      "nodes": [
~~~

~~~
        {
          "$class": "org.accordproject.commonmark.Text",
          "text": "Hello World"
        }
      ]
    },
    {
      "$class": "org.accordproject.commonmark.Paragraph",
      "nodes": [
        {
          "$class": "org.accordproject.commonmark.Text",
          "text": "This is the Hello World of Accord Project Templates. Executing
the clause will simply echo back the text that occurs after the string "
        },
        {
          "$class": "org.accordproject.commonmark.Code",
          "text": "Hello"
        },
        {
          "$class": "org.accordproject.commonmark.Text",
          "text": " prepended to text that is passed in the request."
        }
      ]
    }
  ]
}
```

### `--from` and `--to` options

You can indicate the source and target formats using the `--from` and `--to`
options. For instance, the following transforms from `markdown` to `html`:

```bash
markus transform --from markdown --to html
```

returns:

```md
<html>
<body>
<div class="document">
<h1>Hello World</h1>
<p>This is the Hello World of Accord Project Templates. Executing the clause will
simply echo back the text that occurs after the string <code>Hello</code> prepended
to text that is passed in the request.</p>
</div>
</body>
</html>
```

### `--via` option

When there are several paths between two formats, you can indicate an intermediate
format using the `--via` option. The following transforms from `markdown` to `html`
_via_ `slate`:

```bash
~~~

~~~~
markus transform --from markdown --via slate --to html
```

returns:

```md
<html>
<body>
<div class="document">
<h1>Hello World</h1>
<p>This is the Hello World of Accord Project Templates. Executing the clause will
simply echo back the text that occurs after the string <code>Hello</code> prepended
to text that is passed in the request.</p>
</div>
</body>
</html>
```

### `--roundtrip` option

When the transforms allow, you can roundtrip between two formats, i.e., transform
from a source to a target format and back to the source target. For instance, the
following transform from `markdown` to `slate` and back to markdown:

```md
markus transform --from markdown --to slate --input README.md --roundtrip
```

returns:

```bash
Hello World
====

This is the Hello World of Accord Project Templates. Executing the clause will
simply echo back the text that occurs after the string `Hello` prepended to text
that is passed in the request.
```

:::
Roundtripping might result in small changes in the source markdown, but should
always be semantically equivalent. In the above example the source ATX heading `#
Hello World` has been transformed into a Setext heading equivalent.
:::

### `--model` `--contract` options

When handling [TemplateMark](markup-templatemark), one has to provide a model using
the `--model` option and whether the template is a clause (default) or a contract
(using the `--contract` option).

For instance the following converts markdown with the template extension to a
TemplateMark document object model:
```bash
markus transform --from markdown_template --to templatemark --model model/model.cto
--input text/grammar.tem.md
```

returns:
~~~~

```json
{
  "$class": "org.accordproject.commonmark.Document",
  "xmlns": "http://commonmark.org/xml/1.0",
  "nodes": [
    {
      "$class": "org.accordproject.templatemark.ClauseDefinition",
      "name": "top",
      "elementType": "org.accordproject.helloworld.HelloWorldClause",
      "nodes": [
        {
          "$class": "org.accordproject.commonmark.Paragraph",
          "nodes": [
            {
              "$class": "org.accordproject.commonmark.Text",
              "text": "Name of the person to greet: "
            },
            {
              "$class": "org.accordproject.templatemark.VariableDefinition",
              "name": "name",
              "elementType": "String"
            },
            {
              "$class": "org.accordproject.commonmark.Text",
              "text": "."
            },
            {
              "$class": "org.accordproject.commonmark.Softbreak"
            },
            {
              "$class": "org.accordproject.commonmark.Text",
              "text": "Thank you!"
            }
          ]
        }
      ]
    }
  ]
}
```

### `--template` option

Parsing or drafting contract text using a template can be done using the `--template` option, usually with the corresponding `--model` option to indicate the template model.

For instance, the following parses a markdown with CiceroMark extension to get the correspond contract data:
```bash
markus transform --from markdown_cicero --to data --template text/grammar.tem.md --model model/model.cto --input text/sample.md
```

returns:

```json
{
```

~~~
  "$class": "org.accordproject.helloworld.HelloWorldClause",
  "name": "Fred Blogs",
  "clauseId": "fc345528-2604-420c-9e02-8d85e03cb65b"
}
```

--------------------------------------------------------------------------------
---
id: ref-migrate-0.13-0.20
title: Cicero 0.13 to 0.20
---

Much has changed in the `0.20` release. This guide provides step-by-step
instructions to port your Accord Project templates from version `0.13` or earlier
to version `0.20`.

:::note
Before following those migration instructions, make sure to first install version
`0.20` of Cicero, as described in the [Install Cicero](started-installation.md)
Section of this documentation.
:::

## Metadata Changes

You will first need to update the `package.json` in your template. Remove the Ergo
version which is now unnecessary, and change the Cicero version to `^0.20.0`.

#### Example

After those changes, the `accordproject` field in your `package.json` should look
as follows (with the `template` field being either `clause` or `contract` depending
on the template):
```js
...
    "accordproject": {
        "template": "clause",
        "cicero": "^0.20.0"
    }
...
```

## Template Directory Changes

The layout of templates has changed to reflect the conceptual notion of Accord
Project templates (as a triangle composed of text, model and logic). To migrate a
template directory from version `0.13` or earlier to the new `0.20` layout:
1. Rename your `lib` directory to `logic`
2. Rename your `models` directory to `model`
3. Rename your `grammar` directory to `text`
4. Rename your template grammar from `text/template.tem` to `text/grammar.tem.md`
5. Rename your samples from `sample.txt` to `text/sample.md` (or generally any
other `sample*.txt` files to `text/sample*.md`)

#### Example

Consider the [late delivery and
penalty](https://templates.accordproject.org/latedeliveryandpenalty@0.14.1.html)
clause. After applying those changes, the template directory should look as
follows:
~~~

```
./.cucumber.js
./README.md
./package.json
./request-forcemajeure.json
./request.json
./state.json

./logic:
./logic/logic.ergo

./model:
./model/clause.cto

./test:
./test/logic.feature
./test/logic_default.feature

./text:
./text/grammar.tem.md
./text/sample-noforcemajeure.md
./text/sample.md
```

## Text Changes

Both grammar and sample text for the templates has changed to support rich text annotations through CommonMark and a new syntax for variables. You can find complete information about the new syntax in the [CiceroMark](markup-cicero) Section of this documentation. For an existing template, you should apply the following changes.

### Text Grammar Changes

1. Variables should be changed from `[{variableName}]` to `{{variableName}}`
2. Formatted variables should be changed to from `[{variableName as "FORMAT"}]` to `{{variableName as "FORMAT"}}`
3. Boolean variables should be changed to use the new block syntax, from `[{"This is a force majeure":?forceMajeure}]` to `{{#if forceMajeure}}This is a force majeure{{/if}}`
4. Nested clauses should be changed to use the new block syntax, from `[{#payment}]As consideration in full for the rights granted herein...[{/payment}]` to `{{#clause payment}}As consideration in full for the rights granted herein... {{/clause}}`

:::note
1. Template text is now interpreted as CommonMark which may lead to unexpected results if your text includes CommonMark characters or structure (e.g., `#` or `##` now become headings; `1.` or `-` now become lists). You should review both the grammar and samples so they follow the proper [CommonMark](https://commonmark.org) rules.
2. The new lexer reserves `{{` instead of reserving `[{` which means you should avoid using `{{` in your text unless for Accord Project variables.
:::

### Text Samples Changes

You should ensure that any changes to the grammar text is reflected in the samples. Any change in the grammar text outside of variables should be applied to the corresponding `sample.md` files as well.

:::tip
You can check that the samples and grammar are in agreement by using the `cicero parse` command.
:::

#### Example

Consider the text grammar for the [late delivery and penalty](https://templates.accordproject.org/latedeliveryandpenalty@0.14.1.html) clause:

```md
Late Delivery and Penalty.
In case of delayed delivery[{" except for Force Majeure cases,":? forceMajeure}]
[{seller}] (the Seller) shall pay to [{buyer}] (the Buyer) for every
[{penaltyDuration}]
of delay penalty amounting to [{penaltyPercentage}]% of the total value of the
Equipment
whose delivery has been delayed. Any fractional part of a [{fractionalPart}] is to
be
considered a full [{fractionalPart}]. The total amount of penalty shall not
however,
exceed [{capPercentage}]% of the total value of the Equipment involved in late
delivery.
If the delay is more than [{termination}], the Buyer is entitled to terminate this
Contract.
```

After applying the above rules to the code for the `0.13` version, and identifying the heading for the clause using the new markdown features, the grammar text becomes:

```tem
## Late Delivery and Penalty.

In case of delayed delivery{{#if forceMajeure}} except for Force Majeure
cases,{{/if}}
{{seller}} (the Seller) shall pay to {{buyer}} (the Buyer) for every
{{penaltyDuration}}
of delay penalty amounting to {{penaltyPercentage}}% of the total value of the
Equipment
whose delivery has been delayed. Any fractional part of a {{fractionalPart}} is to
be
considered a full {{fractionalPart}}. The total amount of penalty shall not
however,
exceed {{capPercentage}}% of the total value of the Equipment involved in late
delivery.
If the delay is more than {{termination}}, the Buyer is entitled to terminate this
Contract.
```

To make sure the `sample.md` file parses as well, the heading needs to be similarly identified using markdown:
```md
## Late Delivery and Penalty.

In case of delayed delivery except for Force Majeure cases,
"Dan" (the Seller) shall pay to "Steve" (the Buyer) for every 2 days
of delay penalty amounting to 10.5% of the total value of the Equipment
whose delivery has been delayed. Any fractional part of a days is to be
considered a full days. The total amount of penalty shall not however,
exceed 55% of the total value of the Equipment involved in late delivery.
If the delay is more than 15 days, the Buyer is entitled to terminate this
Contract.
```

## Model Changes

There is no model changes required for this version.

## Logic Changes

Version `0.20` of Ergo has a few new features that are non backward compatible with version `0.13`. Those may require you to change your template logic. The main non-backward compatible feature is the new support for enumerated values.

### Enumerated Values

Enumerated values are now proper values with a proper enum type, and not based on the type `String` anymore.

For instance, consider the enum declaration:
```js
enum Cardsuit {
  o CLUBS
  o DIAMONDS
  o HEARTS
  o SPADES
}
```

In version `0.13` or earlier the Ergo code would write `"CLUBS"` for an enum value and treat the type `Cardsuit` as if it was the type `String`.

As of version `0.20` Ergo writes `CLUBS` for that same enum value and the type `Cardsuit` is now distinct from the type `String`.

If you try to compile Ergo logic written for version `0.13` or earlier that features enumerated values, the compiler will likely throw type errors. You should apply the following changes:

1. Remove the quotes (`"`) around any enum values in your logic. E.g., `"USD"` should now be replaced by `USD` for monetary amounts;
3. If enum values are bound to variables with a type annotation, you should change the type annotation from `String` to the correct enum type. E.g., `let x : String = "DIAMONDS"; ...` should become `let x : Cardsuit = DIAMONDS; ...`;
3. If enum values are passed as parameters in clauses or functions, you should change the type annotation for that parameter from `String` to the correct enum type.
4. In a few cases the same enumerated value may be used in different enum types (e.g., `days` and `weeks` are used in both `TemporalUnit` and `PeriodUnit`). Those two values will now have different types. If you need to distinguish, you can use the fully qualified name for the enum value (e.g., `~org.accordproject.time.TemporalUnit.days` or `~org.accordproject.time.PeriodUnit.days`).

### Other Changes

1. `now` used to return the current time but is treated in `0.20` like any other variables. If your logic used the variable `now` without declaring it, this will raise a `Variable now not found` error. You should change your logic to use the `now()` function instead.

#### Example

Consider the Ergo logic for the [acceptance of delivery](https://templates.accordproject.org/acceptance-of-delivery@0.12.1.html) clause. Applying the above rules to the code for the `0.13` version:

```ergo
  clause acceptanceofdelivery(request : InspectDeliverable) : InspectionResponse {

    let received = request.deliverableReceivedAt;
    enforce isBefore(received,now) else
      throw ErgoErrorResponse{ message : "Transaction time is before the
deliverable date." }
    ;

    let dur =
      Duration{
        amount: contract.businessDays,
        unit: "days"
      };
    let status =
      if isAfter(now(), addDuration(received, dur))
      then "OUTSIDE_INSPECTION_PERIOD"
      else if request.inspectionPassed
      then "PASSED_TESTING"
      else "FAILED_TESTING"
    ;
    return InspectionResponse{
      status : status,
      shipper : contract.shipper,
      receiver : contract.receiver
    }
  }
```

results in the following new logic for the `0.20` version:

```ergo
  clause acceptanceofdelivery(request : InspectDeliverable) : InspectionResponse {

    let received = request.deliverableReceivedAt;
    enforce isBefore(received,now()) else                // changed to now()
      throw ErgoErrorResponse{ message : "Transaction time is before the
deliverable date." }
    ;

    let dur =
      Duration{
        amount: contract.businessDays,
        unit: ~org.accordproject.time.TemporalUnit.days  // enum value with fully
qualified name
      };
    let status =
      if isAfter(now(), addDuration(received, dur))      // changed to now()
      then OUTSIDE_INSPECTION_PERIOD                     // enum value has no
quotes
      else if request.inspectionPassed
      then PASSED_TESTING                                // enum value has no
quotes
      else FAILED_TESTING                                // enum value has no
quotes
    ;
    return InspectionResponse{
      status : status,
      shipper : contract.shipper,
      receiver : contract.receiver
    }
  }
```

## Command Line Changes

The Command Line interface for Cicero and Ergo has been completely overhauled for consistency. Release `0.20` also features new command line interfaces for Concerto and for the new `markdown-transform` project.

If you are familiar with the previous Accord Project command line interfaces (or if you have scripts relying on the previous version of the command line), here is a list of changes:

1. Ergo: A single new `ergo` command replaces both `ergoc` and `ergorun`
   - `ergoc` has been replaced by `ergo compile`
   - `ergorun execute` has been replaced by `ergo trigger`
   - `ergorun init` has been replaced by `ergo initialize`
   - All other `ergorun <command>` commands should use `ergo <command>` instead
2. Cicero:
   - The `cicero execute` command has been replaced by `cicero trigger`
   - The `cicero init` command has been replaced by `cicero initialize`
   - The `cicero generateText` command has been replaced by `cicero draft`
   - the `cicero generate` command has been replaced by `cicero compile`

Note that several options have been renamed for consistency as well. Some of the main option changes are:
1. `--out` and `--outputDirectory` have both been replaced by `--output`
2. `--format` has been replaced by `--target` in the new `cicero compile` command
3. `--contract` has been replaced by `--data` in all `ergo` commands

For more details on the new command line interface, please consult the corresponding [Cicero CLI](cicero-cli), [Concerto CLI](concerto-cli), [Ergo CLI](ergo-cli), and [Markus CLI](markus-cli) Sections in the reference manual.

## API Changes

A number of API changes may affect Node.js applications using Cicero or Ergo packages. The main API changes are:
1. Ergo:
   1. `@accordproject/ergo-engine` package
      - the `Engine.execute()` call has been renamed `Engine.trigger()`
2. Cicero:
   1. `@accordproject/cicero-core` package
      - the `TemplateInstance.generateText()` call has been renamed

`TemplateInstance.draft` **and is now an `async` call**
      - the `Metadata.getErgoVersion()` call has been removed
   2. `@accordproject/cicero-engine` package
      - the `Engine.execute()` call has been renamed `Engine.trigger()`
      - the `Engine.generateText()` call has been renamed `Engine.draft()`

## Cicero Server Changes

Cicero server API has been changed to reflect the new underlying Cicero engine.
Specifically:
1. The `execute` endpoint has been renamed `trigger`
2. The path to the sample has to include the `text` directory, so instead of
`execute/templateName/sample.txt` it should use `trigger/templateName/text
%2Fsample.md`

#### Example

Following the
[README.md](https://github.com/accordproject/cicero/blob/master/packages/cicero-
server/README.md) instructions, instead of calling:
```
curl -X POST --header 'Content-Type: application/json' --header 'Accept:
application/json' http://localhost:6001/execute/latedeliveryandpenalty/sample.txt -
d '{ "request": { "$class":
"org.accordproject.latedeliveryandpenalty.LateDeliveryAndPenaltyRequest",
"forceMajeure": false,"agreedDelivery": "December 17, 2017 03:24:00",
"deliveredAt": null, "goodsValue": 200.00 }, "state": { "$class":
"org.accordproject.cicero.contract.AccordContractState", "stateId" :
"org.accordproject.cicero.contract.AccordContractState#1"}}'
```

You should call:
```
curl -X POST --header 'Content-Type: application/json' --header 'Accept:
application/json' http://localhost:6001/trigger/latedeliveryandpenalty/sample.md -d
'{ "request": { "$class":
"org.accordproject.latedeliveryandpenalty.LateDeliveryAndPenaltyRequest",
"forceMajeure": false,"agreedDelivery": "December 17, 2017 03:24:00",
"deliveredAt": null, "goodsValue": 200.00 }, "state": { "$class":
"org.accordproject.cicero.contract.AccordContractState", "stateId" :
"org.accordproject.cicero.contract.AccordContractState#1"}}'
```

---
id: ref-migrate-0.20-0.21
title: Cicero 0.20 to 0.21

The main change between the `0.20` release and the `0.21` release is the new
markdown syntax and parser infrastructure based on
[`markdown-it`](https://github.com/markdown-it/markdown-it). While most templates
designed for `0.20` should still work on `0.21` some changes might be needed to the
contract or template text to account for this new syntax.

:::note
Before following those migration instructions, make sure to first install version
`0.21` of Cicero, as described in the [Install Cicero](started-installation.md)

Section of this documentation.
:::

## Metadata Changes

You should only have to update the Cicero version in the `package.json` for your template to `^0.21.0`. Remember to also increment the version number for the template itself.

#### Example

After those changes, the `accordproject` field in your `package.json` should look as follows (with the `template` field being either `clause` or `contract` depending on the template):
```js
...
    "accordproject": {
        "template": "clause",
        "cicero": "^0.21.0"
    }
...
```

## Text Changes

Both the markdown for the grammar and sample text have been updated and consolidated in the `0.21` release and may require some adjustments. You can find complete information about the latest syntax in the [Markdown Text](markup-preliminaries) Section of this documentation. For an existing template, you should apply the following changes.

### Text Grammar Changes

1. Clause or list blocks should have their opening and closing tags on a single line terminated by whitespace. I.e., you should change occurrences of :
   ```
   {{#clause clauseName}}...clause text...{{/clauseName}}
   ```
   to
   ```
   {{#clause clauseName}}
   ...clause text...
   {{/clauseName}}
   ```
   and similarly for ordered and unordeded list blocks (`olist` and `ulist`).
2. Markdown container blocks are no longer supported inside inline blocks (`if` `join` and `with` blocks).

:::tip
We recommend using the new [TemplateMark Dingus](https://templatemark-dingus.netlify.app) to check that your template variables, blocks and formula are properly identified following the new markdown parsing rules.
:::

### Text Samples Changes

1. Nested clause template should be now identified within contract templates using clause blocks. I.e., if you use a `paymentClause`, you should change your text from:

~~~~
   ```
   ...negate the notices Licensor provides and requires hereunder.

   Payment. As consideration in full for the rights granted herein, Licensee shall
pay Licensor a one-time fee in the amount of "one hundred US Dollars" (100.0 USD)
upon execution of this Agreement, payable as follows: "bank transfer".

   General.
   ...
   ```
   to
   ```
   ...negate the notices Licensor provides and requires hereunder.

   {{#clause paymentClause}}
   Payment. As consideration in full for the rights granted herein, Licensee shall
pay Licensor a one-time fee in the amount of "one hundred US Dollars" (100.0 USD)
upon execution of this Agreement, payable as follows: "bank transfer".
   {{/clause}}

   General.
   ...
   ```
2. The text corresponding to formulas should be changed from `{{ ...text...}}` to
`{{% ...text... %}}`.

You should also ensure that any changes to the grammar text is reflected in the
samples. Any change in the grammar text outside of variables should be applied to
the corresponding `sample.md` files as well.

:::tip
You can check that the samples and grammar are in agreement by using the `cicero
parse` command.
:::

## Model Changes

There should be no model changes required for this version.

## Logic Changes

There should be no logic changes required for this version.

## API Changes

A number of API changes may affect Node.js applications using Cicero or Ergo
packages. The main API changes are:
1. Cicero:
   1. `@accordproject/cicero-core` package
      - the `ParserManager` class has been completely overhauled and moved to the
`@accordproject/@markdown-template` package.

## CLI Changes

1. The `cicero draft --wrapVariables` option has been removed
2. The `ergo draft` command has been removed

## Cicero Server Changes
~~~~

Cicero server API has been completely overhauled to match the more recent engine interface
1. The contract data is now passed as part of the REST POST request for the `trigger` endpoint

----
---
id: ref-migrate-0.21-0.22
title: Cicero 0.21 to 0.22

The main change between the `0.21` release and the `0.22` release is the switch to version `1.0` of the Concerto modeling language and library. This change comes along with a complete revision for the Accord Project "base models" which define key types for: clause and contract data, parties, obligations and requests / responses. We encourage developers to get familiarized with the [new base models](https://github.com/accordproject/models/tree/master/src/accordproject) before switching to Cicero `0.22`.

:::note
Before following those migration instructions, make sure to first install version `0.22` of Cicero, as described in the [Install Cicero](started-installation.md) Section of this documentation.
:::

## Metadata Changes

You should only have to update the Cicero version in the `package.json` for your template to `^0.22.0`. Remember to also increment the version number for the template itself.

#### Example

After those changes, the `accordproject` field in your `package.json` should look as follows (with the `template` field being either `clause` or `contract` depending on the template):
```js
...
    "accordproject": {
        "template": "clause",
        "cicero": "^0.22.0"
    }
...
```

## Text Changes

There should be no text changes required for this version.

## Model Changes

Most templates will require changes to the model and should be re-written against the new base Accord Project models. Most of the changes should be renaming for key classes:

1. Contract and Clause data
   1. the `org.accordproject.cicero.contract.AccordContract` class is now `org.accordproject.contract.Contract` found in

~~~~
https://models.accordproject.org/accordproject/contract.cto
   2. the `org.accordproject.cicero.contract.AccordClause` class is now
`org.accordproject.contract.Clause` found in
https://models.accordproject.org/accordproject/contract.cto
2. Contract state and parties
   1. the `org.accordproject.cicero.contract.AccordState` class is now
`org.accordproject.runtime.State` found in
https://models.accordproject.org/accordproject/runtime.cto
   2. the `org.accordproject.cicero.contract.AccordParty` class is now
`org.accordproject.party.Party` found in
https://models.accordproject.org/accordproject/party.cto
3. Request and response
   1. the `org.accordproject.cicero.runtime.Request` class is now
`org.accordproject.runtime.Request` found in
https://models.accordproject.org/accordproject/runtime.cto
   2. the `org.accordproject.cicero.runtime.Response` class is now
`org.accordproject.runtime.Response` found in
https://models.accordproject.org/accordproject/runtime.cto
4. Predefined obligations have been moved to their own model file found in
https://models.accordproject.org/accordproject/obligation.cto

:::warning
Some of the properties in those base classes have changed, e.g., the contract state
no longer requires a `stateId`. As a result, corresponding changes to the contract
logic in Ergo or to the application code may be required.
:::

### Example

A typical change to a template model might look as follows, from:
```ergo
import org.accordproject.cicero.contract.* from
https://models.accordproject.org/cicero/contract.cto
import org.accordproject.cicero.runtime.* from
https://models.accordproject.org/cicero/runtime.cto

/**
 * Defines the data model for the Purchase Order Failure
 * template.
 */
asset PurchaseOrderFailure extends AccordContract {
  o AccordParty buyer
  ...
}
```
To:
```ergo
import org.accordproject.contract.* from
https://models.accordproject.org/accordproject/contract.cto
import org.accordproject.runtime.* from
https://models.accordproject.org/accordproject/runtime.cto
import org.accordproject.party.* from
https://models.accordproject.org/accordproject/party.cto
import org.accordproject.obligation.* from
https://models.accordproject.org/accordproject/obligation.cto

asset PurchaseOrderFailure extends Contract {
  --> Party buyer
  ...
~~~~

}
```

Logic Changes

Minimal changes to the contract logic should be required, however a few changes to the base models may affect your Ergo code. Notably:
1. You should import the new Accord Project core models as needed
2. The contract state no longer requires a `stateId` field.
3. The base contract state has been moved to the runtime model, which may need to be imported

API Changes

A number of API changes may affect Node.js applications using Cicero or Ergo packages. The main API changes are:
1. Additional `utcOffset` parameter.
   1. `@accordproject/cicero-core` package
      - the `TemplateInstance.parse` and `TemplateInstance.draft` calls take an additional `utcOffset` parameter to specify the current timezone offset
   2. `@accordproject/cicero-engine` package
      - the `Engine.init`, `Engine.invoke` and `Engine.trigger` calls take an additional `utcOffset` parameter to specify the current timezone offset
   3. `@accordproject/ergo-engine` package
      - the `Engine.init`, `Engine.invoke` and `Engine.trigger` calls take an additional `utcOffset` parameter to specify the current timezone offset
2. New `es6` compilation target for Ergo.
   1. `@accordproject/ergo-compiler` package
      - the `Compiler.compileToJavaScript` compilation target `cicero` has been renamed to `es6`
   2. `@accordproject/cicero-core` package
      - the `Template.toArchive` compilation target `cicero` has been renamed to `es6`

CLI Changes

1. Specific UTC timezone offset now needs to be passed using the new option `--utcOffset` option has been removed

Cicero Server Changes

There should be no text changes required for this version.

---
id: ref-web-components-overview
title: Overview

Accord Project publishes [React](https://reactjs.org) user interface components for use in web applications. Please refer to the web-components project [on GitHub](https://github.com/accordproject/web-components) for detailed usage instructions.

You can preview these components in [the project's storybook](https://ap-web-components.netlify.app/).

Contract Editor

The Contract Editor component provides a rich-text content editor for contract text
```

with embedded clauses.

> Note that the contract editor does not currently support the full expressiveness of Cicero Templates. Please refer to the Limitations section for details.

### Limitations

The contract editor does not support templates which use the following CiceroMark features:

* Lists containing [nested lists](markup-commonmark.md#nested-lists)
* List blocks [list blocks](markup-commonmark.md#list-blocks)

## Error Logger

The Error Logger component is used to display structured error messages from the Contract Editor.

## Navigation

The Navigation component displays an outline view for a contract, allowing the user to quickly navigate between sections.

## Library

The Library component displays a vertical list of library item metadata, and allows the user to add an instance of a library item to a contract.

--------------------------------------------------------------------------------
---
id: started-hello
title: Hello World Template
---

Once you have installed Cicero, you can try it on an existing Accord Project template. This explains how to create an instance of that template and how to run the contract logic.

## Download a Template

You can download a single clause or contract template from the [Accord Project Template Library](https://templates.accordproject.org) as an archive (`.cta`) file. Cicero archives are files with a `.cta` extension, which includes all the different components for the template (text, model and logic).

If you click on the Template Library link, you should see a Web Page which looks as follows:

![Basic-Use-1](/docs/assets/basic/use1.png)

Scrolling down that page, you can see the index for the open-source templates along with their version, and whether they are a Clause or Contract template.

Click on the link to the `helloworld` template. You should be taken to a page which looks as follows:

![Basic-Use-2](/docs/assets/basic/use2.png)

Then click on the `Download Archive` button under the description for the template (highlighted in the red box in the figure). This should download the latest template archive for the `helloworld` template.

## Parse: Extract Deal Data from Text

You can use Cicero to extract deal data from a contract text using the `cicero parse` command.

### Parse Valid Text

Using your terminal, change into the directory (or `cd` into the directory) that contains the template archive you just downloaded, then create a sample clause text `sample.md` which contains the following text:

```md
Name of the person to greet: "Fred Blogs".
Thank you!
```

Then run the `cicero parse` command in your terminal to load the template and parse your sample clause text. This should be echoing the result of parsing back to your terminal.

```bash
cicero parse --template helloworld@0.14.0.cta --sample sample.md
```

:::note
* Templates are tied to a specific version of the cicero tool. Make sure that the version number output from `cicero --version` is compatible with the template. Look for `^0.22.0` or similar at the top of the template web page.
* `cicero parse` requires network access. Make sure that you are online and that your firewall or proxy allows access to `https://models.accordproject.org`
:::

This should extract the data (or "deal points") from the text and output:

```json
{
  "$class": "org.accordproject.helloworld.HelloWorldClause",
  "name": "Fred Blogs",
  "clauseId": "71045314-acfc-441f-92b4-0a2707ea6146",
  "$identifier": "71045314-acfc-441f-92b4-0a2707ea6146"
}
```

You can save the result of `cicero parse` into a file using the `--output` option:
```
cicero parse --template helloworld@0.14.0.cta --sample sample.md --output data.json
```

### Parse Non-Valid Text

If you attempt to parse text which is not valid according to the template, this same command should return an error.

Edit your `sample.md` file to add text that is not consistent with the template:

```text
```

FUBAR Name of the person to greet: "Fred Blogs".
Thank you!

Then run `cicero parse --template helloworld@0.14.0.cta --sample sample.md` again.
The output should now be:

```text
2:13:15 AM - error: Parse error at line 1 column 1
FUBAR Name of the person to greet: "Fred Blogs".
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Expected: 'Name of the person to greet: '
```

## Draft: Create Text from Deal Data

You can use Cicero to create new contract text from deal data using the `cicero
draft` command.

### Draft from Valid Data

If you have saved the deal data earlier in a `data.json` file, you can edit it to
change the name from `Fred Blogs` to `John Doe`, or create a brand new `data.json`
file containing:
```json
{
  "$class": "org.accordproject.helloworld.HelloWorldClause",
  "clauseId": "aa3b9db9-f25f-41f4-88a4-64baba728bfe",
  "name": "John Doe"
}
```

Then run the `cicero draft` command in your terminal:
```
cicero draft --template helloworld@0.14.0.cta --data data.json
```

This should create a new contract text and output:
```
13:17:18 - INFO: Name of the person to greet: "John Doe".
Thank you!
```

You can save the result of `cicero draft` into a file using the `--output` option:
```
cicero draft --template helloworld@0.14.0.cta --data data.json --output new-
sample.md
```

### Draft from Non-Valid Data

If you attempt to draft from data which is not valid according to the template,
this same command should return an error.

Edit your `data.json` file so that the `name` variable is missing:
```json
{
  "$class": "org.accordproject.helloworld.HelloWorldClause",
  "clauseId": "aa3b9db9-f25f-41f4-88a4-64baba728bfe"
```

}
```

Then run `cicero draft --template helloworld@0.14.0.cta --data data.json` again. The output should now be:
```
13:38:11 - ERROR: Instance org.accordproject.helloworld.HelloWorldClause#6f91e060-f837-4108-bead-63891a91ce3a missing required field name

## Trigger: Run the Contract Logic

You can use Cicero to run the logic associated to a contract using the `cicero trigger` command.

### Trigger with a Valid Request

Use the `cicero trigger` command to parse a clause text based (your `sample.md`) *then* send a request to the clause logic.

To do so, you first create one additional file `request.json` which contains:
```json
{
  "$class": "org.accordproject.helloworld.MyRequest",
  "input": "Accord Project"
}
```

This is the request which you will send to trigger the execution of your contract.

Then run the `cicero trigger` command in your terminal to load the template, parse your clause text *and* send the request. This should be echoing the result of execution back to your terminal.

```bash
cicero trigger --template helloworld@0.14.0.cta --sample sample.md --request request.json
```

This should print this output:

```json
13:42:29 - INFO:
{
  "clause": "helloworld@0.14.0-767ffde65292f2f4e8aa474e76bb5f923b80aa29db635cd42afebb6a0cd4c1fa",
  "request": {
    "$class": "org.accordproject.helloworld.MyRequest",
    "input": "Accord Project"
  },
  "response": {
    "$class": "org.accordproject.helloworld.MyResponse",
    "output": "Hello Fred Blogs Accord Project",
    "$timestamp": "2021-06-16T11:38:42.011-04:00"
  },
  "state": {
    "$class": "org.accordproject.runtime.State",
    "$identifier": "f4428ec2-73ca-442b-8006-8e9a290930ad"
  },
```

~~~
  "emit": []
}
```

The results of execution displayed back on your terminal is in JSON format. It
includes the following information:

* Details of the `clause` being triggered (name, version, SHA256 hash of the
template)
* The incoming `request` object (the same request from your `request.json` file)
* The output `response` object
* The output `state` (unchanged in this example)
* An array of `emit`ted events (empty in this example)

That's it! You have successfully parsed and executed your first Accord Project
Clause using the `helloworld` template.

### Trigger with a Non-Valid Request

If you attempt to trigger the contract from a request which is not valid according
to the template, this same command should return an error.

Edit your `request.json` file so that the `input` variable is missing:
```json
{
  "$class": "org.accordproject.helloworld.MyRequest"
}
```

Then run `cicero trigger --template helloworld@0.14.0.cta --sample sample.md --
request request.json` again. The output should now be:
```
13:47:35 - ERROR: Instance org.accordproject.helloworld.MyRequest#null missing
required field input
```

## What Next?

### Try Other Templates

Feel free to try the same commands to parse and execute other templates from the
Accord Project Library. Note that for each template, you can find samples for the
text, for the request and for the state on the corresponding Web page. For
instance, a sample for the [Late Delivery And
Penalty](https://templates.accordproject.org/latedeliveryandpenalty@0.15.0.html)
clause is in the red box in the following image:

![Basic-Use-3](/docs/assets/basic/use3.png)

### More About Cicero

You can find more information on how to create or publish Accord Project templates
in the [Work with Cicero](tutorial-templates) tutorials.

### Run on Different Platforms

Templates may be executed on different platforms, not only from the command line.
You can find more information on how to execute Accord Project templates on
different platforms (Node.js, Hyperledger Fabric, etc.) in the [Template Execution]
~~~

(tutorial-nodejs) tutorials.

---
id: started-installation
title: Install Cicero

To experiment with Accord Project, you can install the Cicero command-line. This will let you author, validate, and run Accord Project templates on your own machine.

## Prerequisites

You must first obtain and configure the following dependency:

* [Node.js (LTS)](http://nodejs.org): We use Node.js to run cicero, generate the documentation, run a development web server, testing, and produce distributable files. Depending on your system, you can install Node either from source or as a pre-packaged bundle.

> We recommend using [nvm](https://github.com/creationix/nvm) (or [nvm-windows](https://github.com/coreybutler/nvm-windows)) to manage and install Node.js, which makes it easy to change the version of Node.js per project.

## Installing Cicero

To install the latest version of the Cicero command-line tools:

```bash
npm install -g @accordproject/cicero-cli
```

:::note
You can install a specific version by appending `@version` at the end of the `npm install` command. For instance to install version `0.20.3` or version `0.13.4`:
```bash
npm install -g @accordproject/cicero-cli@0.20.3
npm install -g @accordproject/cicero-cli@0.13.4
```
:::

To check that Cicero has been properly installed, and display the version number:
```bash
cicero --version
```

To get command line help:
```bash
cicero --help
cicero parse --help     // To parse a sample clause/contract
cicero draft --help     // To draft a sample clause/contract
cicero trigger --help   // To send a request to a clause/contract
```

## Optional Packages

### Template Generator

You may also want to install the template generator tool, which you can use to create an empty template:

```bash
npm install -g yo
npm install -g @accordproject/generator-cicero-template
```

## What next?

That's it! Go to the next page to see how to use your new installation of Cicero on a real Accord Project template.

---
id: started-resources
title: Resources

## Accord Project Resources

- The Main Web site includes latest news, links to working groups, organizational announcements, etc. : https://www.accordproject.org
- This Technical Documentation: https://docs.accordproject.org
- Recording of Working Group discussions, Tutorial Videos are available on Vimeo: https://vimeo.com/accordproject
- Join the [Accord Project Discord](https://discord.com/invite/Zm99SKhhtA) to get involved!

## User Content

Accord Project also maintains libraries containing open source, community-contributed content to help you when authoring your own templates:

- [Model Repository](https://models.accordproject.org/) : a repository of open source Concerto data models for use in templates
- [Template Library](https://templates.accordproject.org/) : a library of open source Clause and Contract templates for various legal domains (supply-chain, loans, intellectual property, etc.)

## Ecosystem & Tools

Accord Project is also developing tools to help with authoring, testing and running accord project templates.

### Editors

- [Template Studio](https://studio.accordproject.org/): a Web-based editor for Accord Project templates
- [VSCode Extension](https://marketplace.visualstudio.com/items?itemName=accordproject.cicero-vscode-extension): an Accord Project extension to the popular [Visual Studio Code](https://visualstudio.microsoft.com/) Editor
- [Emacs Mode](https://github.com/accordproject/ergo/tree/master/ergo.emacs): Emacs Major mode for Ergo (alpha)
- [VIM Plugin](https://github.com/accordproject/ergo/tree/master/ergo.vim): VIM plugin for Ergo (alpha)

### User Interface Components

- [Markdown Editor](https://github.com/accordproject/markdown-editor): a general purpose react component for markdown rendering and editing
- [Cicero UI](https://github.com/accordproject/cicero-ui): a library of react components for visualizing, creating and editing Accord Project templates

## Developers Resources

All the Accord Project technology is being developed as open source. The software packages are being actively maintained on [GitHub](https://github.com/accordproject) and we encourage organizations and individuals to contribute requirements, documentation, issues, new templates, and code.

Join us on the [#technology-wg Discord channel](https://discord.com/invite/Zm99SKhhtA) for technical discussions and weekly updates.

### Cicero

- GitHub: https://github.com/accordproject/cicero
- Technical Questions and Answers on [Stack Overflow](https://stackoverflow.com/questions/tagged/cicero)

### Ergo

- GitHub: https://github.com/accordproject/ergo
- The [Ergo Language Guide](logic-ergo.md) is a good place to get started with Ergo.

---
id: tutorial-create
title: Template Generator

Now that you have executed an existing template, let's create a new template from scratch. To facilitate the creation of new templates, Cicero comes with a template generator.

## The template generator

### Install the generator

If you haven't already done so, first install the template generator::

```bash
npm install -g yo
npm install -g yo @accordproject/generator-cicero-template
```

### Run the generator:

You can now try the template generator by running the following command in a terminal window:
```bash
yo @accordproject/cicero-template
```
This will ask you a series of questions. Give your generator a name (no spaces) and

~~~
then supply a namespace for your template model (again,no spaces). The generator
will then create the files and directories required for a basic template (similar
to the helloworld template).

Here is an example of how it should look like in your terminal window:
```bash
bash-3.2$ yo @accordproject/cicero-template

     _-----_     ╭──────────────────────────╮
    |       |    │      Welcome to the      │
    |--(o)--|    │ generator-cicero-templat │
   `---------´   │       e generator!       │
    ( _´U`_ )    ╰──────────────────────────╯
    /___A___\   /
     |  ~  |
   __'.___.'__
 ´   `  |° ´ Y `

? What is the name of your template? mylease
? Who is the author? me
? What is the namespace for your model? org.acme.lease
   create mylease/README.md
   create mylease/logo.png
   create mylease/package.json
   create mylease/request.json
   create mylease/logic/logic.ergo
   create mylease/model/model.cto
   create mylease/test/logic_default.feature
   create mylease/text/grammar.tem.md
   create mylease/text/sample.md
   create mylease/.cucumber.js
   create mylease/.npmignore
bash-3.2$
```

:::tip
You may find it easier to edit the grammar, model and logic for your template in
[VSCode](https://code.visualstudio.com/), installing the [Accord Project extension]
(https://marketplace.visualstudio.com/items?itemName=accordproject.cicero-vscode-
extension). The extension gives you syntax highlighting and parser errors within VS
Code.

For more information on how to use VS Code with the Accord Project extension,
please consult the [Using the VS Code extension](tutorial-vscode) tutorial.
:::

## Test your template

If you have Cicero installed on your machine, you can go into the newly created
`mylease` directory and try it with cicero, to make sure the contract text parses:
```bash
bash-3.2$ cicero parse
11:51:40 AM - info: Using current directory as template folder
11:51:40 AM - info: Loading a default text/sample.md file.
11:51:41 AM - info:
{
  "$class": "org.acme.lease.MyContract",
  "name": "Dan",
  "contractId": "4a7d5b59-0377-42d3-aa41-15062398d25d",
~~~

```
  "$identifier": "4a7d5b59-0377-42d3-aa41-15062398d25d"
}
```
And that you can trigger the contract:
```bash
bash-3.2$ cicero trigger
11:58:22 AM - info: Using current directory as template folder
11:58:22 AM - info: Loading a default text/sample.md file.
11:58:22 AM - info: Loading a default request.json file.
11:58:23 AM - warn: A state file was not provided, initializing state. Try the --
state flag or create a state.json in the root folder of your template.
11:58:23 AM - info:
{
  "clause": "mylease@0.0.0-
d186ab29c448b0058e4465a54d8376c3817dddb6fda8dc0ca29a88151b3dbecc",
  "request": {
    "$class": "org.acme.lease.MyRequest",
    "input": "World"
  },
  "response": {
    "$class": "org.acme.lease.MyResponse",
    "output": "Hello Dan World",
    "$timestamp": "2021-06-16T12:29:50.317-04:00"
  },
  "state": {
    "$class": "org.accordproject.runtime.State",
    "$identifier": "03515461-7ee7-4c81-a8f0-d4c667db5f4c"
  },
  "emit": []
}
```

The template also comes with a few simple tests which you can run by first doing an `npm install` in the template directory, then by running `npm run test`:
```bash
bash-3.2$ npm install
bash-3.2$ npm run test

> mylease@0.0.0 test /Users/jeromesimeon/tmp/mylease
> cucumber-js test -r .cucumber.js

....

1 scenario (1 passed)
3 steps (3 passed)
0m01.257s
bash-3.2$
```

## Edit your template

### Update Sample.md

First, replace the contents of `./text/sample.md` with the legal text for the contract or clause that you would like to digitize.

Check that when you run `cicero parse` that the new `./text/sample.md` is now _invalid_ with respect to the grammar.

### Edit the Template Grammar

Now update the grammar in `./text/grammar.tem.md`. Start by replacing the existing grammar, making it identical to the contents of your updated `./text/sample.md`.

Now introduce variables into your template grammar as required. The variables are marked-up using `{{` and `}}` with what is between the braces being the name of your variable.

### Edit the Template Model

All of the variables referenced in your template grammar must exist in your template model. Edit
the file `model/model.cto` to include all your variables, making sure the name of the model property matches the name of the variable in the `./text/grammar.tem.md` file.

Note that the Concerto Modeling Language primitive data types are:

- `String`: for character strings
- `Long` or `Integer`: for integer values
- `DateTime`: for dates and times
- `Double`: for floating points numbers
- `Boolean`: for values that are either true or false

:::tip
Note that you can import common types (address, monetary amount, country code, etc.) from the Accord Project Model Repository: https://models.accordproject.org.
:::

### Edit the Transaction Types

Your template expects to receive data as input and will produce data as output. The structure of
this request/response data is captured in the `MyRequest` and `MyResponse` transaction types in your model
namespace. Open up the file `models/model.cto` and edit the definition of the `MyRequest` type to
include all the data you expect to receive from the outside world and that will be used by the
business logic of your template. Similarly edit the definition of the `MyResponse` type to include
all the data that the business logic for your template will compute and would like to return to the
caller.

### Edit the Template Logic

Now edit the business logic of the template itself. This is expressed in the Ergo language, which is a strongly-typed function domain specific language for contract logic. Open the file `logic/logic.ergo`
and edit the `helloworld` clause to perform the calculations your logic requires.

Looking at the Ergo logic for other example templates will help you understand the syntax and capabilities of Ergo.

## Publishing your template

If you would like to publish your new template in the Accord Project Template

Library, please consult the [Template Library](tutorial-library) Section of this documentation.

---
id: tutorial-hyperledger
title: With Hyperledger Fabric

## Hyperledger Fabric 2.2

Sample chaincode for Hyperledger Fabric that shows how to execute a Cicero template:
https://github.com/accordproject/hlf-cicero-contract

Please refer to the project README for detailed instructions on installation and submitting transactions.

---
id: tutorial-library
title: Template Library

This tutorial explains how to get access, and contribute, to all of the public templates available as part of the the [Accord Project Template Library](https://templates.accordproject.org).

## Setting up

### Prerequisites

Accord Project uses [GitHub](https://github.com/) to maintain its open source template library. For this tutorial, you must first obtain and configure the following dependency:

* [Git](https://git-scm.com): a distributed version-control system for
  tracking changes in source code during software development.
* [Lerna](https://lerna.js.org/): A tool for managing JavaScript projects with multiple packages. You can install lerna by running the following command in your terminal:

```bash
npm install -g lerna
```

### Clone the template library

Once you have `git` installed on your machine, you can run `git clone` to create a version of all the templates:

```bash
git clone https://github.com/accordproject/cicero-template-library
```

Alternatively, you can download the library directly by visiting the [GitHub Repository for the Template Library](https://github.com/accordproject/cicero-

~~~
template-library) and use the "Download" button as shown on this snapshot:

![Basic-Library-1](/docs/assets/basic/library1.png)

### Install the Library

Once cloned, you can set up the library for development by running the following
commands inside your template library directory:

```bash
lerna bootstrap
```

### Running all the template tests

To check that the installation was successful, you can run all the tests for all
the Accord Project templates by running:

```bash
lerna run test
```

## Structure of the Repository

You can see the source code for all public Accord Project templates by looking
inside the `./src` directory:

```sh
bash-3.2$ ls src
acceptance-of-delivery
bill-of-lading
car-rental-tr
certificate-of-incorporation
company-information
contact-information
copyright-license
demandforecast
docusign-connect
docusign-po-failure
eat-apples
empty
empty-contract
fixed-interests
...
```

Each of those templates directories have the same structure, as described in the
[Templates Deep Dive](tutorial-templates) Section. For instance for the
`acceptance-of-delivery` template:
```
$ cd src/acceptance-of-delivery
$ bash-3.2$ ls -laR
./README.md
./package.json

./text:
  ./grammar.tem.md
  ./sample.md
~~~

```
./logic:
   logic.ergo

./model:
   model.cto

./test:
  logic.feature
  logic_default.feature

./request.json
./state.json
```

## Use a Template

To use a template, simply run the same Cicero commands we have seen in the previous tutorials. For instance, to extract the deal data from the `./text/sample.md` text sample for the `acceptance-of-delivery` template, run:

```bash
cicero parse --template ./src/acceptance-of-delivery
```
You should see a response as follows:
```json
{
  "$class": "org.accordproject.acceptanceofdelivery.AcceptanceOfDeliveryClause",
  "shipper": "resource:org.accordproject.organization.Organization#Party%20A",
  "receiver": "resource:org.accordproject.organization.Organization#Party%20B",
  "deliverable": "Widgets",
  "businessDays": 10,
  "attachment": "Attachment X",
  "clauseId": "f1b1434b-8500-4672-8678-7c5003d8d66b",
  "$identifier": "f1b1434b-8500-4672-8678-7c5003d8d66b"
}
```

Or, to extract the deal data from the `./text/sample.md` then send the default request in `./request.json` for the `latedeliveryandpenalty` template, run:
```bash
cicero trigger --template ./src/latedeliveryandpenalty
```
You should see a response as follows:

```json
{
  "clause": "latedeliveryandpenalty@0.17.0-
a4e00f4f161e2d343a239a6854bfce92ecd16d891f8e7bc5a5adaab46d242782",
  "request": {
    "$class":
"org.accordproject.latedeliveryandpenalty.LateDeliveryAndPenaltyRequest",
    "forceMajeure": false,
    "agreedDelivery": "2017-12-17T03:24:00-05:00",
    "deliveredAt": null,
    "goodsValue": 200
  },
  "response": {
    "$class":
"org.accordproject.latedeliveryandpenalty.LateDeliveryAndPenaltyResponse",
    "penalty": 110.00000000000001,
    "buyerMayTerminate": true,
    "$timestamp": "2021-06-16T12:26:18.031-04:00"
  },
  "state": {
    "$class": "org.accordproject.runtime.State",
    "$identifier": "54810499-acad-4a3a-9f78-684b0a3bef65"
  },
  "emit": [
    {
      "$class": "org.accordproject.obligation.PaymentObligation",
      "amount": {
        "$class": "org.accordproject.money.MonetaryAmount",
        "doubleValue": 110.00000000000001,
        "currencyCode": "USD"
      },
      "description": ""resource:org.accordproject.party.Party#Dan" should pay
penalty amount to "resource:org.accordproject.party.Party#Steve"",
      "$identifier": "a9482b16-c0dc-4e09-86bc-60bb59b07523",
      "contract":
"resource:org.accordproject.latedeliveryandpenalty.LateDeliveryAndPenaltyContract#3
fecad6b-442c-49d1-99d8-b963616f61d2",
      "promisor": "resource:org.accordproject.party.Party#Dan",
      "promisee": "resource:org.accordproject.party.Party#Steve",
      "$timestamp": "2021-06-16T12:26:18.032-04:00"
    }
  ]
}
```

## Contribute a New Template

To contribute a change to the Accord Project library, please
[fork](https://help.github.com/en/github/getting-started-with-github/fork-a-repo)
the repository and then create a [pull
request](https://help.github.com/en/github/collaborating-with-issues-and-pull-requests/about-pull-requests).

Note that templates should have unit tests. See any of the `./test` directories in the templates contained in the template library for an examples with unit tests, or consult the [Testing Reference](ref-testing) Section of this documentation.

---
id: tutorial-nodejs
title: With Node.js

## Cicero Node.js API

You can work with Accord Project templates directly in JavaScript using Node.js.

Documentation for the API can be found in [Cicero API](ref-cicero-api.html).

## Working with Templates

### Import the Template class

To import the Cicero classes for templates and clauses, we'll also import the Cicero engine and some helper utilities

```js
const fs = require("fs");
const path = require("path");
const { Template, Clause } = require("@accordproject/cicero-core");
const { Engine } = require("@accordproject/cicero-engine");
```

### Load a Template

To create a Template instance in memory call the `fromDirectory`, `fromArchive` or `fromUrl` methods:

```js
const template = await Template.fromDirectory(
  "./test/data/latedeliveryandpenalty"
);
```

These methods are asynchronous and return a `Promise`, so you should use `await` to wait for the promise to be resolved.

> Note that you'll need to wrap this `await` inside an `async` function or use a [top-level await inside a module](https://v8.dev/features/top-level-await)

### Instantiate a Template

Once a Template has been loaded, you can create a Clause based on the Template. You can either instantiate
the Clause using source DSL text (by calling `parse`), or you can set an instance of the template model
as JSON data (by calling `setData`):

```js
// load the DSL text for the template
const testLatePenaltyInput = fs.readFileSync(
  path.resolve(__dirname, "text/", "sample.md"),
  "utf8"
);

const clause = new Clause(template);
clause.parse(testLatePenaltyInput);

// get the JSON object created from the parse
const data = clause.getData();
```

## Executing a Template Instance

Once you have instantiated a clause or contract instance, you can execute it.

### Import the Engine class

To execute a Clause you first need to create an instance of the `Engine` class:

```js
const engine = new Engine();
```

### Send a request to the contract

You can then call `execute` on it, passing in the clause or contract instance, and the request:

```js
const request = {
  $class:
    "org.accordproject.latedeliveryandpenalty.LateDeliveryAndPenaltyRequest",
  forceMajeure: false,
  agreedDelivery: "2017-10-07T16:38:01.412Z",
  goodsValue: 200,
};
const state = {
  $class: "org.accordproject.runtime.State",
};

const result = await engine.trigger(clause, request, state);
console.log(result);
```

---
id: tutorial-studio
title: With Template Studio

This tutorial will walk you through the steps of editing a clause template in [Template Studio](https://studio.accordproject.org/).

We start with a very simple _Late Penalty and Delivery_ Clause and gradually make it more complex, adding both legal text to it and the corresponding business logic in Ergo.

## Initial Late Delivery Clause

### Load the Template

To get started, head to the `minilatedeliveryandpenalty` template in the Accord Project Template Library at [Mini Late Delivery And Penalty](https://templates.accordproject.org/minilatedeliveryandpenalty@0.6.0.html) and click the "Open In Template Studio" button.

![Advanced-Late-1](assets/advanced/late1.png)

Begin by inspecting the `README` and `package.json` tabs within the `Metadata` section. Feel free to change the name of the template to one you like.

### The Contract Text

Then click on the `Text` Section on the left, which should show a `Grammar` tab, for the the natural language of the template.

![Advanced-Late-2](assets/advanced/late2.png)

When the text in the `Grammar` tab is in sync with the text in the `Sample` tab, this means the sample is a valid with respect to the grammar, and data is extracted, showing in `Contract Data` tab. The contract data is represented using the JSON format and contains the value of the variables declared in the contract template. For instance, the value for the `buyer` variable is `Betty Buyer`, highlighted in red:

![Advanced-Late-3](assets/advanced/late3.png)

Changes to the variables in the `Sample` are reflected in the `Contract Data` tab in real time, and vice versa. For instance, change `Betty Buyer` to a different name in the contract text to see the `partyId` change in the contract data.

:::note
The JSON data `resource:org.accordproject.party.Party#Betty%20Buyer` indicate that the value is a relationship of type `Party` whose identifier is `Betty Buyer`. Consult the [Concerto Guide](model-relationships) for more details on modeling relationships.
:::

If you edit part of the text which is not a variable in the template, this results in an error when parsing the `Sample`. The error will be shown in red in the status bar at the bottom of the page. For instance, the following image shows the parsing error obtained when changing the word `delayed` to the word `timely` in the contract text.

![Advanced-Late-4](assets/advanced/late4.png)

This is because the `Sample` relies on the `Grammar` text as a source of truth. This mechanism ensures that the actual contract always reflects the template, and remains faithful to the original legal text. You can, however, edit the `Grammar` itself to change the legal text.

Revert your changes, changing the word `timely` back to the original word `delayed` and the parsing error will disappear.

### The Model

Moving along to the `Model` section, you will find the data model for the template variables (the `MiniLateDeliveryClause` type), as well as for the requests (the `LateRequest` type) and response (the `LateResponse` type) for the late delivery and penalty clause.

![Advanced-Late-5](assets/advanced/late5.png)

Note that a `namespace` is declared at the beginning of the file for the model, and that several existing models are being imported (using e.g., `import org.accordproject.contract.*`). Those imports are needed to access the definition for several types used in the model:
- `Clause` which is a generic type for all Accord Project clause templates, and is defined in the `org.accordproject.contract` namespace;
- `Party` which is a generic type for all Accord Project parties, and is defined in the `org.accordproject.party` namespace;
- `Request` and `Response` which are generic types for responses and requests, and are defined in the `org.accordproject.runtime` namespace;
- `Duration` which is defined in the `org.accordproject.time` namespace.

### The Logic

The final part of the template is the `Ergo` tab of the `Logic` section, which describes the business logic.

![Advanced-Late-6](assets/advanced/late6.png)

Thanks to the `namespace` at the beginning of this file, the Ergo engine can know the definition for the `MiniLateDeliveryClause`, as well as the `LateRequest`, and `LateResponse` types defined in the `Model` tab.

To test the template execution, go to the `Request` tab in the `Logic` section. It should be already populated with a valid request. Press the `Trigger` button to trigger the clause.

![Advanced-Late-7](assets/advanced/late7.png)

Since the value of the `deliveredAt` parameter in the request is after the value of the `agreedDelivery` parameter in the request, this should return a new response which includes the calculated penalty.

Changing the date for the `deliveredAt` parameter in the request and triggering the contract again will result in a different penalty.

![Advanced-Late-8](assets/advanced/late8.png)

Note that the clause will return an error if it is called for a timely delivery.

![Advanced-Late-9](assets/advanced/late9.png)

## Add a Penalty Cap

We can now start building a more advanced clause. Let us first take a moment to notice that there is no limitation to the penalty resulting from a late delivery. Trigger the contract using the following request in the `Request` tab in `Logic`:
```json
{
  "$class": "org.accordproject.minilatedeliveryandpenalty.LateRequest",
  "agreedDelivery": "2019-04-10T12:00:00-05:00",
  "deliveredAt": "2019-04-20T03:24:00-05:00",
  "goodsValue": 200
}
```
The penalty should be rather low. Now send this other request:
```json
{
  "$class": "org.accordproject.minilatedeliveryandpenalty.LateRequest",
  "agreedDelivery": "2005-04-01T12:00:00-05:00",
  "deliveredAt": "2019-04-20T03:24:00-05:00",
  "goodsValue": 200
}
```
Notice that the penalty is now quite a large value. It is not unusual to cap a penalty to a maximum amount. Let us now look at how to change the template to add such a cap based on a percentage of the total value of the delivered goods.

### Update the Legal Text

To implement this, we first go to the `Grammar` tab in the `Text` section and add a sentence indicating: `The total amount of penalty shall not, however, exceed {{capPercentage}}% of the total value of the delayed goods.`

For convenience, you can copy-paste the new template text from here:

```tem
Late Delivery and Penalty.

In case of delayed delivery of Goods, {{seller}} shall pay to
{{buyer}} a penalty amounting to {{penaltyPercentage}}% of the total
value of the Goods for every {{penaltyDuration}} of delay. The total
amount of penalty shall not, however, exceed {{capPercentage}}% of the
total value of the delayed goods. If the delay is more than
{{maximumDelay}}, the Buyer is entitled to terminate this Contract.

```
This should immediately result in an error when parsing the contract text:

![Advanced-Late-10](assets/advanced/late10.png)

As explained in the error message, this is because the new template text uses a
variable `capPercentage` which has not been declared in the model.

### Update the Model

To define this new variable, go to the `Model` tab, and change the
`MiniLateDeliveryClause` type to include `o Double capPercentage`.

![Advanced-Late-11](assets/advanced/late11.png)

For convenience, you can copy-paste the new `MiniLateDeliveryClause` type from
here:
```ergo
asset MiniLateDeliveryClause extends Clause {
  --> Party buyer         // Party to the contract (buyer)
  --> Party seller        // Party to the contract (seller)
  o Duration penaltyDuration  // Length of time resulting in penalty
  o Double penaltyPercentage  // Penalty percentage
  o Double capPercentage      // Maximum penalty percentage
  o Duration maximumDelay     // Maximum delay before termination
}
```

This results in a new error, this time on the sample contract:

![Advanced-Late-12](assets/advanced/late12.png)

To fix it, we need to add that same line we added to the template, replacing the
`capPercentage` by a value in the `Test Contract`: `The total amount of penalty
shall not, however, exceed 52% of the total value of the delayed goods.`

For convenience, you can copy-paste the new test contract from here:
```md
Late Delivery and Penalty.

In case of delayed delivery of Goods, "Steve Seller" shall pay to
"Betty Buyer" a penalty amounting to 10.5% of the total
value of the Goods for every 2 days of delay. The total
amount of penalty shall not, however, exceed 52% of the
total value of the delayed goods. If the delay is more than
15 days, the Buyer is entitled to terminate this Contract.

```

Great, now the edited template should have no more errors, and the contract data should now include the value for the new `capPercentage` variable.

![Advanced-Late-13](assets/advanced/late13.png)

Note that the `Current Template` Tab indicates that the template has been changed.

### Update the Logic

At this point, executing the logic will still result in large penalties. This is because the logic does not take advantage of the new `capPercentage` variable. Edit the `logic.ergo` code to do so. After step `// 2. Penalty formula` in the logic, apply the penalty cap by adding some logic as follows:
```ergo
    // 3. Capped Penalty
    let cap = contract.capPercentage / 100.0 * request.goodsValue;

    let cappedPenalty =
      if penalty > cap
      then cap
      else penalty;

```
Do not forget to also change the value of the penalty in the returned `LateResponse` to use the new variable `cappedPenalty`:
```ergo
    // 5. Return the response
    return LateResponse{
      penalty: cappedPenalty,
      buyerMayTerminate: termination
    }
```
The logic should now look as follows:

![Advanced-Late-14](assets/advanced/late14.png)

### Run the new Logic

As a final test of the new template, you should try again to run the contract with a long delay in delivery. This should now result in a much smaller penalty, which is capped to 52% of the total value of the goods, or 104 USD.

![Advanced-Late-15](assets/advanced/late15.png)

:::tip
A full version of the template after those changes have been applied can be found as the [Mini Late Delivery And Penalty Capped](https://templates.accordproject.org/minilatedeliveryandpenalty-capped@0.6.0.html) in the Template Library.
:::

## Emit a Payment Obligation.

As a final extension to this template, we can modify it to emit a Payment Obligation. This first requires us to switch from a Clause template to a Contract template.

### Switch to a Contract Template

The first place to change is in the metadata for the template. This can be done easily with the `full contract` button in the `Current Template` tab. This will immediately result in an error indicating that the model does not contain an `Contract` type.

![Advanced-Late-16](assets/advanced/late16.png)

### Update the Model

To fix this, change the model to reflect that we are now editing a contract template, and change the type `AccordClause` to `AccordContract` in the type definition for the template variables:
```ergo
asset MiniLateDeliveryContract extends Contract {
  --> Party buyer         // Party to the contract (buyer)
  --> Party seller        // Party to the contract (seller)
  o Duration penaltyDuration  // Length of time resulting in penalty
  o Double penaltyPercentage  // Penalty percentage
  o Double capPercentage      // Maximum penalty percentage
  o Duration maximumDelay     // Maximum delay before termination
}
```

The next error is in the logic, since it still uses the old `MiniLateDeliveryClause` type which does not exist anymore.

### Update the Logic

The `Logic` error that occurs here is:
```bash
Compilation error (at file lib/logic.ergo line 19 col 31). Cannot find type with name 'MiniLateDeliveryClause'
contract MiniLateDelivery over MiniLateDeliveryClause {
                               ^^^^^^^^^^^^^^^^^^^^^^
```
Update the logic to use the the new `MiniLateDeliveryContract` type instead, as follows:
```ergo
contract MiniLateDelivery over MiniLateDeliveryContract {
```

The template should now be without errors.

### Add a Payment Obligation

Our final task is to emit a `PaymentObligation` to indicate that the buyer should pay the seller in the amount of the calculated penalty.

To do so, first import a couple of standard models: for the Cicero's [runtime model](https://models.accordproject.org/cicero/runtime.html) (which contains the definition of a `PaymentObligation`), and for the Accord Project's [money model](https://models.accordproject.org/money.html) (which contains the definition of a `MonetaryAmount`). The `import` statements at the top of your logic should look as follows:
```ergo
import org.accordproject.time.*
import org.accordproject.cicero.runtime.*
import org.accordproject.money.MonetaryAmount
```

~~~
```

Lastly, add a new step between steps `// 4.` and `// 5.` in the logic to emit a
payment obligation in USD:
```ergo
    emit PaymentObligation{
      contract: contract,
      promisor: some(contract.seller),
      promisee: some(contract.buyer),
      deadline: none,
      amount: MonetaryAmount{ doubleValue: cappedPenalty, currencyCode: USD },
      description: contract.seller.partyId ++ " should pay penalty amount to " ++
contract.buyer.partyId
    };

```
That's it! You can observe in the `Request` tab that an `Obligation` is now being
emitted. Try out adjusting values and continuing to send requests and getting
responses and obligations.

![Advanced-Late-17](assets/advanced/late17.png)

:::tip
A full version of the template after those changes have been applied can be found
as the [Mini-Late Delivery and Penalty
Payment](https://templates.accordproject.org/minilatedeliveryandpenalty-
payment@0.6.0.html) in the Template Library.
:::

--------------------------------------------------------------------------------
---
id: tutorial-templates
title: Templates Deep Dive
---

In the [Getting Started](started-hello) section, we learned how to use the existing
[helloworld@0.14.0.cta](https://templates.accordproject.org/archives/
helloworld@0.14.0.cta) template archive. Here we take a look inside that archive to
understand the structure of Accord Project templates.

## Unpack a Template Archive

A `.cta` archive is nothing more than a zip file containing the components of a
template. Let's unzip that archive to see what is inside. First, create a directory
in the place where you have downloaded that archive, then run the unzip command in
a terminal:

```bash
$ mkdir helloworld
$ mv helloworld@0.14.0.cta helloworld
$ cd helloworld
$ unzip helloworld@0.14.0.cta
Archive:  helloworld@0.14.0.cta
 extracting: package.json
   creating: text/
 extracting: text/grammar.tem.md
 extracting: README.md
 extracting: text/sample.md
 extracting: request.json
~~~

```
   creating: model/
 extracting: model/@models.accordproject.org.time@0.2.0.cto
 extracting: model/@models.accordproject.org.accordproject.money@0.2.0.cto
 extracting: model/@models.accordproject.org.accordproject.contract.cto
 extracting: model/@models.accordproject.org.accordproject.runtime.cto
 extracting: model/@org.accordproject.ergo.options.cto
 extracting: model/model.cto
   creating: logic/
 extracting: logic/logic.ergo
```

## Template Components

Once you have unziped the archive, the directory should contain the following files
and sub-directories:

```text
package.json
    Metadata for the template (name, version, description etc)

README.md
    A markdown file that describes the purpose and correct usage for the template

text/grammar.tem.md
    The default grammar for the template

text/sample.md
    A sample clause or contract text that is valid for the template

model/
    A collection of Concerto model files for the template. They define the Template
Model
    and models for the State, Request, Response, and Obligations used during
execution.

logic/
    A collection of Ergo files that implement the business logic for the template

test/
    A collection of unit tests for the template

state.json (optional)
    A sample valid state for the clause or contract

request.json (optional)
    A sample valid request to trigger execution for the template
```

In a nutshell, the template archive contains the three main components of the
[Template Triangle](accordproject-concepts#what-is-a-template) in the corresponding
directories (the natural language text of your clause or contract in the `text`
directory, the data model in the `model` directory, and the contract logic in the
`logic` directory). Additional files include metadata and samples which can be used
to illustrate or test the template.

Let us look at each of those components.

### Template Text

#### Grammar

The file in `text/grammar.tem.md` contains the grammar for the template. It is natural language, with markup to indicate the variable(s) in your Clause or Contract.

```tem
Name of the person to greet: {{name}}.
Thank you!
```

In the `helloworld` template there is only one variable `name` which is indicated between `{{` and `}}`.

#### Sample Text

The file in `text/sample.md` contains a sample valid for that grammar.

```md
Name of the person to greet: "Fred Blogs".
Thank you!
```

### Template Model

The file in `model/model.cto` contains the data model for the template. This includes a description for each of the template variables, including what kind of variable it is (also called their [type](ref-glossary.html#components-of-data-models)).

Here is the model for the `helloworld` template:

```ergo
namespace org.accordproject.helloworld

import org.accordproject.contract.* from
https://models.accordproject.org/accordproject/contract.cto
import org.accordproject.runtime.* from
https://models.accordproject.org/accordproject/runtime.cto

transaction MyRequest extends Request {
  o String input
}

transaction MyResponse extends Response {
  o String output
}

/**
 * The template model
 */
asset HelloWorldClause extends Clause {
  /**
   * The name for the clause
   */
  o String name
}
```

The `HelloWorldClause` as well as the `Request` and `Response` are types which are specified using the [Concerto modeling language](https://github.com/accordproject/concerto).

The `HelloWorldClause` indicate that the template is for a Clause, and should have a variable `name` of type `String` (i.e., text).

```ergo
asset HelloWorldClause extends Clause {
  o String name // variable 'name' is of type String
}
```

Types are always declared within a namespace (here `org.accordproject.helloworld`), which provides a mechanism to disambiguate those types amongst multiple model files.

### Template Logic

The file in `logic/logic.ergo` contains the executable logic. Each Ergo file is identified by a namespace, and contains declarations (e.g., constants, functions, contracts). Here is the Ergo logic for the `helloworld` template:

```ergo
namespace org.accordproject.helloworld

contract HelloWorld over TemplateModel {
  // Simple Clause
  clause greet(request : MyRequest) : MyResponse {
    return MyResponse{ output: "Hello " ++ contract.name ++ " " ++ request.input }
  }
}
```

This declares a single `HelloWorld` contract in the `org.accordproject.helloworld` namespace, with one `greet` clause.

It also declares that this contract `HelloWorld` is parameterized over the given `TemplateModel` found in the `models/model.cto` file.

The `greet` clause takes a request of type `MyRequest` as input and returns a response of type `MyResponse`.

The code for the `greet` clause returns a new `MyResponse` response with a single property `output` which is a string. That string is constructed using the string concatenation operator (`++`) in Ergo from the `name` in the contract (`contract.name`) and the input from the request (`request.input`).

## Use the Template

Even after you have unzipped the template archive, you can use that template from the directory directly, in the same way we did from the `.cta` archive in the [Getting Started](started-hello) section.

For instance you can use `cicero parse` or `cicero trigger` as follows:
```bash
$ cd helloworld
$ cicero parse
12:21:37 PM - INFO: Using current directory as template folder
```

~~~
12:21:37 PM - INFO: Loading a default text/sample.md file.
12:21:38 PM - INFO:
{
  "$class": "org.accordproject.helloworld.HelloWorldClause",
  "name": "Fred Blogs",
  "clauseId": "ca447073-242f-4721-a5b9-c5c14b57233d",
  "$identifier": "ca447073-242f-4721-a5b9-c5c14b57233d"
}
$ cicero trigger
12:21:54 PM - INFO: Using current directory as template folder
12:21:54 PM - INFO: Loading a default text/sample.md file.
12:21:54 PM - INFO: Loading a default request.json file.
12:21:55 PM - WARN: A state file was not provided, initializing state. Try the --
state flag or create a state.json in the root folder of your template.
12:21:55 PM - INFO:
{
  "clause": "helloworld@0.14.0-
4f8006ff0471176f2b5340500ba40c42adb180f26df50b747d8690c6dad79cfa",
  "request": {
    "$class": "org.accordproject.helloworld.MyRequest",
    "input": "Accord Project"
  },
  "response": {
    "$class": "org.accordproject.helloworld.MyResponse",
    "output": "Hello Fred Blogs Accord Project",
    "$timestamp": "2021-06-16T12:21:55.749-04:00"
  },
  "state": {
    "$class": "org.accordproject.runtime.State",
    "$identifier": "3fa15a55-d5db-491c-905a-7fcf5eb64d5f"
  },
  "emit": []
}
```
~~~

:::note
Remark that if your template directory contains a valid `sample.md` or valid `request.json`, Cicero will automatically detect those so you do not need to pass them using the `--sample` or `--request` options.
:::

---
id: tutorial-vscode
title: With VS Code

Cicero comes with a VS Code extension for easier development and testing. It includes support for syntax highlighting, allows you to test your template (contract parsing and logic) and to create template archives directly within VS Code.

## Prerequisites

To follow this tutorial on how to use the Cicero VS Code extension, we recommend you install the following:

1. [Node, LTS version](nodejs.org)
1. [VS Code](https://code.visualstudio.com)

1. [Yeoman](https://yeoman.io)
1. [Accord Project Yeoman Generator](https://github.com/accordproject/cicero/tree/master/packages/generator-cicero-template)
1. [Camel Tooling Yeoman VS Code extension](https://marketplace.visualstudio.com/items?itemName=camel-tooling.yo)
1. [Accord Project VS Code extension](https://marketplace.visualstudio.com/items?itemName=accordproject.cicero-vscode-extension)

## Video Tutorial

<iframe title="vimeo-player" src="https://player.vimeo.com/video/444483242" width="640" height="400" frameborder="0" allowfullscreen></iframe>

---
title: Welcome!
author: Dan Selman
authorURL: http://twitter.com/danielselman

This is a test.

<!--truncate-->

Mauris vestibulum ullamcorper nibh, ut semper purus pulvinar ut. Donec volutpat orci sit amet mauris malesuada, non pulvinar augue aliquam. Vestibulum ultricies at urna ut suscipit. Morbi iaculis, erat at imperdiet semper, ipsum nulla sodales erat, eget tincidunt justo dui quis justo. Pellentesque dictum bibendum diam at aliquet. Sed pulvinar, dolor quis finibus ornare, eros odio facilisis erat, eu rhoncus nunc dui sed ex. Nunc gravida dui massa, sed ornare arcu tincidunt sit amet. Maecenas efficitur sapien neque, a laoreet libero feugiat ut.

---
id: version-0.12-accordproject-installation
title: Installation
original_id: accordproject-installation

To start working on your own Accord Project templates, you should install the Cicero command-line tools. This will let you author, parse, and execute Accord Project templates.

## Prerequisites

Before you can install Cicero, you must first obtain and configure the following dependency:

* [Node.js v8.x (LTS)](http://nodejs.org): We use Node to generate the documentation, run a
  development web server, run tests, and generate distributable files. Depending on your system,
  you can install Node either from source or as a pre-packaged bundle.

> We recommend using [nvm](https://github.com/creationix/nvm) (or [nvm-windows](https://github.com/coreybutler/nvm-windows)) to manage and install Node.js, which makes it easy to change the version of Node.js per project.

## Installing Cicero

To install the latest version:

```bash
npm install -g @accordproject/cicero-cli@0.12
```

To check that Cicero has been properly installed, and display the version number:
```bash
cicero --version
```

To get command line help:
```bash
cicero --help
cicero parse --help
cicero execute --help
```

## Optional Packages

### Template Generator

You may also want to install the template generator tool, which you can use to
create an empty template:

```bash
npm install -g yo
npm install -g @accordproject/generator-cicero-template@0.12
```

### Ergo command line

If you would like to use the Ergo language on its own (i.e., independently of a
Cicero template) you can also install the Ergo command-line tools:

```bash
npm install @accordproject/ergo-cli@0.12 -g
```

To check that the Ergo compiler has been installed, display the version number:
```bash
ergoc --version
```

To get command line help:
```bash
ergoc --help
ergorun --help
```

That's it!

## What next?

The following pages provide links to developers tools and resources. You can also
browse using the navigation bar to the left to find additional information:
tutorials, API reference, the Ergo language guide, etc.

```
--------------------------------------------------------------------------------
---
id: version-0.12-accordproject-models
title: Standard Models
original_id: accordproject-models
---

Accord Project maintains an Open Source [Models
Repository](https://models.accordproject.org) containing standard reusable data
models that you can import and use within your Smart Legal Contract Cicero
templates and in your Ergo logic.

![Model Repository](/docs/assets/bond-model.png)

--------------------------------------------------------------------------------
---
id: version-0.12-accordproject-resources
title: Learning Resources
original_id: accordproject-resources
---

## Accord Project Resources

- The Main Web site includes latest news, links to working groups, organizational
announcements, etc. : https://www.accordproject.org
- This Technical Documentation: https://docs.accordproject.org
- Recording of Working Group discussions, Tutorial Videos are on available on
Vimeo: https://vimeo.com/accordproject
- Accord Project's [Slack](https://accord-project.slack.com/)

## Cicero Resources

- GitHub: https://github.com/accordproject/cicero
- Slack [Channel](https://accord-project.slack.com/messages/CA08NAHQS/details/)
- Technical Questions and Answers on [Stack
Overflow](https://stackoverflow.com/questions/tagged/cicero)

## Ergo Resources

- GitHub: https://github.com/accordproject/ergo
- The [Ergo Language Guide](logic-ergo) is a good place to get started with Ergo.
- Slack [Channel](https://accord-project.slack.com/messages/C9HLJHREG/details/)


--------------------------------------------------------------------------------
---
id: version-0.12-accordproject-studio
title: Template Studio
original_id: accordproject-studio
---

[Template Studio](https://studio.accordproject.org) lets you create, edit and test
legal clause or contract templates built with the Accord Project.

![Model Repository](/img/studio.png)

--------------------------------------------------------------------------------
```

---
id: version-0.12-accordproject-templates
title: Open Source Templates
original_id: accordproject-templates

Accord Project maintains an Open Source [Templates
Library](https://templates.accordproject.org) of legal clauses and contracts that
conform to the Accord Project template specification.

![Template Library](/docs/assets/acceptance-of-delivery.png)

---
id: version-0.12-accordproject-tooling
title: Developers Tools
original_id: accordproject-tooling

Several tools are available in order to help you develop your own smart legal
templates.

## Visual Studio Code Extension

An extension is available for the popular open-source code editor [Visual Studio
Code](https://code.visualstudio.com/).
This provides syntax highlighting, and error reporting when working on source Ergo
logic and on Cicero templates. Syntax highlighting for Composer Concerto models is
available in a [separate plugin](https://marketplace.visualstudio.com/items?
itemName=HyperledgerComposer.composer-support-client).

Install the Accord Project extension by visiting the [Visual Studio marketplace]
(https://marketplace.visualstudio.com/items?itemName=accordproject.accordproject-
vscode-plugin).

![VSCode plugin](/img/ergo-vscode.png)

## Syntax highlighting

Languages modes, which provide syntax highlighting for Ergo, also exist for a
couple of other editors.

### Emacs

A simple Emacs mode for Ergo can be found in the
[ergo-mode](https://github.com/accordproject/ergo-mode) project on GitHub.

### VIM

A simple VIM mode for Ergo can be found in the
[ergo.vim](https://github.com/accordproject/ergo/tree/master/ergo.vim) directory in
the Ergo source code on GitHub.

> Those are not maintained as actively as the rest of the Accord Project. If you
know emacs lisp or are a VIM user and would like to contribute, please contact us
on the Accord Project Slack or directly through GitHub!

## Template Studio

Finally, [Template Studio](https://studio.accordproject.org) lets you create, edit
and test legal clause or contract templates built with the Accord Project directly
from your browser, without having to install anything.

![Model Repository](/img/studio.png)

---
id: version-0.12-accordproject
title: Getting Started with Accord Project
original_id: accordproject

## What is Accord Project?

Accord Project is an open source, non-profit, initiative working to transform
contract management and contract automation by digitizing contracts.

## What is an Accord Project Template?

An Accord Project template is composed of three elements:

- Natural Language, the grammar for the legal text of the template
- Model, the data model that backs the template
- Logic, the executable business logic for the template

![Cicero Template](assets/template.png)

When combined these three elements allow templates to be edited, analyzed, queried
and executed.

## Technology

The Accord Project provides a complete solution for smart legal contract
development. The main project for the Accord Project technology is called [Cicero]
(https://github.com/accordproject/cicero).

### Cicero

Cicero implements a format for legal contract and clause templates based on the
[Accord Project Template Specification](accordproject-specification).

The following screenshot shows a Cicero template for an acceptance of delivery
clause.

![Template Grammar](/docs/assets/grammar.png)

Cicero relies on two other projects:
- [Concerto](https://github.com/hyperledger/composer-concerto): a lightweight,
versatile data modeling language (maintained by the Hyperledger forum)
- [Ergo](https://github.com/accordproject/ergo): a domain specific language to
express the executable logic of legal templates

### Concerto

Concerto is a lightweight modeling language which is used to describe the
information used in Accord Project templates.

The following screenshot shows the model for the acceptance of delivery clause.

![Concerto Model](/img/model-vscode.png)

The Concerto Modeling Language (CML) provides object-oriented style modeling and includes support for inheritance, for describing relationships, nested or optional data structures, enumerations and more.

### Ergo

Ergo is a domain-specific language (DSL) developed by the Accord Project for capturing legal contract logic.

A DSL is a computer language that is targeted to a particular kind of problem, rather than a general-purpose language that is aimed at any kind of software problem. For example, HTML is a DSL targeted at developing web pages. Similarly, Ergo is a DSL meant to capture the execution logic of legal contracts.

The following screenshot shows the Ergo logic for the acceptance of delivery clause.

![Ergo Logic](/img/ergo-vscode.png)

It is important that a developer and a lawyer can together agree that clauses in a computable legal contract have the same semantics as the equivalent computer code. For that reason, Ergo is intended to be accessible to Lawyers who create the corresponding prose for those computable legal contracts. As a programming language, the Ergo syntax also adheres to programming conventions.

## Ecosystem

Beyond a core technology for executable legal templates, Accord Project is building a rich ecosystem which includes community-contributed content based on that technology:

- [Model Repository](https://models.accordproject.org/) : a repository of open source Concerto data models for use in templates
- [Template Library](https://templates.accordproject.org/) : a library of open source Clause and Contract templates for various legal domains (supply-chain, loans, intellectual property, etc.)

Several tools are also available to facilitate authoring of Accord Project templates:

- [Template Studio](https://studio.accordproject.org/): a Web-based editor for Accord Project templates
- [VSCode Plugin](https://marketplace.visualstudio.com/items?itemName=accordproject.accordproject-vscode-plugin): an Accord Project extension to the popular [Visual Studio Code](https://visualstudio.microsoft.com/)

## Open Source Community

The Accord Project technology is being developed as open source. All the software packages are being actively maintained on [GitHub](https://github.com/accordproject) and we encourage organizations and individuals to contribute requirements, documentation, issues, new templates, and code.

Join the Accord Project Technology Working Group <a
href="https://docs.google.com/forms/d/e/1FAIpQLScmPLO6vflTKFTRTJXiopCjGEvS5mMeH-

ZlBnuStiQ3U4k19A/viewform">Slack channel</a> to get involved!

## Try Accord Project Online

The simplest way to get an introduction to the Accord Project technology is through the online [Template Studio](https://studio.accordproject.org) editor (you can open template studio from anywhere in this documentation by clicking the [Try Online!](https://studio.accordproject.org) button located in the top-right of the page).

The following video offers a tour of Template Studio and an introduction to the key concepts behind the Accord Project technology.

<iframe src="https://player.vimeo.com/video/328933628" width="640" height="400" frameborder="0" allow="autoplay; fullscreen" allowfullscreen></iframe>

## Local Installation

If you want to experience the full power of Accord Project, you should install the Cicero command-line tools on your own machine.

---
id: version-0.12-advanced-hyperledger
title: Deploying on Hyperledger Fabric
original_id: advanced-hyperledger

## Hyperledger Fabric 1.3

Sample chaincode for Hyperledger Fabric that shows how to execute a Cicero template:
https://github.com/clauseHQ/fabric-samples/tree/master/chaincode/cicero

This sample shows how you can deploy and execute Smart Legal Contracts on-chain using Hyperledger Fabric v1.3.

Using this guide you can deploy a [Smart Legal Contract Template](https://templates.accordproject.org/) from the [Open Source Accord Project](https://www.accordproject.org/) to your HLF v1.3 blockchain. You can then submit transactions to the Smart Legal Contract, with the contract state persisted to the blockchain, and return values and emitted events propagated back to the client.

Before starting this tutorial, you are encouraged to review an [Introduction to the Accord Project](https://docs.google.com/presentation/d/1lYT-XlY-4UDtYR7Oc0X62nvT7AzOqYZ4QA8YVZasgy8/edit).

These instructions have been tested with:
- MacOS 10.14 / Ubuntu 18.04
- Recent release of Chrome and Safari
- Docker 18.09
- Docker Compose 1.23
- Node 8.10
- Git 2.17

If you're building a new environment yourself, we recommend using a cloud-hosted server (to save the conference WIFI!) as the installations can require large downloads.

A small number of hosted virtual machines are available from the workshop facilitators if you have trouble installing the prerequisites yourself.

## Installing Prerequisites

### Using Amazon Web Services

If you have your own AWS account, you can use the customised Ubuntu image. From your EC2 Dashboard, create a new instance and search for **Cicero** in the Community AMIs. The AMI is available in the Frankfurt and N. Virginia regions.

![Amazon Image](assets/advanced/hlf1.png)

The `t2.medium` instance type is sufficient for this tutorial.

You'll need also need to add an inbound rule to your Security Group to allow connections on port `3389`. This will allow you to make a remote desktop connection to your server.

![Amazon Image Port](assets/advanced/hlf2.png)

The default username and password for the prebuilt image is:
- Username: `guest`
- Password: `hyperledger2018`

> Connect to your image with a Remote Desktop Client, for example, Microsoft Remote Desktop on Mac and Windows.

### Building a Custom Docker Image

If you're feeling a bit more adventurous, you can build your own system. If you don't have the tools locally on your machine, start with an [Ubuntu Bionic 18.04 image from your favourite cloud provider](https://www.ubuntu.com/download/cloud).
You will need to be able to transfer files into your image, so if your server doesn't have a Desktop Manager and browser installed you'll need to find some other way to transfer files, e.g. via SCP or FTP, for example.
Once you've provisioned your server, install all of the required tools using the corresponding installation instructions:
- [Docker](https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-on-ubuntu-18-04)
- [Node](https://www.digitalocean.com/community/tutorials/how-to-install-node-js-on-ubuntu-18-04)
- [Docker Compose](https://www.digitalocean.com/community/tutorials/how-to-install-docker-compose-on-ubuntu-18-04)
- [Git](https://www.digitalocean.com/community/tutorials/how-to-install-git-on-ubuntu-18-04)
- [Fabric](https://hyperledger-fabric.readthedocs.io/en/release-1.3/getting_started.html)

## Create your Smart Legal Contract with Template Studio

Cicero Templates are the magic glue that binds your clever legal words with the logic that will run on your network. In this first step, we'll create a template in [Template Studio](https://studio.accordproject.org/).

Template Studio is a browser-based development environment for Cicero Templates. Your templates are only every stored in your browser (and are not shared with the Accord Project), so you should **Export** your work to save it for another time.

This tutorial uses the `supplyagreement-perishable-goods` template. This Smart Legal Agreement combines a plaintext contract for the shipment of goods that impose conditions on temperature and humidity until the shipment is delivered.

We simulate the submission of IoT events from sensors that get sent to the contract in a supply blockchain network. The contract determines the obligations and actions of the parties according to its logic and legal text.

Once you've connected to your system, open https://studio.accordproject.org in a new browser window.

:::note
In the hosted images, Mozilla Firefox is preinstalled, click the icon on the top-left toolbar to launch it.
:::

The Template Studio allows you to load sample templates for smart legal agreements from the [Accord Project template library](https://templates.accordproject.org/).

> In the Template Studio search bar, type **supplyagreement-perishable-goods**. Select the 0.12.1 version.

Explore the source components of the template.

- Contract Text & Grammar
- Model
- Logic
- Test Execution

Note the placeholders in the Template Grammar (found under **Contract Text** -> **Template**), and the corresponding data model definition in `contract.cto` (found under **Model**).

The logic definition in Ergo defines the behaviour of the contract in response to Requests. The logic also reference the same contract variables, through the `contract` keyword.

You can simulate the performance of the contract using the **Text Execution** page (click **Logic** first). Clicking **Send Request** will trigger the smart clause and produce a response that indicates the total price due, along with any penalties.

> Change some of the values in the Test Contract, for example increase the lower limit for temperature readings from 2°C to 3°C. If you reset the Test Execution and send the same request again you should notice a penalty in the response.

![Change Test Contract](assets/advanced/hlf3.png)

The _Obligations_ that are emitted by the contract are configured to be emitted as Events in Fabric. This allows any party that is involved in the contract to perform an action automatically, for example to add a payment obligation to an invoice.

> Click **Export** to download your template

Save your CTA (Cicero Template Archive) file somewhere safe, as you'll need to use it in a later step. We suggest saving the file in user's the home folder.

> Create a `request.json` file with the contents of the **Request** box from the

Logic -> **Test Execution** page in Template Studio.

For example:
```json
{
    "$class": "org.accordproject.perishablegoods.ShipmentReceived",
    "unitCount": 3002,
    "shipment": {
      "$class": "org.accordproject.perishablegoods.Shipment",
      "shipmentId": "SHIP_001",
      "sensorReadings": [
        {
          "$class": "org.accordproject.perishablegoods.SensorReading",
          "centigrade": 2,
          "humidity": 80,
          "shipment":
"resource:org.accordproject.perishablegoods.Shipment#SHIP_001",
          "transactionId": "a"
        }
      ]
    }
  }
```

Finally, create a `contract.txt` file with the contents of the Test Contract box from the Contract Text page in Template Studio.

:::note
It's important that your sample contract text exactly matches your grammar's structure, this includes trailing spaces and line breaks. To be sure that you copy everything, right click the window and choose Select All, before choosing Copy.

You'll be notified if there are errors in your contract text during the next step by messages such as:
```md
Unexpected "\n"
```
:::

## Provision your Hyperledger Fabric instance

In this step, we will provision a test Hyperledger Fabric network on your machine.

:::note
If you're not using one of the prebuilt images you'll also need run the following command to download the tutorial resources from GitHub.

```sh
git clone https://github.com/clauseHQ/fabric-samples
cd fabric-samples
git checkout master
```
:::

> Open a Terminal window and type the following commands. In the hosted image there is a link start a terminal window on the desktop.
```sh
cd fabric-samples/cicero
```

Next we'll download Fabric and start the docker containers. If you're doing this for the first time, go and get a coffee. This will usually take several minutes.

> In your Terminal, type the following command to download and start Fabric.
```sh
./startFabric.sh
```

> Next install the node dependencies for the client code, with this command. This step can also take a few minutes.
```sh
npm install
```

> You can then enroll the administrator into the network, and register a user that we'll use in later scripts.
```sh
node enrollAdmin.js && node registerUser.js
```

## Deploy your contract to your network

> Using the files that you downloaded earlier, run the `deploy.js` script.

The example below assumes that all of the files are located in the same folder as `deploy.js`.

:::note
The order of the parameters to the `deploy.js` script is important, please follow the pattern shown in the example. You'll also need to give the relative or absolute path to the `sample.txt` file, for example if you saved the file in your home directory, replace `sample.txt` with `../../sample.txt`.
:::

```sh
node deploy.js supplyagreement-perishable-goods.cta sample.txt
```

If deployment of your contract is successful, you should see the following output:
```sh
Transaction proposal was good
Response payload: Successfully deployed contract MYCONTRACT based on
supplyagreement-perishable-goods@0.9.0
Successfully sent Proposal and received ProposalResponse: Status - 200, message -
""
The transaction has been committed on peer localhost:7051
Send transaction promise and event listener promise have completed
Successfully sent transaction to the orderer.
Successfully committed the change to the ledger by the peer
```

Finally, you can trigger your Smart Legal Agreement by sending requests to it. The `submitRequest.js` script is configured to route your request to your deployed contract.

> Run the `submitRequest.js` script using your `request.json` file, by typing the following command into the terminal.
```sh
```

node submitRequest.js request.json
~~~
If the invocation is successful, you should see the following output:
```sh
Assigning transaction_id:
0788d105901c9f12316bb84dc1c5345be6fe96edf626d427de9871cefac4f063
Transaction proposal was good
Response payload:
{"$class":"org.accordproject.perishablegoods.PriceCalculation","totalPrice":
{"$class":"org.accordproject.money.MonetaryAmount","doubleValue":4503,"currencyCode
":"USD"},"penalty":
{"$class":"org.accordproject.money.MonetaryAmount","doubleValue":0,"currencyCode":"
USD"},"late":false,"shipment":"resource:org.accordproject.perishablegoods.Shipment#
SHIP_001","transactionId":"3a30f4b7-7537-4c2e-8186-49ce7e95681d","timestamp":"2018-
12-01T17:49:26.100Z"}
Successfully sent Proposal and received ProposalResponse: Status - 200, message -
""
The transaction has been committed on peer localhost:7051
Send transaction promise and event listener promise have completed
Successfully sent transaction to the orderer.
Successfully committed the change to the ledger by the peer
~~~

Congratulations you now have a legal agreement running on your blockchain network!

Depending on the values in your request, the response payload will indicate whether
a penalty is due.

Because this contract is currently stateless, i.e. it doesn't store any data on-
chain, you can submit multiple requests with different values to simulate the
behaviour of the contract under different circumstances.

:::tip
Can you cause a breach due to out-of-range readings?
:::

## Extension Tasks

We gave you lots of help in the first few steps, but to learn properly, we always
find that it helps to try things for yourself. Here are a few suggestions that will
help you to understand what is going on better.

1. Currently, `the supplyagreement-perishable-goods` template doesn't store any
data on the chain. Modify the template to store sensor readings. The readings
should be looked up from the state object in your logic rather than from your
request when the shipment is accepted.
:::note
Take a look at some of the other stateful Cicero Templates to see what changes you
will need to make. `installment-sale` and `helloworldstate` are good examples.
If you get really stuck, a solution is available for you to
[download](https://drive.google.com/file/d/1cak_P_x01w8dz43aZX8N16G5DUNp6VbG/view?
usp=sharing).
:::
2. Explore the source code of the Cicero chaincode shim that transforms your
requests and deployments into Fabric transactions using the Fabric Node SDK.
https://github.com/clauseHQ/fabric-samples/tree/master/chaincode/cicero
3. Create your own template from scratch using [Template
Studio](https://studio.accordproject.org/), or download the [VSCode plugin]
(https://marketplace.visualstudio.com/items?itemName=accordproject.accordproject-

vscode-plugin).
![Change Test Contract](assets/advanced/hlf4.png)
> A separate tutorial for creating a template using the Cicero CLI tool can be found in [Creating a New Template](basic-create).

4. This template also emits Obligations as Fabric events as well as returning a response to the client. Modify the Cicero chaincode to display the events do something interesting with them. How would you notify the parties that a penalty is due?

:::warning
If you would like to make changes to the cicero chaincode be aware that Docker caches the docker image for the chaincode. If you edit the source and run `./startFabric` you will _not_ see your changes.
For your code changes to take effect you need to `docker stop` the peer (use `docker ps` to get the container id) and then `docker rmi -f` your docker chaincode image. The image name should look something like `dev-peer0.org1.example.com-cicero-2.0-598263b3afa0267a29243ec2ab8d19b7d2016ac628f13641ed1822c4241c5734`
:::
5. Deploy the contract to a Fabric network that includes multiple users and nodes.

## Hyperledger Composer

A separate sample showing how to integrate Cicero with Hyperledger Composer is available here:
https://github.com/accordproject/cicero-perishable-network

---
id: version-0.12-advanced-latedelivery
title: Authoring in Template Studio
original_id: advanced-latedelivery

This tutorial will walk you through the steps of authoring a clause template in [Template Studio](https://studio.accordproject.org/).

We start with a very simple _Late Penalty and Delivery_ Clause and gradually make it more complex, adding both legal text to it and the corresponding business logic in Ergo.

## Initial Late Delivery Clause

### Load the Template

To get started, head to the `minilatedeliveryandpenalty` template in the Accord Project Template Library at
https://templates.accordproject.org/minilatedeliveryandpenalty@0.2.1.html and click the "Open In Template Studio" button.

![Advanced-Late-1](assets/advanced/late1.png)

Begin by inspecting the `README` and `package.json` tabs within the `Metadata` section. Feel free to change the name of the template to one you like.

### The Contract Text

Then head to the `Template` tab in the `Contract Text` section which shows the natural language for the template. You should see the following text.

![Advanced-Late-2](assets/advanced/late2.png)

When the text in the `Template` tab is in sync with the text in the `Test Contract` tab, this results in a valid contract instance in the `Contract Data` tab. The contract data is represented using the JSON format, and contains the value of the variables declared in the contract template. For instance, the value for the `buyer` variable is `Betty Buyer`, highlighted in red:

![Advanced-Late-3](assets/advanced/late3.png)

Changes to the variables in the `Test Contract` are reflected in the Contract Data in real time, and vice versa. For instance, change `Betty Buyer` to a different name in the contract text to see the `partyId` change in the contract data.

If you edit part of the text which is not a variable in the template, this results in an error when parsing the `Contract Text`. The error will be shown in red in the status bar at the bottom of the page. For instance the following image shows the parsing error obtained when changing the word `delayed` to the word `timely` in the contract text.

![Advanced-Late-4](assets/advanced/late4.png)

This is because the `Test Contract` relies on the `Template` text as a source of truth. This mechanism ensures that the actual contract always reflects the template, and remains faithful to the original legal text. You can however edit the `Template` itself in order to change the legal text, thereby creating a new template.

Revert your changes, changing the word `timely` back to the original word `delayed` and the parsing error will disappear.

### The Model

Moving along to the `Model` section, you will find the data model for the template variables (the `MiniLateDeliveryClause` type), as well as for the requests (the `LateRequest` type) and response (the `LateResponse` type) for the late delivery and penalty clause.

![Advanced-Late-5](assets/advanced/late5.png)

Note that a `namespace` is declared at the beginning of the file for the model, and that several existing models are being imported (using e.g., `import org.accordproject.cicero.contract.*`). Those imports are needed to access the definition for several types used in the model:
- `AccordClause` which is a generic type for all Accord Project clause templates, and is defined in the `org.accordproject.contract` namespace;
- `Request` and `Response` which are generic types for responses and requests, and are defined in the `org.accordproject.runtime` namespace;
- `Duration` which is defined in the `org.accordproject.time` namespace.

### The Logic

The final part of the template is the `Ergo` tab of the `Logic` section, which describes the business logic.

![Advanced-Late-6](assets/advanced/late6.png)

Thanks to the `namespace` at the beginning of this file, the Ergo engine can know the definition for the `MiniLateDeliveryClause`, as well as the `LateRequest`, and `LateResponse` types defined in the `Model` tab.

To test the template execution, go to the `Test Execution` tab in the `Logic` section. It should be already populated with a valid `Request`. Press the `Send Request` button to trigger the smart clause.

![Advanced-Late-7](assets/advanced/late7.png)

Since the value of the `deliveredAt` parameter in the request is after the value of the `agreedDelivery` parameter in the request, this should return a new response which includes the calculated penalty.

Changing the date for the `deliveredAt` parameter in the request will result in a different penalty.

![Advanced-Late-8](assets/advanced/late8.png)

Note that the clause will return an error if it is called for a timely delivery.

![Advanced-Late-9](assets/advanced/late9.png)

## Add a Penalty Cap

We can now start building a more advanced clause. Let us first take a moment to notice that there is no limitation to the penalty resulting from a late delivery. Under `Test Execution` in `Logic`, send this request:
```json
{
  "$class": "org.accordproject.minilatedeliveryandpenalty.LateRequest",
  "agreedDelivery": "2019-04-10T12:00:00-05:00",
  "deliveredAt": "2019-04-20T03:24:00-05:00",
  "goodsValue": 200
}
```
The penalty should be rather low. Now send this other request:
```json
{
  "$class": "org.accordproject.minilatedeliveryandpenalty.LateRequest",
  "agreedDelivery": "2005-04-01T12:00:00-05:00",
  "deliveredAt": "2019-04-20T03:24:00-05:00",
  "goodsValue": 200
}
```
Notice that the penalty is now quite a large value. It is not unusual to cap a penalty to a maximum amount. Let us now look at how to change the template to add such a cap based on a percentage of the total value of the delivered goods.

### Update the Legal Text

To implement this, we first go to the `Template` tab and add a sentence indicating: `The total amount of penalty shall not, however, exceed [{capPercentage}]% of the total value of the delayed goods.`

For convenience, you can copy-paste the new template text from here:
```md
Late Delivery and Penalty.
```

~~~
In case of delayed delivery of Goods, [{buyer}] shall pay to
[{seller}] a penalty amounting to [{penaltyPercentage}]% of the total
value of the Goods for every [{penaltyDuration}] of delay. The total
amount of penalty shall not, however, exceed [{capPercentage}]% of the
total value of the delayed goods. If the delay is more than
[{maximumDelay}], the Buyer is entitled to terminate this Contract.

```
This should immediately result in an error when parsing the contract text:

![Advanced-Late-10](assets/advanced/late10.png)

As explained in the error message, this is because the new template text uses a
variable `capPercentage` which has not been declared in the model.

### Update the Model

To define this new variable, go to the `Model` tab, and change the
`MiniLateDeliveryClause` type to include `o Double capPercentage`.

![Advanced-Late-11](assets/advanced/late10b.png)

For convenience, you can copy-paste the new `MiniLateDeliveryClause` type from
here:
```ergo
asset MiniLateDeliveryClause extends AccordClause {
  o AccordParty buyer         // Party to the contract (buyer)
  o AccordParty seller        // Party to the contract (seller)
  o Duration penaltyDuration  // Length of time resulting in penalty
  o Double penaltyPercentage  // Penalty percentage
  o Double capPercentage      // Maximum penalty percentage
  o Duration maximumDelay     // Maximum delay before termination
}
```

This result in a new error, this time on the test contract:

![Advanced-Late-11](assets/advanced/late11.png)

To fix it, we simply need to add that same line we added to the template, replacing
the `capPercentage` by a value in the `Test Contract`: `The total amount of penalty
shall not, however, exceed 52% of the total value of the delayed goods.`

For convenience, you can copy-paste the new test contract from here:
```md
Late Delivery and Penalty.

In case of delayed delivery of Goods, "Betty Buyer" shall pay to
"Steve Seller" a penalty amounting to 10.5% of the total
value of the Goods for every 2 days of delay. The total
amount of penalty shall not, however, exceed 52% of the
total value of the delayed goods. If the delay is more than
15 days, the Buyer is entitled to terminate this Contract.

```

Great, now the edited template should have no more errors, and the contract data
should now include the value for the new `capPercentage` variable.
~~~

![Advanced-Late-12](assets/advanced/late12.png)

Note that the `Current Template` Tab indicates that the template has been changed.

### Update the Logic

At this point, executing the logic will still result in large penalties. This is because the logic does not take advantage of the new `capPercentage` variable. Edit the `logic.ergo` code to do so. After step `// 2. Penalty formula` in the logic, apply the penalty cap by adding some logic as follows:
```
    // 3. Capped Penalty
    let cap = contract.capPercentage / 100.0 * request.goodsValue;

    let cappedPenalty =
      if penalty > cap
      then cap
      else penalty;

```
Do not forget to also change the value of the penalty in the returned `LateResponse` to use the new variable `cappedPenalty`:
```
    // 5. Return the response
    return LateResponse{
      penalty: cappedPenalty,
      buyerMayTerminate: termination
    }
```
The logic should now look as follows:

![Advanced-Late-13](assets/advanced/late13.png)

### Execute the new Logic

As a final test of the new template, you should try again to execute the contract with a long delay in delivery. This should now result into a much smaller penalty, which is capped to 52% of the total value of the goods, or 104 USD.

![Advanced-Late-14](assets/advanced/late14.png)

## Emit a Payment Obligation.

As a final extension to this template, we can modify it to emit a Payment Obligation. This first requires us to switch from a Clause template to a Contract template.

### Switch to a Contract Template

The first place to change is in the metadata for the template. This can be done easily with the `full contract` button in the `Current Template` tab. This will immediately result in an error indicating that the model does not contain an `AccordContract` type.

![Advanced-Late-15](assets/advanced/late15.png)

### Update the Model

To fix this, change the model to reflect that we are now editing a contract

template, and change the type `AccordClause` to `AccordContract` in the type definition for the template variables:
```ergo
asset MiniLateDeliveryContract extends AccordContract {
  o AccordParty buyer         // Party to the contract (buyer)
  o AccordParty seller        // Party to the contract (seller)
  o Duration penaltyDuration  // Length of time resulting in penalty
  o Double penaltyPercentage  // Penalty percentage
  o Double capPercentage      // Maximum penalty percentage
  o Duration maximumDelay     // Maximum delay before termination
}
```

The next error is in the logic, since it still uses the old `MiniLateDeliveryClause` type which does not exist anymore.

### Update the Logic

The `Logic` error that occurs here is:
```ergo
Compilation error (at file lib/logic.ergo line 19 col 31). Cannot find type with name 'MiniLateDeliveryClause'
contract MiniLateDelivery over MiniLateDeliveryClause {
                               ^^^^^^^^^^^^^^^^^^^^^^
```
Update the logic to use the the new `MiniLateDeliveryContract` type instead, as follows:
```ergo
contract MiniLateDelivery over MiniLateDeliveryContract {
```

The template should now be without errors.

### Add a Payment Obligation

Our final task is to emit a `PaymentObligation` to indicate that the seller should pay the buyer in the amount of the calculated penalty.

To do so, first import a couple of standard models: for the Cicero's [runtime model](https://models.accordproject.org/cicero/runtime.html) (which contains the definition of a `PaymentObligation`), and for the Accord Project's [money model](https://models.accordproject.org/money.html) (which contains the definition of a `MonetaryAmount). The `import` statements at the top of your logic should look as follows:
```ergo
import org.accordproject.time.*
import org.accordproject.cicero.runtime.*
import org.accordproject.money.MonetaryAmount

```

Lastly, add a new step between steps `// 4.` and `// 5.` in the logic to emit a payment obligation in USD:
```ergo
    emit PaymentObligation{
      contract: contract,
      promisor: some(contract.seller),
      promisee: some(contract.buyer),
      deadline: none,
      amount: MonetaryAmount{ doubleValue: cappedPenalty, currencyCode: "USD" },
      description: contract.seller.partyId ++ " should pay penalty amount to " ++
contract.buyer.partyId
    };

```
That's it! You can observe in the `Test Execution` that an `Obligation` is now being emitted. Try out adjusting values and continuing to send requests and getting responses and obligations.

![Advanced-Late-16](assets/advanced/late16.png)

---
id: version-0.12-advanced-nodejs
title: Working with Node.js
original_id: advanced-nodejs

## Cicero Node.js API

You can work with Accord Project templates directly in JavaScript using Node.js.

Documentation for the API can be found in [Cicero API](cicero-api).

## Working with Templates

### Import the Template class

To import the Cicero class for templates:

```js
const Template = require('@accordproject/cicero-core').Template;
```

### Load a Template

To create a Template instance in memory call the `fromDirectory`, `fromArchive` or `fromUrl` methods:

```js
    const template = await
Template.fromDirectory('./test/data/latedeliveryandpenalty');
```

These methods are asynchronous and return a `Promise`, so you should use `await` to wait for the promise to be resolved.

### Instantiate a Template

Once a Template has been loaded, you can create a Clause based on the Template. You can either instantiate
the Clause using source DSL text (by calling `parse`), or you can set an instance of the template model
as JSON data (by calling `setData`):

```js
    // load the DSL text for the template
    const testLatePenaltyInput = fs.readFileSync(path.resolve(__dirname, 'data/',
'sample.txt'), 'utf8');

    const clause = new Clause(template);
    clause.parse(testLatePenaltyInput);

    // get the JSON object created from the parse
    const data = clause.getData();
```

OR - create a contract and set the data from a JSON object.

```js
    const clause = new Clause(template);
    clause.setData( {$class: 'org.acme.MyTemplateModel', 'foo': 42 } );
```

## Executing a Template Instance

Once you have instantiated a clause or contract instance, you can execute it.

### Import the Engine class

To execute a Clause you first need to create an instance of the ``Engine`` class:

```js
const Engine = require('@accordproject/cicero-engine').Engine;
```

### Send a request to the contract

You can then call ``execute`` on it, passing in the clause or contract instance, and the request:

```js
    const request = {
        '$class':
'org.accordproject.latedeliveryandpenalty.LateDeliveryAndPenaltyRequest',
        'forceMajeure': false,
        'agreedDelivery': '2017-10-07T16:38:01.412Z',
        'goodsValue': 200,
        'transactionId': '402c8f50-9e61-433e-a7c1-afe61c06ef00',
        'timestamp': '2017-11-12T17:38:01.412Z'
    };
    const state = {};
    state.$class = 'org.accordproject.cicero.contract.AccordContractState';
    state.stateId = 'org.accordproject.cicero.contract.AccordContractState#1';
    const result = await engine.execute(clause, request, state);
```

---
id: version-0.12-basic-create
title: Creating a New Template
original_id: basic-create

Now that you have executed an existing template, let's create a new template.

> If you would like to contribute your template back into the `cicero-template-library` please start by [forking](https://help.github.com/articles/fork-a-repo/) the `cicero-template-library` project on GitHub. This will make it easy for you to submit a pull request to get your new template added to the library.

Install the template generator::

```
    npm install -g yo
    npm install -g yo @accordproject/generator-cicero-template@0.12
```

Run the template generator:

```
    yo @accordproject/cicero-template
```

> If you have forked the `cicero-template-library` `cd` into that directory first.

Give your generator a name (no spaces) and then supply a namespace for your template model (again,
no spaces). The generator will then create the files and directories required for a basic template
(based on the helloworld template).

> You may find it easier to edit the grammar, model and logic for your template in VS Code, installing the Accord Project and Hyperledger Composer extensions. The extensions give you syntax highlighting and parser errors within VS Code.

## Update Sample.txt

First, replace the contents of `sample.txt` with the legal text for the contract or clause that you would like to digitize.
Check that when you run `cicero parse` that the `sample.txt` is now invalid with respect to the grammar.

## Edit the Template Grammar

Now update the grammar. Start by replacing the existing grammar, making it identical to the contents of your updated `sample.txt`.

Now introduce variables into your template grammar as required. The variables are marked-up using `[{` and `}]`
with what is between the braces being the name of your variable.

## Edit the Template Model

All of the variables referenced in your template grammar must exist in your template model. Edit
the file `models/model.cto` to include all your variables, making sure the name of the model property matches the name
of the variable in the `template.tem` file.

Note that the Hyperledger Composer Modeling Language primitive data types are:

- String
- Long

- Integer
- DateTime
- Double
- Boolean

> Note that you can import common types (address, monetary amount, country code, etc.) from the Accord Project Model Repository: https://models.accordproject.org.

## Edit the Transaction Types

Your template expects to receive data as input and will produce data as output. The structure of
this request/response data is captured in the `MyRequest` and `MyResponse` transaction types in your model
namespace. Open up the file `models/model.cto` and edit the definition of the `MyRequest` type to
include all the data you expect to receive from the outside world and that will be used by the
business logic of your template. Similarly edit the definition of the `MyResponse` type to include
all the data that the business logic for your template will compute and would like to return to the
caller.

## Edit the Template Logic

Now edit the business logic of the template itself. This is expressed in the Ergo language, which is a strongly-typed function domain specific language for contract logic. Open the file `lib/logic.ergo`
and edit the `helloworld` clause to perform the calculations your logic requires.

Looking at the Ergo logic for other example templates will help you understand the syntax and capabilities of Ergo.

---
id: version-0.12-basic-library
title: Publishing a Template
original_id: basic-library

This tutorial explains how to contribute a new template to the Accord Project Template Library.

## Prerequisites

Accord Project uses [GitHub](https://github.com/) to maintain its open source template library. For this tutorial, you must first obtain and configure the following dependency:

* [Git](https://git-scm.com): We use Node to generate the
  documentation, run a development web server, run tests, and generate
  distributable files. Depending on your system, you can install Node
  either from source or as a pre-packaged bundle.

You will also need a [GitHub](https://github.com/) account. If this is your first time working with GitHub, you can find a number of guides [here](https://guides.github.com).

## Clone the template library

If you have `git` installed you can `git clone` the template library to download all the templates, or you can use the "Download" button inside GitHub:

```bash
git clone https://github.com/accordproject/cicero-template-library
```

## Add a Template to a Library

The Cicero template library is stored in a GitHub repository:
https://github.com/accordproject/cicero-template-library

To contribute new templates please fork the repository and then create a pull request. Note that templates
should have unit tests. See the ``acceptance-of-delivery`` template for an example with unit tests.

---
id: version-0.12-basic-templates
title: Take a Look Inside
original_id: basic-templates

Now that you have executed an existing template in archive form, let us look inside that archive to understand the structure of that template.

## Unpack a Template Archive

Previously, we downloaded and executed an archive (`helloworld@0.10.1.cta`). A `.cta` archive is nothing more than a zip file containing the components of a template. Let's unzip that archive to see what is inside.

First create a directory in the place where you have downloaded that archive, then use the unzip command in your terminal:

```bash
mkdir helloworld
mv helloworld@0.10.1.cta helloworld
cd helloworld
unzip helloworld@0.10.1.cta
```

## Template Components

The layout of a template is always as follows:

```text
package.json
    Metadata for the template (name, version, description etc)

README.md
    A markdown file that describes the purpose and correct usage for the template

sample.txt (optional)
```

    A sample clause or contract text that is valid for the template

state.json (optional)
    A sample valid state for the clause or contract

request.json (optional)
    A sample valid request transaction for the template

grammar/template.tem
    The default grammar for the template

models/
    A collection of Concerto model files for the template. They define the Template
Model
    and models for the State, Request, Response, and Obligations used during
execution.

lib/
    A collection of Ergo files that implement the business logic for the template

test/
    A collection of unit tests for the template

In a nutshell, the template archive contains the three main components of a
template (the natural language text of your Clause or Contract, the data model for
the template, and the executable logic), along with additional metadata and samples
which can be used to illustrate or test the template.

Let us look at each of those components.

### Grammar

The file in `grammar/template.tem` contains the grammar for the template. It is
natural language, with markup to indicate the variable(s) in your Clause or
Contract.

```md
Name of the person to greet: [{name}]. Thank you!
```

In the `helloworld` template there is only one variable `name` which is indicated
between `[{` and `}]`.

### Model

The file in `models/model.cto` contains the data model for the template. This
includes a description for each of the template variables, including what kind of
variable it is (also called their type).

Here is the model for the `helloworld` template:

```ergo
namespace org.accordproject.helloworld

import org.accordproject.cicero.contract.* from
https://models.accordproject.org/cicero/contract.cto
import org.accordproject.cicero.runtime.* from
https://models.accordproject.org/cicero/runtime.cto
```

~~~~
asset TemplateModel extends AccordClause {
  o String name // variable 'name' is of type String
}

transaction MyRequest extends Request {
  o String input
}

transaction MyResponse extends Response {
  o String output
}
```

The `TemplateModel` as well as the `Request` and `Response` are types which are
specified using the [Composer Concerto modeling
language](https://github.com/hyperledger/composer-concerto).

The `TemplateModel` indicate that the template is for a Clause, and should have a
variable `name` of type `String` (i.e., text).

```ergo
asset TemplateModel extends AccordClause {
  o String name // variable 'name' is of type String
}
```

Types are always declared within a namespace (here `org.accordproject.helloworld`),
which provides a mechanism to disambiguate those types amongst multiple model
files.

### Logic

The file in `logic/logic.ergo` contains the executable logic. Each Ergo file is
identified by a namespace, and contains declarations (e.g., constants, functions,
contracts). Here is the Ergo logic for the `helloworld` template:

```ergo
namespace org.accordproject.helloworld

contract HelloWorld over TemplateModel {
  // Simple Clause
  clause greet(request : MyRequest) : MyResponse {
    return MyResponse{ output: "Hello " ++ contract.name ++ " " ++ request.input }
  }
}
```

This declares a single `HelloWorld` contract in the `org.accordproject.helloworld`
namespace, with one `greet`.

The `greet` clause takes a takes a request of type `MyRequest` as input and returns
a response of type `MyResponse`.

It also declares that this contract `HelloWorld` is parameterized over the given
`TemplateModel` found in the `models/model.cto` file.

The code for the `greet` clause returns a new `MyResponse` with a property `output`
which is a string containing the `name` of from the contract (`contract`) and the
~~~~

`input` from the request (`request`). In Ergo, `++` stands for string concatenation.

## Execute the Template

Even after you have unzipped the template archive, you can still parse and execute that template.

## Run Unit Tests

Templates should have unit tests that cover every line of code of their business logic. You may use any of the
popular unit testing frameworks to implement the tests (mocha, chai, sinon etc). Please refer to the
``acceptance-of-delivery`` template for an example template with unit tests.

---
id: version-0.12-basic-use
title: How to Use a Template
original_id: basic-use

The simplest way to work with an Accord Project template is through the Cicero command line interface (CLI). In this tutorial, we explain how to download an existing Accord Project template, create an instance of that template and how to execute the contract logic.

## Install Cicero CLI

In order to access the Cicero command line interface (CLI), first install the `@accordproject/cicero-cli` npm package:

```bash
npm install -g @accordproject/cicero-cli@0.12
```

> If you're new to `npm` the [installation instructions](accordproject-installation) have some more detailed guidance.

## Download a Template

You can download a single clause or contract template from the [Accord Project Template Library](https://templates.accordproject.org) as an archive (`.cta`) file.

If you click on the Template Library link, you should see a Web Page which looks as follows:

![Basic-Use-1](/docs/assets/basic/use1.png)

Scrolling down that page, you can see the index for the open-source templates along with their version, and whether they are a Clause or Contract template.

Click on the link to the `helloworld` template. You should be taken to a page which looks as follows:

![Basic-Use-2](/docs/assets/basic/use2.png)

Then click on the `Download Archive` button under the description for the template (highlighted in the red box in the figure). This should download the latest template archive for the `helloworld` template.

Cicero archives are files with a `.cta` extension, which includes all the different components for the template (the natural language, model and logic).

> Note that the version of `cicero-cli` needs to match the Cicero version that is required by a template.
> * You can check the version of your CLI with `cicero --version`.
> * You can choose a different version of a template with the *Versions* dropdown in the [Accord Project Template Library](https://templates.accordproject.org).
> * Otherwise, install a specific version of the cli, for example for v0.8, use `npm install -g @accordproject/cicero-cli@0.8`.

## Parse a Valid Clause Text

Using your terminal `cd` into the directory that contains the template archive you just downloaded, then create a sample clause text `sample.txt` which contains the following text:

```text
Name of the person to greet: "Fred Blogs".
Thank you!
```

The use the `cicero parse` command in your terminal to load the template and parse your sample clause text. This should be echoing the result of parsing back to your terminal.

```bash
cicero parse --template helloworld@0.10.1.cta --sample sample.txt
```

> Notes:
> * make sure that the version number in that command matches the one for the archive you have downloaded.
> * `cicero parse` requires network access. Make sure that you are online and that your firewall or proxy allows access to `https://models.accordproject.org`

This should print this output:

```json
{
  "$class": "org.accordproject.helloworld.HelloWorldClause",
  "clauseId": "aa3b9db9-f25f-41f4-88a4-64baba728bfe",
  "name": "Fred Blogs"
}
```

## Parse a Non-Valid Clause Text

If you attempt to parse invalid data, this same command should return with line and column information for the syntax error.

Edit your `sample.txt` file to add text that is not consistent with the template:

```text
FUBAR Name of the person to greet: "Fred Blogs".
```

~~~
Thank you!
```

Rerun `cicero parse --template helloworld@0.10.1.cta --sample sample.txt`. The
output should now be:

```text
18:15:22 - error: invalid syntax at line 1 col 1:

  FUBAR Name of the person to greet: "Fred Blogs".
  ^
Unexpected "F"
```

## Execute the Clause

Use the `cicero execute` command to parse a clause text based (your `sample.txt`)
*and* execute the clause logic using an incoming request in JSON format. To do so
you need to create two additional files.

First, create a `state.json` file which contains:

```json
{
    "$class": "org.accordproject.cicero.contract.AccordContractState",
    "stateId": "org.accordproject.cicero.contract.AccordContractState#1"
}
```

This is the initial state for your contract.

Then, create a `request.json` file which contains:

```json
{
  "$class": "org.accordproject.helloworld.MyRequest",
  "input": "Accord Project"
}
```

This is the request which you will send to trigger the execution of your contract.

Then use the `cicero execute` command in your terminal to load the template, parse
your sample clause text *and* execute the request. This should be echoing the
result of execution back to your terminal.

```bash
cicero execute --template helloworld@0.10.1.cta --sample sample.txt --state
state.json --request request.json
```

> Note that `cicero execute` requires network access. Make sure that you are online
and that your firewall or proxy allows access to `https://models.accordproject.org`

This should print this output:

```json
{
  "clause": "helloworld@0.10.1-
~~~

```
d4aab9b009796f56c45872149c1f97a164856b13056f3d503c76d5e519d9f097",
  "request": {
    "$class": "org.accordproject.helloworld.MyRequest",
    "input": "Accord Project",
    "transactionId": "952515b8-eb87-43d7-a582-4afb30eafc6b",
    "timestamp": "2019-04-15T22:44:14.747Z"
  },
  "response": {
    "$class": "org.accordproject.helloworld.MyResponse",
    "output": "Hello Fred Blogs Accord Project",
    "transactionId": "b9c1b74b-db46-4213-a4d0-fbc43f9c753b",
    "timestamp": "2019-04-15T22:44:14.759Z"
  },
  "state": {
    "$class": "org.accordproject.cicero.contract.AccordContractState",
    "stateId": "org.accordproject.cicero.contract.AccordContractState#1"
  },
  "emit": []
}
```

The results of execution displayed back on your terminal is in JSON format, including the following information:

* Details of the `clause` executed (name, version, SHA256 hash of clause data)
* The incoming `request` object (the same request from your `request.json` file)
* The output `response` object
* The output `state` (unchanged in this example)
* An array of `emit`ted events (empty in this example)

## Try Other Examples

That's it! You have successfully parsed and executed your first Accord Project Clause using the `helloworld` template.

Feel free to try the same commands to parse and execute other templates from the Accord Project Library. Note that for each template you can find samples for the text, for the request and for the state on the corresponding Web page. For instance, a sample for the `latedeliveryandpenalty` clause is in the red box in the following image:

![Basic-Use-3](/docs/assets/basic/use3.png)

## To Execute on Different Platforms

Templates may be executed on different platforms, not just from the command line. In the [Advanced Tutorials](advanced-nodejs), you can find information on how to execute a template in a standalone Node.js process, invoked as RESTful services, or deployed as chaincode in Hyperledger Fabric.

---
id: version-0.12-cicero-api
title: Cicero API
original_id: cicero-api

## Modules

```
<dl>
<dt><a href="#module_cicero-engine">cicero-engine</a></dt>
<dd><p>Clause Engine</p>
</dd>
<dt><a href="#module_cicero-core">cicero-core</a></dt>
<dd><p>Cicero Core - defines the core data types for Cicero.</p>
</dd>
</dl>

<a name="module_cicero-engine"></a>

## cicero-engine
Clause Engine


* [cicero-engine](#module_cicero-engine)
    * [.Engine](#module_cicero-engine.Engine)
        * [new Engine()](#new_module_cicero-engine.Engine_new)
        * [.execute(clause, request, state, currentTime)](#module_cicero-
engine.Engine+execute) ⇒ <code>Promise</code>
        * [.init(clause, currentTime)](#module_cicero-engine.Engine+init) ⇒
<code>Promise</code>
        * [.getErgoEngine()](#module_cicero-engine.Engine+getErgoEngine) ⇒
<code>ErgoEngine</code>

<a name="module_cicero-engine.Engine"></a>

### cicero-engine.Engine
<p>
Engine class. Stateless execution of clauses against a request object, returning a
response to the caller.
</p>

**Kind**: static class of [<code>cicero-engine</code>](#module_cicero-engine)
**Access**: public

* [.Engine](#module_cicero-engine.Engine)
    * [new Engine()](#new_module_cicero-engine.Engine_new)
    * [.execute(clause, request, state, currentTime)](#module_cicero-
engine.Engine+execute) ⇒ <code>Promise</code>
    * [.init(clause, currentTime)](#module_cicero-engine.Engine+init) ⇒
<code>Promise</code>
    * [.getErgoEngine()](#module_cicero-engine.Engine+getErgoEngine) ⇒
<code>ErgoEngine</code>

<a name="new_module_cicero-engine.Engine_new"></a>

#### new Engine()
Create the Engine.

<a name="module_cicero-engine.Engine+execute"></a>

#### engine.execute(clause, request, state, currentTime) ⇒ <code>Promise</code>
Execute a clause, passing in the request object

**Kind**: instance method of [<code>Engine</code>](#module_cicero-engine.Engine)
**Returns**: <code>Promise</code> - a promise that resolves to a result for the
clause
```

Param	Type	Description
clause	<code>Clause</code>	the clause to execute
request	<code>object</code>	the request, a JS object that can be deserialized using the Composer serializer.
state	<code>object</code>	the contract state, a JS object that can be deserialized using the Composer serializer.
currentTime	<code>string</code>	the definition of 'now'

<a name="module_cicero-engine.Engine+init"></a>

#### engine.init(clause, currentTime) ⇒ <code>Promise</code>
Initialize a clause

Kind: instance method of [<code>Engine</code>](#module_cicero-engine.Engine)
Returns: <code>Promise</code> - a promise that resolves to a result for the clause initialization

Param	Type	Description
clause	<code>Clause</code>	the clause to execute
currentTime	<code>string</code>	the definition of 'now'

<a name="module_cicero-engine.Engine+getErgoEngine"></a>

#### engine.getErgoEngine() ⇒ <code>ErgoEngine</code>
Provides access to the Ergo engine.

Kind: instance method of [<code>Engine</code>](#module_cicero-engine.Engine)
Returns: <code>ErgoEngine</code> - the underlying Ergo Engine
<a name="module_cicero-core"></a>

## cicero-core
Cicero Core - defines the core data types for Cicero.

* [cicero-core](#module_cicero-core)
    * [.Clause](#module_cicero-core.Clause)
    * [.Contract](#module_cicero-core.Contract)
    * [.DateTimeFormatParser](#module_cicero-core.DateTimeFormatParser)
        * [.parseDateTimeFormatField(field)](#module_cicero-core.DateTimeFormatParser.parseDateTimeFormatField) ⇒ <code>string</code>
        * [.buildDateTimeFormatRule(formatString)](#module_cicero-core.DateTimeFormatParser.buildDateTimeFormatRule) ⇒ <code>Object</code>
    * [.Metadata](#module_cicero-core.Metadata)
        * [new Metadata(packageJson, readme, samples, request)](#new_module_cicero-core.Metadata_new)
        * [.getTemplateType()](#module_cicero-core.Metadata+getTemplateType) ⇒ <code>number</code>
        * [.getRuntime()](#module_cicero-core.Metadata+getRuntime) ⇒ <code>string</code>
        * [.getErgoVersion()](#module_cicero-core.Metadata+getErgoVersion) ⇒ <code>string</code>
        * [.getCiceroVersion()](#module_cicero-core.Metadata+getCiceroVersion) ⇒ <code>string</code>
        * [.satisfiesCiceroVersion(version)](#module_cicero-core.Metadata+satisfiesCiceroVersion) ⇒ <code>string</code>
        * [.getSamples()](#module_cicero-core.Metadata+getSamples) ⇒

```
<code>object</code>
        * [.getRequest()](#module_cicero-core.Metadata+getRequest) ⇒
<code>object</code>
        * [.getSample(locale)](#module_cicero-core.Metadata+getSample) ⇒
<code>string</code>
        * [.getREADME()](#module_cicero-core.Metadata+getREADME) ⇒
<code>String</code>
        * [.getPackageJson()](#module_cicero-core.Metadata+getPackageJson) ⇒
<code>object</code>
        * [.getName()](#module_cicero-core.Metadata+getName) ⇒ <code>string</code>
        * [.getKeywords()](#module_cicero-core.Metadata+getKeywords) ⇒
<code>Array</code>
        * [.getDescription()](#module_cicero-core.Metadata+getDescription) ⇒
<code>string</code>
        * [.getVersion()](#module_cicero-core.Metadata+getVersion) ⇒
<code>string</code>
        * [.getIdentifier()](#module_cicero-core.Metadata+getIdentifier) ⇒
<code>string</code>
        * [.createTargetMetadata(runtimeName)](#module_cicero-
core.Metadata+createTargetMetadata) ⇒ <code>object</code>
    * [.ParserManager](#module_cicero-core.ParserManager)
        * [new ParserManager(template)](#new_module_cicero-core.ParserManager_new)
        * _instance_
            * [.getParser()](#module_cicero-core.ParserManager+getParser) ⇒
<code>object</code>
            * [.getTemplateAst()](#module_cicero-core.ParserManager+getTemplateAst)
⇒ <code>object</code>
            * [.setGrammar(grammar)](#module_cicero-core.ParserManager+setGrammar)
            * [.buildGrammar(templatizedGrammar)](#module_cicero-
core.ParserManager+buildGrammar)
            * [.buildGrammarRules(ast, templateModel, prefix, parts)]
(#module_cicero-core.ParserManager+buildGrammarRules)
            * [.handleBinding(templateModel, parts, inputRule, element)]
(#module_cicero-core.ParserManager+handleBinding)
            * [.cleanChunk(input)](#module_cicero-core.ParserManager+cleanChunk) ⇒
<code>string</code>
            * [.findFirstBinding(propertyName, elements)](#module_cicero-
core.ParserManager+findFirstBinding) ⇒ <code>int</code>
            * [.getGrammar()](#module_cicero-core.ParserManager+getGrammar) ⇒
<code>String</code>
            * [.getTemplatizedGrammar()](#module_cicero-
core.ParserManager+getTemplatizedGrammar) ⇒ <code>String</code>
        * _static_
            * [.getProperty(templateModel, propertyName)](#module_cicero-
core.ParserManager.getProperty) ⇒ <code>\*</code>
            * [.compileGrammar(sourceCode)](#module_cicero-
core.ParserManager.compileGrammar) ⇒ <code>object</code>
    * *[.Template](#module_cicero-core.Template)*
        * *[new Template(packageJson, readme, samples, request)]
(#new_module_cicero-core.Template_new)*
        * _instance_
            * *[.validate()](#module_cicero-core.Template+validate)*
            * *[.getTemplateModel()](#module_cicero-core.Template+getTemplateModel)
⇒ <code>ClassDeclaration</code>*
            * *[.getIdentifier()](#module_cicero-core.Template+getIdentifier) ⇒
<code>String</code>*
            * *[.getMetadata()](#module_cicero-core.Template+getMetadata) ⇒
<code>Metadata</code>*
            * *[.getName()](#module_cicero-core.Template+getName) ⇒
```

```
<code>String</code>*
            * *[.getVersion()](#module_cicero-core.Template+getVersion) ⇒
<code>String</code>*
            * *[.getDescription()](#module_cicero-core.Template+getDescription) ⇒
<code>String</code>*
            * *[.getHash()](#module_cicero-core.Template+getHash) ⇒
<code>string</code>*
            * *[.toArchive([language], [options])](#module_cicero-
core.Template+toArchive) ⇒ <code>Promise.&lt;Buffer&gt;</code>*
            * *[.getParserManager()](#module_cicero-core.Template+getParserManager)
⇒ <code>ParserManager</code>*
            * *[.getTemplateLogic()](#module_cicero-core.Template+getTemplateLogic)
⇒ <code>TemplateLogic</code>*
            * *[.getIntrospector()](#module_cicero-core.Template+getIntrospector) ⇒
<code>Introspector</code>*
            * *[.getFactory()](#module_cicero-core.Template+getFactory) ⇒
<code>Factory</code>*
            * *[.getSerializer()](#module_cicero-core.Template+getSerializer) ⇒
<code>Serializer</code>*
            * *[.getRequestTypes()](#module_cicero-core.Template+getRequestTypes) ⇒
<code>Array</code>*
            * *[.getResponseTypes()](#module_cicero-core.Template+getResponseTypes)
⇒ <code>Array</code>*
            * *[.getEmitTypes()](#module_cicero-core.Template+getEmitTypes) ⇒
<code>Array</code>*
            * *[.getStateTypes()](#module_cicero-core.Template+getStateTypes) ⇒
<code>Array</code>*
            * *[.hasLogic()](#module_cicero-core.Template+hasLogic) ⇒
<code>boolean</code>*
        * _static_
            * *[.fromDirectory(path, [options])](#module_cicero-
core.Template.fromDirectory) ⇒ <code>Promise.&lt;Template&gt;</code>*
            * *[.fromArchive(buffer)](#module_cicero-core.Template.fromArchive) ⇒
<code>Promise.&lt;Template&gt;</code>*
            * *[.fromUrl(url, options)](#module_cicero-core.Template.fromUrl) ⇒
<code>Promise</code>*
            * *[.instanceOf(classDeclaration, fqt)](#module_cicero-
core.Template.instanceOf) ⇒ <code>boolean</code>*
    * *[.TemplateInstance](#module_cicero-core.TemplateInstance)*
        * *[new TemplateInstance(template)](#new_module_cicero-
core.TemplateInstance_new)*
        * _instance_
            * *[.setData(data)](#module_cicero-core.TemplateInstance+setData)*
            * *[.getData()](#module_cicero-core.TemplateInstance+getData) ⇒
<code>object</code>*
            * *[.getDataAsComposerObject()](#module_cicero-
core.TemplateInstance+getDataAsComposerObject) ⇒ <code>object</code>*
            * *[.parse(text, currentTime)](#module_cicero-
core.TemplateInstance+parse)*
            * *[.generateText()](#module_cicero-core.TemplateInstance+generateText)
⇒ <code>string</code>*
            * *[.getIdentifier()](#module_cicero-
core.TemplateInstance+getIdentifier) ⇒ <code>String</code>*
            * *[.getTemplate()](#module_cicero-core.TemplateInstance+getTemplate) ⇒
<code>Template</code>*
            * *[.getTemplateLogic()](#module_cicero-
core.TemplateInstance+getTemplateLogic) ⇒ <code>TemplateLogic</code>*
            * *[.toJSON()](#module_cicero-core.TemplateInstance+toJSON) ⇒
<code>object</code>*
```

```
        * _static_
            * *[.pad(n, width, z)](#module_cicero-core.TemplateInstance.pad) ⇒
<code>string</code>*
            * *[.convertDateTimes(obj, utcOffset)](#module_cicero-
core.TemplateInstance.convertDateTimes) ⇒ <code>\*</code>*
    * *[.TemplateLoader](#module_cicero-core.TemplateLoader)*
        * *[.loadZipFileContents(zip, path, json, required)](#module_cicero-
core.TemplateLoader.loadZipFileContents) ⇒ <code>Promise.&lt;string&gt;</code>*
        * *[.loadZipFilesContents(zip, regex)](#module_cicero-
core.TemplateLoader.loadZipFilesContents) ⇒
<code>Promise.&lt;Array.&lt;object&gt;&gt;</code>*
        * *[.loadFileContents(path, fileName, json, required)](#module_cicero-
core.TemplateLoader.loadFileContents) ⇒ <code>Promise.&lt;string&gt;</code>*
        * *[.loadFilesContents(path, regex)](#module_cicero-
core.TemplateLoader.loadFilesContents) ⇒
<code>Promise.&lt;Array.&lt;object&gt;&gt;</code>*
        * *[.fromArchive(Template, buffer)](#module_cicero-
core.TemplateLoader.fromArchive) ⇒ <code>Promise.&lt;Template&gt;</code>*
        * *[.fromUrl(Template, url, options)](#module_cicero-
core.TemplateLoader.fromUrl) ⇒ <code>Promise</code>*
        * *[.fromDirectory(Template, path, [options])](#module_cicero-
core.TemplateLoader.fromDirectory) ⇒ <code>Promise.&lt;Template&gt;</code>*
    * [.TemplateSaver](#module_cicero-core.TemplateSaver)
        * [.toArchive(template, [language], [options])](#module_cicero-
core.TemplateSaver.toArchive) ⇒ <code>Promise.&lt;Buffer&gt;</code>

<a name="module_cicero-core.Clause"></a>

### cicero-core.Clause
A Clause is executable business logic, linked to a natural language (legally
enforceable) template.
A Clause must be constructed with a template and then prior to execution the data
for the clause must be set.
Set the data for the clause (an instance of the template model) by either calling
the setData method or by
calling the parse method and passing in natural language text that conforms to the
template grammar.

**Kind**: static class of [<code>cicero-core</code>](#module_cicero-core)
**Access**: public
<a name="module_cicero-core.Contract"></a>

### cicero-core.Contract
A Contract is executable business logic, linked to a natural language (legally
enforceable) template.
A Clause must be constructed with a template and then prior to execution the data
for the clause must be set.
Set the data for the clause (an instance of the template model) by either calling
the setData method or by
calling the parse method and passing in natural language text that conforms to the
template grammar.

**Kind**: static class of [<code>cicero-core</code>](#module_cicero-core)
**Access**: public
<a name="module_cicero-core.DateTimeFormatParser"></a>

### cicero-core.DateTimeFormatParser
Parses a date/time format string
```

```
**Kind**: static class of [<code>cicero-core</code>](#module_cicero-core)
**Access**: public

* [.DateTimeFormatParser](#module_cicero-core.DateTimeFormatParser)
    * [.parseDateTimeFormatField(field)](#module_cicero-
core.DateTimeFormatParser.parseDateTimeFormatField) ⇒ <code>string</code>
    * [.buildDateTimeFormatRule(formatString)](#module_cicero-
core.DateTimeFormatParser.buildDateTimeFormatRule) ⇒ <code>Object</code>

<a name="module_cicero-core.DateTimeFormatParser.parseDateTimeFormatField"></a>

#### DateTimeFormatParser.parseDateTimeFormatField(field) ⇒ <code>string</code>
Given a format field (like HH or D) this method returns
a logical name for the field. Note the logical names
have been picked to align with the moment constructor that takes an object.

**Kind**: static method of [<code>DateTimeFormatParser</code>](#module_cicero-
core.DateTimeFormatParser)
**Returns**: <code>string</code> - the field designator

| Param | Type | Description |
| --- | --- | --- |
| field | <code>string</code> | the input format field |

<a name="module_cicero-core.DateTimeFormatParser.buildDateTimeFormatRule"></a>

#### DateTimeFormatParser.buildDateTimeFormatRule(formatString) ⇒
<code>Object</code>
Converts a format string to a Nearley action

**Kind**: static method of [<code>DateTimeFormatParser</code>](#module_cicero-
core.DateTimeFormatParser)
**Returns**: <code>Object</code> - the tokens and action and name to use for the
Nearley rule

| Param | Type | Description |
| --- | --- | --- |
| formatString | <code>string</code> | the input format string |

<a name="module_cicero-core.Metadata"></a>

### cicero-core.Metadata
Defines the metadata for a Template, including the name, version, README markdown.

**Kind**: static class of [<code>cicero-core</code>](#module_cicero-core)
**Access**: public

* [.Metadata](#module_cicero-core.Metadata)
    * [new Metadata(packageJson, readme, samples, request)](#new_module_cicero-
core.Metadata_new)
    * [.getTemplateType()](#module_cicero-core.Metadata+getTemplateType) ⇒
<code>number</code>
    * [.getRuntime()](#module_cicero-core.Metadata+getRuntime) ⇒
<code>string</code>
    * [.getErgoVersion()](#module_cicero-core.Metadata+getErgoVersion) ⇒
<code>string</code>
    * [.getCiceroVersion()](#module_cicero-core.Metadata+getCiceroVersion) ⇒
<code>string</code>
    * [.satisfiesCiceroVersion(version)](#module_cicero-
core.Metadata+satisfiesCiceroVersion) ⇒ <code>string</code>
    * [.getSamples()](#module_cicero-core.Metadata+getSamples) ⇒
<code>object</code>
    * [.getRequest()](#module_cicero-core.Metadata+getRequest) ⇒
<code>object</code>
    * [.getSample(locale)](#module_cicero-core.Metadata+getSample) ⇒
<code>string</code>
    * [.getREADME()](#module_cicero-core.Metadata+getREADME) ⇒ <code>String</code>
    * [.getPackageJson()](#module_cicero-core.Metadata+getPackageJson) ⇒
<code>object</code>
    * [.getName()](#module_cicero-core.Metadata+getName) ⇒ <code>string</code>
    * [.getKeywords()](#module_cicero-core.Metadata+getKeywords) ⇒
<code>Array</code>
    * [.getDescription()](#module_cicero-core.Metadata+getDescription) ⇒
<code>string</code>
    * [.getVersion()](#module_cicero-core.Metadata+getVersion) ⇒
<code>string</code>
    * [.getIdentifier()](#module_cicero-core.Metadata+getIdentifier) ⇒
<code>string</code>
    * [.createTargetMetadata(runtimeName)](#module_cicero-
core.Metadata+createTargetMetadata) ⇒ <code>object</code>

<a name="new_module_cicero-core.Metadata_new"></a>

#### new Metadata(packageJson, readme, samples, request)
Create the Metadata.
<p>
<strong>Note: Only to be called by framework code. Applications should
retrieve instances from [Template](Template)</strong>
</p>


| Param | Type | Description |
| --- | --- | --- |
| packageJson | <code>object</code> | the JS object for package.json (required) |
| readme | <code>String</code> | the README.md for the template (may be null) |
| samples | <code>object</code> | the sample text for the template in different
locales, |
| request | <code>object</code> | the JS object for the sample request represented
as an object whose keys are the locales and whose values are the sample text. For
example:  {      default: 'default sample text',      en: 'sample text in english',
fr: 'exemple de texte français'  } Locale keys (with the exception of default)
conform to the IETF Language Tag specification (BCP 47). THe `default` key
represents sample template text in a non-specified language, stored in a file
called `sample.txt`. |

<a name="module_cicero-core.Metadata+getTemplateType"></a>

#### metadata.getTemplateType() ⇒ <code>number</code>
Returns either a 0 (for a contract template), or 1 (for a clause template)

**Kind**: instance method of [<code>Metadata</code>](#module_cicero-core.Metadata)
**Returns**: <code>number</code> - the template type
<a name="module_cicero-core.Metadata+getRuntime"></a>

#### metadata.getRuntime() ⇒ <code>string</code>
Returns the name of the runtime target for this template, or null if this template
has not been compiled for a specific runtime.
```

Kind: instance method of [<code>Metadata</code>](#module_cicero-core.Metadata)
Returns: <code>string</code> - the name of the runtime
<a name="module_cicero-core.Metadata+getErgoVersion"></a>

#### metadata.getErgoVersion() ⇒ <code>string</code>
Returns the Ergo version that the Ergo code in this template is compatible with.
This
is null for templates that do not contain source Ergo code.

Kind: instance method of [<code>Metadata</code>](#module_cicero-core.Metadata)
Returns: <code>string</code> - the version of Ergo
<a name="module_cicero-core.Metadata+getCiceroVersion"></a>

#### metadata.getCiceroVersion() ⇒ <code>string</code>
Returns the version of Cicero that this template is compatible with.
i.e. which version of the runtime was this template built for?
The version string conforms to the semver definition

Kind: instance method of [<code>Metadata</code>](#module_cicero-core.Metadata)
Returns: <code>string</code> - the semantic version
<a name="module_cicero-core.Metadata+satisfiesCiceroVersion"></a>

#### metadata.satisfiesCiceroVersion(version) ⇒ <code>string</code>
Only returns true if the current cicero version satisfies the target version of
this template

Kind: instance method of [<code>Metadata</code>](#module_cicero-core.Metadata)
Returns: <code>string</code> - the semantic version

Param	Type	Description
version	<code>string</code>	the cicero version to check against

<a name="module_cicero-core.Metadata+getSamples"></a>

#### metadata.getSamples() ⇒ <code>object</code>
Returns the samples for this template.

Kind: instance method of [<code>Metadata</code>](#module_cicero-core.Metadata)
Returns: <code>object</code> - the sample files for the template
<a name="module_cicero-core.Metadata+getRequest"></a>

#### metadata.getRequest() ⇒ <code>object</code>
Returns the sample request for this template.

Kind: instance method of [<code>Metadata</code>](#module_cicero-core.Metadata)
Returns: <code>object</code> - the sample request for the template
<a name="module_cicero-core.Metadata+getSample"></a>

#### metadata.getSample(locale) ⇒ <code>string</code>
Returns the sample for this template in the given locale. This may be null.
If no locale is specified returns the default sample if it has been specified.

Kind: instance method of [<code>Metadata</code>](#module_cicero-core.Metadata)
Returns: <code>string</code> - the sample file for the template in the given
locale or null

Param	Type	Default	Description

| locale | <code>string</code> | <code>null</code> | the IETF language code for the language. |

<a name="module_cicero-core.Metadata+getREADME"></a>

#### metadata.getREADME() ⇒ <code>String</code>
Returns the README.md for this template. This may be null if the template does not have a README.md

Kind: instance method of [<code>Metadata</code>](#module_cicero-core.Metadata)
Returns: <code>String</code> - the README.md file for the template or null
<a name="module_cicero-core.Metadata+getPackageJson"></a>

#### metadata.getPackageJson() ⇒ <code>object</code>
Returns the package.json for this template.

Kind: instance method of [<code>Metadata</code>](#module_cicero-core.Metadata)
Returns: <code>object</code> - the Javascript object for package.json
<a name="module_cicero-core.Metadata+getName"></a>

#### metadata.getName() ⇒ <code>string</code>
Returns the name for this template.

Kind: instance method of [<code>Metadata</code>](#module_cicero-core.Metadata)
Returns: <code>string</code> - the name of the template
<a name="module_cicero-core.Metadata+getKeywords"></a>

#### metadata.getKeywords() ⇒ <code>Array</code>
Returns the name for this template.

Kind: instance method of [<code>Metadata</code>](#module_cicero-core.Metadata)
Returns: <code>Array</code> - the name of the template
<a name="module_cicero-core.Metadata+getDescription"></a>

#### metadata.getDescription() ⇒ <code>string</code>
Returns the description for this template.

Kind: instance method of [<code>Metadata</code>](#module_cicero-core.Metadata)
Returns: <code>string</code> - the description of the template
<a name="module_cicero-core.Metadata+getVersion"></a>

#### metadata.getVersion() ⇒ <code>string</code>
Returns the version for this template.

Kind: instance method of [<code>Metadata</code>](#module_cicero-core.Metadata)
Returns: <code>string</code> - the description of the template
<a name="module_cicero-core.Metadata+getIdentifier"></a>

#### metadata.getIdentifier() ⇒ <code>string</code>
Returns the identifier for this template, formed from name@version.

Kind: instance method of [<code>Metadata</code>](#module_cicero-core.Metadata)
Returns: <code>string</code> - the identifier of the template
<a name="module_cicero-core.Metadata+createTargetMetadata"></a>

#### metadata.createTargetMetadata(runtimeName) ⇒ <code>object</code>
Return new Metadata for a target runtime

Kind: instance method of [<code>Metadata</code>](#module_cicero-core.Metadata)

Returns: <code>object</code> - the new Metadata

Param	Type	Description
runtimeName	<code>string</code>	the target runtime name

<a name="module_cicero-core.ParserManager"></a>

### cicero-core.ParserManager
Generates and manages a Nearley parser for a template.

Kind: static class of [<code>cicero-core</code>](#module_cicero-core)
Access: public

* [.ParserManager](#module_cicero-core.ParserManager)
    * [new ParserManager(template)](#new_module_cicero-core.ParserManager_new)
    * _instance_
        * [.getParser()](#module_cicero-core.ParserManager+getParser) ⇒ <code>object</code>
        * [.getTemplateAst()](#module_cicero-core.ParserManager+getTemplateAst) ⇒ <code>object</code>
        * [.setGrammar(grammar)](#module_cicero-core.ParserManager+setGrammar)
        * [.buildGrammar(templatizedGrammar)](#module_cicero-core.ParserManager+buildGrammar)
        * [.buildGrammarRules(ast, templateModel, prefix, parts)](#module_cicero-core.ParserManager+buildGrammarRules)
        * [.handleBinding(templateModel, parts, inputRule, element)](#module_cicero-core.ParserManager+handleBinding)
        * [.cleanChunk(input)](#module_cicero-core.ParserManager+cleanChunk) ⇒ <code>string</code>
        * [.findFirstBinding(propertyName, elements)](#module_cicero-core.ParserManager+findFirstBinding) ⇒ <code>int</code>
        * [.getGrammar()](#module_cicero-core.ParserManager+getGrammar) ⇒ <code>String</code>
        * [.getTemplatizedGrammar()](#module_cicero-core.ParserManager+getTemplatizedGrammar) ⇒ <code>String</code>
    * _static_
        * [.getProperty(templateModel, propertyName)](#module_cicero-core.ParserManager.getProperty) ⇒ <code>*</code>
        * [.compileGrammar(sourceCode)](#module_cicero-core.ParserManager.compileGrammar) ⇒ <code>object</code>

<a name="new_module_cicero-core.ParserManager_new"></a>

#### new ParserManager(template)
Create the ParserManager.

Param	Type	Description
template	<code>object</code>	the template instance

<a name="module_cicero-core.ParserManager+getParser"></a>

#### parserManager.getParser() ⇒ <code>object</code>
Gets a parser object for this template

Kind: instance method of [<code>ParserManager</code>](#module_cicero-core.ParserManager)

Returns: <code>object</code> - the parser for this template
<a name="module_cicero-core.ParserManager+getTemplateAst"></a>

#### parserManager.getTemplateAst() ⇒ <code>object</code>
Gets the AST for the template

Kind: instance method of [<code>ParserManager</code>](#module_cicero-core.ParserManager)
Returns: <code>object</code> - the AST for the template
<a name="module_cicero-core.ParserManager+setGrammar"></a>

#### parserManager.setGrammar(grammar)
Set the grammar for the template

Kind: instance method of [<code>ParserManager</code>](#module_cicero-core.ParserManager)

Param	Type	Description
grammar	<code>String</code>	the grammar for the template

<a name="module_cicero-core.ParserManager+buildGrammar"></a>

#### parserManager.buildGrammar(templatizedGrammar)
Build a grammar from a template

Kind: instance method of [<code>ParserManager</code>](#module_cicero-core.ParserManager)

Param	Type	Description
templatizedGrammar	<code>String</code>	the annotated template

<a name="module_cicero-core.ParserManager+buildGrammarRules"></a>

#### parserManager.buildGrammarRules(ast, templateModel, prefix, parts)
Build grammar rules from a template

Kind: instance method of [<code>ParserManager</code>](#module_cicero-core.ParserManager)

Param	Type	Description
ast	<code>object</code>	the AST from which to build the grammar
templateModel	<code>ClassDeclaration</code>	the type of the parent class for this AST
prefix	<code>String</code>	A unique prefix for the grammar rules
parts	<code>Object</code>	Result object to acculumate rules and required sub-grammars

<a name="module_cicero-core.ParserManager+handleBinding"></a>

#### parserManager.handleBinding(templateModel, parts, inputRule, element)
Utility method to generate a grammar rule for a variable binding

Kind: instance method of [<code>ParserManager</code>](#module_cicero-core.ParserManager)

| Param | Type | Description |

| --- | --- | --- |
templateModel	<code>ClassDeclaration</code>	the current template model
parts	<code>*</code>	the parts, where the rule will be added
inputRule	<code>*</code>	the rule we are processing in the AST
element	<code>*</code>	the current element in the AST

<a name="module_cicero-core.ParserManager+cleanChunk"></a>

#### parserManager.cleanChunk(input) ⇒ <code>string</code>
Cleans a chunk of text to make it safe to include
as a grammar rule. We need to remove linefeeds and
escape any '"' characters.

Kind: instance method of [<code>ParserManager</code>](#module_cicero-core.ParserManager)
Returns: <code>string</code> - cleaned text

Param	Type	Description
input	<code>string</code>	the input text from the template

<a name="module_cicero-core.ParserManager+findFirstBinding"></a>

#### parserManager.findFirstBinding(propertyName, elements) ⇒ <code>int</code>
Finds the first binding for the given property

Kind: instance method of [<code>ParserManager</code>](#module_cicero-core.ParserManager)
Returns: <code>int</code> - the index of the element or -1

Param	Type	Description
propertyName	<code>string</code>	the name of the property
elements	<code>Array.&lt;object&gt;</code>	the result of parsing the template_txt.

<a name="module_cicero-core.ParserManager+getGrammar"></a>

#### parserManager.getGrammar() ⇒ <code>String</code>
Get the (compiled) grammar for the template

Kind: instance method of [<code>ParserManager</code>](#module_cicero-core.ParserManager)
Returns: <code>String</code> - - the grammar for the template
<a name="module_cicero-core.ParserManager+getTemplatizedGrammar"></a>

#### parserManager.getTemplatizedGrammar() ⇒ <code>String</code>
Returns the templatized grammar

Kind: instance method of [<code>ParserManager</code>](#module_cicero-core.ParserManager)
Returns: <code>String</code> - the contents of the templatized grammar
<a name="module_cicero-core.ParserManager.getProperty"></a>

#### ParserManager.getProperty(templateModel, propertyName) ⇒ <code>*</code>
Throws an error if a template variable doesn't exist on the model.

Kind: static method of [<code>ParserManager</code>](#module_cicero-core.ParserManager)

Returns: <code>*</code> - the property

Param	Type	Description
templateModel	<code>*</code>	the model for the template
propertyName	<code>String</code>	the name of the property

<a name="module_cicero-core.ParserManager.compileGrammar"></a>

#### ParserManager.compileGrammar(sourceCode) ⇒ <code>object</code>
Compiles a Nearley grammar to its AST

Kind: static method of [<code>ParserManager</code>](#module_cicero-
core.ParserManager)
Returns: <code>object</code> - the AST for the grammar

Param	Type	Description
sourceCode	<code>string</code>	the source text for the grammar

<a name="module_cicero-core.Template"></a>

### *cicero-core.Template*
A template for a legal clause or contract. A Template has a template model,
request/response transaction types,
a template grammar (natural language for the template) as well as Ergo code for the
business logic of the
template.

Kind: static abstract class of [<code>cicero-core</code>](#module_cicero-core)
Access: public

* *[.Template](#module_cicero-core.Template)*
    * *[new Template(packageJson, readme, samples, request)](#new_module_cicero-
core.Template_new)*
    * _instance_
        * *[.validate()](#module_cicero-core.Template+validate)*
        * *[.getTemplateModel()](#module_cicero-core.Template+getTemplateModel) ⇒
<code>ClassDeclaration</code>*
        * *[.getIdentifier()](#module_cicero-core.Template+getIdentifier) ⇒
<code>String</code>*
        * *[.getMetadata()](#module_cicero-core.Template+getMetadata) ⇒
<code>Metadata</code>*
        * *[.getName()](#module_cicero-core.Template+getName) ⇒
<code>String</code>*
        * *[.getVersion()](#module_cicero-core.Template+getVersion) ⇒
<code>String</code>*
        * *[.getDescription()](#module_cicero-core.Template+getDescription) ⇒
<code>String</code>*
        * *[.getHash()](#module_cicero-core.Template+getHash) ⇒
<code>string</code>*
        * *[.toArchive([language], [options])](#module_cicero-
core.Template+toArchive) ⇒ <code>Promise.&lt;Buffer&gt;</code>*
        * *[.getParserManager()](#module_cicero-core.Template+getParserManager) ⇒
<code>ParserManager</code>*
        * *[.getTemplateLogic()](#module_cicero-core.Template+getTemplateLogic) ⇒
<code>TemplateLogic</code>*
        * *[.getIntrospector()](#module_cicero-core.Template+getIntrospector) ⇒
<code>Introspector</code>*

```
        * *[.getFactory()](#module_cicero-core.Template+getFactory) ⇒
<code>Factory</code>*
        * *[.getSerializer()](#module_cicero-core.Template+getSerializer) ⇒
<code>Serializer</code>*
        * *[.getRequestTypes()](#module_cicero-core.Template+getRequestTypes) ⇒
<code>Array</code>*
        * *[.getResponseTypes()](#module_cicero-core.Template+getResponseTypes) ⇒
<code>Array</code>*
        * *[.getEmitTypes()](#module_cicero-core.Template+getEmitTypes) ⇒
<code>Array</code>*
        * *[.getStateTypes()](#module_cicero-core.Template+getStateTypes) ⇒
<code>Array</code>*
        * *[.hasLogic()](#module_cicero-core.Template+hasLogic) ⇒
<code>boolean</code>*
    * _static_
        * *[.fromDirectory(path, [options])](#module_cicero-
core.Template.fromDirectory) ⇒ <code>Promise.&lt;Template&gt;</code>*
        * *[.fromArchive(buffer)](#module_cicero-core.Template.fromArchive) ⇒
<code>Promise.&lt;Template&gt;</code>*
        * *[.fromUrl(url, options)](#module_cicero-core.Template.fromUrl) ⇒
<code>Promise</code>*
        * *[.instanceOf(classDeclaration, fqt)](#module_cicero-
core.Template.instanceOf) ⇒ <code>boolean</code>*

<a name="new_module_cicero-core.Template_new"></a>

#### *new Template(packageJson, readme, samples, request)*
Create the Template.
Note: Only to be called by framework code. Applications should
retrieve instances from [Template.fromArchive](Template.fromArchive) or
[Template.fromDirectory](Template.fromDirectory).


| Param | Type | Description |
| --- | --- | --- |
| packageJson | <code>object</code> | the JS object for package.json |
| readme | <code>String</code> | the readme in markdown for the template (optional)
|
| samples | <code>object</code> | the sample text for the template in different
locales |
| request | <code>object</code> | the JS object for the sample request |

<a name="module_cicero-core.Template+validate"></a>

#### *template.validate()*
Verifies that the template is well formed.
Throws an exception with the details of any validation errors.

**Kind**: instance method of [<code>Template</code>](#module_cicero-core.Template)
<a name="module_cicero-core.Template+getTemplateModel"></a>

#### *template.getTemplateModel() ⇒ <code>ClassDeclaration</code>*
Returns the template model for the template

**Kind**: instance method of [<code>Template</code>](#module_cicero-core.Template)
**Returns**: <code>ClassDeclaration</code> - the template model for the template
**Throws**:

- <code>Error</code> if no template model is found, or multiple template models are
```

found

<a name="module_cicero-core.Template+getIdentifier"></a>

#### *template.getIdentifier() ⇒ <code>String</code>*
Returns the identifier for this template

Kind: instance method of [<code>Template</code>](#module_cicero-core.Template)
Returns: <code>String</code> - the identifier of this template
<a name="module_cicero-core.Template+getMetadata"></a>

#### *template.getMetadata() ⇒ <code>Metadata</code>*
Returns the metadata for this template

Kind: instance method of [<code>Template</code>](#module_cicero-core.Template)
Returns: <code>Metadata</code> - the metadata for this template
<a name="module_cicero-core.Template+getName"></a>

#### *template.getName() ⇒ <code>String</code>*
Returns the name for this template

Kind: instance method of [<code>Template</code>](#module_cicero-core.Template)
Returns: <code>String</code> - the name of this template
<a name="module_cicero-core.Template+getVersion"></a>

#### *template.getVersion() ⇒ <code>String</code>*
Returns the version for this template

Kind: instance method of [<code>Template</code>](#module_cicero-core.Template)
Returns: <code>String</code> - the version of this template. Use semver module
to parse.
<a name="module_cicero-core.Template+getDescription"></a>

#### *template.getDescription() ⇒ <code>String</code>*
Returns the description for this template

Kind: instance method of [<code>Template</code>](#module_cicero-core.Template)
Returns: <code>String</code> - the description of this template
<a name="module_cicero-core.Template+getHash"></a>

#### *template.getHash() ⇒ <code>string</code>*
Gets a content based SHA-256 hash for this template. Hash
is based on the metadata for the template plus the contents of
all the models and all the script files.

Kind: instance method of [<code>Template</code>](#module_cicero-core.Template)
Returns: <code>string</code> - the SHA-256 hash in hex format
<a name="module_cicero-core.Template+toArchive"></a>

#### *template.toArchive([language], [options]) ⇒
<code>Promise.&lt;Buffer&gt;</code>*
Persists this template to a Cicero Template Archive (cta) file.

Kind: instance method of [<code>Template</code>](#module_cicero-core.Template)
Returns: <code>Promise.&lt;Buffer&gt;</code> - the zlib buffer

Param	Type	Description
[language]	<code>string</code>	target language for the archive (should be

```
'ergo') |
| [options] | <code>Object</code> | JSZip options |

<a name="module_cicero-core.Template+getParserManager"></a>

#### *template.getParserManager() ⇒ <code>ParserManager</code>*
Provides access to the parser manager for this template.
The parser manager can convert template data to and from
natural language text.

**Kind**: instance method of [<code>Template</code>](#module_cicero-core.Template)
**Returns**: <code>ParserManager</code> - the ParserManager for this template
<a name="module_cicero-core.Template+getTemplateLogic"></a>

#### *template.getTemplateLogic() ⇒ <code>TemplateLogic</code>*
Provides access to the template logic for this template.
The template logic encapsulate the code necessary to
execute the clause or contract.

**Kind**: instance method of [<code>Template</code>](#module_cicero-core.Template)
**Returns**: <code>TemplateLogic</code> - the TemplateLogic for this template
<a name="module_cicero-core.Template+getIntrospector"></a>

#### *template.getIntrospector() ⇒ <code>Introspector</code>*
Provides access to the Introspector for this template. The Introspector
is used to reflect on the types defined within this template.

**Kind**: instance method of [<code>Template</code>](#module_cicero-core.Template)
**Returns**: <code>Introspector</code> - the Introspector for this template
<a name="module_cicero-core.Template+getFactory"></a>

#### *template.getFactory() ⇒ <code>Factory</code>*
Provides access to the Factory for this template. The Factory
is used to create the types defined in this template.

**Kind**: instance method of [<code>Template</code>](#module_cicero-core.Template)
**Returns**: <code>Factory</code> - the Factory for this template
<a name="module_cicero-core.Template+getSerializer"></a>

#### *template.getSerializer() ⇒ <code>Serializer</code>*
Provides access to the Serializer for this template. The Serializer
is used to serialize instances of the types defined within this template.

**Kind**: instance method of [<code>Template</code>](#module_cicero-core.Template)
**Returns**: <code>Serializer</code> - the Serializer for this template
<a name="module_cicero-core.Template+getRequestTypes"></a>

#### *template.getRequestTypes() ⇒ <code>Array</code>*
Provides a list of the input types that are accepted by this Template. Types use
the fully-qualified form.

**Kind**: instance method of [<code>Template</code>](#module_cicero-core.Template)
**Returns**: <code>Array</code> - a list of the request types
<a name="module_cicero-core.Template+getResponseTypes"></a>

#### *template.getResponseTypes() ⇒ <code>Array</code>*
Provides a list of the response types that are returned by this Template. Types use
the fully-qualified form.
```

Kind: instance method of [<code>Template</code>](#module_cicero-core.Template)
Returns: <code>Array</code> - a list of the response types
<a name="module_cicero-core.Template+getEmitTypes"></a>

#### *template.getEmitTypes() ⇒ <code>Array</code>*
Provides a list of the emit types that are emitted by this Template. Types use the fully-qualified form.

Kind: instance method of [<code>Template</code>](#module_cicero-core.Template)
Returns: <code>Array</code> - a list of the emit types
<a name="module_cicero-core.Template+getStateTypes"></a>

#### *template.getStateTypes() ⇒ <code>Array</code>*
Provides a list of the state types that are expected by this Template. Types use the fully-qualified form.

Kind: instance method of [<code>Template</code>](#module_cicero-core.Template)
Returns: <code>Array</code> - a list of the state types
<a name="module_cicero-core.Template+hasLogic"></a>

#### *template.hasLogic() ⇒ <code>boolean</code>*
Returns true if the template has logic, i.e. has more than one script file.

Kind: instance method of [<code>Template</code>](#module_cicero-core.Template)
Returns: <code>boolean</code> - true if the template has logic
<a name="module_cicero-core.Template.fromDirectory"></a>

#### *Template.fromDirectory(path, [options]) ⇒ <code>Promise.&lt;Template&gt;</code>*
Builds a Template from the contents of a directory.
The directory must include a package.json in the root (used to specify
the name, version and description of the template).

Kind: static method of [<code>Template</code>](#module_cicero-core.Template)
Returns: <code>Promise.&lt;Template&gt;</code> - a Promise to the instantiated template

Param	Type	Description
path	<code>String</code>	to a local directory
[options]	<code>Object</code>	an optional set of options to configure the instance.

<a name="module_cicero-core.Template.fromArchive"></a>

#### *Template.fromArchive(buffer) ⇒ <code>Promise.&lt;Template&gt;</code>*
Create a template from an archive.

Kind: static method of [<code>Template</code>](#module_cicero-core.Template)
Returns: <code>Promise.&lt;Template&gt;</code> - a Promise to the template

Param	Type	Description
buffer	<code>Buffer</code>	the buffer to a Cicero Template Archive (cta) file

<a name="module_cicero-core.Template.fromUrl"></a>

#### *Template.fromUrl(url, options) ⇒ <code>Promise</code>*

Create a template from an URL.

Kind: static method of [<code>Template</code>](#module_cicero-core.Template)
Returns: <code>Promise</code> - a Promise to the template

Param	Type	Default	Description
url	<code>String</code>		the URL to a Cicero Template Archive (cta) file
options	<code>object</code>	<code></code>	additional options

<a name="module_cicero-core.Template.instanceOf"></a>

#### *Template.instanceOf(classDeclaration, fqt) ⇒ <code>boolean</code>*
Check to see if a ClassDeclaration is an instance of the specified fully qualified
type name.

Kind: static method of [<code>Template</code>](#module_cicero-core.Template)
Returns: <code>boolean</code> - True if classDeclaration an instance of the
specified fully
qualified type name, false otherwise.
Internal:

Param	Type	Description
classDeclaration	<code>ClassDeclaration</code>	The class to test
fqt	<code>String</code>	The fully qualified type name.

<a name="module_cicero-core.TemplateInstance"></a>

### *cicero-core.TemplateInstance*
A TemplateInstance is an instance of a Clause or Contract template. It is
executable business logic, linked to
a natural language (legally enforceable) template.
A TemplateInstance must be constructed with a template and then prior to execution
the data for the clause must be set.
Set the data for the TemplateInstance by either calling the setData method or by
calling the parse method and passing in natural language text that conforms to the
template grammar.

Kind: static abstract class of [<code>cicero-core</code>](#module_cicero-core)
Access: public

* *[.TemplateInstance](#module_cicero-core.TemplateInstance)*
    * *[new TemplateInstance(template)](#new_module_cicero-
core.TemplateInstance_new)*
    * _instance_
        * *[.setData(data)](#module_cicero-core.TemplateInstance+setData)*
        * *[.getData()](#module_cicero-core.TemplateInstance+getData) ⇒
<code>object</code>*
        * *[.getDataAsComposerObject()](#module_cicero-
core.TemplateInstance+getDataAsComposerObject) ⇒ <code>object</code>*
        * *[.parse(text, currentTime)](#module_cicero-core.TemplateInstance+parse)*
        * *[.generateText()](#module_cicero-core.TemplateInstance+generateText) ⇒
<code>string</code>*
        * *[.getIdentifier()](#module_cicero-core.TemplateInstance+getIdentifier) ⇒
<code>String</code>*
        * *[.getTemplate()](#module_cicero-core.TemplateInstance+getTemplate) ⇒
<code>Template</code>*
        * *[.getTemplateLogic()](#module_cicero-

core.TemplateInstance+getTemplateLogic) ⇒ <code>TemplateLogic</code>*
        * *[.toJSON()](#module_cicero-core.TemplateInstance+toJSON) ⇒
<code>object</code>*
    * _static_
        * *[.pad(n, width, z)](#module_cicero-core.TemplateInstance.pad) ⇒
<code>string</code>*
        * *[.convertDateTimes(obj, utcOffset)](#module_cicero-
core.TemplateInstance.convertDateTimes) ⇒ <code>*</code>*

<a name="new_module_cicero-core.TemplateInstance_new"></a>

#### *new TemplateInstance(template)*
Create the Clause and link it to a Template.

Param	Type	Description
template	<code>Template</code>	the template for the clause

<a name="module_cicero-core.TemplateInstance+setData"></a>

#### *templateInstance.setData(data)*
Set the data for the clause

Kind: instance method of [<code>TemplateInstance</code>](#module_cicero-
core.TemplateInstance)

Param	Type	Description
data	<code>object</code>	the data for the clause, must be an instance of the template model for the clause's template. This should be a plain JS object and will be deserialized and validated into the Composer object before assignment.

<a name="module_cicero-core.TemplateInstance+getData"></a>

#### *templateInstance.getData() ⇒ <code>object</code>*
Get the data for the clause. This is a plain JS object. To retrieve the Composer
object call getComposerData().

Kind: instance method of [<code>TemplateInstance</code>](#module_cicero-
core.TemplateInstance)
Returns: <code>object</code> - - the data for the clause, or null if it has not
been set
<a name="module_cicero-core.TemplateInstance+getDataAsComposerObject"></a>

#### *templateInstance.getDataAsComposerObject() ⇒ <code>object</code>*
Get the data for the clause. This is a Composer object. To retrieve the
plain JS object suitable for serialization call toJSON() and retrieve the `data`
property.

Kind: instance method of [<code>TemplateInstance</code>](#module_cicero-
core.TemplateInstance)
Returns: <code>object</code> - - the data for the clause, or null if it has not
been set
<a name="module_cicero-core.TemplateInstance+parse"></a>

#### *templateInstance.parse(text, currentTime)*
Set the data for the clause by parsing natural language text.

Kind: instance method of [<code>TemplateInstance</code>](#module_cicero-core.TemplateInstance)

Param	Type	Description
text	<code>string</code>	the data for the clause
currentTime	<code>string</code>	the definition of 'now' (optional)

<a name="module_cicero-core.TemplateInstance+generateText"></a>

#### *templateInstance.generateText() ⇒ <code>string</code>*
Generates the natural language text for a clause; combining the text from the template
and the clause data.

Kind: instance method of [<code>TemplateInstance</code>](#module_cicero-core.TemplateInstance)
Returns: <code>string</code> - the natural language text for the clause; created by combining the structure of
the template with the JSON data for the clause.
<a name="module_cicero-core.TemplateInstance+getIdentifier"></a>

#### *templateInstance.getIdentifier() ⇒ <code>String</code>*
Returns the identifier for this clause. The identifier is the identifier of
the template plus '-' plus a hash of the data for the clause (if set).

Kind: instance method of [<code>TemplateInstance</code>](#module_cicero-core.TemplateInstance)
Returns: <code>String</code> - the identifier of this clause
<a name="module_cicero-core.TemplateInstance+getTemplate"></a>

#### *templateInstance.getTemplate() ⇒ <code>Template</code>*
Returns the template for this clause

Kind: instance method of [<code>TemplateInstance</code>](#module_cicero-core.TemplateInstance)
Returns: <code>Template</code> - the template for this clause
<a name="module_cicero-core.TemplateInstance+getTemplateLogic"></a>

#### *templateInstance.getTemplateLogic() ⇒ <code>TemplateLogic</code>*
Returns the template logic for this clause

Kind: instance method of [<code>TemplateInstance</code>](#module_cicero-core.TemplateInstance)
Returns: <code>TemplateLogic</code> - the template for this clause
<a name="module_cicero-core.TemplateInstance+toJSON"></a>

#### *templateInstance.toJSON() ⇒ <code>object</code>*
Returns a JSON representation of the clause

Kind: instance method of [<code>TemplateInstance</code>](#module_cicero-core.TemplateInstance)
Returns: <code>object</code> - the JS object for serialization
<a name="module_cicero-core.TemplateInstance.pad"></a>

#### *TemplateInstance.pad(n, width, z) ⇒ <code>string</code>*
Left pads a number

Kind: static method of [<code>TemplateInstance</code>](#module_cicero-

core.TemplateInstance)
Returns: <code>string</code> - the left padded string

Param	Type	Default	Description
n	<code>*</code>		the number
width	<code>*</code>		the number of chars to pad to
z	<code>string</code>	<code>&quot;0&quot;</code>	the pad character

<a name="module_cicero-core.TemplateInstance.convertDateTimes"></a>

#### *TemplateInstance.convertDateTimes(obj, utcOffset) ⇒ <code>*</code>*
Recursive function that converts all instances of ParsedDateTime
to a Moment.

Kind: static method of [<code>TemplateInstance</code>](#module_cicero-
core.TemplateInstance)
Returns: <code>*</code> - the converted object

Param	Type	Description
obj	<code>*</code>	the input object
utcOffset	<code>number</code>	the default utcOffset

<a name="module_cicero-core.TemplateLoader"></a>

### *cicero-core.TemplateLoader*
A utility class to create templates from data sources.

Kind: static abstract class of [<code>cicero-core</code>](#module_cicero-core)
Interal:

* *[.TemplateLoader](#module_cicero-core.TemplateLoader)*
    * *[.loadZipFileContents(zip, path, json, required)](#module_cicero-
core.TemplateLoader.loadZipFileContents) ⇒ <code>Promise.&lt;string&gt;</code>*
    * *[.loadZipFilesContents(zip, regex)](#module_cicero-
core.TemplateLoader.loadZipFilesContents) ⇒
<code>Promise.&lt;Array.&lt;object&gt;&gt;</code>*
    * *[.loadFileContents(path, fileName, json, required)](#module_cicero-
core.TemplateLoader.loadFileContents) ⇒ <code>Promise.&lt;string&gt;</code>*
    * *[.loadFilesContents(path, regex)](#module_cicero-
core.TemplateLoader.loadFilesContents) ⇒
<code>Promise.&lt;Array.&lt;object&gt;&gt;</code>*
    * *[.fromArchive(Template, buffer)](#module_cicero-
core.TemplateLoader.fromArchive) ⇒ <code>Promise.&lt;Template&gt;</code>*
    * *[.fromUrl(Template, url, options)](#module_cicero-
core.TemplateLoader.fromUrl) ⇒ <code>Promise</code>*
    * *[.fromDirectory(Template, path, [options])](#module_cicero-
core.TemplateLoader.fromDirectory) ⇒ <code>Promise.&lt;Template&gt;</code>*

<a name="module_cicero-core.TemplateLoader.loadZipFileContents"></a>

#### *TemplateLoader.loadZipFileContents(zip, path, json, required) ⇒
<code>Promise.&lt;string&gt;</code>*
Loads a required file from the zip, displaying an error if missing

Kind: static method of [<code>TemplateLoader</code>](#module_cicero-
core.TemplateLoader)
Returns: <code>Promise.&lt;string&gt;</code> - a promise to the contents of the

zip file or null if it does not exist and
required is false
Internal:

Param	Type	Default	Description
zip	<code>*</code>		the JSZip instance
path	<code>string</code>		the file path within the zip
json	<code>boolean</code>	<code>false</code>	if true the file is converted
to a JS Object using JSON.parse
required	<code>boolean</code>	<code>false</code>	whether the file is
required

<a name="module_cicero-core.TemplateLoader.loadZipFilesContents"></a>

#### *TemplateLoader.loadZipFilesContents(zip, regex) ⇒
<code>Promise.&lt;Array.&lt;object&gt;&gt;</code>*
Loads the contents of all files in the zip that match a regex

Kind: static method of [<code>TemplateLoader</code>](#module_cicero-
core.TemplateLoader)
Returns: <code>Promise.&lt;Array.&lt;object&gt;&gt;</code> - a promise to an
array of objects with the name and contents of the zip files
Internal:

Param	Type	Description
zip	<code>*</code>	the JSZip instance
regex	<code>RegExp</code>	the regex to use to match files

<a name="module_cicero-core.TemplateLoader.loadFileContents"></a>

#### *TemplateLoader.loadFileContents(path, fileName, json, required) ⇒
<code>Promise.&lt;string&gt;</code>*
Loads a required file from a directory, displaying an error if missing

Kind: static method of [<code>TemplateLoader</code>](#module_cicero-
core.TemplateLoader)
Returns: <code>Promise.&lt;string&gt;</code> - a promise to the contents of the
file or null if it does not exist and
required is false
Internal:

Param	Type	Default	Description
path	<code>*</code>		the root path
fileName	<code>string</code>		the relative file name
json	<code>boolean</code>	<code>false</code>	if true the file is converted
to a JS Object using JSON.parse
required	<code>boolean</code>	<code>false</code>	whether the file is
required

<a name="module_cicero-core.TemplateLoader.loadFilesContents"></a>

#### *TemplateLoader.loadFilesContents(path, regex) ⇒
<code>Promise.&lt;Array.&lt;object&gt;&gt;</code>*
Loads the contents of all files under a path that match a regex
Note that any directories called node_modules are ignored.

```
**Kind**: static method of [<code>TemplateLoader</code>](#module_cicero-
core.TemplateLoader)
**Returns**: <code>Promise.&lt;Array.&lt;object&gt;&gt;</code> - a promise to an
array of objects with the name and contents of the files
**Internal**:

| Param | Type | Description |
| --- | --- | --- |
| path | <code>\*</code> | the file path |
| regex | <code>RegExp</code> | the regex to match files |

<a name="module_cicero-core.TemplateLoader.fromArchive"></a>

#### *TemplateLoader.fromArchive(Template, buffer) ⇒
<code>Promise.&lt;Template&gt;</code>*
Create a template from an archive.

**Kind**: static method of [<code>TemplateLoader</code>](#module_cicero-
core.TemplateLoader)
**Returns**: <code>Promise.&lt;Template&gt;</code> - a Promise to the template

| Param | Type | Description |
| --- | --- | --- |
| Template | <code>\*</code> | the type to construct |
| buffer | <code>Buffer</code> | the buffer to a Cicero Template Archive (cta) file
|

<a name="module_cicero-core.TemplateLoader.fromUrl"></a>

#### *TemplateLoader.fromUrl(Template, url, options) ⇒ <code>Promise</code>*
Create a template from an URL.

**Kind**: static method of [<code>TemplateLoader</code>](#module_cicero-
core.TemplateLoader)
**Returns**: <code>Promise</code> - a Promise to the template

| Param | Type | Description |
| --- | --- | --- |
| Template | <code>\*</code> | the type to construct |
| url | <code>String</code> | the URL to a Cicero Template Archive (cta) file |
| options | <code>object</code> | additional options |

<a name="module_cicero-core.TemplateLoader.fromDirectory"></a>

#### *TemplateLoader.fromDirectory(Template, path, [options]) ⇒
<code>Promise.&lt;Template&gt;</code>*
Builds a Template from the contents of a directory.
The directory must include a package.json in the root (used to specify
the name, version and description of the template).

**Kind**: static method of [<code>TemplateLoader</code>](#module_cicero-
core.TemplateLoader)
**Returns**: <code>Promise.&lt;Template&gt;</code> - a Promise to the instantiated
template

| Param | Type | Description |
| --- | --- | --- |
| Template | <code>\*</code> | the type to construct |
| path | <code>String</code> | to a local directory |
```

| [options] | <code>Object</code> | an optional set of options to configure the instance. |

<a name="module_cicero-core.TemplateSaver"></a>

### cicero-core.TemplateSaver
A utility to persist templates to data sources.

Kind: static class of [<code>cicero-core</code>](#module_cicero-core)
Internal:
<a name="module_cicero-core.TemplateSaver.toArchive"></a>

#### TemplateSaver.toArchive(template, [language], [options]) ⇒ <code>Promise.&lt;Buffer&gt;</code>
Persists this template to a Cicero Template Archive (cta) file.

Kind: static method of [<code>TemplateSaver</code>](#module_cicero-core.TemplateSaver)
Returns: <code>Promise.&lt;Buffer&gt;</code> - the zlib buffer

Param	Type	Description
template	<code>Template</code>	the template to persist
[language]	<code>string</code>	target language for the archive (should be 'ergo')
[options]	<code>Object</code>	JSZip options

---
id: version-0.12-cicero-cli
title: Cicero CLI
original_id: cicero-cli

Install the `@accordproject/cicero-cli` npm package to access the Cicero command line interface (CLI). After installation you can use the `cicero` command and its sub-commands as described below.

## cicero parse

Loads a template from a directory on disk and then parses input clause (or contract) text using the template.
If successful the template model is printed to console. If there are syntax errors in the DSL
text the line and column and error information are printed.

```bash
    cicero parse

    Options:
      --help         Show help
[boolean]
      --version      Show version number
[boolean]
      --template     path to the directory with the template
[string]
      --sample       path to the clause text
[string]
```

~~~~text
      --out          path to the output file
[string]
      --verbose, -v                                                 [default:
false]
```

## cicero execute

Loads a template from a directory on disk and then attempts to create a clause (or
contract)from a given input
text. If the clause (or contract) is successfully created, it is then executed by
the engine, passing in JSON data. If successful the
engine response is printed to the console.

```bash
   cicero execute

   Options:
     --help         Show help                                         [boolean]
     --version      Show version number                               [boolean]
     --template     path to the directory with the template            [string]
     --sample       path to the clause text                            [string]
     --request      path to the JSON request                            [array]
     --state        path to the JSON state                             [string]
     --verbose, -v                                           [default: false]
```

## cicero archive

Creates a Cicero Template Archive (.cta) file from a template stored in a local
directory.

```sh
    cicero archive

    Options:
      --help         Show help
[boolean]
      --version      Show version number
[boolean]
      --template     path to the directory with the template
[string]
      --archiveFile  file name for the archive
[string]
      --verbose, -v                                                 [default:
false]
```

## cicero generate

Loads a template from a directory on disk and then attempts to generate versions of
the template model in the specified format.
The available formats include: `Go`, `PlantUML`, `Typescript`, `Java`, and
`JSONSchema`.

```bash
   cicero generate

   Options:
~~~~

```
  --help              Show help                                         [boolean]
  --version           Show version number                               [boolean]
  --template          path to the directory with the template
                                                        [string] [default: "."]
  --format            format of the code to generate
                                               [string] [default: "JSONSchema"]
  --outputDirectory   output directory path       [string] [default: "./output/"]
  --verbose, -v                                                  [default: false]
```

---
id: version-0.12-ergo-api
title: Ergo API
original_id: ergo-api

## Classes

<dl>
<dt><a href="#Commands">Commands</a></dt>
<dd><p>Utility class that implements the commands exposed by the Ergo CLI.</p>
</dd>
</dl>

## Functions

<dl>
<dt><a href="#getJson">getJson(input)</a> ⇒ <code>object</code></dt>
<dd><p>Load a file or JSON string</p>
</dd>
<dt><a href="#setCurrentTime">setCurrentTime(currentTime)</a> ⇒
<code>object</code></dt>
<dd><p>Ensures there is a proper current time</p>
</dd>
<dt><a href="#init">init(engine, templateLogic, contractJson, currentTime)</a> ⇒
<code>object</code></dt>
<dd><p>Invoke Ergo contract initialization</p>
</dd>
<dt><a href="#execute">execute(engine, templateLogic, contractJson, stateJson,
currentTime, requestJson)</a> ⇒ <code>object</code></dt>
<dd><p>Execute the Ergo contract with a request</p>
</dd>
<dt><a href="#resolveRootDir">resolveRootDir(parameters)</a> ⇒
<code>string</code></dt>
<dd><p>Resolve the root directory</p>
</dd>
<dt><a href="#compareComponent">compareComponent(expected, actual)</a></dt>
<dd><p>Compare actual and expected result components</p>
</dd>
<dt><a href="#compareSuccess">compareSuccess(expected, actual)</a></dt>
<dd><p>Compare actual result and expected result</p>
</dd>
</dl>

<a name="Commands"></a>

## Commands
Utility class that implements the commands exposed by the Ergo CLI.

```
**Kind**: global class

* [Commands](#Commands)
    * [.execute(ergoPaths, ctoPaths, contractName, contractInput, stateInput,
currentTime, requestsInput)](#Commands.execute) ⇒ <code>object</code>
    * [.invoke(ergoPaths, ctoPaths, contractName, clauseName, contractInput,
stateInput, currentTime, paramsInput)](#Commands.invoke) ⇒ <code>object</code>
    * [.init(ergoPaths, ctoPaths, contractName, contractInput, currentTime,
paramsInput)](#Commands.init) ⇒ <code>object</code>
    * [.parseCTOtoFileSync(ctoPath)](#Commands.parseCTOtoFileSync) ⇒
<code>string</code>
    * [.parseCTOtoFile(ctoPath)](#Commands.parseCTOtoFile) ⇒ <code>string</code>

<a name="Commands.execute"></a>

### Commands.execute(ergoPaths, ctoPaths, contractName, contractInput, stateInput,
currentTime, requestsInput) ⇒ <code>object</code>
Execute an Ergo contract with a request

**Kind**: static method of [<code>Commands</code>](#Commands)
**Returns**: <code>object</code> - Promise to the result of execution

| Param | Type | Description |
| --- | --- | --- |
| ergoPaths | <code>Array.&lt;string&gt;</code> | paths to the Ergo modules |
| ctoPaths | <code>Array.&lt;string&gt;</code> | paths to CTO models |
| contractName | <code>string</code> | of the contract |
| contractInput | <code>string</code> | the contract data |
| stateInput | <code>string</code> | the contract state |
| currentTime | <code>string</code> | the definition of 'now' |
| requestsInput | <code>Array.&lt;string&gt;</code> | the requests |

<a name="Commands.invoke"></a>

### Commands.invoke(ergoPaths, ctoPaths, contractName, clauseName, contractInput,
stateInput, currentTime, paramsInput) ⇒ <code>object</code>
Invoke an Ergo contract's clause

**Kind**: static method of [<code>Commands</code>](#Commands)
**Returns**: <code>object</code> - Promise to the result of invocation

| Param | Type | Description |
| --- | --- | --- |
| ergoPaths | <code>Array.&lt;string&gt;</code> | paths to the Ergo modules |
| ctoPaths | <code>Array.&lt;string&gt;</code> | paths to CTO models |
| contractName | <code>string</code> | the contract |
| clauseName | <code>string</code> | the name of the clause to invoke |
| contractInput | <code>string</code> | the contract data |
| stateInput | <code>string</code> | the contract state |
| currentTime | <code>string</code> | the definition of 'now' |
| paramsInput | <code>object</code> | the parameters for the clause |

<a name="Commands.init"></a>

### Commands.init(ergoPaths, ctoPaths, contractName, contractInput, currentTime,
paramsInput) ⇒ <code>object</code>
Invoke init for an Ergo contract
```

Kind: static method of [<code>Commands</code>](#Commands)
Returns: <code>object</code> - Promise to the result of execution

Param	Type	Description
ergoPaths	<code>Array.&lt;string&gt;</code>	paths to the Ergo modules
ctoPaths	<code>Array.&lt;string&gt;</code>	paths to CTO models
contractName	<code>string</code>	the contract name
contractInput	<code>string</code>	the contract data
currentTime	<code>string</code>	the definition of 'now'
paramsInput	<code>object</code>	the parameters for the clause

<a name="Commands.parseCTOtoFileSync"></a>

### Commands.parseCTOtoFileSync(ctoPath) ⇒ <code>string</code>
Parse CTO to JSON File

Kind: static method of [<code>Commands</code>](#Commands)
Returns: <code>string</code> - The name of the generated CTOJ model file

Param	Type	Description
ctoPath	<code>string</code>	path to CTO model file

<a name="Commands.parseCTOtoFile"></a>

### Commands.parseCTOtoFile(ctoPath) ⇒ <code>string</code>
Parse CTO to JSON File

Kind: static method of [<code>Commands</code>](#Commands)
Returns: <code>string</code> - The name of the generated CTOJ model file

Param	Type	Description
ctoPath	<code>string</code>	path to CTO model file

<a name="getJson"></a>

## getJson(input) ⇒ <code>object</code>
Load a file or JSON string

Kind: global function
Returns: <code>object</code> - JSON object

Param	Type	Description
input	<code>object</code>	either a file name or a json string

<a name="setCurrentTime"></a>

## setCurrentTime(currentTime) ⇒ <code>object</code>
Ensures there is a proper current time

Kind: global function
Returns: <code>object</code> - if valid, the moment object for the current time

Param	Type	Description
currentTime	<code>string</code>	the definition of 'now'

```
<a name="init"></a>

## init(engine, templateLogic, contractJson, currentTime) ⇒ <code>object</code>
Invoke Ergo contract initialization

**Kind**: global function
**Returns**: <code>object</code> - Promise to the initial state of the contract

| Param | Type | Description |
| --- | --- | --- |
| engine | <code>object</code> | the execution engine |
| templateLogic | <code>object</code> | the Template Logic |
| contractJson | <code>object</code> | contract data in JSON |
| currentTime | <code>string</code> | the definition of 'now' |

<a name="execute"></a>

## execute(engine, templateLogic, contractJson, stateJson, currentTime,
requestJson) ⇒ <code>object</code>
Execute the Ergo contract with a request

**Kind**: global function
**Returns**: <code>object</code> - Promise to the response

| Param | Type | Description |
| --- | --- | --- |
| engine | <code>object</code> | the execution engine |
| templateLogic | <code>object</code> | the Template Logic |
| contractJson | <code>object</code> | contract data in JSON |
| stateJson | <code>object</code> | state data in JSON |
| currentTime | <code>string</code> | the definition of 'now' |
| requestJson | <code>object</code> | state data in JSON |

<a name="resolveRootDir"></a>

## resolveRootDir(parameters) ⇒ <code>string</code>
Resolve the root directory

**Kind**: global function
**Returns**: <code>string</code> - root directory used to resolve file names

| Param | Type | Description |
| --- | --- | --- |
| parameters | <code>string</code> | Cucumber's World parameters |

<a name="compareComponent"></a>

## compareComponent(expected, actual)
Compare actual and expected result components

**Kind**: global function

| Param | Type | Description |
| --- | --- | --- |
| expected | <code>string</code> | the expected component as specified in the test
workload |
| actual | <code>string</code> | the actual component as returned by the engine |
```

<a name="compareSuccess"></a>

## compareSuccess(expected, actual)
Compare actual result and expected result

Kind: global function

Param	Type	Description
expected	<code>string</code>	the expected successful result as specified in the test workload
actual	<code>string</code>	the successful result as returned by the engine

----
---
id: version-0.12-ergo-cli
title: Ergo CLI
original_id: ergo-cli

To install the Ergo command-line interface (CLI):

```term
    npm install -g @accordproject/ergo-cli@0.12
```

This will install `ergoc`, the Ergo compiler, `ergorun` to run your contracts locally on your machine, and `ergotop` which is a _read-eval-print-loop_ utility to write Ergo interactively.

## ergoc

### Usage

Compile an Ergo contract to a target platform

```term
    ergoc [options] [cto files] [ergo files]

    Options:
      --version       Show version and exit
      --target <lang> Target language (default: es6) (available: es5,es6,cicero,java)
      --link          Adds the Ergo runtime to the target code (for es5,es6 and cicero only)
      --monitor       Produce compilation time information
      --help          Show help and exit
```

`ergoc` takes your input models (cto files) and input contracts (ergo files) and generates code for execution. By default it generates JavaScript code (ES6 compliant).

### Examples

For instance, to compile the helloworld contract to JavaScript:

```term
```

~~~~
    $ ergoc ./examples/volumediscount/model.cto
./examples/volumediscount/logic.ergo
    Compiling Ergo './examples/volumediscount/logic.ergo' -- creating
'./examples/volumediscount/logic.js'
```

To compile the helloworld contract to JavaScript and link the Ergo runtime for
execution:

```term
    $ ergoc ./examples/volumediscount/model.cto
./examples/volumediscount/logic.ergo --link
    Compiling Ergo './examples/volumediscount/logic.ergo' -- creating
'./examples/volumediscount/logic.js'
```

To compile the helloworld contract to Java:

```term
    $ ergoc ./examples/volumediscount/model.cto
./examples/volumediscount/logic.ergo --target java
    Compiling Ergo './examples/volumediscount/logic.ergo' -- creating
'./examples/volumediscount/logic.java'
```

## ergorun

### Usage

Invoke an ergo contract

```term
    ergorun --contract [file] --state [file] --request [file] [ctos] [ergos]

    Options:
      --help          Show help
[boolean]
      --version       Show version number
[boolean]
      --contract      path to the contract data
[required]
      --request       path to the request data                   [array]
[required]
      --state         path to the state data                [string] [default:
null]
      --contractname
[required]
      --verbose, -v                                                 [default:
false]
```

`ergorun` lets you invoke your Ergo contract. You need to pass the CTO and Ergo
files, the name of the contract that you want to execute, and JSON files for: the
contract parameters, the current state of the contract, and for the request.

### Examples

For instance, to send one request to the `volumediscount` contract:
~~~~

```term
    $ ergorun ./examples/volumediscount/model.cto
./examples/volumediscount/logic.ergo --contractname
org.accordproject.volumediscount.VolumeDiscount --contract
./examples/volumediscount/contract.json --request
./examples/volumediscount/request.json --state ./examples/volumediscount/state.json
    02:33:50 - info: {"response":
{"discountRate":2.8,"$class":"org.accordproject.volumediscount.VolumeDiscountRespon
se"},"state":
{"$class":"org.accordproject.cicero.contract.AccordContractState","stateId":"1"},"e
mit":[]}
```

If contract invokation is successful, `ergorun` will print out the response, the new contract state and any emitted events.

## ergotop (REPL)

### Starting the REPL

`ergotop` is a convenient tool to try-out Ergo contracts in an interactive way. You can write commands, or expressions and see the result. It is often called the Ergo REPL, for _read-eval-print-loop_, since it literally: reads your input Ergo from the command-line, evaluates it, prints the result and loops back to read your next input.

To start the REPL:

```
    $ ergotop
    02:39:37 - info: Logging initialized. 2018-09-25T06:39:37.209Z
    ergo$
```

It should print the prompt `ergo$` which indicates it is ready to read your command. For instance:

```ergo
    ergo$ return 42
    Response. 42 : Integer
```

`ergotop` prints back both the resulting value and its type. You can then keep typing commands:

```ergo
    ergo$ return "hello " ++ "world!"
    Response. "hello world!" : String
    ergo$ define constant pi = 3.14
    ergo$ return pi ^ 2.0
    Response. 9.8596 : Double
```

If your expression is not valid, or not well-typed, it will return an error:

```ergo
    ergo$ return if true else "hello"
    Parse error (at line 1 col 15).
    return if true else "hello"
```

```
                   ^^^^
    ergo$ return if "hello" then 1 else 2
    Type error (at line 1 col 10). 'if' condition not boolean.
    return if "hello" then true else false
              ^^^^^^^
```

If what you are trying to write is too long to fit on one line, you can use `\` to go to a new line:

```ergo
    ergo$ define function squares(l:Double[]) : Double[] { \
      ...   return \
      ...     foreach x in l return x * x \
      ... }
    ergo$ return squares([2.3,4.5,6.7])
    Response. [5.29, 20.25, 44.89] : Double[]
```

### Loading files

You can load CTO and Ergo files to use in your REPL session. Once the REPL is launched you will have to import the corresponding namespace. For instance, if you want to use the `compoundInterestMultiple` function defined in the `./examples/promissory-note/money.ergo` file, you can do it as follows:

```ergo
$ ergotop ./examples/promissory-note/money.ergo
08:45:26 - info: Logging initialized. 2018-09-25T12:45:26.481Z
ergo$ import org.accordproject.ergo.money.*
ergo$ return compoundInterestMultiple(0.035, 100.0)
Response. 1.00946960405 : Double
ergo$
```

### Calling contracts

To call a contract, you first needs to _instantiate_ it, which means setting its parameters and initializing its state. You can do this by using the `set contract` and `call init` commands respectively. Here is an example using the `volumediscount` template:

```ergo
$ ergotop ./examples/volumediscount/model.cto ./examples/volumediscount/logic.ergo
ergo$ import org.accordproject.cicero.contract.*
ergo$ import org.accordproject.cicero.runtime.*
ergo$ import org.accordproject.volumediscount.*
ergo$ set contract VolumeDiscount over TemplateModel{ \
  ...   firstVolume: 1.0, \
  ...   secondVolume: 10.0, \
  ...   firstRate: 3.0, \
  ...   secondRate: 2.9, \
  ...   thirdRate: 2.8 \
  ... }
ergo$ call init(Request{})
Response. unit : Unit
State. AccordContractState{stateId: "1"} : AccordContractState
```

You can then invoke clauses of the contract:

```ergo
ergo$ call volumediscount(VolumeDiscountRequest{ netAnnualChargeVolume : 0.1 })
Response. VolumeDiscountResponse{discountRate: 3.0} : VolumeDiscountResponse
ergo$ call volumediscount(VolumeDiscountRequest{ netAnnualChargeVolume : 10.5 })
Response. VolumeDiscountResponse{discountRate: 2.8} : VolumeDiscountResponse
```

You can also invoke the contract without explicitely naming the clause by simply sending a request. The Ergo engine dispatches that request to the first clause which can handle it:
```ergo
ergo$ send VolumeDiscountRequest{ netAnnualChargeVolume : 0.1 }
Response. VolumeDiscountResponse{discountRate: 3.0} : VolumeDiscountResponse
ergo$ send VolumeDiscountRequest{ netAnnualChargeVolume : 10.5 }
Response. VolumeDiscountResponse{discountRate: 2.8} : VolumeDiscountResponse
```

---
id: version-0.12-ergo-tutorial
title: Ergo: A Tutorial
original_id: ergo-tutorial

## Overview of Accord

Cicero is an Open Source implementation of the Accord Project Template Specification. It defines the structure of natural language templates, bound to a data model, that can be executed using Ergo and request/response JSON messages. You can read the latest user documentation here: http://docs.accordproject.org.

In short with the Accord Project you can take a classic contract, e.g. Word document and use Cicero to define natural language contract and clause templates that can be executed by an event driven computer program (aka Smart contract). For the tutorial, Cicero will be used to define natural language contract and clause templates. These clause templates handle the syllogistic language of contracts.

For example,
```md
 if the goods are more than [{DAYS}] late,
 then notify the supplier of the goods, with the message [{MESSAGE}].
```
DAYS and MESSAGE are variables

You can browse the library of Open Source Cicero contract and clause templates at: https://templates.accordproject.org.

So how goes the contract get executed? That is where Ergo comes in Ergo is a strongly-typed functional programming language designed to capture the legal intent of legal contracts and clauses. We will use Ergo to create the contract logic consisting of a contract class with executable embedded clauses. Note: prior to the emergence of Ergo, the Cicero JavaScript component was primary to the execution of code.

Ergo obviates the Cicero JavaScript component for the execution phase with a new more comprehensive language which we explore in this tutorial.

## Cicero

The Open Source Cicero project defines the format of clause and contract templates based on to the Cicero Template Specification. The templates are the link between the natural language of contracts usually composed in a Word document and the specification of a machine executable transaction. Cicero templates define the API by specifying request and response elements for the logic associated with functional transaction executed by Ergo.

Cicero templates are composed of two elements:
* Template Grammar (the natural language text for the template),
* Template Model (the data model that includes the variables contained within the template).
* The Logic (the executable business logic for the template) will be handled by Ergo.

When combined these three elements allow templates to be edited, analyzed, queried and executed.

## Setup Ergo Development environment

Before you can build Ergo, you must install and configure the following dependencies on your machine:

### Git

* Git: The [Github Guide to Installing Git][git-setup] is a good source of information.

### Node.js

* Node.js v8.x (LTS): We use Node to generate the documentation, run a development web server, run tests, and generate distributable files. Depending on your system, you can install Node either from source or as a pre-packaged bundle.
> Tip: Use nvm (or nvm-windows) to manage and install Node.js, This facilitates a version change of Node.js per project.
* Lerna: This is a tool which helps when handling multiple npm packages in the Ergo repository. To install:
npm install -g lerna@2.11.0

### Visual Studio Code

Visual Studio Code is a lightweight but powerful source code editor which runs on your desktop and is available for Windows, macOS and Linux. It comes with built-in support for JavaScript and Node.js and has a rich ecosystem of extensions for other languages (such Ergo).

Follow the platform specific guides below:
See, https://code.visualstudio.com/docs/setup/
* macOS
* Linux
* Windows

#### Install Ergo VisualStudio Plugin

### Validate Development Environment and Toolset

Clone https://github.com/accordproject/ergo to your local machine

### Getting started

Install Ergo

The easiest way to install Ergo is as a Node.js package. Once you have Node.js installed on your machine, you can get the Ergo compiler and command-line using the Node.js package manager by typing the following in a terminal:
$ npm install -g @accordproject/ergo-cli@0.12

This will install the compiler itself (ergoc) and a command-line tool (ergo) to execute Ergo code. You can check that both have been installed and print the version number by typing the following in a terminal:
```sh
$ ergoc --version
$ ergo --version
```
Then, to get command line help:
```
$ ergoc --help
$ ergo execute --help
```
Compiling your first contract
```ergo
namespace org.accordproject.volumediscount

contract VolumeDiscount over VolumeDiscountContract {
   // Clause for volume discount
   clause volumediscount(request : VolumeDiscountRequest) : VolumeDiscountResponse
{
        if request.netAnnualChargeVolume < contract.firstVolume
        then return VolumeDiscountResponse{ discountRate: contract.firstRate }
        else if request.netAnnualChargeVolume < contract.secondVolume
        then return VolumeDiscountResponse{ discountRate: contract.secondRate }
        else return VolumeDiscountResponse{ discountRate: contract.thirdRate }
  }
}
```

To compile your first Ergo contract to JavaScript , within Visual Studio code
* Open the folder where you cloned https://github.com/accordproject/ergo
* Use View/Terminal to run the Ergo compiler:
```sh
$ ergoc ./examples/volumediscount/model.cto ./examples/volumediscount/logic.ergo
Compiling Ergo './examples/volumediscount/logic.ergo' -- creating
'./examples/volumediscount/logic.js'
```

By default, Ergo compiles to JavaScript for execution. This may change in the future to support other languages. The compiled code for the result in stored as `./examples/volumediscount/logic.js`

### Execute a contract
To execute a contract, we pass the necessary parameters including the CTO, Ergo files, the name of a contract and the json files containing request and contract state
ergorun [ctos] [ergos] --contractname [file] --contract [file] --state [file] --request [file]

So for example we use ergorun with :
```sh
$ ergorun ./examples/volumediscount/model.cto ./examples/volumediscount/logic.ergo
--contractname org.accordproject.volumediscount.VolumeDiscount
--contract ./examples/volumediscount/contract.json
--request ./examples/volumediscount/request.json
--state ./examples/volumediscount/state.json
```

Here contract.json contains the following values
```json
{
  "$class": "org.accordproject.volumediscount.VolumeDiscountContract",
  "parties": null,
  "contractId": "cr1",
  "firstVolume": 1,
  "secondVolume": 10,
  "firstRate": 3,
  "secondRate": 2.9,
  "thirdRate": 2.8
}
```

Request.json contains
```json
{
  "$class": "org.accordproject.volumediscount.VolumeDiscountRequest",
  "netAnnualChargeVolume": 10.4
}
```

logic.ergo contains:
```ergo
namespace org.accordproject.volumediscount

contract VolumeDiscount over VolumeDiscountContract {
  // Clause for volume discount
  clause volumediscount(request : VolumeDiscountRequest) : VolumeDiscountResponse {
    if request.netAnnualChargeVolume < contract.firstVolume
    then return VolumeDiscountResponse{ discountRate: contract.firstRate }
    else if request.netAnnualChargeVolume < contract.secondVolume
    then return VolumeDiscountResponse{ discountRate: contract.secondRate }
    else return VolumeDiscountResponse{ discountRate: contract.thirdRate }
  }
}
```

Here netAnnualCharge Volume equals 10.4 which is not less than firstVolume and secondVolume which are equal to 1 and 10 respectively so the logic for the volumediscount clause returns thirdRate which equals 2.8

```
7:31:58 PM - info: Logging initialized. 2018-09-27T23:31:58.623Z
7:31:59 PM - info: {"response":
{"discountRate":2.8,"$class":"org.accordproject.volumediscount.VolumeDiscountRespon
se"},"state":
{"$class":"org.accordproject.cicero.contract.AccordContractState","stateId":"1"},"e
mit":[]}
```

PS D:\Users\jbambara\Github\ergo>

## Ergo Development

Create Template
Start with basic agreement in natural language and locate the variables
Here in the example see the bold
Volume-Based Card Acceptance Agreement [Abbreviated]
This Agreement is by and between ……….you agree to be bound by the Agreement.
Discount means an amount that we charge you for accepting the Card, which amount is:
(i) a percentage (Discount Rate) of the face amount of the Charge that you submit, or a flat per-
Transaction fee, or a combination of both; and/or
(ii) a Monthly Flat Fee (if you meet our requirements).

Transaction Processing and Payments. ………………… less all applicable deductions, rejections, and withholdings, which include:
………………………….

SETTLEMENT
a) Settlement Amount. Our agent will pay you according to your payment plan, ……………………..which include:
(i) the Discount,
…………………………………………..
b) Discount. The Discount is determined according to the following table:

Annual Dollar Volume	Discount
Less than $1 million	3.00%
$1 million to $10 million	2.90%
Greater than $10 million	2.80%

Identify the request variables and contract instance variables
Codify the variables with $[{request}] or [{contract instance}]

Annual Dollar Volume	Discount
Less than $[{firstVolume}] million	[{firstRate}]%
$[{firstVolume}] million to $[{secondVolume}] million	[{secondRate}]%
Greater than $[{secondVolume}] million	[{thirdRate}]%

Create Model
Define the model asset which contains the contract instance variables and the transaction request and response. Defines the data model for the VolumeDiscount template. This defines the structure that the parser for the template generates from input source text. See model.cto below:

```
 namespace org.accordproject.volumediscount
import org.accordproject.cicero.contract.* from
https://models.accordproject.org/cicero/contract.cto
import org.accordproject.cicero.runtime.* from
https://models.accordproject.org/cicero/runtime.cto
asset VolumeDiscountContract extends AccordContract {
  o Double firstVolume
  o Double secondVolume
  o Double firstRate
  o Double secondRate
  o Double thirdRate
}
transaction VolumeDiscountRequest {
  o Double netAnnualChargeVolume
}
transaction VolumeDiscountResponse {
        o Double discountRate
}
```

Create Logic
The contract logic is accomplished by coding ERGO statements and expressions to consume the request and use contract instance variables to produce the desired response. In our example, request.netAnnualChargeVolume is tested against contract rates to produce the result.

```
namespace org.accordproject.volumediscount

define the contract
contract VolumeDiscount over VolumeDiscountContract {

define the contract clause and request : response

   clause volumediscount(request : VolumeDiscountRequest) : VolumeDiscountResponse
{

define the logic ; here we use if /then /else statement to test request parameter
against contract instance variable
 and return

        if request.netAnnualChargeVolume < contract.firstVolume
        then return VolumeDiscountResponse{ discountRate: contract.firstRate }
        else if request.netAnnualChargeVolume < contract.secondVolume
        then return VolumeDiscountResponse{ discountRate: contract.secondRate }
        else return VolumeDiscountResponse{ discountRate: contract.thirdRate }
  }
```

Ergo Language
As you have seen in this tutorial, Ergo is a domain-specific language (DSL) that captures the execution logic of legal contracts. In this simple example, you see that Ergo aims to have contracts and clauses as first-class elements of the language. To accommodate the maturation of distributed ledger implementations, Ergo will be blockchain neutral, i.e., the same contract logic can be executed either on and off chain on distributed ledger technologies like HyperLedger Fabric. Most importantly, Ergo is consistent with the Accord Protocol Template Specification.
Follow the links below to learn more about
Introduction to Ergo
Ergo Language Guide
Ergo Reference Guide

October 12, 2018

---
id: version-0.12-example-eatapple
title: A Healthy Clause
original_id: example-eatapple

## Eat Apples!

The healthy eating clause is inspired by the not so serious [terms of services contract](https://www.grahamcluley.com/page-46-apples-new-ios-agreement-funny-fake-makes-serious-point/).

For this example, let us look first at the template for that legal clause written
in natural language:

```markdown
Eating healthy clause between [{employee}] (the Employee) and [{company}] (the
Company).
The canteen only sells apple products. Apples, apple juice, apple flapjacks, toffee
apples. Employee gets fired if caught eating anything without apples in it.
THE EMPLOYEE, IF ALLERGIC TO APPLES, SHALL ALWAYS BE HUNGRY.
Apple products at the canteen are subject to a [{tax}]% tax.
```

The text specify the terms for the legal clause, and includes a few
variables such as `employee`, `company` and `tax`.

The second component of a smart legal template is the model, which is
expressed using the [Composer Concerto modeling
language](https://github.com/hyperledger/composer-concerto). The model
describe the variables of the contract, as well as additional
information required to execute the contract logic. In our example,
this includes the input request for the clause (`Food`), the response
to that request (`Outcome`) and possible events emitted during the
clause execution (`Bill`).

```ergo
namespace org.accordproject.canteen

@AccordTemplateModel("eat-apples")
concept CanteenContract {
  o String employee
  o String company
  o Double tax
}

transaction Food {
  o String produce
  o Double price
}

transaction Outcome {
  o String notice
}

event Bill {
  o String billTo
  o Double amount
}
```

The last component of a smart legal template is the Ergo logic. In our example it
is a single clause `eathealthy` which can be used to process a `Food` request.

```ergo
namespace org.accordproject.canteen

contract EatApples over CanteenContract {
  clause eathealthy(request : Food) : Outcome {
    enforce request.produce = "apple"
```

~~~~
    else return Outcome{ notice : "You're fired!" };

    emit Bill{
      billTo: contract.employee,
      amount: request.price * (1.0 + contract.tax / 100.0)
    };
    return Outcome{ notice : "Very healthy!" }
  }
}
```

As in the "Hello World!" example, this is a smart legal contract with
a single clause, but it illustrate a few new ideas.

The `enforce` expression is used to check conditions that must be true
for normal execution of the clause. In this example, the `enforce`
makes sure the food is an apple and if not returns a new outcome
indicating termination of employment.

If the condition is true, the contract proceeds by emitting a bill for
the purchase of the apple. The employee to be billed is obtained from
the contract (`contract.employee`). The total amount is calculated by
adding the tax, which is obtained from the contract (`contract.tax`),
to the purchase price, which is obtained from the request
(`request.price`). The calculation is done using a simple arithmetic
expression (`request.price * (1.0 + contract.tax / 100.0)`).


--------------------------------------------------------------------------------
---
id: version-0.12-logic-advanced-expr
title: Advanced Expressions
original_id: logic-advanced-expr
---

## Match

Match expressions allow to check an expression against multiple possible
values:

```ergo
    match fruitcode
      with 1 then "Apple"
      with 2 then "Apricot"
      else "Strange Fruit"
```

## Foreach

Foreach expressions allow to apply an expression of every element in
an input array of values and returns a new array:

```ergo
  foreach x in [1.0,-2.0,3.0] return x + 1.0
```

Foreach expressions can have an optional condition of the values being
iterated over:
~~~~

```ergo
  foreach x in [1.0,-2.0,3.0] where x > 0.0 return x + 1.0
```

---
id: version-0.12-logic-complex-type
title: Complex Values & Types
original_id: logic-complex-type

So far we only considered atomic values and types, such as string values or integers, which are not sufficient for most contracts. In Ergo, values and types are based on the [Composer Concerto Modeling Language](https://github.com/hyperledger/composer-concerto) (often referred to as CTO files). This provides a rich vocabulary to define the parameters of your contract, the information associated to contract participants, the structure of contract obligation, etc.

In Ergo, you can either import an existing CTO file or declare types directly within your code. Let us look at the different kinds of types you can define and how to create values with those types.

## Arrays

Array types lets you define collections of values and are denoted with `[]` after the type of elements in that collection:

```ergo
    String[]                         // a String array
    Double[]                         // a Double array
```

You can write arrays as follows:
```ergo
    ["pear","apple","strawberries"]  // an array of String values
    [3.14,2.72,1.62]                 // an array of Double values
```

You can construct arrays using other expressions:
```ergo
    let pi = 3.14;
    let e = 2.72;
    let golden = 1.62;
    [pi,e,golden]
```

Ergo also provides functions to manipulate arrays as parts of its [standard library](ergo-stdlib.html#functions-on-arrays):
```ergo
    let pi = 3.14;
    let e = 2.72;
    let golden = 1.62;
    let prettynumbers : Double[] = [pi,e,golden];
    sum(prettynumbers)
```

## Classes

You can declare classes in the Composer Modeling Language (concepts, transactions, events, participants or assets) by importing them from a CTO file or directly within your Ergo program:

```ergo
   define concept Seminar {
     name : String,
     fee : Double
   }
   define asset Product {
     id : String
   }
   define asset Car extends Product {
     range : String
   }
   define transaction Response {
     rate : Double,
     penalty : Double
   }
  define event PaymentObligation{
    amount : Double,
    description : String
  }
```

Once a class type has been defined, you can create an instance of that type using the class name along with the values for each fields:

```ergo
    Seminar{
      name: "Law for developers",
      fee: 29.99
    }
    Car{
      id: "Batmobile4156",
      range: "Unknown"
    }
```

> **TechNote:** When extending an existing class (e.g., `Car extends Product`), the sub-class includes the fields from the super-class. So `Car` includes the field `range` which is locally declared and the field `id` which is declared in `Product`.

You can access the field of a class using the `.` operator:
```ergo
    Seminar{
      name: "Law for developers",
      fee: 29.99
    }.fee                        // Returns 29.99
```

## Records

Sometimes it is convenient to declare a structure without having to declare it first. You can do that using a record, which is similar to a class but without its name:

```ergo
   {
     name : String,  // A record with a name of type String
     fee : Double    // and a fee of type Double
   }
```

You do not need to declare that record, and can directly write an instance of that record as follows:

```ergo
    {
      name: "Law for developers",
      fee: 29.99
    }
```

> Typing `return { name: "Law for developers", fee: 29.99 }` in the [Ergo REPL](https://ergorepl.netlify.com), should answer `Response. {name: "Law for developers", fee: 29.99} : {fee: Double, name: String}`.

You can access the field of a record using the `.` operator:
```ergo
    {
      name: "Law for developers",
      fee: 29.99
    }.fee                        // Returns 29.99
```
## Enums

Here is how to declare an enumerated type:

```ergo
define enum ProductType {
   DAIRY,
   BEEF,
   VEGETABLES
}
```

> **TechNote:** Enumerated types are handled as `String` at the moment.

To create an instance of that enum:
```ergo
"DAIRY"
"BEEF"
```

## Optional types

An optional type can contain a value or not and is indicated with a `?`.

```ergo
Integer?           // An optional integer
PaymentObligation // An optional payment obligation
Double[]?          // An optional array of doubles
```

A an optional value can be either present, written `some(v)`, or absent, written `none`.

```ergo
let i1 : Integer? = some(1); i1
let i2 : Integer? = none; i2
```

To operate on an optional type, you need to say what to do when the value is present and what to do when the value is not present. You can do that with a match statement:

This example:
```ergo
match some(1)
with let? x then "I found 1 :-)"
else "I found nothing :-("
```
should return `"I found 1 :-)"`.

This example:
```
match none
with let? x then "I found 1 :-)"
else "I found nothing :-("
```
should return `"I found nothing :-("`.

For conciseness, a few operators are also available on optional values. One can give a default value when the optional is `none` using the operator `??`. For instance:

```ergo
some(1) ?? 0       // Returns the integer 1
none ?? 0          // Returns the integer 0
```

You can also access the field inside an optional concept or an optional record using the operator `?.`. For instance:

```ergo
some({a:1})?.a     // Returns the optional value: some(1)
none?.a            // Returns the optional value: none
```

---
id: version-0.12-logic-decl
title: Declarations
original_id: logic-decl

Now that we have values, types, expressions and statements available, we can start writing more complex Ergo logic using by declaring functions, clauses and contracts.

## Constants and functions

It is possible to declare global constants and functions in Ergo:

```ergo
    define constant pi = 3.1416
    define function area(radius : Double) : Double {
      pi * r * r
    }
    area(1.5)
```

Global variables can also be declared with a type, and the return type of functions can be omitted:

```ergo
    define constant pi : Double = 3.1416
```

The return type of functions can be omitted:

```ergo
    define function area(radius : Double) {
      pi * r * r
    }
```

## Clauses

In Ergo, a logical clause like the example clause noted below is represented as a “function” (akin to a “method” in languages like Java) that resides within its parent contract (akin to a “class” in a language like Java).

> Functions are "self contained" modules of code that accomplish a specific task. Functions usually "take in" data, process it, and "return" a result. Once a function is written, it is reusable , i.e., it can be used over and over and over again.
> Functions can be "called" from within other functions or from a clause.
> Functions have to be declared before they can be used. So functions "encapsulate" a task. They combine statements and expressions carried out as instructions which accomplish a specific task to allow their execution using a single line of code. Most programming languages provide libraries of built in functions (i.e., commonly used tasks like computing the square root of a number).
> Functions accelerate development and facilitate the reuse of code which performs common tasks.

The declaration of a Clause that contains the clause’s name, request type and return type collectively referred to as the ‘signature’ of the function.

### Example Prose

Additionally the Equipment should have proper devices on it to record any shock during transportation as any instance of acceleration outside the bounds of -0.5g and 0.5g. Each shock shall reduce the Contract Price by $5.00

### Syntax

```ergo
    clause fragileGoods(request : DeliveryUpdate) : ContractPrice {
        ... // A statement computing the clause response
    }
```

Inside a contract, the `contract` variable contains the instance of the template model for the current contract.

## Contract Declarations

The legal requirements for a valid contract at law vary by jurisdiction and contract type. The requisite elements that typically necessary for the formation of a legally binding contract are (1) _offer_; (2) _acceptance_; (3) _consideration_; (4) _mutuality of obligation_; (5) _competency and capacity_; and, in certain circumstances, (6) _a written instrument_.

Ergo contacts address consideration, mutuality of obligation, competency and capacity through statements that are described in this document.

Furthermore, an Ergo contract is an immutable written document which obviates a good deal of the issues and conflicts which emerge from existing contracts in use today. In Ergo, a contract:
- represents an agreement between parties creating mutual and enforceable obligations; and
- is a code module that uses conditionals and functions to describe execution by the parties with their obligations. Contracts accept input data either directly from the associated natural language text or through request _transactions_. The contract then uses _clause functions_ to process it, and _return_ a result.
Once a contract logic has been written within a template, it can be used over and over and over again.

Instantiated contracts correspond to particular domain agreement. They combine functions and clauses to execute a specific agreement and to allow its automation. Many traditional contracts are “boilerplate” and as such are reusable in their specific legal domain, e.g., sale of goods.

You can declare a contract over a template model as follows. The `TemplateModel` is the data model for the parameters of the contract text.

```ergo
    contract ContractName over TemplateModel {
      clause C1(request : ReqType1) : RespType1 {
        // Statement
      }

      clause C2(request : ReqType2) : RespType2 {
        // Statement
      }
    }
```

## Contract State and Obligations

If your contract requires a state, or emits only certain kinds of obligations but not other, you can specify the corresponding types when declaring your contract:

```ergo
    contract ContractName over TemplateModel state MyState {
      clause C1(request : ReqType1) : RespType1 emits MyObligation {
        // Statement
      }

      clause C2(request : ReqType2) : RespType2 {
        // Statement
      }
    }
```

The state is always declared for the whole contract, while obligations can be declared individually for each clause.

---
id: version-0.12-logic-ergo
title: Overview
original_id: logic-ergo

## Language Goals

Ergo aims to:
- have contracts and clauses as first-class elements of the language
- help legal-tech developers quickly and safely write computable legal contracts
- be modular, facilitating reuse of existing contract or clause logic
- ensure safe execution: the language should prevent run-time errors and non-terminating logic
- be blockchain neutral: the same contract logic can be executed either on and off chain on a variety of distributed ledger technologies
- be formally specified: the meaning of contracts should be well defined so it can be verified, and preserved during execution
- be consistent with the [Accord Project Template Specification](accordproject-specification)

## Design Choices

To achieve those goals the design of Ergo is based on the following principles:

- Ergo contracts have a class-like structure with clauses akin to methods
- Ergo can handle types (concepts, transations, etc) defined with the [Composer Concerto Modeling Language](https://github.com/hyperledger/composer-concerto) (so called CML models), as mandated by the Accord Project Template Specification
- Ergo borrows from strongly-typed functional programming languages: clauses have a well-defined type signature (input and output), they are functions without side effects
- The compiler guarantees error-free execution for well-typed Ergo programs
- Clauses and functions are written in an expression language with limited expressiveness (it allows conditional and bounded iteration)
- Most of the compiler is written in Coq as a stepping stone for formal specification and verification

## Status

- The current implementation is considered *in development*, we welcome contributions (be it bug reports, suggestions for new features or improvements, or pull requests)
- The currently compiler targets JavaScript (either standalone or for use in Cicero Templates and Hyperledger Fabric) and Java (experimental)

## This Guide

Ergo provides a simple expression language to describe computation. From those expressions one can write functions, clauses, and then whole contract logic. This guide explains most of the Ergo concepts starting from simple expressions all the way to contracts.

Ergo is a _strongly typed_ language, which means it checks that the expressions you use are consistent (e.g., you can take the square root of `3.14` but not of `"pi!"`). The type system is here to help you write better and safer contract logic, but it also takes a little getting used to. This page also introduces Ergo types and how to work with them.

---
id: version-0.12-logic-module
title: Modularity
original_id: logic-module

Finally, we can place multiple Ergo declarations (functions, contracts, etc) into a library so it can be shared with other developers.

## Namespaces

Each Ergo file starts with a namespace declaration which provides a way to identify it uniquely:
```ergo
    namespace org.acme.mynamespace
```

## Libraries

A library is simply an Ergo file in a namespace which defines useful constants or functions. For instance:

```ergo
    namespace org.accordproject.ergo.money

    define constant days_in_a_year = 365.0
    define function compoundInterests(
      annualInterest : Double,
      numberOfDays : Double
    ) : Double {
        return (1.0 + annualInterest) ^ (numberOfDays / days_in_a_year)
    }
```

## Import

You can then access this library in another Ergo file using import:
```ergo
namespace org.accordproject.promissorynote

import org.accordproject.cicero.runtime.*
import org.accordproject.time.*           // Imports the DateTime library
import org.accordproject.ergo.money.*     // Imports the money.ergo library

contract PromissoryNote over PromissoryNoteContract {
  clause check(request : Payment) : Result {
```

    let interestRate = contract.interestRate ?? 3.4;
    let outstanding = contract.amount.doubleValue - request.amountPaid.doubleValue;

    let numberOfDays =
      diffDurationAs(dateTime("17 May 2018 13:53:33
EST"),contract.date,"days").amount;
    let compounded =
      outstanding
    * compoundInterests(interestRate, // Calls compoundInterests from the library!
                        numberOfDays);

    return Result{
      outstandingBalance: compounded
    }
  }
}

> **TechNote:** the namespace and import handling in Ergo allows you to access either existing CTO models or Ergo libraries in the same way.

---
id: version-0.12-logic-simple-expr
title: Simple Expressions
original_id: logic-simple-expr

## Literal values

The simplest kind of expressions in Ergo are literal values.

```ergo
    "John Smith" // a String literal
    1            // an Integer literal
    3.0          // a Double literal
    3.5e-10      // another Double literal
    true         // the Boolean true
    false        // the Boolean false
```

Each line here is a separate expression. At the end of the line, the notation `// write something here` is a _comment_, which means it is a part of your Ergo program which is ignored by the Ergo compiler. It can be useful to document your code.

Every Ergo expression can be _evaluated_, which means it should compute some value. In the case of a literal value, the result of evaluation is simply itself (e.g., the expression `1` evaluates to the integer `1`).

> You can actually see the result of evaluating expressions by trying them out in the [Ergo REPL](https://ergorepl.netlify.com). You just have to prefix them with `return`: for instance, to evaluate the String literal `"John Smith"` type: `return "John Smith"` (followed by clicking the button 'Evaluate') in the REPL. This should answer: `Response. "John Smith" : String`.

## Operators

You can apply operators to values. Those can be used for arithmetic operations, to compare two values, to concatenate two string values, etc.

```ergo
    1.0 + 2.0 * 3.0      // arithmetic operators on Double
    -1.0
    1 + 2 * 3            // arithmetic operators on Integer
    -1

    1.0 <= 3.0           // comparison operators on Double
    1.0 = 2.0
    2.0 > 1.0
    1 <= 3               // comparison operators on Integer
    1 = 2
    2 > 1.0

    true or false        // Boolean disjunction
    true and false       // Boolean conjunction
    !true                // Negation

    "Hello" ++ " World!" // String concatenation
```

> Again, you can try those in the [Ergo REPL](https://ergorepl.netlify.com). For instance, typing `return true and false` should answer `Response. false : Boolean`, and typing `return 1.0 + 2.0 * 3.0` should answer: `Response. 7.0 : Double`.

## Conditional expressions

Conditional expressions can be used to perform different computations depending on some condition:

```ergo
    if 1.0 < 0.0     // Condition
    then "negative"  // Expression if condition is true
    else "positive"  // Expression if condition is false
```

> Typing `return if 1.0 < 0.0 then "negative" else "positive"` in the [Ergo REPL](https://ergorepl.netlify.com), should answer `Response. "positive" : String`.

See also the [Conditional Expression Reference](ergo-reference.html#condition-expressions)

## Let bindings

Local variables can be declared with `let`:

```ergo
    let x = 1;              // declares and initialize a variable
    x+2                     // rest of the expression, where x is in scope
```

Let bindings give a name to some intermediate result and allows you to reuse the corresponding value in multiple places:

```ergo
   let x = -1.0;            // bind x to the value -1.0
   if x < 0.0               // if x is negative
   then -x                  // then return the opposite of x
   else x                  // else return x
```

> **TechNote:** let bindings in Ergo are immutable, in a way similar to other functional languages. A nice explanation can be found e.g., in the documentation for let bindings in [ReasonML](https://reasonml.github.io/docs/en/let-binding).

---
id: version-0.12-logic-simple-type
title: Introducing Types
original_id: logic-simple-type

We have so far talked about types only informally. When we wrote earlier:
```ergo
    "John Smith" // a String literal
    1            // an Integer literal
    ...
```
the comments mention that `"John Smith"` is of type `String`, and that `1` is of type `Integer`.

In reality, the Ergo compiler understands which types your expressions have and can detect whether those expressions apply to the right kinds of values or not.

## Atomic types

The simplest of types are atomic types which describe the various kinds of atomic values allowed in Ergo. Those atomic types are:

```ergo
    Boolean
    String
    Double
    Integer
    DateTime
```

## Type errors

The Ergo compiler understand types and can detect type errors when you write expressions. For instance, if you write: `1.0 + 2.0 * 3.0`, the Ergo compiler checks that the parameters for the operators `+` and `*` are indeed of type `Double`.

If you write `1.0 + 2.0 * "some text"` the Ergo compiler will detect that `"some text"` is of type `String`, which is not of the right type for the operator `*` and return an error.

> Typing `return 1.0 + 2.0 * "some text"` in the [Ergo REPL](https://ergorepl.netlify.com), should answer a type error:
> ```text
> Type error (at line 1 col 13). Operator * expected operands of
> type Double and Double but received operands of type Double and String.
> return 1.0 + 2.0 * "some text"
>              ^^^^^^^^^^^^^^^^^
> ```

## Type annotations

In a let bindings, you can also use a _type annotation_ to indicate which type you
expect it to have.

```ergo
    let name : String = "John"; // declares and initialize a string variable
    name ++ " Smith"            // rest of the expression
```
or
```ergo
    let x : Double = 3.1416     // declares and initialize a double variable
    sqrt(x)                     // rest of the expression
```

This can be useful to document your code, or to remember what type you expect from
an expression.

Again, the Ergo compiler will return a type error if the annotation is not
consistent with the expression that computes the value for that let binding. For
instance, the following will return a type error since `"pi!"` is not of type
`Double`.

```ergo
    let x : Double = "pi!"; // TYPE ERROR: "pi!" is not a Double
    sqrt(x)
```

> Typing `return let x : Double = "pi!"; sqrt(x)` in the [Ergo
REPL](https://ergorepl.netlify.com), should answer a type error:
> ```text
> Type error (at line 1 col 7). The let type annotation Double for
> the name x does not match the actual type String.
> return let x : Double = "pi!"; sqrt(x)
>        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
> ```

This becomes particularly useful as your code becomes more complex. For instance
the following expression will also trigger a type error:

```ergo
    let rate = 3.5;
    let name : String =
      if rate > 0.0
      then 3.14         // TYPE ERROR: 3.14 is not a String
      else "John";
    name ++ " Smith"
```

Since not all the cases of the `if ... then ... else ...` expressions return a
value of type `String` which is the type annotation for `name`.

---
id: version-0.12-logic-stmt
title: Statements
original_id: logic-stmt

A clause's body is composed of statements. Statements are a special kind of expression which can manipulate the contract state and emit obligations. Unlike other expressions they may return a response or an error.

## Contract data

When inside a statement, data about the contract -- either the contract parameters, clause parameters or contract state are available using the following Ergo keywords:
```ergo
    contract   // The contract parameters (from a contract template)
    clause     // Local clause parameters (from a clause template)
    state      // The contract state
```

For instance, if your contract template parameters and state information:
```ergo
    // Template parameters
    asset InstallmentSaleContract extends AccordContract {
      o AccordParty BUYER
      o AccordParty SELLER
      o Double INITIAL_DUE
      o Double INTEREST_RATE
      o Double TOTAL_DUE_BEFORE_CLOSING
      o Double MIN_PAYMENT
      o Double DUE_AT_CLOSING
      o Integer FIRST_MONTH
    }
    // Contract state
    enum ContractStatus {
      o WaitingForFirstDayOfNextMonth
      o Fulfilled
    }
    asset InstallmentSaleState extends AccordContractState {
      o ContractStatus status
      o Double balance_remaining
      o Integer next_payment_month
      o Double total_paid
   }
```

You can use the following expressions:
```ergo
    contract.BUYER
    state.balance_remaining
```

## Returning a response

Returning a response from a clause can be done by using a `return` statement:

```ergo
     return 1                        // Return the integer one
     return Payout{ amount: 39.99 } // Return a new Payout object
     return                          // Return nothing
```

> **TechNote:** the [Ergo REPL](https://ergorepl.netlify.com) takes statements as input which is why we had to add `return` to expressions in previous examples.

## Returning a failure

Returning a failure from a clause can be done by using a `throw` statement:
```ergo
    throw ErgoErrorResponse{ message: "This is wrong" }
    define concept MyOwnError extends ErgoErrorResponse{ fee: Double }
    throw MyOwnError{ message: "This is wrong and costs a fee", fee: 29.99 }
```

For convenience, Ergo provides a `failure` function which takes a string as part of its [standard library](ergo-stdlib.html#other-functions), so you can also write:
```ergo
    throw failure("This is wrong")
```

## Enforce statement

Before a contract is enforceable some preconditions must be satisfied:
- Competent parties who have the legal capacity to contract
- Lawful subject matter
- Mutuality of obligation
- Consideration

The constructs below will be used to determine if the preconditions have been met and what actions to take if they are not

```test
Example Prose
    Do the parties have adequate funds to execute this contract?
```

One can check preconditions in a clause using enforce statements, as follows:

```ergo
    enforce x >= 0.0                      // Condition
    else throw "Something went wrong"; // Statement if condition is false
    return x+1.0                          // Statement if condition is true
```

The else part of the statement can be omitted in which case Ergo returns an error by default.

```ergo
    enforce x >= 0.0;          // Condition
    return x+1.0               // Statement if condition is true
```

## Emitting obligations

When inside a clause or contract, you can emit (one or more) obligations as follows:
```ergo
   emit PaymentObligation{ amount: 29.99, description: "12 red roses" };
   emit PaymentObligation{ amount: 19.99, description: "12 white tulips" };
   return
```

Note that `emit` is always terminated by a `;` followed by another statement.

## Setting the contract state

When inside a clause or contract, you can change the contract state as follows:
```ergo
    set state InstallmentSaleState{
      stateId: "#1",
      status: "WaitingForFirstDayOfNextMonth",
      balance_remaining: contract.INITIAL_DUE,
      total_paid: 0.0,
      next_payment_month: contract.FIRST_MONTH
    };
    return
```

Note that `set state` is always terminated by a `;` followed by another statement.

---
id: version-0.12-ref-logic-specification
title: Ergo Compiler
original_id: ref-logic-specification

A large part of the Ergo compiler is written as a Coq specification
from which the compiler is extracted.

Ultimately, one of our goals is to provide a full formal semantics for
Ergo in Coq, and prove correct as much of the compilation pipeline as
possible.

## Compiler architecture

### Frontend

![Frontend](/docs/assets/architecture/frontend.svg)

### Code generation

![Codegen](/docs/assets/architecture/codegen.svg)

## Code Overview

The Coq source serves a dual purpose: as Ergo's formal semantics and as part of its
implementation through extraction. Here are some entry points to the code.

A browsable version of the Coq code (generated using
[coq2html](https://github.com/xavierleroy/coq2html)) is
available. Below are some of the main intermediate represntations and
compilation phases.

### Intermediate representations

- Ergo: [Ergo/Lang/Ergo](assets/specification/ErgoSpec.Ergo.Lang.Ergo.html)
- Ergo calculus:

[ErgoC/Lang/ErgoC](assets/specification/ErgoSpec.ErgoC.Lang.ErgoC.html)
- Ergo NNRC (Named Nested Relational Calculus):
[ErgoNNRC/Lang/ErgoNNRC](assets/specification/ErgoSpec.ErgoNNRC.Lang.ErgoNNRC.html)

### Macro expansion

- Ergo to Ergo:
[Ergo/Lang/ErgoExpand](assets/specification/ErgoSpec.Ergo.Lang.ErgoExpand.html)

### Namespace resolution

- Ergo to Ergo:
[Translation/ErgoNameResolve](assets/specification/ErgoSpec.Translation.ErgoNameRes
olve.html)

### Translations between intermediate representations

- Ergo to Ergo calculus:
[Translation/ErgotoErgoC](assets/specification/ErgoSpec.Translation.ErgotoErgoC.htm
l)
- ErgoC to Ergo NNRC:
[Translation/ErgoCtoErgoNNRC](assets/specification/ErgoSpec.Translation.ErgoCtoErgo
NNRC.html)

### Type checking

- ErgoC to ErgoC with types:
[ErgoCType](assets/specification/ErgoSpec.ErgoC.Lang.ErgoCType.html)

---
id: version-0.12-ref-logic-stdlib
title: Ergo Libraries
original_id: ref-logic-stdlib

The following libraries are provided with the Ergo compiler.

## Stdlib

The following functions are in the `org.accordproject.ergo.stdlib` namespace and
available by default.

### Functions on Integer

Name	Signature	Description
`integerAbs`	`(x:Integer) : Integer`	Absolute value
`integerLog2`	`(x:Integer) : Integer`	Base 2 integer logarithm
`integerSqrt`	`(x:Integer) : Integer`	Integer square root
`integerToDouble`	`(x:Integer) : Double`	Cast to a Double
`integerMod`	`(x:Integer, y:Integer) : Integer`	Integer remainder
`integerMin`	`(x:Integer, y:Integer) : Integer`	Smallest of `x` and `y`
`integerMax`	`(x:Integer, y:Integer) : Integer`	Largest of `x` and `y`

### Functions on Double

| Name | Signature | Description |

`abs`	`(x:Double) : Double`	Absolute value
`sqrt`	`(x:Double) : Double`	Square root
`exp`	`(x:Double) : Double`	Exponential
`log`	`(x:Double) : Double`	Natural logarithm
`log10`	`(x:Double) : Double`	Base 10 logarithm
`ceil`	`(x:Double) : Double`	Round to closest integer above
`floor`	`(x:Double) : Double`	Round to closest integer below
`truncate`	`(x:Double) : Integer`	Cast to an Integer
`doubleToInteger`	`(x:Double) : Integer`	Same as `truncate`
`minPair`	`(x:Double, y:Double) : Double`	Smallest of `x` and `y`
`maxPair`	`(x:Double, y:Double) : Double`	Largest of `x` and `y`

### Functions on Arrays

Name	Signature	Description
`count`	(x:Any[]) : Integer	Number of elements
`flatten`	(x:Any[][]) : Any[]	Flattens a nested array
`arrayAdd`	`(x:Any[],y:Any[]) : Any[]`	Array concatenation
`arraySubtract`	`(x:Any[],y:Any[]) : Any[]`	Removes elements of `y` in `x`
`inArray`	`(x:Any,y:Any[]) : Boolean`	Whether `x` is in `y`
`containsAll`	`(x:Any[],y:Any[]) : Boolean`	Whether all elements of `y` are in `x`
`distinct`	`(x:Any[]) : Any[]`	Duplicates elimination

Note: For most of these functions, the type-checker infers more precise types than indicated here. For instance `concat([1,2],[3,4])` will return `[1,2,3,4]` and have the type `Integer[]`.

### Aggregate functions

Name	Signature	Description
`max`	(x:Double[]) : Double	The largest element in `x`
`min`	(x:Double[]) : Double	The smallest element in `x`
`sum`	(x:Double[]) : Double	Sum of the elements in `x`
`average`	(x:Double[]) : Double	Arithmetic mean

### Math functions

Name	Signature	Description
`acos`	(x:Double) : Double	The inverse cosine of x
`asin`	(x:Double) : Double	The inverse sine of x
`atan`	(x:Double) : Double	The inverse tangent of x
`atan2`	(x:Double, y:Double) : Double	The inverse tangent of `x / y`
`cos`	(x:Double) : Double	The cosine of x
`cosh`	(x:Double) : Double	The hyperbolic cosine of x
`sin`	(x:Double) : Double	The sine of x
`sinh`	(x:Double) : Double	The hyperbolic sine of x
`tan`	(x:Double) : Double	The tangent of x
`tanh`	(x:Double) : Double	The hyperbolic tangent of x

### Other functions

Name	Signature	Description
`toString`	`(x:Any) : String`	Prints any value to a string

`failure`	`(x:String) : ErgoErrorResponse`	Ergo error from a string

## Time

The following functions are in the `org.accordproject.time` namespace and are available by importing that namespace.
They rely on the [time.cto](https://models.accordproject.org/v2.0/time.html) types from the Accord Project models.

### Functions on DateTime

Name	Signature	Description
`now`	`() : DateTime`	Returns the time when execution started
`dateTime`	`(x:String) : DateTime`	Parse a date and time
`getSecond`	`(x:DateTime) : Long`	Second component of a DateTime
`getMinute`	`(x:DateTime) : Long`	Minute component of a DateTime
`getHour`	`(x:DateTime) : Long`	Hour component of a DateTime
`getDay`	`(x:DateTime) : Long`	Day of the month component of a DateTime
`getWeek`	`(x:DateTime) : Long`	Week of the year component of a DateTime
`getMonth`	`(x:DateTime) : Long`	Month component in a DateTime
`getYear`	`(x:DateTime) : Long`	Year component in a DateTime
`isAfter`	`(x:DateTime, y:DateTime) : Boolean`	Whether `x` if after `y`
`isBefore`	`(x:DateTime, y:DateTime) : Boolean`	Whether `x` is before `y`
`isSame`	`(x:DateTime, y:DateTime) : Boolean`	Whether `x` is the same DateTime as `y`
`dateTimeMin`	`(x:DateTime[]) : DateTime`	The earliest in an array of DateTime
`dateTimeMax`	`(x:DateTime[]) : DateTime`	The latest in an array of DateTime

### Functions on Duration

Name	Signature	Description
`durationAs`	`(x:Duration, y:TemporalUnit) : Duration`	Change the unit for duration `x` to `y`
`diffDurationAs`	`(x:DateTime, y:DateTime, z:TemporalUnit) : Duration`	Duration between `x` and `y` in unit `z`
`diffDuration`	`(x:DateTime, y:DateTime) : Duration`	Duration between `x` and `y` in seconds
`addDuration`	`(x:DateTime, y:Duration) : DateTime`	Add duration `y` to `x`
`subtractDuration`	`(x:DateTime, y:Duration) : DateTime`	Subtract duration `y` to `x`
`divideDuration`	`(x:Duration, y:Duration) : Double`	Ratio between durations `x` and `y`

### Functions on Period

Name	Signature	Description
`diffPeriodAs`	`(x:DateTime, y:DateTime, z:PeriodUnit) : Period`	Time period between `x` and `y` in unit `z`
`addPeriod`	`(x:DateTime, y:Period) : DateTime`	Add time period `y` to `x`
`subtractPeriod`	`(x:DateTime, y:Period) : DateTime`	Subtract time period `y` to `x`
`startOf`	`(x:DateTime, y:PeriodUnit) : DateTime`	Start of period `y` nearest to DateTime `x`
`endOf`	`(x:DateTime, y:PeriodUnit) : DateTime`	End of period `y` nearest to

DateTime `x` |

---
id: version-0.12-ref-logic
title: Ergo Language Reference
original_id: ref-logic

## Lexical conventions

### File Extension

Ergo files have the ``.ergo`` extension.

### Blanks

Blank characters (such as space, tabulation, carriage return) are
ignored but they are used to separate identifiers.

### Comments

Comments come in two forms. Single line comments are introduced by the
two characters `//` and are terminated by the end of the current
line. Multi-line comments start with the two characters `/*` and are
terminated by the two characters `*/`. Multi-line comments can be
nested.

Here are examples of comments:

```ergo
    // This is a single line comment
    /* This comment spans multiple lines
       and it can also be /* nested */ */
```

### Reserved words

The following are reserved as keywords in Ergo. They cannot be used as identifiers.

```text
namespace, import, define, function, transaction, concept, event, asset,
participant, enum, extends, contract, over, clause, throws, emits, state, call,
enforce, if, then, else, let, foreach, return, in, where, throw,
constant, match, set, emit, with, or, and, true, false, unit, none
```

## Condition Expressions

Conditional statements, conditional expressions and conditional constructs are
features of a programming language which perform different computations or actions
depending on whether a programmer-specified boolean condition evaluates to true or
false. Conditional expressions (also known as `if` statements) allow us to
conditionally execute Ergo code depending on the value of a test condition. If the
test condition evaluates to `true` then the code on the `then` branch is evaluated.
Otherwise, when the test condition evaluates to `false` then the `else` branch is
evaluated.

### Example

```ergo
if delayInDays > 15.0 then
  BuyerMayTerminateResponse{};
else
  BuyerMayNotTerminateResponse{}
```

### Legal Prose

For example, this corresponds to a conditional logic statement in legal
prose.

    If the delay is more than 15 days, the Buyer is entitled to terminate this
Contract.

### Syntax

```ergo
    if expression1 then     // Condition
      expression2           // Expression if condition is true
    else
      expression3           // Expression if condition is false
```

Where `expression1` is an Ergo expression that evaluates to a Boolean
value (i.e. `true` or `false`), and `expression2` and `expression3` are
Ergo expressions.

> Note that as with all Ergo expressions, new lines and indentation
> don't change the meaning of your code. However it is good practice to
> standardise the way that you using whitespace in your code to make it
> easier to read.

### Usage

If statements can be chained , i.e., `if ... then .... else if ... then ...
else ...` to build more compound conditionals.

```ergo
if request.netAnnualChargeVolume < contract.firstVolume then
  return VolumeDiscountResponse{ discountRate: contract.firstRate }
else if request.netAnnualChargeVolume < contract.secondVolume then
  return VolumeDiscountResponse{ discountRate: contract.secondRate }
else
  return VolumeDiscountResponse{ discountRate: contract.thirdRate }
```

Conditional expressions can also be used as expressions, e.g., inside a constant
declaration:

```ergo
define constant price = 42;
define constant message =  if price >i 100 then "High price" else "Low Price";
message;
```

The value of message after running this code will be `"Low Price"`.

### Related

-   [Match expression](ergo-reference#match-expressions) - where many
    alternative conditions check the same variable

## Match Expressions

Match expressions allow us to check an expression against multiple
possible values or patterns. If a match is found, then Ergo will
evaluate the corresponding expression.

> Match expressions are similar to `switch` statements in other
> programming languages

### Example

```ergo
match request.status
  with "CREATED" then
    new PayOut{ amount : contract.deliveryPrice }
  with "ARRIVED" then
    new PayOut{ amount : contract.deliveryPrice - shockPenalty }
  else
    new PayOut{ amount : 0.0 }
```

### Legal Prose

> Example needed.

### Syntax

```ergo
match expression0
  with pattern1 then       // Repeat this line
    expression1            //    and this line
  else
    expression2
```

### Usage

You can use a match expression to look for patterns based on the type of
an expression.

```ergo
match response
    with let b1 : BuyerMayTerminateResponse then
        // Do something with b1
    with let b2 : BuyerMayNotTerminateResponse then
        // Do something with b2
    else
        // Do a default action
```

You can use it to match against an optional value.

```ergo
```

~~~~text
match maybe_response
    with let? b1 : BuyerMayTerminateResponse then
        // Do something when there is a response
    else
        // Do something else when there is no response
```

Often a match expression is a more concise way to represent a
conditional expression with a repeating, regular condition. For example:

```ergo
    if x = 1 then
      ...
    else if x = 2 then
      ...
    else if x = 3 then
      ...
    else if x = 4 then
      ...
    else
      ...
```

This is equivalent to the match expression:

```ergo
    match x
      with 1 then
        ...
      with 2 then
        ...
      with 3 then
        ...
      with 4 then
        ...
      else
        ...
```

--------------------------------------------------------------------------------
---
id: version-0.12-ref-markup
title: Markup Reference
original_id: ref-markup
---

## Markup Syntax

A template is UTF-8 text with markup to introduce named variables. Each variable
starts with `[{` and ends with `}]`. There are three kinds of variables:
- Standard variables
- Formatted variables
- Boolean variables

### Standard Variables

Standard variables are written `[{variableName}]` where `variableName` is a
variable declared in the model.
~~~~

The following example shows a template text with three variables (`buyer`,
`amount`, and `seller`):

```md
Upon the signing of this Agreement, [{buyer}] shall pay [{amount}] to [{seller}].
```

### Formatted Variables

Formatted variables are written `[{variableName as "FORMAT"}]` where `variableName`
is a variable declared in the model and the `FORMAT` is a type-dependent
description for the syntax of the variables in the contract.

The following example shows a template text with one variable with a format
`DD/MM/YYYY`.

```md
The contract was signed on [{contractDate as "DD/MM/YYYY"}].
```

### Boolean Variables

Boolean variables are written `[{"OPTIONALTEXT":? variableName}] where
`variableName` is a variable declared in the model and `OPTIONALTEXT` is text that
is optionally present in the contract.

## Instance Text

A template defines the set of clauses or contracts instances which are valid
against that template. An instance is UTF-8 text without markup, where variables
have been replaced by values.

Except for variables, the instance text has to exactly match the text in the
template. The syntax of a value in the instance text for each variable
`variableName` depends on the type of that variable.

## String Variable

### Description

If the variable `variableName` has type `String` in the model:
```ergo
o String variableName
```
The corresponding instance should contain text between quotes (`"`).

### Examples

For example, consider the following model:

```md
asset Template extends AccordClause {
  o String buyer
  o String supplier
}
```

the following instance text:
```md
```

~~~~
This Supply Sales Agreement is made between "Steve Supplier" and "Betty Byer".
```

matches the template:
```md
This Supply Sales Agreement is made between [{supplier}] and [{buyer}].
```

while the following instance texts do not match:
```md
This Supply Sales Agreement is made between 2019 and 2020.
```
or
```md
This Supply Sales Agreement is made between Steve Supplier and Betty Byer.
```

## Numeric Variable

### Description

If the variable `variableName` has type `Double`, `Integer` or `Long` in the model:
```ergo
o Double variableName
o Integer variableName2
o Long variableName3
```
The corresponding instance should contain the corresponding number.

### Examples

For example, consider the following model:

```md
asset Template extends AccordClause {
  o Double penaltyPercentage
}
```

the following instance text:
```md
The penalty amount is 10.5% of the total value of the Equipment whose delivery has
been delayed.
```

matches the template:
```md
The penalty amount is [{penaltyPercentage}]% of the total value of the Equipment
whose delivery has been delayed.
```

while the following instance texts do not match:
```md
The penalty amount is ten% of the total value of the Equipment whose delivery has
been delayed.
```
or
```md
The penalty amount is "10.5"% of the total value of the Equipment whose delivery
~~~~

has been delayed.

## DateTime Variables

### Description

If the variable `variableName` has type `DateTime`:
```ergo
o DateTime variableName
```
The corresponding instance should contain the corresponding date using the format `MM/DD/YYYY`, commonly used in the US.

### DateTime Formats

DateTime format can be customized inline in a template grammar by including an optional format string using the `as` keyword. The following formatting tokens are supported:

Tokens are case-sensitive.

```md
Input         Example.           Description

YYYY          2014               4 or 2 digit year
YY            14                 2 digit year
M             12                 1 or 2 digit month number
MM            04                 2 digit month number
MMM           Feb.               Short month name
MMMM          December           Long month name
D             3                  1 or 2 digit day of month
DD            04                 2 digit day of month
H             3                  1 or 2 digit hours
HH            04                 2 digit hours
mm            59                 2 digit minutes
ss            34                 2 digit seconds
SSS           002                3 digit milliseconds
Z             +01:00             UTC offset
```

> Note that if `Z` is specified, it must occur as the last token in the format string.

### Examples

The format of the `contractDate` variable of type `DateTime` can be specified with the `DD/MM/YYYY` format, as is commonly used in Europe.

```md
The contract was signed on [{contractDate as "DD/MM/YYYY"}].
The contract was signed on 26/04/2019.
```

Other examples:

```md
dateTimeProperty: [{dateTimeProperty as "D MMM YYYY HH:mm:ss.SSSZ"}]
dateTimeProperty: 1 Jan 2018 05:15:20.123+01:02
```

~~~~
```

```md
dateTimeProperty: [{dateTimeProperty as "D MMMM YYYY HH:mm:ss.SSSZ"}]
dateTimeProperty: 1 January 2018 05:15:20.123+01:02
```

```md
dateTimeProperty: [{dateTimeProperty as "D-M-YYYY H mm:ss.SSSZ"}]
dateTimeProperty: 31-12-2019 2 59:01.001+01:01
```

```md
dateTimeProperty: [{dateTimeProperty as "DD/MM/YYYY"}]
dateTimeProperty: 01/12/2018
```

```md
dateTimeProperty: [{dateTimeProperty as "DD-MMM-YYYY H mm:ss.SSSZ"}]
dateTimeProperty: 04-Jan-2019 2 59:01.001+01:01
```

## Duration Variables

### Description

If the variable `variableName` has type `Duration`:
```ergo
import org.accordproject.time.Duration
o Duration variableName
```

The corresponding instance should contain the corresponding duration written with
the amount as a number and the duration unit as literal text.

### Examples

For example, consider the following model:
```md
asset Template extends AccordClause {
  o Duration termination
}
```

the following instance texts:
```md
If the delay is more than 15 days, the Buyer is entitled to terminate this
Contract.
```
and
```md
If the delay is more than 1 week, the Buyer is entitled to terminate this Contract.
```

both match the template:
```md
If the delay is more than [{termination}], the Buyer is entitled to terminate this
Contract.
```
~~~~

while the following instance texts do not match:
```md
If the delay is more than a month, the Buyer is entitled to terminate this
Contract.
```
or
```md
If the delay is more than "two weeks", the Buyer is entitled to terminate this
Contract.
```

## Enumerated Variables

### Decription

If the variable `variableName` has an enumerated type:
```ergo
o EnumType variableName
```

The corresponding instance should contain a corresponding enumerated value without
quotes.

### Examples

For example, consider the following model:
```
import org.accordproject.money.CurrencyCode from
https://models.accordproject.org/money.cto
asset Template extends AccordClause {
  o CurrencyCode currency
}

```
the following instance text:
```md
Monetary amounts in this contract are denominated in USD.
```

matches the template:
```md
Monetary amounts in this contract are denominated in [{currency}].
```

while the following instance texts do not match:
```md
Monetary amounts in this contract are denominated in "USD".
```
or
```md
Monetary amounts in this contract are denominated in $.
```

## Complex Variables

### Description

If the variable `variableName` has a complex type `ComplextType` (such as an

~~~
`asset`, a `concept`, etc.)
```ergo
o ComplextType variableName
```

The corresponding instance should contain all fields in the corresponding complex
type in the order they occur in the model, separated by a single white space
character.

### Examples

For example, consider the following model:
```md
import org.accordproject.address.PostalAddress from
https://models.accordproject.org/address.cto
asset Template extends AccordClause {
  o PostalAddress address
}
```

the following instance text:
```md
Address of the supplier: "555 main street" "10290" "" "NY" "New York" "10001".
```

matches the template:
```md
Address of the supplier: [{address}].
```

Consider the following model:
```md
import org.accordproject.money.MonetaryAmount from
https://models.accordproject.org/money.cto
asset Template extends AccordClause {
  o MonetaryAmount amount
}
```

the following instance text:
```md
Total value of the goods: 50 USD.
```

matches the template:
```md
Total value of the goods: [{amount}].
```


--------------------------------------------------------------------------------
---
id: version-0.12-ref-testing
title: Testing Reference
original_id: ref-testing
---

Cicero uses [Cucumber](https://cucumber.io/docs) for writing template tests, which
provides a human readable syntax.
~~~

This documents the syntax available to write Cicero tests.

## Test Structure

Tests are located in the `./test/` directory for each template, which contains files with the `.feature` extension.

Each file has the following structure:

```gherkin
Feature: Name of the template being tested
  Description for the test

  Background:
    Given that the contract says
"""
Text of the contract instance.
"""

  Scenario: Description for scenario 1
    [[First Scenario Sequence]]

  Scenario: Description for scenario 2
    [[Second Scenario Sequence]]

etc.
```

Each scenario can be thought of as a description for the behavior of the clause or contract template for the contract given as background.

Each scenario corresponds to one call to the contract. I.e., for a given current time, request and contract state, it says what the expected result of executing the contract should be. This can be either:
- A response, a new contract state, and a list of emitted obligations
- An error

If you are not familiar with the execution model for Accord Project contract, we recommend reading through the corresponding part of the [Template Specification](spec-concepts#step-7-execute-and-return-response), including the Section on [Contract Execution](spec-execution).

## Scenarios

A complete scenario is described in the [Gherkin Syntax](https://cucumber.io/docs/gherkin/reference/) through a sequence of **Step**.

Each step starts with a keyword, either **Given**, **When**, **And**, or **Then**:

- **Given**, **When** and **And** are used to specify the input for a call to the contract;
- **Then** and **And** are used to specify the expected result.

### Request and Response

The simplest kind of scenario specifies the response expected for a given request.

For instance, the following scenario describe the expected response for a given request to the [helloworld template](https://templates.accordproject.org/helloworld@0.10.1.html):

```gherkin
  Scenario: The contract should say Hello to Betty Buyer, from the ACME Corporation
    When it receives the request
"""
{
    "$class": "org.accordproject.helloworld.MyRequest",
    "input": "ACME Corporation"
}
"""
    Then it should respond with
"""
{
    "$class": "org.accordproject.helloworld.MyResponse",
    "output": "Hello Betty Buyer ACME Corporation"
}
"""
```

Both the request and the response are written inside triple quotes `"""` using JSON. If the request or response is not valid wrt. to the data model, this will result in a failing test.

:::warning
While the syntax for each scenario uses _pseudo_ natural language (e.g., `When it receives the request`), the tests much use very specific sentences as illustrated in this guide.
:::

### Defaults

You can use the sample contract `sample.txt` and request `request.json` provided with a template by using specific steps.

For instance, the following scenario describe the expected response for the default contract text when sending the default request to the [helloworld template](https://templates.accordproject.org/helloworld@0.10.1.html):
```gherkin
Feature: HelloWorld
  This describe the expected behavior for the Accord Project's "Hello World!" contract

  Background:
    Given the default contract

  Scenario: The contract should say Hello to Fred Blogs, from the Accord Project, for the default request
    When it receives the default request
    Then it should respond with
"""
{
    "$class": "org.accordproject.helloworld.MyResponse",
    "output": "Hello Fred Blogs Accord Project"
}
"""
```

### Errors

Whenever appropriate, it is good practice to include both successful executions, as
well as scenarios for cases when a call to a template might fail. This can be
written using a **Then** step that describes the error.

For instance, the following scenario describe an expected error for a given request
to the [Interest Rate Swap](https://templates.accordproject.org/interest-rate-
swap@0.4.1.html) template:
```gherkin
Feature: Interest Rate Swap
  This describes the expected behavior for the Accord Project's interest rate swap
contract

  Background:
    Given that the contract says
"""
INTEREST RATE SWAP TRANSACTION LETTER AGREEMENT
"Deutsche Bank"

Date: 06/30/2005
To: "MagnaChip Semiconductor S.A."
Attention: Swaps Documentation Department
Our Reference: "Global No. N397355N"
Re: Interest Rate Swap Transaction

Ladies and Gentlemen:

The purpose of this letter agreement is to set forth the terms and conditions of
the Transaction entered into between "Deutsche Bank" and "MagnaChip Semiconductor
S.A." (“Counterparty”) on the Trade Date specified below (the “Transaction”). This
letter agreement constitutes a “Confirmation” as referred to in the Agreement
specified below.

The definitions and provisions contained in the 2000 ISDA Definitions (the
“Definitions”) as published by the International Swaps and Derivatives Association,
Inc. are incorporated by reference herein. In the event of any inconsistency
between the Definitions and this Confirmation, this Confirmation will govern.

For the purpose of this Confirmation, all references in the Definitions or the
Agreement to a “Swap Transaction” shall be deemed to be references to this
Transaction.

1. This Confirmation evidences a complete and binding agreement between "Deutsche
Bank" (“Party A”) and Counterparty (“Party B”) as to the terms of the Transaction
to which this Confirmation relates. In addition, Party A and Party B agree to use
all reasonable efforts to negotiate, execute and deliver an agreement in the form
of the ISDA 2002 Master Agreement with such modifications as Party A and Party B
will in good faith agree (the “ISDA Form” or the “Agreement”). Upon execution by
the parties of such Agreement, this Confirmation will supplement, form a part of
and be subject to the Agreement. All provisions contained or incorporated by
reference in such Agreement upon its execution shall govern this Confirmation
except as expressly modified below. Until Party A and Party B execute and deliver
the Agreement, this Confirmation, together with all other documents referring to
the ISDA Form (each a “Confirmation”) confirming Transactions (each a
“Transaction”) entered into between us (notwithstanding anything to the contrary in
a Confirmation) shall supplement, form a part of, and be subject to an agreement in
the form of the ISDA Form as if Party A and Party B had executed an agreement on
```

~~~~
the Trade Date of the first such Transaction between us in such form, with the
Schedule thereto (i) specifying only that (a) the governing law is English law,
provided, that such choice of law shall be superseded by any choice of law
provision specified in the Agreement upon its execution, and (b) the Termination
Currency is U.S. Dollars and (ii) incorporating the addition to the definition of
“Indemnifiable Tax” contained in (page 49 of) the ISDA “User’s Guide to the 2002
ISDA Master Agreements”.
2. The terms of the particular Transaction to which this Confirmation relates are
as follows:

Notional Amount: 300000000.00 USD
Trade Date: 06/23/2005
Effective Date: 06/27/2005
Termination Date: 06/18/2008

Fixed Amounts:
Fixed Rate Payer: "Counterparty"
Fixed Rate Payer Period End Dates: "The 15th day of March, June, September and
December of each year, commencing September 15, 2005, through and including the
Termination Date with No Adjustment"
Fixed Rate Payer Payment Dates: "The 15th day of March, June, September and
December of each year, commencing September 15, 2005, through and including the
Termination Date"
Fixed Rate: -4.09%
Fixed Rate Day Count Fraction: "30" "360"
Fixed Rate Payer Business Days:"New York"
Fixed Rate Payer Business Day Convention: "Modified Following"

Floating Amounts:
Floating Rate Payer: "DBAG"
Floating Rate Payer Period End Dates: "The 15th day of March, June, September and
December of each year, commencing September 15, 2005, through and including the
Termination Date with No Adjustment"
Floating Rate Payer Payment Dates: "The 15th day of March, June, September and
December of each year, commencing September 15, 2005, through and including the
Termination Date"
Floating Rate for initial Calculation Period: 3.41%
Floating Rate Option: "USD-LIBOR-BBA"
Designated Maturity: "Three months"
Spread: "None"
Floating Rate Day Count Fraction: "30" "360"
Reset Dates: "The first Floating Rate Payer Business Day of each Calculation Period
or Compounding Period, if Compounding is applicable."
Compounding: "Inapplicable"
Floating Rate Payer Business Days: "New York"
Floating Rate Payer Business Day Convention: "Modified Following"
"""

  Scenario: The fixed rate is negative
    When it receives the request
"""
{
    "$class": "org.accordproject.isda.irs.RateObservation"
}
"""
    Then it should reject the request with the error "[Ergo] Fixed rate cannot be
negative"
```
~~~~

~~~~
The reason for the error is that the contract has been defined with a negative
interest rate (the line: `Fixed Rate: -4.09%` in the contract given as
**Background** for the scenario).

### State Change

For templates which assume and can modify the contract state, the scenario should
also include pre- and post- conditions for that state. In addition, some steps are
available to define scenarios that specify the expected initial step for the
contract.

For instance, the following scenario for the [Installment
Sale](https://templates.accordproject.org/installment-sale@0.12.1.html) template
describe the expected initial state and execution of one installment:
```gherkin
Feature: Installment Sale
  This describe the expected behavior for the Accord Project's installment sale
contract

  Background:
    Given that the contract says
"""
"Dan" agrees to pay to "Ned" the total sum e10000, in the manner following:

E500 is to be paid at closing, and the remaining balance of E9500 shall be paid as
follows:

E500 or more per month on the first day of each and every month, and continuing
until the entire balance, including both principal and interest, shall be paid in
full -- provided, however, that the entire balance due plus accrued interest and
any other amounts due here-under shall be paid in full on or before 24 months.

Monthly payments, which shall start on month 3, include both principal and interest
with interest at the rate of 1.5%, computed monthly on the remaining balance from
time to time unpaid.

"""

  Scenario: The contract should be in the correct initial state
    Then the initial state of the contract should be
"""
{
  "$class": "org.accordproject.installmentsale.InstallmentSaleState",
  "status" : "WaitingForFirstDayOfNextMonth",
  "balance_remaining" : 10000.00,
  "total_paid" : 0.00,
  "next_payment_month" : 3,
  "stateId": "#1"
}
"""

  Scenario: The contract accepts a first payment, and maintain the remaining
balance
    Given the state
"""
{
  "$class": "org.accordproject.installmentsale.InstallmentSaleState",
  "status" : "WaitingForFirstDayOfNextMonth",
  "balance_remaining" : 10000.00,
~~~~

~~~
  "total_paid" : 0.00,
  "next_payment_month" : 3,
  "stateId": "#1"
}
"""
    When it receives the request
"""
{
    "$class": "org.accordproject.installmentsale.Installment",
    "amount": 2500.00
}
"""
    Then it should respond with
"""
{
  "total_paid": 2500,
  "balance": 7612.499999999999,
  "$class": "org.accordproject.installmentsale.Balance"
}
"""
    And the new state of the contract should be
"""
{
  "$class": "org.accordproject.installmentsale.InstallmentSaleState",
  "status" : "WaitingForFirstDayOfNextMonth",
  "balance_remaining" : 7612.499999999999,
  "total_paid" : 2500,
  "next_payment_month" : 4,
  "stateId": "#1"
}
"""

```

### Current Time

The logic for some clause or contract template are time-dependent. It can be useful
to specify multiple scenarios for the behavior under different date and time
assumptions. This can be described with an additional **When** step to set the
current time to a specific value.

For instance, the following shows two scenarios for the [IP
Payment](https://templates.accordproject.org/ip-payment@0.10.1.html) template,
which describe its expected behavior for two distinct current times:

```gherkin
Feature: IP Payment Contract
  This describes the expected behavior for the Accord Project’s IP Payment Contract
contract

  Background:
    Given the default contract

  Scenario: Payment of a specified amount should be made
    When the current time is "2019-03-04T16:34:00-05:00"
    And it receives the request
"""
{
    "$class": "org.accordproject.ippayment.PaymentRequest",
~~~

~~~~
    "netSaleRevenue": 1200,
    "sublicensingRevenue": 450,
    "permissionGrantedBy": "2018-04-05T00:00:00-05:00"
}
"""
    Then it should respond with
"""
{
    "$class": "org.accordproject.ippayment.PayOut",
    "totalAmount": 77.4,
    "dueBy": "2018-04-12T00:00:00.000-05:00"
}
"""

  Scenario: Payment of a specified amount should be made
    When the current time is "2019-03-01T16:34:00-02:00"
    And it receives the request
"""
{
    "$class": "org.accordproject.ippayment.PaymentRequest",
    "netSaleRevenue": 1550,
    "sublicensingRevenue": 225,
    "permissionGrantedBy": "2018-04-05T00:00:00-05:00"
}
"""
    Then it should respond with
"""
{
    "$class": "org.accordproject.ippayment.PayOut",
    "totalAmount": 81.45,
    "dueBy": "2018-04-12T03:00:00.000-02:00"
}
"""
```

### Emitting Obligations

If the template execution emits obligations, those can also be specified in the
scenario as one of the **Then** steps.

For instance, the following shows a scenario for the [Rental
Deposit](https://templates.accordproject.org/ip-payment@0.10.1.html) template,
which describes the expected list of obligations that should be emitted for a given
request:
```gherkin
Feature: Rental Deposit
  This describe the expected behavior for the Accord Project's rental deposit
contract

  Background:
    Given the default contract

  Scenario: The property was inspected and there was damage
    When the current time is "2018-01-02T16:34:00Z"
    And it receives the default request
    Then it should respond with
"""
{
   "$class": "org.accordproject.rentaldeposit.PropertyInspectionResponse",
~~~~

~~~
    "balance": {
      "$class": "org.accordproject.money.MonetaryAmount",
      "currencyCode" : "USD",
      "doubleValue" : 1550
    }
}
"""
    And the following obligations should have been emitted
"""
[
    {
        "$class": "org.accordproject.cicero.runtime.PaymentObligation",
        "amount": {
            "$class": "org.accordproject.money.MonetaryAmount",
            "doubleValue": 1550,
            "currencyCode": "USD"
        }
    }
]
"""
```
~~~

---
id: version-0.12-spec-concepts
title: Concepts and High-level Architecture
original_id: spec-concepts

![Overview](assets/cicero-spec-overview.png)

## Step 1: Creation of Clause Template
A legal professional analyzes a contract to determine the frequently used/standard clauses that are present. Clauses that are amenable to automation are extracted into a clause template. The template (more details follow) is comprised of the annotated legal text and an accompanying template data model that defines the assets, participants, concepts and events that are relevant to the clause. The business logic for the clause is coded by a developer (with review and in collaboration with the legal professional, or the suitably trained legal professional can code the contract logic themselves using the [Ergo](logic-ergo) DSL).

## Step 2: Data Modeling

Concept Template Data Model. The variables and expressions in a template are expressed in terms of a typed data model, that captures all the concepts of relevance to the clause. The data model technology allows importing concepts from namespaces, allowing concepts to be shared across templates.

## Step 3: Generation of the Template Parser
The [Cicero Open Source project](https://github.com/accordproject/cicero) contains code that can automatically generate a parser from the annotated template text (template grammar) and the associated template data model. The parser generation is completely automatic and supports customization of types and nested grammars.

## Step 4: Create a Clause
The generated template parser can now be used to dynamically edit and validate source clause text (potentially using code completion, error reporting etc). The editor technology can be embedded on a webpage, or executed as a SaaS service, or run within an IDE.

## Concept 5: Clause (instance of a Template)
The output of the Template Parser is an instance of the Template Model (a JSON
abstract syntax tree that can be deployed to the engine). It captures a machine
readable (and hashable) representation of all the executable data extracted from
the clause text.

## Step 6: Invoke Engine with a Request
The application feeds JSON documents to the engine that represents the request
instances, which themselves have been modelled in the Template Data Model. These
requests represent events of significance to the clause from the outside world.

## Step 7: Execute and Return Response
The engine invokes the business logic for the template, passing in the
parameterization data, a context object and the incoming request. The engine
validates the response and then returns it to the caller. The structure of the
response is modelled in the Template Data Model.

Once a template has been created (Steps 1 to 4), it can be used to _instantiate_ a
contract (Step 5) which itself can be executed by the template engine (Steps 5 to
6). Both contract instantiation and execution are shown in greater details on the
following figure:

![Execution Context](assets/execution_context.png)

> Note that the Accord Project specification does not assume a specific execution
environment. [Cicero](https://github.com/accordproject/cicero), which implements
the Accord Project specification, includes a Node.js VM based execution engine for
contracts created from Accord Project templates. The engine routes incoming
requests to template functions, performs data validation, executes the functions
within a sandboxed environment, and then validates the response. It can also update
the contract state and emit events and/or contract obligations back to the caller.

---
id: version-0.12-spec-example
title: Example: Late Delivery Clause
original_id: spec-example

In the rest of this specification, we will use the Late Delivery And Penalty legal
clause as an example. It is a common clause in a legal contract related to the
delivery of good or services, and in some circumstances may be amenable to
automation.

The Late Delivery And Penalty clause in the typical legal contract looks like this:

```text
   Late Delivery and Penalty. In case of delayed delivery except for Force
   Majeure cases, the Seller shall pay to the Buyer for every
   2 weeks of delay penalty amounting to 10.5% of
   the total value of the Equipment whose delivery has been delayed. Any
   fractional part of a week is to be considered a full
   week. The total amount of penalty shall not however, exceed
   55% of the total value of the Equipment involved in late
   delivery. If the delay is more than 10 weeks, the Buyer is entitled to
   terminate this Contract.
```

This specification can be used to convert the late delivery and penalty clause into a reusable fragment (a template), that can be executed by a suitable runtime. The complete template can be found in the [Accord Project Template Library](https://templates.accordproject.org/latedeliveryandpenalty@0.13.0.html).

---
id: version-0.12-spec-execution
title: Contract Execution
original_id: spec-execution

## Interfacing the Template with the Outside World

Given the template grammar and the template model above we can now edit (parameterise) the template to create a clause (an instance of the template).

Next we need to ground the template to events that are happening in the real-world: packages are getting shipped, delivered, signed-for etc. We want those transactions to be routed to the template, so that it is aware of them and can take appropriate action. In this case the action is simply to calculate the penalty amount and signal whether the buyer may terminate the contract.

## Transactions

Transactions are used to model the interactions between a contract or clause and the real world. Transactions are used for both inbound messages (requests) and as the synchronous return values (responses) from the logic of clauses or contracts.

### Requests

Example of requests include:

- A party has made a payment
- A party has signed for the goods, accepting delivery
- A temperature sensor reading from a shipping container
- A location of a truck from a GPS sensor

Accord Project uses CML again to define the structure of the data that the template requires from the outside world. For the late delivery and penalty clause, this looks as follows:

```
/**
 * Defines the input data required by the template
 */
transaction LateDeliveryAndPenaltyRequest {

  /**
   * Are we in a force majeure situation?
   */
  o Boolean forceMajeure

  /**
   * What was the agreed delivery date for the goods?
   */
  o DateTime agreedDelivery
  /**
   * If the goods have been delivered, when were they delivered?
   * the “optional” keyword means that if the goods have not yet been delivered,
the deliveredAt parameter may be omitted from the request.
   */
  o DateTime deliveredAt optional

  /**
   * What is the value of the goods?
   */
  o Double goodsValue
}
```

Given an instance of LateDeliveryAndPenaltyRequest the clause can calculate the current penalty amount and whether the buyer may terminate. The result of that calculation can be returned back to the caller as a response.

### Responses

Example responses:

- Cost of shipping the goods (minus late penalty) was $256
- Party A has breached the terms of the contract
- Volume discount for this quarter is now 3.5%

Accord Project uses CML again to define the structure of the data that the template returns to the outside world. For the late delivery and penalty clause, this looks as follows:

```
/**
 * Defines the output data for the template
 */
transaction LateDeliveryAndPenaltyResponse {
  /**
   * The penalty to be paid by the seller.
   * In a scenario where deliveredAt was omitted, we might expect “penalty” to be
NULL.
   * Arguably, “penalty” should also be an “optional” type, to distinguish between
a scenario where penalty is undefined, and a scenario where penalty is known to be
0.
   */
  o Double penalty

  /**
   * Whether the buyer may terminate the contract
   */
  o Boolean buyerMayTerminate
}
```

Here we are simply stating that execution this template will produce an instance of LateDeliveryAndPenaltyResponse.

### Event

The logic of a contract or clause may optionally emit events. Events are typically used by the contract to indicate that some asynchronous action should occur in the real-world, such as notifying a party to the contract that they need to take some action, or even, triggering an automated payment or invoice.

## State

A contract template may optionally be stateful, and declare a state type.

## Emitted Types

The logic for a template may optionally emit event types.

---
id: version-0.12-spec-instance
title: Template Instantiation
original_id: spec-instance

## Contract

A contract is an instance of a Contract Template, where the variables for the template have been set to specific values. A Contract may be instantiated by either parsing natural language text that conforms to the structure of the template grammar, or may be instantiated from a JSON object that is an instance of the Template Model for the template.

A Contract is composed of a set of Clauses. A Contract may have a state. Many contracts require state, for example, to remember the last time a contracting party made a payment. A contract may optionally be attached to a state object, giving the logic of the contract read and write access to the contract state.

## Clause

Clause is an instance of a Clause Template, where the variables for the template have been set to specific values.
A Clause may be instantiated by either parsing natural language text that conforms to the structure of the
template grammar, or may be instantiated from a JSON object that is an instance of the Template Model for the
template.

The logic for a Clause is implemented as a function, each of which takes a request and produces a response. The logic for a clause may optionally emit events. A clause has access to the properties and the state of its owning contract.

---
id: version-0.12-spec-overview
title: Overview
original_id: spec-overview

This specification defines the structure of Accord Project Templates: legally enforceable natural language text that is bound to executable business logic.

> Version: 0.8 (Working Draft)

## Goals

Accord Project templates bind legally enforceable natural language text to executable business logic. They provide the foundational technology for legal professionals to formalise a set of legally enforceable executable clauses and contracts.

The templates are designed to be easy and quick to create from existing legal contracts by legal professionals, and then made executable by legal technologists or programmers using the [Ergo](logic-ergo) domain specific language.

Templates may support one or more locales, allowing the template to be edited or visualized in different languages, whilst preserving a single locale-neutral executable representation.

Executable clauses are easy to hash for storage in content-based addressing systems (out of scope for this specification).

The [reference engine](https://github.com/accordproject/cicero) for the Accord Project Template Specification is designed to be easily embeddable across a wide-variety of form factors: web, middleware, SaaS, on-blockchain execution and off-blockchain execution.
The templates, clauses and the engine are designed to integrate into a traditional DevOps practices and CI/CD, including unit and system testing and code coverage analysis.

---
id: version-0.12-spec-packaging
title: Packaging
original_id: spec-packaging

The artifacts that define a template are:
- Metadata, such as name and version
- CML models, which define the template model, request, response and any required types
- Template grammar for each supported locale
- A sample instance, used to bootstrap editing, for each supported locale
- Executable business logic

Templates are typically packaged and distributed as Cicero Template Archive (`.cta`) files, however they may also be read from: a directory, http(s) URL, the Accord Project [template library](https://templates.accordproject.org), the npm package manager. Each of these distribution mechanisms support slightly different use cases:

- Directory: useful during testing, allows changes to the template to be quickly tested with no need to re-package
- URL: allows templates to be published to a stable web address
- Accord Project template library maintains a curated set of Open Source templates
- npm: allows dependencies on templates to be easily declared for Node.js and browser based applications. Integrates with CI/CD tools.

## Metadata
The metadata for a Template is stored in the /package.json text file in JSON format.

```
{
    "name": "latedeliveryandpenalty",
    "version": "0.11.1",
    "description": "A sample Late Delivery And Penalty clause.",
    "accordproject": {
        "template": "clause",
        "ergo": "0.8.0",
        "cicero": "^0.12.0"
    },
    "keywords": ["clause", "delivery", "acceptance", "obligation"]
}
```

The name property must consist of [a-z][A-Z][.]. It is strongly recommended that the name be prefixed with the domain name of the author of the clause template, to minimise naming collisions. The version property must be a semantic version of the form major.minor.micro [0-9].[0-9].[0.9]. Note that this data format ensures that a Template can be published to the npm package manager for either global or private (enterprise-wide) distribution.

The `accordproject` property of a template specifies the following metadata:
- `template`: must either be `clause` for a clause template or `contract` for a contract templates
- `ergo`: the Ergo version used to to draft the template. This is an npm [semver](https://semver.npmjs.com) specification.
- `cicero`: the Cicero version with which the template is compatible. This is an npm [semver](https://semver.npmjs.com) specification.

The `keywords` property should list words relevant to the template, and can be useful for search and classification.

Note that additional properties such as locales and jurisdictions may be added as future needs arise.

## README.md
The root of the template may also contain a markdown file to explain the purpose and semantics of the template. This file may be displayed by tools to preview the template or provide usage instructions.

## Template Grammars
The grammar files for the template are stored in the /grammar/ folder.

> Note that support for per-locale grammar files in TBD.

## Data Model

The data model for a clause template is stored in a set of files under the `/model` folder. The data model files must be in the format defined using the Composer Concerto modeling language. All data models for the template are in-scope and types from all namespaces may be imported.

Using the ability to convert CML models to UML we can even visualise all the required types (model, request, response) we have modelled graphically:

![UML diagram](assets/cicero-spec-uml.png)

## Execution Logic

The Ergo execution logic for a clause template is stored under the /lib folder.

---
id: version-0.12-spec-template
title: Template Structure
original_id: spec-template

An Accord Project template is composed of three elements:

- Natural Language, the grammar for the legal text of the template
- Model, the data model that backs the template
- Logic, the executable business logic for the template

![Cicero Template](assets/template.png)

When combined these three elements allow templates to be edited, analyzed, queried and executed.

## Model

The first step in converting the late delivery and penalty clause to use Accord Project is to identify the data elements that are captured by the clause (aka variables). These are:
- Whether the clause includes a force majeure provision
- Temporal duration for the penalty provision
- Percentage for the penalty provision
- Maximum penalty percentage (cap)
- Temporal duration after which the buyer may terminate the contract

These data elements comprise the Template Model for the clause. The template model is critical in that it defines formal semantics for the clause, and it is a locale neutral representation of the data that a template requires. It also enables powerful search, filtering and organization of templates, for example by finding all templates related to concept X, or all templates that can process a request of type Y.

They are captured formally using the [Concerto modeling language](https://github.com/hyperledger/composer-concerto) (CML). CML is a lightweight schema language that defines namespaces, types, and relationships between types. It includes first-class support for modeling participants (individuals or companies), assets, transactions, enumerations, concepts, and events, and includes the typical features of an Object Oriented modeling language, including inheritance, meta-annotations (decorators), and field specific validators. CML also defines a serialization of instances to JSON and a validator for instances, making it easy to integrate with a wide variety of JSON-capable external systems.

> Note that while the CML format was originally designed for use in Hyperledger Composer, the Accord Project Specification is not limited to executing on Hyperledger Composer. CML is merely a schema language and can be supported on any distributed ledger, or even non-distributed ledger technology, written in any programming language.

Concerto models may be published to GitHub or any HTTP(S) website, and models can declare dependencies on other models, reducing the technical barrier to entry to creating an eco-system of mutually reinforcing industry standard models.

In CML format the Template Model for the late delivery and penalty clause looks like this:

```ergo
/**
 * Defines the data model for the LateDeliveryAndPenalty template.
 * This defines the structure of the abstract syntax tree that the parser for the
template
 * must generate from input source text.
 */
asset TemplateModel extends AccordClause {
  /**
   * Does the clause include a force majeure provision?
   */
  o Boolean forceMajeure

  /**
   * For every penaltyDuration that the goods are late
   */
  o Duration penaltyDuration

  /**
   * Seller pays the buyer penaltyPercentage % of the value of the goods
   */
  o Double penaltyPercentage

  /**
   * Round up to the minimum fraction of a penaltyDuration
   */
  o Duration fractionalPart

  /**
   * Up to capPercentage % of the value of the goods
   */
  o Double capPercentage

  /**
   * If the goods are >= termination late then the buyer may terminate the contract
   */
  o Duration termination
}
```

The template model for the clause captures unambiguously the data types defined by the clause. The Duration data type is imported from an Accord Project namespace, which defines a library of useful reusable basic types for contracts. Only one asset within the model files for the template may extend the `AccordClause` or `AccordContract` base type.

> Terminology: a Template has a Template Model

## Grammar

The next step in making the clause executable is to relate the template model to the natural language text that describes the legally enforceable clause. This is accomplished by taking the natural language for the clause and inserting bindings to the template model, using the Accord Project markup language. We call this the

grammar for the template (or template grammar) as it determines what a
syntactically valid clause can look like.

Here is the marked-up template:

```md
   Late Delivery and Penalty. In case of delayed delivery[{" except for Force
   Majeure cases,":? forceMajeure}] the Seller shall pay to the Buyer for every
   [{penaltyDuration}] of delay penalty amounting to [{penaltyPercentage}]% of
   the total value of the Equipment whose delivery has been delayed. Any
   fractional part of a [{fractionalPart}] is to be considered a full
   [{fractionalPart}]. The total amount of penalty shall not however, exceed
   [{capPercentage}]% of the total value of the Equipment involved in late
   delivery. If the delay is more than [{termination}], the Buyer is entitled to
   terminate this Contract.
```

The marked-up template is UTF-8 text with markup to introduce named variables. Each
variable starts with `[{` and ends with `}]`. Let’s take a look at each variable in
turn.

- `[{"except for Force Majeure cases,":? forceMajeure}]` : this variable definition
is called a Boolean Assignment. It states that if the optional text “except for
Force Majeure cases,” is present in the clause, then the Boolean forceMajeure
property on the template model should be set to true. Otherwise the property should
be set to false.
- `[{penaltyDuration}]` : this variable definition is a binding. It states that the
variable is bound to the penaltyDuration property in the template model.
Implicitly it also states that the variable is of type Duration because that is the
type of penaltyDuration in the model.
- `[{penaltyPercentage}]` : another variable binding, this time to the
penaltyPercentage property in the model.
- `[{fractionalPart}]` : another variable binding, this time to the fractionalPart
property in the model. Note that this occurs twice in the template grammar -
however a smart editor for the clause should auto-replace all occurrences.
- `[{capPercentage}]` : this is a binding, setting the capPercentage property on
the template model.
- `[{termination}]` : this is a binding, setting the termination property on the
template model.

To recap, there are currently 2 kinds of variable definition supported:

1. Boolean Assignment: sets a boolean property in the model based on the presence
of text in the clause
2. Binding: set a property in the model based on a value supplied in the clause

> Note: Support for other types of binding may be added in the future as the need
arise.

Note that any types within the model may have an associated template grammar file.
For example the Duration type may have a template grammar that captures the syntax
for how to enter calendar durations in English or French etc. These dependent
grammars are merged into the template grammar for the root type for the template
(the asset that either extends `AccordClause` or `AccordContract`).

> Terminology: a Template Grammar is a marked-up template that declares variables.
Variables are bound to the Template Model. The Template Grammar and the Template
Model are used to generate a parser for the template, allowing syntactically valid
instances (clauses) to be created.

Reference information for the markup that is supported in Cicero can be found in [Markup Reference](cicero-markup).

## Logic
The last part of the puzzle for the template is to capture the logic of the template in a form that a computer can execute. No, computers cannot (yet) execute the natural language text, with all its interesting legal ambiguities!
Accord Project is extensible and supports pluggable mechanisms to capture the template logic. The accord-engine package acts as a shim, bootstrapping a kernel for a given template logic language.

Accord Project ships with the ability to execute template logic expressed using the [Ergo domain specific language](logic-ergo).

The example below illustrates the [Ergo](logic-ergo) logic for the late delivery and penalty clause.

```
contract LateDeliveryAndPenalty over LateDeliveryAndPenaltyContract {
  clause latedeliveryandpenalty(request : LateDeliveryAndPenaltyRequest) :
LateDeliveryAndPenaltyResponse emits PaymentObligation {
    // Guard against calling late delivery clause too early
    let agreed = request.agreedDelivery;
    enforce isBefore(agreed,now()) else
    throw ErgoErrorResponse{ message : "Cannot exercise late delivery before
delivery date" };

    // Handling for force majeure
    enforce !contract.forceMajeure or !request.forceMajeure else
    return LateDeliveryAndPenaltyResponse{
      penalty: 0.0,
      buyerMayTerminate: true
    };

    // If force majeure does not apply, calculate the penalty
    let diff = diffDurationAs(now,agreed,"days");
    let diffRatio =
divideDuration(diff,durationAs(contract.penaltyDuration,"days"));
    // Penalty formula
    let penalty = diffRatio * contract.penaltyPercentage/100.0 *
request.goodsValue;
    // Penalty may be capped
    let capped = min([penalty, contract.capPercentage/100.0 * request.goodsValue]);
    // Return the response with the penalty and termination determination
    return LateDeliveryAndPenaltyResponse{
      penalty: capped,
      buyerMayTerminate: diff.amount >
durationAs(contract.termination,"days").amount
    }
  }
}
```

It contains a single clause called `latedeliveryandpenalty` that produces a `LateDeliveryAndPenaltyResponse` in response to a `LateDeliveryAndPenaltyRequest`. This contract is stateless and does not emit events. See below for a description of contract state and events.

---
id: version-0.13-accordproject-installation
title: Installation
original_id: accordproject-installation

To start working on your own Accord Project templates, you should install the
Cicero command-line tools. This will let you author, parse, and execute Accord
Project templates.

## Prerequisites

Before you can install Cicero, you must first obtain and configure the following
dependency:

* [Node.js v8.x (LTS)](http://nodejs.org): We use Node to generate the
documentation, run a
  development web server, run tests, and generate distributable files. Depending on
your system,
  you can install Node either from source or as a pre-packaged bundle.

> We recommend using [nvm](https://github.com/creationix/nvm) (or [nvm-windows]
(https://github.com/coreybutler/nvm-windows)) to manage and install Node.js, which
makes it easy to change the version of Node.js per project.

## Installing Cicero

To install the latest version:

```bash
npm install -g @accordproject/cicero-cli@0.13
```

To check that Cicero has been properly installed, and display the version number:
```bash
cicero --version
```

To get command line help:
```bash
cicero --help
cicero parse --help
cicero execute --help
```

## Optional Packages

### Template Generator

You may also want to install the template generator tool, which you can use to
create an empty template:

```bash
npm install -g yo
npm install -g @accordproject/generator-cicero-template@0.13
```

### Ergo command line

If you would like to use the Ergo language on its own (i.e., independently of a
Cicero template) you can also install the Ergo command-line tools:

```bash
npm install @accordproject/ergo-cli@0.13 -g
```

To check that the Ergo compiler has been installed, display the version number:
```bash
ergoc --version
```

To get command line help:
```bash
ergoc --help
ergorun --help
```

That's it!

## What next?

The following pages provide links to developers tools and resources. You can also
browse using the navigation bar to the left to find additional information:
tutorials, API reference, the Ergo language guide, etc.

---
id: version-0.13-basic-create
title: Creating a New Template
original_id: basic-create

Now that you have executed an existing template, let's create a new template.

> If you would like to contribute your template back into the `cicero-template-
library` please start by [forking](https://help.github.com/articles/fork-a-repo/)
the `cicero-template-library` project on GitHub. This will make it easy for you to
submit a pull request to get your new template added to the library.

Install the template generator::

```
    npm install -g yo
    npm install -g yo @accordproject/generator-cicero-template@0.13
```

Run the template generator:

```
    yo @accordproject/cicero-template
```

> If you have forked the `cicero-template-library` `cd` into that directory first.

Give your generator a name (no spaces) and then supply a namespace for your

template model (again,
no spaces). The generator will then create the files and directories required for a basic template
(based on the helloworld template).

> You may find it easier to edit the grammar, model and logic for your template in VS Code, installing the Accord Project and Hyperledger Composer extensions. The extensions give you syntax highlighting and parser errors within VS Code.

## Update Sample.txt

First, replace the contents of `sample.txt` with the legal text for the contract or clause that you would like to digitize.
Check that when you run `cicero parse` that the `sample.txt` is now invalid with respect to the grammar.

## Edit the Template Grammar

Now update the grammar. Start by replacing the existing grammar, making it identical to the contents of your updated `sample.txt`.

Now introduce variables into your template grammar as required. The variables are marked-up using `[{` and `}]`
with what is between the braces being the name of your variable.

## Edit the Template Model

All of the variables referenced in your template grammar must exist in your template model. Edit
the file `models/model.cto` to include all your variables, making sure the name of the model property matches the name
of the variable in the `template.tem` file.

Note that the Hyperledger Composer Modeling Language primitive data types are:

- String
- Long
- Integer
- DateTime
- Double
- Boolean

> Note that you can import common types (address, monetary amount, country code, etc.) from the Accord Project Model Repository: https://models.accordproject.org.

## Edit the Transaction Types

Your template expects to receive data as input and will produce data as output. The structure of
this request/response data is captured in the `MyRequest` and `MyResponse` transaction types in your model
namespace. Open up the file `models/model.cto` and edit the definition of the `MyRequest` type to
include all the data you expect to receive from the outside world and that will be used by the
business logic of your template. Similarly edit the definition of the `MyResponse` type to include
all the data that the business logic for your template will compute and would like to return to the

caller.

## Edit the Template Logic

Now edit the business logic of the template itself. This is expressed in the Ergo language, which is a strongly-typed function domain specific language for contract logic. Open the file `lib/logic.ergo`
and edit the `helloworld` clause to perform the calculations your logic requires.

Looking at the Ergo logic for other example templates will help you understand the syntax and capabilities of Ergo.

---
id: version-0.13-basic-use
title: How to Use a Template
original_id: basic-use

The simplest way to work with an Accord Project template is through the Cicero command line interface (CLI). In this tutorial, we explain how to download an existing Accord Project template, create an instance of that template and how to execute the contract logic.

## Install Cicero CLI

In order to access the Cicero command line interface (CLI), first install the `@accordproject/cicero-cli` npm package:

```bash
npm install -g @accordproject/cicero-cli@0.13
```

> If you're new to `npm` the [installation instructions](accordproject-installation) have some more detailed guidance.

## Download a Template

You can download a single clause or contract template from the [Accord Project Template Library](https://templates.accordproject.org) as an archive (`.cta`) file.

If you click on the Template Library link, you should see a Web Page which looks as follows:

![Basic-Use-1](/docs/assets/basic/use1.png)

Scrolling down that page, you can see the index for the open-source templates along with their version, and whether they are a Clause or Contract template.

Click on the link to the `helloworld` template. You should be taken to a page which looks as follows:

![Basic-Use-2](/docs/assets/basic/use2.png)

Then click on the `Download Archive` button under the description for the template (highlighted in the red box in the figure). This should download the latest template archive for the `helloworld` template.

Cicero archives are files with a `.cta` extension, which includes all the different components for the template (the natural language, model and logic).

> Note that the version of `cicero-cli` needs to match the Cicero version that is required by a template.
> * You can check the version of your CLI with `cicero --version`.
> * You can choose a different version of a template with the *Versions* dropdown in the [Accord Project Template Library](https://templates.accordproject.org).
> * Otherwise, install a specific version of the cli, for example for v0.8, use `npm install -g @accordproject/cicero-cli@0.8`.

## Parse a Valid Clause Text

Using your terminal `cd` into the directory that contains the template archive you just downloaded, then create a sample clause text `sample.txt` which contains the following text:

```text
Name of the person to greet: "Fred Blogs".
Thank you!
```

The use the `cicero parse` command in your terminal to load the template and parse your sample clause text. This should be echoing the result of parsing back to your terminal.

```bash
cicero parse --template helloworld@0.10.1.cta --sample sample.txt
```

> Notes:
> * make sure that the version number in that command matches the one for the archive you have downloaded.
> * `cicero parse` requires network access. Make sure that you are online and that your firewall or proxy allows access to `https://models.accordproject.org`

This should print this output:

```json
{
  "$class": "org.accordproject.helloworld.HelloWorldClause",
  "clauseId": "aa3b9db9-f25f-41f4-88a4-64baba728bfe",
  "name": "Fred Blogs"
}
```

## Parse a Non-Valid Clause Text

If you attempt to parse invalid data, this same command should return with line and column information for the syntax error.

Edit your `sample.txt` file to add text that is not consistent with the template:

```text
FUBAR Name of the person to greet: "Fred Blogs".
Thank you!
```

Rerun `cicero parse --template helloworld@0.10.1.cta --sample sample.txt`. The output should now be:

```text
18:15:22 - error: invalid syntax at line 1 col 1:

  FUBAR Name of the person to greet: "Fred Blogs".
  ^
Unexpected "F"
```

## Execute the Clause

Use the `cicero execute` command to parse a clause text based (your `sample.txt`) *and* execute the clause logic using an incoming request in JSON format. To do so you need to create two additional files.

First, create a `state.json` file which contains:

```json
{
    "$class": "org.accordproject.cicero.contract.AccordContractState",
    "stateId": "org.accordproject.cicero.contract.AccordContractState#1"
}
```

This is the initial state for your contract.

Then, create a `request.json` file which contains:

```json
{
  "$class": "org.accordproject.helloworld.MyRequest",
  "input": "Accord Project"
}
```

This is the request which you will send to trigger the execution of your contract.

Then use the `cicero execute` command in your terminal to load the template, parse your sample clause text *and* execute the request. This should be echoing the result of execution back to your terminal.

```bash
cicero execute --template helloworld@0.10.1.cta --sample sample.txt --state
state.json --request request.json
```

> Note that `cicero execute` requires network access. Make sure that you are online and that your firewall or proxy allows access to `https://models.accordproject.org`

This should print this output:

```json
{
  "clause": "helloworld@0.10.1-
d4aab9b009796f56c45872149c1f97a164856b13056f3d503c76d5e519d9f097",
  "request": {
    "$class": "org.accordproject.helloworld.MyRequest",
    "input": "Accord Project",
```

~~~
    "transactionId": "952515b8-eb87-43d7-a582-4afb30eafc6b",
    "timestamp": "2019-04-15T22:44:14.747Z"
  },
  "response": {
    "$class": "org.accordproject.helloworld.MyResponse",
    "output": "Hello Fred Blogs Accord Project",
    "transactionId": "b9c1b74b-db46-4213-a4d0-fbc43f9c753b",
    "timestamp": "2019-04-15T22:44:14.759Z"
  },
  "state": {
    "$class": "org.accordproject.cicero.contract.AccordContractState",
    "stateId": "org.accordproject.cicero.contract.AccordContractState#1"
  },
  "emit": []
}
```

The results of execution displayed back on your terminal is in JSON format,
including the following information:

* Details of the `clause` executed (name, version, SHA256 hash of clause data)
* The incoming `request` object (the same request from your `request.json` file)
* The output `response` object
* The output `state` (unchanged in this example)
* An array of `emit`ted events (empty in this example)

## Try Other Examples

That's it! You have successfully parsed and executed your first Accord Project
Clause using the `helloworld` template.

Feel free to try the same commands to parse and execute other templates from the
Accord Project Library. Note that for each template you can find samples for the
text, for the request and for the state on the corresponding Web page. For
instance, a sample for the `latedeliveryandpenalty` clause is in the red box in the
following image:

![Basic-Use-3](/docs/assets/basic/use3.png)

## To Execute on Different Platforms

Templates may be executed on different platforms, not just from the command line.
In the [Advanced Tutorials](advanced-nodejs), you can find information on how to
execute a template in a standalone Node.js process, invoked as RESTful services, or
deployed as chaincode in Hyperledger Fabric.


--------------------------------------------------------------------------------
---
id: version-0.13-ergo-cli
title: Ergo CLI
original_id: ergo-cli
---

To install the Ergo command-line interface (CLI):

```term
    npm install -g @accordproject/ergo-cli@0.13
```
~~~

This will install `ergoc`, the Ergo compiler, `ergorun` to run your contracts locally on your machine, and `ergotop` which is a _read-eval-print-loop_ utility to write Ergo interactively.

## ergoc

### Usage

Compile an Ergo contract to a target platform

```term
    ergoc [options] [cto files] [ergo files]

    Options:
      --version       Show version and exit
      --target <lang> Target language (default: es6) (available:
es5,es6,cicero,java)
      --link          Adds the Ergo runtime to the target code (for es5,es6 and
cicero only)
      --monitor       Produce compilation time information
      --warnings      Print warnings
      --help          Show help and exit
```

`ergoc` takes your input models (cto files) and input contracts (ergo files) and generates code for execution. By default it generates JavaScript code (ES6 compliant).

### Examples

For instance, to compile the helloworld contract to JavaScript:

```term
    $ ergoc ./examples/volumediscount/model.cto
./examples/volumediscount/logic.ergo
    Compiling Ergo './examples/volumediscount/logic.ergo' -- creating
'./examples/volumediscount/logic.js'
```

To compile the helloworld contract to JavaScript and link the Ergo runtime for execution:

```term
    $ ergoc ./examples/volumediscount/model.cto
./examples/volumediscount/logic.ergo --link
    Compiling Ergo './examples/volumediscount/logic.ergo' -- creating
'./examples/volumediscount/logic.js'
```

To compile the helloworld contract to Java:

```term
    $ ergoc ./examples/volumediscount/model.cto
./examples/volumediscount/logic.ergo --target java
    Compiling Ergo './examples/volumediscount/logic.ergo' -- creating
'./examples/volumediscount/logic.java'
```

## ergorun

### Usage

Invoke an ergo contract

```term
    ergorun --contract [file] --state [file] --request [file] [ctos] [ergos]

    Options:
      --help          Show help
[boolean]
      --version       Show version number
[boolean]
      --contract      path to the contract data
[required]
      --request       path to the request data                    [array]
[required]
      --state         path to the state data                [string] [default:
null]
      --warnings      print warnings                      [boolean] [default:
false]
      --contractname
[required]
      --verbose, -v                                                 [default:
false]
```

`ergorun` lets you invoke your Ergo contract. You need to pass the CTO and Ergo files, the name of the contract that you want to execute, and JSON files for: the contract parameters, the current state of the contract, and for the request.

### Examples

For instance, to send one request to the `volumediscount` contract:

```term
    $ ergorun ./examples/volumediscount/model.cto
./examples/volumediscount/logic.ergo --contractname
org.accordproject.volumediscount.VolumeDiscount --contract
./examples/volumediscount/contract.json --request
./examples/volumediscount/request.json --state ./examples/volumediscount/state.json
    02:33:50 - info: {"response":
{"discountRate":2.8,"$class":"org.accordproject.volumediscount.VolumeDiscountRespon
se"},"state":
{"$class":"org.accordproject.cicero.contract.AccordContractState","stateId":"1"},"e
mit":[]}
```

If contract invokation is successful, `ergorun` will print out the response, the new contract state and any emitted events.

## ergotop (REPL)

### Starting the REPL

`ergotop` is a convenient tool to try-out Ergo contracts in an interactive way. You can write commands, or expressions and see the result. It is often called the Ergo REPL, for _read-eval-print-loop_, since it literally: reads your input Ergo from the command-line, evaluates it, prints the result and loops back to read your next input.

To start the REPL:

```
    $ ergotop
    02:39:37 - info: Logging initialized. 2018-09-25T06:39:37.209Z
    ergo$
```

It should print the prompt `ergo$` which indicates it is ready to read your command. For instance:

```ergo
    ergo$ return 42
    Response. 42 : Integer
```

`ergotop` prints back both the resulting value and its type. You can then keep typing commands:

```ergo
    ergo$ return "hello " ++ "world!"
    Response. "hello world!" : String
    ergo$ define constant pi = 3.14
    ergo$ return pi ^ 2.0
    Response. 9.8596 : Double
```

If your expression is not valid, or not well-typed, it will return an error:

```ergo
    ergo$ return if true else "hello"
    Parse error (at line 1 col 15).
    return if true else "hello"
                   ^^^^
    ergo$ return if "hello" then 1 else 2
    Type error (at line 1 col 10). 'if' condition not boolean.
    return if "hello" then true else false
              ^^^^^^^
```

If what you are trying to write is too long to fit on one line, you can use `\` to go to a new line:

```ergo
    ergo$ define function squares(l:Double[]) : Double[] { \
      ...   return \
      ...     foreach x in l return x * x \
      ... }
    ergo$ return squares([2.3,4.5,6.7])
    Response. [5.29, 20.25, 44.89] : Double[]
```

### Loading files

You can load CTO and Ergo files to use in your REPL session. Once the REPL is launched you will have to import the corresponding namespace. For instance, if you

want to use the `compoundInterestMultiple` function defined in the
`./examples/promissory-note/money.ergo` file, you can do it as follows:

```ergo
$ ergotop ./examples/promissory-note/money.ergo
08:45:26 - info: Logging initialized. 2018-09-25T12:45:26.481Z
ergo$ import org.accordproject.ergo.money.*
ergo$ return compoundInterestMultiple(0.035, 100.0)
Response. 1.00946960405 : Double
ergo$
```

### Calling contracts

To call a contract, you first needs to _instantiate_ it, which means setting its
parameters and initializing its state. You can do this by using the `set contract`
and `call init` commands respectively. Here is an example using the
`volumediscount` template:

```ergo
$ ergotop ./examples/volumediscount/model.cto ./examples/volumediscount/logic.ergo
ergo$ import org.accordproject.cicero.contract.*
ergo$ import org.accordproject.cicero.runtime.*
ergo$ import org.accordproject.volumediscount.*
ergo$ set contract VolumeDiscount over TemplateModel{ \
  ...   firstVolume: 1.0, \
  ...   secondVolume: 10.0, \
  ...   firstRate: 3.0, \
  ...   secondRate: 2.9, \
  ...   thirdRate: 2.8 \
  ... }
ergo$ call init(Request{})
Response. unit : Unit
State. AccordContractState{stateId: "1"} : AccordContractState
```

You can then invoke clauses of the contract:

```ergo
ergo$ call volumediscount(VolumeDiscountRequest{ netAnnualChargeVolume : 0.1 })
Response. VolumeDiscountResponse{discountRate: 3.0} : VolumeDiscountResponse
ergo$ call volumediscount(VolumeDiscountRequest{ netAnnualChargeVolume : 10.5 })
Response. VolumeDiscountResponse{discountRate: 2.8} : VolumeDiscountResponse
```

You can also invoke the contract without explicitly naming the clause by simply
sending a request. The Ergo engine dispatches that request to the first clause
which can handle it:
```ergo
ergo$ send VolumeDiscountRequest{ netAnnualChargeVolume : 0.1 }
Response. VolumeDiscountResponse{discountRate: 3.0} : VolumeDiscountResponse
ergo$ send VolumeDiscountRequest{ netAnnualChargeVolume : 10.5 }
Response. VolumeDiscountResponse{discountRate: 2.8} : VolumeDiscountResponse
```

---
id: version-0.13-ergo-tutorial

title: Ergo: A Tutorial
original_id: ergo-tutorial

## Overview of Accord

Cicero is an Open Source implementation of the Accord Project Template Specification. It defines the structure of natural language templates, bound to a data model, that can be executed using Ergo and request/response JSON messages. You can read the latest user documentation here: http://docs.accordproject.org.

In short with the Accord Project you can take a classic contract, e.g. Word document and use Cicero to define natural language contract and clause templates that can be executed by an event driven computer program (aka Smart contract). For the tutorial, Cicero will be used to define natural language contract and clause templates. These clause templates handle the syllogistic language of contracts.

For example,
```md
 if the goods are more than [{DAYS}] late,
 then notify the supplier of the goods, with the message [{MESSAGE}].
```
DAYS and MESSAGE are variables

You can browse the library of Open Source Cicero contract and clause templates at: https://templates.accordproject.org.

So how goes the contract get executed? That is where Ergo comes in Ergo is a strongly-typed functional programming language designed to capture the legal intent of legal contracts and clauses. We will use Ergo to create the contract logic consisting of a contract class with executable embedded clauses. Note: prior to the emergence of Ergo, the Cicero JavaScript component was primary to the execution of code.

Ergo obviates the Cicero JavaScript component for the execution phase with a new more comprehensive language which we explore in this tutorial.

## Cicero

The Open Source Cicero project defines the format of clause and contract templates based on to the Cicero Template Specification. The templates are the link between the natural language of contracts usually composed in a Word document and the specification of a machine executable transaction. Cicero templates define the API by specifying request and response elements for the logic associated with functional transaction executed by Ergo.

Cicero templates are composed of two elements:
* Template Grammar (the natural language text for the template),
* Template Model (the data model that includes the variables contained within the template).
* The Logic (the executable business logic for the template) will be handled by Ergo.

When combined these three elements allow templates to be edited, analyzed, queried and executed.

## Setup Ergo Development environment

Before you can build Ergo, you must install and configure the following dependencies on your machine:

### Git

* Git: The [Github Guide to Installing Git][git-setup] is a good source of information.

### Node.js

* Node.js v8.x (LTS): We use Node to generate the documentation, run a development web server, run tests, and generate distributable files. Depending on your system, you can install Node either from source or as a pre-packaged bundle.
> Tip: Use nvm (or nvm-windows) to manage and install Node.js, This facilitates a version change of Node.js per project.
* Lerna: This is a tool which helps when handling multiple npm packages in the Ergo repository. To install:
npm install -g lerna@2.11.0

### Visual Studio Code

Visual Studio Code is a lightweight but powerful source code editor which runs on your desktop and is available for Windows, macOS and Linux. It comes with built-in support for JavaScript and Node.js and has a rich ecosystem of extensions for other languages (such Ergo).

Follow the platform specific guides below:
See, https://code.visualstudio.com/docs/setup/
* macOS
* Linux
* Windows

#### Install Ergo VisualStudio Plugin

### Validate Development Environment and Toolset

Clone https://github.com/accordproject/ergo to your local machine

### Getting started

Install Ergo

The easiest way to install Ergo is as a Node.js package. Once you have Node.js installed on your machine, you can get the Ergo compiler and command-line using the Node.js package manager by typing the following in a terminal:
$ npm install -g @accordproject/ergo-cli@0.13

This will install the compiler itself (ergoc) and a command-line tool (ergo) to execute Ergo code. You can check that both have been installed and print the version number by typing the following in a terminal:
```sh
$ ergoc --version
$ ergo --version
```
Then, to get command line help:
```
$ ergoc --help
$ ergo execute --help
```
Compiling your first contract

```ergo
namespace org.accordproject.volumediscount

contract VolumeDiscount over VolumeDiscountContract {
   // Clause for volume discount
   clause volumediscount(request : VolumeDiscountRequest) : VolumeDiscountResponse
{
        if request.netAnnualChargeVolume < contract.firstVolume
        then return VolumeDiscountResponse{ discountRate: contract.firstRate }
        else if request.netAnnualChargeVolume < contract.secondVolume
        then return VolumeDiscountResponse{ discountRate: contract.secondRate }
        else return VolumeDiscountResponse{ discountRate: contract.thirdRate }
  }
}
```

To compile your first Ergo contract to JavaScript , within Visual Studio code
* Open the folder where you cloned https://github.com/accordproject/ergo
* Use View/Terminal to run the Ergo compiler:
```sh
$ ergoc ./examples/volumediscount/model.cto ./examples/volumediscount/logic.ergo
Compiling Ergo './examples/volumediscount/logic.ergo' -- creating
'./examples/volumediscount/logic.js'
```

By default, Ergo compiles to JavaScript for execution. This may change in the future to support other languages. The compiled code for the result in stored as `./examples/volumediscount/logic.js`

### Execute a contract
To execute a contract, we pass the necessary parameters including the CTO, Ergo files, the name of a contract and the json files containing request and contract state
ergorun [ctos] [ergos] --contractname [file] --contract [file] --state [file] --
request [file]

So for example we use ergorun with :
```sh
$ ergorun ./examples/volumediscount/model.cto ./examples/volumediscount/logic.ergo
--contractname org.accordproject.volumediscount.VolumeDiscount
--contract ./examples/volumediscount/contract.json
--request ./examples/volumediscount/request.json
--state ./examples/volumediscount/state.json
```

Here contract.json contains the following values
```json
{
  "$class": "org.accordproject.volumediscount.VolumeDiscountContract",
  "parties": null,
  "contractId": "cr1",
  "firstVolume": 1,
  "secondVolume": 10,
  "firstRate": 3,
  "secondRate": 2.9,
  "thirdRate": 2.8
}
```

Request.json contains
```json
{
  "$class": "org.accordproject.volumediscount.VolumeDiscountRequest",
  "netAnnualChargeVolume": 10.4
}
```

logic.ergo contains:
```ergo
namespace org.accordproject.volumediscount

contract VolumeDiscount over VolumeDiscountContract {
  // Clause for volume discount
  clause volumediscount(request : VolumeDiscountRequest) : VolumeDiscountResponse {
    if request.netAnnualChargeVolume < contract.firstVolume
    then return VolumeDiscountResponse{ discountRate: contract.firstRate }
    else if request.netAnnualChargeVolume < contract.secondVolume
    then return VolumeDiscountResponse{ discountRate: contract.secondRate }
    else return VolumeDiscountResponse{ discountRate: contract.thirdRate }
  }
}
```

Here netAnnualCharge Volume equals 10.4 which is not less than firstVolume and secondVolume which are equal to 1 and 10 respectively so the logic for the volumediscount clause returns thirdRate which equals 2.8

```
7:31:58 PM - info: Logging initialized. 2018-09-27T23:31:58.623Z
7:31:59 PM - info: {"response":
{"discountRate":2.8,"$class":"org.accordproject.volumediscount.VolumeDiscountRespon
se"},"state":
{"$class":"org.accordproject.cicero.contract.AccordContractState","stateId":"1"},"e
mit":[]}
```

PS D:\Users\jbambara\Github\ergo>

## Ergo Development

Create Template
Start with basic agreement in natural language and locate the variables
Here in the example see the bold
Volume-Based Card Acceptance Agreement [Abbreviated]
This Agreement is by and between ………..you agree to be bound by the Agreement.
Discount means an amount that we charge you for accepting the Card, which amount is:
(i) a percentage (Discount Rate) of the face amount of the Charge that you submit, or a flat per-
Transaction fee, or a combination of both; and/or
(ii) a Monthly Flat Fee (if you meet our requirements).

Transaction Processing and Payments. ………………… less all applicable deductions, rejections, and withholdings, which include:
………………………….

SETTLEMENT
a) Settlement Amount. Our agent will pay you according to your payment plan,

……………………..which include:
        (i) the Discount,
………………………………………..
b) Discount. The Discount is determined according to the following table:

Annual Dollar Volume	Discount
Less than $1 million	3.00%
$1 million to $10 million	2.90%
Greater than $10 million	2.80%

Identify the request variables and contract instance variables
Codify the variables with $[{request}] or [{contract instance}]

Annual Dollar Volume	Discount
Less than $[{firstVolume}] million	[{firstRate}]%
$[{firstVolume}] million to $[{secondVolume}] million	[{secondRate}]%
Greater than $[{secondVolume}] million	[{thirdRate}]%

Create Model
Define the model asset which contains the contract instance variables and the transaction request and response. Defines the data model for the VolumeDiscount template. This defines the structure that the parser for the template generates from input source text. See model.cto below:

```
 namespace org.accordproject.volumediscount
import org.accordproject.cicero.contract.* from
https://models.accordproject.org/cicero/contract.cto
import org.accordproject.cicero.runtime.* from
https://models.accordproject.org/cicero/runtime.cto
asset VolumeDiscountContract extends AccordContract {
  o Double firstVolume
  o Double secondVolume
  o Double firstRate
  o Double secondRate
  o Double thirdRate
}
transaction VolumeDiscountRequest {
  o Double netAnnualChargeVolume
}
transaction VolumeDiscountResponse {
        o Double discountRate
}
```

Create Logic
The contract logic is accomplished by coding ERGO statements and expressions to consume the request and use contract instance variables to produce the desired response. In our example, request.netAnnualChargeVolume is tested against contract rates to produce the result.

```
namespace org.accordproject.volumediscount

define the contract
contract VolumeDiscount over VolumeDiscountContract {

define the contract clause and request : response

   clause volumediscount(request : VolumeDiscountRequest) : VolumeDiscountResponse
{

define the logic ; here we use if /then /else statement to test request parameter
against contract instance variable
 and return
```

```
        if request.netAnnualChargeVolume < contract.firstVolume
        then return VolumeDiscountResponse{ discountRate: contract.firstRate }
        else if request.netAnnualChargeVolume < contract.secondVolume
        then return VolumeDiscountResponse{ discountRate: contract.secondRate }
        else return VolumeDiscountResponse{ discountRate: contract.thirdRate }
  }
```

Ergo Language
As you have seen in this tutorial, Ergo is a domain-specific language (DSL) that captures the execution logic of legal contracts. In this simple example, you see that Ergo aims to have contracts and clauses as first-class elements of the language. To accommodate the maturation of distributed ledger implementations, Ergo will be blockchain neutral, i.e., the same contract logic can be executed either on and off chain on distributed ledger technologies like HyperLedger Fabric. Most importantly, Ergo is consistent with the Accord Protocol Template Specification.
Follow the links below to learn more about
Introduction to Ergo
Ergo Language Guide
Ergo Reference Guide

October 12, 2018

---
id: version-0.13-logic-complex-type
title: Complex Values & Types
original_id: logic-complex-type

So far we only considered atomic values and types, such as string values or integers, which are not sufficient for most contracts. In Ergo, values and types are based on the [Composer Concerto Modeling Language](https://github.com/hyperledger/composer-concerto) (often referred to as CTO files). This provides a rich vocabulary to define the parameters of your contract, the information associated to contract participants, the structure of contract obligation, etc.

In Ergo, you can either import an existing CTO file or declare types directly within your code. Let us look at the different kinds of types you can define and how to create values with those types.

## Arrays

Array types lets you define collections of values and are denoted with `[]` after the type of elements in that collection:

```ergo
    String[]                         // a String array
    Double[]                         // a Double array
```

You can write arrays as follows:
```ergo
    ["pear","apple","strawberries"]  // an array of String values
    [3.14,2.72,1.62]                 // an array of Double values
```

You can construct arrays using other expressions:
```ergo
    let pi = 3.14;
    let e = 2.72;
    let golden = 1.62;
    [pi,e,golden]
```

Ergo also provides functions to manipulate arrays as parts of its [standard library](ergo-stdlib.html#functions-on-arrays):
```ergo
    let pi = 3.14;
    let e = 2.72;
    let golden = 1.62;
    let prettynumbers : Double[] = [pi,e,golden];
    sum(prettynumbers)
```

You can access the element at a given position inside the array using an index:
```ergo
    let fruits = ["pear","apple","strawberries"];
    fruits[0]          // Returns: some("pear")
     let fruits = ["pear","apple","strawberries"];
    fruits[2]          // Returns: some("strawberries")
     let fruits = ["pear","apple","strawberries"];
    fruits[4]          // Returns: none
```

 Note that the index starts at `0` for the first element and that indexed-based access returns an optional value, since Ergo compiler cannot statically determine whether there will be an element at the corresponding index.

## Classes

You can declare classes in the Composer Modeling Language (concepts, transactions, events, participants or assets) by importing them from a CTO file or directly within your Ergo program:

```ergo
   define concept Seminar {
     name : String,
     fee : Double
   }
   define asset Product {
     id : String
   }
   define asset Car extends Product {
     range : String
   }
   define transaction Response {
     rate : Double,
     penalty : Double
   }
  define event PaymentObligation{
    amount : Double,
    description : String
  }
```

Once a class type has been defined, you can create an instance of that type using the class name along with the values for each fields:

```ergo
    Seminar{
      name: "Law for developers",
      fee: 29.99
    }
    Car{
      id: "Batmobile4156",
      range: "Unknown"
    }
```

> **TechNote:** When extending an existing class (e.g., `Car extends Product`), the sub-class includes the fields from the super-class. So `Car` includes the field `range` which is locally declared and the field `id` which is declared in `Product`.

You can access the field of a class using the `.` operator:
```ergo
    Seminar{
      name: "Law for developers",
      fee: 29.99
    }.fee                        // Returns 29.99
```

## Records

Sometimes it is convenient to declare a structure without having to declare it first. You can do that using a record, which is similar to a class but without its name:

```ergo
   {
     name : String,  // A record with a name of type String
     fee : Double    // and a fee of type Double
   }
```

You do not need to declare that record, and can directly write an instance of that record as follows:

```ergo
    {
      name: "Law for developers",
      fee: 29.99
    }
```

> Typing `return { name: "Law for developers", fee: 29.99 }` in the [Ergo REPL](https://ergorepl.netlify.com), should answer `Response. {name: "Law for developers", fee: 29.99} : {fee: Double, name: String}`.

You can access the field of a record using the `.` operator:
```ergo
    {
      name: "Law for developers",
      fee: 29.99
```

~~~
    }.fee                        // Returns 29.99
```
## Enums

Here is how to declare an enumerated type:

```ergo
define enum ProductType {
   DAIRY,
   BEEF,
   VEGETABLES
}
```

> **TechNote:** Enumerated types are handled as `String` at the moment.

To create an instance of that enum:
```ergo
"DAIRY"
"BEEF"
```

## Optional types

An optional type can contain a value or not and is indicated with a `?`.

```ergo
Integer?          // An optional integer
PaymentObligation // An optional payment obligation
Double[]?         // An optional array of doubles
```

A an optional value can be either present, written `some(v)`, or absent, written
`none`.

```ergo
let i1 : Integer? = some(1); i1
let i2 : Integer? = none; i2
```

To operate on an optional type, you need to say what to do when the value is
present and what to do when the value is not present. You can do that with a match
statement:

This example:
```ergo
match some(1)
with let? x then "I found 1 :-)"
else "I found nothing :-("
```
should return `"I found 1 :-)"`.

This example:
```
match none
with let? x then "I found 1 :-)"
else "I found nothing :-("
```
should return `"I found nothing :-("`.
~~~

For conciseness, a few operators are also available on optional values. One can give a default value when the optional is `none` using the operator `??`. For instance:

```ergo
some(1) ?? 0       // Returns the integer 1
none ?? 0          // Returns the integer 0
```

You can also access the field inside an optional concept or an optional record using the operator `?.`. For instance:

```ergo
some({a:1})?.a     // Returns the optional value: some(1)
none?.a            // Returns the optional value: none
```

---
id: version-0.13-logic-stmt
title: Statements
original_id: logic-stmt

A clause's body is composed of statements. Statements are a special kind of expression which can manipulate the contract state and emit obligations. Unlike other expressions they may return a response or an error.

## Contract data

When inside a statement, data about the contract -- either the contract parameters, clause parameters or contract state are available using the following Ergo keywords:
```ergo
    contract   // The contract parameters (from a contract template)
    clause     // Local clause parameters (from a clause template)
    state      // The contract state
```

For instance, if your contract template parameters and state information:
```ergo
    // Template parameters
    asset InstallmentSaleContract extends AccordContract {
      o AccordParty BUYER
      o AccordParty SELLER
      o Double INITIAL_DUE
      o Double INTEREST_RATE
      o Double TOTAL_DUE_BEFORE_CLOSING
      o Double MIN_PAYMENT
      o Double DUE_AT_CLOSING
      o Integer FIRST_MONTH
    }
    // Contract state
    enum ContractStatus {
      o WaitingForFirstDayOfNextMonth
      o Fulfilled
    }
```

    asset InstallmentSaleState extends AccordContractState {
      o ContractStatus status
      o Double balance_remaining
      o Integer next_payment_month
      o Double total_paid
   }

You can use the following expressions:
```ergo
    contract.BUYER
    state.balance_remaining
```

## Returning a response

Returning a response from a clause can be done by using a `return` statement:

```ergo
     return 1                         // Return the integer one
     return Payout{ amount: 39.99 } // Return a new Payout object
     return                           // Return nothing
```

> **TechNote:** the [Ergo REPL](https://ergorepl.netlify.com) takes statements as
input which is why we had to add `return` to expressions in previous examples.

## Returning a failure

Returning a failure from a clause can be done by using a `throw` statement:
```ergo
    throw ErgoErrorResponse{ message: "This is wrong" }
    define concept MyOwnError extends ErgoErrorResponse{ fee: Double }
    throw MyOwnError{ message: "This is wrong and costs a fee", fee: 29.99 }
```

For convenience, Ergo provides a `failure` function which takes a string as part of
its [standard library](ergo-stdlib.html#other-functions), so you can also write:
```ergo
    throw failure("This is wrong")
```

## Enforce statement

Before a contract is enforceable some preconditions must be satisfied:
- Competent parties who have the legal capacity to contract
- Lawful subject matter
- Mutuality of obligation
- Consideration

The constructs below will be used to determine if the preconditions have been met
and what actions to take if they are not

```test
Example Prose
    Do the parties have adequate funds to execute this contract?
```

One can check preconditions in a clause using enforce statements, as follows:

```ergo
    enforce x >= 0.0                    // Condition
    else throw "Something went wrong"; // Statement if condition is false
    return x+1.0                        // Statement if condition is true
```

The else part of the statement can be omitted in which case Ergo
returns an error by default.

```ergo
    enforce x >= 0.0;         // Condition
    return x+1.0              // Statement if condition is true
```

## Emitting obligations

When inside a clause or contract, you can emit (one or more) obligations as
follows:
```ergo
   emit PaymentObligation{ amount: 29.99, description: "12 red roses" };
   emit PaymentObligation{ amount: 19.99, description: "12 white tulips" };
   return
```

Note that `emit` is always terminated by a `;` followed by another statement.

## Setting the contract state

When inside a clause or contract, you can change the contract state as follows:
```ergo
    set state InstallmentSaleState{
      stateId: "#1",
      status: "WaitingForFirstDayOfNextMonth",
      balance_remaining: contract.INITIAL_DUE,
      total_paid: 0.0,
      next_payment_month: contract.FIRST_MONTH
    };
    return
```

Note that `set state` is always terminated by a `;` followed by another statement.

## Printing intermediate results

 For debugging purposes a special `info` statement can be used in your contract
logic. For instance, the following indicates that you would like the Ergo execution
engine to print out the result of expression `state.status` on the standard output.

 ```ergo
    set state InstallmentSaleState{
      stateId: "#1",
      status: "WaitingForFirstDayOfNextMonth",
      balance_remaining: contract.INITIAL_DUE,
      total_paid: 0.0,
      next_payment_month: contract.FIRST_MONTH
    };
    info(state.status);     // Directive to print to standard output
 ```

~~~
    return
```
--------------------------------------------------------------------------------
---
id: version-0.13-ref-errors
title: Errors Reference
original_id: ref-errors
---

 As much as possible, errors returned by Cicero or the Ergo compiler are normalized
and categorized in order to facilitate handling of those error by the application
code. Those errors are raised in Cicero as JavaScript _exceptions_.

 ## Errors Hierarchy

 The hierarchy of errors (or exceptions) is shown on the following diagram:

 ![Error Hierarchy](assets/exceptions.png)

 ## CTO Model

 For reference, those can also be described using the following CTO model:

 ```ergo
namespace org.accordproject.errors
 /** Common */
concept LocationPoint {
  o Integer line
  o Integer column
  o Integer offset optional
}
concept FileLocation {
  o LocationPoint start
  o LocationPoint end
}
 concept BaseException {
      o String component // Node component the error originates from
  o String name      // name of the class
  o String message
}
concept BaseFileException extends BaseException {
      o FileLocation fileLocation
      o String shortMessage
      o String fileName
}
concept ParseException extends BaseFileException {
}
 /* Model errors */
concept ValidationException extends BaseException {
}
concept TypeNotFoundException extends BaseException {
      o String typeName
}
concept IllegalModelException extends BaseFileException {
      o String modelFile
}
 /* Ergo errors */
concept CompilerException extends BaseFileException {
}
~~~

```
concept TypeException extends BaseFileException {
}
concept SystemException extends BaseFileException {
}
 /* Cicero errors */
concept TemplateException extends ParseException {
}
```

---
id: version-0.13-ref-logic-stdlib
title: Ergo Libraries
original_id: ref-logic-stdlib

The following libraries are provided with the Ergo compiler.

## Stdlib

The following functions are in the `org.accordproject.ergo.stdlib` namespace and available by default.

### Functions on Integer

Name	Signature	Description
`integerAbs`	`(x:Integer) : Integer`	Absolute value
`integerLog2`	`(x:Integer) : Integer`	Base 2 integer logarithm
`integerSqrt`	`(x:Integer) : Integer`	Integer square root
`integerToDouble`	`(x:Integer) : Double`	Cast to a Double
`integerMod`	`(x:Integer, y:Integer) : Integer`	Integer remainder
`integerMin`	`(x:Integer, y:Integer) : Integer`	Smallest of `x` and `y`
`integerMax`	`(x:Integer, y:Integer) : Integer`	Largest of `x` and `y`

### Functions on Double

Name	Signature	Description
`abs`	`(x:Double) : Double`	Absolute value
`sqrt`	`(x:Double) : Double`	Square root
`exp`	`(x:Double) : Double`	Exponential
`log`	`(x:Double) : Double`	Natural logarithm
`log10`	`(x:Double) : Double`	Base 10 logarithm
`ceil`	`(x:Double) : Double`	Round to closest integer above
`floor`	`(x:Double) : Double`	Round to closest integer below
`truncate`	`(x:Double) : Integer`	Cast to an Integer
`doubleToInteger`	`(x:Double) : Integer`	Same as `truncate`
`minPair`	`(x:Double, y:Double) : Double`	Smallest of `x` and `y`
`maxPair`	`(x:Double, y:Double) : Double`	Largest of `x` and `y`

### Functions on Arrays

Name	Signature	Description
`count`	(x:Any[]) : Integer	Number of elements
`flatten`	(x:Any[][]) : Any[]	Flattens a nested array
`arrayAdd`	`(x:Any[],y:Any[]) : Any[]`	Array concatenation
`arraySubtract`	`(x:Any[],y:Any[]) : Any[]`	Removes elements of `y` in `x`
`inArray`	`(x:Any,y:Any[]) : Boolean`	Whether `x` is in `y`

`containsAll`	`(x:Any[],y:Any[]) : Boolean`	Whether all elements of `y` are in `x`
`distinct`	`(x:Any[]) : Any[]`	Duplicates elimination

Note: For most of these functions, the type-checker infers more precise types than indicated here. For instance `concat([1,2],[3,4])` will return `[1,2,3,4]` and have the type `Integer[]`.

### Aggregate functions

Name	Signature	Description
`max`	(x:Double[]) : Double	The largest element in `x`
`min`	(x:Double[]) : Double	The smallest element in `x`
`sum`	(x:Double[]) : Double	Sum of the elements in `x`
`average`	(x:Double[]) : Double	Arithmetic mean

### Math functions

Name	Signature	Description
`acos`	(x:Double) : Double	The inverse cosine of x
`asin`	(x:Double) : Double	The inverse sine of x
`atan`	(x:Double) : Double	The inverse tangent of x
`atan2`	(x:Double, y:Double) : Double	The inverse tangent of `x / y`
`cos`	(x:Double) : Double	The cosine of x
`cosh`	(x:Double) : Double	The hyperbolic cosine of x
`sin`	(x:Double) : Double	The sine of x
`sinh`	(x:Double) : Double	The hyperbolic sine of x
`tan`	(x:Double) : Double	The tangent of x
`tanh`	(x:Double) : Double	The hyperbolic tangent of x

### Other functions

Name	Signature	Description
`toString`	`(x:Any) : String`	Prints any value to a string
`length`	`(x:String) : Integer`	Prints length of a string
`failure`	`(x:String) : ErgoErrorResponse`	Ergo error from a string

## Time

The following functions are in the `org.accordproject.time` namespace and are available by importing that namespace.
They rely on the [time.cto](https://models.accordproject.org/v2.0/time.html) types from the Accord Project models.

### Functions on DateTime

Name	Signature	Description
`now`	`() : DateTime`	Returns the time when execution started
`dateTime`	`(x:String) : DateTime`	Parse a date and time
`getSecond`	`(x:DateTime) : Long`	Second component of a DateTime
`getMinute`	`(x:DateTime) : Long`	Minute component of a DateTime
`getHour`	`(x:DateTime) : Long`	Hour component of a DateTime
`getDay`	`(x:DateTime) : Long`	Day of the month component of a DateTime
`getWeek`	`(x:DateTime) : Long`	Week of the year component of a DateTime
`getMonth`	`(x:DateTime) : Long`	Month component in a DateTime

`getYear`	`(x:DateTime) : Long`	Year component in a DateTime
`isAfter`	`(x:DateTime, y:DateTime) : Boolean`	Whether `x` if after `y`
`isBefore`	`(x:DateTime, y:DateTime) : Boolean`	Whether `x` is before `y`
`isSame`	`(x:DateTime, y:DateTime) : Boolean`	Whether `x` is the same DateTime as `y`
`dateTimeMin`	`(x:DateTime[]) : DateTime`	The earliest in an array of DateTime
`dateTimeMax`	`(x:DateTime[]) : DateTime`	The latest in an array of DateTime

### Functions on Duration

Name	Signature	Description
`durationAs`	`(x:Duration, y:TemporalUnit) : Duration`	Change the unit for duration `x` to `y`
`diffDurationAs`	`(x:DateTime, y:DateTime, z:TemporalUnit) : Duration`	Duration between `x` and `y` in unit `z`
`diffDuration`	`(x:DateTime, y:DateTime) : Duration`	Duration between `x` and `y` in seconds
`addDuration`	`(x:DateTime, y:Duration) : DateTime`	Add duration `y` to `x`
`subtractDuration`	`(x:DateTime, y:Duration) : DateTime`	Subtract duration `y` to `x`
`divideDuration`	`(x:Duration, y:Duration) : Double`	Ratio between durations `x` and `y`

### Functions on Period

Name	Signature	Description
`diffPeriodAs`	`(x:DateTime, y:DateTime, z:PeriodUnit) : Period`	Time period between `x` and `y` in unit `z`
`addPeriod`	`(x:DateTime, y:Period) : DateTime`	Add time period `y` to `x`
`subtractPeriod`	`(x:DateTime, y:Period) : DateTime`	Subtract time period `y` to `x`
`startOf`	`(x:DateTime, y:PeriodUnit) : DateTime`	Start of period `y` nearest to DateTime `x`
`endOf`	`(x:DateTime, y:PeriodUnit) : DateTime`	End of period `y` nearest to DateTime `x`

---
id: version-0.13-spec-packaging
title: Packaging
original_id: spec-packaging

The artifacts that define a template are:
- Metadata, such as name and version
- CML models, which define the template model, request, response and any required types
- Template grammar for each supported locale
- A sample instance, used to bootstrap editing, for each supported locale
- Executable business logic

Templates are typically packaged and distributed as Cicero Template Archive (`.cta`) files, however they may also be read from: a directory, http(s) URL, the Accord Project [template library](https://templates.accordproject.org), the npm package manager. Each of these distribution mechanisms support slightly different use cases:

- Directory: useful during testing, allows changes to the template to be quickly tested with no need to re-package
- URL: allows templates to be published to a stable web address
- Accord Project template library maintains a curated set of Open Source templates
- npm: allows dependencies on templates to be easily declared for Node.js and browser based applications. Integrates with CI/CD tools.

## Metadata
The metadata for a Template is stored in the /package.json text file in JSON format.

```
{
    "name": "latedeliveryandpenalty",
    "displayName": "Late Delivery and Penalty",
    "version": "0.11.1",
    "description": "A sample Late Delivery And Penalty clause.",
    "accordproject": {
        "template": "clause",
        "ergo": "0.9.0",
        "cicero": "^0.13.0"
    },
    "keywords": ["clause", "delivery", "acceptance", "obligation"]
}
```

The name property must consist of `[a-z][A-Z][.]`. It is strongly recommended that the name be prefixed with the domain name of the author of the smart clause, to minimise naming collisions. The version property must be a semantic version of the form `major.minor.micro [0-9].[0-9].[0.9]`. Note that this data format ensures that a Template can be published to the npm package manager for either global or private (enterprise-wide) distribution.

The `accordproject` property of a template specifies the following metadata:
- `template`: must either be `clause` for a clause template or `contract` for a contract templates
- `ergo`: the Ergo version used to to draft the template. This is an npm [semver](https://semver.npmjs.com) specification.
- `cicero`: the Cicero version with which the template is compatible. This is an npm [semver](https://semver.npmjs.com) specification.

The `keywords` property should list words relevant to the template, and can be useful for search and classification.

Note that additional properties such as locales and jurisdictions may be added as future needs arise.

## README.md
The root of the template may also contain a markdown file to explain the purpose and semantics of the template. This file may be displayed by tools to preview the template or provide usage instructions.

## Template Grammars
The grammar files for the template are stored in the /grammar/ folder.

> Note that support for per-locale grammar files in TBD.

## Data Model

The data model for a smart clause is stored in a set of files under the `/model` folder. The data model files must be in the format defined using the Composer Concerto modeling language. All data models for the template are in-scope and types from all namespaces may be imported.

Using the ability to convert CML models to UML we can even visualise all the required types (model, request, response) we have modelled graphically:

![UML diagram](assets/cicero-spec-uml.png)

## Execution Logic

The Ergo execution logic for a smart clause is stored under the /lib folder.

---
id: version-0.20-accordproject-business
title: For Business
original_id: accordproject-business

## Why is the Accord Project relevant for business?

Contracting is undergoing a digital transformation driven by a need to deliver customer-centric legal and business solutions faster, and at lower cost. This imperative is fueling the adoption of a broad range of new technologies to improve the efficiency of drafting, managing, and executing legal contracting operations; the Accord Project is proud to be part of that movement.

In addition, contributions from businesses are crucial for the development of the Accord Project. The expertise of stakeholders, such as business professionals and attorneys, is invaluable in improving the functionality and content of the Accord Project's codebase and specifications, to ensure that the templates meet real-world business requirements.

If this interests you, please visit our [Lifecycle and Industry Working Groups](https://www.accordproject.org/liwg) page for more information.

## Why is the Accord Project relevant for lawyers?

Even the best code would be useless if it does not meet industry needs and requirements, so input from the legal community to steer the direction of smart legal contract technology is crucial. By creating templates for contract clauses or reviewing templates, lawyers can influence the direction of the Accord Project. In addition, feedback using domain-specific expertise can enhance the value and usability of the Accord Project templates to users.

If this interests you, please visit our [Lifecycle and Industry Working Groups](https://www.accordproject.org/liwg) page for more information.

As an individual, contributing to the Accord Project would be a great opportunity to learn about smart legal contracts. Through the Accord Project, you can understand the foundations of open source technologies and learn how to develop smart agreements.

If your organization wants to become a member of the Accord Project, please [join our community](https://www.accordproject.org/membership).

## How to navigate this documentation?

If you are new to the Accord Project, the best place to start is our [Online Tour](started-studio). This video is a great way to see the Accord Project in action! If you want to read more about the key concepts behind the Accord Project technology (or learn more about the template model), please read the [Key Concepts](accordproject-concepts) page. This will allow you to understand the three components of a template (text, model, and logic) and how they work together.

If any of the technical terms are confusing or hard to understand, we have a [Glossary](ref-glossary) page that may help you. If there are any concepts or terms that you want to include in the Glossary, please join our [slack channel](https://accord-project-slack-signup.herokuapp.com/) and make suggestions!

If you want to create a template yourself, please see [Authoring in Template Studio](tutorial-latedelivery) for a step-by-step guide on how to create your first template. The tutorial will provide an accessible starting point for those without significant development experience to begin building smart legal contracts using the Accord Project technology.

---
id: version-0.20-accordproject-concepts
title: Key Concepts
original_id: accordproject-concepts

## What is a Template?

An Accord Project template ties legal text to computer code. It is composed of three elements:

- **Template Text**: the natural language of the template
- **Template Model**: the data model that backs the template, acting as a bridge between the text and the logic
- **Template Logic**: the executable business logic for the template

![Template](assets/020/template.png)

The three components (Text - Model - Logic) can also be intuitively understood as a **progression**, from _human-readable_ legal text to _machine-readable_ and _machine-executable_. When combined these three elements allow templates to be edited, validated, and then executed on any computer platform (on your own machine, on a Cloud platform, on Blockchain, etc).

> We use the computing term 'executed' here, which means run by a computer. This is distinct from the legal term 'executed', which usually refers to the process of signing an agreement.

### Cicero

The implementation for the Accord Project templates is called [Cicero](https://github.com/accordproject/cicero). It defines and can read the structure of templates, with natural language bound to a data model and logic. By doing this, Cicero allows users to create, validate and execute software templates which embody all three components in the template triangle above.

More information about how to install Cicero and get started with Accord Project templates can be found in the [Installation](started-installation) Section of this documentation.

Let's look at each component of the template triangle, starting with the text.

## Template Text

![Template Text](assets/020/template_text.png)

The template text is the natural language of the clause or contract. It can include markup to indicate [variables](ref-glossary#variable) for that template.

The following shows the text of an **Acceptance of Delivery** clause.

```tem
## Acceptance of Delivery.

{{shipper}} will be deemed to have completed its delivery obligations
if in {{receiver}}'s opinion, the {{deliverable}} satisfies the
Acceptance Criteria, and {{receiver}} notifies {{shipper}} in writing
that it is accepting the {{deliverable}}.

## Inspection and Notice.

{{receiver}} will have {{businessDays}} Business Days to inspect and
evaluate the {{deliverable}} on the delivery date before notifying
{{shipper}} that it is either accepting or rejecting the
{{deliverable}}.

## Acceptance Criteria.

The 'Acceptance Criteria' are the specifications the {{deliverable}}
must meet for the {{shipper}} to comply with its requirements and
obligations under this agreement, detailed in {{attachment}}, attached
to this agreement.
```

The text is written in plain English, with variables between `{{` and `}}`. Variables allows template to be used in different agreements by replacing them with different values.

For instance, the following show the same **Acceptance of Delivery** clause where the `shipper` is `"Party A"`, the `receiver` is `"Party B"`, the `deliverable` is `"Widgets"`, etc.

```md
## Acceptance of Delivery.

"Party A" will be deemed to have completed its delivery obligations
if in "Party B"'s opinion, the "Widgets" satisfies the
Acceptance Criteria, and "Party B" notifies "Party A" in writing
that it is accepting the "Widgets".

## Inspection and Notice.

"Party B" will have 10 Business Days to inspect and
```

~~~
evaluate the "Widgets" on the delivery date before notifying
"Party A" that it is either accepting or rejecting the
"Widgets".

## Acceptance Criteria.

The "Acceptance Criteria" are the specifications the "Widgets"
must meet for the "Party A" to comply with its requirements and
obligations under this agreement, detailed in "Attachment X", attached
to this agreement.
```

### CiceroMark

CiceroMark is the markup format in which the text for Accord Project templates is
written. It defines notations (such as the `{{` and `}}` notation for variables
used in the **Acceptance of Delivery** clause) which allows a computer to make
sense of your templates.

It also provides the ability to specify the document structure (e.g., headings,
lists), to highlight certain terms (e.g., in bold or italics), to indicate text
which is optional in the agreement, and more.

_More information about the Accord Project markup can be found in the [CiceroMark]
(markup-cicero) Section of this documentation._

## Template Model

![Template Model](assets/020/template_model.png)

Unlike a standard document template (e.g., in Word or pdf), Accord Project
templates associate a _model_ to the natural language text. The model acts as a
bridge between the text and logic; it gives the users an overview of the
components, as well as the types of different components.

The model categorizes variables (is it a number, a monetary amount, a date, a
reference to a business or organization, etc.). This is crucial as it allows the
computer to make sense of the information contained in the template.

The following shows the model for the **Acceptance of Delivery** clause.

```ergo
/* The template model */
asset AcceptanceOfDeliveryClause extends AccordClause {

  /* the shipper of the goods*/
  --> Organization shipper

  /* the receiver of the goods */
  --> Organization receiver

  /* what we are delivering */
  o String deliverable

  /* how long does the receiver have to inspect the goods */
  o Integer businessDays

  /* additional information */
  o String attachment
~~~

}

Thanks to that model, the computer knows that the `shipper` variable (`"Party A"` in the example) and the `receiver` variable (`"Party B"` in the example) are both `Organization` types. The computer also knows that variable `businessDays` (`10` in the example) is an `Integer` type; and that the variable `deliverable` (`"Widgets"` in the example) is a `String` type, and can contain any text description.

> If you are unfamiliar with the different types of variables, or want a more thorough explanation of what variables are, please refer to our [Glossary](ref-glossary#data-models) for a more detailed explanation.

### Concerto

Concerto is the language which is used to write models in Accord Project templates. Concerto offers modern modeling capabilities including support for primitive types (numbers, dates, etc), nested or optional data structures, enumerations, relationships, object-oriented style inheritance, and more.

More information about Concerto can be found in the [Concerto Modeling](model-concerto) section of this documentation.

## Template Logic

![Template Logic](assets/020/template_logic.png)

The combination of text and model already makes templates _machine-readable_, while the logic makes it _machine-executable_.

### During Drafting

In the [Overview](accordproject) Section, we already saw how logic can be embedded in the text of the template itself to automatically calculate a monthly payment for a [fixed rate loan]():

```tem
## Fixed rate loan

This is a *fixed interest* loan to the amount of {{loanAmount}}
at a yearly interest rate of {{rate}}%
with a loan term of {{loanDuration}},
and monthly payments of {{% monthlyPaymentFormula(loanAmount,rate,loanDuration) %}}.
```

This uses a `monthlyPaymentFormula` function which calculates the monthly payment based on the other data points in the text:
```ergo
define function monthlyPaymentFormula(loanAmount: Double, rate: Double, loanDuration: Integer) : Double {
  let term = longToDouble(loanDuration * 12);       // Term in months
  if (rate = 0.0) then return (loanAmount / term)   // If the rate is 0
  else
    let monthlyRate = (rate / 12.0) / 100.0;        // Rate in months
    let monthlyPayment =                            // Payment calculation
      (monthlyRate * loanAmount)
      / (1.0 - ((1.0 + monthlyRate) ^ (-term)));
    return roundn(monthlyPayment, 0)                // Rounding
```

}
~~~
Each logic function has a _name_ (e.g., `monthlyPayment`), a _signature_ indicating the parameters with their types (e.g., `loanAmount:Double`), and a _body_ which performs the appropriate computation based on the parameters. The main payment calculation is here based on the [standardized calculation used in the United States](https://en.wikipedia.org/wiki/Mortgage_calculator#Monthly_payment_formula) with `*` standing for multiplication, `/` for division, and `^` for exponentiation.

### After Signature

The logic can also be used to associate behavior to the template _after_ the contract has been signed. This can be used for instance to specify what happens when a delivery is received late, to check conditions for payment, determine if there has been a breach of contract, etc.

The following shows post-signature logic for the **Acceptance of Delivery** clause.

```ergo
contract SupplyAgreement over SupplyAgreementModel {
  clause acceptanceofdelivery(request : InspectDeliverable) : InspectionResponse {

    let received = request.deliverableReceivedAt;
    enforce isBefore(received,now()) else
      throw ErgoErrorResponse{ message : "Transaction time is before the 
deliverable date." }
    ;

    let status =
      if isAfter(now(), addDuration(received, Duration{ amount: 
contract.businessDays, unit: ~org.accordproject.time.TemporalUnit.days}))
      then OUTSIDE_INSPECTION_PERIOD
      else if request.inspectionPassed
      then PASSED_TESTING
      else FAILED_TESTING
    ;
    return InspectionResponse{
      status : status,
      shipper : contract.shipper,
      receiver : contract.receiver
    }
  }
}
~~~

This logic describes what conditions must be met for a delivery to be accepted. It checks whether the delivery has already been made; whether the acceptance is timely, within the specified inspection date; and whether the inspection has passed or not.

### Ergo

Ergo is the programming language which is used to express contractual logic in templates. Ergo is specifically designed for legal agreements, and is intended to be accessible for those creating the corresponding prose for those computable legal contracts. Ergo expressions can also be embedded in the text for a template.

More information about Ergo can be found in the [Ergo Logic](logic-ergo) Section of this documentation.

## What next?

Build your first smart legal contract templates, either [online](tutorial-latedelivery) with Template Studio, or by [installing Cicero](started-installation).

Explore [sample templates](started-resources) and other resources in the rest of this documentation.

If some of technical words are unfamiliar, please consult the [Glossary](ref-glossary) for more detailed explanations.

---
id: version-0.20-accordproject-developers
title: For Developers
original_id: accordproject-developers

## Why is the Accord Project relevant for developers?

The Accord Project provides a universal format for smart legal contracts, and this format is embodied in a variety of open source projects that comprise the Accord Project technology stack. Input from developers are crucial for the Accord Project.

Developers can contribute by converting legal text into corresponding computer code, creating Accord Project templates to be used by lawyers and businesses. In addition, developers can provide input on the development of its technology stack: language, models, templating, and other tools.

If this interests you, please visit our [Technology Working Group](https://www.accordproject.org/working-groups/technology) page, and join our [slack channel](https://accord-project-slack-signup.herokuapp.com/)!

## How to navigate this documentation?

If you want to author, validate, and run Accord Project templates locally, please visit our [Install Cicero](https://docs.accordproject.org/docs/next/started-installation.html) page for instructions.

If you are new to the Accord Project, please read the [Key Concepts](accordproject-concepts) page. This will allow you to understand the three components of a template (text, model, and logic) and how they work together. If you want some guidance on creating your first template, please see [Authoring in Template Studio](tutorial-latedelivery) for a step-by-step guide on how to create your first template.

If you want to dive into our technology stack, you can find more information about:
- Software implementation: [Cicero](https://github.com/accordproject/cicero)
- Template text: [CiceroMark](markup-cicero)
- Template model: [Concerto Modeling](model-concerto)
- Template logic: [Ergo Logic](logic-ergo)

---
id: version-0.20-accordproject
title: Overview

original_id: accordproject

## What is the Accord Project?

Accord Project is an open source, non-profit initiative aimed at transforming contract management and contract automation by digitizing contracts.

The Accord Project defines a notion of a legal template with associated computing logic which is expressive, open-source, and portable. Accord Project templates are similar to a clause or contract template in any document format, but they can be read, interpreted, and run by a computer.

The goal of the Accord Project is to provide an open, standardized format for Smart Legal Contracts.

## What is a Smart Legal Contract?

A Smart Legal Contract is a human-readable _and_ machine-readable agreement that is digital, consisting of natural language and computable components.

The human-readable nature of the document ensures that signatories, lawyers, contracting parties and others are able to understand the contract.

The machine-readable nature of the document enables it to be interpreted and executed by computers, making the document "smart".

Contracts drafted with Accord Project can contain both traditional and machine-readable clauses. For example, a Smart Legal Contract may include a smart payment clause while all of the other provisions of the contract (Definitions, Jurisdiction clause, Force Majeure clause, ...) are being documented solely in regular natural language text.

A Smart Legal Contract is a general term to refer to two compatible, architectural forms of contract:
- Machine-Readable Contracts, which tie legal text to data
- Machine-Executable Contracts, which tie legal text to data and executable code

### Machine-Readable Contracts

By combining Text and a data, a clause or contract becomes machine-readable.

For instance, the clause below for a [fixed rate loan](https://templates.accordproject.org/fixed-interests-static@0.2.0.html) includes natural language text coupled with variables. Together, these variables refer to some data for the clause and correspond to the 'deal points':

```tem
## Fixed rate loan

This is a *fixed interest* loan to the amount of {{loanAmount}}
at the yearly interest rate of {{rate}}%
with a loan term of {{loanDuration}},
and monthly payments of {{monthlyPayment}}.
```

To make sense of the data, a _Data Model_, expressed in the Concerto schema language, defines the variables for the template and their associated Data Types:

```ergo
  o Double loanAmount     // loanAmount is a floating-point number
  o Double rate           // rate is a floating-point number
  o Integer loanDuration  // loanDuration is an integer
  o Double monthlyPayment // monthlyPayment is a floating-point number
```

The Data Types allow a computer to validate values inserted into each of the `{{variable}}` placeholders (e.g., `2.5` is a valid `{{rate}}` but `January` isn't). In other words, the Data Model lets a computer make sense of the structure of (and data in) the clause. To learn more about Data Types see [Concerto Modeling](model-concerto).

The clause data (the 'deal points') can then be capture as a machine-readable representation:

```js
{
  "$class": "org.accordproject.interests.TemplateModel",
  "clauseId": "cec0a194-cd45-42f7-ab3e-7a673978602a",
  "loanAmount": 100000.0,
  "rate": 2.5,
  "loanDuration": 15
  "monthlyPayment": 667.0
}
```

The values entered into the template text are associated with the name of the variable e.g. `{{rate}} = 2.5%`. This provides the structure for understanding the clause and its contents.

### Machine-Executable Contracts

By adding Logic to a machine-readable clause or contract in the form of expressions - much like in a spreadsheet - the contract is able to execute operations based upon data included in the contract.

For instance, the clause below is a variant of the earlier [fixed rate loan](https://templates.accordproject.org/fixed-interests@0.2.0.html). While it is consistent with the previous one, the `{{monthlyPayment}}` variable is replaced with an [Ergo](logic-ergo) expression `monthlyPaymentFormula(loanAmount,rate,loanDuration)` which calculates the monthly interest rate based upon the values of the other variables: `{{loanAmount}}`, `{{rate}}`, and `{{loanDuration}}`. To learn more about contract Logic see [Ergo Logic](logic-ergo).

```tem
## Fixed rate loan

This is a *fixed interest* loan to the amount of {{loanAmount}}
at a yearly interest rate of {{rate}}%
with a loan term of {{loanDuration}},
and monthly payments of {{% monthlyPaymentFormula(loanAmount,rate,loanDuration)
%}}.
```

This is a simple example of the benefits of Machine-Executable contract, here adding logic to ensure that the value of the `{{monthlyPayment}}` in the text is always consistent with the other variables in the clause. In this example, we display the contract text using the underlying [CiceroMark](markup-cicero) format, instead of the rich-text output that would be found in [editor tools](started-resources#ecosystem-tools) and PDF outputs.

More complex examples, (e.g., how to add post-signature logic which responds to data sent to the contract or which triggers operations on external systems) can be found in the rest of this documentation.

## What are the Benefits of Smart Legal Contracts?

Smart Legal Contracts can be easily searched, analyzed, queried, and understood. By associating a data model to a contract, it is possible to extract a host of valuable data about a contract or draft a series of contracts from existing data points (i.e., variables and their values).

The data model is used to ensure that all of the necessary data is present in the contract, and that this data is valid. In addition, it provides the necessary structure to enable contracts to "come alive" by adding executable logic.

For more information about Smart Legal Contracts, and how they are different from other kinds of "smart contracts", please visit the [Accord Project FAQ](https://www.accordproject.org/frequently-asked-questions).

---
id: version-0.20-cicero-api
title: Cicero API
original_id: cicero-api

## Modules

<dl>
<dt><a href="#module_cicero-engine">cicero-engine</a></dt>
<dd><p>Clause Engine</p>
</dd>
<dt><a href="#module_cicero-core">cicero-core</a></dt>
<dd><p>Cicero Core - defines the core data types for Cicero.</p>
</dd>
</dl>

## Classes

<dl>
<dt><a href="#Clause">Clause</a></dt>
<dd><p>A Clause is executable business logic, linked to a natural language (legally enforceable) template.
A Clause must be constructed with a template and then prior to execution the data for the clause must be set.
Set the data for the clause (an instance of the template model) by either calling the setData method or by
calling the parse method and passing in natural language text that conforms to the template grammar.</p>
</dd>
<dt><a href="#Contract">Contract</a></dt>
<dd><p>A Contract is executable business logic, linked to a natural language (legally enforceable) template.
A Clause must be constructed with a template and then prior to execution the data for the clause must be set.

Set the data for the clause (an instance of the template model) by either calling the setData method or by
calling the parse method and passing in natural language text that conforms to the template grammar.</p>
</dd>
<dt><a href="#DateTimeFormatParser">DateTimeFormatParser</a></dt>
<dd><p>Parses a date/time format string</p>
</dd>
<dt><a href="#Metadata">Metadata</a></dt>
<dd><p>Defines the metadata for a Template, including the name, version, README markdown.</p>
</dd>
<dt><a href="#ParserManager">ParserManager</a></dt>
<dd><p>Generates and manages a Nearley parser for a template.</p>
</dd>
<dt><a href="#Template">Template</a></dt>
<dd><p>A template for a legal clause or contract. A Template has a template model, request/response transaction types,
a template grammar (natural language for the template) as well as Ergo code for the business logic of the
template.</p>
</dd>
<dt><a href="#TemplateInstance">TemplateInstance</a></dt>
<dd><p>A TemplateInstance is an instance of a Clause or Contract template. It is executable business logic, linked to
a natural language (legally enforceable) template.
A TemplateInstance must be constructed with a template and then prior to execution the data for the clause must be set.
Set the data for the TemplateInstance by either calling the setData method or by
calling the parse method and passing in natural language text that conforms to the template grammar.</p>
</dd>
<dt><a href="#CompositeArchiveLoader">CompositeArchiveLoader</a></dt>
<dd><p>Manages a set of archive loaders, delegating to the first archive
loader that accepts a URL.</p>
</dd>
</dl>

## Functions

<dl>
<dt><a href="#locationOfError">locationOfError(error)</a> ⇒
<code>object</code></dt>
<dd><p>Extract the file location from the parse error</p>
</dd>
</dl>

<a name="module_cicero-engine"></a>

## cicero-engine
Clause Engine

* [cicero-engine](#module_cicero-engine)
    * [~Engine](#module_cicero-engine.Engine)
        * [new Engine()](#new_module_cicero-engine.Engine_new)
        * [.trigger(clause, request, state, currentTime)](#module_cicero-engine.Engine+trigger) ⇒ <code>Promise</code>
        * [.invoke(clause, clauseName, params, state, currentTime)](#module_cicero-

```
engine.Engine+invoke) ⇒ <code>Promise</code>
        * [.init(clause, currentTime)](#module_cicero-engine.Engine+init) ⇒
<code>Promise</code>
        * [.draft(clause, [options], currentTime)](#module_cicero-
engine.Engine+draft) ⇒ <code>Promise</code>
        * [.getErgoEngine()](#module_cicero-engine.Engine+getErgoEngine) ⇒
<code>ErgoEngine</code>

<a name="module_cicero-engine.Engine"></a>

### cicero-engine~Engine
<p>
Engine class. Stateless execution of clauses against a request object, returning a
response to the caller.
</p>

**Kind**: inner class of [<code>cicero-engine</code>](#module_cicero-engine)
**Access**: public

* [~Engine](#module_cicero-engine.Engine)
    * [new Engine()](#new_module_cicero-engine.Engine_new)
    * [.trigger(clause, request, state, currentTime)](#module_cicero-
engine.Engine+trigger) ⇒ <code>Promise</code>
    * [.invoke(clause, clauseName, params, state, currentTime)](#module_cicero-
engine.Engine+invoke) ⇒ <code>Promise</code>
    * [.init(clause, currentTime)](#module_cicero-engine.Engine+init) ⇒
<code>Promise</code>
    * [.draft(clause, [options], currentTime)](#module_cicero-engine.Engine+draft)
⇒ <code>Promise</code>
    * [.getErgoEngine()](#module_cicero-engine.Engine+getErgoEngine) ⇒
<code>ErgoEngine</code>

<a name="new_module_cicero-engine.Engine_new"></a>

#### new Engine()
Create the Engine.

<a name="module_cicero-engine.Engine+trigger"></a>

#### engine.trigger(clause, request, state, currentTime) ⇒ <code>Promise</code>
Send a request to a clause for execution

**Kind**: instance method of [<code>Engine</code>](#module_cicero-engine.Engine)
**Returns**: <code>Promise</code> - a promise that resolves to a result for the
clause

| Param | Type | Description |
| --- | --- | --- |
| clause | [<code>Clause</code>](#Clause) | the clause |
| request | <code>object</code> | the request, a JS object that can be deserialized
using the Composer serializer. |
| state | <code>object</code> | the contract state, a JS object that can be
deserialized using the Composer serializer. |
| currentTime | <code>string</code> | the definition of 'now' |

<a name="module_cicero-engine.Engine+invoke"></a>

#### engine.invoke(clause, clauseName, params, state, currentTime) ⇒
<code>Promise</code>
```

Invoke a specific clause for execution

Kind: instance method of [<code>Engine</code>](#module_cicero-engine.Engine)
Returns: <code>Promise</code> - a promise that resolves to a result for the
clause

Param	Type	Description
clause	[<code>Clause</code>](#Clause)	the clause
clauseName	<code>string</code>	the clause name
params	<code>object</code>	the clause parameters, a JS object whose fields
that can be deserialized using the Composer serializer.
state	<code>object</code>	the contract state, a JS object that can be
deserialized using the Composer serializer.
currentTime	<code>string</code>	the definition of 'now'

<a name="module_cicero-engine.Engine+init"></a>

#### engine.init(clause, currentTime) ⇒ <code>Promise</code>
Initialize a clause

Kind: instance method of [<code>Engine</code>](#module_cicero-engine.Engine)
Returns: <code>Promise</code> - a promise that resolves to a result for the
clause initialization

Param	Type	Description
clause	[<code>Clause</code>](#Clause)	the clause
currentTime	<code>string</code>	the definition of 'now'

<a name="module_cicero-engine.Engine+draft"></a>

#### engine.draft(clause, [options], currentTime) ⇒ <code>Promise</code>
Generate Text for a clause

Kind: instance method of [<code>Engine</code>](#module_cicero-engine.Engine)
Returns: <code>Promise</code> - a promise that resolves to a result for the
clause initialization

Param	Type	Description
clause	[<code>Clause</code>](#Clause)	the clause
[options]	<code>*</code>	text generation options. options.wrapVariables
encloses variables and editable sections in '<variable ...' and '/>'
currentTime	<code>string</code>	the definition of 'now'

<a name="module_cicero-engine.Engine+getErgoEngine"></a>

#### engine.getErgoEngine() ⇒ <code>ErgoEngine</code>
Provides access to the underlying Ergo engine.

Kind: instance method of [<code>Engine</code>](#module_cicero-engine.Engine)
Returns: <code>ErgoEngine</code> - the Ergo Engine for this Engine
<a name="module_cicero-core"></a>

## cicero-core
Cicero Core - defines the core data types for Cicero.

<a name="Clause"></a>

## Clause
A Clause is executable business logic, linked to a natural language (legally
enforceable) template.
A Clause must be constructed with a template and then prior to execution the data
for the clause must be set.
Set the data for the clause (an instance of the template model) by either calling
the setData method or by
calling the parse method and passing in natural language text that conforms to the
template grammar.

Kind: global class
Access: public
<a name="Contract"></a>

## Contract
A Contract is executable business logic, linked to a natural language (legally
enforceable) template.
A Clause must be constructed with a template and then prior to execution the data
for the clause must be set.
Set the data for the clause (an instance of the template model) by either calling
the setData method or by
calling the parse method and passing in natural language text that conforms to the
template grammar.

Kind: global class
Access: public
<a name="DateTimeFormatParser"></a>

## DateTimeFormatParser
Parses a date/time format string

Kind: global class
Access: public

* [DateTimeFormatParser](#DateTimeFormatParser)
    * [.parseDateTimeFormatField(field)]
(#DateTimeFormatParser.parseDateTimeFormatField) ⇒ <code>string</code>
    * [.buildDateTimeFormatRule(formatString)]
(#DateTimeFormatParser.buildDateTimeFormatRule) ⇒ <code>Object</code>

<a name="DateTimeFormatParser.parseDateTimeFormatField"></a>

### DateTimeFormatParser.parseDateTimeFormatField(field) ⇒ <code>string</code>
Given a format field (like HH or D) this method returns
a logical name for the field. Note the logical names
have been picked to align with the moment constructor that takes an object.

Kind: static method of [<code>DateTimeFormatParser</code>]
(#DateTimeFormatParser)
Returns: <code>string</code> - the field designator

Param	Type	Description
field	<code>string</code>	the input format field

<a name="DateTimeFormatParser.buildDateTimeFormatRule"></a>

### DateTimeFormatParser.buildDateTimeFormatRule(formatString) ⇒

```
<code>Object</code>
Converts a format string to a Nearley action

**Kind**: static method of [<code>DateTimeFormatParser</code>]
(#DateTimeFormatParser)
**Returns**: <code>Object</code> - the tokens and action and name to use for the
Nearley rule

| Param | Type | Description |
| --- | --- | --- |
| formatString | <code>string</code> | the input format string |

<a name="Metadata"></a>

## Metadata
Defines the metadata for a Template, including the name, version, README markdown.

**Kind**: global class
**Access**: public

* [Metadata](#Metadata)
    * [new Metadata(packageJson, readme, samples, request)](#new_Metadata_new)
    * [.getTemplateType()](#Metadata+getTemplateType) ⇒ <code>number</code>
    * [.getRuntime()](#Metadata+getRuntime) ⇒ <code>string</code>
    * [.getCiceroVersion()](#Metadata+getCiceroVersion) ⇒ <code>string</code>
    * [.satisfiesCiceroVersion(version)](#Metadata+satisfiesCiceroVersion) ⇒
<code>string</code>
    * [.getSamples()](#Metadata+getSamples) ⇒ <code>object</code>
    * [.getRequest()](#Metadata+getRequest) ⇒ <code>object</code>
    * [.getSample(locale)](#Metadata+getSample) ⇒ <code>string</code>
    * [.getREADME()](#Metadata+getREADME) ⇒ <code>String</code>
    * [.getPackageJson()](#Metadata+getPackageJson) ⇒ <code>object</code>
    * [.getName()](#Metadata+getName) ⇒ <code>string</code>
    * [.getDisplayName()](#Metadata+getDisplayName) ⇒ <code>string</code>
    * [.getKeywords()](#Metadata+getKeywords) ⇒ <code>Array</code>
    * [.getDescription()](#Metadata+getDescription) ⇒ <code>string</code>
    * [.getVersion()](#Metadata+getVersion) ⇒ <code>string</code>
    * [.getIdentifier()](#Metadata+getIdentifier) ⇒ <code>string</code>
    * [.createTargetMetadata(runtimeName)](#Metadata+createTargetMetadata) ⇒
<code>object</code>

<a name="new_Metadata_new"></a>

### new Metadata(packageJson, readme, samples, request)
Create the Metadata.
<p>
<strong>Note: Only to be called by framework code. Applications should
retrieve instances from [Template](#Template)</strong>
</p>


| Param | Type | Description |
| --- | --- | --- |
| packageJson | <code>object</code> | the JS object for package.json (required) |
| readme | <code>String</code> | the README.md for the template (may be null) |
| samples | <code>object</code> | the sample markdown for the template in different
locales, |
| request | <code>object</code> | the JS object for the sample request represented
as an object whose keys are the locales and whose values are the sample markdown.
```

```
For example:  {      default: 'default sample markdown',      en: 'sample text in
english',      fr: 'exemple de texte français'  } Locale keys (with the exception
of default) conform to the IETF Language Tag specification (BCP 47). THe `default`
key represents sample template text in a non-specified language, stored in a file
called `sample.md`. |

<a name="Metadata+getTemplateType"></a>

### metadata.getTemplateType() ⇒ <code>number</code>
Returns either a 0 (for a contract template), or 1 (for a clause template)

**Kind**: instance method of [<code>Metadata</code>](#Metadata)
**Returns**: <code>number</code> - the template type
<a name="Metadata+getRuntime"></a>

### metadata.getRuntime() ⇒ <code>string</code>
Returns the name of the runtime target for this template, or null if this template
has not been compiled for a specific runtime.

**Kind**: instance method of [<code>Metadata</code>](#Metadata)
**Returns**: <code>string</code> - the name of the runtime
<a name="Metadata+getCiceroVersion"></a>

### metadata.getCiceroVersion() ⇒ <code>string</code>
Returns the version of Cicero that this template is compatible with.
i.e. which version of the runtime was this template built for?
The version string conforms to the semver definition

**Kind**: instance method of [<code>Metadata</code>](#Metadata)
**Returns**: <code>string</code> - the semantic version
<a name="Metadata+satisfiesCiceroVersion"></a>

### metadata.satisfiesCiceroVersion(version) ⇒ <code>string</code>
Only returns true if the current cicero version satisfies the target version of
this template

**Kind**: instance method of [<code>Metadata</code>](#Metadata)
**Returns**: <code>string</code> - the semantic version

| Param | Type | Description |
| --- | --- | --- |
| version | <code>string</code> | the cicero version to check against |

<a name="Metadata+getSamples"></a>

### metadata.getSamples() ⇒ <code>object</code>
Returns the samples for this template.

**Kind**: instance method of [<code>Metadata</code>](#Metadata)
**Returns**: <code>object</code> - the sample files for the template
<a name="Metadata+getRequest"></a>

### metadata.getRequest() ⇒ <code>object</code>
Returns the sample request for this template.

**Kind**: instance method of [<code>Metadata</code>](#Metadata)
**Returns**: <code>object</code> - the sample request for the template
<a name="Metadata+getSample"></a>
```

### metadata.getSample(locale) ⇒ <code>string</code>
Returns the sample for this template in the given locale. This may be null.
If no locale is specified returns the default sample if it has been specified.

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>string</code> - the sample file for the template in the given
locale or null

Param	Type	Default	Description
locale	<code>string</code>	<code>null</code>	the IETF language code for the
language.

<a name="Metadata+getREADME"></a>

### metadata.getREADME() ⇒ <code>String</code>
Returns the README.md for this template. This may be null if the template does not
have a README.md

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>String</code> - the README.md file for the template or null
<a name="Metadata+getPackageJson"></a>

### metadata.getPackageJson() ⇒ <code>object</code>
Returns the package.json for this template.

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>object</code> - the Javascript object for package.json
<a name="Metadata+getName"></a>

### metadata.getName() ⇒ <code>string</code>
Returns the name for this template.

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>string</code> - the name of the template
<a name="Metadata+getDisplayName"></a>

### metadata.getDisplayName() ⇒ <code>string</code>
Returns the display name for this template.

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>string</code> - the display name of the template
<a name="Metadata+getKeywords"></a>

### metadata.getKeywords() ⇒ <code>Array</code>
Returns the name for this template.

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>Array</code> - the name of the template
<a name="Metadata+getDescription"></a>

### metadata.getDescription() ⇒ <code>string</code>
Returns the description for this template.

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>string</code> - the description of the template
<a name="Metadata+getVersion"></a>

### metadata.getVersion() ⇒ <code>string</code>

Returns the version for this template.

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>string</code> - the description of the template
<a name="Metadata+getIdentifier"></a>

### metadata.getIdentifier() ⇒ <code>string</code>
Returns the identifier for this template, formed from name@version.

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>string</code> - the identifier of the template
<a name="Metadata+createTargetMetadata"></a>

### metadata.createTargetMetadata(runtimeName) ⇒ <code>object</code>
Return new Metadata for a target runtime

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>object</code> - the new Metadata

Param	Type	Description
runtimeName	<code>string</code>	the target runtime name

<a name="ParserManager"></a>

## ParserManager
Generates and manages a Nearley parser for a template.

Kind: global class

* [ParserManager](#ParserManager)
    * [new ParserManager(template)](#new_ParserManager_new)
    * _instance_
        * [.getParser()](#ParserManager+getParser) ⇒ <code>object</code>
        * [.getTemplateAst()](#ParserManager+getTemplateAst) ⇒ <code>object</code>
        * [.setGrammar(grammar)](#ParserManager+setGrammar)
        * [.buildGrammar(templatizedGrammar)](#ParserManager+buildGrammar)
        * [.buildGrammarRules(ast, templateModel, prefix, parts)](#ParserManager+buildGrammarRules)
        * [.handleBinding(templateModel, parts, inputRule, element)](#ParserManager+handleBinding)
        * [.cleanChunk(input)](#ParserManager+cleanChunk) ⇒ <code>string</code>
        * [.findFirstBinding(propertyName, elements)](#ParserManager+findFirstBinding) ⇒ <code>int</code>
        * [.getGrammar()](#ParserManager+getGrammar) ⇒ <code>String</code>
        * [.getTemplatizedGrammar()](#ParserManager+getTemplatizedGrammar) ⇒ <code>String</code>
        * [.roundtripMarkdown(text)](#ParserManager+roundtripMarkdown) ⇒ <code>string</code>
    * _static_
        * [.adjustListBlock(x, separator)](#ParserManager.adjustListBlock) ⇒ <code>object</code>
        * [.getProperty(templateModel, element)](#ParserManager.getProperty) ⇒ <code>*</code>
        * [._throwTemplateExceptionForElement(message, element)](#ParserManager._throwTemplateExceptionForElement)
        * [.compileGrammar(sourceCode)](#ParserManager.compileGrammar) ⇒ <code>object</code>

<a name="new_ParserManager_new"></a>

### new ParserManager(template)
Create the ParserManager.

Param	Type	Description
template	<code>object</code>	the template instance

<a name="ParserManager+getParser"></a>

### parserManager.getParser() ⇒ <code>object</code>
Gets a parser object for this template

Kind: instance method of [<code>ParserManager</code>](#ParserManager)
Returns: <code>object</code> - the parser for this template
<a name="ParserManager+getTemplateAst"></a>

### parserManager.getTemplateAst() ⇒ <code>object</code>
Gets the AST for the template

Kind: instance method of [<code>ParserManager</code>](#ParserManager)
Returns: <code>object</code> - the AST for the template
<a name="ParserManager+setGrammar"></a>

### parserManager.setGrammar(grammar)
Set the grammar for the template

Kind: instance method of [<code>ParserManager</code>](#ParserManager)

Param	Type	Description
grammar	<code>String</code>	the grammar for the template

<a name="ParserManager+buildGrammar"></a>

### parserManager.buildGrammar(templatizedGrammar)
Build a grammar from a template

Kind: instance method of [<code>ParserManager</code>](#ParserManager)

Param	Type	Description
templatizedGrammar	<code>String</code>	the annotated template using the markdown parser

<a name="ParserManager+buildGrammarRules"></a>

### parserManager.buildGrammarRules(ast, templateModel, prefix, parts)
Build grammar rules from a template

Kind: instance method of [<code>ParserManager</code>](#ParserManager)

Param	Type	Description
ast	<code>object</code>	the AST from which to build the grammar
templateModel	<code>ClassDeclaration</code>	the type of the parent class for this AST

```markdown
| prefix | <code>String</code> | A unique prefix for the grammar rules |
| parts | <code>Object</code> | Result object to acculumate rules and required sub-
grammars |

<a name="ParserManager+handleBinding"></a>

### parserManager.handleBinding(templateModel, parts, inputRule, element)
Utility method to generate a grammar rule for a variable binding

**Kind**: instance method of [<code>ParserManager</code>](#ParserManager)

| Param | Type | Description |
| --- | --- | --- |
| templateModel | <code>ClassDeclaration</code> | the current template model |
| parts | <code>\*</code> | the parts, where the rule will be added |
| inputRule | <code>\*</code> | the rule we are processing in the AST |
| element | <code>\*</code> | the current element in the AST |

<a name="ParserManager+cleanChunk"></a>

### parserManager.cleanChunk(input) ⇒ <code>string</code>
Cleans a chunk of text to make it safe to include
as a grammar rule. We need to remove linefeeds and
escape any '"' characters.

**Kind**: instance method of [<code>ParserManager</code>](#ParserManager)
**Returns**: <code>string</code> - cleaned text

| Param | Type | Description |
| --- | --- | --- |
| input | <code>string</code> | the input text from the template |

<a name="ParserManager+findFirstBinding"></a>

### parserManager.findFirstBinding(propertyName, elements) ⇒ <code>int</code>
Finds the first binding for the given property

**Kind**: instance method of [<code>ParserManager</code>](#ParserManager)
**Returns**: <code>int</code> - the index of the element or -1

| Param | Type | Description |
| --- | --- | --- |
| propertyName | <code>string</code> | the name of the property |
| elements | <code>Array.&lt;object&gt;</code> | the result of parsing the
template_txt. |

<a name="ParserManager+getGrammar"></a>

### parserManager.getGrammar() ⇒ <code>String</code>
Get the (compiled) grammar for the template

**Kind**: instance method of [<code>ParserManager</code>](#ParserManager)
**Returns**: <code>String</code> - - the grammar for the template
<a name="ParserManager+getTemplatizedGrammar"></a>

### parserManager.getTemplatizedGrammar() ⇒ <code>String</code>
Returns the templatized grammar

**Kind**: instance method of [<code>ParserManager</code>](#ParserManager)
```

Returns: <code>String</code> - the contents of the templatized grammar
<a name="ParserManager+roundtripMarkdown"></a>

### parserManager.roundtripMarkdown(text) ⇒ <code>string</code>
Round-trip markdown

Kind: instance method of [<code>ParserManager</code>](#ParserManager)
Returns: <code>string</code> - the result of parsing and printing back the text

Param	Type	Description
text	<code>string</code>	the markdown text

<a name="ParserManager.adjustListBlock"></a>

### ParserManager.adjustListBlock(x, separator) ⇒ <code>object</code>
Adjust the template for list blocks

Kind: static method of [<code>ParserManager</code>](#ParserManager)
Returns: <code>object</code> - the new template AST node

Param	Type	Description
x	<code>object</code>	The current template AST node
separator	<code>String</code>	The list separator

<a name="ParserManager.getProperty"></a>

### ParserManager.getProperty(templateModel, element) ⇒ <code>*</code>
Throws an error if a template variable doesn't exist on the model.

Kind: static method of [<code>ParserManager</code>](#ParserManager)
Returns: <code>*</code> - the property

Param	Type	Description
templateModel	<code>*</code>	the model for the template
element	<code>*</code>	the current element in the AST

<a name="ParserManager._throwTemplateExceptionForElement"></a>

### ParserManager._throwTemplateExceptionForElement(message, element)
Throw a template exception for the element

Kind: static method of [<code>ParserManager</code>](#ParserManager)
Throws:

- <code>TemplateException</code>

Param	Type	Description
message	<code>string</code>	the error message
element	<code>object</code>	the AST

<a name="ParserManager.compileGrammar"></a>

### ParserManager.compileGrammar(sourceCode) ⇒ <code>object</code>
Compiles a Nearley grammar to its AST

```
**Kind**: static method of [<code>ParserManager</code>](#ParserManager)
**Returns**: <code>object</code> - the AST for the grammar

| Param | Type | Description |
| --- | --- | --- |
| sourceCode | <code>string</code> | the source text for the grammar |

<a name="Template"></a>

## *Template*
A template for a legal clause or contract. A Template has a template model,
request/response transaction types,
a template grammar (natural language for the template) as well as Ergo code for the
business logic of the
template.

**Kind**: global abstract class
**Access**: public

* *[Template](#Template)*
    * *[new Template(packageJson, readme, samples, request, options)]
(#new_Template_new)*
    * _instance_
        * *[.validate()](#Template+validate)*
        * *[.getTemplateModel()](#Template+getTemplateModel) ⇒
<code>ClassDeclaration</code>*
        * *[.getIdentifier()](#Template+getIdentifier) ⇒ <code>String</code>*
        * *[.getMetadata()](#Template+getMetadata) ⇒ [<code>Metadata</code>]
(#Metadata)*
        * *[.getName()](#Template+getName) ⇒ <code>String</code>*
        * *[.getDisplayName()](#Template+getDisplayName) ⇒ <code>string</code>*
        * *[.getVersion()](#Template+getVersion) ⇒ <code>String</code>*
        * *[.getDescription()](#Template+getDescription) ⇒ <code>String</code>*
        * *[.getHash()](#Template+getHash) ⇒ <code>string</code>*
        * *[.toArchive([language], [options])](#Template+toArchive) ⇒
<code>Promise.&lt;Buffer&gt;</code>*
        * *[.getParserManager()](#Template+getParserManager) ⇒
[<code>ParserManager</code>](#ParserManager)*
        * *[.getLogicManager()](#Template+getLogicManager) ⇒
<code>LogicManager</code>*
        * *[.getIntrospector()](#Template+getIntrospector) ⇒
<code>Introspector</code>*
        * *[.getFactory()](#Template+getFactory) ⇒ <code>Factory</code>*
        * *[.getSerializer()](#Template+getSerializer) ⇒ <code>Serializer</code>*
        * *[.getRequestTypes()](#Template+getRequestTypes) ⇒ <code>Array</code>*
        * *[.getResponseTypes()](#Template+getResponseTypes) ⇒ <code>Array</code>*
        * *[.getEmitTypes()](#Template+getEmitTypes) ⇒ <code>Array</code>*
        * *[.getStateTypes()](#Template+getStateTypes) ⇒ <code>Array</code>*
        * *[.hasLogic()](#Template+hasLogic) ⇒ <code>boolean</code>*
        * *[.grammarHasErgoExpression()](#Template+grammarHasErgoExpression) ⇒
<code>boolean</code>*
    * _static_
        * *[.fromDirectory(path, [options])](#Template.fromDirectory) ⇒
[<code>Promise.&lt;Template&gt;</code>](#Template)*
        * *[.fromArchive(buffer, [options])](#Template.fromArchive) ⇒
[<code>Promise.&lt;Template&gt;</code>](#Template)*
        * *[.fromUrl(url, [options])](#Template.fromUrl) ⇒ <code>Promise</code>*
        * *[.instanceOf(classDeclaration, fqt)](#Template.instanceOf) ⇒
```

<code>boolean</code>*

<a name="new_Template_new"></a>

### *new Template(packageJson, readme, samples, request, options)*
Create the Template.
Note: Only to be called by framework code. Applications should
retrieve instances from [fromArchive](#Template.fromArchive) or [fromDirectory](#Template.fromDirectory).

Param	Type	Description
packageJson	<code>object</code>	the JS object for package.json
readme	<code>String</code>	the readme in markdown for the template (optional)
samples	<code>object</code>	the sample text for the template in different locales
request	<code>object</code>	the JS object for the sample request
options	<code>Object</code>	e.g., { warnings: true }

<a name="Template+validate"></a>

### *template.validate()*
Verifies that the template is well formed.
Throws an exception with the details of any validation errors.

Kind: instance method of [<code>Template</code>](#Template)
<a name="Template+getTemplateModel"></a>

### *template.getTemplateModel() ⇒ <code>ClassDeclaration</code>*
Returns the template model for the template

Kind: instance method of [<code>Template</code>](#Template)
Returns: <code>ClassDeclaration</code> - the template model for the template
Throws:

- <code>Error</code> if no template model is found, or multiple template models are found

<a name="Template+getIdentifier"></a>

### *template.getIdentifier() ⇒ <code>String</code>*
Returns the identifier for this template

Kind: instance method of [<code>Template</code>](#Template)
Returns: <code>String</code> - the identifier of this template
<a name="Template+getMetadata"></a>

### *template.getMetadata() ⇒ [<code>Metadata</code>](#Metadata)*
Returns the metadata for this template

Kind: instance method of [<code>Template</code>](#Template)
Returns: [<code>Metadata</code>](#Metadata) - the metadata for this template
<a name="Template+getName"></a>

### *template.getName() ⇒ <code>String</code>*
Returns the name for this template

Kind: instance method of [<code>Template</code>](#Template)
Returns: <code>String</code> - the name of this template
<a name="Template+getDisplayName"></a>

### *template.getDisplayName() ⇒ <code>string</code>*
Returns the display name for this template.

Kind: instance method of [<code>Template</code>](#Template)
Returns: <code>string</code> - the display name of the template
<a name="Template+getVersion"></a>

### *template.getVersion() ⇒ <code>String</code>*
Returns the version for this template

Kind: instance method of [<code>Template</code>](#Template)
Returns: <code>String</code> - the version of this template. Use semver module
to parse.
<a name="Template+getDescription"></a>

### *template.getDescription() ⇒ <code>String</code>*
Returns the description for this template

Kind: instance method of [<code>Template</code>](#Template)
Returns: <code>String</code> - the description of this template
<a name="Template+getHash"></a>

### *template.getHash() ⇒ <code>string</code>*
Gets a content based SHA-256 hash for this template. Hash
is based on the metadata for the template plus the contents of
all the models and all the script files.

Kind: instance method of [<code>Template</code>](#Template)
Returns: <code>string</code> - the SHA-256 hash in hex format
<a name="Template+toArchive"></a>

### *template.toArchive([language], [options]) ⇒
<code>Promise.&lt;Buffer&gt;</code>*
Persists this template to a Cicero Template Archive (cta) file.

Kind: instance method of [<code>Template</code>](#Template)
Returns: <code>Promise.&lt;Buffer&gt;</code> - the zlib buffer

Param	Type	Description
[language]	<code>string</code>	target language for the archive (should be 'ergo')
[options]	<code>Object</code>	JSZip options

<a name="Template+getParserManager"></a>

### *template.getParserManager() ⇒ [<code>ParserManager</code>](#ParserManager)*
Provides access to the parser manager for this template.
The parser manager can convert template data to and from
natural language text.

Kind: instance method of [<code>Template</code>](#Template)
Returns: [<code>ParserManager</code>](#ParserManager) - the ParserManager for
this template
<a name="Template+getLogicManager"></a>

```
### *template.getLogicManager() ⇒ <code>LogicManager</code>*
Provides access to the template logic for this template.
The template logic encapsulate the code necessary to
execute the clause or contract.

**Kind**: instance method of [<code>Template</code>](#Template)
**Returns**: <code>LogicManager</code> - the LogicManager for this template
<a name="Template+getIntrospector"></a>

### *template.getIntrospector() ⇒ <code>Introspector</code>*
Provides access to the Introspector for this template. The Introspector
is used to reflect on the types defined within this template.

**Kind**: instance method of [<code>Template</code>](#Template)
**Returns**: <code>Introspector</code> - the Introspector for this template
<a name="Template+getFactory"></a>

### *template.getFactory() ⇒ <code>Factory</code>*
Provides access to the Factory for this template. The Factory
is used to create the types defined in this template.

**Kind**: instance method of [<code>Template</code>](#Template)
**Returns**: <code>Factory</code> - the Factory for this template
<a name="Template+getSerializer"></a>

### *template.getSerializer() ⇒ <code>Serializer</code>*
Provides access to the Serializer for this template. The Serializer
is used to serialize instances of the types defined within this template.

**Kind**: instance method of [<code>Template</code>](#Template)
**Returns**: <code>Serializer</code> - the Serializer for this template
<a name="Template+getRequestTypes"></a>

### *template.getRequestTypes() ⇒ <code>Array</code>*
Provides a list of the input types that are accepted by this Template. Types use
the fully-qualified form.

**Kind**: instance method of [<code>Template</code>](#Template)
**Returns**: <code>Array</code> - a list of the request types
<a name="Template+getResponseTypes"></a>

### *template.getResponseTypes() ⇒ <code>Array</code>*
Provides a list of the response types that are returned by this Template. Types use
the fully-qualified form.

**Kind**: instance method of [<code>Template</code>](#Template)
**Returns**: <code>Array</code> - a list of the response types
<a name="Template+getEmitTypes"></a>

### *template.getEmitTypes() ⇒ <code>Array</code>*
Provides a list of the emit types that are emitted by this Template. Types use the
fully-qualified form.

**Kind**: instance method of [<code>Template</code>](#Template)
**Returns**: <code>Array</code> - a list of the emit types
<a name="Template+getStateTypes"></a>

### *template.getStateTypes() ⇒ <code>Array</code>*
```

```
Provides a list of the state types that are expected by this Template. Types use
the fully-qualified form.

**Kind**: instance method of [<code>Template</code>](#Template)
**Returns**: <code>Array</code> - a list of the state types
<a name="Template+hasLogic"></a>

### *template.hasLogic() ⇒ <code>boolean</code>*
Returns true if the template has logic, i.e. has more than one script file.

**Kind**: instance method of [<code>Template</code>](#Template)
**Returns**: <code>boolean</code> - true if the template has logic
<a name="Template+grammarHasErgoExpression"></a>

### *template.grammarHasErgoExpression() ⇒ <code>boolean</code>*
Checks whether the template grammar has computer (Ergo) expressions

**Kind**: instance method of [<code>Template</code>](#Template)
**Returns**: <code>boolean</code> - True if the template grammar has Ergo
expressions (`{{% ... %}}`)
<a name="Template.fromDirectory"></a>

### *Template.fromDirectory(path, [options]) ⇒
[<code>Promise.&lt;Template&gt;</code>](#Template)*
Builds a Template from the contents of a directory.
The directory must include a package.json in the root (used to specify
the name, version and description of the template).

**Kind**: static method of [<code>Template</code>](#Template)
**Returns**: [<code>Promise.&lt;Template&gt;</code>](#Template) - a Promise to the
instantiated template

| Param | Type | Default | Description |
| --- | --- | --- | --- |
| path | <code>String</code> |  | to a local directory |
| [options] | <code>Object</code> | <code></code> | an optional set of options to
configure the instance. |

<a name="Template.fromArchive"></a>

### *Template.fromArchive(buffer, [options]) ⇒
[<code>Promise.&lt;Template&gt;</code>](#Template)*
Create a template from an archive.

**Kind**: static method of [<code>Template</code>](#Template)
**Returns**: [<code>Promise.&lt;Template&gt;</code>](#Template) - a Promise to the
template

| Param | Type | Default | Description |
| --- | --- | --- | --- |
| buffer | <code>Buffer</code> |  | the buffer to a Cicero Template Archive (cta)
file |
| [options] | <code>Object</code> | <code></code> | an optional set of options to
configure the instance. |

<a name="Template.fromUrl"></a>

### *Template.fromUrl(url, [options]) ⇒ <code>Promise</code>*
Create a template from an URL.
```

Kind: static method of [<code>Template</code>](#Template)
Returns: <code>Promise</code> - a Promise to the template

Param	Type	Default	Description
url	<code>String</code>		the URL to a Cicero Template Archive (cta) file
[options]	<code>Object</code>	<code></code>	an optional set of options to configure the instance.

<a name="Template.instanceOf"></a>

### *Template.instanceOf(classDeclaration, fqt) ⇒ <code>boolean</code>*
Check to see if a ClassDeclaration is an instance of the specified fully qualified type name.

Kind: static method of [<code>Template</code>](#Template)
Returns: <code>boolean</code> - True if classDeclaration an instance of the specified fully
qualified type name, false otherwise.
Internal:

Param	Type	Description
classDeclaration	<code>ClassDeclaration</code>	The class to test
fqt	<code>String</code>	The fully qualified type name.

<a name="TemplateInstance"></a>

## *TemplateInstance*
A TemplateInstance is an instance of a Clause or Contract template. It is executable business logic, linked to
a natural language (legally enforceable) template.
A TemplateInstance must be constructed with a template and then prior to execution the data for the clause must be set.
Set the data for the TemplateInstance by either calling the setData method or by calling the parse method and passing in natural language text that conforms to the template grammar.

Kind: global abstract class
Access: public

* *[TemplateInstance](#TemplateInstance)*
    * *[new TemplateInstance(template)](#new_TemplateInstance_new)*
    * _instance_
        * *[.setData(data)](#TemplateInstance+setData)*
        * *[.getData()](#TemplateInstance+getData) ⇒ <code>object</code>*
        * *[.getEngine()](#TemplateInstance+getEngine) ⇒ <code>object</code>*
        * *[.getDataAsConcertoObject()](#TemplateInstance+getDataAsConcertoObject) ⇒ <code>object</code>*
        * *[.parse(input, [currentTime], [fileName])](#TemplateInstance+parse)*
        * *[.draft([options], currentTime)](#TemplateInstance+draft) ⇒ <code>string</code>*
        * *[.getIdentifier()](#TemplateInstance+getIdentifier) ⇒ <code>String</code>*
        * *[.getTemplate()](#TemplateInstance+getTemplate) ⇒ [<code>Template</code>](#Template)*
        * *[.getLogicManager()](#TemplateInstance+getLogicManager) ⇒ <code>LogicManager</code>*

```
        * *[.toJSON()](#TemplateInstance+toJSON) ⇒ <code>object</code>*
    * _static_
        * *[.convertDateTimes(obj, utcOffset)](#TemplateInstance.convertDateTimes)
⇒ <code>\*</code>*

<a name="new_TemplateInstance_new"></a>

### *new TemplateInstance(template)*
Create the Clause and link it to a Template.


| Param | Type | Description |
| --- | --- | --- |
| template | [<code>Template</code>](#Template) | the template for the clause |

<a name="TemplateInstance+setData"></a>

### *templateInstance.setData(data)*
Set the data for the clause

**Kind**: instance method of [<code>TemplateInstance</code>](#TemplateInstance)

| Param | Type | Description |
| --- | --- | --- |
| data | <code>object</code> | the data for the clause, must be an instance of the
template model for the clause's template. This should be a plain JS object and will
be deserialized and validated into the Concerto object before assignment. |

<a name="TemplateInstance+getData"></a>

### *templateInstance.getData() ⇒ <code>object</code>*
Get the data for the clause. This is a plain JS object. To retrieve the Concerto
object call getConcertoData().

**Kind**: instance method of [<code>TemplateInstance</code>](#TemplateInstance)
**Returns**: <code>object</code> - - the data for the clause, or null if it has not
been set
<a name="TemplateInstance+getEngine"></a>

### *templateInstance.getEngine() ⇒ <code>object</code>*
Get the current Ergo engine

**Kind**: instance method of [<code>TemplateInstance</code>](#TemplateInstance)
**Returns**: <code>object</code> - - the data for the clause, or null if it has not
been set
<a name="TemplateInstance+getDataAsConcertoObject"></a>

### *templateInstance.getDataAsConcertoObject() ⇒ <code>object</code>*
Get the data for the clause. This is a Concerto object. To retrieve the
plain JS object suitable for serialization call toJSON() and retrieve the `data`
property.

**Kind**: instance method of [<code>TemplateInstance</code>](#TemplateInstance)
**Returns**: <code>object</code> - - the data for the clause, or null if it has not
been set
<a name="TemplateInstance+parse"></a>

### *templateInstance.parse(input, [currentTime], [fileName])*
Set the data for the clause by parsing natural language text.
```

Kind: instance method of [<code>TemplateInstance</code>](#TemplateInstance)

Param	Type	Description
input	<code>string</code>	the text for the clause
[currentTime]	<code>string</code>	the definition of 'now' (optional)
[fileName]	<code>string</code>	the fileName for the text (optional)

<a name="TemplateInstance+draft"></a>

### *templateInstance.draft([options], currentTime) ⇒ <code>string</code>*
Generates the natural language text for a contract or clause clause; combining the text from the template
and the instance data.

Kind: instance method of [<code>TemplateInstance</code>](#TemplateInstance)
Returns: <code>string</code> - the natural language text for the contract or clause; created by combining the structure of
the template with the JSON data for the clause.

Param	Type	Description
[options]	<code>*</code>	text generation options. options.wrapVariables encloses variables and editable sections in '<variable ...' and '/>'
currentTime	<code>string</code>	the definition of 'now' (optional)

<a name="TemplateInstance+getIdentifier"></a>

### *templateInstance.getIdentifier() ⇒ <code>String</code>*
Returns the identifier for this clause. The identifier is the identifier of
the template plus '-' plus a hash of the data for the clause (if set).

Kind: instance method of [<code>TemplateInstance</code>](#TemplateInstance)
Returns: <code>String</code> - the identifier of this clause
<a name="TemplateInstance+getTemplate"></a>

### *templateInstance.getTemplate() ⇒ [<code>Template</code>](#Template)*
Returns the template for this clause

Kind: instance method of [<code>TemplateInstance</code>](#TemplateInstance)
Returns: [<code>Template</code>](#Template) - the template for this clause
<a name="TemplateInstance+getLogicManager"></a>

### *templateInstance.getLogicManager() ⇒ <code>LogicManager</code>*
Returns the template logic for this clause

Kind: instance method of [<code>TemplateInstance</code>](#TemplateInstance)
Returns: <code>LogicManager</code> - the template for this clause
<a name="TemplateInstance+toJSON"></a>

### *templateInstance.toJSON() ⇒ <code>object</code>*
Returns a JSON representation of the clause

Kind: instance method of [<code>TemplateInstance</code>](#TemplateInstance)
Returns: <code>object</code> - the JS object for serialization
<a name="TemplateInstance.convertDateTimes"></a>

### *TemplateInstance.convertDateTimes(obj, utcOffset) ⇒ <code>*</code>*

```
Recursive function that converts all instances of ParsedDateTime
to a Moment.

**Kind**: static method of [<code>TemplateInstance</code>](#TemplateInstance)
**Returns**: <code>\*</code> - the converted object

| Param | Type | Description |
| --- | --- | --- |
| obj | <code>\*</code> | the input object |
| utcOffset | <code>number</code> | the default utcOffset |

<a name="CompositeArchiveLoader"></a>

## CompositeArchiveLoader
Manages a set of archive loaders, delegating to the first archive
loader that accepts a URL.

**Kind**: global class

* [CompositeArchiveLoader](#CompositeArchiveLoader)
    * [new CompositeArchiveLoader()](#new_CompositeArchiveLoader_new)
    * [.addArchiveLoader(archiveLoader)](#CompositeArchiveLoader+addArchiveLoader)
    * [.clearArchiveLoaders()](#CompositeArchiveLoader+clearArchiveLoaders)
    * *[.accepts(url)](#CompositeArchiveLoader+accepts) ⇒ <code>boolean</code>*
    * [.load(url, options)](#CompositeArchiveLoader+load) ⇒ <code>Promise</code>

<a name="new_CompositeArchiveLoader_new"></a>

### new CompositeArchiveLoader()
Create the CompositeArchiveLoader. Used to delegate to a set of ArchiveLoaders.

<a name="CompositeArchiveLoader+addArchiveLoader"></a>

### compositeArchiveLoader.addArchiveLoader(archiveLoader)
Adds a ArchiveLoader implemenetation to the ArchiveLoader

**Kind**: instance method of [<code>CompositeArchiveLoader</code>]
(#CompositeArchiveLoader)

| Param | Type | Description |
| --- | --- | --- |
| archiveLoader | <code>ArchiveLoader</code> | The archive to add to the
CompositeArchiveLoader |

<a name="CompositeArchiveLoader+clearArchiveLoaders"></a>

### compositeArchiveLoader.clearArchiveLoaders()
Remove all registered ArchiveLoaders

**Kind**: instance method of [<code>CompositeArchiveLoader</code>]
(#CompositeArchiveLoader)
<a name="CompositeArchiveLoader+accepts"></a>

### *compositeArchiveLoader.accepts(url) ⇒ <code>boolean</code>*
Returns true if this ArchiveLoader can process the URL

**Kind**: instance abstract method of [<code>CompositeArchiveLoader</code>]
(#CompositeArchiveLoader)
**Returns**: <code>boolean</code> - true if this ArchiveLoader accepts the URL
```

Param	Type	Description
url	<code>string</code>	the URL

<a name="CompositeArchiveLoader+load"></a>

### compositeArchiveLoader.load(url, options) ⇒ <code>Promise</code>
Load a Archive from a URL and return it

Kind: instance method of [<code>CompositeArchiveLoader</code>]
(#CompositeArchiveLoader)
Returns: <code>Promise</code> - a promise to the Archive

Param	Type	Description
url	<code>string</code>	the url to get
options	<code>object</code>	additional options

<a name="locationOfError"></a>

## locationOfError(error) ⇒ <code>object</code>
Extract the file location from the parse error

Kind: global function
Returns: <code>object</code> - - the file location information

Param	Type	Description
error	<code>object</code>	the error object

---
id: version-0.20-cicero-cli
title: Cicero CLI
original_id: cicero-cli

Install the `@accordproject/cicero-cli` npm package to access the Cicero command line interface (CLI). After installation you can use the `cicero` command and its sub-commands as described below.

To install the Cicero CLI:
```
npm install -g @accordproject/cicero-cli@0.20
```

## Usage

```md
Commands:
  cicero parse       parse a contract text
  cicero draft       create contract text from data
  cicero normalize   normalize markdown (parse & redraft)
  cicero trigger     send a request to the contract
  cicero invoke      invoke a clause of the contract
  cicero initialize  initialize a clause
  cicero archive     create a template archive
```

~~~~
  cicero compile     generate code for a target platform
  cicero get         save local copies of external dependencies

Options:
  --version      Show version number                                   [boolean]
  --verbose, -v                                                 [default: false]
  --help         Show help                                             [boolean]

```

## cicero parse

`cicero parse` loads a template from a directory on disk and then parses input
clause (or contract) text using the template. If successful, the template model is
printed to console. If there are syntax errors, the line and column and error
information are printed.

```md
cicero parse

parse a contract text

Options:
  --version      Show version number                                   [boolean]
  --verbose, -v                                                 [default: false]
  --help         Show help                                             [boolean]
  --template     path to the template                                   [string]
  --sample       path to the contract text                              [string]
  --output       path to the output file                                [string]
  --currentTime  set current time                       [string] [default: null]
  --warnings     print warnings                       [boolean] [default: false]
```

## cicero draft

`cicero draft` creates contract text from data.

```md
create contract text from data

Options:
  --version        Show version number                                 [boolean]
  --verbose, -v                                                 [default: false]
  --help           Show help                                           [boolean]
  --template       path to the template                                 [string]
  --data           path to the contract data                            [string]
  --output         path to the output file                              [string]
  --currentTime    set current time                     [string] [default: null]
  --wrapVariables  wrap variables as XML tags         [boolean] [default: false]
  --warnings       print warnings                     [boolean] [default: false]
```

## cicero normalize

`cicero normalize` normalizes markdown text by parsing and redrafting the text.

```md
normalize markdown (parse & redraft)
~~~~

~~~~
Options:
  --version        Show version number                                 [boolean]
  --verbose, -v                                                 [default: false]
  --help           Show help                                           [boolean]
  --template       path to the template                                 [string]
  --sample         path to the contract text                            [string]
  --overwrite      overwrite the contract text        [boolean] [default: false]
  --output         path to the output file                              [string]
  --currentTime    set current time                     [string] [default: null]
  --warnings       print warnings                     [boolean] [default: false]
  --wrapVariables  wrap variables as XML tags         [boolean] [default: false]
```

## cicero trigger

`cicero trigger` sends a request to the contract.

```md
send a request to the contract

Options:
  --version      Show version number                                   [boolean]
  --verbose, -v                                                 [default: false]
  --help         Show help                                             [boolean]
  --template     path to the template                                   [string]
  --sample       path to the contract text                              [string]
  --request      path to the JSON request                                [array]
  --state        path to the JSON state                                 [string]
  --currentTime  set current time                       [string] [default: null]
  --warnings     print warnings                       [boolean] [default: false]

```

## cicero invoke

`cicero invoke` invokes a specific clause (`--clauseName`) of the contract.

```md
invoke a clause of the contract

Options:
  --version      Show version number                                   [boolean]
  --verbose, -v                                                 [default: false]
  --help         Show help                                             [boolean]
  --template     path to the template                                   [string]
  --sample       path to the contract text                              [string]
  --clauseName   the name of the clause to invoke                       [string]
  --params       path to the parameters                                 [string]
  --state        path to the JSON state                                 [string]
  --currentTime  set current time                       [string] [default: null]
  --warnings     print warnings                       [boolean] [default: false]

```

## cicero initialize

`cicero initialize` initializes a clause.

```md
~~~~

~~~~
cicero initialize

initialize a clause

Options:
  --version      Show version number                                   [boolean]
  --verbose, -v                                                 [default: false]
  --help         Show help                                             [boolean]
  --template     path to the template                                   [string]
  --sample       path to the contract text                              [string]
  --currentTime  initialize with this current time      [string] [default: null]
  --warnings     print warnings                       [boolean] [default: false]

```
## cicero archive

`cicero archive` creates a Cicero Template Archive (`.cta`) file from a template
stored in a local directory.

```md
cicero archive

create a template archive

Options:
  --version      Show version number                                   [boolean]
  --verbose, -v                                                 [default: false]
  --help         Show help                                             [boolean]
  --template     path to the template                                   [string]
  --target       the target language of the archive   [string] [default: "ergo"]
  --output       file name for new archive              [string] [default: null]
  --warnings     print warnings                       [boolean] [default: false]

```

## cicero compile

`cicero compile` generates code for a target platform. It loads a template from a
directory on disk and then attempts to generate versions of the template model in
the specified format. The available formats include: `Go`, `PlantUML`,
`Typescript`, `Java`, and `JSONSchema`.

```md
cicero compile

generate code for a target platform

Options:
  --version      Show version number                                   [boolean]
  --verbose, -v                                                 [default: false]
  --help         Show help                                             [boolean]
  --template     path to the template                                   [string]
  --target       target of the code generation  [string] [default: "JSONSchema"]
  --output       path to the output directory    [string] [default: "./output/"]
  --warnings     print warnings                       [boolean] [default: false]

```
## cicero get
~~~~

~~~
`cicero get` saves local copies of external dependencies.

```md
cicero get

save local copies of external dependencies

Options:
  --version      Show version number                                [boolean]
  --verbose, -v                                              [default: false]
  --help         Show help                                          [boolean]
  --template     path to the template                                [string]
  --output       output directory path                               [string]
```

--------------------------------------------------------------------------------
---
id: version-0.20-concerto-api
title: Concerto API
original_id: concerto-api
---

<a name="module_concerto-core"></a>

## concerto-core
Concerto module. Concerto is a framework for defining domain
specific models.


* [concerto-core](#module_concerto-core)
    * _static_
        * [.ModelLoader](#module_concerto-core.ModelLoader)
            * [.loadModelManager(ctoSystemFile, ctoFiles)](#module_concerto-
core.ModelLoader.loadModelManager) ⇒ <code>object</code>
        * [.DecoratorFactory](#module_concerto-core.DecoratorFactory)
            * *[.newDecorator(parent, ast)](#module_concerto-
core.DecoratorFactory+newDecorator) ⇒ <code>Decorator</code>*
    * _inner_
        * [~BaseException](#module_concerto-core.BaseException) ⇐
<code>Error</code>
            * [new BaseException(message, component)](#new_module_concerto-
core.BaseException_new)
        * [~BaseFileException](#module_concerto-core.BaseFileException) ⇐
<code>BaseException</code>
            * [new BaseFileException(message, fileLocation, fullMessage, fileName,
component)](#new_module_concerto-core.BaseFileException_new)
            * [.getFileLocation()](#module_concerto-
core.BaseFileException+getFileLocation) ⇒ <code>string</code>
            * [.getShortMessage()](#module_concerto-
core.BaseFileException+getShortMessage) ⇒ <code>string</code>
            * [.getFileName()](#module_concerto-core.BaseFileException+getFileName)
⇒ <code>string</code>
        * [~Factory](#module_concerto-core.Factory)
            * [new Factory(modelManager)](#new_module_concerto-core.Factory_new)
            * _instance_
                * [.newResource(ns, type, id, [options])](#module_concerto-
core.Factory+newResource) ⇒ <code>Resource</code>
                * [.newConcept(ns, type, [options])](#module_concerto-
core.Factory+newConcept) ⇒ <code>Resource</code>
~~~

```
                * [.newRelationship(ns, type, id)](#module_concerto-
core.Factory+newRelationship) ⇒ <code>Relationship</code>
                * [.newTransaction(ns, type, [id], [options])](#module_concerto-
core.Factory+newTransaction) ⇒ <code>Resource</code>
                * [.newEvent(ns, type, [id], [options])](#module_concerto-
core.Factory+newEvent) ⇒ <code>Resource</code>
            * _static_
                * [.Symbol.hasInstance(object)](#module_concerto-
core.Factory.Symbol.hasInstance) ⇒ <code>boolean</code>
        * [~ModelManager](#module_concerto-core.ModelManager)
            * [new ModelManager()](#new_module_concerto-core.ModelManager_new)
            * _instance_
                * [.validateModelFile(modelFile, fileName)](#module_concerto-
core.ModelManager+validateModelFile)
                * [.addModelFile(modelFile, fileName, [disableValidation],
[systemModelTable])](#module_concerto-core.ModelManager+addModelFile) ⇒
<code>Object</code>
                * [.updateModelFile(modelFile, fileName, [disableValidation])]
(#module_concerto-core.ModelManager+updateModelFile) ⇒ <code>Object</code>
                * [.deleteModelFile(namespace)](#module_concerto-
core.ModelManager+deleteModelFile)
                * [.addModelFiles(modelFiles, [fileNames], [disableValidation],
[systemModelTable])](#module_concerto-core.ModelManager+addModelFiles) ⇒
<code>Array.&lt;Object&gt;</code>
                * [.validateModelFiles()](#module_concerto-
core.ModelManager+validateModelFiles)
                * [.updateExternalModels([options], [modelFileDownloader])]
(#module_concerto-core.ModelManager+updateExternalModels) ⇒ <code>Promise</code>
                * [.writeModelsToFileSystem(path, [options])](#module_concerto-
core.ModelManager+writeModelsToFileSystem)
                * [.getModels([options])](#module_concerto-
core.ModelManager+getModels) ⇒ <code>Array.&lt;{name:string,
content:string}&gt;</code>
                * [.clearModelFiles()](#module_concerto-
core.ModelManager+clearModelFiles)
                * [.getNamespaces()](#module_concerto-
core.ModelManager+getNamespaces) ⇒ <code>Array.&lt;string&gt;</code>
                * [.getSystemTypes()](#module_concerto-
core.ModelManager+getSystemTypes) ⇒ <code>Array.&lt;ClassDeclaration&gt;</code>
                * [.getAssetDeclarations(includeSystemType)](#module_concerto-
core.ModelManager+getAssetDeclarations) ⇒
<code>Array.&lt;AssetDeclaration&gt;</code>
                * [.getTransactionDeclarations(includeSystemType)]
(#module_concerto-core.ModelManager+getTransactionDeclarations) ⇒
<code>Array.&lt;TransactionDeclaration&gt;</code>
                * [.getEventDeclarations(includeSystemType)](#module_concerto-
core.ModelManager+getEventDeclarations) ⇒
<code>Array.&lt;EventDeclaration&gt;</code>
                * [.getParticipantDeclarations(includeSystemType)]
(#module_concerto-core.ModelManager+getParticipantDeclarations) ⇒
<code>Array.&lt;ParticipantDeclaration&gt;</code>
                * [.getEnumDeclarations(includeSystemType)](#module_concerto-
core.ModelManager+getEnumDeclarations) ⇒ <code>Array.&lt;EnumDeclaration&gt;</code>
                * [.getConceptDeclarations(includeSystemType)](#module_concerto-
core.ModelManager+getConceptDeclarations) ⇒
<code>Array.&lt;ConceptDeclaration&gt;</code>
                * [.getFactory()](#module_concerto-core.ModelManager+getFactory) ⇒
<code>Factory</code>
                * [.getSerializer()](#module_concerto-
core.ModelManager+getSerializer) ⇒ <code>Serializer</code>
                * [.getDecoratorFactories()](#module_concerto-
core.ModelManager+getDecoratorFactories) ⇒
<code>Array.&lt;DecoratorFactory&gt;</code>
                * [.addDecoratorFactory(factory)](#module_concerto-
core.ModelManager+addDecoratorFactory)
            * _static_
                * [.Symbol.hasInstance(object)](#module_concerto-
core.ModelManager.Symbol.hasInstance) ⇒ <code>boolean</code>
        * [~SecurityException](#module_concerto-core.SecurityException) ⇐
<code>BaseException</code>
            * [new SecurityException(message)](#new_module_concerto-
core.SecurityException_new)
        * [~Serializer](#module_concerto-core.Serializer)
            * [new Serializer(factory, modelManager)](#new_module_concerto-
core.Serializer_new)
            * _instance_
                * [.setDefaultOptions(newDefaultOptions)](#module_concerto-
core.Serializer+setDefaultOptions)
                * [.toJSON(resource, [options])](#module_concerto-
core.Serializer+toJSON) ⇒ <code>Object</code>
                * [.fromJSON(jsonObject, options)](#module_concerto-
core.Serializer+fromJSON) ⇒ <code>Resource</code>
            * _static_
                * [.Symbol.hasInstance(object)](#module_concerto-
core.Serializer.Symbol.hasInstance) ⇒ <code>boolean</code>
        * [~AssetDeclaration](#module_concerto-core.AssetDeclaration) ⇐
<code>ClassDeclaration</code>
            * [new AssetDeclaration(modelFile, ast)](#new_module_concerto-
core.AssetDeclaration_new)
            * _instance_
                * [.isRelationshipTarget()](#module_concerto-
core.AssetDeclaration+isRelationshipTarget) ⇒ <code>boolean</code>
                * [.getSystemType()](#module_concerto-
core.AssetDeclaration+getSystemType) ⇒ <code>string</code>
            * _static_
                * [.Symbol.hasInstance(object)](#module_concerto-
core.AssetDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>
        * *[~ClassDeclaration](#module_concerto-core.ClassDeclaration)*
            * *[new ClassDeclaration(modelFile, ast)](#new_module_concerto-
core.ClassDeclaration_new)*
            * _instance_
                * *[._resolveSuperType()](#module_concerto-
core.ClassDeclaration+_resolveSuperType) ⇒ <code>ClassDeclaration</code>*
                * *[.getSystemType()](#module_concerto-
core.ClassDeclaration+getSystemType) ⇒ <code>string</code>*
                * *[.isAbstract()](#module_concerto-
core.ClassDeclaration+isAbstract) ⇒ <code>boolean</code>*
                * *[.isEnum()](#module_concerto-core.ClassDeclaration+isEnum) ⇒
<code>boolean</code>*
                * *[.isConcept()](#module_concerto-core.ClassDeclaration+isConcept)
⇒ <code>boolean</code>*
                * *[.isEvent()](#module_concerto-core.ClassDeclaration+isEvent) ⇒
<code>boolean</code>*
                * *[.isRelationshipTarget()](#module_concerto-
core.ClassDeclaration+isRelationshipTarget) ⇒ <code>boolean</code>*
                * *[.isSystemRelationshipTarget()](#module_concerto-
core.ClassDeclaration+isSystemRelationshipTarget) ⇒ <code>boolean</code>*
                * *[.isSystemType()](#module_concerto-
core.ClassDeclaration+isSystemType) ⇒ <code>boolean</code>*
                * *[.isSystemCoreType()](#module_concerto-
core.ClassDeclaration+isSystemCoreType) ⇒ <code>boolean</code>*
                * *[.getName()](#module_concerto-core.ClassDeclaration+getName) ⇒
<code>string</code>*
                * *[.getNamespace()](#module_concerto-
core.ClassDeclaration+getNamespace) ⇒ <code>String</code>*
                * *[.getFullyQualifiedName()](#module_concerto-
core.ClassDeclaration+getFullyQualifiedName) ⇒ <code>string</code>*
                * *[.getIdentifierFieldName()](#module_concerto-
core.ClassDeclaration+getIdentifierFieldName) ⇒ <code>string</code>*
                * *[.getOwnProperty(name)](#module_concerto-
core.ClassDeclaration+getOwnProperty) ⇒ <code>Property</code>*
                * *[.getOwnProperties()](#module_concerto-
core.ClassDeclaration+getOwnProperties) ⇒ <code>Array.&lt;Property&gt;</code>*
                * *[.getSuperType()](#module_concerto-
core.ClassDeclaration+getSuperType) ⇒ <code>string</code>*
                * *[.getSuperTypeDeclaration()](#module_concerto-
core.ClassDeclaration+getSuperTypeDeclaration) ⇒ <code>ClassDeclaration</code>*
                * *[.getAssignableClassDeclarations()](#module_concerto-
core.ClassDeclaration+getAssignableClassDeclarations) ⇒
<code>Array.&lt;ClassDeclaration&gt;</code>*
                * *[.getAllSuperTypeDeclarations()](#module_concerto-
core.ClassDeclaration+getAllSuperTypeDeclarations) ⇒
<code>Array.&lt;ClassDeclaration&gt;</code>*
                * *[.getProperty(name)](#module_concerto-
core.ClassDeclaration+getProperty) ⇒ <code>Property</code>*
                * *[.getProperties()](#module_concerto-
core.ClassDeclaration+getProperties) ⇒ <code>Array.&lt;Property&gt;</code>*
                * *[.getNestedProperty(propertyPath)](#module_concerto-
core.ClassDeclaration+getNestedProperty) ⇒ <code>Property</code>*
                * *[.toString()](#module_concerto-core.ClassDeclaration+toString) ⇒
<code>String</code>*
            * _static_
                * *[.Symbol.hasInstance(object)](#module_concerto-
core.ClassDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>*
        * [~ConceptDeclaration](#module_concerto-core.ConceptDeclaration) ⇐
<code>ClassDeclaration</code>
            * [new ConceptDeclaration(modelFile, ast)](#new_module_concerto-
core.ConceptDeclaration_new)
            * _instance_
                * [.isConcept()](#module_concerto-
core.ConceptDeclaration+isConcept) ⇒ <code>boolean</code>
            * _static_
                * [.Symbol.hasInstance(object)](#module_concerto-
core.ConceptDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>
        * [~Decorator](#module_concerto-core.Decorator)
            * [new Decorator(parent, ast)](#new_module_concerto-core.Decorator_new)
            * [.getParent()](#module_concerto-core.Decorator+getParent) ⇒
<code>ClassDeclaration</code> \| <code>Property</code>
            * [.getName()](#module_concerto-core.Decorator+getName) ⇒
<code>string</code>
            * [.getArguments()](#module_concerto-core.Decorator+getArguments) ⇒
<code>Array.&lt;object&gt;</code>
        * [~EnumDeclaration](#module_concerto-core.EnumDeclaration) ⇐
<code>ClassDeclaration</code>
            * [new EnumDeclaration(modelFile, ast)](#new_module_concerto-
core.EnumDeclaration_new)
            * _instance_
```

```
                * [.isEnum()](#module_concerto-core.EnumDeclaration+isEnum) ⇒
<code>boolean</code>
                * [.toString()](#module_concerto-core.EnumDeclaration+toString) ⇒
<code>String</code>
            * _static_
                * [.Symbol.hasInstance(object)](#module_concerto-
core.EnumDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>
        * [~EnumValueDeclaration](#module_concerto-core.EnumValueDeclaration) ⇐
<code>Property</code>
            * [new EnumValueDeclaration(parent, ast)](#new_module_concerto-
core.EnumValueDeclaration_new)
            * [.Symbol.hasInstance(object)](#module_concerto-
core.EnumValueDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>
        * [~EventDeclaration](#module_concerto-core.EventDeclaration) ⇐
<code>ClassDeclaration</code>
            * [new EventDeclaration(modelFile, ast)](#new_module_concerto-
core.EventDeclaration_new)
            * _instance_
                * [.getSystemType()](#module_concerto-
core.EventDeclaration+getSystemType) ⇒ <code>string</code>
                * [.isEvent()](#module_concerto-core.EventDeclaration+isEvent) ⇒
<code>boolean</code>
            * _static_
                * [.Symbol.hasInstance(object)](#module_concerto-
core.EventDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>
        * [~Introspector](#module_concerto-core.Introspector)
            * [new Introspector(modelManager)](#new_module_concerto-
core.Introspector_new)
            * [.getClassDeclarations()](#module_concerto-
core.Introspector+getClassDeclarations) ⇒
<code>Array.&lt;ClassDeclaration&gt;</code>
            * [.getClassDeclaration(fullyQualifiedTypeName)](#module_concerto-
core.Introspector+getClassDeclaration) ⇒ <code>ClassDeclaration</code>
        * [~ModelFile](#module_concerto-core.ModelFile)
            * [new ModelFile(modelManager, definitions, [fileName],
[isSystemModelFile])](#new_module_concerto-core.ModelFile_new)
            * _instance_
                * [.isExternal()](#module_concerto-core.ModelFile+isExternal) ⇒
<code>boolean</code>
                * [.getModelManager()](#module_concerto-
core.ModelFile+getModelManager) ⇒ <code>ModelManager</code>
                * [.getImports()](#module_concerto-core.ModelFile+getImports) ⇒
<code>Array.&lt;string&gt;</code>
                * [.isDefined(type)](#module_concerto-core.ModelFile+isDefined) ⇒
<code>boolean</code>
                * [.getLocalType(type)](#module_concerto-
core.ModelFile+getLocalType) ⇒ <code>ClassDeclaration</code>
                * [.getAssetDeclaration(name)](#module_concerto-
core.ModelFile+getAssetDeclaration) ⇒ <code>AssetDeclaration</code>
                * [.getTransactionDeclaration(name)](#module_concerto-
core.ModelFile+getTransactionDeclaration) ⇒ <code>TransactionDeclaration</code>
                * [.getEventDeclaration(name)](#module_concerto-
core.ModelFile+getEventDeclaration) ⇒ <code>EventDeclaration</code>
                * [.getParticipantDeclaration(name)](#module_concerto-
core.ModelFile+getParticipantDeclaration) ⇒ <code>ParticipantDeclaration</code>
                * [.getNamespace()](#module_concerto-core.ModelFile+getNamespace) ⇒
<code>string</code>
                * [.getName()](#module_concerto-core.ModelFile+getName) ⇒
<code>string</code>
```

```
                * [.getAssetDeclarations(includeSystemType)](#module_concerto-
core.ModelFile+getAssetDeclarations) ⇒ <code>Array.&lt;AssetDeclaration&gt;</code>
                * [.getTransactionDeclarations(includeSystemType)]
(#module_concerto-core.ModelFile+getTransactionDeclarations) ⇒
<code>Array.&lt;TransactionDeclaration&gt;</code>
                * [.getEventDeclarations(includeSystemType)](#module_concerto-
core.ModelFile+getEventDeclarations) ⇒ <code>Array.&lt;EventDeclaration&gt;</code>
                * [.getParticipantDeclarations(includeSystemType)]
(#module_concerto-core.ModelFile+getParticipantDeclarations) ⇒
<code>Array.&lt;ParticipantDeclaration&gt;</code>
                * [.getConceptDeclarations(includeSystemType)](#module_concerto-
core.ModelFile+getConceptDeclarations) ⇒
<code>Array.&lt;ConceptDeclaration&gt;</code>
                * [.getEnumDeclarations(includeSystemType)](#module_concerto-
core.ModelFile+getEnumDeclarations) ⇒ <code>Array.&lt;EnumDeclaration&gt;</code>
                * [.getDeclarations(type, includeSystemType)](#module_concerto-
core.ModelFile+getDeclarations) ⇒ <code>Array.&lt;ClassDeclaration&gt;</code>
                * [.getAllDeclarations()](#module_concerto-
core.ModelFile+getAllDeclarations) ⇒ <code>Array.&lt;ClassDeclaration&gt;</code>
                * [.getDefinitions()](#module_concerto-
core.ModelFile+getDefinitions) ⇒ <code>string</code>
                * [.isSystemModelFile()](#module_concerto-
core.ModelFile+isSystemModelFile) ⇒ <code>boolean</code>
            * _static_
                * [.Symbol.hasInstance(object)](#module_concerto-
core.ModelFile.Symbol.hasInstance) ⇒ <code>boolean</code>
        * [~ParticipantDeclaration](#module_concerto-core.ParticipantDeclaration) ⇐
<code>ClassDeclaration</code>
            * [new ParticipantDeclaration(modelFile, ast)](#new_module_concerto-
core.ParticipantDeclaration_new)
            * _instance_
                * [.isRelationshipTarget()](#module_concerto-
core.ParticipantDeclaration+isRelationshipTarget) ⇒ <code>boolean</code>
                * [.getSystemType()](#module_concerto-
core.ParticipantDeclaration+getSystemType) ⇒ <code>string</code>
            * _static_
                * [.Symbol.hasInstance(object)](#module_concerto-
core.ParticipantDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>
        * [~Property](#module_concerto-core.Property)
            * [new Property(parent, ast)](#new_module_concerto-core.Property_new)
            * _instance_
                * [.getParent()](#module_concerto-core.Property+getParent) ⇒
<code>ClassDeclaration</code>
                * [.getName()](#module_concerto-core.Property+getName) ⇒
<code>string</code>
                * [.getType()](#module_concerto-core.Property+getType) ⇒
<code>string</code>
                * [.isOptional()](#module_concerto-core.Property+isOptional) ⇒
<code>boolean</code>
                * [.getFullyQualifiedTypeName()](#module_concerto-
core.Property+getFullyQualifiedTypeName) ⇒ <code>string</code>
                * [.getFullyQualifiedName()](#module_concerto-
core.Property+getFullyQualifiedName) ⇒ <code>string</code>
                * [.getNamespace()](#module_concerto-core.Property+getNamespace) ⇒
<code>string</code>
                * [.isArray()](#module_concerto-core.Property+isArray) ⇒
<code>boolean</code>
                * [.isTypeEnum()](#module_concerto-core.Property+isTypeEnum) ⇒
<code>boolean</code>
```

                * [.isPrimitive()](#module_concerto-core.Property+isPrimitive) ⇒ <code>boolean</code>
            * _static_
                * [.Symbol.hasInstance(object)](#module_concerto-core.Property.Symbol.hasInstance) ⇒ <code>boolean</code>
        * [~RelationshipDeclaration](#module_concerto-core.RelationshipDeclaration) ⇐ <code>Property</code>
            * [new RelationshipDeclaration(parent, ast)](#new_module_concerto-core.RelationshipDeclaration_new)
            * _instance_
                * [.toString()](#module_concerto-core.RelationshipDeclaration+toString) ⇒ <code>String</code>
            * _static_
                * [.Symbol.hasInstance(object)](#module_concerto-core.RelationshipDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>
        * [~TransactionDeclaration](#module_concerto-core.TransactionDeclaration) ⇐ <code>ClassDeclaration</code>
            * [new TransactionDeclaration(modelFile, ast)](#new_module_concerto-core.TransactionDeclaration_new)
            * _instance_
                * [.getSystemType()](#module_concerto-core.TransactionDeclaration+getSystemType) ⇒ <code>string</code>
            * _static_
                * [.Symbol.hasInstance(object)](#module_concerto-core.TransactionDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>

<a name="module_concerto-core.ModelLoader"></a>

### concerto-core.ModelLoader
Create a ModelManager from model files, with an optional system model.

If a ctoFile is not provided, the Accord Project system model is used.

Kind: static class of [<code>concerto-core</code>](#module_concerto-core)
<a name="module_concerto-core.ModelLoader.loadModelManager"></a>

#### ModelLoader.loadModelManager(ctoSystemFile, ctoFiles) ⇒ <code>object</code>
Load system and models in a new model manager

Kind: static method of [<code>ModelLoader</code>](#module_concerto-core.ModelLoader)
Returns: <code>object</code> - the model manager

Param	Type	Description
ctoSystemFile	<code>string</code>	the system model file
ctoFiles	<code>Array.&lt;string&gt;</code>	the CTO files (can be local file paths or URLs)

<a name="module_concerto-core.DecoratorFactory"></a>

### concerto-core.DecoratorFactory
An interface for a class that processes a decorator and returns a specific implementation class for that decorator.

Kind: static class of [<code>concerto-core</code>](#module_concerto-core)
<a name="module_concerto-core.DecoratorFactory+newDecorator"></a>

#### *decoratorFactory.newDecorator(parent, ast) ⇒ <code>Decorator</code>*

Process the decorator, and return a specific implementation class for that
decorator, or return null if this decorator is not handled by this processor.

Kind: instance abstract method of [<code>DecoratorFactory</code>]
(#module_concerto-core.DecoratorFactory)
Returns: <code>Decorator</code> - The decorator.

Param	Type	Description
parent	<code>ClassDeclaration</code> | <code>Property</code>	the owner of this property
ast	<code>Object</code>	The AST created by the parser

<a name="module_concerto-core.BaseException"></a>

### concerto-core~BaseException ⇐ <code>Error</code>
A base class for all Concerto exceptions

Kind: inner class of [<code>concerto-core</code>](#module_concerto-core)
Extends: <code>Error</code>
<a name="new_module_concerto-core.BaseException_new"></a>

#### new BaseException(message, component)
Create the BaseException.

Param	Type	Description
message	<code>string</code>	The exception message.
component	<code>string</code>	The optional component which throws this error.

<a name="module_concerto-core.BaseFileException"></a>

### concerto-core~BaseFileException ⇐ <code>BaseException</code>
Exception throws when a Concerto file is semantically invalid

Kind: inner class of [<code>concerto-core</code>](#module_concerto-core)
Extends: <code>BaseException</code>
See: BaseException

* [~BaseFileException](#module_concerto-core.BaseFileException) ⇐
<code>BaseException</code>
    * [new BaseFileException(message, fileLocation, fullMessage, fileName,
component)](#new_module_concerto-core.BaseFileException_new)
    * [.getFileLocation()](#module_concerto-core.BaseFileException+getFileLocation)
⇒ <code>string</code>
    * [.getShortMessage()](#module_concerto-core.BaseFileException+getShortMessage)
⇒ <code>string</code>
    * [.getFileName()](#module_concerto-core.BaseFileException+getFileName) ⇒
<code>string</code>

<a name="new_module_concerto-core.BaseFileException_new"></a>

#### new BaseFileException(message, fileLocation, fullMessage, fileName, component)
Create an BaseFileException

| Param | Type | Description |

```
| --- | --- | --- |
| message | <code>string</code> | the message for the exception |
| fileLocation | <code>string</code> | the optional file location associated with
the exception |
| fullMessage | <code>string</code> | the optional full message text |
| fileName | <code>string</code> | the optional file name |
| component | <code>string</code> | the optional component which throws this error
|

<a name="module_concerto-core.BaseFileException+getFileLocation"></a>

#### baseFileException.getFileLocation() ⇒ <code>string</code>
Returns the file location associated with the exception or null

**Kind**: instance method of [<code>BaseFileException</code>](#module_concerto-
core.BaseFileException)
**Returns**: <code>string</code> - the optional location associated with the
exception
<a name="module_concerto-core.BaseFileException+getShortMessage"></a>

#### baseFileException.getShortMessage() ⇒ <code>string</code>
Returns the error message without the location of the error

**Kind**: instance method of [<code>BaseFileException</code>](#module_concerto-
core.BaseFileException)
**Returns**: <code>string</code> - the error message
<a name="module_concerto-core.BaseFileException+getFileName"></a>

#### baseFileException.getFileName() ⇒ <code>string</code>
Returns the fileName for the error

**Kind**: instance method of [<code>BaseFileException</code>](#module_concerto-
core.BaseFileException)
**Returns**: <code>string</code> - the file name or null
<a name="module_concerto-core.Factory"></a>

### concerto-core~Factory
Use the Factory to create instances of Resource: transactions, participants
and assets.

**Kind**: inner class of [<code>concerto-core</code>](#module_concerto-core)

* [~Factory](#module_concerto-core.Factory)
    * [new Factory(modelManager)](#new_module_concerto-core.Factory_new)
    * _instance_
        * [.newResource(ns, type, id, [options])](#module_concerto-
core.Factory+newResource) ⇒ <code>Resource</code>
        * [.newConcept(ns, type, [options])](#module_concerto-
core.Factory+newConcept) ⇒ <code>Resource</code>
        * [.newRelationship(ns, type, id)](#module_concerto-
core.Factory+newRelationship) ⇒ <code>Relationship</code>
        * [.newTransaction(ns, type, [id], [options])](#module_concerto-
core.Factory+newTransaction) ⇒ <code>Resource</code>
        * [.newEvent(ns, type, [id], [options])](#module_concerto-
core.Factory+newEvent) ⇒ <code>Resource</code>
    * _static_
        * [.Symbol.hasInstance(object)](#module_concerto-
core.Factory.Symbol.hasInstance) ⇒ <code>boolean</code>
```

```
<a name="new_module_concerto-core.Factory_new"></a>

#### new Factory(modelManager)
Create the factory.


| Param | Type | Description |
| --- | --- | --- |
| modelManager | <code>ModelManager</code> | The ModelManager to use for this
registry |

<a name="module_concerto-core.Factory+newResource"></a>

#### factory.newResource(ns, type, id, [options]) ⇒ <code>Resource</code>
Create a new Resource with a given namespace, type name and id

**Kind**: instance method of [<code>Factory</code>](#module_concerto-core.Factory)
**Returns**: <code>Resource</code> - the new instance
**Throws**:

- <code>TypeNotFoundException</code> if the type is not registered with the
ModelManager


| Param | Type | Description |
| --- | --- | --- |
| ns | <code>String</code> | the namespace of the Resource |
| type | <code>String</code> | the type of the Resource |
| id | <code>String</code> | the identifier |
| [options] | <code>Object</code> | an optional set of options |
| [options.disableValidation] | <code>boolean</code> | pass true if you want the
factory to return a [Resource](Resource) instead of a [ValidatedResource]
(ValidatedResource). Defaults to false. |
| [options.generate] | <code>String</code> | Pass one of: <dl>
<dt>sample</dt><dd>return a resource instance with generated sample data.</dd>
<dt>empty</dt><dd>return a resource instance with empty property values.</dd></dl>
|
| [options.includeOptionalFields] | <code>boolean</code> | if
<code>options.generate</code> is specified, whether optional fields should be
generated. |
| [options.allowEmptyId] | <code>boolean</code> | if
<code>options.allowEmptyId</code> is specified as true, a zero length string for id
is allowed (allows it to be filled in later). |

<a name="module_concerto-core.Factory+newConcept"></a>

#### factory.newConcept(ns, type, [options]) ⇒ <code>Resource</code>
Create a new Concept with a given namespace and type name

**Kind**: instance method of [<code>Factory</code>](#module_concerto-core.Factory)
**Returns**: <code>Resource</code> - the new instance
**Throws**:

- <code>TypeNotFoundException</code> if the type is not registered with the
ModelManager


| Param | Type | Description |
| --- | --- | --- |
```

ns	<code>String</code>	the namespace of the Concept
type	<code>String</code>	the type of the Concept
[options]	<code>Object</code>	an optional set of options
[options.disableValidation]	<code>boolean</code>	pass true if you want the factory to return a Concept instead of a ValidatedConcept. Defaults to false.
[options.generate]	<code>String</code>	Pass one of: <dl>
<dt>sample</dt><dd>return a resource instance with generated sample data.</dd>
<dt>empty</dt><dd>return a resource instance with empty property values.</dd></dl>
[options.includeOptionalFields]	<code>boolean</code>	if <code>options.generate</code> is specified, whether optional fields should be generated.

<a name="module_concerto-core.Factory+newRelationship"></a>

#### factory.newRelationship(ns, type, id) ⇒ <code>Relationship</code>
Create a new Relationship with a given namespace, type and identifier.
A relationship is a typed pointer to an instance. I.e the relationship
with `namespace = 'org.example'`, `type = 'Vehicle'` and `id = 'ABC' creates`
a pointer that points at an instance of org.example.Vehicle with the id
ABC.

Kind: instance method of [<code>Factory</code>](#module_concerto-core.Factory)
Returns: <code>Relationship</code> - - the new relationship instance
Throws:

- <code>TypeNotFoundException</code> if the type is not registered with the ModelManager

Param	Type	Description
ns	<code>String</code>	the namespace of the Resource
type	<code>String</code>	the type of the Resource
id	<code>String</code>	the identifier

<a name="module_concerto-core.Factory+newTransaction"></a>

#### factory.newTransaction(ns, type, [id], [options]) ⇒ <code>Resource</code>
Create a new transaction object. The identifier of the transaction is
set to a UUID.

Kind: instance method of [<code>Factory</code>](#module_concerto-core.Factory)
Returns: <code>Resource</code> - A resource for the new transaction.

Param	Type	Description
ns	<code>String</code>	the namespace of the transaction.
type	<code>String</code>	the type of the transaction.
[id]	<code>String</code>	an optional identifier for the transaction; if you do not specify one then an identifier will be automatically generated.
[options]	<code>Object</code>	an optional set of options
[options.generate]	<code>String</code>	Pass one of: <dl>
<dt>sample</dt><dd>return a resource instance with generated sample data.</dd>
<dt>empty</dt><dd>return a resource instance with empty property values.</dd></dl>
[options.includeOptionalFields]	<code>boolean</code>	if <code>options.generate</code> is specified, whether optional fields should be

```
generated. |
| [options.allowEmptyId] | <code>boolean</code> | if
<code>options.allowEmptyId</code> is specified as true, a zero length string for id
is allowed (allows it to be filled in later). |

<a name="module_concerto-core.Factory+newEvent"></a>

#### factory.newEvent(ns, type, [id], [options]) ⇒ <code>Resource</code>
Create a new event object. The identifier of the event is
set to a UUID.

**Kind**: instance method of [<code>Factory</code>](#module_concerto-core.Factory)
**Returns**: <code>Resource</code> - A resource for the new event.

| Param | Type | Description |
| --- | --- | --- |
| ns | <code>String</code> | the namespace of the event. |
| type | <code>String</code> | the type of the event. |
| [id] | <code>String</code> | an optional identifier for the event; if you do not
specify one then an identifier will be automatically generated. |
| [options] | <code>Object</code> | an optional set of options |
| [options.generate] | <code>String</code> | Pass one of: <dl>
<dt>sample</dt><dd>return a resource instance with generated sample data.</dd>
<dt>empty</dt><dd>return a resource instance with empty property values.</dd></dl>
|
| [options.includeOptionalFields] | <code>boolean</code> | if
<code>options.generate</code> is specified, whether optional fields should be
generated. |
| [options.allowEmptyId] | <code>boolean</code> | if
<code>options.allowEmptyId</code> is specified as true, a zero length string for id
is allowed (allows it to be filled in later). |

<a name="module_concerto-core.Factory.Symbol.hasInstance"></a>

#### Factory.Symbol.hasInstance(object) ⇒ <code>boolean</code>
Alternative instanceof that is reliable across different module instances

**Kind**: static method of [<code>Factory</code>](#module_concerto-core.Factory)
**Returns**: <code>boolean</code> - - True, if the object is an instance of a
Factory
**See**: https://github.com/hyperledger/composer-concerto/issues/47

| Param | Type | Description |
| --- | --- | --- |
| object | <code>object</code> | The object to test against |

<a name="module_concerto-core.ModelManager"></a>

### concerto-core~ModelManager
Manages the Concerto model files.

The structure of [Resource](Resource)s (Assets, Transactions, Participants) is
modelled
in a set of Concerto files. The contents of these files are managed
by the [ModelManager](ModelManager). Each Concerto file has a single namespace and
contains
a set of asset, transaction and participant type definitions.

Concerto applications load their Concerto files and then call the [addModelFile]
```

```
(ModelManager#addModelFile)
method to register the Concerto file(s) with the ModelManager. The ModelManager
parses the text of the Concerto file and will make all defined types available
to other Concerto services, such as the [Serializer](Serializer) (to convert
instances to/from JSON)
and [Factory](Factory) (to create instances).

**Kind**: inner class of [<code>concerto-core</code>](#module_concerto-core)

* [~ModelManager](#module_concerto-core.ModelManager)
    * [new ModelManager()](#new_module_concerto-core.ModelManager_new)
    * _instance_
        * [.validateModelFile(modelFile, fileName)](#module_concerto-
core.ModelManager+validateModelFile)
        * [.addModelFile(modelFile, fileName, [disableValidation],
[systemModelTable])](#module_concerto-core.ModelManager+addModelFile) ⇒
<code>Object</code>
        * [.updateModelFile(modelFile, fileName, [disableValidation])]
(#module_concerto-core.ModelManager+updateModelFile) ⇒ <code>Object</code>
        * [.deleteModelFile(namespace)](#module_concerto-
core.ModelManager+deleteModelFile)
        * [.addModelFiles(modelFiles, [fileNames], [disableValidation],
[systemModelTable])](#module_concerto-core.ModelManager+addModelFiles) ⇒
<code>Array.&lt;Object&gt;</code>
        * [.validateModelFiles()](#module_concerto-
core.ModelManager+validateModelFiles)
        * [.updateExternalModels([options], [modelFileDownloader])]
(#module_concerto-core.ModelManager+updateExternalModels) ⇒ <code>Promise</code>
        * [.writeModelsToFileSystem(path, [options])](#module_concerto-
core.ModelManager+writeModelsToFileSystem)
        * [.getModels([options])](#module_concerto-core.ModelManager+getModels) ⇒
<code>Array.&lt;{name:string, content:string}&gt;</code>
        * [.clearModelFiles()](#module_concerto-core.ModelManager+clearModelFiles)
        * [.getNamespaces()](#module_concerto-core.ModelManager+getNamespaces) ⇒
<code>Array.&lt;string&gt;</code>
        * [.getSystemTypes()](#module_concerto-core.ModelManager+getSystemTypes) ⇒
<code>Array.&lt;ClassDeclaration&gt;</code>
        * [.getAssetDeclarations(includeSystemType)](#module_concerto-
core.ModelManager+getAssetDeclarations) ⇒
<code>Array.&lt;AssetDeclaration&gt;</code>
        * [.getTransactionDeclarations(includeSystemType)](#module_concerto-
core.ModelManager+getTransactionDeclarations) ⇒
<code>Array.&lt;TransactionDeclaration&gt;</code>
        * [.getEventDeclarations(includeSystemType)](#module_concerto-
core.ModelManager+getEventDeclarations) ⇒
<code>Array.&lt;EventDeclaration&gt;</code>
        * [.getParticipantDeclarations(includeSystemType)](#module_concerto-
core.ModelManager+getParticipantDeclarations) ⇒
<code>Array.&lt;ParticipantDeclaration&gt;</code>
        * [.getEnumDeclarations(includeSystemType)](#module_concerto-
core.ModelManager+getEnumDeclarations) ⇒ <code>Array.&lt;EnumDeclaration&gt;</code>
        * [.getConceptDeclarations(includeSystemType)](#module_concerto-
core.ModelManager+getConceptDeclarations) ⇒
<code>Array.&lt;ConceptDeclaration&gt;</code>
        * [.getFactory()](#module_concerto-core.ModelManager+getFactory) ⇒
<code>Factory</code>
        * [.getSerializer()](#module_concerto-core.ModelManager+getSerializer) ⇒
<code>Serializer</code>
        * [.getDecoratorFactories()](#module_concerto-
core.ModelManager+getDecoratorFactories) ⇒
<code>Array.&lt;DecoratorFactory&gt;</code>
        * [.addDecoratorFactory(factory)](#module_concerto-
core.ModelManager+addDecoratorFactory)
    * _static_
        * [.Symbol.hasInstance(object)](#module_concerto-
core.ModelManager.Symbol.hasInstance) ⇒ <code>boolean</code>

<a name="new_module_concerto-core.ModelManager_new"></a>

#### new ModelManager()
Create the ModelManager.

<a name="module_concerto-core.ModelManager+validateModelFile"></a>

#### modelManager.validateModelFile(modelFile, fileName)
Validates a Concerto file (as a string) to the ModelManager.
Concerto files have a single namespace.

Note that if there are dependencies between multiple files the files
must be added in dependency order, or the addModelFiles method can be
used to add a set of files irrespective of dependencies.

**Kind**: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)
**Throws**:

- <code>IllegalModelException</code>


| Param | Type | Description |
| --- | --- | --- |
| modelFile | <code>string</code> | The Concerto file as a string |
| fileName | <code>string</code> | an optional file name to associate with the
model file |

<a name="module_concerto-core.ModelManager+addModelFile"></a>

#### modelManager.addModelFile(modelFile, fileName, [disableValidation],
[systemModelTable]) ⇒ <code>Object</code>
Adds a Concerto file (as a string) to the ModelManager.
Concerto files have a single namespace. If a Concerto file with the
same namespace has already been added to the ModelManager then it
will be replaced.
Note that if there are dependencies between multiple files the files
must be added in dependency order, or the addModelFiles method can be
used to add a set of files irrespective of dependencies.

**Kind**: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)
**Returns**: <code>Object</code> - The newly added model file (internal).
**Throws**:

- <code>IllegalModelException</code>


| Param | Type | Description |
| --- | --- | --- |
| modelFile | <code>string</code> | The Concerto file as a string |
| fileName | <code>string</code> | an optional file name to associate with the
model file |
| [disableValidation] | <code>boolean</code> | If true then the model files are not
validated |
| [systemModelTable] | <code>boolean</code> | A table that maps classes in the new
models to system types |

<a name="module_concerto-core.ModelManager+updateModelFile"></a>

#### modelManager.updateModelFile(modelFile, fileName, [disableValidation]) ⇒
<code>Object</code>
Updates a Concerto file (as a string) on the ModelManager.
Concerto files have a single namespace. If a Concerto file with the
same namespace has already been added to the ModelManager then it
will be replaced.

**Kind**: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)
**Returns**: <code>Object</code> - The newly added model file (internal).
**Throws**:

- <code>IllegalModelException</code>


| Param | Type | Description |
| --- | --- | --- |
| modelFile | <code>string</code> | The Concerto file as a string |
| fileName | <code>string</code> | an optional file name to associate with the
model file |
| [disableValidation] | <code>boolean</code> | If true then the model files are not
validated |

<a name="module_concerto-core.ModelManager+deleteModelFile"></a>

#### modelManager.deleteModelFile(namespace)
Remove the Concerto file for a given namespace

**Kind**: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)

| Param | Type | Description |
| --- | --- | --- |
| namespace | <code>string</code> | The namespace of the model file to delete. |

<a name="module_concerto-core.ModelManager+addModelFiles"></a>

#### modelManager.addModelFiles(modelFiles, [fileNames], [disableValidation],
[systemModelTable]) ⇒ <code>Array.&lt;Object&gt;</code>
Add a set of Concerto files to the model manager.

**Kind**: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)
**Returns**: <code>Array.&lt;Object&gt;</code> - The newly added model files
(internal).

| Param | Type | Description |
| --- | --- | --- |
| modelFiles | <code>Array.&lt;string&gt;</code> | An array of Concerto files as
strings. |
```

[fileNames]	<code>Array.&lt;string&gt;</code>	An optional array of file names to associate with the model files
[disableValidation]	<code>boolean</code>	If true then the model files are not validated
[systemModelTable]	<code>boolean</code>	A table that maps classes in the new models to system types

<a name="module_concerto-core.ModelManager+validateModelFiles"></a>

#### modelManager.validateModelFiles()
Validates all models files in this model manager

Kind: instance method of [<code>ModelManager</code>](#module_concerto-core.ModelManager)
<a name="module_concerto-core.ModelManager+updateExternalModels"></a>

#### modelManager.updateExternalModels([options], [modelFileDownloader]) ⇒ <code>Promise</code>
Downloads all ModelFiles that are external dependencies and adds or
updates them in this ModelManager.

Kind: instance method of [<code>ModelManager</code>](#module_concerto-core.ModelManager)
Returns: <code>Promise</code> - a promise when the download and update operation is completed.
Throws:

- <code>IllegalModelException</code> if the models fail validation

Param	Type	Description
[options]	<code>Object</code>	Options object passed to ModelFileLoaders
[modelFileDownloader]	<code>ModelFileDownloader</code>	an optional ModelFileDownloader

<a name="module_concerto-core.ModelManager+writeModelsToFileSystem"></a>

#### modelManager.writeModelsToFileSystem(path, [options])
Write all models in this model manager to the specified path in the file system

Kind: instance method of [<code>ModelManager</code>](#module_concerto-core.ModelManager)

Param	Type	Description
path	<code>String</code>	to a local directory
[options]	<code>Object</code>	Options object
options.includeExternalModels	<code>boolean</code>	If true, external models are written to the file system. Defaults to true
options.includeSystemModels	<code>boolean</code>	If true, system models are written to the file system. Defaults to false

<a name="module_concerto-core.ModelManager+getModels"></a>

#### modelManager.getModels([options]) ⇒ <code>Array.&lt;{name:string, content:string}&gt;</code>
Gets all the CTO models

```
**Kind**: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)
**Returns**: <code>Array.&lt;{name:string, content:string}&gt;</code> - the name
and content of each CTO file

| Param | Type | Description |
| --- | --- | --- |
| [options] | <code>Object</code> | Options object |
| options.includeExternalModels | <code>boolean</code> | If true, external models
are written to the file system. Defaults to true |
| options.includeSystemModels | <code>boolean</code> | If true, system models are
written to the file system. Defaults to false |

<a name="module_concerto-core.ModelManager+clearModelFiles"></a>

#### modelManager.clearModelFiles()
Remove all registered Concerto files

**Kind**: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)
<a name="module_concerto-core.ModelManager+getNamespaces"></a>

#### modelManager.getNamespaces() ⇒ <code>Array.&lt;string&gt;</code>
Get the namespaces registered with the ModelManager.

**Kind**: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)
**Returns**: <code>Array.&lt;string&gt;</code> - namespaces - the namespaces that
have been registered.
<a name="module_concerto-core.ModelManager+getSystemTypes"></a>

#### modelManager.getSystemTypes() ⇒ <code>Array.&lt;ClassDeclaration&gt;</code>
Get all class declarations from system namespaces

**Kind**: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)
**Returns**: <code>Array.&lt;ClassDeclaration&gt;</code> - the ClassDeclarations
from system namespaces
<a name="module_concerto-core.ModelManager+getAssetDeclarations"></a>

#### modelManager.getAssetDeclarations(includeSystemType) ⇒
<code>Array.&lt;AssetDeclaration&gt;</code>
Get the AssetDeclarations defined in this model manager

**Kind**: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)
**Returns**: <code>Array.&lt;AssetDeclaration&gt;</code> - the AssetDeclarations
defined in the model manager

| Param | Type | Default | Description |
| --- | --- | --- | --- |
| includeSystemType | <code>Boolean</code> | <code>true</code> | Include the
decalarations of system type in returned data |

<a name="module_concerto-core.ModelManager+getTransactionDeclarations"></a>

#### modelManager.getTransactionDeclarations(includeSystemType) ⇒
<code>Array.&lt;TransactionDeclaration&gt;</code>
Get the TransactionDeclarations defined in this model manager
```

```
**Kind**: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)
**Returns**: <code>Array.&lt;TransactionDeclaration&gt;</code> - the
TransactionDeclarations defined in the model manager

| Param | Type | Default | Description |
| --- | --- | --- | --- |
| includeSystemType | <code>Boolean</code> | <code>true</code> | Include the
decalarations of system type in returned data |

<a name="module_concerto-core.ModelManager+getEventDeclarations"></a>

#### modelManager.getEventDeclarations(includeSystemType) ⇒
<code>Array.&lt;EventDeclaration&gt;</code>
Get the EventDeclarations defined in this model manager

**Kind**: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)
**Returns**: <code>Array.&lt;EventDeclaration&gt;</code> - the EventDeclaration
defined in the model manager

| Param | Type | Default | Description |
| --- | --- | --- | --- |
| includeSystemType | <code>Boolean</code> | <code>true</code> | Include the
decalarations of system type in returned data |

<a name="module_concerto-core.ModelManager+getParticipantDeclarations"></a>

#### modelManager.getParticipantDeclarations(includeSystemType) ⇒
<code>Array.&lt;ParticipantDeclaration&gt;</code>
Get the ParticipantDeclarations defined in this model manager

**Kind**: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)
**Returns**: <code>Array.&lt;ParticipantDeclaration&gt;</code> - the
ParticipantDeclaration defined in the model manager

| Param | Type | Default | Description |
| --- | --- | --- | --- |
| includeSystemType | <code>Boolean</code> | <code>true</code> | Include the
decalarations of system type in returned data |

<a name="module_concerto-core.ModelManager+getEnumDeclarations"></a>

#### modelManager.getEnumDeclarations(includeSystemType) ⇒
<code>Array.&lt;EnumDeclaration&gt;</code>
Get the EnumDeclarations defined in this model manager

**Kind**: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)
**Returns**: <code>Array.&lt;EnumDeclaration&gt;</code> - the EnumDeclaration
defined in the model manager

| Param | Type | Default | Description |
| --- | --- | --- | --- |
| includeSystemType | <code>Boolean</code> | <code>true</code> | Include the
decalarations of system type in returned data |
<a name="module_concerto-core.ModelManager+getConceptDeclarations"></a>

#### modelManager.getConceptDeclarations(includeSystemType) ⇒
<code>Array.&lt;ConceptDeclaration&gt;</code>
Get the Concepts defined in this model manager

**Kind**: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)
**Returns**: <code>Array.&lt;ConceptDeclaration&gt;</code> - the ConceptDeclaration
defined in the model manager

| Param | Type | Default | Description |
| --- | --- | --- | --- |
| includeSystemType | <code>Boolean</code> | <code>true</code> | Include the
decalarations of system type in returned data |

<a name="module_concerto-core.ModelManager+getFactory"></a>

#### modelManager.getFactory() ⇒ <code>Factory</code>
Get a factory for creating new instances of types defined in this model manager.

**Kind**: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)
**Returns**: <code>Factory</code> - A factory for creating new instances of types
defined in this model manager.
<a name="module_concerto-core.ModelManager+getSerializer"></a>

#### modelManager.getSerializer() ⇒ <code>Serializer</code>
Get a serializer for serializing instances of types defined in this model manager.

**Kind**: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)
**Returns**: <code>Serializer</code> - A serializer for serializing instances of
types defined in this model manager.
<a name="module_concerto-core.ModelManager+getDecoratorFactories"></a>

#### modelManager.getDecoratorFactories() ⇒
<code>Array.&lt;DecoratorFactory&gt;</code>
Get the decorator factories for this model manager.

**Kind**: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)
**Returns**: <code>Array.&lt;DecoratorFactory&gt;</code> - The decorator factories
for this model manager.
<a name="module_concerto-core.ModelManager+addDecoratorFactory"></a>

#### modelManager.addDecoratorFactory(factory)
Add a decorator factory to this model manager.

**Kind**: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)

| Param | Type | Description |
| --- | --- | --- |
| factory | <code>DecoratorFactory</code> | The decorator factory to add to this
model manager. |

<a name="module_concerto-core.ModelManager.Symbol.hasInstance"></a>
```

```
#### ModelManager.Symbol.hasInstance(object) ⇒ <code>boolean</code>
Alternative instanceof that is reliable across different module instances

**Kind**: static method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)
**Returns**: <code>boolean</code> - - True, if the object is an instance of a
ModelManager
**See**: https://github.com/hyperledger/composer-concerto/issues/47

| Param | Type | Description |
| --- | --- | --- |
| object | <code>object</code> | The object to test against |

<a name="module_concerto-core.SecurityException"></a>

### concerto-core~SecurityException ⇐ <code>BaseException</code>
Class representing a security exception

**Kind**: inner class of [<code>concerto-core</code>](#module_concerto-core)
**Extends**: <code>BaseException</code>
**See**: See [BaseException](BaseException)
<a name="new_module_concerto-core.SecurityException_new"></a>

#### new SecurityException(message)
Create the SecurityException.


| Param | Type | Description |
| --- | --- | --- |
| message | <code>string</code> | The exception message. |

<a name="module_concerto-core.Serializer"></a>

### concerto-core~Serializer
Serialize Resources instances to/from various formats for long-term storage
(e.g. on the blockchain).

**Kind**: inner class of [<code>concerto-core</code>](#module_concerto-core)

* [~Serializer](#module_concerto-core.Serializer)
    * [new Serializer(factory, modelManager)](#new_module_concerto-
core.Serializer_new)
    * _instance_
        * [.setDefaultOptions(newDefaultOptions)](#module_concerto-
core.Serializer+setDefaultOptions)
        * [.toJSON(resource, [options])](#module_concerto-core.Serializer+toJSON) ⇒
<code>Object</code>
        * [.fromJSON(jsonObject, options)](#module_concerto-
core.Serializer+fromJSON) ⇒ <code>Resource</code>
    * _static_
        * [.Symbol.hasInstance(object)](#module_concerto-
core.Serializer.Symbol.hasInstance) ⇒ <code>boolean</code>

<a name="new_module_concerto-core.Serializer_new"></a>

#### new Serializer(factory, modelManager)
Create a Serializer.
```

Param	Type	Description
factory	<code>Factory</code>	The Factory to use to create instances
modelManager	<code>ModelManager</code>	The ModelManager to use for validation etc.

<a name="module_concerto-core.Serializer+setDefaultOptions"></a>

#### serializer.setDefaultOptions(newDefaultOptions)
Set the default options for the serializer.

Kind: instance method of [<code>Serializer</code>](#module_concerto-core.Serializer)

Param	Type	Description
newDefaultOptions	<code>Object</code>	The new default options for the serializer.

<a name="module_concerto-core.Serializer+toJSON"></a>

#### serializer.toJSON(resource, [options]) ⇒ <code>Object</code>
<p>
Convert a Resource to a JavaScript object suitable for long-term peristent storage.
</p>

Kind: instance method of [<code>Serializer</code>](#module_concerto-core.Serializer)
Returns: <code>Object</code> - - The Javascript Object that represents the resource
Throws:

- <code>Error</code> - throws an exception if resource is not an instance of Resource or fails validation.

Param	Type	Description
resource	<code>Resource</code>	The instance to convert to JSON
[options]	<code>Object</code>	the optional serialization options.
[options.validate]	<code>boolean</code>	validate the structure of the Resource with its model prior to serialization (default to true)
[options.convertResourcesToRelationships]	<code>boolean</code>	Convert resources that are specified for relationship fields into relationships, false by default.
[options.permitResourcesForRelationships]	<code>boolean</code>	Permit resources in the place of relationships (serializing them as resources), false by default.
[options.deduplicateResources]	<code>boolean</code>	Generate $id for resources and if a resources appears multiple times in the object graph only the first instance is serialized in full, subsequent instances are replaced with a reference to the $id
[options.convertResourcesToId]	<code>boolean</code>	Convert resources that are specified for relationship fields into their id, false by default.

<a name="module_concerto-core.Serializer+fromJSON"></a>

#### serializer.fromJSON(jsonObject, options) ⇒ <code>Resource</code>

```
Create a [Resource](Resource) from a JavaScript Object representation.
The JavaScript Object should have been created by calling the
[toJSON](Serializer#toJSON) API.

The Resource is populated based on the JavaScript object.

**Kind**: instance method of [<code>Serializer</code>](#module_concerto-
core.Serializer)
**Returns**: <code>Resource</code> - The new populated resource

| Param | Type | Description |
| --- | --- | --- |
| jsonObject | <code>Object</code> | The JavaScript Object for a Resource |
| options | <code>Object</code> | the optional serialization options |
| options.acceptResourcesForRelationships | <code>boolean</code> | handle JSON
objects in the place of strings for relationships, defaults to false. |
| options.validate | <code>boolean</code> | validate the structure of the Resource
with its model prior to serialization (default to true) |

<a name="module_concerto-core.Serializer.Symbol.hasInstance"></a>

#### Serializer.Symbol.hasInstance(object) ⇒ <code>boolean</code>
Alternative instanceof that is reliable across different module instances

**Kind**: static method of [<code>Serializer</code>](#module_concerto-
core.Serializer)
**Returns**: <code>boolean</code> - - True, if the object is an instance of a
Serializer
**See**: https://github.com/hyperledger/composer-concerto/issues/47

| Param | Type | Description |
| --- | --- | --- |
| object | <code>object</code> | The object to test against |

<a name="module_concerto-core.AssetDeclaration"></a>

### concerto-core~AssetDeclaration ⇐ <code>ClassDeclaration</code>
AssetDeclaration defines the schema (aka model or class) for
an Asset. It extends ClassDeclaration which manages a set of
fields, a super-type and the specification of an
identifying field.

**Kind**: inner class of [<code>concerto-core</code>](#module_concerto-core)
**Extends**: <code>ClassDeclaration</code>
**See**: See [ClassDeclaration](ClassDeclaration)

* [~AssetDeclaration](#module_concerto-core.AssetDeclaration) ⇐
<code>ClassDeclaration</code>
    * [new AssetDeclaration(modelFile, ast)](#new_module_concerto-
core.AssetDeclaration_new)
    * _instance_
        * [.isRelationshipTarget()](#module_concerto-
core.AssetDeclaration+isRelationshipTarget) ⇒ <code>boolean</code>
        * [.getSystemType()](#module_concerto-core.AssetDeclaration+getSystemType)
⇒ <code>string</code>
    * _static_
        * [.Symbol.hasInstance(object)](#module_concerto-
core.AssetDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>
```

<a name="new_module_concerto-core.AssetDeclaration_new"></a>

#### new AssetDeclaration(modelFile, ast)
Create an AssetDeclaration.

Throws:

- <code>IllegalModelException</code>

Param	Type	Description
modelFile	<code>ModelFile</code>	the ModelFile for this class
ast	<code>Object</code>	The AST created by the parser

<a name="module_concerto-core.AssetDeclaration+isRelationshipTarget"></a>

#### assetDeclaration.isRelationshipTarget() ⇒ <code>boolean</code>
Returns true if this class can be pointed to by a relationship

Kind: instance method of [<code>AssetDeclaration</code>](#module_concerto-core.AssetDeclaration)
Returns: <code>boolean</code> - true if the class may be pointed to by a relationship
<a name="module_concerto-core.AssetDeclaration+getSystemType"></a>

#### assetDeclaration.getSystemType() ⇒ <code>string</code>
Returns the base system type for Assets from the system namespace

Kind: instance method of [<code>AssetDeclaration</code>](#module_concerto-core.AssetDeclaration)
Returns: <code>string</code> - the short name of the base system type
<a name="module_concerto-core.AssetDeclaration.Symbol.hasInstance"></a>

#### AssetDeclaration.Symbol.hasInstance(object) ⇒ <code>boolean</code>
Alternative instanceof that is reliable across different module instances

Kind: static method of [<code>AssetDeclaration</code>](#module_concerto-core.AssetDeclaration)
Returns: <code>boolean</code> - - True, if the object is an instance of a AssetDeclaration
See: https://github.com/hyperledger/composer-concerto/issues/47

Param	Type	Description
object	<code>object</code>	The object to test against

<a name="module_concerto-core.ClassDeclaration"></a>

### *concerto-core~ClassDeclaration*
ClassDeclaration defines the structure (model/schema) of composite data.
It is composed of a set of Properties, may have an identifying field, and may have a super-type.
A ClassDeclaration is conceptually owned by a ModelFile which
defines all the classes that are part of a namespace.

Kind: inner abstract class of [<code>concerto-core</code>](#module_concerto-core)

```
* *[~ClassDeclaration](#module_concerto-core.ClassDeclaration)*
    * *[new ClassDeclaration(modelFile, ast)](#new_module_concerto-
core.ClassDeclaration_new)*
    * _instance_
        * *[._resolveSuperType()](#module_concerto-
core.ClassDeclaration+_resolveSuperType) ⇒ <code>ClassDeclaration</code>*
        * *[.getSystemType()](#module_concerto-core.ClassDeclaration+getSystemType)
⇒ <code>string</code>*
        * *[.isAbstract()](#module_concerto-core.ClassDeclaration+isAbstract) ⇒
<code>boolean</code>*
        * *[.isEnum()](#module_concerto-core.ClassDeclaration+isEnum) ⇒
<code>boolean</code>*
        * *[.isConcept()](#module_concerto-core.ClassDeclaration+isConcept) ⇒
<code>boolean</code>*
        * *[.isEvent()](#module_concerto-core.ClassDeclaration+isEvent) ⇒
<code>boolean</code>*
        * *[.isRelationshipTarget()](#module_concerto-
core.ClassDeclaration+isRelationshipTarget) ⇒ <code>boolean</code>*
        * *[.isSystemRelationshipTarget()](#module_concerto-
core.ClassDeclaration+isSystemRelationshipTarget) ⇒ <code>boolean</code>*
        * *[.isSystemType()](#module_concerto-core.ClassDeclaration+isSystemType) ⇒
<code>boolean</code>*
        * *[.isSystemCoreType()](#module_concerto-
core.ClassDeclaration+isSystemCoreType) ⇒ <code>boolean</code>*
        * *[.getName()](#module_concerto-core.ClassDeclaration+getName) ⇒
<code>string</code>*
        * *[.getNamespace()](#module_concerto-core.ClassDeclaration+getNamespace) ⇒
<code>String</code>*
        * *[.getFullyQualifiedName()](#module_concerto-
core.ClassDeclaration+getFullyQualifiedName) ⇒ <code>string</code>*
        * *[.getIdentifierFieldName()](#module_concerto-
core.ClassDeclaration+getIdentifierFieldName) ⇒ <code>string</code>*
        * *[.getOwnProperty(name)](#module_concerto-
core.ClassDeclaration+getOwnProperty) ⇒ <code>Property</code>*
        * *[.getOwnProperties()](#module_concerto-
core.ClassDeclaration+getOwnProperties) ⇒ <code>Array.&lt;Property&gt;</code>*
        * *[.getSuperType()](#module_concerto-core.ClassDeclaration+getSuperType) ⇒
<code>string</code>*
        * *[.getSuperTypeDeclaration()](#module_concerto-
core.ClassDeclaration+getSuperTypeDeclaration) ⇒ <code>ClassDeclaration</code>*
        * *[.getAssignableClassDeclarations()](#module_concerto-
core.ClassDeclaration+getAssignableClassDeclarations) ⇒
<code>Array.&lt;ClassDeclaration&gt;</code>*
        * *[.getAllSuperTypeDeclarations()](#module_concerto-
core.ClassDeclaration+getAllSuperTypeDeclarations) ⇒
<code>Array.&lt;ClassDeclaration&gt;</code>*
        * *[.getProperty(name)](#module_concerto-core.ClassDeclaration+getProperty)
⇒ <code>Property</code>*
        * *[.getProperties()](#module_concerto-core.ClassDeclaration+getProperties)
⇒ <code>Array.&lt;Property&gt;</code>*
        * *[.getNestedProperty(propertyPath)](#module_concerto-
core.ClassDeclaration+getNestedProperty) ⇒ <code>Property</code>*
        * *[.toString()](#module_concerto-core.ClassDeclaration+toString) ⇒
<code>String</code>*
    * _static_
        * *[.Symbol.hasInstance(object)](#module_concerto-
core.ClassDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>*

<a name="new_module_concerto-core.ClassDeclaration_new"></a>
```

#### *new ClassDeclaration(modelFile, ast)*
Create a ClassDeclaration from an Abstract Syntax Tree. The AST is the
result of parsing.

Throws:

- <code>IllegalModelException</code>

Param	Type	Description
modelFile	<code>ModelFile</code>	the ModelFile for this class
ast	<code>string</code>	the AST created by the parser

<a name="module_concerto-core.ClassDeclaration+_resolveSuperType"></a>

#### *classDeclaration._resolveSuperType() ⇒ <code>ClassDeclaration</code>*
Resolve the super type on this class and store it as an internal property.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-
core.ClassDeclaration)
Returns: <code>ClassDeclaration</code> - The super type, or null if non
specified.
<a name="module_concerto-core.ClassDeclaration+getSystemType"></a>

#### *classDeclaration.getSystemType() ⇒ <code>string</code>*
Returns the base system type for this type of class declaration. Override
this method in derived classes to specify a base system type.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-
core.ClassDeclaration)
Returns: <code>string</code> - the short name of the base system type or null
<a name="module_concerto-core.ClassDeclaration+isAbstract"></a>

#### *classDeclaration.isAbstract() ⇒ <code>boolean</code>*
Returns true if this class is declared as abstract in the model file

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-
core.ClassDeclaration)
Returns: <code>boolean</code> - true if the class is abstract
<a name="module_concerto-core.ClassDeclaration+isEnum"></a>

#### *classDeclaration.isEnum() ⇒ <code>boolean</code>*
Returns true if this class is an enumeration.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-
core.ClassDeclaration)
Returns: <code>boolean</code> - true if the class is an enumerated type
<a name="module_concerto-core.ClassDeclaration+isConcept"></a>

#### *classDeclaration.isConcept() ⇒ <code>boolean</code>*
Returns true if this class is the definition of a concept.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-
core.ClassDeclaration)
Returns: <code>boolean</code> - true if the class is a concept
<a name="module_concerto-core.ClassDeclaration+isEvent"></a>

```
#### *classDeclaration.isEvent() ⇒ <code>boolean</code>*
Returns true if this class is the definition of an event.

**Kind**: instance method of [<code>ClassDeclaration</code>](#module_concerto-
core.ClassDeclaration)
**Returns**: <code>boolean</code> - true if the class is an event
<a name="module_concerto-core.ClassDeclaration+isRelationshipTarget"></a>

#### *classDeclaration.isRelationshipTarget() ⇒ <code>boolean</code>*
Returns true if this class can be pointed to by a relationship

**Kind**: instance method of [<code>ClassDeclaration</code>](#module_concerto-
core.ClassDeclaration)
**Returns**: <code>boolean</code> - true if the class may be pointed to by a
relationship
<a name="module_concerto-core.ClassDeclaration+isSystemRelationshipTarget"></a>

#### *classDeclaration.isSystemRelationshipTarget() ⇒ <code>boolean</code>*
Returns true if this class can be pointed to by a relationship in a
system model

**Kind**: instance method of [<code>ClassDeclaration</code>](#module_concerto-
core.ClassDeclaration)
**Returns**: <code>boolean</code> - true if the class may be pointed to by a
relationship
<a name="module_concerto-core.ClassDeclaration+isSystemType"></a>

#### *classDeclaration.isSystemType() ⇒ <code>boolean</code>*
Returns true is this type is in the system namespace

**Kind**: instance method of [<code>ClassDeclaration</code>](#module_concerto-
core.ClassDeclaration)
**Returns**: <code>boolean</code> - true if the class may be pointed to by a
relationship
<a name="module_concerto-core.ClassDeclaration+isSystemCoreType"></a>

#### *classDeclaration.isSystemCoreType() ⇒ <code>boolean</code>*
Returns true if this class is a system core type - both in the system
namespace, and also one of the system core types (Asset, Participant, etc).

**Kind**: instance method of [<code>ClassDeclaration</code>](#module_concerto-
core.ClassDeclaration)
**Returns**: <code>boolean</code> - true if the class may be pointed to by a
relationship
<a name="module_concerto-core.ClassDeclaration+getName"></a>

#### *classDeclaration.getName() ⇒ <code>string</code>*
Returns the short name of a class. This name does not include the
namespace from the owning ModelFile.

**Kind**: instance method of [<code>ClassDeclaration</code>](#module_concerto-
core.ClassDeclaration)
**Returns**: <code>string</code> - the short name of this class
<a name="module_concerto-core.ClassDeclaration+getNamespace"></a>

#### *classDeclaration.getNamespace() ⇒ <code>String</code>*
Return the namespace of this class.

**Kind**: instance method of [<code>ClassDeclaration</code>](#module_concerto-
```

core.ClassDeclaration)
Returns: <code>String</code> - namespace - a namespace.
<a name="module_concerto-core.ClassDeclaration+getFullyQualifiedName"></a>

#### *classDeclaration.getFullyQualifiedName() ⇒ <code>string</code>*
Returns the fully qualified name of this class.
The name will include the namespace if present.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>string</code> - the fully-qualified name of this class
<a name="module_concerto-core.ClassDeclaration+getIdentifierFieldName"></a>

#### *classDeclaration.getIdentifierFieldName() ⇒ <code>string</code>*
Returns the name of the identifying field for this class. Note
that the identifying field may come from a super type.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>string</code> - the name of the id field for this class
<a name="module_concerto-core.ClassDeclaration+getOwnProperty"></a>

#### *classDeclaration.getOwnProperty(name) ⇒ <code>Property</code>*
Returns the field with a given name or null if it does not exist.
The field must be directly owned by this class -- the super-type is
not introspected.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>Property</code> - the field definition or null if it does not exist.

Param	Type	Description
name	<code>string</code>	the name of the field

<a name="module_concerto-core.ClassDeclaration+getOwnProperties"></a>

#### *classDeclaration.getOwnProperties() ⇒ <code>Array.&lt;Property&gt;</code>*
Returns the fields directly defined by this class.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>Array.&lt;Property&gt;</code> - the array of fields
<a name="module_concerto-core.ClassDeclaration+getSuperType"></a>

#### *classDeclaration.getSuperType() ⇒ <code>string</code>*
Returns the FQN of the super type for this class or null if this
class does not have a super type.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>string</code> - the FQN name of the super type or null
<a name="module_concerto-core.ClassDeclaration+getSuperTypeDeclaration"></a>

#### *classDeclaration.getSuperTypeDeclaration() ⇒ <code>ClassDeclaration</code>*
Get the super type class declaration for this class.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-

core.ClassDeclaration)
Returns: <code>ClassDeclaration</code> - the super type declaration, or null if there is no super type.
<a name="module_concerto-core.ClassDeclaration+getAssignableClassDeclarations"></a>

#### *classDeclaration.getAssignableClassDeclarations() ⇒ <code>Array.&lt;ClassDeclaration&gt;</code>*
Get the class declarations for all subclasses of this class, including this class.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>Array.&lt;ClassDeclaration&gt;</code> - subclass declarations.
<a name="module_concerto-core.ClassDeclaration+getAllSuperTypeDeclarations"></a>

#### *classDeclaration.getAllSuperTypeDeclarations() ⇒ <code>Array.&lt;ClassDeclaration&gt;</code>*
Get all the super-type declarations for this type.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>Array.&lt;ClassDeclaration&gt;</code> - super-type declarations.
<a name="module_concerto-core.ClassDeclaration+getProperty"></a>

#### *classDeclaration.getProperty(name) ⇒ <code>Property</code>*
Returns the property with a given name or null if it does not exist.
Fields defined in super-types are also introspected.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>Property</code> - the field, or null if it does not exist

Param	Type	Description
name	<code>string</code>	the name of the field

<a name="module_concerto-core.ClassDeclaration+getProperties"></a>

#### *classDeclaration.getProperties() ⇒ <code>Array.&lt;Property&gt;</code>*
Returns the properties defined in this class and all super classes.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>Array.&lt;Property&gt;</code> - the array of fields
<a name="module_concerto-core.ClassDeclaration+getNestedProperty"></a>

#### *classDeclaration.getNestedProperty(propertyPath) ⇒ <code>Property</code>*
Get a nested property using a dotted property path

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>Property</code> - the property
Throws:

- <code>IllegalModelException</code> if the property path is invalid or the property does not exist

Param	Type	Description

```
| propertyPath | <code>string</code> | The property name or name with nested
structure e.g a.b.c |

<a name="module_concerto-core.ClassDeclaration+toString"></a>

#### *classDeclaration.toString() ⇒ <code>String</code>*
Returns the string representation of this class

**Kind**: instance method of [<code>ClassDeclaration</code>](#module_concerto-
core.ClassDeclaration)
**Returns**: <code>String</code> - the string representation of the class
<a name="module_concerto-core.ClassDeclaration.Symbol.hasInstance"></a>

#### *ClassDeclaration.Symbol.hasInstance(object) ⇒ <code>boolean</code>*
Alternative instanceof that is reliable across different module instances

**Kind**: static method of [<code>ClassDeclaration</code>](#module_concerto-
core.ClassDeclaration)
**Returns**: <code>boolean</code> - - True, if the object is an instance of a Class
Declaration
**See**: https://github.com/hyperledger/composer-concerto/issues/47

| Param | Type | Description |
| --- | --- | --- |
| object | <code>object</code> | The object to test against |

<a name="module_concerto-core.ConceptDeclaration"></a>

### concerto-core~ConceptDeclaration ⇐ <code>ClassDeclaration</code>
ConceptDeclaration defines the schema (aka model or class) for
an Concept. It extends ClassDeclaration which manages a set of
fields, a super-type and the specification of an
identifying field.

**Kind**: inner class of [<code>concerto-core</code>](#module_concerto-core)
**Extends**: <code>ClassDeclaration</code>
**See**: [ClassDeclaration](ClassDeclaration)

* [~ConceptDeclaration](#module_concerto-core.ConceptDeclaration) ⇐
<code>ClassDeclaration</code>
    * [new ConceptDeclaration(modelFile, ast)](#new_module_concerto-
core.ConceptDeclaration_new)
    * _instance_
        * [.isConcept()](#module_concerto-core.ConceptDeclaration+isConcept) ⇒
<code>boolean</code>
    * _static_
        * [.Symbol.hasInstance(object)](#module_concerto-
core.ConceptDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>

<a name="new_module_concerto-core.ConceptDeclaration_new"></a>

#### new ConceptDeclaration(modelFile, ast)
Create an AssetDeclaration.

**Throws**:

- <code>IllegalModelException</code>
```

```
| Param | Type | Description |
| --- | --- | --- |
| modelFile | <code>ModelFile</code> | the ModelFile for this class |
| ast | <code>Object</code> | The AST created by the parser |

<a name="module_concerto-core.ConceptDeclaration+isConcept"></a>

#### conceptDeclaration.isConcept() ⇒ <code>boolean</code>
Returns true if this class is the definition of a concept.

**Kind**: instance method of [<code>ConceptDeclaration</code>](#module_concerto-
core.ConceptDeclaration)
**Returns**: <code>boolean</code> - true if the class is a concept
<a name="module_concerto-core.ConceptDeclaration.Symbol.hasInstance"></a>

#### ConceptDeclaration.Symbol.hasInstance(object) ⇒ <code>boolean</code>
Alternative instanceof that is reliable across different module instances

**Kind**: static method of [<code>ConceptDeclaration</code>](#module_concerto-
core.ConceptDeclaration)
**Returns**: <code>boolean</code> - - True, if the object is an instance of a
ConceptDeclaration
**See**: https://github.com/hyperledger/composer-concerto/issues/47

| Param | Type | Description |
| --- | --- | --- |
| object | <code>object</code> | The object to test against |

<a name="module_concerto-core.Decorator"></a>

### concerto-core~Decorator
Decorator encapsulates a decorator (annotation) on a class or property.

**Kind**: inner class of [<code>concerto-core</code>](#module_concerto-core)

* [~Decorator](#module_concerto-core.Decorator)
    * [new Decorator(parent, ast)](#new_module_concerto-core.Decorator_new)
    * [.getParent()](#module_concerto-core.Decorator+getParent) ⇒
<code>ClassDeclaration</code> \| <code>Property</code>
    * [.getName()](#module_concerto-core.Decorator+getName) ⇒ <code>string</code>
    * [.getArguments()](#module_concerto-core.Decorator+getArguments) ⇒
<code>Array.&lt;object&gt;</code>

<a name="new_module_concerto-core.Decorator_new"></a>

#### new Decorator(parent, ast)
Create a Decorator.

**Throws**:

- <code>IllegalModelException</code>


| Param | Type | Description |
| --- | --- | --- |
| parent | <code>ClassDeclaration</code> \| <code>Property</code> | the owner of
this property |
| ast | <code>Object</code> | The AST created by the parser |
<a name="module_concerto-core.Decorator+getParent"></a>

#### decorator.getParent() ⇒ <code>ClassDeclaration</code> \| <code>Property</code>
Returns the owner of this property

**Kind**: instance method of [<code>Decorator</code>](#module_concerto-
core.Decorator)
**Returns**: <code>ClassDeclaration</code> \| <code>Property</code> - the parent
class or property declaration
<a name="module_concerto-core.Decorator+getName"></a>

#### decorator.getName() ⇒ <code>string</code>
Returns the name of a decorator

**Kind**: instance method of [<code>Decorator</code>](#module_concerto-
core.Decorator)
**Returns**: <code>string</code> - the name of this decorator
<a name="module_concerto-core.Decorator+getArguments"></a>

#### decorator.getArguments() ⇒ <code>Array.&lt;object&gt;</code>
Returns the arguments for this decorator

**Kind**: instance method of [<code>Decorator</code>](#module_concerto-
core.Decorator)
**Returns**: <code>Array.&lt;object&gt;</code> - the arguments for this decorator
or null if it does not have any arguments
<a name="module_concerto-core.EnumDeclaration"></a>

### concerto-core~EnumDeclaration ⇐ <code>ClassDeclaration</code>
EnumDeclaration defines an enumeration of static values.

**Kind**: inner class of [<code>concerto-core</code>](#module_concerto-core)
**Extends**: <code>ClassDeclaration</code>
**See**: See [ClassDeclaration](ClassDeclaration)

* [~EnumDeclaration](#module_concerto-core.EnumDeclaration) ⇐
<code>ClassDeclaration</code>
    * [new EnumDeclaration(modelFile, ast)](#new_module_concerto-
core.EnumDeclaration_new)
    * _instance_
        * [.isEnum()](#module_concerto-core.EnumDeclaration+isEnum) ⇒
<code>boolean</code>
        * [.toString()](#module_concerto-core.EnumDeclaration+toString) ⇒
<code>String</code>
    * _static_
        * [.Symbol.hasInstance(object)](#module_concerto-
core.EnumDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>

<a name="new_module_concerto-core.EnumDeclaration_new"></a>

#### new EnumDeclaration(modelFile, ast)
Create an AssetDeclaration.

**Throws**:

- <code>IllegalModelException</code>


| Param | Type | Description |
| --- | --- | --- |
| modelFile | <code>ModelFile</code> | the ModelFile for this class |
| ast | <code>Object</code> | The AST created by the parser |

<a name="module_concerto-core.EnumDeclaration+isEnum"></a>

#### enumDeclaration.isEnum() ⇒ <code>boolean</code>
Returns true if this class is an enumeration.

**Kind**: instance method of [<code>EnumDeclaration</code>](#module_concerto-
core.EnumDeclaration)
**Returns**: <code>boolean</code> - true if the class is an enumerated type
<a name="module_concerto-core.EnumDeclaration+toString"></a>

#### enumDeclaration.toString() ⇒ <code>String</code>
Returns the string representation of this class

**Kind**: instance method of [<code>EnumDeclaration</code>](#module_concerto-
core.EnumDeclaration)
**Returns**: <code>String</code> - the string representation of the class
<a name="module_concerto-core.EnumDeclaration.Symbol.hasInstance"></a>

#### EnumDeclaration.Symbol.hasInstance(object) ⇒ <code>boolean</code>
Alternative instanceof that is reliable across different module instances

**Kind**: static method of [<code>EnumDeclaration</code>](#module_concerto-
core.EnumDeclaration)
**Returns**: <code>boolean</code> - - True, if the object is an instance of a Class
Declaration
**See**: https://github.com/hyperledger/composer-concerto/issues/47

| Param | Type | Description |
| --- | --- | --- |
| object | <code>object</code> | The object to test against |

<a name="module_concerto-core.EnumValueDeclaration"></a>

### concerto-core~EnumValueDeclaration ⇐ <code>Property</code>
Class representing a value from a set of enumerated values

**Kind**: inner class of [<code>concerto-core</code>](#module_concerto-core)
**Extends**: <code>Property</code>
**See**: See [Property](Property)

* [~EnumValueDeclaration](#module_concerto-core.EnumValueDeclaration) ⇐
<code>Property</code>
    * [new EnumValueDeclaration(parent, ast)](#new_module_concerto-
core.EnumValueDeclaration_new)
    * [.Symbol.hasInstance(object)](#module_concerto-
core.EnumValueDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>

<a name="new_module_concerto-core.EnumValueDeclaration_new"></a>

#### new EnumValueDeclaration(parent, ast)
Create a EnumValueDeclaration.

**Throws**:

- <code>IllegalModelException</code>
```

```
| Param | Type | Description |
| --- | --- | --- |
| parent | <code>ClassDeclaration</code> | The owner of this property |
| ast | <code>Object</code> | The AST created by the parser |

<a name="module_concerto-core.EnumValueDeclaration.Symbol.hasInstance"></a>

#### EnumValueDeclaration.Symbol.hasInstance(object) ⇒ <code>boolean</code>
Alternative instanceof that is reliable across different module instances

**Kind**: static method of [<code>EnumValueDeclaration</code>](#module_concerto-
core.EnumValueDeclaration)
**Returns**: <code>boolean</code> - - True, if the object is an instance of a
EnumValueDeclaration
**See**: https://github.com/hyperledger/composer-concerto/issues/47

| Param | Type | Description |
| --- | --- | --- |
| object | <code>object</code> | The object to test against |

<a name="module_concerto-core.EventDeclaration"></a>

### concerto-core~EventDeclaration ⇐ <code>ClassDeclaration</code>
Class representing the definition of an Event.

**Kind**: inner class of [<code>concerto-core</code>](#module_concerto-core)
**Extends**: <code>ClassDeclaration</code>
**See**: See  [ClassDeclaration](ClassDeclaration)

* [~EventDeclaration](#module_concerto-core.EventDeclaration) ⇐
<code>ClassDeclaration</code>
    * [new EventDeclaration(modelFile, ast)](#new_module_concerto-
core.EventDeclaration_new)
    * _instance_
        * [.getSystemType()](#module_concerto-core.EventDeclaration+getSystemType)
⇒ <code>string</code>
        * [.isEvent()](#module_concerto-core.EventDeclaration+isEvent) ⇒
<code>boolean</code>
    * _static_
        * [.Symbol.hasInstance(object)](#module_concerto-
core.EventDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>

<a name="new_module_concerto-core.EventDeclaration_new"></a>

#### new EventDeclaration(modelFile, ast)
Create an EventDeclaration.

**Throws**:

- <code>IllegalModelException</code>


| Param | Type | Description |
| --- | --- | --- |
| modelFile | <code>ModelFile</code> | the ModelFile for this class |
| ast | <code>Object</code> | The AST created by the parser |
<a name="module_concerto-core.EventDeclaration+getSystemType"></a>

#### eventDeclaration.getSystemType() ⇒ <code>string</code>
Returns the base system type for Events from the system namespace

**Kind**: instance method of [<code>EventDeclaration</code>](#module_concerto-
core.EventDeclaration)
**Returns**: <code>string</code> - the short name of the base system type
<a name="module_concerto-core.EventDeclaration+isEvent"></a>

#### eventDeclaration.isEvent() ⇒ <code>boolean</code>
Returns true if this class is the definition of an event

**Kind**: instance method of [<code>EventDeclaration</code>](#module_concerto-
core.EventDeclaration)
**Returns**: <code>boolean</code> - true if the class is an event
<a name="module_concerto-core.EventDeclaration.Symbol.hasInstance"></a>

#### EventDeclaration.Symbol.hasInstance(object) ⇒ <code>boolean</code>
Alternative instanceof that is reliable across different module instances

**Kind**: static method of [<code>EventDeclaration</code>](#module_concerto-
core.EventDeclaration)
**Returns**: <code>boolean</code> - - True, if the object is an instance of a
EventDeclaration
**See**: https://github.com/hyperledger/composer-concerto/issues/47

| Param | Type | Description |
| --- | --- | --- |
| object | <code>object</code> | The object to test against |

<a name="module_concerto-core.Introspector"></a>

### concerto-core~Introspector
Provides access to the structure of transactions, assets and participants.

**Kind**: inner class of [<code>concerto-core</code>](#module_concerto-core)

* [~Introspector](#module_concerto-core.Introspector)
    * [new Introspector(modelManager)](#new_module_concerto-core.Introspector_new)
    * [.getClassDeclarations()](#module_concerto-
core.Introspector+getClassDeclarations) ⇒
<code>Array.&lt;ClassDeclaration&gt;</code>
    * [.getClassDeclaration(fullyQualifiedTypeName)](#module_concerto-
core.Introspector+getClassDeclaration) ⇒ <code>ClassDeclaration</code>

<a name="new_module_concerto-core.Introspector_new"></a>

#### new Introspector(modelManager)
Create the Introspector.


| Param | Type | Description |
| --- | --- | --- |
| modelManager | <code>ModelManager</code> | the ModelManager that backs this
Introspector |

<a name="module_concerto-core.Introspector+getClassDeclarations"></a>
```

```
#### introspector.getClassDeclarations() ⇒
<code>Array.&lt;ClassDeclaration&gt;</code>
Returns all the class declarations for the business network.

**Kind**: instance method of [<code>Introspector</code>](#module_concerto-
core.Introspector)
**Returns**: <code>Array.&lt;ClassDeclaration&gt;</code> - the array of class
declarations
<a name="module_concerto-core.Introspector+getClassDeclaration"></a>

#### introspector.getClassDeclaration(fullyQualifiedTypeName) ⇒
<code>ClassDeclaration</code>
Returns the class declaration with the given fully qualified name.
Throws an error if the class declaration does not exist.

**Kind**: instance method of [<code>Introspector</code>](#module_concerto-
core.Introspector)
**Returns**: <code>ClassDeclaration</code> - the class declaration
**Throws**:

- <code>Error</code> if the class declaration does not exist


| Param | Type | Description |
| --- | --- | --- |
| fullyQualifiedTypeName | <code>String</code> | the fully qualified name of the
type |

<a name="module_concerto-core.ModelFile"></a>

### concerto-core~ModelFile
Class representing a Model File. A Model File contains a single namespace
and a set of model elements: assets, transactions etc.

**Kind**: inner class of [<code>concerto-core</code>](#module_concerto-core)

* [~ModelFile](#module_concerto-core.ModelFile)
    * [new ModelFile(modelManager, definitions, [fileName], [isSystemModelFile])]
(#new_module_concerto-core.ModelFile_new)
    * _instance_
        * [.isExternal()](#module_concerto-core.ModelFile+isExternal) ⇒
<code>boolean</code>
        * [.getModelManager()](#module_concerto-core.ModelFile+getModelManager) ⇒
<code>ModelManager</code>
        * [.getImports()](#module_concerto-core.ModelFile+getImports) ⇒
<code>Array.&lt;string&gt;</code>
        * [.isDefined(type)](#module_concerto-core.ModelFile+isDefined) ⇒
<code>boolean</code>
        * [.getLocalType(type)](#module_concerto-core.ModelFile+getLocalType) ⇒
<code>ClassDeclaration</code>
        * [.getAssetDeclaration(name)](#module_concerto-
core.ModelFile+getAssetDeclaration) ⇒ <code>AssetDeclaration</code>
        * [.getTransactionDeclaration(name)](#module_concerto-
core.ModelFile+getTransactionDeclaration) ⇒ <code>TransactionDeclaration</code>
        * [.getEventDeclaration(name)](#module_concerto-
core.ModelFile+getEventDeclaration) ⇒ <code>EventDeclaration</code>
        * [.getParticipantDeclaration(name)](#module_concerto-
core.ModelFile+getParticipantDeclaration) ⇒ <code>ParticipantDeclaration</code>
        * [.getNamespace()](#module_concerto-core.ModelFile+getNamespace) ⇒
```

```
<code>string</code>
        * [.getName()](#module_concerto-core.ModelFile+getName) ⇒
<code>string</code>
        * [.getAssetDeclarations(includeSystemType)](#module_concerto-
core.ModelFile+getAssetDeclarations) ⇒ <code>Array.&lt;AssetDeclaration&gt;</code>
        * [.getTransactionDeclarations(includeSystemType)](#module_concerto-
core.ModelFile+getTransactionDeclarations) ⇒
<code>Array.&lt;TransactionDeclaration&gt;</code>
        * [.getEventDeclarations(includeSystemType)](#module_concerto-
core.ModelFile+getEventDeclarations) ⇒ <code>Array.&lt;EventDeclaration&gt;</code>
        * [.getParticipantDeclarations(includeSystemType)](#module_concerto-
core.ModelFile+getParticipantDeclarations) ⇒
<code>Array.&lt;ParticipantDeclaration&gt;</code>
        * [.getConceptDeclarations(includeSystemType)](#module_concerto-
core.ModelFile+getConceptDeclarations) ⇒
<code>Array.&lt;ConceptDeclaration&gt;</code>
        * [.getEnumDeclarations(includeSystemType)](#module_concerto-
core.ModelFile+getEnumDeclarations) ⇒ <code>Array.&lt;EnumDeclaration&gt;</code>
        * [.getDeclarations(type, includeSystemType)](#module_concerto-
core.ModelFile+getDeclarations) ⇒ <code>Array.&lt;ClassDeclaration&gt;</code>
        * [.getAllDeclarations()](#module_concerto-
core.ModelFile+getAllDeclarations) ⇒ <code>Array.&lt;ClassDeclaration&gt;</code>
        * [.getDefinitions()](#module_concerto-core.ModelFile+getDefinitions) ⇒
<code>string</code>
        * [.isSystemModelFile()](#module_concerto-core.ModelFile+isSystemModelFile)
⇒ <code>boolean</code>
    * _static_
        * [.Symbol.hasInstance(object)](#module_concerto-
core.ModelFile.Symbol.hasInstance) ⇒ <code>boolean</code>

<a name="new_module_concerto-core.ModelFile_new"></a>

#### new ModelFile(modelManager, definitions, [fileName], [isSystemModelFile])
Create a ModelFile. This should only be called by framework code.
Use the ModelManager to manage ModelFiles.

**Throws**:

- <code>IllegalModelException</code>


| Param | Type | Default | Description |
| --- | --- | --- | --- |
| modelManager | <code>ModelManager</code> |  | the ModelManager that manages this
ModelFile |
| definitions | <code>string</code> |  | The DSL model as a string. |
| [fileName] | <code>string</code> |  | The optional filename for this modelfile |
| [isSystemModelFile] | <code>boolean</code> | <code>false</code> | If true, this
is a system model file, defaults to false |

<a name="module_concerto-core.ModelFile+isExternal"></a>

#### modelFile.isExternal() ⇒ <code>boolean</code>
Returns true if this ModelFile was downloaded from an external URI.

**Kind**: instance method of [<code>ModelFile</code>](#module_concerto-
core.ModelFile)
**Returns**: <code>boolean</code> - true iff this ModelFile was downloaded from an
external URI
```

<a name="module_concerto-core.ModelFile+getModelManager"></a>

#### modelFile.getModelManager() ⇒ <code>ModelManager</code>
Returns the ModelManager associated with this ModelFile

Kind: instance method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)
Returns: <code>ModelManager</code> - The ModelManager for this ModelFile
<a name="module_concerto-core.ModelFile+getImports"></a>

#### modelFile.getImports() ⇒ <code>Array.&lt;string&gt;</code>
Returns the types that have been imported into this ModelFile.

Kind: instance method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)
Returns: <code>Array.&lt;string&gt;</code> - The array of imports for this ModelFile
<a name="module_concerto-core.ModelFile+isDefined"></a>

#### modelFile.isDefined(type) ⇒ <code>boolean</code>
Returns true if the type is defined in the model file

Kind: instance method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)
Returns: <code>boolean</code> - true if the type (asset or transaction) is defined

Param	Type	Description
type	<code>string</code>	the name of the type

<a name="module_concerto-core.ModelFile+getLocalType"></a>

#### modelFile.getLocalType(type) ⇒ <code>ClassDeclaration</code>
Returns the type with the specified name or null

Kind: instance method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)
Returns: <code>ClassDeclaration</code> - the ClassDeclaration, or null if the type does not exist

Param	Type	Description
type	<code>string</code>	the short OR FQN name of the type

<a name="module_concerto-core.ModelFile+getAssetDeclaration"></a>

#### modelFile.getAssetDeclaration(name) ⇒ <code>AssetDeclaration</code>
Get the AssetDeclarations defined in this ModelFile or null

Kind: instance method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)
Returns: <code>AssetDeclaration</code> - the AssetDeclaration with the given short name

Param	Type	Description
name	<code>string</code>	the name of the type

<a name="module_concerto-core.ModelFile+getTransactionDeclaration"></a>

#### modelFile.getTransactionDeclaration(name) ⇒
<code>TransactionDeclaration</code>
Get the TransactionDeclaration defined in this ModelFile or null

Kind: instance method of [<code>ModelFile</code>](#module_concerto-
core.ModelFile)
Returns: <code>TransactionDeclaration</code> - the TransactionDeclaration with
the given short name

Param	Type	Description
name	<code>string</code>	the name of the type

<a name="module_concerto-core.ModelFile+getEventDeclaration"></a>

#### modelFile.getEventDeclaration(name) ⇒ <code>EventDeclaration</code>
Get the EventDeclaration defined in this ModelFile or null

Kind: instance method of [<code>ModelFile</code>](#module_concerto-
core.ModelFile)
Returns: <code>EventDeclaration</code> - the EventDeclaration with the given
short name

Param	Type	Description
name	<code>string</code>	the name of the type

<a name="module_concerto-core.ModelFile+getParticipantDeclaration"></a>

#### modelFile.getParticipantDeclaration(name) ⇒
<code>ParticipantDeclaration</code>
Get the ParticipantDeclaration defined in this ModelFile or null

Kind: instance method of [<code>ModelFile</code>](#module_concerto-
core.ModelFile)
Returns: <code>ParticipantDeclaration</code> - the ParticipantDeclaration with
the given short name

Param	Type	Description
name	<code>string</code>	the name of the type

<a name="module_concerto-core.ModelFile+getNamespace"></a>

#### modelFile.getNamespace() ⇒ <code>string</code>
Get the Namespace for this model file.

Kind: instance method of [<code>ModelFile</code>](#module_concerto-
core.ModelFile)
Returns: <code>string</code> - The Namespace for this model file
<a name="module_concerto-core.ModelFile+getName"></a>

#### modelFile.getName() ⇒ <code>string</code>
Get the filename for this model file. Note that this may be null.

Kind: instance method of [<code>ModelFile</code>](#module_concerto-
core.ModelFile)

```
**Returns**: <code>string</code> - The filename for this model file
<a name="module_concerto-core.ModelFile+getAssetDeclarations"></a>

#### modelFile.getAssetDeclarations(includeSystemType) ⇒
<code>Array.&lt;AssetDeclaration&gt;</code>
Get the AssetDeclarations defined in this ModelFile

**Kind**: instance method of [<code>ModelFile</code>](#module_concerto-
core.ModelFile)
**Returns**: <code>Array.&lt;AssetDeclaration&gt;</code> - the AssetDeclarations
defined in the model file

| Param | Type | Default | Description |
| --- | --- | --- | --- |
| includeSystemType | <code>Boolean</code> | <code>true</code> | Include the
decalarations of system type in returned data |

<a name="module_concerto-core.ModelFile+getTransactionDeclarations"></a>

#### modelFile.getTransactionDeclarations(includeSystemType) ⇒
<code>Array.&lt;TransactionDeclaration&gt;</code>
Get the TransactionDeclarations defined in this ModelFile

**Kind**: instance method of [<code>ModelFile</code>](#module_concerto-
core.ModelFile)
**Returns**: <code>Array.&lt;TransactionDeclaration&gt;</code> - the
TransactionDeclarations defined in the model file

| Param | Type | Default | Description |
| --- | --- | --- | --- |
| includeSystemType | <code>Boolean</code> | <code>true</code> | Include the
decalarations of system type in returned data |

<a name="module_concerto-core.ModelFile+getEventDeclarations"></a>

#### modelFile.getEventDeclarations(includeSystemType) ⇒
<code>Array.&lt;EventDeclaration&gt;</code>
Get the EventDeclarations defined in this ModelFile

**Kind**: instance method of [<code>ModelFile</code>](#module_concerto-
core.ModelFile)
**Returns**: <code>Array.&lt;EventDeclaration&gt;</code> - the EventDeclarations
defined in the model file

| Param | Type | Default | Description |
| --- | --- | --- | --- |
| includeSystemType | <code>Boolean</code> | <code>true</code> | Include the
decalarations of system type in returned data |

<a name="module_concerto-core.ModelFile+getParticipantDeclarations"></a>

#### modelFile.getParticipantDeclarations(includeSystemType) ⇒
<code>Array.&lt;ParticipantDeclaration&gt;</code>
Get the ParticipantDeclarations defined in this ModelFile

**Kind**: instance method of [<code>ModelFile</code>](#module_concerto-
core.ModelFile)
**Returns**: <code>Array.&lt;ParticipantDeclaration&gt;</code> - the
ParticipantDeclaration defined in the model file
```

Param	Type	Default	Description
includeSystemType	<code>Boolean</code>	<code>true</code>	Include the decalarations of system type in returned data

<a name="module_concerto-core.ModelFile+getConceptDeclarations"></a>

#### modelFile.getConceptDeclarations(includeSystemType) ⇒ <code>Array.&lt;ConceptDeclaration&gt;</code>
Get the ConceptDeclarations defined in this ModelFile

Kind: instance method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)
Returns: <code>Array.&lt;ConceptDeclaration&gt;</code> - the ParticipantDeclaration defined in the model file

Param	Type	Default	Description
includeSystemType	<code>Boolean</code>	<code>true</code>	Include the decalarations of system type in returned data

<a name="module_concerto-core.ModelFile+getEnumDeclarations"></a>

#### modelFile.getEnumDeclarations(includeSystemType) ⇒ <code>Array.&lt;EnumDeclaration&gt;</code>
Get the EnumDeclarations defined in this ModelFile

Kind: instance method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)
Returns: <code>Array.&lt;EnumDeclaration&gt;</code> - the EnumDeclaration defined in the model file

Param	Type	Default	Description
includeSystemType	<code>Boolean</code>	<code>true</code>	Include the decalarations of system type in returned data

<a name="module_concerto-core.ModelFile+getDeclarations"></a>

#### modelFile.getDeclarations(type, includeSystemType) ⇒ <code>Array.&lt;ClassDeclaration&gt;</code>
Get the instances of a given type in this ModelFile

Kind: instance method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)
Returns: <code>Array.&lt;ClassDeclaration&gt;</code> - the ClassDeclaration defined in the model file

Param	Type	Default	Description
type	<code>function</code>		the type of the declaration
includeSystemType	<code>Boolean</code>	<code>true</code>	Include the decalarations of system type in returned data

<a name="module_concerto-core.ModelFile+getAllDeclarations"></a>

#### modelFile.getAllDeclarations() ⇒ <code>Array.&lt;ClassDeclaration&gt;</code>
Get all declarations in this ModelFile

Kind: instance method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)
Returns: <code>Array.&lt;ClassDeclaration&gt;</code> - the ClassDeclarations defined in the model file
<a name="module_concerto-core.ModelFile+getDefinitions"></a>

#### modelFile.getDefinitions() ⇒ <code>string</code>
Get the definitions for this model.

Kind: instance method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)
Returns: <code>string</code> - The definitions for this model.
<a name="module_concerto-core.ModelFile+isSystemModelFile"></a>

#### modelFile.isSystemModelFile() ⇒ <code>boolean</code>
Returns true if this ModelFile is a system model

Kind: instance method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)
Returns: <code>boolean</code> - true of this ModelFile is a system model
<a name="module_concerto-core.ModelFile.Symbol.hasInstance"></a>

#### ModelFile.Symbol.hasInstance(object) ⇒ <code>boolean</code>
Alternative to instanceof that is reliable across different module instances

Kind: static method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)
Returns: <code>boolean</code> - - True, if the object is an instance of a ModelFile
See: https://github.com/hyperledger/composer-concerto/issues/47

Param	Type	Description
object	<code>object</code>	The object to test against

<a name="module_concerto-core.ParticipantDeclaration"></a>

### concerto-core~ParticipantDeclaration ⇐ <code>ClassDeclaration</code>
Class representing the definition of a Participant.

Kind: inner class of [<code>concerto-core</code>](#module_concerto-core)
Extends: <code>ClassDeclaration</code>
See: See ClassDeclaration

* [~ParticipantDeclaration](#module_concerto-core.ParticipantDeclaration) ⇐ <code>ClassDeclaration</code>
    * [new ParticipantDeclaration(modelFile, ast)](#new_module_concerto-core.ParticipantDeclaration_new)
    * _instance_
        * [.isRelationshipTarget()](#module_concerto-core.ParticipantDeclaration+isRelationshipTarget) ⇒ <code>boolean</code>
        * [.getSystemType()](#module_concerto-core.ParticipantDeclaration+getSystemType) ⇒ <code>string</code>
    * _static_
        * [.Symbol.hasInstance(object)](#module_concerto-core.ParticipantDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>

<a name="new_module_concerto-core.ParticipantDeclaration_new"></a>

```
#### new ParticipantDeclaration(modelFile, ast)
Create an ParticipantDeclaration.

**Throws**:

- <code>IllegalModelException</code>


| Param | Type | Description |
| --- | --- | --- |
| modelFile | <code>ModelFile</code> | the ModelFile for this class |
| ast | <code>Object</code> | The AST created by the parser |

<a name="module_concerto-core.ParticipantDeclaration+isRelationshipTarget"></a>

#### participantDeclaration.isRelationshipTarget() ⇒ <code>boolean</code>
Returns true if this class can be pointed to by a relationship

**Kind**: instance method of [<code>ParticipantDeclaration</code>]
(#module_concerto-core.ParticipantDeclaration)
**Returns**: <code>boolean</code> - true if the class may be pointed to by a
relationship
<a name="module_concerto-core.ParticipantDeclaration+getSystemType"></a>

#### participantDeclaration.getSystemType() ⇒ <code>string</code>
Returns the base system type for Participants from the system namespace

**Kind**: instance method of [<code>ParticipantDeclaration</code>]
(#module_concerto-core.ParticipantDeclaration)
**Returns**: <code>string</code> - the short name of the base system type
<a name="module_concerto-core.ParticipantDeclaration.Symbol.hasInstance"></a>

#### ParticipantDeclaration.Symbol.hasInstance(object) ⇒ <code>boolean</code>
Alternative instanceof that is reliable across different module instances

**Kind**: static method of [<code>ParticipantDeclaration</code>](#module_concerto-
core.ParticipantDeclaration)
**Returns**: <code>boolean</code> - - True, if the object is an instance of a
ParticipantDeclaration
**See**: https://github.com/hyperledger/composer-concerto/issues/47

| Param | Type | Description |
| --- | --- | --- |
| object | <code>object</code> | The object to test against |

<a name="module_concerto-core.Property"></a>

### concerto-core~Property
Property representing an attribute of a class declaration,
either a Field or a Relationship.

**Kind**: inner class of [<code>concerto-core</code>](#module_concerto-core)

* [~Property](#module_concerto-core.Property)
    * [new Property(parent, ast)](#new_module_concerto-core.Property_new)
    * _instance_
        * [.getParent()](#module_concerto-core.Property+getParent) ⇒
<code>ClassDeclaration</code>
```

```
        * [.getName()](#module_concerto-core.Property+getName) ⇒
<code>string</code>
        * [.getType()](#module_concerto-core.Property+getType) ⇒
<code>string</code>
        * [.isOptional()](#module_concerto-core.Property+isOptional) ⇒
<code>boolean</code>
        * [.getFullyQualifiedTypeName()](#module_concerto-
core.Property+getFullyQualifiedTypeName) ⇒ <code>string</code>
        * [.getFullyQualifiedName()](#module_concerto-
core.Property+getFullyQualifiedName) ⇒ <code>string</code>
        * [.getNamespace()](#module_concerto-core.Property+getNamespace) ⇒
<code>string</code>
        * [.isArray()](#module_concerto-core.Property+isArray) ⇒
<code>boolean</code>
        * [.isTypeEnum()](#module_concerto-core.Property+isTypeEnum) ⇒
<code>boolean</code>
        * [.isPrimitive()](#module_concerto-core.Property+isPrimitive) ⇒
<code>boolean</code>
    * _static_
        * [.Symbol.hasInstance(object)](#module_concerto-
core.Property.Symbol.hasInstance) ⇒ <code>boolean</code>

<a name="new_module_concerto-core.Property_new"></a>

#### new Property(parent, ast)
Create a Property.

**Throws**:

- <code>IllegalModelException</code>


| Param | Type | Description |
| --- | --- | --- |
| parent | <code>ClassDeclaration</code> | the owner of this property |
| ast | <code>Object</code> | The AST created by the parser |

<a name="module_concerto-core.Property+getParent"></a>

#### property.getParent() ⇒ <code>ClassDeclaration</code>
Returns the owner of this property

**Kind**: instance method of [<code>Property</code>](#module_concerto-
core.Property)
**Returns**: <code>ClassDeclaration</code> - the parent class declaration
<a name="module_concerto-core.Property+getName"></a>

#### property.getName() ⇒ <code>string</code>
Returns the name of a property

**Kind**: instance method of [<code>Property</code>](#module_concerto-
core.Property)
**Returns**: <code>string</code> - the name of this field
<a name="module_concerto-core.Property+getType"></a>

#### property.getType() ⇒ <code>string</code>
Returns the type of a property

**Kind**: instance method of [<code>Property</code>](#module_concerto-
core.Property)
**Returns**: <code>string</code> - the type of this field
<a name="module_concerto-core.Property+isOptional"></a>

#### property.isOptional() ⇒ <code>boolean</code>
Returns true if the field is optional

**Kind**: instance method of [<code>Property</code>](#module_concerto-
core.Property)
**Returns**: <code>boolean</code> - true if the field is optional
<a name="module_concerto-core.Property+getFullyQualifiedTypeName"></a>

#### property.getFullyQualifiedTypeName() ⇒ <code>string</code>
Returns the fully qualified type name of a property

**Kind**: instance method of [<code>Property</code>](#module_concerto-
core.Property)
**Returns**: <code>string</code> - the fully qualified type of this property
<a name="module_concerto-core.Property+getFullyQualifiedName"></a>

#### property.getFullyQualifiedName() ⇒ <code>string</code>
Returns the fully name of a property (ns + class name + property name)

**Kind**: instance method of [<code>Property</code>](#module_concerto-
core.Property)
**Returns**: <code>string</code> - the fully qualified name of this property
<a name="module_concerto-core.Property+getNamespace"></a>

#### property.getNamespace() ⇒ <code>string</code>
Returns the namespace of the parent of this property

**Kind**: instance method of [<code>Property</code>](#module_concerto-
core.Property)
**Returns**: <code>string</code> - the namespace of the parent of this property
<a name="module_concerto-core.Property+isArray"></a>

#### property.isArray() ⇒ <code>boolean</code>
Returns true if the field is declared as an array type

**Kind**: instance method of [<code>Property</code>](#module_concerto-
core.Property)
**Returns**: <code>boolean</code> - true if the property is an array type
<a name="module_concerto-core.Property+isTypeEnum"></a>

#### property.isTypeEnum() ⇒ <code>boolean</code>
Returns true if the field is declared as an enumerated value

**Kind**: instance method of [<code>Property</code>](#module_concerto-
core.Property)
**Returns**: <code>boolean</code> - true if the property is an enumerated value
<a name="module_concerto-core.Property+isPrimitive"></a>

#### property.isPrimitive() ⇒ <code>boolean</code>
Returns true if this property is a primitive type.

**Kind**: instance method of [<code>Property</code>](#module_concerto-
core.Property)
**Returns**: <code>boolean</code> - true if the property is a primitive type.
<a name="module_concerto-core.Property.Symbol.hasInstance"></a>
```

```
#### Property.Symbol.hasInstance(object) ⇒ <code>boolean</code>
Alternative instanceof that is reliable across different module instances

**Kind**: static method of [<code>Property</code>](#module_concerto-core.Property)
**Returns**: <code>boolean</code> - - True, if the object is an instance of a
Property
**See**: https://github.com/hyperledger/composer-concerto/issues/47

| Param | Type | Description |
| --- | --- | --- |
| object | <code>object</code> | The object to test against |

<a name="module_concerto-core.RelationshipDeclaration"></a>

### concerto-core~RelationshipDeclaration ⇐ <code>Property</code>
Class representing a relationship between model elements

**Kind**: inner class of [<code>concerto-core</code>](#module_concerto-core)
**Extends**: <code>Property</code>
**See**: See  [Property](Property)

* [~RelationshipDeclaration](#module_concerto-core.RelationshipDeclaration) ⇐
<code>Property</code>
    * [new RelationshipDeclaration(parent, ast)](#new_module_concerto-
core.RelationshipDeclaration_new)
    * _instance_
        * [.toString()](#module_concerto-core.RelationshipDeclaration+toString) ⇒
<code>String</code>
    * _static_
        * [.Symbol.hasInstance(object)](#module_concerto-
core.RelationshipDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>

<a name="new_module_concerto-core.RelationshipDeclaration_new"></a>

#### new RelationshipDeclaration(parent, ast)
Create a Relationship.

**Throws**:

- <code>IllegalModelException</code>


| Param | Type | Description |
| --- | --- | --- |
| parent | <code>ClassDeclaration</code> | The owner of this property |
| ast | <code>Object</code> | The AST created by the parser |

<a name="module_concerto-core.RelationshipDeclaration+toString"></a>

#### relationshipDeclaration.toString() ⇒ <code>String</code>
Returns a string representation of this property

**Kind**: instance method of [<code>RelationshipDeclaration</code>]
(#module_concerto-core.RelationshipDeclaration)
**Returns**: <code>String</code> - the string version of the property.
<a name="module_concerto-core.RelationshipDeclaration.Symbol.hasInstance"></a>

#### RelationshipDeclaration.Symbol.hasInstance(object) ⇒ <code>boolean</code>
```

Alternative instanceof that is reliable across different module instances

Kind: static method of [<code>RelationshipDeclaration</code>](#module_concerto-core.RelationshipDeclaration)
Returns: <code>boolean</code> - - True, if the object is an instance of a RelationshipDeclaration
See: https://github.com/hyperledger/composer-concerto/issues/47

Param	Type	Description
object	<code>object</code>	The object to test against

<a name="module_concerto-core.TransactionDeclaration"></a>

### concerto-core~TransactionDeclaration ⇐ <code>ClassDeclaration</code>
Class representing the definition of an Transaction.

Kind: inner class of [<code>concerto-core</code>](#module_concerto-core)
Extends: <code>ClassDeclaration</code>
See: See ClassDeclaration

* [~TransactionDeclaration](#module_concerto-core.TransactionDeclaration) ⇐ <code>ClassDeclaration</code>
    * [new TransactionDeclaration(modelFile, ast)](#new_module_concerto-core.TransactionDeclaration_new)
    * _instance_
        * [.getSystemType()](#module_concerto-core.TransactionDeclaration+getSystemType) ⇒ <code>string</code>
    * _static_
        * [.Symbol.hasInstance(object)](#module_concerto-core.TransactionDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>

<a name="new_module_concerto-core.TransactionDeclaration_new"></a>

#### new TransactionDeclaration(modelFile, ast)
Create an TransactionDeclaration.

Throws:

- <code>IllegalModelException</code>

Param	Type	Description
modelFile	<code>ModelFile</code>	the ModelFile for this class
ast	<code>Object</code>	The AST created by the parser

<a name="module_concerto-core.TransactionDeclaration+getSystemType"></a>

#### transactionDeclaration.getSystemType() ⇒ <code>string</code>
Returns the base system type for Transactions from the system namespace

Kind: instance method of [<code>TransactionDeclaration</code>](#module_concerto-core.TransactionDeclaration)
Returns: <code>string</code> - the short name of the base system type
<a name="module_concerto-core.TransactionDeclaration.Symbol.hasInstance"></a>

#### TransactionDeclaration.Symbol.hasInstance(object) ⇒ <code>boolean</code>
Alternative instanceof that is reliable across different module instances

Kind: static method of [<code>TransactionDeclaration</code>](#module_concerto-core.TransactionDeclaration)
Returns: <code>boolean</code> - - True, if the object is an instance of a TransactionDeclaration
See: https://github.com/hyperledger/composer-concerto/issues/47

Param	Type	Description
object	<code>object</code>	The object to test against

---
id: version-0.20-concerto-cli
title: Concerto CLI
original_id: concerto-cli

Install the `@accordproject/concerto-cli` npm package to access the Concerto command line interface (CLI). After installation you can use the `concerto` command and its sub-commands as described below.

To install the Concerto CLI:
```
npm install -g @accordproject/concerto-cli@0.82
```

## Usage

```md
concerto <cmd> [args]

Commands:
  concerto validate  validate JSON against model files
  concerto compile   generate code for a target platform
  concerto get       save local copies of external model dependencies

Options:
  --version      Show version number                                   [boolean]
  --verbose, -v                                                 [default: false]
  --help         Show help                                             [boolean]
```

## concerto validate
`concerto validate` lets you check whether a JSON sample is a valid instance of the given model.

```md
## concerto validate

validate JSON against model files

Options:
  --version      Show version number                                   [boolean]
  --verbose, -v                                                 [default: false]
  --help         Show help                                             [boolean]
  --sample       sample JSON to validate       [string] [default: "sample.json"]
  --ctoSystem    system model to be used                                [string]
```

  --ctoFiles   array of CTO files   [array] [default: "."]
```

Example
For example, using the `validate` command to check the sample `request.json` file from a [Late Delivery and Penalty](https://github.com/accordproject/cicero-template-library/tree/master/src/latedeliveryandpenalty) clause:

```
concerto validate --sample request.json --ctoFiles model/clause.cto

returns:

```json
info:
{
  "$class": 
"org.accordproject.latedeliveryandpenalty.LateDeliveryAndPenaltyRequest",
  "forceMajeure": false,
  "agreedDelivery": "2017-12-17T03:24:00.000-05:00",
  "goodsValue": 200,
  "transactionId": "9f241804-118e-439e-bef4-49ee8cf57875",
  "timestamp": "2019-10-29T15:08:46.219Z"
}
```

## concerto compile
`Concerto compile` takes an array of local CTO files, downloads any external dependencies (imports) and then converts all the model to the target format.

```md
## concerto compile

generate code for a target platform

Options:
  --version      Show version number                                   [boolean]
  --verbose, -v                                                 [default: false]
  --help         Show help                                             [boolean]
  --ctoSystem    system model to be used                                [string]
  --ctoFiles     array of CTO files                       [array] [default: "."]
  --target       target of the code generation  [string] [default: "JSONSchema"]
  --output       output directory path           [string] [default: "./output/"]
```

At the moment, the available target formats are as follows:
- Go Lang: `concerto compile --ctoFiles modelfile.cto --target Go`
- Plant UML: `concerto compile --ctoFiles modelfile.cto --target PlantUML`
- Typescript: `concerto compile --ctoFiles modelfile.cto --target Typescript`
- Java: `concerto compile --ctoFiles modelfile.cto --target Java`
- JSONSchema: `concerto compile --ctoFiles modelfile.cto --target JSONSchema`
- XMLSchema: `concerto compile --ctoFiles modelfile.cto --target XMLSchema`

### Example
For example, using the `compile` command to export the `clause.cto` file from a [Late Delivery and Penalty](https://github.com/accordproject/cicero-template-library/tree/master/src/latedeliveryandpenalty) clause into `Go Lang` format:

```md
cd ./model
concerto compile --ctoFiles clause.cto --target Go
```

returns:
```md
info: Compiled to Go in './output/'.
```

## concerto get
`Concerto get` allows you to resolve and download external models from a set of local CTO files.

```md
## concerto get

save local copies of external model dependencies

Options:
  --version      Show version number                                  [boolean]
  --verbose, -v                                                [default: false]
  --help         Show help                                            [boolean]
  --ctoFiles     array of local CTO files                 [array] [default: "."]
  --ctoSystem    system model to be used                               [string]
  --output       output directory path                  [string] [default: "./"]

```

### Example
For example, using the `get` command to get the external models in the `clause.cto` file from a [Late Delivery and Penalty](https://github.com/accordproject/cicero-template-library/tree/master/src/latedeliveryandpenalty) clause:

```md
concerto get --ctoFiles clause.cto
```

returns:
```md
info: Loaded external models in './'.
```

---
id: version-0.20-ergo-api
title: Ergo API
original_id: ergo-api

## Classes

<dl>
<dt><a href="#Commands">Commands</a></dt>
<dd><p>Utility class that implements the commands exposed by the Ergo CLI.</p>
</dd>
</dl>

```
## Functions

<dl>
<dt><a href="#getJson">getJson(input)</a> ⇒ <code>object</code></dt>
<dd><p>Load a file or JSON string</p>
</dd>
<dt><a href="#loadTemplate">loadTemplate(template, files)</a> ⇒
<code>Promise.&lt;LogicManager&gt;</code></dt>
<dd><p>Load a template from directory or files</p>
</dd>
<dt><a href="#fromDirectory">fromDirectory(path, [options])</a> ⇒
<code>Promise.&lt;LogicManager&gt;</code></dt>
<dd><p>Builds a LogicManager from a directory.</p>
</dd>
<dt><a href="#fromZip">fromZip(buffer, [options])</a> ⇒
<code>Promise.&lt;LogicManager&gt;</code></dt>
<dd><p>Builds a LogicManager from a Zip.</p>
</dd>
<dt><a href="#fromFiles">fromFiles(files, [options])</a> ⇒
<code>Promise.&lt;LogicManager&gt;</code></dt>
<dd><p>Builds a LogicManager from files.</p>
</dd>
<dt><a href="#setCurrentTime">setCurrentTime(currentTime)</a> ⇒
<code>object</code></dt>
<dd><p>Ensures there is a proper current time</p>
</dd>
<dt><a href="#init">init(engine, logicManager, contractJson, currentTime)</a> ⇒
<code>object</code></dt>
<dd><p>Invoke Ergo contract initialization</p>
</dd>
<dt><a href="#trigger">trigger(engine, logicManager, contractJson, stateJson,
currentTime, requestJson)</a> ⇒ <code>object</code></dt>
<dd><p>Trigger the Ergo contract with a request</p>
</dd>
<dt><a href="#resolveRootDir">resolveRootDir(parameters)</a> ⇒
<code>string</code></dt>
<dd><p>Resolve the root directory</p>
</dd>
<dt><a href="#compareComponent">compareComponent(expected, actual)</a></dt>
<dd><p>Compare actual and expected result components</p>
</dd>
<dt><a href="#compareSuccess">compareSuccess(expected, actual)</a></dt>
<dd><p>Compare actual result and expected result</p>
</dd>
</dl>

<a name="Commands"></a>

## Commands
Utility class that implements the commands exposed by the Ergo CLI.

**Kind**: global class

* [Commands](#Commands)
    * [.draft(template, files, contractInput, currentTime, options)]
(#Commands.draft) ⇒ <code>object</code>
    * [.trigger(template, files, contractInput, stateInput, currentTime,
requestsInput, warnings)](#Commands.trigger) ⇒ <code>object</code>
    * [.invoke(template, files, clauseName, contractInput, stateInput, currentTime,
paramsInput, warnings)](#Commands.invoke) ⇒ <code>object</code>
    * [.initialize(template, files, contractInput, currentTime, paramsInput,
warnings)](#Commands.initialize) ⇒ <code>object</code>
    * [.parseCTOtoFileSync(ctoPath)](#Commands.parseCTOtoFileSync) ⇒
<code>string</code>
    * [.parseCTOtoFile(ctoPath)](#Commands.parseCTOtoFile) ⇒ <code>string</code>

<a name="Commands.draft"></a>

### Commands.draft(template, files, contractInput, currentTime, options) ⇒
<code>object</code>
Invoke draft for an Ergo contract

**Kind**: static method of [<code>Commands</code>](#Commands)
**Returns**: <code>object</code> - Promise to the result of execution

| Param | Type | Description |
| --- | --- | --- |
| template | <code>string</code> | template directory |
| files | <code>Array.&lt;string&gt;</code> | input files |
| contractInput | <code>string</code> | the contract data |
| currentTime | <code>string</code> | the definition of 'now' |
| options | <code>object</code> | to the text generation |

<a name="Commands.trigger"></a>

### Commands.trigger(template, files, contractInput, stateInput, currentTime,
requestsInput, warnings) ⇒ <code>object</code>
Send a request an Ergo contract

**Kind**: static method of [<code>Commands</code>](#Commands)
**Returns**: <code>object</code> - Promise to the result of execution

| Param | Type | Description |
| --- | --- | --- |
| template | <code>string</code> | template directory |
| files | <code>Array.&lt;string&gt;</code> | input files |
| contractInput | <code>string</code> | the contract data |
| stateInput | <code>string</code> | the contract state |
| currentTime | <code>string</code> | the definition of 'now' |
| requestsInput | <code>Array.&lt;string&gt;</code> | the requests |
| warnings | <code>boolean</code> | whether to print warnings |

<a name="Commands.invoke"></a>

### Commands.invoke(template, files, clauseName, contractInput, stateInput,
currentTime, paramsInput, warnings) ⇒ <code>object</code>
Invoke an Ergo contract's clause

**Kind**: static method of [<code>Commands</code>](#Commands)
**Returns**: <code>object</code> - Promise to the result of invocation

| Param | Type | Description |
| --- | --- | --- |
| template | <code>string</code> | template directory |
| files | <code>Array.&lt;string&gt;</code> | input files |
| clauseName | <code>string</code> | the name of the clause to invoke |
| contractInput | <code>string</code> | the contract data |
| stateInput | <code>string</code> | the contract state |
| currentTime | <code>string</code> | the definition of 'now' |
| paramsInput | <code>object</code> | the parameters for the clause |
| warnings | <code>boolean</code> | whether to print warnings |

<a name="Commands.initialize"></a>

### Commands.initialize(template, files, contractInput, currentTime, paramsInput,
warnings) ⇒ <code>object</code>
Invoke init for an Ergo contract

**Kind**: static method of [<code>Commands</code>](#Commands)
**Returns**: <code>object</code> - Promise to the result of execution

| Param | Type | Description |
| --- | --- | --- |
| template | <code>string</code> | template directory |
| files | <code>Array.&lt;string&gt;</code> | input files |
| contractInput | <code>string</code> | the contract data |
| currentTime | <code>string</code> | the definition of 'now' |
| paramsInput | <code>object</code> | the parameters for the clause |
| warnings | <code>boolean</code> | whether to print warnings |

<a name="Commands.parseCTOtoFileSync"></a>

### Commands.parseCTOtoFileSync(ctoPath) ⇒ <code>string</code>
Parse CTO to JSON File

**Kind**: static method of [<code>Commands</code>](#Commands)
**Returns**: <code>string</code> - The name of the generated CTOJ model file

| Param | Type | Description |
| --- | --- | --- |
| ctoPath | <code>string</code> | path to CTO model file |

<a name="Commands.parseCTOtoFile"></a>

### Commands.parseCTOtoFile(ctoPath) ⇒ <code>string</code>
Parse CTO to JSON File

**Kind**: static method of [<code>Commands</code>](#Commands)
**Returns**: <code>string</code> - The name of the generated CTOJ model file

| Param | Type | Description |
| --- | --- | --- |
| ctoPath | <code>string</code> | path to CTO model file |

<a name="getJson"></a>

## getJson(input) ⇒ <code>object</code>
Load a file or JSON string

**Kind**: global function
**Returns**: <code>object</code> - JSON object

| Param | Type | Description |
| --- | --- | --- |
| input | <code>object</code> | either a file name or a json string |

<a name="loadTemplate"></a>
```

## loadTemplate(template, files) ⇒ <code>Promise.&lt;LogicManager&gt;</code>
Load a template from directory or files

Kind: global function
Returns: <code>Promise.&lt;LogicManager&gt;</code> - a Promise to the instantiated logicmanager

Param	Type	Description
template	<code>string</code>	template directory
files	<code>Array.&lt;string&gt;</code>	input files

<a name="fromDirectory"></a>

## fromDirectory(path, [options]) ⇒ <code>Promise.&lt;LogicManager&gt;</code>
Builds a LogicManager from a directory.

Kind: global function
Returns: <code>Promise.&lt;LogicManager&gt;</code> - a Promise to the instantiated logicmanager

Param	Type	Description
path	<code>String</code>	to a local directory
[options]	<code>Object</code>	an optional set of options to configure the instance.

<a name="fromZip"></a>

## fromZip(buffer, [options]) ⇒ <code>Promise.&lt;LogicManager&gt;</code>
Builds a LogicManager from a Zip.

Kind: global function
Returns: <code>Promise.&lt;LogicManager&gt;</code> - a Promise to the instantiated logicmanager

Param	Type	Description
buffer	<code>Buffer</code>	the buffer to a Zip (zip) file
[options]	<code>Object</code>	an optional set of options to configure the instance.

<a name="fromFiles"></a>

## fromFiles(files, [options]) ⇒ <code>Promise.&lt;LogicManager&gt;</code>
Builds a LogicManager from files.

Kind: global function
Returns: <code>Promise.&lt;LogicManager&gt;</code> - a Promise to the instantiated logicmanager

Param	Type	Description
files	<code>Array.&lt;String&gt;</code>	file names
[options]	<code>Object</code>	an optional set of options to configure the instance.

<a name="setCurrentTime"></a>

## setCurrentTime(currentTime) ⇒ <code>object</code>
Ensures there is a proper current time

Kind: global function
Returns: <code>object</code> - if valid, the moment object for the current time

Param	Type	Description
currentTime	<code>string</code>	the definition of 'now'

<a name="init"></a>

## init(engine, logicManager, contractJson, currentTime) ⇒ <code>object</code>
Invoke Ergo contract initialization

Kind: global function
Returns: <code>object</code> - Promise to the initial state of the contract

Param	Type	Description
engine	<code>object</code>	the execution engine
logicManager	<code>object</code>	the Template Logic
contractJson	<code>object</code>	contract data in JSON
currentTime	<code>string</code>	the definition of 'now'

<a name="trigger"></a>

## trigger(engine, logicManager, contractJson, stateJson, currentTime, requestJson) ⇒ <code>object</code>
Trigger the Ergo contract with a request

Kind: global function
Returns: <code>object</code> - Promise to the response

Param	Type	Description
engine	<code>object</code>	the execution engine
logicManager	<code>object</code>	the Template Logic
contractJson	<code>object</code>	contract data in JSON
stateJson	<code>object</code>	state data in JSON
currentTime	<code>string</code>	the definition of 'now'
requestJson	<code>object</code>	state data in JSON

<a name="resolveRootDir"></a>

## resolveRootDir(parameters) ⇒ <code>string</code>
Resolve the root directory

Kind: global function
Returns: <code>string</code> - root directory used to resolve file names

Param	Type	Description
parameters	<code>string</code>	Cucumber's World parameters

<a name="compareComponent"></a>

## compareComponent(expected, actual)

Compare actual and expected result components

Kind: global function

Param	Type	Description
expected	<code>string</code>	the expected component as specified in the test workload
actual	<code>string</code>	the actual component as returned by the engine

<a name="compareSuccess"></a>

## compareSuccess(expected, actual)
Compare actual result and expected result

Kind: global function

Param	Type	Description
expected	<code>string</code>	the expected successful result as specified in the test workload
actual	<code>string</code>	the successful result as returned by the engine

----
---
id: version-0.20-ergo-cli
title: Ergo CLI
original_id: ergo-cli

Install the `@accordproject/ergo-cli` npm package to access the Ergo command line interface (CLI). After installation you can use the ergo command and its sub-commands as described below.

To install the Ergo CLI:
```
npm install -g @accordproject/ergo-cli@0.20
```

This will install `ergo`, to compile and run contracts locally on your machine, and `ergotop`, which is a _read-eval-print-loop_ utility to write Ergo interactively.

> In ergo-cli `0.20` release, `ergoc`, the Ergo compiler, and `ergorun`, used to run contracts locally on your machine, which were previously part of the `ergo-cli` npm package, have been merged into `ergo` commands.
>
> For more information about the changes that were made for the `0.20` release, please refer to our [Migrating from 0.13.*](0.13to0.20.html) guide.

## Ergo

### Usage

```md
ergo <command>

Commands:
```

~~~~
  ergo draft       create a contract text from data
  ergo trigger     send a request to the contract
  ergo invoke      invoke a clause of the contract
  ergo initialize  initialize the state for a contract
  ergo compile     compile a contract

Options:
  --help         Show help                                             [boolean]
  --version      Show version number                                   [boolean]
  --verbose, -v                                                 [default: false]
```

### ergo draft

`ergo draft` allows you to create a contract text from data. Note that `--data` is
a required field.

```md
## ergo draft
Usage: ergo draft --data [file] [ctos] [ergos]

Options:
  --help           Show help                                           [boolean]
  --version        Show version number                                 [boolean]
  --verbose, -v                                                 [default: false]
  --data           path to the contract data                          [required]
  --currentTime    the current time
                              [string] [default: "2019-10-30T12:03:24+00:00"]
  --wrapVariables  wrap variables in curly braces     [boolean] [default: false]
  --template       path to the template directory       [string] [default: null]
  --warnings       print warnings                     [boolean] [default: false]
```

#### Example

For example, using the `draft` command for the [Volume Discount
example](https://github.com/accordproject/ergo/tree/master/examples/volumediscount)
in the [Ergo Directory](https://github.com/accordproject/ergo):

```md
ergo draft --template ./examples/volumediscount --data
./examples/volumediscount/data.json
```

returns:

```md
info: Volume-Based Card Acceptance Agreement [Abbreviated]

This Agreement is by and between Card, Inc., a New York corporation, and you, the
Merchant.
...
b) Discount. The Discount is determined according to the following table:

| Annual Dollar Volume      | Discount             |
| Less than $1.0 million      | 3.0%                |
| $1.0 million to $10.0 million | 2.9%                |
| Greater than $10.0 million  | 2.8%                |
```
~~~~

The variables specified in the `data.json` file (such as `firstVolume`: 1, `firstRate`: 3) are incorporated into the text of the contract (which can be found in the `./examples/volumediscount` template directory).

> In general, the `data.json` files aren’t part of the template archive from the [Cicero Template Library](https://github.com/accordproject/cicero-template-library/tree/js-release-0.20/src). It is possible to generate one with [cicero parse](cicero-cli#cicero-parse).

## ergo trigger
`ergo trigger` allows you to send a request to the contract.

```md
Usage: ergo trigger --data [file] --state [file] --request [file] [cto files]
[ergo files]

Options:
  --help         Show help                                            [boolean]
  --version      Show version number                                  [boolean]
  --verbose, -v                                                [default: false]
  --data         path to the contract data                           [required]
  --state        path to the state data                 [string] [default: null]
  --currentTime  the current time[string] [default: "2019-10-30T20:18:28+00:00"]
  --request      path to the request data                    [array] [required]
  --template     path to the template directory         [string] [default: null]
  --warnings     print warnings                       [boolean] [default: false]
```

### Example

For example, using the `trigger` command for the [Volume Discount example](https://github.com/accordproject/ergo/tree/master/examples/volumediscount) in the [Ergo Directory](https://github.com/accordproject/ergo):

```md
ergo trigger --template ./examples/volumediscount --data
./examples/volumediscount/data.json --request
./examples/volumediscount/request.json --state ./examples/volumediscount/state.json
```

returns:

```json
{
  "clause": "orgXaccordprojectXvolumediscountXVolumeDiscount",
  "request": {
    "$class": "org.accordproject.volumediscount.VolumeDiscountRequest",
    "netAnnualChargeVolume": 10.4
  },
  "response": {
    "$class": "org.accordproject.volumediscount.VolumeDiscountResponse",
    "discountRate": 2.8,
    "transactionId": "3024ea58-ad82-4c83-87b1-d8fea8436d49",
    "timestamp": "2019-10-31T11:17:36.038Z"
  },
  "state": {
    "$class": "org.accordproject.cicero.contract.AccordContractState",
```

    "stateId": "1"
  },
  "emit": []
}

As the `request` was sent for an annual charge volume of 10.4, which falls into the third discount rate category (as specified in the `data.json` file), the `response` returns with a discount rate of 2.8%.

## ergo invoke
`ergo invoke` allows you to invoke a specific clause of the contract. The main difference between `ergo invoke` and `ergo trigger` is that `ergo invoke` sends data to a specific clause, whereas `ergo trigger` lets the contract choose which clause to invoke. This is why `--clauseName` (the name of the contract you want to execute) is a required field for `ergo invoke`.

You need to pass the CTO and Ergo files (`--template`), the name of the contract that you want to execute (`--clauseName`), and JSON files for: the contract data (`--data`), the contract parameters (`--params`), the current state of the contract (`--state`), and the request.

If contract invocation is successful, `ergorun` will print out the response, the new contract state and any emitted events.

```md
Usage: ergo invoke --data [file] --state [file] --params [file] [cto files]
[ergo files]

Options:
  --help         Show help                                             [boolean]
  --version      Show version number                                   [boolean]
  --verbose, -v                                                 [default: false]
  --clauseName   the name of the clause to invoke                     [required]
  --data         path to the contract data                            [required]
  --state        path to the state data                      [string] [required]
  --currentTime  the current time[string] [default: "2019-10-30T20:18:57+00:00"]
  --params       path to the parameters        [string] [required] [default: {}]
  --template     path to the template directory         [string] [default: null]
  --warnings     print warnings                       [boolean] [default: false]
```

### Example

For example, using the `invoke` command for the [Volume Discount example](https://github.com/accordproject/ergo/tree/master/examples/volumediscount) in the [Ergo Directory](https://github.com/accordproject/ergo):

```md
ergo invoke --template ./examples/volumediscount --clauseName volumediscount --data
./examples/volumediscount/data.json --params ./examples/volumediscount/params.json
--state ./examples/volumediscount/state.json
```

returns:

```json
info:
```

```json
{
  "clause": "orgXaccordprojectXvolumediscountXVolumeDiscount",
  "params": {
    "request": {
      "$class": "org.accordproject.volumediscount.VolumeDiscountRequest",
      "netAnnualChargeVolume": 10.4
    }
  },
  "response": {
    "$class": "org.accordproject.volumediscount.VolumeDiscountResponse",
    "discountRate": 2.8,
    "transactionId": "2a979363-56bc-48ff-a6b4-49994a848a0c",
    "timestamp": "2019-10-31T11:22:37.368Z"
  },
  "state": {
    "$class": "org.accordproject.cicero.contract.AccordContractState",
    "stateId": "1"
  },
  "emit": []
}
```

Although this looks very similar to what `ergo trigger` returns, it is important to note that `--clauseName volumediscount` was specifically invoked.

## ergo initialize
`ergo initialize` allows you to obtain the initial state of the contract. This is the state of the contract without requests or responses.

```md
Usage: ergo intialize --data [file] --params [file] [cto files] [ergo files]

Options:
  --help         Show help                                             [boolean]
  --version      Show version number                                   [boolean]
  --verbose, -v                                                 [default: false]
  --data         path to the contract data                            [required]
  --currentTime  the current time[string] [default: "2019-10-30T20:19:24+00:00"]
  --params       path to the parameters                 [string] [default: null]
  --template     path to the template directory         [string] [default: null]
  --warnings     print warnings                       [boolean] [default: false]

```

### Example

For example, using the `initialize` command for the [Volume Discount example](https://github.com/accordproject/ergo/tree/master/examples/volumediscount) in the [Ergo Directory](https://github.com/accordproject/ergo):

```md
ergo initialize --template ./examples/volumediscount --data ./examples/volumediscount/data.json
```

returns:

```json
info:
```

{
  "clause": "orgXaccordprojectXvolumediscountXVolumeDiscount",
  "params": {
  },
  "response": null,
  "state": {
    "$class": "org.accordproject.cicero.contract.AccordContractState",
    "stateId": "org.accordproject.cicero.contract.AccordContractState#1"
  },
  "emit": []
}

## ergo compile
`ergo compile` takes your input models (`.cto` files) and input contracts (`.ergo` files) and allows you to compile a contract into a target platform. By default, Ergo compiles to JavaScript (ES6 compliant) for execution.

```md
Usage: ergo compile --target [lang] --link --monitor --warnings [cto files]
[ergo files]

Options:
  --help         Show help                                           [boolean]
  --version      Show version number                                 [boolean]
  --verbose, -v                                               [default: false]
  --target       Target platform (available: es5,es6,cicero,java)
                                                  [string] [default: "es6"]
  --link         Link the Ergo runtime with the target code (es5,es6,cicero
                 only)                                 [boolean] [default: false]
  --monitor      Produce compilation time information [boolean] [default: false]
  --warnings     print warnings                        [boolean] [default: false]
```

### Example
For example, using the `compile` command on the [Volume Discount example](https://github.com/accordproject/ergo/tree/master/examples/volumediscount) in the [Ergo Directory](https://github.com/accordproject/ergo):

```md
ergo compile ./examples/volumediscount/model/model.cto
./examples/volumediscount/logic/logic.ergo
```

returns:

```md
Compiling Ergo './examples/volumediscount/logic/logic.ergo' --
'./examples/volumediscount/logic/logic.js'
```

Which means a new `logic.js` file is located in the `./examples/volumediscount/logic` directory.

To compile the contract to Javascript and **link the Ergo runtime for execution**:
```md
ergo compile ./examples/volumediscount/model/model.cto
./examples/volumediscount/logic/logic.ergo --link
```

~~~
```

returns:

```md
Compiling Ergo './examples/volumediscount/logic/logic.ergo' --
'./examples/volumediscount/logic/logic.js'
```


--------------------------------------------------------------------------------
---
id: version-0.20-ergo-repl
title: Ergo REPL
original_id: ergo-repl
---

`ergotop` is a convenient tool to try-out Ergo contracts in an interactive way. You
can write commands, or expressions and see the result. It is often called the Ergo
REPL, for _read-eval-print-loop_, since it literally: reads your input Ergo from
the command-line, evaluates it, prints the result and loops back to read your next
input.

## Starting the REPL

To start the REPL:

```
$ ergotop
Welcome to ERGOTOP version 0.20.0
ergo$
```

It should print the prompt `ergo$` which indicates it is ready to read your
command. For instance:

```ergo
    ergo$ return 42
    Response. 42 : Integer
```

`ergotop` prints back both the resulting value and its type. You can then keep
typing commands:

```ergo
    ergo$ return "hello " ++ "world!"
    Response. "hello world!" : String
    ergo$ define constant pi = 3.14
    ergo$ return pi ^ 2.0
    Response. 9.8596 : Double
```

If your expression is not valid, or not well-typed, it will return an error:

```ergo
    ergo$ return if true else "hello"
    Parse error (at line 1 col 15).
    return if true else "hello"
                   ^^^^
~~~

~~~~
    ergo$ return if "hello" then 1 else 2
    Type error (at line 1 col 10). 'if' condition not boolean.
    return if "hello" then 1 else 2
              ^^^^^^^
```

If what you are trying to write is too long to fit on one line, you can use `\` to
go to a new line:

```ergo
    ergo$ define function squares(l:Double[]) : Double[] { \
      ...   return \
      ...     foreach x in l return x * x \
      ... }
    ergo$ return squares([2.4,4.5,6.7])
    Response. [5.76, 20.25, 44.89] : Double[]
```

## Loading files

You can load CTO and Ergo files to use in your REPL session. Once the REPL is
launched you will have to import the corresponding namespace. For instance, if you
want to use the `compoundInterestMultiple` function defined in the
`./examples/promissory-note/money.ergo` file, you can do it as follows:

```ergo
$ ergotop ./examples/promissory-note/logic/money.ergo
Welcome to ERGOTOP version 0.20.0

ergo$ import org.accordproject.ergo.money.*
ergo$ return compoundInterestMultiple(0.035, 100)
Response. 1.00946960405 : Double
```

## Calling contracts

To call a contract, you first needs to _instantiate_ it, which means setting its
parameters and initializing its state. You can do this by using the `set contract`
and `call init` commands respectively. Here is an example using the
`volumediscount` template:

```ergo
$ ergotop ./examples/volumediscount/model/model.cto
./examples/volumediscount/logic/logic.ergo
ergo$ import org.accordproject.cicero.contract.*
ergo$ import org.accordproject.volumediscount.*
ergo$ set contract VolumeDiscount over VolumeDiscountContract {firstVolume: 1.0,
secondVolume: 10.0, firstRate: 3.0, secondRate: 2.9, thirdRate: 2.8, contractId:
"0", parties: none }
ergo$ call init()
Response. unit : Unit
State. AccordContractState{stateId:
"org.accordproject.cicero.contract.AccordContractState#1"} : AccordContractState
```

You can then invoke clauses of the contract:

```ergo
ergo$ call volumediscount(VolumeDiscountRequest{ netAnnualChargeVolume : 0.1 })
~~~~

Response. VolumeDiscountResponse{discountRate: 3.0} : VolumeDiscountResponse
ergo$ call volumediscount(VolumeDiscountRequest{ netAnnualChargeVolume : 10.5 })
Response. VolumeDiscountResponse{discountRate: 2.8} : VolumeDiscountResponse

You can also invoke the contract without explicitly naming the clause by sending a request. The Ergo engine dispatches that request to the first clause which can handle it:
```ergo
ergo$ send VolumeDiscountRequest{ netAnnualChargeVolume : 0.1 }
Response. VolumeDiscountResponse{discountRate: 3.0} : VolumeDiscountResponse
ergo$ send VolumeDiscountRequest{ netAnnualChargeVolume : 10.5 }
Response. VolumeDiscountResponse{discountRate: 2.8} : VolumeDiscountResponse
```

---
id: version-0.20-ergo-tutorial
title: Ergo: A Tutorial
original_id: ergo-tutorial

## Overview of Accord

Cicero is an Open Source implementation of the Accord Project Template Specification. It defines the structure of natural language templates, bound to a data model, that can be executed using Ergo and request/response JSON messages. You can read the latest user documentation here: http://docs.accordproject.org.

In short, with the Accord Project you can take a classic contract, e.g. Word document and use Cicero to define natural language contract and clause templates that can be executed by an event driven computer program (aka Smart contract). For the tutorial, Cicero will be used to define natural language contract and clause templates. These clause templates handle the syllogistic language of contracts.

For example,
```md
 if the goods are more than [{DAYS}] late,
 then notify the supplier of the goods, with the message [{MESSAGE}].
```
DAYS and MESSAGE are variables

You can browse the library of Open Source Cicero contract and clause templates at: https://templates.accordproject.org.

So how goes the contract get executed? That is where Ergo comes in Ergo is a strongly-typed functional programming language designed to capture the legal intent of legal contracts and clauses. We will use Ergo to create the contract logic consisting of a contract class with executable embedded clauses. Note: prior to the emergence of Ergo, the Cicero JavaScript component was primary to the execution of code.

Ergo obviates the Cicero JavaScript component for the execution phase with a new more comprehensive language which we explore in this tutorial.

## Cicero

The Open Source Cicero project defines the format of clause and contract templates based on to the Cicero Template Specification. The templates are the link between the natural language of contracts usually composed in a Word document and the specification of a machine executable transaction. Cicero templates define the API by specifying request and response elements for the logic associated with functional transaction executed by Ergo.

Cicero templates are composed of two elements:
* Template Grammar (the natural language text for the template),
* Template Model (the data model that includes the variables contained within the template).
* The Logic (the executable business logic for the template) will be handled by Ergo.

When combined these three elements allow templates to be edited, analyzed, queried and executed.

## Setup Ergo Development environment

Before you can build Ergo, you must install and configure the following dependencies on your machine:

### Git

* Git: The [Github Guide to Installing Git][git-setup] is a good source of information.

### Node.js

* Node.js v8.x (LTS): We use Node to generate the documentation, run a development web server, run tests, and generate distributable files. Depending on your system, you can install Node either from source or as a pre-packaged bundle.
> Tip: Use nvm (or nvm-windows) to manage and install Node.js, This facilitates a version change of Node.js per project.
* Lerna: This is a tool which helps when handling multiple npm packages in the Ergo repository. To install:
npm install -g lerna@^3.15.0

### Visual Studio Code

Visual Studio Code is a lightweight but powerful source code editor which runs on your desktop and is available for Windows, macOS and Linux. It comes with built-in support for JavaScript and Node.js and has a rich ecosystem of extensions for other languages (such Ergo).

Follow the platform specific guides below:
See, https://code.visualstudio.com/docs/setup/
* macOS
* Linux
* Windows

#### Install Ergo VisualStudio Plugin

### Validate Development Environment and Toolset

Clone https://github.com/accordproject/ergo to your local machine

### Getting started

Install Ergo

The easiest way to install Ergo is as a Node.js package. Once you have Node.js installed on your machine, you can get the Ergo compiler and command-line using the Node.js package manager by typing the following in a terminal:
$ npm install -g @accordproject/ergo-cli@0.20

This will install the compiler itself (ergoc) and a command-line tool (ergo) to execute Ergo code. You can check that both have been installed and print the version number by typing the following in a terminal:
```sh
$ ergoc --version
$ ergo --version
```
Then, to get command line help:
```
$ ergoc --help
$ ergo execute --help
```
Compiling your first contract
```ergo
namespace org.accordproject.volumediscount

contract VolumeDiscount over VolumeDiscountContract {
   // Clause for volume discount
   clause volumediscount(request : VolumeDiscountRequest) : VolumeDiscountResponse
{
        if request.netAnnualChargeVolume < contract.firstVolume
        then return VolumeDiscountResponse{ discountRate: contract.firstRate }
        else if request.netAnnualChargeVolume < contract.secondVolume
        then return VolumeDiscountResponse{ discountRate: contract.secondRate }
        else return VolumeDiscountResponse{ discountRate: contract.thirdRate }
  }
}
```

To compile your first Ergo contract to JavaScript , within Visual Studio code
* Open the folder where you cloned https://github.com/accordproject/ergo
* Use View/Terminal to run the Ergo compiler:
```sh
$ ergoc ./examples/volumediscount/model.cto ./examples/volumediscount/logic.ergo
Compiling Ergo './examples/volumediscount/logic.ergo' -- creating
'./examples/volumediscount/logic.js'
```

By default, Ergo compiles to JavaScript for execution. This may change in the future to support other languages. The compiled code for the result in stored as `./examples/volumediscount/logic.js`

### Execute a contract
To execute a contract, we pass the necessary parameters including the CTO, Ergo files, the name of a contract and the json files containing request and contract state
ergorun [ctos] [ergos] --contractname [file] --contract [file] --state [file] --request [file]

So for example we use ergorun with :
```sh
$ ergorun ./examples/volumediscount/model.cto ./examples/volumediscount/logic.ergo
--contractname org.accordproject.volumediscount.VolumeDiscount
--contract ./examples/volumediscount/contract.json
```

~~~~
--request ./examples/volumediscount/request.json
--state ./examples/volumediscount/state.json
```

Here contract.json contains the following values
```json
{
  "$class": "org.accordproject.volumediscount.VolumeDiscountContract",
  "parties": null,
  "contractId": "cr1",
  "firstVolume": 1,
  "secondVolume": 10,
  "firstRate": 3,
  "secondRate": 2.9,
  "thirdRate": 2.8
}
```

Request.json contains
```json
{
  "$class": "org.accordproject.volumediscount.VolumeDiscountRequest",
  "netAnnualChargeVolume": 10.4
}
```

logic.ergo contains:
```ergo
namespace org.accordproject.volumediscount

contract VolumeDiscount over VolumeDiscountContract {
  // Clause for volume discount
  clause volumediscount(request : VolumeDiscountRequest) : VolumeDiscountResponse {
    if request.netAnnualChargeVolume < contract.firstVolume
    then return VolumeDiscountResponse{ discountRate: contract.firstRate }
    else if request.netAnnualChargeVolume < contract.secondVolume
    then return VolumeDiscountResponse{ discountRate: contract.secondRate }
    else return VolumeDiscountResponse{ discountRate: contract.thirdRate }
  }
}
```

Here netAnnualCharge Volume equals 10.4 which is not less than firstVolume and
secondVolume which are equal to 1 and 10 respectively so the logic for the
volumediscount clause returns thirdRate which equals 2.8

```
7:31:58 PM - info: Logging initialized. 2018-09-27T23:31:58.623Z
7:31:59 PM - info: {"response":
{"discountRate":2.8,"$class":"org.accordproject.volumediscount.VolumeDiscountRespon
se"},"state":
{"$class":"org.accordproject.cicero.contract.AccordContractState","stateId":"1"},"e
mit":[]}
```

PS D:\Users\jbambara\Github\ergo>

## Ergo Development
~~~~

Create Template
Start with basic agreement in natural language and locate the variables
Here in the example see the bold
Volume-Based Card Acceptance Agreement [Abbreviated]
This Agreement is by and between ………..you agree to be bound by the Agreement.
Discount means an amount that we charge you for accepting the Card, which amount is:
(i) a percentage (Discount Rate) of the face amount of the Charge that you submit, or a flat per-
Transaction fee, or a combination of both; and/or
(ii) a Monthly Flat Fee (if you meet our requirements).

Transaction Processing and Payments. ………………… less all applicable deductions, rejections, and withholdings, which include:
………………………….

SETTLEMENT
a) Settlement Amount. Our agent will pay you according to your payment plan, ……………………..which include:
(i) the Discount,
…………………………………………..
b) Discount. The Discount is determined according to the following table:

Annual Dollar Volume	Discount
Less than $1 million	3.00%
$1 million to $10 million	2.90%
Greater than $10 million	2.80%

Identify the request variables and contract instance variables
Codify the variables with $[{request}] or [{contract instance}]

Annual Dollar Volume	Discount
Less than $[{firstVolume}] million	[{firstRate}]%
$[{firstVolume}] million to $[{secondVolume}] million	[{secondRate}]%
Greater than $[{secondVolume}] million	[{thirdRate}]%

Create Model
Define the model asset which contains the contract instance variables and the transaction request and response. Defines the data model for the VolumeDiscount template. This defines the structure that the parser for the template generates from input source text. See model.cto below:

```
 namespace org.accordproject.volumediscount
import org.accordproject.cicero.contract.* from
https://models.accordproject.org/cicero/contract.cto
import org.accordproject.cicero.runtime.* from
https://models.accordproject.org/cicero/runtime.cto
asset VolumeDiscountContract extends AccordContract {
  o Double firstVolume
  o Double secondVolume
  o Double firstRate
  o Double secondRate
  o Double thirdRate
}
transaction VolumeDiscountRequest {
  o Double netAnnualChargeVolume
}
transaction VolumeDiscountResponse {
        o Double discountRate
}
```

Create Logic
The contract logic is accomplished by coding ERGO statements and expressions to consume the request and use contract instance variables to produce the desired response. In our example, request.netAnnualChargeVolume is tested against contract rates to produce the result.

```
namespace org.accordproject.volumediscount

define the contract
contract VolumeDiscount over VolumeDiscountContract {

define the contract clause and request : response

   clause volumediscount(request : VolumeDiscountRequest) : VolumeDiscountResponse
{

define the logic ; here we use if /then /else statement to test request parameter
against contract instance variable
 and return

        if request.netAnnualChargeVolume < contract.firstVolume
        then return VolumeDiscountResponse{ discountRate: contract.firstRate }
        else if request.netAnnualChargeVolume < contract.secondVolume
        then return VolumeDiscountResponse{ discountRate: contract.secondRate }
        else return VolumeDiscountResponse{ discountRate: contract.thirdRate }
  }
```

Ergo Language
As you have seen in this tutorial, Ergo is a domain-specific language (DSL) that captures the execution logic of legal contracts. In this simple example, you see that Ergo aims to have contracts and clauses as first-class elements of the language. To accommodate the maturation of distributed ledger implementations, Ergo will be blockchain neutral, i.e., the same contract logic can be executed either on and off chain on distributed ledger technologies like HyperLedger Fabric. Most importantly, Ergo is consistent with the Accord Protocol Template Specification.
Follow the links below to learn more about
Introduction to Ergo
Ergo Language Guide
Ergo Reference Guide

October 12, 2018

---
id: version-0.20-example-eatapple
title: A Healthy Clause
original_id: example-eatapple

## Eat Apples!

The healthy eating clause is inspired by the not so serious [terms of services contract](https://www.grahamcluley.com/page-46-apples-new-ios-agreement-funny-fake-makes-serious-point/).

For this example, let us look first at the template for that legal clause written in natural language:

```markdown
```

```
Eating healthy clause between [{employee}] (the Employee) and [{company}] (the
Company).
The canteen only sells apple products. Apples, apple juice, apple flapjacks, toffee
apples. Employee gets fired if caught eating anything without apples in it.
THE EMPLOYEE, IF ALLERGIC TO APPLES, SHALL ALWAYS BE HUNGRY.
Apple products at the canteen are subject to a [{tax}]% tax.
```

The text specifies the terms for the legal clause and includes a few
variables such as `employee`, `company` and `tax`.

The second component of a smart legal template is the model, which is
expressed using the [Concerto modeling
language](https://github.com/accordproject/concerto).
The model describes the variables of the contract, as well as additional
information required to execute the contract logic. In our example,
this includes the input request for the clause (`Food`), the response
to that request (`Outcome`) and possible events emitted during the
clause execution (`Bill`).

```ergo
namespace org.accordproject.canteen

@AccordTemplateModel("eat-apples")
concept CanteenContract {
  o String employee
  o String company
  o Double tax
}

transaction Food {
  o String produce
  o Double price
}

transaction Outcome {
  o String notice
}

event Bill {
  o String billTo
  o Double amount
}
```

The last component of a smart legal template is the Ergo logic. In our example, it
is a single clause `eathealthy` which can be used to process a `Food` request.

```ergo
namespace org.accordproject.canteen

contract EatApples over CanteenContract {
  clause eathealthy(request : Food) : Outcome {
    enforce request.produce = "apple"
    else return Outcome{ notice : "You're fired!" };

    emit Bill{
      billTo: contract.employee,
      amount: request.price * (1.0 + contract.tax / 100.0)
```

    };
    return Outcome{ notice : "Very healthy!" }
  }
}

As in the "Hello World!" example, this is a smart legal contract with
a single clause, but it illustrate a few new ideas.

The `enforce` expression is used to check conditions that must be true
for normal execution of the clause. In this example, the `enforce`
makes sure the food is an apple and if not returns a new outcome
indicating termination of employment.

If the condition is true, the contract proceeds by emitting a bill for
the purchase of the apple. The employee to be billed is obtained from
the contract (`contract.employee`). The total amount is calculated by
adding the tax, which is obtained from the contract (`contract.tax`),
to the purchase price, which is obtained from the request
(`request.price`). The calculation is done using a simple arithmetic
expression (`request.price * (1.0 + contract.tax / 100.0)`).

---
id: version-0.20-logic-advanced-expr
title: Advanced Expressions
original_id: logic-advanced-expr

## Match

### Match against Values

Match expressions allow to check an expression against multiple possible
values:

```ergo
    match fruitcode
      with 1 then "Apple"
      with 2 then "Apricot"
      else "Strange Fruit"
```

Match expressions can also be used to match against enumerated values:
```ergo
    match state
      with NY then "Empire State"
      with NJ then "Garden State"
      else "Far from home state"
```

### Match against Types

Match expressions can be used to match a value against a class type:.

```
define constant products = [
    Product{ id : "Blender" },
```

~~~~
    Car{ id : "Batmobile", range: "Infinite" },
    Product{ id : "Cup" }
  ]

foreach p in products
return
  match p
    with let x : Car then "Car (" ++ x.id ++ ") with range " ++ x.range
    with let x : Product then "Product (" ++ x.id ++ ")"
    else "Not a product"
```
Should return the array `["Product (Blender)", "Car (Batmobile) with range
Infinite", "Product (Cup)"]`

## Foreach

Foreach expressions allow to apply an expression of every element in
an input array of values and returns a new array:

```ergo
  foreach x in [1.0,-2.0,3.0] return x + 1.0
```

Foreach expressions can have an optional condition of the values being
iterated over:

```ergo
  foreach x in [1.0,-2.0,3.0] where x > 0.0 return x + 1.0
```

Foreach expressions can iterate over multiple arrays. For example, the following
foreach expression returns all all [Pythagorean
triples](https://en.wikipedia.org/wiki/Pythagorean_triple):
```ergo
let nums = [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0];
foreach x in nums
foreach y in nums
foreach z in nums
where (x^2.0 + y^2.0 = z^2.0)
return {a: x, b: y, c: z}
```
and should return the array `[{a: 3.0, b: 4.0, c: 5.0}, {a: 4.0, b: 3.0, c: 5.0},
{a: 6.0, b: 8.0, c: 10.0}, {a: 8.0, b: 6.0, c: 10.0}]`.

## Template Literals

Template literals are similar to [String literals](Literal values) but with the
ability to embed Ergo expressions. They are written with between `` ` `` and may
contains Ergo expressions inside `{{%` and `%}}`.

The following Ergo expressions illustrates the use of a template literal to
construct a String describing the content of a record.
```
let law101 = {
    name: "Law for developers",
    fee: 29.99
  };
`Course "{{% law101.name %}}" (Cost: {{% law101.fee %}})`
```
~~~~

Should return the string literal `"Course \"Law for developers\" (Cost: 29.99)"`.

---
id: version-0.20-logic-complex-type
title: Complex Values & Types
original_id: logic-complex-type

So far we only considered atomic values and types, such as string values or integers, which are not sufficient for most contracts. In Ergo, values and types are based on the [Concerto Modeling](model-concerto) (often referred to as CTO models after the `.cto` file extension). This provides a rich vocabulary to define the parameters of your contract, the information associated to contract participants, the structure of contract obligation, etc.

In Ergo, you can either import an existing CTO model or declare types directly within your code. Let us look at the different kinds of types you can define and how to create values with those types.

## Arrays

Array types lets you define collections of values and are denoted with `[]` after the type of elements in that collection:

```ergo
    String[]                         // a String array
    Double[]                         // a Double array
```

You can write arrays as follows:
```ergo
    ["pear","apple","strawberries"]  // an array of String values
    [3.14,2.72,1.62]                 // an array of Double values
```

You can construct arrays using other expressions:
```ergo
    let pi = 3.14;
    let e = 2.72;
    let golden = 1.62;
    [pi,e,golden]
```

Ergo also provides functions to manipulate arrays as parts of its [standard library](ref-logic-stdlib.html#functions-on-arrays). The following example uses the `sum` function to calculate the sum of all the elements in the `prettynumbers` array.
```ergo
    let pi = 3.14;
    let e = 2.72;
    let golden = 1.62;
    let prettynumbers : Double[] = [pi,e,golden];
    sum(prettynumbers)
```

You can access the element at a given position inside the array using an index:
```ergo
```

    let fruits = ["pear","apple","strawberries"];
    fruits[0]   // Returns: some("pear")
     let fruits = ["pear","apple","strawberries"];
    fruits[2]   // Returns: some("strawberries")
     let fruits = ["pear","apple","strawberries"];
    fruits[4]   // Returns: none

 Note that the index starts at `0` for the first element and that indexed-based access returns an optional value, since Ergo compiler cannot statically determine whether there will be an element at the corresponding index. You can learn more about how to handle optional values and types in the [Optionals](logic-complex-type#optionals) Section below.

## Classes

You can declare classes in the Concerto Modeling Language (concepts, transactions, events, participants or assets) by importing them from a CTO file or directly within your Ergo program:

```ergo
   define concept Seminar {
     name : String,
     fee : Double
   }
   define asset Product {
     id : String
   }
   define asset Car extends Product {
     range : String
   }
   define transaction Response {
     rate : Double,
     penalty : Double
   }
  define event PaymentObligation{
    amount : Double,
    description : String
  }
```

Once a class type has been defined, you can create an instance of that type using the class name along with the values for each fields:

```ergo
    Seminar{
      name: "Law for developers",
      fee: 29.99
    }
    Car{
      id: "Batmobile4156",
      range: "Infinite"
    }
```

> **TechNote:** When extending an existing class (e.g., `Car extends Product`), the sub-class includes the fields from the super-class. So `Car` includes the field `range` which is locally declared and the field `id` which is declared in `Product`.

You can access fields for values of a class type by using the `.` operator:
```ergo
    Seminar{
      name: "Law for developers",
      fee: 29.99
    }.fee                        // Returns 29.99
```

## Records

Sometimes it is convenient to declare a structure without having to declare it first. You can do that using a record, which is similar to a class but without a name attached to it:

```ergo
   {
     name : String,  // A record with a name of type String
     fee : Double    // and a fee of type Double
   }
```

You do not need to declare that record, and can directly write an instance of that record as follows:

```ergo
    {
      name: "Law for developers",
      fee: 29.99
    }
```

> Typing `return { name: "Law for developers", fee: 29.99 }` in the [Ergo REPL](https://ergorepl.netlify.com), should answer `Response. {name: "Law for developers", fee: 29.99} : {fee: Double, name: String}`.

You can access the field of a record using the `.` operator:
```ergo
    {
      name: "Law for developers",
      fee: 29.99
    }.fee                        // Returns 29.99
```
## Enums

Here is how to declare an enumerated type:

```ergo
define enum ProductType {
   DAIRY,
   BEEF,
   VEGETABLES
}
```

To create an instance of that enum:
```ergo
DAIRY
BEEF
```

~~~
```

## Optionals

An optional type can contain a value or not and is indicated with a `?`.

```ergo
Integer?           // An optional integer
PaymentObligation? // An optional payment obligation
Double[]?          // An optional array of doubles
```

A an optional value can be either present, written `some(v)`, or absent, written
`none`.

```ergo
let i1 : Integer? = some(1); i1
let i2 : Integer? = none; i2
```

To operate on an optional type, you need to say what to do when the value is
present and what to do when the value is not present. The most general way to do
that is with a match expression:

This example matches a value which is present:
```ergo
match some(1)
with let? x then "I found " ++ toString(x) ++ " :-)"
else "I found nothing :-("
```
and should return `"I found 1 :-)"`.

While this example matches against a value which is absent:
```
match none
with let? x then "I found " ++ toString(x) ++ " :-)"
else "I found nothing :-("
```
and should return `"I found nothing :-("`.

More details on match expressions can be found in [Advanced Expressions](logic-
advanced-expr#match).

For conciseness, a few operators are also available on optional values. One can
give a default value when the optional is `none` using the operator `??`. For
instance:

```ergo
some(1) ?? 0       // Returns the integer 1
none ?? 0          // Returns the integer 0
```

You can also access the field inside an optional concept or an optional record
using the operator `?.`. For instance:

```ergo
some({a:1})?.a     // Returns the optional value: some(1)
none?.a            // Returns the optional value: none
```
~~~

---
id: version-0.20-logic-decl
title: Declarations
original_id: logic-decl

Now that we have values, types, expressions and statements available, we can start writing more complex Ergo logic using by declaring functions, clauses and contracts.

## Constants and functions

It is possible to declare global constants and functions in Ergo:

```ergo
define constant pi = 3.1416
define function area(radius : Double) : Double {
   return pi * radius * radius
}
area(1.5)
```

Global variables can also be declared with a type:

```ergo
define constant pi : Double = 3.1416
```

The return type of functions can be omitted:

```ergo
define function area(radius : Double) {
   return pi * radius * radius
}
area(1.5)
```

## Clauses

In Ergo, a logical clause like the example clause noted below is represented as a “function” (akin to a “method” in languages like Java) that resides within its parent contract (akin to a “class” in a language like Java).

> Functions are "self contained" modules of code that accomplish a specific task. Functions usually "take in" data, process it, and "return" a result. Once a function is written, it is reusable , i.e., it can be used over and over and over again.
> Functions can be "called" from within other functions or from a clause.
> Functions have to be declared before they can be used. So functions "encapsulate" a task. They combine statements and expressions carried out as instructions which accomplish a specific task to allow their execution using a single line of code. Most programming languages provide libraries of built in functions (i.e., commonly used tasks like computing the square root of a number).
> Functions accelerate development and facilitate the reuse of code which performs common tasks.

The declaration of a Clause that contains the clause’s name, request type and return type collectively referred to as the ‘signature’ of the function.

### Example Prose

Additionally the Equipment should have proper devices on it to record any shock during transportation as any instance of acceleration outside the bounds of -0.5g and 0.5g. Each shock shall reduce the Contract Price by $5.00

### Syntax

```ergo
    clause fragileGoods(request : DeliveryUpdate) : ContractPrice {
        ... // A statement computing the clause response
    }
```

Inside a contract, the `contract` variable contains the instance of the template model for the current contract.

## Contract Declarations

The legal requirements for a valid contract at law vary by jurisdiction and contract type. The requisite elements that typically necessary for the formation of a legally binding contract are (1) _offer_; (2) _acceptance_; (3) _consideration_; (4) _mutuality of obligation_; (5) _competency and capacity_; and, in certain circumstances, (6) _a written instrument_.

Ergo contacts address consideration, mutuality of obligation, competency and capacity through statements that are described in this document.

Furthermore, an Ergo contract is an immutable written document which obviates a good deal of the issues and conflicts which emerge from existing contracts in use today. In Ergo, a contract:
- represents an agreement between parties creating mutual and enforceable obligations; and
- is a code module that uses conditionals and functions to describe execution by the parties with their obligations. Contracts accept input data either directly from the associated natural language text or through request _transactions_. The contract then uses _clause functions_ to process it, and _return_ a result.
Once a contract logic has been written within a template, it can be used over and over and over again.

Instantiated contracts correspond to particular domain agreement. They combine functions and clauses to execute a specific agreement and to allow its automation. Many traditional contracts are “boilerplate” and as such are reusable in their specific legal domain, e.g., sale of goods.

You can declare a contract over a template model as follows. The `TemplateModel` is the data model for the parameters of the contract text.

```ergo
    contract ContractName over TemplateModel {
      clause C1(request : ReqType1) : RespType1 {
        // Statement
      }

      clause C2(request : ReqType2) : RespType2 {
        // Statement
      }
    }
```

## Contract State and Obligations

If your contract requires a state, or emits only certain kinds of obligations but not other, you can specify the corresponding types when declaring your contract:

```ergo
    contract ContractName over TemplateModel state MyState {
      clause C1(request : ReqType1) : RespType1 emits MyObligation {
        // Statement
      }

      clause C2(request : ReqType2) : RespType2 {
        // Statement
      }
    }
```

The state is always declared for the whole contract, while obligations can be declared individually for each clause.

---
id: version-0.20-logic-ergo
title: Ergo Overview
original_id: logic-ergo

## Language Goals

Ergo aims to:
- have contracts and clauses as first-class elements of the language
- help legal-tech developers quickly and safely write computable legal contracts
- be modular, facilitating reuse of existing contract or clause logic
- ensure safe execution: the language should prevent run-time errors and non-terminating logic
- be blockchain neutral: the same contract logic can be executed either on and off chain on a variety of distributed ledger technologies
- be formally specified: the meaning of contracts should be well defined so it can be verified, and preserved during execution
- be consistent with the [Accord Project Template Specification](accordproject-specification)

## Design Choices

To achieve those goals the design of Ergo is based on the following principles:

- Ergo contracts have a class-like structure with clauses akin to methods
- Ergo can handle types (concepts, transactions, etc) defined with the [Concerto Modeling Language](https://github.com/accordproject/concerto) (so called CML models), as mandated by the Accord Project Template Specification
- Ergo borrows from strongly-typed functional programming languages: clauses have a well-defined type signature (input and output), they are functions without side effects

- The compiler guarantees error-free execution for well-typed Ergo programs
- Clauses and functions are written in an expression language with limited expressiveness (it allows conditional and bounded iteration)
- Most of the compiler is written in Coq as a stepping stone for formal specification and verification

## Status

- The current implementation is considered *in development*, we welcome contributions (be it bug reports, suggestions for new features or improvements, or pull requests)
- The current compiler targets JavaScript (either standalone or for use in Cicero Templates and Hyperledger Fabric) and Java (experimental)

## This Guide

Ergo provides a simple expression language to describe computation. From those expressions, one can write functions, clauses, and then whole contract logic. This guide explains most of the Ergo concepts starting from simple expressions all the way to contracts.

Ergo is a _strongly typed_ language, which means it checks that the expressions you use are consistent (e.g., you can take the square root of `3.14` but not of `"pi!"`). The type system is here to help you write better and safer contract logic, but it also takes a little getting used to. This page also introduces Ergo types and how to work with them.

---
id: version-0.20-logic-module
title: Modules
original_id: logic-module

Finally, we can place multiple Ergo declarations (functions, contracts, etc) into a library so it can be shared with other developers.

## Namespaces

Each Ergo file starts with a namespace declaration which provides a way to identify it uniquely:
```ergo
namespace org.acme.mynamespace
```

## Libraries

A library is an Ergo file in a namespace which defines useful constants or functions. For instance:

```ergo
namespace org.accordproject.ergo.money

define constant days_in_a_year = 365.0
define function compoundInterests(
  annualInterest : Double,
  numberOfDays : Double
) : Double {
```

    return (1.0 + annualInterest) ^ (numberOfDays / days_in_a_year)
}

## Import

You can then access this library in another Ergo file using import:
```ergo
namespace org.accordproject.promissorynote

contract PromissoryNote over PromissoryNoteContract {
  clause check(request : Payment) : Result {
        let interestRate = contract.interestRate ?? 3.4;
    enforce interestRate >= 0.0;
    enforce contract.amount.doubleValue >= 0.0;
    let outstanding = contract.amount.doubleValue - request.amountPaid.doubleValue;
    enforce outstanding >= 0.0;

    let numberOfDays =
      diffDurationAs(
        dateTime("17 May 2018 13:53:33 EST"),
        contract.date,
        ~org.accordproject.time.TemporalUnit.days).amount;

    enforce numberOfDays >= 0;

    let compounded =  outstanding
      * compoundInterestMultiple(interestRate, numberOfDays); // Defined in
ergo.money module

    return Result{
      outstandingBalance: compounded
    }
  }
}
```

> **TechNote:** the namespace and import handling in Ergo allows you to access either existing CTO models or Ergo libraries in the same way.

---
id: version-0.20-logic-simple-expr
title: Simple Expressions
original_id: logic-simple-expr

## Literal values

The simplest kind of expressions in Ergo are literal values.

```ergo
    "John Smith" // a String literal
    1            // an Integer literal
    3.0          // a Double literal
    3.5e-10      // another Double literal
    true         // the Boolean true
    false        // the Boolean false
```

Each line here is a separate expression. At the end of the line, the notation `// write something here` is a _comment_, which means it is a part of your Ergo program which is ignored by the Ergo compiler. It can be useful to document your code.

Every Ergo expression can be _evaluated_, which means it should compute some value. In the case of a literal value, the result of evaluation is itself (e.g., the expression `1` evaluates to the integer `1`).

> You can actually see the result of evaluating expressions by trying them out in the [Ergo REPL](https://ergorepl.netlify.com). You have to prefix them with `return`: for instance, to evaluate the String literal `"John Smith"` type: `return "John Smith"` (followed by clicking the button 'Evaluate') in the REPL. This should answer: `Response. "John Smith" : String`.

## Operators

You can apply operators to values. Those can be used for arithmetic operations, to compare two values, to concatenate two string values, etc.

```ergo
    1.0 + 2.0 * 3.0      // arithmetic operators on Double
    -1.0
    1 + 2 * 3            // arithmetic operators on Integer
    -1

    1.0 <= 3.0           // comparison operators on Double
    1.0 = 2.0
    2.0 > 1.0
    1 <= 3               // comparison operators on Integer
    1 = 2
    2 > 1.0

    true or false        // Boolean disjunction
    true and false       // Boolean conjunction
    !true                // Negation

    "Hello" ++ " World!" // String concatenation
```

> Again, you can try those in the [Ergo REPL](https://ergorepl.netlify.com). For instance, typing `return true and false` should answer `Response. false : Boolean`, and typing `return 1.0 + 2.0 * 3.0` should answer: `Response. 7.0 : Double`.

## Conditional expressions

Conditional expressions can be used to perform different computations depending on some condition:

```ergo
    if 1.0 < 0.0     // Condition
    then "negative"  // Expression if condition is true
    else "positive"  // Expression if condition is false
```

> Typing `return if 1.0 < 0.0 then "negative" else "positive"` in the [Ergo REPL](https://ergorepl.netlify.com), should answer `Response. "positive" : String`.

See also the [Conditional Expression Reference](ergo-reference.html#condition-
expressions)

## Let bindings

Local variables can be declared with `let`:

```ergo
    let x = 1;              // declares and initialize a variable
    x+2                     // rest of the expression, where x is in scope
```

Let bindings give a name to some intermediate result and allows you to reuse the
corresponding value in multiple places:

```ergo
   let x = -1.0;            // bind x to the value -1.0
   if x < 0.0               // if x is negative
   then -x                  // then return the opposite of x
   else x                   // else return x
```

> **TechNote:** let bindings in Ergo are immutable, in a way similar to other
functional languages. A nice explanation can be found e.g., in the documentation
for let bindings in [ReasonML](https://reasonml.github.io/docs/en/let-binding).

---
id: version-0.20-logic-simple-type
title: Introducing Types
original_id: logic-simple-type

We have so far talked about types only informally. When we wrote earlier:
```ergo
    "John Smith" // a String literal
    1            // an Integer literal
    ...
```
the comments mention that `"John Smith"` is of type `String`, and that `1` is of
type `Integer`.

In reality, the Ergo compiler understands which types your expressions have and can
detect whether those expressions apply to the right type(s) or not.

Ergo types are based on the [Concerto Modeling](model-concerto) Language.

## Primitive types

The simplest of types are primitive types which describe the various kinds of
literal values we saw in the previous section. Those primitive types are:

```ergo
    Boolean
    String
    Double
    Integer
    Long
```

~~~~
    DateTime
```

:::note
The two primitive types `Integer` and `Long` are currently treated as the same type
by the Ergo compiler.
:::

## Type errors

The Ergo compiler understand types and can detect type errors when you write
expressions. For instance, if you write: `1.0 + 2.0 * 3.0`, the Ergo compiler knows
that the expression is correct since all parameters for the operators `+` and `*`
are of type `Double`, and it knows the result of that expression will be a `Double`
as well.

If you write `1.0 + 2.0 * "some text"` the Ergo compiler will detect that `"some
text"` is of type `String`, which is not of the right type for the operator `*` and
return a type error.

> Typing `return 1.0 + 2.0 * "some text"` in the [Ergo
REPL](https://ergorepl.netlify.com), should answer a type error:
> ```text
> Type error (at line 1 col 13). This operator received
> unexpected arguments of type Double  and String.
> return 1.0 + 2.0 * "some text"
>              ^^^^^^^^^^^^^^^^^
> ```

## Type annotations

In a let bindings, you can also use a _type annotation_ to indicate which type you
expect it to have.

```ergo
    let name : String = "John"; // declares and initialize a string variable
    name ++ " Smith"            // rest of the expression
```
or
```ergo
    let x : Double = 3.1416     // declares and initialize a double variable
    sqrt(x)                     // rest of the expression
```

This can be useful to document your code, or to remember what type you expect from
an expression.

The Ergo compiler will return a type error if the annotation is not consistent with
the expression that computes the value for that let binding. For instance, the
following will return a type error since `"pi!"` is not of type `Double`.

```ergo
    let x : Double = "pi!"; // TYPE ERROR: "pi!" is not a Double
    sqrt(x)
```

> Typing `return let x : Double = "pi!"; sqrt(x)` in the [Ergo
REPL](https://ergorepl.netlify.com), should answer a type error:
> ```text
~~~~

> Type error (at line 1 col 7). The let type annotation Double for
> the name x does not match the actual type String.
> return let x : Double = "pi!"; sqrt(x)
>   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
> ```

This becomes particularly useful as your code becomes more complex. For instance the following expression will also trigger a type error:

```ergo
    let rate = 3.5;
    let name : String =
      if rate > 0.0
      then 3.14         // TYPE ERROR: 3.14 is not a String
      else "John";
    name ++ " Smith"
```

Since not all the cases of the `if ... then ... else ...` expressions return a value of type `String` which is the type annotation for the `name` variable.

---
id: version-0.20-logic-stmt
title: Statements
original_id: logic-stmt

A clause's body is composed of statements. Statements are a special kind of expression which can manipulate the contract state and emit obligations. Unlike other expressions they may return a response or an error.

## Contract data

When inside a statement, data about the contract -- either the contract parameters, clause parameters or contract state are available using the following Ergo keywords:
```ergo
    contract   // The contract parameters (from a contract template)
    clause     // Local clause parameters (from a clause template)
    state      // The contract state
```

For instance, if your contract template parameters and state information have the following types:
```ergo
    // Template parameters
    asset InstallmentSaleContract extends AccordContract {
      o AccordParty BUYER
      o AccordParty SELLER
      o Double INITIAL_DUE
      o Double INTEREST_RATE
      o Double TOTAL_DUE_BEFORE_CLOSING
      o Double MIN_PAYMENT
      o Double DUE_AT_CLOSING
      o Integer FIRST_MONTH
    }
    // Contract state
```

    enum ContractStatus {
      o WaitingForFirstDayOfNextMonth
      o Fulfilled
    }
    asset InstallmentSaleState extends AccordContractState {
      o ContractStatus status
      o Double balance_remaining
      o Integer next_payment_month
      o Double total_paid
   }

You can use the following expressions:
```ergo
    contract.BUYER
    state.balance_remaining
```

## Returning a response

Returning a response from a clause can be done by using a `return` statement:

```ergo
     return 1                          // Return the integer one
     return Payout{ amount: 39.99 } // Return a new Payout object
     return                            // Return nothing
```

> **TechNote:** the [Ergo REPL](https://ergorepl.netlify.com) takes statements as input which is why we had to add `return` to expressions in previous examples.

## Returning a failure

Returning a failure from a clause can be done by using a `throw` statement:
```ergo
    throw ErgoErrorResponse{ message: "This is wrong" }
    define concept MyOwnError extends ErgoErrorResponse{ fee: Double }
    throw MyOwnError{ message: "This is wrong and costs a fee", fee: 29.99 }
```

For convenience, Ergo provides a `failure` function which takes a string as part of its [standard library](ref-logic-stdlib#other-functions), so you can also write:
```ergo
    throw failure("This is wrong")
```

## Enforce statement

Before a contract is enforceable some preconditions must be satisfied:
- Competent parties who have the legal capacity to contract
- Lawful subject matter
- Mutuality of obligation
- Consideration

The constructs below will be used to determine if the preconditions have been met and what actions to take if they are not

```test
Example Prose
```

    Do the parties have adequate funds to execute this contract?

One can check preconditions in a clause using enforce statements, as
follows:

```ergo
    enforce x >= 0.0                     // Condition
    else throw failure("not positive"); // Statement if condition is false
    return x+1.0                         // Statement if condition is true
```

The else part of the statement can be omitted in which case Ergo
returns an error by default.

```ergo
    enforce x >= 0.0;          // Condition
    return x+1.0               // Statement if condition is true
```

## Emitting obligations

When inside a clause or contract, you can emit (one or more) obligations as
follows:
```ergo
   emit PaymentObligation{ amount: 29.99, description: "12 red roses" };
   emit PaymentObligation{ amount: 19.99, description: "12 white tulips" };
   return
```

Note that `emit` is always terminated by a `;` followed by another statement.

## Setting the contract state

When inside a clause or contract, you can create a contract state as follows:
```ergo
    set state InstallmentSaleState{
      stateId: "#1",
      status: "WaitingForFirstDayOfNextMonth",
      balance_remaining: contract.INITIAL_DUE,
      total_paid: 0.0,
      next_payment_month: contract.FIRST_MONTH
    };
    return
```

Note that `set state` is always terminated by a `;` followed by another statement.

Once the state is set, you can change its properties individually with the shorter:
```ergo
set state.total_paid = 100.0;
return
```

## Printing intermediate results

For debugging purposes a special `info` statement can be used in your contract
logic. For instance, the following indicates that you would like the Ergo execution
engine to print out the result of expression `state.status` on the standard output.

```ergo
    set state InstallmentSaleState{
      stateId: "#1",
      status: "WaitingForFirstDayOfNextMonth",
      balance_remaining: contract.INITIAL_DUE,
      total_paid: 0.0,
      next_payment_month: contract.FIRST_MONTH
    };
    info(state.status);     // Directive to print to standard output
    return
```

---
id: version-0.20-markup-blocks
title: Block Expressions
original_id: markup-blocks

CiceroMark uses block expressions to enable more advanced scenarios, to handle
optional or repeated text (e.g., lists), to change the variables in scope for a
given section of the text, etc.

Block expressions always have the following syntactic structure:

```tem
{{#blockName variableName parameters}}
...
{{/blockName}}
```

where `blockName` indicates which kind of block it is (e.g., conditional block or
list block), `variableName` indicates the template variable which is in scope
within the block. For certain blocks, additional `parameters` can be passed to
control the behavior of that block (e.g., the `join` block creates text from a list
with an optional separator).

## List Blocks

### Unordered Lists

```tem
{{#ulist rates}}{{volumeAbove}}$ M<= Volume < {{volumeUpTo}}$ M :
{{rate}}%{{/ulist}}
```

#### Example

Drafting text with this block using the following JSON data:
```json
{
  "$class": "org.accordproject.volumediscountlist.VolumeDiscountContract",
  "contractId": "19243313-adc2-4ff1-aa41-993816ed2cdc",
  "rates": [
    {
      "$class": "org.accordproject.volumediscountlist.RateRange",
      "volumeUpTo": 1,
```

      "volumeAbove": 0,
      "rate": 3.1
    },
    {
      "$class": "org.accordproject.volumediscountlist.RateRange",
      "volumeUpTo": 10,
      "volumeAbove": 1,
      "rate": 3.1
    },
    {
      "$class": "org.accordproject.volumediscountlist.RateRange",
      "volumeUpTo": 50,
      "volumeAbove": 10,
      "rate": 2.9
    }
  ]
}

results in the following markdown text:

```md
- 0.0$ M <= Volume < 1.0$ M : 3.1%
- 1.0$ M <= Volume < 10.0$ M : 3.1%
- 10.0$ M <= Volume < 50.0$ M : 2.9%
```

### Ordered Lists

```tem
{{#olist rates}}{{volumeAbove}}$ M <= Volume < {{volumeUpTo}}$ M :
{{rate}}%{{/olist}}
```

#### Example

Drafting text with this block using the following JSON data:
```json
{
  "$class": "org.accordproject.volumediscountlist.VolumeDiscountContract",
  "contractId": "19243313-adc2-4ff1-aa41-993816ed2cdc",
  "rates": [
    {
      "$class": "org.accordproject.volumediscountlist.RateRange",
      "volumeUpTo": 1,
      "volumeAbove": 0,
      "rate": 3.1
    },
    {
      "$class": "org.accordproject.volumediscountlist.RateRange",
      "volumeUpTo": 10,
      "volumeAbove": 1,
      "rate": 3.1
    },
    {
      "$class": "org.accordproject.volumediscountlist.RateRange",
      "volumeUpTo": 50,
      "volumeAbove": 10,
      "rate": 2.9
```

~~~
    }
  ]
}
```

results in the following markdown text:
```md
1. 0.0$ M <= Volume < 1.0$ M : 3.1%
2. 1.0$ M <= Volume < 10.0$ M : 3.1%
3. 10.0$ M <= Volume < 50.0$ M : 2.9%
```

## Conditional Blocks

Conditional blocks enables text which depends on a value of a `Boolean` variable in
your model:

```tem
{{#if forceMajeure}}This is a force majeure{{/if}}
```

:::note
Conditional blocks replace the notion of conditional variables used in Cicero
version `0.13` or earlier. If you need to migrate templates created for a previous
version of Cicero, please refer to the [0.13 to 0.20 Migration Guide](ref-migrate-
0.13-0.20).
:::

Conditional blocks can also include an `else` branch to indicate that some other
text should be use when the value of the variable is `false`:

```tem
{{#if forceMajeure}}This is a force majeure{{else}}This is *not* a force
majeure{{/if}}
```

#### Examples

Drafting text with the first conditional block above using the following JSON data:
```json
{
  "$class": "org.accordproject.foo.Status",
  "forceMajeure": true
}
```

results in the following markdown text:

```md
This is a force majeure
```

Drafting text with this block using the following JSON data:
```json
{
  "$class": "org.accordproject.foo.Status",
  "forceMajeure": false
}
```
~~~

results in the following markdown text:

```md

```

## Clause Blocks

Blocks can be used to inline a clause's text within a contract template:

```tem
Payment
-------
{{#clause payment}}
As consideration in full for the rights granted herein, Licensee shall pay Licensor
a one-time
fee in the amount of {{amountText}} ({{amount}}) upon execution of this Agreement,
payable as
follows: {{paymentProcedure}}.
{{/clause}}
```

#### Example

Drafting text with this block using the following JSON data:
```json
{
  "$class": "org.accordproject.copyrightlicense.CopyrightLicenseContract",
  "contractId": "944535e8-213c-4649-9e60-cc062cce24e8",
  ...
  "paymentClause": {
    "$class": "org.accordproject.copyrightlicense.PaymentClause",
    "clauseId": "6c7611dc-410c-4134-a9ec-17fb6aad5607",
    "amountText": "one hundred US Dollars",
    "amount": {
      "$class": "org.accordproject.money.MonetaryAmount",
      "doubleValue": 100,
      "currencyCode": "USD"
    },
    "paymentProcedure": "bank transfer"
  }
}
```

results in the following markdown text:

```md
Payment
----

As consideration in full for the rights granted herein, Licensee shall pay Licensor
a one-time
fee in the amount of "one hundred US Dollars" (100.0 USD) upon execution of this
Agreement, payable as
follows: "bank transfer".

```

## With Blocks

A `with` block can be used to changes variables that are in scope in a specific part of a template grammar:

```tem
For the Tenant: {{#with tenant}}{{partyId}}, domiciled at {{address}}{{/with}}
For the Landlord: {{#with landlord}}{{partyId}}, domiciled at {{address}}{{/with}}
```

#### Example

Drafting text with this block using the following JSON data:
```json
{
  "$class": "org.accordproject.rentaldeposit.RentalDepositClause",
  "contractId": "31d817e2-d62a-4b70-b395-acd0d5da09f5",
  "tenant": {
    "$class": "org.accordproject.rentaldeposit.RentalParty",
    "partyId": "Michael",
    "address": "111, main street"
  }
  ...
}
```

results in the following markdown text:

```md
For the Tenant: "Michael", domiciled at "111, main street"
For the Landlord: "Parsa", domiciled at "222, chestnut road"
```

## User Feedback

:::note
Other templating systems, e.g., Handlebars, offer a variety of features which are not currently supported in CiceroMark, such as:
- Iterators Blocks `{{#each}}...{{/each}}`
- Comments `{{!-- ... --}}`
- Whitespace control `{{~price~}}`
We welcome user feedback on whether those (or other) features would be useful. Please visit the issues list in the [Cicero](https://github.com/accordproject/cicero) GitHub repository for related discussions or to contribute.
:::

---
id: version-0.20-markup-cicero
title: CiceroMark Overview
original_id: markup-cicero

## Preliminaries

The Cicero markup language (or CiceroMark) is used to express the natural language text for legal clauses or contracts. As with other markup languages, CiceroMark can

express document structure (e.g., headings, paragraphs, lists) as well as
formatting useful for readability (e.g., italics, bold, quotations). In addition,
CiceroMark can also include variables for the template and calculations based on
the values of those variables.

CiceroMark looks like regular [Markdown](https://en.wikipedia.org/wiki/Markdown),
with embedded CiceroMark expressions. Let us look again at the [fixed rate loan]
(https://templates.accordproject.org/fixed-interests-static@0.2.0.html) clause that
was used in the [Overview](accordproject) Section of this documentation.

```tem
## Fixed rate loan

This is a *fixed interest* loan to the amount of {{loanAmount}}
at the yearly interest rate of {{rate}}%
with a loan term of {{loanDuration}},
and monthly payments of {{% monthlyPaymentFormula(loanAmount,rate,loanDuration) %}}
```

This example illustrates all the key features of CiceroMark:
1. _Markdown_: e.g., `## Fixed rate loan` for level 2 heading, and `*fixed
interest*` for italics.
2. _Variable expressions_: e.g., `{{loanAmount}}` which is the amount for the loan.
3. _Ergo expressions_: e.g., `{{%
monthlyPaymentFormula(loanAmount,rate,loanDuration) %}}` which calculates the
monthly payment based on the `loanAmount`, `rate`, and `loanDuration` variables.

### Lexical Conventions

CiceroMark is a string of `UTF-8` characters.

:::note
By convention, CiceroMark files have the `.md` extensions for a sample text, or
`.tem.md` extension for a grammar.
:::

The two sequences of characters `{{` and `}}` are reserved and used for CiceroMark
expressions.

### Markdown

CiceroMark is based on the [CommonMark](https://commonmark.org) markdown syntax.

Except for the two sequences of characters `{{` and `}}` CiceroMark follows the
[CommonMark specification](https://spec.commonmark.org/0.29/) (a new line followed
by the character `#` is interpreted as a level-1 heading, two new lines followed by
text is interpreted as a paragraph, etc).

An introduction to CommonMark can be found in the [Rich Text Markdown](markup-
commonmark) Section.

### CiceroMark Expressions

There are three kinds of CiceroMark expressions, which are similar to expressions
found in other templating systems such as [Handlebars](https://handlebarsjs.com):
- variable expressions (written `{{variableName}}`) which may include an optional
formatting (written `{{variableName as "FORMAT"}}`).
- block expressions which may contain additional text or markup (written
`{{#blockName variableName}} ... text and markup ... {{/blockName}}`).

- Ergo expressions (written `{{% ergoExpression %}}`).

Details and examples of CiceroMark expressions can be found in the [Variable Expressions](markup-variables), [Block Expressions](markup-blocks) and [Ergo Expressions](markup-ergo) Sections.

## Processing CiceroMark

There are two main operations on CiceroMark: **drafting** and **parsing**.

Drafting creates text from data in the [JSON](http://json.org) format. Parsing extracts data in the JSON format from text.

We have already illustrated those operations in the [Hello World Template](started-hello) tutorial, but we review them again here with a focus on how they behave with respect to CiceroMark.

### Drafting: From Data to Text

The most direct way to understand CiceroMark is as a way to generate text from input data. Let's consider the following data, which associates values to the variables in the fixed rate loan template.
```json
{
  "$class": "org.accordproject.interests.TemplateModel",
  "clauseId": "1055c2eb-1cf7-438d-81c7-ec7a91374d9e",
  "loanAmount": 100000.0,
  "rate": 2.5,
  "loanDuration": 15
}
```

From the template and that data, one can create the corresponding clause text:

```md
## Fixed rate loan

This is a *fixed interest* loan to the amount of 10000.0
at the yearly interest rate of 2.5%
with a loan term of 15,
and monthly payments of {{667}}
```

:::tip
Going from data to text can be done using the [`cicero draft`](cicero-cli.html#cicero-draft) command.
:::

The variables are replaced by their values in the text, and the Ergo expression which calculates the monthly payment is evaluated (yielding `667` in this example). In most of this guide, we use text generation to illustrate CiceroMark. However we can also use the template to go from text to data.

### Parsing: From Text to Data

Parsing performs the reverse operation and extract data from text. So from the previous clause text:

```md
```

~~~
## Fixed rate loan

This is a *fixed interest* loan to the amount of 150000.0
at the yearly interest rate of 3.5%
with a loan term of 20,
and monthly payments of {{870}}
```

parsing will extract the following deal data in JSON format:
```json
{
  "$class": "org.accordproject.interests.TemplateModel",
  "clauseId": "1055c2eb-1cf7-438d-81c7-ec7a91374d9e",
  "loanAmount": 150000.0,
  "rate": 3.5,
  "loanDuration": 20
}
```

:::tip
Going from text to data can be done using the [`cicero parse`](cicero-
cli.html#cicero-parse) command.
:::

Several important remarks are important about parsing.

First, parsing will expect the values corresponding to variable or Ergo expressions
to adhere to a specific syntax. Notably, text corresponding to variable expressions
with the type `String` have to be quoted with `"`, and text corresponding to Ergo
expressions have to be quoted with `{{ ... }}`.

Second, because markdown sometimes allows the same thing to be written in different
ways, parsing is resilient to those variations. For instance, the following text
uses a [Setext Heading](https://spec.commonmark.org/0.29/#setext-headings) rather
than an [ATX Heading](https://spec.commonmark.org/0.29/#atx-headings) and uses
`_fixed interest_` rather than `*fixed interest*` for italics, but it will parse
with the same grammar and yield the same data:
```md
Fixed rate loan
----

This is a _fixed interest_ loan to the amount of 150000.0
at the yearly interest rate of 3.5%
with a loan term of 20,
and monthly payments of {{870}}
```

Finally, parsing ignores the text corresponding to the Ergo expression `{{870}}`
since it only needs to extract the other variables. This means the following text
will parse with the same grammar and yield the same data:
```md
Fixed rate loan
----

This is a _fixed interest_ loan to the amount of 150000.0
at the yearly interest rate of 3.5%
with a loan term of 20,
and monthly payments of {{anything else here}}
```
~~~

:::tip
For convenience, the command [`cicero normalize`](cicero-cli.html#cicero-normalize)
can be used to parse, then re-draft a CiceroMark document, allowing a user to both
normalize the markdown and to recalculate the Ergo expressions. For instance,
[`cicero normalize`](cicero-cli.html#cicero-normalize) applied to the following
document:
```md
## Fixed rate loan

This is a _fixed interest_ loan to the amount of 150000.0
at the yearly interest rate of 3.5%
with a loan term of 20,
and monthly payments of {{anything else here}}
```
yields the normalized document:
```md
Fixed rate loan
----

This is a *fixed interest* loan to the amount of 150000.0
at the yearly interest rate of 3.5%
with a loan term of 20,
and monthly payments of {{870}}
```
:::

---
id: version-0.20-markup-commonmark
title: Rich Text Markdown
original_id: markup-commonmark

The following CommonMark guide is non normative, but included for convenience. For
a more detailed introduction and the official reference, we refer the reader the
[CommonMark page](https://commonmark.org/).

## Formatting

### Italics

To italicize text, add one asterisk `*` or underscore `_` both before and after the
relevant text.

##### Example

```md
_Donoghue v Stevenson_ is a landmark tort law case.
```
will be rendered as:

> _Donoghue v Stevenson_ is a landmark tort law case.

### Bold
To bold text, add two asterisks `**` or two underscores `__` both before and after
the relevant text.

##### Example

~~~
```md
**Price** is defined in the Appendix.
```

will be rendered as:

> **Price** is defined in the Appendix.


### Bold and Italic
To bold _and_ italicize text, add `***` both before and after the relevant text.

##### Example

```md
***WARNING***: This product contains chemicals that may cause cancer.
```
will be rendered as:

> ***WARNING***: This product contains chemicals that may cause cancer.

## Paragraphs
To start a new paragraph, insert one or more blank lines. (In other words, all
paragraphs in markdown need to have one or more blank lines between them.)

##### Example

```md
This is the first paragraph.

This is the second paragraph.
This is not a third paragraph.
```
will be rendered as:

>This is the first paragraph.
>
>This is the second paragraph.
>This is not a third paragraph.


## Headings

### Using `#` (ATX Headings)

Level-1 through level-6 headings from are written with a `#` for each level.

#### Example

```md
# US Constitution
## Statutes enacted by Congress
### Rules promulgated by federal agencies
#### State constitution
##### Laws enacted by state legislature
###### Local laws and ordinances
```

will be rendered as:
~~~

~~~
> <h1>US Constitution</h1>
> <h2>Statutes enacted by Congress</h2>
> <h3>Rules promulgated by federal agencies</h3>
> <h4>State constitution</h4>
> <h5>Laws enacted by state legislature</h5>
> <h6>Local laws and ordinances</h6>

### Using `=` or `-` (Setext Headings)

Alternatively, headings with level 1 or 2 can be represented by using `=` and `-`
under the text of the heading.

#### Example

```md
Linux Foundation
================

Accord Project
--------------
```

will be rendered as:
> <h1>Linux Foundation</h1>
> <h2>Accord Project</h2>

## Lists

### Unordered Lists
To create an unordered list, use asterisks `*`, plus `+`, or hyphens `-` in the
beginning as list markers.

#### Example

```md
* Cicero
* Ergo
* Concerto
```

Will be rendered as:
>* Cicero
>* Ergo
>* Concerto

### Ordered Lists

To create an ordered list, use numbers followed by a period `.`.

#### Example

```md
1. One
2. Two
3. Three
```

will be rendered as:
>1. One
~~~

~~~~
>2. Two
>3. Three

### Nested Lists

To create a list within another, indent each item in the sublist by four spaces.

#### Example
```md
1. Matters related to the business
    - enter into an agreement...
    - enter into any abnormal contracts...
2. Matters related to the assets
    - sell or otherwise dispose...
    - mortage, ...
```

will be rendered as:
>1. Matters related to the business
>    - enter into an agreement...
>    - enter into any abnormal contracts...
>2. Matters related to the assets
>    - sell or otherwise dispose...
>    - mortgage, ...

## Horizontal Rule

A horizontal rule may be used to create a "thematic break" between paragraph-level
elements. In markdown, you can create a thematic break using either of the
following:

* `___`: three consecutive underscores
* `---`: three consecutive dashes
* `***`: three consecutive asterisks

#### Example

```md
___
---
***
```

Will be rendered as:
>___
>
>---
>
>***

## Escaping

Any markdown character that is used for a special purpose may be _escaped_ by
placing a backslash in front of it.

For instance avoid creating bold or italic when using `*` or `_` in a sentence,
place a backslash `\` in the front, like: `\*` or `\_`.

#### Example
~~~~

```md
This is \_not\_ italics but _this_ is!
```
Will be rendered as:
> This is _not_ italics but _this_ is!

<!--References:
Commonmark official page and tutorial: https://commonmark.org/help/
OpenLaw Beginner's Guide: https://docs.openlaw.io/beginners-guide/
Markdown cheatsheet: https://gist.github.com/jonschlinkert/5854601
Headings example:
http://www.nyc.gov/html/conflicts/downloads/pdf2/municipal_ethics_laws_ny_state/introduction_to_american_law.pdf
-->

---
id: version-0.20-markup-ergo
title: Ergo Expressions
original_id: markup-ergo

CiceroMark allows you to embed computation inside the text of your contract or clause, in the form of Ergo expressions.

## Syntax

Ergo expressions in template text are essentially similar to Excel formulas, and enable to create legal text dynamically, based on the other variables in your contract. They are written `{{% ergoExpression %}}` where `ergoExpression` is any valid [Ergo Expression](logic-ergo).

## Evaluation Context

The context in which expressions within templates text are evaluated includes:
- The contract variables, which can be accessed using the variable name (or `contract.variableName`)
- All constants or functions declared or imported in the main [Ergo module](logic-module) for your template.

#### Fixed Interests Clause

For instance, let us look one more time at [fixed rate loan](https://templates.accordproject.org/fixed-interests-static@0.2.0.html) clause that was used previously:

```tem
## Fixed rate loan

This is a *fixed interest* loan to the amount of {{loanAmount}}
at the yearly interest rate of {{rate}}%
with a loan term of {{loanDuration}},
and monthly payments of {{% monthlyPaymentFormula(loanAmount,rate,loanDuration) %}}
```

The [`logic` directory](https://github.com/accordproject/cicero-template-library/tree/master/src/fixed-interests/logic) for that template includes two Ergo modules:

```
./logic/interests.ergo  // Module containing the monthlyPaymentFormula function
./logic/logic.ergo      // Main module
```

A look inside the `logic.ergo` module shows the corresponding import, which ensures the `monthlyPaymentFormula` function is also in scope in the text for the template:
```
namespace org.accordproject.interests

import org.accordproject.loan.interests.*

contract Interests over TemplateModel {
  ...
}
```

## Other Examples

Ergo provides a wide range of capabilities which you can use to construct the text that should be included in the final clause or contract. Below are a few examples for illustrations, but we encourage you to consult the [Ergo Logic](logic-ergo) guide for a more comprehensive overview of Ergo.

### Path expressions

The contents of complex values can be accessed using the `.` notation.

#### Example

For instance the following template uses the `.` notation to access the first name and last name of the contract author.

```tem
This contract was drafted by {{% author.name.firstName %}} {{% author.name.lastName %}}
```

### Built-in Functions

Ergo offers a number of pre-defined functions for a variety of primitive types. Please consult the [Ergo Standard Library](ref-logic-stdlib) reference for the complete list of built-in functions.

#### Example

For instance the following template uses the `addPeriod` function to automatically include the date at which a lease expires in the text of the contract:

```tem
This lease was signed on {{signatureDate}}, and is valid for a {{leaseTerm}} period.
This lease will expire on {{% addPeriod(signatureDate, leaseTerm) %}}`
```

### Iterators

Ergo's `foreach` expressions lets you iterate over collections of values.

#### Example

For instance the following template uses a `foreach` expression combined with the
`avg` built-in function to include the average product price in the text of the
contract:

```tem
The average price of the products included in this purchase
order is {{% avg(foreach p in products return p.price) %}}.
```

### Conditionals

Conditional expressions lets you include alternative text based on arbitrary
conditions.

#### Example

For instance, the following template uses a conditional expression to indicate the
governing jurisdiction:

```tem
Each party hereby irrevocably agrees that process may be served on it in
any manner authorized by the Laws of {{%
    if address.country = US and getYear(now()) > 1959
    then "the State of " ++ address.state
    else "the Country of " ++ address.country
%}}
```

---
id: version-0.20-markup-variables
title: Variable Expressions
original_id: markup-variables

Each variable starts with `{{` and ends with `}}` and may include an optional
formatting using the keyword `as`:

```tem
{{firstName}}                       // Source variable (used for parsing and
rendering)
{{deliveryDate as "MMMM, DD YYYY"}} // Source variable with date formatting
```

The way variables are handled (both during parsing and drafting) is based on their
type.

## Primitive Types

Standard variables are written `{{variableName}}` where `variableName` is a
variable declared in the model.

The following example shows a template text with three variables (`buyer`,
`amount`, and `seller`):

```tem
```

~~~~
Upon the signing of this Agreement, {{buyer}} shall pay {{amount}} to {{seller}}.
```

### String Variable

#### Description

If the variable `variableName` has type `String` in the model:
```ergo
o String variableName
```
The corresponding instance should contain text between quotes (`"`).

#### Examples

For example, consider the following model:

```ergo
asset Template extends AccordClause {
  o String buyer
  o String supplier
}
```

the following instance text:
```md
This Supply Sales Agreement is made between "Steve Supplier" and "Betty Byer".
```

matches the template:
```tem
This Supply Sales Agreement is made between {{supplier}} and {{buyer}}.
```

while the following instance texts do not match:
```md
This Supply Sales Agreement is made between 2019 and 2020.
```
or
```md
This Supply Sales Agreement is made between Steve Supplier and Betty Byer.
```

### Numeric Variable

#### Description

If the variable `variableName` has type `Double`, `Integer` or `Long` in the model:
```ergo
o Double variableName
o Integer variableName2
o Long variableName3
```
The corresponding instance should contain the corresponding number.

#### Examples

For example, consider the following model:
~~~~

```ergo
asset Template extends AccordClause {
  o Double penaltyPercentage
}
```

the following instance text:
```md
The penalty amount is 10.5% of the total value of the Equipment whose delivery has
been delayed.
```

matches the template:
```tem
The penalty amount is {{penaltyPercentage}}% of the total value of the Equipment
whose delivery has been delayed.
```

while the following instance texts do not match:
```md
The penalty amount is ten% of the total value of the Equipment whose delivery has
been delayed.
```
or
```md
The penalty amount is "10.5"% of the total value of the Equipment whose delivery
has been delayed.
```

## Primitive Types with a Format

Formatted variables are written `{{variableName as "FORMAT"}}` where `variableName` is a variable declared in the model and the `FORMAT` is a type-dependent description for the syntax of the variables in the contract.

The following example shows a template text with one variable with a format `DD/MM/YYYY`.

```tem
The contract was signed on {{contractDate as "DD/MM/YYYY"}}.
```

### DateTime Variables

#### Description

If the variable `variableName` has type `DateTime`:
```ergo
o DateTime variableName
```
The corresponding instance should contain the corresponding date using the format `MM/DD/YYYY`, commonly used in the US.

#### DateTime Formats

DateTime format can be customized inline in a template grammar by including an optional format string using the `as` keyword. The following formatting tokens are supported:

Tokens are case-sensitive.

Input	Example .	Description
`YYYY`	`2014`	4 or 2 digit year
`YY`	`14`	2 digit year
`M`	`12`	1 or 2 digit month number
`MM`	`04`	2 digit month number
`MMM`	`Feb.`	Short month name
`MMMM`	`December`	Long month name
`D`	`3`	1 or 2 digit day of month
`DD`	`04`	2 digit day of month
`H`	`3`	1 or 2 digit hours
`HH`	`04`	2 digit hours
`mm`	`59`	2 digit minutes
`ss`	`34`	2 digit seconds
`SSS`	`002`	3 digit milliseconds
`Z`	`+01:00`	UTC offset

:::note
If `Z` is specified, it must occur as the last token in the format string.
:::

#### Examples

The format of the `contractDate` variable of type `DateTime` can be specified with the `DD/MM/YYYY` format, as is commonly used in Europe.

```tem
The contract was signed on {{contractDate as "DD/MM/YYYY"}}.
The contract was signed on 26/04/2019.
```

Other examples:

```tem
dateTimeProperty: {{dateTimeProperty as "D MMM YYYY HH:mm:ss.SSSZ"}}
dateTimeProperty: 1 Jan 2018 05:15:20.123+01:02
```

```tem
dateTimeProperty: {{dateTimeProperty as "D MMMM YYYY HH:mm:ss.SSSZ"}}
dateTimeProperty: 1 January 2018 05:15:20.123+01:02
```

```tem
dateTimeProperty: {{dateTimeProperty as "D-M-YYYY H mm:ss.SSSZ"}}
dateTimeProperty: 31-12-2019 2 59:01.001+01:01
```

```tem
dateTimeProperty: {{dateTimeProperty as "DD/MM/YYYY"}}
dateTimeProperty: 01/12/2018
```

```tem
dateTimeProperty: {{dateTimeProperty as "DD-MMM-YYYY H mm:ss.SSSZ"}}
dateTimeProperty: 04-Jan-2019 2 59:01.001+01:01
```

## Complex Types

### Enum Types

#### Description

If the variable `variableName` has an enumerated type:
```ergo
o EnumType variableName
```

The corresponding instance should contain a corresponding enumerated value without quotes.

#### Examples

For example, consider the following model:
```ergo
import org.accordproject.money.CurrencyCode from
https://models.accordproject.org/money.cto
asset Template extends AccordClause {
  o CurrencyCode currency
}

```
the following instance text:
```md
Monetary amounts in this contract are denominated in USD.
```

matches the template:
```tem
Monetary amounts in this contract are denominated in {{currency}}.
```

while the following instance texts do not match:
```md
Monetary amounts in this contract are denominated in "USD".
```
or
```md
Monetary amounts in this contract are denominated in $.
```

### Duration Types

#### Description

If the variable `variableName` has type `Duration`:
```ergo
import org.accordproject.time.Duration
o Duration variableName
```

The corresponding instance should contain the corresponding duration written with the amount as a number and the duration unit as literal text.

#### Examples

For example, consider the following model:
```ergo
asset Template extends AccordClause {
  o Duration termination
}
```

the following instance texts:
```md
If the delay is more than 15 days, the Buyer is entitled to terminate this
Contract.
```
and
```md
If the delay is more than 1 week, the Buyer is entitled to terminate this Contract.
```

both match the template:
```tem
If the delay is more than {{termination}}, the Buyer is entitled to terminate this
Contract.
```

while the following instance texts do not match:
```md
If the delay is more than a month, the Buyer is entitled to terminate this
Contract.
```
or
```md
If the delay is more than "two weeks", the Buyer is entitled to terminate this
Contract.
```

### Other Complex Types

#### Description

If the variable `variableName` has a complex type `ComplexType` (such as an
`asset`, a `concept`, etc.)
```ergo
o ComplextType variableName
```

The corresponding instance should contain all fields in the corresponding complex
type in the order they occur in the model, separated by a single white space
character.

#### Examples

For example, consider the following model:
```ergo
import org.accordproject.address.PostalAddress from
https://models.accordproject.org/address.cto
asset Template extends AccordClause {
  o PostalAddress address
}
```

the following instance text:
```md
Address of the supplier: "555 main street" "10290" "" "NY" "New York" "10001".
```

matches the template:
```tem
Address of the supplier: {{address}}.
```

Consider the following model:
```md
import org.accordproject.money.MonetaryAmount from
https://models.accordproject.org/money.cto
asset Template extends AccordClause {
  o MonetaryAmount amount
}
```

the following instance text:
```md
Total value of the goods: 50.0 USD.
```

matches the template:
```tem
Total value of the goods: {{amount}}.
```

---
id: version-0.20-markus-cli
title: Markdown Transform CLI
original_id: markus-cli

Install the `@accordproject/markdown-cli` npm package to access the Markdown
Transform command line interface (CLI). After installation you can use the `markus`
command and its sub-commands as described below.

To install the Markdown CLI:
```
npm install -g @accordproject/markdown-cli@0.8
```

## Usage

`markus` is a command line tool to debug and use markdown transformations.

```md
markus <cmd> [args]

Commands:
  markus parse      parse and transform sample markdown
  markus draft      create markdown text from data
  markus normalize  normalize markdown in a sample (parse & draft)
```

~~~~
Options:
  --version      Show version number                                   [boolean]
  --verbose, -v                                                 [default: false]
  --help         Show help                                             [boolean]
```

## markus parse

The `markus parse` command lets you parse markdown and create a document object
model from it.

```md
## markus parse

parse and transform a sample markdown

Options:
  --version      Show version number                                   [boolean]
  --verbose, -v  verbose output                       [boolean] [default: false]
  --help         Show help                                             [boolean]
  --sample       path to the markdown text                              [string]
  --output       path to the output file                                [string]
  --cicero       further transform to CiceroMark      [boolean] [default: false]
  --slate        further transform to Slate DOM       [boolean] [default: false]
  --html         further transform to HTML            [boolean] [default: false]
```

### Example

For example, using the `parse` command on the `README.md` file from the [Hello
World](https://github.com/accordproject/cicero-template-library/tree/js-release-
0.20/src/helloworld) template:

```
markus parse --sample README.md
```

returns:
```json
info:
{
  "$class": "org.accordproject.commonmark.Document",
  "xmlns": "http://commonmark.org/xml/1.0",
  "nodes": [
    {
      "$class": "org.accordproject.commonmark.Heading",
      "level": "1",
      "nodes": [
        {
          "$class": "org.accordproject.commonmark.Text",
          "text": "Hello World"
        }
      ]
    },
    {
      "$class": "org.accordproject.commonmark.Paragraph",
      "nodes": [
        {
          "$class": "org.accordproject.commonmark.Text",
~~~~

~~~~
          "text": "This is the Hello World of Accord Project Templates. Executing
the clause will simply echo back the text that occurs after the string "
        },
        {
          "$class": "org.accordproject.commonmark.Code",
          "text": "Hello"
        },
        {
          "$class": "org.accordproject.commonmark.Text",
          "text": " prepended to text that is passed in the request."
        }
      ]
    }
  ]
}

```

Try the command yourself with this
[sample.md](https://github.com/accordproject/markdown-transform/blob/master/
packages/markdown-cli/test/data/sample.md) file.

## markus draft

The `markus draft` command lets you take a document object model and generate
markdown text from it. It is the reverse of `markus parse`.

```md
## markus draft

create markdown text from data

Options:
  --version      Show version number                               [boolean]
  --verbose, -v  verbose output                     [boolean] [default: false]
  --help         Show help                                         [boolean]
  --data         path to the data                                   [string]
  --output       path to the output file                            [string]
  --cicero       input data is a CiceroMark DOM     [boolean] [default: false]
  --slate        input data is a Slate DOM          [boolean] [default: false]
  --noWrap       do not wrap CiceroMark variables as XML tags
                                                    [boolean] [default: false]
  --noIndex      do not index ordered lists         [boolean] [default: false]
```

### Example

For example, using the `draft` command for the [sample
acceptance.json](https://github.com/accordproject/markdown-transform/blob/master/
packages/markdown-cli/test/data/acceptance.json) file:

```md
markus draft --data acceptance.json
```

returns:

```md
Heading
~~~~

~~~~
====

And below is a **clause**.

`<clause src="ap://acceptance-of-
delivery@0.12.1#721d1aa0999a5d278653e211ae2a64b75fdd8ca6fa1f34255533c942404c5c1f"
clauseid="479adbb4-dc55-4d1a-ab12-b6c5e16900c0"/>
Acceptance of Delivery. <variable id="shipper" value="%22Party%20A%22"/> will be
deemed to have completed its delivery obligations if in <variable id="receiver"
value="%22Party%20B%22"/>'s opinion, the <variable id="deliverable"
value="%22Widgets%22"/> satisfies the Acceptance Criteria, and <variable
id="receiver" value="%22Party%20B%22"/> notifies <variable id="shipper"
value="%22Party%20A%22"/> in writing that it is accepting the <variable
id="deliverable" value="%22Widgets%22"/>.

Inspection and Notice. <variable id="receiver" value="%22Party%20B%22"/> will have
<variable id="businessDays" value="10"/> Business Days' to inspect and evaluate the
<variable id="deliverable" value="%22Widgets%22"/> on the delivery date before
notifying <variable id="shipper" value="%22Party%20A%22"/> that it is either
accepting or rejecting the <variable id="deliverable" value="%22Widgets%22"/>.

Acceptance Criteria. The "Acceptance Criteria" are the specifications the <variable
id="deliverable" value="%22Widgets%22"/> must meet for the <variable id="shipper"
value="%22Party%20A%22"/> to comply with its requirements and obligations under
this agreement, detailed in <variable id="attachment" value="%22Attachment%20X
%22"/>, attached to this agreement.`

```


If you have tried to parse the
[sample.md](https://github.com/accordproject/markdown-transform/blob/master/
packages/markdown-cli/test/data/sample.md) file, you would see some similarities
between `sample.md` and this code block :)

### `--cicero` flag

However, `markus draft --data` may not work if the `.json` file contains
`CiceroMark` nodes like Clause or Variables.

For example, trying to use the `draft` command for the [acceptance-cicero.md]
(https://github.com/accordproject/markdown-transform/blob/master/packages/markdown-
cli/test/data/acceptance-cicero.json) using:

```md
markus draft --data acceptance-cicero.json
```


returns the following error:

```md
14:13:13 - error: Namespace is not defined for type
org.accordproject.ciceromark.Clause
```


In this case, the problem can be solved by using the `--cicero` flag, to indicate
your input has `CiceroMark` nodes. Therefore, the correct command to use will be:
```md
markus draft --data acceptance-cicero.json --cicero
```
~~~~

## markus normalize

The `markus normalize` command lets you parse markdown and re-draft it from its document object model.

```md
## markus normalize

normalize a sample markdown (parse & redraft)

Options:
  --version      Show version number                                  [boolean]
  --verbose, -v  verbose output                      [boolean] [default: false]
  --help         Show help                                            [boolean]
  --sample       path to the markdown text                             [string]
  --output       path to the output file                               [string]
  --cicero       further transform to CiceroMark     [boolean] [default: false]
  --slate        further transform to Slate DOM      [boolean] [default: false]
  --noWrap       do not wrap variables as XML tags   [boolean] [default: false]
  --noIndex      do not index ordered lists          [boolean] [default: false]
```

### Example

For example, using the `normalize` command on the `README.md` file from the [Hello World](https://github.com/accordproject/cicero-template-library/tree/js-release-0.20/src/helloworld) template:

```md
markus normalize --sample README.md
```

returns:

```md
info:
Hello World
====

This is the Hello World of Accord Project Templates. Executing the clause will simply echo back the text that occurs after the string `Hello` prepended to text that is passed in the request.
```

### Benefits of using `normalize`

In the previous example, using the `normalize` command did not change the `README.md` file much. This is because the `README.md` file was already well formatted.

However, the true benefits of `markus normalize` can be seen when it is used for a `.md` file that is not well formatted.

For example, consider the following markdown text (to try it, please save this as `markus-test.md` on your computer):

```md
## Lists
```

~~~~
1. This is number one
1. This is number one, too.
1. This is another number one.
1. This is the fourth number one.
1. Let's see how this gets normalized.

## Paragraphs

This is the first paragraph.








This is the second paragraph, with many lines in between.
```

Using `markdown normalize` on this code:

```md
markus normalize --sample markus-test.md
```

returns:

```md
14:25:50 - info:
Lists
----

1. This is number one
2. This is number one, too.
3. This is another number one.
4. This is the fourth number one.
5. Let's see how this gets normalized.

Paragraphs
----

This is the first paragraph.

This is the second paragraph, with many lines in between.
```

--------------------------------------------------------------------------------
---
id: version-0.20-model-api
title: Using the API
original_id: model-api
---

## Install the Core Library

To install the core model library in your project:
```
~~~~

~~~
npm install @accordproject/concerto-core@0.20 --save
```

Below are examples of API use.

## Create a Concerto File

```js
namespace org.acme.address

/**
 * This is a concept
 */
concept PostalAddress {
  o String streetAddress optional
  o String postalCode optional
  o String postOfficeBoxNumber optional
  o String addressRegion optional
  o String addressLocality optional
  o String addressCountry optional
}
```

## Create a Model Manager

```js
const ModelManager = require('@accordproject/concerto-core').ModelManager;

const modelManager = new ModelManager();
modelManager.addModelFile( concertoFileText, 'filename.cto');
```

## Create an Instance

```js
const Factory = require('@accordproject/concerto-core').Factory;

const factory = new Factory(modelManager);
const postalAddress = factory.newConcept('org.acme.address', 'PostalAddress');
postalAddress.streetAddress = '1 Maine Street';
```

## Serialize an Instance to JSON

```js
const Serializer = require('@accordproject/concerto-core').Serializer;

const serializer = new Serializer(factory, modelManager);
const plainJsObject = serializer.toJSON(postalAddress); // instance will be
validated
console.log(JSON.stringify(plainJsObject, null, 4);
```

## Deserialize an Instance from JSON

```js
const postalAddress = serializer.fromJSON(plainJsObject); // JSON will be validated
console.log(postalAddress.streetAddress);
```
~~~

---
id: version-0.20-model-classes
title: Classes
original_id: model-classes

## Concepts

Concepts are similar to class declarations in most object-oriented languages, in that they may have a super-type and a set of typed properties:

```js
abstract concept Animal {
  o DateTime dob
}

concept Dog extends Animal {
 o String breed
}
```

Concepts can be declared `abstract` if it should not be instantiated (must be subclassed).

## Assets

An asset is a class declaration that has a single `String` property which acts as an identifier. You can use the `modelManager.getAssetDeclarations` API to look up all assets.

```js
asset Vehicle identified by vin {
  o String vin
}
```

Assets are typically used in your models for the long-lived identifiable Things (or nouns) in the model: cars, orders, shipping containers, products, etc.

## Participants

Participants are class declarations that have a single `String` property acting as an identifier. You can use the `modelManager.getParticipantDeclarations` API to look up all participants.

```js
participant Customer identified by email {
  o String email
}
```

Participants are typically used for the identifiable people or organizations in the model: person, customer, company, business, auditor, etc.

## Transactions

Transactions are similar to participants in that they are also class declarations that have a single `String` property acting as an identifier. You can use the `modelManager.getTransactionDeclarations` API to look up all transactions.

```js
transaction Order identified by orderId {
  o String orderId
}
```

Transactions are typically used in models for the identifiable business events or messages that are submitted by Participants to change the state of Assets: cart check out, change of address, identity verification, place order, etc.

## Events

** TBD **

---
id: version-0.20-model-concerto
title: Concerto Overview
original_id: model-concerto

### Principles

The Concerto Modeling Language (CML) allows you to:
- Define an object-oriented model using a domain-specific language that is much easier to read and write than JSON/XML Schema, XMI or equivalents.
- Optionally edit your models using a powerful [VS Code add-on](https://marketplace.visualstudio.com/items?itemName=accordproject.cicero-vscode-extension) with syntax highlighting and validation
- Create runtime instances of your model
- Serialize your instances to JSON
- Deserialize (and optionally validate) instances from JSON
- Pass JS object instances around your application
- Introspect the model using a powerful set of APIs
- Convert the model to other formats using [concerto-tools](https://github.com/accordproject/concerto/tree/master/packages/concerto-tools)
- Import models from URLs
- Publish your reusable models to any website, including the [Accord Project Open Source model repository](https://models.accordproject.org)

### Metamodel Components

The Concerto metamodel contains the following:
- [Namespaces](model-namespaces)
- [Imports](model-namespaces#imports)
- [Concepts](model-classes#concepts)
- [Assets](model-classes#assets)
- [Participants](model-classes#participants)
- [Transactions](model-classes#transactions)
- [Enumerations & Enumeration Values](model-enum)
- [Properties & Meta Properties](model-properties)
- [Relationships](model-relationships)
- [Decorators](model-decorators)

~~~
--------------------------------------------------------------------------------
---
id: version-0.20-model-decorators
title: Decorators
original_id: model-decorators
---

Model elements may have arbitrary decorators (aka annotations) placed on them.
These are available via API and can be useful for tools to extend the model.

```js
@foo("arg1", 2)
asset Order identified by orderId {
  o String orderId
}
```

Decorators have an arbitrary number of arguments. They support arguments of type:
- String
- Boolean
- Number
- Type reference

Resource definitions and properties may be decorated with 0 or more decorations.
Note that only a single instance of a decorator is allowed on each element type.
I.e. it is invalid to have the @bar decorator listed twice on the same element.

Decorators are accessible at runtime via the `ModelManager` introspect APIs. This
allows tools and utilities to use Concerto to describe a core model, while
decorating it with sufficient metadata for their own purposes.

The example below retrieves the 3rd argument to the foo decorator attached to the
myField property of a class declaration:

```js
const val = myField.getDecorator('foo').getArguments()[2];
```

--------------------------------------------------------------------------------
---
id: version-0.20-model-enums
title: Enumerations
original_id: model-enums
---

Enumerations are used to capture lists of domain values.

```js
enum Cardsuit {
  o CLUBS
  o DIAMONDS
  o HEARTS
  o SPADES
}
```
~~~

---
id: version-0.20-model-namespaces
title: Namespaces
original_id: model-namespaces

Each Concerto file starts with the name of a single namespace, which contains the base definitions of asset, event, participant and transaction. All definitions within a single file belong to the same namespace.

```js
namespace foo
```

### Imports

In order for one namespace to reference types defined in another namespace, the types must be imported. Imports can be either qualified or can use wildcards.

```js
import org.accordproject.address.PostalAddress
```

```js
import org.accordproject.address.*
```

Import also can use the optional `from` declaration to import a model file that has been deployed to a URL.

```js
import org.accordproject.address.PostalAddress from
https://models.accordproject.org/address.cto
```

Imports using a `from` declaration can be downloaded into the model manager by calling `modelManager.updateExternalModels`.

The Model Manager will resolve all imports to ensure that the set of declarations that have been loaded are globally consistent.

---
id: version-0.20-model-properties
title: Properties
original_id: model-properties

Class declarations contain properties. Each property has a type which can either be a type defined in the same namespace, an imported type, or a primitive type.

### Primitive types

Concerto supports the following primitive types:

Type	Description

|`String` | a UTF8 encoded String.
|`Double` | a double precision 64 bit numeric value.
|`Integer` | a 32 bit signed whole number.
|`Long` | a 64 bit signed whole number.
|`DateTime` | an ISO-8601 compatible time instance, with optional time zone and UTZ offset.
|`Boolean` | a Boolean value, either true or false.

### Meta Properties

Property	Description
`[]`	declares that the property is an array
`optional`	declares that the property is not required for the instance to be valid
`default`	declares a default value for the property, if not value is specified
`range`	declares a valid range for numeric properties
`regex`	declares a validation regex for string properties

`String` fields may include an optional regular expression, which is used to validate the contents of the field. Careful use of field validators allows Concerto to perform rich data validation, leading to fewer errors and less boilerplate application code.

`Double`, `Long` or `Integer` fields may include an optional range expression, which is used to validate the contents of the field.

---
id: version-0.20-model-relationships
title: Relationships
original_id: model-relationships

A relationship in Concerto Modeling Language (CML) is a tuple composed of:

1. The namespace of the type being referenced
2. The type name of the type being referenced
3. The identifier of the instance being referenced

Hence a relationship could be: `org.example.Vehicle#123456`

This would be a relationship to the `Vehicle` _type_ declared in the `org.example` _namespace_ with the _identifier_ `123456`.

Relationships are unidirectional and deletes do not cascade, ie. removing the relationship has no impact on the thing that is being pointed to. Removing the thing being pointed to does not invalidate the relationship.

Relationships must be resolved to retrieve an instance of the object being referenced. The act of resolution may result in null, if the object no longer exists or the information in the relationship is invalid. Resolution of relationships is outside of the scope of Concerto.

A property of a class may be declared as a relationship using the `-->` syntax instead of the `o` syntax. The `o` syntax declares that the class contains (has-a) property of that type, whereas the `-->` syntax declares a typed pointer to an external identifiable instance.

In this example, the model declares that an `Order` has-an array of reference to `OrderLines`. Deleting the `Order` has no impact on the `OrderLine`. When the `Order` is serialized the JSON only the IDs of the `OrderLines` are stored within the `Order`, not the `OrderLines` themselves.

```js
asset OrderLine identified by orderLineId {
  o String orderLineId
  o String sku
}

asset Order identified by orderId {
  o String orderId
  --> OrderLine[] orderlines
}
```

---
id: version-0.20-ref-cicero-ui
title: Cicero UI Reference
original_id: ref-cicero-ui

Accord Project publishes [React](https://reactjs.org) user interface components for use in web-applications. Please refer to the cicero-ui project [on GitHub](https://github.com/accordproject/cicero-ui) for detailed usage instructions.

You can preview these components in [Template Studio v2](https://accordproject-studio.netlify.com).

![Template-Studio-V2](/docs/assets/reference/tsv2.png)

## Contract Editor

The Contract Editor component provides a rich-text content editor for contract text with embedded clauses.

> Note that the contract editor does not currently support the full expressiveness of Cicero Templates. Please refer to the Limitations section for details.

### Limitations

The contract editor does not support templates which use the following CiceroMark features:

* Lists containing [nested lists](markup-commonmark#nested-lists)
* Templates [list blocks](markup-blocks#list-blocks)

## Error Logger

The Error Logger component is used to display structured error messages from the Contract Editor.

## Navigation

The Navigation component displays an outline view for a contract, allowing the user to quickly navigate between sections.

## Template Library

The Template Library component displays a vertical list of template metadata, and allows the user to add a clause (instance of a template) to a contract.

---
id: version-0.20-ref-glossary
title: Glossary
original_id: ref-glossary

## Variable

Variables function as ‘placeholders’ for information to be added when a template is used. Variables are the information that will change between usages of a Template. Here as an example of the text of a contract with three Variables defined:

```
Upon the signing of this Agreement, {{buyer}} shall pay {{amount}} to {{seller}}.
```

## Data Model

A Data Model is used to express the variables that are contained within a Template in a structured way. A Data Model provides us with the structure of the ‘fields’ in the contract which are completed when the templated is used for an agreement. For example, ‘Price’ would be a variable name against which a value, say $100, is entered. This enables us to create a contract document that has a definite structure underlying it rather than simply lines of text.

The Data Model is used to create a machine-readable representation of the text of the contract. It does this by creating a link with the text of the contract by defining the Variables that should exist within the contract along with an associated data type. The linkage between the text and the data model is created by referencing the variable in both the model and the text.

For example, if the text is:
```
Upon the signing of this Agreement, {{buyer}} shall pay {{amount}} to {{seller}}.
```

The model will include `buyer`, `amount`, and `seller`. A data type and a value is assigned against each of these variables.

### Components of Data Models

Data models consist of two core components:

- Variable Name: The name of the ‘placeholder’ for data to be added into a template to create an instance of the contract. By naming variables, we are able to specify what data should be entered into that placeholder in the contract.
- Data Type: The data type defines what type, or format, of data should be inserted in the ‘placeholder’.

The **value** is the actual data that is input into the ‘placeholder’. The value is displayed instead of the name of the variable in the Text. For example:

```md
Upon the signing of this Agreement, "Steve" shall pay 100.0 USD to "Dan".
```

In the model, this is represented as:

Variable Name	Data type	Value
buyer	`String`	Steve
amount	`MonetaryAmount`	100.0 USD
seller	`String`	Dan

Here, `amount` should be a combination of a decimalized value (a double) and a currency code, as opposed to an alphanumeric value, and `buyer` and `seller` should be a combination of alphanumeric characters. This ensures that invalid data cannot be added into the contract, much like letters cannot be added into a credit card section of a web form.

---
id: version-0.20-ref-logic-specification
title: Ergo Compiler
original_id: ref-logic-specification

A large part of the Ergo compiler is written as a Coq specification
from which the compiler is extracted.

Ultimately, one of our goals is to provide a full formal semantics for
Ergo in Coq, and prove correct as much of the compilation pipeline as
possible.

## Compiler architecture

### Frontend

![Frontend](/docs/assets/architecture/frontend.svg)

### Code generation

![Codegen](/docs/assets/architecture/codegen.svg)

## Code Overview

The Coq source serves a dual purpose: as Ergo's formal semantics and as part of its implementation through extraction. Here are some entry points to the code.

A browsable version of the Coq code (generated using
[coq2html](https://github.com/xavierleroy/coq2html)) is
available. Below are some of the main intermediate representations and
compilation phases.

### Intermediate representations

- Ergo: [Ergo/Lang/Ergo](assets/specification/ErgoSpec.Ergo.Lang.Ergo.html)
- Ergo calculus:
[ErgoC/Lang/ErgoC](assets/specification/ErgoSpec.ErgoC.Lang.ErgoC.html)

- Ergo NNRC (Named Nested Relational Calculus): [ErgoNNRC/Lang/ErgoNNRC](assets/specification/ErgoSpec.ErgoNNRC.Lang.ErgoNNRC.html)

### Macro expansion

- Ergo to Ergo: [Ergo/Lang/ErgoExpand](assets/specification/ErgoSpec.Ergo.Lang.ErgoExpand.html)

### Namespace resolution

- Ergo to Ergo: [Translation/ErgoNameResolve](assets/specification/ErgoSpec.Translation.ErgoNameResolve.html)

### Translations between intermediate representations

- Ergo to Ergo calculus: [Translation/ErgotoErgoC](assets/specification/ErgoSpec.Translation.ErgotoErgoC.html)
- ErgoC to Ergo NNRC: [Translation/ErgoCtoErgoNNRC](assets/specification/ErgoSpec.Translation.ErgoCtoErgoNNRC.html)

### Type checking

- ErgoC to ErgoC with types: [ErgoCType](assets/specification/ErgoSpec.ErgoC.Lang.ErgoCType.html)

---
id: version-0.20-ref-logic-stdlib
title: Ergo Standard Library
original_id: ref-logic-stdlib

The following libraries are provided with the Ergo compiler.

## Stdlib

The following functions are in the `org.accordproject.ergo.stdlib` namespace and available by default.

### Functions on Integer

Name	Signature	Description
`integerAbs`	`(x:Integer) : Integer`	Absolute value
`integerLog2`	`(x:Integer) : Integer`	Base 2 integer logarithm
`integerSqrt`	`(x:Integer) : Integer`	Integer square root
`integerToDouble`	`(x:Integer) : Double`	Cast to a Double
`integerMod`	`(x:Integer, y:Integer) : Integer`	Integer remainder
`integerMin`	`(x:Integer, y:Integer) : Integer`	Smallest of `x` and `y`
`integerMax`	`(x:Integer, y:Integer) : Integer`	Largest of `x` and `y`

### Functions on Double

Name	Signature	Description

`abs`	`(x:Double) : Double`	Absolute value
`sqrt`	`(x:Double) : Double`	Square root
`exp`	`(x:Double) : Double`	Exponential
`log`	`(x:Double) : Double`	Natural logarithm
`log10`	`(x:Double) : Double`	Base 10 logarithm
`ceil`	`(x:Double) : Double`	Round to closest integer above
`floor`	`(x:Double) : Double`	Round to closest integer below
`truncate`	`(x:Double) : Integer`	Cast to an Integer
`doubleToInteger`	`(x:Double) : Integer`	Same as `truncate`
`minPair`	`(x:Double, y:Double) : Double`	Smallest of `x` and `y`
`maxPair`	`(x:Double, y:Double) : Double`	Largest of `x` and `y`

### Functions on String

Name	Signature	Description
`length`	`(x:String) : Integer`	Prints length of a string
`encode`	`(x:String) : String`	Encode as URI component
`decode`	`(x:String) : String`	Decode as URI component

### Functions on Arrays

Name	Signature	Description
`count`	(x:Any[]) : Integer	Number of elements
`flatten`	(x:Any[][]) : Any[]	Flattens a nested array
`arrayAdd`	`(x:Any[],y:Any[]) : Any[]`	Array concatenation
`arraySubtract`	`(x:Any[],y:Any[]) : Any[]`	Removes elements of `y` in `x`
`inArray`	`(x:Any,y:Any[]) : Boolean`	Whether `x` is in `y`
`containsAll`	`(x:Any[],y:Any[]) : Boolean`	Whether all elements of `y` are in `x`
`distinct`	`(x:Any[]) : Any[]`	Duplicates elimination

Note: For most of these functions, the type-checker infers more precise types than indicated here. For instance `concat([1,2],[3,4])` will return `[1,2,3,4]` and have the type `Integer[]`.

### Aggregate functions

Name	Signature	Description
`max`	(x:Double[]) : Double	The largest element in `x`
`min`	(x:Double[]) : Double	The smallest element in `x`
`sum`	(x:Double[]) : Double	Sum of the elements in `x`
`average`	(x:Double[]) : Double	Arithmetic mean

### Math functions

Name	Signature	Description
`acos`	(x:Double) : Double	The inverse cosine of x
`asin`	(x:Double) : Double	The inverse sine of x
`atan`	(x:Double) : Double	The inverse tangent of x
`atan2`	(x:Double, y:Double) : Double	The inverse tangent of `x / y`
`cos`	(x:Double) : Double	The cosine of x
`cosh`	(x:Double) : Double	The hyperbolic cosine of x
`sin`	(x:Double) : Double	The sine of x
`sinh`	(x:Double) : Double	The hyperbolic sine of x
`tan`	(x:Double) : Double	The tangent of x

| `tanh` | (x:Double) : Double | The hyperbolic tangent of x |

### Other functions

Name	Signature	Description
`failure`	`(x:String) : ErgoErrorResponse`	Ergo error from a string
`toString`	`(x:Any) : String`	Prints any value to a string
`toText`	`(x:Any) : String`	Template variant of `toString` (internal)

## Time

The following functions are in the `org.accordproject.time` namespace and are available by importing that namespace.
They rely on the [time.cto](https://models.accordproject.org/v2.0/time.html) types from the Accord Project models.

### Functions on DateTime

Name	Signature	Description
`now`	`() : DateTime`	Returns the time when execution started
`dateTime`	`(x:String) : DateTime`	Parse a date and time
`getSecond`	`(x:DateTime) : Long`	Second component of a DateTime
`getMinute`	`(x:DateTime) : Long`	Minute component of a DateTime
`getHour`	`(x:DateTime) : Long`	Hour component of a DateTime
`getDay`	`(x:DateTime) : Long`	Day of the month component of a DateTime
`getWeek`	`(x:DateTime) : Long`	Week of the year component of a DateTime
`getMonth`	`(x:DateTime) : Long`	Month component in a DateTime
`getYear`	`(x:DateTime) : Long`	Year component in a DateTime
`isAfter`	`(x:DateTime, y:DateTime) : Boolean`	Whether `x` if after `y`
`isBefore`	`(x:DateTime, y:DateTime) : Boolean`	Whether `x` is before `y`
`isSame`	`(x:DateTime, y:DateTime) : Boolean`	Whether `x` is the same DateTime as `y`
`dateTimeMin`	`(x:DateTime[]) : DateTime`	The earliest in an array of DateTime
`dateTimeMax`	`(x:DateTime[]) : DateTime`	The latest in an array of DateTime
`format`	`(x:DateTime,f:String) : String`	Prints date `x` according to [format](markup-variables#datetime-formats) `f`

### Functions on Duration

Name	Signature	Description
`durationAs`	`(x:Duration, y:TemporalUnit) : Duration`	Change the unit for duration `x` to `y`
`diffDurationAs`	`(x:DateTime, y:DateTime, z:TemporalUnit) : Duration`	Duration between `x` and `y` in unit `z`
`diffDuration`	`(x:DateTime, y:DateTime) : Duration`	Duration between `x` and `y` in seconds
`addDuration`	`(x:DateTime, y:Duration) : DateTime`	Add duration `y` to `x`
`subtractDuration`	`(x:DateTime, y:Duration) : DateTime`	Subtract duration `y` to `x`
`divideDuration`	`(x:Duration, y:Duration) : Double`	Ratio between durations `x` and `y`

### Functions on Period

Name	Signature	Description
`diffPeriodAs`	`(x:DateTime, y:DateTime, z:PeriodUnit) : Period`	Time period between `x` and `y` in unit `z`
`addPeriod`	`(x:DateTime, y:Period) : DateTime`	Add time period `y` to `x`
`subtractPeriod`	`(x:DateTime, y:Period) : DateTime`	Subtract time period `y` to `x`
`startOf`	`(x:DateTime, y:PeriodUnit) : DateTime`	Start of period `y` nearest to DateTime `x`
`endOf`	`(x:DateTime, y:PeriodUnit) : DateTime`	End of period `y` nearest to DateTime `x`

---
id: version-0.20-ref-logic
title: Ergo Language Reference
original_id: ref-logic

## Lexical Conventions

### File Extension

Ergo files have the ``.ergo`` extension.

### Blanks

Blank characters (such as space, tabulation, carriage return) are ignored but they are used to separate identifiers.

### Comments

Comments come in two forms. Single line comments are introduced by the two characters `//` and are terminated by the end of the current line. Multi-line comments start with the two characters `/*` and are terminated by the two characters `*/`. Multi-line comments can be nested.

Here are examples of comments:

```ergo
    // This is a single line comment
    /* This comment spans multiple lines
       and it can also be /* nested */ */
```

### Reserved Words

The following are reserved as keywords in Ergo. They cannot be used as identifiers.

```text
namespace, import, define, function, transaction, concept, event, asset,
participant, enum, extends, contract, over, clause, throws, emits, state, call,
enforce, if, then, else, let, foreach, return, in, where, throw,
constant, match, set, emit, with, or, and, true, false, unit, none
```

## Condition Expressions

Conditional statements, conditional expressions and conditional constructs are features of a programming language which perform different computations or actions depending on whether a programmer-specified boolean condition evaluates to true or false.

Conditional expressions (also known as `if` statements) allow us to conditionally execute Ergo code depending on the value of a test condition. If the test condition evaluates to `true` then the code on the `then` branch is evaluated. Otherwise, when the test condition evaluates to `false` then the `else` branch is evaluated.

### Example

```ergo
if delayInDays > 15.0 then
  BuyerMayTerminateResponse{};
else
  BuyerMayNotTerminateResponse{}
```

### Legal Prose

For example, this corresponds to a conditional logic statement in legal
prose.

    If the delay is more than 15 days, the Buyer is entitled to terminate this
Contract.

### Syntax

```ergo
    if expression1 then     // Condition
      expression2           // Expression if condition is true
    else
      expression3           // Expression if condition is false
```

Where `expression1` is an Ergo expression that evaluates to a Boolean
value (i.e. `true` or `false`), and `expression2` and `expression3` are
Ergo expressions.

> Note that as with all Ergo expressions, new lines and indentation
> don't change the meaning of your code. However it is good practice to
> standardise the way that you using whitespace in your code to make it
> easier to read.

### Usage

If statements can be chained , i.e., `if ... then .... else if ... then ...
else ...` to build more compound conditionals.

```ergo
if request.netAnnualChargeVolume < contract.firstVolume then
  return VolumeDiscountResponse{ discountRate: contract.firstRate }
else if request.netAnnualChargeVolume < contract.secondVolume then
  return VolumeDiscountResponse{ discountRate: contract.secondRate }
else
  return VolumeDiscountResponse{ discountRate: contract.thirdRate }
```

Conditional expressions can also be used as expressions, e.g., inside a constant declaration:

```ergo
define constant price = 42;
define constant message =  if price >i 100 then "High price" else "Low Price";
message;
```

The value of message after running this code will be `"Low Price"`.

### Related

-   [Match expression](ref-logic#match-expressions) - where many
    alternative conditions check the same variable

## Match Expressions

Match expressions allow us to check an expression against multiple
possible values or patterns. If a match is found, then Ergo will
evaluate the corresponding expression.

> Match expressions are similar to `switch` statements in other
> programming languages

### Example

```ergo
match request.status
  with "CREATED" then
    new PayOut{ amount : contract.deliveryPrice }
  with "ARRIVED" then
    new PayOut{ amount : contract.deliveryPrice - shockPenalty }
  else
    new PayOut{ amount : 0.0 }
```

### Legal Prose

> Example needed.

### Syntax

```ergo
match expression0
  with pattern1 then       // Repeat this line
    expression1            //    and this line
  else
    expression2
```

### Usage

You can use a match expression to look for patterns based on the type of
an expression.

```ergo
match response
```

    with let b1 : BuyerMayTerminateResponse then
        // Do something with b1
    with let b2 : BuyerMayNotTerminateResponse then
        // Do something with b2
    else
        // Do a default action

You can use it to match against an optional value.

```ergo
match maybe_response
    with let? b1 : BuyerMayTerminateResponse then
        // Do something when there is a response
    else
        // Do something else when there is no response
```

Often a match expression is a more concise way to represent a
conditional expression with a repeating, regular condition. For example:

```ergo
    if x = 1 then
      ...
    else if x = 2 then
      ...
    else if x = 3 then
      ...
    else if x = 4 then
      ...
    else
      ...
```

This is equivalent to the match expression:

```ergo
    match x
      with 1 then
        ...
      with 2 then
        ...
      with 3 then
        ...
      with 4 then
        ...
      else
        ...
```

## Operator Precedence

Precedence determines the order of operations in expressions with operators of
different priority. In the case of the same precedence, it is based on the
associativity of operators.

### Example

`a = b * c ^ d + e` is the same as `(a = (b * (c ^ d)) + e)` `a = b * c * d / e` is the same as `(a = (((b * c) * d) / e)`

`a.b.c.d.e ^ f` is the same as `(((((a.b).c).d).e) ^ f)`

### Table of precedence

Table of operators in Ergo with their associativity and precedence from highest to lowest:

Order	**Operator(s)**	**Description**	**Associativity**
1	. <br> ?.	field access <br> field access of optional type	left to right
2	[]	array index access	right to left
3	!	logical not	right to left
4	\-	arithmetic negation	right to left
5	++	string concatenation	left to right
6	^	floating point number power	left to right
7	* <br> / <br> %	multiplication <br> division <br> remainder	left to right
8	\+ <br> -	addition <br> subtraction	left to right
9	??	default value of optional type	left to right
10	and	logical conjunction	left to right
11	or	logical disjunction	left to right
12	< <br> > <br> <= <br> >= <br> = <br> !=	less than <br> greater than <br> less or equal <br> greater or equal <br> equal <br> not equal	left to right

----
---
id: version-0.20-ref-migrate-0.13-0.20
title: 0.13 to 0.20
original_id: ref-migrate-0.13-0.20

Much has changed in the new `0.20` release. This guide provides step-by-step instructions to port your Accord Project templates from version `0.13` or earlier to version `0.20`.

:::note
Before following those migration instructions, make sure to first install version `0.20` of Cicero, as described in the [Installation](started-installation) Section of this documentation.
:::

## Metadata Changes

You will first need to update the `package.json` in your template. Remove the Ergo version which is now unnecessary, and change the Cicero version to `^0.20.0`.

#### Example

After those changes, the `accordproject` field in your `package.json` should look as follows (with the `template` field being either `clause` or `contract` depending on the template):
```js
...
    "accordproject": {
        "template": "clause",
        "cicero": "^0.20.0"
    }
```

...
```

Template Directory Changes

The layout of templates has changed to reflect the conceptual notion of Accord Project templates (as a triangle composed of text, model and logic). To migrate a template directory from version `0.13` or earlier to the new `0.20` layout:
1. Rename your `lib` directory to `logic`
2. Rename your `models` directory to `model`
3. Rename your `grammar` directory to `text`
4. Rename your template grammar from `text/template.tem` to `text/grammar.tem.md`
5. Rename your samples from `sample.txt` to `text/sample.md` (or generally any other `sample*.txt` files to `text/sample*.md`)

Example

Consider the [late delivery and penalty](https://templates.accordproject.org/latedeliveryandpenalty@0.14.1.html) clause. After applying those changes, the template directory should look as follows:
```
./.cucumber.js
./README.md
./package.json
./request-forcemajeure.json
./request.json
./state.json

./logic:
./logic/logic.ergo

./model:
./model/clause.cto

./test:
./test/logic.feature
./test/logic_default.feature

./text:
./text/grammar.tem.md
./text/sample-noforcemajeure.md
./text/sample.md
```

Text Changes

Both grammar and sample text for the templates has changed to support rich text annotations through CommonMark and a new syntax for variables. You can find complete information about the new syntax in the [CiceroMark](markup-cicero) Section of this documentation. For an existing template, you should apply the following changes.

Text Grammar Changes

1. Variables should be changed from `[{variableName}]` to `{{variableName}}`
2. Formatted variables should be changed to from `[{variableName as "FORMAT"}]` to `{{variableName as "FORMAT"}}`
3. Boolean variables should be changed to use the new block syntax, from `[{"This
```

is a force majeure":?forceMajeure}]` to `{{#if forceMajeure}}This is a force
majeure{{/if}}`
4. Nested clauses should be changed to use the new block syntax, from
`[{#payment}]As consideration in full for the rights granted herein...[{/payment}]`
to `{{#clause payment}}As consideration in full for the rights granted herein...
{{/clause}}`

:::note
1. Template text is now interpreted as CommonMark which may lead to unexpected
results if your text includes CommonMark characters or structure (e.g., `#` or `##`
now become headings; `1.` or `-` now become lists). You should review both the
grammar and samples so they follow the proper [CommonMark](https://commonmark.org)
rules.
2. The new lexer reserves `{{` instead of reserving `[{` which means you should
avoid using `{{` in your text unless for Accord Project variables.
:::

### Text Samples Changes

You should ensure that any changes to the grammar text is reflected in the samples.
Any change in the grammar text outside of variables should be applied to the
corresponding `sample.md` files as well.

:::note
You can check that the samples and grammar are in agreement by using the `cicero
parse` command.
:::

#### Example

Consider the text grammar for the [late delivery and
penalty](https://templates.accordproject.org/latedeliveryandpenalty@0.14.1.html)
clause:

```md
Late Delivery and Penalty.
In case of delayed delivery[{" except for Force Majeure cases,":? forceMajeure}]
[{seller}] (the Seller) shall pay to [{buyer}] (the Buyer) for every
[{penaltyDuration}]
of delay penalty amounting to [{penaltyPercentage}]% of the total value of the
Equipment
whose delivery has been delayed. Any fractional part of a [{fractionalPart}] is to
be
considered a full [{fractionalPart}]. The total amount of penalty shall not
however,
exceed [{capPercentage}]% of the total value of the Equipment involved in late
delivery.
If the delay is more than [{termination}], the Buyer is entitled to terminate this
Contract.
```

After applying the above rules to the code for the `0.13` version, and identifying
the heading for the clause using the new markdown features, the grammar text
becomes:

```tem
## Late Delivery and Penalty.

In case of delayed delivery{{#if forceMajeure}} except for Force Majeure
```

```
cases,{{/if}}
{{seller}} (the Seller) shall pay to {{buyer}} (the Buyer) for every
{{penaltyDuration}}
of delay penalty amounting to {{penaltyPercentage}}% of the total value of the
Equipment
whose delivery has been delayed. Any fractional part of a {{fractionalPart}} is to
be
considered a full {{fractionalPart}}. The total amount of penalty shall not
however,
exceed {{capPercentage}}% of the total value of the Equipment involved in late
delivery.
If the delay is more than {{termination}}, the Buyer is entitled to terminate this
Contract.
```

To make sure the `sample.md` file parses as well, the heading needs to be similarly identified using markdown:
```md
## Late Delivery and Penalty.

In case of delayed delivery except for Force Majeure cases,
"Dan" (the Seller) shall pay to "Steve" (the Buyer) for every 2 days
of delay penalty amounting to 10.5% of the total value of the Equipment
whose delivery has been delayed. Any fractional part of a days is to be
considered a full days. The total amount of penalty shall not however,
exceed 55% of the total value of the Equipment involved in late delivery.
If the delay is more than 15 days, the Buyer is entitled to terminate this
Contract.
```

## Model Changes

There is no model changes required for this version.

## Logic Changes

Version `0.20` of Ergo has a few new features that are non backward compatible with version `0.13`. Those may require you to change your template logic. The main non-backward compatible feature is the new support for enumerated values.

### Enumerated Values

Enumerated values are now proper values with a proper enum type, and not based on the type `String` anymore.

For instance, consider the enum declaration:
```js
enum Cardsuit {
  o CLUBS
  o DIAMONDS
  o HEARTS
  o SPADES
}
```

In version `0.13` or earlier the Ergo code would write `"CLUBS"` for an enum value and treat the type `Cardsuit` as if it was the type `String`.

As of version `0.20` Ergo writes `CLUBS` for that same enum value and the type

`Cardsuit` is now distinct from the type `String`.

If you try to compile Ergo logic written for version `0.13` or earlier that
features enumerated values, the compiler will likely throw type errors. You should
apply the following changes:

1. Remove the quotes (`"`) around any enum values in your logic. E.g., `"USD"`
should now be replaced by `USD` for monetary amounts;
3. If enum values are bound to variables with a type annotation, you should change
the type annotation from `String` to the correct enum type. E.g., `let x : String =
"DIAMONDS"; ...` should become `let x : Cardsuit = DIAMONDS; ...`;
3. If enum values are passed as parameters in clauses or functions, you should
change the type annotation for that parameter from `String` to the correct enum
type.
4. In a few cases the same enumerated value may be used in different enum types
(e.g., `days` and `weeks` are used in both `TemporalUnit` and `PeriodUnit`). Those
two values will now have different types. If you need to distinguish, you can use
the fully qualified name for the enum value (e.g.,
`~org.accordproject.time.TemporalUnit.days` or
`~org.accordproject.time.PeriodUnit.days`).

### Other Changes

1. `now` used to return the current time but is treated in `0.20` like any other
variables. If your logic used the variable `now` without declaring it, this will
raise a `Variable now not found` error. You should change your logic to use the
`now()` function instead.

#### Example

Consider the Ergo logic for the [acceptance of
delivery](https://templates.accordproject.org/acceptance-of-delivery@0.12.1.html)
clause. Applying the above rules to the code for the `0.13` version:

```ergo
  clause acceptanceofdelivery(request : InspectDeliverable) : InspectionResponse {

    let received = request.deliverableReceivedAt;
    enforce isBefore(received,now) else
      throw ErgoErrorResponse{ message : "Transaction time is before the
deliverable date." }
    ;

    let dur =
      Duration{
        amount: contract.businessDays,
        unit: "days"
      };
    let status =
      if isAfter(now(), addDuration(received, dur))
      then "OUTSIDE_INSPECTION_PERIOD"
      else if request.inspectionPassed
      then "PASSED_TESTING"
      else "FAILED_TESTING"
    ;
    return InspectionResponse{
      status : status,
      shipper : contract.shipper,
      receiver : contract.receiver
```

~~~
    }
  }
```

results in the following new logic for the `0.20` version:

```ergo
  clause acceptanceofdelivery(request : InspectDeliverable) : InspectionResponse {

    let received = request.deliverableReceivedAt;
    enforce isBefore(received,now()) else                // changed to now()
      throw ErgoErrorResponse{ message : "Transaction time is before the
deliverable date." }
    ;

    let dur =
      Duration{
        amount: contract.businessDays,
        unit: ~org.accordproject.time.TemporalUnit.days  // enum value with fully
qualified name
      };
    let status =
      if isAfter(now(), addDuration(received, dur))      // changed to now()
      then OUTSIDE_INSPECTION_PERIOD                     // enum value has no
quotes
      else if request.inspectionPassed
      then PASSED_TESTING                                // enum value has no
quotes
      else FAILED_TESTING                                // enum value has no
quotes
    ;
    return InspectionResponse{
      status : status,
      shipper : contract.shipper,
      receiver : contract.receiver
    }
  }
```

## Command Line Changes

The Command Line interface for Cicero and Ergo has been completely overhauled for
consistency. Release `0.20` also features new command line interfaces for Concerto
and for the new `markdown-transform` project.

If you are familiar with the previous Accord Project command line interfaces (or if
you have scripts relying on the previous version of the command line), here is a
list of changes:

1. Ergo: A single new `ergo` command replaces both `ergoc` and `ergorun`
   - `ergoc` has been replaced by `ergo compile`
   - `ergorun execute` has been replaced by `ergo trigger`
   - `ergorun init` has been replaced by `ergo initialize`
   - All other `ergorun <command>` commands should use `ergo <command>` instead
2. Cicero:
   - The `cicero execute` command has been replaced by `cicero trigger`
   - The `cicero init` command has been replaced by `cicero initialize`
   - The `cicero generateText` command has been replaced by `cicero draft`
   - the `cicero generate` command has been replaced by `cicero compile`
~~~

Note that several options have been renamed for consistency as well. Some of the main option changes are:
1. `--out` and `--outputDirectory` have both been replaced by `--output`
2. `--format` has been replaced by `--target` in the new `cicero compile` command
3. `--contract` has been replaced by `--data` in all `ergo` commands

For more details on the new command line interface, please consult the corresponding [Cicero CLI](cicero-cli), [Concerto CLI](concerto-cli), [Ergo CLI](ergo-cli), and [Markus CLI](markus-cli) Sections in the reference manual.

## API Changes

A number of API changes may affect Node.js applications using Cicero or Ergo packages. The main API changes are:
1. Ergo:
   1. `ergo-engine` package
      - the `Engine.execute()` call has been renamed `Engine.trigger()`
2. Cicero:
   1. `cicero-core` package
      - the `TemplateInstance.generateText()` call has been renamed `TemplateInstance.draft` **and is now an `async` call**
      - the `Metadata.getErgoVersion()` call has been removed
   2. `cicero-engine` package
      - the `Engine.execute()` call has been renamed `Engine.trigger()`
      - the `Engine.generateText()` call has been renamed `Engine.draft()`

## Cicero Server Changes

Cicero server API has been changed to reflect the new underlying Cicero engine. Specifically:
1. The `execute` endpoint has been renamed `trigger`
2. The path to the sample has to include the `text` directory, so instead of `execute/templateName/sample.txt` it should use `trigger/templateName/text%2Fsample.md`

#### Example

Following the [README.md](https://github.com/accordproject/cicero/blob/master/packages/cicero-server/README.md) instructions, instead of calling:
```
curl -X POST --header 'Content-Type: application/json' --header 'Accept: application/json' http://localhost:6001/execute/latedeliveryandpenalty/sample.txt -d '{ "request": { "$class": "org.accordproject.latedeliveryandpenalty.LateDeliveryAndPenaltyRequest", "forceMajeure": false,"agreedDelivery": "December 17, 2017 03:24:00", "deliveredAt": null, "goodsValue": 200.00 }, "state": { "$class": "org.accordproject.cicero.contract.AccordContractState", "stateId" : "org.accordproject.cicero.contract.AccordContractState#1"}}'
```

You should call:
```
curl -X POST --header 'Content-Type: application/json' --header 'Accept: application/json' http://localhost:6001/trigger/latedeliveryandpenalty/sample.md -d '{ "request": { "$class": "org.accordproject.latedeliveryandpenalty.LateDeliveryAndPenaltyRequest", "forceMajeure": false,"agreedDelivery": "December 17, 2017 03:24:00",
```

"deliveredAt": null, "goodsValue": 200.00 }, "state": { "$class":
"org.accordproject.cicero.contract.AccordContractState", "stateId" :
"org.accordproject.cicero.contract.AccordContractState#1"}}'

---
id: version-0.20-ref-testing
title: Testing Reference
original_id: ref-testing

Cicero uses [Cucumber](https://cucumber.io/docs) for writing template tests, which provides a human readable syntax.

This documents the syntax available to write Cicero tests.

## Test Structure

Tests are located in the `./test/` directory for each template, which contains files with the `.feature` extension.

Each file has the following structure:

```gherkin
Feature: Name of the template being tested
  Description for the test

  Background:
    Given that the contract says
"""
Text of the contract instance.
"""

  Scenario: Description for scenario 1
    [[First Scenario Sequence]]

  Scenario: Description for scenario 2
    [[Second Scenario Sequence]]

etc.
```

Each scenario can be thought of as a description for the behavior of the clause or contract template for the contract given as background.

Each scenario corresponds to one call to the contract. I.e., for a given current time, request and contract state, it says what the expected result of executing the contract should be. This can be either:
- A response, a new contract state, and a list of emitted obligations
- An error

## Scenarios

A complete scenario is described in the [Gherkin Syntax](https://cucumber.io/docs/gherkin/reference/) through a sequence of **Step**.

Each step starts with a keyword, either **Given**, **When**, **And**, or **Then**:

- **Given**, **When** and **And** are used to specify the input for a call to the contract;
- **Then** and **And** are used to specify the expected result.

### Request and Response

The simplest kind of scenario specifies the response expected for a given request.

For instance, the following scenario describe the expected response for a given request to the [helloworld template](https://templates.accordproject.org/helloworld@0.10.1.html):

```gherkin
  Scenario: The contract should say Hello to Betty Buyer, from the ACME Corporation
    When it receives the request
"""
{
    "$class": "org.accordproject.helloworld.MyRequest",
    "input": "ACME Corporation"
}
"""
    Then it should respond with
"""
{
    "$class": "org.accordproject.helloworld.MyResponse",
    "output": "Hello Betty Buyer ACME Corporation"
}
"""
```

Both the request and the response are written inside triple quotes `"""` using JSON. If the request or response is not valid wrt. to the data model, this will result in a failing test.

:::warning
While the syntax for each scenario uses _pseudo_ natural language (e.g., `When it receives the request`), the tests much use very specific sentences as illustrated in this guide.
:::

### Defaults

You can use the sample contract `sample.txt` and request `request.json` provided with a template by using specific steps.

For instance, the following scenario describe the expected response for the default contract text when sending the default request to the [helloworld template](https://templates.accordproject.org/helloworld@0.10.1.html):
```gherkin
Feature: HelloWorld
  This describe the expected behavior for the Accord Project's "Hello World!" contract

  Background:
    Given the default contract

  Scenario: The contract should say Hello to Fred Blogs, from the Accord Project,
```

~~~
for the default request
    When it receives the default request
    Then it should respond with
"""
{
    "$class": "org.accordproject.helloworld.MyResponse",
    "output": "Hello Fred Blogs Accord Project"
}
"""
```

### Errors

Whenever appropriate, it is good practice to include both successful executions, as
well as scenarios for cases when a call to a template might fail. This can be
written using a **Then** step that describes the error.

For instance, the following scenario describe an expected error for a given request
to the [Interest Rate Swap](https://templates.accordproject.org/interest-rate-
swap@0.4.1.html) template:
```gherkin
Feature: Interest Rate Swap
  This describes the expected behavior for the Accord Project's interest rate swap
contract

  Background:
    Given that the contract says
"""
INTEREST RATE SWAP TRANSACTION LETTER AGREEMENT
"Deutsche Bank"

Date: 06/30/2005
To: "MagnaChip Semiconductor S.A."
Attention: Swaps Documentation Department
Our Reference: "Global No. N397355N"
Re: Interest Rate Swap Transaction

Ladies and Gentlemen:

The purpose of this letter agreement is to set forth the terms and conditions of
the Transaction entered into between "Deutsche Bank" and "MagnaChip Semiconductor
S.A." (“Counterparty”) on the Trade Date specified below (the “Transaction”). This
letter agreement constitutes a “Confirmation” as referred to in the Agreement
specified below.

The definitions and provisions contained in the 2000 ISDA Definitions (the
“Definitions”) as published by the International Swaps and Derivatives Association,
Inc. are incorporated by reference herein. In the event of any inconsistency
between the Definitions and this Confirmation, this Confirmation will govern.

For the purpose of this Confirmation, all references in the Definitions or the
Agreement to a “Swap Transaction” shall be deemed to be references to this
Transaction.

1. This Confirmation evidences a complete and binding agreement between "Deutsche
Bank" (“Party A”) and Counterparty (“Party B”) as to the terms of the Transaction
to which this Confirmation relates. In addition, Party A and Party B agree to use
all reasonable efforts to negotiate, execute and deliver an agreement in the form
of the ISDA 2002 Master Agreement with such modifications as Party A and Party B
~~~

```
will in good faith agree (the “ISDA Form” or the “Agreement”). Upon execution by
the parties of such Agreement, this Confirmation will supplement, form a part of
and be subject to the Agreement. All provisions contained or incorporated by
reference in such Agreement upon its execution shall govern this Confirmation
except as expressly modified below. Until Party A and Party B execute and deliver
the Agreement, this Confirmation, together with all other documents referring to
the ISDA Form (each a “Confirmation”) confirming Transactions (each a
“Transaction”) entered into between us (notwithstanding anything to the contrary in
a Confirmation) shall supplement, form a part of, and be subject to an agreement in
the form of the ISDA Form as if Party A and Party B had executed an agreement on
the Trade Date of the first such Transaction between us in such form, with the
Schedule thereto (i) specifying only that (a) the governing law is English law,
provided, that such choice of law shall be superseded by any choice of law
provision specified in the Agreement upon its execution, and (b) the Termination
Currency is U.S. Dollars and (ii) incorporating the addition to the definition of
“Indemnifiable Tax” contained in (page 49 of) the ISDA “User’s Guide to the 2002
ISDA Master Agreements”.
2. The terms of the particular Transaction to which this Confirmation relates are
as follows:

Notional Amount: 300000000.00 USD
Trade Date: 06/23/2005
Effective Date: 06/27/2005
Termination Date: 06/18/2008

Fixed Amounts:
Fixed Rate Payer: "Counterparty"
Fixed Rate Payer Period End Dates: "The 15th day of March, June, September and
December of each year, commencing September 15, 2005, through and including the
Termination Date with No Adjustment"
Fixed Rate Payer Payment Dates: "The 15th day of March, June, September and
December of each year, commencing September 15, 2005, through and including the
Termination Date"
Fixed Rate: -4.09%
Fixed Rate Day Count Fraction: "30" "360"
Fixed Rate Payer Business Days:"New York"
Fixed Rate Payer Business Day Convention: "Modified Following"

Floating Amounts:
Floating Rate Payer: "DBAG"
Floating Rate Payer Period End Dates: "The 15th day of March, June, September and
December of each year, commencing September 15, 2005, through and including the
Termination Date with No Adjustment"
Floating Rate Payer Payment Dates: "The 15th day of March, June, September and
December of each year, commencing September 15, 2005, through and including the
Termination Date"
Floating Rate for initial Calculation Period: 3.41%
Floating Rate Option: "USD-LIBOR-BBA"
Designated Maturity: "Three months"
Spread: "None"
Floating Rate Day Count Fraction: "30" "360"
Reset Dates: "The first Floating Rate Payer Business Day of each Calculation Period
or Compounding Period, if Compounding is applicable."
Compounding: "Inapplicable"
Floating Rate Payer Business Days: "New York"
Floating Rate Payer Business Day Convention: "Modified Following"
"""

  Scenario: The fixed rate is negative
```

~~~~
    When it receives the request
"""
{
    "$class": "org.accordproject.isda.irs.RateObservation"
}
"""
    Then it should reject the request with the error "[Ergo] Fixed rate cannot be
negative"
```

The reason for the error is that the contract has been defined with a negative
interest rate (the line: `Fixed Rate: -4.09%` in the contract given as
**Background** for the scenario).

### State Change

For templates which assume and can modify the contract state, the scenario should
also include pre- and post- conditions for that state. In addition, some steps are
available to define scenarios that specify the expected initial step for the
contract.

For instance, the following scenario for the [Installment
Sale](https://templates.accordproject.org/installment-sale@0.12.1.html) template
describe the expected initial state and execution of one installment:
```gherkin
Feature: Installment Sale
  This describe the expected behavior for the Accord Project's installment sale
contract

  Background:
    Given that the contract says
"""
"Dan" agrees to pay to "Ned" the total sum e10000, in the manner following:

E500 is to be paid at closing, and the remaining balance of E9500 shall be paid as
follows:

E500 or more per month on the first day of each and every month, and continuing
until the entire balance, including both principal and interest, shall be paid in
full -- provided, however, that the entire balance due plus accrued interest and
any other amounts due here-under shall be paid in full on or before 24 months.

Monthly payments, which shall start on month 3, include both principal and interest
with interest at the rate of 1.5%, computed monthly on the remaining balance from
time to time unpaid.

"""

  Scenario: The contract should be in the correct initial state
    Then the initial state of the contract should be
"""
{
  "$class": "org.accordproject.installmentsale.InstallmentSaleState",
  "status" : "WaitingForFirstDayOfNextMonth",
  "balance_remaining" : 10000.00,
  "total_paid" : 0.00,
  "next_payment_month" : 3,
  "stateId": "#1"
}
~~~~

~~~
"""

  Scenario: The contract accepts a first payment, and maintain the remaining
balance
    Given the state
"""
{
  "$class": "org.accordproject.installmentsale.InstallmentSaleState",
  "status" : "WaitingForFirstDayOfNextMonth",
  "balance_remaining" : 10000.00,
  "total_paid" : 0.00,
  "next_payment_month" : 3,
  "stateId": "#1"
}
"""
    When it receives the request
"""
{
    "$class": "org.accordproject.installmentsale.Installment",
    "amount": 2500.00
}
"""
    Then it should respond with
"""
{
  "total_paid": 2500,
  "balance": 7612.499999999999,
  "$class": "org.accordproject.installmentsale.Balance"
}
"""
    And the new state of the contract should be
"""
{
  "$class": "org.accordproject.installmentsale.InstallmentSaleState",
  "status" : "WaitingForFirstDayOfNextMonth",
  "balance_remaining" : 7612.499999999999,
  "total_paid" : 2500,
  "next_payment_month" : 4,
  "stateId": "#1"
}
"""

```

### Current Time

The logic for some clause or contract templates is time-dependent. It can be useful
to specify multiple scenarios for the behavior under different date and time
assumptions. This can be described with an additional **When** step to set the
current time to a specific value.

For instance, the following shows two scenarios for the [IP
Payment](https://templates.accordproject.org/ip-payment@0.10.1.html) template,
which describe its expected behavior for two distinct current times:

```gherkin
Feature: IP Payment Contract
  This describes the expected behavior for the Accord Project’s IP Payment Contract
contract
~~~

```
  Background:
    Given the default contract

  Scenario: Payment of a specified amount should be made
    When the current time is "2019-03-04T16:34:00-05:00"
    And it receives the request
"""
{
    "$class": "org.accordproject.ippayment.PaymentRequest",
    "netSaleRevenue": 1200,
    "sublicensingRevenue": 450,
    "permissionGrantedBy": "2018-04-05T00:00:00-05:00"
}
"""
    Then it should respond with
"""
{
    "$class": "org.accordproject.ippayment.PayOut",
    "totalAmount": 77.4,
    "dueBy": "2018-04-12T00:00:00.000-05:00"
}
"""

  Scenario: Payment of a specified amount should be made
    When the current time is "2019-03-01T16:34:00-02:00"
    And it receives the request
"""
{
    "$class": "org.accordproject.ippayment.PaymentRequest",
    "netSaleRevenue": 1550,
    "sublicensingRevenue": 225,
    "permissionGrantedBy": "2018-04-05T00:00:00-05:00"
}
"""
    Then it should respond with
"""
{
    "$class": "org.accordproject.ippayment.PayOut",
    "totalAmount": 81.45,
    "dueBy": "2018-04-12T03:00:00.000-02:00"
}
"""
```

### Emitting Obligations

If the template execution emits obligations, those can also be specified in the scenario as one of the **Then** steps.

For instance, the following shows a scenario for the [Rental Deposit](https://templates.accordproject.org/ip-payment@0.10.1.html) template, which describes the expected list of obligations that should be emitted for a given request:
```gherkin
Feature: Rental Deposit
  This describe the expected behavior for the Accord Project's rental deposit contract
  Background:
    Given the default contract

  Scenario: The property was inspected and there was damage
    When the current time is "2018-01-02T16:34:00Z"
    And it receives the default request
    Then it should respond with
"""
{
   "$class": "org.accordproject.rentaldeposit.PropertyInspectionResponse",
    "balance": {
      "$class": "org.accordproject.money.MonetaryAmount",
      "currencyCode" : "USD",
      "doubleValue" : 1550
    }
}
"""
    And the following obligations should have been emitted
"""
[
    {
        "$class": "org.accordproject.cicero.runtime.PaymentObligation",
        "amount": {
            "$class": "org.accordproject.money.MonetaryAmount",
            "doubleValue": 1550,
            "currencyCode": "USD"
        }
    }
]
"""
```

---
id: version-0.20-started-hello
title: Hello World Template
original_id: started-hello

Once you have installed Cicero, you can try it on an existing Accord Project template. This explains how to create an instance of that template and how to run the contract logic.

## Download a Template

You can download a single clause or contract template from the [Accord Project Template Library](https://templates.accordproject.org) as an archive (`.cta`) file. Cicero archives are files with a `.cta` extension, which includes all the different components for the template (text, model and logic).

If you click on the Template Library link, you should see a Web Page which looks as follows:

![Basic-Use-1](/docs/assets/basic/use1.png)

Scrolling down that page, you can see the index for the open-source templates along with their version, and whether they are a Clause or Contract template.

Click on the link to the `helloworld` template. You should be taken to a page which looks as follows:

![Basic-Use-2](/docs/assets/basic/use2.png)

Then click on the `Download Archive` button under the description for the template
(highlighted in the red box in the figure). This should download the latest
template archive for the `helloworld` template.

## Parse: Extract Deal Data from Text

You can use Cicero to extract deal data from a contract text using the `cicero
parse` command.

### Parse Valid Text

Using your terminal, change into the directory (or `cd` into the directory) that
contains the template archive you just downloaded, then create a sample clause text
`sample.md` which contains the following text:

```md
Name of the person to greet: "Fred Blogs".
Thank you!
```

Then run the `cicero parse` command in your terminal to load the template and parse
your sample clause text. This should be echoing the result of parsing back to your
terminal.

```bash
cicero parse --template helloworld@0.12.0.cta --sample sample.md
```

:::note
* Templates are tied to a specific version of the cicero tool. Make sure that the
version number output from `cicero --version` is compatible with the template. Look
for `^0.20.0` or similar at the top of the template web page.
* `cicero parse` requires network access. Make sure that you are online and that
your firewall or proxy allows access to `https://models.accordproject.org`
:::

This should extract the data (or "deal points") from the text and output:

```json
{
  "$class": "org.accordproject.helloworld.HelloWorldClause",
  "clauseId": "aa3b9db9-f25f-41f4-88a4-64baba728bfe",
  "name": "Fred Blogs"
}
```

You can save the result of `cicero parse` into a file using the `--output` option:
```
cicero parse --template helloworld@0.12.0.cta --sample sample.md --output data.json
```

### Parse Non-Valid Text

If you attempt to parse text which is not valid according to the template, this
same command should return an error.

Edit your `sample.md` file to add text that is not consistent with the template:

```text
FUBAR Name of the person to greet: "Fred Blogs".
Thank you!
```

Then run `cicero parse --template helloworld@0.12.0.cta --sample sample.md` again.
The output should now be:

```text
18:15:22 - error: invalid syntax at line 1 col 1:

  FUBAR Name of the person to greet: "Fred Blogs".
  ^
Unexpected "F"
```

## Draft: Create Text from Deal Data

You can use Cicero to create new contract text from deal data using the `cicero draft` command.

### Draft from Valid Data

If you have saved the deal data earlier in a `data.json` file, you can edit it to change the name from `Fred Blogs` to `John Doe`, or create a brand new `data.json` file containing:
```json
{
  "$class": "org.accordproject.helloworld.HelloWorldClause",
  "clauseId": "aa3b9db9-f25f-41f4-88a4-64baba728bfe",
  "name": "John Doe"
}
```

Then run the `cicero draft` command in your terminal:
```
cicero draft --template helloworld@0.12.0.cta --data data.json
```

This should create a new contract text and output:
```
13:17:18 - info: Name of the person to greet: "John Doe".
Thank you!
```

You can save the result of `cicero draft` into a file using the `--output` option:
```
cicero draft --template helloworld@0.12.0.cta --data data.json --output new-
sample.md
```

### Draft from Non-Valid Data

If you attempt to draft from data which is not valid according to the template, this same command should return an error.

Edit your `data.json` file so that the `name` variable is missing:

```json
{
  "$class": "org.accordproject.helloworld.HelloWorldClause",
  "clauseId": "aa3b9db9-f25f-41f4-88a4-64baba728bfe"
}
```

Then run `cicero draft --template helloworld@0.12.0.cta --data data.json` again.
The output should now be:
```
13:38:11 - error: Instance org.accordproject.helloworld.HelloWorldClause#6f91e060-f837-4108-bead-63891a91ce3a missing required field name
```

## Trigger: Run the Contract Logic

You can use Cicero to run the logic associated to a contract using the `cicero trigger` command.

### Trigger with a Valid Request

Use the `cicero trigger` command to parse a clause text based (your `sample.md`) *then* send a request to the clause logic.

To do so you, first create one additional file `request.json` which contains:
```json
{
  "$class": "org.accordproject.helloworld.MyRequest",
  "input": "Accord Project"
}
```

This is the request which you will send to trigger the execution of your contract.

Then run the `cicero trigger` command in your terminal to load the template, parse your clause text *and* send the request. This should be echoing the result of execution back to your terminal.

```bash
cicero trigger --template helloworld@0.12.0.cta --sample sample.md --request request.json
```

This should print this output:

```json
13:42:29 - info:
{
  "clause": "helloworld@0.12.0-c03393f7e50865012e6005050fcaccb2716481fa7599905f7306673cf15857cf",
  "request": {
    "$class": "org.accordproject.helloworld.MyRequest",
    "input": "Accord Project"
  },
  "response": {
    "$class": "org.accordproject.helloworld.MyResponse",
    "output": "Hello Fred Blogs Accord Project",
    "transactionId": "ecc56a61-713c-4113-9842-550efb09ac74",
    "timestamp": "2019-11-03T18:42:29.984Z"
```

  },
  "state": {
    "$class": "org.accordproject.cicero.contract.AccordContractState",
    "stateId": "org.accordproject.cicero.contract.AccordContractState#1"
  },
  "emit": []
}

The results of execution displayed back on your terminal is in JSON format. It includes the following information:

* Details of the `clause` being triggered (name, version, SHA256 hash of the template)
* The incoming `request` object (the same request from your `request.json` file)
* The output `response` object
* The output `state` (unchanged in this example)
* An array of `emit`ted events (empty in this example)

That's it! You have successfully parsed and executed your first Accord Project Clause using the `helloworld` template.

### Trigger with a Non-Valid Request

If you attempt to trigger the contract from a request which is not valid according to the template, this same command should return an error.

Edit your `request.json` file so that the `input` variable is missing:
```json
{
  "$class": "org.accordproject.helloworld.MyRequest"
}
```

Then run `cicero trigger --template helloworld@0.12.0.cta --sample sample.md --request request.json ` again. The output should now be:
```
13:47:35 - error: Instance org.accordproject.helloworld.MyRequest#b0b1cbcc-dcae-4758-b9fc-254a43aa10a8 missing required field input
```

## What Next?

### Try Other Templates

Feel free to try the same commands to parse and execute other templates from the Accord Project Library. Note that for each template you can find samples for the text, for the request and for the state on the corresponding Web page. For instance, a sample for the [Late Delivery And Penalty](https://templates.accordproject.org/latedeliveryandpenalty@0.15.0.html) clause is in the red box in the following image:

![Basic-Use-3](/docs/assets/basic/use3.png)

### More About Cicero

You can find more information on how to create or publish Accord Project templates in the [Work with Cicero](tutorial-templates) tutorials.

~~~text
### Run on Different Platforms

Templates may be executed on different platforms, not only from the command line.
You can find more information on how to execute Accord Project templates on
different platforms (Node.js, Hyperledger Fabric, etc.) in the [Template Execution]
(tutorial-nodejs) tutorials.


-------------------------------------------------------------------------------
---
id: version-0.20-started-installation
title: Install Cicero
original_id: started-installation
---

To experience the full power of Accord Project, you should install the Cicero
command-line tools. This will let you author, validate, and run Accord Project
templates on your own machine.

## Prerequisites

Before you install Cicero, you must first obtain and configure the following
dependency:

* [Node.js v10.x (LTS)](http://nodejs.org): We use Node.js to run cicero, generate
the documentation, run a development web server, testing, and produce distributable
files. Depending on your system, you can install Node either from source or as a
pre-packaged bundle.

>  We recommend using [nvm](https://github.com/creationix/nvm) (or [nvm-windows]
(https://github.com/coreybutler/nvm-windows)) to manage and install Node.js, which
makes it easy to change the version of Node.js per project.

## Installing Cicero

To install the latest version of the Cicero command-line tools:

```bash
npm install -g @accordproject/cicero-cli@0.20
```

:::note
You can install a specific version by appending `@version` at the end of the `npm
install` command. For instance to install version `0.20.0` or version `0.13.4`:
```bash
npm install -g @accordproject/cicero-cli@0.20.3
npm install -g @accordproject/cicero-cli@0.13.4
```
:::

To check that Cicero has been properly installed, and display the version number:
```bash
cicero --version
```

To get command line help:
```bash
cicero --help
cicero parse --help     // To parse a sample clause/contract
~~~

~~~~
cicero draft --help     // To draft a sample clause/contract
cicero trigger --help   // To send a request to a clause/contract
```

## Optional Packages

### Template Generator

You may also want to install the template generator tool, which you can use to
create an empty template:

```bash
npm install -g yo
npm install -g @accordproject/generator-cicero-template@0.20
```

## What next?

That's it! Go to the next page to see how to use your new installation of Cicero on
a real Accord Project template.


--------------------------------------------------------------------------------
---
id: version-0.20-started-resources
title: Resources
original_id: started-resources
---

## Accord Project Resources

- The Main Web site includes latest news, links to working groups, organizational
announcements, etc. : https://www.accordproject.org
- This Technical Documentation: https://docs.accordproject.org
- Recording of Working Group discussions, Tutorial Videos are available on Vimeo:
https://vimeo.com/accordproject
- Join the [Accord Project Slack](https://accord-project-slack-
signup.herokuapp.com) to get involved!

## User Content

Accord Project also maintains libraries containing open source, community-
contributed content to help you when authoring your own templates:

- [Model Repository](https://models.accordproject.org/) : a repository of open
source Concerto data models for use in templates
- [Template Library](https://templates.accordproject.org/) : a library of open
source Clause and Contract templates for various legal domains (supply-chain,
loans, intellectual property, etc.)

## Ecosystem & Tools

Accord Project is also developing tools to help with authoring, testing and running
accord project templates.

### Editors

- [Template Studio](https://studio.accordproject.org/): a Web-based editor for
Accord Project templates
~~~~

- [VSCode Extension](https://marketplace.visualstudio.com/items?itemName=accordproject.cicero-vscode-extension): an Accord Project extension to the popular [Visual Studio Code](https://visualstudio.microsoft.com/) Editor
- [Emacs Mode](https://github.com/accordproject/ergo/tree/master/ergo.emacs): Emacs Major mode for Ergo (alpha)
- [VIM Plugin](https://github.com/accordproject/ergo/tree/master/ergo.vim): VIM plugin for Ergo (alpha)

### User Interface Components

- [Markdown Editor](https://github.com/accordproject/markdown-editor): a general purpose react component for markdown rendering editing-
- [Cicero UI](https://github.com/accordproject/cicero-ui): a library of react components for visualizing, creating, editing Accord Project templates

## Developers Resources

All the Accord Project technology is being developed as open source. The software packages are being actively maintained on
[GitHub](https://github.com/accordproject) and we encourage organizations and individuals to contribute requirements, documentation, issues, new templates, and code.

Join us on the [#technology-wg Slack channel](https://accord-project-slack-signup.herokuapp.com) for technical discussions and weekly updates.

### Cicero

- GitHub: https://github.com/accordproject/cicero
- [Cicero Slack Channel](https://accord-project.slack.com/messages/CA08NAHQS/details/)
- Technical Questions and Answers on [Stack Overflow](https://stackoverflow.com/questions/tagged/cicero)

### Ergo

- GitHub: https://github.com/accordproject/ergo
- The [Ergo Language Guide](logic-ergo) is a good place to get started with Ergo.
- [Ergo Slack Channel](https://accord-project.slack.com/messages/C9HLJHREG/details/)

```
--------------------------------------------------------------------------------
---
id: version-0.20-started-studio
title: Online Tour
original_id: started-studio
---
```

To get an better acquainted with Accord Project templates, the easiest way is through the online [Template Studio](https://studio.accordproject.org) editor.

:::tip
You can open template studio from anywhere in this documentation by clicking the [Try Online!](https://studio.accordproject.org) button located in the top-right of the page.
:::

The following video offers a tour of Template Studio and an introduction to the key concepts behind the Accord Project technology.

<iframe src="https://player.vimeo.com/video/328933628" width="640" height="400" frameborder="0" allow="autoplay; fullscreen" allowfullscreen></iframe>

Here is a timestamp of what is covered in the video:

- **00:08** : Introduction to template Studio & pointers the other parts of the docs / AP website
- **03:51** : Helloworld template tutorial start
- **04:48** : Template Studio - Metadata
- **06:56** : Template Studio - Contract Text
  - **07:52** : The 'live' nature of the text; how to read errors
  - **08:39** : Changing the template text itself
  - **09:17** : Changing the variables in the template text
- **10:00** : Template Studio - Model update
- **11:28** : Template Studio - Logic update
  - **13:17** : Explanation about request types / response types
- **14:44** : Template Studio - Logic - Test Execution (request, response)
  - **15:57** : Test Execution - contract state
  - **16:49** : Test Execution - obligations
- **18:20** : Wrap-up

## What next?

Learn more about authoring Accord Project templates in template studio with the [Editing a Late Delivery Clause](tutorial-latedelivery) tutorial.

---
id: version-0.20-tutorial-create
title: Template Generator
original_id: tutorial-create

Now that you have executed an existing template, let's create a new template from scratch. To facilitate the creation of new templates, Cicero comes with a template generator.

## The template generator

### Install the generator

If you haven't already done so, first install the template generator::

```bash
npm install -g yo
npm install -g yo @accordproject/generator-cicero-template@0.20
```

### Run the generator:

You can now try the template generator by running the following command in a terminal window:
```bash
yo @accordproject/cicero-template
```
This will ask you a series of questions. Give your generator a name (no spaces) and then supply a namespace for your template model (again,no spaces). The generator will then create the files and directories required for a basic template (similar

to the helloworld template).

Here is an example of how it should look like in your terminal window:
```bash
bash-3.2$ yo @accordproject/cicero-template

     _-----_     ╭──────────────────────────╮
    |       |    │      Welcome to the      │
    |--(o)--|    │ generator-cicero-templat │
   `---------´   │       e generator!       │
    ( _´U`_ )    ╰──────────────────────────╯
    /___A___\   /
     |  ~  |
   __'.___.'__
 ´   `  |° ´ Y `

? What is the name of your template? mylease
? What is the namespace for your model? org.acme.lease
   create mylease/README.md
   create mylease/package.json
   create mylease/request.json
   create mylease/logic/logic.ergo
   create mylease/model/model.cto
   create mylease/test/logic_default.feature
   create mylease/text/grammar.tem.md
   create mylease/text/sample.md
   create mylease/.cucumber.js
   create mylease/.npmignore
```

:::tip
You may find it easier to edit the grammar, model and logic for your template in [VSCode](https://code.visualstudio.com/), installing the [Accord Project extension](https://marketplace.visualstudio.com/items?itemName=accordproject.cicero-vscode-extension). The extension gives you syntax highlighting and parser errors within VS Code.
:::

## Edit your template

### Update Sample.md

First, replace the contents of `./text/sample.md` with the legal text for the contract or clause that you would like to digitize.

Check that when you run `cicero parse` that the new `./text/sample.md` is now _invalid_ with respect to the grammar.

### Edit the Template Grammar

Now update the grammar in `./text/grammar.tem.md`. Start by replacing the existing grammar, making it identical to the contents of your updated `./text/sample.md`.

Now introduce variables into your template grammar as required. The variables are marked-up using `{{` and `}}` with what is between the braces being the name of your variable.

### Edit the Template Model

All of the variables referenced in your template grammar must exist in your template model. Edit
the file `model/model.cto` to include all your variables, making sure the name of the model property matches the name of the variable in the `./text/grammar.tem.md` file.

Note that the Concerto Modeling Language primitive data types are:

- `String`: for character strings
- `Long` or `Integer`: for integer values
- `DateTime`: for dates and times
- `Double`: for floating points numbers
- `Boolean`: for values that are either true or false

:::tip
Note that you can import common types (address, monetary amount, country code, etc.) from the Accord Project Model Repository: https://models.accordproject.org.
:::

### Edit the Transaction Types

Your template expects to receive data as input and will produce data as output. The structure of
this request/response data is captured in the `MyRequest` and `MyResponse` transaction types in your model
namespace. Open up the file `models/model.cto` and edit the definition of the `MyRequest` type to
include all the data you expect to receive from the outside world and that will be used by the
business logic of your template. Similarly edit the definition of the `MyResponse` type to include
all the data that the business logic for your template will compute and would like to return to the
caller.

### Edit the Template Logic

Now edit the business logic of the template itself. This is expressed in the Ergo language, which is a strongly-typed function domain specific language for contract logic. Open the file `logic/logic.ergo`
and edit the `helloworld` clause to perform the calculations your logic requires.

Looking at the Ergo logic for other example templates will help you understand the syntax and capabilities of Ergo.

## Publishing your template

If you would like to publish your new template in the Accord Project Template Library, please consult the [Template Library](tutorial-library) Section of this documentation.

---
id: version-0.20-tutorial-hyperledger
title: Deploying on Hyperledger Fabric
original_id: tutorial-hyperledger

## Hyperledger Fabric 1.3

Sample chaincode for Hyperledger Fabric that shows how to execute a Cicero template:
https://github.com/clauseHQ/fabric-samples/tree/master/chaincode/cicero

This sample shows how you can deploy and execute Smart Legal Contracts on-chain using Hyperledger Fabric v1.3.

Using this guide you can deploy a [Smart Legal Contract Template](https://templates.accordproject.org/) from the [Open Source Accord Project](https://www.accordproject.org/) to your HLF v1.3 blockchain. You can then submit transactions to the Smart Legal Contract, with the contract state persisted to the blockchain, and return values and emitted events propagated back to the client.

Before starting this tutorial, you are encouraged to review an [Introduction to the Accord Project](https://docs.google.com/presentation/d/1lYT-XlY-4UDtYR7Oc0X62nvT7AzOqYZ4QA8YVZasgy8/edit).

These instructions have been tested with:
- MacOS 10.14 / Ubuntu 18.04
- Recent release of Chrome and Safari
- Docker 18.09
- Docker Compose 1.23
- Node 8.10
- Git 2.17

If you're building a new environment yourself, we recommend using a cloud-hosted server (to save the conference WIFI!) as the installations can require large downloads.

A small number of hosted virtual machines are available from the workshop facilitators if you have trouble installing the prerequisites yourself.

## Installing Prerequisites

### Using Amazon Web Services

If you have your own AWS account, you can use the customised Ubuntu image. From your EC2 Dashboard, create a new instance and search for **Cicero** in the Community AMIs. The AMI is available in the Frankfurt and N. Virginia regions.

![Amazon Image](assets/advanced/hlf1.png)

The `t2.medium` instance type is sufficient for this tutorial.

You'll need also need to add an inbound rule to your Security Group to allow connections on port `3389`. This will allow you to make a remote desktop connection to your server.

![Amazon Image Port](assets/advanced/hlf2.png)

The default username and password for the prebuilt image is:
- Username: `guest`
- Password: `hyperledger2018`

> Connect to your image with a Remote Desktop Client, for example, Microsoft Remote Desktop on Mac and Windows.

### Building a Custom Docker Image

If you're feeling a bit more adventurous, you can build your own system. If you don't have the tools locally on your machine, start with an [Ubuntu Bionic 18.04 image from your favourite cloud provider](https://www.ubuntu.com/download/cloud).
You will need to be able to transfer files into your image, so if your server doesn't have a Desktop Manager and browser installed you'll need to find some other way to transfer files, e.g. via SCP or FTP, for example.
Once you've provisioned your server, install all of the required tools using the corresponding installation instructions:
- [Docker](https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-on-ubuntu-18-04)
- [Node](https://www.digitalocean.com/community/tutorials/how-to-install-node-js-on-ubuntu-18-04)
- [Docker Compose](https://www.digitalocean.com/community/tutorials/how-to-install-docker-compose-on-ubuntu-18-04)
- [Git](https://www.digitalocean.com/community/tutorials/how-to-install-git-on-ubuntu-18-04)
- [Fabric](https://hyperledger-fabric.readthedocs.io/en/release-1.3/getting_started.html)

## Create your Smart Legal Contract with Template Studio

Cicero Templates are the magic glue that binds your clever legal words with the logic that will run on your network. In this first step, we'll create a template in [Template Studio](https://studio.accordproject.org/).

Template Studio is a browser-based development environment for Cicero Templates. Your templates are only every stored in your browser (and are not shared with the Accord Project), so you should **Export** your work to save it for another time.

This tutorial uses the `supplyagreement-perishable-goods` template. This Smart Legal Agreement combines a plaintext contract for the shipment of goods that impose conditions on temperature and humidity until the shipment is delivered.

We simulate the submission of IoT events from sensors that get sent to the contract in a supply blockchain network. The contract determines the obligations and actions of the parties according to its logic and legal text.

Once you've connected to your system, open https://studio.accordproject.org in a new browser window.

:::note
In the hosted images, Mozilla Firefox is preinstalled, click the icon on the top-left toolbar to launch it.
:::

The Template Studio allows you to load sample templates for smart legal agreements from the [Accord Project template library](https://templates.accordproject.org/).

> In the Template Studio search bar, type **supplyagreement-perishable-goods**. Select the 0.12.1 version.

Explore the source components of the template.

- Contract Text & Grammar
- Model
- Logic

- Test Execution

Note the placeholders in the Template Grammar (found under **Contract Text** ->
Template), and the corresponding data model definition in `contract.cto` (found
under **Model**).

The logic definition in Ergo defines the behaviour of the contract in response to
Requests. The logic also reference the same contract variables, through the
`contract` keyword.

You can simulate the performance of the contract using the **Text Execution** page
(click **Logic** first). Clicking **Send Request** will trigger the clause and
produce a response that indicates the total price due, along with any penalties.

> Change some of the values in the Test Contract, for example increase the lower
limit for temperature readings from 2°C to 3°C. If you reset the Test Execution and
send the same request again you should notice a penalty in the response.

![Change Test Contract](assets/advanced/hlf3.png)

The _Obligations_ that are emitted by the contract are configured to be emitted as
Events in Fabric. This allows any party that is involved in the contract to perform
an action automatically, for example to add a payment obligation to an invoice.

> Click **Export** to download your template

Save your CTA (Cicero Template Archive) file somewhere safe, as you'll need to use
it in a later step. We suggest saving the file in user's the home folder.

> Create a `request.json` file with the contents of the **Request** box from the
Logic -> **Test Execution** page in Template Studio.

For example:
```json
{
    "$class": "org.accordproject.perishablegoods.ShipmentReceived",
    "unitCount": 3002,
    "shipment": {
      "$class": "org.accordproject.perishablegoods.Shipment",
      "shipmentId": "SHIP_001",
      "sensorReadings": [
        {
          "$class": "org.accordproject.perishablegoods.SensorReading",
          "centigrade": 2,
          "humidity": 80,
          "shipment":
"resource:org.accordproject.perishablegoods.Shipment#SHIP_001",
          "transactionId": "a"
        }
      ]
    }
  }
```

Finally, create a `contract.txt` file with the contents of the Test Contract box
from the Contract Text page in Template Studio.

:::note
It's important that your sample contract text exactly matches your grammar's

structure, this includes trailing spaces and line breaks. To be sure that you copy everything, right click the window and choose Select All, before choosing Copy.

You'll be notified if there are errors in your contract text during the next step by messages such as:
```md
Unexpected "\n"
```
:::

## Provision your Hyperledger Fabric instance

In this step, we will provision a test Hyperledger Fabric network on your machine.

:::note
If you're not using one of the prebuilt images you'll also need run the following command to download the tutorial resources from GitHub.

```sh
git clone https://github.com/clauseHQ/fabric-samples
cd fabric-samples
git checkout master
```
:::

> Open a Terminal window and type the following commands. In the hosted image there is a link start a terminal window on the desktop.
```sh
cd fabric-samples/cicero
```

Next we'll download Fabric and start the docker containers. If you're doing this for the first time, go and get a coffee. This will usually take several minutes.

> In your Terminal, type the following command to download and start Fabric.
```sh
./startFabric.sh
```

> Next install the node dependencies for the client code, with this command. This step can also take a few minutes.
```sh
npm install
```

> You can then enroll the administrator into the network, and register a user that we'll use in later scripts.
```sh
node enrollAdmin.js && node registerUser.js
```

## Deploy your contract to your network

> Using the files that you downloaded earlier, run the `deploy.js` script.

The example below assumes that all of the files are located in the same folder as `deploy.js`.

:::note

The order of the parameters to the `deploy.js` script is important, please follow the pattern shown in the example. You'll also need to give the relative or absolute path to the `sample.txt` file, for example if you saved the file in your home directory, replace `sample.txt` with `../../sample.txt`.
:::

```sh
node deploy.js supplyagreement-perishable-goods.cta sample.txt
```

If deployment of your contract is successful, you should see the following output:
```sh
Transaction proposal was good
Response payload: Successfully deployed contract MYCONTRACT based on
supplyagreement-perishable-goods@0.9.0
Successfully sent Proposal and received ProposalResponse: Status - 200, message -
""
The transaction has been committed on peer localhost:7051
Send transaction promise and event listener promise have completed
Successfully sent transaction to the orderer.
Successfully committed the change to the ledger by the peer
```

Finally, you can trigger your Smart Legal Agreement by sending requests to it. The `submitRequest.js` script is configured to route your request to your deployed contract.

> Run the `submitRequest.js` script using your `request.json` file, by typing the following command into the terminal.
```sh
node submitRequest.js request.json
```
If the invocation is successful, you should see the following output:
```sh
Assigning transaction_id:
0788d105901c9f12316bb84dc1c5345be6fe96edf626d427de9871cefac4f063
Transaction proposal was good
Response payload:
{"$class":"org.accordproject.perishablegoods.PriceCalculation","totalPrice":
{"$class":"org.accordproject.money.MonetaryAmount","doubleValue":4503,"currencyCode
":"USD"},"penalty":
{"$class":"org.accordproject.money.MonetaryAmount","doubleValue":0,"currencyCode":"
USD"},"late":false,"shipment":"resource:org.accordproject.perishablegoods.Shipment#
SHIP_001","transactionId":"3a30f4b7-7537-4c2e-8186-49ce7e95681d","timestamp":"2018-
12-01T17:49:26.100Z"}
Successfully sent Proposal and received ProposalResponse: Status - 200, message -
""
The transaction has been committed on peer localhost:7051
Send transaction promise and event listener promise have completed
Successfully sent transaction to the orderer.
Successfully committed the change to the ledger by the peer
```

Congratulations you now have a legal agreement running on your blockchain network!

Depending on the values in your request, the response payload will indicate whether a penalty is due.

Because this contract is currently stateless, i.e. it doesn't store any data on-chain, you can submit multiple requests with different values to simulate the behaviour of the contract under different circumstances.

:::tip
Can you cause a breach due to out-of-range readings?
:::

## Extension Tasks

We gave you lots of help in the first few steps, but to learn properly, we always find that it helps to try things for yourself. Here are a few suggestions that will help you to understand what is going on better.

1. Currently, `the supplyagreement-perishable-goods` template doesn't store any data on the chain. Modify the template to store sensor readings. The readings should be looked up from the state object in your logic rather than from your request when the shipment is accepted.
:::note
Take a look at some of the other stateful Cicero Templates to see what changes you will need to make. `installment-sale` and `helloworldstate` are good examples.
If you get really stuck, a solution is available for you to [download](https://drive.google.com/file/d/1cak_P_x01w8dz43aZX8N16G5DUNp6VbG/view?usp=sharing).
:::
2. Explore the source code of the Cicero chaincode shim that transforms your requests and deployments into Fabric transactions using the Fabric Node SDK.
https://github.com/clauseHQ/fabric-samples/tree/master/chaincode/cicero
3. Create your own template from scratch using [Template Studio](https://studio.accordproject.org/), or download the [VSCode plugin](https://marketplace.visualstudio.com/items?itemName=accordproject.accordproject-vscode-plugin).
![Change Test Contract](assets/advanced/hlf4.png)
> A separate tutorial for creating a template using the Cicero CLI tool can be found in [Creating a New Template](basic-create).

4. This template also emits Obligations as Fabric events as well as returning a response to the client. Modify the Cicero chaincode to display the events do something interesting with them. How would you notify the parties that a penalty is due?

:::warning
If you would like to make changes to the cicero chaincode be aware that Docker caches the docker image for the chaincode. If you edit the source and run `./startFabric` you will _not_ see your changes.
For your code changes to take effect you need to `docker stop` the peer (use `docker ps` to get the container id) and then `docker rmi -f` your docker chaincode image. The image name should look something like `dev-peer0.org1.example.com-cicero-2.0-598263b3afa0267a29243ec2ab8d19b7d2016ac628f13641ed1822c4241c5734`
:::
5. Deploy the contract to a Fabric network that includes multiple users and nodes.

## Hyperledger Composer

A separate sample showing how to integrate Cicero with Hyperledger Composer is available here:
https://github.com/accordproject/cicero-perishable-network

---
id: version-0.20-tutorial-latedelivery
title: Editing a Late Delivery Clause
original_id: tutorial-latedelivery

This tutorial will walk you through the steps of authoring a clause template in
[Template Studio](https://studio.accordproject.org/).

We start with a very simple _Late Penalty and Delivery_ Clause and gradually make
it more complex, adding both legal text to it and the corresponding business logic
in Ergo.

## Initial Late Delivery Clause

### Load the Template

To get started, head to the `minilatedeliveryandpenalty` template in the Accord
Project Template Library at [Mini Late Delivery And
Penalty](https://templates.accordproject.org/minilatedeliveryandpenalty@0.4.0.html)
and click the "Open In Template Studio" button.

![Advanced-Late-1](assets/advanced/late1.png)

Begin by inspecting the `README` and `package.json` tabs within the `Metadata`
section. Feel free to change the name of the template to one you like.

### The Contract Text

Then click on the `Text` Section on the left, which should show a `Grammar` tab,
for the the natural language of the template.

![Advanced-Late-2](assets/advanced/late2.png)

When the text in the `Grammar` tab is in sync with the text in the `Test Sample`
tab, this means the sample is a valid with respect to the grammar, and data is
extracted, showing in `Contract Data` tab. The contract data is represented using
the JSON format and contains the value of the variables declared in the contract
template. For instance, the value for the `buyer` variable is `Betty Buyer`,
highlighted in red:

![Advanced-Late-3](assets/advanced/late3.png)

Changes to the variables in the `Test Sample` are reflected in the `Contract Data`
tab in real time, and vice versa. For instance, change `Betty Buyer` to a different
name in the contract text to see the `partyId` change in the contract data.

If you edit part of the text which is not a variable in the template, this results
in an error when parsing the `Test Sample`. The error will be shown in red in the
status bar at the bottom of the page. For instance, the following image shows the
parsing error obtained when changing the word `delayed` to the word `timely` in the
contract text.

![Advanced-Late-4](assets/advanced/late4.png)

This is because the `Test Sample` relies on the `Grammar` text as a source of
truth. This mechanism ensures that the actual contract always reflects the
template, and remains faithful to the original legal text. You can, however, edit
the `Grammar` itself to change the legal text.

Revert your changes, changing the word `timely` back to the original word `delayed` and the parsing error will disappear.

### The Model

Moving along to the `Model` section, you will find the data model for the template variables (the `MiniLateDeliveryClause` type), as well as for the requests (the `LateRequest` type) and response (the `LateResponse` type) for the late delivery and penalty clause.

![Advanced-Late-5](assets/advanced/late5.png)

Note that a `namespace` is declared at the beginning of the file for the model, and that several existing models are being imported (using e.g., `import org.accordproject.cicero.contract.*`). Those imports are needed to access the definition for several types used in the model:
- `AccordClause` which is a generic type for all Accord Project clause templates, and is defined in the `org.accordproject.contract` namespace;
- `Request` and `Response` which are generic types for responses and requests, and are defined in the `org.accordproject.runtime` namespace;
- `Duration` which is defined in the `org.accordproject.time` namespace.

### The Logic

The final part of the template is the `Ergo` tab of the `Logic` section, which describes the business logic.

![Advanced-Late-6](assets/advanced/late6.png)

Thanks to the `namespace` at the beginning of this file, the Ergo engine can know the definition for the `MiniLateDeliveryClause`, as well as the `LateRequest`, and `LateResponse` types defined in the `Model` tab.

To test the template execution, go to the `Test Request` tab in the `Logic` section. It should be already populated with a valid `Request`. Press the `Send Request` button to trigger the clause.

![Advanced-Late-7](assets/advanced/late7.png)

Since the value of the `deliveredAt` parameter in the request is after the value of the `agreedDelivery` parameter in the request, this should return a new response which includes the calculated penalty.

Changing the date for the `deliveredAt` parameter in the request will result in a different penalty.

![Advanced-Late-8](assets/advanced/late8.png)

Note that the clause will return an error if it is called for a timely delivery.

![Advanced-Late-9](assets/advanced/late9.png)

## Add a Penalty Cap

We can now start building a more advanced clause. Let us first take a moment to notice that there is no limitation to the penalty resulting from a late delivery. Under `Test Execution` in `Logic`, send this request:
```json
```

~~~
{
  "$class": "org.accordproject.minilatedeliveryandpenalty.LateRequest",
  "agreedDelivery": "2019-04-10T12:00:00-05:00",
  "deliveredAt": "2019-04-20T03:24:00-05:00",
  "goodsValue": 200
}
```
The penalty should be rather low. Now send this other request:
```json
{
  "$class": "org.accordproject.minilatedeliveryandpenalty.LateRequest",
  "agreedDelivery": "2005-04-01T12:00:00-05:00",
  "deliveredAt": "2019-04-20T03:24:00-05:00",
  "goodsValue": 200
}
```
Notice that the penalty is now quite a large value. It is not unusual to cap a
penalty to a maximum amount. Let us now look at how to change the template to add
such a cap based on a percentage of the total value of the delivered goods.

### Update the Legal Text

To implement this, we first go to the `Template` tab and add a sentence indicating:
`The total amount of penalty shall not, however, exceed {{capPercentage}}% of the
total value of the delayed goods.`

For convenience, you can copy-paste the new template text from here:
```tem
Late Delivery and Penalty.

In case of delayed delivery of Goods, {{seller}} shall pay to
{{buyer}} a penalty amounting to {{penaltyPercentage}}% of the total
value of the Goods for every {{penaltyDuration}} of delay. The total
amount of penalty shall not, however, exceed {{capPercentage}}% of the
total value of the delayed goods. If the delay is more than
{{maximumDelay}}, the Buyer is entitled to terminate this Contract.

```
This should immediately result in an error when parsing the contract text:

![Advanced-Late-10](assets/advanced/late10.png)

As explained in the error message, this is because the new template text uses a
variable `capPercentage` which has not been declared in the model.

### Update the Model

To define this new variable, go to the `Model` tab, and change the
`MiniLateDeliveryClause` type to include `o Double capPercentage`.

![Advanced-Late-11](assets/advanced/late10b.png)

For convenience, you can copy-paste the new `MiniLateDeliveryClause` type from
here:
```ergo
asset MiniLateDeliveryClause extends AccordClause {
  o AccordParty buyer         // Party to the contract (buyer)
  o AccordParty seller        // Party to the contract (seller)
  o Duration penaltyDuration  // Length of time resulting in penalty
~~~

  o Double penaltyPercentage // Penalty percentage
  o Double capPercentage   // Maximum penalty percentage
  o Duration maximumDelay   // Maximum delay before termination
}

This results in a new error, this time on the test contract:

![Advanced-Late-11](assets/advanced/late11.png)

To fix it, we need to add that same line we added to the template, replacing the `capPercentage` by a value in the `Test Contract`: `The total amount of penalty shall not, however, exceed 52% of the total value of the delayed goods.`

For convenience, you can copy-paste the new test contract from here:
```md
Late Delivery and Penalty.

In case of delayed delivery of Goods, "Steve Seller" shall pay to
"Betty Buyer" a penalty amounting to 10.5% of the total
value of the Goods for every 2 days of delay. The total
amount of penalty shall not, however, exceed 52% of the
total value of the delayed goods. If the delay is more than
15 days, the Buyer is entitled to terminate this Contract.

```

Great, now the edited template should have no more errors, and the contract data should now include the value for the new `capPercentage` variable.

![Advanced-Late-12](assets/advanced/late12.png)

Note that the `Current Template` Tab indicates that the template has been changed.

### Update the Logic

At this point, executing the logic will still result in large penalties. This is because the logic does not take advantage of the new `capPercentage` variable. Edit the `logic.ergo` code to do so. After step `// 2. Penalty formula` in the logic, apply the penalty cap by adding some logic as follows:
```ergo
    // 3. Capped Penalty
    let cap = contract.capPercentage / 100.0 * request.goodsValue;

    let cappedPenalty =
      if penalty > cap
      then cap
      else penalty;

```
Do not forget to also change the value of the penalty in the returned `LateResponse` to use the new variable `cappedPenalty`:
```ergo
    // 5. Return the response
    return LateResponse{
      penalty: cappedPenalty,
      buyerMayTerminate: termination
    }
```

The logic should now look as follows:

![Advanced-Late-13](assets/advanced/late13.png)

### Run the new Logic

As a final test of the new template, you should try again to run the contract with a long delay in delivery. This should now result in a much smaller penalty, which is capped to 52% of the total value of the goods, or 104 USD.

![Advanced-Late-14](assets/advanced/late14.png)

:::tip
A full version of the template after those changes have been applied can be found as the [Mini Late Delivery And Penalty Capped](https://templates.accordproject.org/minilatedeliveryandpenalty-capped@0.4.0.html) in the Template Library.
:::

## Emit a Payment Obligation.

As a final extension to this template, we can modify it to emit a Payment Obligation. This first requires us to switch from a Clause template to a Contract template.

### Switch to a Contract Template

The first place to change is in the metadata for the template. This can be done easily with the `full contract` button in the `Current Template` tab. This will immediately result in an error indicating that the model does not contain an `AccordContract` type.

![Advanced-Late-15](assets/advanced/late15.png)

### Update the Model

To fix this, change the model to reflect that we are now editing a contract template, and change the type `AccordClause` to `AccordContract` in the type definition for the template variables:
```ergo
asset MiniLateDeliveryContract extends AccordContract {
  o AccordParty buyer         // Party to the contract (buyer)
  o AccordParty seller        // Party to the contract (seller)
  o Duration penaltyDuration  // Length of time resulting in penalty
  o Double penaltyPercentage  // Penalty percentage
  o Double capPercentage      // Maximum penalty percentage
  o Duration maximumDelay     // Maximum delay before termination
}
```

The next error is in the logic, since it still uses the old `MiniLateDeliveryClause` type which does not exist anymore.

### Update the Logic

The `Logic` error that occurs here is:
```bash
Compilation error (at file lib/logic.ergo line 19 col 31). Cannot find type with name 'MiniLateDeliveryClause'
```

~~~
contract MiniLateDelivery over MiniLateDeliveryClause {
                               ^^^^^^^^^^^^^^^^^^^^^^
```
Update the logic to use the the new `MiniLateDeliveryContract` type instead, as
follows:
```ergo
contract MiniLateDelivery over MiniLateDeliveryContract {
```

The template should now be without errors.

### Add a Payment Obligation

Our final task is to emit a `PaymentObligation` to indicate that the buyer should
pay the seller in the amount of the calculated penalty.

To do so, first import a couple of standard models: for the Cicero's [runtime
model](https://models.accordproject.org/cicero/runtime.html) (which contains the
definition of a `PaymentObligation`), and for the Accord Project's [money model]
(https://models.accordproject.org/money.html) (which contains the definition of a
`MonetaryAmount`). The `import` statements at the top of your logic should look as
follows:
```ergo
import org.accordproject.time.*
import org.accordproject.cicero.runtime.*
import org.accordproject.money.MonetaryAmount

```

Lastly, add a new step between steps `// 4.` and `// 5.` in the logic to emit a
payment obligation in USD:
```ergo
    emit PaymentObligation{
      contract: contract,
      promisor: some(contract.seller),
      promisee: some(contract.buyer),
      deadline: none,
      amount: MonetaryAmount{ doubleValue: cappedPenalty, currencyCode: "USD" },
      description: contract.seller.partyId ++ " should pay penalty amount to " ++
contract.buyer.partyId
    };

```
That's it! You can observe in the `Test Execution` that an `Obligation` is now
being emitted. Try out adjusting values and continuing to send requests and getting
responses and obligations.

![Advanced-Late-16](assets/advanced/late16.png)

--------------------------------------------------------------------------------
---
id: version-0.20-tutorial-library
title: Template Library
original_id: tutorial-library
---

This tutorial explains how to get access, and contribute, to all of the public
templates available as part of the the [Accord Project Template
Library](https://templates.accordproject.org).
~~~

## Setting up

### Prerequisites

Accord Project uses [GitHub](https://github.com/) to maintain its open source template library. For this tutorial, you must first obtain and configure the following dependency:

* [Git](https://git-scm.com): a distributed version-control system for
  tracking changes in source code during software development.
* [Lerna](https://lerna.js.org/): A tool for managing JavaScript projects with multiple packages. You can install lerna by running the following command in your terminal:

```bash
npm install -g lerna@^3.15.0
```

### Clone the template library

Once you have `git` installed on your machine, you can run `git clone` to create a version of all the templates:

```bash
git clone https://github.com/accordproject/cicero-template-library
```

Alternatively, you can download the library directly by visiting the [GitHub Repository for the Template Library](https://github.com/accordproject/cicero-template-library) and use the "Download" button as shown on this snapshot:

![Basic-Library-1](/docs/assets/basic/library1.png)

### Install the Library

Once cloned, you can set up the library for development by running the following commands inside your template library directory:

```bash
lerna bootstrap
```

### Running all the template tests

To check that the installation was successful, you can run all the tests for all the Accord Project templates by running:

```bash
lerna run test
```

## Structure of the Repository

You can see the source code for all public Accord Project templates by looking inside the `./src` directory:

```sh
bash-3.2$ ls src
```

acceptance-of-delivery
car-rental-tr
certificate-of-incorporation
copyright-license
demandforecast
docusign-connect
docusign-po-failure
eat-apples
empty
empty-contract
fixed-interests
...
```

Each of those templates directories have the same structure, as described in the [Templates Deep Dive](tutorial-templates) Section. For instance for the `acceptance-of-delivery` template:
```
$ cd src/acceptance-of-delivery
$ bash-3.2$ ls -laR
./README.md
./package.json

./text:
  ./grammar.tem.md
  ./sample.md

./logic:
   logic.ergo

./model:
   model.cto

./test:
  logic.feature
  logic_default.feature

./request.json
./state.json

## Use a Template

To use a template, simply run the same Cicero commands we have seen in the previous tutorials. For instance, to extract the deal data from the `./text/sample.md` text sample for the `acceptance-of-delivery` template, run:

```bash
cicero parse --template ./src/acceptance-of-delivery
```
You should see a response as follows:
```json
{
  "$class": "org.accordproject.acceptanceofdelivery.AcceptanceOfDeliveryClause",
  "clauseId": "9ed9d255-3bb6-4928-be7b-c6305e083246",
  "shipper": "Party A",
  "receiver": "Party B",
  "deliverable": "Widgets",
  "businessDays": 10,
```

~~~
  "attachment": "Attachment X"
}
```

Or, to extract the deal data from the `./text/sample.md` then send the default
request in `./request.json` for the `latedeliveryandpenalty` template, run:
```bash
cicero trigger --template ./src/latedeliveryandpenalty
```
You should see a response as follows:

```json
{
  "clause": "latedeliveryandpenalty@0.15.0-
988570f09c5da08526a2c0a3bf9a5b6226c6265c267a60be62fdeaeb44661ee3",
  "request": {
    "$class":
"org.accordproject.latedeliveryandpenalty.LateDeliveryAndPenaltyRequest",
    "forceMajeure": false,
    "agreedDelivery": "2017-12-17T03:24:00-05:00",
    "deliveredAt": null,
    "goodsValue": 200
  },
  "response": {
    "$class":
"org.accordproject.latedeliveryandpenalty.LateDeliveryAndPenaltyResponse",
    "penalty": 110.00000000000001,
    "buyerMayTerminate": true,
    "transactionId": "1e285c3f-0394-4543-aa67-c324d9ad5b6f",
    "timestamp": "2019-11-04T00:05:43.923Z"
  },
  "state": {
    "$class": "org.accordproject.cicero.contract.AccordContractState",
    "stateId": "org.accordproject.cicero.contract.AccordContractState#1"
  },
  "emit": [
    {
      "$class": "org.accordproject.cicero.runtime.PaymentObligation",
      "amount": {
        "$class": "org.accordproject.money.MonetaryAmount",
        "doubleValue": 110.00000000000001,
        "currencyCode": "USD"
      },
      "description": "Dan should pay penalty amount to Steve",
      "contract":
"resource:org.accordproject.latedeliveryandpenalty.LateDeliveryAndPenaltyContract#4
be4de30-0aeb-47c6-8791-3a64f4491437",
      "promisor": "resource:org.accordproject.cicero.contract.AccordParty#Dan",
      "promisee": "resource:org.accordproject.cicero.contract.AccordParty#Steve",
      "eventId": "valid",
      "timestamp": "2019-11-04T00:05:43.925Z"
    }
  ]
}
```

## Contribute a New Template

To contribute a change to the Accord Project library, please
~~~

[fork](https://help.github.com/en/github/getting-started-with-github/fork-a-repo)
the repository and then create a [pull
request](https://help.github.com/en/github/collaborating-with-issues-and-pull-
requests/about-pull-requests).

Note that templates should have unit tests. See any of the `./test` directories in
the templates contained in the template library for an examples with unit tests, or
consult the [Testing Reference](ref-testing) Section of this documentation.

---
id: version-0.20-tutorial-nodejs
title: Working with Node.js
original_id: tutorial-nodejs

## Cicero Node.js API

You can work with Accord Project templates directly in JavaScript using Node.js.

Documentation for the API can be found in [Cicero
API](https://docs.accordproject.org/docs/cicero-api.html).

## Working with Templates

### Import the Template class

To import the Cicero class for templates:

```js
const Template = require('@accordproject/cicero-core').Template;
```

### Load a Template

To create a Template instance in memory call the `fromDirectory`, `fromArchive` or
`fromUrl` methods:

```js
    const template = await
Template.fromDirectory('./test/data/latedeliveryandpenalty');
```

These methods are asynchronous and return a `Promise`, so you should use `await` to
wait for the promise to be resolved.

### Instantiate a Template

Once a Template has been loaded, you can create a Clause based on the Template. You
can either instantiate
the Clause using source DSL text (by calling `parse`), or you can set an instance
of the template model
as JSON data (by calling `setData`):

```js
    // load the DSL text for the template
    const testLatePenaltyInput = fs.readFileSync(path.resolve(__dirname, 'text/',
'sample.md'), 'utf8');
```

    const clause = new Clause(template);
    clause.parse(testLatePenaltyInput);

    // get the JSON object created from the parse
    const data = clause.getData();

OR - create a contract and set the data from a JSON object.

```js
    const clause = new Clause(template);
    clause.setData( {$class: 'org.acme.MyTemplateModel', 'foo': 42 } );
```

## Executing a Template Instance

Once you have instantiated a clause or contract instance, you can execute it.

### Import the Engine class

To execute a Clause you first need to create an instance of the ``Engine`` class:

```js
const Engine = require('@accordproject/cicero-engine').Engine;
```

### Send a request to the contract

You can then call ``execute`` on it, passing in the clause or contract instance,
and the request:

```js
    const request = {
        '$class':
'org.accordproject.latedeliveryandpenalty.LateDeliveryAndPenaltyRequest',
        'forceMajeure': false,
        'agreedDelivery': '2017-10-07T16:38:01.412Z',
        'goodsValue': 200,
        'transactionId': '402c8f50-9e61-433e-a7c1-afe61c06ef00',
        'timestamp': '2017-11-12T17:38:01.412Z'
    };
    const state = {};
    state.$class = 'org.accordproject.cicero.contract.AccordContractState';
    state.stateId = 'org.accordproject.cicero.contract.AccordContractState#1';
    const result = await engine.execute(clause, request, state);
```

---
id: version-0.20-tutorial-templates
title: Templates Deep Dive
original_id: tutorial-templates

In the [Getting Started](started-hello) section, we learned how to use the existing
[helloworld@0.12.0.cta](https://templates.accordproject.org/archives/
helloworld@0.12.0.cta) template archive. Here we take a look inside that archive to

understand the structure of Accord Project templates.

## Unpack a Template Archive

A `.cta` archive is nothing more than a zip file containing the components of a template. Let's unzip that archive to see what is inside. First, create a directory in the place where you have downloaded that archive, then run the unzip command in a terminal:

```bash
$ mkdir helloworld
$ mv helloworld@0.12.0.cta helloworld
$ cd helloworld
$ unzip helloworld@0.12.0.cta
Archive:  helloworld@0.12.0.cta
 extracting: package.json
   creating: text/
 extracting: text/grammar.tem.md
 extracting: README.md
 extracting: text/sample.md
 extracting: request.json
   creating: model/
 extracting: model/@models.accordproject.org.cicero.contract.cto
 extracting: model/@models.accordproject.org.cicero.runtime.cto
 extracting: model/@models.accordproject.org.money.cto
 extracting: model/model.cto
   creating: logic/
 extracting: logic/logic.ergo
```

## Template Components

Once you have unziped the archive, the directory should contain the following files and sub-directories:

```text
package.json
    Metadata for the template (name, version, description etc)

README.md
    A markdown file that describes the purpose and correct usage for the template

text/grammar.tem.md
    The default grammar for the template

text/sample.md
    A sample clause or contract text that is valid for the template

model/
    A collection of Concerto model files for the template. They define the Template
Model
    and models for the State, Request, Response, and Obligations used during
execution.

logic/
    A collection of Ergo files that implement the business logic for the template

test/
    A collection of unit tests for the template
```

state.json (optional)
    A sample valid state for the clause or contract

request.json (optional)
    A sample valid request to trigger execution for the template

In a nutshell, the template archive contains the three main components of the
[Template Triangle](accordproject-concepts#what-is-a-template) in the corresponding
directories (the natural language text of your clause or contract in the `text`
directory, the data model in the `model` directory, and the contract logic in the
`logic` directory). Additional files include metadata and samples which can be used
to illustrate or test the template.

Let us look at each of those components.

### Template Text

#### Grammar

The file in `text/grammar.tem.md` contains the grammar for the template. It is
natural language, with markup to indicate the variable(s) in your Clause or
Contract.

```tem
Name of the person to greet: {{name}}.
Thank you!
```

In the `helloworld` template there is only one variable `name` which is indicated
between `{{` and `}}`.

#### Sample Text

The file in `text/sample.md` contains a sample valid for that grammar.

```md
Name of the person to greet: "Fred Blogs".
Thank you!
```

### Template Model

The file in `model/model.cto` contains the data model for the template. This
includes a description for each of the template variables, including what kind of
variable it is (also called their [type](ref-glossary.html#components-of-data-
models)).

Here is the model for the `helloworld` template:

```ergo
namespace org.accordproject.helloworld

import org.accordproject.cicero.contract.* from
https://models.accordproject.org/cicero/contract.cto
import org.accordproject.cicero.runtime.* from
https://models.accordproject.org/cicero/runtime.cto
```

~~~
asset TemplateModel extends AccordClause {
  o String name // variable 'name' is of type String
}

transaction MyRequest extends Request {
  o String input
}

transaction MyResponse extends Response {
  o String output
}
```

The `TemplateModel` as well as the `Request` and `Response` are types which are
specified using the [Concerto modeling
language](https://github.com/accordproject/concerto).

The `TemplateModel` indicate that the template is for a Clause, and should have a
variable `name` of type `String` (i.e., text).

```ergo
asset TemplateModel extends AccordClause {
  o String name // variable 'name' is of type String
}
```

Types are always declared within a namespace (here `org.accordproject.helloworld`),
which provides a mechanism to disambiguate those types amongst multiple model
files.

### Template Logic

The file in `logic/logic.ergo` contains the executable logic. Each Ergo file is
identified by a namespace, and contains declarations (e.g., constants, functions,
contracts). Here is the Ergo logic for the `helloworld` template:

```ergo
namespace org.accordproject.helloworld

contract HelloWorld over TemplateModel {
  // Simple Clause
  clause greet(request : MyRequest) : MyResponse {
    return MyResponse{ output: "Hello " ++ contract.name ++ " " ++ request.input }
  }
}
```

This declares a single `HelloWorld` contract in the `org.accordproject.helloworld`
namespace, with one `greet` clause.

It also declares that this contract `HelloWorld` is parameterized over the given
`TemplateModel` found in the `models/model.cto` file.

The `greet` clause takes a request of type `MyRequest` as input and returns a
response of type `MyResponse`.

The code for the `greet` clause returns a new `MyResponse` response with a single
property `output` which is a string. That string is constructed using the string
concatenation operator (`++`) in Ergo from the `name` in the contract
~~~

(`contract.name`) and the input from the request (`request.input`).

## Use the Template

Even after you have unzipped the template archive, you can use that template from the directory directly, in the same way we did from the `.cta` archive in the [Getting Started](started-hello) section.

For instance you can use `cicero parse` or `cicero trigger` as follows:
```bash
$ cd helloworld
$ cicero parse
15:35:12 - info: Using current directory as template folder
15:35:12 - info: Loading a default text/sample.md file.
15:35:14 - info:
{
  "$class": "org.accordproject.helloworld.HelloWorldClause",
  "clauseId": "7258ecf6-cf64-4f9b-807d-c4a3ae6b83ed",
  "name": "Fred Blogs"
}
$ cicero trigger
15:35:17 - info: Using current directory as template folder
15:35:17 - info: Loading a default text/sample.md file.
15:35:17 - info: Loading a default request.json file.
15:35:19 - warn: A state file was not provided, initializing state. Try the --state
flag or create a state.json in the root folder of your template.
15:35:19 - info:
{
  "clause": "helloworld@0.12.0-
13f7230894084cc568853771a6b5c928a1a3b71699512f763f8734fcca38dc5c",
  "request": {
    "$class": "org.accordproject.helloworld.MyRequest",
    "input": "Accord Project"
  },
  "response": {
    "$class": "org.accordproject.helloworld.MyResponse",
    "output": "Hello Fred Blogs Accord Project",
    "transactionId": "c65d3161-eb77-4d52-8abb-6953a664d190",
    "timestamp": "2019-11-03T20:35:19.526Z"
  },
  "state": {
    "$class": "org.accordproject.cicero.contract.AccordContractState",
    "stateId": "org.accordproject.cicero.contract.AccordContractState#1"
  },
  "emit": []
}
```

:::note
Remark that if your template directory contains a valid `sample.md` or valid `request.json`, Cicero will automatically detect those so you do not need to pass them using the `--sample` or `--request` options.
:::

---
id: version-0.21-accordproject-faq
title: FAQ
original_id: accordproject-faq

## Accord Project Frequently asked Questions

### What is a "Smart Contract" in the Accord Project?

A “smart” legal contract is a legally binding agreement that is digital and able to connect its terms and the performance of its obligations to external sources of data and software systems. The benefit is to enable a wide variety of efficiencies, automation, and real time visibility for lawyers, businesses, nonprofits, and government. The potential applications of smart legal contracts are limitless. Although the operation of smart legal contracts may be enhanced by using blockchain technology, a blockchain is not necessary, smart legal contracts can operate using traditional software systems without blockchain. A central goal of Accord Project technology is to be blockchain agnostic.

A smart legal contract consists of natural language text with certain parts (e.g. clauses, sections) of the agreement constructed as machine executable components. The libraries provided by the Accord Project enable a document to be:

* Structured as machine readable data objects; and
* Executed on, or integrated with, external systems (e.g. to initiate a payment or update an invoice)

While the Accord Project technology is targeted at the development of smart legal contracts, the open source codebase may also be used to develop other forms of machine-readable and executable documentation.

### How is an Accord Project "Smart Contract" different from "Smart Contracts" on the blockchain?

Accord Project Smart legal contracts should not be confused with so-called blockchain “smart contracts”, which are scripts that necesarily operate on a blockchain. On the blockchain a smart contract is often written in a specific language like solidity that executes and operates on the blockchain. It lives in a closed world. An Accord Project Smart Contract contains text based template that integrates with a data model and the Ergo language. The three components are integrated into a whole. Using Ergo an Accord Project Smart contract can communicate with other systems, it can send and receive data, it can perform calculations and it can interact with a blockchain.

### What benefits do Smart Legal Contracts provide?

Contractual agreements sit at the heart of any organization, governing relationships with employees, shareholders, customers, suppliers, financiers, and more. Yet contracts today are not capable of being efficiently managed as the valuable assets they are. Currently contracts exist as static text documents stored in cloud storage services, dated contract management systems, or even email inboxes. Often these documents are Word files or PDFs that can only be interpreted by humans. A smart legal contract, by contrast, can be interpreted by machines.

Smart Legal Contracts can be easily searched, analyzed, queried, and understood. By associating a data model to a contract, it is possible to extract a host of valuable data about a contract or draft a series of contracts from existing data points (i.e., variables and their values).

The data model is used to ensure that all of the necessary data is present in the contract, and that this data is valid. In addition, it provides the necessary structure to enable contracts to "come alive" by adding executable logic.

The result is a contract that is:

* Searchable
* Analyzable
* Real-time
* Integrated

Consequently, contracts are transformed from business liabilities in constant need of management to assets capable of providing real business intelligence and value. A Smart Contract contains a data model so that the data is part of the contract and not something held in an external system. The logical operations of the contract are also part of the contract. The contract can update itself and react to the outside world. Rather than being stored in filing cabinet it is a living breathing process.

### What is the Accord Project and what is its purpose?

The Accord Project is a non-profit, member-driven organization that builds open source code and documentation for smart legal contracts for use by transactional attorneys, business and finance professionals, and other contract users. Open source means that anyone can use and contribute to the code and documentation and use it in their own software applications and systems free of charge.

The purpose of the Accord Project is to establish and maintain a common and consistent legal and technical foundation for smart legal contracts. The Accord Project is organized into working groups focused on various use cases for Smart Contracts. The specific working groups are assisted by the Technology Working Group, which builds the underlying open source code and specifications to codify the knowledge of the transactional working groups. More details about the internal governance of the Accord Project are available [here](https://github.com/accordproject/governance).

### How can I get involved?

The Accord Project Community is developing several working groups focusing on different applications of smart contracts. The working groups have frequent calls and use the Accord Project’s Slack group chat application (join by clicking [here](https://accord-project-slack-signup.herokuapp.com/)) for discussion. The dates, dial-in instructions, and agendas for the working groups are all listed in the Project’s public calendar and typically also in working group’s respective slack channels.

A primary purpose of the working groups is to develop a universally accessible and widely used open source library of modular, smart legal contracts, smart templates and models that reflect input from the community. Smart legal contract templates are built according to the Project’s [Cicero Specification](https://github.com/accordproject/cicero).

Members can provide feedback into the templates and models relevant to a particular working group. You can immediately start contributing smart legal contract templates and models by using the Accord Project’s [Template Studio](https://studio.accordproject.org/).

The Accord Project has developed an easy-to-use programming language for building and executing smart legal contracts called Ergo. The goals of Ergo are to be accessible and usable by non-technical professionals, portable across, and compatible with, a variety of environments such as SaaS platforms and different blockchains, and meeting security, safety, and other requirements.

You can use the Accord Project’s [Template
Studio](https://studio.accordproject.org/) to create and test your smart legal
contracts.

---
id: version-0.21-accordproject-slc
title: Smart Legal Contracts
original_id: accordproject-slc

A Smart Legal Contract is a human-readable _and_ machine-readable agreement that is
digital, consisting of natural language and computable components.

The human-readable nature of the document ensures that signatories, lawyers,
contracting parties and others are able to understand the contract.

The machine-readable nature of the document enables it to be interpreted and
executed by computers, making the document "smart".

Contracts drafted with Accord Project can contain both traditional and machine-
readable clauses. For example, a Smart Legal Contract may include a smart payment
clause while all of the other provisions of the contract (Definitions, Jurisdiction
clause, Force Majeure clause, ...) are being documented solely in regular natural
language text.

A Smart Legal Contract is a general term to refer to two compatible, architectural
forms of contract:
- Machine-Readable Contracts, which tie legal text to data
- Machine-Executable Contracts, which tie legal text to data and executable code

### Machine-Readable Contracts

By combining Text and a data, a clause or contract becomes machine-readable.

For instance, the clause below for a [fixed rate
loan](https://templates.accordproject.org/fixed-interests-static@0.2.0.html)
includes natural language text coupled with variables. Together, these variables
refer to some data for the clause and correspond to the 'deal points':

```tem
## Fixed rate loan

This is a *fixed interest* loan to the amount of {{loanAmount}}
at the yearly interest rate of {{rate}}%
with a loan term of {{loanDuration}},
and monthly payments of {{monthlyPayment}}.
```

To make sense of the data, a _Data Model_, expressed in the Concerto schema
language, defines the variables for the template and their associated Data Types:

```ergo
  o Double loanAmount     // loanAmount is a floating-point number
  o Double rate           // rate is a floating-point number
  o Integer loanDuration  // loanDuration is an integer
```

  o Double monthlyPayment // monthlyPayment is a floating-point number

The Data Types allow a computer to validate values inserted into each of the `{{variable}}` placeholders (e.g., `2.5` is a valid `{{rate}}` but `January` isn't). In other words, the Data Model lets a computer make sense of the structure of (and data in) the clause. To learn more about Data Types see [Concerto Modeling](model-concerto).

The clause data (the 'deal points') can then be capture as a machine-readable representation:

```js
{
  "$class": "org.accordproject.interests.TemplateModel",
  "clauseId": "cec0a194-cd45-42f7-ab3e-7a673978602a",
  "loanAmount": 100000.0,
  "rate": 2.5,
  "loanDuration": 15
  "monthlyPayment": 667.0
}
```

The values entered into the template text are associated with the name of the variable e.g. `{{rate}} = 2.5%`. This provides the structure for understanding the clause and its contents.

### Machine-Executable Contracts

By adding Logic to a machine-readable clause or contract in the form of expressions - much like in a spreadsheet - the contract is able to execute operations based upon data included in the contract.

For instance, the clause below is a variant of the earlier [fixed rate loan](https://templates.accordproject.org/fixed-interests@0.2.0.html). While it is consistent with the previous one, the `{{monthlyPayment}}` variable is replaced with an [Ergo](logic-ergo) expression `monthlyPaymentFormula(loanAmount,rate,loanDuration)` which calculates the monthly interest rate based upon the values of the other variables: `{{loanAmount}}`, `{{rate}}`, and `{{loanDuration}}`. To learn more about contract Logic see [Ergo Logic](logic-ergo).

```tem
## Fixed rate loan

This is a *fixed interest* loan to the amount of {{loanAmount}}
at a yearly interest rate of {{rate}}%
with a loan term of {{loanDuration}},
and monthly payments of {{% monthlyPaymentFormula(loanAmount,rate,loanDuration)
%}}.
```

This is a simple example of the benefits of Machine-Executable contract, here adding logic to ensure that the value of the `{{monthlyPayment}}` in the text is always consistent with the other variables in the clause. In this example, we display the contract text using the underlying [Markup](markup-preliminaries) format, instead of the rich-text output that would be found in [editor tools](started-resources#ecosystem-tools) and PDF outputs.

More complex examples, (e.g., how to add post-signature logic which responds to
data sent to the contract or which triggers operations on external systems) can be
found in the rest of this documentation.

---
id: version-0.21-accordproject-template
title: Accord Project Templates
original_id: accordproject-template

An Accord Project template ties legal text to computer code. It is composed of
three elements:

- **Template Text**: the natural language of the template
- **Template Model**: the data model that backs the template, acting as a bridge
between the text and the logic
- **Template Logic**: the executable business logic for the template

![Template](assets/020/template.png)

The three components (Text - Model - Logic) can also be intuitively understood as a
progression, from _human-readable_ legal text to _machine-readable_ and
machine-executable. When combined these three elements allow templates to be
edited, validated, and then executed on any computer platform (on your own machine,
on a Cloud platform, on Blockchain, etc).

> We use the computing term 'executed' here, which means run by a computer. This is
distinct from the legal term 'executed', which usually refers to the process of
signing an agreement.

## Template Text

![Template Text](assets/020/template_text.png)

The template text is the natural language of the clause or contract. It can include
markup to indicate [variables](ref-glossary#variable) for that template.

The following shows the text of an **Acceptance of Delivery** clause.

```tem
## Acceptance of Delivery.

{{shipper}} will be deemed to have completed its delivery obligations
if in {{receiver}}'s opinion, the {{deliverable}} satisfies the
Acceptance Criteria, and {{receiver}} notifies {{shipper}} in writing
that it is accepting the {{deliverable}}.

## Inspection and Notice.

{{receiver}} will have {{businessDays}} Business Days to inspect and
evaluate the {{deliverable}} on the delivery date before notifying
{{shipper}} that it is either accepting or rejecting the
{{deliverable}}.

## Acceptance Criteria.

The 'Acceptance Criteria' are the specifications the {{deliverable}}
```

~~~
must meet for the {{shipper}} to comply with its requirements and
obligations under this agreement, detailed in {{attachment}}, attached
to this agreement.
```

The text is written in plain English, with variables between `{{` and `}}`.
Variables allows template to be used in different agreements by replacing them with
different values.

For instance, the following show the same **Acceptance of Delivery** clause where
the `shipper` is `"Party A"`, the `receiver` is `"Party B"`, the `deliverable` is
`"Widgets"`, etc.

```md
## Acceptance of Delivery.

"Party A" will be deemed to have completed its delivery obligations
if in "Party B"'s opinion, the "Widgets" satisfies the
Acceptance Criteria, and "Party B" notifies "Party A" in writing
that it is accepting the "Widgets".

## Inspection and Notice.

"Party B" will have 10 Business Days to inspect and
evaluate the "Widgets" on the delivery date before notifying
"Party A" that it is either accepting or rejecting the
"Widgets".

## Acceptance Criteria.

The "Acceptance Criteria" are the specifications the "Widgets"
must meet for the "Party A" to comply with its requirements and
obligations under this agreement, detailed in "Attachment X", attached
to this agreement.
```

### TemplateMark

TemplateMark is the markup format in which the text for Accord Project templates is
written. It defines notations (such as the `{{` and `}}` notation for variables
used in the **Acceptance of Delivery** clause) which allows a computer to make
sense of your templates.

It also provides the ability to specify the document structure (e.g., headings,
lists), to highlight certain terms (e.g., in bold or italics), to indicate text
which is optional in the agreement, and more.

_More information about the Accord Project markup can be found in the [Markdown
Text](markup-templatemark) Section of this documentation._

## Template Model

![Template Model](assets/020/template_model.png)

Unlike a standard document template (e.g., in Word or pdf), Accord Project
templates associate a _model_ to the natural language text. The model acts as a
bridge between the text and logic; it gives the users an overview of the
components, as well as the types of different components.
~~~

The model categorizes variables (is it a number, a monetary amount, a date, a reference to a business or organization, etc.). This is crucial as it allows the computer to make sense of the information contained in the template.

The following shows the model for the **Acceptance of Delivery** clause.

```ergo
/* The template model */
asset AcceptanceOfDeliveryClause extends AccordClause {

  /* the shipper of the goods*/
  --> Organization shipper

  /* the receiver of the goods */
  --> Organization receiver

  /* what we are delivering */
  o String deliverable

  /* how long does the receiver have to inspect the goods */
  o Integer businessDays

  /* additional information */
  o String attachment
}
```

Thanks to that model, the computer knows that the `shipper` variable (`"Party A"` in the example) and the `receiver` variable (`"Party B"` in the example) are both `Organization` types. The computer also knows that variable `businessDays` (`10` in the example) is an `Integer` type; and that the variable `deliverable` (`"Widgets"` in the example) is a `String` type, and can contain any text description.

> If you are unfamiliar with the different types of variables, or want a more thorough explanation of what variables are, please refer to our [Glossary](ref-glossary#data-models) for a more detailed explanation.

### Concerto

Concerto is the language which is used to write models in Accord Project templates. Concerto offers modern modeling capabilities including support for primitive types (numbers, dates, etc), nested or optional data structures, enumerations, relationships, object-oriented style inheritance, and more.

More information about Concerto can be found in the [Concerto Model](model-concerto) section of this documentation.

## Template Logic

![Template Logic](assets/020/template_logic.png)

The combination of text and model already makes templates _machine-readable_, while the logic makes it _machine-executable_.

### During Drafting

In the [Overview](accordproject) Section, we already saw how logic can be embedded in the text of the template itself to automatically calculate a monthly payment for a [fixed rate loan]():

```tem
## Fixed rate loan

This is a *fixed interest* loan to the amount of {{loanAmount}}
at a yearly interest rate of {{rate}}%
with a loan term of {{loanDuration}},
and monthly payments of {{% monthlyPaymentFormula(loanAmount,rate,loanDuration)
%}}.
```

This uses a `monthlyPaymentFormula` function which calculates the monthly payment
based on the other data points in the text:
```ergo
define function monthlyPaymentFormula(loanAmount: Double, rate: Double,
loanDuration: Integer) : Double {
  let term = longToDouble(loanDuration * 12);       // Term in months
  if (rate = 0.0) then return (loanAmount / term)   // If the rate is 0
  else
    let monthlyRate = (rate / 12.0) / 100.0;        // Rate in months
    let monthlyPayment =                            // Payment calculation
      (monthlyRate * loanAmount)
      / (1.0 - ((1.0 + monthlyRate) ^ (-term)));
    return roundn(monthlyPayment, 0)                // Rounding
}
```
Each logic function has a _name_ (e.g., `monthlyPayment`), a _signature_ indicating
the parameters with their types (e.g., `loanAmount:Double`), and a _body_ which
performs the appropriate computation based on the parameters. The main payment
calculation is here based on the [standardized calculation used in the United
States](https://en.wikipedia.org/wiki/Mortgage_calculator#Monthly_payment_formula)
with `*` standing for multiplication, `/` for division, and `^` for exponentiation.

### After Signature

The logic can also be used to associate behavior to the template _after_ the
contract has been signed. This can be used for instance to specify what happens
when a delivery is received late, to check conditions for payment, determine if
there has been a breach of contract, etc.

The following shows post-signature logic for the **Acceptance of Delivery** clause.

```ergo
contract SupplyAgreement over SupplyAgreementModel {
  clause acceptanceofdelivery(request : InspectDeliverable) : InspectionResponse {

    let received = request.deliverableReceivedAt;
    enforce isBefore(received,now()) else
      throw ErgoErrorResponse{ message : "Transaction time is before the
deliverable date." }
    ;

    let status =
      if isAfter(now(), addDuration(received, Duration{ amount:
contract.businessDays, unit: ~org.accordproject.time.TemporalUnit.days}))
      then OUTSIDE_INSPECTION_PERIOD
      else if request.inspectionPassed
      then PASSED_TESTING
      else FAILED_TESTING
    ;
    return InspectionResponse{
      status : status,
      shipper : contract.shipper,
      receiver : contract.receiver
    }
  }
}
```

This logic describes what conditions must be met for a delivery to be accepted. It checks whether the delivery has already been made; whether the acceptance is timely, within the specified inspection date; and whether the inspection has passed or not.

### Ergo

Ergo is the programming language which is used to express contractual logic in templates. Ergo is specifically designed for legal agreements, and is intended to be accessible for those creating the corresponding prose for those computable legal contracts. Ergo expressions can also be embedded in the text for a template.

More information about Ergo can be found in the [Ergo Logic](logic-ergo) Section of this documentation.

## Cicero

The implementation for the Accord Project templates is called [Cicero](https://github.com/accordproject/cicero). It defines and can read the structure of templates, with natural language bound to a data model and logic. By doing this, Cicero allows users to create, validate and execute software templates which embody all three components in the template triangle above.

More information about how to install Cicero and get started with Accord Project templates can be found in the [Installation](started-installation) Section of this documentation.

Let's look at each component of the template triangle, starting with the text.

### What next?

Build your first smart legal contract templates, either [online](tutorial-latedelivery) with Template Studio, or by [installing Cicero](started-installation).

Explore [sample templates](started-resources) and other resources in the rest of this documentation.

If some of technical words are unfamiliar, please consult the [Glossary](ref-glossary) for more detailed explanations.

---
id: version-0.21-accordproject-tour
title: Online Tour
original_id: accordproject-tour

To get an better acquainted with Accord Project templates, the easiest way is through the online [Template Studio](https://studio.accordproject.org) editor.

:::tip
You can open template studio from anywhere in this documentation by clicking the [Try Online!](https://studio.accordproject.org) button located in the top-right of the page.
:::

The following video offers a tour of Template Studio and an introduction to the key concepts behind the Accord Project technology.

<iframe src="https://player.vimeo.com/video/328933628" width="640" height="400" frameborder="0" allow="autoplay; fullscreen" allowfullscreen></iframe>

Here is a timestamp of what is covered in the video:

- **00:08** : Introduction to template Studio & pointers the other parts of the docs / AP website
- **03:51** : Helloworld template tutorial start
- **04:48** : Template Studio - Metadata
- **06:56** : Template Studio - Contract Text
  - **07:52** : The 'live' nature of the text; how to read errors
  - **08:39** : Changing the template text itself
  - **09:17** : Changing the variables in the template text
- **10:00** : Template Studio - Model update
- **11:28** : Template Studio - Logic update
  - **13:17** : Explanation about request types / response types
- **14:44** : Template Studio - Logic - Test Execution (request, response)
  - **15:57** : Test Execution - contract state
  - **16:49** : Test Execution - obligations
- **18:20** : Wrap-up

---
id: version-0.21-accordproject
title: Overview
original_id: accordproject

## What is the Accord Project?

Accord Project is an open source, non-profit initiative aimed at transforming contract management and contract automation by digitizing contracts. It provides an open, standardized format for Smart Legal Contracts.

The Accord Project defines a notion of a legal template with associated computing logic which is expressive, open-source, and portable. Accord Project templates are similar to a clause or contract template in any document format, but they can be read, interpreted, and run by a computer.

## Why is the Accord Project relevant?

The Accord Project provides a universal format for smart legal contracts, and this format is embodied in a variety of open source projects that comprise the Accord Project technology stack. Input from businesses, lawyers and developers is crucial for the Accord Project.

### For Businesses

Contracting is undergoing a digital transformation driven by a need to deliver
customer-centric legal and business solutions faster, and at lower cost. This
imperative is fueling the adoption of a broad range of new technologies to improve
the efficiency of drafting, managing, and executing legal contracting operations;
the Accord Project is proud to be part of that movement.

The Accord Project provides a Smart Contract that does not depend on a blockchain,
that can integrate text
and data and that can continue operating over its lifespan. The Accord Project
smart contract can integrate with your technology platforms and become part of you
digital infrastructure.

In addition, contributions from businesses are crucial for the development of the
Accord Project. The expertise of stakeholders, such as business professionals and
attorneys, is invaluable in improving the functionality and content of the Accord
Project's codebase and specifications, to ensure that the templates meet real-world
business requirements.

If this interests you, please visit our [Lifecycle and Industry Working Groups]
(https://www.accordproject.org/liwg) page for more information.

### For Lawyers

The Legal world is changing and Legal Tech is growing into a billion dollar
industry. The modern lawyer has to be at home in the digital world. Law Schools now
teach courses in coding for lawyers, computational law, blockchain and artificial
intelligence. Legal Hackers is a world wide movement uniting lawyers across the
world in a shared passion for law and technology. Lawyers need to move beyond the
the written word on paper.

The template in an Accord Project Contract is pure legal text that can be drafted
by lawyers and interpreted by courts. An existing contract can easily be
transformed into a template by adding data points between curly braces that
represent the Concerto model and Ergo logic can be added as an integral part of the
contract. The template language is subject to judicial interpretation and the
Concerto model and Ergo logic can be interpreted by a computer creating a bridge
between the two worlds.

As a lawyer, contributing to the Accord Project would be a great opportunity to
learn about smart legal contracts. Through the Accord Project, you can understand
the foundations of open source technologies and learn how to develop smart
agreements.

If your organization wants to become a member of the Accord Project, please [join
our community](https://www.accordproject.org/membership).

### For Developers

The Accord Project provides a universal format for smart legal contracts, and this
format is embodied in a variety of open source projects that comprise the Accord
Project technology stack. The Accord Project is an open source project and welcomes
contributions from anyone.

The Accord Project is developing tools including a [Visual Studio Code plugin]
(https://marketplace.visualstudio.com/items?itemName=accordproject.cicero-vscode-
extension), [React based web components](https://github.com/accordproject/web-
components) and a command line interface for working with Accord Project Contracts.

You can integrate contracts into existing applications, create new applications or simply assist lawyers with developing applications with the Ergo language.

There is a welcoming community on Slack that is eager to help. [Join our Community](https://www.accordproject.org/membership/)

## About this documentation

If you are new to Accord Project, you may want to first read about the notion of [Smart Legal Contracts](accordproject-slc) and about [Accord Project Templates](accordproject-template). We also recommend taking the [Online Tour](accordproject-tour).

To start using Accord Project templates, follow the [Install Cicero](https://docs.accordproject.org/docs/next/started-installation.html) instructions in the _Getting Started_ Section of the documentation.

You can find in-depth guides for the different components of a template in the _Template Guides_ part of the documentation:
- Learn how to write contract or template text in the [Markdown Text](markup-preliminaries) Guide
- Learn how to design your data model in the [Concerto Model](model-concerto) Guide
- Learn how to write smart contract logic in the [Ergo Logic](logic-ergo) Guide

Finally, the documentation includes several step by step [Tutorials](tutorial-templates) and some reference information (for APIs, command-line tools, etc.) can be found in the [Reference Manual](ref-glossary).

---
id: version-0.21-ergo-tutorial
title: Ergo: A Tutorial
original_id: ergo-tutorial

## Overview of Accord

Cicero is an Open Source implementation of the Accord Project Template Specification. It defines the structure of natural language templates, bound to a data model, that can be executed using Ergo and request/response JSON messages. You can read the latest user documentation here: http://docs.accordproject.org.

In short, with the Accord Project you can take a classic contract, e.g. Word document and use Cicero to define natural language contract and clause templates that can be executed by an event driven computer program (aka Smart contract). For the tutorial, Cicero will be used to define natural language contract and clause templates. These clause templates handle the syllogistic language of contracts.

For example,
```md
 if the goods are more than [{DAYS}] late,
 then notify the supplier of the goods, with the message [{MESSAGE}].
```
DAYS and MESSAGE are variables

You can browse the library of Open Source Cicero contract and clause templates at: https://templates.accordproject.org.

So how goes the contract get executed? That is where Ergo comes in Ergo is a strongly-typed functional programming language designed to capture the legal intent of legal contracts and clauses. We will use Ergo to create the contract logic consisting of a contract class with executable embedded clauses. Note: prior to the emergence of Ergo, the Cicero JavaScript component was primary to the execution of code.

Ergo obviates the Cicero JavaScript component for the execution phase with a new more comprehensive language which we explore in this tutorial.

## Cicero

The Open Source Cicero project defines the format of clause and contract templates based on to the Cicero Template Specification. The templates are the link between the natural language of contracts usually composed in a Word document and the specification of a machine executable transaction. Cicero templates define the API by specifying request and response elements for the logic associated with functional transaction executed by Ergo.

Cicero templates are composed of two elements:
* Template Grammar (the natural language text for the template),
* Template Model (the data model that includes the variables contained within the template).
* The Logic (the executable business logic for the template) will be handled by Ergo.

When combined these three elements allow templates to be edited, analyzed, queried and executed.

## Setup Ergo Development environment

Before you can build Ergo, you must install and configure the following dependencies on your machine:

### Git

* Git: The [Github Guide to Installing Git][git-setup] is a good source of information.

### Node.js

* Node.js (LTS): We use Node to generate the documentation, run a development web server, run tests, and generate distributable files. Depending on your system, you can install Node either from source or as a pre-packaged bundle.
> Tip: Use nvm (or nvm-windows) to manage and install Node.js, This facilitates a version change of Node.js per project.
* Lerna: This is a tool which helps when handling multiple npm packages in the Ergo repository. To install:
```
npm install -g lerna@^3.15.0
```

### Visual Studio Code

Visual Studio Code is a lightweight but powerful source code editor which runs on your desktop and is available for Windows, macOS and Linux. It comes with built-in support for JavaScript and Node.js and has a rich ecosystem of extensions for other languages (such Ergo).

Follow the platform specific guides below:

See, https://code.visualstudio.com/docs/setup/
* macOS
* Linux
* Windows

#### Install Ergo VisualStudio Plugin

### Validate Development Environment and Toolset

Clone https://github.com/accordproject/ergo to your local machine

### Getting started

Install Ergo

The easiest way to install Ergo is as a Node.js package. Once you have Node.js installed on your machine, you can get the Ergo compiler and command-line using the Node.js package manager by typing the following in a terminal:
$ npm install -g @accordproject/ergo-cli@0.20

This will install the compiler itself (ergoc) and a command-line tool (ergo) to execute Ergo code. You can check that both have been installed and print the version number by typing the following in a terminal:
```sh
$ ergoc --version
$ ergo --version
```
Then, to get command line help:
```
$ ergoc --help
$ ergo execute --help
```
Compiling your first contract
```ergo
namespace org.accordproject.volumediscount

contract VolumeDiscount over VolumeDiscountContract {
   // Clause for volume discount
   clause volumediscount(request : VolumeDiscountRequest) : VolumeDiscountResponse
{
        if request.netAnnualChargeVolume < contract.firstVolume
        then return VolumeDiscountResponse{ discountRate: contract.firstRate }
        else if request.netAnnualChargeVolume < contract.secondVolume
        then return VolumeDiscountResponse{ discountRate: contract.secondRate }
        else return VolumeDiscountResponse{ discountRate: contract.thirdRate }
  }
}
```

To compile your first Ergo contract to JavaScript , within Visual Studio code
* Open the folder where you cloned https://github.com/accordproject/ergo
* Use View/Terminal to run the Ergo compiler:
```sh
$ ergoc ./examples/volumediscount/model.cto ./examples/volumediscount/logic.ergo
Compiling Ergo './examples/volumediscount/logic.ergo' -- creating
'./examples/volumediscount/logic.js'
```

By default, Ergo compiles to JavaScript for execution. This may change in the

future to support other languages. The compiled code for the result in stored as
`./examples/volumediscount/logic.js`

### Execute a contract
To execute a contract, we pass the necessary parameters including the CTO, Ergo
files, the name of a contract and the json files containing request and contract
state
ergorun [ctos] [ergos] --contractname [file] --contract [file] --state [file] --
request [file]

So for example we use ergorun with :
```sh
$ ergorun ./examples/volumediscount/model.cto ./examples/volumediscount/logic.ergo
--contractname org.accordproject.volumediscount.VolumeDiscount
--contract ./examples/volumediscount/contract.json
--request ./examples/volumediscount/request.json
--state ./examples/volumediscount/state.json
```

Here contract.json contains the following values
```json
{
  "$class": "org.accordproject.volumediscount.VolumeDiscountContract",
  "parties": null,
  "contractId": "cr1",
  "firstVolume": 1,
  "secondVolume": 10,
  "firstRate": 3,
  "secondRate": 2.9,
  "thirdRate": 2.8
}
```

Request.json contains
```json
{
  "$class": "org.accordproject.volumediscount.VolumeDiscountRequest",
  "netAnnualChargeVolume": 10.4
}
```

logic.ergo contains:
```ergo
namespace org.accordproject.volumediscount

contract VolumeDiscount over VolumeDiscountContract {
  // Clause for volume discount
  clause volumediscount(request : VolumeDiscountRequest) : VolumeDiscountResponse {
    if request.netAnnualChargeVolume < contract.firstVolume
    then return VolumeDiscountResponse{ discountRate: contract.firstRate }
    else if request.netAnnualChargeVolume < contract.secondVolume
    then return VolumeDiscountResponse{ discountRate: contract.secondRate }
    else return VolumeDiscountResponse{ discountRate: contract.thirdRate }
  }
}
```

Here netAnnualCharge Volume equals 10.4 which is not less than firstVolume and
secondVolume which are equal to 1 and 10 respectively so the logic for the

volumediscount clause returns thirdRate which equals 2.8

```
7:31:58 PM - info: Logging initialized. 2018-09-27T23:31:58.623Z
7:31:59 PM - info: {"response":
{"discountRate":2.8,"$class":"org.accordproject.volumediscount.VolumeDiscountRespon
se"},"state":
{"$class":"org.accordproject.cicero.contract.AccordContractState","stateId":"1"},"e
mit":[]}
```

PS D:\Users\jbambara\Github\ergo>

## Ergo Development

Create Template
Start with basic agreement in natural language and locate the variables
Here in the example see the bold
Volume-Based Card Acceptance Agreement [Abbreviated]
This Agreement is by and between ………..you agree to be bound by the Agreement.
Discount means an amount that we charge you for accepting the Card, which amount is:
(i) a percentage (Discount Rate) of the face amount of the Charge that you submit, or a flat per-
Transaction fee, or a combination of both; and/or
(ii) a Monthly Flat Fee (if you meet our requirements).

Transaction Processing and Payments. ………………… less all applicable deductions, rejections, and withholdings, which include:
………………………….

SETTLEMENT
a) Settlement Amount. Our agent will pay you according to your payment plan,
……………………..which include:
(i) the Discount,
……………………………………………..
b) Discount. The Discount is determined according to the following table:

Annual Dollar Volume	Discount
Less than $1 million	3.00%
$1 million to $10 million	2.90%
Greater than $10 million	2.80%

Identify the request variables and contract instance variables
Codify the variables with $[{request}] or [{contract instance}]

Annual Dollar Volume	Discount
Less than $[{firstVolume}] million	[{firstRate}]%
$[{firstVolume}] million to $[{secondVolume}] million	[{secondRate}]%
Greater than $[{secondVolume}] million	[{thirdRate}]%

Create Model
Define the model asset which contains the contract instance variables and the transaction request and response. Defines the data model for the VolumeDiscount template. This defines the structure that the parser for the template generates from input source text. See model.cto below:

```
 namespace org.accordproject.volumediscount
import org.accordproject.cicero.contract.* from
https://models.accordproject.org/cicero/contract.cto
import org.accordproject.cicero.runtime.* from
```

```
https://models.accordproject.org/cicero/runtime.cto
asset VolumeDiscountContract extends AccordContract {
  o Double firstVolume
  o Double secondVolume
  o Double firstRate
  o Double secondRate
  o Double thirdRate
}
transaction VolumeDiscountRequest {
  o Double netAnnualChargeVolume
}
transaction VolumeDiscountResponse {
        o Double discountRate
}
```

Create Logic
The contract logic is accomplished by coding ERGO statements and expressions to consume the request and use contract instance variables to produce the desired response. In our example, request.netAnnualChargeVolume is tested against contract rates to produce the result.

```
namespace org.accordproject.volumediscount

define the contract
contract VolumeDiscount over VolumeDiscountContract {

define the contract clause and request : response

   clause volumediscount(request : VolumeDiscountRequest) : VolumeDiscountResponse
{

define the logic ; here we use if /then /else statement to test request parameter
against contract instance variable
 and return

        if request.netAnnualChargeVolume < contract.firstVolume
        then return VolumeDiscountResponse{ discountRate: contract.firstRate }
        else if request.netAnnualChargeVolume < contract.secondVolume
        then return VolumeDiscountResponse{ discountRate: contract.secondRate }
        else return VolumeDiscountResponse{ discountRate: contract.thirdRate }
  }
```

Ergo Language
As you have seen in this tutorial, Ergo is a domain-specific language (DSL) that captures the execution logic of legal contracts. In this simple example, you see that Ergo aims to have contracts and clauses as first-class elements of the language. To accommodate the maturation of distributed ledger implementations, Ergo will be blockchain neutral, i.e., the same contract logic can be executed either on and off chain on distributed ledger technologies like HyperLedger Fabric. Most importantly, Ergo is consistent with the Accord Protocol Template Specification.
Follow the links below to learn more about
Introduction to Ergo
Ergo Language Guide
Ergo Reference Guide

October 12, 2018

---

id: version-0.21-logic-advanced-expr
title: Advanced Expressions
original_id: logic-advanced-expr

## Match

### Match against Values

Match expressions allow to check an expression against multiple possible
values:

```ergo
    match fruitcode
      with 1 then "Apple"
      with 2 then "Apricot"
      else "Strange Fruit"
```

Match expressions can also be used to match against enumerated values:
```ergo
    match state
      with NY then "Empire State"
      with NJ then "Garden State"
      else "Far from home state"
```

### Match against Types

Match expressions can be used to match a value against a class type:.

```
define constant products = [
    Product{ id : "Blender" },
    Car{ id : "Batmobile", range: "Infinite" },
    Product{ id : "Cup" }
  ]

foreach p in products
return
  match p
    with let x : Car then "Car (" ++ x.id ++ ") with range " ++ x.range
    with let x : Product then "Product (" ++ x.id ++ ")"
    else "Not a product"
```
Should return the array `["Product (Blender)", "Car (Batmobile) with range
Infinite", "Product (Cup)"]`

## Foreach

Foreach expressions allow to apply an expression of every element in
an input array of values and returns a new array:

```ergo
  foreach x in [1.0,-2.0,3.0] return x + 1.0
```

Foreach expressions can have an optional condition of the values being
iterated over:

```ergo
  foreach x in [1.0,-2.0,3.0] where x > 0.0 return x + 1.0
```

Foreach expressions can iterate over multiple arrays. For example, the following foreach expression returns all all [Pythagorean triples](https://en.wikipedia.org/wiki/Pythagorean_triple):
```ergo
let nums = [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0];
foreach x in nums
foreach y in nums
foreach z in nums
where (x^2.0 + y^2.0 = z^2.0)
return {a: x, b: y, c: z}
```
and should return the array `[{a: 3.0, b: 4.0, c: 5.0}, {a: 4.0, b: 3.0, c: 5.0}, {a: 6.0, b: 8.0, c: 10.0}, {a: 8.0, b: 6.0, c: 10.0}]`.

## Template Literals

Template literals are similar to [String literals](logic-simple-expr#literal-values) but with the ability to embed Ergo expressions. They are written with between `` ` `` and may contains Ergo expressions inside `{{%` and `%}}`.

The following Ergo expressions illustrates the use of a template literal to construct a String describing the content of a record.
```
let law101 = {
    name: "Law for developers",
    fee: 29.99
  };
`Course "{{% law101.name %}}" (Cost: {{% law101.fee %}})`
```
Should return the string literal `"Course \"Law for developers\" (Cost: 29.99)"`.

## Formatting

One can use template formatting using the `Expr as "FORMAT"` Ergo expression. Supported formats are the same as those available in TemplateMark [Formatted Variables](markup-templatemark#formatted-variables).

For instance:
```
let payment = MonetaryAmount{ currencyCode: USD, doubleValue: 1129.99 };
payment as "K0,0.00"
```
Should return the string literal `"$1,129.99"`.

---
id: version-0.21-logic-complex-type
title: Complex Values & Types
original_id: logic-complex-type

So far we only considered atomic values and types, such as string values or integers, which are not sufficient for most contracts. In Ergo, values and types are based on the [Concerto Modeling](model-concerto) (often referred to as CTO

models after the `.cto` file extension). This provides a rich vocabulary to define
the parameters of your contract, the information associated to contract
participants, the structure of contract obligation, etc.

In Ergo, you can either import an existing CTO model or declare types directly
within your code. Let us look at the different kinds of types you can define and
how to create values with those types.

## Arrays

Array types lets you define collections of values and are denoted with `[]` after
the type of elements in that collection:

```ergo
    String[]                         // a String array
    Double[]                         // a Double array
```

You can write arrays as follows:
```ergo
    ["pear","apple","strawberries"]  // an array of String values
    [3.14,2.72,1.62]                 // an array of Double values
```

You can construct arrays using other expressions:
```ergo
    let pi = 3.14;
    let e = 2.72;
    let golden = 1.62;
    [pi,e,golden]
```

Ergo also provides functions to manipulate arrays as parts of its [standard
library](ref-logic-stdlib.html#functions-on-arrays). The following example uses the
`sum` function to calculate the sum of all the elements in the `prettynumbers`
array.
```ergo
    let pi = 3.14;
    let e = 2.72;
    let golden = 1.62;
    let prettynumbers : Double[] = [pi,e,golden];
    sum(prettynumbers)
```

You can access the element at a given position inside the array using an index:
```ergo
    let fruits = ["pear","apple","strawberries"];
    fruits[0]          // Returns: some("pear")
     let fruits = ["pear","apple","strawberries"];
    fruits[2]          // Returns: some("strawberries")
     let fruits = ["pear","apple","strawberries"];
    fruits[4]          // Returns: none
```

 Note that the index starts at `0` for the first element and that indexed-based
access returns an optional value, since Ergo compiler cannot statically determine
whether there will be an element at the corresponding index. You can learn more
about how to handle optional values and types in the [Optionals](logic-complex-
type#optionals) Section below.

## Classes

You can declare classes in the Concerto Modeling Language (concepts, transactions, events, participants or assets) by importing them from a CTO file or directly within your Ergo program:

```ergo
   define concept Seminar {
     name : String,
     fee : Double
   }
   define asset Product {
     id : String
   }
   define asset Car extends Product {
     range : String
   }
   define transaction Response {
     rate : Double,
     penalty : Double
   }
  define event PaymentObligation{
    amount : Double,
    description : String
  }
```

Once a class type has been defined, you can create an instance of that type using the class name along with the values for each fields:

```ergo
    Seminar{
      name: "Law for developers",
      fee: 29.99
    }
    Car{
      id: "Batmobile4156",
      range: "Infinite"
    }
```

> **TechNote:** When extending an existing class (e.g., `Car extends Product`), the sub-class includes the fields from the super-class. So `Car` includes the field `range` which is locally declared and the field `id` which is declared in `Product`.

You can access fields for values of a class type by using the `.` operator:
```ergo
    Seminar{
      name: "Law for developers",
      fee: 29.99
    }.fee                        // Returns 29.99
```

## Records

Sometimes it is convenient to declare a structure without having to declare it first. You can do that using a record, which is similar to a class but without a name attached to it:

```ergo
   {
     name : String,  // A record with a name of type String
     fee : Double    // and a fee of type Double
   }
```

You do not need to declare that record, and can directly write an instance of that record as follows:

```ergo
    {
      name: "Law for developers",
      fee: 29.99
    }
```

> Typing `return { name: "Law for developers", fee: 29.99 }` in the [Ergo REPL](https://ergorepl.netlify.com), should answer `Response. {name: "Law for developers", fee: 29.99} : {fee: Double, name: String}`.

You can access the field of a record using the `.` operator:
```ergo
    {
      name: "Law for developers",
      fee: 29.99
    }.fee                        // Returns 29.99
```
## Enums

Here is how to declare an enumerated type:

```ergo
define enum ProductType {
   DAIRY,
   BEEF,
   VEGETABLES
}
```

To create an instance of that enum:
```ergo
DAIRY
BEEF
```

## Optionals

An optional type can contain a value or not and is indicated with a `?`.

```ergo
Integer?           // An optional integer
PaymentObligation? // An optional payment obligation
Double[]?          // An optional array of doubles
```

An optional value can be either present, written `some(v)`, or absent, written `none`.

```ergo
let i1 : Integer? = some(1); i1
let i2 : Integer? = none; i2
```

To operate on an optional type, you need to say what to do when the value is present and what to do when the value is not present. The most general way to do that is with a match expression:

This example matches a value which is present:
```ergo
match some(1)
with let? x then "I found " ++ toString(x) ++ " :-)"
else "I found nothing :-("
```
and should return `"I found 1 :-)"`.

While this example matches against a value which is absent:
```
match none
with let? x then "I found " ++ toString(x) ++ " :-)"
else "I found nothing :-("
```
and should return `"I found nothing :-("`.

More details on match expressions can be found in [Advanced Expressions](logic-advanced-expr#match).

For conciseness, a few operators are also available on optional values. One can give a default value when the optional is `none` using the operator `??`. For instance:

```ergo
some(1) ?? 0       // Returns the integer 1
none ?? 0          // Returns the integer 0
```

You can also access the field inside an optional concept or an optional record using the operator `?.`. For instance:

```ergo
some({a:1})?.a     // Returns the optional value: some(1)
none?.a            // Returns the optional value: none
```

---
id: version-0.21-logic-ergo
title: Ergo Overview
original_id: logic-ergo

## Language Goals

Ergo aims to:
- have contracts and clauses as first-class elements of the language

- help legal-tech developers quickly and safely write computable legal contracts
- be modular, facilitating reuse of existing contract or clause logic
- ensure safe execution: the language should prevent run-time errors and non-terminating logic
- be blockchain neutral: the same contract logic can be executed either on and off chain on a variety of distributed ledger technologies
- be formally specified: the meaning of contracts should be well defined so it can be verified, and preserved during execution
- be consistent with the [Accord Project Templates](accordproject-template)

## Design Choices

To achieve those goals the design of Ergo is based on the following principles:

- Ergo contracts have a class-like structure with clauses akin to methods
- Ergo can handle types (concepts, transactions, etc) defined with the [Concerto Modeling Language](https://github.com/accordproject/concerto) (so called CML models), as mandated by the Accord Project Template Specification
- Ergo borrows from strongly-typed functional programming languages: clauses have a well-defined type signature (input and output), they are functions without side effects
- The compiler guarantees error-free execution for well-typed Ergo programs
- Clauses and functions are written in an expression language with limited expressiveness (it allows conditional and bounded iteration)
- Most of the compiler is written in Coq as a stepping stone for formal specification and verification

## Status

- The current implementation is considered *in development*, we welcome contributions (be it bug reports, suggestions for new features or improvements, or pull requests)
- The current compiler targets JavaScript (either standalone or for use in Cicero Templates and Hyperledger Fabric) and Java (experimental)

## This Guide

Ergo provides a simple expression language to describe computation. From those expressions, one can write functions, clauses, and then whole contract logic. This guide explains most of the Ergo concepts starting from simple expressions all the way to contracts.

Ergo is a _strongly typed_ language, which means it checks that the expressions you use are consistent (e.g., you can take the square root of `3.14` but not of `"pi!"`). The type system is here to help you write better and safer contract logic, but it also takes a little getting used to. This page also introduces Ergo types and how to work with them.

---
id: version-0.21-logic-simple-expr
title: Simple Expressions
original_id: logic-simple-expr

## Literal values

The simplest kind of expressions in Ergo are literal values.

```ergo
    "John Smith" // a String literal
    1            // an Integer literal
    3.0          // a Double literal
    3.5e-10      // another Double literal
    true         // the Boolean true
    false        // the Boolean false
```

Each line here is a separate expression. At the end of the line, the notation `// write something here` is a _comment_, which means it is a part of your Ergo program which is ignored by the Ergo compiler. It can be useful to document your code.

Every Ergo expression can be _evaluated_, which means it should compute some value. In the case of a literal value, the result of evaluation is itself (e.g., the expression `1` evaluates to the integer `1`).

> You can actually see the result of evaluating expressions by trying them out in the [Ergo REPL](https://ergorepl.netlify.com). You have to prefix them with `return`: for instance, to evaluate the String literal `"John Smith"` type: `return "John Smith"` (followed by clicking the button 'Evaluate') in the REPL. This should answer: `Response. "John Smith" : String`.

## Operators

You can apply operators to values. Those can be used for arithmetic operations, to compare two values, to concatenate two string values, etc.

```ergo
    1.0 + 2.0 * 3.0      // arithmetic operators on Double
    -1.0
    1 + 2 * 3            // arithmetic operators on Integer
    -1

    1.0 <= 3.0           // comparison operators on Double
    1.0 = 2.0
    2.0 > 1.0
    1 <= 3               // comparison operators on Integer
    1 = 2
    2 > 1.0

    true or false        // Boolean disjunction
    true and false       // Boolean conjunction
    !true                // Negation

    "Hello" ++ " World!" // String concatenation
```

> Again, you can try those in the [Ergo REPL](https://ergorepl.netlify.com). For instance, typing `return true and false` should answer `Response. false : Boolean`, and typing `return 1.0 + 2.0 * 3.0` should answer: `Response. 7.0 : Double`.

## Conditional expressions

Conditional expressions can be used to perform different computations depending on some condition:

```ergo
    if 1.0 < 0.0     // Condition
    then "negative"  // Expression if condition is true
    else "positive"  // Expression if condition is false
```

> Typing `return if 1.0 < 0.0 then "negative" else "positive"` in the [Ergo REPL](https://ergorepl.netlify.com), should answer `Response. "positive" : String`.

See also the [Conditional Expression Reference](ref-ergo-spec.html#condition-expressions)

## Let bindings

Local variables can be declared with `let`:

```ergo
    let x = 1;              // declares and initialize a variable
    x+2                     // rest of the expression, where x is in scope
```

Let bindings give a name to some intermediate result and allows you to reuse the corresponding value in multiple places:

```ergo
   let x = -1.0;            // bind x to the value -1.0
   if x < 0.0               // if x is negative
   then -x                  // then return the opposite of x
   else x                   // else return x
```

> **TechNote:** let bindings in Ergo are immutable, in a way similar to other functional languages. A nice explanation can be found e.g., in the documentation for let bindings in [ReasonML](https://reasonml.github.io/docs/en/let-binding).

---
id: version-0.21-logic-stmt
title: Statements
original_id: logic-stmt

A clause's body is composed of statements. Statements are a special kind of expression which can manipulate the contract state and emit obligations. Unlike other expressions they may return a response or an error.

## Contract data

When inside a statement, data about the contract -- either the contract parameters, clause parameters or contract state are available using the following Ergo keywords:
```ergo
    contract   // The contract parameters (from a contract template)
    clause     // Local clause parameters (from a clause template)
    state      // The contract state
```

For instance, if your contract template parameters and state information have the

following types:
```ergo
    // Template parameters
    asset InstallmentSaleContract extends AccordContract {
      o AccordParty BUYER
      o AccordParty SELLER
      o Double INITIAL_DUE
      o Double INTEREST_RATE
      o Double TOTAL_DUE_BEFORE_CLOSING
      o Double MIN_PAYMENT
      o Double DUE_AT_CLOSING
      o Integer FIRST_MONTH
    }
    // Contract state
    enum ContractStatus {
      o WaitingForFirstDayOfNextMonth
      o Fulfilled
    }
    asset InstallmentSaleState extends AccordContractState {
      o ContractStatus status
      o Double balance_remaining
      o Integer next_payment_month
      o Double total_paid
   }
```

You can use the following expressions:
```ergo
    contract.BUYER
    state.balance_remaining
```

## Returning a response

Returning a response from a clause can be done by using a `return` statement:

```ergo
     return 1                        // Return the integer one
     return Payout{ amount: 39.99 } // Return a new Payout object
     return                          // Return nothing
```

> **TechNote:** the [Ergo REPL](https://ergorepl.netlify.com) takes statements as input which is why we had to add `return` to expressions in previous examples.

## Returning a failure

Returning a failure from a clause can be done by using a `throw` statement:
```ergo
    throw ErgoErrorResponse{ message: "This is wrong" }
    define concept MyOwnError extends ErgoErrorResponse{ fee: Double }
    throw MyOwnError{ message: "This is wrong and costs a fee", fee: 29.99 }
```

For convenience, Ergo provides a `failure` function which takes a string as part of its [standard library](ref-logic-stdlib#other-functions), so you can also write:
```ergo
    throw failure("This is wrong")
```

## Enforce statement

Before a contract is enforceable some preconditions must be satisfied:
- Competent parties who have the legal capacity to contract
- Lawful subject matter
- Mutuality of obligation
- Consideration

The constructs below will be used to determine if the preconditions have been met
and what actions to take if they are not

```test
Example Prose
    Do the parties have adequate funds to execute this contract?
```

One can check preconditions in a clause using enforce statements, as
follows:

```ergo
    enforce x >= 0.0                       // Condition
    else throw failure("not positive"); // Statement if condition is false
    return x+1.0                           // Statement if condition is true
```

The else part of the statement can be omitted in which case Ergo
returns an error by default.

```ergo
    enforce x >= 0.0;          // Condition
    return x+1.0               // Statement if condition is true
```

## Emitting obligations

When inside a clause or contract, you can emit (one or more) obligations as
follows:
```ergo
   emit PaymentObligation{ amount: 29.99, description: "12 red roses" };
   emit PaymentObligation{ amount: 19.99, description: "12 white tulips" };
   return
```

Note that `emit` is always terminated by a `;` followed by another statement.

To conditionally emit an obligation you must ensure that both the `then` and the
`else` branches
in your Ergo code have a `return` value. For example:

```ergo
  let response = VehiclePaymentResponse{ message: "A missed payment was declared
for " ++
      contract.buyer.partyId ++ ". There have been " ++ toString(newCounter) ++ "
missed payments." };

    if state.numMissedPayments > contract.numMissedPayments then
      emit DisableVehicle {
        vehicleId : contract.vehicleId,
        contract: contract,
        deadline: none,
        promisor: some(contract.buyer),
        promisee: some(contract.seller),
        numMissedPayments : newCounter
      };
      return response
    else
      return response
```

## Setting the contract state

When inside a clause or contract, you can create a contract state as follows:
```ergo
    set state InstallmentSaleState{
      stateId: "#1",
      status: "WaitingForFirstDayOfNextMonth",
      balance_remaining: contract.INITIAL_DUE,
      total_paid: 0.0,
      next_payment_month: contract.FIRST_MONTH
    };
    return
```

Note that `set state` is always terminated by a `;` followed by another statement.

Once the state is set, you can change its properties individually with the shorter:
```ergo
set state.total_paid = 100.0;
return
```

## Printing intermediate results

 For debugging purposes a special `info` statement can be used in your contract logic. For instance, the following indicates that you would like the Ergo execution engine to print out the result of expression `state.status` on the standard output.

 ```ergo
    set state InstallmentSaleState{
      stateId: "#1",
      status: "WaitingForFirstDayOfNextMonth",
      balance_remaining: contract.INITIAL_DUE,
      total_paid: 0.0,
      next_payment_month: contract.FIRST_MONTH
    };
    info(state.status);     // Directive to print to standard output
    return
```

---
id: version-0.21-markup-ciceromark
title: CiceroMark
original_id: markup-ciceromark

CiceroMark is an extension to CommonMark used to write the text in Accord Project contracts. The extension is limited in scope, and designed to help with parsing clauses inside contracts and the result of formulas.

## Blocks

### Clause Blocks

Clause blocks can be used to identify a specific clause within a contract. Contract blocks are written:
```
{{#clause clauseName}}
...Markdown of the clause...
{{/clause}}
```

### Example

For instance, the following is a valid contract text containing a payment clause:

```tem
## Copyright Notices.

Licensee shall ensure that its use of the Work is marked with the appropriate
copyright notices specified by Licensor in a reasonably prominent position in the
order and manner provided by Licensor. Licensee shall abide by the copyright laws
and what are considered to be sound practices for copyright notice provisions in
the Territory. Licensee shall not use any copyright notices that conflict with,
confuse, or negate the notices Licensor provides and requires hereunder.

{{#clause payment}}
Payment
-------
As consideration in full for the rights granted herein, Licensee shall pay Licensor
a one-time
fee in the amount of "one hundred US Dollars" (100.0 USD) upon execution of this
Agreement, payable as
follows: "bank transfer".
{{/clause}}

## General.

### Interpretation.

For purposes of this Agreement, (a) the words "include," "includes," and
"including" are deemed to be followed by the words "without limitation"; (b) the
word "or" is not exclusive; and (c) the words "herein," "hereof," "hereby,"
"hereto," and "hereunder" refer to this Agreement as a whole. This Agreement is
intended to be construed without regard to any presumption or rule requiring
construction or interpretation against the party drafting an instrument or causing
any instrument to be drafted.
```

## Ergo Formulas

Ergo formulas in template text are essentially similar to Excel formulas. They let you create legal text dynamically based on the other variables in your contract.

If your contract contains the result of evaluating a formula, the corresponding

~~~~
text should be written `{{% resultOfFormula %}}` where `resultOfFormula` is the
expected result of that formula.

### Example

For instance, the following is a valid contract text for the [fixed rate loan]
(https://templates.accordproject.org/fixed-interests@0.5.2.html) template:

```tem
## Fixed rate loan

This is a _fixed interest_ loan to the amount of £100,000.00
at the yearly interest rate of 2.5%
with a loan term of 15,
and monthly payments of {{%£667.00%}}
```

--------------------------------------------------------------------------------
---
id: version-0.21-markup-commonmark
title: CommonMark
original_id: markup-commonmark
---

The following CommonMark guide is non normative, but included for convenience. For
a more detailed introduction we refer the reader the [CommonMark
Webpage](https://commonmark.org/) and
[Specification](https://spec.commonmark.org/0.29/).

## Formatting

### Italics

To italicize text, add one asterisk `*` or underscore `_` both before and after the
relevant text.

##### Example

```md
_Donoghue v Stevenson_ is a landmark tort law case.
```
will be rendered as:

> _Donoghue v Stevenson_ is a landmark tort law case.

### Bold
To bold text, add two asterisks `**` or two underscores `__` both before and after
the relevant text.

##### Example

```md
**Price** is defined in the Appendix.
```

will be rendered as:

> **Price** is defined in the Appendix.
~~~~

### Bold and Italic
To bold _and_ italicize text, add `***` both before and after the relevant text.

##### Example

```md
***WARNING***: This product contains chemicals that may cause cancer.
```
will be rendered as:

> ***WARNING***: This product contains chemicals that may cause cancer.

## Paragraphs
To start a new paragraph, insert one or more blank lines. (In other words, all paragraphs in markdown need to have one or more blank lines between them.)

##### Example

```md
This is the first paragraph.

This is the second paragraph.
This is not a third paragraph.
```
will be rendered as:

>This is the first paragraph.
>
>This is the second paragraph.
>This is not a third paragraph.

## Headings

### Using `#` (ATX Headings)

Level-1 through level-6 headings from are written with a `#` for each level.

#### Example

```md
# US Constitution
## Statutes enacted by Congress
### Rules promulgated by federal agencies
#### State constitution
##### Laws enacted by state legislature
###### Local laws and ordinances
```

will be rendered as:
> <h1>US Constitution</h1>
> <h2>Statutes enacted by Congress</h2>
> <h3>Rules promulgated by federal agencies</h3>
> <h4>State constitution</h4>
> <h5>Laws enacted by state legislature</h5>
> <h6>Local laws and ordinances</h6>

### Using `=` or `-` (Setext Headings)

Alternatively, headings with level 1 or 2 can be represented by using `=` and `-` under the text of the heading.

#### Example

```md
Linux Foundation
================

Accord Project
--------------
```

will be rendered as:
> <h1>Linux Foundation</h1>
> <h2>Accord Project</h2>

## Lists

### Unordered Lists
To create an unordered list, use asterisks `*`, plus `+`, or hyphens `-` in the beginning as list markers.

#### Example

```md
* Cicero
* Ergo
* Concerto
```

Will be rendered as:
>* Cicero
>* Ergo
>* Concerto

### Ordered Lists

To create an ordered list, use numbers followed by a period `.`.

#### Example

```md
1. One
2. Two
3. Three
```

will be rendered as:
>1. One
>2. Two
>3. Three

### Nested Lists

To create a list within another, indent each item in the sublist by four spaces.

#### Example

~~~~
```md
1. Matters related to the business
    - enter into an agreement...
    - enter into any abnormal contracts...
2. Matters related to the assets
    - sell or otherwise dispose...
    - mortage, ...
```

will be rendered as:
>1. Matters related to the business
>    - enter into an agreement...
>    - enter into any abnormal contracts...
>2. Matters related to the assets
>    - sell or otherwise dispose...
>    - mortgage, ...

## Horizontal Rule

A horizontal rule may be used to create a "thematic break" between paragraph-level
elements. In markdown, you can create a thematic break using either of the
following:

* `___`: three consecutive underscores
* `---`: three consecutive dashes
* `***`: three consecutive asterisks

#### Example

```md
___
---
***
```

Will be rendered as:
>___
>
>---
>
>***

## Escaping

Any markdown character that is used for a special purpose may be _escaped_ by
placing a backslash in front of it.

For instance avoid creating bold or italic when using `*` or `_` in a sentence,
place a backslash `\` in the front, like: `\*` or `\_`.

#### Example

```md
This is \_not\_ italics but _this_ is!
```
Will be rendered as:
> This is \_not\_ italics but _this_ is!
~~~~

~~~~
<!--References:
Commonmark official page and tutorial: https://commonmark.org/help/
OpenLaw Beginner's Guide: https://docs.openlaw.io/beginners-guide/
Markdown cheatsheet: https://gist.github.com/jonschlinkert/5854601
Headings example:
http://www.nyc.gov/html/conflicts/downloads/pdf2/municipal_ethics_laws_ny_state/
introduction_to_american_law.pdf
-->

--------------------------------------------------------------------------------
---
id: version-0.21-markup-preliminaries
title: Preliminaries
original_id: markup-preliminaries
---

## Markdown & CommonMark

The text for Accord Project templates is written using markdown. It builds on the
[CommonMark](https://commonmark.org) standard so that any CommonMark document is
valid text for a template or contract.

As with other markup languages, CommonMark can express the document structure
(e.g., headings, paragraphs, lists) and formatting useful for readability (e.g.,
italics, bold, quotations).

The main reference is the [CommonMark
Specification](https://spec.commonmark.org/0.29/) but you can find an overview of
CommonMark main features in the [CommonMark](markup-commonmark) Section of this
guide.

## Accord Project Extensions

Accord Project uses two extensions to CommonMark: CiceroMark for the contract text,
and TemplateMark for the template grammar.

### Lexical Conventions

Accord Project contract or template text is a string of `UTF-8` characters.

:::note
By convention, CiceroMark files have the `.md` extensions, and TemplateMark files
have the `.tem.md` extension.
:::

The two sequences of characters `{{` and `}}` are reserved and used for the
CiceroMark and TemplateMark extensions to CommonMark. There are three kinds of
extensions:
1. Variables (written `{{variableName}}`) which may include an optional formatting
(written `{{variableName as "FORMAT"}}`).
2. Formulas (written `{{% expression %}}`).
3. Blocks which may contain additional text or markdown. Blocks come in two
flavors:
   1. Blocks corresponding to [markdown inline
elements](https://spec.commonmark.org/0.29/#inlines) which may contain only other
markdown inline elements (e.g., text, emphasis, links). Those have to be written on
a single line as follows:
      ```
      {{#blockName variableName}} ... text or markdown ... {{/blockName}}
~~~~

```
   2. Blocks corresponding to [markdown container
elements](https://spec.commonmark.org/0.29/#container-blocks) which may contain
arbitrary markdown elements (e.g., paragraphs, lists, headings). Those have to be
written with each opening and closing tags on their own line as follows:
      ```
      {{#blockName variableName}}
      ... text or markdown ...
      {{/blockBane}}
      ```

CiceroMark

CiceroMark is used to express the natural language text for legal clauses or
contracts. It uses two specific extensions to CommonMark to facilitate contract
parsing:
1. Clauses within a contract can be identified using a `clause` block:
   ```
   {{#clause clauseName}}
   text of the clause
   {{/clause}}
   ```
2. The result of formulas within a contract or clause can be identified using:
   ```
   {{% result_of_the_formula %}}

For instance, the following CiceroMark for a loan between `John Smith` and `Jane
Doe` includes a title (`Loan agreement`) followed by some text, followed by a fixed
rate interest clause. The clause contains the terms for the loan and the result of
calculating the monthly payment.
```tem
# Loan agreement

This is a loan agreement between "John Smith" and "Jane Doe", which shall be
entered into
by the parties on January 21, 2021 - 3 PM, except in the event of a force majeure.

{{#clause fixedRate}}
## Fixed rate loan

This is a _fixed interest_ loan to the amount of £100,000.00
at the yearly interest rate of 2.5%
with a loan term of 15,
and monthly payments of {{%£667.00%}}
{{/clause}}
```

More information and examples can be found in the [CiceroMark](markup-ciceromark)
part of this guide.

### TemplateMark

TemplateMark is used to describe families of contracts or clauses with some
variable parts. It is based on CommonMark with several extensions to indicate those
variables parts:
1. _Variables_: e.g., `{{loanAmount}}` indicates the amount for a loan.
2. _Template Blocks_: e.g., `{{#if forceMajeure}}, except in the event of a force
majeure{{/if}}` indicates some optional text in the contract.

3. _Formulas_: e.g., `{{% monthlyPaymentFormula(loanAmount,rate,loanDuration) %}}` calculates a monthly payment based on the `loanAmount`, `rate`, and `loanDuration` variables.

For instance, the following TemplateMark for a loan between a `borrower` and a `lender` includes a title (`Loan agreement`) followed by some text, followed by a fixed rate interest clause. This template allows for either taking force majeure into account or not, and calls into a formula to calculate the monthly payment.
```tem
# Loan agreement

This is a loan agreement between {{borrower}} and {{lender}}, which shall be entered into
by the parties on {{date as "MMMM DD, YYYY - h A"}}{{#if forceMajeure}}, except in the event of a force majeure{{/if}}.

{{#clause fixedRate}}
## Fixed rate loan

This is a _fixed interest_ loan to the amount of {{loanAmount as "K0,0.00"}}
at the yearly interest rate of {{rate}}%
with a loan term of {{loanDuration}},
and monthly payments of {{% monthlyPaymentFormula(loanAmount,rate,loanDuration) as "K0,0.00" %}}
{{/clause}}
```

More information and examples can be found in the [TemplateMark](markup-templatemark) part of this guide.

## Dingus

You can test your template or contract text using the [TemplateMark Dingus](https://templatemark-dingus.netlify.app), an online tool which lets you edit the markdown and see it rendered as HTML, or as a document object model.

![TemplateMark Dingus](assets/dingus1.png)

You can select whether to parse your text as pure CommonMark (i.e., according to the CommonMark specification), or with the CiceroMark or TemplateMark extensions.

![TemplateMark Dingus](assets/dingus2.png)

You can also inspect the HTML source, or the document object model (abstract syntax tree or AST).

![TemplateMark Dingus](assets/dingus3.png)

For instance, you can open the TemplateMark from the loan example on this page by clicking [this link](https://templatemark-dingus.netlify.app/#md3=%7B%22source%22%3A%22%23%20Loan%20agreement%5Cn%5CnThis%20is%20a%20loan%20agreement%20between%20%7B%7Bborrower%7D%7D%20and%20%7B%7Blender%7D%7D%2C%20which%20shall%20be%20entered%20into%5Cnby%20the%20parties%20on%20%7B%7Bdate%20as%20%5C%22MMMMM%20DD%2C%20YYYY%20-%20hhA%5C%22%7D%7D%7B%7B%23if%20forceMajeure%7D%7D%2C%20except%20in%20the%20event%20of%20a%20force%20majeure%7B%7B%2Fif%7D%7D.%5Cn%5Cn%7B%7B%23clause%20fixedRate%7D%7D%5Cn%23%23%20Fixed%20rate%20loan%5Cn%5CnThis%20is%20a%20_fixed%20interest_%20loan%20to%20the%20amount%20of%20%7B%7BloanAmount%20as%20%5C%22K0%2C0.00%5C%22%7D%7D%5Cnat%20the%20yearly%20interest%20rate%20of%20%7B%7Brate%7D%7D%25%5Cnwith%20a%20loan%20term%20of%20%7B%7BloanDuration%7D%7D%2C%5Cnand

%20monthly%20payments%20of%20%7B%7B%25%20monthlyPaymentFormula%28loanAmount%2Crate
%2CloanDuration%29%20as%20%5C%22K0%2C0.00%5C%22%20%25%7D%7D%5Cn%7B%7B%2Fclause%7D
%7D%5Cn%22%2C%22defaults%22%3A%7B%22templateMark%22%3Atrue%2C%22ciceroMark
%22%3Afalse%2C%22html%22%3Atrue%2C%22_highlight%22%3Atrue%2C%22_strict%22%3Afalse
%2C%22_view%22%3A%22html%22%7D%7D).

![TemplateMark Dingus](assets/dingus4.png)

---
id: version-0.21-markup-templatemark
title: TemplateMark
original_id: markup-templatemark

TemplateMark is an extension to CommonMark used to write the text in Accord Project templates. The extension includes new markdown for variables, inline and container elements of the markdown and template formulas.

The kind of extension which can be used is based on the _type_ of the variable in the [Concerto Model](model-concerto) for your template. For each type in your model differrent markdown elements apply: variable markdown for atomic types in the model, list blocks for array types in the model, optional blocks for optional types in the model, etc.

## Variables

Standard variables are written `{{variableName}}` where `variableName` is a variable declared in the model.

The following example shows a template text with three variables (`buyer`, `amount`, and `seller`):

```tem
Upon the signing of this Agreement, {{buyer}} shall pay {{amount}} to {{seller}}.
```

The way variables are handled (both during parsing and drafting) is based on their type.

### String Variable

#### Description

If the variable `variableName` has type `String` in the model:
```ergo
o String variableName
```
The corresponding instance should contain text between quotes (`"`).

#### Examples

For example, consider the following model:

```ergo
asset Template extends AccordClause {
  o String buyer
  o String supplier
```

~~~text
}
```

the following instance text:
```md
This Supply Sales Agreement is made between "Steve Supplier" and "Betty Byer".
```

matches the template:
```tem
This Supply Sales Agreement is made between {{supplier}} and {{buyer}}.
```

while the following instance texts do not match:
```md
This Supply Sales Agreement is made between 2019 and 2020.
```
or
```md
This Supply Sales Agreement is made between Steve Supplier and Betty Byer.
```

### Numeric Variable

#### Description

If the variable `variableName` has type `Double`, `Integer` or `Long` in the model:
```ergo
o Double variableName
o Integer variableName2
o Long variableName3
```
The corresponding instance should contain the corresponding number.

#### Examples

For example, consider the following model:

```ergo
asset Template extends AccordClause {
  o Double penaltyPercentage
}
```

the following instance text:
```md
The penalty amount is 10.5% of the total value of the Equipment whose delivery has
been delayed.
```

matches the template:
```tem
The penalty amount is {{penaltyPercentage}}% of the total value of the Equipment
whose delivery has been delayed.
```

while the following instance texts do not match:
```md
The penalty amount is ten% of the total value of the Equipment whose delivery has
~~~

been delayed.
~~~
or
```md
The penalty amount is "10.5"% of the total value of the Equipment whose delivery
has been delayed.
~~~

### Enum Variables

#### Description

If the variable `variableName` has an enumerated type:
```ergo
o EnumType variableName
```

The corresponding instance should contain a corresponding enumerated value without
quotes.

#### Examples

For example, consider the following model:
```ergo
import org.accordproject.money.CurrencyCode from
https://models.accordproject.org/money.cto
asset Template extends AccordClause {
  o CurrencyCode currency
}

```
the following instance text:
```md
Monetary amounts in this contract are denominated in USD.
```

matches the template:
```tem
Monetary amounts in this contract are denominated in {{currency}}.
```

while the following instance texts do not match:
```md
Monetary amounts in this contract are denominated in "USD".
```
or
```md
Monetary amounts in this contract are denominated in $.
```

## Formatted Variables

Formatted variables are written `{{variableName as "FORMAT"}}` where `variableName`
is a variable declared in the model and the `FORMAT` is a type-dependent
description for the syntax of the variables in the contract.

The following example shows a template text with one variable with a format
`DD/MM/YYYY`.

```tem
The contract was signed on {{contractDate as "DD/MM/YYYY"}}.
```

### DateTime Variables

#### Description

If the variable `variableName` has type `DateTime`:
```ergo
o DateTime variableName
```
The corresponding instance should be a date and time, and can optionally be formatted. The default format is `MM/DD/YYYY`, commonly used in the US.

#### DateTime Formats

The textual representation of a DateTime can be customized by including an optional format string using the `as` keyword directly in a template grammar. The following formatting tokens are supported:

Tokens are case-sensitive.

Input	Example	Description
`YYYY`	`2014`	4 or 2 digit year
`M`	`12`	1 or 2 digit month number
`MM`	`04`	2 digit month number
`MMM`	`Feb.`	Short month name
`MMMM`	`December`	Long month name
`D`	`3`	1 or 2 digit day of month
`DD`	`04`	2 digit day of month
`H`	`3`	24 hours (1 or 2 digits)
`HH`	`04`	24 hours (2 digits)
`h`	`1`	12 hours (1 or 2 digits)
`hh`	`02`	12 hours (2 digits)
`a`	`am` or `pm`	morning/afternoon (lowercase)
`A`	`AM` or `PM`	morning/afternoon (uppercase)
`mm`	`59`	2 digit minutes
`ss`	`34`	2 digit seconds
`SSS`	`002`	3 digit milliseconds
`Z`	`+01:00`	UTC offset

:::note
If `Z` is specified, it must occur as the last token in the format string.
:::

#### Examples

The format of the `contractDate` variable of type `DateTime` can be specified with the `DD/MM/YYYY` format, as is commonly used in Europe.

```tem
The contract was signed on {{contractDate as "DD/MM/YYYY"}}.
The contract was signed on 26/04/2019.
```

Other examples:

```tem
dateTimeProperty: {{dateTimeProperty as "D MMM YYYY HH:mm:ss.SSSZ"}}
dateTimeProperty: 1 Jan 2018 05:15:20.123+01:02
```

```tem
dateTimeProperty: {{dateTimeProperty as "D MMMM YYYY HH:mm:ss.SSSZ"}}
dateTimeProperty: 1 January 2018 05:15:20.123+01:02
```

```tem
dateTimeProperty: {{dateTimeProperty as "D-M-YYYY H mm:ss.SSSZ"}}
dateTimeProperty: 31-12-2019 2 59:01.001+01:01
```

```tem
dateTimeProperty: {{dateTimeProperty as "DD/MM/YYYY"}}
dateTimeProperty: 01/12/2018
```

```tem
dateTimeProperty: {{dateTimeProperty as "DD-MMM-YYYY H mm:ss.SSSZ"}}
dateTimeProperty: 04-Jan-2019 2 59:01.001+01:01
```

### Amount Variables

#### Description

If the variable `variableName` is of type `Integer`, `Long`, `Double` or `MonetaryAmount`:
```ergo
o Integer integerVariable
o Long longVariable
o Double doubleVariable
o MonetaryAmount monetaryVariable
```

The corresponding instance should be a numeric value (with a currency code in the case of monetary amounts), and can optionally be formatted.

#### Amount Formats

The textual representation of an amount can be customized by including an optional format string using the `as` keyword directly in a template grammar. The following formatting tokens are supported:

Tokens are case-sensitive.

Input	Example	Description	Type Supported
`0,0`	`3,100,200`	integer part with `,` separator	Integer,Long,Double,MonetaryAmount
`0 0`	`3 100 200`	integer part with ` ` separator	Integer,Long,Double,MonetaryAmount
`0,0.00`	`3,100,200.95`	decimal with two digits precision	Double,MonetaryAmount

`0 0,00`	`3 100 200,95`	decimal with two digits precision	Double,MonetaryAmount
`0,0.0000`	`3,100,200.95`	decimal with four digits precision	Double,MonetaryAmount
`CCC`	`USD`	currency code	MonetaryAmount
`K`	`$`	currency symbol	MonetaryAmount

The general format for the amount is `0{sep}0({sep}0+)?` where `{sep}` is a single character (e.g., `,` or `.`). The first `{sep}` is used to separate every three digits of the integer part. The second `{sep}` is used as a decimal point. And the number of `0` after the second separator is used to indicate precision (number of digits after the decimal point).

#### Examples

The following examples show formating for `Integer` or `Long` values.

```
The manuscript shall be completed within {{days as "0,0"}} days.
The manuscript shall be completed within 1,001 days.
```

```
The manuscript shall contain at most {{words as "0 0"}} words.
The manuscript shall contain at most 1 500 001 words.
```

The following examples show formatting for `Double` values.

```
The effective range of the device should be at least {{distance as "0,0.00mm"}}.
The effective range of the device should be at least 1,250,400.99mm.
```

```
The effective range of the device should be at least {{distance as "0 0,0000mm"}}.
The effective range of the device should be at least 1 250 400,9900mm.
```

The following examples show formatting for `MonetaryAmount` values.

```
The loan principal is {{principal as "0,0.00 CCC"}}.
The loan principal is 2,000,500,000.00 GBP.
```

```
The loan principal is {{principal as "K0,0.00"}}.
The loan principal is £2,000,500,000.00.
```

```
The loan principal is {{principal as "0 0,00 K"}}.
The loan principal is 2 000 500 000,00 €.
```

## Complex Types Variables

### Duration Types

#### Description

If the variable `variableName` has type `Duration`:
```ergo
import org.accordproject.time.Duration
o Duration variableName
```

The corresponding instance should contain the corresponding duration written with
the amount as a number and the duration unit as literal text.

#### Examples

For example, consider the following model:
```ergo
asset Template extends AccordClause {
  o Duration termination
}
```

the following instance texts:
```md
If the delay is more than 15 days, the Buyer is entitled to terminate this
Contract.
```
and
```md
If the delay is more than 1 week, the Buyer is entitled to terminate this Contract.
```

both match the template:
```tem
If the delay is more than {{termination}}, the Buyer is entitled to terminate this
Contract.
```

while the following instance texts do not match:
```md
If the delay is more than a month, the Buyer is entitled to terminate this
Contract.
```
or
```md
If the delay is more than "two weeks", the Buyer is entitled to terminate this
Contract.
```

### Other Complex Types

#### Description

If the variable `variableName` has a complex type `ComplexType` (such as an
`asset`, a `concept`, etc.)
```ergo
o ComplexType variableName
```

~~~
```

The corresponding instance should contain all fields in the corresponding complex
type in the order they occur in the model, separated by a single white space
character.

#### Examples

For example, consider the following model:
```ergo
import org.accordproject.address.PostalAddress from
https://models.accordproject.org/address.cto
asset Template extends AccordClause {
  o PostalAddress address
}
```

the following instance text:
```md
Address of the supplier: "555 main street" "10290" "" "NY" "New York" "10001".
```

matches the template:
```tem
Address of the supplier: {{address}}.
```

Consider the following model:
```md
import org.accordproject.money.MonetaryAmount from
https://models.accordproject.org/money.cto
asset Template extends AccordClause {
  o MonetaryAmount amount
}
```

the following instance text:
```md
Total value of the goods: 50.0 USD.
```

matches the template:
```tem
Total value of the goods: {{amount}}.
```

## Inline Blocks

CiceroMark uses blocks to enable more advanced scenarios, to handle optional or
repeated text (e.g., lists), to change the variables in scope for a given section
of the text, etc. Inline blocks correspond to inline elements in the markdown.

Inline blocks always have the following syntactic structure:

```tem
{{#blockName variableName parameters}}...{{/blockName}}
```

where `blockName` indicates which kind of block it is (e.g., conditional block or
~~~

optional block), `variableName` indicates the template variable which is in scope within the block. For certain blocks, additional `parameters` can be passed to control the behavior of that block (e.g., the `join` block creates text from a list with an optional separator).

### Conditional Blocks

Conditional blocks enables text which depends on a value of a `Boolean` variable in your model:

```tem
{{#if forceMajeure}}This is a force majeure{{/if}}
```

Conditional blocks can also include an `else` branch to indicate that some other text should be use when the value of the variable is `false`:

```tem
{{#if forceMajeure}}This is a force majeure{{else}}This is *not* a force
majeure{{/if}}
```

#### Examples

Drafting text with the first conditional block above using the following JSON data:
```json
{
  "$class": "org.accordproject.foo.Status",
  "forceMajeure": true
}
```

results in the following markdown text:

```md
This is a force majeure
```

Drafting text with this block using the following JSON data:
```json
{
  "$class": "org.accordproject.foo.Status",
  "forceMajeure": false
}
```

results in the following markdown text:

```md

```

### Optional Blocks

Optional blocks enable text which depends on the presence or absence of an `optional` variable in your model:

```tem
{{#optional forceMajeure}}This applies except for Force Majeure cases in a
```

~~~~
{{miles}} miles radius.{{/optional}}
```

Optional blocks can also include an `else` branch to indicate that some other text
should be use when the value of the variable is absent (`null` in the JSON data):

```tem
{{#optional forceMajeure}}This applies except for Force Majeure cases in a
{{miles}} miles radius.{{else}}This applies even in case a force
majeure.{{/optional}}
```

#### Examples

Drafting text with the second optional block above using the following JSON data:
```json
{
  "$class": "org.accordproject.foo.Status",
  "forceMajeure": {
    "$class": "org.accordproject.foo.Distance",
    "miles": 250
  }
}
```

results in the following markdown text:

```md
This applies except for Force Majeure cases in a 250 miles radius.
```

Drafting text with this block using the following JSON data:
```json
{
  "$class": "org.accordproject.foo.Status",
  "forceMajeure": null
}
```

results in the following markdown text:

```md
This applies even in case a force majeure.
```

### With Blocks

A `with` block can be used to change variables that are in scope in a specific part
of a template grammar:

```tem
For the Tenant: {{#with tenant}}{{partyId}}, domiciled at {{address}}{{/with}}
For the Landlord: {{#with landlord}}{{partyId}}, domiciled at {{address}}{{/with}}
```

#### Example

Drafting text with this block using the following JSON data:
```json
~~~~

~~~~
{
  "$class": "org.accordproject.rentaldeposit.RentalDepositClause",
  "contractId": "31d817e2-d62a-4b70-b395-acd0d5da09f5",
  "tenant": {
    "$class": "org.accordproject.rentaldeposit.RentalParty",
    "partyId": "Michael",
    "address": "111, main street"
  }
  ...
}
```

results in the following markdown text:

```md
For the Tenant: "Michael", domiciled at "111, main street"
For the Landlord: "Parsa", domiciled at "222, chestnut road"
```

### Join Blocks

A `join` block can be used to iterate over a variable containing an array of
values, and can use an (optional) separator.

```tem
Discount applies to the following items: {{#join items separator=", "}}{{name}}
({{id}}){{/join}}.
```

#### Example

Drafting text with this block using the following JSON data:
```json
{
  "$class": "org.accordproject.sale.Order",
  "contractId": "31d817e2-d62a-4b70-b395-acd0d5da09f5",
  "items": [{
      "$class": "org.accordproject.slate.Item",
      "id": "111",
      "name": "Pineapple"
    },{
      "$class": "org.accordproject.slate.Item",
      "id": "222",
      "name": "Strawberries"
    },{
      "$class": "org.accordproject.slate.Item",
      "id": "333",
      "name": "Pomegranate"
    }
  ]
}
```

results in the following markdown text:

```md
Discount applies to the following items: Pineapple (111), Strawberries (222),
Pomegranate (333).
```
~~~~

## Container Blocks

CiceroMark uses block expressions to enable more advanced scenarios, to handle optional or repeated text (e.g., lists), to change the variables in scope for a given section of the text, etc.

Container blocks always have the following syntactic structure:

```tem
{{#blockName variableName parameters}}
...
{{/blockName}}
```

where `blockName` indicates which kind of block it is (e.g., conditional block or list block), `variableName` indicates the template variable which is in scope within the block. For certain blocks, additional `parameters` can be passed to control the behavior of that block (e.g., the `join` block creates text from a list with an optional separator).

### Unordered Lists

```tem
{{#ulist rates}}
{{volumeAbove}}$ M<= Volume < {{volumeUpTo}}$ M : {{rate}}%
{{/ulist}}
```

#### Example

Drafting text with this block using the following JSON data:
```json
{
  "$class": "org.accordproject.volumediscountlist.VolumeDiscountContract",
  "contractId": "19243313-adc2-4ff1-aa41-993816ed2cdc",
  "rates": [
    {
      "$class": "org.accordproject.volumediscountlist.RateRange",
      "volumeUpTo": 1,
      "volumeAbove": 0,
      "rate": 3.1
    },
    {
      "$class": "org.accordproject.volumediscountlist.RateRange",
      "volumeUpTo": 10,
      "volumeAbove": 1,
      "rate": 3.1
    },
    {
      "$class": "org.accordproject.volumediscountlist.RateRange",
      "volumeUpTo": 50,
      "volumeAbove": 10,
      "rate": 2.9
    }
  ]
}
```

results in the following markdown text:

```md
- 0.0$ M <= Volume < 1.0$ M : 3.1%
- 1.0$ M <= Volume < 10.0$ M : 3.1%
- 10.0$ M <= Volume < 50.0$ M : 2.9%
```

### Ordered Lists

```tem
{{#olist rates}}
{{volumeAbove}}$ M <= Volume < {{volumeUpTo}}$ M : {{rate}}%
{{/olist}}
```

#### Example

Drafting text with this block using the following JSON data:
```json
{
  "$class": "org.accordproject.volumediscountlist.VolumeDiscountContract",
  "contractId": "19243313-adc2-4ff1-aa41-993816ed2cdc",
  "rates": [
    {
      "$class": "org.accordproject.volumediscountlist.RateRange",
      "volumeUpTo": 1,
      "volumeAbove": 0,
      "rate": 3.1
    },
    {
      "$class": "org.accordproject.volumediscountlist.RateRange",
      "volumeUpTo": 10,
      "volumeAbove": 1,
      "rate": 3.1
    },
    {
      "$class": "org.accordproject.volumediscountlist.RateRange",
      "volumeUpTo": 50,
      "volumeAbove": 10,
      "rate": 2.9
    }
  ]
}
```

results in the following markdown text:
```md
1. 0.0$ M <= Volume < 1.0$ M : 3.1%
2. 1.0$ M <= Volume < 10.0$ M : 3.1%
3. 10.0$ M <= Volume < 50.0$ M : 2.9%
```

### Clause Blocks

Clause blocks can be used to include a clause template within a contract template:

```tem
Payment
```

~~~~
-------
{{#clause payment}}
As consideration in full for the rights granted herein, Licensee shall pay Licensor
a one-time
fee in the amount of {{amountText}} ({{amount}}) upon execution of this Agreement,
payable as
follows: {{paymentProcedure}}.
{{/clause}}
```

#### Example

Drafting text with this block using the following JSON data:
```json
{
  "$class": "org.accordproject.copyrightlicense.CopyrightLicenseContract",
  "contractId": "944535e8-213c-4649-9e60-cc062cce24e8",
  ...
  "paymentClause": {
    "$class": "org.accordproject.copyrightlicense.PaymentClause",
    "clauseId": "6c7611dc-410c-4134-a9ec-17fb6aad5607",
    "amountText": "one hundred US Dollars",
    "amount": {
      "$class": "org.accordproject.money.MonetaryAmount",
      "doubleValue": 100,
      "currencyCode": "USD"
    },
    "paymentProcedure": "bank transfer"
  }
}
```

results in the following markdown text:

```md
Payment
----

As consideration in full for the rights granted herein, Licensee shall pay Licensor
a one-time
fee in the amount of "one hundred US Dollars" (100.0 USD) upon execution of this
Agreement, payable as
follows: "bank transfer".

```

## Ergo Formulas

Ergo formulas in template text are essentially similar to Excel formulas, and
enable to create legal text dynamically, based on the other variables in your
contract. They are written `{{% ergoExpression %}}` where `ergoExpression` is any
valid [Ergo Expression](logic-ergo).

::: note
Formulas allow the template developer to generate arbitrary contract text from
other contract and clause variables. They therefore cannot be used to set a
template model variable during parsing. In other words formulas are evaluated when
drafting a contract but are ignored when parsing the contract text.
:::
~~~~

### Evaluation Context

The context in which expressions within templates text are evaluated includes:
- The contract variables, which can be accessed using the variable name (or
`contract.variableName`)
- All constants or functions declared or imported in the main [Ergo module](logic-
module) for your template.

#### Fixed Interests Clause

For instance, let us look one more time at [fixed rate
loan](https://templates.accordproject.org/fixed-interests-static@0.2.0.html) clause
that was used previously:

```tem
## Fixed rate loan

This is a *fixed interest* loan to the amount of {{loanAmount}}
at the yearly interest rate of {{rate}}%
with a loan term of {{loanDuration}},
and monthly payments of {{% monthlyPaymentFormula(loanAmount,rate,loanDuration) %}}
```

The [`logic` directory](https://github.com/accordproject/cicero-template-library/
tree/master/src/fixed-interests/logic) for that template includes two Ergo modules:
```
./logic/interests.ergo  // Module containing the monthlyPaymentFormula function
./logic/logic.ergo      // Main module
```

A look inside the `logic.ergo` module shows the corresponding import, which ensures
the `monthlyPaymentFormula` function is also in scope in the text for the template:
```
namespace org.accordproject.interests

import org.accordproject.loan.interests.*

contract Interests over TemplateModel {
  ...
}
```

### Examples

Ergo provides a wide range of capabilities which you can use to construct the text
that should be included in the final clause or contract. Below are a few examples
for illustrations, but we encourage you to consult the [Ergo Logic](logic-ergo)
guide for a more comprehensive overview of Ergo.

#### Path expressions

The contents of complex values can be accessed using the `.` notation.

For instance the following template uses the `.` notation to access the first name
and last name of the contract author.

```tem
This contract was drafted by {{% author.name.firstName %}} {{% author.name.lastName
```

~~~~markdown
%}}
```

#### Built-in Functions

Ergo offers a number of pre-defined functions for a variety of primitive types.
Please consult the [Ergo Standard Library](ref-logic-stdlib) reference for the
complete list of built-in functions.

For instance the following template uses the `addPeriod` function to automatically
include the date at which a lease expires in the text of the contract:

```tem
This lease was signed on {{signatureDate}}, and is valid for a {{leaseTerm}}
period.
This lease will expire on {{% addPeriod(signatureDate, leaseTerm) %}}`
```

#### Iterators

Ergo's `foreach` expressions lets you iterate over collections of values.

For instance the following template uses a `foreach` expression combined with the
`avg` built-in function to include the average product price in the text of the
contract:

```tem
The average price of the products included in this purchase
order is {{% avg(foreach p in products return p.price) %}}.
```

#### Conditionals

Conditional expressions lets you include alternative text based on arbitrary
conditions.

For instance, the following template uses a conditional expression to indicate the
governing jurisdiction:

```tem
Each party hereby irrevocably agrees that process may be served on it in
any manner authorized by the Laws of {{%
    if address.country = US and getYear(now()) > 1959
    then "the State of " ++ address.state
    else "the Country of " ++ address.country
%}}
```



--------------------------------------------------------------------------------
---
id: version-0.21-model-api
title: Using the API
original_id: model-api
---

## Install the Core Library

To install the core model library in your project:
~~~~

```
npm install @accordproject/concerto-core --save
```

Below are examples of API use.

## Validating JSON data using a Model

```js
const ModelManager = require('@accordproject/concerto-core').ModelManager;
const Concerto = require('@accordproject/concerto-core').Concerto;
const modelManager = new ModelManager();
modelManager.addModelFile( `namespace org.acme.address
concept PostalAddress {
  o String streetAddress optional
  o String postalCode optional
  o String postOfficeBoxNumber optional
  o String addressRegion optional
  o String addressLocality optional
  o String addressCountry optional
}`, 'model.cto');

const postalAddress = {
    $class : 'org.acme.address.PostalAddress',
    streetAddress : '1 Maine Street'
};
const concerto = new Concerto(modelManager);
concerto.validate(postalAddress);
```

Now try validating this instance:

```
const postalAddress = {
    $class : 'org.acme.address.PostalAddress',
    missing : '1 Maine Street'
};
```

Validation should fail with the message:

```
Instance undefined has a property named missing which is not declared in
org.acme.address.PostalAddress
```

## Runtime introspection of the model

You can use the Concerto `introspect` APIs to retrieve model information at
runtime:

```
const typeDeclaration = concerto.getTypeDeclaration(postalAddress);
const fqn = typeDeclaration.getFullyQualifiedName();
console.log(fqn); // should equal 'org.acme.address.PostalAddress'
```

These APIs allow you to examine the declared properties, super types and meta-
properies for a modelled type.

---
id: version-0.21-model-classes
title: Classes
original_id: model-classes

## Concepts

Concepts are similar to class declarations in most object-oriented languages, in that they may have a super-type and a set of typed properties:

```js
abstract concept Animal {
  o DateTime dob
}

concept Dog extends Animal {
 o String breed
}
```

Concepts can be declared `abstract` if it should not be instantiated (must be subclassed).

## Identity

Concepts may optionally declare an identifying field, using either the `identified by` (explicitly named identity field) or `identified` (`$identifier` system identity field) syntax.

`Person` below is defined to use the `email` property as its identifying field.

```
concept Person identified by email {
  o String email
  o String firstName
  o String lastName
}
```

While `Product` below will use `$identifier` as its identifying field.

```
concept Product identified {
  o String name
  o Double price
}
```

## Assets

An asset is a class declaration that has a single `String` property which acts as an identifier. You can use the `modelManager.getAssetDeclarations` API to look up all assets.

```js
asset Vehicle identified by vin {
  o String vin
}
```

Assets are typically used in your models for the long-lived identifiable Things (or nouns) in the model: cars, orders, shipping containers, products, etc.

## Participants

Participants are class declarations that have a single `String` property acting as an identifier. You can use the `modelManager.getParticipantDeclarations` API to look up all participants.

```js
participant Customer identified by email {
  o String email
}
```

Participants are typically used for the identifiable people or organizations in the model: person, customer, company, business, auditor, etc.

## Transactions

Transactions are similar to participants in that they are also class declarations that have a single `String` property acting as an identifier. You can use the `modelManager.getTransactionDeclarations` API to look up all transactions.

```js
transaction Order identified by orderId {
  o String orderId
}
```

Transactions are typically used in models for the identifiable business events or messages that are submitted by Participants to change the state of Assets: cart check out, change of address, identity verification, place order, etc.

## Events

Events are similar to participants in that they are also class declarations that have a single `String` property acting as an identifier. You can use the `modelManager.getEventDeclarations` API to look up all transactions.

```js
event LateDelivery identified by eventId {
  o String eventId
}
```

Events are typically used in models for the identifiable business events or messages that are emitted by logic to signify that something of interest has occurred.

---
id: version-0.21-model-concerto
title: Concerto Overview
original_id: model-concerto

### Principles

The Concerto Modeling Language (CML) allows you to:
- Define an object-oriented model using a domain-specific language that is much easier to read and write than JSON/XML Schema, XMI or equivalents.
- Optionally edit your models using a powerful [VS Code add-on](https://marketplace.visualstudio.com/items?itemName=accordproject.cicero-vscode-extension) with syntax highlighting and validation
- Create runtime instances of your model
- Serialize your instances to JSON
- Deserialize (and optionally validate) instances from JSON
- Pass JS object instances around your application
- Introspect the model using a powerful set of APIs
- Convert the model to other formats using [concerto-tools](https://github.com/accordproject/concerto/tree/master/packages/concerto-tools)
- Import models from URLs
- Publish your reusable models to any website, including the [Accord Project Open Source model repository](https://models.accordproject.org)

### Metamodel Components

The Concerto metamodel contains the following:
- [Namespaces](model-namespaces)
- [Imports](model-namespaces#imports)
- [Concepts](model-classes#concepts)
- [Assets](model-classes#assets)
- [Participants](model-classes#participants)
- [Transactions](model-classes#transactions)
- [Enumerations](model-enums)
- [Properties & Meta Properties](model-properties)
- [Relationships](model-relationships)
- [Decorators](model-decorators)

---
id: version-0.21-model-decorators
title: Decorators
original_id: model-decorators

Model elements may have arbitrary decorators (aka annotations) placed on them. These are available via API and can be useful for tools to extend the model. Accord Project decorators are defined in the [Decorators Reference](ref-concerto-decorators).

```js
@foo("arg1", 2)
asset Order identified by orderId {
  o String orderId
}
```

Decorators have an arbitrary number of arguments. They support arguments of type:
- String
- Boolean
- Number

- Type reference

Resource definitions and properties may be decorated with 0 or more decorations. Note that only a single instance of a decorator is allowed on each element type. I.e. it is invalid to have the @bar decorator listed twice on the same element.

Decorators are accessible at runtime via the `ModelManager` introspect APIs. This allows tools and utilities to use Concerto to describe a core model, while decorating it with sufficient metadata for their own purposes.

The example below retrieves the 3rd argument to the foo decorator attached to the myField property of a class declaration:

```js
const val = myField.getDecorator('foo').getArguments()[2];
```

---
id: version-0.21-model-properties
title: Properties
original_id: model-properties

Class declarations contain properties. Each property has a type which can either be a type defined in the same namespace, an imported type, or a primitive type.

### Primitive types

Concerto supports the following primitive types:

Type	Description
`String`	a UTF8 encoded String.
`Double`	a double precision 64 bit numeric value.
`Integer`	a 32 bit signed whole number.
`Long`	a 64 bit signed whole number.
`DateTime`	an ISO-8601 compatible time instance, with optional time zone and UTZ offset.
`Boolean`	a Boolean value, either true or false.

### Meta Properties

Property	Description
`[]`	declares that the property is an array
`optional`	declares that the property is not required for the instance to be valid
`default`	declares a default value for the property, if no value is specified
`range`	declares a valid range for numeric properties
`regex`	declares a validation regex for string properties

`String` fields may include an optional regular expression, which is used to validate the contents of the field. Careful use of field validators allows Concerto to perform rich data validation, leading to fewer errors and less boilerplate application code.

The example below validates that a `String` variable starts with `abc`:

```
  o String myString regex=/abc.*/
```

`Double`, `Long` or `Integer` fields may include an optional range expression,
which is used to validate the contents of the field. Both the lower and upper bound
are optional, however at least one must be specified. The upper bound must be
greater than or equal to the lower bound.

```
  o Integer intLowerUpper range=[-1,1] // greater than or equal to -1 and less than
1
  o Integer intLower range=[-1,] // greater than or equal to -1
  o Integer intUpper range=[,1] // less than 1
```

#### Example

```
asset Vehicle {
  o String model default="F150"
  o String make default="FORD"
  o Integer year default=2016 range=[1990,] optional // model year must be 1990 or
higher
  o String V5cID regex=/^[A-z][A-z][0-9]{7}/
}
```

---
id: version-0.21-model-relationships
title: Relationships
original_id: model-relationships

A relationship in Concerto Modeling Language (CML) is a tuple composed of:

1. The namespace of the type being referenced
2. The type name of the type being referenced
3. The identifier of the instance being referenced

Hence a relationship could be: `org.example.Vehicle#123456`

This would be a relationship to the `Vehicle` _type_ declared in the `org.example`
namespace with the _identifier_ `123456`.

> A relationship can be defined to any *identifiable* type, that is a type that has
been declared with either the `identified by` or `identified` properties.

Relationships are unidirectional and deletes do not cascade, ie. removing the
relationship has no impact on the thing that is being pointed to. Removing the
thing being pointed to does not invalidate the relationship.

Relationships must be resolved to retrieve an instance of the object being
referenced. The act of resolution may result in null, if the object no longer
exists or the information in the relationship is invalid. Resolution of
relationships is outside of the scope of Concerto.

A property of a class may be declared as a relationship using the `-->` syntax

instead of the `o` syntax. The `o` syntax declares that the class contains (has-a)
property of that type, whereas the `-->` syntax declares a typed pointer to an
external identifiable instance.

In this example, the model declares that an `Order` has-an array of reference to
`OrderLines`. Deleting the `Order` has no impact on the `OrderLine`. When the
`Order` is serialized the JSON only the IDs of the `OrderLines` are stored within
the `Order`, not the `OrderLines` themselves.

```js
asset OrderLine identified by orderLineId {
  o String orderLineId
  o String sku
}

asset Order identified by orderId {
  o String orderId
  --> OrderLine[] orderlines
}
```

---
id: version-0.21-ref-cicero-api
title: Node.js API
original_id: ref-cicero-api

## Modules

<dl>
<dt><a href="#module_cicero-engine">cicero-engine</a></dt>
<dd><p>Clause Engine</p>
</dd>
<dt><a href="#module_cicero-core">cicero-core</a></dt>
<dd><p>Cicero Core - defines the core data types for Cicero.</p>
</dd>
</dl>

## Classes

<dl>
<dt><a href="#AmountFormatParser">AmountFormatParser</a></dt>
<dd><p>Parses an amount format string</p>
</dd>
<dt><a href="#Clause">Clause</a></dt>
<dd><p>A Clause is executable business logic, linked to a natural language (legally
enforceable) template.
A Clause must be constructed with a template and then prior to execution the data
for the clause must be set.
Set the data for the clause (an instance of the template model) by either calling
the setData method or by
calling the parse method and passing in natural language text that conforms to the
template grammar.</p>
</dd>
<dt><a href="#Contract">Contract</a></dt>
<dd><p>A Contract is executable business logic, linked to a natural language
(legally enforceable) template.
A Clause must be constructed with a template and then prior to execution the data

```
for the clause must be set.
Set the data for the clause (an instance of the template model) by either calling
the setData method or by
calling the parse method and passing in natural language text that conforms to the
template grammar.</p>
</dd>
<dt><a href="#DateTimeFormatParser">DateTimeFormatParser</a></dt>
<dd><p>Parses a date/time format string</p>
</dd>
<dt><a href="#FormatParser">FormatParser</a></dt>
<dd><p>Parses a format string</p>
</dd>
<dt><a href="#Metadata">Metadata</a></dt>
<dd><p>Defines the metadata for a Template, including the name, version, README
markdown.</p>
</dd>
<dt><a href="#MonetaryAmountFormatParser">MonetaryAmountFormatParser</a></dt>
<dd><p>Parses a monetary/amount format string</p>
</dd>
<dt><a href="#ParserManager">ParserManager</a></dt>
<dd><p>Generates and manages a Nearley parser for a template.</p>
</dd>
<dt><a href="#Template">Template</a></dt>
<dd><p>A template for a legal clause or contract. A Template has a template model,
request/response transaction types,
a template grammar (natural language for the template) as well as Ergo code for the
business logic of the
template.</p>
</dd>
<dt><a href="#TemplateInstance">TemplateInstance</a></dt>
<dd><p>A TemplateInstance is an instance of a Clause or Contract template. It is
executable business logic, linked to
a natural language (legally enforceable) template.
A TemplateInstance must be constructed with a template and then prior to execution
the data for the clause must be set.
Set the data for the TemplateInstance by either calling the setData method or by
calling the parse method and passing in natural language text that conforms to the
template grammar.</p>
</dd>
<dt><a href="#CompositeArchiveLoader">CompositeArchiveLoader</a></dt>
<dd><p>Manages a set of archive loaders, delegating to the first archive
loader that accepts a URL.</p>
</dd>
</dl>

## Functions

<dl>
<dt><a href="#locationOfError">locationOfError(error)</a> ⇒
<code>object</code></dt>
<dd><p>Extract the file location from the parse error</p>
</dd>
<dt><a href="#isPNG">isPNG(buffer)</a> ⇒ <code>Boolean</code></dt>
<dd><p>Checks whether the file is PNG</p>
</dd>
<dt><a href="#getMimeType">getMimeType(buffer)</a> ⇒ <code>Object</code></dt>
<dd><p>Returns the mime-type of the file</p>
</dd>
</dl>
```

<a name="module_cicero-engine"></a>

## cicero-engine
Clause Engine

* [cicero-engine](#module_cicero-engine)
    * [~Engine](#module_cicero-engine.Engine)
        * [new Engine()](#new_module_cicero-engine.Engine_new)
        * [.trigger(clause, request, state, currentTime)](#module_cicero-engine.Engine+trigger) ⇒ <code>Promise</code>
        * [.invoke(clause, clauseName, params, state, currentTime)](#module_cicero-engine.Engine+invoke) ⇒ <code>Promise</code>
        * [.init(clause, currentTime)](#module_cicero-engine.Engine+init) ⇒ <code>Promise</code>
        * [.draft(clause, [options], currentTime)](#module_cicero-engine.Engine+draft) ⇒ <code>Promise</code>
        * [.getErgoEngine()](#module_cicero-engine.Engine+getErgoEngine) ⇒ <code>ErgoEngine</code>

<a name="module_cicero-engine.Engine"></a>

### cicero-engine~Engine
<p>
Engine class. Stateless execution of clauses against a request object, returning a response to the caller.
</p>

Kind: inner class of [<code>cicero-engine</code>](#module_cicero-engine)
Access: public

* [~Engine](#module_cicero-engine.Engine)
    * [new Engine()](#new_module_cicero-engine.Engine_new)
    * [.trigger(clause, request, state, currentTime)](#module_cicero-engine.Engine+trigger) ⇒ <code>Promise</code>
    * [.invoke(clause, clauseName, params, state, currentTime)](#module_cicero-engine.Engine+invoke) ⇒ <code>Promise</code>
    * [.init(clause, currentTime)](#module_cicero-engine.Engine+init) ⇒ <code>Promise</code>
    * [.draft(clause, [options], currentTime)](#module_cicero-engine.Engine+draft) ⇒ <code>Promise</code>
    * [.getErgoEngine()](#module_cicero-engine.Engine+getErgoEngine) ⇒ <code>ErgoEngine</code>

<a name="new_module_cicero-engine.Engine_new"></a>

#### new Engine()
Create the Engine.

<a name="module_cicero-engine.Engine+trigger"></a>

#### engine.trigger(clause, request, state, currentTime) ⇒ <code>Promise</code>
Send a request to a clause for execution

Kind: instance method of [<code>Engine</code>](#module_cicero-engine.Engine)
Returns: <code>Promise</code> - a promise that resolves to a result for the clause

Param	Type	Description
clause	[<code>Clause</code>](#Clause)	the clause
request	<code>object</code>	the request, a JS object that can be deserialized using the Composer serializer.
state	<code>object</code>	the contract state, a JS object that can be deserialized using the Composer serializer.
currentTime	<code>string</code>	the definition of 'now'

<a name="module_cicero-engine.Engine+invoke"></a>

#### engine.invoke(clause, clauseName, params, state, currentTime) ⇒ <code>Promise</code>
Invoke a specific clause for execution

Kind: instance method of [<code>Engine</code>](#module_cicero-engine.Engine)
Returns: <code>Promise</code> - a promise that resolves to a result for the clause

Param	Type	Description
clause	[<code>Clause</code>](#Clause)	the clause
clauseName	<code>string</code>	the clause name
params	<code>object</code>	the clause parameters, a JS object whose fields that can be deserialized using the Composer serializer.
state	<code>object</code>	the contract state, a JS object that can be deserialized using the Composer serializer.
currentTime	<code>string</code>	the definition of 'now'

<a name="module_cicero-engine.Engine+init"></a>

#### engine.init(clause, currentTime) ⇒ <code>Promise</code>
Initialize a clause

Kind: instance method of [<code>Engine</code>](#module_cicero-engine.Engine)
Returns: <code>Promise</code> - a promise that resolves to a result for the clause initialization

Param	Type	Description
clause	[<code>Clause</code>](#Clause)	the clause
currentTime	<code>string</code>	the definition of 'now'

<a name="module_cicero-engine.Engine+draft"></a>

#### engine.draft(clause, [options], currentTime) ⇒ <code>Promise</code>
Generate Text for a clause

Kind: instance method of [<code>Engine</code>](#module_cicero-engine.Engine)
Returns: <code>Promise</code> - a promise that resolves to a result for the clause initialization

Param	Type	Description
clause	[<code>Clause</code>](#Clause)	the clause
[options]	<code>*</code>	text generation options. options.wrapVariables encloses variables and editable sections in '<variable ...' and '/>'
currentTime	<code>string</code>	the definition of 'now'

<a name="module_cicero-engine.Engine+getErgoEngine"></a>

#### engine.getErgoEngine() ⇒ <code>ErgoEngine</code>
Provides access to the underlying Ergo engine.

Kind: instance method of [<code>Engine</code>](#module_cicero-engine.Engine)
Returns: <code>ErgoEngine</code> - the Ergo Engine for this Engine
<a name="module_cicero-core"></a>

## cicero-core
Cicero Core - defines the core data types for Cicero.

<a name="AmountFormatParser"></a>

## AmountFormatParser
Parses an amount format string

Kind: global class
Access: public

* [AmountFormatParser](#AmountFormatParser)
    * _instance_
        * [.addGrammars(grammars, field)](#AmountFormatParser+addGrammars)
        * [.buildFormatRules(formatString)](#AmountFormatParser+buildFormatRules) ⇒
<code>Object</code>
    * _static_
        * [.parseAmountFormatField(field)]
(#AmountFormatParser.parseAmountFormatField) ⇒ <code>string</code>
        * [.amountFormatField(field)](#AmountFormatParser.amountFormatField) ⇒
<code>string</code>

<a name="AmountFormatParser+addGrammars"></a>

### amountFormatParser.addGrammars(grammars, field)
Given current grammar parts, add necessary grammars parts for the format.

Kind: instance method of [<code>AmountFormatParser</code>](#AmountFormatParser)

Param	Type	Description
grammars	<code>Array.&lt;object&gt;</code>	the current grammar parts
field	<code>string</code>	grammar field

<a name="AmountFormatParser+buildFormatRules"></a>

### amountFormatParser.buildFormatRules(formatString) ⇒ <code>Object</code>
Converts a format string to a Nearley action

Kind: instance method of [<code>AmountFormatParser</code>](#AmountFormatParser)
Returns: <code>Object</code> - the tokens and action and name to use for the
Nearley rule

Param	Type	Description
formatString	<code>string</code>	the input format string

<a name="AmountFormatParser.parseAmountFormatField"></a>

### AmountFormatParser.parseAmountFormatField(field) ⇒ <code>string</code>

Given a format field (like 0,0.0) this method returns
a logical name for the field. Note the logical names
have been picked to align with the moment constructor that takes an object.

Kind: static method of [<code>AmountFormatParser</code>](#AmountFormatParser)
Returns: <code>string</code> - the field designator

Param	Type	Description
field	<code>string</code>	the input format field

<a name="AmountFormatParser.amountFormatField"></a>

### AmountFormatParser.amountFormatField(field) ⇒ <code>string</code>
Given a double format field (like 0,0.0) this method returns a new unique field
name

Kind: static method of [<code>AmountFormatParser</code>](#AmountFormatParser)
Returns: <code>string</code> - the field designator

Param	Type	Description
field	<code>string</code>	the input format field

<a name="Clause"></a>

## Clause
A Clause is executable business logic, linked to a natural language (legally
enforceable) template.
A Clause must be constructed with a template and then prior to execution the data
for the clause must be set.
Set the data for the clause (an instance of the template model) by either calling
the setData method or by
calling the parse method and passing in natural language text that conforms to the
template grammar.

Kind: global class
Access: public
<a name="Contract"></a>

## Contract
A Contract is executable business logic, linked to a natural language (legally
enforceable) template.
A Clause must be constructed with a template and then prior to execution the data
for the clause must be set.
Set the data for the clause (an instance of the template model) by either calling
the setData method or by
calling the parse method and passing in natural language text that conforms to the
template grammar.

Kind: global class
Access: public
<a name="DateTimeFormatParser"></a>

## DateTimeFormatParser
Parses a date/time format string

Kind: global class
Access: public

* [DateTimeFormatParser](#DateTimeFormatParser)
    * _instance_
        * [.addGrammars(grammars, field)](#DateTimeFormatParser+addGrammars)
        * [.buildFormatRules(formatString)](#DateTimeFormatParser+buildFormatRules)
⇒ <code>Object</code>
    * _static_
        * [.parseDateTimeFormatField(field)]
(#DateTimeFormatParser.parseDateTimeFormatField) ⇒ <code>string</code>

<a name="DateTimeFormatParser+addGrammars"></a>

### dateTimeFormatParser.addGrammars(grammars, field)
Given current grammar parts, add necessary grammars parts for the format.

Kind: instance method of [<code>DateTimeFormatParser</code>]
(#DateTimeFormatParser)

Param	Type	Description
grammars	<code>Array.&lt;object&gt;</code>	the current grammar parts
field	<code>string</code>	grammar field

<a name="DateTimeFormatParser+buildFormatRules"></a>

### dateTimeFormatParser.buildFormatRules(formatString) ⇒ <code>Object</code>
Converts a format string to a Nearley action

Kind: instance method of [<code>DateTimeFormatParser</code>]
(#DateTimeFormatParser)
Returns: <code>Object</code> - the tokens and action and name to use for the
Nearley rule

Param	Type	Description
formatString	<code>string</code>	the input format string

<a name="DateTimeFormatParser.parseDateTimeFormatField"></a>

### DateTimeFormatParser.parseDateTimeFormatField(field) ⇒ <code>string</code>
Given a format field (like HH or D) this method returns
a logical name for the field. Note the logical names
have been picked to align with the moment constructor that takes an object.

Kind: static method of [<code>DateTimeFormatParser</code>]
(#DateTimeFormatParser)
Returns: <code>string</code> - the field designator

Param	Type	Description
field	<code>string</code>	the input format field

<a name="FormatParser"></a>

## FormatParser
Parses a format string

Kind: global class
Access: public

* [FormatParser](#FormatParser)
    * _instance_
        * [.addGrammars(grammars, field)](#FormatParser+addGrammars)
    * _static_
        * [.buildFormatRules(format)](#FormatParser.buildFormatRules)

<a name="FormatParser+addGrammars"></a>

### formatParser.addGrammars(grammars, field)
Given current grammar parts, add necessary grammars parts for the format.

Kind: instance method of [<code>FormatParser</code>](#FormatParser)

Param	Type	Description
grammars	<code>Array.&lt;object&gt;</code>	the current grammar parts
field	<code>string</code>	grammar field

<a name="FormatParser.buildFormatRules"></a>

### FormatParser.buildFormatRules(format)
Given a format, returns grammar rules to parse that format

Kind: static method of [<code>FormatParser</code>](#FormatParser)

Param	Type	Description
format	<code>string</code>	the format
	<code>Array.&lt;object&gt;</code>	grammar rules for the format

<a name="Metadata"></a>

## Metadata
Defines the metadata for a Template, including the name, version, README markdown.

Kind: global class
Access: public

* [Metadata](#Metadata)
    * [new Metadata(packageJson, readme, samples, request, logo)]
(#new_Metadata_new)
    * _instance_
        * [.getTemplateType()](#Metadata+getTemplateType) ⇒ <code>number</code>
        * [.getLogo()](#Metadata+getLogo) ⇒ <code>Buffer</code>
        * [.getAuthor()](#Metadata+getAuthor) ⇒ <code>*</code>
        * [.getRuntime()](#Metadata+getRuntime) ⇒ <code>string</code>
        * [.getCiceroVersion()](#Metadata+getCiceroVersion) ⇒ <code>string</code>
        * [.satisfiesCiceroVersion(version)](#Metadata+satisfiesCiceroVersion) ⇒
<code>string</code>
        * [.getSamples()](#Metadata+getSamples) ⇒ <code>object</code>
        * [.getRequest()](#Metadata+getRequest) ⇒ <code>object</code>
        * [.getSample(locale)](#Metadata+getSample) ⇒ <code>string</code>
        * [.getREADME()](#Metadata+getREADME) ⇒ <code>String</code>
        * [.getPackageJson()](#Metadata+getPackageJson) ⇒ <code>object</code>
        * [.getName()](#Metadata+getName) ⇒ <code>string</code>
        * [.getDisplayName()](#Metadata+getDisplayName) ⇒ <code>string</code>
        * [.getKeywords()](#Metadata+getKeywords) ⇒ <code>Array</code>
        * [.getDescription()](#Metadata+getDescription) ⇒ <code>string</code>

```
        * [.getVersion()](#Metadata+getVersion) ⇒ <code>string</code>
        * [.getIdentifier()](#Metadata+getIdentifier) ⇒ <code>string</code>
        * [.createTargetMetadata(runtimeName)](#Metadata+createTargetMetadata) ⇒
<code>object</code>
        * [.toJSON()](#Metadata+toJSON) ⇒ <code>object</code>
    * _static_
        * [.checkImage(buffer)](#Metadata.checkImage)
        * [.checkImageDimensions(buffer, mimeType)](#Metadata.checkImageDimensions)

<a name="new_Metadata_new"></a>

### new Metadata(packageJson, readme, samples, request, logo)
Create the Metadata.
<p>
<strong>Note: Only to be called by framework code. Applications should
retrieve instances from [Template](#Template)</strong>
</p>


| Param | Type | Description |
| --- | --- | --- |
| packageJson | <code>object</code> | the JS object for package.json (required) |
| readme | <code>String</code> | the README.md for the template (may be null) |
| samples | <code>object</code> | the sample markdown for the template in different
locales, |
| request | <code>object</code> | the JS object for the sample request |
| logo | <code>Buffer</code> | the bytes data for the image represented as an
object whose keys are the locales and whose values are the sample markdown. For
example:  {      default: 'default sample markdown',      en: 'sample text in
english',      fr: 'exemple de texte français'  } Locale keys (with the exception
of default) conform to the IETF Language Tag specification (BCP 47). THe `default`
key represents sample template text in a non-specified language, stored in a file
called `sample.md`. |

<a name="Metadata+getTemplateType"></a>

### metadata.getTemplateType() ⇒ <code>number</code>
Returns either a 0 (for a contract template), or 1 (for a clause template)

**Kind**: instance method of [<code>Metadata</code>](#Metadata)
**Returns**: <code>number</code> - the template type
<a name="Metadata+getLogo"></a>

### metadata.getLogo() ⇒ <code>Buffer</code>
Returns the logo at the root of the template

**Kind**: instance method of [<code>Metadata</code>](#Metadata)
**Returns**: <code>Buffer</code> - the bytes data of logo
<a name="Metadata+getAuthor"></a>

### metadata.getAuthor() ⇒ <code>\*</code>
Returns the author for this template.

**Kind**: instance method of [<code>Metadata</code>](#Metadata)
**Returns**: <code>\*</code> - the author information
<a name="Metadata+getRuntime"></a>

### metadata.getRuntime() ⇒ <code>string</code>
Returns the name of the runtime target for this template, or null if this template
```

```
has not been compiled for a specific runtime.

**Kind**: instance method of [<code>Metadata</code>](#Metadata)
**Returns**: <code>string</code> - the name of the runtime
<a name="Metadata+getCiceroVersion"></a>

### metadata.getCiceroVersion() ⇒ <code>string</code>
Returns the version of Cicero that this template is compatible with.
i.e. which version of the runtime was this template built for?
The version string conforms to the semver definition

**Kind**: instance method of [<code>Metadata</code>](#Metadata)
**Returns**: <code>string</code> - the semantic version
<a name="Metadata+satisfiesCiceroVersion"></a>

### metadata.satisfiesCiceroVersion(version) ⇒ <code>string</code>
Only returns true if the current cicero version satisfies the target version of
this template

**Kind**: instance method of [<code>Metadata</code>](#Metadata)
**Returns**: <code>string</code> - the semantic version

| Param | Type | Description |
| --- | --- | --- |
| version | <code>string</code> | the cicero version to check against |

<a name="Metadata+getSamples"></a>

### metadata.getSamples() ⇒ <code>object</code>
Returns the samples for this template.

**Kind**: instance method of [<code>Metadata</code>](#Metadata)
**Returns**: <code>object</code> - the sample files for the template
<a name="Metadata+getRequest"></a>

### metadata.getRequest() ⇒ <code>object</code>
Returns the sample request for this template.

**Kind**: instance method of [<code>Metadata</code>](#Metadata)
**Returns**: <code>object</code> - the sample request for the template
<a name="Metadata+getSample"></a>

### metadata.getSample(locale) ⇒ <code>string</code>
Returns the sample for this template in the given locale. This may be null.
If no locale is specified returns the default sample if it has been specified.

**Kind**: instance method of [<code>Metadata</code>](#Metadata)
**Returns**: <code>string</code> - the sample file for the template in the given
locale or null

| Param | Type | Default | Description |
| --- | --- | --- | --- |
| locale | <code>string</code> | <code>null</code> | the IETF language code for the
language. |

<a name="Metadata+getREADME"></a>

### metadata.getREADME() ⇒ <code>String</code>
Returns the README.md for this template. This may be null if the template does not
```

```
have a README.md

**Kind**: instance method of [<code>Metadata</code>](#Metadata)
**Returns**: <code>String</code> - the README.md file for the template or null
<a name="Metadata+getPackageJson"></a>

### metadata.getPackageJson() ⇒ <code>object</code>
Returns the package.json for this template.

**Kind**: instance method of [<code>Metadata</code>](#Metadata)
**Returns**: <code>object</code> - the Javascript object for package.json
<a name="Metadata+getName"></a>

### metadata.getName() ⇒ <code>string</code>
Returns the name for this template.

**Kind**: instance method of [<code>Metadata</code>](#Metadata)
**Returns**: <code>string</code> - the name of the template
<a name="Metadata+getDisplayName"></a>

### metadata.getDisplayName() ⇒ <code>string</code>
Returns the display name for this template.

**Kind**: instance method of [<code>Metadata</code>](#Metadata)
**Returns**: <code>string</code> - the display name of the template
<a name="Metadata+getKeywords"></a>

### metadata.getKeywords() ⇒ <code>Array</code>
Returns the keywords for this template.

**Kind**: instance method of [<code>Metadata</code>](#Metadata)
**Returns**: <code>Array</code> - the keywords of the template
<a name="Metadata+getDescription"></a>

### metadata.getDescription() ⇒ <code>string</code>
Returns the description for this template.

**Kind**: instance method of [<code>Metadata</code>](#Metadata)
**Returns**: <code>string</code> - the description of the template
<a name="Metadata+getVersion"></a>

### metadata.getVersion() ⇒ <code>string</code>
Returns the version for this template.

**Kind**: instance method of [<code>Metadata</code>](#Metadata)
**Returns**: <code>string</code> - the description of the template
<a name="Metadata+getIdentifier"></a>

### metadata.getIdentifier() ⇒ <code>string</code>
Returns the identifier for this template, formed from name@version.

**Kind**: instance method of [<code>Metadata</code>](#Metadata)
**Returns**: <code>string</code> - the identifier of the template
<a name="Metadata+createTargetMetadata"></a>

### metadata.createTargetMetadata(runtimeName) ⇒ <code>object</code>
Return new Metadata for a target runtime

**Kind**: instance method of [<code>Metadata</code>](#Metadata)
```

Returns: <code>object</code> - the new Metadata

Param	Type	Description
runtimeName	<code>string</code>	the target runtime name

<a name="Metadata+toJSON"></a>

### metadata.toJSON() ⇒ <code>object</code>
Return the whole metadata content, for hashing

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>object</code> - the content of the metadata object
<a name="Metadata.checkImage"></a>

### Metadata.checkImage(buffer)
Check the buffer is a png file with the right size

Kind: static method of [<code>Metadata</code>](#Metadata)

Param	Type	Description
buffer	<code>Buffer</code>	the buffer object

<a name="Metadata.checkImageDimensions"></a>

### Metadata.checkImageDimensions(buffer, mimeType)
Checks if dimensions for the image are correct.

Kind: static method of [<code>Metadata</code>](#Metadata)

Param	Type	Description
buffer	<code>Buffer</code>	the buffer object
mimeType	<code>string</code>	the mime type of the object

<a name="MonetaryAmountFormatParser"></a>

## MonetaryAmountFormatParser
Parses a monetary/amount format string

Kind: global class
Access: public

* [MonetaryAmountFormatParser](#MonetaryAmountFormatParser)
    * _instance_
        * [.addGrammars(grammars, field)](#MonetaryAmountFormatParser+addGrammars)
        * [.buildFormatRules(formatString)]
(#MonetaryAmountFormatParser+buildFormatRules) ⇒ <code>Object</code>
    * _static_
        * [.parseMonetaryAmountFormatField(field)]
(#MonetaryAmountFormatParser.parseMonetaryAmountFormatField) ⇒ <code>string</code>

<a name="MonetaryAmountFormatParser+addGrammars"></a>

### monetaryAmountFormatParser.addGrammars(grammars, field)
Given current grammar parts, add necessary grammars parts for the format.

Kind: instance method of [<code>MonetaryAmountFormatParser</code>]

```
(#MonetaryAmountFormatParser)

| Param | Type | Description |
| --- | --- | --- |
| grammars | <code>Array.&lt;object&gt;</code> | the current grammar parts |
| field | <code>string</code> | grammar field |

<a name="MonetaryAmountFormatParser+buildFormatRules"></a>

### monetaryAmountFormatParser.buildFormatRules(formatString) ⇒ <code>Object</code>
Converts a format string to a Nearley action

**Kind**: instance method of [<code>MonetaryAmountFormatParser</code>]
(#MonetaryAmountFormatParser)
**Returns**: <code>Object</code> - the tokens and action and name to use for the
Nearley rule

| Param | Type | Description |
| --- | --- | --- |
| formatString | <code>string</code> | the input format string |

<a name="MonetaryAmountFormatParser.parseMonetaryAmountFormatField"></a>

### MonetaryAmountFormatParser.parseMonetaryAmountFormatField(field) ⇒
<code>string</code>
Given a format field (like CCC or 0,0.0) this method returns
a logical name for the field. Note the logical names
have been picked to align with the moment constructor that takes an object.

**Kind**: static method of [<code>MonetaryAmountFormatParser</code>]
(#MonetaryAmountFormatParser)
**Returns**: <code>string</code> - the field designator

| Param | Type | Description |
| --- | --- | --- |
| field | <code>string</code> | the input format field |

<a name="ParserManager"></a>

## ParserManager
Generates and manages a Nearley parser for a template.

**Kind**: global class

* [ParserManager](#ParserManager)
    * [new ParserManager(template)](#new_ParserManager_new)
    * _instance_
        * [.getParser()](#ParserManager+getParser) ⇒ <code>object</code>
        * [.getTemplateAst()](#ParserManager+getTemplateAst) ⇒ <code>object</code>
        * [.setGrammar(grammar)](#ParserManager+setGrammar)
        * [.buildGrammar(templatizedGrammar)](#ParserManager+buildGrammar)
        * [.buildGrammarRules(ast, templateModel, prefix, parts)]
(#ParserManager+buildGrammarRules)
        * [.handleBinding(templateModel, parts, inputRule, element)]
(#ParserManager+handleBinding)
        * [.cleanChunk(input)](#ParserManager+cleanChunk) ⇒ <code>string</code>
        * [.findFirstBinding(propertyName, elements)]
(#ParserManager+findFirstBinding) ⇒ <code>int</code>
        * [.getGrammar()](#ParserManager+getGrammar) ⇒ <code>String</code>
        * [.getTemplatizedGrammar()](#ParserManager+getTemplatizedGrammar) ⇒
<code>String</code>
        * [.roundtripMarkdown(text)](#ParserManager+roundtripMarkdown) ⇒
<code>string</code>
        * [.formatText(text, options, format)](#ParserManager+formatText) ⇒
<code>string</code>
    * _static_
        * [.adjustListBlock(x, separator)](#ParserManager.adjustListBlock) ⇒
<code>object</code>
        * [.getProperty(templateModel, element)](#ParserManager.getProperty) ⇒
<code>\*</code>
        * [._throwTemplateExceptionForElement(message, element)]
(#ParserManager._throwTemplateExceptionForElement)
        * [.compileGrammar(sourceCode)](#ParserManager.compileGrammar) ⇒
<code>object</code>

<a name="new_ParserManager_new"></a>

### new ParserManager(template)
Create the ParserManager.


| Param | Type | Description |
| --- | --- | --- |
| template | <code>object</code> | the template instance |

<a name="ParserManager+getParser"></a>

### parserManager.getParser() ⇒ <code>object</code>
Gets a parser object for this template

**Kind**: instance method of [<code>ParserManager</code>](#ParserManager)
**Returns**: <code>object</code> - the parser for this template
<a name="ParserManager+getTemplateAst"></a>

### parserManager.getTemplateAst() ⇒ <code>object</code>
Gets the AST for the template

**Kind**: instance method of [<code>ParserManager</code>](#ParserManager)
**Returns**: <code>object</code> - the AST for the template
<a name="ParserManager+setGrammar"></a>

### parserManager.setGrammar(grammar)
Set the grammar for the template

**Kind**: instance method of [<code>ParserManager</code>](#ParserManager)

| Param | Type | Description |
| --- | --- | --- |
| grammar | <code>String</code> | the grammar for the template |

<a name="ParserManager+buildGrammar"></a>

### parserManager.buildGrammar(templatizedGrammar)
Build a grammar from a template

**Kind**: instance method of [<code>ParserManager</code>](#ParserManager)

| Param | Type | Description |
| --- | --- | --- |
| templatizedGrammar | <code>String</code> | the annotated template using the
markdown parser |

<a name="ParserManager+buildGrammarRules"></a>

### parserManager.buildGrammarRules(ast, templateModel, prefix, parts)
Build grammar rules from a template

**Kind**: instance method of [<code>ParserManager</code>](#ParserManager)

| Param | Type | Description |
| --- | --- | --- |
| ast | <code>object</code> | the AST from which to build the grammar |
| templateModel | <code>ClassDeclaration</code> | the type of the parent class for
this AST |
| prefix | <code>String</code> | A unique prefix for the grammar rules |
| parts | <code>Object</code> | Result object to acculumate rules and required sub-
grammars |

<a name="ParserManager+handleBinding"></a>

### parserManager.handleBinding(templateModel, parts, inputRule, element)
Utility method to generate a grammar rule for a variable binding

**Kind**: instance method of [<code>ParserManager</code>](#ParserManager)

| Param | Type | Description |
| --- | --- | --- |
| templateModel | <code>ClassDeclaration</code> | the current template model |
| parts | <code>\*</code> | the parts, where the rule will be added |
| inputRule | <code>\*</code> | the rule we are processing in the AST |
| element | <code>\*</code> | the current element in the AST |

<a name="ParserManager+cleanChunk"></a>

### parserManager.cleanChunk(input) ⇒ <code>string</code>
Cleans a chunk of text to make it safe to include
as a grammar rule. We need to remove linefeeds and
escape any '"' characters.

**Kind**: instance method of [<code>ParserManager</code>](#ParserManager)
**Returns**: <code>string</code> - cleaned text

| Param | Type | Description |
| --- | --- | --- |
| input | <code>string</code> | the input text from the template |

<a name="ParserManager+findFirstBinding"></a>

### parserManager.findFirstBinding(propertyName, elements) ⇒ <code>int</code>
Finds the first binding for the given property

**Kind**: instance method of [<code>ParserManager</code>](#ParserManager)
**Returns**: <code>int</code> - the index of the element or -1

| Param | Type | Description |
| --- | --- | --- |
| propertyName | <code>string</code> | the name of the property |
| elements | <code>Array.&lt;object&gt;</code> | the result of parsing the
template_txt. |

<a name="ParserManager+getGrammar"></a>

### parserManager.getGrammar() ⇒ <code>String</code>
Get the (compiled) grammar for the template

**Kind**: instance method of [<code>ParserManager</code>](#ParserManager)
**Returns**: <code>String</code> - - the grammar for the template
<a name="ParserManager+getTemplatizedGrammar"></a>

### parserManager.getTemplatizedGrammar() ⇒ <code>String</code>
Returns the templatized grammar

**Kind**: instance method of [<code>ParserManager</code>](#ParserManager)
**Returns**: <code>String</code> - the contents of the templatized grammar
<a name="ParserManager+roundtripMarkdown"></a>

### parserManager.roundtripMarkdown(text) ⇒ <code>string</code>
Round-trip markdown

**Kind**: instance method of [<code>ParserManager</code>](#ParserManager)
**Returns**: <code>string</code> - the result of parsing and printing back the text

| Param | Type | Description |
| --- | --- | --- |
| text | <code>string</code> | the markdown text |

<a name="ParserManager+formatText"></a>

### parserManager.formatText(text, options, format) ⇒ <code>string</code>
Format text

**Kind**: instance method of [<code>ParserManager</code>](#ParserManager)
**Returns**: <code>string</code> - the result of parsing and printing back the text

| Param | Type | Description |
| --- | --- | --- |
| text | <code>string</code> | the markdown text |
| options | <code>object</code> | parameters to the formatting |
| format | <code>string</code> | to the text generation |

<a name="ParserManager.adjustListBlock"></a>

### ParserManager.adjustListBlock(x, separator) ⇒ <code>object</code>
Adjust the template for list blocks

**Kind**: static method of [<code>ParserManager</code>](#ParserManager)
**Returns**: <code>object</code> - the new template AST node

| Param | Type | Description |
| --- | --- | --- |
| x | <code>object</code> | The current template AST node |
| separator | <code>String</code> | The list separator |

<a name="ParserManager.getProperty"></a>

### ParserManager.getProperty(templateModel, element) ⇒ <code>\*</code>
```

```
Throws an error if a template variable doesn't exist on the model.

**Kind**: static method of [<code>ParserManager</code>](#ParserManager)
**Returns**: <code>\*</code> - the property

| Param | Type | Description |
| --- | --- | --- |
| templateModel | <code>\*</code> | the model for the template |
| element | <code>\*</code> | the current element in the AST |

<a name="ParserManager._throwTemplateExceptionForElement"></a>

### ParserManager.\_throwTemplateExceptionForElement(message, element)
Throw a template exception for the element

**Kind**: static method of [<code>ParserManager</code>](#ParserManager)
**Throws**:

- <code>TemplateException</code>


| Param | Type | Description |
| --- | --- | --- |
| message | <code>string</code> | the error message |
| element | <code>object</code> | the AST |

<a name="ParserManager.compileGrammar"></a>

### ParserManager.compileGrammar(sourceCode) ⇒ <code>object</code>
Compiles a Nearley grammar to its AST

**Kind**: static method of [<code>ParserManager</code>](#ParserManager)
**Returns**: <code>object</code> - the AST for the grammar

| Param | Type | Description |
| --- | --- | --- |
| sourceCode | <code>string</code> | the source text for the grammar |

<a name="Template"></a>

## *Template*
A template for a legal clause or contract. A Template has a template model,
request/response transaction types,
a template grammar (natural language for the template) as well as Ergo code for the
business logic of the
template.

**Kind**: global abstract class
**Access**: public

* *[Template](#Template)*
    * *[new Template(packageJson, readme, samples, request, logo, options)]
(#new_Template_new)*
    * _instance_
        * *[.validate()](#Template+validate)*
        * *[.getTemplateModel()](#Template+getTemplateModel) ⇒
<code>ClassDeclaration</code>*
        * *[.getIdentifier()](#Template+getIdentifier) ⇒ <code>String</code>*
        * *[.getMetadata()](#Template+getMetadata) ⇒ [<code>Metadata</code>]
```

```
(#Metadata)*
        * *[.getName()](#Template+getName) ⇒ <code>String</code>*
        * *[.getDisplayName()](#Template+getDisplayName) ⇒ <code>string</code>*
        * *[.getVersion()](#Template+getVersion) ⇒ <code>String</code>*
        * *[.getDescription()](#Template+getDescription) ⇒ <code>String</code>*
        * *[.getHash()](#Template+getHash) ⇒ <code>string</code>*
        * *[.toArchive([language], [options], logo)](#Template+toArchive) ⇒
<code>Promise.&lt;Buffer&gt;</code>*
        * *[.getParserManager()](#Template+getParserManager) ⇒
[<code>ParserManager</code>](#ParserManager)*
        * *[.getLogicManager()](#Template+getLogicManager) ⇒
<code>LogicManager</code>*
        * *[.getIntrospector()](#Template+getIntrospector) ⇒
<code>Introspector</code>*
        * *[.getFactory()](#Template+getFactory) ⇒ <code>Factory</code>*
        * *[.getSerializer()](#Template+getSerializer) ⇒ <code>Serializer</code>*
        * *[.getRequestTypes()](#Template+getRequestTypes) ⇒ <code>Array</code>*
        * *[.getResponseTypes()](#Template+getResponseTypes) ⇒ <code>Array</code>*
        * *[.getEmitTypes()](#Template+getEmitTypes) ⇒ <code>Array</code>*
        * *[.getStateTypes()](#Template+getStateTypes) ⇒ <code>Array</code>*
        * *[.hasLogic()](#Template+hasLogic) ⇒ <code>boolean</code>*
        * *[.grammarHasErgoExpression()](#Template+grammarHasErgoExpression) ⇒
<code>boolean</code>*
    * _static_
        * *[.fromDirectory(path, [options])](#Template.fromDirectory) ⇒
[<code>Promise.&lt;Template&gt;</code>](#Template)*
        * *[.fromArchive(buffer, [options])](#Template.fromArchive) ⇒
[<code>Promise.&lt;Template&gt;</code>](#Template)*
        * *[.fromUrl(url, [options])](#Template.fromUrl) ⇒ <code>Promise</code>*
        * *[.instanceOf(classDeclaration, fqt)](#Template.instanceOf) ⇒
<code>boolean</code>*

<a name="new_Template_new"></a>

### *new Template(packageJson, readme, samples, request, logo, options)*
Create the Template.
Note: Only to be called by framework code. Applications should
retrieve instances from [fromArchive](#Template.fromArchive) or [fromDirectory]
(#Template.fromDirectory).


| Param | Type | Description |
| --- | --- | --- |
| packageJson | <code>object</code> | the JS object for package.json |
| readme | <code>String</code> | the readme in markdown for the template (optional)
|
| samples | <code>object</code> | the sample text for the template in different
locales |
| request | <code>object</code> | the JS object for the sample request |
| logo | <code>Buffer</code> | the bytes data of logo |
| options | <code>Object</code> | e.g., { warnings: true } |

<a name="Template+validate"></a>

### *template.validate()*
Verifies that the template is well formed.
Throws an exception with the details of any validation errors.

**Kind**: instance method of [<code>Template</code>](#Template)
```

```
<a name="Template+getTemplateModel"></a>

### *template.getTemplateModel() ⇒ <code>ClassDeclaration</code>*
Returns the template model for the template

**Kind**: instance method of [<code>Template</code>](#Template)
**Returns**: <code>ClassDeclaration</code> - the template model for the template
**Throws**:

- <code>Error</code> if no template model is found, or multiple template models are
found

<a name="Template+getIdentifier"></a>

### *template.getIdentifier() ⇒ <code>String</code>*
Returns the identifier for this template

**Kind**: instance method of [<code>Template</code>](#Template)
**Returns**: <code>String</code> - the identifier of this template
<a name="Template+getMetadata"></a>

### *template.getMetadata() ⇒ [<code>Metadata</code>](#Metadata)*
Returns the metadata for this template

**Kind**: instance method of [<code>Template</code>](#Template)
**Returns**: [<code>Metadata</code>](#Metadata) - the metadata for this template
<a name="Template+getName"></a>

### *template.getName() ⇒ <code>String</code>*
Returns the name for this template

**Kind**: instance method of [<code>Template</code>](#Template)
**Returns**: <code>String</code> - the name of this template
<a name="Template+getDisplayName"></a>

### *template.getDisplayName() ⇒ <code>string</code>*
Returns the display name for this template.

**Kind**: instance method of [<code>Template</code>](#Template)
**Returns**: <code>string</code> - the display name of the template
<a name="Template+getVersion"></a>

### *template.getVersion() ⇒ <code>String</code>*
Returns the version for this template

**Kind**: instance method of [<code>Template</code>](#Template)
**Returns**: <code>String</code> - the version of this template. Use semver module
to parse.
<a name="Template+getDescription"></a>

### *template.getDescription() ⇒ <code>String</code>*
Returns the description for this template

**Kind**: instance method of [<code>Template</code>](#Template)
**Returns**: <code>String</code> - the description of this template
<a name="Template+getHash"></a>

### *template.getHash() ⇒ <code>string</code>*
Gets a content based SHA-256 hash for this template. Hash
```

```
is based on the metadata for the template plus the contents of
all the models and all the script files.

**Kind**: instance method of [<code>Template</code>](#Template)
**Returns**: <code>string</code> - the SHA-256 hash in hex format
<a name="Template+toArchive"></a>

### *template.toArchive([language], [options], logo) ⇒
<code>Promise.&lt;Buffer&gt;</code>*
Persists this template to a Cicero Template Archive (cta) file.

**Kind**: instance method of [<code>Template</code>](#Template)
**Returns**: <code>Promise.&lt;Buffer&gt;</code> - the zlib buffer

| Param | Type | Description |
| --- | --- | --- |
| [language] | <code>string</code> | target language for the archive (should be
'ergo') |
| [options] | <code>Object</code> | JSZip options |
| logo | <code>Buffer</code> | Bytes data of the PNG file |

<a name="Template+getParserManager"></a>

### *template.getParserManager() ⇒ [<code>ParserManager</code>](#ParserManager)*
Provides access to the parser manager for this template.
The parser manager can convert template data to and from
natural language text.

**Kind**: instance method of [<code>Template</code>](#Template)
**Returns**: [<code>ParserManager</code>](#ParserManager) - the ParserManager for
this template
<a name="Template+getLogicManager"></a>

### *template.getLogicManager() ⇒ <code>LogicManager</code>*
Provides access to the template logic for this template.
The template logic encapsulate the code necessary to
execute the clause or contract.

**Kind**: instance method of [<code>Template</code>](#Template)
**Returns**: <code>LogicManager</code> - the LogicManager for this template
<a name="Template+getIntrospector"></a>

### *template.getIntrospector() ⇒ <code>Introspector</code>*
Provides access to the Introspector for this template. The Introspector
is used to reflect on the types defined within this template.

**Kind**: instance method of [<code>Template</code>](#Template)
**Returns**: <code>Introspector</code> - the Introspector for this template
<a name="Template+getFactory"></a>

### *template.getFactory() ⇒ <code>Factory</code>*
Provides access to the Factory for this template. The Factory
is used to create the types defined in this template.

**Kind**: instance method of [<code>Template</code>](#Template)
**Returns**: <code>Factory</code> - the Factory for this template
<a name="Template+getSerializer"></a>

### *template.getSerializer() ⇒ <code>Serializer</code>*
```

```
Provides access to the Serializer for this template. The Serializer
is used to serialize instances of the types defined within this template.

**Kind**: instance method of [<code>Template</code>](#Template)
**Returns**: <code>Serializer</code> - the Serializer for this template
<a name="Template+getRequestTypes"></a>

### *template.getRequestTypes() ⇒ <code>Array</code>*
Provides a list of the input types that are accepted by this Template. Types use
the fully-qualified form.

**Kind**: instance method of [<code>Template</code>](#Template)
**Returns**: <code>Array</code> - a list of the request types
<a name="Template+getResponseTypes"></a>

### *template.getResponseTypes() ⇒ <code>Array</code>*
Provides a list of the response types that are returned by this Template. Types use
the fully-qualified form.

**Kind**: instance method of [<code>Template</code>](#Template)
**Returns**: <code>Array</code> - a list of the response types
<a name="Template+getEmitTypes"></a>

### *template.getEmitTypes() ⇒ <code>Array</code>*
Provides a list of the emit types that are emitted by this Template. Types use the
fully-qualified form.

**Kind**: instance method of [<code>Template</code>](#Template)
**Returns**: <code>Array</code> - a list of the emit types
<a name="Template+getStateTypes"></a>

### *template.getStateTypes() ⇒ <code>Array</code>*
Provides a list of the state types that are expected by this Template. Types use
the fully-qualified form.

**Kind**: instance method of [<code>Template</code>](#Template)
**Returns**: <code>Array</code> - a list of the state types
<a name="Template+hasLogic"></a>

### *template.hasLogic() ⇒ <code>boolean</code>*
Returns true if the template has logic, i.e. has more than one script file.

**Kind**: instance method of [<code>Template</code>](#Template)
**Returns**: <code>boolean</code> - true if the template has logic
<a name="Template+grammarHasErgoExpression"></a>

### *template.grammarHasErgoExpression() ⇒ <code>boolean</code>*
Checks whether the template grammar has computer (Ergo) expressions

**Kind**: instance method of [<code>Template</code>](#Template)
**Returns**: <code>boolean</code> - True if the template grammar has Ergo
expressions (`{{% ... %}}`)
<a name="Template.fromDirectory"></a>

### *Template.fromDirectory(path, [options]) ⇒
[<code>Promise.&lt;Template&gt;</code>](#Template)*
Builds a Template from the contents of a directory.
The directory must include a package.json in the root (used to specify
the name, version and description of the template).
```

```
**Kind**: static method of [<code>Template</code>](#Template)
**Returns**: [<code>Promise.&lt;Template&gt;</code>](#Template) - a Promise to the
instantiated template

| Param | Type | Default | Description |
| --- | --- | --- | --- |
| path | <code>String</code> |  | to a local directory |
| [options] | <code>Object</code> | <code></code> | an optional set of options to
configure the instance. |

<a name="Template.fromArchive"></a>

### *Template.fromArchive(buffer, [options]) ⇒
[<code>Promise.&lt;Template&gt;</code>](#Template)*
Create a template from an archive.

**Kind**: static method of [<code>Template</code>](#Template)
**Returns**: [<code>Promise.&lt;Template&gt;</code>](#Template) - a Promise to the
template

| Param | Type | Default | Description |
| --- | --- | --- | --- |
| buffer | <code>Buffer</code> |  | the buffer to a Cicero Template Archive (cta)
file |
| [options] | <code>Object</code> | <code></code> | an optional set of options to
configure the instance. |

<a name="Template.fromUrl"></a>

### *Template.fromUrl(url, [options]) ⇒ <code>Promise</code>*
Create a template from an URL.

**Kind**: static method of [<code>Template</code>](#Template)
**Returns**: <code>Promise</code> - a Promise to the template

| Param | Type | Default | Description |
| --- | --- | --- | --- |
| url | <code>String</code> |  | the URL to a Cicero Template Archive (cta) file |
| [options] | <code>Object</code> | <code></code> | an optional set of options to
configure the instance. |

<a name="Template.instanceOf"></a>

### *Template.instanceOf(classDeclaration, fqt) ⇒ <code>boolean</code>*
Check to see if a ClassDeclaration is an instance of the specified fully qualified
type name.

**Kind**: static method of [<code>Template</code>](#Template)
**Returns**: <code>boolean</code> - True if classDeclaration an instance of the
specified fully
qualified type name, false otherwise.
**Internal**:

| Param | Type | Description |
| --- | --- | --- |
| classDeclaration | <code>ClassDeclaration</code> | The class to test |
| fqt | <code>String</code> | The fully qualified type name. |
<a name="TemplateInstance"></a>

## *TemplateInstance*
A TemplateInstance is an instance of a Clause or Contract template. It is
executable business logic, linked to
a natural language (legally enforceable) template.
A TemplateInstance must be constructed with a template and then prior to execution
the data for the clause must be set.
Set the data for the TemplateInstance by either calling the setData method or by
calling the parse method and passing in natural language text that conforms to the
template grammar.

**Kind**: global abstract class
**Access**: public

* *[TemplateInstance](#TemplateInstance)*
    * *[new TemplateInstance(template)](#new_TemplateInstance_new)*
    * _instance_
        * *[.setData(data)](#TemplateInstance+setData)*
        * *[.getData()](#TemplateInstance+getData) ⇒ <code>object</code>*
        * *[.getEngine()](#TemplateInstance+getEngine) ⇒ <code>object</code>*
        * *[.getDataAsConcertoObject()](#TemplateInstance+getDataAsConcertoObject)
⇒ <code>object</code>*
        * *[.parse(input, [currentTime], [fileName])](#TemplateInstance+parse)*
        * *[.draft([options], currentTime)](#TemplateInstance+draft) ⇒
<code>string</code>*
        * *[.getIdentifier()](#TemplateInstance+getIdentifier) ⇒
<code>String</code>*
        * *[.getTemplate()](#TemplateInstance+getTemplate) ⇒
[<code>Template</code>](#Template)*
        * *[.getLogicManager()](#TemplateInstance+getLogicManager) ⇒
<code>LogicManager</code>*
        * *[.toJSON()](#TemplateInstance+toJSON) ⇒ <code>object</code>*
    * _static_
        * *[.convertFormattedParsed(obj, utcOffset)]
(#TemplateInstance.convertFormattedParsed) ⇒ <code>\*</code>*

<a name="new_TemplateInstance_new"></a>

### *new TemplateInstance(template)*
Create the Clause and link it to a Template.


| Param | Type | Description |
| --- | --- | --- |
| template | [<code>Template</code>](#Template) | the template for the clause |

<a name="TemplateInstance+setData"></a>

### *templateInstance.setData(data)*
Set the data for the clause

**Kind**: instance method of [<code>TemplateInstance</code>](#TemplateInstance)

| Param | Type | Description |
| --- | --- | --- |
| data | <code>object</code> | the data for the clause, must be an instance of the
template model for the clause's template. This should be a plain JS object and will
be deserialized and validated into the Concerto object before assignment. |
```

<a name="TemplateInstance+getData"></a>

### *templateInstance.getData() ⇒ <code>object</code>*
Get the data for the clause. This is a plain JS object. To retrieve the Concerto
object call getConcertoData().

Kind: instance method of [<code>TemplateInstance</code>](#TemplateInstance)
Returns: <code>object</code> - - the data for the clause, or null if it has not
been set
<a name="TemplateInstance+getEngine"></a>

### *templateInstance.getEngine() ⇒ <code>object</code>*
Get the current Ergo engine

Kind: instance method of [<code>TemplateInstance</code>](#TemplateInstance)
Returns: <code>object</code> - - the data for the clause, or null if it has not
been set
<a name="TemplateInstance+getDataAsConcertoObject"></a>

### *templateInstance.getDataAsConcertoObject() ⇒ <code>object</code>*
Get the data for the clause. This is a Concerto object. To retrieve the
plain JS object suitable for serialization call toJSON() and retrieve the `data`
property.

Kind: instance method of [<code>TemplateInstance</code>](#TemplateInstance)
Returns: <code>object</code> - - the data for the clause, or null if it has not
been set
<a name="TemplateInstance+parse"></a>

### *templateInstance.parse(input, [currentTime], [fileName])*
Set the data for the clause by parsing natural language text.

Kind: instance method of [<code>TemplateInstance</code>](#TemplateInstance)

Param	Type	Description
input	<code>string</code>	the text for the clause
[currentTime]	<code>string</code>	the definition of 'now' (optional)
[fileName]	<code>string</code>	the fileName for the text (optional)

<a name="TemplateInstance+draft"></a>

### *templateInstance.draft([options], currentTime) ⇒ <code>string</code>*
Generates the natural language text for a contract or clause clause; combining the
text from the template
and the instance data.

Kind: instance method of [<code>TemplateInstance</code>](#TemplateInstance)
Returns: <code>string</code> - the natural language text for the contract or
clause; created by combining the structure of
the template with the JSON data for the clause.

Param	Type	Description
[options]	<code>*</code>	text generation options. options.wrapVariables
encloses variables and editable sections in '<variable ...' and '/>'
currentTime	<code>string</code>	the definition of 'now' (optional)

```
<a name="TemplateInstance+getIdentifier"></a>

### *templateInstance.getIdentifier() ⇒ <code>String</code>*
Returns the identifier for this clause. The identifier is the identifier of
the template plus '-' plus a hash of the data for the clause (if set).

**Kind**: instance method of [<code>TemplateInstance</code>](#TemplateInstance)
**Returns**: <code>String</code> - the identifier of this clause
<a name="TemplateInstance+getTemplate"></a>

### *templateInstance.getTemplate() ⇒ [<code>Template</code>](#Template)*
Returns the template for this clause

**Kind**: instance method of [<code>TemplateInstance</code>](#TemplateInstance)
**Returns**: [<code>Template</code>](#Template) - the template for this clause
<a name="TemplateInstance+getLogicManager"></a>

### *templateInstance.getLogicManager() ⇒ <code>LogicManager</code>*
Returns the template logic for this clause

**Kind**: instance method of [<code>TemplateInstance</code>](#TemplateInstance)
**Returns**: <code>LogicManager</code> - the template for this clause
<a name="TemplateInstance+toJSON"></a>

### *templateInstance.toJSON() ⇒ <code>object</code>*
Returns a JSON representation of the clause

**Kind**: instance method of [<code>TemplateInstance</code>](#TemplateInstance)
**Returns**: <code>object</code> - the JS object for serialization
<a name="TemplateInstance.convertFormattedParsed"></a>

### *TemplateInstance.convertFormattedParsed(obj, utcOffset) ⇒ <code>\*</code>*
Recursive function that converts all instances of Formated objects (ParsedDateTime
or ParsedMonetaryAmount)
to a Moment.

**Kind**: static method of [<code>TemplateInstance</code>](#TemplateInstance)
**Returns**: <code>\*</code> - the converted object

| Param | Type | Description |
| --- | --- | --- |
| obj | <code>\*</code> | the input object |
| utcOffset | <code>number</code> | the default utcOffset |

<a name="CompositeArchiveLoader"></a>

## CompositeArchiveLoader
Manages a set of archive loaders, delegating to the first archive
loader that accepts a URL.

**Kind**: global class

* [CompositeArchiveLoader](#CompositeArchiveLoader)
    * [new CompositeArchiveLoader()](#new_CompositeArchiveLoader_new)
    * [.addArchiveLoader(archiveLoader)](#CompositeArchiveLoader+addArchiveLoader)
    * [.clearArchiveLoaders()](#CompositeArchiveLoader+clearArchiveLoaders)
    * *[.accepts(url)](#CompositeArchiveLoader+accepts) ⇒ <code>boolean</code>*
    * [.load(url, options)](#CompositeArchiveLoader+load) ⇒ <code>Promise</code>
```

```
<a name="new_CompositeArchiveLoader_new"></a>

### new CompositeArchiveLoader()
Create the CompositeArchiveLoader. Used to delegate to a set of ArchiveLoaders.

<a name="CompositeArchiveLoader+addArchiveLoader"></a>

### compositeArchiveLoader.addArchiveLoader(archiveLoader)
Adds a ArchiveLoader implemenetation to the ArchiveLoader

**Kind**: instance method of [<code>CompositeArchiveLoader</code>]
(#CompositeArchiveLoader)

| Param | Type | Description |
| --- | --- | --- |
| archiveLoader | <code>ArchiveLoader</code> | The archive to add to the
CompositeArchiveLoader |

<a name="CompositeArchiveLoader+clearArchiveLoaders"></a>

### compositeArchiveLoader.clearArchiveLoaders()
Remove all registered ArchiveLoaders

**Kind**: instance method of [<code>CompositeArchiveLoader</code>]
(#CompositeArchiveLoader)
<a name="CompositeArchiveLoader+accepts"></a>

### *compositeArchiveLoader.accepts(url) ⇒ <code>boolean</code>*
Returns true if this ArchiveLoader can process the URL

**Kind**: instance abstract method of [<code>CompositeArchiveLoader</code>]
(#CompositeArchiveLoader)
**Returns**: <code>boolean</code> - true if this ArchiveLoader accepts the URL

| Param | Type | Description |
| --- | --- | --- |
| url | <code>string</code> | the URL |

<a name="CompositeArchiveLoader+load"></a>

### compositeArchiveLoader.load(url, options) ⇒ <code>Promise</code>
Load a Archive from a URL and return it

**Kind**: instance method of [<code>CompositeArchiveLoader</code>]
(#CompositeArchiveLoader)
**Returns**: <code>Promise</code> - a promise to the Archive

| Param | Type | Description |
| --- | --- | --- |
| url | <code>string</code> | the url to get |
| options | <code>object</code> | additional options |

<a name="locationOfError"></a>

## locationOfError(error) ⇒ <code>object</code>
Extract the file location from the parse error

**Kind**: global function
**Returns**: <code>object</code> - - the file location information
```

Param	Type	Description
error	<code>object</code>	the error object

<a name="isPNG"></a>

## isPNG(buffer) ⇒ <code>Boolean</code>
Checks whether the file is PNG

Kind: global function
Returns: <code>Boolean</code> - whether the file in PNG

Param	Type	Description
buffer	<code>Buffer</code>	buffer of the file

<a name="getMimeType"></a>

## getMimeType(buffer) ⇒ <code>Object</code>
Returns the mime-type of the file

Kind: global function
Returns: <code>Object</code> - the mime-type of the file

Param	Type	Description
buffer	<code>Buffer</code>	buffer of the file

---
id: version-0.21-ref-cicero-cli
title: Command Line
original_id: ref-cicero-cli

Install the `@accordproject/cicero-cli` npm package to access the Cicero command line interface (CLI). After installation you can use the `cicero` command and its sub-commands as described below.

To install the Cicero CLI:
```
npm install -g @accordproject/cicero-cli
```

## Usage

```md
cicero <cmd> [args]

Commands:
  cicero parse       parse a contract text
  cicero draft       create contract text from data
  cicero normalize   normalize markdown (parse & redraft)
  cicero trigger     send a request to the contract
  cicero invoke      invoke a clause of the contract
  cicero initialize  initialize a clause
  cicero archive     create a template archive
```

~~~~
  cicero compile     generate code for a target platform
  cicero get         save local copies of external dependencies

Options:
  --version      Show version number                                   [boolean]
  --verbose, -v                                                 [default: false]
  --help         Show help                                             [boolean]
```

## cicero parse

`cicero parse` loads a template from a directory on disk and then parses input
clause (or contract) text using the template. If successful, the template model is
printed to console. If there are syntax errors, the line and column and error
information are printed.

```md
cicero parse

parse a contract text

Options:
  --version      Show version number                                   [boolean]
  --verbose, -v                                                 [default: false]
  --help         Show help                                             [boolean]
  --template     path to the template                                   [string]
  --sample       path to the contract text                              [string]
  --output       path to the output file                                [string]
  --currentTime  set current time                       [string] [default: null]
  --warnings     print warnings                       [boolean] [default: false]
```

## cicero draft

`cicero draft` creates contract text from data.

```md
cicero draft

create contract text from data

Options:
  --version           Show version number                              [boolean]
  --verbose, -v                                                 [default: false]
  --help              Show help                                        [boolean]
  --template          path to the template                              [string]
  --data              path to the contract data                         [string]
  --output            path to the output file                           [string]
  --currentTime       set current time                  [string] [default: null]
  --format            target format                                     [string]
  --unquoteVariables  remove variables quoting        [boolean] [default: false]
  --warnings          print warnings                  [boolean] [default: false]
```

## cicero normalize

`cicero normalize` normalizes markdown text by parsing and redrafting the text.

```md
~~~~

```
cicero normalize

normalize markdown (parse & redraft)

Options:
  --version           Show version number                              [boolean]
  --verbose, -v                                                 [default: false]
  --help              Show help                                        [boolean]
  --template          path to the template                              [string]
  --sample            path to the contract text                         [string]
  --overwrite         overwrite the contract text     [boolean] [default: false]
  --output            path to the output file                           [string]
  --currentTime       set current time                  [string] [default: null]
  --warnings          print warnings                  [boolean] [default: false]
  --wrapVariables     wrap variables as XML tags      [boolean] [default: false]
  --format            target format                                     [string]
  --unquoteVariables  remove variables quoting        [boolean] [default: false]
```

## cicero trigger

`cicero trigger` sends a request to the contract.

```md
cicero trigger

send a request to the contract

Options:
  --version      Show version number                                   [boolean]
  --verbose, -v                                                 [default: false]
  --help         Show help                                             [boolean]
  --template     path to the template                                   [string]
  --sample       path to the contract text                              [string]
  --request      path to the JSON request                                [array]
  --state        path to the JSON state                                 [string]
  --currentTime  set current time                       [string] [default: null]
  --warnings     print warnings                       [boolean] [default: false]
```

## cicero invoke

`cicero invoke` invokes a specific clause (`--clauseName`) of the contract.

```md
cicero invoke

invoke a clause of the contract

Options:
  --version      Show version number                                   [boolean]
  --verbose, -v                                                 [default: false]
  --help         Show help                                             [boolean]
  --template     path to the template                                   [string]
  --sample       path to the contract text                              [string]
  --clauseName   the name of the clause to invoke                       [string]
  --params       path to the parameters                                 [string]
  --state        path to the JSON state                                 [string]
  --currentTime  set current time                       [string] [default: null]
```

  --warnings   print warnings   [boolean] [default: false]

## cicero initialize

`cicero initialize` initializes a clause.

```md
cicero initialize

initialize a clause

Options:
  --version      Show version number                                   [boolean]
  --verbose, -v                                                 [default: false]
  --help         Show help                                             [boolean]
  --template     path to the template                                   [string]
  --sample       path to the contract text                              [string]
  --currentTime  initialize with this current time      [string] [default: null]
  --warnings     print warnings                       [boolean] [default: false]
```
## cicero archive

`cicero archive` creates a Cicero Template Archive (`.cta`) file from a template
stored in a local directory.

```md
cicero archive

create a template archive

Options:
  --version      Show version number                                   [boolean]
  --verbose, -v                                                 [default: false]
  --help         Show help                                             [boolean]
  --template     path to the template                                   [string]
  --target       the target language of the archive   [string] [default: "ergo"]
  --output       file name for new archive              [string] [default: null]
  --warnings     print warnings                       [boolean] [default: false]
```

## cicero compile

`cicero compile` generates code for a target platform. It loads a template from a
directory on disk and then attempts to generate versions of the template model in
the specified format. The available formats include: `Go`, `PlantUML`,
`Typescript`, `Java`, and `JSONSchema`.

```md
cicero compile

generate code for a target platform

Options:
  --version      Show version number                                   [boolean]
  --verbose, -v                                                 [default: false]
  --help         Show help                                             [boolean]
  --template     path to the template                                   [string]
  --target       target of the code generation  [string] [default: "JSONSchema"]
```

~~~~
  --output       path to the output directory    [string] [default: "./output/"]
  --warnings     print warnings                       [boolean] [default: false]
```
## cicero get

`cicero get` saves local copies of external dependencies.

```md
cicero get

save local copies of external dependencies

Options:
  --version      Show version number                                   [boolean]
  --verbose, -v                                                 [default: false]
  --help         Show help                                             [boolean]
  --template     path to the template                                   [string]
  --output       output directory path                                  [string]
```

--------------------------------------------------------------------------------
---
id: version-0.21-ref-cicero-testing
title: Template Testing
original_id: ref-cicero-testing
---

Cicero uses [Cucumber](https://cucumber.io/docs) for writing template tests, which
provides a human readable syntax.

This documents the syntax available to write Cicero tests.

## Test Structure

Tests are located in the `./test/` directory for each template, which contains
files with the `.feature` extension.

Each file has the following structure:

```gherkin
Feature: Name of the template being tested
  Description for the test

  Background:
    Given that the contract says
"""
Text of the contract instance.
"""

  Scenario: Description for scenario 1
    [[First Scenario Sequence]]

  Scenario: Description for scenario 2
    [[Second Scenario Sequence]]

etc.
```

Each scenario can be thought of as a description for the behavior of the clause or
~~~~

~~~
contract template for the contract given as background.

Each scenario corresponds to one call to the contract. I.e., for a given current
time, request and contract state, it says what the expected result of executing the
contract should be. This can be either:
- A response, a new contract state, and a list of emitted obligations
- An error

## Scenarios

A complete scenario is described in the [Gherkin
Syntax](https://cucumber.io/docs/gherkin/reference/) through a sequence of
**Step**.

Each step starts with a keyword, either **Given**, **When**, **And**, or **Then**:

- **Given**, **When** and **And** are used to specify the input for a call to the
contract;
- **Then** and **And** are used to specify the expected result.

### Request and Response

The simplest kind of scenario specifies the response expected for a given request.

For instance, the following scenario describes the expected response for a given
request to the [helloworld
template](https://templates.accordproject.org/helloworld@0.10.1.html):

```gherkin
  Scenario: The contract should say Hello to Betty Buyer, from the ACME Corporation
    When it receives the request
"""
{
    "$class": "org.accordproject.helloworld.MyRequest",
    "input": "ACME Corporation"
}
"""
    Then it should respond with
"""
{
    "$class": "org.accordproject.helloworld.MyResponse",
    "output": "Hello Betty Buyer ACME Corporation"
}
"""
```

Both the request and the response are written inside triple quotes `"""` using
JSON. If the request or response is not valid wrt. to the data model, this will
result in a failing test.

:::warning
While the syntax for each scenario uses _pseudo_ natural language (e.g., `When it
receives the request`), the tests use very specific sentences as illustrated in
this guide.
:::

### Defaults

You can use the sample contract `sample.txt` and request `request.json` provided
~~~

with a template by using specific steps.

For instance, the following scenario describes the expected response for the default contract text when sending the default request to the [helloworld template](https://templates.accordproject.org/helloworld@0.10.1.html):
```gherkin
Feature: HelloWorld
  This describe the expected behavior for the Accord Project's "Hello World!"
contract

  Background:
    Given the default contract

  Scenario: The contract should say Hello to Fred Blogs, from the Accord Project,
for the default request
    When it receives the default request
    Then it should respond with
"""
{
    "$class": "org.accordproject.helloworld.MyResponse",
    "output": "Hello Fred Blogs Accord Project"
}
"""
```

### Errors

Whenever appropriate, it is good practice to include both successful executions, as well as scenarios for cases when a call to a template might fail. This can be written using a **Then** step that describes the error.

For instance, the following scenario describes an expected error for a given request to the [Interest Rate Swap](https://templates.accordproject.org/interest-rate-swap@0.4.1.html) template:
```gherkin
Feature: Interest Rate Swap
  This describes the expected behavior for the Accord Project's interest rate swap
contract

  Background:
    Given that the contract says
"""
INTEREST RATE SWAP TRANSACTION LETTER AGREEMENT
"Deutsche Bank"

Date: 06/30/2005
To: "MagnaChip Semiconductor S.A."
Attention: Swaps Documentation Department
Our Reference: "Global No. N397355N"
Re: Interest Rate Swap Transaction

Ladies and Gentlemen:

The purpose of this letter agreement is to set forth the terms and conditions of
the Transaction entered into between "Deutsche Bank" and "MagnaChip Semiconductor
S.A." (“Counterparty”) on the Trade Date specified below (the “Transaction”). This
letter agreement constitutes a “Confirmation” as referred to in the Agreement
specified below.
```

The definitions and provisions contained in the 2000 ISDA Definitions (the “Definitions”) as published by the International Swaps and Derivatives Association, Inc. are incorporated by reference herein. In the event of any inconsistency between the Definitions and this Confirmation, this Confirmation will govern.

For the purpose of this Confirmation, all references in the Definitions or the Agreement to a “Swap Transaction” shall be deemed to be references to this Transaction.

1. This Confirmation evidences a complete and binding agreement between "Deutsche Bank" (“Party A”) and Counterparty (“Party B”) as to the terms of the Transaction to which this Confirmation relates. In addition, Party A and Party B agree to use all reasonable efforts to negotiate, execute and deliver an agreement in the form of the ISDA 2002 Master Agreement with such modifications as Party A and Party B will in good faith agree (the “ISDA Form” or the “Agreement”). Upon execution by the parties of such Agreement, this Confirmation will supplement, form a part of and be subject to the Agreement. All provisions contained or incorporated by reference in such Agreement upon its execution shall govern this Confirmation except as expressly modified below. Until Party A and Party B execute and deliver the Agreement, this Confirmation, together with all other documents referring to the ISDA Form (each a “Confirmation”) confirming Transactions (each a “Transaction”) entered into between us (notwithstanding anything to the contrary in a Confirmation) shall supplement, form a part of, and be subject to an agreement in the form of the ISDA Form as if Party A and Party B had executed an agreement on the Trade Date of the first such Transaction between us in such form, with the Schedule thereto (i) specifying only that (a) the governing law is English law, provided, that such choice of law shall be superseded by any choice of law provision specified in the Agreement upon its execution, and (b) the Termination Currency is U.S. Dollars and (ii) incorporating the addition to the definition of “Indemnifiable Tax” contained in (page 49 of) the ISDA “User’s Guide to the 2002 ISDA Master Agreements”.
2. The terms of the particular Transaction to which this Confirmation relates are as follows:

Notional Amount: 300000000.00 USD
Trade Date: 06/23/2005
Effective Date: 06/27/2005
Termination Date: 06/18/2008

Fixed Amounts:
Fixed Rate Payer: "Counterparty"
Fixed Rate Payer Period End Dates: "The 15th day of March, June, September and December of each year, commencing September 15, 2005, through and including the Termination Date with No Adjustment"
Fixed Rate Payer Payment Dates: "The 15th day of March, June, September and December of each year, commencing September 15, 2005, through and including the Termination Date"
Fixed Rate: -4.09%
Fixed Rate Day Count Fraction: "30" "360"
Fixed Rate Payer Business Days:"New York"
Fixed Rate Payer Business Day Convention: "Modified Following"

Floating Amounts:
Floating Rate Payer: "DBAG"
Floating Rate Payer Period End Dates: "The 15th day of March, June, September and December of each year, commencing September 15, 2005, through and including the Termination Date with No Adjustment"
Floating Rate Payer Payment Dates: "The 15th day of March, June, September and December of each year, commencing September 15, 2005, through and including the

Termination Date"
Floating Rate for initial Calculation Period: 3.41%
Floating Rate Option: "USD-LIBOR-BBA"
Designated Maturity: "Three months"
Spread: "None"
Floating Rate Day Count Fraction: "30" "360"
Reset Dates: "The first Floating Rate Payer Business Day of each Calculation Period
or Compounding Period, if Compounding is applicable."
Compounding: "Inapplicable"
Floating Rate Payer Business Days: "New York"
Floating Rate Payer Business Day Convention: "Modified Following"
"""

  Scenario: The fixed rate is negative
    When it receives the request
"""
{
    "$class": "org.accordproject.isda.irs.RateObservation"
}
"""
    Then it should reject the request with the error "[Ergo] Fixed rate cannot be
negative"
~~~

The reason for the error is that the contract has been defined with a negative
interest rate (the line: `Fixed Rate: -4.09%` in the contract given as
Background for the scenario).

State Change

For templates which assume and can modify the contract state, the scenario should
also include pre- and post- conditions for that state. In addition, some steps are
available to define scenarios that specify the expected initial step for the
contract.

For instance, the following scenario for the [Installment
Sale](https://templates.accordproject.org/installment-sale@0.12.1.html) template
describes the expected initial state and execution of one installment:
```gherkin
Feature: Installment Sale
  This describe the expected behavior for the Accord Project's installment sale
contract

  Background:
    Given that the contract says
"""
"Dan" agrees to pay to "Ned" the total sum e10000, in the manner following:

E500 is to be paid at closing, and the remaining balance of E9500 shall be paid as
follows:

E500 or more per month on the first day of each and every month, and continuing
until the entire balance, including both principal and interest, shall be paid in
full -- provided, however, that the entire balance due plus accrued interest and
any other amounts due here-under shall be paid in full on or before 24 months.

Monthly payments, which shall start on month 3, include both principal and interest
with interest at the rate of 1.5%, computed monthly on the remaining balance from
time to time unpaid.
~~~

```
"""

  Scenario: The contract should be in the correct initial state
    Then the initial state of the contract should be
"""
{
  "$class": "org.accordproject.installmentsale.InstallmentSaleState",
  "status" : "WaitingForFirstDayOfNextMonth",
  "balance_remaining" : 10000.00,
  "total_paid" : 0.00,
  "next_payment_month" : 3,
  "stateId": "#1"
}
"""

  Scenario: The contract accepts a first payment, and maintain the remaining
balance
    Given the state
"""
{
  "$class": "org.accordproject.installmentsale.InstallmentSaleState",
  "status" : "WaitingForFirstDayOfNextMonth",
  "balance_remaining" : 10000.00,
  "total_paid" : 0.00,
  "next_payment_month" : 3,
  "stateId": "#1"
}
"""
    When it receives the request
"""
{
    "$class": "org.accordproject.installmentsale.Installment",
    "amount": 2500.00
}
"""
    Then it should respond with
"""
{
  "total_paid": 2500,
  "balance": 7612.499999999999,
  "$class": "org.accordproject.installmentsale.Balance"
}
"""
    And the new state of the contract should be
"""
{
  "$class": "org.accordproject.installmentsale.InstallmentSaleState",
  "status" : "WaitingForFirstDayOfNextMonth",
  "balance_remaining" : 7612.499999999999,
  "total_paid" : 2500,
  "next_payment_month" : 4,
  "stateId": "#1"
}
"""

```

### Current Time

The logic for some clause or contract templates is time-dependent. It can be useful to specify multiple scenarios for the behavior under different date and time assumptions. This can be described with an additional **When** step to set the current time to a specific value.

For instance, the following shows two scenarios for the [IP Payment](https://templates.accordproject.org/ip-payment@0.10.1.html) template, which describe its expected behavior for two distinct current times:

```gherkin
Feature: IP Payment Contract
  This describes the expected behavior for the Accord Project’s IP Payment Contract
contract

  Background:
    Given the default contract

  Scenario: Payment of a specified amount should be made
    When the current time is "2019-03-04T16:34:00-05:00"
    And it receives the request
"""
{
    "$class": "org.accordproject.ippayment.PaymentRequest",
    "netSaleRevenue": 1200,
    "sublicensingRevenue": 450,
    "permissionGrantedBy": "2018-04-05T00:00:00-05:00"
}
"""
    Then it should respond with
"""
{
    "$class": "org.accordproject.ippayment.PayOut",
    "totalAmount": 77.4,
    "dueBy": "2018-04-12T00:00:00.000-05:00"
}
"""

  Scenario: Payment of a specified amount should be made
    When the current time is "2019-03-01T16:34:00-02:00"
    And it receives the request
"""
{
    "$class": "org.accordproject.ippayment.PaymentRequest",
    "netSaleRevenue": 1550,
    "sublicensingRevenue": 225,
    "permissionGrantedBy": "2018-04-05T00:00:00-05:00"
}
"""
    Then it should respond with
"""
{
    "$class": "org.accordproject.ippayment.PayOut",
    "totalAmount": 81.45,
    "dueBy": "2018-04-12T03:00:00.000-02:00"
}
"""
```

### Emitting Obligations

If the template execution emits obligations, those can also be specified in the scenario as one of the **Then** steps.

For instance, the following shows a scenario for the [Rental Deposit](https://templates.accordproject.org/ip-payment@0.10.1.html) template, which describes the expected list of obligations that should be emitted for a given request:
```gherkin
Feature: Rental Deposit
  This describe the expected behavior for the Accord Project's rental deposit contract

  Background:
    Given the default contract

  Scenario: The property was inspected and there was damage
    When the current time is "2018-01-02T16:34:00Z"
    And it receives the default request
    Then it should respond with
"""
{
   "$class": "org.accordproject.rentaldeposit.PropertyInspectionResponse",
    "balance": {
      "$class": "org.accordproject.money.MonetaryAmount",
      "currencyCode" : "USD",
      "doubleValue" : 1550
    }
}
"""
    And the following obligations should have been emitted
"""
[
    {
        "$class": "org.accordproject.cicero.runtime.PaymentObligation",
        "amount": {
            "$class": "org.accordproject.money.MonetaryAmount",
            "doubleValue": 1550,
            "currencyCode": "USD"
        }
    }
]
"""
```

---
id: version-0.21-ref-concerto-api
title: Node.js API
original_id: ref-concerto-api

<a name="module_concerto-core"></a>

## concerto-core
Concerto module. Concerto is a framework for defining domain
specific models.

```
* [concerto-core](#module_concerto-core)
    * _static_
        * [.ModelLoader](#module_concerto-core.ModelLoader)
            * [.loadModelManager(ctoSystemFile, ctoFiles)](#module_concerto-
core.ModelLoader.loadModelManager) ⇒ <code>object</code>
            * [.loadModelManagerFromModelFiles(ctoSystemFile, modelFiles,
[fileNames])](#module_concerto-core.ModelLoader.loadModelManagerFromModelFiles) ⇒
<code>object</code>
        * [.DecoratorFactory](#module_concerto-core.DecoratorFactory)
            * *[.newDecorator(parent, ast)](#module_concerto-
core.DecoratorFactory+newDecorator) ⇒ <code>Decorator</code>*
    * _inner_
        * [~BaseException](#module_concerto-core.BaseException) ⇐
<code>Error</code>
            * [new BaseException(message, component)](#new_module_concerto-
core.BaseException_new)
        * [~BaseFileException](#module_concerto-core.BaseFileException) ⇐
<code>BaseException</code>
            * [new BaseFileException(message, fileLocation, fullMessage, fileName,
component)](#new_module_concerto-core.BaseFileException_new)
            * [.getFileLocation()](#module_concerto-
core.BaseFileException+getFileLocation) ⇒ <code>string</code>
            * [.getShortMessage()](#module_concerto-
core.BaseFileException+getShortMessage) ⇒ <code>string</code>
            * [.getFileName()](#module_concerto-core.BaseFileException+getFileName)
⇒ <code>string</code>
        * [~Factory](#module_concerto-core.Factory)
            * [new Factory(modelManager)](#new_module_concerto-core.Factory_new)
            * _instance_
                * [.newResource(ns, type, id, [options])](#module_concerto-
core.Factory+newResource) ⇒ <code>Resource</code>
                * [.newConcept(ns, type, [options])](#module_concerto-
core.Factory+newConcept) ⇒ <code>Resource</code>
                * [.newRelationship(ns, type, id)](#module_concerto-
core.Factory+newRelationship) ⇒ <code>Relationship</code>
                * [.newTransaction(ns, type, [id], [options])](#module_concerto-
core.Factory+newTransaction) ⇒ <code>Resource</code>
                * [.newEvent(ns, type, [id], [options])](#module_concerto-
core.Factory+newEvent) ⇒ <code>Resource</code>
            * _static_
                * [.Symbol.hasInstance(object)](#module_concerto-
core.Factory.Symbol.hasInstance) ⇒ <code>boolean</code>
        * [~ModelManager](#module_concerto-core.ModelManager)
            * [new ModelManager()](#new_module_concerto-core.ModelManager_new)
            * _instance_
                * [.validateModelFile(modelFile, fileName)](#module_concerto-
core.ModelManager+validateModelFile)
                * [.addModelFile(modelFile, fileName, [disableValidation],
[systemModelTable])](#module_concerto-core.ModelManager+addModelFile) ⇒
<code>Object</code>
                * [.updateModelFile(modelFile, fileName, [disableValidation])]
(#module_concerto-core.ModelManager+updateModelFile) ⇒ <code>Object</code>
                * [.deleteModelFile(namespace)](#module_concerto-
core.ModelManager+deleteModelFile)
                * [.addModelFiles(modelFiles, [fileNames], [disableValidation],
[systemModelTable])](#module_concerto-core.ModelManager+addModelFiles) ⇒
<code>Array.&lt;Object&gt;</code>
                * [.validateModelFiles()](#module_concerto-
core.ModelManager+validateModelFiles)
```

```
                * [.updateExternalModels([options], [modelFileDownloader])]
(#module_concerto-core.ModelManager+updateExternalModels) ⇒ <code>Promise</code>
                * [.writeModelsToFileSystem(path, [options])](#module_concerto-
core.ModelManager+writeModelsToFileSystem)
                * [.getModels([options])](#module_concerto-
core.ModelManager+getModels) ⇒ <code>Array.&lt;{name:string,
content:string}&gt;</code>
                * [.clearModelFiles()](#module_concerto-
core.ModelManager+clearModelFiles)
                * [.getNamespaces()](#module_concerto-
core.ModelManager+getNamespaces) ⇒ <code>Array.&lt;string&gt;</code>
                * [.getSystemTypes()](#module_concerto-
core.ModelManager+getSystemTypes) ⇒ <code>Array.&lt;ClassDeclaration&gt;</code>
                * [.getAssetDeclarations(includeSystemType)](#module_concerto-
core.ModelManager+getAssetDeclarations) ⇒
<code>Array.&lt;AssetDeclaration&gt;</code>
                * [.getTransactionDeclarations(includeSystemType)]
(#module_concerto-core.ModelManager+getTransactionDeclarations) ⇒
<code>Array.&lt;TransactionDeclaration&gt;</code>
                * [.getEventDeclarations(includeSystemType)](#module_concerto-
core.ModelManager+getEventDeclarations) ⇒
<code>Array.&lt;EventDeclaration&gt;</code>
                * [.getParticipantDeclarations(includeSystemType)]
(#module_concerto-core.ModelManager+getParticipantDeclarations) ⇒
<code>Array.&lt;ParticipantDeclaration&gt;</code>
                * [.getEnumDeclarations(includeSystemType)](#module_concerto-
core.ModelManager+getEnumDeclarations) ⇒ <code>Array.&lt;EnumDeclaration&gt;</code>
                * [.getConceptDeclarations(includeSystemType)](#module_concerto-
core.ModelManager+getConceptDeclarations) ⇒
<code>Array.&lt;ConceptDeclaration&gt;</code>
                * [.getFactory()](#module_concerto-core.ModelManager+getFactory) ⇒
<code>Factory</code>
                * [.getSerializer()](#module_concerto-
core.ModelManager+getSerializer) ⇒ <code>Serializer</code>
                * [.getDecoratorFactories()](#module_concerto-
core.ModelManager+getDecoratorFactories) ⇒
<code>Array.&lt;DecoratorFactory&gt;</code>
                * [.addDecoratorFactory(factory)](#module_concerto-
core.ModelManager+addDecoratorFactory)
            * _static_
                * [.Symbol.hasInstance(object)](#module_concerto-
core.ModelManager.Symbol.hasInstance) ⇒ <code>boolean</code>
        * [~SecurityException](#module_concerto-core.SecurityException) ⇐
<code>BaseException</code>
            * [new SecurityException(message)](#new_module_concerto-
core.SecurityException_new)
        * [~Serializer](#module_concerto-core.Serializer)
            * [new Serializer(factory, modelManager)](#new_module_concerto-
core.Serializer_new)
            * _instance_
                * [.setDefaultOptions(newDefaultOptions)](#module_concerto-
core.Serializer+setDefaultOptions)
                * [.toJSON(resource, [options])](#module_concerto-
core.Serializer+toJSON) ⇒ <code>Object</code>
                * [.fromJSON(jsonObject, options)](#module_concerto-
core.Serializer+fromJSON) ⇒ <code>Resource</code>
            * _static_
                * [.Symbol.hasInstance(object)](#module_concerto-
core.Serializer.Symbol.hasInstance) ⇒ <code>boolean</code>
```

```
        * [~AssetDeclaration](#module_concerto-core.AssetDeclaration) ⇐
<code>ClassDeclaration</code>
            * [new AssetDeclaration(modelFile, ast)](#new_module_concerto-
core.AssetDeclaration_new)
            * _instance_
                * [.isRelationshipTarget()](#module_concerto-
core.AssetDeclaration+isRelationshipTarget) ⇒ <code>boolean</code>
                * [.getSystemType()](#module_concerto-
core.AssetDeclaration+getSystemType) ⇒ <code>string</code>
            * _static_
                * [.Symbol.hasInstance(object)](#module_concerto-
core.AssetDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>
        * *[~ClassDeclaration](#module_concerto-core.ClassDeclaration)*
            * *[new ClassDeclaration(modelFile, ast)](#new_module_concerto-
core.ClassDeclaration_new)*
            * _instance_
                * *[._resolveSuperType()](#module_concerto-
core.ClassDeclaration+_resolveSuperType) ⇒ <code>ClassDeclaration</code>*
                * *[.getSystemType()](#module_concerto-
core.ClassDeclaration+getSystemType) ⇒ <code>string</code>*
                * *[.isAbstract()](#module_concerto-
core.ClassDeclaration+isAbstract) ⇒ <code>boolean</code>*
                * *[.isEnum()](#module_concerto-core.ClassDeclaration+isEnum) ⇒
<code>boolean</code>*
                * *[.isConcept()](#module_concerto-core.ClassDeclaration+isConcept)
⇒ <code>boolean</code>*
                * *[.isEvent()](#module_concerto-core.ClassDeclaration+isEvent) ⇒
<code>boolean</code>*
                * *[.isRelationshipTarget()](#module_concerto-
core.ClassDeclaration+isRelationshipTarget) ⇒ <code>boolean</code>*
                * *[.isSystemRelationshipTarget()](#module_concerto-
core.ClassDeclaration+isSystemRelationshipTarget) ⇒ <code>boolean</code>*
                * *[.isSystemType()](#module_concerto-
core.ClassDeclaration+isSystemType) ⇒ <code>boolean</code>*
                * *[.isSystemCoreType()](#module_concerto-
core.ClassDeclaration+isSystemCoreType) ⇒ <code>boolean</code>*
                * *[.getName()](#module_concerto-core.ClassDeclaration+getName) ⇒
<code>string</code>*
                * *[.getNamespace()](#module_concerto-
core.ClassDeclaration+getNamespace) ⇒ <code>String</code>*
                * *[.getFullyQualifiedName()](#module_concerto-
core.ClassDeclaration+getFullyQualifiedName) ⇒ <code>string</code>*
                * *[.getIdentifierFieldName()](#module_concerto-
core.ClassDeclaration+getIdentifierFieldName) ⇒ <code>string</code>*
                * *[.getOwnProperty(name)](#module_concerto-
core.ClassDeclaration+getOwnProperty) ⇒ <code>Property</code>*
                * *[.getOwnProperties()](#module_concerto-
core.ClassDeclaration+getOwnProperties) ⇒ <code>Array.&lt;Property&gt;</code>*
                * *[.getSuperType()](#module_concerto-
core.ClassDeclaration+getSuperType) ⇒ <code>string</code>*
                * *[.getSuperTypeDeclaration()](#module_concerto-
core.ClassDeclaration+getSuperTypeDeclaration) ⇒ <code>ClassDeclaration</code>*
                * *[.getAssignableClassDeclarations()](#module_concerto-
core.ClassDeclaration+getAssignableClassDeclarations) ⇒
<code>Array.&lt;ClassDeclaration&gt;</code>*
                * *[.getAllSuperTypeDeclarations()](#module_concerto-
core.ClassDeclaration+getAllSuperTypeDeclarations) ⇒
<code>Array.&lt;ClassDeclaration&gt;</code>*
                * *[.getProperty(name)](#module_concerto-
core.ClassDeclaration+getProperty) ⇒ <code>Property</code>*
                * *[.getProperties()](#module_concerto-
core.ClassDeclaration+getProperties) ⇒ <code>Array.&lt;Property&gt;</code>*
                * *[.getNestedProperty(propertyPath)](#module_concerto-
core.ClassDeclaration+getNestedProperty) ⇒ <code>Property</code>*
                * *[.toString()](#module_concerto-core.ClassDeclaration+toString) ⇒
<code>String</code>*
            * _static_
                * *[.Symbol.hasInstance(object)](#module_concerto-
core.ClassDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>*
        * [~ConceptDeclaration](#module_concerto-core.ConceptDeclaration) ⇐
<code>ClassDeclaration</code>
            * [new ConceptDeclaration(modelFile, ast)](#new_module_concerto-
core.ConceptDeclaration_new)
            * _instance_
                * [.isConcept()](#module_concerto-
core.ConceptDeclaration+isConcept) ⇒ <code>boolean</code>
            * _static_
                * [.Symbol.hasInstance(object)](#module_concerto-
core.ConceptDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>
        * [~Decorator](#module_concerto-core.Decorator)
            * [new Decorator(parent, ast)](#new_module_concerto-core.Decorator_new)
            * [.getParent()](#module_concerto-core.Decorator+getParent) ⇒
<code>ClassDeclaration</code> \| <code>Property</code>
            * [.getName()](#module_concerto-core.Decorator+getName) ⇒
<code>string</code>
            * [.getArguments()](#module_concerto-core.Decorator+getArguments) ⇒
<code>Array.&lt;object&gt;</code>
        * [~EnumDeclaration](#module_concerto-core.EnumDeclaration) ⇐
<code>ClassDeclaration</code>
            * [new EnumDeclaration(modelFile, ast)](#new_module_concerto-
core.EnumDeclaration_new)
            * _instance_
                * [.isEnum()](#module_concerto-core.EnumDeclaration+isEnum) ⇒
<code>boolean</code>
                * [.toString()](#module_concerto-core.EnumDeclaration+toString) ⇒
<code>String</code>
            * _static_
                * [.Symbol.hasInstance(object)](#module_concerto-
core.EnumDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>
        * [~EnumValueDeclaration](#module_concerto-core.EnumValueDeclaration) ⇐
<code>Property</code>
            * [new EnumValueDeclaration(parent, ast)](#new_module_concerto-
core.EnumValueDeclaration_new)
            * [.Symbol.hasInstance(object)](#module_concerto-
core.EnumValueDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>
        * [~EventDeclaration](#module_concerto-core.EventDeclaration) ⇐
<code>ClassDeclaration</code>
            * [new EventDeclaration(modelFile, ast)](#new_module_concerto-
core.EventDeclaration_new)
            * _instance_
                * [.getSystemType()](#module_concerto-
core.EventDeclaration+getSystemType) ⇒ <code>string</code>
                * [.isEvent()](#module_concerto-core.EventDeclaration+isEvent) ⇒
<code>boolean</code>
            * _static_
                * [.Symbol.hasInstance(object)](#module_concerto-
core.EventDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>
        * [~Introspector](#module_concerto-core.Introspector)
```

```
            * [new Introspector(modelManager)](#new_module_concerto-
core.Introspector_new)
            * [.getClassDeclarations()](#module_concerto-
core.Introspector+getClassDeclarations) ⇒
<code>Array.&lt;ClassDeclaration&gt;</code>
            * [.getClassDeclaration(fullyQualifiedTypeName)](#module_concerto-
core.Introspector+getClassDeclaration) ⇒ <code>ClassDeclaration</code>
        * [~ModelFile](#module_concerto-core.ModelFile)
            * [new ModelFile(modelManager, definitions, [fileName],
[isSystemModelFile])](#new_module_concerto-core.ModelFile_new)
            * _instance_
                * [.isExternal()](#module_concerto-core.ModelFile+isExternal) ⇒
<code>boolean</code>
                * [.getModelManager()](#module_concerto-
core.ModelFile+getModelManager) ⇒ <code>ModelManager</code>
                * [.getImports()](#module_concerto-core.ModelFile+getImports) ⇒
<code>Array.&lt;string&gt;</code>
                * [.isDefined(type)](#module_concerto-core.ModelFile+isDefined) ⇒
<code>boolean</code>
                * [.getLocalType(type)](#module_concerto-
core.ModelFile+getLocalType) ⇒ <code>ClassDeclaration</code>
                * [.getAssetDeclaration(name)](#module_concerto-
core.ModelFile+getAssetDeclaration) ⇒ <code>AssetDeclaration</code>
                * [.getTransactionDeclaration(name)](#module_concerto-
core.ModelFile+getTransactionDeclaration) ⇒ <code>TransactionDeclaration</code>
                * [.getEventDeclaration(name)](#module_concerto-
core.ModelFile+getEventDeclaration) ⇒ <code>EventDeclaration</code>
                * [.getParticipantDeclaration(name)](#module_concerto-
core.ModelFile+getParticipantDeclaration) ⇒ <code>ParticipantDeclaration</code>
                * [.getNamespace()](#module_concerto-core.ModelFile+getNamespace) ⇒
<code>string</code>
                * [.getName()](#module_concerto-core.ModelFile+getName) ⇒
<code>string</code>
                * [.getAssetDeclarations(includeSystemType)](#module_concerto-
core.ModelFile+getAssetDeclarations) ⇒ <code>Array.&lt;AssetDeclaration&gt;</code>
                * [.getTransactionDeclarations(includeSystemType)]
(#module_concerto-core.ModelFile+getTransactionDeclarations) ⇒
<code>Array.&lt;TransactionDeclaration&gt;</code>
                * [.getEventDeclarations(includeSystemType)](#module_concerto-
core.ModelFile+getEventDeclarations) ⇒ <code>Array.&lt;EventDeclaration&gt;</code>
                * [.getParticipantDeclarations(includeSystemType)]
(#module_concerto-core.ModelFile+getParticipantDeclarations) ⇒
<code>Array.&lt;ParticipantDeclaration&gt;</code>
                * [.getConceptDeclarations(includeSystemType)](#module_concerto-
core.ModelFile+getConceptDeclarations) ⇒
<code>Array.&lt;ConceptDeclaration&gt;</code>
                * [.getEnumDeclarations(includeSystemType)](#module_concerto-
core.ModelFile+getEnumDeclarations) ⇒ <code>Array.&lt;EnumDeclaration&gt;</code>
                * [.getDeclarations(type, includeSystemType)](#module_concerto-
core.ModelFile+getDeclarations) ⇒ <code>Array.&lt;ClassDeclaration&gt;</code>
                * [.getAllDeclarations()](#module_concerto-
core.ModelFile+getAllDeclarations) ⇒ <code>Array.&lt;ClassDeclaration&gt;</code>
                * [.getDefinitions()](#module_concerto-
core.ModelFile+getDefinitions) ⇒ <code>string</code>
                * [.isSystemModelFile()](#module_concerto-
core.ModelFile+isSystemModelFile) ⇒ <code>boolean</code>
            * _static_
                * [.Symbol.hasInstance(object)](#module_concerto-
core.ModelFile.Symbol.hasInstance) ⇒ <code>boolean</code>
```

```
        * [~ParticipantDeclaration](#module_concerto-core.ParticipantDeclaration) ⇐
<code>ClassDeclaration</code>
            * [new ParticipantDeclaration(modelFile, ast)](#new_module_concerto-
core.ParticipantDeclaration_new)
            * _instance_
                * [.isRelationshipTarget()](#module_concerto-
core.ParticipantDeclaration+isRelationshipTarget) ⇒ <code>boolean</code>
                * [.getSystemType()](#module_concerto-
core.ParticipantDeclaration+getSystemType) ⇒ <code>string</code>
            * _static_
                * [.Symbol.hasInstance(object)](#module_concerto-
core.ParticipantDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>
        * [~Property](#module_concerto-core.Property)
            * [new Property(parent, ast)](#new_module_concerto-core.Property_new)
            * _instance_
                * [.getParent()](#module_concerto-core.Property+getParent) ⇒
<code>ClassDeclaration</code>
                * [.getName()](#module_concerto-core.Property+getName) ⇒
<code>string</code>
                * [.getType()](#module_concerto-core.Property+getType) ⇒
<code>string</code>
                * [.isOptional()](#module_concerto-core.Property+isOptional) ⇒
<code>boolean</code>
                * [.getFullyQualifiedTypeName()](#module_concerto-
core.Property+getFullyQualifiedTypeName) ⇒ <code>string</code>
                * [.getFullyQualifiedName()](#module_concerto-
core.Property+getFullyQualifiedName) ⇒ <code>string</code>
                * [.getNamespace()](#module_concerto-core.Property+getNamespace) ⇒
<code>string</code>
                * [.isArray()](#module_concerto-core.Property+isArray) ⇒
<code>boolean</code>
                * [.isTypeEnum()](#module_concerto-core.Property+isTypeEnum) ⇒
<code>boolean</code>
                * [.isPrimitive()](#module_concerto-core.Property+isPrimitive) ⇒
<code>boolean</code>
            * _static_
                * [.Symbol.hasInstance(object)](#module_concerto-
core.Property.Symbol.hasInstance) ⇒ <code>boolean</code>
        * [~RelationshipDeclaration](#module_concerto-core.RelationshipDeclaration)
⇐ <code>Property</code>
            * [new RelationshipDeclaration(parent, ast)](#new_module_concerto-
core.RelationshipDeclaration_new)
            * _instance_
                * [.toString()](#module_concerto-
core.RelationshipDeclaration+toString) ⇒ <code>String</code>
            * _static_
                * [.Symbol.hasInstance(object)](#module_concerto-
core.RelationshipDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>
        * [~TransactionDeclaration](#module_concerto-core.TransactionDeclaration) ⇐
<code>ClassDeclaration</code>
            * [new TransactionDeclaration(modelFile, ast)](#new_module_concerto-
core.TransactionDeclaration_new)
            * _instance_
                * [.getSystemType()](#module_concerto-
core.TransactionDeclaration+getSystemType) ⇒ <code>string</code>
            * _static_
                * [.Symbol.hasInstance(object)](#module_concerto-
core.TransactionDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>
```

```
<a name="module_concerto-core.ModelLoader"></a>

### concerto-core.ModelLoader
Create a ModelManager from model files, with an optional system model.

If a ctoFile is not provided, the Accord Project system model is used.

**Kind**: static class of [<code>concerto-core</code>](#module_concerto-core)

* [.ModelLoader](#module_concerto-core.ModelLoader)
    * [.loadModelManager(ctoSystemFile, ctoFiles)](#module_concerto-
core.ModelLoader.loadModelManager) ⇒ <code>object</code>
    * [.loadModelManagerFromModelFiles(ctoSystemFile, modelFiles, [fileNames])]
(#module_concerto-core.ModelLoader.loadModelManagerFromModelFiles) ⇒
<code>object</code>

<a name="module_concerto-core.ModelLoader.loadModelManager"></a>

#### ModelLoader.loadModelManager(ctoSystemFile, ctoFiles) ⇒ <code>object</code>
Load system and models in a new model manager

**Kind**: static method of [<code>ModelLoader</code>](#module_concerto-
core.ModelLoader)
**Returns**: <code>object</code> - the model manager

| Param | Type | Description |
| --- | --- | --- |
| ctoSystemFile | <code>string</code> | the system model file |
| ctoFiles | <code>Array.&lt;string&gt;</code> | the CTO files (can be local file
paths or URLs) |

<a name="module_concerto-core.ModelLoader.loadModelManagerFromModelFiles"></a>

#### ModelLoader.loadModelManagerFromModelFiles(ctoSystemFile, modelFiles,
[fileNames]) ⇒ <code>object</code>
Load system and models in a new model manager from model files objects

**Kind**: static method of [<code>ModelLoader</code>](#module_concerto-
core.ModelLoader)
**Returns**: <code>object</code> - the model manager

| Param | Type | Description |
| --- | --- | --- |
| ctoSystemFile | <code>string</code> | the system model file |
| modelFiles | <code>Array.&lt;object&gt;</code> | An array of Concerto files as
strings or ModelFile objects. |
| [fileNames] | <code>Array.&lt;string&gt;</code> | An optional array of file names
to associate with the model files |

<a name="module_concerto-core.DecoratorFactory"></a>

### concerto-core.DecoratorFactory
An interface for a class that processes a decorator and returns a specific
implementation class for that decorator.

**Kind**: static class of [<code>concerto-core</code>](#module_concerto-core)
<a name="module_concerto-core.DecoratorFactory+newDecorator"></a>

#### *decoratorFactory.newDecorator(parent, ast) ⇒ <code>Decorator</code>*
```

```
Process the decorator, and return a specific implementation class for that
decorator, or return null if this decorator is not handled by this processor.

**Kind**: instance abstract method of [<code>DecoratorFactory</code>]
(#module_concerto-core.DecoratorFactory)
**Returns**: <code>Decorator</code> - The decorator.

| Param | Type | Description |
| --- | --- | --- |
| parent | <code>ClassDeclaration</code> \| <code>Property</code> | the owner of
this property |
| ast | <code>Object</code> | The AST created by the parser |

<a name="module_concerto-core.BaseException"></a>

### concerto-core~BaseException ⇐ <code>Error</code>
A base class for all Concerto exceptions

**Kind**: inner class of [<code>concerto-core</code>](#module_concerto-core)
**Extends**: <code>Error</code>
<a name="new_module_concerto-core.BaseException_new"></a>

#### new BaseException(message, component)
Create the BaseException.


| Param | Type | Description |
| --- | --- | --- |
| message | <code>string</code> | The exception message. |
| component | <code>string</code> | The optional component which throws this error.
|

<a name="module_concerto-core.BaseFileException"></a>

### concerto-core~BaseFileException ⇐ <code>BaseException</code>
Exception throws when a Concerto file is semantically invalid

**Kind**: inner class of [<code>concerto-core</code>](#module_concerto-core)
**Extends**: <code>BaseException</code>
**See**: [BaseException](BaseException)

* [~BaseFileException](#module_concerto-core.BaseFileException) ⇐
<code>BaseException</code>
    * [new BaseFileException(message, fileLocation, fullMessage, fileName,
component)](#new_module_concerto-core.BaseFileException_new)
    * [.getFileLocation()](#module_concerto-core.BaseFileException+getFileLocation)
⇒ <code>string</code>
    * [.getShortMessage()](#module_concerto-core.BaseFileException+getShortMessage)
⇒ <code>string</code>
    * [.getFileName()](#module_concerto-core.BaseFileException+getFileName) ⇒
<code>string</code>

<a name="new_module_concerto-core.BaseFileException_new"></a>

#### new BaseFileException(message, fileLocation, fullMessage, fileName, component)
Create an BaseFileException


| Param | Type | Description |
| --- | --- | --- |
| message | <code>string</code> | the message for the exception |
| fileLocation | <code>string</code> | the optional file location associated with
the exception |
| fullMessage | <code>string</code> | the optional full message text |
| fileName | <code>string</code> | the optional file name |
| component | <code>string</code> | the optional component which throws this error
|

<a name="module_concerto-core.BaseFileException+getFileLocation"></a>

#### baseFileException.getFileLocation() ⇒ <code>string</code>
Returns the file location associated with the exception or null

**Kind**: instance method of [<code>BaseFileException</code>](#module_concerto-
core.BaseFileException)
**Returns**: <code>string</code> - the optional location associated with the
exception
<a name="module_concerto-core.BaseFileException+getShortMessage"></a>

#### baseFileException.getShortMessage() ⇒ <code>string</code>
Returns the error message without the location of the error

**Kind**: instance method of [<code>BaseFileException</code>](#module_concerto-
core.BaseFileException)
**Returns**: <code>string</code> - the error message
<a name="module_concerto-core.BaseFileException+getFileName"></a>

#### baseFileException.getFileName() ⇒ <code>string</code>
Returns the fileName for the error

**Kind**: instance method of [<code>BaseFileException</code>](#module_concerto-
core.BaseFileException)
**Returns**: <code>string</code> - the file name or null
<a name="module_concerto-core.Factory"></a>

### concerto-core~Factory
Use the Factory to create instances of Resource: transactions, participants
and assets.

**Kind**: inner class of [<code>concerto-core</code>](#module_concerto-core)

* [~Factory](#module_concerto-core.Factory)
    * [new Factory(modelManager)](#new_module_concerto-core.Factory_new)
    * _instance_
        * [.newResource(ns, type, id, [options])](#module_concerto-
core.Factory+newResource) ⇒ <code>Resource</code>
        * [.newConcept(ns, type, [options])](#module_concerto-
core.Factory+newConcept) ⇒ <code>Resource</code>
        * [.newRelationship(ns, type, id)](#module_concerto-
core.Factory+newRelationship) ⇒ <code>Relationship</code>
        * [.newTransaction(ns, type, [id], [options])](#module_concerto-
core.Factory+newTransaction) ⇒ <code>Resource</code>
        * [.newEvent(ns, type, [id], [options])](#module_concerto-
core.Factory+newEvent) ⇒ <code>Resource</code>
    * _static_
        * [.Symbol.hasInstance(object)](#module_concerto-
core.Factory.Symbol.hasInstance) ⇒ <code>boolean</code>
```

```
<a name="new_module_concerto-core.Factory_new"></a>

#### new Factory(modelManager)
Create the factory.


| Param | Type | Description |
| --- | --- | --- |
| modelManager | <code>ModelManager</code> | The ModelManager to use for this
registry |

<a name="module_concerto-core.Factory+newResource"></a>

#### factory.newResource(ns, type, id, [options]) ⇒ <code>Resource</code>
Create a new Resource with a given namespace, type name and id

**Kind**: instance method of [<code>Factory</code>](#module_concerto-core.Factory)
**Returns**: <code>Resource</code> - the new instance
**Throws**:

- <code>TypeNotFoundException</code> if the type is not registered with the
ModelManager


| Param | Type | Description |
| --- | --- | --- |
| ns | <code>String</code> | the namespace of the Resource |
| type | <code>String</code> | the type of the Resource |
| id | <code>String</code> | the identifier |
| [options] | <code>Object</code> | an optional set of options |
| [options.disableValidation] | <code>boolean</code> | pass true if you want the
factory to return a [Resource](Resource) instead of a [ValidatedResource]
(ValidatedResource). Defaults to false. |
| [options.generate] | <code>String</code> | Pass one of: <dl>
<dt>sample</dt><dd>return a resource instance with generated sample data.</dd>
<dt>empty</dt><dd>return a resource instance with empty property values.</dd></dl>
|
| [options.includeOptionalFields] | <code>boolean</code> | if
<code>options.generate</code> is specified, whether optional fields should be
generated. |
| [options.allowEmptyId] | <code>boolean</code> | if
<code>options.allowEmptyId</code> is specified as true, a zero length string for id
is allowed (allows it to be filled in later). |

<a name="module_concerto-core.Factory+newConcept"></a>

#### factory.newConcept(ns, type, [options]) ⇒ <code>Resource</code>
Create a new Concept with a given namespace and type name

**Kind**: instance method of [<code>Factory</code>](#module_concerto-core.Factory)
**Returns**: <code>Resource</code> - the new instance
**Throws**:

- <code>TypeNotFoundException</code> if the type is not registered with the
ModelManager


| Param | Type | Description |
| --- | --- | --- |
```

ns	<code>String</code>	the namespace of the Concept
type	<code>String</code>	the type of the Concept
[options]	<code>Object</code>	an optional set of options
[options.disableValidation]	<code>boolean</code>	pass true if you want the
factory to return a Concept instead of a [ValidatedConcept]
(ValidatedConcept). Defaults to false.
[options.generate]	<code>String</code>	Pass one of: <dl>
<dt>sample</dt><dd>return a resource instance with generated sample data.</dd>
<dt>empty</dt><dd>return a resource instance with empty property values.</dd></dl>
[options.includeOptionalFields]	<code>boolean</code>	if
<code>options.generate</code> is specified, whether optional fields should be
generated.

<a name="module_concerto-core.Factory+newRelationship"></a>

#### factory.newRelationship(ns, type, id) ⇒ <code>Relationship</code>
Create a new Relationship with a given namespace, type and identifier.
A relationship is a typed pointer to an instance. I.e the relationship
with `namespace = 'org.example'`, `type = 'Vehicle'` and `id = 'ABC' creates`
a pointer that points at an instance of org.example.Vehicle with the id
ABC.

Kind: instance method of [<code>Factory</code>](#module_concerto-core.Factory)
Returns: <code>Relationship</code> - - the new relationship instance
Throws:

- <code>TypeNotFoundException</code> if the type is not registered with the
ModelManager

Param	Type	Description
ns	<code>String</code>	the namespace of the Resource
type	<code>String</code>	the type of the Resource
id	<code>String</code>	the identifier

<a name="module_concerto-core.Factory+newTransaction"></a>

#### factory.newTransaction(ns, type, [id], [options]) ⇒ <code>Resource</code>
Create a new transaction object. The identifier of the transaction is
set to a UUID.

Kind: instance method of [<code>Factory</code>](#module_concerto-core.Factory)
Returns: <code>Resource</code> - A resource for the new transaction.

Param	Type	Description
ns	<code>String</code>	the namespace of the transaction.
type	<code>String</code>	the type of the transaction.
[id]	<code>String</code>	an optional identifier for the transaction; if you
do not specify one then an identifier will be automatically generated.
[options]	<code>Object</code>	an optional set of options
[options.generate]	<code>String</code>	Pass one of: <dl>
<dt>sample</dt><dd>return a resource instance with generated sample data.</dd>
<dt>empty</dt><dd>return a resource instance with empty property values.</dd></dl>
[options.includeOptionalFields]	<code>boolean</code>	if
<code>options.generate</code> is specified, whether optional fields should be

generated. |
| [options.allowEmptyId] | <code>boolean</code> | if
<code>options.allowEmptyId</code> is specified as true, a zero length string for id
is allowed (allows it to be filled in later). |

<a name="module_concerto-core.Factory+newEvent"></a>

#### factory.newEvent(ns, type, [id], [options]) ⇒ <code>Resource</code>
Create a new event object. The identifier of the event is
set to a UUID.

Kind: instance method of [<code>Factory</code>](#module_concerto-core.Factory)
Returns: <code>Resource</code> - A resource for the new event.

Param	Type	Description
ns	<code>String</code>	the namespace of the event.
type	<code>String</code>	the type of the event.
[id]	<code>String</code>	an optional identifier for the event; if you do not
specify one then an identifier will be automatically generated.
[options]	<code>Object</code>	an optional set of options
[options.generate]	<code>String</code>	Pass one of: <dl>
<dt>sample</dt><dd>return a resource instance with generated sample data.</dd>
<dt>empty</dt><dd>return a resource instance with empty property values.</dd></dl>
[options.includeOptionalFields]	<code>boolean</code>	if
<code>options.generate</code> is specified, whether optional fields should be
generated.
[options.allowEmptyId]	<code>boolean</code>	if
<code>options.allowEmptyId</code> is specified as true, a zero length string for id
is allowed (allows it to be filled in later).

<a name="module_concerto-core.Factory.Symbol.hasInstance"></a>

#### Factory.Symbol.hasInstance(object) ⇒ <code>boolean</code>
Alternative instanceof that is reliable across different module instances

Kind: static method of [<code>Factory</code>](#module_concerto-core.Factory)
Returns: <code>boolean</code> - - True, if the object is an instance of a
Factory
See: https://github.com/hyperledger/composer-concerto/issues/47

Param	Type	Description
object	<code>object</code>	The object to test against

<a name="module_concerto-core.ModelManager"></a>

### concerto-core~ModelManager
Manages the Concerto model files.

The structure of Resources (Assets, Transactions, Participants) is
modelled
in a set of Concerto files. The contents of these files are managed
by the ModelManager. Each Concerto file has a single namespace and
contains
a set of asset, transaction and participant type definitions.

Concerto applications load their Concerto files and then call the [addModelFile]

(ModelManager#addModelFile)
method to register the Concerto file(s) with the ModelManager. The ModelManager
parses the text of the Concerto file and will make all defined types available
to other Concerto services, such as the Serializer (to convert
instances to/from JSON)
and Factory (to create instances).

Kind: inner class of [<code>concerto-core</code>](#module_concerto-core)

* [~ModelManager](#module_concerto-core.ModelManager)
    * [new ModelManager()](#new_module_concerto-core.ModelManager_new)
    * _instance_
        * [.validateModelFile(modelFile, fileName)](#module_concerto-
core.ModelManager+validateModelFile)
        * [.addModelFile(modelFile, fileName, [disableValidation],
[systemModelTable])](#module_concerto-core.ModelManager+addModelFile) ⇒
<code>Object</code>
        * [.updateModelFile(modelFile, fileName, [disableValidation])]
(#module_concerto-core.ModelManager+updateModelFile) ⇒ <code>Object</code>
        * [.deleteModelFile(namespace)](#module_concerto-
core.ModelManager+deleteModelFile)
        * [.addModelFiles(modelFiles, [fileNames], [disableValidation],
[systemModelTable])](#module_concerto-core.ModelManager+addModelFiles) ⇒
<code>Array.&lt;Object&gt;</code>
        * [.validateModelFiles()](#module_concerto-
core.ModelManager+validateModelFiles)
        * [.updateExternalModels([options], [modelFileDownloader])]
(#module_concerto-core.ModelManager+updateExternalModels) ⇒ <code>Promise</code>
        * [.writeModelsToFileSystem(path, [options])](#module_concerto-
core.ModelManager+writeModelsToFileSystem)
        * [.getModels([options])](#module_concerto-core.ModelManager+getModels) ⇒
<code>Array.&lt;{name:string, content:string}&gt;</code>
        * [.clearModelFiles()](#module_concerto-core.ModelManager+clearModelFiles)
        * [.getNamespaces()](#module_concerto-core.ModelManager+getNamespaces) ⇒
<code>Array.&lt;string&gt;</code>
        * [.getSystemTypes()](#module_concerto-core.ModelManager+getSystemTypes) ⇒
<code>Array.&lt;ClassDeclaration&gt;</code>
        * [.getAssetDeclarations(includeSystemType)](#module_concerto-
core.ModelManager+getAssetDeclarations) ⇒
<code>Array.&lt;AssetDeclaration&gt;</code>
        * [.getTransactionDeclarations(includeSystemType)](#module_concerto-
core.ModelManager+getTransactionDeclarations) ⇒
<code>Array.&lt;TransactionDeclaration&gt;</code>
        * [.getEventDeclarations(includeSystemType)](#module_concerto-
core.ModelManager+getEventDeclarations) ⇒
<code>Array.&lt;EventDeclaration&gt;</code>
        * [.getParticipantDeclarations(includeSystemType)](#module_concerto-
core.ModelManager+getParticipantDeclarations) ⇒
<code>Array.&lt;ParticipantDeclaration&gt;</code>
        * [.getEnumDeclarations(includeSystemType)](#module_concerto-
core.ModelManager+getEnumDeclarations) ⇒ <code>Array.&lt;EnumDeclaration&gt;</code>
        * [.getConceptDeclarations(includeSystemType)](#module_concerto-
core.ModelManager+getConceptDeclarations) ⇒
<code>Array.&lt;ConceptDeclaration&gt;</code>
        * [.getFactory()](#module_concerto-core.ModelManager+getFactory) ⇒
<code>Factory</code>
        * [.getSerializer()](#module_concerto-core.ModelManager+getSerializer) ⇒
<code>Serializer</code>
        * [.getDecoratorFactories()](#module_concerto-

```
core.ModelManager+getDecoratorFactories) ⇒
<code>Array.&lt;DecoratorFactory&gt;</code>
        * [.addDecoratorFactory(factory)](#module_concerto-
core.ModelManager+addDecoratorFactory)
    * _static_
        * [.Symbol.hasInstance(object)](#module_concerto-
core.ModelManager.Symbol.hasInstance) ⇒ <code>boolean</code>

<a name="new_module_concerto-core.ModelManager_new"></a>

#### new ModelManager()
Create the ModelManager.

<a name="module_concerto-core.ModelManager+validateModelFile"></a>

#### modelManager.validateModelFile(modelFile, fileName)
Validates a Concerto file (as a string) to the ModelManager.
Concerto files have a single namespace.

Note that if there are dependencies between multiple files the files
must be added in dependency order, or the addModelFiles method can be
used to add a set of files irrespective of dependencies.

**Kind**: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)
**Throws**:

- <code>IllegalModelException</code>


| Param | Type | Description |
| --- | --- | --- |
| modelFile | <code>string</code> | The Concerto file as a string |
| fileName | <code>string</code> | an optional file name to associate with the
model file |

<a name="module_concerto-core.ModelManager+addModelFile"></a>

#### modelManager.addModelFile(modelFile, fileName, [disableValidation],
[systemModelTable]) ⇒ <code>Object</code>
Adds a Concerto file (as a string) to the ModelManager.
Concerto files have a single namespace. If a Concerto file with the
same namespace has already been added to the ModelManager then it
will be replaced.
Note that if there are dependencies between multiple files the files
must be added in dependency order, or the addModelFiles method can be
used to add a set of files irrespective of dependencies.

**Kind**: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)
**Returns**: <code>Object</code> - The newly added model file (internal).
**Throws**:

- <code>IllegalModelException</code>


| Param | Type | Description |
| --- | --- | --- |
| modelFile | <code>string</code> | The Concerto file as a string |
```

| fileName | <code>string</code> | an optional file name to associate with the
model file |
| [disableValidation] | <code>boolean</code> | If true then the model files are not
validated |
| [systemModelTable] | <code>boolean</code> | A table that maps classes in the new
models to system types |

<a name="module_concerto-core.ModelManager+updateModelFile"></a>

#### modelManager.updateModelFile(modelFile, fileName, [disableValidation]) ⇒
<code>Object</code>
Updates a Concerto file (as a string) on the ModelManager.
Concerto files have a single namespace. If a Concerto file with the
same namespace has already been added to the ModelManager then it
will be replaced.

Kind: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)
Returns: <code>Object</code> - The newly added model file (internal).
Throws:

- <code>IllegalModelException</code>

Param	Type	Description
modelFile	<code>string</code>	The Concerto file as a string
fileName	<code>string</code>	an optional file name to associate with the
model file
[disableValidation]	<code>boolean</code>	If true then the model files are not
validated

<a name="module_concerto-core.ModelManager+deleteModelFile"></a>

#### modelManager.deleteModelFile(namespace)
Remove the Concerto file for a given namespace

Kind: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)

Param	Type	Description
namespace	<code>string</code>	The namespace of the model file to delete.

<a name="module_concerto-core.ModelManager+addModelFiles"></a>

#### modelManager.addModelFiles(modelFiles, [fileNames], [disableValidation],
[systemModelTable]) ⇒ <code>Array.&lt;Object&gt;</code>
Add a set of Concerto files to the model manager.

Kind: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)
Returns: <code>Array.&lt;Object&gt;</code> - The newly added model files
(internal).

Param	Type	Description
modelFiles	<code>Array.&lt;object&gt;</code>	An array of Concerto files as
strings or ModelFile objects.

[fileNames]	<code>Array.&lt;string&gt;</code>	An optional array of file names to associate with the model files
[disableValidation]	<code>boolean</code>	If true then the model files are not validated
[systemModelTable]	<code>boolean</code>	A table that maps classes in the new models to system types

<a name="module_concerto-core.ModelManager+validateModelFiles"></a>

#### modelManager.validateModelFiles()
Validates all models files in this model manager

Kind: instance method of [<code>ModelManager</code>](#module_concerto-core.ModelManager)
<a name="module_concerto-core.ModelManager+updateExternalModels"></a>

#### modelManager.updateExternalModels([options], [modelFileDownloader]) ⇒ <code>Promise</code>
Downloads all ModelFiles that are external dependencies and adds or
updates them in this ModelManager.

Kind: instance method of [<code>ModelManager</code>](#module_concerto-core.ModelManager)
Returns: <code>Promise</code> - a promise when the download and update operation is completed.
Throws:

- <code>IllegalModelException</code> if the models fail validation

Param	Type	Description
[options]	<code>Object</code>	Options object passed to ModelFileLoaders
[modelFileDownloader]	<code>ModelFileDownloader</code>	an optional ModelFileDownloader

<a name="module_concerto-core.ModelManager+writeModelsToFileSystem"></a>

#### modelManager.writeModelsToFileSystem(path, [options])
Write all models in this model manager to the specified path in the file system

Kind: instance method of [<code>ModelManager</code>](#module_concerto-core.ModelManager)

Param	Type	Description
path	<code>String</code>	to a local directory
[options]	<code>Object</code>	Options object
options.includeExternalModels	<code>boolean</code>	If true, external models are written to the file system. Defaults to true
options.includeSystemModels	<code>boolean</code>	If true, system models are written to the file system. Defaults to false

<a name="module_concerto-core.ModelManager+getModels"></a>

#### modelManager.getModels([options]) ⇒ <code>Array.&lt;{name:string, content:string}&gt;</code>
Gets all the CTO models

```
**Kind**: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)
**Returns**: <code>Array.&lt;{name:string, content:string}&gt;</code> - the name
and content of each CTO file

| Param | Type | Description |
| --- | --- | --- |
| [options] | <code>Object</code> | Options object |
| options.includeExternalModels | <code>boolean</code> | If true, external models
are written to the file system. Defaults to true |
| options.includeSystemModels | <code>boolean</code> | If true, system models are
written to the file system. Defaults to false |

<a name="module_concerto-core.ModelManager+clearModelFiles"></a>

#### modelManager.clearModelFiles()
Remove all registered Concerto files

**Kind**: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)
<a name="module_concerto-core.ModelManager+getNamespaces"></a>

#### modelManager.getNamespaces() ⇒ <code>Array.&lt;string&gt;</code>
Get the namespaces registered with the ModelManager.

**Kind**: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)
**Returns**: <code>Array.&lt;string&gt;</code> - namespaces - the namespaces that
have been registered.
<a name="module_concerto-core.ModelManager+getSystemTypes"></a>

#### modelManager.getSystemTypes() ⇒ <code>Array.&lt;ClassDeclaration&gt;</code>
Get all class declarations from system namespaces

**Kind**: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)
**Returns**: <code>Array.&lt;ClassDeclaration&gt;</code> - the ClassDeclarations
from system namespaces
<a name="module_concerto-core.ModelManager+getAssetDeclarations"></a>

#### modelManager.getAssetDeclarations(includeSystemType) ⇒
<code>Array.&lt;AssetDeclaration&gt;</code>
Get the AssetDeclarations defined in this model manager

**Kind**: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)
**Returns**: <code>Array.&lt;AssetDeclaration&gt;</code> - the AssetDeclarations
defined in the model manager

| Param | Type | Default | Description |
| --- | --- | --- | --- |
| includeSystemType | <code>Boolean</code> | <code>true</code> | Include the
decalarations of system type in returned data |

<a name="module_concerto-core.ModelManager+getTransactionDeclarations"></a>

#### modelManager.getTransactionDeclarations(includeSystemType) ⇒
<code>Array.&lt;TransactionDeclaration&gt;</code>
Get the TransactionDeclarations defined in this model manager
```

```
**Kind**: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)
**Returns**: <code>Array.&lt;TransactionDeclaration&gt;</code> - the
TransactionDeclarations defined in the model manager

| Param | Type | Default | Description |
| --- | --- | --- | --- |
| includeSystemType | <code>Boolean</code> | <code>true</code> | Include the
decalarations of system type in returned data |

<a name="module_concerto-core.ModelManager+getEventDeclarations"></a>

#### modelManager.getEventDeclarations(includeSystemType) ⇒
<code>Array.&lt;EventDeclaration&gt;</code>
Get the EventDeclarations defined in this model manager

**Kind**: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)
**Returns**: <code>Array.&lt;EventDeclaration&gt;</code> - the EventDeclaration
defined in the model manager

| Param | Type | Default | Description |
| --- | --- | --- | --- |
| includeSystemType | <code>Boolean</code> | <code>true</code> | Include the
decalarations of system type in returned data |

<a name="module_concerto-core.ModelManager+getParticipantDeclarations"></a>

#### modelManager.getParticipantDeclarations(includeSystemType) ⇒
<code>Array.&lt;ParticipantDeclaration&gt;</code>
Get the ParticipantDeclarations defined in this model manager

**Kind**: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)
**Returns**: <code>Array.&lt;ParticipantDeclaration&gt;</code> - the
ParticipantDeclaration defined in the model manager

| Param | Type | Default | Description |
| --- | --- | --- | --- |
| includeSystemType | <code>Boolean</code> | <code>true</code> | Include the
decalarations of system type in returned data |

<a name="module_concerto-core.ModelManager+getEnumDeclarations"></a>

#### modelManager.getEnumDeclarations(includeSystemType) ⇒
<code>Array.&lt;EnumDeclaration&gt;</code>
Get the EnumDeclarations defined in this model manager

**Kind**: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)
**Returns**: <code>Array.&lt;EnumDeclaration&gt;</code> - the EnumDeclaration
defined in the model manager

| Param | Type | Default | Description |
| --- | --- | --- | --- |
| includeSystemType | <code>Boolean</code> | <code>true</code> | Include the
decalarations of system type in returned data |
```

<a name="module_concerto-core.ModelManager+getConceptDeclarations"></a>

#### modelManager.getConceptDeclarations(includeSystemType) ⇒
<code>Array.&lt;ConceptDeclaration&gt;</code>
Get the Concepts defined in this model manager

Kind: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)
Returns: <code>Array.&lt;ConceptDeclaration&gt;</code> - the ConceptDeclaration
defined in the model manager

Param	Type	Default	Description
includeSystemType	<code>Boolean</code>	<code>true</code>	Include the
decalarations of system type in returned data

<a name="module_concerto-core.ModelManager+getFactory"></a>

#### modelManager.getFactory() ⇒ <code>Factory</code>
Get a factory for creating new instances of types defined in this model manager.

Kind: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)
Returns: <code>Factory</code> - A factory for creating new instances of types
defined in this model manager.
<a name="module_concerto-core.ModelManager+getSerializer"></a>

#### modelManager.getSerializer() ⇒ <code>Serializer</code>
Get a serializer for serializing instances of types defined in this model manager.

Kind: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)
Returns: <code>Serializer</code> - A serializer for serializing instances of
types defined in this model manager.
<a name="module_concerto-core.ModelManager+getDecoratorFactories"></a>

#### modelManager.getDecoratorFactories() ⇒
<code>Array.&lt;DecoratorFactory&gt;</code>
Get the decorator factories for this model manager.

Kind: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)
Returns: <code>Array.&lt;DecoratorFactory&gt;</code> - The decorator factories
for this model manager.
<a name="module_concerto-core.ModelManager+addDecoratorFactory"></a>

#### modelManager.addDecoratorFactory(factory)
Add a decorator factory to this model manager.

Kind: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)

Param	Type	Description
factory	<code>DecoratorFactory</code>	The decorator factory to add to this
model manager.

<a name="module_concerto-core.ModelManager.Symbol.hasInstance"></a>

```
#### ModelManager.Symbol.hasInstance(object) ⇒ <code>boolean</code>
Alternative instanceof that is reliable across different module instances

**Kind**: static method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)
**Returns**: <code>boolean</code> - - True, if the object is an instance of a
ModelManager
**See**: https://github.com/hyperledger/composer-concerto/issues/47

| Param | Type | Description |
| --- | --- | --- |
| object | <code>object</code> | The object to test against |

<a name="module_concerto-core.SecurityException"></a>

### concerto-core~SecurityException ⇐ <code>BaseException</code>
Class representing a security exception

**Kind**: inner class of [<code>concerto-core</code>](#module_concerto-core)
**Extends**: <code>BaseException</code>
**See**: See [BaseException](BaseException)
<a name="new_module_concerto-core.SecurityException_new"></a>

#### new SecurityException(message)
Create the SecurityException.


| Param | Type | Description |
| --- | --- | --- |
| message | <code>string</code> | The exception message. |

<a name="module_concerto-core.Serializer"></a>

### concerto-core~Serializer
Serialize Resources instances to/from various formats for long-term storage
(e.g. on the blockchain).

**Kind**: inner class of [<code>concerto-core</code>](#module_concerto-core)

* [~Serializer](#module_concerto-core.Serializer)
    * [new Serializer(factory, modelManager)](#new_module_concerto-
core.Serializer_new)
    * _instance_
        * [.setDefaultOptions(newDefaultOptions)](#module_concerto-
core.Serializer+setDefaultOptions)
        * [.toJSON(resource, [options])](#module_concerto-core.Serializer+toJSON) ⇒
<code>Object</code>
        * [.fromJSON(jsonObject, options)](#module_concerto-
core.Serializer+fromJSON) ⇒ <code>Resource</code>
    * _static_
        * [.Symbol.hasInstance(object)](#module_concerto-
core.Serializer.Symbol.hasInstance) ⇒ <code>boolean</code>

<a name="new_module_concerto-core.Serializer_new"></a>

#### new Serializer(factory, modelManager)
Create a Serializer.
```

Param	Type	Description
factory	<code>Factory</code>	The Factory to use to create instances
modelManager	<code>ModelManager</code>	The ModelManager to use for validation etc.

<a name="module_concerto-core.Serializer+setDefaultOptions"></a>

#### serializer.setDefaultOptions(newDefaultOptions)
Set the default options for the serializer.

Kind: instance method of [<code>Serializer</code>](#module_concerto-core.Serializer)

Param	Type	Description
newDefaultOptions	<code>Object</code>	The new default options for the serializer.

<a name="module_concerto-core.Serializer+toJSON"></a>

#### serializer.toJSON(resource, [options]) ⇒ <code>Object</code>
<p>
Convert a Resource to a JavaScript object suitable for long-term peristent storage.
</p>

Kind: instance method of [<code>Serializer</code>](#module_concerto-core.Serializer)
Returns: <code>Object</code> - - The Javascript Object that represents the resource
Throws:

- <code>Error</code> - throws an exception if resource is not an instance of Resource or fails validation.

Param	Type	Description
resource	<code>Resource</code>	The instance to convert to JSON
[options]	<code>Object</code>	the optional serialization options.
[options.validate]	<code>boolean</code>	validate the structure of the Resource with its model prior to serialization (default to true)
[options.convertResourcesToRelationships]	<code>boolean</code>	Convert resources that are specified for relationship fields into relationships, false by default.
[options.permitResourcesForRelationships]	<code>boolean</code>	Permit resources in the place of relationships (serializing them as resources), false by default.
[options.deduplicateResources]	<code>boolean</code>	Generate $id for resources and if a resources appears multiple times in the object graph only the first instance is serialized in full, subsequent instances are replaced with a reference to the $id
[options.convertResourcesToId]	<code>boolean</code>	Convert resources that are specified for relationship fields into their id, false by default.

<a name="module_concerto-core.Serializer+fromJSON"></a>

#### serializer.fromJSON(jsonObject, options) ⇒ <code>Resource</code>

```
Create a [Resource](Resource) from a JavaScript Object representation.
The JavaScript Object should have been created by calling the
[toJSON](Serializer#toJSON) API.

The Resource is populated based on the JavaScript object.

**Kind**: instance method of [<code>Serializer</code>](#module_concerto-
core.Serializer)
**Returns**: <code>Resource</code> - The new populated resource

| Param | Type | Description |
| --- | --- | --- |
| jsonObject | <code>Object</code> | The JavaScript Object for a Resource |
| options | <code>Object</code> | the optional serialization options |
| options.acceptResourcesForRelationships | <code>boolean</code> | handle JSON
objects in the place of strings for relationships, defaults to false. |
| options.validate | <code>boolean</code> | validate the structure of the Resource
with its model prior to serialization (default to true) |

<a name="module_concerto-core.Serializer.Symbol.hasInstance"></a>

#### Serializer.Symbol.hasInstance(object) ⇒ <code>boolean</code>
Alternative instanceof that is reliable across different module instances

**Kind**: static method of [<code>Serializer</code>](#module_concerto-
core.Serializer)
**Returns**: <code>boolean</code> - - True, if the object is an instance of a
Serializer
**See**: https://github.com/hyperledger/composer-concerto/issues/47

| Param | Type | Description |
| --- | --- | --- |
| object | <code>object</code> | The object to test against |

<a name="module_concerto-core.AssetDeclaration"></a>

### concerto-core~AssetDeclaration ⇐ <code>ClassDeclaration</code>
AssetDeclaration defines the schema (aka model or class) for
an Asset. It extends ClassDeclaration which manages a set of
fields, a super-type and the specification of an
identifying field.

**Kind**: inner class of [<code>concerto-core</code>](#module_concerto-core)
**Extends**: <code>ClassDeclaration</code>
**See**: See [ClassDeclaration](ClassDeclaration)

* [~AssetDeclaration](#module_concerto-core.AssetDeclaration) ⇐
<code>ClassDeclaration</code>
    * [new AssetDeclaration(modelFile, ast)](#new_module_concerto-
core.AssetDeclaration_new)
    * _instance_
        * [.isRelationshipTarget()](#module_concerto-
core.AssetDeclaration+isRelationshipTarget) ⇒ <code>boolean</code>
        * [.getSystemType()](#module_concerto-core.AssetDeclaration+getSystemType)
⇒ <code>string</code>
    * _static_
        * [.Symbol.hasInstance(object)](#module_concerto-
core.AssetDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>
```

<a name="new_module_concerto-core.AssetDeclaration_new"></a>

#### new AssetDeclaration(modelFile, ast)
Create an AssetDeclaration.

Throws:

- <code>IllegalModelException</code>

Param	Type	Description
modelFile	<code>ModelFile</code>	the ModelFile for this class
ast	<code>Object</code>	The AST created by the parser

<a name="module_concerto-core.AssetDeclaration+isRelationshipTarget"></a>

#### assetDeclaration.isRelationshipTarget() ⇒ <code>boolean</code>
Returns true if this class can be pointed to by a relationship

Kind: instance method of [<code>AssetDeclaration</code>](#module_concerto-
core.AssetDeclaration)
Returns: <code>boolean</code> - true if the class may be pointed to by a
relationship
<a name="module_concerto-core.AssetDeclaration+getSystemType"></a>

#### assetDeclaration.getSystemType() ⇒ <code>string</code>
Returns the base system type for Assets from the system namespace

Kind: instance method of [<code>AssetDeclaration</code>](#module_concerto-
core.AssetDeclaration)
Returns: <code>string</code> - the short name of the base system type
<a name="module_concerto-core.AssetDeclaration.Symbol.hasInstance"></a>

#### AssetDeclaration.Symbol.hasInstance(object) ⇒ <code>boolean</code>
Alternative instanceof that is reliable across different module instances

Kind: static method of [<code>AssetDeclaration</code>](#module_concerto-
core.AssetDeclaration)
Returns: <code>boolean</code> - - True, if the object is an instance of a
AssetDeclaration
See: https://github.com/hyperledger/composer-concerto/issues/47

Param	Type	Description
object	<code>object</code>	The object to test against

<a name="module_concerto-core.ClassDeclaration"></a>

### *concerto-core~ClassDeclaration*
ClassDeclaration defines the structure (model/schema) of composite data.
It is composed of a set of Properties, may have an identifying field, and may
have a super-type.
A ClassDeclaration is conceptually owned by a ModelFile which
defines all the classes that are part of a namespace.

Kind: inner abstract class of [<code>concerto-core</code>](#module_concerto-
core)

```
* *[~ClassDeclaration](#module_concerto-core.ClassDeclaration)*
    * *[new ClassDeclaration(modelFile, ast)](#new_module_concerto-
core.ClassDeclaration_new)*
    * _instance_
        * *[._resolveSuperType()](#module_concerto-
core.ClassDeclaration+_resolveSuperType) ⇒ <code>ClassDeclaration</code>*
        * *[.getSystemType()](#module_concerto-core.ClassDeclaration+getSystemType)
⇒ <code>string</code>*
        * *[.isAbstract()](#module_concerto-core.ClassDeclaration+isAbstract) ⇒
<code>boolean</code>*
        * *[.isEnum()](#module_concerto-core.ClassDeclaration+isEnum) ⇒
<code>boolean</code>*
        * *[.isConcept()](#module_concerto-core.ClassDeclaration+isConcept) ⇒
<code>boolean</code>*
        * *[.isEvent()](#module_concerto-core.ClassDeclaration+isEvent) ⇒
<code>boolean</code>*
        * *[.isRelationshipTarget()](#module_concerto-
core.ClassDeclaration+isRelationshipTarget) ⇒ <code>boolean</code>*
        * *[.isSystemRelationshipTarget()](#module_concerto-
core.ClassDeclaration+isSystemRelationshipTarget) ⇒ <code>boolean</code>*
        * *[.isSystemType()](#module_concerto-core.ClassDeclaration+isSystemType) ⇒
<code>boolean</code>*
        * *[.isSystemCoreType()](#module_concerto-
core.ClassDeclaration+isSystemCoreType) ⇒ <code>boolean</code>*
        * *[.getName()](#module_concerto-core.ClassDeclaration+getName) ⇒
<code>string</code>*
        * *[.getNamespace()](#module_concerto-core.ClassDeclaration+getNamespace) ⇒
<code>String</code>*
        * *[.getFullyQualifiedName()](#module_concerto-
core.ClassDeclaration+getFullyQualifiedName) ⇒ <code>string</code>*
        * *[.getIdentifierFieldName()](#module_concerto-
core.ClassDeclaration+getIdentifierFieldName) ⇒ <code>string</code>*
        * *[.getOwnProperty(name)](#module_concerto-
core.ClassDeclaration+getOwnProperty) ⇒ <code>Property</code>*
        * *[.getOwnProperties()](#module_concerto-
core.ClassDeclaration+getOwnProperties) ⇒ <code>Array.&lt;Property&gt;</code>*
        * *[.getSuperType()](#module_concerto-core.ClassDeclaration+getSuperType) ⇒
<code>string</code>*
        * *[.getSuperTypeDeclaration()](#module_concerto-
core.ClassDeclaration+getSuperTypeDeclaration) ⇒ <code>ClassDeclaration</code>*
        * *[.getAssignableClassDeclarations()](#module_concerto-
core.ClassDeclaration+getAssignableClassDeclarations) ⇒
<code>Array.&lt;ClassDeclaration&gt;</code>*
        * *[.getAllSuperTypeDeclarations()](#module_concerto-
core.ClassDeclaration+getAllSuperTypeDeclarations) ⇒
<code>Array.&lt;ClassDeclaration&gt;</code>*
        * *[.getProperty(name)](#module_concerto-core.ClassDeclaration+getProperty)
⇒ <code>Property</code>*
        * *[.getProperties()](#module_concerto-core.ClassDeclaration+getProperties)
⇒ <code>Array.&lt;Property&gt;</code>*
        * *[.getNestedProperty(propertyPath)](#module_concerto-
core.ClassDeclaration+getNestedProperty) ⇒ <code>Property</code>*
        * *[.toString()](#module_concerto-core.ClassDeclaration+toString) ⇒
<code>String</code>*
    * _static_
        * *[.Symbol.hasInstance(object)](#module_concerto-
core.ClassDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>*

<a name="new_module_concerto-core.ClassDeclaration_new"></a>
```

#### *new ClassDeclaration(modelFile, ast)*
Create a ClassDeclaration from an Abstract Syntax Tree. The AST is the
result of parsing.

Throws:

- <code>IllegalModelException</code>

Param	Type	Description
modelFile	<code>ModelFile</code>	the ModelFile for this class
ast	<code>string</code>	the AST created by the parser

<a name="module_concerto-core.ClassDeclaration+_resolveSuperType"></a>

#### *classDeclaration._resolveSuperType() ⇒ <code>ClassDeclaration</code>*
Resolve the super type on this class and store it as an internal property.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>ClassDeclaration</code> - The super type, or null if non specified.
<a name="module_concerto-core.ClassDeclaration+getSystemType"></a>

#### *classDeclaration.getSystemType() ⇒ <code>string</code>*
Returns the base system type for this type of class declaration. Override
this method in derived classes to specify a base system type.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>string</code> - the short name of the base system type or null
<a name="module_concerto-core.ClassDeclaration+isAbstract"></a>

#### *classDeclaration.isAbstract() ⇒ <code>boolean</code>*
Returns true if this class is declared as abstract in the model file

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>boolean</code> - true if the class is abstract
<a name="module_concerto-core.ClassDeclaration+isEnum"></a>

#### *classDeclaration.isEnum() ⇒ <code>boolean</code>*
Returns true if this class is an enumeration.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>boolean</code> - true if the class is an enumerated type
<a name="module_concerto-core.ClassDeclaration+isConcept"></a>

#### *classDeclaration.isConcept() ⇒ <code>boolean</code>*
Returns true if this class is the definition of a concept.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>boolean</code> - true if the class is a concept
<a name="module_concerto-core.ClassDeclaration+isEvent"></a>

#### *classDeclaration.isEvent() ⇒ <code>boolean</code>*
Returns true if this class is the definition of an event.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>boolean</code> - true if the class is an event
<a name="module_concerto-core.ClassDeclaration+isRelationshipTarget"></a>

#### *classDeclaration.isRelationshipTarget() ⇒ <code>boolean</code>*
Returns true if this class can be pointed to by a relationship

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>boolean</code> - true if the class may be pointed to by a relationship
<a name="module_concerto-core.ClassDeclaration+isSystemRelationshipTarget"></a>

#### *classDeclaration.isSystemRelationshipTarget() ⇒ <code>boolean</code>*
Returns true if this class can be pointed to by a relationship in a
system model

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>boolean</code> - true if the class may be pointed to by a relationship
<a name="module_concerto-core.ClassDeclaration+isSystemType"></a>

#### *classDeclaration.isSystemType() ⇒ <code>boolean</code>*
Returns true is this type is in the system namespace

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>boolean</code> - true if the class may be pointed to by a relationship
<a name="module_concerto-core.ClassDeclaration+isSystemCoreType"></a>

#### *classDeclaration.isSystemCoreType() ⇒ <code>boolean</code>*
Returns true if this class is a system core type - both in the system
namespace, and also one of the system core types (Asset, Participant, etc).

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>boolean</code> - true if the class may be pointed to by a relationship
<a name="module_concerto-core.ClassDeclaration+getName"></a>

#### *classDeclaration.getName() ⇒ <code>string</code>*
Returns the short name of a class. This name does not include the
namespace from the owning ModelFile.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>string</code> - the short name of this class
<a name="module_concerto-core.ClassDeclaration+getNamespace"></a>

#### *classDeclaration.getNamespace() ⇒ <code>String</code>*
Return the namespace of this class.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-

core.ClassDeclaration)
Returns: <code>String</code> - namespace - a namespace.
<a name="module_concerto-core.ClassDeclaration+getFullyQualifiedName"></a>

#### *classDeclaration.getFullyQualifiedName() ⇒ <code>string</code>*
Returns the fully qualified name of this class.
The name will include the namespace if present.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-
core.ClassDeclaration)
Returns: <code>string</code> - the fully-qualified name of this class
<a name="module_concerto-core.ClassDeclaration+getIdentifierFieldName"></a>

#### *classDeclaration.getIdentifierFieldName() ⇒ <code>string</code>*
Returns the name of the identifying field for this class. Note
that the identifying field may come from a super type.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-
core.ClassDeclaration)
Returns: <code>string</code> - the name of the id field for this class
<a name="module_concerto-core.ClassDeclaration+getOwnProperty"></a>

#### *classDeclaration.getOwnProperty(name) ⇒ <code>Property</code>*
Returns the field with a given name or null if it does not exist.
The field must be directly owned by this class -- the super-type is
not introspected.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-
core.ClassDeclaration)
Returns: <code>Property</code> - the field definition or null if it does not
exist.

Param	Type	Description
name	<code>string</code>	the name of the field

<a name="module_concerto-core.ClassDeclaration+getOwnProperties"></a>

#### *classDeclaration.getOwnProperties() ⇒ <code>Array.&lt;Property&gt;</code>*
Returns the fields directly defined by this class.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-
core.ClassDeclaration)
Returns: <code>Array.&lt;Property&gt;</code> - the array of fields
<a name="module_concerto-core.ClassDeclaration+getSuperType"></a>

#### *classDeclaration.getSuperType() ⇒ <code>string</code>*
Returns the FQN of the super type for this class or null if this
class does not have a super type.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-
core.ClassDeclaration)
Returns: <code>string</code> - the FQN name of the super type or null
<a name="module_concerto-core.ClassDeclaration+getSuperTypeDeclaration"></a>

#### *classDeclaration.getSuperTypeDeclaration() ⇒ <code>ClassDeclaration</code>*
Get the super type class declaration for this class.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-

core.ClassDeclaration)
Returns: <code>ClassDeclaration</code> - the super type declaration, or null if
there is no super type.
<a name="module_concerto-core.ClassDeclaration+getAssignableClassDeclarations"></a>

#### *classDeclaration.getAssignableClassDeclarations() ⇒
<code>Array.&lt;ClassDeclaration&gt;</code>*
Get the class declarations for all subclasses of this class, including this class.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-
core.ClassDeclaration)
Returns: <code>Array.&lt;ClassDeclaration&gt;</code> - subclass declarations.
<a name="module_concerto-core.ClassDeclaration+getAllSuperTypeDeclarations"></a>

#### *classDeclaration.getAllSuperTypeDeclarations() ⇒
<code>Array.&lt;ClassDeclaration&gt;</code>*
Get all the super-type declarations for this type.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-
core.ClassDeclaration)
Returns: <code>Array.&lt;ClassDeclaration&gt;</code> - super-type declarations.
<a name="module_concerto-core.ClassDeclaration+getProperty"></a>

#### *classDeclaration.getProperty(name) ⇒ <code>Property</code>*
Returns the property with a given name or null if it does not exist.
Fields defined in super-types are also introspected.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-
core.ClassDeclaration)
Returns: <code>Property</code> - the field, or null if it does not exist

Param	Type	Description
name	<code>string</code>	the name of the field

<a name="module_concerto-core.ClassDeclaration+getProperties"></a>

#### *classDeclaration.getProperties() ⇒ <code>Array.&lt;Property&gt;</code>*
Returns the properties defined in this class and all super classes.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-
core.ClassDeclaration)
Returns: <code>Array.&lt;Property&gt;</code> - the array of fields
<a name="module_concerto-core.ClassDeclaration+getNestedProperty"></a>

#### *classDeclaration.getNestedProperty(propertyPath) ⇒ <code>Property</code>*
Get a nested property using a dotted property path

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-
core.ClassDeclaration)
Returns: <code>Property</code> - the property
Throws:

- <code>IllegalModelException</code> if the property path is invalid or the
property does not exist

Param	Type	Description

```
| propertyPath | <code>string</code> | The property name or name with nested
structure e.g a.b.c |

<a name="module_concerto-core.ClassDeclaration+toString"></a>

#### *classDeclaration.toString() ⇒ <code>String</code>*
Returns the string representation of this class

**Kind**: instance method of [<code>ClassDeclaration</code>](#module_concerto-
core.ClassDeclaration)
**Returns**: <code>String</code> - the string representation of the class
<a name="module_concerto-core.ClassDeclaration.Symbol.hasInstance"></a>

#### *ClassDeclaration.Symbol.hasInstance(object) ⇒ <code>boolean</code>*
Alternative instanceof that is reliable across different module instances

**Kind**: static method of [<code>ClassDeclaration</code>](#module_concerto-
core.ClassDeclaration)
**Returns**: <code>boolean</code> - - True, if the object is an instance of a Class
Declaration
**See**: https://github.com/hyperledger/composer-concerto/issues/47

| Param | Type | Description |
| --- | --- | --- |
| object | <code>object</code> | The object to test against |

<a name="module_concerto-core.ConceptDeclaration"></a>

### concerto-core~ConceptDeclaration ⇐ <code>ClassDeclaration</code>
ConceptDeclaration defines the schema (aka model or class) for
an Concept. It extends ClassDeclaration which manages a set of
fields, a super-type and the specification of an
identifying field.

**Kind**: inner class of [<code>concerto-core</code>](#module_concerto-core)
**Extends**: <code>ClassDeclaration</code>
**See**: [ClassDeclaration](ClassDeclaration)

* [~ConceptDeclaration](#module_concerto-core.ConceptDeclaration) ⇐
<code>ClassDeclaration</code>
    * [new ConceptDeclaration(modelFile, ast)](#new_module_concerto-
core.ConceptDeclaration_new)
    * _instance_
        * [.isConcept()](#module_concerto-core.ConceptDeclaration+isConcept) ⇒
<code>boolean</code>
    * _static_
        * [.Symbol.hasInstance(object)](#module_concerto-
core.ConceptDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>

<a name="new_module_concerto-core.ConceptDeclaration_new"></a>

#### new ConceptDeclaration(modelFile, ast)
Create an AssetDeclaration.

**Throws**:

- <code>IllegalModelException</code>
```

Param	Type	Description
modelFile	<code>ModelFile</code>	the ModelFile for this class
ast	<code>Object</code>	The AST created by the parser

<a name="module_concerto-core.ConceptDeclaration+isConcept"></a>

#### conceptDeclaration.isConcept() ⇒ <code>boolean</code>
Returns true if this class is the definition of a concept.

Kind: instance method of [<code>ConceptDeclaration</code>](#module_concerto-core.ConceptDeclaration)
Returns: <code>boolean</code> - true if the class is a concept
<a name="module_concerto-core.ConceptDeclaration.Symbol.hasInstance"></a>

#### ConceptDeclaration.Symbol.hasInstance(object) ⇒ <code>boolean</code>
Alternative instanceof that is reliable across different module instances

Kind: static method of [<code>ConceptDeclaration</code>](#module_concerto-core.ConceptDeclaration)
Returns: <code>boolean</code> - - True, if the object is an instance of a ConceptDeclaration
See: https://github.com/hyperledger/composer-concerto/issues/47

Param	Type	Description
object	<code>object</code>	The object to test against

<a name="module_concerto-core.Decorator"></a>

### concerto-core~Decorator
Decorator encapsulates a decorator (annotation) on a class or property.

Kind: inner class of [<code>concerto-core</code>](#module_concerto-core)

* [~Decorator](#module_concerto-core.Decorator)
    * [new Decorator(parent, ast)](#new_module_concerto-core.Decorator_new)
    * [.getParent()](#module_concerto-core.Decorator+getParent) ⇒ <code>ClassDeclaration</code> \| <code>Property</code>
    * [.getName()](#module_concerto-core.Decorator+getName) ⇒ <code>string</code>
    * [.getArguments()](#module_concerto-core.Decorator+getArguments) ⇒ <code>Array.&lt;object&gt;</code>

<a name="new_module_concerto-core.Decorator_new"></a>

#### new Decorator(parent, ast)
Create a Decorator.

Throws:

- <code>IllegalModelException</code>

Param	Type	Description
parent	<code>ClassDeclaration</code> | <code>Property</code>	the owner of this property
ast	<code>Object</code>	The AST created by the parser

```
<a name="module_concerto-core.Decorator+getParent"></a>

#### decorator.getParent() ⇒ <code>ClassDeclaration</code> \| <code>Property</code>
Returns the owner of this property

**Kind**: instance method of [<code>Decorator</code>](#module_concerto-
core.Decorator)
**Returns**: <code>ClassDeclaration</code> \| <code>Property</code> - the parent
class or property declaration
<a name="module_concerto-core.Decorator+getName"></a>

#### decorator.getName() ⇒ <code>string</code>
Returns the name of a decorator

**Kind**: instance method of [<code>Decorator</code>](#module_concerto-
core.Decorator)
**Returns**: <code>string</code> - the name of this decorator
<a name="module_concerto-core.Decorator+getArguments"></a>

#### decorator.getArguments() ⇒ <code>Array.&lt;object&gt;</code>
Returns the arguments for this decorator

**Kind**: instance method of [<code>Decorator</code>](#module_concerto-
core.Decorator)
**Returns**: <code>Array.&lt;object&gt;</code> - the arguments for this decorator
or null if it does not have any arguments
<a name="module_concerto-core.EnumDeclaration"></a>

### concerto-core~EnumDeclaration ⇐ <code>ClassDeclaration</code>
EnumDeclaration defines an enumeration of static values.

**Kind**: inner class of [<code>concerto-core</code>](#module_concerto-core)
**Extends**: <code>ClassDeclaration</code>
**See**: See [ClassDeclaration](ClassDeclaration)

* [~EnumDeclaration](#module_concerto-core.EnumDeclaration) ⇐
<code>ClassDeclaration</code>
    * [new EnumDeclaration(modelFile, ast)](#new_module_concerto-
core.EnumDeclaration_new)
    * _instance_
        * [.isEnum()](#module_concerto-core.EnumDeclaration+isEnum) ⇒
<code>boolean</code>
        * [.toString()](#module_concerto-core.EnumDeclaration+toString) ⇒
<code>String</code>
    * _static_
        * [.Symbol.hasInstance(object)](#module_concerto-
core.EnumDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>

<a name="new_module_concerto-core.EnumDeclaration_new"></a>

#### new EnumDeclaration(modelFile, ast)
Create an AssetDeclaration.

**Throws**:

- <code>IllegalModelException</code>


| Param | Type | Description |
```

| --- | --- | --- |
| modelFile | <code>ModelFile</code> | the ModelFile for this class |
| ast | <code>Object</code> | The AST created by the parser |

<a name="module_concerto-core.EnumDeclaration+isEnum"></a>

#### enumDeclaration.isEnum() ⇒ <code>boolean</code>
Returns true if this class is an enumeration.

Kind: instance method of [<code>EnumDeclaration</code>](#module_concerto-core.EnumDeclaration)
Returns: <code>boolean</code> - true if the class is an enumerated type
<a name="module_concerto-core.EnumDeclaration+toString"></a>

#### enumDeclaration.toString() ⇒ <code>String</code>
Returns the string representation of this class

Kind: instance method of [<code>EnumDeclaration</code>](#module_concerto-core.EnumDeclaration)
Returns: <code>String</code> - the string representation of the class
<a name="module_concerto-core.EnumDeclaration.Symbol.hasInstance"></a>

#### EnumDeclaration.Symbol.hasInstance(object) ⇒ <code>boolean</code>
Alternative instanceof that is reliable across different module instances

Kind: static method of [<code>EnumDeclaration</code>](#module_concerto-core.EnumDeclaration)
Returns: <code>boolean</code> - - True, if the object is an instance of a Class Declaration
See: https://github.com/hyperledger/composer-concerto/issues/47

Param	Type	Description
object	<code>object</code>	The object to test against

<a name="module_concerto-core.EnumValueDeclaration"></a>

### concerto-core~EnumValueDeclaration ⇐ <code>Property</code>
Class representing a value from a set of enumerated values

Kind: inner class of [<code>concerto-core</code>](#module_concerto-core)
Extends: <code>Property</code>
See: See Property

* [~EnumValueDeclaration](#module_concerto-core.EnumValueDeclaration) ⇐ <code>Property</code>
    * [new EnumValueDeclaration(parent, ast)](#new_module_concerto-core.EnumValueDeclaration_new)
    * [.Symbol.hasInstance(object)](#module_concerto-core.EnumValueDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>

<a name="new_module_concerto-core.EnumValueDeclaration_new"></a>

#### new EnumValueDeclaration(parent, ast)
Create a EnumValueDeclaration.

Throws:

- <code>IllegalModelException</code>

```
| Param | Type | Description |
| --- | --- | --- |
| parent | <code>ClassDeclaration</code> | The owner of this property |
| ast | <code>Object</code> | The AST created by the parser |

<a name="module_concerto-core.EnumValueDeclaration.Symbol.hasInstance"></a>

#### EnumValueDeclaration.Symbol.hasInstance(object) ⇒ <code>boolean</code>
Alternative instanceof that is reliable across different module instances

**Kind**: static method of [<code>EnumValueDeclaration</code>](#module_concerto-
core.EnumValueDeclaration)
**Returns**: <code>boolean</code> - - True, if the object is an instance of a
EnumValueDeclaration
**See**: https://github.com/hyperledger/composer-concerto/issues/47

| Param | Type | Description |
| --- | --- | --- |
| object | <code>object</code> | The object to test against |

<a name="module_concerto-core.EventDeclaration"></a>

### concerto-core~EventDeclaration ⇐ <code>ClassDeclaration</code>
Class representing the definition of an Event.

**Kind**: inner class of [<code>concerto-core</code>](#module_concerto-core)
**Extends**: <code>ClassDeclaration</code>
**See**: See  [ClassDeclaration](ClassDeclaration)

* [~EventDeclaration](#module_concerto-core.EventDeclaration) ⇐
<code>ClassDeclaration</code>
    * [new EventDeclaration(modelFile, ast)](#new_module_concerto-
core.EventDeclaration_new)
    * _instance_
        * [.getSystemType()](#module_concerto-core.EventDeclaration+getSystemType)
⇒ <code>string</code>
        * [.isEvent()](#module_concerto-core.EventDeclaration+isEvent) ⇒
<code>boolean</code>
    * _static_
        * [.Symbol.hasInstance(object)](#module_concerto-
core.EventDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>

<a name="new_module_concerto-core.EventDeclaration_new"></a>

#### new EventDeclaration(modelFile, ast)
Create an EventDeclaration.

**Throws**:

- <code>IllegalModelException</code>


| Param | Type | Description |
| --- | --- | --- |
| modelFile | <code>ModelFile</code> | the ModelFile for this class |
| ast | <code>Object</code> | The AST created by the parser |
```

<a name="module_concerto-core.EventDeclaration+getSystemType"></a>

#### eventDeclaration.getSystemType() ⇒ <code>string</code>
Returns the base system type for Events from the system namespace

Kind: instance method of [<code>EventDeclaration</code>](#module_concerto-core.EventDeclaration)
Returns: <code>string</code> - the short name of the base system type
<a name="module_concerto-core.EventDeclaration+isEvent"></a>

#### eventDeclaration.isEvent() ⇒ <code>boolean</code>
Returns true if this class is the definition of an event

Kind: instance method of [<code>EventDeclaration</code>](#module_concerto-core.EventDeclaration)
Returns: <code>boolean</code> - true if the class is an event
<a name="module_concerto-core.EventDeclaration.Symbol.hasInstance"></a>

#### EventDeclaration.Symbol.hasInstance(object) ⇒ <code>boolean</code>
Alternative instanceof that is reliable across different module instances

Kind: static method of [<code>EventDeclaration</code>](#module_concerto-core.EventDeclaration)
Returns: <code>boolean</code> - - True, if the object is an instance of a EventDeclaration
See: https://github.com/hyperledger/composer-concerto/issues/47

Param	Type	Description
object	<code>object</code>	The object to test against

<a name="module_concerto-core.Introspector"></a>

### concerto-core~Introspector
Provides access to the structure of transactions, assets and participants.

Kind: inner class of [<code>concerto-core</code>](#module_concerto-core)

* [~Introspector](#module_concerto-core.Introspector)
    * [new Introspector(modelManager)](#new_module_concerto-core.Introspector_new)
    * [.getClassDeclarations()](#module_concerto-core.Introspector+getClassDeclarations) ⇒ <code>Array.&lt;ClassDeclaration&gt;</code>
    * [.getClassDeclaration(fullyQualifiedTypeName)](#module_concerto-core.Introspector+getClassDeclaration) ⇒ <code>ClassDeclaration</code>

<a name="new_module_concerto-core.Introspector_new"></a>

#### new Introspector(modelManager)
Create the Introspector.

Param	Type	Description
modelManager	<code>ModelManager</code>	the ModelManager that backs this Introspector

<a name="module_concerto-core.Introspector+getClassDeclarations"></a>

```
#### introspector.getClassDeclarations() ⇒
<code>Array.&lt;ClassDeclaration&gt;</code>
Returns all the class declarations for the business network.

**Kind**: instance method of [<code>Introspector</code>](#module_concerto-
core.Introspector)
**Returns**: <code>Array.&lt;ClassDeclaration&gt;</code> - the array of class
declarations
<a name="module_concerto-core.Introspector+getClassDeclaration"></a>

#### introspector.getClassDeclaration(fullyQualifiedTypeName) ⇒
<code>ClassDeclaration</code>
Returns the class declaration with the given fully qualified name.
Throws an error if the class declaration does not exist.

**Kind**: instance method of [<code>Introspector</code>](#module_concerto-
core.Introspector)
**Returns**: <code>ClassDeclaration</code> - the class declaration
**Throws**:

- <code>Error</code> if the class declaration does not exist


| Param | Type | Description |
| --- | --- | --- |
| fullyQualifiedTypeName | <code>String</code> | the fully qualified name of the
type |

<a name="module_concerto-core.ModelFile"></a>

### concerto-core~ModelFile
Class representing a Model File. A Model File contains a single namespace
and a set of model elements: assets, transactions etc.

**Kind**: inner class of [<code>concerto-core</code>](#module_concerto-core)

* [~ModelFile](#module_concerto-core.ModelFile)
    * [new ModelFile(modelManager, definitions, [fileName], [isSystemModelFile])]
(#new_module_concerto-core.ModelFile_new)
    * _instance_
        * [.isExternal()](#module_concerto-core.ModelFile+isExternal) ⇒
<code>boolean</code>
        * [.getModelManager()](#module_concerto-core.ModelFile+getModelManager) ⇒
<code>ModelManager</code>
        * [.getImports()](#module_concerto-core.ModelFile+getImports) ⇒
<code>Array.&lt;string&gt;</code>
        * [.isDefined(type)](#module_concerto-core.ModelFile+isDefined) ⇒
<code>boolean</code>
        * [.getLocalType(type)](#module_concerto-core.ModelFile+getLocalType) ⇒
<code>ClassDeclaration</code>
        * [.getAssetDeclaration(name)](#module_concerto-
core.ModelFile+getAssetDeclaration) ⇒ <code>AssetDeclaration</code>
        * [.getTransactionDeclaration(name)](#module_concerto-
core.ModelFile+getTransactionDeclaration) ⇒ <code>TransactionDeclaration</code>
        * [.getEventDeclaration(name)](#module_concerto-
core.ModelFile+getEventDeclaration) ⇒ <code>EventDeclaration</code>
        * [.getParticipantDeclaration(name)](#module_concerto-
core.ModelFile+getParticipantDeclaration) ⇒ <code>ParticipantDeclaration</code>
        * [.getNamespace()](#module_concerto-core.ModelFile+getNamespace) ⇒
```

```
<code>string</code>
        * [.getName()](#module_concerto-core.ModelFile+getName) ⇒
<code>string</code>
        * [.getAssetDeclarations(includeSystemType)](#module_concerto-
core.ModelFile+getAssetDeclarations) ⇒ <code>Array.&lt;AssetDeclaration&gt;</code>
        * [.getTransactionDeclarations(includeSystemType)](#module_concerto-
core.ModelFile+getTransactionDeclarations) ⇒
<code>Array.&lt;TransactionDeclaration&gt;</code>
        * [.getEventDeclarations(includeSystemType)](#module_concerto-
core.ModelFile+getEventDeclarations) ⇒ <code>Array.&lt;EventDeclaration&gt;</code>
        * [.getParticipantDeclarations(includeSystemType)](#module_concerto-
core.ModelFile+getParticipantDeclarations) ⇒
<code>Array.&lt;ParticipantDeclaration&gt;</code>
        * [.getConceptDeclarations(includeSystemType)](#module_concerto-
core.ModelFile+getConceptDeclarations) ⇒
<code>Array.&lt;ConceptDeclaration&gt;</code>
        * [.getEnumDeclarations(includeSystemType)](#module_concerto-
core.ModelFile+getEnumDeclarations) ⇒ <code>Array.&lt;EnumDeclaration&gt;</code>
        * [.getDeclarations(type, includeSystemType)](#module_concerto-
core.ModelFile+getDeclarations) ⇒ <code>Array.&lt;ClassDeclaration&gt;</code>
        * [.getAllDeclarations()](#module_concerto-
core.ModelFile+getAllDeclarations) ⇒ <code>Array.&lt;ClassDeclaration&gt;</code>
        * [.getDefinitions()](#module_concerto-core.ModelFile+getDefinitions) ⇒
<code>string</code>
        * [.isSystemModelFile()](#module_concerto-core.ModelFile+isSystemModelFile)
⇒ <code>boolean</code>
    * _static_
        * [.Symbol.hasInstance(object)](#module_concerto-
core.ModelFile.Symbol.hasInstance) ⇒ <code>boolean</code>

<a name="new_module_concerto-core.ModelFile_new"></a>

#### new ModelFile(modelManager, definitions, [fileName], [isSystemModelFile])
Create a ModelFile. This should only be called by framework code.
Use the ModelManager to manage ModelFiles.

**Throws**:

- <code>IllegalModelException</code>


| Param | Type | Default | Description |
| --- | --- | --- | --- |
| modelManager | <code>ModelManager</code> |  | the ModelManager that manages this
ModelFile |
| definitions | <code>string</code> |  | The DSL model as a string. |
| [fileName] | <code>string</code> |  | The optional filename for this modelfile |
| [isSystemModelFile] | <code>boolean</code> | <code>false</code> | If true, this
is a system model file, defaults to false |

<a name="module_concerto-core.ModelFile+isExternal"></a>

#### modelFile.isExternal() ⇒ <code>boolean</code>
Returns true if this ModelFile was downloaded from an external URI.

**Kind**: instance method of [<code>ModelFile</code>](#module_concerto-
core.ModelFile)
**Returns**: <code>boolean</code> - true iff this ModelFile was downloaded from an
external URI
```

```
<a name="module_concerto-core.ModelFile+getModelManager"></a>

#### modelFile.getModelManager() ⇒ <code>ModelManager</code>
Returns the ModelManager associated with this ModelFile

**Kind**: instance method of [<code>ModelFile</code>](#module_concerto-
core.ModelFile)
**Returns**: <code>ModelManager</code> - The ModelManager for this ModelFile
<a name="module_concerto-core.ModelFile+getImports"></a>

#### modelFile.getImports() ⇒ <code>Array.&lt;string&gt;</code>
Returns the types that have been imported into this ModelFile.

**Kind**: instance method of [<code>ModelFile</code>](#module_concerto-
core.ModelFile)
**Returns**: <code>Array.&lt;string&gt;</code> - The array of imports for this
ModelFile
<a name="module_concerto-core.ModelFile+isDefined"></a>

#### modelFile.isDefined(type) ⇒ <code>boolean</code>
Returns true if the type is defined in the model file

**Kind**: instance method of [<code>ModelFile</code>](#module_concerto-
core.ModelFile)
**Returns**: <code>boolean</code> - true if the type (asset or transaction) is
defined

| Param | Type | Description |
| --- | --- | --- |
| type | <code>string</code> | the name of the type |

<a name="module_concerto-core.ModelFile+getLocalType"></a>

#### modelFile.getLocalType(type) ⇒ <code>ClassDeclaration</code>
Returns the type with the specified name or null

**Kind**: instance method of [<code>ModelFile</code>](#module_concerto-
core.ModelFile)
**Returns**: <code>ClassDeclaration</code> - the ClassDeclaration, or null if the
type does not exist

| Param | Type | Description |
| --- | --- | --- |
| type | <code>string</code> | the short OR FQN name of the type |

<a name="module_concerto-core.ModelFile+getAssetDeclaration"></a>

#### modelFile.getAssetDeclaration(name) ⇒ <code>AssetDeclaration</code>
Get the AssetDeclarations defined in this ModelFile or null

**Kind**: instance method of [<code>ModelFile</code>](#module_concerto-
core.ModelFile)
**Returns**: <code>AssetDeclaration</code> - the AssetDeclaration with the given
short name

| Param | Type | Description |
| --- | --- | --- |
| name | <code>string</code> | the name of the type |
```

<a name="module_concerto-core.ModelFile+getTransactionDeclaration"></a>

#### modelFile.getTransactionDeclaration(name) ⇒ <code>TransactionDeclaration</code>
Get the TransactionDeclaration defined in this ModelFile or null

Kind: instance method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)
Returns: <code>TransactionDeclaration</code> - the TransactionDeclaration with the given short name

Param	Type	Description
name	<code>string</code>	the name of the type

<a name="module_concerto-core.ModelFile+getEventDeclaration"></a>

#### modelFile.getEventDeclaration(name) ⇒ <code>EventDeclaration</code>
Get the EventDeclaration defined in this ModelFile or null

Kind: instance method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)
Returns: <code>EventDeclaration</code> - the EventDeclaration with the given short name

Param	Type	Description
name	<code>string</code>	the name of the type

<a name="module_concerto-core.ModelFile+getParticipantDeclaration"></a>

#### modelFile.getParticipantDeclaration(name) ⇒ <code>ParticipantDeclaration</code>
Get the ParticipantDeclaration defined in this ModelFile or null

Kind: instance method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)
Returns: <code>ParticipantDeclaration</code> - the ParticipantDeclaration with the given short name

Param	Type	Description
name	<code>string</code>	the name of the type

<a name="module_concerto-core.ModelFile+getNamespace"></a>

#### modelFile.getNamespace() ⇒ <code>string</code>
Get the Namespace for this model file.

Kind: instance method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)
Returns: <code>string</code> - The Namespace for this model file
<a name="module_concerto-core.ModelFile+getName"></a>

#### modelFile.getName() ⇒ <code>string</code>
Get the filename for this model file. Note that this may be null.

Kind: instance method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)

Returns: <code>string</code> - The filename for this model file
<a name="module_concerto-core.ModelFile+getAssetDeclarations"></a>

#### modelFile.getAssetDeclarations(includeSystemType) ⇒ <code>Array.&lt;AssetDeclaration&gt;</code>
Get the AssetDeclarations defined in this ModelFile

Kind: instance method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)
Returns: <code>Array.&lt;AssetDeclaration&gt;</code> - the AssetDeclarations defined in the model file

Param	Type	Default	Description
includeSystemType	<code>Boolean</code>	<code>true</code>	Include the decalarations of system type in returned data

<a name="module_concerto-core.ModelFile+getTransactionDeclarations"></a>

#### modelFile.getTransactionDeclarations(includeSystemType) ⇒ <code>Array.&lt;TransactionDeclaration&gt;</code>
Get the TransactionDeclarations defined in this ModelFile

Kind: instance method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)
Returns: <code>Array.&lt;TransactionDeclaration&gt;</code> - the TransactionDeclarations defined in the model file

Param	Type	Default	Description
includeSystemType	<code>Boolean</code>	<code>true</code>	Include the decalarations of system type in returned data

<a name="module_concerto-core.ModelFile+getEventDeclarations"></a>

#### modelFile.getEventDeclarations(includeSystemType) ⇒ <code>Array.&lt;EventDeclaration&gt;</code>
Get the EventDeclarations defined in this ModelFile

Kind: instance method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)
Returns: <code>Array.&lt;EventDeclaration&gt;</code> - the EventDeclarations defined in the model file

Param	Type	Default	Description
includeSystemType	<code>Boolean</code>	<code>true</code>	Include the decalarations of system type in returned data

<a name="module_concerto-core.ModelFile+getParticipantDeclarations"></a>

#### modelFile.getParticipantDeclarations(includeSystemType) ⇒ <code>Array.&lt;ParticipantDeclaration&gt;</code>
Get the ParticipantDeclarations defined in this ModelFile

Kind: instance method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)
Returns: <code>Array.&lt;ParticipantDeclaration&gt;</code> - the ParticipantDeclaration defined in the model file

Param	Type	Default	Description
includeSystemType	<code>Boolean</code>	<code>true</code>	Include the decalarations of system type in returned data

<a name="module_concerto-core.ModelFile+getConceptDeclarations"></a>

#### modelFile.getConceptDeclarations(includeSystemType) ⇒ <code>Array.&lt;ConceptDeclaration&gt;</code>
Get the ConceptDeclarations defined in this ModelFile

Kind: instance method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)
Returns: <code>Array.&lt;ConceptDeclaration&gt;</code> - the ParticipantDeclaration defined in the model file

Param	Type	Default	Description
includeSystemType	<code>Boolean</code>	<code>true</code>	Include the decalarations of system type in returned data

<a name="module_concerto-core.ModelFile+getEnumDeclarations"></a>

#### modelFile.getEnumDeclarations(includeSystemType) ⇒ <code>Array.&lt;EnumDeclaration&gt;</code>
Get the EnumDeclarations defined in this ModelFile

Kind: instance method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)
Returns: <code>Array.&lt;EnumDeclaration&gt;</code> - the EnumDeclaration defined in the model file

Param	Type	Default	Description
includeSystemType	<code>Boolean</code>	<code>true</code>	Include the decalarations of system type in returned data

<a name="module_concerto-core.ModelFile+getDeclarations"></a>

#### modelFile.getDeclarations(type, includeSystemType) ⇒ <code>Array.&lt;ClassDeclaration&gt;</code>
Get the instances of a given type in this ModelFile

Kind: instance method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)
Returns: <code>Array.&lt;ClassDeclaration&gt;</code> - the ClassDeclaration defined in the model file

Param	Type	Default	Description
type	<code>function</code>		the type of the declaration
includeSystemType	<code>Boolean</code>	<code>true</code>	Include the decalarations of system type in returned data

<a name="module_concerto-core.ModelFile+getAllDeclarations"></a>

#### modelFile.getAllDeclarations() ⇒ <code>Array.&lt;ClassDeclaration&gt;</code>
Get all declarations in this ModelFile

```
**Kind**: instance method of [<code>ModelFile</code>](#module_concerto-
core.ModelFile)
**Returns**: <code>Array.&lt;ClassDeclaration&gt;</code> - the ClassDeclarations
defined in the model file
<a name="module_concerto-core.ModelFile+getDefinitions"></a>

#### modelFile.getDefinitions() ⇒ <code>string</code>
Get the definitions for this model.

**Kind**: instance method of [<code>ModelFile</code>](#module_concerto-
core.ModelFile)
**Returns**: <code>string</code> - The definitions for this model.
<a name="module_concerto-core.ModelFile+isSystemModelFile"></a>

#### modelFile.isSystemModelFile() ⇒ <code>boolean</code>
Returns true if this ModelFile is a system model

**Kind**: instance method of [<code>ModelFile</code>](#module_concerto-
core.ModelFile)
**Returns**: <code>boolean</code> - true of this ModelFile is a system model
<a name="module_concerto-core.ModelFile.Symbol.hasInstance"></a>

#### ModelFile.Symbol.hasInstance(object) ⇒ <code>boolean</code>
Alternative to instanceof that is reliable across different module instances

**Kind**: static method of [<code>ModelFile</code>](#module_concerto-
core.ModelFile)
**Returns**: <code>boolean</code> - - True, if the object is an instance of a
ModelFile
**See**: https://github.com/hyperledger/composer-concerto/issues/47

| Param | Type | Description |
| --- | --- | --- |
| object | <code>object</code> | The object to test against |

<a name="module_concerto-core.ParticipantDeclaration"></a>

### concerto-core~ParticipantDeclaration ⇐ <code>ClassDeclaration</code>
Class representing the definition of a Participant.

**Kind**: inner class of [<code>concerto-core</code>](#module_concerto-core)
**Extends**: <code>ClassDeclaration</code>
**See**: See  [ClassDeclaration](ClassDeclaration)

* [~ParticipantDeclaration](#module_concerto-core.ParticipantDeclaration) ⇐
<code>ClassDeclaration</code>
    * [new ParticipantDeclaration(modelFile, ast)](#new_module_concerto-
core.ParticipantDeclaration_new)
    * _instance_
        * [.isRelationshipTarget()](#module_concerto-
core.ParticipantDeclaration+isRelationshipTarget) ⇒ <code>boolean</code>
        * [.getSystemType()](#module_concerto-
core.ParticipantDeclaration+getSystemType) ⇒ <code>string</code>
    * _static_
        * [.Symbol.hasInstance(object)](#module_concerto-
core.ParticipantDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>

<a name="new_module_concerto-core.ParticipantDeclaration_new"></a>
```

```
#### new ParticipantDeclaration(modelFile, ast)
Create an ParticipantDeclaration.

**Throws**:

- <code>IllegalModelException</code>


| Param | Type | Description |
| --- | --- | --- |
| modelFile | <code>ModelFile</code> | the ModelFile for this class |
| ast | <code>Object</code> | The AST created by the parser |

<a name="module_concerto-core.ParticipantDeclaration+isRelationshipTarget"></a>

#### participantDeclaration.isRelationshipTarget() ⇒ <code>boolean</code>
Returns true if this class can be pointed to by a relationship

**Kind**: instance method of [<code>ParticipantDeclaration</code>]
(#module_concerto-core.ParticipantDeclaration)
**Returns**: <code>boolean</code> - true if the class may be pointed to by a
relationship
<a name="module_concerto-core.ParticipantDeclaration+getSystemType"></a>

#### participantDeclaration.getSystemType() ⇒ <code>string</code>
Returns the base system type for Participants from the system namespace

**Kind**: instance method of [<code>ParticipantDeclaration</code>]
(#module_concerto-core.ParticipantDeclaration)
**Returns**: <code>string</code> - the short name of the base system type
<a name="module_concerto-core.ParticipantDeclaration.Symbol.hasInstance"></a>

#### ParticipantDeclaration.Symbol.hasInstance(object) ⇒ <code>boolean</code>
Alternative instanceof that is reliable across different module instances

**Kind**: static method of [<code>ParticipantDeclaration</code>](#module_concerto-
core.ParticipantDeclaration)
**Returns**: <code>boolean</code> - - True, if the object is an instance of a
ParticipantDeclaration
**See**: https://github.com/hyperledger/composer-concerto/issues/47

| Param | Type | Description |
| --- | --- | --- |
| object | <code>object</code> | The object to test against |

<a name="module_concerto-core.Property"></a>

### concerto-core~Property
Property representing an attribute of a class declaration,
either a Field or a Relationship.

**Kind**: inner class of [<code>concerto-core</code>](#module_concerto-core)

* [~Property](#module_concerto-core.Property)
    * [new Property(parent, ast)](#new_module_concerto-core.Property_new)
    * _instance_
        * [.getParent()](#module_concerto-core.Property+getParent) ⇒
<code>ClassDeclaration</code>
```

```
        * [.getName()](#module_concerto-core.Property+getName) ⇒
<code>string</code>
        * [.getType()](#module_concerto-core.Property+getType) ⇒
<code>string</code>
        * [.isOptional()](#module_concerto-core.Property+isOptional) ⇒
<code>boolean</code>
        * [.getFullyQualifiedTypeName()](#module_concerto-
core.Property+getFullyQualifiedTypeName) ⇒ <code>string</code>
        * [.getFullyQualifiedName()](#module_concerto-
core.Property+getFullyQualifiedName) ⇒ <code>string</code>
        * [.getNamespace()](#module_concerto-core.Property+getNamespace) ⇒
<code>string</code>
        * [.isArray()](#module_concerto-core.Property+isArray) ⇒
<code>boolean</code>
        * [.isTypeEnum()](#module_concerto-core.Property+isTypeEnum) ⇒
<code>boolean</code>
        * [.isPrimitive()](#module_concerto-core.Property+isPrimitive) ⇒
<code>boolean</code>
    * _static_
        * [.Symbol.hasInstance(object)](#module_concerto-
core.Property.Symbol.hasInstance) ⇒ <code>boolean</code>

<a name="new_module_concerto-core.Property_new"></a>

#### new Property(parent, ast)
Create a Property.

**Throws**:

- <code>IllegalModelException</code>


| Param | Type | Description |
| --- | --- | --- |
| parent | <code>ClassDeclaration</code> | the owner of this property |
| ast | <code>Object</code> | The AST created by the parser |

<a name="module_concerto-core.Property+getParent"></a>

#### property.getParent() ⇒ <code>ClassDeclaration</code>
Returns the owner of this property

**Kind**: instance method of [<code>Property</code>](#module_concerto-
core.Property)
**Returns**: <code>ClassDeclaration</code> - the parent class declaration
<a name="module_concerto-core.Property+getName"></a>

#### property.getName() ⇒ <code>string</code>
Returns the name of a property

**Kind**: instance method of [<code>Property</code>](#module_concerto-
core.Property)
**Returns**: <code>string</code> - the name of this field
<a name="module_concerto-core.Property+getType"></a>

#### property.getType() ⇒ <code>string</code>
Returns the type of a property

**Kind**: instance method of [<code>Property</code>](#module_concerto-
core.Property)
**Returns**: <code>string</code> - the type of this field
<a name="module_concerto-core.Property+isOptional"></a>

#### property.isOptional() ⇒ <code>boolean</code>
Returns true if the field is optional

**Kind**: instance method of [<code>Property</code>](#module_concerto-
core.Property)
**Returns**: <code>boolean</code> - true if the field is optional
<a name="module_concerto-core.Property+getFullyQualifiedTypeName"></a>

#### property.getFullyQualifiedTypeName() ⇒ <code>string</code>
Returns the fully qualified type name of a property

**Kind**: instance method of [<code>Property</code>](#module_concerto-
core.Property)
**Returns**: <code>string</code> - the fully qualified type of this property
<a name="module_concerto-core.Property+getFullyQualifiedName"></a>

#### property.getFullyQualifiedName() ⇒ <code>string</code>
Returns the fully name of a property (ns + class name + property name)

**Kind**: instance method of [<code>Property</code>](#module_concerto-
core.Property)
**Returns**: <code>string</code> - the fully qualified name of this property
<a name="module_concerto-core.Property+getNamespace"></a>

#### property.getNamespace() ⇒ <code>string</code>
Returns the namespace of the parent of this property

**Kind**: instance method of [<code>Property</code>](#module_concerto-
core.Property)
**Returns**: <code>string</code> - the namespace of the parent of this property
<a name="module_concerto-core.Property+isArray"></a>

#### property.isArray() ⇒ <code>boolean</code>
Returns true if the field is declared as an array type

**Kind**: instance method of [<code>Property</code>](#module_concerto-
core.Property)
**Returns**: <code>boolean</code> - true if the property is an array type
<a name="module_concerto-core.Property+isTypeEnum"></a>

#### property.isTypeEnum() ⇒ <code>boolean</code>
Returns true if the field is declared as an enumerated value

**Kind**: instance method of [<code>Property</code>](#module_concerto-
core.Property)
**Returns**: <code>boolean</code> - true if the property is an enumerated value
<a name="module_concerto-core.Property+isPrimitive"></a>

#### property.isPrimitive() ⇒ <code>boolean</code>
Returns true if this property is a primitive type.

**Kind**: instance method of [<code>Property</code>](#module_concerto-
core.Property)
**Returns**: <code>boolean</code> - true if the property is a primitive type.
<a name="module_concerto-core.Property.Symbol.hasInstance"></a>
```

#### Property.Symbol.hasInstance(object) ⇒ <code>boolean</code>
Alternative instanceof that is reliable across different module instances

Kind: static method of [<code>Property</code>](#module_concerto-core.Property)
Returns: <code>boolean</code> - - True, if the object is an instance of a Property
See: https://github.com/hyperledger/composer-concerto/issues/47

Param	Type	Description
object	<code>object</code>	The object to test against

<a name="module_concerto-core.RelationshipDeclaration"></a>

### concerto-core~RelationshipDeclaration ⇐ <code>Property</code>
Class representing a relationship between model elements

Kind: inner class of [<code>concerto-core</code>](#module_concerto-core)
Extends: <code>Property</code>
See: See Property

* [~RelationshipDeclaration](#module_concerto-core.RelationshipDeclaration) ⇐ <code>Property</code>
    * [new RelationshipDeclaration(parent, ast)](#new_module_concerto-core.RelationshipDeclaration_new)
    * _instance_
        * [.toString()](#module_concerto-core.RelationshipDeclaration+toString) ⇒ <code>String</code>
    * _static_
        * [.Symbol.hasInstance(object)](#module_concerto-core.RelationshipDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>

<a name="new_module_concerto-core.RelationshipDeclaration_new"></a>

#### new RelationshipDeclaration(parent, ast)
Create a Relationship.

Throws:

- <code>IllegalModelException</code>

Param	Type	Description
parent	<code>ClassDeclaration</code>	The owner of this property
ast	<code>Object</code>	The AST created by the parser

<a name="module_concerto-core.RelationshipDeclaration+toString"></a>

#### relationshipDeclaration.toString() ⇒ <code>String</code>
Returns a string representation of this property

Kind: instance method of [<code>RelationshipDeclaration</code>](#module_concerto-core.RelationshipDeclaration)
Returns: <code>String</code> - the string version of the property.
<a name="module_concerto-core.RelationshipDeclaration.Symbol.hasInstance"></a>

#### RelationshipDeclaration.Symbol.hasInstance(object) ⇒ <code>boolean</code>

Alternative instanceof that is reliable across different module instances

Kind: static method of [<code>RelationshipDeclaration</code>](#module_concerto-core.RelationshipDeclaration)
Returns: <code>boolean</code> - - True, if the object is an instance of a RelationshipDeclaration
See: https://github.com/hyperledger/composer-concerto/issues/47

Param	Type	Description
object	<code>object</code>	The object to test against

<a name="module_concerto-core.TransactionDeclaration"></a>

### concerto-core~TransactionDeclaration ⇐ <code>ClassDeclaration</code>
Class representing the definition of an Transaction.

Kind: inner class of [<code>concerto-core</code>](#module_concerto-core)
Extends: <code>ClassDeclaration</code>
See: See ClassDeclaration

* [~TransactionDeclaration](#module_concerto-core.TransactionDeclaration) ⇐ <code>ClassDeclaration</code>
    * [new TransactionDeclaration(modelFile, ast)](#new_module_concerto-core.TransactionDeclaration_new)
    * _instance_
        * [.getSystemType()](#module_concerto-core.TransactionDeclaration+getSystemType) ⇒ <code>string</code>
    * _static_
        * [.Symbol.hasInstance(object)](#module_concerto-core.TransactionDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>

<a name="new_module_concerto-core.TransactionDeclaration_new"></a>

#### new TransactionDeclaration(modelFile, ast)
Create an TransactionDeclaration.

Throws:

- <code>IllegalModelException</code>

Param	Type	Description
modelFile	<code>ModelFile</code>	the ModelFile for this class
ast	<code>Object</code>	The AST created by the parser

<a name="module_concerto-core.TransactionDeclaration+getSystemType"></a>

#### transactionDeclaration.getSystemType() ⇒ <code>string</code>
Returns the base system type for Transactions from the system namespace

Kind: instance method of [<code>TransactionDeclaration</code>](#module_concerto-core.TransactionDeclaration)
Returns: <code>string</code> - the short name of the base system type
<a name="module_concerto-core.TransactionDeclaration.Symbol.hasInstance"></a>

#### TransactionDeclaration.Symbol.hasInstance(object) ⇒ <code>boolean</code>
Alternative instanceof that is reliable across different module instances

Kind: static method of [<code>TransactionDeclaration</code>](#module_concerto-core.TransactionDeclaration)
Returns: <code>boolean</code> - - True, if the object is an instance of a TransactionDeclaration
See: https://github.com/hyperledger/composer-concerto/issues/47

Param	Type	Description
object	<code>object</code>	The object to test against

---
id: version-0.21-ref-concerto-cli
title: Command Line
original_id: ref-concerto-cli

Install the `@accordproject/concerto-cli` npm package to access the Concerto command line interface (CLI). After installation you can use the `concerto` command and its sub-commands as described below.

To install the Concerto CLI:
```
npm install -g @accordproject/concerto-cli
```

## Usage

```md
concerto <cmd> [args]

Commands:
  concerto validate  validate JSON against model files
  concerto compile   generate code for a target platform
  concerto get       save local copies of external model dependencies

Options:
  --version      Show version number                                   [boolean]
  --verbose, -v                                                 [default: false]
  --help         Show help                                             [boolean]
```

## concerto validate
`concerto validate` lets you check whether a JSON sample is a valid instance of the given model.

```md
concerto validate

validate JSON against model files

Options:
  --version      Show version number                                   [boolean]
  --verbose, -v                                                 [default: false]
  --help         Show help                                             [boolean]
  --sample       sample JSON to validate                                [string]
  --ctoSystem    system model to be used                                [string]
```

~~~
  --ctoFiles     array of CTO files                                      [array]
  --offline      do not resolve external models       [boolean] [default: false]
```

### Example
For example, using the `validate` command to check the sample `request.json` file
from a [Late Delivery and Penalty](https://github.com/accordproject/cicero-
template-library/tree/master/src/latedeliveryandpenalty) clause:

```
concerto validate --sample request.json --ctoFiles model/clause.cto
```

returns:

```json
info:
{
  "$class":
"org.accordproject.latedeliveryandpenalty.LateDeliveryAndPenaltyRequest",
  "forceMajeure": false,
  "agreedDelivery": "2017-12-17T03:24:00.000-05:00",
  "goodsValue": 200,
  "transactionId": "9f241804-118e-439e-bef4-49ee8cf57875",
  "timestamp": "2019-10-29T15:08:46.219Z"
}
```

## concerto compile
`Concerto compile` takes an array of local CTO files, downloads any external
dependencies (imports) and then converts all the model to the target format.

```md
concerto compile

generate code for a target platform

Options:
  --version      Show version number                                   [boolean]
  --verbose, -v                                                 [default: false]
  --help         Show help                                             [boolean]
  --ctoSystem    system model to be used                                [string]
  --ctoFiles     array of CTO files                           [array] [required]
  --offline      do not resolve external models       [boolean] [default: false]
  --target       target of the code generation  [string] [default: "JSONSchema"]
  --output       output directory path            [string] [default: "./output/"]
```

At the moment, the available target formats are as follows:
- Go Lang: `concerto compile --ctoFiles modelfile.cto --target Go`
- Plant UML: `concerto compile --ctoFiles modelfile.cto --target PlantUML`
- Typescript: `concerto compile --ctoFiles modelfile.cto --target Typescript`
- Java: `concerto compile --ctoFiles modelfile.cto --target Java`
- JSONSchema: `concerto compile --ctoFiles modelfile.cto --target JSONSchema`
- XMLSchema: `concerto compile --ctoFiles modelfile.cto --target XMLSchema`

### Example
For example, using the `compile` command to export the `clause.cto` file from a
[Late Delivery and Penalty](https://github.com/accordproject/cicero-template-
~~~

library/tree/master/src/latedeliveryandpenalty) clause into `Go Lang` format:

```md
cd ./model
concerto compile --ctoFiles clause.cto --target Go
```

returns:
```md
info: Compiled to Go in './output/'.
```

## concerto get
`Concerto get` allows you to resolve and download external models from a set of local CTO files.

```md
concerto get

save local copies of external model dependencies

Options:
  --version      Show version number                                   [boolean]
  --verbose, -v                                                 [default: false]
  --help         Show help                                             [boolean]
  --ctoFiles     array of local CTO files                     [array] [required]
  --ctoSystem    system model to be used                                [string]
  --output       output directory path                  [string] [default: "./"]
```

### Example
For example, using the `get` command to get the external models in the `clause.cto` file from a [Late Delivery and Penalty](https://github.com/accordproject/cicero-template-library/tree/master/src/latedeliveryandpenalty) clause:

```md
concerto get --ctoFiles clause.cto
```

returns:
```md
info: Loaded external models in './'.
```

---
id: version-0.21-ref-concerto-decorators
title: Decorators
original_id: ref-concerto-decorators

Decorators are used to add metadata to Concerto model elements, typically to control how variables are edited, printed or transformed.

## Pdf

The `@Pdf` decorator is used to control how a variable is rendered by the `markdown-pdf` transformation, which is used to convert CiceroMark rich text to PDF.

### Attributes

style : specifies the style name used to render the variable. Default styles are [defined in the code](https://github.com/accordproject/markdown-transform/blob/master/packages/markdown-pdf/src/PdfTransformer.js#L278) and may be overridden or supplemented via the `options.styles` parameter.

### Example

The example below renders the `title` variable using the PDF background style, which is defined to have the color `white`.

```
asset ExampleClause extends AccordClause {
   @Pdf("style", "background")
   o String title
}
```

## ContractEditor

The `@ContractEditor` decorator is used to control how a variable is edited using the `ContractEditor` React [web-components](https://github.com/accordproject/web-components).

### Attributes

readOnly : when set to true the variable value cannot be edited

fontFamily : the name of the HTML font-family to use when rendering the variable

backgroundColor : the HTML background color to use when rendering the variable

border : the HTML border color to use when rendering the variable

### Example

The example below renders the `title` variable using custom font, background color and border color. The variable is read-only and cannot be edited.

```
asset ExampleClause extends AccordClause {
   @ContractEditor("readOnly", true,
    "fontFamily", "Lucida Console, Courier, monospace",
    "backgroundColor", "#FAE094", "border", '#CCA855' )
   o String title
}
```

## FormEditor

The `@FormEditor` decorator is used to control whether the `ConcertoForm` React [web-components](https://github.com/accordproject/web-components) creates an input field for the variable.

### Attributes

hide : when set to true an input field for the variable is not created

### Example

The example specifies that an input field for the `title` variable should not be created by the Concerto Form component.

```
asset ExampleClause extends AccordClause {
   @FormEditor("hide", true)
   o String title
}
```

## DocuSignTab

The `@DocuSignTab` decorator is used to specify how a variable is mapped to a DocuSign tab. This decorator is not currently supported by existing Accord Project transformations but is reserved for future use, and may be used by upstream consumers.

### Attributes

type : the type of the DocuSign tab. See the documentation for DocuSign [EnvelopeRecipientTabs](https://developers.docusign.com/docs/esign-rest-api/reference/Envelopes/EnvelopeRecipientTabs/#tab-types) for the list of supported tab types.

optional : whether the tab is optional or required

### Example

The example below maps the `title` variable to the DocuSign tab type `Title` and marks it as optional.

```
asset ExampleClause extends AccordClause {
  @DocuSignTab("type", "Title", "optional", true)
   o String title
}
```

---
id: version-0.21-ref-ergo-api
title: Node.js API
original_id: ref-ergo-api

## Classes

<dl>
<dt><a href="#Commands">Commands</a></dt>
<dd><p>Utility class that implements the commands exposed by the Ergo CLI.</p>
</dd>
</dl>

## Functions

<dl>
<dt><a href="#getJson">getJson(input)</a> ⇒ <code>object</code></dt>
<dd><p>Load a file or JSON string</p>
</dd>
<dt><a href="#loadTemplate">loadTemplate(template, files)</a> ⇒
<code>Promise.&lt;LogicManager&gt;</code></dt>
<dd><p>Load a template from directory or files</p>
</dd>
<dt><a href="#fromDirectory">fromDirectory(path, [options])</a> ⇒
<code>Promise.&lt;LogicManager&gt;</code></dt>
<dd><p>Builds a LogicManager from a directory.</p>
</dd>
<dt><a href="#fromZip">fromZip(buffer, [options])</a> ⇒
<code>Promise.&lt;LogicManager&gt;</code></dt>
<dd><p>Builds a LogicManager from a Zip.</p>
</dd>
<dt><a href="#fromFiles">fromFiles(files, [options])</a> ⇒
<code>Promise.&lt;LogicManager&gt;</code></dt>
<dd><p>Builds a LogicManager from files.</p>
</dd>
<dt><a href="#setCurrentTime">setCurrentTime(currentTime)</a> ⇒
<code>object</code></dt>
<dd><p>Ensures there is a proper current time</p>
</dd>
<dt><a href="#init">init(engine, logicManager, contractJson, currentTime)</a> ⇒
<code>object</code></dt>
<dd><p>Invoke Ergo contract initialization</p>
</dd>
<dt><a href="#trigger">trigger(engine, logicManager, contractJson, stateJson,
currentTime, requestJson)</a> ⇒ <code>object</code></dt>
<dd><p>Trigger the Ergo contract with a request</p>
</dd>
<dt><a href="#resolveRootDir">resolveRootDir(parameters)</a> ⇒
<code>string</code></dt>
<dd><p>Resolve the root directory</p>
</dd>
<dt><a href="#compareComponent">compareComponent(expected, actual)</a></dt>
<dd><p>Compare actual and expected result components</p>
</dd>
<dt><a href="#compareSuccess">compareSuccess(expected, actual)</a></dt>
<dd><p>Compare actual result and expected result</p>
</dd>
</dl>

<a name="Commands"></a>

## Commands
Utility class that implements the commands exposed by the Ergo CLI.

Kind: global class

* [Commands](#Commands)
    * [.draft(template, files, contractInput, currentTime, options)]
(#Commands.draft) ⇒ <code>object</code>
    * [.trigger(template, files, contractInput, stateInput, currentTime,
requestsInput, warnings)](#Commands.trigger) ⇒ <code>object</code>
    * [.invoke(template, files, clauseName, contractInput, stateInput, currentTime,
paramsInput, warnings)](#Commands.invoke) ⇒ <code>object</code>

```
    * [.initialize(template, files, contractInput, currentTime, paramsInput,
warnings)](#Commands.initialize) ⇒ <code>object</code>
    * [.parseCTOtoFileSync(ctoPath)](#Commands.parseCTOtoFileSync) ⇒
<code>string</code>
    * [.parseCTOtoFile(ctoPath)](#Commands.parseCTOtoFile) ⇒ <code>string</code>

<a name="Commands.draft"></a>

### Commands.draft(template, files, contractInput, currentTime, options) ⇒
<code>object</code>
Invoke draft for an Ergo contract

**Kind**: static method of [<code>Commands</code>](#Commands)
**Returns**: <code>object</code> - Promise to the result of execution

| Param | Type | Description |
| --- | --- | --- |
| template | <code>string</code> | template directory |
| files | <code>Array.&lt;string&gt;</code> | input files |
| contractInput | <code>string</code> | the contract data |
| currentTime | <code>string</code> | the definition of 'now' |
| options | <code>object</code> | to the text generation |

<a name="Commands.trigger"></a>

### Commands.trigger(template, files, contractInput, stateInput, currentTime,
requestsInput, warnings) ⇒ <code>object</code>
Send a request an Ergo contract

**Kind**: static method of [<code>Commands</code>](#Commands)
**Returns**: <code>object</code> - Promise to the result of execution

| Param | Type | Description |
| --- | --- | --- |
| template | <code>string</code> | template directory |
| files | <code>Array.&lt;string&gt;</code> | input files |
| contractInput | <code>string</code> | the contract data |
| stateInput | <code>string</code> | the contract state |
| currentTime | <code>string</code> | the definition of 'now' |
| requestsInput | <code>Array.&lt;string&gt;</code> | the requests |
| warnings | <code>boolean</code> | whether to print warnings |

<a name="Commands.invoke"></a>

### Commands.invoke(template, files, clauseName, contractInput, stateInput,
currentTime, paramsInput, warnings) ⇒ <code>object</code>
Invoke an Ergo contract's clause

**Kind**: static method of [<code>Commands</code>](#Commands)
**Returns**: <code>object</code> - Promise to the result of invocation

| Param | Type | Description |
| --- | --- | --- |
| template | <code>string</code> | template directory |
| files | <code>Array.&lt;string&gt;</code> | input files |
| clauseName | <code>string</code> | the name of the clause to invoke |
| contractInput | <code>string</code> | the contract data |
| stateInput | <code>string</code> | the contract state |
| currentTime | <code>string</code> | the definition of 'now' |
```

| paramsInput | <code>object</code> | the parameters for the clause |
| warnings | <code>boolean</code> | whether to print warnings |

<a name="Commands.initialize"></a>

### Commands.initialize(template, files, contractInput, currentTime, paramsInput, warnings) ⇒ <code>object</code>
Invoke init for an Ergo contract

Kind: static method of [<code>Commands</code>](#Commands)
Returns: <code>object</code> - Promise to the result of execution

Param	Type	Description
template	<code>string</code>	template directory
files	<code>Array.&lt;string&gt;</code>	input files
contractInput	<code>string</code>	the contract data
currentTime	<code>string</code>	the definition of 'now'
paramsInput	<code>object</code>	the parameters for the clause
warnings	<code>boolean</code>	whether to print warnings

<a name="Commands.parseCTOtoFileSync"></a>

### Commands.parseCTOtoFileSync(ctoPath) ⇒ <code>string</code>
Parse CTO to JSON File

Kind: static method of [<code>Commands</code>](#Commands)
Returns: <code>string</code> - The name of the generated CTOJ model file

Param	Type	Description
ctoPath	<code>string</code>	path to CTO model file

<a name="Commands.parseCTOtoFile"></a>

### Commands.parseCTOtoFile(ctoPath) ⇒ <code>string</code>
Parse CTO to JSON File

Kind: static method of [<code>Commands</code>](#Commands)
Returns: <code>string</code> - The name of the generated CTOJ model file

Param	Type	Description
ctoPath	<code>string</code>	path to CTO model file

<a name="getJson"></a>

## getJson(input) ⇒ <code>object</code>
Load a file or JSON string

Kind: global function
Returns: <code>object</code> - JSON object

Param	Type	Description
input	<code>object</code>	either a file name or a json string

<a name="loadTemplate"></a>

```
## loadTemplate(template, files) ⇒ <code>Promise.&lt;LogicManager&gt;</code>
Load a template from directory or files

**Kind**: global function
**Returns**: <code>Promise.&lt;LogicManager&gt;</code> - a Promise to the
instantiated logicmanager

| Param | Type | Description |
| --- | --- | --- |
| template | <code>string</code> | template directory |
| files | <code>Array.&lt;string&gt;</code> | input files |

<a name="fromDirectory"></a>

## fromDirectory(path, [options]) ⇒ <code>Promise.&lt;LogicManager&gt;</code>
Builds a LogicManager from a directory.

**Kind**: global function
**Returns**: <code>Promise.&lt;LogicManager&gt;</code> - a Promise to the
instantiated logicmanager

| Param | Type | Description |
| --- | --- | --- |
| path | <code>String</code> | to a local directory |
| [options] | <code>Object</code> | an optional set of options to configure the
instance. |

<a name="fromZip"></a>

## fromZip(buffer, [options]) ⇒ <code>Promise.&lt;LogicManager&gt;</code>
Builds a LogicManager from a Zip.

**Kind**: global function
**Returns**: <code>Promise.&lt;LogicManager&gt;</code> - a Promise to the
instantiated logicmanager

| Param | Type | Description |
| --- | --- | --- |
| buffer | <code>Buffer</code> | the buffer to a Zip (zip) file |
| [options] | <code>Object</code> | an optional set of options to configure the
instance. |

<a name="fromFiles"></a>

## fromFiles(files, [options]) ⇒ <code>Promise.&lt;LogicManager&gt;</code>
Builds a LogicManager from files.

**Kind**: global function
**Returns**: <code>Promise.&lt;LogicManager&gt;</code> - a Promise to the
instantiated logicmanager

| Param | Type | Description |
| --- | --- | --- |
| files | <code>Array.&lt;String&gt;</code> | file names |
| [options] | <code>Object</code> | an optional set of options to configure the
instance. |

<a name="setCurrentTime"></a>
```

## setCurrentTime(currentTime) ⇒ <code>object</code>
Ensures there is a proper current time

Kind: global function
Returns: <code>object</code> - if valid, the moment object for the current time

Param	Type	Description
currentTime	<code>string</code>	the definition of 'now'

<a name="init"></a>

## init(engine, logicManager, contractJson, currentTime) ⇒ <code>object</code>
Invoke Ergo contract initialization

Kind: global function
Returns: <code>object</code> - Promise to the initial state of the contract

Param	Type	Description
engine	<code>object</code>	the execution engine
logicManager	<code>object</code>	the Template Logic
contractJson	<code>object</code>	contract data in JSON
currentTime	<code>string</code>	the definition of 'now'

<a name="trigger"></a>

## trigger(engine, logicManager, contractJson, stateJson, currentTime, requestJson) ⇒ <code>object</code>
Trigger the Ergo contract with a request

Kind: global function
Returns: <code>object</code> - Promise to the response

Param	Type	Description
engine	<code>object</code>	the execution engine
logicManager	<code>object</code>	the Template Logic
contractJson	<code>object</code>	contract data in JSON
stateJson	<code>object</code>	state data in JSON
currentTime	<code>string</code>	the definition of 'now'
requestJson	<code>object</code>	state data in JSON

<a name="resolveRootDir"></a>

## resolveRootDir(parameters) ⇒ <code>string</code>
Resolve the root directory

Kind: global function
Returns: <code>string</code> - root directory used to resolve file names

Param	Type	Description
parameters	<code>string</code>	Cucumber's World parameters

<a name="compareComponent"></a>

## compareComponent(expected, actual)
Compare actual and expected result components

Kind: global function

Param	Type	Description
expected	<code>string</code>	the expected component as specified in the test workload
actual	<code>string</code>	the actual component as returned by the engine

<a name="compareSuccess"></a>

## compareSuccess(expected, actual)
Compare actual result and expected result

Kind: global function

Param	Type	Description
expected	<code>string</code>	the expected successful result as specified in the test workload
actual	<code>string</code>	the successful result as returned by the engine

----
---
id: version-0.21-ref-ergo-cli
title: Command Line
original_id: ref-ergo-cli

Install the `@accordproject/ergo-cli` npm package to access the Ergo command line interface (CLI). After installation you can use the ergo command and its sub-commands as described below.

To install the Ergo CLI:
```
npm install -g @accordproject/ergo-cli
```

This will install `ergo`, to compile and run contracts locally on your machine, and `ergotop`, which is a _read-eval-print-loop_ utility to write Ergo interactively.

> In ergo-cli `0.20` release, `ergoc`, the Ergo compiler, and `ergorun`, used to run contracts locally on your machine, which were previously part of the `ergo-cli` npm package, have been merged into `ergo` commands.
>
> For more information about the changes that were made for the `0.20` release, please refer to our [Migrating from 0.13.*](ref-migrate-0.13-0.20.html) guide.

## Ergo

### Usage

```md
ergo <command>

Commands:
  ergo trigger     send a request to the contract
```

~~~~
  ergo invoke      invoke a clause of the contract
  ergo initialize  initialize the state for a contract
  ergo compile     compile a contract

Options:
  --help         Show help                                             [boolean]
  --version      Show version number                                   [boolean]
  --verbose, -v                                                 [default: false]
```

## ergo trigger
`ergo trigger` allows you to send a request to the contract.

```md
Usage: ergo trigger --data [file] --state [file] --request [file] [cto files] [ergo
files]

Options:
  --help         Show help                                             [boolean]
  --version      Show version number                                   [boolean]
  --verbose, -v                                                 [default: false]
  --data         path to the contract data                            [required]
  --state        path to the state data               [string] [default: null]
  --currentTime  the current time[string] [default: "2020-09-29T11:06:48-04:00"]
  --request      path to the request data                   [array] [required]
  --template     path to the template directory       [string] [default: null]
  --warnings     print warnings                     [boolean] [default: false]
```

### Example

For example, using the `trigger` command for the [Volume Discount example]
(https://github.com/accordproject/ergo/tree/master/examples/volumediscount) in the
[Ergo Directory](https://github.com/accordproject/ergo):

```md
ergo trigger --template ./examples/volumediscount --data
./examples/volumediscount/data.json --request
./examples/volumediscount/request.json --state ./examples/volumediscount/state.json
```

returns:

```json
{
  "clause": "orgXaccordprojectXvolumediscountXVolumeDiscount",
  "request": {
    "$class": "org.accordproject.volumediscount.VolumeDiscountRequest",
    "netAnnualChargeVolume": 10.4
  },
  "response": {
    "$class": "org.accordproject.volumediscount.VolumeDiscountResponse",
    "discountRate": 2.8,
    "transactionId": "3024ea58-ad82-4c83-87b1-d8fea8436d49",
    "timestamp": "2019-10-31T11:17:36.038Z"
  },
  "state": {
    "$class": "org.accordproject.cicero.contract.AccordContractState",
    "stateId": "1"
~~~~

~~~~
  },
  "emit": []
}
```

As the `request` was sent for an annual charge volume of 10.4, which falls into the
third discount rate category (as specified in the `data.json` file), the `response`
returns with a discount rate of 2.8%.

## ergo invoke
`ergo invoke` allows you to invoke a specific clause of the contract. The main
difference between `ergo invoke` and `ergo trigger` is that `ergo invoke` sends
data to a specific clause, whereas `ergo trigger` lets the contract choose which
clause to invoke. This is why `--clauseName` (the name of the contract you want to
execute) is a required field for `ergo invoke`.

You need to pass the CTO and Ergo files (`--template`), the name of the contract
that you want to execute (`--clauseName`), and JSON files for: the contract data
(`--data`), the contract parameters (`--params`), the current state of the contract
(`--state`), and the request.

If contract invocation is successful, `ergorun` will print out the response, the
new contract state and any emitted events.

```md
Usage: ergo invoke --data [file] --state [file] --params [file] [cto files] [ergo
files]

Options:
  --help         Show help                                           [boolean]
  --version      Show version number                                 [boolean]
  --verbose, -v                                               [default: false]
  --clauseName   the name of the clause to invoke                   [required]
  --data         path to the contract data                          [required]
  --state        path to the state data                    [string] [required]
  --currentTime  the current time[string] [default: "2020-09-29T11:07:24-04:00"]
  --params       path to the parameters       [string] [required] [default: {}]
  --template     path to the template directory        [string] [default: null]
  --warnings     print warnings                       [boolean] [default: false]
```

### Example

For example, using the `invoke` command for the [Volume Discount
example](https://github.com/accordproject/ergo/tree/master/examples/volumediscount)
in the [Ergo Directory](https://github.com/accordproject/ergo):

```md
ergo invoke --template ./examples/volumediscount --clauseName volumediscount --data
./examples/volumediscount/data.json --params ./examples/volumediscount/params.json
--state ./examples/volumediscount/state.json
```

returns:

```json
info:
{
  "clause": "orgXaccordprojectXvolumediscountXVolumeDiscount",
~~~~

~~~
  "params": {
    "request": {
      "$class": "org.accordproject.volumediscount.VolumeDiscountRequest",
      "netAnnualChargeVolume": 10.4
    }
  },
  "response": {
    "$class": "org.accordproject.volumediscount.VolumeDiscountResponse",
    "discountRate": 2.8,
    "transactionId": "2a979363-56bc-48ff-a6b4-49994a848a0c",
    "timestamp": "2019-10-31T11:22:37.368Z"
  },
  "state": {
    "$class": "org.accordproject.cicero.contract.AccordContractState",
    "stateId": "1"
  },
  "emit": []
}
```

Although this looks very similar to what `ergo trigger` returns, it is important to
note that `--clauseName volumediscount` was specifically invoked.

## ergo initialize
`ergo initialize` allows you to obtain the initial state of the contract. This is
the state of the contract without requests or responses.

```md
Usage: ergo intialize --data [file] --params [file] [cto files] [ergo files]

Options:
  --help         Show help                                             [boolean]
  --version      Show version number                                   [boolean]
  --verbose, -v                                                 [default: false]
  --data         path to the contract data                            [required]
  --currentTime  the current time[string] [default: "2020-09-29T11:07:47-04:00"]
  --params       path to the parameters                [string] [default: null]
  --template     path to the template directory        [string] [default: null]
  --warnings     print warnings                      [boolean] [default: false]
```

### Example

For example, using the `initialize` command for the [Volume Discount example]
(https://github.com/accordproject/ergo/tree/master/examples/volumediscount) in the
[Ergo Directory](https://github.com/accordproject/ergo):

```md
ergo initialize --template ./examples/volumediscount --data
./examples/volumediscount/data.json
```

returns:

```json
info:
{
  "clause": "orgXaccordprojectXvolumediscountXVolumeDiscount",
  "params": {
~~~

~~~~
  },
  "response": null,
  "state": {
    "$class": "org.accordproject.cicero.contract.AccordContractState",
    "stateId": "org.accordproject.cicero.contract.AccordContractState#1"
  },
  "emit": []
}
```

## ergo compile
`ergo compile` takes your input models (`.cto` files) and input contracts (`.ergo`
files) and allows you to compile a contract into a target platform. By default,
Ergo compiles to JavaScript (ES6 compliant) for execution.


```md
Usage: ergo compile --target [lang] --link --monitor --warnings [cto files] [ergo
files]

Options:
  --help         Show help                                             [boolean]
  --version      Show version number                                   [boolean]
  --verbose, -v                                                 [default: false]
  --target       Target platform (available: es5,es6,cicero,java)
                                                      [string] [default: "es6"]
  --link         Link the Ergo runtime with the target code (es5,es6,cicero
                 only)                               [boolean] [default: false]
  --monitor      Produce compilation time information [boolean] [default: false]
  --warnings     print warnings                      [boolean] [default: false]
```

### Example
For example, using the `compile` command on the [Volume Discount
example](https://github.com/accordproject/ergo/tree/master/examples/volumediscount)
in the [Ergo Directory](https://github.com/accordproject/ergo):

```md
ergo compile ./examples/volumediscount/model/model.cto
./examples/volumediscount/logic/logic.ergo
```

returns:

```md
Compiling Ergo './examples/volumediscount/logic/logic.ergo' --
'./examples/volumediscount/logic/logic.js'
```

Which means a new `logic.js` file is located in the
`./examples/volumediscount/logic` directory.

To compile the contract to Javascript and **link the Ergo runtime for execution**:
```md
ergo compile ./examples/volumediscount/model/model.cto
./examples/volumediscount/logic/logic.ergo --link
```

returns:
~~~~

~~~~
```md
Compiling Ergo './examples/volumediscount/logic/logic.ergo' --
'./examples/volumediscount/logic/logic.js'
```


--------------------------------------------------------------------------------
---
id: version-0.21-ref-ergo-repl
title: Read-Eval-Print Loop
original_id: ref-ergo-repl
---

`ergotop` is a convenient tool to try-out Ergo contracts in an interactive way. You
can write commands, or expressions and see the result. It is often called the Ergo
REPL, for _read-eval-print-loop_, since it literally: reads your input Ergo from
the command-line, evaluates it, prints the result and loops back to read your next
input.

## Starting the REPL

To start the REPL:

```
$ ergotop
Welcome to ERGOTOP version 0.20.0
ergo$
```

It should print the prompt `ergo$` which indicates it is ready to read your
command. For instance:

```ergo
    ergo$ return 42
    Response. 42 : Integer
```

`ergotop` prints back both the resulting value and its type. You can then keep
typing commands:

```ergo
    ergo$ return "hello " ++ "world!"
    Response. "hello world!" : String
    ergo$ define constant pi = 3.14
    ergo$ return pi ^ 2.0
    Response. 9.8596 : Double
```

If your expression is not valid, or not well-typed, it will return an error:

```ergo
    ergo$ return if true else "hello"
    Parse error (at line 1 col 15).
    return if true else "hello"
                   ^^^^
    ergo$ return if "hello" then 1 else 2
    Type error (at line 1 col 10). 'if' condition not boolean.
    return if "hello" then 1 else 2
~~~~

              ^^^^^^^

If what you are trying to write is too long to fit on one line, you can use `\` to go to a new line:

```ergo
    ergo$ define function squares(l:Double[]) : Double[] { \
      ...   return \
      ...     foreach x in l return x * x \
      ... }
    ergo$ return squares([2.4,4.5,6.7])
    Response. [5.76, 20.25, 44.89] : Double[]
```

## Loading files

You can load CTO and Ergo files to use in your REPL session. Once the REPL is launched you will have to import the corresponding namespace. For instance, if you want to use the `compoundInterestMultiple` function defined in the `./examples/promissory-note/money.ergo` file, you can do it as follows:

```ergo
$ ergotop ./examples/promissory-note/logic/money.ergo
Welcome to ERGOTOP version 0.20.0

ergo$ import org.accordproject.ergo.money.*
ergo$ return compoundInterestMultiple(0.035, 100)
Response. 1.00946960405 : Double
```

## Calling contracts

To call a contract, you first needs to _instantiate_ it, which means setting its parameters and initializing its state. You can do this by using the `set contract` and `call init` commands respectively. Here is an example using the `volumediscount` template:

```ergo
$ ergotop ./examples/volumediscount/model/model.cto
./examples/volumediscount/logic/logic.ergo
ergo$ import org.accordproject.cicero.contract.*
ergo$ import org.accordproject.volumediscount.*
ergo$ set contract VolumeDiscount over VolumeDiscountContract {firstVolume: 1.0,
secondVolume: 10.0, firstRate: 3.0, secondRate: 2.9, thirdRate: 2.8, contractId:
"0", parties: none }
ergo$ call init()
Response. unit : Unit
State. AccordContractState{stateId:
"org.accordproject.cicero.contract.AccordContractState#1"} : AccordContractState
```

You can then invoke clauses of the contract:

```ergo
ergo$ call volumediscount(VolumeDiscountRequest{ netAnnualChargeVolume : 0.1 })
Response. VolumeDiscountResponse{discountRate: 3.0} : VolumeDiscountResponse
ergo$ call volumediscount(VolumeDiscountRequest{ netAnnualChargeVolume : 10.5 })
Response. VolumeDiscountResponse{discountRate: 2.8} : VolumeDiscountResponse
```

You can also invoke the contract without explicitly naming the clause by sending a
request. The Ergo engine dispatches that request to the first clause which can
handle it:
```ergo
ergo$ send VolumeDiscountRequest{ netAnnualChargeVolume : 0.1 }
Response. VolumeDiscountResponse{discountRate: 3.0} : VolumeDiscountResponse
ergo$ send VolumeDiscountRequest{ netAnnualChargeVolume : 10.5 }
Response. VolumeDiscountResponse{discountRate: 2.8} : VolumeDiscountResponse
```

---
id: version-0.21-ref-ergo-spec
title: Specification
original_id: ref-ergo-spec

## Lexical Conventions

### File Extension

Ergo files have the ``.ergo`` extension.

### Blanks

Blank characters (such as space, tabulation, carriage return) are
ignored but they are used to separate identifiers.

### Comments

Comments come in two forms. Single line comments are introduced by the
two characters `//` and are terminated by the end of the current
line. Multi-line comments start with the two characters `/*` and are
terminated by the two characters `*/`. Multi-line comments can be
nested.

Here are examples of comments:

```ergo
    // This is a single line comment
    /* This comment spans multiple lines
       and it can also be /* nested */ */
```

### Reserved Words

The following are reserved as keywords in Ergo. They cannot be used as identifiers.

```text
namespace, import, define, function, transaction, concept, event, asset,
participant, enum, extends, contract, over, clause, throws, emits, state, call,
enforce, if, then, else, let, foreach, return, in, where, throw,
constant, match, set, emit, with, or, and, true, false, unit, none
```

## Condition Expressions

Conditional statements, conditional expressions and conditional constructs are
features of a programming language which perform different computations or actions
depending on whether a programmer-specified boolean condition evaluates to true or
false.

Conditional expressions (also known as `if` statements) allow us to conditionally
execute Ergo code depending on the value of a test condition. If the test condition
evaluates to `true` then the code on the `then` branch is evaluated. Otherwise,
when the test condition evaluates to `false` then the `else` branch is evaluated.

### Example

```ergo
if delayInDays > 15.0 then
  BuyerMayTerminateResponse{};
else
  BuyerMayNotTerminateResponse{}
```

### Legal Prose

For example, this corresponds to a conditional logic statement in legal
prose.

    If the delay is more than 15 days, the Buyer is entitled to terminate this
Contract.

### Syntax

```ergo
    if expression1 then     // Condition
      expression2           // Expression if condition is true
    else
      expression3           // Expression if condition is false
```

Where `expression1` is an Ergo expression that evaluates to a Boolean
value (i.e. `true` or `false`), and `expression2` and `expression3` are
Ergo expressions.

> Note that as with all Ergo expressions, new lines and indentation
> don't change the meaning of your code. However it is good practice to
> standardise the way that you using whitespace in your code to make it
> easier to read.

### Usage

If statements can be chained , i.e., `if ... then .... else if ... then ...
else ...` to build more compound conditionals.

```ergo
if request.netAnnualChargeVolume < contract.firstVolume then
  return VolumeDiscountResponse{ discountRate: contract.firstRate }
else if request.netAnnualChargeVolume < contract.secondVolume then
  return VolumeDiscountResponse{ discountRate: contract.secondRate }
else
  return VolumeDiscountResponse{ discountRate: contract.thirdRate }
```

Conditional expressions can also be used as expressions, e.g., inside a constant declaration:

```ergo
define constant price = 42;
define constant message =  if price >i 100 then "High price" else "Low Price";
message;
```

The value of message after running this code will be `"Low Price"`.

### Related

-   [Match expression](ref-logic#match-expressions) - where many
    alternative conditions check the same variable

## Match Expressions

Match expressions allow us to check an expression against multiple
possible values or patterns. If a match is found, then Ergo will
evaluate the corresponding expression.

> Match expressions are similar to `switch` statements in other
> programming languages

### Example

```ergo
match request.status
  with "CREATED" then
    new PayOut{ amount : contract.deliveryPrice }
  with "ARRIVED" then
    new PayOut{ amount : contract.deliveryPrice - shockPenalty }
  else
    new PayOut{ amount : 0.0 }
```

### Legal Prose

> Example needed.

### Syntax

```ergo
match expression0
  with pattern1 then       // Repeat this line
    expression1            //    and this line
  else
    expression2
```

### Usage

You can use a match expression to look for patterns based on the type of
an expression.

```ergo
match response
    with let b1 : BuyerMayTerminateResponse then
```

```
        // Do something with b1
    with let b2 : BuyerMayNotTerminateResponse then
        // Do something with b2
    else
        // Do a default action
```

You can use it to match against an optional value.

```ergo
match maybe_response
    with let? b1 : BuyerMayTerminateResponse then
        // Do something when there is a response
    else
        // Do something else when there is no response
```

Often a match expression is a more concise way to represent a
conditional expression with a repeating, regular condition. For example:

```ergo
    if x = 1 then
      ...
    else if x = 2 then
      ...
    else if x = 3 then
      ...
    else if x = 4 then
      ...
    else
      ...
```

This is equivalent to the match expression:

```ergo
    match x
      with 1 then
        ...
      with 2 then
        ...
      with 3 then
        ...
      with 4 then
        ...
      else
        ...
```

## Operator Precedence

Precedence determines the order of operations in expressions with operators of
different priority. In the case of the same precedence, it is based on the
associativity of operators.

### Example

`a = b * c ^ d + e` is the same as `(a = (b * (c ^ d)) + e)` `a = b * c * d / e` is the same as `(a = (((b * c) * d) / e)`

`a.b.c.d.e ^ f` is the same as `(((((a.b).c).d).e) ^ f)`

### Table of precedence

Table of operators in Ergo with their associativity and precedence from highest to lowest:

Order	**Operator(s)**	**Description**	**Associativity**
1	. <br> ?.	field access <br> field access of optional type	left to right
2	[]	array index access	right to left
3	!	logical not	right to left
4	\-	arithmetic negation	right to left
5	++	string concatenation	left to right
6	^	floating point number power	left to right
7	* <br> / <br> %	multiplication <br> division <br> remainder	left to right
8	\+ <br> -	addition <br> subtraction	left to right
9	??	default value of optional type	left to right
10	and	logical conjunction	left to right
11	or	logical disjunction	left to right
12	< <br> > <br> <= <br> >= <br> = <br> !=	less than <br> greater than <br> less or equal <br> greater or equal <br> equal <br> not equal	left to right

----
---
id: version-0.21-ref-ergo-stdlib
title: Standard Library
original_id: ref-ergo-stdlib

The following libraries are provided with the Ergo compiler.

## Stdlib

The following functions are in the `org.accordproject.ergo.stdlib` namespace and available by default.

### Functions on Integer

Name	Signature	Description
`integerAbs`	`(x:Integer) : Integer`	Absolute value
`integerLog2`	`(x:Integer) : Integer`	Base 2 integer logarithm
`integerSqrt`	`(x:Integer) : Integer`	Integer square root
`integerToDouble`	`(x:Integer) : Double`	Cast to a Double
`integerModulo`	`(x:Integer, y:Integer) : Integer`	Integer remainder
`integerMin`	`(x:Integer, y:Integer) : Integer`	Smallest of `x` and `y`
`integerMax`	`(x:Integer, y:Integer) : Integer`	Largest of `x` and `y`

### Functions on Long

Name	Signature	Description
`longAbs`	`(x:Long) : Long`	Absolute value
`longLog2`	`(x:Long) : Long`	Base 2 long logarithm
`longSqrt`	`(x:Long) : Long`	Long square root
`longToDouble`	`(x:Long) : Double`	Cast to a Double

`longModulo`	`(x:Long, y:Long) : Long`	Long remainder
`longMin`	`(x:Long, y:Long) : Long`	Smallest of `x` and `y`
`longMax`	`(x:Long, y:Long) : Long`	Largest of `x` and `y`

### Functions on Double

Name	Signature	Description
`abs`	`(x:Double) : Double`	Absolute value
`sqrt`	`(x:Double) : Double`	Square root
`exp`	`(x:Double) : Double`	Exponential
`log`	`(x:Double) : Double`	Natural logarithm
`log10`	`(x:Double) : Double`	Base 10 logarithm
`ceil`	`(x:Double) : Double`	Round to closest integer above
`floor`	`(x:Double) : Double`	Round to closest integer below
`truncate`	`(x:Double) : Integer`	Cast to an Integer
`doubleToInteger`	`(x:Double) : Integer`	Same as `truncate`
`doubleToLong`	`(x:Double) : Long`	Cast to a Long
`minPair`	`(x:Double, y:Double) : Double`	Smallest of `x` and `y`
`maxPair`	`(x:Double, y:Double) : Double`	Largest of `x` and `y`

### Functions on String

Name	Signature	Description
`length`	`(x:String) : Integer`	Prints length of a string
`encode`	`(x:String) : String`	Encode as URI component
`decode`	`(x:String) : String`	Decode as URI component
`doubleOpt`	`(x:String) : Double?`	Cast to a Double
`double`	`(x:String) : Double`	Cast to a Double or NaN
`integerOpt`	`(x:String) : Integer?`	Cast to an Integer
`integer`	`(x:String) : Integer`	Cast to a Integer or 0
`longOpt`	`(x:String) : Long?`	Cast to a Long
`long`	`(x:String) : Long`	Cast to a Long or 0

### Functions on Arrays

Name	Signature	Description
`count`	(x:Any[]) : Integer	Number of elements
`flatten`	(x:Any[][]) : Any[]	Flattens a nested array
`arrayAdd`	`(x:Any[],y:Any[]) : Any[]`	Array concatenation
`arraySubtract`	`(x:Any[],y:Any[]) : Any[]`	Removes elements of `y` in `x`
`inArray`	`(x:Any,y:Any[]) : Boolean`	Whether `x` is in `y`
`containsAll`	`(x:Any[],y:Any[]) : Boolean`	Whether all elements of `y` are in `x`
`distinct`	`(x:Any[]) : Any[]`	Duplicates elimination
`singleton`	`(x:Any[]) : Any?`	Single value from singleton array

Note: For most of these functions, the type-checker infers more precise types than indicated here. For instance `concat([1,2],[3,4])` will return `[1,2,3,4]` and have the type `Integer[]`.

### Log functions

Name	Signature	Description
`logString`	(x:String) : Unit	Adds string to the log

### Aggregate functions

Name	Signature	Description
`max`	(x:Double[]) : Double	The largest element in `x`
`min`	(x:Double[]) : Double	The smallest element in `x`
`sum`	(x:Double[]) : Double	Sum of the elements in `x`
`average`	(x:Double[]) : Double	Arithmetic mean

### Math functions

Name	Signature	Description
`acos`	(x:Double) : Double	The inverse cosine of x
`asin`	(x:Double) : Double	The inverse sine of x
`atan`	(x:Double) : Double	The inverse tangent of x
`atan2`	(x:Double, y:Double) : Double	The inverse tangent of `x / y`
`cos`	(x:Double) : Double	The cosine of x
`cosh`	(x:Double) : Double	The hyperbolic cosine of x
`sin`	(x:Double) : Double	The sine of x
`sinh`	(x:Double) : Double	The hyperbolic sine of x
`tan`	(x:Double) : Double	The tangent of x
`tanh`	(x:Double) : Double	The hyperbolic tangent of x

### Other functions

Name	Signature	Description
`failure`	`(x:String) : ErgoErrorResponse`	Ergo error from a string
`toString`	`(x:Any) : String`	Prints any value to a string
`toText`	`(x:Any) : String`	Template variant of `toString` (internal)

## Time

The following functions are in the `org.accordproject.time` namespace and are available by importing that namespace.
They rely on the [time.cto](https://models.accordproject.org/v2.0/time.html) types from the Accord Project models.

### Functions on DateTime

Name	Signature	Description
`now`	`() : DateTime`	Returns the time when execution started
`dateTime`	`(x:String) : DateTime`	Parse a date and time
`getSecond`	`(x:DateTime) : Long`	Second component of a DateTime
`getMinute`	`(x:DateTime) : Long`	Minute component of a DateTime
`getHour`	`(x:DateTime) : Long`	Hour component of a DateTime
`getDay`	`(x:DateTime) : Long`	Day of the month component of a DateTime
`getWeek`	`(x:DateTime) : Long`	Week of the year component of a DateTime
`getMonth`	`(x:DateTime) : Long`	Month component in a DateTime
`getYear`	`(x:DateTime) : Long`	Year component in a DateTime
`isAfter`	`(x:DateTime, y:DateTime) : Boolean`	Whether `x` if after `y`
`isBefore`	`(x:DateTime, y:DateTime) : Boolean`	Whether `x` is before `y`
`isSame`	`(x:DateTime, y:DateTime) : Boolean`	Whether `x` is the same DateTime as `y`
`dateTimeMin`	`(x:DateTime[]) : DateTime`	The earliest in an array of DateTime
`dateTimeMax`	`(x:DateTime[]) : DateTime`	The latest in an array of DateTime

|
| `format` | `(x:DateTime,f:String) : String` | Prints date `x` according to [format](markup-variables#datetime-formats) `f` |

### Functions on Duration

Name	Signature	Description
`durationAs`	`(x:Duration, y:TemporalUnit) : Duration`	Change the unit for duration `x` to `y`
`diffDurationAs`	`(x:DateTime, y:DateTime, z:TemporalUnit) : Duration`	Duration between `x` and `y` in unit `z`
`diffDuration`	`(x:DateTime, y:DateTime) : Duration`	Duration between `x` and `y` in seconds
`addDuration`	`(x:DateTime, y:Duration) : DateTime`	Add duration `y` to `x`
`subtractDuration`	`(x:DateTime, y:Duration) : DateTime`	Subtract duration `y` to `x`
`divideDuration`	`(x:Duration, y:Duration) : Double`	Ratio between durations `x` and `y`

### Functions on Period

Name	Signature	Description
`diffPeriodAs`	`(x:DateTime, y:DateTime, z:PeriodUnit) : Period`	Time period between `x` and `y` in unit `z`
`addPeriod`	`(x:DateTime, y:Period) : DateTime`	Add time period `y` to `x`
`subtractPeriod`	`(x:DateTime, y:Period) : DateTime`	Subtract time period `y` to `x`
`startOf`	`(x:DateTime, y:PeriodUnit) : DateTime`	Start of period `y` nearest to DateTime `x`
`endOf`	`(x:DateTime, y:PeriodUnit) : DateTime`	End of period `y` nearest to DateTime `x`

---
id: version-0.21-ref-errors
title: Errors
original_id: ref-errors

As much as possible, errors returned by all projects (notably Cicero and the Ergo compiler) are normalized and categorized in order to facilitate handling of those error by the application code. Those errors are raised as JavaScript _exceptions_.

## Errors Hierarchy

The hierarchy of errors (or exceptions) is shown on the following diagram:

![Error Hierarchy](assets/exceptions.png)

## Errors Model

For reference, those can also be described using the following Concerto model:

```ergo
namespace org.accordproject.errors
 /** Common */
```

```
concept LocationPoint {
  o Integer line
  o Integer column
  o Integer offset optional
}
concept FileLocation {
  o LocationPoint start
  o LocationPoint end
}
 concept BaseException {
      o String component // Node component the error originates from
  o String name      // name of the class
  o String message
}
concept BaseFileException extends BaseException {
      o FileLocation fileLocation
      o String shortMessage
      o String fileName
}
concept ParseException extends BaseFileException {
}
 /* Model errors */
concept ValidationException extends BaseException {
}
concept TypeNotFoundException extends BaseException {
      o String typeName
}
concept IllegalModelException extends BaseFileException {
      o String modelFile
}
 /* Ergo errors */
concept CompilerException extends BaseFileException {
}
concept TypeException extends BaseFileException {
}
concept SystemException extends BaseFileException {
}
 /* Cicero errors */
concept TemplateException extends ParseException {
}
```

---
id: version-0.21-ref-markus-cli
title: Command Line
original_id: ref-markus-cli

Install the `@accordproject/markdown-cli` npm package to access the Markdown Transform command line interface (CLI). After installation you can use the `markus` command and its sub-commands as described below.

To install the Markdown CLI:
```bash
npm install -g @accordproject/markdown-cli
```

## Usage

`markus` is a command line tool to debug and use markdown transformations.

```md
markus <cmd> [args]

Commands:
  markus transform  transform between two formats

Options:
  --version      Show version number                                   [boolean]
  --verbose, -v                                                 [default: false]
  --help         Show help                                             [boolean]
```

## markus transform

The `markus transform` command lets you transform between any two of the supported formats

```md
markus transform

transform between two formats

Options:
  --version      Show version number                                   [boolean]
  --verbose, -v  verbose output                       [boolean] [default: false]
  --help         Show help                                             [boolean]
  --input        path to the input                                      [string]
  --from         source format                    [string] [default: "markdown"]
  --to           target format                  [string] [default: "commonmark"]
  --via          intermediate formats                      [array] [default: []]
  --roundtrip    roundtrip transform                  [boolean] [default: false]
  --output       path to the output file                                [string]
  --model        array of concerto model files                           [array]
  --template     template grammar                                       [string]
  --contract     contract template                    [boolean] [default: false]
  --currentTime  set current time                       [string] [default: null]
  --plugin       path to a parser plugin                                [string]
  --sourcePos    enable source position               [boolean] [default: false]
  --offline      do not resolve external models       [boolean] [default: false]
```

### Example

For example, you can use the `transform` command on the `README.md` file from the [Hello World](https://github.com/accordproject/cicero-template-library/blob/master/src/helloworld) template:

```bash
markus transform --input README.md
```

returns:
```json
{
  "$class": "org.accordproject.commonmark.Document",
  "xmlns": "http://commonmark.org/xml/1.0",
  "nodes": [
    {
      "$class": "org.accordproject.commonmark.Heading",
      "level": "1",
      "nodes": [
        {
          "$class": "org.accordproject.commonmark.Text",
          "text": "Hello World"
        }
      ]
    },
    {
      "$class": "org.accordproject.commonmark.Paragraph",
      "nodes": [
        {
          "$class": "org.accordproject.commonmark.Text",
          "text": "This is the Hello World of Accord Project Templates. Executing
the clause will simply echo back the text that occurs after the string "
        },
        {
          "$class": "org.accordproject.commonmark.Code",
          "text": "Hello"
        },
        {
          "$class": "org.accordproject.commonmark.Text",
          "text": " prepended to text that is passed in the request."
        }
      ]
    }
  ]
}
```

### `--from` and `--to` options

You can indicate the source and target formats using the `--from` and `--to`
options. For instance, the following transforms from `markdown` to `html`:

```bash
markus transform --from markdown --to html
```

returns:

```md
<html>
<body>
<div class="document">
<h1>Hello World</h1>
<p>This is the Hello World of Accord Project Templates. Executing the clause will
simply echo back the text that occurs after the string <code>Hello</code> prepended
to text that is passed in the request.</p>
</div>
</body>
</html>
```

### `--via` option

When there are several paths between two formats, you can indicate an intermediate format using the `--via` option. The following transforms from `markdown` to `html` _via_ `slate`:

```bash
markus transform --from markdown --via slate --to html
```

returns:

```md
<html>
<body>
<div class="document">
<h1>Hello World</h1>
<p>This is the Hello World of Accord Project Templates. Executing the clause will
simply echo back the text that occurs after the string <code>Hello</code> prepended
to text that is passed in the request.</p>
</div>
</body>
</html>
```

### `--roundtrip` option

When the transforms allow, you can roundtrip between two formats, i.e., transform from a source to a target format and back to the source target. For instance, the following transform from `markdown` to `slate` and back to markdown:

```md
markus transform --from markdown --to slate --input README.md --roundtrip
```

returns:

```bash
Hello World
====

This is the Hello World of Accord Project Templates. Executing the clause will
simply echo back the text that occurs after the string `Hello` prepended to text
that is passed in the request.
```

:::
Roundtripping might result in small changes in the source markdown, but should always be semantically equivalent. In the above example the source ATX heading `# Hello World` has been transformed into a Setext heading equivalent.
:::

### `--model` `--contract` options

When handling [TemplateMark](markup-templatemark), one has to provide a model using the `--model` option and whether the template is a clause (default) or a contract (using the `--contract` option).

For instance the following converts markdown with the template extension to a TemplateMark document object model:
```bash
markus transform --from markdown_template --to templatemark --model model/model.cto
```

```
--input text/grammar.tem.md
```

returns:

```json
{
  "$class": "org.accordproject.commonmark.Document",
  "xmlns": "http://commonmark.org/xml/1.0",
  "nodes": [
    {
      "$class": "org.accordproject.templatemark.ClauseDefinition",
      "name": "top",
      "elementType": "org.accordproject.helloworld.HelloWorldClause",
      "nodes": [
        {
          "$class": "org.accordproject.commonmark.Paragraph",
          "nodes": [
            {
              "$class": "org.accordproject.commonmark.Text",
              "text": "Name of the person to greet: "
            },
            {
              "$class": "org.accordproject.templatemark.VariableDefinition",
              "name": "name",
              "elementType": "String"
            },
            {
              "$class": "org.accordproject.commonmark.Text",
              "text": "."
            },
            {
              "$class": "org.accordproject.commonmark.Softbreak"
            },
            {
              "$class": "org.accordproject.commonmark.Text",
              "text": "Thank you!"
            }
          ]
        }
      ]
    }
  ]
}
```

### `--template` option

Parsing or drafting contract text using a template can be done using the `--template` option, usually with the corresponding `--model` option to indicate the template model.

For instance, the following parses a markdown with CiceroMark extension to get the correspond contract data:
```bash
markus transform --from markdown_cicero --to data --template text/grammar.tem.md --model model/model.cto --input text/sample.md
```

returns:

```json
{
  "$class": "org.accordproject.helloworld.HelloWorldClause",
  "name": "Fred Blogs",
  "clauseId": "fc345528-2604-420c-9e02-8d85e03cb65b"
}
```

---
id: version-0.21-ref-migrate-0.13-0.20
title: Cicero 0.13 to 0.20
original_id: ref-migrate-0.13-0.20

Much has changed in the `0.20` release. This guide provides step-by-step instructions to port your Accord Project templates from version `0.13` or earlier to version `0.20`.

:::note
Before following those migration instructions, make sure to first install version `0.20` of Cicero, as described in the [Install Cicero](started-installation) Section of this documentation.
:::

## Metadata Changes

You will first need to update the `package.json` in your template. Remove the Ergo version which is now unnecessary, and change the Cicero version to `^0.20.0`.

#### Example

After those changes, the `accordproject` field in your `package.json` should look as follows (with the `template` field being either `clause` or `contract` depending on the template):
```js
...
    "accordproject": {
        "template": "clause",
        "cicero": "^0.20.0"
    }
...
```

## Template Directory Changes

The layout of templates has changed to reflect the conceptual notion of Accord Project templates (as a triangle composed of text, model and logic). To migrate a template directory from version `0.13` or earlier to the new `0.20` layout:
1. Rename your `lib` directory to `logic`
2. Rename your `models` directory to `model`
3. Rename your `grammar` directory to `text`
4. Rename your template grammar from `text/template.tem` to `text/grammar.tem.md`
5. Rename your samples from `sample.txt` to `text/sample.md` (or generally any other `sample*.txt` files to `text/sample*.md`)

#### Example

Consider the [late delivery and penalty](https://templates.accordproject.org/latedeliveryandpenalty@0.14.1.html) clause. After applying those changes, the template directory should look as follows:
```
./.cucumber.js
./README.md
./package.json
./request-forcemajeure.json
./request.json
./state.json

./logic:
./logic/logic.ergo

./model:
./model/clause.cto

./test:
./test/logic.feature
./test/logic_default.feature

./text:
./text/grammar.tem.md
./text/sample-noforcemajeure.md
./text/sample.md
```

## Text Changes

Both grammar and sample text for the templates has changed to support rich text annotations through CommonMark and a new syntax for variables. You can find complete information about the new syntax in the [CiceroMark](markup-cicero) Section of this documentation. For an existing template, you should apply the following changes.

### Text Grammar Changes

1. Variables should be changed from `[{variableName}]` to `{{variableName}}`
2. Formatted variables should be changed to from `[{variableName as "FORMAT"}]` to `{{variableName as "FORMAT"}}`
3. Boolean variables should be changed to use the new block syntax, from `[{"This is a force majeure":?forceMajeure}]` to `{{#if forceMajeure}}This is a force majeure{{/if}}`
4. Nested clauses should be changed to use the new block syntax, from `[{#payment}]As consideration in full for the rights granted herein...[{/payment}]` to `{{#clause payment}}As consideration in full for the rights granted herein... {{/clause}}`

:::note
1. Template text is now interpreted as CommonMark which may lead to unexpected results if your text includes CommonMark characters or structure (e.g., `#` or `##` now become headings; `1.` or `-` now become lists). You should review both the grammar and samples so they follow the proper [CommonMark](https://commonmark.org) rules.
2. The new lexer reserves `{{` instead of reserving `[{` which means you should avoid using `{{` in your text unless for Accord Project variables.
:::

### Text Samples Changes

You should ensure that any changes to the grammar text is reflected in the samples.
Any change in the grammar text outside of variables should be applied to the
corresponding `sample.md` files as well.

:::tip
You can check that the samples and grammar are in agreement by using the `cicero
parse` command.
:::

#### Example

Consider the text grammar for the [late delivery and
penalty](https://templates.accordproject.org/latedeliveryandpenalty@0.14.1.html)
clause:

```md
Late Delivery and Penalty.
In case of delayed delivery[{" except for Force Majeure cases,":? forceMajeure}]
[{seller}] (the Seller) shall pay to [{buyer}] (the Buyer) for every
[{penaltyDuration}]
of delay penalty amounting to [{penaltyPercentage}]% of the total value of the
Equipment
whose delivery has been delayed. Any fractional part of a [{fractionalPart}] is to
be
considered a full [{fractionalPart}]. The total amount of penalty shall not
however,
exceed [{capPercentage}]% of the total value of the Equipment involved in late
delivery.
If the delay is more than [{termination}], the Buyer is entitled to terminate this
Contract.
```

After applying the above rules to the code for the `0.13` version, and identifying
the heading for the clause using the new markdown features, the grammar text
becomes:

```tem
## Late Delivery and Penalty.

In case of delayed delivery{{#if forceMajeure}} except for Force Majeure
cases,{{/if}}
{{seller}} (the Seller) shall pay to {{buyer}} (the Buyer) for every
{{penaltyDuration}}
of delay penalty amounting to {{penaltyPercentage}}% of the total value of the
Equipment
whose delivery has been delayed. Any fractional part of a {{fractionalPart}} is to
be
considered a full {{fractionalPart}}. The total amount of penalty shall not
however,
exceed {{capPercentage}}% of the total value of the Equipment involved in late
delivery.
If the delay is more than {{termination}}, the Buyer is entitled to terminate this
Contract.
```

To make sure the `sample.md` file parses as well, the heading needs to be similarly

identified using markdown:
```md
## Late Delivery and Penalty.

In case of delayed delivery except for Force Majeure cases,
"Dan" (the Seller) shall pay to "Steve" (the Buyer) for every 2 days
of delay penalty amounting to 10.5% of the total value of the Equipment
whose delivery has been delayed. Any fractional part of a days is to be
considered a full days. The total amount of penalty shall not however,
exceed 55% of the total value of the Equipment involved in late delivery.
If the delay is more than 15 days, the Buyer is entitled to terminate this
Contract.
```

## Model Changes

There is no model changes required for this version.

## Logic Changes

Version `0.20` of Ergo has a few new features that are non backward compatible with version `0.13`. Those may require you to change your template logic. The main non-backward compatible feature is the new support for enumerated values.

### Enumerated Values

Enumerated values are now proper values with a proper enum type, and not based on the type `String` anymore.

For instance, consider the enum declaration:
```js
enum Cardsuit {
  o CLUBS
  o DIAMONDS
  o HEARTS
  o SPADES
}
```

In version `0.13` or earlier the Ergo code would write `"CLUBS"` for an enum value and treat the type `Cardsuit` as if it was the type `String`.

As of version `0.20` Ergo writes `CLUBS` for that same enum value and the type `Cardsuit` is now distinct from the type `String`.

If you try to compile Ergo logic written for version `0.13` or earlier that features enumerated values, the compiler will likely throw type errors. You should apply the following changes:

1. Remove the quotes (`"`) around any enum values in your logic. E.g., `"USD"` should now be replaced by `USD` for monetary amounts;
3. If enum values are bound to variables with a type annotation, you should change the type annotation from `String` to the correct enum type. E.g., `let x : String = "DIAMONDS"; ...` should become `let x : Cardsuit = DIAMONDS; ...`;
3. If enum values are passed as parameters in clauses or functions, you should change the type annotation for that parameter from `String` to the correct enum type.
4. In a few cases the same enumerated value may be used in different enum types (e.g., `days` and `weeks` are used in both `TemporalUnit` and `PeriodUnit`). Those

two values will now have different types. If you need to distinguish, you can use the fully qualified name for the enum value (e.g., `~org.accordproject.time.TemporalUnit.days` or `~org.accordproject.time.PeriodUnit.days`).

### Other Changes

1. `now` used to return the current time but is treated in `0.20` like any other variables. If your logic used the variable `now` without declaring it, this will raise a `Variable now not found` error. You should change your logic to use the `now()` function instead.

#### Example

Consider the Ergo logic for the [acceptance of delivery](https://templates.accordproject.org/acceptance-of-delivery@0.12.1.html) clause. Applying the above rules to the code for the `0.13` version:

```ergo
  clause acceptanceofdelivery(request : InspectDeliverable) : InspectionResponse {

    let received = request.deliverableReceivedAt;
    enforce isBefore(received,now) else
      throw ErgoErrorResponse{ message : "Transaction time is before the
deliverable date." }
    ;

    let dur =
      Duration{
        amount: contract.businessDays,
        unit: "days"
      };
    let status =
      if isAfter(now(), addDuration(received, dur))
      then "OUTSIDE_INSPECTION_PERIOD"
      else if request.inspectionPassed
      then "PASSED_TESTING"
      else "FAILED_TESTING"
    ;
    return InspectionResponse{
      status : status,
      shipper : contract.shipper,
      receiver : contract.receiver
    }
  }
```

results in the following new logic for the `0.20` version:

```ergo
  clause acceptanceofdelivery(request : InspectDeliverable) : InspectionResponse {

    let received = request.deliverableReceivedAt;
    enforce isBefore(received,now()) else                // changed to now()
      throw ErgoErrorResponse{ message : "Transaction time is before the
deliverable date." }
    ;

    let dur =
      Duration{
        amount: contract.businessDays,
        unit: ~org.accordproject.time.TemporalUnit.days  // enum value with fully
qualified name
      };
    let status =
      if isAfter(now(), addDuration(received, dur))      // changed to now()
      then OUTSIDE_INSPECTION_PERIOD                     // enum value has no
quotes
      else if request.inspectionPassed
      then PASSED_TESTING                                // enum value has no
quotes
      else FAILED_TESTING                                // enum value has no
quotes
    ;
    return InspectionResponse{
      status : status,
      shipper : contract.shipper,
      receiver : contract.receiver
    }
  }
```

## Command Line Changes

The Command Line interface for Cicero and Ergo has been completely overhauled for consistency. Release `0.20` also features new command line interfaces for Concerto and for the new `markdown-transform` project.

If you are familiar with the previous Accord Project command line interfaces (or if you have scripts relying on the previous version of the command line), here is a list of changes:

1. Ergo: A single new `ergo` command replaces both `ergoc` and `ergorun`
   - `ergoc` has been replaced by `ergo compile`
   - `ergorun execute` has been replaced by `ergo trigger`
   - `ergorun init` has been replaced by `ergo initialize`
   - All other `ergorun <command>` commands should use `ergo <command>` instead
2. Cicero:
   - The `cicero execute` command has been replaced by `cicero trigger`
   - The `cicero init` command has been replaced by `cicero initialize`
   - The `cicero generateText` command has been replaced by `cicero draft`
   - the `cicero generate` command has been replaced by `cicero compile`

Note that several options have been renamed for consistency as well. Some of the main option changes are:
1. `--out` and `--outputDirectory` have both been replaced by `--output`
2. `--format` has been replaced by `--target` in the new `cicero compile` command
3. `--contract` has been replaced by `--data` in all `ergo` commands

For more details on the new command line interface, please consult the corresponding [Cicero CLI](cicero-cli), [Concerto CLI](concerto-cli), [Ergo CLI](ergo-cli), and [Markus CLI](markus-cli) Sections in the reference manual.

## API Changes

A number of API changes may affect Node.js applications using Cicero or Ergo packages. The main API changes are:
1. Ergo:

~~~
   1. `@accordproject/ergo-engine` package
      - the `Engine.execute()` call has been renamed `Engine.trigger()`
2. Cicero:
   1. `@accordproject/cicero-core` package
      - the `TemplateInstance.generateText()` call has been renamed
`TemplateInstance.draft` **and is now an `async` call**
      - the `Metadata.getErgoVersion()` call has been removed
   2. `@accordproject/cicero-engine` package
      - the `Engine.execute()` call has been renamed `Engine.trigger()`
      - the `Engine.generateText()` call has been renamed `Engine.draft()`

## Cicero Server Changes

Cicero server API has been changed to reflect the new underlying Cicero engine.
Specifically:
1. The `execute` endpoint has been renamed `trigger`
2. The path to the sample has to include the `text` directory, so instead of
`execute/templateName/sample.txt` it should use `trigger/templateName/text
%2Fsample.md`

#### Example

Following the
[README.md](https://github.com/accordproject/cicero/blob/master/packages/cicero-
server/README.md) instructions, instead of calling:
```
curl -X POST --header 'Content-Type: application/json' --header 'Accept:
application/json' http://localhost:6001/execute/latedeliveryandpenalty/sample.txt -
d '{ "request": { "$class":
"org.accordproject.latedeliveryandpenalty.LateDeliveryAndPenaltyRequest",
"forceMajeure": false,"agreedDelivery": "December 17, 2017 03:24:00",
"deliveredAt": null, "goodsValue": 200.00 }, "state": { "$class":
"org.accordproject.cicero.contract.AccordContractState", "stateId" :
"org.accordproject.cicero.contract.AccordContractState#1"}}'
```

You should call:
```
curl -X POST --header 'Content-Type: application/json' --header 'Accept:
application/json' http://localhost:6001/trigger/latedeliveryandpenalty/sample.md -d
'{ "request": { "$class":
"org.accordproject.latedeliveryandpenalty.LateDeliveryAndPenaltyRequest",
"forceMajeure": false,"agreedDelivery": "December 17, 2017 03:24:00",
"deliveredAt": null, "goodsValue": 200.00 }, "state": { "$class":
"org.accordproject.cicero.contract.AccordContractState", "stateId" :
"org.accordproject.cicero.contract.AccordContractState#1"}}'
```



--------------------------------------------------------------------------------
---
id: version-0.21-ref-migrate-0.20-0.21
title: Cicero 0.20 to 0.21
original_id: ref-migrate-0.20-0.21
---

The main change between the `0.20` release and the `0.21` release is the new
markdown syntax and parser infrastructure based on
[`markdown-it`](https://github.com/markdown-it/markdown-it). While most templates
~~~

~~~
designed for `0.20` should still work on `0.21` some changes might be needed to the
contract or template text to account for this new syntax.

:::note
Before following those migration instructions, make sure to first install version
`0.21` of Cicero, as described in the [Install Cicero](started-installation)
Section of this documentation.
:::

## Metadata Changes

You should only have to update the Cicero version in the `package.json` for your
template to `^0.21.0`. Remember to also increment the version number for the
template itself.

#### Example

After those changes, the `accordproject` field in your `package.json` should look
as follows (with the `template` field being either `clause` or `contract` depending
on the template):
```js
...
    "accordproject": {
        "template": "clause",
        "cicero": "^0.21.0"
    }
...
```

## Text Changes

Both the markdown for the grammar and sample text have been updated and
consolidated in the `0.21` release and may require some adjustments. You can find
complete information about the latest syntax in the [Markdown Text](markup-
preliminaries) Section of this documentation. For an existing template, you should
apply the following changes.

### Text Grammar Changes

1. Clause or list blocks should have their opening and closing tags on a single
line terminated by whitespace. I.e., you should change occurrences of :
   ```
   {{#clause clauseName}}...clause text...{{/clauseName}}
   ```
   to
   ```
   {{#clause clauseName}}
   ...clause text...
   {{/clauseName}}
   ```
   and similarly for ordered and unordeded list blocks (`olist` and `ulist`).
2. Markdown container blocks are no longer supported inside inline blocks (`if`
`join` and `with` blocks).

:::tip
We recommend using the new [TemplateMark Dingus](https://templatemark-
dingus.netlify.app) to check that your template variables, blocks and formula are
properly identified following the new markdown parsing rules.
:::
~~~

### Text Samples Changes

1. Nested clause template should be now identified within contract templates using clause blocks. I.e., if you use a `paymentClause`, you should change your text from:
   ```
   ...negate the notices Licensor provides and requires hereunder.

   Payment. As consideration in full for the rights granted herein, Licensee shall pay Licensor a one-time fee in the amount of "one hundred US Dollars" (100.0 USD) upon execution of this Agreement, payable as follows: "bank transfer".

   General.
   ...
   ```
   to
   ```
   ...negate the notices Licensor provides and requires hereunder.

   {{#clause paymentClause}}
   Payment. As consideration in full for the rights granted herein, Licensee shall pay Licensor a one-time fee in the amount of "one hundred US Dollars" (100.0 USD) upon execution of this Agreement, payable as follows: "bank transfer".
   {{/clause}}

   General.
   ...
   ```
2. The text corresponding to formulas should be changed from `{{ ...text...}}` to `{{% ...text... %}}`.

You should also ensure that any changes to the grammar text is reflected in the samples. Any change in the grammar text outside of variables should be applied to the corresponding `sample.md` files as well.

:::tip
You can check that the samples and grammar are in agreement by using the `cicero parse` command.
:::

## Model Changes

There should be no model changes required for this version.

## Logic Changes

There should be no logic changes required for this version.

## API Changes

A number of API changes may affect Node.js applications using Cicero or Ergo packages. The main API changes are:
1. Cicero:
   1. `@accordproject/cicero-core` package
      - the `ParserManager` class has been completely overhauled and moved to the `@accordproject/@markdown-template` package.

## CLI Changes

1. The `cicero draft --wrapVariables` option has been removed
2. The `ergo draft` command has been removed

## Cicero Server Changes

Cicero server API has been completely overhauled to match the more recent engine interface
1. The contract data is now passed as part of the REST POST request for the `trigger` endpoint

----
---
id: version-0.21-ref-migrate-concerto-0.82-1.0
title: Concerto 0.82 to 1.0
original_id: ref-migrate-concerto-0.82-1.0

Concerto `1.0` delivers fundamental improvements over previous releases, whilst maintaining a high-degree (though not total!) of backwards compatibility with `0.82`. In particular all of the `0.82` Concerto syntax remains valid in `1.0`.

:::note
We are currently in the process of migrating the Accord Project stack to Concero v1.0. Until the migration is complete Concero v1 is tagged as an `alpha` and may undergo minor additional changes and fixes.
:::

## Summary of Breaking Changes

- Systems models are no longer supported
- DateTime values do not preserve the timezone offset and are always converted to UTC
- Validation has been made stricter, which means some previously allowed instances will now fail validation
- The syntax and semantic of relationships has been changed

## Removal of System Models

See: [#62](https://github.com/accordproject/concerto/issues/62)

An advanced feature of prior versions of Concerto was the ability to add a _system model_ to the `ModelManager`, which functioned as an implicit set of base types for concepts, assets, participants etc within the scope of the `ModelManager`. This feature complicated the APIs considerably, and it had the effect of making CTO models non-portable, in as far as they could only be loaded into a `ModelManager` that used the same set of system models.

System models have therefore been removed from Concerto v1 — any base types should now be imported and referenced explicitly into model files.

## Strict Validation

See: [#157](https://github.com/accordproject/concerto/issues/158) [#158](https://github.com/accordproject/concerto/issues/158)

Prior to Concerto v1 validation suffered from some side-effects of JavaScript type-coercion. For example, `"NaN"` would be a valid value for a `Double` field. These edge cases have now been tightened up, so some instances that were valid prior to v1 may now fail validation.

## Identifiers and Relationships

See: [#181](https://github.com/accordproject/concerto/issues/181)

Prior to v1 a relationship could only be declared to an asset, participant, transaction or event (as these must be `identified by`). In v1 two new capabilities have been added:

1. Concepts can now be declared to be `identified by` an identifying field, allowing them to be used with relationships
2. Any type can be declared as `identified` — to automatically create a system `$identifier` field.

The model below is valid with Concerto v1.

```
namespace org.accordproject

concept Address {
   o String country
}

concept Product identified by sku {
   o String sku
}

concept Order identified {
   o Double ammount
   o Address address
   --> Product product
}
```

## Root of Type Hierarchy

See: [#180](https://github.com/accordproject/concerto/issues/180)

All declared types now have `concerto.Concept` as their super type, and the `concerto` namespace is reserved. Note that the super type for `concerto.Concept` is null.

## Functional API (experimental)

See: [#188](https://github.com/accordproject/concerto/issues/188)

A new streamlined `Concerto` API has been added, to replace some of the existing runtime APIs. Existing runtime APIs have been preserved, but will be progressively removed.

The `Concerto` API is much more functional in nature, and allows plain-old-JavaScript objects to be validated using a `ModelManager` — removing the need to use the `Factory` API to create JS objects prior to validation, or to use the `Serializer` to convert them back to plain-old-JavaScript objects. This should reduce the learning-curve for the library and improve performance.

```
```

~~~~
const { ModelManager, Concerto } = require('@accordproject/concerto-core');
const modelManager = new ModelManager();

modelManager.addModelFile( `namespace org.acme.address
concept PostalAddress {
  o String streetAddress optional
  o String postalCode optional
  o String postOfficeBoxNumber optional
  o String addressRegion optional
  o String addressLocality optional
  o String addressCountry optional
}`, 'model.cto');

const postalAddress = {
   $class : 'org.acme.address.PostalAddress',
   streetAddress : '1 Maine Street'
};

const concerto = new Concerto(modelManager);
concerto.validate(postalAddress);
```

## API Changes

API additions are prefix by a `>` character, while API removals are prefixed by a
`<`.

:::note
A new simplified `Concerto` class has been created to validate JSON data against a
Concerto model. The `Concerto` class wraps a `ModelManager` and allows JS objects
to be validates without using the `Factory` or `Serializer` classes.
:::

```
> class Concerto {
>    + void constructor()
>    + void validate(undefined) throws Error
>    + void getModelManager()
>    + boolean isObject()
>    + void getTypeDeclaration()
>    + string getIdentifier()
>    + boolean isIdentifiable()
>    + boolean isRelationship()
>    + void setIdentifier(string)
>    + string getFullyQualifiedIdentifier()
>    + string toURI()
>    + void fromURI(string) throws Error
>    + string getType()
>    + string getNamespace()
>    + boolean instanceOf(String)
> }
```

:::note
`AssetDeclaration` and other stereotypes now extend `IdentifiedDeclaration` rather
than `ClassDeclaration`. Methods relating to whether the type can be the target of
a relationship have been removed as all types can now be used with relationships,
and methods have been added to denote whether a type has an automatic (system)
identifying field (primary key), no identifying field, or is using an explicitly
~~~~

defined identifying field.
:::

```
< class AssetDeclaration extends ClassDeclaration {
> class AssetDeclaration extends IdentifiedDeclaration {
<    + boolean isRelationshipTarget()
<    + string getSystemType()
<    + boolean isRelationshipTarget()
<    + boolean isSystemRelationshipTarget()
<    + boolean isSystemType()
<    + boolean isSystemCoreType()
>    + Boolean isIdentified()
>    + Boolean isSystemIdentified()
>    + Boolean isExplicitlyIdentified()
```

```
< class EventDeclaration extends ClassDeclaration {
> class EventDeclaration extends IdentifiedDeclaration {
<    + string getSystemType()
```

```
> class IdentifiedDeclaration extends ClassDeclaration {
>    + void constructor(ModelFile,Object) throws IllegalModelException
>    + boolean hasInstance(object)
> }
```

```
< class ParticipantDeclaration extends ClassDeclaration {
> class ParticipantDeclaration extends IdentifiedDeclaration {
<    + boolean isRelationshipTarget()
<    + string getSystemType()
```

```
< class TransactionDeclaration extends ClassDeclaration {
> class TransactionDeclaration extends IdentifiedDeclaration {
<    + string getSystemType()
```

:::note
`ModelFile` has been updated to remove system model files.
:::

```
class ModelFile {
<    + void constructor(ModelManager,string,string,boolean) throws
IllegalModelException
>    + void constructor(ModelManager,string,string) throws IllegalModelException
<    + ClassDeclaration[] getDeclarations(Function,Boolean)
>    + ClassDeclaration[] getDeclarations(Function)
<    + boolean isSystemModelFile()
}
```

:::note

`Concept` has been removed, as all types are now identifiable and now extend
`Resource`.
:::

```
< class Concept extends Typed {
<    + boolean isConcept()
< }
```

:::note
`Resource` has extended to capture that some resources are concepts, and are
identifiable.
:::

```
class Resource extends Identifiable {
>    + boolean isConcept()
>    + boolean isIdentifiable()
}
```

:::note
`ValidatedConcept` has been removed. Users should now call the `validate` method
explicitly to validate data.
:::

```
< class ValidatedConcept extends Concept {
<    + void setPropertyValue(string,string) throws Error
<    + void addArrayValue(string,string) throws Error
<    + void validate() throws Error
< }
```

:::note
`ModelLoader` constructor has been updated to remove system models.
:::

```
class ModelLoader {
<    + object loadModelManager(string,string[],object,boolean)
<    + object
loadModelManagerFromModelFiles(string,object[],undefined,object,boolean)
>    + object loadModelManager(string[],object,boolean)
>    + object loadModelManagerFromModelFiles(object[],undefined,object,boolean)
}
```

:::note
`ModelManager` has been updated to remove system models. These are non-breaking
changes that remove the last argument to method calls.
:::

```
class ModelManager {
<    + Object addModelFile(string,string,boolean,boolean) throws
IllegalModelException
>    + Object addModelFile(string,string,boolean) throws IllegalModelException
<    + Object[] addModelFiles(object[],undefined,boolean,boolean)
>    + Object[] addModelFiles(object[],undefined,boolean)
<    + void writeModelsToFileSystem(String,Object,boolean,boolean)
<    + Object[] getModels(Object,boolean,boolean)
>    + void writeModelsToFileSystem(String,Object,boolean)
>    + Object[] getModels(Object,boolean)
<    + ClassDeclaration[] getSystemTypes()
}
```

---
id: version-0.21-ref-web-components-overview
title: Overview
original_id: ref-web-components-overview

Accord Project publishes [React](https://reactjs.org) user interface components for use in web applications. Please refer to the web-components project [on GitHub](https://github.com/accordproject/web-components) for detailed usage instructions.

You can preview these components in [the project's storybook](https://ap-web-components.netlify.app/).

## Contract Editor

The Contract Editor component provides a rich-text content editor for contract text with embedded clauses.

> Note that the contract editor does not currently support the full expressiveness of Cicero Templates. Please refer to the Limitations section for details.

### Limitations

The contract editor does not support templates which use the following CiceroMark features:

* Lists containing [nested lists](markup-commonmark#nested-lists)
* Templates [list blocks](markup-blocks#list-blocks)

## Error Logger

The Error Logger component is used to display structured error messages from the Contract Editor.

## Navigation

The Navigation component displays an outline view for a contract, allowing the user to quickly navigate between sections.

## Library

The Library component displays a vertical list of library item metadata, and allows the user to add an instance of a library item to a contract.

---
id: version-0.21-started-hello

title: Hello World Template
original_id: started-hello

Once you have installed Cicero, you can try it on an existing Accord Project template. This explains how to create an instance of that template and how to run the contract logic.

## Download a Template

You can download a single clause or contract template from the [Accord Project Template Library](https://templates.accordproject.org) as an archive (`.cta`) file. Cicero archives are files with a `.cta` extension, which includes all the different components for the template (text, model and logic).

If you click on the Template Library link, you should see a Web Page which looks as follows:

![Basic-Use-1](/docs/assets/basic/use1.png)

Scrolling down that page, you can see the index for the open-source templates along with their version, and whether they are a Clause or Contract template.

Click on the link to the `helloworld` template. You should be taken to a page which looks as follows:

![Basic-Use-2](/docs/assets/basic/use2.png)

Then click on the `Download Archive` button under the description for the template (highlighted in the red box in the figure). This should download the latest template archive for the `helloworld` template.

## Parse: Extract Deal Data from Text

You can use Cicero to extract deal data from a contract text using the `cicero parse` command.

### Parse Valid Text

Using your terminal, change into the directory (or `cd` into the directory) that contains the template archive you just downloaded, then create a sample clause text `sample.md` which contains the following text:

```md
Name of the person to greet: "Fred Blogs".
Thank you!
```

Then run the `cicero parse` command in your terminal to load the template and parse your sample clause text. This should be echoing the result of parsing back to your terminal.

```bash
cicero parse --template helloworld@0.13.0.cta --sample sample.md
```

:::note
* Templates are tied to a specific version of the cicero tool. Make sure that the version number output from `cicero --version` is compatible with the template. Look

for `^0.21.0` or similar at the top of the template web page.
* `cicero parse` requires network access. Make sure that you are online and that your firewall or proxy allows access to `https://models.accordproject.org`
:::

This should extract the data (or "deal points") from the text and output:

```json
{
  "$class": "org.accordproject.helloworld.HelloWorldClause",
  "clauseId": "aa3b9db9-f25f-41f4-88a4-64baba728bfe",
  "name": "Fred Blogs"
}
```

You can save the result of `cicero parse` into a file using the `--output` option:
```
cicero parse --template helloworld@0.13.0.cta --sample sample.md --output data.json
```

### Parse Non-Valid Text

If you attempt to parse text which is not valid according to the template, this same command should return an error.

Edit your `sample.md` file to add text that is not consistent with the template:

```text
FUBAR Name of the person to greet: "Fred Blogs".
Thank you!
```

Then run `cicero parse --template helloworld@0.13.0.cta --sample sample.md` again. The output should now be:

```text
2:13:15 AM - error: Parse error at line 1 column 1
FUBAR Name of the person to greet: "Fred Blogs".
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Expected: 'Name of the person to greet: '
```

## Draft: Create Text from Deal Data

You can use Cicero to create new contract text from deal data using the `cicero draft` command.

### Draft from Valid Data

If you have saved the deal data earlier in a `data.json` file, you can edit it to change the name from `Fred Blogs` to `John Doe`, or create a brand new `data.json` file containing:
```json
{
  "$class": "org.accordproject.helloworld.HelloWorldClause",
  "clauseId": "aa3b9db9-f25f-41f4-88a4-64baba728bfe",
  "name": "John Doe"
}
```

Then run the `cicero draft` command in your terminal:
```
cicero draft --template helloworld@0.13.0.cta --data data.json
```

This should create a new contract text and output:
```
13:17:18 - info: Name of the person to greet: "John Doe".
Thank you!
```

You can save the result of `cicero draft` into a file using the `--output` option:
```
cicero draft --template helloworld@0.13.0.cta --data data.json --output new-
sample.md
```

### Draft from Non-Valid Data

If you attempt to draft from data which is not valid according to the template, this same command should return an error.

Edit your `data.json` file so that the `name` variable is missing:
```json
{
  "$class": "org.accordproject.helloworld.HelloWorldClause",
  "clauseId": "aa3b9db9-f25f-41f4-88a4-64baba728bfe"
}
```

Then run `cicero draft --template helloworld@0.13.0.cta --data data.json` again. The output should now be:
```
13:38:11 - error: Instance org.accordproject.helloworld.HelloWorldClause#6f91e060-
f837-4108-bead-63891a91ce3a missing required field name
```

## Trigger: Run the Contract Logic

You can use Cicero to run the logic associated to a contract using the `cicero trigger` command.

### Trigger with a Valid Request

Use the `cicero trigger` command to parse a clause text based (your `sample.md`) *then* send a request to the clause logic.

To do so, you first create one additional file `request.json` which contains:
```json
{
  "$class": "org.accordproject.helloworld.MyRequest",
  "input": "Accord Project"
}
```

This is the request which you will send to trigger the execution of your contract.

Then run the `cicero trigger` command in your terminal to load the template, parse your clause text *and* send the request. This should be echoing the result of execution back to your terminal.

```bash
cicero trigger --template helloworld@0.13.0.cta --sample sample.md --request
request.json
```

This should print this output:

```json
13:42:29 - info:
{
  "clause": "helloworld@0.13.0-
c03393f7e50865012e6005050fcaccb2716481fa7599905f7306673cf15857cf",
  "request": {
    "$class": "org.accordproject.helloworld.MyRequest",
    "input": "Accord Project"
  },
  "response": {
    "$class": "org.accordproject.helloworld.MyResponse",
    "output": "Hello Fred Blogs Accord Project",
    "transactionId": "ecc56a61-713c-4113-9842-550efb09ac74",
    "timestamp": "2019-11-03T18:42:29.984Z"
  },
  "state": {
    "$class": "org.accordproject.cicero.contract.AccordContractState",
    "stateId": "org.accordproject.cicero.contract.AccordContractState#1"
  },
  "emit": []
}
```

The results of execution displayed back on your terminal is in JSON format. It includes the following information:

* Details of the `clause` being triggered (name, version, SHA256 hash of the template)
* The incoming `request` object (the same request from your `request.json` file)
* The output `response` object
* The output `state` (unchanged in this example)
* An array of `emit`ted events (empty in this example)

That's it! You have successfully parsed and executed your first Accord Project Clause using the `helloworld` template.

### Trigger with a Non-Valid Request

If you attempt to trigger the contract from a request which is not valid according to the template, this same command should return an error.

Edit your `request.json` file so that the `input` variable is missing:
```json
{
  "$class": "org.accordproject.helloworld.MyRequest"
}
```

Then run `cicero trigger --template helloworld@0.13.0.cta --sample sample.md --

request request.json` again. The output should now be:
```
13:47:35 - error: Instance org.accordproject.helloworld.MyRequest#b0b1cbcc-dcae-4758-b9fc-254a43aa10a8 missing required field input
```

## What Next?

### Try Other Templates

Feel free to try the same commands to parse and execute other templates from the Accord Project Library. Note that for each template, you can find samples for the text, for the request and for the state on the corresponding Web page. For instance, a sample for the [Late Delivery And Penalty](https://templates.accordproject.org/latedeliveryandpenalty@0.15.0.html) clause is in the red box in the following image:

![Basic-Use-3](/docs/assets/basic/use3.png)

### More About Cicero

You can find more information on how to create or publish Accord Project templates in the [Work with Cicero](tutorial-templates) tutorials.

### Run on Different Platforms

Templates may be executed on different platforms, not only from the command line. You can find more information on how to execute Accord Project templates on different platforms (Node.js, Hyperledger Fabric, etc.) in the [Template Execution](tutorial-nodejs) tutorials.

---
id: version-0.21-started-installation
title: Install Cicero
original_id: started-installation

To experiment with Accord Project, you can install the Cicero command-line. This will let you author, validate, and run Accord Project templates on your own machine.

## Prerequisites

You must first obtain and configure the following dependency:

* [Node.js (LTS)](http://nodejs.org): We use Node.js to run cicero, generate the documentation, run a development web server, testing, and produce distributable files. Depending on your system, you can install Node either from source or as a pre-packaged bundle.

> We recommend using [nvm](https://github.com/creationix/nvm) (or [nvm-windows](https://github.com/coreybutler/nvm-windows)) to manage and install Node.js, which makes it easy to change the version of Node.js per project.

## Installing Cicero

To install the latest version of the Cicero command-line tools:

```bash
npm install -g @accordproject/cicero-cli
```

:::note
You can install a specific version by appending `@version` at the end of the `npm install` command. For instance to install version `0.20.3` or version `0.13.4`:
```bash
npm install -g @accordproject/cicero-cli@0.20.3
npm install -g @accordproject/cicero-cli@0.13.4
```
:::

To check that Cicero has been properly installed, and display the version number:
```bash
cicero --version
```

To get command line help:
```bash
cicero --help
cicero parse --help     // To parse a sample clause/contract
cicero draft --help     // To draft a sample clause/contract
cicero trigger --help   // To send a request to a clause/contract
```

## Optional Packages

### Template Generator

You may also want to install the template generator tool, which you can use to create an empty template:

```bash
npm install -g yo
npm install -g @accordproject/generator-cicero-template
```

## What next?

That's it! Go to the next page to see how to use your new installation of Cicero on a real Accord Project template.

---
id: version-0.21-started-resources
title: Resources
original_id: started-resources

## Accord Project Resources

- The Main Web site includes latest news, links to working groups, organizational announcements, etc. : https://www.accordproject.org
- This Technical Documentation: https://docs.accordproject.org
- Recording of Working Group discussions, Tutorial Videos are available on Vimeo: https://vimeo.com/accordproject

- Join the [Accord Project Slack](https://accord-project-slack-signup.herokuapp.com) to get involved!

## User Content

Accord Project also maintains libraries containing open source, community-contributed content to help you when authoring your own templates:

- [Model Repository](https://models.accordproject.org/) : a repository of open source Concerto data models for use in templates
- [Template Library](https://templates.accordproject.org/) : a library of open source Clause and Contract templates for various legal domains (supply-chain, loans, intellectual property, etc.)

## Ecosystem & Tools

Accord Project is also developing tools to help with authoring, testing and running accord project templates.

### Editors

- [Template Studio](https://studio.accordproject.org/): a Web-based editor for Accord Project templates
- [VSCode Extension](https://marketplace.visualstudio.com/items?itemName=accordproject.cicero-vscode-extension): an Accord Project extension to the popular [Visual Studio Code](https://visualstudio.microsoft.com/) Editor
- [Emacs Mode](https://github.com/accordproject/ergo/tree/master/ergo.emacs): Emacs Major mode for Ergo (alpha)
- [VIM Plugin](https://github.com/accordproject/ergo/tree/master/ergo.vim): VIM plugin for Ergo (alpha)

### User Interface Components

- [Markdown Editor](https://github.com/accordproject/markdown-editor): a general purpose react component for markdown rendering and editing
- [Cicero UI](https://github.com/accordproject/cicero-ui): a library of react components for visualizing, creating and editing Accord Project templates

## Developers Resources

All the Accord Project technology is being developed as open source. The software packages are being actively maintained on [GitHub](https://github.com/accordproject) and we encourage organizations and individuals to contribute requirements, documentation, issues, new templates, and code.

Join us on the [#technology-wg Slack channel](https://accord-project-slack-signup.herokuapp.com) for technical discussions and weekly updates.

### Cicero

- GitHub: https://github.com/accordproject/cicero
- [Cicero Slack Channel](https://accord-project.slack.com/messages/CA08NAHQS/details/)
- Technical Questions and Answers on [Stack Overflow](https://stackoverflow.com/questions/tagged/cicero)

### Ergo

- GitHub: https://github.com/accordproject/ergo
- The [Ergo Language Guide](logic-ergo) is a good place to get started with Ergo.
- [Ergo Slack
Channel](https://accord-project.slack.com/messages/C9HLJHREG/details/)

---
id: version-0.21-tutorial-create
title: Template Generator
original_id: tutorial-create

Now that you have executed an existing template, let's create a new template from scratch. To facilitate the creation of new templates, Cicero comes with a template generator.

## The template generator

### Install the generator

If you haven't already done so, first install the template generator::

```bash
npm install -g yo
npm install -g yo @accordproject/generator-cicero-template
```

### Run the generator:

You can now try the template generator by running the following command in a terminal window:
```bash
yo @accordproject/cicero-template
```
This will ask you a series of questions. Give your generator a name (no spaces) and then supply a namespace for your template model (again,no spaces). The generator will then create the files and directories required for a basic template (similar to the helloworld template).

Here is an example of how it should look like in your terminal window:
```bash
bash-3.2$ yo @accordproject/cicero-template

     _-----_     ╭──────────────────────────╮
    |       |    │      Welcome to the      │
    |--(o)--|    │ generator-cicero-templat │
   `---------´   │       e generator!       │
    ( _´U`_ )    ╰──────────────────────────╯
    /___A___\   /
     |  ~  |
   __'.___.'__
 ´   `  |° ´ Y `

? What is the name of your template? mylease
? Who is the author? me
? What is the namespace for your model? org.acme.lease
   create mylease/README.md
   create mylease/logo.png
   create mylease/package.json
```

~~~~
   create mylease/request.json
   create mylease/logic/logic.ergo
   create mylease/model/model.cto
   create mylease/test/logic_default.feature
   create mylease/text/grammar.tem.md
   create mylease/text/sample.md
   create mylease/.cucumber.js
   create mylease/.npmignore
bash-3.2$
```

:::tip
You may find it easier to edit the grammar, model and logic for your template in
[VSCode](https://code.visualstudio.com/), installing the [Accord Project extension]
(https://marketplace.visualstudio.com/items?itemName=accordproject.cicero-vscode-
extension). The extension gives you syntax highlighting and parser errors within VS
Code.

For more information on how to use VS Code with the Accord Project extension,
please consult the [Using the VS Code extension](tutorial-vscode) tutorial.
:::

## Test your template

If you have Cicero installed on your machine, you can go into the newly created
`mylease` directory and try it with cicero, to make sure the contract text parses:
```bash
bash-3.2$ cicero parse
11:51:40 AM - info: Using current directory as template folder
11:51:40 AM - info: Loading a default text/sample.md file.
11:51:41 AM - info:
{
  "$class": "org.acme.lease.MyContract",
  "name": "Dan",
  "contractId": "635633f9-e188-4d79-a867-6850d8ad6c66"
}
```
And that you can trigger the contract:
```bash
bash-3.2$ cicero trigger
11:58:22 AM - info: Using current directory as template folder
11:58:22 AM - info: Loading a default text/sample.md file.
11:58:22 AM - info: Loading a default request.json file.
11:58:23 AM - warn: A state file was not provided, initializing state. Try the --
state flag or create a state.json in the root folder of your template.
11:58:23 AM - info:
{
  "clause": "mylease@0.0.0-
db65db8a6022ef8dbbc25f2fd9fdc2778596d8ff3473a33c0dab66ae76f1d86e",
  "request": {
    "$class": "org.acme.lease.MyRequest",
    "input": "World"
  },
  "response": {
    "$class": "org.acme.lease.MyResponse",
    "output": "Hello Dan World",
    "transactionId": "c5ed5a39-5fd3-4013-b53f-bdd46bd96406",
    "timestamp": "2020-09-22T15:58:23.798Z"
  },
~~~~

  "state": {
    "$class": "org.accordproject.cicero.contract.AccordContractState",
    "stateId": "org.accordproject.cicero.contract.AccordContractState#1"
  },
  "emit": []
}

The template also comes with a few simple tests which you can run by first doing an `npm install` in the template directory, then by running `npm run test`:
```bash
bash-3.2$ npm install
bash-3.2$ npm run test

> mylease@0.0.0 test /Users/jeromesimeon/tmp/mylease
> cucumber-js test -r .cucumber.js

....

1 scenario (1 passed)
3 steps (3 passed)
0m01.257s
bash-3.2$
```

## Edit your template

### Update Sample.md

First, replace the contents of `./text/sample.md` with the legal text for the contract or clause that you would like to digitize.

Check that when you run `cicero parse` that the new `./text/sample.md` is now _invalid_ with respect to the grammar.

### Edit the Template Grammar

Now update the grammar in `./text/grammar.tem.md`. Start by replacing the existing grammar, making it identical to the contents of your updated `./text/sample.md`.

Now introduce variables into your template grammar as required. The variables are marked-up using `{{` and `}}` with what is between the braces being the name of your variable.

### Edit the Template Model

All of the variables referenced in your template grammar must exist in your template model. Edit
the file `model/model.cto` to include all your variables, making sure the name of the model property matches the name of the variable in the `./text/grammar.tem.md` file.

Note that the Concerto Modeling Language primitive data types are:

- `String`: for character strings
- `Long` or `Integer`: for integer values
- `DateTime`: for dates and times
- `Double`: for floating points numbers
- `Boolean`: for values that are either true or false

:::tip
Note that you can import common types (address, monetary amount, country code,
etc.) from the Accord Project Model Repository: https://models.accordproject.org.
:::

### Edit the Transaction Types

Your template expects to receive data as input and will produce data as output. The
structure of
this request/response data is captured in the `MyRequest` and `MyResponse`
transaction types in your model
namespace. Open up the file `models/model.cto` and edit the definition of the
`MyRequest` type to
include all the data you expect to receive from the outside world and that will be
used by the
business logic of your template. Similarly edit the definition of the `MyResponse`
type to include
all the data that the business logic for your template will compute and would like
to return to the
caller.

### Edit the Template Logic

Now edit the business logic of the template itself. This is expressed in the Ergo
language, which is a strongly-typed function domain specific language for contract
logic. Open the file `logic/logic.ergo`
and edit the `helloworld` clause to perform the calculations your logic requires.

Looking at the Ergo logic for other example templates will help you understand the
syntax and capabilities of Ergo.

## Publishing your template

If you would like to publish your new template in the Accord Project Template
Library, please consult the [Template Library](tutorial-library) Section of this
documentation.

---
id: version-0.21-tutorial-hyperledger
title: With Hyperledger Fabric
original_id: tutorial-hyperledger

## Hyperledger Fabric 2.2

Sample chaincode for Hyperledger Fabric that shows how to execute a Cicero
template:
https://github.com/accordproject/hlf-cicero-contract

Please refer to the project README for detailed instructions on installation and
submitting transactions.

---
id: version-0.21-tutorial-library

title: Template Library
original_id: tutorial-library

This tutorial explains how to get access, and contribute, to all of the public templates available as part of the the [Accord Project Template Library](https://templates.accordproject.org).

## Setting up

### Prerequisites

Accord Project uses [GitHub](https://github.com/) to maintain its open source template library. For this tutorial, you must first obtain and configure the following dependency:

* [Git](https://git-scm.com): a distributed version-control system for
  tracking changes in source code during software development.
* [Lerna](https://lerna.js.org/): A tool for managing JavaScript projects with multiple packages. You can install lerna by running the following command in your terminal:

```bash
npm install -g lerna@^3.15.0
```

### Clone the template library

Once you have `git` installed on your machine, you can run `git clone` to create a version of all the templates:

```bash
git clone https://github.com/accordproject/cicero-template-library
```

Alternatively, you can download the library directly by visiting the [GitHub Repository for the Template Library](https://github.com/accordproject/cicero-template-library) and use the "Download" button as shown on this snapshot:

![Basic-Library-1](/docs/assets/basic/library1.png)

### Install the Library

Once cloned, you can set up the library for development by running the following commands inside your template library directory:

```bash
lerna bootstrap
```

### Running all the template tests

To check that the installation was successful, you can run all the tests for all the Accord Project templates by running:

```bash
lerna run test
```

## Structure of the Repository

You can see the source code for all public Accord Project templates by looking inside the `./src` directory:

```sh
bash-3.2$ ls src
acceptance-of-delivery
car-rental-tr
certificate-of-incorporation
copyright-license
demandforecast
docusign-connect
docusign-po-failure
eat-apples
empty
empty-contract
fixed-interests
...
```

Each of those templates directories have the same structure, as described in the [Templates Deep Dive](tutorial-templates) Section. For instance for the `acceptance-of-delivery` template:
```
$ cd src/acceptance-of-delivery
$ bash-3.2$ ls -laR
./README.md
./package.json

./text:
  ./grammar.tem.md
  ./sample.md

./logic:
   logic.ergo

./model:
   model.cto

./test:
  logic.feature
  logic_default.feature

./request.json
./state.json
```

## Use a Template

To use a template, simply run the same Cicero commands we have seen in the previous tutorials. For instance, to extract the deal data from the `./text/sample.md` text sample for the `acceptance-of-delivery` template, run:

```bash
cicero parse --template ./src/acceptance-of-delivery
```
You should see a response as follows:
```json
```

```json
{
  "$class": "org.accordproject.acceptanceofdelivery.AcceptanceOfDeliveryClause",
  "clauseId": "9ed9d255-3bb6-4928-be7b-c6305e083246",
  "shipper": "Party A",
  "receiver": "Party B",
  "deliverable": "Widgets",
  "businessDays": 10,
  "attachment": "Attachment X"
}
```

Or, to extract the deal data from the `./text/sample.md` then send the default request in `./request.json` for the `latedeliveryandpenalty` template, run:
```bash
cicero trigger --template ./src/latedeliveryandpenalty
```
You should see a response as follows:

```json
{
  "clause": "latedeliveryandpenalty@0.16.0-
f4070225d9792aa6494b2ea1f0ffe7a794f8c671977d43fa25c75e83b3eacc3d",
  "request": {
    "$class":
"org.accordproject.latedeliveryandpenalty.LateDeliveryAndPenaltyRequest",
    "forceMajeure": false,
    "agreedDelivery": "2017-12-17T03:24:00-05:00",
    "deliveredAt": null,
    "goodsValue": 200
  },
  "response": {
    "$class":
"org.accordproject.latedeliveryandpenalty.LateDeliveryAndPenaltyResponse",
    "penalty": 110.00000000000001,
    "buyerMayTerminate": true,
    "transactionId": "cca97517-21e8-46b4-8524-e9523d10b6bc",
    "timestamp": "2020-09-22T15:42:27.464Z"
  },
  "state": {
    "$class": "org.accordproject.cicero.contract.AccordContractState",
    "stateId": "org.accordproject.cicero.contract.AccordContractState#1"
  },
  "emit": [
    {
      "$class": "org.accordproject.cicero.runtime.PaymentObligation",
      "amount": {
        "$class": "org.accordproject.money.MonetaryAmount",
        "doubleValue": 110.00000000000001,
        "currencyCode": "USD"
      },
      "description": "Dan should pay penalty amount to Steve",
      "contract":
"resource:org.accordproject.latedeliveryandpenalty.LateDeliveryAndPenaltyContract#1
99b219e-783f-4451-8992-2e5605310d6d",
      "promisor": "resource:org.accordproject.cicero.contract.AccordParty#Dan",
      "promisee": "resource:org.accordproject.cicero.contract.AccordParty#Steve",
      "eventId": "valid",
      "timestamp": "2020-09-22T15:42:27.465Z"
    }
  ]
}
```

## Contribute a New Template

To contribute a change to the Accord Project library, please
[fork](https://help.github.com/en/github/getting-started-with-github/fork-a-repo)
the repository and then create a [pull
request](https://help.github.com/en/github/collaborating-with-issues-and-pull-requests/about-pull-requests).

Note that templates should have unit tests. See any of the `./test` directories in
the templates contained in the template library for an examples with unit tests, or
consult the [Testing Reference](ref-testing) Section of this documentation.

---
id: version-0.21-tutorial-nodejs
title: With Node.js
original_id: tutorial-nodejs

## Cicero Node.js API

You can work with Accord Project templates directly in JavaScript using Node.js.

Documentation for the API can be found in [Cicero API](ref-cicero-api.html).

## Working with Templates

### Import the Template class

To import the Cicero class for templates:

```js
const Template = require('@accordproject/cicero-core').Template;
```

### Load a Template

To create a Template instance in memory call the `fromDirectory`, `fromArchive` or
`fromUrl` methods:

```js
    const template = await
Template.fromDirectory('./test/data/latedeliveryandpenalty');
```

These methods are asynchronous and return a `Promise`, so you should use `await` to
wait for the promise to be resolved.

### Instantiate a Template

Once a Template has been loaded, you can create a Clause based on the Template. You
can either instantiate
the Clause using source DSL text (by calling `parse`), or you can set an instance
of the template model

as JSON data (by calling `setData`):

```js
    // load the DSL text for the template
    const testLatePenaltyInput = fs.readFileSync(path.resolve(__dirname, 'text/',
'sample.md'), 'utf8');

    const clause = new Clause(template);
    clause.parse(testLatePenaltyInput);

    // get the JSON object created from the parse
    const data = clause.getData();
```

OR - create a contract and set the data from a JSON object.

```js
    const clause = new Clause(template);
    clause.setData( {$class: 'org.acme.MyTemplateModel', 'foo': 42 } );
```

## Executing a Template Instance

Once you have instantiated a clause or contract instance, you can execute it.

### Import the Engine class

To execute a Clause you first need to create an instance of the ``Engine`` class:

```js
const Engine = require('@accordproject/cicero-engine').Engine;
```

### Send a request to the contract

You can then call ``execute`` on it, passing in the clause or contract instance,
and the request:

```js
    const request = {
        '$class':
'org.accordproject.latedeliveryandpenalty.LateDeliveryAndPenaltyRequest',
        'forceMajeure': false,
        'agreedDelivery': '2017-10-07T16:38:01.412Z',
        'goodsValue': 200,
        'transactionId': '402c8f50-9e61-433e-a7c1-afe61c06ef00',
        'timestamp': '2017-11-12T17:38:01.412Z'
    };
    const state = {};
    state.$class = 'org.accordproject.cicero.contract.AccordContractState';
    state.stateId = 'org.accordproject.cicero.contract.AccordContractState#1';
    const result = await engine.execute(clause, request, state);
```

---
id: version-0.21-tutorial-studio
title: With Template Studio
original_id: tutorial-studio

This tutorial will walk you through the steps of editing a clause template in [Template Studio](https://studio.accordproject.org/).

We start with a very simple _Late Penalty and Delivery_ Clause and gradually make it more complex, adding both legal text to it and the corresponding business logic in Ergo.

## Initial Late Delivery Clause

### Load the Template

To get started, head to the `minilatedeliveryandpenalty` template in the Accord Project Template Library at [Mini Late Delivery And Penalty](https://templates.accordproject.org/minilatedeliveryandpenalty@0.5.0.html) and click the "Open In Template Studio" button.

![Advanced-Late-1](assets/advanced/late1.png)

Begin by inspecting the `README` and `package.json` tabs within the `Metadata` section. Feel free to change the name of the template to one you like.

### The Contract Text

Then click on the `Text` Section on the left, which should show a `Grammar` tab, for the the natural language of the template.

![Advanced-Late-2](assets/advanced/late2.png)

When the text in the `Grammar` tab is in sync with the text in the `Sample` tab, this means the sample is a valid with respect to the grammar, and data is extracted, showing in `Contract Data` tab. The contract data is represented using the JSON format and contains the value of the variables declared in the contract template. For instance, the value for the `buyer` variable is `Betty Buyer`, highlighted in red:

![Advanced-Late-3](assets/advanced/late3.png)

Changes to the variables in the `Sample` are reflected in the `Contract Data` tab in real time, and vice versa. For instance, change `Betty Buyer` to a different name in the contract text to see the `partyId` change in the contract data.

If you edit part of the text which is not a variable in the template, this results in an error when parsing the `Sample`. The error will be shown in red in the status bar at the bottom of the page. For instance, the following image shows the parsing error obtained when changing the word `delayed` to the word `timely` in the contract text.

![Advanced-Late-4](assets/advanced/late4.png)

This is because the `Sample` relies on the `Grammar` text as a source of truth. This mechanism ensures that the actual contract always reflects the template, and remains faithful to the original legal text. You can, however, edit the `Grammar` itself to change the legal text.

Revert your changes, changing the word `timely` back to the original word `delayed` and the parsing error will disappear.

### The Model

Moving along to the `Model` section, you will find the data model for the template variables (the `MiniLateDeliveryClause` type), as well as for the requests (the `LateRequest` type) and response (the `LateResponse` type) for the late delivery and penalty clause.

![Advanced-Late-5](assets/advanced/late5.png)

Note that a `namespace` is declared at the beginning of the file for the model, and that several existing models are being imported (using e.g., `import org.accordproject.cicero.contract.*`). Those imports are needed to access the definition for several types used in the model:
- `AccordClause` which is a generic type for all Accord Project clause templates, and is defined in the `org.accordproject.contract` namespace;
- `Request` and `Response` which are generic types for responses and requests, and are defined in the `org.accordproject.runtime` namespace;
- `Duration` which is defined in the `org.accordproject.time` namespace.

### The Logic

The final part of the template is the `Ergo` tab of the `Logic` section, which describes the business logic.

![Advanced-Late-6](assets/advanced/late6.png)

Thanks to the `namespace` at the beginning of this file, the Ergo engine can know the definition for the `MiniLateDeliveryClause`, as well as the `LateRequest`, and `LateResponse` types defined in the `Model` tab.

To test the template execution, go to the `Request` tab in the `Logic` section. It should be already populated with a valid request. Press the `Trigger` button to trigger the clause.

![Advanced-Late-7](assets/advanced/late7.png)

Since the value of the `deliveredAt` parameter in the request is after the value of the `agreedDelivery` parameter in the request, this should return a new response which includes the calculated penalty.

Changing the date for the `deliveredAt` parameter in the request and triggering the contract again will result in a different penalty.

![Advanced-Late-8](assets/advanced/late8.png)

Note that the clause will return an error if it is called for a timely delivery.

![Advanced-Late-9](assets/advanced/late9.png)

## Add a Penalty Cap

We can now start building a more advanced clause. Let us first take a moment to notice that there is no limitation to the penalty resulting from a late delivery. Trigger the contract using the following request in the `Request` tab in `Logic`:
```json
{
  "$class": "org.accordproject.minilatedeliveryandpenalty.LateRequest",
  "agreedDelivery": "2019-04-10T12:00:00-05:00",
  "deliveredAt": "2019-04-20T03:24:00-05:00",
```

  "goodsValue": 200
}
~~~
The penalty should be rather low. Now send this other request:
```json
{
  "$class": "org.accordproject.minilatedeliveryandpenalty.LateRequest",
  "agreedDelivery": "2005-04-01T12:00:00-05:00",
  "deliveredAt": "2019-04-20T03:24:00-05:00",
  "goodsValue": 200
}
~~~
Notice that the penalty is now quite a large value. It is not unusual to cap a penalty to a maximum amount. Let us now look at how to change the template to add such a cap based on a percentage of the total value of the delivered goods.

### Update the Legal Text

To implement this, we first go to the `Grammar` tab in the `Text` section and add a sentence indicating: `The total amount of penalty shall not, however, exceed {{capPercentage}}% of the total value of the delayed goods.`

For convenience, you can copy-paste the new template text from here:
```tem
Late Delivery and Penalty.

In case of delayed delivery of Goods, {{seller}} shall pay to
{{buyer}} a penalty amounting to {{penaltyPercentage}}% of the total
value of the Goods for every {{penaltyDuration}} of delay. The total
amount of penalty shall not, however, exceed {{capPercentage}}% of the
total value of the delayed goods. If the delay is more than
{{maximumDelay}}, the Buyer is entitled to terminate this Contract.

```
This should immediately result in an error when parsing the contract text:

![Advanced-Late-10](assets/advanced/late10.png)

As explained in the error message, this is because the new template text uses a variable `capPercentage` which has not been declared in the model.

### Update the Model

To define this new variable, go to the `Model` tab, and change the `MiniLateDeliveryClause` type to include `o Double capPercentage`.

![Advanced-Late-11](assets/advanced/late11.png)

For convenience, you can copy-paste the new `MiniLateDeliveryClause` type from here:
```ergo
asset MiniLateDeliveryClause extends AccordClause {
  o AccordParty buyer         // Party to the contract (buyer)
  o AccordParty seller        // Party to the contract (seller)
  o Duration penaltyDuration  // Length of time resulting in penalty
  o Double penaltyPercentage  // Penalty percentage
  o Double capPercentage      // Maximum penalty percentage
  o Duration maximumDelay     // Maximum delay before termination
}
```

This results in a new error, this time on the sample contract:

![Advanced-Late-12](assets/advanced/late12.png)

To fix it, we need to add that same line we added to the template, replacing the
`capPercentage` by a value in the `Test Contract`: `The total amount of penalty
shall not, however, exceed 52% of the total value of the delayed goods.`

For convenience, you can copy-paste the new test contract from here:
```md
Late Delivery and Penalty.

In case of delayed delivery of Goods, "Steve Seller" shall pay to
"Betty Buyer" a penalty amounting to 10.5% of the total
value of the Goods for every 2 days of delay. The total
amount of penalty shall not, however, exceed 52% of the
total value of the delayed goods. If the delay is more than
15 days, the Buyer is entitled to terminate this Contract.

```

Great, now the edited template should have no more errors, and the contract data
should now include the value for the new `capPercentage` variable.

![Advanced-Late-13](assets/advanced/late13.png)

Note that the `Current Template` Tab indicates that the template has been changed.

### Update the Logic

At this point, executing the logic will still result in large penalties. This is
because the logic does not take advantage of the new `capPercentage` variable. Edit
the `logic.ergo` code to do so. After step `// 2. Penalty formula` in the logic,
apply the penalty cap by adding some logic as follows:
```ergo
    // 3. Capped Penalty
    let cap = contract.capPercentage / 100.0 * request.goodsValue;

    let cappedPenalty =
      if penalty > cap
      then cap
      else penalty;

```
Do not forget to also change the value of the penalty in the returned
`LateResponse` to use the new variable `cappedPenalty`:
```ergo
    // 5. Return the response
    return LateResponse{
      penalty: cappedPenalty,
      buyerMayTerminate: termination
    }
```
The logic should now look as follows:

![Advanced-Late-14](assets/advanced/late14.png)

### Run the new Logic

As a final test of the new template, you should try again to run the contract with
a long delay in delivery. This should now result in a much smaller penalty, which
is capped to 52% of the total value of the goods, or 104 USD.

![Advanced-Late-15](assets/advanced/late15.png)

:::tip
A full version of the template after those changes have been applied can be found
as the [Mini Late Delivery And Penalty
Capped](https://templates.accordproject.org/minilatedeliveryandpenalty-
capped@0.5.0.html) in the Template Library.
:::

## Emit a Payment Obligation.

As a final extension to this template, we can modify it to emit a Payment
Obligation. This first requires us to switch from a Clause template to a Contract
template.

### Switch to a Contract Template

The first place to change is in the metadata for the template. This can be done
easily with the `full contract` button in the `Current Template` tab. This will
immediately result in an error indicating that the model does not contain an
`AccordContract` type.

![Advanced-Late-16](assets/advanced/late16.png)

### Update the Model

To fix this, change the model to reflect that we are now editing a contract
template, and change the type `AccordClause` to `AccordContract` in the type
definition for the template variables:
```ergo
asset MiniLateDeliveryContract extends AccordContract {
  o AccordParty buyer         // Party to the contract (buyer)
  o AccordParty seller        // Party to the contract (seller)
  o Duration penaltyDuration  // Length of time resulting in penalty
  o Double penaltyPercentage  // Penalty percentage
  o Double capPercentage      // Maximum penalty percentage
  o Duration maximumDelay     // Maximum delay before termination
}
```

The next error is in the logic, since it still uses the old
`MiniLateDeliveryClause` type which does not exist anymore.

### Update the Logic

The `Logic` error that occurs here is:
```bash
Compilation error (at file lib/logic.ergo line 19 col 31). Cannot find type with
name 'MiniLateDeliveryClause'
contract MiniLateDelivery over MiniLateDeliveryClause {
                               ^^^^^^^^^^^^^^^^^^^^^^
```
Update the logic to use the the new `MiniLateDeliveryContract` type instead, as

follows:
```ergo
contract MiniLateDelivery over MiniLateDeliveryContract {
```

The template should now be without errors.

### Add a Payment Obligation

Our final task is to emit a `PaymentObligation` to indicate that the buyer should pay the seller in the amount of the calculated penalty.

To do so, first import a couple of standard models: for the Cicero's [runtime model](https://models.accordproject.org/cicero/runtime.html) (which contains the definition of a `PaymentObligation`), and for the Accord Project's [money model](https://models.accordproject.org/money.html) (which contains the definition of a `MonetaryAmount`). The `import` statements at the top of your logic should look as follows:
```ergo
import org.accordproject.time.*
import org.accordproject.cicero.runtime.*
import org.accordproject.money.MonetaryAmount

```

Lastly, add a new step between steps `// 4.` and `// 5.` in the logic to emit a payment obligation in USD:
```ergo
    emit PaymentObligation{
      contract: contract,
      promisor: some(contract.seller),
      promisee: some(contract.buyer),
      deadline: none,
      amount: MonetaryAmount{ doubleValue: cappedPenalty, currencyCode: USD },
      description: contract.seller.partyId ++ " should pay penalty amount to " ++
contract.buyer.partyId
    };

```
That's it! You can observe in the `Request` tab that an `Obligation` is now being emitted. Try out adjusting values and continuing to send requests and getting responses and obligations.

![Advanced-Late-17](assets/advanced/late17.png)

:::tip
A full version of the template after those changes have been applied can be found as the [Mini-Late Delivery and Penalty Payment](https://templates.accordproject.org/minilatedeliveryandpenalty-payment@0.5.0.html) in the Template Library.
:::

---
id: version-0.21-tutorial-templates
title: Templates Deep Dive
original_id: tutorial-templates

In the [Getting Started](started-hello) section, we learned how to use the existing [helloworld@0.13.0.cta](https://templates.accordproject.org/archives/helloworld@0.13.0.cta) template archive. Here we take a look inside that archive to understand the structure of Accord Project templates.

## Unpack a Template Archive

A `.cta` archive is nothing more than a zip file containing the components of a template. Let's unzip that archive to see what is inside. First, create a directory in the place where you have downloaded that archive, then run the unzip command in a terminal:

```bash
$ mkdir helloworld
$ mv helloworld@0.13.0.cta helloworld
$ cd helloworld
$ unzip helloworld@0.13.0.cta
Archive:  helloworld@0.13.0.cta
 extracting: package.json
   creating: text/
 extracting: text/grammar.tem.md
 extracting: README.md
 extracting: text/sample.md
 extracting: request.json
   creating: model/
 extracting: model/@models.accordproject.org.cicero.contract.cto
 extracting: model/@models.accordproject.org.cicero.runtime.cto
 extracting: model/@models.accordproject.org.money.cto
 extracting: model/model.cto
   creating: logic/
 extracting: logic/logic.ergo
```

## Template Components

Once you have unziped the archive, the directory should contain the following files and sub-directories:

```text
package.json
    Metadata for the template (name, version, description etc)

README.md
    A markdown file that describes the purpose and correct usage for the template

text/grammar.tem.md
    The default grammar for the template

text/sample.md
    A sample clause or contract text that is valid for the template

model/
    A collection of Concerto model files for the template. They define the Template 
Model
    and models for the State, Request, Response, and Obligations used during 
execution.

logic/
    A collection of Ergo files that implement the business logic for the template
```

~~~
test/
    A collection of unit tests for the template

state.json (optional)
    A sample valid state for the clause or contract

request.json (optional)
    A sample valid request to trigger execution for the template
```

In a nutshell, the template archive contains the three main components of the
[Template Triangle](accordproject-concepts#what-is-a-template) in the corresponding
directories (the natural language text of your clause or contract in the `text`
directory, the data model in the `model` directory, and the contract logic in the
`logic` directory). Additional files include metadata and samples which can be used
to illustrate or test the template.

Let us look at each of those components.

### Template Text

#### Grammar

The file in `text/grammar.tem.md` contains the grammar for the template. It is
natural language, with markup to indicate the variable(s) in your Clause or
Contract.

```tem
Name of the person to greet: {{name}}.
Thank you!
```

In the `helloworld` template there is only one variable `name` which is indicated
between `{{` and `}}`.

#### Sample Text

The file in `text/sample.md` contains a sample valid for that grammar.

```md
Name of the person to greet: "Fred Blogs".
Thank you!
```

### Template Model

The file in `model/model.cto` contains the data model for the template. This
includes a description for each of the template variables, including what kind of
variable it is (also called their [type](ref-glossary.html#components-of-data-
models)).

Here is the model for the `helloworld` template:

```ergo
namespace org.accordproject.helloworld

import org.accordproject.cicero.contract.* from
https://models.accordproject.org/cicero/contract.cto
~~~

~~~~
import org.accordproject.cicero.runtime.* from
https://models.accordproject.org/cicero/runtime.cto

asset TemplateModel extends AccordClause {
  o String name // variable 'name' is of type String
}

transaction MyRequest extends Request {
  o String input
}

transaction MyResponse extends Response {
  o String output
}
```

The `TemplateModel` as well as the `Request` and `Response` are types which are
specified using the [Concerto modeling
language](https://github.com/accordproject/concerto).

The `TemplateModel` indicate that the template is for a Clause, and should have a
variable `name` of type `String` (i.e., text).

```ergo
asset TemplateModel extends AccordClause {
  o String name // variable 'name' is of type String
}
```

Types are always declared within a namespace (here `org.accordproject.helloworld`),
which provides a mechanism to disambiguate those types amongst multiple model
files.

### Template Logic

The file in `logic/logic.ergo` contains the executable logic. Each Ergo file is
identified by a namespace, and contains declarations (e.g., constants, functions,
contracts). Here is the Ergo logic for the `helloworld` template:

```ergo
namespace org.accordproject.helloworld

contract HelloWorld over TemplateModel {
  // Simple Clause
  clause greet(request : MyRequest) : MyResponse {
    return MyResponse{ output: "Hello " ++ contract.name ++ " " ++ request.input }
  }
}
```

This declares a single `HelloWorld` contract in the `org.accordproject.helloworld`
namespace, with one `greet` clause.

It also declares that this contract `HelloWorld` is parameterized over the given
`TemplateModel` found in the `models/model.cto` file.

The `greet` clause takes a request of type `MyRequest` as input and returns a
response of type `MyResponse`.
~~~~

The code for the `greet` clause returns a new `MyResponse` response with a single property `output` which is a string. That string is constructed using the string concatenation operator (`++`) in Ergo from the `name` in the contract (`contract.name`) and the input from the request (`request.input`).

## Use the Template

Even after you have unzipped the template archive, you can use that template from the directory directly, in the same way we did from the `.cta` archive in the [Getting Started](started-hello) section.

For instance you can use `cicero parse` or `cicero trigger` as follows:
```bash
$ cd helloworld
$ cicero parse
15:35:12 - info: Using current directory as template folder
15:35:12 - info: Loading a default text/sample.md file.
15:35:14 - info:
{
  "$class": "org.accordproject.helloworld.HelloWorldClause",
  "clauseId": "7258ecf6-cf64-4f9b-807d-c4a3ae6b83ed",
  "name": "Fred Blogs"
}
$ cicero trigger
15:35:17 - info: Using current directory as template folder
15:35:17 - info: Loading a default text/sample.md file.
15:35:17 - info: Loading a default request.json file.
15:35:19 - warn: A state file was not provided, initializing state. Try the --state
flag or create a state.json in the root folder of your template.
15:35:19 - info:
{
  "clause": "helloworld@0.13.0-
ba2600ef11675ad55a036361c1d99e1e9df9a6025c0a35dd5fbe3fc20a0edd07",
  "request": {
    "$class": "org.accordproject.helloworld.MyRequest",
    "input": "Accord Project"
  },
  "response": {
    "$class": "org.accordproject.helloworld.MyResponse",
    "output": "Hello Fred Blogs Accord Project",
    "transactionId": "2d49bd9e-10b1-4ddd-bca6-79a67fe18c9f",
    "timestamp": "2020-09-22T15:40:03.175Z"
  },
  "state": {
    "$class": "org.accordproject.cicero.contract.AccordContractState",
    "stateId": "org.accordproject.cicero.contract.AccordContractState#1"
  },
  "emit": []
}
```

:::note
Remark that if your template directory contains a valid `sample.md` or valid `request.json`, Cicero will automatically detect those so you do not need to pass them using the `--sample` or `--request` options.
:::

---

id: version-0.21-tutorial-vscode
title: With VS Code
original_id: tutorial-vscode

Cicero comes with a VS Code extension for easier development and testing. It includes support for syntax highlighting, allows you to test your template (contract parsing and logic) and to create template archives directly within VS Code.

## Prerequisites

To follow this tutorial on how to use the Cicero VS Code extension, we recommend you install the following:

1. [Node, LTS version](nodejs.org)
1. [VS Code](https://code.visualstudio.com)
1. [Yeoman](https://yeoman.io)
1. [Accord Project Yeoman Generator](https://github.com/accordproject/cicero/tree/master/packages/generator-cicero-template)
1. [Camel Tooling Yeoman VS Code extension](https://marketplace.visualstudio.com/items?itemName=camel-tooling.yo)
1. [Accord Project VS Code extension](https://marketplace.visualstudio.com/items?itemName=accordproject.cicero-vscode-extension)

## Video Tutorial

<iframe title="vimeo-player" src="https://player.vimeo.com/video/444483242" width="640" height="400" frameborder="0" allowfullscreen></iframe>

---
id: version-0.22-markup-preliminaries
title: Preliminaries
original_id: markup-preliminaries

## Markdown & CommonMark

The text for Accord Project templates is written using markdown. It builds on the [CommonMark](https://commonmark.org) standard so that any CommonMark document is valid text for a template or contract.

As with other markup languages, CommonMark can express the document structure (e.g., headings, paragraphs, lists) and formatting useful for readability (e.g., italics, bold, quotations).

The main reference is the [CommonMark Specification](https://spec.commonmark.org/0.29/) but you can find an overview of CommonMark main features in the [CommonMark](markup-commonmark) Section of this guide.

## Accord Project Extensions

Accord Project uses two extensions to CommonMark: CiceroMark for the contract text, and TemplateMark for the template grammar.

### Lexical Conventions

Accord Project contract or template text is a string of `UTF-8` characters.

:::note
By convention, CiceroMark files have the `.md` extensions, and TemplateMark files have the `.tem.md` extension.
:::

The two sequences of characters `{{` and `}}` are reserved and used for the CiceroMark and TemplateMark extensions to CommonMark. There are three kinds of extensions:
1. Variables (written `{{variableName}}`) which may include an optional formatting (written `{{variableName as "FORMAT"}}`).
2. Formulas (written `{{% expression %}}`).
3. Blocks which may contain additional text or markdown. Blocks come in two flavors:
   1. Blocks corresponding to [markdown inline elements](https://spec.commonmark.org/0.29/#inlines) which may contain only other markdown inline elements (e.g., text, emphasis, links). Those have to be written on a single line as follows:
      ```
      {{#blockName variableName}} ... text or markdown ... {{/blockName}}
      ```
   2. Blocks corresponding to [markdown container elements](https://spec.commonmark.org/0.29/#container-blocks) which may contain arbitrary markdown elements (e.g., paragraphs, lists, headings). Those have to be written with each opening and closing tags on their own line as follows:
      ```
      {{#blockName variableName}}
      ... text or markdown ...
      {{/blockBane}}
      ```

### CiceroMark

CiceroMark is used to express the natural language text for legal clauses or contracts. It uses two specific extensions to CommonMark to facilitate contract parsing:
1. Clauses within a contract can be identified using a `clause` block:
   ```
   {{#clause clauseName}}
   text of the clause
   {{/clause}}
   ```
2. The result of formulas within a contract or clause can be identified using:
   ```
   {{% result_of_the_formula %}}
   ```

For instance, the following CiceroMark for a loan between `John Smith` and `Jane Doe` includes a title (`Loan agreement`) followed by some text, followed by a fixed rate interest clause. The clause contains the terms for the loan and the result of calculating the monthly payment.
```tem
# Loan agreement

This is a loan agreement between "John Smith" and "Jane Doe", which shall be entered into
by the parties on January 21, 2021 - 3 PM, except in the event of a force majeure.
```

~~~
{{#clause fixedRate}}
## Fixed rate loan

This is a _fixed interest_ loan to the amount of £100,000.00
at the yearly interest rate of 2.5%
with a loan term of 15,
and monthly payments of {{%£667.00%}}
{{/clause}}
```

More information and examples can be found in the [CiceroMark](markup-ciceromark)
part of this guide.

### TemplateMark

TemplateMark is used to describe families of contracts or clauses with some
variable parts. It is based on CommonMark with several extensions to indicate those
variables parts:
1. _Variables_: e.g., `{{loanAmount}}` indicates the amount for a loan.
2. _Template Blocks_: e.g., `{{#if forceMajeure}}, except in the event of a force
majeure{{/if}}` indicates some optional text in the contract.
3. _Formulas_: e.g., `{{% monthlyPaymentFormula(loanAmount,rate,loanDuration) %}}`
calculates a monthly payment based on the `loanAmount`, `rate`, and `loanDuration`
variables.

For instance, the following TemplateMark for a loan between a `borrower` and a
`lender` includes a title (`Loan agreement`) followed by some text, followed by a
fixed rate interest clause. This template allows for either taking force majeure
into account or not, and calls into a formula to calculate the monthly payment.
```tem
# Loan agreement

This is a loan agreement between {{borrower}} and {{lender}}, which shall be
entered into
by the parties on {{date as "MMMM DD, YYYY - h A"}}{{#if forceMajeure}}, except in
the event of a force majeure{{/if}}.

{{#clause fixedRate}}
## Fixed rate loan

This is a _fixed interest_ loan to the amount of {{loanAmount as "K0,0.00"}}
at the yearly interest rate of {{rate}}%
with a loan term of {{loanDuration}},
and monthly payments of {{% monthlyPaymentFormula(loanAmount,rate,loanDuration) as
"K0,0.00" %}}
{{/clause}}
```

More information and examples can be found in the [TemplateMark](markup-
templatemark) part of this guide.

## Dingus

You can test your template or contract text using the [TemplateMark Dingus]
(https://templatemark-dingus.netlify.app), an online tool which lets you edit the
markdown and see it rendered as HTML, or as a document object model.

![TemplateMark Dingus](assets/dingus1.png)
~~~

You can select whether to parse your text as pure CommonMark (i.e., according to the CommonMark specification), or with the CiceroMark or TemplateMark extensions.

![TemplateMark Dingus](assets/dingus2.png)

You can also inspect the HTML source, or the document object model (abstract syntax tree or AST), even see a pdf rendering for your template.

![TemplateMark Dingus](assets/dingus3.png)

For instance, you can open the TemplateMark from the loan example on this page by clicking [this link](https://templatemark-dingus.netlify.app/#md3=%7B%22source
%22%3A%22%23%20Loan%20agreement%5Cn%5CnThis%20is%20a%20loan%20agreement%20between
%20%7B%7Bborrower%7D%7D%20and%20%7B%7Blender%7D%7D%2C%20which%20shall%20be
%20entered%20into%5Cnby%20the%20parties%20on%20%7B%7Bdate%20as%20%5C%22MMMMM%20DD
%2C%20YYYY%20-%20hhA%5C%22%7D%7D%7B%7B%23if%20forceMajeure%7D%7D%2C%20except%20in
%20the%20event%20of%20a%20force%20majeure%7B%7B%2Fif%7D%7D.%5Cn%5Cn%7B%7B%23clause
%20fixedRate%7D%7D%5Cn%23%23%20Fixed%20rate%20loan%5Cn%5CnThis%20is%20a%20_fixed
%20interest_%20loan%20to%20the%20amount%20of%20%7B%7BloanAmount%20as%20%5C
%22K0%2C0.00%5C%22%7D%7D%5Cnat%20the%20yearly%20interest%20rate%20of%20%7B%7Brate
%7D%7D%25%5Cnwith%20a%20loan%20term%20of%20%7B%7BloanDuration%7D%7D%2C%5Cnand
%20monthly%20payments%20of%20%7B%7B%25%20monthlyPaymentFormula%28loanAmount%2Crate
%2CloanDuration%29%20as%20%5C%22K0%2C0.00%5C%22%20%25%7D%7D%5Cn%7B%7B%2Fclause%7D
%7D%5Cn%22%2C%22defaults%22%3A%7B%22templateMark%22%3Atrue%2C%22ciceroMark
%22%3Afalse%2C%22html%22%3Atrue%2C%22_highlight%22%3Atrue%2C%22_strict%22%3Afalse
%2C%22_view%22%3A%22html%22%7D%7D).

![TemplateMark Dingus](assets/dingus4.png)

---
id: version-0.22-ref-cicero-api
title: Cicero API
original_id: ref-cicero-api

## Modules

<dl>
<dt><a href="#module_cicero-engine">cicero-engine</a></dt>
<dd><p>Clause Engine</p>
</dd>
<dt><a href="#module_cicero-core">cicero-core</a></dt>
<dd><p>Cicero Core - defines the core data types for Cicero.</p>
</dd>
</dl>

## Classes

<dl>
<dt><a href="#Clause">Clause</a></dt>
<dd><p>A Clause is executable business logic, linked to a natural language (legally enforceable) template.
A Clause must be constructed with a template and then prior to execution the data for the clause must be set.
Set the data for the clause (an instance of the template model) by either calling the setData method or by

calling the parse method and passing in natural language text that conforms to the
template grammar.</p>
</dd>
<dt><a href="#Contract">Contract</a></dt>
<dd><p>A Contract is executable business logic, linked to a natural language
(legally enforceable) template.
A Clause must be constructed with a template and then prior to execution the data
for the clause must be set.
Set the data for the clause (an instance of the template model) by either calling
the setData method or by
calling the parse method and passing in natural language text that conforms to the
template grammar.</p>
</dd>
<dt><a href="#Metadata">Metadata</a></dt>
<dd><p>Defines the metadata for a Template, including the name, version, README
markdown.</p>
</dd>
<dt><a href="#Template">Template</a></dt>
<dd><p>A template for a legal clause or contract. A Template has a template model,
request/response transaction types,
a template grammar (natural language for the template) as well as Ergo code for the
business logic of the
template.</p>
</dd>
<dt><a href="#TemplateInstance">TemplateInstance</a></dt>
<dd><p>A TemplateInstance is an instance of a Clause or Contract template. It is
executable business logic, linked to
a natural language (legally enforceable) template.
A TemplateInstance must be constructed with a template and then prior to execution
the data for the clause must be set.
Set the data for the TemplateInstance by either calling the setData method or by
calling the parse method and passing in natural language text that conforms to the
template grammar.</p>
</dd>
<dt><a href="#CompositeArchiveLoader">CompositeArchiveLoader</a></dt>
<dd><p>Manages a set of archive loaders, delegating to the first archive
loader that accepts a URL.</p>
</dd>
</dl>

## Functions

<dl>
<dt><a href="#isPNG">isPNG(buffer)</a> ⇒ <code>Boolean</code></dt>
<dd><p>Checks whether the file is PNG</p>
</dd>
<dt><a href="#getMimeType">getMimeType(buffer)</a> ⇒ <code>Object</code></dt>
<dd><p>Returns the mime-type of the file</p>
</dd>
</dl>

<a name="module_cicero-engine"></a>

## cicero-engine
Clause Engine

* [cicero-engine](#module_cicero-engine)
    * [.Engine](#module_cicero-engine.Engine)

```
        * [new Engine()](#new_module_cicero-engine.Engine_new)
        * [.trigger(clause, request, state, [currentTime], [utcOffset])]
(#module_cicero-engine.Engine+trigger) ⇒ <code>Promise</code>
        * [.invoke(clause, clauseName, params, state, [currentTime], [utcOffset])]
(#module_cicero-engine.Engine+invoke) ⇒ <code>Promise</code>
        * [.init(clause, [currentTime], [utcOffset], params)](#module_cicero-
engine.Engine+init) ⇒ <code>Promise</code>
        * [.getErgoEngine()](#module_cicero-engine.Engine+getErgoEngine) ⇒
<code>ErgoEngine</code>

<a name="module_cicero-engine.Engine"></a>

### cicero-engine.Engine
<p>
Engine class. Stateless execution of clauses against a request object, returning a
response to the caller.
</p>

**Kind**: static class of [<code>cicero-engine</code>](#module_cicero-engine)
**Access**: public

* [.Engine](#module_cicero-engine.Engine)
    * [new Engine()](#new_module_cicero-engine.Engine_new)
    * [.trigger(clause, request, state, [currentTime], [utcOffset])]
(#module_cicero-engine.Engine+trigger) ⇒ <code>Promise</code>
    * [.invoke(clause, clauseName, params, state, [currentTime], [utcOffset])]
(#module_cicero-engine.Engine+invoke) ⇒ <code>Promise</code>
    * [.init(clause, [currentTime], [utcOffset], params)](#module_cicero-
engine.Engine+init) ⇒ <code>Promise</code>
    * [.getErgoEngine()](#module_cicero-engine.Engine+getErgoEngine) ⇒
<code>ErgoEngine</code>

<a name="new_module_cicero-engine.Engine_new"></a>

#### new Engine()
Create the Engine.

<a name="module_cicero-engine.Engine+trigger"></a>

#### engine.trigger(clause, request, state, [currentTime], [utcOffset]) ⇒
<code>Promise</code>
Send a request to a clause for execution

**Kind**: instance method of [<code>Engine</code>](#module_cicero-engine.Engine)
**Returns**: <code>Promise</code> - a promise that resolves to a result for the
clause

| Param | Type | Description |
| --- | --- | --- |
| clause | [<code>Clause</code>](#Clause) | the clause |
| request | <code>object</code> | the request, a JS object that can be deserialized
using the Composer serializer. |
| state | <code>object</code> | the contract state, a JS object that can be
deserialized using the Composer serializer. |
| [currentTime] | <code>string</code> | the definition of 'now', defaults to
current time |
| [utcOffset] | <code>number</code> | UTC Offset for this execution, defaults to
local offset |
```

<a name="module_cicero-engine.Engine+invoke"></a>

#### engine.invoke(clause, clauseName, params, state, [currentTime], [utcOffset]) ⇒ <code>Promise</code>
Invoke a specific clause for execution

Kind: instance method of [<code>Engine</code>](#module_cicero-engine.Engine)
Returns: <code>Promise</code> - a promise that resolves to a result for the clause

Param	Type	Description
clause	[<code>Clause</code>](#Clause)	the clause
clauseName	<code>string</code>	the clause name
params	<code>object</code>	the clause parameters, a JS object whose fields that can be deserialized using the Composer serializer.
state	<code>object</code>	the contract state, a JS object that can be deserialized using the Composer serializer.
[currentTime]	<code>string</code>	the definition of 'now', defaults to current time
[utcOffset]	<code>number</code>	UTC Offset for this execution, defaults to local offset

<a name="module_cicero-engine.Engine+init"></a>

#### engine.init(clause, [currentTime], [utcOffset], params) ⇒ <code>Promise</code>
Initialize a clause

Kind: instance method of [<code>Engine</code>](#module_cicero-engine.Engine)
Returns: <code>Promise</code> - a promise that resolves to a result for the clause initialization

Param	Type	Description
clause	[<code>Clause</code>](#Clause)	the clause
[currentTime]	<code>string</code>	the definition of 'now', defaults to current time
[utcOffset]	<code>number</code>	UTC Offset for this execution, defaults to local offset
params	<code>object</code>	the clause parameters, a JS object whose fields that can be deserialized using the Composer serializer.

<a name="module_cicero-engine.Engine+getErgoEngine"></a>

#### engine.getErgoEngine() ⇒ <code>ErgoEngine</code>
Provides access to the underlying Ergo engine.

Kind: instance method of [<code>Engine</code>](#module_cicero-engine.Engine)
Returns: <code>ErgoEngine</code> - the Ergo Engine for this Engine
<a name="module_cicero-core"></a>

## cicero-core
Cicero Core - defines the core data types for Cicero.

<a name="Clause"></a>

## Clause
A Clause is executable business logic, linked to a natural language (legally enforceable) template.

A Clause must be constructed with a template and then prior to execution the data
for the clause must be set.
Set the data for the clause (an instance of the template model) by either calling
the setData method or by
calling the parse method and passing in natural language text that conforms to the
template grammar.

Kind: global class
Access: public
<a name="Contract"></a>

## Contract
A Contract is executable business logic, linked to a natural language (legally
enforceable) template.
A Clause must be constructed with a template and then prior to execution the data
for the clause must be set.
Set the data for the clause (an instance of the template model) by either calling
the setData method or by
calling the parse method and passing in natural language text that conforms to the
template grammar.

Kind: global class
Access: public
<a name="Metadata"></a>

## Metadata
Defines the metadata for a Template, including the name, version, README markdown.

Kind: global class
Access: public

* [Metadata](#Metadata)
    * [new Metadata(packageJson, readme, samples, request, logo)]
(#new_Metadata_new)
    * _instance_
        * [.getTemplateType()](#Metadata+getTemplateType) ⇒ <code>number</code>
        * [.getLogo()](#Metadata+getLogo) ⇒ <code>Buffer</code>
        * [.getAuthor()](#Metadata+getAuthor) ⇒ <code>*</code>
        * [.getRuntime()](#Metadata+getRuntime) ⇒ <code>string</code>
        * [.getCiceroVersion()](#Metadata+getCiceroVersion) ⇒ <code>string</code>
        * [.satisfiesCiceroVersion(version)](#Metadata+satisfiesCiceroVersion) ⇒
<code>string</code>
        * [.getSamples()](#Metadata+getSamples) ⇒ <code>object</code>
        * [.getRequest()](#Metadata+getRequest) ⇒ <code>object</code>
        * [.getSample(locale)](#Metadata+getSample) ⇒ <code>string</code>
        * [.getREADME()](#Metadata+getREADME) ⇒ <code>String</code>
        * [.getPackageJson()](#Metadata+getPackageJson) ⇒ <code>object</code>
        * [.getName()](#Metadata+getName) ⇒ <code>string</code>
        * [.getDisplayName()](#Metadata+getDisplayName) ⇒ <code>string</code>
        * [.getKeywords()](#Metadata+getKeywords) ⇒ <code>Array</code>
        * [.getDescription()](#Metadata+getDescription) ⇒ <code>string</code>
        * [.getVersion()](#Metadata+getVersion) ⇒ <code>string</code>
        * [.getIdentifier()](#Metadata+getIdentifier) ⇒ <code>string</code>
        * [.createTargetMetadata(runtimeName)](#Metadata+createTargetMetadata) ⇒
<code>object</code>
        * [.toJSON()](#Metadata+toJSON) ⇒ <code>object</code>
    * _static_
        * [.checkImage(buffer)](#Metadata.checkImage)
        * [.checkImageDimensions(buffer, mimeType)](#Metadata.checkImageDimensions)

```
<a name="new_Metadata_new"></a>

### new Metadata(packageJson, readme, samples, request, logo)
Create the Metadata.
<p>
<strong>Note: Only to be called by framework code. Applications should
retrieve instances from [Template](#Template)</strong>
</p>


| Param | Type | Description |
| --- | --- | --- |
| packageJson | <code>object</code> | the JS object for package.json (required) |
| readme | <code>String</code> | the README.md for the template (may be null) |
| samples | <code>object</code> | the sample markdown for the template in different
locales, |
| request | <code>object</code> | the JS object for the sample request |
| logo | <code>Buffer</code> | the bytes data for the image represented as an
object whose keys are the locales and whose values are the sample markdown. For
example:  {      default: 'default sample markdown',      en: 'sample text in
english',      fr: 'exemple de texte français'  } Locale keys (with the exception
of default) conform to the IETF Language Tag specification (BCP 47). THe `default`
key represents sample template text in a non-specified language, stored in a file
called `sample.md`. |

<a name="Metadata+getTemplateType"></a>

### metadata.getTemplateType() ⇒ <code>number</code>
Returns either a 0 (for a contract template), or 1 (for a clause template)

**Kind**: instance method of [<code>Metadata</code>](#Metadata)
**Returns**: <code>number</code> - the template type
<a name="Metadata+getLogo"></a>

### metadata.getLogo() ⇒ <code>Buffer</code>
Returns the logo at the root of the template

**Kind**: instance method of [<code>Metadata</code>](#Metadata)
**Returns**: <code>Buffer</code> - the bytes data of logo
<a name="Metadata+getAuthor"></a>

### metadata.getAuthor() ⇒ <code>\*</code>
Returns the author for this template.

**Kind**: instance method of [<code>Metadata</code>](#Metadata)
**Returns**: <code>\*</code> - the author information
<a name="Metadata+getRuntime"></a>

### metadata.getRuntime() ⇒ <code>string</code>
Returns the name of the runtime target for this template, or null if this template
has not been compiled for a specific runtime.

**Kind**: instance method of [<code>Metadata</code>](#Metadata)
**Returns**: <code>string</code> - the name of the runtime
<a name="Metadata+getCiceroVersion"></a>

### metadata.getCiceroVersion() ⇒ <code>string</code>
Returns the version of Cicero that this template is compatible with.
```

i.e. which version of the runtime was this template built for?
The version string conforms to the semver definition

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>string</code> - the semantic version
<a name="Metadata+satisfiesCiceroVersion"></a>

### metadata.satisfiesCiceroVersion(version) ⇒ <code>string</code>
Only returns true if the current cicero version satisfies the target version of this template

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>string</code> - the semantic version

Param	Type	Description
version	<code>string</code>	the cicero version to check against

<a name="Metadata+getSamples"></a>

### metadata.getSamples() ⇒ <code>object</code>
Returns the samples for this template.

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>object</code> - the sample files for the template
<a name="Metadata+getRequest"></a>

### metadata.getRequest() ⇒ <code>object</code>
Returns the sample request for this template.

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>object</code> - the sample request for the template
<a name="Metadata+getSample"></a>

### metadata.getSample(locale) ⇒ <code>string</code>
Returns the sample for this template in the given locale. This may be null.
If no locale is specified returns the default sample if it has been specified.

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>string</code> - the sample file for the template in the given locale or null

Param	Type	Default	Description
locale	<code>string</code>	<code>null</code>	the IETF language code for the language.

<a name="Metadata+getREADME"></a>

### metadata.getREADME() ⇒ <code>String</code>
Returns the README.md for this template. This may be null if the template does not have a README.md

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>String</code> - the README.md file for the template or null
<a name="Metadata+getPackageJson"></a>

### metadata.getPackageJson() ⇒ <code>object</code>
Returns the package.json for this template.

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>object</code> - the Javascript object for package.json
<a name="Metadata+getName"></a>

### metadata.getName() ⇒ <code>string</code>
Returns the name for this template.

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>string</code> - the name of the template
<a name="Metadata+getDisplayName"></a>

### metadata.getDisplayName() ⇒ <code>string</code>
Returns the display name for this template.

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>string</code> - the display name of the template
<a name="Metadata+getKeywords"></a>

### metadata.getKeywords() ⇒ <code>Array</code>
Returns the keywords for this template.

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>Array</code> - the keywords of the template
<a name="Metadata+getDescription"></a>

### metadata.getDescription() ⇒ <code>string</code>
Returns the description for this template.

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>string</code> - the description of the template
<a name="Metadata+getVersion"></a>

### metadata.getVersion() ⇒ <code>string</code>
Returns the version for this template.

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>string</code> - the description of the template
<a name="Metadata+getIdentifier"></a>

### metadata.getIdentifier() ⇒ <code>string</code>
Returns the identifier for this template, formed from name@version.

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>string</code> - the identifier of the template
<a name="Metadata+createTargetMetadata"></a>

### metadata.createTargetMetadata(runtimeName) ⇒ <code>object</code>
Return new Metadata for a target runtime

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>object</code> - the new Metadata

Param	Type	Description
runtimeName	<code>string</code>	the target runtime name

<a name="Metadata+toJSON"></a>

```
### metadata.toJSON() ⇒ <code>object</code>
Return the whole metadata content, for hashing

**Kind**: instance method of [<code>Metadata</code>](#Metadata)
**Returns**: <code>object</code> - the content of the metadata object
<a name="Metadata.checkImage"></a>

### Metadata.checkImage(buffer)
Check the buffer is a png file with the right size

**Kind**: static method of [<code>Metadata</code>](#Metadata)

| Param | Type | Description |
| --- | --- | --- |
| buffer | <code>Buffer</code> | the buffer object |

<a name="Metadata.checkImageDimensions"></a>

### Metadata.checkImageDimensions(buffer, mimeType)
Checks if dimensions for the image are correct.

**Kind**: static method of [<code>Metadata</code>](#Metadata)

| Param | Type | Description |
| --- | --- | --- |
| buffer | <code>Buffer</code> | the buffer object |
| mimeType | <code>string</code> | the mime type of the object |

<a name="Template"></a>

## *Template*
A template for a legal clause or contract. A Template has a template model,
request/response transaction types,
a template grammar (natural language for the template) as well as Ergo code for the
business logic of the
template.

**Kind**: global abstract class
**Access**: public

* *[Template](#Template)*
    * *[new Template(packageJson, readme, samples, request, logo, options)]
(#new_Template_new)*
    * _instance_
        * *[.validate()](#Template+validate)*
        * *[.getTemplateModel()](#Template+getTemplateModel) ⇒
<code>ClassDeclaration</code>*
        * *[.getIdentifier()](#Template+getIdentifier) ⇒ <code>String</code>*
        * *[.getMetadata()](#Template+getMetadata) ⇒ [<code>Metadata</code>]
(#Metadata)*
        * *[.getName()](#Template+getName) ⇒ <code>String</code>*
        * *[.getDisplayName()](#Template+getDisplayName) ⇒ <code>string</code>*
        * *[.getVersion()](#Template+getVersion) ⇒ <code>String</code>*
        * *[.getDescription()](#Template+getDescription) ⇒ <code>String</code>*
        * *[.getHash()](#Template+getHash) ⇒ <code>string</code>*
        * *[.toArchive([language], [options], logo)](#Template+toArchive) ⇒
<code>Promise.&lt;Buffer&gt;</code>*
        * *[.getParserManager()](#Template+getParserManager) ⇒
<code>ParserManager</code>*
```

```
        * *[.getLogicManager()](#Template+getLogicManager) ⇒
<code>LogicManager</code>*
        * *[.getIntrospector()](#Template+getIntrospector) ⇒
<code>Introspector</code>*
        * *[.getFactory()](#Template+getFactory) ⇒ <code>Factory</code>*
        * *[.getSerializer()](#Template+getSerializer) ⇒ <code>Serializer</code>*
        * *[.getRequestTypes()](#Template+getRequestTypes) ⇒ <code>Array</code>*
        * *[.getResponseTypes()](#Template+getResponseTypes) ⇒ <code>Array</code>*
        * *[.getEmitTypes()](#Template+getEmitTypes) ⇒ <code>Array</code>*
        * *[.getStateTypes()](#Template+getStateTypes) ⇒ <code>Array</code>*
        * *[.hasLogic()](#Template+hasLogic) ⇒ <code>boolean</code>*
    * _static_
        * *[.fromDirectory(path, [options])](#Template.fromDirectory) ⇒
[<code>Promise.&lt;Template&gt;</code>](#Template)*
        * *[.fromArchive(buffer, [options])](#Template.fromArchive) ⇒
[<code>Promise.&lt;Template&gt;</code>](#Template)*
        * *[.fromUrl(url, [options])](#Template.fromUrl) ⇒ <code>Promise</code>*
        * *[.instanceOf(classDeclaration, fqt)](#Template.instanceOf) ⇒
<code>boolean</code>*

<a name="new_Template_new"></a>

### *new Template(packageJson, readme, samples, request, logo, options)*
Create the Template.
Note: Only to be called by framework code. Applications should
retrieve instances from [fromArchive](#Template.fromArchive) or [fromDirectory]
(#Template.fromDirectory).


| Param | Type | Description |
| --- | --- | --- |
| packageJson | <code>object</code> | the JS object for package.json |
| readme | <code>String</code> | the readme in markdown for the template (optional)
|
| samples | <code>object</code> | the sample text for the template in different
locales |
| request | <code>object</code> | the JS object for the sample request |
| logo | <code>Buffer</code> | the bytes data of logo |
| options | <code>Object</code> | e.g., { warnings: true } |

<a name="Template+validate"></a>

### *template.validate()*
Verifies that the template is well formed.
Compiles the Ergo logic.
Throws an exception with the details of any validation errors.

**Kind**: instance method of [<code>Template</code>](#Template)
<a name="Template+getTemplateModel"></a>

### *template.getTemplateModel() ⇒ <code>ClassDeclaration</code>*
Returns the template model for the template

**Kind**: instance method of [<code>Template</code>](#Template)
**Returns**: <code>ClassDeclaration</code> - the template model for the template
**Throws**:

- <code>Error</code> if no template model is found, or multiple template models are
found
```

```markdown
<a name="Template+getIdentifier"></a>

### *template.getIdentifier() ⇒ <code>String</code>*
Returns the identifier for this template

**Kind**: instance method of [<code>Template</code>](#Template)
**Returns**: <code>String</code> - the identifier of this template
<a name="Template+getMetadata"></a>

### *template.getMetadata() ⇒ [<code>Metadata</code>](#Metadata)*
Returns the metadata for this template

**Kind**: instance method of [<code>Template</code>](#Template)
**Returns**: [<code>Metadata</code>](#Metadata) - the metadata for this template
<a name="Template+getName"></a>

### *template.getName() ⇒ <code>String</code>*
Returns the name for this template

**Kind**: instance method of [<code>Template</code>](#Template)
**Returns**: <code>String</code> - the name of this template
<a name="Template+getDisplayName"></a>

### *template.getDisplayName() ⇒ <code>string</code>*
Returns the display name for this template.

**Kind**: instance method of [<code>Template</code>](#Template)
**Returns**: <code>string</code> - the display name of the template
<a name="Template+getVersion"></a>

### *template.getVersion() ⇒ <code>String</code>*
Returns the version for this template

**Kind**: instance method of [<code>Template</code>](#Template)
**Returns**: <code>String</code> - the version of this template. Use semver module
to parse.
<a name="Template+getDescription"></a>

### *template.getDescription() ⇒ <code>String</code>*
Returns the description for this template

**Kind**: instance method of [<code>Template</code>](#Template)
**Returns**: <code>String</code> - the description of this template
<a name="Template+getHash"></a>

### *template.getHash() ⇒ <code>string</code>*
Gets a content based SHA-256 hash for this template. Hash
is based on the metadata for the template plus the contents of
all the models and all the script files.

**Kind**: instance method of [<code>Template</code>](#Template)
**Returns**: <code>string</code> - the SHA-256 hash in hex format
<a name="Template+toArchive"></a>

### *template.toArchive([language], [options], logo) ⇒
<code>Promise.&lt;Buffer&gt;</code>*
Persists this template to a Cicero Template Archive (cta) file.
```

Kind: instance method of [<code>Template</code>](#Template)
Returns: <code>Promise.&lt;Buffer&gt;</code> - the zlib buffer

Param	Type	Description
[language]	<code>string</code>	target language for the archive (should be 'ergo')
[options]	<code>Object</code>	JSZip options
logo	<code>Buffer</code>	Bytes data of the PNG file

<a name="Template+getParserManager"></a>

### *template.getParserManager() ⇒ <code>ParserManager</code>*
Provides access to the parser manager for this template.
The parser manager can convert template data to and from
natural language text.

Kind: instance method of [<code>Template</code>](#Template)
Returns: <code>ParserManager</code> - the ParserManager for this template
<a name="Template+getLogicManager"></a>

### *template.getLogicManager() ⇒ <code>LogicManager</code>*
Provides access to the template logic for this template.
The template logic encapsulate the code necessary to
execute the clause or contract.

Kind: instance method of [<code>Template</code>](#Template)
Returns: <code>LogicManager</code> - the LogicManager for this template
<a name="Template+getIntrospector"></a>

### *template.getIntrospector() ⇒ <code>Introspector</code>*
Provides access to the Introspector for this template. The Introspector
is used to reflect on the types defined within this template.

Kind: instance method of [<code>Template</code>](#Template)
Returns: <code>Introspector</code> - the Introspector for this template
<a name="Template+getFactory"></a>

### *template.getFactory() ⇒ <code>Factory</code>*
Provides access to the Factory for this template. The Factory
is used to create the types defined in this template.

Kind: instance method of [<code>Template</code>](#Template)
Returns: <code>Factory</code> - the Factory for this template
<a name="Template+getSerializer"></a>

### *template.getSerializer() ⇒ <code>Serializer</code>*
Provides access to the Serializer for this template. The Serializer
is used to serialize instances of the types defined within this template.

Kind: instance method of [<code>Template</code>](#Template)
Returns: <code>Serializer</code> - the Serializer for this template
<a name="Template+getRequestTypes"></a>

### *template.getRequestTypes() ⇒ <code>Array</code>*
Provides a list of the input types that are accepted by this Template. Types use
the fully-qualified form.

Kind: instance method of [<code>Template</code>](#Template)

Returns: <code>Array</code> - a list of the request types
<a name="Template+getResponseTypes"></a>

### *template.getResponseTypes() ⇒ <code>Array</code>*
Provides a list of the response types that are returned by this Template. Types use
the fully-qualified form.

Kind: instance method of [<code>Template</code>](#Template)
Returns: <code>Array</code> - a list of the response types
<a name="Template+getEmitTypes"></a>

### *template.getEmitTypes() ⇒ <code>Array</code>*
Provides a list of the emit types that are emitted by this Template. Types use the
fully-qualified form.

Kind: instance method of [<code>Template</code>](#Template)
Returns: <code>Array</code> - a list of the emit types
<a name="Template+getStateTypes"></a>

### *template.getStateTypes() ⇒ <code>Array</code>*
Provides a list of the state types that are expected by this Template. Types use
the fully-qualified form.

Kind: instance method of [<code>Template</code>](#Template)
Returns: <code>Array</code> - a list of the state types
<a name="Template+hasLogic"></a>

### *template.hasLogic() ⇒ <code>boolean</code>*
Returns true if the template has logic, i.e. has more than one script file.

Kind: instance method of [<code>Template</code>](#Template)
Returns: <code>boolean</code> - true if the template has logic
<a name="Template.fromDirectory"></a>

### *Template.fromDirectory(path, [options]) ⇒
[<code>Promise.&lt;Template&gt;</code>](#Template)*
Builds a Template from the contents of a directory.
The directory must include a package.json in the root (used to specify
the name, version and description of the template).

Kind: static method of [<code>Template</code>](#Template)
Returns: [<code>Promise.&lt;Template&gt;</code>](#Template) - a Promise to the
instantiated template

Param	Type	Default	Description
path	<code>String</code>		to a local directory
[options]	<code>Object</code>	<code></code>	an optional set of options to
configure the instance.

<a name="Template.fromArchive"></a>

### *Template.fromArchive(buffer, [options]) ⇒
[<code>Promise.&lt;Template&gt;</code>](#Template)*
Create a template from an archive.

Kind: static method of [<code>Template</code>](#Template)
Returns: [<code>Promise.&lt;Template&gt;</code>](#Template) - a Promise to the
template

Param	Type	Default	Description
buffer	<code>Buffer</code>		the buffer to a Cicero Template Archive (cta) file
[options]	<code>Object</code>	<code></code>	an optional set of options to configure the instance.

<a name="Template.fromUrl"></a>

### *Template.fromUrl(url, [options]) ⇒ <code>Promise</code>*
Create a template from an URL.

Kind: static method of [<code>Template</code>](#Template)
Returns: <code>Promise</code> - a Promise to the template

Param	Type	Default	Description
url	<code>String</code>		the URL to a Cicero Template Archive (cta) file
[options]	<code>Object</code>	<code></code>	an optional set of options to configure the instance.

<a name="Template.instanceOf"></a>

### *Template.instanceOf(classDeclaration, fqt) ⇒ <code>boolean</code>*
Check to see if a ClassDeclaration is an instance of the specified fully qualified type name.

Kind: static method of [<code>Template</code>](#Template)
Returns: <code>boolean</code> - True if classDeclaration an instance of the specified fully
qualified type name, false otherwise.
Internal:

Param	Type	Description
classDeclaration	<code>ClassDeclaration</code>	The class to test
fqt	<code>String</code>	The fully qualified type name.

<a name="TemplateInstance"></a>

## *TemplateInstance*
A TemplateInstance is an instance of a Clause or Contract template. It is executable business logic, linked to
a natural language (legally enforceable) template.
A TemplateInstance must be constructed with a template and then prior to execution the data for the clause must be set.
Set the data for the TemplateInstance by either calling the setData method or by calling the parse method and passing in natural language text that conforms to the template grammar.

Kind: global abstract class
Access: public

* *[TemplateInstance](#TemplateInstance)*
    * *[new TemplateInstance(template)](#new_TemplateInstance_new)*
    * _instance_
        * *[.setData(data)](#TemplateInstance+setData)*
        * *[.getData()](#TemplateInstance+getData) ⇒ <code>object</code>*

        * *[.getEngine()](#TemplateInstance+getEngine) ⇒ <code>object</code>*
        * *[.getDataAsConcertoObject()](#TemplateInstance+getDataAsConcertoObject) ⇒ <code>object</code>*
        * *[.parse(input, [currentTime], [utcOffset], [fileName])](#TemplateInstance+parse)*
        * *[.draft([options], [currentTime], [utcOffset])](#TemplateInstance+draft) ⇒ <code>string</code>*
        * *[.formatCiceroMark(ciceroMarkParsed, options, format)](#TemplateInstance+formatCiceroMark) ⇒ <code>string</code>*
        * *[.getIdentifier()](#TemplateInstance+getIdentifier) ⇒ <code>String</code>*
        * *[.getTemplate()](#TemplateInstance+getTemplate) ⇒ [<code>Template</code>](#Template)*
        * *[.getLogicManager()](#TemplateInstance+getLogicManager) ⇒ <code>LogicManager</code>*
        * *[.toJSON()](#TemplateInstance+toJSON) ⇒ <code>object</code>*
    * _static_
        * *[.ciceroFormulaEval(logicManager, clauseId, ergoEngine, name)](#TemplateInstance.ciceroFormulaEval) ⇒ <code>*</code>*
        * *[.rebuildParser(parserManager, logicManager, ergoEngine, templateName, grammar)](#TemplateInstance.rebuildParser)*

<a name="new_TemplateInstance_new"></a>

### *new TemplateInstance(template)*
Create the Clause and link it to a Template.

Param	Type	Description
template	[<code>Template</code>](#Template)	the template for the clause

<a name="TemplateInstance+setData"></a>

### *templateInstance.setData(data)*
Set the data for the clause

Kind: instance method of [<code>TemplateInstance</code>](#TemplateInstance)

Param	Type	Description
data	<code>object</code>	the data for the clause, must be an instance of the template model for the clause's template. This should be a plain JS object and will be deserialized and validated into the Concerto object before assignment.

<a name="TemplateInstance+getData"></a>

### *templateInstance.getData() ⇒ <code>object</code>*
Get the data for the clause. This is a plain JS object. To retrieve the Concerto object call getConcertoData().

Kind: instance method of [<code>TemplateInstance</code>](#TemplateInstance)
Returns: <code>object</code> - - the data for the clause, or null if it has not been set
<a name="TemplateInstance+getEngine"></a>

### *templateInstance.getEngine() ⇒ <code>object</code>*
Get the current Ergo engine

```
**Kind**: instance method of [<code>TemplateInstance</code>](#TemplateInstance)
**Returns**: <code>object</code> - - the data for the clause, or null if it has not
been set
<a name="TemplateInstance+getDataAsConcertoObject"></a>

### *templateInstance.getDataAsConcertoObject() ⇒ <code>object</code>*
Get the data for the clause. This is a Concerto object. To retrieve the
plain JS object suitable for serialization call toJSON() and retrieve the `data`
property.

**Kind**: instance method of [<code>TemplateInstance</code>](#TemplateInstance)
**Returns**: <code>object</code> - - the data for the clause, or null if it has not
been set
<a name="TemplateInstance+parse"></a>

### *templateInstance.parse(input, [currentTime], [utcOffset], [fileName])*
Set the data for the clause by parsing natural language text.

**Kind**: instance method of [<code>TemplateInstance</code>](#TemplateInstance)

| Param | Type | Description |
| --- | --- | --- |
| input | <code>string</code> | the text for the clause |
| [currentTime] | <code>string</code> | the definition of 'now', defaults to
current time |
| [utcOffset] | <code>number</code> | UTC Offset for this execution, defaults to
local offset |
| [fileName] | <code>string</code> | the fileName for the text (optional) |

<a name="TemplateInstance+draft"></a>

### *templateInstance.draft([options], [currentTime], [utcOffset]) ⇒
<code>string</code>*
Generates the natural language text for a contract or clause clause; combining the
text from the template
and the instance data.

**Kind**: instance method of [<code>TemplateInstance</code>](#TemplateInstance)
**Returns**: <code>string</code> - the natural language text for the contract or
clause; created by combining the structure of
the template with the JSON data for the clause.

| Param | Type | Description |
| --- | --- | --- |
| [options] | <code>\*</code> | text generation options. |
| [currentTime] | <code>string</code> | the definition of 'now', defaults to
current time |
| [utcOffset] | <code>number</code> | UTC Offset for this execution, defaults to
local offset |

<a name="TemplateInstance+formatCiceroMark"></a>

### *templateInstance.formatCiceroMark(ciceroMarkParsed, options, format) ⇒
<code>string</code>*
Format CiceroMark

**Kind**: instance method of [<code>TemplateInstance</code>](#TemplateInstance)
**Returns**: <code>string</code> - the result of parsing and printing back the text
```

Param	Type	Description
ciceroMarkParsed	<code>object</code>	the parsed CiceroMark DOM
options	<code>object</code>	parameters to the formatting
format	<code>string</code>	to the text generation

<a name="TemplateInstance+getIdentifier"></a>

### *templateInstance.getIdentifier() ⇒ <code>String</code>*
Returns the identifier for this clause. The identifier is the identifier of
the template plus '-' plus a hash of the data for the clause (if set).

Kind: instance method of [<code>TemplateInstance</code>](#TemplateInstance)
Returns: <code>String</code> - the identifier of this clause
<a name="TemplateInstance+getTemplate"></a>

### *templateInstance.getTemplate() ⇒ [<code>Template</code>](#Template)*
Returns the template for this clause

Kind: instance method of [<code>TemplateInstance</code>](#TemplateInstance)
Returns: [<code>Template</code>](#Template) - the template for this clause
<a name="TemplateInstance+getLogicManager"></a>

### *templateInstance.getLogicManager() ⇒ <code>LogicManager</code>*
Returns the template logic for this clause

Kind: instance method of [<code>TemplateInstance</code>](#TemplateInstance)
Returns: <code>LogicManager</code> - the template for this clause
<a name="TemplateInstance+toJSON"></a>

### *templateInstance.toJSON() ⇒ <code>object</code>*
Returns a JSON representation of the clause

Kind: instance method of [<code>TemplateInstance</code>](#TemplateInstance)
Returns: <code>object</code> - the JS object for serialization
<a name="TemplateInstance.ciceroFormulaEval"></a>

### *TemplateInstance.ciceroFormulaEval(logicManager, clauseId, ergoEngine, name) ⇒
<code>*</code>*
Constructs a function for formula evaluation based for this template instance

Kind: static method of [<code>TemplateInstance</code>](#TemplateInstance)
Returns: <code>*</code> - A function from formula code + input data to result

Param	Type	Description
logicManager	<code>*</code>	the logic manager
clauseId	<code>string</code>	this instance identifier
ergoEngine	<code>*</code>	the evaluation engine
name	<code>string</code>	the name of the formula

<a name="TemplateInstance.rebuildParser"></a>

### *TemplateInstance.rebuildParser(parserManager, logicManager, ergoEngine,
templateName, grammar)*
Utility to rebuild a parser when the grammar changes

Kind: static method of [<code>TemplateInstance</code>](#TemplateInstance)

Param	Type	Description
parserManager	<code>*</code>	the parser manager
logicManager	<code>*</code>	the logic manager
ergoEngine	<code>*</code>	the evaluation engine
templateName	<code>string</code>	this template name
grammar	<code>string</code>	the new grammar

<a name="CompositeArchiveLoader"></a>

## CompositeArchiveLoader
Manages a set of archive loaders, delegating to the first archive
loader that accepts a URL.

Kind: global class

* [CompositeArchiveLoader](#CompositeArchiveLoader)
    * [new CompositeArchiveLoader()](#new_CompositeArchiveLoader_new)
    * [.addArchiveLoader(archiveLoader)](#CompositeArchiveLoader+addArchiveLoader)
    * [.clearArchiveLoaders()](#CompositeArchiveLoader+clearArchiveLoaders)
    * *[.accepts(url)](#CompositeArchiveLoader+accepts) ⇒ <code>boolean</code>*
    * [.load(url, options)](#CompositeArchiveLoader+load) ⇒ <code>Promise</code>

<a name="new_CompositeArchiveLoader_new"></a>

### new CompositeArchiveLoader()
Create the CompositeArchiveLoader. Used to delegate to a set of ArchiveLoaders.

<a name="CompositeArchiveLoader+addArchiveLoader"></a>

### compositeArchiveLoader.addArchiveLoader(archiveLoader)
Adds a ArchiveLoader implemenetation to the ArchiveLoader

Kind: instance method of [<code>CompositeArchiveLoader</code>]
(#CompositeArchiveLoader)

Param	Type	Description
archiveLoader	<code>ArchiveLoader</code>	The archive to add to the CompositeArchiveLoader

<a name="CompositeArchiveLoader+clearArchiveLoaders"></a>

### compositeArchiveLoader.clearArchiveLoaders()
Remove all registered ArchiveLoaders

Kind: instance method of [<code>CompositeArchiveLoader</code>]
(#CompositeArchiveLoader)
<a name="CompositeArchiveLoader+accepts"></a>

### *compositeArchiveLoader.accepts(url) ⇒ <code>boolean</code>*
Returns true if this ArchiveLoader can process the URL

Kind: instance abstract method of [<code>CompositeArchiveLoader</code>]
(#CompositeArchiveLoader)
Returns: <code>boolean</code> - true if this ArchiveLoader accepts the URL

Param	Type	Description

| url | <code>string</code> | the URL |

<a name="CompositeArchiveLoader+load"></a>

### compositeArchiveLoader.load(url, options) ⇒ <code>Promise</code>
Load a Archive from a URL and return it

Kind: instance method of [<code>CompositeArchiveLoader</code>](#CompositeArchiveLoader)
Returns: <code>Promise</code> - a promise to the Archive

Param	Type	Description
url	<code>string</code>	the url to get
options	<code>object</code>	additional options

<a name="isPNG"></a>

## isPNG(buffer) ⇒ <code>Boolean</code>
Checks whether the file is PNG

Kind: global function
Returns: <code>Boolean</code> - whether the file in PNG

Param	Type	Description
buffer	<code>Buffer</code>	buffer of the file

<a name="getMimeType"></a>

## getMimeType(buffer) ⇒ <code>Object</code>
Returns the mime-type of the file

Kind: global function
Returns: <code>Object</code> - the mime-type of the file

Param	Type	Description
buffer	<code>Buffer</code>	buffer of the file

---
id: version-0.22-ref-cicero-cli
title: Command Line
original_id: ref-cicero-cli

Install the `@accordproject/cicero-cli` npm package to access the Cicero command line interface (CLI). After installation you can use the `cicero` command and its sub-commands as described below.

To install the Cicero CLI:
```
npm install -g @accordproject/cicero-cli
```

## Usage

```md
cicero <cmd> [args]

Commands:
  cicero parse       parse a contract text
  cicero draft       create contract text from data
  cicero normalize   normalize markdown (parse & redraft)
  cicero trigger     send a request to the contract
  cicero invoke      invoke a clause of the contract
  cicero initialize  initialize a clause
  cicero archive     create a template archive
  cicero compile     generate code for a target platform
  cicero get         save local copies of external dependencies

Options:
  --version      Show version number                                   [boolean]
  --verbose, -v                                                 [default: false]
  --help         Show help                                             [boolean]
```

## cicero parse

`cicero parse` loads a template from a directory on disk and then parses input
clause (or contract) text using the template. If successful, the template model is
printed to console. If there are syntax errors, the line and column and error
information are printed.

```md
cicero parse

parse a contract text

Options:
  --version      Show version number                                   [boolean]
  --verbose, -v                                                 [default: false]
  --help         Show help                                             [boolean]
  --template     path to the template                                   [string]
  --sample       path to the contract text                              [string]
  --output       path to the output file                                [string]
  --currentTime  set current time                       [string] [default: null]
  --utcOffset    set UTC offset                         [number] [default: null]
  --offline      do not resolve external models       [boolean] [default: false]
  --warnings     print warnings                       [boolean] [default: false]
```

## cicero draft

`cicero draft` creates contract text from data.

```md
cicero draft

create contract text from data

Options:
  --version           Show version number                              [boolean]
  --verbose, -v                                                 [default: false]
  --help              Show help                                        [boolean]
  --template          path to the template                              [string]
```

  --data   path to the contract data   [string]
  --output   path to the output file   [string]
  --currentTime   set current time   [string] [default: null]
  --utcOffset   set UTC offset   [number] [default: null]
  --offline   do not resolve external models [boolean] [default: false]
  --format   target format   [string]
  --unquoteVariables remove variables quoting   [boolean] [default: false]
  --warnings   print warnings   [boolean] [default: false]

## cicero normalize

`cicero normalize` normalizes markdown text by parsing and redrafting the text.

```md
cicero normalize

normalize markdown (parse & redraft)

Options:
  --version           Show version number                              [boolean]
  --verbose, -v                                                 [default: false]
  --help              Show help                                        [boolean]
  --template          path to the template                              [string]
  --sample            path to the contract text                         [string]
  --overwrite         overwrite the contract text     [boolean] [default: false]
  --output            path to the output file                           [string]
  --currentTime       set current time                  [string] [default: null]
  --utcOffset         set UTC offset                    [number] [default: null]
  --offline           do not resolve external models  [boolean] [default: false]
  --warnings          print warnings                  [boolean] [default: false]
  --format            target format                                     [string]
  --unquoteVariables  remove variables quoting        [boolean] [default: false]
```

## cicero trigger

`cicero trigger` sends a request to the contract.

```md
cicero trigger

send a request to the contract

Options:
  --version      Show version number                                   [boolean]
  --verbose, -v                                                 [default: false]
  --help         Show help                                             [boolean]
  --template     path to the template                                   [string]
  --sample       path to the contract text                              [string]
  --request      path to the JSON request                                [array]
  --state        path to the JSON state                                 [string]
  --currentTime  set current time                       [string] [default: null]
  --utcOffset    set UTC offset                         [number] [default: null]
  --offline      do not resolve external models       [boolean] [default: false]
  --warnings     print warnings                       [boolean] [default: false]
```

## cicero invoke

`cicero invoke` invokes a specific clause (`--clauseName`) of the contract.

```md
cicero invoke

invoke a clause of the contract

Options:
  --version      Show version number                                 [boolean]
  --verbose, -v                                               [default: false]
  --help         Show help                                           [boolean]
  --template     path to the template                                 [string]
  --sample       path to the contract text                            [string]
  --clauseName   the name of the clause to invoke                     [string]
  --params       path to the parameters                               [string]
  --state        path to the JSON state                               [string]
  --currentTime  set current time                     [string] [default: null]
  --utcOffset    set UTC offset                       [number] [default: null]
  --offline      do not resolve external models     [boolean] [default: false]
  --warnings     print warnings                     [boolean] [default: false]
```

## cicero initialize

`cicero initialize` initializes a clause.

```md
cicero initialize

initialize a clause

Options:
  --version      Show version number                                 [boolean]
  --verbose, -v                                               [default: false]
  --help         Show help                                           [boolean]
  --template     path to the template                                 [string]
  --sample       path to the contract text                            [string]
  --params       path to the parameters                               [string]
  --currentTime  initialize with this current time    [string] [default: null]
  --utcOffset    set UTC offset                       [number] [default: null]
  --offline      do not resolve external models     [boolean] [default: false]
  --warnings     print warnings                     [boolean] [default: false]
```

## cicero archive

`cicero archive` creates a Cicero Template Archive (`.cta`) file from a template
stored in a local directory.

```md
cicero archive

create a template archive

Options:
  --version      Show version number                                 [boolean]
  --verbose, -v                                               [default: false]
  --help         Show help                                           [boolean]
```

~~~
  --template     path to the template                                  [string]
  --target       the target language of the archive   [string] [default: "ergo"]
  --output       file name for new archive               [string] [default: null]
  --warnings     print warnings                       [boolean] [default: false]
```

## cicero compile

`cicero compile` generates code for a target platform. It loads a template from a
directory on disk and then attempts to generate versions of the template model in
the specified format. The available formats include: `Go`, `PlantUML`,
`Typescript`, `Java`, and `JSONSchema`.

```md
cicero compile

generate code for a target platform

Options:
  --version      Show version number                                  [boolean]
  --verbose, -v                                                [default: false]
  --help         Show help                                            [boolean]
  --template     path to the template                                  [string]
  --target       target of the code generation  [string] [default: "JSONSchema"]
  --output       path to the output directory    [string] [default: "./output/"]
  --warnings     print warnings                       [boolean] [default: false]
```

## cicero get

`cicero get` saves local copies of external dependencies.

```md
cicero get

save local copies of external dependencies

Options:
  --version      Show version number                                  [boolean]
  --verbose, -v                                                [default: false]
  --help         Show help                                            [boolean]
  --template     path to the template                                  [string]
  --output       output directory path                                 [string]
```

--------------------------------------------------------------------------------
---
id: version-0.22-ref-concerto-api
title: Concerto API
original_id: ref-concerto-api
---

## Modules

<dl>
<dt><a href="#module_concerto-core">concerto-core</a></dt>
<dd><p>Concerto module. Concerto is a framework for defining domain
specific models.</p>
</dd>
~~~

```
</dl>

## Constants

<dl>
<dt><a href="#levels">levels</a> : <code>Object</code></dt>
<dd><p>Default levels for the npm configuration.</p>
</dd>
<dt><a href="#colorMap">colorMap</a> : <code>Object</code></dt>
<dd><p>Default levels for the npm configuration.</p>
</dd>
</dl>

## Functions

<dl>
<dt><a href="#setCurrentTime">setCurrentTime([currentTime], [utcOffset])</a> ⇒
<code>object</code></dt>
<dd><p>Ensures there is a proper current time</p>
</dd>
<dt><a
href="#randomNumberInRangeWithPrecision">randomNumberInRangeWithPrecision(userMin,
userMax, precision, systemMin, systemMax)</a> ⇒ <code>number</code></dt>
<dd><p>Generate a random number within a given range with
a prescribed precision and inside a global range</p>
</dd>
</dl>

<a name="module_concerto-core"></a>

## concerto-core
Concerto module. Concerto is a framework for defining domain
specific models.


* [concerto-core](#module_concerto-core)
    * [.BaseException](#module_concerto-core.BaseException) ⇐ <code>Error</code>
        * [new BaseException(message, component)](#new_module_concerto-
core.BaseException_new)
    * [.BaseFileException](#module_concerto-core.BaseFileException) ⇐
<code>BaseException</code>
        * [new BaseFileException(message, fileLocation, fullMessage, [fileName],
[component])](#new_module_concerto-core.BaseFileException_new)
        * [.getFileLocation()](#module_concerto-
core.BaseFileException+getFileLocation) ⇒ <code>string</code>
        * [.getShortMessage()](#module_concerto-
core.BaseFileException+getShortMessage) ⇒ <code>string</code>
        * [.getFileName()](#module_concerto-core.BaseFileException+getFileName) ⇒
<code>string</code>
    * [.Concerto](#module_concerto-core.Concerto)
        * [new Concerto(modelManager)](#new_module_concerto-core.Concerto_new)
        * [.validate(obj, [options])](#module_concerto-core.Concerto+validate)
        * [.getModelManager()](#module_concerto-core.Concerto+getModelManager) ⇒
<code>\*</code>
        * [.isObject(obj)](#module_concerto-core.Concerto+isObject) ⇒
<code>boolean</code>
        * [.getTypeDeclaration(obj)](#module_concerto-
core.Concerto+getTypeDeclaration) ⇒ <code>\*</code>
        * [.getIdentifier(obj)](#module_concerto-core.Concerto+getIdentifier) ⇒
```

```
<code>string</code>
        * [.isIdentifiable(obj)](#module_concerto-core.Concerto+isIdentifiable) ⇒
<code>boolean</code>
        * [.isRelationship(obj)](#module_concerto-core.Concerto+isRelationship) ⇒
<code>boolean</code>
        * [.setIdentifier(obj, id)](#module_concerto-core.Concerto+setIdentifier) ⇒
<code>\*</code>
        * [.getFullyQualifiedIdentifier(obj)](#module_concerto-
core.Concerto+getFullyQualifiedIdentifier) ⇒ <code>string</code>
        * [.toURI(obj)](#module_concerto-core.Concerto+toURI) ⇒ <code>string</code>
        * [.fromURI(uri)](#module_concerto-core.Concerto+fromURI) ⇒ <code>\*</code>
        * [.getType(obj)](#module_concerto-core.Concerto+getType) ⇒
<code>string</code>
        * [.getNamespace(obj)](#module_concerto-core.Concerto+getNamespace) ⇒
<code>string</code>
    * [.Factory](#module_concerto-core.Factory)
        * [new Factory(modelManager)](#new_module_concerto-core.Factory_new)
        * _instance_
            * [.newResource(ns, type, [id], [options])](#module_concerto-
core.Factory+newResource) ⇒ <code>Resource</code>
            * [.newConcept(ns, type, [id], [options])](#module_concerto-
core.Factory+newConcept) ⇒ <code>Resource</code>
            * [.newRelationship(ns, type, id)](#module_concerto-
core.Factory+newRelationship) ⇒ <code>Relationship</code>
            * [.newTransaction(ns, type, [id], [options])](#module_concerto-
core.Factory+newTransaction) ⇒ <code>Resource</code>
            * [.newEvent(ns, type, [id], [options])](#module_concerto-
core.Factory+newEvent) ⇒ <code>Resource</code>
        * _static_
            * [.Symbol.hasInstance(object)](#module_concerto-
core.Factory.Symbol.hasInstance) ⇒ <code>boolean</code>
    * [.ModelLoader](#module_concerto-core.ModelLoader)
        * [.loadModelManager(ctoFiles, options)](#module_concerto-
core.ModelLoader.loadModelManager) ⇒ <code>object</code>
        * [.loadModelManagerFromModelFiles(modelFiles, [fileNames], options)]
(#module_concerto-core.ModelLoader.loadModelManagerFromModelFiles) ⇒
<code>object</code>
    * [.ModelManager](#module_concerto-core.ModelManager)
        * [new ModelManager([options])](#new_module_concerto-core.ModelManager_new)
        * _instance_
            * [.accept(visitor, parameters)](#module_concerto-
core.ModelManager+accept) ⇒ <code>Object</code>
            * [.validateModelFile(modelFile, [fileName])](#module_concerto-
core.ModelManager+validateModelFile)
            * [.addModelFile(modelFile, fileName, [disableValidation])]
(#module_concerto-core.ModelManager+addModelFile) ⇒ <code>Object</code>
            * [.updateModelFile(modelFile, [fileName], [disableValidation])]
(#module_concerto-core.ModelManager+updateModelFile) ⇒ <code>Object</code>
            * [.deleteModelFile(namespace)](#module_concerto-
core.ModelManager+deleteModelFile)
            * [.addModelFiles(modelFiles, [fileNames], [disableValidation])]
(#module_concerto-core.ModelManager+addModelFiles) ⇒
<code>Array.&lt;Object&gt;</code>
            * [.validateModelFiles()](#module_concerto-
core.ModelManager+validateModelFiles)
            * [.updateExternalModels([options], [modelFileDownloader])]
(#module_concerto-core.ModelManager+updateExternalModels) ⇒ <code>Promise</code>
            * [.writeModelsToFileSystem(path, [options])](#module_concerto-
core.ModelManager+writeModelsToFileSystem)
```

```
            * [.getModels([options])](#module_concerto-core.ModelManager+getModels)
⇒ <code>Array.&lt;{name:string, content:string}&gt;</code>
            * [.clearModelFiles()](#module_concerto-
core.ModelManager+clearModelFiles)
            * [.getModelFile(namespace)](#module_concerto-
core.ModelManager+getModelFile) ⇒ <code>ModelFile</code>
            * [.getNamespaces()](#module_concerto-core.ModelManager+getNamespaces)
⇒ <code>Array.&lt;string&gt;</code>
            * [.getAssetDeclarations()](#module_concerto-
core.ModelManager+getAssetDeclarations) ⇒
<code>Array.&lt;AssetDeclaration&gt;</code>
            * [.getTransactionDeclarations()](#module_concerto-
core.ModelManager+getTransactionDeclarations) ⇒
<code>Array.&lt;TransactionDeclaration&gt;</code>
            * [.getEventDeclarations()](#module_concerto-
core.ModelManager+getEventDeclarations) ⇒
<code>Array.&lt;EventDeclaration&gt;</code>
            * [.getParticipantDeclarations()](#module_concerto-
core.ModelManager+getParticipantDeclarations) ⇒
<code>Array.&lt;ParticipantDeclaration&gt;</code>
            * [.getEnumDeclarations()](#module_concerto-
core.ModelManager+getEnumDeclarations) ⇒ <code>Array.&lt;EnumDeclaration&gt;</code>
            * [.getConceptDeclarations()](#module_concerto-
core.ModelManager+getConceptDeclarations) ⇒
<code>Array.&lt;ConceptDeclaration&gt;</code>
            * [.getFactory()](#module_concerto-core.ModelManager+getFactory) ⇒
<code>Factory</code>
            * [.getSerializer()](#module_concerto-core.ModelManager+getSerializer)
⇒ <code>Serializer</code>
            * [.getDecoratorFactories()](#module_concerto-
core.ModelManager+getDecoratorFactories) ⇒
<code>Array.&lt;DecoratorFactory&gt;</code>
            * [.addDecoratorFactory(factory)](#module_concerto-
core.ModelManager+addDecoratorFactory)
            * [.derivesFrom(fqt1, fqt2)](#module_concerto-
core.ModelManager+derivesFrom) ⇒ <code>boolean</code>
        * _static_
            * [.Symbol.hasInstance(object)](#module_concerto-
core.ModelManager.Symbol.hasInstance) ⇒ <code>boolean</code>
    * [.SecurityException](#module_concerto-core.SecurityException) ⇐
<code>BaseException</code>
        * [new SecurityException(message)](#new_module_concerto-
core.SecurityException_new)
    * [.Serializer](#module_concerto-core.Serializer)
        * [new Serializer(factory, modelManager, [options])](#new_module_concerto-
core.Serializer_new)
        * _instance_
            * [.setDefaultOptions(newDefaultOptions)](#module_concerto-
core.Serializer+setDefaultOptions)
            * [.toJSON(resource, [options])](#module_concerto-
core.Serializer+toJSON) ⇒ <code>Object</code>
            * [.fromJSON(jsonObject, options)](#module_concerto-
core.Serializer+fromJSON) ⇒ <code>Resource</code>
        * _static_
            * [.Symbol.hasInstance(object)](#module_concerto-
core.Serializer.Symbol.hasInstance) ⇒ <code>boolean</code>
    * [.TypeNotFoundException](#module_concerto-core.TypeNotFoundException) ⇐
<code>BaseException</code>
        * [new TypeNotFoundException(typeName, [message], component)]
```

```
(#new_module_concerto-core.TypeNotFoundException_new)
        * [.getTypeName()](#module_concerto-core.TypeNotFoundException+getTypeName)
⇒ <code>string</code>
    * [.AssetDeclaration](#module_concerto-core.AssetDeclaration) ⇐
<code>ClassDeclaration</code>
        * [new AssetDeclaration(modelFile, ast)](#new_module_concerto-
core.AssetDeclaration_new)
        * [.Symbol.hasInstance(object)](#module_concerto-
core.AssetDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>
    * *[.ClassDeclaration](#module_concerto-core.ClassDeclaration)*
        * *[new ClassDeclaration(modelFile, ast)](#new_module_concerto-
core.ClassDeclaration_new)*
        * _instance_
            * *[._resolveSuperType()](#module_concerto-
core.ClassDeclaration+_resolveSuperType) ⇒ <code>ClassDeclaration</code>*
            * *[.isAbstract()](#module_concerto-core.ClassDeclaration+isAbstract) ⇒
<code>boolean</code>*
            * *[.isEnum()](#module_concerto-core.ClassDeclaration+isEnum) ⇒
<code>boolean</code>*
            * *[.isConcept()](#module_concerto-core.ClassDeclaration+isConcept) ⇒
<code>boolean</code>*
            * *[.isEvent()](#module_concerto-core.ClassDeclaration+isEvent) ⇒
<code>boolean</code>*
            * *[.getName()](#module_concerto-core.ClassDeclaration+getName) ⇒
<code>string</code>*
            * *[.getNamespace()](#module_concerto-
core.ClassDeclaration+getNamespace) ⇒ <code>string</code>*
            * *[.getFullyQualifiedName()](#module_concerto-
core.ClassDeclaration+getFullyQualifiedName) ⇒ <code>string</code>*
            * *[.isIdentified()](#module_concerto-
core.ClassDeclaration+isIdentified) ⇒ <code>Boolean</code>*
            * *[.isSystemIdentified()](#module_concerto-
core.ClassDeclaration+isSystemIdentified) ⇒ <code>Boolean</code>*
            * *[.isExplicitlyIdentified()](#module_concerto-
core.ClassDeclaration+isExplicitlyIdentified) ⇒ <code>Boolean</code>*
            * *[.getIdentifierFieldName()](#module_concerto-
core.ClassDeclaration+getIdentifierFieldName) ⇒ <code>string</code>*
            * *[.getOwnProperty(name)](#module_concerto-
core.ClassDeclaration+getOwnProperty) ⇒ <code>Property</code>*
            * *[.getOwnProperties()](#module_concerto-
core.ClassDeclaration+getOwnProperties) ⇒ <code>Array.&lt;Property&gt;</code>*
            * *[.getSuperType()](#module_concerto-
core.ClassDeclaration+getSuperType) ⇒ <code>string</code>*
            * *[.getSuperTypeDeclaration()](#module_concerto-
core.ClassDeclaration+getSuperTypeDeclaration) ⇒ <code>ClassDeclaration</code>*
            * *[.getAssignableClassDeclarations()](#module_concerto-
core.ClassDeclaration+getAssignableClassDeclarations) ⇒
<code>Array.&lt;ClassDeclaration&gt;</code>*
            * *[.getAllSuperTypeDeclarations()](#module_concerto-
core.ClassDeclaration+getAllSuperTypeDeclarations) ⇒
<code>Array.&lt;ClassDeclaration&gt;</code>*
            * *[.getProperty(name)](#module_concerto-
core.ClassDeclaration+getProperty) ⇒ <code>Property</code>*
            * *[.getProperties()](#module_concerto-
core.ClassDeclaration+getProperties) ⇒ <code>Array.&lt;Property&gt;</code>*
            * *[.getNestedProperty(propertyPath)](#module_concerto-
core.ClassDeclaration+getNestedProperty) ⇒ <code>Property</code>*
            * *[.toString()](#module_concerto-core.ClassDeclaration+toString) ⇒
<code>String</code>*
```

```
        * _static_
            * *[.Symbol.hasInstance(object)](#module_concerto-
core.ClassDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>*
    * [.ConceptDeclaration](#module_concerto-core.ConceptDeclaration) ⇐
<code>ClassDeclaration</code>
        * [new ConceptDeclaration(modelFile, ast)](#new_module_concerto-
core.ConceptDeclaration_new)
        * _instance_
            * [.isConcept()](#module_concerto-core.ConceptDeclaration+isConcept) ⇒
<code>boolean</code>
        * _static_
            * [.Symbol.hasInstance(object)](#module_concerto-
core.ConceptDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>
    * [.Decorator](#module_concerto-core.Decorator)
        * [new Decorator(parent, ast)](#new_module_concerto-core.Decorator_new)
        * [.getParent()](#module_concerto-core.Decorator+getParent) ⇒
<code>ClassDeclaration</code> \| <code>Property</code>
        * [.getName()](#module_concerto-core.Decorator+getName) ⇒
<code>string</code>
        * [.getArguments()](#module_concerto-core.Decorator+getArguments) ⇒
<code>Array.&lt;object&gt;</code>
    * [.DecoratorFactory](#module_concerto-core.DecoratorFactory)
        * *[.newDecorator(parent, ast)](#module_concerto-
core.DecoratorFactory+newDecorator) ⇒ <code>Decorator</code>*
    * [.EnumDeclaration](#module_concerto-core.EnumDeclaration) ⇐
<code>ClassDeclaration</code>
        * [new EnumDeclaration(modelFile, ast)](#new_module_concerto-
core.EnumDeclaration_new)
        * _instance_
            * [.isEnum()](#module_concerto-core.EnumDeclaration+isEnum) ⇒
<code>boolean</code>
            * [.toString()](#module_concerto-core.EnumDeclaration+toString) ⇒
<code>String</code>
        * _static_
            * [.Symbol.hasInstance(object)](#module_concerto-
core.EnumDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>
    * [.EnumValueDeclaration](#module_concerto-core.EnumValueDeclaration) ⇐
<code>Property</code>
        * [new EnumValueDeclaration(parent, ast)](#new_module_concerto-
core.EnumValueDeclaration_new)
        * [.Symbol.hasInstance(object)](#module_concerto-
core.EnumValueDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>
    * [.EventDeclaration](#module_concerto-core.EventDeclaration) ⇐
<code>ClassDeclaration</code>
        * [new EventDeclaration(modelFile, ast)](#new_module_concerto-
core.EventDeclaration_new)
        * _instance_
            * [.isEvent()](#module_concerto-core.EventDeclaration+isEvent) ⇒
<code>boolean</code>
        * _static_
            * [.Symbol.hasInstance(object)](#module_concerto-
core.EventDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>
    * *[.IdentifiedDeclaration](#module_concerto-core.IdentifiedDeclaration) ⇐
<code>ClassDeclaration</code>*
        * *[new IdentifiedDeclaration(modelFile, ast)](#new_module_concerto-
core.IdentifiedDeclaration_new)*
        * *[.Symbol.hasInstance(object)](#module_concerto-
core.IdentifiedDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>*
    * [.IllegalModelException](#module_concerto-core.IllegalModelException) ⇐
```

```
<code>BaseFileException</code>
        * [new IllegalModelException(message, [modelFile], [fileLocation],
[component])](#new_module_concerto-core.IllegalModelException_new)
    * [.Introspector](#module_concerto-core.Introspector)
        * [new Introspector(modelManager)](#new_module_concerto-
core.Introspector_new)
        * [.getClassDeclarations()](#module_concerto-
core.Introspector+getClassDeclarations) ⇒
<code>Array.&lt;ClassDeclaration&gt;</code>
        * [.getClassDeclaration(fullyQualifiedTypeName)](#module_concerto-
core.Introspector+getClassDeclaration) ⇒ <code>ClassDeclaration</code>
    * [.ModelFile](#module_concerto-core.ModelFile)
        * [new ModelFile(modelManager, definitions, [fileName])]
(#new_module_concerto-core.ModelFile_new)
        * _instance_
            * [.isSystemModelFile()](#module_concerto-
core.ModelFile+isSystemModelFile) ⇒ <code>Boolean</code>
            * [.isExternal()](#module_concerto-core.ModelFile+isExternal) ⇒
<code>boolean</code>
            * [.getModelManager()](#module_concerto-core.ModelFile+getModelManager)
⇒ <code>ModelManager</code>
            * [.getImports()](#module_concerto-core.ModelFile+getImports) ⇒
<code>Array.&lt;string&gt;</code>
            * [.isDefined(type)](#module_concerto-core.ModelFile+isDefined) ⇒
<code>boolean</code>
            * [.getLocalType(type)](#module_concerto-core.ModelFile+getLocalType) ⇒
<code>ClassDeclaration</code>
            * [.getAssetDeclaration(name)](#module_concerto-
core.ModelFile+getAssetDeclaration) ⇒ <code>AssetDeclaration</code>
            * [.getTransactionDeclaration(name)](#module_concerto-
core.ModelFile+getTransactionDeclaration) ⇒ <code>TransactionDeclaration</code>
            * [.getEventDeclaration(name)](#module_concerto-
core.ModelFile+getEventDeclaration) ⇒ <code>EventDeclaration</code>
            * [.getParticipantDeclaration(name)](#module_concerto-
core.ModelFile+getParticipantDeclaration) ⇒ <code>ParticipantDeclaration</code>
            * [.getNamespace()](#module_concerto-core.ModelFile+getNamespace) ⇒
<code>string</code>
            * [.getName()](#module_concerto-core.ModelFile+getName) ⇒
<code>string</code>
            * [.getAssetDeclarations()](#module_concerto-
core.ModelFile+getAssetDeclarations) ⇒ <code>Array.&lt;AssetDeclaration&gt;</code>
            * [.getTransactionDeclarations()](#module_concerto-
core.ModelFile+getTransactionDeclarations) ⇒
<code>Array.&lt;TransactionDeclaration&gt;</code>
            * [.getEventDeclarations()](#module_concerto-
core.ModelFile+getEventDeclarations) ⇒ <code>Array.&lt;EventDeclaration&gt;</code>
            * [.getParticipantDeclarations()](#module_concerto-
core.ModelFile+getParticipantDeclarations) ⇒
<code>Array.&lt;ParticipantDeclaration&gt;</code>
            * [.getConceptDeclarations()](#module_concerto-
core.ModelFile+getConceptDeclarations) ⇒
<code>Array.&lt;ConceptDeclaration&gt;</code>
            * [.getEnumDeclarations()](#module_concerto-
core.ModelFile+getEnumDeclarations) ⇒ <code>Array.&lt;EnumDeclaration&gt;</code>
            * [.getDeclarations(type)](#module_concerto-
core.ModelFile+getDeclarations) ⇒ <code>Array.&lt;ClassDeclaration&gt;</code>
            * [.getAllDeclarations()](#module_concerto-
core.ModelFile+getAllDeclarations) ⇒ <code>Array.&lt;ClassDeclaration&gt;</code>
            * [.getDefinitions()](#module_concerto-core.ModelFile+getDefinitions) ⇒
```

```
<code>string</code>
            * [.getConcertoVersion()](#module_concerto-
core.ModelFile+getConcertoVersion) ⇒ <code>string</code>
        * _static_
            * [.Symbol.hasInstance(object)](#module_concerto-
core.ModelFile.Symbol.hasInstance) ⇒ <code>boolean</code>
    * [.ParticipantDeclaration](#module_concerto-core.ParticipantDeclaration) ⇐
<code>ClassDeclaration</code>
        * [new ParticipantDeclaration(modelFile, ast)](#new_module_concerto-
core.ParticipantDeclaration_new)
        * [.Symbol.hasInstance(object)](#module_concerto-
core.ParticipantDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>
    * [.Property](#module_concerto-core.Property)
        * [new Property(parent, ast)](#new_module_concerto-core.Property_new)
        * _instance_
            * [.getParent()](#module_concerto-core.Property+getParent) ⇒
<code>ClassDeclaration</code>
            * [.getName()](#module_concerto-core.Property+getName) ⇒
<code>string</code>
            * [.getType()](#module_concerto-core.Property+getType) ⇒
<code>string</code>
            * [.isOptional()](#module_concerto-core.Property+isOptional) ⇒
<code>boolean</code>
            * [.getFullyQualifiedTypeName()](#module_concerto-
core.Property+getFullyQualifiedTypeName) ⇒ <code>string</code>
            * [.getFullyQualifiedName()](#module_concerto-
core.Property+getFullyQualifiedName) ⇒ <code>string</code>
            * [.getNamespace()](#module_concerto-core.Property+getNamespace) ⇒
<code>string</code>
            * [.isArray()](#module_concerto-core.Property+isArray) ⇒
<code>boolean</code>
            * [.isTypeEnum()](#module_concerto-core.Property+isTypeEnum) ⇒
<code>boolean</code>
            * [.isPrimitive()](#module_concerto-core.Property+isPrimitive) ⇒
<code>boolean</code>
        * _static_
            * [.Symbol.hasInstance(object)](#module_concerto-
core.Property.Symbol.hasInstance) ⇒ <code>boolean</code>
    * [.RelationshipDeclaration](#module_concerto-core.RelationshipDeclaration) ⇐
<code>Property</code>
        * [new RelationshipDeclaration(parent, ast)](#new_module_concerto-
core.RelationshipDeclaration_new)
        * _instance_
            * [.toString()](#module_concerto-core.RelationshipDeclaration+toString)
⇒ <code>String</code>
        * _static_
            * [.Symbol.hasInstance(object)](#module_concerto-
core.RelationshipDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>
    * [.TransactionDeclaration](#module_concerto-core.TransactionDeclaration) ⇐
<code>ClassDeclaration</code>
        * [new TransactionDeclaration(modelFile, ast)](#new_module_concerto-
core.TransactionDeclaration_new)
        * [.Symbol.hasInstance(object)](#module_concerto-
core.TransactionDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>
    * [.Resource](#module_concerto-core.Resource) ⇐ <code>Identifiable</code>
        * [new Resource(modelManager, classDeclaration, ns, type, id, timestamp)]
(#new_module_concerto-core.Resource_new)
        * [.toString()](#module_concerto-core.Resource+toString) ⇒
<code>String</code>
```

```
        * [.isResource()](#module_concerto-core.Resource+isResource) ⇒
<code>boolean</code>
        * [.isConcept()](#module_concerto-core.Resource+isConcept) ⇒
<code>boolean</code>
        * [.isIdentifiable()](#module_concerto-core.Resource+isIdentifiable) ⇒
<code>boolean</code>
        * [.toJSON()](#module_concerto-core.Resource+toJSON) ⇒ <code>Object</code>
    * [.TypedStack](#module_concerto-core.TypedStack)
        * [new TypedStack(resource)](#new_module_concerto-core.TypedStack_new)
        * [.push(obj, expectedType)](#module_concerto-core.TypedStack+push)
        * [.pop(expectedType)](#module_concerto-core.TypedStack+pop) ⇒
<code>Object</code>
        * [.peek(expectedType)](#module_concerto-core.TypedStack+peek) ⇒
<code>Object</code>
        * [.clear()](#module_concerto-core.TypedStack+clear)

<a name="module_concerto-core.BaseException"></a>

### concerto-core.BaseException ⇐ <code>Error</code>
A base class for all Concerto exceptions

**Kind**: static class of [<code>concerto-core</code>](#module_concerto-core)
**Extends**: <code>Error</code>
<a name="new_module_concerto-core.BaseException_new"></a>

#### new BaseException(message, component)
Create the BaseException.


| Param | Type | Description |
| --- | --- | --- |
| message | <code>string</code> | The exception message. |
| component | <code>string</code> | The optional component which throws this error.
|

<a name="module_concerto-core.BaseFileException"></a>

### concerto-core.BaseFileException ⇐ <code>BaseException</code>
Exception throws when a Concerto file is semantically invalid

**Kind**: static class of [<code>concerto-core</code>](#module_concerto-core)
**Extends**: <code>BaseException</code>
**See**: [BaseException](BaseException)

* [.BaseFileException](#module_concerto-core.BaseFileException) ⇐
<code>BaseException</code>
    * [new BaseFileException(message, fileLocation, fullMessage, [fileName],
[component])](#new_module_concerto-core.BaseFileException_new)
    * [.getFileLocation()](#module_concerto-core.BaseFileException+getFileLocation)
⇒ <code>string</code>
    * [.getShortMessage()](#module_concerto-core.BaseFileException+getShortMessage)
⇒ <code>string</code>
    * [.getFileName()](#module_concerto-core.BaseFileException+getFileName) ⇒
<code>string</code>

<a name="new_module_concerto-core.BaseFileException_new"></a>

#### new BaseFileException(message, fileLocation, fullMessage, [fileName],
[component])
```

Create an BaseFileException

Param	Type	Description
message	<code>string</code>	the message for the exception
fileLocation	<code>string</code>	the optional file location associated with the exception
fullMessage	<code>string</code>	the optional full message text
[fileName]	<code>string</code>	the file name
[component]	<code>string</code>	the component which throws this error

<a name="module_concerto-core.BaseFileException+getFileLocation"></a>

#### baseFileException.getFileLocation() ⇒ <code>string</code>
Returns the file location associated with the exception or null

Kind: instance method of [<code>BaseFileException</code>](#module_concerto-core.BaseFileException)
Returns: <code>string</code> - the optional location associated with the exception
<a name="module_concerto-core.BaseFileException+getShortMessage"></a>

#### baseFileException.getShortMessage() ⇒ <code>string</code>
Returns the error message without the location of the error

Kind: instance method of [<code>BaseFileException</code>](#module_concerto-core.BaseFileException)
Returns: <code>string</code> - the error message
<a name="module_concerto-core.BaseFileException+getFileName"></a>

#### baseFileException.getFileName() ⇒ <code>string</code>
Returns the fileName for the error

Kind: instance method of [<code>BaseFileException</code>](#module_concerto-core.BaseFileException)
Returns: <code>string</code> - the file name or null
<a name="module_concerto-core.Concerto"></a>

### concerto-core.Concerto
Runtime API for Concerto.

Kind: static class of [<code>concerto-core</code>](#module_concerto-core)

* [.Concerto](#module_concerto-core.Concerto)
    * [new Concerto(modelManager)](#new_module_concerto-core.Concerto_new)
    * [.validate(obj, [options])](#module_concerto-core.Concerto+validate)
    * [.getModelManager()](#module_concerto-core.Concerto+getModelManager) ⇒ <code>*</code>
    * [.isObject(obj)](#module_concerto-core.Concerto+isObject) ⇒ <code>boolean</code>
    * [.getTypeDeclaration(obj)](#module_concerto-core.Concerto+getTypeDeclaration) ⇒ <code>*</code>
    * [.getIdentifier(obj)](#module_concerto-core.Concerto+getIdentifier) ⇒ <code>string</code>
    * [.isIdentifiable(obj)](#module_concerto-core.Concerto+isIdentifiable) ⇒ <code>boolean</code>
    * [.isRelationship(obj)](#module_concerto-core.Concerto+isRelationship) ⇒ <code>boolean</code>

```
    * [.setIdentifier(obj, id)](#module_concerto-core.Concerto+setIdentifier) ⇒
<code>\*</code>
    * [.getFullyQualifiedIdentifier(obj)](#module_concerto-
core.Concerto+getFullyQualifiedIdentifier) ⇒ <code>string</code>
    * [.toURI(obj)](#module_concerto-core.Concerto+toURI) ⇒ <code>string</code>
    * [.fromURI(uri)](#module_concerto-core.Concerto+fromURI) ⇒ <code>\*</code>
    * [.getType(obj)](#module_concerto-core.Concerto+getType) ⇒ <code>string</code>
    * [.getNamespace(obj)](#module_concerto-core.Concerto+getNamespace) ⇒
<code>string</code>

<a name="new_module_concerto-core.Concerto_new"></a>

#### new Concerto(modelManager)
Create a Concerto instance.


| Param | Type | Description |
| --- | --- | --- |
| modelManager | <code>\*</code> | The this.modelManager to use for validation etc.
|

<a name="module_concerto-core.Concerto+validate"></a>

#### concerto.validate(obj, [options])
Validates the instance against its model.

**Kind**: instance method of [<code>Concerto</code>](#module_concerto-
core.Concerto)
**Throws**:

- <code>Error</code> - if the instance if invalid with respect to the model


| Param | Type | Description |
| --- | --- | --- |
| obj | <code>\*</code> | the input object |
| [options] | <code>\*</code> | the validation options |

<a name="module_concerto-core.Concerto+getModelManager"></a>

#### concerto.getModelManager() ⇒ <code>\*</code>
Returns the model manager

**Kind**: instance method of [<code>Concerto</code>](#module_concerto-
core.Concerto)
**Returns**: <code>\*</code> - the model manager associated with this Concerto
class
<a name="module_concerto-core.Concerto+isObject"></a>

#### concerto.isObject(obj) ⇒ <code>boolean</code>
Returns true if the input object is a Concerto object

**Kind**: instance method of [<code>Concerto</code>](#module_concerto-
core.Concerto)
**Returns**: <code>boolean</code> - true if the object has a $class attribute

| Param | Type | Description |
| --- | --- | --- |
| obj | <code>\*</code> | the input object |
```

<a name="module_concerto-core.Concerto+getTypeDeclaration"></a>

#### concerto.getTypeDeclaration(obj) ⇒ <code>*</code>
Returns the ClassDeclaration for an object, or throws an exception

Kind: instance method of [<code>Concerto</code>](#module_concerto-core.Concerto)
Returns: <code>*</code> - the ClassDeclaration for the type
Throw: <code>Error</code> an error if the object does not have a $class attribute

Param	Type	Description
obj	<code>*</code>	the input object

<a name="module_concerto-core.Concerto+getIdentifier"></a>

#### concerto.getIdentifier(obj) ⇒ <code>string</code>
Gets the identifier for an object

Kind: instance method of [<code>Concerto</code>](#module_concerto-core.Concerto)
Returns: <code>string</code> - The identifier for this object

Param	Type	Description
obj	<code>*</code>	the input object

<a name="module_concerto-core.Concerto+isIdentifiable"></a>

#### concerto.isIdentifiable(obj) ⇒ <code>boolean</code>
Returns true if the object has an identifier

Kind: instance method of [<code>Concerto</code>](#module_concerto-core.Concerto)
Returns: <code>boolean</code> - is the object has been defined with an identifier in the model

Param	Type	Description
obj	<code>*</code>	the input object

<a name="module_concerto-core.Concerto+isRelationship"></a>

#### concerto.isRelationship(obj) ⇒ <code>boolean</code>
Returns true if the object is a relationship. Relationships are strings
of the form: 'resource:org.accordproject.Order#001' (a relationship)
to the 'Order' identifiable, with the id 001.

Kind: instance method of [<code>Concerto</code>](#module_concerto-core.Concerto)
Returns: <code>boolean</code> - true if the object is a relationship

Param	Type	Description
obj	<code>*</code>	the input object

<a name="module_concerto-core.Concerto+setIdentifier"></a>

```
#### concerto.setIdentifier(obj, id) ⇒ <code>\*</code>
Set the identifier for an object. This method does *not* mutate the
input object, use the return object.

**Kind**: instance method of [<code>Concerto</code>](#module_concerto-
core.Concerto)
**Returns**: <code>\*</code> - the input object with the identifier set

| Param | Type | Description |
| --- | --- | --- |
| obj | <code>\*</code> | the input object |
| id | <code>string</code> | the new identifier |

<a name="module_concerto-core.Concerto+getFullyQualifiedIdentifier"></a>

#### concerto.getFullyQualifiedIdentifier(obj) ⇒ <code>string</code>
Returns the fully qualified identifier for an object

**Kind**: instance method of [<code>Concerto</code>](#module_concerto-
core.Concerto)
**Returns**: <code>string</code> - the fully qualified identifier

| Param | Type | Description |
| --- | --- | --- |
| obj | <code>\*</code> | the input object |

<a name="module_concerto-core.Concerto+toURI"></a>

#### concerto.toURI(obj) ⇒ <code>string</code>
Returns a URI for an object

**Kind**: instance method of [<code>Concerto</code>](#module_concerto-
core.Concerto)
**Returns**: <code>string</code> - the URI for the object

| Param | Type | Description |
| --- | --- | --- |
| obj | <code>\*</code> | the input object |

<a name="module_concerto-core.Concerto+fromURI"></a>

#### concerto.fromURI(uri) ⇒ <code>\*</code>
Parses a resource URI into typeDeclaration and id components.

**Kind**: instance method of [<code>Concerto</code>](#module_concerto-
core.Concerto)
**Returns**: <code>\*</code> - an object with typeDeclaration and id attributes
**Throws**:

- <code>Error</code> if the URI is invalid or the type does not exist
in the model manager


| Param | Type | Description |
| --- | --- | --- |
| uri | <code>string</code> | the input URI |

<a name="module_concerto-core.Concerto+getType"></a>
```

```
#### concerto.getType(obj) ⇒ <code>string</code>
Returns the short type name

**Kind**: instance method of [<code>Concerto</code>](#module_concerto-
core.Concerto)
**Returns**: <code>string</code> - the short type name

| Param | Type | Description |
| --- | --- | --- |
| obj | <code>\*</code> | the input object |

<a name="module_concerto-core.Concerto+getNamespace"></a>

#### concerto.getNamespace(obj) ⇒ <code>string</code>
Returns the namespace for the object

**Kind**: instance method of [<code>Concerto</code>](#module_concerto-
core.Concerto)
**Returns**: <code>string</code> - the namespace

| Param | Type | Description |
| --- | --- | --- |
| obj | <code>\*</code> | the input object |

<a name="module_concerto-core.Factory"></a>

### concerto-core.Factory
Use the Factory to create instances of Resource: transactions, participants
and assets.

**Kind**: static class of [<code>concerto-core</code>](#module_concerto-core)

* [.Factory](#module_concerto-core.Factory)
    * [new Factory(modelManager)](#new_module_concerto-core.Factory_new)
    * _instance_
        * [.newResource(ns, type, [id], [options])](#module_concerto-
core.Factory+newResource) ⇒ <code>Resource</code>
        * [.newConcept(ns, type, [id], [options])](#module_concerto-
core.Factory+newConcept) ⇒ <code>Resource</code>
        * [.newRelationship(ns, type, id)](#module_concerto-
core.Factory+newRelationship) ⇒ <code>Relationship</code>
        * [.newTransaction(ns, type, [id], [options])](#module_concerto-
core.Factory+newTransaction) ⇒ <code>Resource</code>
        * [.newEvent(ns, type, [id], [options])](#module_concerto-
core.Factory+newEvent) ⇒ <code>Resource</code>
    * _static_
        * [.Symbol.hasInstance(object)](#module_concerto-
core.Factory.Symbol.hasInstance) ⇒ <code>boolean</code>

<a name="new_module_concerto-core.Factory_new"></a>

#### new Factory(modelManager)
Create the factory.


| Param | Type | Description |
| --- | --- | --- |
| modelManager | <code>ModelManager</code> | The ModelManager to use for this
```

registry |

<a name="module_concerto-core.Factory+newResource"></a>

#### factory.newResource(ns, type, [id], [options]) ⇒ <code>Resource</code>
Create a new Resource with a given namespace, type name and id

Kind: instance method of [<code>Factory</code>](#module_concerto-core.Factory)
Returns: <code>Resource</code> - the new instance
Throws:

- <code>TypeNotFoundException</code> if the type is not registered with the
ModelManager

Param	Type	Description
ns	<code>String</code>	the namespace of the Resource
type	<code>String</code>	the type of the Resource
[id]	<code>String</code>	an optional string identifier
[options]	<code>Object</code>	an optional set of options
[options.disableValidation]	<code>boolean</code>	pass true if you want the
factory to return a Resource instead of a [ValidatedResource]
(ValidatedResource). Defaults to false.
[options.generate]	<code>String</code>	Pass one of: <dl>
<dt>sample</dt><dd>return a resource instance with generated sample data.</dd>
<dt>empty</dt><dd>return a resource instance with empty property values.</dd></dl>
[options.includeOptionalFields]	<code>boolean</code>	if
<code>options.generate</code> is specified, whether optional fields should be
generated.

<a name="module_concerto-core.Factory+newConcept"></a>

#### factory.newConcept(ns, type, [id], [options]) ⇒ <code>Resource</code>
Create a new Concept with a given namespace and type name

Kind: instance method of [<code>Factory</code>](#module_concerto-core.Factory)
Returns: <code>Resource</code> - the new instance
Throws:

- <code>TypeNotFoundException</code> if the type is not registered with the
ModelManager

Param	Type	Description
ns	<code>String</code>	the namespace of the Concept
type	<code>String</code>	the type of the Concept
[id]	<code>String</code>	an optional string identifier
[options]	<code>Object</code>	an optional set of options
[options.disableValidation]	<code>boolean</code>	pass true if you want the
factory to return a Concept instead of a [ValidatedConcept]
(ValidatedConcept). Defaults to false.
[options.generate]	<code>String</code>	Pass one of: <dl>
<dt>sample</dt><dd>return a resource instance with generated sample data.</dd>
<dt>empty</dt><dd>return a resource instance with empty property values.</dd></dl>
[options.includeOptionalFields]	<code>boolean</code>	if

<code>options.generate</code> is specified, whether optional fields should be
generated. |

<a name="module_concerto-core.Factory+newRelationship"></a>

#### factory.newRelationship(ns, type, id) ⇒ <code>Relationship</code>
Create a new Relationship with a given namespace, type and identifier.
A relationship is a typed pointer to an instance. I.e the relationship
with `namespace = 'org.example'`, `type = 'Vehicle'` and `id = 'ABC' creates`
a pointer that points at an instance of org.example.Vehicle with the id
ABC.

Kind: instance method of [<code>Factory</code>](#module_concerto-core.Factory)
Returns: <code>Relationship</code> - - the new relationship instance
Throws:

- <code>TypeNotFoundException</code> if the type is not registered with the
ModelManager

Param	Type	Description
ns	<code>String</code>	the namespace of the Resource
type	<code>String</code>	the type of the Resource
id	<code>String</code>	the identifier

<a name="module_concerto-core.Factory+newTransaction"></a>

#### factory.newTransaction(ns, type, [id], [options]) ⇒ <code>Resource</code>
Create a new transaction object. The identifier of the transaction is set to a
UUID.

Kind: instance method of [<code>Factory</code>](#module_concerto-core.Factory)
Returns: <code>Resource</code> - A resource for the new transaction.

Param	Type	Description
ns	<code>String</code>	the namespace of the transaction.
type	<code>String</code>	the type of the transaction.
[id]	<code>String</code>	an optional string identifier
[options]	<code>Object</code>	an optional set of options
[options.generate]	<code>String</code>	Pass one of: <dl>
<dt>sample</dt><dd>return a resource instance with generated sample data.</dd>
<dt>empty</dt><dd>return a resource instance with empty property values.</dd></dl>
[options.includeOptionalFields]	<code>boolean</code>	if
<code>options.generate</code> is specified, whether optional fields should be
generated.

<a name="module_concerto-core.Factory+newEvent"></a>

#### factory.newEvent(ns, type, [id], [options]) ⇒ <code>Resource</code>
Create a new event object. The identifier of the event is
set to a UUID.

Kind: instance method of [<code>Factory</code>](#module_concerto-core.Factory)
Returns: <code>Resource</code> - A resource for the new event.

| Param | Type | Description |

| --- | --- | --- |
ns	<code>String</code>	the namespace of the event.
type	<code>String</code>	the type of the event.
[id]	<code>String</code>	an optional string identifier
[options]	<code>Object</code>	an optional set of options
[options.generate]	<code>String</code>	Pass one of: <dl> <dt>sample</dt><dd>return a resource instance with generated sample data.</dd> <dt>empty</dt><dd>return a resource instance with empty property values.</dd></dl>
[options.includeOptionalFields]	<code>boolean</code>	if <code>options.generate</code> is specified, whether optional fields should be generated.

<a name="module_concerto-core.Factory.Symbol.hasInstance"></a>

#### Factory.Symbol.hasInstance(object) ⇒ <code>boolean</code>
Alternative instanceof that is reliable across different module instances

Kind: static method of [<code>Factory</code>](#module_concerto-core.Factory)
Returns: <code>boolean</code> - - True, if the object is an instance of a Factory
See: https://github.com/hyperledger/composer-concerto/issues/47

Param	Type	Description
object	<code>object</code>	The object to test against

<a name="module_concerto-core.ModelLoader"></a>

### concerto-core.ModelLoader
Create a ModelManager from model files, with an optional system model.

If a ctoFile is not provided, the Accord Project system model is used.

Kind: static class of [<code>concerto-core</code>](#module_concerto-core)

* [.ModelLoader](#module_concerto-core.ModelLoader)
    * [.loadModelManager(ctoFiles, options)](#module_concerto-core.ModelLoader.loadModelManager) ⇒ <code>object</code>
    * [.loadModelManagerFromModelFiles(modelFiles, [fileNames], options)](#module_concerto-core.ModelLoader.loadModelManagerFromModelFiles) ⇒ <code>object</code>

<a name="module_concerto-core.ModelLoader.loadModelManager"></a>

#### ModelLoader.loadModelManager(ctoFiles, options) ⇒ <code>object</code>
Load models in a new model manager

Kind: static method of [<code>ModelLoader</code>](#module_concerto-core.ModelLoader)
Returns: <code>object</code> - the model manager

Param	Type	Description
ctoFiles	<code>Array.&lt;string&gt;</code>	the CTO files (can be local file paths or URLs)
options	<code>object</code>	optional parameters
[options.offline]	<code>boolean</code>	do not resolve external models
[options.utcOffset]	<code>number</code>	UTC Offset for this execution

<a name="module_concerto-core.ModelLoader.loadModelManagerFromModelFiles"></a>

#### ModelLoader.loadModelManagerFromModelFiles(modelFiles, [fileNames], options) ⇒ <code>object</code>
Load system and models in a new model manager from model files objects

Kind: static method of [<code>ModelLoader</code>](#module_concerto-core.ModelLoader)
Returns: <code>object</code> - the model manager

Param	Type	Description
modelFiles	<code>Array.&lt;object&gt;</code>	An array of Concerto files as strings or ModelFile objects.
[fileNames]	<code>Array.&lt;string&gt;</code>	An optional array of file names to associate with the model files
options	<code>object</code>	optional parameters
[options.offline]	<code>boolean</code>	do not resolve external models
[options.utcOffset]	<code>number</code>	UTC Offset for this execution

<a name="module_concerto-core.ModelManager"></a>

### concerto-core.ModelManager
Manages the Concerto model files.

The structure of Resources (Assets, Transactions, Participants) is modelled
in a set of Concerto files. The contents of these files are managed
by the ModelManager. Each Concerto file has a single namespace and contains
a set of asset, transaction and participant type definitions.

Concerto applications load their Concerto files and then call the [addModelFile](ModelManager#addModelFile)
method to register the Concerto file(s) with the ModelManager.

Use the Concerto class to validate instances.

Kind: static class of [<code>concerto-core</code>](#module_concerto-core)

* [.ModelManager](#module_concerto-core.ModelManager)
    * [new ModelManager([options])](#new_module_concerto-core.ModelManager_new)
    * _instance_
        * [.accept(visitor, parameters)](#module_concerto-core.ModelManager+accept) ⇒ <code>Object</code>
        * [.validateModelFile(modelFile, [fileName])](#module_concerto-core.ModelManager+validateModelFile)
        * [.addModelFile(modelFile, fileName, [disableValidation])](#module_concerto-core.ModelManager+addModelFile) ⇒ <code>Object</code>
        * [.updateModelFile(modelFile, [fileName], [disableValidation])](#module_concerto-core.ModelManager+updateModelFile) ⇒ <code>Object</code>
        * [.deleteModelFile(namespace)](#module_concerto-core.ModelManager+deleteModelFile)
        * [.addModelFiles(modelFiles, [fileNames], [disableValidation])](#module_concerto-core.ModelManager+addModelFiles) ⇒ <code>Array.&lt;Object&gt;</code>
        * [.validateModelFiles()](#module_concerto-core.ModelManager+validateModelFiles)

```
        * [.updateExternalModels([options], [modelFileDownloader])]
(#module_concerto-core.ModelManager+updateExternalModels) ⇒ <code>Promise</code>
        * [.writeModelsToFileSystem(path, [options])](#module_concerto-
core.ModelManager+writeModelsToFileSystem)
        * [.getModels([options])](#module_concerto-core.ModelManager+getModels) ⇒
<code>Array.&lt;{name:string, content:string}&gt;</code>
        * [.clearModelFiles()](#module_concerto-core.ModelManager+clearModelFiles)
        * [.getModelFile(namespace)](#module_concerto-
core.ModelManager+getModelFile) ⇒ <code>ModelFile</code>
        * [.getNamespaces()](#module_concerto-core.ModelManager+getNamespaces) ⇒
<code>Array.&lt;string&gt;</code>
        * [.getAssetDeclarations()](#module_concerto-
core.ModelManager+getAssetDeclarations) ⇒
<code>Array.&lt;AssetDeclaration&gt;</code>
        * [.getTransactionDeclarations()](#module_concerto-
core.ModelManager+getTransactionDeclarations) ⇒
<code>Array.&lt;TransactionDeclaration&gt;</code>
        * [.getEventDeclarations()](#module_concerto-
core.ModelManager+getEventDeclarations) ⇒
<code>Array.&lt;EventDeclaration&gt;</code>
        * [.getParticipantDeclarations()](#module_concerto-
core.ModelManager+getParticipantDeclarations) ⇒
<code>Array.&lt;ParticipantDeclaration&gt;</code>
        * [.getEnumDeclarations()](#module_concerto-
core.ModelManager+getEnumDeclarations) ⇒ <code>Array.&lt;EnumDeclaration&gt;</code>
        * [.getConceptDeclarations()](#module_concerto-
core.ModelManager+getConceptDeclarations) ⇒
<code>Array.&lt;ConceptDeclaration&gt;</code>
        * [.getFactory()](#module_concerto-core.ModelManager+getFactory) ⇒
<code>Factory</code>
        * [.getSerializer()](#module_concerto-core.ModelManager+getSerializer) ⇒
<code>Serializer</code>
        * [.getDecoratorFactories()](#module_concerto-
core.ModelManager+getDecoratorFactories) ⇒
<code>Array.&lt;DecoratorFactory&gt;</code>
        * [.addDecoratorFactory(factory)](#module_concerto-
core.ModelManager+addDecoratorFactory)
        * [.derivesFrom(fqt1, fqt2)](#module_concerto-
core.ModelManager+derivesFrom) ⇒ <code>boolean</code>
    * _static_
        * [.Symbol.hasInstance(object)](#module_concerto-
core.ModelManager.Symbol.hasInstance) ⇒ <code>boolean</code>

<a name="new_module_concerto-core.ModelManager_new"></a>

#### new ModelManager([options])
Create the ModelManager.


| Param | Type | Description |
| --- | --- | --- |
| [options] | <code>object</code> | Serializer options |

<a name="module_concerto-core.ModelManager+accept"></a>

#### modelManager.accept(visitor, parameters) ⇒ <code>Object</code>
Visitor design pattern

**Kind**: instance method of [<code>ModelManager</code>](#module_concerto-
```

core.ModelManager)
Returns: <code>Object</code> - the result of visiting or null

Param	Type	Description
visitor	<code>Object</code>	the visitor
parameters	<code>Object</code>	the parameter

<a name="module_concerto-core.ModelManager+validateModelFile"></a>

#### modelManager.validateModelFile(modelFile, [fileName])
Validates a Concerto file (as a string) to the ModelManager.
Concerto files have a single namespace.

Note that if there are dependencies between multiple files the files
must be added in dependency order, or the addModelFiles method can be
used to add a set of files irrespective of dependencies.

Kind: instance method of [<code>ModelManager</code>](#module_concerto-core.ModelManager)
Throws:

- <code>IllegalModelException</code>

Param	Type	Description
modelFile	<code>string</code>	The Concerto file as a string
[fileName]	<code>string</code>	a file name to associate with the model file

<a name="module_concerto-core.ModelManager+addModelFile"></a>

#### modelManager.addModelFile(modelFile, fileName, [disableValidation]) ⇒ <code>Object</code>
Adds a Concerto file (as a string) to the ModelManager.
Concerto files have a single namespace. If a Concerto file with the
same namespace has already been added to the ModelManager then it
will be replaced.
Note that if there are dependencies between multiple files the files
must be added in dependency order, or the addModelFiles method can be
used to add a set of files irrespective of dependencies.

Kind: instance method of [<code>ModelManager</code>](#module_concerto-core.ModelManager)
Returns: <code>Object</code> - The newly added model file (internal).
Throws:

- <code>IllegalModelException</code>

Param	Type	Description
modelFile	<code>string</code>	The Concerto file as a string
fileName	<code>string</code>	an optional file name to associate with the model file
[disableValidation]	<code>boolean</code>	If true then the model files are not validated

<a name="module_concerto-core.ModelManager+updateModelFile"></a>

```
#### modelManager.updateModelFile(modelFile, [fileName], [disableValidation]) ⇒
<code>Object</code>
Updates a Concerto file (as a string) on the ModelManager.
Concerto files have a single namespace. If a Concerto file with the
same namespace has already been added to the ModelManager then it
will be replaced.

**Kind**: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)
**Returns**: <code>Object</code> - The newly added model file (internal).
**Throws**:

- <code>IllegalModelException</code>


| Param | Type | Description |
| --- | --- | --- |
| modelFile | <code>string</code> | The Concerto file as a string |
| [fileName] | <code>string</code> | a file name to associate with the model file |
| [disableValidation] | <code>boolean</code> | If true then the model files are not
validated |

<a name="module_concerto-core.ModelManager+deleteModelFile"></a>

#### modelManager.deleteModelFile(namespace)
Remove the Concerto file for a given namespace

**Kind**: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)

| Param | Type | Description |
| --- | --- | --- |
| namespace | <code>string</code> | The namespace of the model file to delete. |

<a name="module_concerto-core.ModelManager+addModelFiles"></a>

#### modelManager.addModelFiles(modelFiles, [fileNames], [disableValidation]) ⇒
<code>Array.&lt;Object&gt;</code>
Add a set of Concerto files to the model manager.

**Kind**: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)
**Returns**: <code>Array.&lt;Object&gt;</code> - The newly added model files
(internal).

| Param | Type | Description |
| --- | --- | --- |
| modelFiles | <code>Array.&lt;object&gt;</code> | An array of Concerto files as
strings or ModelFile objects. |
| [fileNames] | <code>Array.&lt;string&gt;</code> | A array of file names to
associate with the model files |
| [disableValidation] | <code>boolean</code> | If true then the model files are not
validated |

<a name="module_concerto-core.ModelManager+validateModelFiles"></a>

#### modelManager.validateModelFiles()
Validates all models files in this model manager
```

```
**Kind**: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)
<a name="module_concerto-core.ModelManager+updateExternalModels"></a>

#### modelManager.updateExternalModels([options], [modelFileDownloader]) ⇒
<code>Promise</code>
Downloads all ModelFiles that are external dependencies and adds or
updates them in this ModelManager.

**Kind**: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)
**Returns**: <code>Promise</code> - a promise when the download and update
operation is completed.
**Throws**:

- <code>IllegalModelException</code> if the models fail validation


| Param | Type | Description |
| --- | --- | --- |
| [options] | <code>Object</code> | Options object passed to ModelFileLoaders |
| [modelFileDownloader] | <code>ModelFileDownloader</code> | an optional
ModelFileDownloader |

<a name="module_concerto-core.ModelManager+writeModelsToFileSystem"></a>

#### modelManager.writeModelsToFileSystem(path, [options])
Write all models in this model manager to the specified path in the file system

**Kind**: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)

| Param | Type | Description |
| --- | --- | --- |
| path | <code>string</code> | to a local directory |
| [options] | <code>Object</code> | Options object |
| options.includeExternalModels | <code>boolean</code> | If true, external models
are written to the file system. Defaults to true |

<a name="module_concerto-core.ModelManager+getModels"></a>

#### modelManager.getModels([options]) ⇒ <code>Array.&lt;{name:string,
content:string}&gt;</code>
Gets all the Concerto models

**Kind**: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)
**Returns**: <code>Array.&lt;{name:string, content:string}&gt;</code> - the name
and content of each CTO file

| Param | Type | Description |
| --- | --- | --- |
| [options] | <code>Object</code> | Options object |
| options.includeExternalModels | <code>boolean</code> | If true, external models
are written to the file system. Defaults to true |

<a name="module_concerto-core.ModelManager+clearModelFiles"></a>
```

#### modelManager.clearModelFiles()
Remove all registered Concerto files

Kind: instance method of [<code>ModelManager</code>](#module_concerto-core.ModelManager)
<a name="module_concerto-core.ModelManager+getModelFile"></a>

#### modelManager.getModelFile(namespace) ⇒ <code>ModelFile</code>
Get the ModelFile associated with a namespace

Kind: instance method of [<code>ModelManager</code>](#module_concerto-core.ModelManager)
Returns: <code>ModelFile</code> - registered ModelFile for the namespace or null

Param	Type	Description
namespace	<code>string</code>	the namespace containing the ModelFile

<a name="module_concerto-core.ModelManager+getNamespaces"></a>

#### modelManager.getNamespaces() ⇒ <code>Array.&lt;string&gt;</code>
Get the namespaces registered with the ModelManager.

Kind: instance method of [<code>ModelManager</code>](#module_concerto-core.ModelManager)
Returns: <code>Array.&lt;string&gt;</code> - namespaces - the namespaces that have been registered.
<a name="module_concerto-core.ModelManager+getAssetDeclarations"></a>

#### modelManager.getAssetDeclarations() ⇒ <code>Array.&lt;AssetDeclaration&gt;</code>
Get the AssetDeclarations defined in this model manager

Kind: instance method of [<code>ModelManager</code>](#module_concerto-core.ModelManager)
Returns: <code>Array.&lt;AssetDeclaration&gt;</code> - the AssetDeclarations defined in the model manager
<a name="module_concerto-core.ModelManager+getTransactionDeclarations"></a>

#### modelManager.getTransactionDeclarations() ⇒ <code>Array.&lt;TransactionDeclaration&gt;</code>
Get the TransactionDeclarations defined in this model manager

Kind: instance method of [<code>ModelManager</code>](#module_concerto-core.ModelManager)
Returns: <code>Array.&lt;TransactionDeclaration&gt;</code> - the TransactionDeclarations defined in the model manager
<a name="module_concerto-core.ModelManager+getEventDeclarations"></a>

#### modelManager.getEventDeclarations() ⇒ <code>Array.&lt;EventDeclaration&gt;</code>
Get the EventDeclarations defined in this model manager

Kind: instance method of [<code>ModelManager</code>](#module_concerto-core.ModelManager)
Returns: <code>Array.&lt;EventDeclaration&gt;</code> - the EventDeclaration defined in the model manager
<a name="module_concerto-core.ModelManager+getParticipantDeclarations"></a>

```
#### modelManager.getParticipantDeclarations() ⇒
<code>Array.&lt;ParticipantDeclaration&gt;</code>
Get the ParticipantDeclarations defined in this model manager

**Kind**: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)
**Returns**: <code>Array.&lt;ParticipantDeclaration&gt;</code> - the
ParticipantDeclaration defined in the model manager
<a name="module_concerto-core.ModelManager+getEnumDeclarations"></a>

#### modelManager.getEnumDeclarations() ⇒
<code>Array.&lt;EnumDeclaration&gt;</code>
Get the EnumDeclarations defined in this model manager

**Kind**: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)
**Returns**: <code>Array.&lt;EnumDeclaration&gt;</code> - the EnumDeclaration
defined in the model manager
<a name="module_concerto-core.ModelManager+getConceptDeclarations"></a>

#### modelManager.getConceptDeclarations() ⇒
<code>Array.&lt;ConceptDeclaration&gt;</code>
Get the Concepts defined in this model manager

**Kind**: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)
**Returns**: <code>Array.&lt;ConceptDeclaration&gt;</code> - the ConceptDeclaration
defined in the model manager
<a name="module_concerto-core.ModelManager+getFactory"></a>

#### modelManager.getFactory() ⇒ <code>Factory</code>
Get a factory for creating new instances of types defined in this model manager.

**Kind**: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)
**Returns**: <code>Factory</code> - A factory for creating new instances of types
defined in this model manager.
<a name="module_concerto-core.ModelManager+getSerializer"></a>

#### modelManager.getSerializer() ⇒ <code>Serializer</code>
Get a serializer for serializing instances of types defined in this model manager.

**Kind**: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)
**Returns**: <code>Serializer</code> - A serializer for serializing instances of
types defined in this model manager.
<a name="module_concerto-core.ModelManager+getDecoratorFactories"></a>

#### modelManager.getDecoratorFactories() ⇒
<code>Array.&lt;DecoratorFactory&gt;</code>
Get the decorator factories for this model manager.

**Kind**: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)
**Returns**: <code>Array.&lt;DecoratorFactory&gt;</code> - The decorator factories
for this model manager.
<a name="module_concerto-core.ModelManager+addDecoratorFactory"></a>
```

```
#### modelManager.addDecoratorFactory(factory)
Add a decorator factory to this model manager.

**Kind**: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)

| Param | Type | Description |
| --- | --- | --- |
| factory | <code>DecoratorFactory</code> | The decorator factory to add to this
model manager. |

<a name="module_concerto-core.ModelManager+derivesFrom"></a>

#### modelManager.derivesFrom(fqt1, fqt2) ⇒ <code>boolean</code>
Checks if this fully qualified type name is derived from another.

**Kind**: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)
**Returns**: <code>boolean</code> - True if this instance is an instance of the
specified fully
qualified type name, false otherwise.

| Param | Type | Description |
| --- | --- | --- |
| fqt1 | <code>string</code> | The fully qualified type name to check. |
| fqt2 | <code>string</code> | The fully qualified type name it is may be derived
from. |

<a name="module_concerto-core.ModelManager.Symbol.hasInstance"></a>

#### ModelManager.Symbol.hasInstance(object) ⇒ <code>boolean</code>
Alternative instanceof that is reliable across different module instances

**Kind**: static method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)
**Returns**: <code>boolean</code> - - True, if the object is an instance of a
ModelManager
**See**: https://github.com/hyperledger/composer-concerto/issues/47

| Param | Type | Description |
| --- | --- | --- |
| object | <code>object</code> | The object to test against |

<a name="module_concerto-core.SecurityException"></a>

### concerto-core.SecurityException ⇐ <code>BaseException</code>
Class representing a security exception

**Kind**: static class of [<code>concerto-core</code>](#module_concerto-core)
**Extends**: <code>BaseException</code>
**See**: See [BaseException](BaseException)
<a name="new_module_concerto-core.SecurityException_new"></a>

#### new SecurityException(message)
Create the SecurityException.


| Param | Type | Description |
| --- | --- | --- |
```

| message | <code>string</code> | The exception message. |

<a name="module_concerto-core.Serializer"></a>

### concerto-core.Serializer
Serialize Resources instances to/from various formats for long-term storage
(e.g. on the blockchain).

Kind: static class of [<code>concerto-core</code>](#module_concerto-core)

* [.Serializer](#module_concerto-core.Serializer)
    * [new Serializer(factory, modelManager, [options])](#new_module_concerto-core.Serializer_new)
    * _instance_
        * [.setDefaultOptions(newDefaultOptions)](#module_concerto-core.Serializer+setDefaultOptions)
        * [.toJSON(resource, [options])](#module_concerto-core.Serializer+toJSON) ⇒ <code>Object</code>
        * [.fromJSON(jsonObject, options)](#module_concerto-core.Serializer+fromJSON) ⇒ <code>Resource</code>
    * _static_
        * [.Symbol.hasInstance(object)](#module_concerto-core.Serializer.Symbol.hasInstance) ⇒ <code>boolean</code>

<a name="new_module_concerto-core.Serializer_new"></a>

#### new Serializer(factory, modelManager, [options])
Create a Serializer.

Param	Type	Description
factory	<code>Factory</code>	The Factory to use to create instances
modelManager	<code>ModelManager</code>	The ModelManager to use for validation etc.
[options]	<code>object</code>	Serializer options

<a name="module_concerto-core.Serializer+setDefaultOptions"></a>

#### serializer.setDefaultOptions(newDefaultOptions)
Set the default options for the serializer.

Kind: instance method of [<code>Serializer</code>](#module_concerto-core.Serializer)

Param	Type	Description
newDefaultOptions	<code>Object</code>	The new default options for the serializer.

<a name="module_concerto-core.Serializer+toJSON"></a>

#### serializer.toJSON(resource, [options]) ⇒ <code>Object</code>
<p>
Convert a Resource to a JavaScript object suitable for long-term
peristent storage.
</p>

Kind: instance method of [<code>Serializer</code>](#module_concerto-

core.Serializer)
Returns: <code>Object</code> - - The Javascript Object that represents the resource
Throws:

- <code>Error</code> - throws an exception if resource is not an instance of Resource or fails validation.

Param	Type	Description
resource	<code>Resource</code>	The instance to convert to JSON
[options]	<code>Object</code>	the optional serialization options.
[options.validate]	<code>boolean</code>	validate the structure of the Resource with its model prior to serialization (default to true)
[options.convertResourcesToRelationships]	<code>boolean</code>	Convert resources that are specified for relationship fields into relationships, false by default.
[options.permitResourcesForRelationships]	<code>boolean</code>	Permit resources in the place of relationships (serializing them as resources), false by default.
[options.deduplicateResources]	<code>boolean</code>	Generate $id for resources and if a resources appears multiple times in the object graph only the first instance is serialized in full, subsequent instances are replaced with a reference to the $id
[options.convertResourcesToId]	<code>boolean</code>	Convert resources that are specified for relationship fields into their id, false by default.
[options.utcOffset]	<code>number</code>	UTC Offset for DateTime values.

<a name="module_concerto-core.Serializer+fromJSON"></a>

#### serializer.fromJSON(jsonObject, options) ⇒ <code>Resource</code>
Create a Resource from a JavaScript Object representation.
The JavaScript Object should have been created by calling the
[toJSON](Serializer#toJSON) API.

The Resource is populated based on the JavaScript object.

Kind: instance method of [<code>Serializer</code>](#module_concerto-core.Serializer)
Returns: <code>Resource</code> - The new populated resource

Param	Type	Description
jsonObject	<code>Object</code>	The JavaScript Object for a Resource
options	<code>Object</code>	the optional serialization options
options.acceptResourcesForRelationships	<code>boolean</code>	handle JSON objects in the place of strings for relationships, defaults to false.
options.validate	<code>boolean</code>	validate the structure of the Resource with its model prior to serialization (default to true)
[options.utcOffset]	<code>number</code>	UTC Offset for DateTime values.

<a name="module_concerto-core.Serializer.Symbol.hasInstance"></a>

#### Serializer.Symbol.hasInstance(object) ⇒ <code>boolean</code>
Alternative instanceof that is reliable across different module instances

Kind: static method of [<code>Serializer</code>](#module_concerto-core.Serializer)

```
**Returns**: <code>boolean</code> - - True, if the object is an instance of a
Serializer
**See**: https://github.com/hyperledger/composer-concerto/issues/47

| Param | Type | Description |
| --- | --- | --- |
| object | <code>object</code> | The object to test against |

<a name="module_concerto-core.TypeNotFoundException"></a>

### concerto-core.TypeNotFoundException ⇐ <code>BaseException</code>
Error thrown when a Concerto type does not exist.

**Kind**: static class of [<code>concerto-core</code>](#module_concerto-core)
**Extends**: <code>BaseException</code>
**See**: see [BaseException](BaseException)

* [.TypeNotFoundException](#module_concerto-core.TypeNotFoundException) ⇐
<code>BaseException</code>
    * [new TypeNotFoundException(typeName, [message], component)]
(#new_module_concerto-core.TypeNotFoundException_new)
    * [.getTypeName()](#module_concerto-core.TypeNotFoundException+getTypeName) ⇒
<code>string</code>

<a name="new_module_concerto-core.TypeNotFoundException_new"></a>

#### new TypeNotFoundException(typeName, [message], component)
Constructor. If the optional 'message' argument is not supplied, it will be set to
a default value that
includes the type name.


| Param | Type | Description |
| --- | --- | --- |
| typeName | <code>string</code> | fully qualified type name. |
| [message] | <code>string</code> | error message. |
| component | <code>string</code> | the optional component which throws this error
|

<a name="module_concerto-core.TypeNotFoundException+getTypeName"></a>

#### typeNotFoundException.getTypeName() ⇒ <code>string</code>
Get the name of the type that was not found.

**Kind**: instance method of [<code>TypeNotFoundException</code>](#module_concerto-
core.TypeNotFoundException)
**Returns**: <code>string</code> - fully qualified type name.
<a name="module_concerto-core.AssetDeclaration"></a>

### concerto-core.AssetDeclaration ⇐ <code>ClassDeclaration</code>
AssetDeclaration defines the schema (aka model or class) for
an Asset. It extends ClassDeclaration which manages a set of
fields, a super-type and the specification of an
identifying field.

**Kind**: static class of [<code>concerto-core</code>](#module_concerto-core)
**Extends**: <code>ClassDeclaration</code>
**See**: See [ClassDeclaration](ClassDeclaration)
```

* [.AssetDeclaration](#module_concerto-core.AssetDeclaration) ⇐
<code>ClassDeclaration</code>
    * [new AssetDeclaration(modelFile, ast)](#new_module_concerto-
core.AssetDeclaration_new)
    * [.Symbol.hasInstance(object)](#module_concerto-
core.AssetDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>

<a name="new_module_concerto-core.AssetDeclaration_new"></a>

#### new AssetDeclaration(modelFile, ast)
Create an AssetDeclaration.

Throws:

- <code>IllegalModelException</code>

Param	Type	Description
modelFile	<code>ModelFile</code>	the ModelFile for this class
ast	<code>Object</code>	The AST created by the parser

<a name="module_concerto-core.AssetDeclaration.Symbol.hasInstance"></a>

#### AssetDeclaration.Symbol.hasInstance(object) ⇒ <code>boolean</code>
Alternative instanceof that is reliable across different module instances

Kind: static method of [<code>AssetDeclaration</code>](#module_concerto-
core.AssetDeclaration)
Returns: <code>boolean</code> - - True, if the object is an instance of a
AssetDeclaration
See: https://github.com/hyperledger/composer-concerto/issues/47

Param	Type	Description
object	<code>object</code>	The object to test against

<a name="module_concerto-core.ClassDeclaration"></a>

### *concerto-core.ClassDeclaration*
ClassDeclaration defines the structure (model/schema) of composite data.
It is composed of a set of Properties, may have an identifying field, and may
have a super-type.
A ClassDeclaration is conceptually owned by a ModelFile which
defines all the classes that are part of a namespace.

Kind: static abstract class of [<code>concerto-core</code>](#module_concerto-
core)

* *[.ClassDeclaration](#module_concerto-core.ClassDeclaration)*
    * *[new ClassDeclaration(modelFile, ast)](#new_module_concerto-
core.ClassDeclaration_new)*
    * _instance_
        * *[._resolveSuperType()](#module_concerto-
core.ClassDeclaration+_resolveSuperType) ⇒ <code>ClassDeclaration</code>*
        * *[.isAbstract()](#module_concerto-core.ClassDeclaration+isAbstract) ⇒
<code>boolean</code>*
        * *[.isEnum()](#module_concerto-core.ClassDeclaration+isEnum) ⇒
<code>boolean</code>*

```
        * *[.isConcept()](#module_concerto-core.ClassDeclaration+isConcept) ⇒
<code>boolean</code>*
        * *[.isEvent()](#module_concerto-core.ClassDeclaration+isEvent) ⇒
<code>boolean</code>*
        * *[.getName()](#module_concerto-core.ClassDeclaration+getName) ⇒
<code>string</code>*
        * *[.getNamespace()](#module_concerto-core.ClassDeclaration+getNamespace) ⇒
<code>string</code>*
        * *[.getFullyQualifiedName()](#module_concerto-
core.ClassDeclaration+getFullyQualifiedName) ⇒ <code>string</code>*
        * *[.isIdentified()](#module_concerto-core.ClassDeclaration+isIdentified) ⇒
<code>Boolean</code>*
        * *[.isSystemIdentified()](#module_concerto-
core.ClassDeclaration+isSystemIdentified) ⇒ <code>Boolean</code>*
        * *[.isExplicitlyIdentified()](#module_concerto-
core.ClassDeclaration+isExplicitlyIdentified) ⇒ <code>Boolean</code>*
        * *[.getIdentifierFieldName()](#module_concerto-
core.ClassDeclaration+getIdentifierFieldName) ⇒ <code>string</code>*
        * *[.getOwnProperty(name)](#module_concerto-
core.ClassDeclaration+getOwnProperty) ⇒ <code>Property</code>*
        * *[.getOwnProperties()](#module_concerto-
core.ClassDeclaration+getOwnProperties) ⇒ <code>Array.&lt;Property&gt;</code>*
        * *[.getSuperType()](#module_concerto-core.ClassDeclaration+getSuperType) ⇒
<code>string</code>*
        * *[.getSuperTypeDeclaration()](#module_concerto-
core.ClassDeclaration+getSuperTypeDeclaration) ⇒ <code>ClassDeclaration</code>*
        * *[.getAssignableClassDeclarations()](#module_concerto-
core.ClassDeclaration+getAssignableClassDeclarations) ⇒
<code>Array.&lt;ClassDeclaration&gt;</code>*
        * *[.getAllSuperTypeDeclarations()](#module_concerto-
core.ClassDeclaration+getAllSuperTypeDeclarations) ⇒
<code>Array.&lt;ClassDeclaration&gt;</code>*
        * *[.getProperty(name)](#module_concerto-core.ClassDeclaration+getProperty)
⇒ <code>Property</code>*
        * *[.getProperties()](#module_concerto-core.ClassDeclaration+getProperties)
⇒ <code>Array.&lt;Property&gt;</code>*
        * *[.getNestedProperty(propertyPath)](#module_concerto-
core.ClassDeclaration+getNestedProperty) ⇒ <code>Property</code>*
        * *[.toString()](#module_concerto-core.ClassDeclaration+toString) ⇒
<code>String</code>*
    * _static_
        * *[.Symbol.hasInstance(object)](#module_concerto-
core.ClassDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>*

<a name="new_module_concerto-core.ClassDeclaration_new"></a>

#### *new ClassDeclaration(modelFile, ast)*
Create a ClassDeclaration from an Abstract Syntax Tree. The AST is the
result of parsing.

**Throws**:

- <code>IllegalModelException</code>


| Param | Type | Description |
| --- | --- | --- |
| modelFile | <code>ModelFile</code> | the ModelFile for this class |
| ast | <code>Object</code> | the AST created by the parser |
<a name="module_concerto-core.ClassDeclaration+_resolveSuperType"></a>

#### *classDeclaration.\_resolveSuperType() ⇒ <code>ClassDeclaration</code>*
Resolve the super type on this class and store it as an internal property.

**Kind**: instance method of [<code>ClassDeclaration</code>](#module_concerto-
core.ClassDeclaration)
**Returns**: <code>ClassDeclaration</code> - The super type, or null if non
specified.
<a name="module_concerto-core.ClassDeclaration+isAbstract"></a>

#### *classDeclaration.isAbstract() ⇒ <code>boolean</code>*
Returns true if this class is declared as abstract in the model file

**Kind**: instance method of [<code>ClassDeclaration</code>](#module_concerto-
core.ClassDeclaration)
**Returns**: <code>boolean</code> - true if the class is abstract
<a name="module_concerto-core.ClassDeclaration+isEnum"></a>

#### *classDeclaration.isEnum() ⇒ <code>boolean</code>*
Returns true if this class is an enumeration.

**Kind**: instance method of [<code>ClassDeclaration</code>](#module_concerto-
core.ClassDeclaration)
**Returns**: <code>boolean</code> - true if the class is an enumerated type
<a name="module_concerto-core.ClassDeclaration+isConcept"></a>

#### *classDeclaration.isConcept() ⇒ <code>boolean</code>*
Returns true if this class is the definition of a concept.

**Kind**: instance method of [<code>ClassDeclaration</code>](#module_concerto-
core.ClassDeclaration)
**Returns**: <code>boolean</code> - true if the class is a concept
<a name="module_concerto-core.ClassDeclaration+isEvent"></a>

#### *classDeclaration.isEvent() ⇒ <code>boolean</code>*
Returns true if this class is the definition of an event.

**Kind**: instance method of [<code>ClassDeclaration</code>](#module_concerto-
core.ClassDeclaration)
**Returns**: <code>boolean</code> - true if the class is an event
<a name="module_concerto-core.ClassDeclaration+getName"></a>

#### *classDeclaration.getName() ⇒ <code>string</code>*
Returns the short name of a class. This name does not include the
namespace from the owning ModelFile.

**Kind**: instance method of [<code>ClassDeclaration</code>](#module_concerto-
core.ClassDeclaration)
**Returns**: <code>string</code> - the short name of this class
<a name="module_concerto-core.ClassDeclaration+getNamespace"></a>

#### *classDeclaration.getNamespace() ⇒ <code>string</code>*
Return the namespace of this class.

**Kind**: instance method of [<code>ClassDeclaration</code>](#module_concerto-
core.ClassDeclaration)
**Returns**: <code>string</code> - namespace - a namespace.
```

```
<a name="module_concerto-core.ClassDeclaration+getFullyQualifiedName"></a>

#### *classDeclaration.getFullyQualifiedName() ⇒ <code>string</code>*
Returns the fully qualified name of this class.
The name will include the namespace if present.

**Kind**: instance method of [<code>ClassDeclaration</code>](#module_concerto-
core.ClassDeclaration)
**Returns**: <code>string</code> - the fully-qualified name of this class
<a name="module_concerto-core.ClassDeclaration+isIdentified"></a>

#### *classDeclaration.isIdentified() ⇒ <code>Boolean</code>*
Returns true if this class declaration declares an identifying field
(system or explicit)

**Kind**: instance method of [<code>ClassDeclaration</code>](#module_concerto-
core.ClassDeclaration)
**Returns**: <code>Boolean</code> - true if the class declaration includes an
identifier
<a name="module_concerto-core.ClassDeclaration+isSystemIdentified"></a>

#### *classDeclaration.isSystemIdentified() ⇒ <code>Boolean</code>*
Returns true if this class declaration declares a system identifier
$identifier

**Kind**: instance method of [<code>ClassDeclaration</code>](#module_concerto-
core.ClassDeclaration)
**Returns**: <code>Boolean</code> - true if the class declaration includes a system
identifier
<a name="module_concerto-core.ClassDeclaration+isExplicitlyIdentified"></a>

#### *classDeclaration.isExplicitlyIdentified() ⇒ <code>Boolean</code>*
Returns true if this class declaration declares an explicit identifier

**Kind**: instance method of [<code>ClassDeclaration</code>](#module_concerto-
core.ClassDeclaration)
**Returns**: <code>Boolean</code> - true if the class declaration includes an
explicit identifier
<a name="module_concerto-core.ClassDeclaration+getIdentifierFieldName"></a>

#### *classDeclaration.getIdentifierFieldName() ⇒ <code>string</code>*
Returns the name of the identifying field for this class. Note
that the identifying field may come from a super type.

**Kind**: instance method of [<code>ClassDeclaration</code>](#module_concerto-
core.ClassDeclaration)
**Returns**: <code>string</code> - the name of the id field for this class or null
if it does not exist
<a name="module_concerto-core.ClassDeclaration+getOwnProperty"></a>

#### *classDeclaration.getOwnProperty(name) ⇒ <code>Property</code>*
Returns the field with a given name or null if it does not exist.
The field must be directly owned by this class -- the super-type is
not introspected.

**Kind**: instance method of [<code>ClassDeclaration</code>](#module_concerto-
core.ClassDeclaration)
**Returns**: <code>Property</code> - the field definition or null if it does not
exist
```

Param	Type	Description
name	<code>string</code>	the name of the field

<a name="module_concerto-core.ClassDeclaration+getOwnProperties"></a>

#### *classDeclaration.getOwnProperties() ⇒ <code>Array.&lt;Property&gt;</code>*
Returns the fields directly defined by this class.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>Array.&lt;Property&gt;</code> - the array of fields
<a name="module_concerto-core.ClassDeclaration+getSuperType"></a>

#### *classDeclaration.getSuperType() ⇒ <code>string</code>*
Returns the FQN of the super type for this class or null if this
class does not have a super type.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>string</code> - the FQN name of the super type or null
<a name="module_concerto-core.ClassDeclaration+getSuperTypeDeclaration"></a>

#### *classDeclaration.getSuperTypeDeclaration() ⇒ <code>ClassDeclaration</code>*
Get the super type class declaration for this class.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>ClassDeclaration</code> - the super type declaration, or null if there is no super type.
<a name="module_concerto-core.ClassDeclaration+getAssignableClassDeclarations"></a>

#### *classDeclaration.getAssignableClassDeclarations() ⇒ <code>Array.&lt;ClassDeclaration&gt;</code>*
Get the class declarations for all subclasses of this class, including this class.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>Array.&lt;ClassDeclaration&gt;</code> - subclass declarations.
<a name="module_concerto-core.ClassDeclaration+getAllSuperTypeDeclarations"></a>

#### *classDeclaration.getAllSuperTypeDeclarations() ⇒ <code>Array.&lt;ClassDeclaration&gt;</code>*
Get all the super-type declarations for this type.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>Array.&lt;ClassDeclaration&gt;</code> - super-type declarations.
<a name="module_concerto-core.ClassDeclaration+getProperty"></a>

#### *classDeclaration.getProperty(name) ⇒ <code>Property</code>*
Returns the property with a given name or null if it does not exist.
Fields defined in super-types are also introspected.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>Property</code> - the field, or null if it does not exist

Param	Type	Description
name	<code>string</code>	the name of the field

<a name="module_concerto-core.ClassDeclaration+getProperties"></a>

#### *classDeclaration.getProperties() ⇒ <code>Array.&lt;Property&gt;</code>*
Returns the properties defined in this class and all super classes.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-
core.ClassDeclaration)
Returns: <code>Array.&lt;Property&gt;</code> - the array of fields
<a name="module_concerto-core.ClassDeclaration+getNestedProperty"></a>

#### *classDeclaration.getNestedProperty(propertyPath) ⇒ <code>Property</code>*
Get a nested property using a dotted property path

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-
core.ClassDeclaration)
Returns: <code>Property</code> - the property
Throws:

- <code>IllegalModelException</code> if the property path is invalid or the
property does not exist

Param	Type	Description
propertyPath	<code>string</code>	The property name or name with nested
structure e.g a.b.c

<a name="module_concerto-core.ClassDeclaration+toString"></a>

#### *classDeclaration.toString() ⇒ <code>String</code>*
Returns the string representation of this class

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-
core.ClassDeclaration)
Returns: <code>String</code> - the string representation of the class
<a name="module_concerto-core.ClassDeclaration.Symbol.hasInstance"></a>

#### *ClassDeclaration.Symbol.hasInstance(object) ⇒ <code>boolean</code>*
Alternative instanceof that is reliable across different module instances

Kind: static method of [<code>ClassDeclaration</code>](#module_concerto-
core.ClassDeclaration)
Returns: <code>boolean</code> - - True, if the object is an instance of a Class
Declaration
See: https://github.com/hyperledger/composer-concerto/issues/47

Param	Type	Description
object	<code>object</code>	The object to test against

<a name="module_concerto-core.ConceptDeclaration"></a>

### concerto-core.ConceptDeclaration ⇐ <code>ClassDeclaration</code>
ConceptDeclaration defines the schema (aka model or class) for
an Concept. It extends ClassDeclaration which manages a set of

```
fields, a super-type and the specification of an
identifying field.

**Kind**: static class of [<code>concerto-core</code>](#module_concerto-core)
**Extends**: <code>ClassDeclaration</code>
**See**: [ClassDeclaration](ClassDeclaration)

* [.ConceptDeclaration](#module_concerto-core.ConceptDeclaration) ⇐
<code>ClassDeclaration</code>
    * [new ConceptDeclaration(modelFile, ast)](#new_module_concerto-
core.ConceptDeclaration_new)
    * _instance_
        * [.isConcept()](#module_concerto-core.ConceptDeclaration+isConcept) ⇒
<code>boolean</code>
    * _static_
        * [.Symbol.hasInstance(object)](#module_concerto-
core.ConceptDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>

<a name="new_module_concerto-core.ConceptDeclaration_new"></a>

#### new ConceptDeclaration(modelFile, ast)
Create a ConceptDeclaration.

**Throws**:

- <code>IllegalModelException</code>


| Param | Type | Description |
| --- | --- | --- |
| modelFile | <code>ModelFile</code> | the ModelFile for this class |
| ast | <code>Object</code> | The AST created by the parser |

<a name="module_concerto-core.ConceptDeclaration+isConcept"></a>

#### conceptDeclaration.isConcept() ⇒ <code>boolean</code>
Returns true if this class is the definition of a concept.

**Kind**: instance method of [<code>ConceptDeclaration</code>](#module_concerto-
core.ConceptDeclaration)
**Returns**: <code>boolean</code> - true if the class is a concept
<a name="module_concerto-core.ConceptDeclaration.Symbol.hasInstance"></a>

#### ConceptDeclaration.Symbol.hasInstance(object) ⇒ <code>boolean</code>
Alternative instanceof that is reliable across different module instances

**Kind**: static method of [<code>ConceptDeclaration</code>](#module_concerto-
core.ConceptDeclaration)
**Returns**: <code>boolean</code> - - True, if the object is an instance of a
ConceptDeclaration
**See**: https://github.com/hyperledger/composer-concerto/issues/47

| Param | Type | Description |
| --- | --- | --- |
| object | <code>object</code> | The object to test against |

<a name="module_concerto-core.Decorator"></a>

### concerto-core.Decorator
```

```
Decorator encapsulates a decorator (annotation) on a class or property.

**Kind**: static class of [<code>concerto-core</code>](#module_concerto-core)

* [.Decorator](#module_concerto-core.Decorator)
    * [new Decorator(parent, ast)](#new_module_concerto-core.Decorator_new)
    * [.getParent()](#module_concerto-core.Decorator+getParent) ⇒
<code>ClassDeclaration</code> \| <code>Property</code>
    * [.getName()](#module_concerto-core.Decorator+getName) ⇒ <code>string</code>
    * [.getArguments()](#module_concerto-core.Decorator+getArguments) ⇒
<code>Array.&lt;object&gt;</code>

<a name="new_module_concerto-core.Decorator_new"></a>

#### new Decorator(parent, ast)
Create a Decorator.

**Throws**:

- <code>IllegalModelException</code>


| Param | Type | Description |
| --- | --- | --- |
| parent | <code>ClassDeclaration</code> \| <code>Property</code> | the owner of
this property |
| ast | <code>Object</code> | The AST created by the parser |

<a name="module_concerto-core.Decorator+getParent"></a>

#### decorator.getParent() ⇒ <code>ClassDeclaration</code> \| <code>Property</code>
Returns the owner of this property

**Kind**: instance method of [<code>Decorator</code>](#module_concerto-
core.Decorator)
**Returns**: <code>ClassDeclaration</code> \| <code>Property</code> - the parent
class or property declaration
<a name="module_concerto-core.Decorator+getName"></a>

#### decorator.getName() ⇒ <code>string</code>
Returns the name of a decorator

**Kind**: instance method of [<code>Decorator</code>](#module_concerto-
core.Decorator)
**Returns**: <code>string</code> - the name of this decorator
<a name="module_concerto-core.Decorator+getArguments"></a>

#### decorator.getArguments() ⇒ <code>Array.&lt;object&gt;</code>
Returns the arguments for this decorator

**Kind**: instance method of [<code>Decorator</code>](#module_concerto-
core.Decorator)
**Returns**: <code>Array.&lt;object&gt;</code> - the arguments for this decorator
<a name="module_concerto-core.DecoratorFactory"></a>

### concerto-core.DecoratorFactory
An interface for a class that processes a decorator and returns a specific
implementation class for that decorator.
```

```
**Kind**: static class of [<code>concerto-core</code>](#module_concerto-core)
<a name="module_concerto-core.DecoratorFactory+newDecorator"></a>

#### *decoratorFactory.newDecorator(parent, ast) ⇒ <code>Decorator</code>*
Process the decorator, and return a specific implementation class for that
decorator, or return null if this decorator is not handled by this processor.

**Kind**: instance abstract method of [<code>DecoratorFactory</code>]
(#module_concerto-core.DecoratorFactory)
**Returns**: <code>Decorator</code> - The decorator.

| Param | Type | Description |
| --- | --- | --- |
| parent | <code>ClassDeclaration</code> \| <code>Property</code> | the owner of
this property |
| ast | <code>Object</code> | The AST created by the parser |

<a name="module_concerto-core.EnumDeclaration"></a>

### concerto-core.EnumDeclaration ⇐ <code>ClassDeclaration</code>
EnumDeclaration defines an enumeration of static values.

**Kind**: static class of [<code>concerto-core</code>](#module_concerto-core)
**Extends**: <code>ClassDeclaration</code>
**See**: See [ClassDeclaration](ClassDeclaration)

* [.EnumDeclaration](#module_concerto-core.EnumDeclaration) ⇐
<code>ClassDeclaration</code>
    * [new EnumDeclaration(modelFile, ast)](#new_module_concerto-
core.EnumDeclaration_new)
    * _instance_
        * [.isEnum()](#module_concerto-core.EnumDeclaration+isEnum) ⇒
<code>boolean</code>
        * [.toString()](#module_concerto-core.EnumDeclaration+toString) ⇒
<code>String</code>
    * _static_
        * [.Symbol.hasInstance(object)](#module_concerto-
core.EnumDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>

<a name="new_module_concerto-core.EnumDeclaration_new"></a>

#### new EnumDeclaration(modelFile, ast)
Create an EnumDeclaration.

**Throws**:

- <code>IllegalModelException</code>


| Param | Type | Description |
| --- | --- | --- |
| modelFile | <code>ModelFile</code> | the ModelFile for this class |
| ast | <code>Object</code> | The AST created by the parser |

<a name="module_concerto-core.EnumDeclaration+isEnum"></a>

#### enumDeclaration.isEnum() ⇒ <code>boolean</code>
Returns true if this class is an enumeration.
```

Kind: instance method of [<code>EnumDeclaration</code>](#module_concerto-core.EnumDeclaration)
Returns: <code>boolean</code> - true if the class is an enumerated type
<a name="module_concerto-core.EnumDeclaration+toString"></a>

#### enumDeclaration.toString() ⇒ <code>String</code>
Returns the string representation of this class

Kind: instance method of [<code>EnumDeclaration</code>](#module_concerto-core.EnumDeclaration)
Returns: <code>String</code> - the string representation of the class
<a name="module_concerto-core.EnumDeclaration.Symbol.hasInstance"></a>

#### EnumDeclaration.Symbol.hasInstance(object) ⇒ <code>boolean</code>
Alternative instanceof that is reliable across different module instances

Kind: static method of [<code>EnumDeclaration</code>](#module_concerto-core.EnumDeclaration)
Returns: <code>boolean</code> - - True, if the object is an instance of a Class Declaration
See: https://github.com/hyperledger/composer-concerto/issues/47

Param	Type	Description
object	<code>object</code>	The object to test against

<a name="module_concerto-core.EnumValueDeclaration"></a>

### concerto-core.EnumValueDeclaration ⇐ <code>Property</code>
Class representing a value from a set of enumerated values

Kind: static class of [<code>concerto-core</code>](#module_concerto-core)
Extends: <code>Property</code>
See: See Property

* [.EnumValueDeclaration](#module_concerto-core.EnumValueDeclaration) ⇐ <code>Property</code>
    * [new EnumValueDeclaration(parent, ast)](#new_module_concerto-core.EnumValueDeclaration_new)
    * [.Symbol.hasInstance(object)](#module_concerto-core.EnumValueDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>

<a name="new_module_concerto-core.EnumValueDeclaration_new"></a>

#### new EnumValueDeclaration(parent, ast)
Create a EnumValueDeclaration.

Throws:

- <code>IllegalModelException</code>

Param	Type	Description
parent	<code>ClassDeclaration</code>	The owner of this property
ast	<code>Object</code>	The AST created by the parser

<a name="module_concerto-core.EnumValueDeclaration.Symbol.hasInstance"></a>

```
#### EnumValueDeclaration.Symbol.hasInstance(object) ⇒ <code>boolean</code>
Alternative instanceof that is reliable across different module instances

**Kind**: static method of [<code>EnumValueDeclaration</code>](#module_concerto-
core.EnumValueDeclaration)
**Returns**: <code>boolean</code> - - True, if the object is an instance of a
EnumValueDeclaration
**See**: https://github.com/hyperledger/composer-concerto/issues/47

| Param | Type | Description |
| --- | --- | --- |
| object | <code>object</code> | The object to test against |

<a name="module_concerto-core.EventDeclaration"></a>

### concerto-core.EventDeclaration ⇐ <code>ClassDeclaration</code>
Class representing the definition of an Event.

**Kind**: static class of [<code>concerto-core</code>](#module_concerto-core)
**Extends**: <code>ClassDeclaration</code>
**See**: See  [ClassDeclaration](ClassDeclaration)

* [.EventDeclaration](#module_concerto-core.EventDeclaration) ⇐
<code>ClassDeclaration</code>
    * [new EventDeclaration(modelFile, ast)](#new_module_concerto-
core.EventDeclaration_new)
    * _instance_
        * [.isEvent()](#module_concerto-core.EventDeclaration+isEvent) ⇒
<code>boolean</code>
    * _static_
        * [.Symbol.hasInstance(object)](#module_concerto-
core.EventDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>

<a name="new_module_concerto-core.EventDeclaration_new"></a>

#### new EventDeclaration(modelFile, ast)
Create an EventDeclaration.

**Throws**:

- <code>IllegalModelException</code>


| Param | Type | Description |
| --- | --- | --- |
| modelFile | <code>ModelFile</code> | the ModelFile for this class |
| ast | <code>Object</code> | The AST created by the parser |

<a name="module_concerto-core.EventDeclaration+isEvent"></a>

#### eventDeclaration.isEvent() ⇒ <code>boolean</code>
Returns true if this class is the definition of an event

**Kind**: instance method of [<code>EventDeclaration</code>](#module_concerto-
core.EventDeclaration)
**Returns**: <code>boolean</code> - true if the class is an event
<a name="module_concerto-core.EventDeclaration.Symbol.hasInstance"></a>

#### EventDeclaration.Symbol.hasInstance(object) ⇒ <code>boolean</code>
```

Alternative instanceof that is reliable across different module instances

Kind: static method of [<code>EventDeclaration</code>](#module_concerto-core.EventDeclaration)
Returns: <code>boolean</code> - - True, if the object is an instance of a EventDeclaration
See: https://github.com/hyperledger/composer-concerto/issues/47

Param	Type	Description
object	<code>object</code>	The object to test against

<a name="module_concerto-core.IdentifiedDeclaration"></a>

### *concerto-core.IdentifiedDeclaration ⇐ <code>ClassDeclaration</code>*
IdentifiedDeclaration

Kind: static abstract class of [<code>concerto-core</code>](#module_concerto-core)
Extends: <code>ClassDeclaration</code>
See: See ClassDeclaration

* *[.IdentifiedDeclaration](#module_concerto-core.IdentifiedDeclaration) ⇐ <code>ClassDeclaration</code>*
    * *[new IdentifiedDeclaration(modelFile, ast)](#new_module_concerto-core.IdentifiedDeclaration_new)*
    * *[.Symbol.hasInstance(object)](#module_concerto-core.IdentifiedDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>*

<a name="new_module_concerto-core.IdentifiedDeclaration_new"></a>

#### *new IdentifiedDeclaration(modelFile, ast)*
Create an AssetDeclaration.

Throws:

- <code>IllegalModelException</code>

Param	Type	Description
modelFile	<code>ModelFile</code>	the ModelFile for this class
ast	<code>Object</code>	The AST created by the parser

<a name="module_concerto-core.IdentifiedDeclaration.Symbol.hasInstance"></a>

#### *IdentifiedDeclaration.Symbol.hasInstance(object) ⇒ <code>boolean</code>*
Alternative instanceof that is reliable across different module instances

Kind: static method of [<code>IdentifiedDeclaration</code>](#module_concerto-core.IdentifiedDeclaration)
Returns: <code>boolean</code> - - True, if the object is an instance of a AssetDeclaration
See: https://github.com/hyperledger/composer-concerto/issues/47

Param	Type	Description
object	<code>object</code>	The object to test against

<a name="module_concerto-core.IllegalModelException"></a>

### concerto-core.IllegalModelException ⇐ <code>BaseFileException</code>
Exception throws when a composer file is semantically invalid

Kind: static class of [<code>concerto-core</code>](#module_concerto-core)
Extends: <code>BaseFileException</code>
See: See BaseFileException
<a name="new_module_concerto-core.IllegalModelException_new"></a>

#### new IllegalModelException(message, [modelFile], [fileLocation], [component])
Create an IllegalModelException.

Param	Type	Description
message	<code>string</code>	the message for the exception
[modelFile]	<code>ModelFile</code>	the modelfile associated with the exception
[fileLocation]	<code>Object</code>	location details of the error within the model file.
fileLocation.start.line	<code>number</code>	start line of the error location.
fileLocation.start.column	<code>number</code>	start column of the error location.
fileLocation.end.line	<code>number</code>	end line of the error location.
fileLocation.end.column	<code>number</code>	end column of the error location.
[component]	<code>string</code>	the component which throws this error

<a name="module_concerto-core.Introspector"></a>

### concerto-core.Introspector
Provides access to the structure of transactions, assets and participants.

Kind: static class of [<code>concerto-core</code>](#module_concerto-core)

* [.Introspector](#module_concerto-core.Introspector)
    * [new Introspector(modelManager)](#new_module_concerto-core.Introspector_new)
    * [.getClassDeclarations()](#module_concerto-core.Introspector+getClassDeclarations) ⇒ <code>Array.&lt;ClassDeclaration&gt;</code>
    * [.getClassDeclaration(fullyQualifiedTypeName)](#module_concerto-core.Introspector+getClassDeclaration) ⇒ <code>ClassDeclaration</code>

<a name="new_module_concerto-core.Introspector_new"></a>

#### new Introspector(modelManager)
Create the Introspector.

Param	Type	Description
modelManager	<code>ModelManager</code>	the ModelManager that backs this Introspector

<a name="module_concerto-core.Introspector+getClassDeclarations"></a>

#### introspector.getClassDeclarations() ⇒

```
<code>Array.&lt;ClassDeclaration&gt;</code>
Returns all the class declarations for the business network.

**Kind**: instance method of [<code>Introspector</code>](#module_concerto-
core.Introspector)
**Returns**: <code>Array.&lt;ClassDeclaration&gt;</code> - the array of class
declarations
<a name="module_concerto-core.Introspector+getClassDeclaration"></a>

#### introspector.getClassDeclaration(fullyQualifiedTypeName) ⇒
<code>ClassDeclaration</code>
Returns the class declaration with the given fully qualified name.
Throws an error if the class declaration does not exist.

**Kind**: instance method of [<code>Introspector</code>](#module_concerto-
core.Introspector)
**Returns**: <code>ClassDeclaration</code> - the class declaration
**Throws**:

- <code>Error</code> if the class declaration does not exist


| Param | Type | Description |
| --- | --- | --- |
| fullyQualifiedTypeName | <code>String</code> | the fully qualified name of the
type |

<a name="module_concerto-core.ModelFile"></a>

### concerto-core.ModelFile
Class representing a Model File. A Model File contains a single namespace
and a set of model elements: assets, transactions etc.

**Kind**: static class of [<code>concerto-core</code>](#module_concerto-core)

* [.ModelFile](#module_concerto-core.ModelFile)
    * [new ModelFile(modelManager, definitions, [fileName])](#new_module_concerto-
core.ModelFile_new)
    * _instance_
        * [.isSystemModelFile()](#module_concerto-core.ModelFile+isSystemModelFile)
⇒ <code>Boolean</code>
        * [.isExternal()](#module_concerto-core.ModelFile+isExternal) ⇒
<code>boolean</code>
        * [.getModelManager()](#module_concerto-core.ModelFile+getModelManager) ⇒
<code>ModelManager</code>
        * [.getImports()](#module_concerto-core.ModelFile+getImports) ⇒
<code>Array.&lt;string&gt;</code>
        * [.isDefined(type)](#module_concerto-core.ModelFile+isDefined) ⇒
<code>boolean</code>
        * [.getLocalType(type)](#module_concerto-core.ModelFile+getLocalType) ⇒
<code>ClassDeclaration</code>
        * [.getAssetDeclaration(name)](#module_concerto-
core.ModelFile+getAssetDeclaration) ⇒ <code>AssetDeclaration</code>
        * [.getTransactionDeclaration(name)](#module_concerto-
core.ModelFile+getTransactionDeclaration) ⇒ <code>TransactionDeclaration</code>
        * [.getEventDeclaration(name)](#module_concerto-
core.ModelFile+getEventDeclaration) ⇒ <code>EventDeclaration</code>
        * [.getParticipantDeclaration(name)](#module_concerto-
core.ModelFile+getParticipantDeclaration) ⇒ <code>ParticipantDeclaration</code>
```

```
        * [.getNamespace()](#module_concerto-core.ModelFile+getNamespace) ⇒
<code>string</code>
        * [.getName()](#module_concerto-core.ModelFile+getName) ⇒
<code>string</code>
        * [.getAssetDeclarations()](#module_concerto-
core.ModelFile+getAssetDeclarations) ⇒ <code>Array.&lt;AssetDeclaration&gt;</code>
        * [.getTransactionDeclarations()](#module_concerto-
core.ModelFile+getTransactionDeclarations) ⇒
<code>Array.&lt;TransactionDeclaration&gt;</code>
        * [.getEventDeclarations()](#module_concerto-
core.ModelFile+getEventDeclarations) ⇒ <code>Array.&lt;EventDeclaration&gt;</code>
        * [.getParticipantDeclarations()](#module_concerto-
core.ModelFile+getParticipantDeclarations) ⇒
<code>Array.&lt;ParticipantDeclaration&gt;</code>
        * [.getConceptDeclarations()](#module_concerto-
core.ModelFile+getConceptDeclarations) ⇒
<code>Array.&lt;ConceptDeclaration&gt;</code>
        * [.getEnumDeclarations()](#module_concerto-
core.ModelFile+getEnumDeclarations) ⇒ <code>Array.&lt;EnumDeclaration&gt;</code>
        * [.getDeclarations(type)](#module_concerto-core.ModelFile+getDeclarations)
⇒ <code>Array.&lt;ClassDeclaration&gt;</code>
        * [.getAllDeclarations()](#module_concerto-
core.ModelFile+getAllDeclarations) ⇒ <code>Array.&lt;ClassDeclaration&gt;</code>
        * [.getDefinitions()](#module_concerto-core.ModelFile+getDefinitions) ⇒
<code>string</code>
        * [.getConcertoVersion()](#module_concerto-
core.ModelFile+getConcertoVersion) ⇒ <code>string</code>
    * _static_
        * [.Symbol.hasInstance(object)](#module_concerto-
core.ModelFile.Symbol.hasInstance) ⇒ <code>boolean</code>

<a name="new_module_concerto-core.ModelFile_new"></a>

#### new ModelFile(modelManager, definitions, [fileName])
Create a ModelFile. This should only be called by framework code.
Use the ModelManager to manage ModelFiles.

**Throws**:

- <code>IllegalModelException</code>


| Param | Type | Description |
| --- | --- | --- |
| modelManager | <code>ModelManager</code> | the ModelManager that manages this
ModelFile |
| definitions | <code>string</code> | The DSL model as a string. |
| [fileName] | <code>string</code> | The optional filename for this modelfile |

<a name="module_concerto-core.ModelFile+isSystemModelFile"></a>

#### modelFile.isSystemModelFile() ⇒ <code>Boolean</code>
Returns true if the ModelFile is a system namespace

**Kind**: instance method of [<code>ModelFile</code>](#module_concerto-
core.ModelFile)
**Returns**: <code>Boolean</code> - true if this is a system model file
<a name="module_concerto-core.ModelFile+isExternal"></a>
```

#### modelFile.isExternal() ⇒ <code>boolean</code>
Returns true if this ModelFile was downloaded from an external URI.

Kind: instance method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)
Returns: <code>boolean</code> - true iff this ModelFile was downloaded from an external URI
<a name="module_concerto-core.ModelFile+getModelManager"></a>

#### modelFile.getModelManager() ⇒ <code>ModelManager</code>
Returns the ModelManager associated with this ModelFile

Kind: instance method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)
Returns: <code>ModelManager</code> - The ModelManager for this ModelFile
<a name="module_concerto-core.ModelFile+getImports"></a>

#### modelFile.getImports() ⇒ <code>Array.&lt;string&gt;</code>
Returns the types that have been imported into this ModelFile.

Kind: instance method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)
Returns: <code>Array.&lt;string&gt;</code> - The array of imports for this ModelFile
<a name="module_concerto-core.ModelFile+isDefined"></a>

#### modelFile.isDefined(type) ⇒ <code>boolean</code>
Returns true if the type is defined in the model file

Kind: instance method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)
Returns: <code>boolean</code> - true if the type (asset or transaction) is defined

Param	Type	Description
type	<code>string</code>	the name of the type

<a name="module_concerto-core.ModelFile+getLocalType"></a>

#### modelFile.getLocalType(type) ⇒ <code>ClassDeclaration</code>
Returns the type with the specified name or null

Kind: instance method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)
Returns: <code>ClassDeclaration</code> - the ClassDeclaration, or null if the type does not exist

Param	Type	Description
type	<code>string</code>	the short OR FQN name of the type

<a name="module_concerto-core.ModelFile+getAssetDeclaration"></a>

#### modelFile.getAssetDeclaration(name) ⇒ <code>AssetDeclaration</code>
Get the AssetDeclarations defined in this ModelFile or null

Kind: instance method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)

```
**Returns**: <code>AssetDeclaration</code> - the AssetDeclaration with the given
short name

| Param | Type | Description |
| --- | --- | --- |
| name | <code>string</code> | the name of the type |

<a name="module_concerto-core.ModelFile+getTransactionDeclaration"></a>

#### modelFile.getTransactionDeclaration(name) ⇒
<code>TransactionDeclaration</code>
Get the TransactionDeclaration defined in this ModelFile or null

**Kind**: instance method of [<code>ModelFile</code>](#module_concerto-
core.ModelFile)
**Returns**: <code>TransactionDeclaration</code> - the TransactionDeclaration with
the given short name

| Param | Type | Description |
| --- | --- | --- |
| name | <code>string</code> | the name of the type |

<a name="module_concerto-core.ModelFile+getEventDeclaration"></a>

#### modelFile.getEventDeclaration(name) ⇒ <code>EventDeclaration</code>
Get the EventDeclaration defined in this ModelFile or null

**Kind**: instance method of [<code>ModelFile</code>](#module_concerto-
core.ModelFile)
**Returns**: <code>EventDeclaration</code> - the EventDeclaration with the given
short name

| Param | Type | Description |
| --- | --- | --- |
| name | <code>string</code> | the name of the type |

<a name="module_concerto-core.ModelFile+getParticipantDeclaration"></a>

#### modelFile.getParticipantDeclaration(name) ⇒
<code>ParticipantDeclaration</code>
Get the ParticipantDeclaration defined in this ModelFile or null

**Kind**: instance method of [<code>ModelFile</code>](#module_concerto-
core.ModelFile)
**Returns**: <code>ParticipantDeclaration</code> - the ParticipantDeclaration with
the given short name

| Param | Type | Description |
| --- | --- | --- |
| name | <code>string</code> | the name of the type |

<a name="module_concerto-core.ModelFile+getNamespace"></a>

#### modelFile.getNamespace() ⇒ <code>string</code>
Get the Namespace for this model file.

**Kind**: instance method of [<code>ModelFile</code>](#module_concerto-
core.ModelFile)
**Returns**: <code>string</code> - The Namespace for this model file
```

<a name="module_concerto-core.ModelFile+getName"></a>

#### modelFile.getName() ⇒ <code>string</code>
Get the filename for this model file. Note that this may be null.

Kind: instance method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)
Returns: <code>string</code> - The filename for this model file
<a name="module_concerto-core.ModelFile+getAssetDeclarations"></a>

#### modelFile.getAssetDeclarations() ⇒ <code>Array.&lt;AssetDeclaration&gt;</code>
Get the AssetDeclarations defined in this ModelFile

Kind: instance method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)
Returns: <code>Array.&lt;AssetDeclaration&gt;</code> - the AssetDeclarations defined in the model file
<a name="module_concerto-core.ModelFile+getTransactionDeclarations"></a>

#### modelFile.getTransactionDeclarations() ⇒ <code>Array.&lt;TransactionDeclaration&gt;</code>
Get the TransactionDeclarations defined in this ModelFile

Kind: instance method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)
Returns: <code>Array.&lt;TransactionDeclaration&gt;</code> - the TransactionDeclarations defined in the model file
<a name="module_concerto-core.ModelFile+getEventDeclarations"></a>

#### modelFile.getEventDeclarations() ⇒ <code>Array.&lt;EventDeclaration&gt;</code>
Get the EventDeclarations defined in this ModelFile

Kind: instance method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)
Returns: <code>Array.&lt;EventDeclaration&gt;</code> - the EventDeclarations defined in the model file
<a name="module_concerto-core.ModelFile+getParticipantDeclarations"></a>

#### modelFile.getParticipantDeclarations() ⇒ <code>Array.&lt;ParticipantDeclaration&gt;</code>
Get the ParticipantDeclarations defined in this ModelFile

Kind: instance method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)
Returns: <code>Array.&lt;ParticipantDeclaration&gt;</code> - the ParticipantDeclaration defined in the model file
<a name="module_concerto-core.ModelFile+getConceptDeclarations"></a>

#### modelFile.getConceptDeclarations() ⇒ <code>Array.&lt;ConceptDeclaration&gt;</code>
Get the ConceptDeclarations defined in this ModelFile

Kind: instance method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)
Returns: <code>Array.&lt;ConceptDeclaration&gt;</code> - the ParticipantDeclaration defined in the model file
<a name="module_concerto-core.ModelFile+getEnumDeclarations"></a>

#### modelFile.getEnumDeclarations() ⇒ <code>Array.&lt;EnumDeclaration&gt;</code>

Get the EnumDeclarations defined in this ModelFile

Kind: instance method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)
Returns: <code>Array.&lt;EnumDeclaration&gt;</code> - the EnumDeclaration defined in the model file
<a name="module_concerto-core.ModelFile+getDeclarations"></a>

#### modelFile.getDeclarations(type) ⇒ <code>Array.&lt;ClassDeclaration&gt;</code>
Get the instances of a given type in this ModelFile

Kind: instance method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)
Returns: <code>Array.&lt;ClassDeclaration&gt;</code> - the ClassDeclaration defined in the model file

Param	Type	Description
type	<code>function</code>	the type of the declaration

<a name="module_concerto-core.ModelFile+getAllDeclarations"></a>

#### modelFile.getAllDeclarations() ⇒ <code>Array.&lt;ClassDeclaration&gt;</code>
Get all declarations in this ModelFile

Kind: instance method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)
Returns: <code>Array.&lt;ClassDeclaration&gt;</code> - the ClassDeclarations defined in the model file
<a name="module_concerto-core.ModelFile+getDefinitions"></a>

#### modelFile.getDefinitions() ⇒ <code>string</code>
Get the definitions for this model.

Kind: instance method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)
Returns: <code>string</code> - The definitions for this model.
<a name="module_concerto-core.ModelFile+getConcertoVersion"></a>

#### modelFile.getConcertoVersion() ⇒ <code>string</code>
Get the expected concerto version

Kind: instance method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)
Returns: <code>string</code> - The semver range for compatible concerto versions
<a name="module_concerto-core.ModelFile.Symbol.hasInstance"></a>

#### ModelFile.Symbol.hasInstance(object) ⇒ <code>boolean</code>
Alternative to instanceof that is reliable across different module instances

Kind: static method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)
Returns: <code>boolean</code> - - True, if the object is an instance of a ModelFile
See: https://github.com/hyperledger/composer-concerto/issues/47

Param	Type	Description

```
| object | <code>object</code> | The object to test against |

<a name="module_concerto-core.ParticipantDeclaration"></a>

### concerto-core.ParticipantDeclaration ⇐ <code>ClassDeclaration</code>
Class representing the definition of a Participant.

**Kind**: static class of [<code>concerto-core</code>](#module_concerto-core)
**Extends**: <code>ClassDeclaration</code>
**See**: See  [ClassDeclaration](ClassDeclaration)

* [.ParticipantDeclaration](#module_concerto-core.ParticipantDeclaration) ⇐
<code>ClassDeclaration</code>
    * [new ParticipantDeclaration(modelFile, ast)](#new_module_concerto-
core.ParticipantDeclaration_new)
    * [.Symbol.hasInstance(object)](#module_concerto-
core.ParticipantDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>

<a name="new_module_concerto-core.ParticipantDeclaration_new"></a>

#### new ParticipantDeclaration(modelFile, ast)
Create an ParticipantDeclaration.

**Throws**:

- <code>IllegalModelException</code>


| Param | Type | Description |
| --- | --- | --- |
| modelFile | <code>ModelFile</code> | the ModelFile for this class |
| ast | <code>Object</code> | The AST created by the parser |

<a name="module_concerto-core.ParticipantDeclaration.Symbol.hasInstance"></a>

#### ParticipantDeclaration.Symbol.hasInstance(object) ⇒ <code>boolean</code>
Alternative instanceof that is reliable across different module instances

**Kind**: static method of [<code>ParticipantDeclaration</code>](#module_concerto-
core.ParticipantDeclaration)
**Returns**: <code>boolean</code> - - True, if the object is an instance of a
ParticipantDeclaration
**See**: https://github.com/hyperledger/composer-concerto/issues/47

| Param | Type | Description |
| --- | --- | --- |
| object | <code>object</code> | The object to test against |

<a name="module_concerto-core.Property"></a>

### concerto-core.Property
Property representing an attribute of a class declaration,
either a Field or a Relationship.

**Kind**: static class of [<code>concerto-core</code>](#module_concerto-core)

* [.Property](#module_concerto-core.Property)
    * [new Property(parent, ast)](#new_module_concerto-core.Property_new)
    * _instance_
```

```
        * [.getParent()](#module_concerto-core.Property+getParent) ⇒
<code>ClassDeclaration</code>
        * [.getName()](#module_concerto-core.Property+getName) ⇒
<code>string</code>
        * [.getType()](#module_concerto-core.Property+getType) ⇒
<code>string</code>
        * [.isOptional()](#module_concerto-core.Property+isOptional) ⇒
<code>boolean</code>
        * [.getFullyQualifiedTypeName()](#module_concerto-
core.Property+getFullyQualifiedTypeName) ⇒ <code>string</code>
        * [.getFullyQualifiedName()](#module_concerto-
core.Property+getFullyQualifiedName) ⇒ <code>string</code>
        * [.getNamespace()](#module_concerto-core.Property+getNamespace) ⇒
<code>string</code>
        * [.isArray()](#module_concerto-core.Property+isArray) ⇒
<code>boolean</code>
        * [.isTypeEnum()](#module_concerto-core.Property+isTypeEnum) ⇒
<code>boolean</code>
        * [.isPrimitive()](#module_concerto-core.Property+isPrimitive) ⇒
<code>boolean</code>
    * _static_
        * [.Symbol.hasInstance(object)](#module_concerto-
core.Property.Symbol.hasInstance) ⇒ <code>boolean</code>

<a name="new_module_concerto-core.Property_new"></a>

#### new Property(parent, ast)
Create a Property.

**Throws**:

- <code>IllegalModelException</code>


| Param | Type | Description |
| --- | --- | --- |
| parent | <code>ClassDeclaration</code> | the owner of this property |
| ast | <code>Object</code> | The AST created by the parser |

<a name="module_concerto-core.Property+getParent"></a>

#### property.getParent() ⇒ <code>ClassDeclaration</code>
Returns the owner of this property

**Kind**: instance method of [<code>Property</code>](#module_concerto-
core.Property)
**Returns**: <code>ClassDeclaration</code> - the parent class declaration
<a name="module_concerto-core.Property+getName"></a>

#### property.getName() ⇒ <code>string</code>
Returns the name of a property

**Kind**: instance method of [<code>Property</code>](#module_concerto-
core.Property)
**Returns**: <code>string</code> - the name of this field
<a name="module_concerto-core.Property+getType"></a>

#### property.getType() ⇒ <code>string</code>
Returns the type of a property
```

```
**Kind**: instance method of [<code>Property</code>](#module_concerto-
core.Property)
**Returns**: <code>string</code> - the type of this field
<a name="module_concerto-core.Property+isOptional"></a>

#### property.isOptional() ⇒ <code>boolean</code>
Returns true if the field is optional

**Kind**: instance method of [<code>Property</code>](#module_concerto-
core.Property)
**Returns**: <code>boolean</code> - true if the field is optional
<a name="module_concerto-core.Property+getFullyQualifiedTypeName"></a>

#### property.getFullyQualifiedTypeName() ⇒ <code>string</code>
Returns the fully qualified type name of a property

**Kind**: instance method of [<code>Property</code>](#module_concerto-
core.Property)
**Returns**: <code>string</code> - the fully qualified type of this property
<a name="module_concerto-core.Property+getFullyQualifiedName"></a>

#### property.getFullyQualifiedName() ⇒ <code>string</code>
Returns the fully name of a property (ns + class name + property name)

**Kind**: instance method of [<code>Property</code>](#module_concerto-
core.Property)
**Returns**: <code>string</code> - the fully qualified name of this property
<a name="module_concerto-core.Property+getNamespace"></a>

#### property.getNamespace() ⇒ <code>string</code>
Returns the namespace of the parent of this property

**Kind**: instance method of [<code>Property</code>](#module_concerto-
core.Property)
**Returns**: <code>string</code> - the namespace of the parent of this property
<a name="module_concerto-core.Property+isArray"></a>

#### property.isArray() ⇒ <code>boolean</code>
Returns true if the field is declared as an array type

**Kind**: instance method of [<code>Property</code>](#module_concerto-
core.Property)
**Returns**: <code>boolean</code> - true if the property is an array type
<a name="module_concerto-core.Property+isTypeEnum"></a>

#### property.isTypeEnum() ⇒ <code>boolean</code>
Returns true if the field is declared as an enumerated value

**Kind**: instance method of [<code>Property</code>](#module_concerto-
core.Property)
**Returns**: <code>boolean</code> - true if the property is an enumerated value
<a name="module_concerto-core.Property+isPrimitive"></a>

#### property.isPrimitive() ⇒ <code>boolean</code>
Returns true if this property is a primitive type.

**Kind**: instance method of [<code>Property</code>](#module_concerto-
core.Property)
```

```
**Returns**: <code>boolean</code> - true if the property is a primitive type.
<a name="module_concerto-core.Property.Symbol.hasInstance"></a>

#### Property.Symbol.hasInstance(object) ⇒ <code>boolean</code>
Alternative instanceof that is reliable across different module instances

**Kind**: static method of [<code>Property</code>](#module_concerto-core.Property)
**Returns**: <code>boolean</code> - - True, if the object is an instance of a
Property
**See**: https://github.com/hyperledger/composer-concerto/issues/47

| Param | Type | Description |
| --- | --- | --- |
| object | <code>object</code> | The object to test against |

<a name="module_concerto-core.RelationshipDeclaration"></a>

### concerto-core.RelationshipDeclaration ⇐ <code>Property</code>
Class representing a relationship between model elements

**Kind**: static class of [<code>concerto-core</code>](#module_concerto-core)
**Extends**: <code>Property</code>
**See**: See  [Property](Property)

* [.RelationshipDeclaration](#module_concerto-core.RelationshipDeclaration) ⇐
<code>Property</code>
    * [new RelationshipDeclaration(parent, ast)](#new_module_concerto-
core.RelationshipDeclaration_new)
    * _instance_
        * [.toString()](#module_concerto-core.RelationshipDeclaration+toString) ⇒
<code>String</code>
    * _static_
        * [.Symbol.hasInstance(object)](#module_concerto-
core.RelationshipDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>

<a name="new_module_concerto-core.RelationshipDeclaration_new"></a>

#### new RelationshipDeclaration(parent, ast)
Create a Relationship.

**Throws**:

- <code>IllegalModelException</code>


| Param | Type | Description |
| --- | --- | --- |
| parent | <code>ClassDeclaration</code> | The owner of this property |
| ast | <code>Object</code> | The AST created by the parser |

<a name="module_concerto-core.RelationshipDeclaration+toString"></a>

#### relationshipDeclaration.toString() ⇒ <code>String</code>
Returns a string representation of this property

**Kind**: instance method of [<code>RelationshipDeclaration</code>]
(#module_concerto-core.RelationshipDeclaration)
**Returns**: <code>String</code> - the string version of the property.
<a name="module_concerto-core.RelationshipDeclaration.Symbol.hasInstance"></a>
```

#### RelationshipDeclaration.Symbol.hasInstance(object) ⇒ <code>boolean</code>
Alternative instanceof that is reliable across different module instances

Kind: static method of [<code>RelationshipDeclaration</code>](#module_concerto-core.RelationshipDeclaration)
Returns: <code>boolean</code> - - True, if the object is an instance of a RelationshipDeclaration
See: https://github.com/hyperledger/composer-concerto/issues/47

Param	Type	Description
object	<code>object</code>	The object to test against

<a name="module_concerto-core.TransactionDeclaration"></a>

### concerto-core.TransactionDeclaration ⇐ <code>ClassDeclaration</code>
Class representing the definition of an Transaction.

Kind: static class of [<code>concerto-core</code>](#module_concerto-core)
Extends: <code>ClassDeclaration</code>
See: See ClassDeclaration

* [.TransactionDeclaration](#module_concerto-core.TransactionDeclaration) ⇐ <code>ClassDeclaration</code>
    * [new TransactionDeclaration(modelFile, ast)](#new_module_concerto-core.TransactionDeclaration_new)
    * [.Symbol.hasInstance(object)](#module_concerto-core.TransactionDeclaration.Symbol.hasInstance) ⇒ <code>boolean</code>

<a name="new_module_concerto-core.TransactionDeclaration_new"></a>

#### new TransactionDeclaration(modelFile, ast)
Create an TransactionDeclaration.

Throws:

- <code>IllegalModelException</code>

Param	Type	Description
modelFile	<code>ModelFile</code>	the ModelFile for this class
ast	<code>Object</code>	The AST created by the parser

<a name="module_concerto-core.TransactionDeclaration.Symbol.hasInstance"></a>

#### TransactionDeclaration.Symbol.hasInstance(object) ⇒ <code>boolean</code>
Alternative instanceof that is reliable across different module instances

Kind: static method of [<code>TransactionDeclaration</code>](#module_concerto-core.TransactionDeclaration)
Returns: <code>boolean</code> - - True, if the object is an instance of a TransactionDeclaration
See: https://github.com/hyperledger/composer-concerto/issues/47

Param	Type	Description
object	<code>object</code>	The object to test against

```
<a name="module_concerto-core.Resource"></a>

### concerto-core.Resource ⇐ <code>Identifiable</code>
Resource is an instance that has a type. The type of the resource
specifies a set of properites (which themselves have types).


Type information in Concerto is used to validate the structure of
Resource instances and for serialization.


Resources are used in Concerto to represent Assets, Participants, Transactions and
other domain classes that can be serialized for long-term persistent storage.

**Kind**: static class of [<code>concerto-core</code>](#module_concerto-core)
**Extends**: <code>Identifiable</code>
**Access**: public
**See**: See [Resource](Resource)

* [.Resource](#module_concerto-core.Resource) ⇐ <code>Identifiable</code>
    * [new Resource(modelManager, classDeclaration, ns, type, id, timestamp)]
(#new_module_concerto-core.Resource_new)
    * [.toString()](#module_concerto-core.Resource+toString) ⇒ <code>String</code>
    * [.isResource()](#module_concerto-core.Resource+isResource) ⇒
<code>boolean</code>
    * [.isConcept()](#module_concerto-core.Resource+isConcept) ⇒
<code>boolean</code>
    * [.isIdentifiable()](#module_concerto-core.Resource+isIdentifiable) ⇒
<code>boolean</code>
    * [.toJSON()](#module_concerto-core.Resource+toJSON) ⇒ <code>Object</code>

<a name="new_module_concerto-core.Resource_new"></a>

#### new Resource(modelManager, classDeclaration, ns, type, id, timestamp)
This constructor should not be called directly.
<p>
<strong>Note: Only to be called by framework code. Applications should
retrieve instances from [Factory](Factory)</strong>
</p>


| Param | Type | Description |
| --- | --- | --- |
| modelManager | <code>ModelManager</code> | The ModelManager for this instance |
| classDeclaration | <code>ClassDeclaration</code> | The class declaration for this
instance. |
| ns | <code>string</code> | The namespace this instance. |
| type | <code>string</code> | The type this instance. |
| id | <code>string</code> | The identifier of this instance. |
| timestamp | <code>string</code> | The timestamp of this instance |

<a name="module_concerto-core.Resource+toString"></a>

#### resource.toString() ⇒ <code>String</code>
Returns the string representation of this class

**Kind**: instance method of [<code>Resource</code>](#module_concerto-
core.Resource)
```

Returns: <code>String</code> - the string representation of the class
<a name="module_concerto-core.Resource+isResource"></a>

#### resource.isResource() ⇒ <code>boolean</code>
Determine if this identifiable is a resource.

Kind: instance method of [<code>Resource</code>](#module_concerto-core.Resource)
Returns: <code>boolean</code> - True if this identifiable is a resource, false if not.
<a name="module_concerto-core.Resource+isConcept"></a>

#### resource.isConcept() ⇒ <code>boolean</code>
Determine if this identifiable is a concept.

Kind: instance method of [<code>Resource</code>](#module_concerto-core.Resource)
Returns: <code>boolean</code> - True if this identifiable is a concept, false if not.
<a name="module_concerto-core.Resource+isIdentifiable"></a>

#### resource.isIdentifiable() ⇒ <code>boolean</code>
Determine if this object is identifiable.

Kind: instance method of [<code>Resource</code>](#module_concerto-core.Resource)
Returns: <code>boolean</code> - True if this object has an identifiying field false if not.
<a name="module_concerto-core.Resource+toJSON"></a>

#### resource.toJSON() ⇒ <code>Object</code>
Serialize this resource into a JavaScript object suitable for serialization to JSON,
using the default options for the serializer. If you need to set additional options
for the serializer, use the Serializer#toJSON method instead.

Kind: instance method of [<code>Resource</code>](#module_concerto-core.Resource)
Returns: <code>Object</code> - A JavaScript object suitable for serialization to JSON.
<a name="module_concerto-core.TypedStack"></a>

### concerto-core.TypedStack
Tracks a stack of typed instances. The type information is used to detect
overflow / underflow bugs by the caller. It also performs basic sanity
checking on push/pop to make detecting bugs easier.

Kind: static class of [<code>concerto-core</code>](#module_concerto-core)

* [.TypedStack](#module_concerto-core.TypedStack)
    * [new TypedStack(resource)](#new_module_concerto-core.TypedStack_new)
    * [.push(obj, expectedType)](#module_concerto-core.TypedStack+push)
    * [.pop(expectedType)](#module_concerto-core.TypedStack+pop) ⇒ <code>Object</code>
    * [.peek(expectedType)](#module_concerto-core.TypedStack+peek) ⇒ <code>Object</code>
    * [.clear()](#module_concerto-core.TypedStack+clear)

<a name="new_module_concerto-core.TypedStack_new"></a>

```
#### new TypedStack(resource)
Create the Stack with the resource at the head.


| Param | Type | Description |
| --- | --- | --- |
| resource | <code>Object</code> | the resource to be put at the head of the stack
|

<a name="module_concerto-core.TypedStack+push"></a>

#### typedStack.push(obj, expectedType)
Push a new object.

**Kind**: instance method of [<code>TypedStack</code>](#module_concerto-
core.TypedStack)

| Param | Type | Description |
| --- | --- | --- |
| obj | <code>Object</code> | the object being visited |
| expectedType | <code>Object</code> | the expected type of the object being pushed
|

<a name="module_concerto-core.TypedStack+pop"></a>

#### typedStack.pop(expectedType) ⇒ <code>Object</code>
Push a new object.

**Kind**: instance method of [<code>TypedStack</code>](#module_concerto-
core.TypedStack)
**Returns**: <code>Object</code> - the result of pop

| Param | Type | Description |
| --- | --- | --- |
| expectedType | <code>Object</code> | the type that should be the result of pop |

<a name="module_concerto-core.TypedStack+peek"></a>

#### typedStack.peek(expectedType) ⇒ <code>Object</code>
Peek the top of the stack

**Kind**: instance method of [<code>TypedStack</code>](#module_concerto-
core.TypedStack)
**Returns**: <code>Object</code> - the result of peek

| Param | Type | Description |
| --- | --- | --- |
| expectedType | <code>Object</code> | the type that should be the result of pop |

<a name="module_concerto-core.TypedStack+clear"></a>

#### typedStack.clear()
Clears the stack

**Kind**: instance method of [<code>TypedStack</code>](#module_concerto-
core.TypedStack)
<a name="levels"></a>
```

```
## levels : <code>Object</code>
Default levels for the npm configuration.

**Kind**: global constant
<a name="colorMap"></a>

## colorMap : <code>Object</code>
Default levels for the npm configuration.

**Kind**: global constant
<a name="setCurrentTime"></a>

## setCurrentTime([currentTime], [utcOffset]) ⇒ <code>object</code>
Ensures there is a proper current time

**Kind**: global function
**Returns**: <code>object</code> - if valid, the dayjs object for the current time

| Param | Type | Description |
| --- | --- | --- |
| [currentTime] | <code>string</code> | the definition of 'now' |
| [utcOffset] | <code>number</code> | UTC Offset for this execution |

<a name="randomNumberInRangeWithPrecision"></a>

## randomNumberInRangeWithPrecision(userMin, userMax, precision, systemMin,
systemMax) ⇒ <code>number</code>
Generate a random number within a given range with
a prescribed precision and inside a global range

**Kind**: global function
**Returns**: <code>number</code> - a number

| Param | Type | Description |
| --- | --- | --- |
| userMin | <code>\*</code> | Lower bound on the range, inclusive. Defaults to
systemMin |
| userMax | <code>\*</code> | Upper bound on the range, inclusive. Defaults to
systemMax |
| precision | <code>\*</code> | The precision of values returned, e.g. a value of
`1` returns only whole numbers |
| systemMin | <code>\*</code> | Global minimum on the range, takes precidence over
the userMin |
| systemMax | <code>\*</code> | Global maximum on the range, takes precidence over
the userMax |


--------------------------------------------------------------------------------
---
id: version-0.22-ref-concerto-cli
title: Command Line
original_id: ref-concerto-cli
---

Install the `@accordproject/concerto-cli` npm package to access the Concerto
command line interface (CLI). After installation you can use the `concerto` command
and its sub-commands as described below.

To install the Concerto CLI:
```

```
npm install -g @accordproject/concerto-cli
```

## Usage

```md
concerto <cmd> [args]

Commands:
  concerto validate  validate JSON against model files
  concerto compile   generate code for a target platform
  concerto get       save local copies of external model dependencies

Options:
      --version  Show version number                                [boolean]
  -v, --verbose                                              [default: false]
      --help     Show help                                          [boolean]
```

## concerto validate
`concerto validate` lets you check whether a JSON sample is a valid instance of the given model.

```md
concerto validate

validate JSON against model files

Options:
      --version     Show version number                             [boolean]
  -v, --verbose                                              [default: false]
      --help        Show help                                       [boolean]
      --input       JSON to validate                                 [string]
      --model       array of concerto (cto) model files               [array]
      --utcOffset   set UTC offset                                   [number]
      --offline     do not resolve external models    [boolean] [default: false]
      --functional  new validation API                [boolean] [default: false]
      --ergo        validation and emit for Ergo      [boolean] [default: false]
```

### Example
For example, using the `validate` command to check the sample `request.json` file from a [Late Delivery and Penalty](https://github.com/accordproject/cicero-template-library/tree/master/src/latedeliveryandpenalty) clause:

```
concerto validate --input request.json --model model/clause.cto
```

returns:

```json
{
  "$class":
"org.accordproject.latedeliveryandpenalty.LateDeliveryAndPenaltyRequest",
  "forceMajeure": false,
  "agreedDelivery": "2017-12-17T04:24:00.000-04:00",
  "goodsValue": 200,
```

~~~~
  "$timestamp": "2021-06-17T09:41:54.207-04:00"
}
```

## concerto compile
`Concerto compile` takes an array of local CTO files, downloads any external
dependencies (imports) and then converts all the model to the target format.

```md
concerto compile

generate code for a target platform

Options:
      --version  Show version number                                   [boolean]
  -v, --verbose                                               [default: false]
      --help     Show help                                             [boolean]
      --model    array of concerto (cto) model files          [array] [required]
      --offline  do not resolve external models       [boolean] [default: false]
      --target   target of the code generation  [string] [default: "JSONSchema"]
      --output   output directory path           [string] [default: "./output/"]
```

At the moment, the available target formats are as follows:
- Go Lang: `concerto compile --model modelfile.cto --target Go`
- Plant UML: `concerto compile --model modelfile.cto --target PlantUML`
- Typescript: `concerto compile --model modelfile.cto --target Typescript`
- Java: `concerto compile --model modelfile.cto --target Java`
- JSONSchema: `concerto compile --model modelfile.cto --target JSONSchema`
- XMLSchema: `concerto compile --model modelfile.cto --target XMLSchema`

### Example
For example, using the `compile` command to export the `clause.cto` file from a
[Late Delivery and Penalty](https://github.com/accordproject/cicero-template-
library/tree/master/src/latedeliveryandpenalty) clause into `Go Lang` format:

```md
cd ./model
concerto compile --model clause.cto --target Go
```

returns:
```md
info: Compiled to Go in './output/'.
```

## concerto get
`Concerto get` allows you to resolve and download external models from a set of
local CTO files.

```md
concerto get

save local copies of external model dependencies

Options:
      --version  Show version number                                   [boolean]
  -v, --verbose                                               [default: false]
      --help     Show help                                             [boolean]
~~~~

~~~~
      --model    array of concerto (cto) model files          [array] [required]
      --output   output directory path                  [string] [default: "./"]
```

### Example
For example, using the `get` command to get the external models in the `clause.cto`
file from a [Late Delivery and Penalty](https://github.com/accordproject/cicero-
template-library/tree/master/src/latedeliveryandpenalty) clause:

```md
concerto get --model clause.cto
```

returns:
```md
info: Loaded external models in './'.
```

--------------------------------------------------------------------------------
---
id: version-0.22-ref-ergo-api
title: Ergo API
original_id: ref-ergo-api
---

## Classes

<dl>
<dt><a href="#Commands">Commands</a></dt>
<dd><p>Utility class that implements the commands exposed by the Ergo CLI.</p>
</dd>
</dl>

## Functions

<dl>
<dt><a href="#getJson">getJson(input)</a> ⇒ <code>object</code></dt>
<dd><p>Load a file or JSON string</p>
</dd>
<dt><a href="#loadTemplate">loadTemplate(template, files)</a> ⇒
<code>Promise.&lt;LogicManager&gt;</code></dt>
<dd><p>Load a template from directory or files</p>
</dd>
<dt><a href="#fromDirectory">fromDirectory(path, [options])</a> ⇒
<code>Promise.&lt;LogicManager&gt;</code></dt>
<dd><p>Builds a LogicManager from a directory.</p>
</dd>
<dt><a href="#fromZip">fromZip(buffer, [options])</a> ⇒
<code>Promise.&lt;LogicManager&gt;</code></dt>
<dd><p>Builds a LogicManager from a Zip.</p>
</dd>
<dt><a href="#fromFiles">fromFiles(files, [options])</a> ⇒
<code>Promise.&lt;LogicManager&gt;</code></dt>
<dd><p>Builds a LogicManager from files.</p>
</dd>
<dt><a href="#validateContract">validateContract(modelManager, contract, utcOffset,
options)</a> ⇒ <code>object</code></dt>
<dd><p>Validate contract JSON</p>
</dd>
~~~~

```
<dt><a href="#validateInput">validateInput(modelManager, input, utcOffset)</a> ⇒
<code>object</code></dt>
<dd><p>Validate input JSON</p>
</dd>
<dt><a href="#validateInputRecord">validateInputRecord(modelManager, input,
utcOffset)</a> ⇒ <code>object</code></dt>
<dd><p>Validate input JSON record</p>
</dd>
<dt><a href="#validateOutput">validateOutput(modelManager, output, utcOffset)</a> ⇒
<code>object</code></dt>
<dd><p>Validate output JSON</p>
</dd>
<dt><a href="#validateOutputArray">validateOutputArray(modelManager, output,
utcOffset)</a> ⇒ <code>Array.&lt;object&gt;</code></dt>
<dd><p>Validate output JSON array</p>
</dd>
<dt><a href="#init">init(engine, logicManager, contractJson, currentTime,
utcOffset)</a> ⇒ <code>object</code></dt>
<dd><p>Invoke Ergo contract initialization</p>
</dd>
<dt><a href="#trigger">trigger(engine, logicManager, contractJson, stateJson,
currentTime, utcOffset, requestJson)</a> ⇒ <code>object</code></dt>
<dd><p>Trigger the Ergo contract with a request</p>
</dd>
<dt><a href="#resolveRootDir">resolveRootDir(parameters)</a> ⇒
<code>string</code></dt>
<dd><p>Resolve the root directory</p>
</dd>
<dt><a href="#compareComponent">compareComponent(expected, actual)</a></dt>
<dd><p>Compare actual and expected result components</p>
</dd>
<dt><a href="#compareSuccess">compareSuccess(expected, actual)</a></dt>
<dd><p>Compare actual result and expected result</p>
</dd>
</dl>

<a name="Commands"></a>

## Commands
Utility class that implements the commands exposed by the Ergo CLI.

**Kind**: global class

* [Commands](#Commands)
    * [.trigger(template, files, contractInput, stateInput, [currentTime],
[utcOffset], requestsInput, warnings)](#Commands.trigger) ⇒ <code>object</code>
    * [.invoke(template, files, clauseName, contractInput, stateInput,
[currentTime], [utcOffset], paramsInput, warnings)](#Commands.invoke) ⇒
<code>object</code>
    * [.initialize(template, files, contractInput, [currentTime], [utcOffset],
paramsInput, warnings)](#Commands.initialize) ⇒ <code>object</code>
    * [.parseCTOtoFileSync(ctoPath)](#Commands.parseCTOtoFileSync) ⇒
<code>string</code>
    * [.parseCTOtoFile(ctoPath)](#Commands.parseCTOtoFile) ⇒ <code>string</code>

<a name="Commands.trigger"></a>

### Commands.trigger(template, files, contractInput, stateInput, [currentTime],
[utcOffset], requestsInput, warnings) ⇒ <code>object</code>
```

Send a request an Ergo contract

Kind: static method of [<code>Commands</code>](#Commands)
Returns: <code>object</code> - Promise to the result of execution

Param	Type	Description
template	<code>string</code>	template directory
files	<code>Array.&lt;string&gt;</code>	input files
contractInput	<code>string</code>	the contract data
stateInput	<code>string</code>	the contract state
[currentTime]	<code>string</code>	the definition of 'now', defaults to current time
[utcOffset]	<code>number</code>	UTC Offset for this execution, defaults to local offset
requestsInput	<code>Array.&lt;string&gt;</code>	the requests
warnings	<code>boolean</code>	whether to print warnings

<a name="Commands.invoke"></a>

### Commands.invoke(template, files, clauseName, contractInput, stateInput, [currentTime], [utcOffset], paramsInput, warnings) ⇒ <code>object</code>
Invoke an Ergo contract's clause

Kind: static method of [<code>Commands</code>](#Commands)
Returns: <code>object</code> - Promise to the result of invocation

Param	Type	Description
template	<code>string</code>	template directory
files	<code>Array.&lt;string&gt;</code>	input files
clauseName	<code>string</code>	the name of the clause to invoke
contractInput	<code>string</code>	the contract data
stateInput	<code>string</code>	the contract state
[currentTime]	<code>string</code>	the definition of 'now', defaults to current time
[utcOffset]	<code>number</code>	UTC Offset for this execution, defaults to local offset
paramsInput	<code>object</code>	the parameters for the clause
warnings	<code>boolean</code>	whether to print warnings

<a name="Commands.initialize"></a>

### Commands.initialize(template, files, contractInput, [currentTime], [utcOffset], paramsInput, warnings) ⇒ <code>object</code>
Invoke init for an Ergo contract

Kind: static method of [<code>Commands</code>](#Commands)
Returns: <code>object</code> - Promise to the result of execution

Param	Type	Description
template	<code>string</code>	template directory
files	<code>Array.&lt;string&gt;</code>	input files
contractInput	<code>string</code>	the contract data
[currentTime]	<code>string</code>	the definition of 'now', defaults to current time
[utcOffset]	<code>number</code>	UTC Offset for this execution, defaults to local offset

```markdown
| paramsInput | <code>object</code> | the parameters for the clause |
| warnings | <code>boolean</code> | whether to print warnings |

<a name="Commands.parseCTOtoFileSync"></a>

### Commands.parseCTOtoFileSync(ctoPath) ⇒ <code>string</code>
Parse CTO to JSON File

**Kind**: static method of [<code>Commands</code>](#Commands)
**Returns**: <code>string</code> - The name of the generated CTOJ model file

| Param | Type | Description |
| --- | --- | --- |
| ctoPath | <code>string</code> | path to CTO model file |

<a name="Commands.parseCTOtoFile"></a>

### Commands.parseCTOtoFile(ctoPath) ⇒ <code>string</code>
Parse CTO to JSON File

**Kind**: static method of [<code>Commands</code>](#Commands)
**Returns**: <code>string</code> - The name of the generated CTOJ model file

| Param | Type | Description |
| --- | --- | --- |
| ctoPath | <code>string</code> | path to CTO model file |

<a name="getJson"></a>

## getJson(input) ⇒ <code>object</code>
Load a file or JSON string

**Kind**: global function
**Returns**: <code>object</code> - JSON object

| Param | Type | Description |
| --- | --- | --- |
| input | <code>object</code> | either a file name or a json string |

<a name="loadTemplate"></a>

## loadTemplate(template, files) ⇒ <code>Promise.&lt;LogicManager&gt;</code>
Load a template from directory or files

**Kind**: global function
**Returns**: <code>Promise.&lt;LogicManager&gt;</code> - a Promise to the
instantiated logicmanager

| Param | Type | Description |
| --- | --- | --- |
| template | <code>string</code> | template directory |
| files | <code>Array.&lt;string&gt;</code> | input files |

<a name="fromDirectory"></a>

## fromDirectory(path, [options]) ⇒ <code>Promise.&lt;LogicManager&gt;</code>
Builds a LogicManager from a directory.

**Kind**: global function
```

Returns: <code>Promise.&lt;LogicManager&gt;</code> - a Promise to the instantiated logicmanager

Param	Type	Description
path	<code>String</code>	to a local directory
[options]	<code>Object</code>	an optional set of options to configure the instance.

<a name="fromZip"></a>

## fromZip(buffer, [options]) ⇒ <code>Promise.&lt;LogicManager&gt;</code>
Builds a LogicManager from a Zip.

Kind: global function
Returns: <code>Promise.&lt;LogicManager&gt;</code> - a Promise to the instantiated logicmanager

Param	Type	Description
buffer	<code>Buffer</code>	the buffer to a Zip (zip) file
[options]	<code>Object</code>	an optional set of options to configure the instance.

<a name="fromFiles"></a>

## fromFiles(files, [options]) ⇒ <code>Promise.&lt;LogicManager&gt;</code>
Builds a LogicManager from files.

Kind: global function
Returns: <code>Promise.&lt;LogicManager&gt;</code> - a Promise to the instantiated logicmanager

Param	Type	Description
files	<code>Array.&lt;String&gt;</code>	file names
[options]	<code>Object</code>	an optional set of options to configure the instance.

<a name="validateContract"></a>

## validateContract(modelManager, contract, utcOffset, options) ⇒ <code>object</code>
Validate contract JSON

Kind: global function
Returns: <code>object</code> - the validated contract

Param	Type	Description
modelManager	<code>object</code>	the Concerto model manager
contract	<code>object</code>	the contract JSON
utcOffset	<code>number</code>	UTC Offset for DateTime values
options	<code>object</code>	parameters for contract variables validation

<a name="validateInput"></a>

## validateInput(modelManager, input, utcOffset) ⇒ <code>object</code>
Validate input JSON

```
**Kind**: global function
**Returns**: <code>object</code> - the validated input

| Param | Type | Description |
| --- | --- | --- |
| modelManager | <code>object</code> | the Concerto model manager |
| input | <code>object</code> | the input JSON |
| utcOffset | <code>number</code> | UTC Offset for DateTime values |

<a name="validateInputRecord"></a>

## validateInputRecord(modelManager, input, utcOffset) ⇒ <code>object</code>
Validate input JSON record

**Kind**: global function
**Returns**: <code>object</code> - the validated input

| Param | Type | Description |
| --- | --- | --- |
| modelManager | <code>object</code> | the Concerto model manager |
| input | <code>object</code> | the input JSON record |
| utcOffset | <code>number</code> | UTC Offset for DateTime values |

<a name="validateOutput"></a>

## validateOutput(modelManager, output, utcOffset) ⇒ <code>object</code>
Validate output JSON

**Kind**: global function
**Returns**: <code>object</code> - the validated output

| Param | Type | Description |
| --- | --- | --- |
| modelManager | <code>object</code> | the Concerto model manager |
| output | <code>object</code> | the output JSON |
| utcOffset | <code>number</code> | UTC Offset for DateTime values |

<a name="validateOutputArray"></a>

## validateOutputArray(modelManager, output, utcOffset) ⇒
<code>Array.&lt;object&gt;</code>
Validate output JSON array

**Kind**: global function
**Returns**: <code>Array.&lt;object&gt;</code> - the validated output array

| Param | Type | Description |
| --- | --- | --- |
| modelManager | <code>object</code> | the Concerto model manager |
| output | <code>\*</code> | the output JSON array |
| utcOffset | <code>number</code> | UTC Offset for DateTime values |

<a name="init"></a>

## init(engine, logicManager, contractJson, currentTime, utcOffset) ⇒
<code>object</code>
Invoke Ergo contract initialization
```

```
**Kind**: global function
**Returns**: <code>object</code> - Promise to the initial state of the contract

| Param | Type | Description |
| --- | --- | --- |
| engine | <code>object</code> | the execution engine |
| logicManager | <code>object</code> | the Template Logic |
| contractJson | <code>object</code> | contract data in JSON |
| currentTime | <code>string</code> | the definition of 'now' |
| utcOffset | <code>utcOffset</code> | UTC Offset for this execution |

<a name="trigger"></a>

## trigger(engine, logicManager, contractJson, stateJson, currentTime, utcOffset,
requestJson) ⇒ <code>object</code>
Trigger the Ergo contract with a request

**Kind**: global function
**Returns**: <code>object</code> - Promise to the response

| Param | Type | Description |
| --- | --- | --- |
| engine | <code>object</code> | the execution engine |
| logicManager | <code>object</code> | the Template Logic |
| contractJson | <code>object</code> | contract data in JSON |
| stateJson | <code>object</code> | state data in JSON |
| currentTime | <code>string</code> | the definition of 'now' |
| utcOffset | <code>utcOffset</code> | UTC Offset for this execution |
| requestJson | <code>object</code> | state data in JSON |

<a name="resolveRootDir"></a>

## resolveRootDir(parameters) ⇒ <code>string</code>
Resolve the root directory

**Kind**: global function
**Returns**: <code>string</code> - root directory used to resolve file names

| Param | Type | Description |
| --- | --- | --- |
| parameters | <code>string</code> | Cucumber's World parameters |

<a name="compareComponent"></a>

## compareComponent(expected, actual)
Compare actual and expected result components

**Kind**: global function

| Param | Type | Description |
| --- | --- | --- |
| expected | <code>string</code> | the expected component as specified in the test
workload |
| actual | <code>string</code> | the actual component as returned by the engine |

<a name="compareSuccess"></a>

## compareSuccess(expected, actual)
Compare actual result and expected result
```

Kind: global function

Param	Type	Description
expected	<code>string</code>	the expected successful result as specified in the test workload
actual	<code>string</code>	the successful result as returned by the engine

----
---
id: version-0.22-ref-ergo-cli
title: Command Line
original_id: ref-ergo-cli

Install the `@accordproject/ergo-cli` npm package to access the Ergo command line interface (CLI). After installation you can use the ergo command and its sub-commands as described below.

To install the Ergo CLI:
```
npm install -g @accordproject/ergo-cli
```

This will install `ergo`, to compile and run contracts locally on your machine, and `ergotop`, which is a _read-eval-print-loop_ utility to write Ergo interactively.

## Ergo

### Usage

```md
ergo <command>

Commands:
  ergo trigger     send a request to the contract
  ergo invoke      invoke a clause of the contract
  ergo initialize  initialize the state for a contract
  ergo compile     compile a contract

Options:
  --help         Show help                                             [boolean]
  --version      Show version number                                   [boolean]
  --verbose, -v                                                 [default: false]
```

## ergo trigger

`ergo trigger` allows you to send a request to the contract.

```md
Usage: ergo trigger --data [file] --state [file] --request [file] [cto files] [ergo files]

Options:
  --help         Show help                                             [boolean]
  --version      Show version number                                   [boolean]
```

~~~
  --verbose, -v                                                  [default: false]
  --data        path to the contract data                              [required]
  --state       path to the state data                [string] [default: null]
  --currentTime set current time                      [string] [default: null]
  --utcOffset   set UTC offset                        [number] [default: null]
  --request     path to the request data                    [array] [required]
  --template    path to the template directory        [string] [default: null]
  --warnings    print warnings                      [boolean] [default: false]
```

### Example

For example, using the `trigger` command for the [Volume Discount example]
(https://github.com/accordproject/ergo/tree/master/tests/volumediscount) in the
[Ergo Directory](https://github.com/accordproject/ergo):

```md
ergo trigger --template ./tests/volumediscount --data
./tests/volumediscount/data.json --request ./tests/volumediscount/request.json --
state ./tests/volumediscount/state.json
```

returns:

```json
{
  "clause": "orgXaccordprojectXvolumediscountXVolumeDiscount",
  "request": {
    "$class": "org.accordproject.volumediscount.VolumeDiscountRequest",
    "netAnnualChargeVolume": 10.4
  },
  "response": {
    "$class": "org.accordproject.volumediscount.VolumeDiscountResponse",
    "discountRate": 2.8,
    "$timestamp": "2021-06-17T09:36:53.847-04:00"
  },
  "state": {
    "$class": "org.accordproject.runtime.State",
    "$identifier": "7c19d1e3-1f70-4b30-8c3d-086dc45b1dd1"
  },
  "emit": []
}
```

As the `request` was sent for an annual charge volume of 10.4, which falls into the
third discount rate category (as specified in the `data.json` file), the `response`
returns with a discount rate of 2.8%.

## ergo invoke

`ergo invoke` allows you to invoke a specific clause of the contract. The main
difference between `ergo invoke` and `ergo trigger` is that `ergo invoke` sends
data to a specific clause, whereas `ergo trigger` lets the contract choose which
clause to invoke. This is why `--clauseName` (the name of the contract you want to
execute) is a required field for `ergo invoke`.

You need to pass the CTO and Ergo files (`--template`), the name of the contract
that you want to execute (`--clauseName`), and JSON files for: the contract data
(`--data`), the contract parameters (`--params`), the current state of the contract
~~~

(`--state`), and the request.

If contract invocation is successful, `ergorun` will print out the response, the new contract state and any emitted events.

```md
Usage: ergo invoke --data [file] --state [file] --params [file] [cto files] [ergo
files]

Options:
  --help         Show help                                             [boolean]
  --version      Show version number                                   [boolean]
  --verbose, -v                                                 [default: false]
  --clauseName   the name of the clause to invoke                     [required]
  --data         path to the contract data                            [required]
  --state        path to the state data                      [string] [required]
  --currentTime  set current time                       [string] [default: null]
  --utcOffset    set UTC offset                         [number] [default: null]
  --params       path to the parameters        [string] [required] [default: {}]
  --template     path to the template directory         [string] [default: null]
  --warnings     print warnings                       [boolean] [default: false]
```

### Example

For example, using the `invoke` command for the [Volume Discount example](https://github.com/accordproject/ergo/tree/master/tests/volumediscount) in the [Ergo Directory](https://github.com/accordproject/ergo):

```md
ergo invoke --template ./tests/volumediscount --clauseName volumediscount --
data ./tests/volumediscount/data.json --params ./tests/volumediscount/params.json
--state ./tests/volumediscount/state.json
```

returns:

```json
{
  "clause": "orgXaccordprojectXvolumediscountXVolumeDiscount",
  "params": {
    "request": {
      "$class": "org.accordproject.volumediscount.VolumeDiscountRequest",
      "netAnnualChargeVolume": 10.4
    }
  },
  "response": {
    "$class": "org.accordproject.volumediscount.VolumeDiscountResponse",
    "discountRate": 2.8,
    "$timestamp": "2021-06-17T09:38:03.189-04:00"
  },
  "state": {
    "$class": "org.accordproject.runtime.State",
    "$identifier": "b757ad1f-e011-4fda-9b37-e7157512300f"
  },
  "emit": []
}
```

Although this looks very similar to what `ergo trigger` returns, it is important to note that `--clauseName volumediscount` was specifically invoked.

## ergo initialize
`ergo initialize` allows you to obtain the initial state of the contract. This is the state of the contract without requests or responses.

```md
Usage: ergo intialize --data [file] --params [file] [cto files] [ergo files]

Options:
  --help         Show help                                           [boolean]
  --version      Show version number                                 [boolean]
  --verbose, -v                                               [default: false]
  --data         path to the contract data                          [required]
  --currentTime  set current time                     [string] [default: null]
  --utcOffset    set UTC offset                       [number] [default: null]
  --params       path to the parameters               [string] [default: null]
  --template     path to the template directory       [string] [default: null]
  --warnings     print warnings                     [boolean] [default: false]
```

### Example

For example, using the `initialize` command for the [Volume Discount example](https://github.com/accordproject/ergo/tree/master/tests/volumediscount) in the [Ergo Directory](https://github.com/accordproject/ergo):

```md
ergo initialize --template ./tests/volumediscount --data
./tests/volumediscount/data.json
```

returns:

```json
{
  "clause": "orgXaccordprojectXvolumediscountXVolumeDiscount",
  "params": {
  },
  "response": null,
  "state": {
    "$class": "org.accordproject.runtime.State",
    "$identifier": "af4f0f49-2658-4465-87f4-780e7d2e38a8"
  },
  "emit": []
}
```

## ergo compile
`ergo compile` takes your input models (`.cto` files) and input contracts (`.ergo` files) and allows you to compile a contract into a target platform. By default, Ergo compiles to JavaScript (ES6 compliant) for execution.

```md
Usage: ergo compile --target [lang] --link --monitor --warnings [cto files] [ergo
files]
```

~~~~
Options:
  --help          Show help                                            [boolean]
  --version       Show version number                                  [boolean]
  --verbose, -v                                                 [default: false]
  --target        Target platform (available: es5,es6,cicero,java)
                                                       [string] [default: "es6"]
  --link          Link the Ergo runtime with the target code (es5,es6,cicero
                  only)                               [boolean] [default: false]
  --monitor       Produce compilation time information [boolean] [default: false]
  --warnings      print warnings                      [boolean] [default: false]
```

### Example
For example, using the `compile` command on the [Volume Discount
example](https://github.com/accordproject/ergo/tree/master/tests/volumediscount) in
the [Ergo Directory](https://github.com/accordproject/ergo):

```md
ergo compile ./tests/volumediscount/model/model.cto
./tests/volumediscount/logic/logic.ergo
```

returns:

```md
Compiling Ergo './tests/volumediscount/logic/logic.ergo' --
'./tests/volumediscount/logic/logic.js'
```

Which means a new `logic.js` file is located in the `./tests/volumediscount/logic`
directory.

To compile the contract to Javascript and **link the Ergo runtime for execution**:
```md
ergo compile ./tests/volumediscount/model/model.cto
./tests/volumediscount/logic/logic.ergo --link
```

returns:

```md
Compiling Ergo './tests/volumediscount/logic/logic.ergo' --
'./tests/volumediscount/logic/logic.js'
```



--------------------------------------------------------------------------------
---
id: version-0.22-ref-migrate-0.21-0.22
title: Cicero 0.21 to 0.22
original_id: ref-migrate-0.21-0.22
---

The main change between the `0.21` release and the `0.22` release is the switch to
version `1.0` of the Concerto modeling language and library. This change comes
along with a complete revision for the Accord Project "base models" which define
key types for: clause and contract data, parties, obligations and requests /
responses. We encourage developers to get familiarized with the [new base models]
(https://github.com/accordproject/models/tree/master/src/accordproject) before
~~~~

switching to Cicero `0.22`.

:::note
Before following those migration instructions, make sure to first install version `0.22` of Cicero, as described in the [Install Cicero](started-installation) Section of this documentation.
:::

## Metadata Changes

You should only have to update the Cicero version in the `package.json` for your template to `^0.22.0`. Remember to also increment the version number for the template itself.

#### Example

After those changes, the `accordproject` field in your `package.json` should look as follows (with the `template` field being either `clause` or `contract` depending on the template):
```js
...
    "accordproject": {
        "template": "clause",
        "cicero": "^0.22.0"
    }
...
```

## Text Changes

There should be no text changes required for this version.

## Model Changes

Most templates will require changes to the model and should be re-written against the new base Accord Project models. Most of the changes should be renaming for key classes:

1. Contract and Clause data
   1. the `org.accordproject.cicero.contract.AccordContract` class is now `org.accordproject.contract.Contract` found in https://models.accordproject.org/accordproject/contract.cto
   2. the `org.accordproject.cicero.contract.AccordClause` class is now `org.accordproject.contract.Clause` found in https://models.accordproject.org/accordproject/contract.cto
2. Contract state and parties
   1. the `org.accordproject.cicero.contract.AccordState` class is now `org.accordproject.runtime.State` found in https://models.accordproject.org/accordproject/runtime.cto
   2. the `org.accordproject.cicero.contract.AccordParty` class is now `org.accordproject.party.Party` found in https://models.accordproject.org/accordproject/party.cto
3. Request and response
   1. the `org.accordproject.cicero.runtime.Request` class is now `org.accordproject.runtime.Request` found in https://models.accordproject.org/accordproject/runtime.cto
   2. the `org.accordproject.cicero.runtime.Response` class is now `org.accordproject.runtime.Response` found in https://models.accordproject.org/accordproject/runtime.cto

4. Predefined obligations have been moved to their own model file found in https://models.accordproject.org/accordproject/obligation.cto

:::warning
Some of the properties in those base classes have changed, e.g., the contract state no longer requires a `stateId`. As a result, corresponding changes to the contract logic in Ergo or to the application code may be required.
:::

### Example

A typical change to a template model might look as follows, from:
```ergo
import org.accordproject.cicero.contract.* from
https://models.accordproject.org/cicero/contract.cto
import org.accordproject.cicero.runtime.* from
https://models.accordproject.org/cicero/runtime.cto

/**
 * Defines the data model for the Purchase Order Failure
 * template.
 */
asset PurchaseOrderFailure extends AccordContract {
  o AccordParty buyer
  ...
}
```
To:
```ergo
import org.accordproject.contract.* from
https://models.accordproject.org/accordproject/contract.cto
import org.accordproject.runtime.* from
https://models.accordproject.org/accordproject/runtime.cto
import org.accordproject.party.* from
https://models.accordproject.org/accordproject/party.cto
import org.accordproject.obligation.* from
https://models.accordproject.org/accordproject/obligation.cto

asset PurchaseOrderFailure extends Contract {
  --> Party buyer
  ...
}
```

## Logic Changes

Minimal changes to the contract logic should be required, however a few changes to the base models may affect your Ergo code. Notably:
1. You should import the new Accord Project core models as needed
2. The contract state no longer requires a `stateId` field.
3. The base contract state has been moved to the runtime model, which may need to be imported

## API Changes

A number of API changes may affect Node.js applications using Cicero or Ergo packages. The main API changes are:
1. Additional `utcOffset` parameter.
   1. `@accordproject/cicero-core` package

      - the `TemplateInstance.parse` and `TemplateInstance.draft` calls take an additional `utcOffset` parameter to specify the current timezone offset
   2. `@accordproject/cicero-engine` package
      - the `Engine.init`, `Engine.invoke` and `Engine.trigger` calls take an additional `utcOffset` parameter to specify the current timezone offset
   3. `@accordproject/ergo-engine` package
      - the `Engine.init`, `Engine.invoke` and `Engine.trigger` calls take an additional `utcOffset` parameter to specify the current timezone offset
2. New `es6` compilation target for Ergo.
   1. `@accordproject/ergo-compiler` package
      - the `Compiler.compileToJavaScript` compilation target `cicero` has been renamed to `es6`
   2. `@accordproject/cicero-core` package
      - the `Template.toArchive` compilation target `cicero` has been renamed to `es6`

## CLI Changes

1. Specific UTC timezone offset now needs to be passed using the new option `--utcOffset` option has been removed

## Cicero Server Changes

There should be no text changes required for this version.

---
id: version-0.22-started-hello
title: Hello World Template
original_id: started-hello

Once you have installed Cicero, you can try it on an existing Accord Project template. This explains how to create an instance of that template and how to run the contract logic.

## Download a Template

You can download a single clause or contract template from the [Accord Project Template Library](https://templates.accordproject.org) as an archive (`.cta`) file. Cicero archives are files with a `.cta` extension, which includes all the different components for the template (text, model and logic).

If you click on the Template Library link, you should see a Web Page which looks as follows:

![Basic-Use-1](/docs/assets/basic/use1.png)

Scrolling down that page, you can see the index for the open-source templates along with their version, and whether they are a Clause or Contract template.

Click on the link to the `helloworld` template. You should be taken to a page which looks as follows:

![Basic-Use-2](/docs/assets/basic/use2.png)

Then click on the `Download Archive` button under the description for the template (highlighted in the red box in the figure). This should download the latest template archive for the `helloworld` template.

## Parse: Extract Deal Data from Text

You can use Cicero to extract deal data from a contract text using the `cicero parse` command.

### Parse Valid Text

Using your terminal, change into the directory (or `cd` into the directory) that contains the template archive you just downloaded, then create a sample clause text `sample.md` which contains the following text:

```md
Name of the person to greet: "Fred Blogs".
Thank you!
```

Then run the `cicero parse` command in your terminal to load the template and parse your sample clause text. This should be echoing the result of parsing back to your terminal.

```bash
cicero parse --template helloworld@0.14.0.cta --sample sample.md
```

:::note
* Templates are tied to a specific version of the cicero tool. Make sure that the version number output from `cicero --version` is compatible with the template. Look for `^0.22.0` or similar at the top of the template web page.
* `cicero parse` requires network access. Make sure that you are online and that your firewall or proxy allows access to `https://models.accordproject.org`
:::

This should extract the data (or "deal points") from the text and output:

```json
{
  "$class": "org.accordproject.helloworld.HelloWorldClause",
  "name": "Fred Blogs",
  "clauseId": "71045314-acfc-441f-92b4-0a2707ea6146",
  "$identifier": "71045314-acfc-441f-92b4-0a2707ea6146"
}
```

You can save the result of `cicero parse` into a file using the `--output` option:
```
cicero parse --template helloworld@0.14.0.cta --sample sample.md --output data.json
```

### Parse Non-Valid Text

If you attempt to parse text which is not valid according to the template, this same command should return an error.

Edit your `sample.md` file to add text that is not consistent with the template:

```text
FUBAR Name of the person to greet: "Fred Blogs".
Thank you!
```

Then run `cicero parse --template helloworld@0.14.0.cta --sample sample.md` again. The output should now be:

```text
2:13:15 AM - error: Parse error at line 1 column 1
FUBAR Name of the person to greet: "Fred Blogs".
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Expected: 'Name of the person to greet: '
```

## Draft: Create Text from Deal Data

You can use Cicero to create new contract text from deal data using the `cicero draft` command.

### Draft from Valid Data

If you have saved the deal data earlier in a `data.json` file, you can edit it to change the name from `Fred Blogs` to `John Doe`, or create a brand new `data.json` file containing:
```json
{
  "$class": "org.accordproject.helloworld.HelloWorldClause",
  "clauseId": "aa3b9db9-f25f-41f4-88a4-64baba728bfe",
  "name": "John Doe"
}
```

Then run the `cicero draft` command in your terminal:
```
cicero draft --template helloworld@0.14.0.cta --data data.json
```

This should create a new contract text and output:
```
13:17:18 - INFO: Name of the person to greet: "John Doe".
Thank you!
```

You can save the result of `cicero draft` into a file using the `--output` option:
```
cicero draft --template helloworld@0.14.0.cta --data data.json --output new-sample.md
```

### Draft from Non-Valid Data

If you attempt to draft from data which is not valid according to the template, this same command should return an error.

Edit your `data.json` file so that the `name` variable is missing:
```json
{
  "$class": "org.accordproject.helloworld.HelloWorldClause",
  "clauseId": "aa3b9db9-f25f-41f4-88a4-64baba728bfe"
}
```

Then run `cicero draft --template helloworld@0.14.0.cta --data data.json` again. The output should now be:
```
13:38:11 - ERROR: Instance org.accordproject.helloworld.HelloWorldClause#6f91e060-f837-4108-bead-63891a91ce3a missing required field name
```

## Trigger: Run the Contract Logic

You can use Cicero to run the logic associated to a contract using the `cicero trigger` command.

### Trigger with a Valid Request

Use the `cicero trigger` command to parse a clause text based (your `sample.md`) *then* send a request to the clause logic.

To do so, you first create one additional file `request.json` which contains:
```json
{
  "$class": "org.accordproject.helloworld.MyRequest",
  "input": "Accord Project"
}
```

This is the request which you will send to trigger the execution of your contract.

Then run the `cicero trigger` command in your terminal to load the template, parse your clause text *and* send the request. This should be echoing the result of execution back to your terminal.

```bash
cicero trigger --template helloworld@0.14.0.cta --sample sample.md --request request.json
```

This should print this output:

```json
13:42:29 - INFO:
{
  "clause": "helloworld@0.14.0-767ffde65292f2f4e8aa474e76bb5f923b80aa29db635cd42afebb6a0cd4c1fa",
  "request": {
    "$class": "org.accordproject.helloworld.MyRequest",
    "input": "Accord Project"
  },
  "response": {
    "$class": "org.accordproject.helloworld.MyResponse",
    "output": "Hello Fred Blogs Accord Project",
    "$timestamp": "2021-06-16T11:38:42.011-04:00"
  },
  "state": {
    "$class": "org.accordproject.runtime.State",
    "$identifier": "f4428ec2-73ca-442b-8006-8e9a290930ad"
  },
  "emit": []
}
```

The results of execution displayed back on your terminal is in JSON format. It includes the following information:

* Details of the `clause` being triggered (name, version, SHA256 hash of the template)
* The incoming `request` object (the same request from your `request.json` file)
* The output `response` object
* The output `state` (unchanged in this example)
* An array of `emit`ted events (empty in this example)

That's it! You have successfully parsed and executed your first Accord Project Clause using the `helloworld` template.

### Trigger with a Non-Valid Request

If you attempt to trigger the contract from a request which is not valid according to the template, this same command should return an error.

Edit your `request.json` file so that the `input` variable is missing:
```json
{
  "$class": "org.accordproject.helloworld.MyRequest"
}
```

Then run `cicero trigger --template helloworld@0.14.0.cta --sample sample.md --request request.json` again. The output should now be:
```
13:47:35 - ERROR: Instance org.accordproject.helloworld.MyRequest#null missing required field input
```

## What Next?

### Try Other Templates

Feel free to try the same commands to parse and execute other templates from the Accord Project Library. Note that for each template, you can find samples for the text, for the request and for the state on the corresponding Web page. For instance, a sample for the [Late Delivery And Penalty](https://templates.accordproject.org/latedeliveryandpenalty@0.15.0.html) clause is in the red box in the following image:

![Basic-Use-3](/docs/assets/basic/use3.png)

### More About Cicero

You can find more information on how to create or publish Accord Project templates in the [Work with Cicero](tutorial-templates) tutorials.

### Run on Different Platforms

Templates may be executed on different platforms, not only from the command line. You can find more information on how to execute Accord Project templates on different platforms (Node.js, Hyperledger Fabric, etc.) in the [Template Execution](tutorial-nodejs) tutorials.

~~~
--------------------------------------------------------------------------------
---
id: version-0.22-tutorial-create
title: Template Generator
original_id: tutorial-create
---

Now that you have executed an existing template, let's create a new template from
scratch. To facilitate the creation of new templates, Cicero comes with a template
generator.

## The template generator

### Install the generator

If you haven't already done so, first install the template generator::

```bash
npm install -g yo
npm install -g yo @accordproject/generator-cicero-template
```

### Run the generator:

You can now try the template generator by running the following command in a
terminal window:
```bash
yo @accordproject/cicero-template
```
This will ask you a series of questions. Give your generator a name (no spaces) and
then supply a namespace for your template model (again,no spaces). The generator
will then create the files and directories required for a basic template (similar
to the helloworld template).

Here is an example of how it should look like in your terminal window:
```bash
bash-3.2$ yo @accordproject/cicero-template

     _-----_     ╭──────────────────────────╮
    |       |    │      Welcome to the      │
    |--(o)--|    │ generator-cicero-templat │
   `---------´   │       e generator!       │
    ( _´U`_ )    ╰──────────────────────────╯
    /___A___\   /
     |  ~  |
   __'.___.'__
 ´   `  |° ´ Y `

? What is the name of your template? mylease
? Who is the author? me
? What is the namespace for your model? org.acme.lease
   create mylease/README.md
   create mylease/logo.png
   create mylease/package.json
   create mylease/request.json
   create mylease/logic/logic.ergo
   create mylease/model/model.cto
   create mylease/test/logic_default.feature
~~~

~~~~
   create mylease/text/grammar.tem.md
   create mylease/text/sample.md
   create mylease/.cucumber.js
   create mylease/.npmignore
bash-3.2$
```

:::tip
You may find it easier to edit the grammar, model and logic for your template in
[VSCode](https://code.visualstudio.com/), installing the [Accord Project extension]
(https://marketplace.visualstudio.com/items?itemName=accordproject.cicero-vscode-
extension). The extension gives you syntax highlighting and parser errors within VS
Code.

For more information on how to use VS Code with the Accord Project extension,
please consult the [Using the VS Code extension](tutorial-vscode) tutorial.
:::

## Test your template

If you have Cicero installed on your machine, you can go into the newly created
`mylease` directory and try it with cicero, to make sure the contract text parses:
```bash
bash-3.2$ cicero parse
11:51:40 AM - info: Using current directory as template folder
11:51:40 AM - info: Loading a default text/sample.md file.
11:51:41 AM - info:
{
  "$class": "org.acme.lease.MyContract",
  "name": "Dan",
  "contractId": "4a7d5b59-0377-42d3-aa41-15062398d25d",
  "$identifier": "4a7d5b59-0377-42d3-aa41-15062398d25d"
}
```
And that you can trigger the contract:
```bash
bash-3.2$ cicero trigger
11:58:22 AM - info: Using current directory as template folder
11:58:22 AM - info: Loading a default text/sample.md file.
11:58:22 AM - info: Loading a default request.json file.
11:58:23 AM - warn: A state file was not provided, initializing state. Try the --
state flag or create a state.json in the root folder of your template.
11:58:23 AM - info:
{
  "clause": "mylease@0.0.0-
d186ab29c448b0058e4465a54d8376c3817dddb6fda8dc0ca29a88151b3dbecc",
  "request": {
    "$class": "org.acme.lease.MyRequest",
    "input": "World"
  },
  "response": {
    "$class": "org.acme.lease.MyResponse",
    "output": "Hello Dan World",
    "$timestamp": "2021-06-16T12:29:50.317-04:00"
  },
  "state": {
    "$class": "org.accordproject.runtime.State",
    "$identifier": "03515461-7ee7-4c81-a8f0-d4c667db5f4c"
  },
~~~~

```
  "emit": []
}
```

The template also comes with a few simple tests which you can run by first doing an
`npm install` in the template directory, then by running `npm run test`:
```bash
bash-3.2$ npm install
bash-3.2$ npm run test

> mylease@0.0.0 test /Users/jeromesimeon/tmp/mylease
> cucumber-js test -r .cucumber.js

....

1 scenario (1 passed)
3 steps (3 passed)
0m01.257s
bash-3.2$
```

## Edit your template

### Update Sample.md

First, replace the contents of `./text/sample.md` with the legal text for the
contract or clause that you would like to digitize.

Check that when you run `cicero parse` that the new `./text/sample.md` is now
invalid with respect to the grammar.

### Edit the Template Grammar

Now update the grammar in `./text/grammar.tem.md`. Start by replacing the existing
grammar, making it identical to the contents of your updated `./text/sample.md`.

Now introduce variables into your template grammar as required. The variables are
marked-up using `{{` and `}}` with what is between the braces being the name of
your variable.

### Edit the Template Model

All of the variables referenced in your template grammar must exist in your
template model. Edit
the file `model/model.cto` to include all your variables, making sure the name of
the model property matches the name of the variable in the `./text/grammar.tem.md`
file.

Note that the Concerto Modeling Language primitive data types are:

- `String`: for character strings
- `Long` or `Integer`: for integer values
- `DateTime`: for dates and times
- `Double`: for floating points numbers
- `Boolean`: for values that are either true or false

:::tip
Note that you can import common types (address, monetary amount, country code,
etc.) from the Accord Project Model Repository: https://models.accordproject.org.

:::

### Edit the Transaction Types

Your template expects to receive data as input and will produce data as output. The structure of
this request/response data is captured in the `MyRequest` and `MyResponse` transaction types in your model
namespace. Open up the file `models/model.cto` and edit the definition of the `MyRequest` type to
include all the data you expect to receive from the outside world and that will be used by the
business logic of your template. Similarly edit the definition of the `MyResponse` type to include
all the data that the business logic for your template will compute and would like to return to the
caller.

### Edit the Template Logic

Now edit the business logic of the template itself. This is expressed in the Ergo language, which is a strongly-typed function domain specific language for contract logic. Open the file `logic/logic.ergo`
and edit the `helloworld` clause to perform the calculations your logic requires.

Looking at the Ergo logic for other example templates will help you understand the syntax and capabilities of Ergo.

## Publishing your template

If you would like to publish your new template in the Accord Project Template Library, please consult the [Template Library](tutorial-library) Section of this documentation.

---
id: version-0.22-tutorial-library
title: Template Library
original_id: tutorial-library

This tutorial explains how to get access, and contribute, to all of the public templates available as part of the the [Accord Project Template Library](https://templates.accordproject.org).

## Setting up

### Prerequisites

Accord Project uses [GitHub](https://github.com/) to maintain its open source template library. For this tutorial, you must first obtain and configure the following dependency:

* [Git](https://git-scm.com): a distributed version-control system for
  tracking changes in source code during software development.
* [Lerna](https://lerna.js.org/): A tool for managing JavaScript projects with multiple packages. You can install lerna by running the following command in your terminal:

```bash
npm install -g lerna
```

### Clone the template library

Once you have `git` installed on your machine, you can run `git clone` to create a
version of all the templates:

```bash
git clone https://github.com/accordproject/cicero-template-library
```

Alternatively, you can download the library directly by visiting the [GitHub
Repository for the Template Library](https://github.com/accordproject/cicero-
template-library) and use the "Download" button as shown on this snapshot:

![Basic-Library-1](/docs/assets/basic/library1.png)

### Install the Library

Once cloned, you can set up the library for development by running the following
commands inside your template library directory:

```bash
lerna bootstrap
```

### Running all the template tests

To check that the installation was successful, you can run all the tests for all
the Accord Project templates by running:

```bash
lerna run test
```

## Structure of the Repository

You can see the source code for all public Accord Project templates by looking
inside the `./src` directory:

```sh
bash-3.2$ ls src
acceptance-of-delivery
bill-of-lading
car-rental-tr
certificate-of-incorporation
company-information
contact-information
copyright-license
demandforecast
docusign-connect
docusign-po-failure
eat-apples
empty
empty-contract
fixed-interests
```

...
```

Each of those templates directories have the same structure, as described in the
[Templates Deep Dive](tutorial-templates) Section. For instance for the
`acceptance-of-delivery` template:
```
$ cd src/acceptance-of-delivery
$ bash-3.2$ ls -laR
./README.md
./package.json

./text:
  ./grammar.tem.md
  ./sample.md

./logic:
   logic.ergo

./model:
   model.cto

./test:
  logic.feature
  logic_default.feature

./request.json
./state.json

## Use a Template

To use a template, simply run the same Cicero commands we have seen in the previous
tutorials. For instance, to extract the deal data from the `./text/sample.md` text
sample for the `acceptance-of-delivery` template, run:

```bash
cicero parse --template ./src/acceptance-of-delivery
```
You should see a response as follows:
```json
{
  "$class": "org.accordproject.acceptanceofdelivery.AcceptanceOfDeliveryClause",
  "shipper": "resource:org.accordproject.organization.Organization#Party%20A",
  "receiver": "resource:org.accordproject.organization.Organization#Party%20B",
  "deliverable": "Widgets",
  "businessDays": 10,
  "attachment": "Attachment X",
  "clauseId": "f1b1434b-8500-4672-8678-7c5003d8d66b",
  "$identifier": "f1b1434b-8500-4672-8678-7c5003d8d66b"
}
```

Or, to extract the deal data from the `./text/sample.md` then send the default
request in `./request.json` for the `latedeliveryandpenalty` template, run:
```bash
cicero trigger --template ./src/latedeliveryandpenalty
```
You should see a response as follows:

```json
{
  "clause": "latedeliveryandpenalty@0.17.0-
a4e00f4f161e2d343a239a6854bfce92ecd16d891f8e7bc5a5adaab46d242782",
  "request": {
    "$class":
"org.accordproject.latedeliveryandpenalty.LateDeliveryAndPenaltyRequest",
    "forceMajeure": false,
    "agreedDelivery": "2017-12-17T03:24:00-05:00",
    "deliveredAt": null,
    "goodsValue": 200
  },
  "response": {
    "$class":
"org.accordproject.latedeliveryandpenalty.LateDeliveryAndPenaltyResponse",
    "penalty": 110.00000000000001,
    "buyerMayTerminate": true,
    "$timestamp": "2021-06-16T12:26:18.031-04:00"
  },
  "state": {
    "$class": "org.accordproject.runtime.State",
    "$identifier": "54810499-acad-4a3a-9f78-684b0a3bef65"
  },
  "emit": [
    {
      "$class": "org.accordproject.obligation.PaymentObligation",
      "amount": {
        "$class": "org.accordproject.money.MonetaryAmount",
        "doubleValue": 110.00000000000001,
        "currencyCode": "USD"
      },
      "description": ""resource:org.accordproject.party.Party#Dan" should pay
penalty amount to "resource:org.accordproject.party.Party#Steve"",
      "$identifier": "a9482b16-c0dc-4e09-86bc-60bb59b07523",
      "contract":
"resource:org.accordproject.latedeliveryandpenalty.LateDeliveryAndPenaltyContract#3
fecad6b-442c-49d1-99d8-b963616f61d2",
      "promisor": "resource:org.accordproject.party.Party#Dan",
      "promisee": "resource:org.accordproject.party.Party#Steve",
      "$timestamp": "2021-06-16T12:26:18.032-04:00"
    }
  ]
}
```

## Contribute a New Template

To contribute a change to the Accord Project library, please [fork](https://help.github.com/en/github/getting-started-with-github/fork-a-repo) the repository and then create a [pull request](https://help.github.com/en/github/collaborating-with-issues-and-pull-requests/about-pull-requests).

Note that templates should have unit tests. See any of the `./test` directories in the templates contained in the template library for an examples with unit tests, or consult the [Testing Reference](ref-testing) Section of this documentation.

---
id: version-0.22-tutorial-nodejs
title: With Node.js
original_id: tutorial-nodejs

## Cicero Node.js API

You can work with Accord Project templates directly in JavaScript using Node.js.

Documentation for the API can be found in [Cicero API](ref-cicero-api.html).

## Working with Templates

### Import the Template class

To import the Cicero classes for templates and clauses, we'll also import the
Cicero engine and some helper utilities

```js
const fs = require("fs");
const path = require("path");
const { Template, Clause } = require("@accordproject/cicero-core");
const { Engine } = require("@accordproject/cicero-engine");
```

### Load a Template

To create a Template instance in memory call the `fromDirectory`, `fromArchive` or
`fromUrl` methods:

```js
const template = await Template.fromDirectory(
  "./test/data/latedeliveryandpenalty"
);
```

These methods are asynchronous and return a `Promise`, so you should use `await` to
wait for the promise to be resolved.

> Note that you'll need to wrap this `await` inside an `async` function or use a
[top-level await inside a module](https://v8.dev/features/top-level-await)

### Instantiate a Template

Once a Template has been loaded, you can create a Clause based on the Template. You
can either instantiate
the Clause using source DSL text (by calling `parse`), or you can set an instance
of the template model
as JSON data (by calling `setData`):

```js
// load the DSL text for the template
const testLatePenaltyInput = fs.readFileSync(
  path.resolve(__dirname, "text/", "sample.md"),
  "utf8"
);
```

~~~
const clause = new Clause(template);
clause.parse(testLatePenaltyInput);

// get the JSON object created from the parse
const data = clause.getData();
```

## Executing a Template Instance

Once you have instantiated a clause or contract instance, you can execute it.

### Import the Engine class

To execute a Clause you first need to create an instance of the `Engine` class:

```js
const engine = new Engine();
```

### Send a request to the contract

You can then call `execute` on it, passing in the clause or contract instance, and
the request:

```js
const request = {
  $class:
    "org.accordproject.latedeliveryandpenalty.LateDeliveryAndPenaltyRequest",
  forceMajeure: false,
  agreedDelivery: "2017-10-07T16:38:01.412Z",
  goodsValue: 200,
};
const state = {
  $class: "org.accordproject.runtime.State",
};

const result = await engine.trigger(clause, request, state);
console.log(result);
```

--------------------------------------------------------------------------------
---
id: version-0.22-tutorial-studio
title: With Template Studio
original_id: tutorial-studio
---

This tutorial will walk you through the steps of editing a clause template in
[Template Studio](https://studio.accordproject.org/).

We start with a very simple _Late Penalty and Delivery_ Clause and gradually make
it more complex, adding both legal text to it and the corresponding business logic
in Ergo.

## Initial Late Delivery Clause

### Load the Template

To get started, head to the `minilatedeliveryandpenalty` template in the Accord
~~~

Project Template Library at [Mini Late Delivery And Penalty](https://templates.accordproject.org/minilatedeliveryandpenalty@0.6.0.html) and click the "Open In Template Studio" button.

![Advanced-Late-1](assets/advanced/late1.png)

Begin by inspecting the `README` and `package.json` tabs within the `Metadata` section. Feel free to change the name of the template to one you like.

### The Contract Text

Then click on the `Text` Section on the left, which should show a `Grammar` tab, for the the natural language of the template.

![Advanced-Late-2](assets/advanced/late2.png)

When the text in the `Grammar` tab is in sync with the text in the `Sample` tab, this means the sample is a valid with respect to the grammar, and data is extracted, showing in `Contract Data` tab. The contract data is represented using the JSON format and contains the value of the variables declared in the contract template. For instance, the value for the `buyer` variable is `Betty Buyer`, highlighted in red:

![Advanced-Late-3](assets/advanced/late3.png)

Changes to the variables in the `Sample` are reflected in the `Contract Data` tab in real time, and vice versa. For instance, change `Betty Buyer` to a different name in the contract text to see the `partyId` change in the contract data.

:::note
The JSON data `resource:org.accordproject.party.Party#Betty%20Buyer` indicate that the value is a relationship of type `Party` whose identifier is `Betty Buyer`. Consult the [Concerto Guide](model-relationships) for more details on modeling relationships.
:::

If you edit part of the text which is not a variable in the template, this results in an error when parsing the `Sample`. The error will be shown in red in the status bar at the bottom of the page. For instance, the following image shows the parsing error obtained when changing the word `delayed` to the word `timely` in the contract text.

![Advanced-Late-4](assets/advanced/late4.png)

This is because the `Sample` relies on the `Grammar` text as a source of truth. This mechanism ensures that the actual contract always reflects the template, and remains faithful to the original legal text. You can, however, edit the `Grammar` itself to change the legal text.

Revert your changes, changing the word `timely` back to the original word `delayed` and the parsing error will disappear.

### The Model

Moving along to the `Model` section, you will find the data model for the template variables (the `MiniLateDeliveryClause` type), as well as for the requests (the `LateRequest` type) and response (the `LateResponse` type) for the late delivery and penalty clause.

![Advanced-Late-5](assets/advanced/late5.png)

Note that a `namespace` is declared at the beginning of the file for the model, and that several existing models are being imported (using e.g., `import org.accordproject.contract.*`). Those imports are needed to access the definition for several types used in the model:
- `Clause` which is a generic type for all Accord Project clause templates, and is defined in the `org.accordproject.contract` namespace;
- `Party` which is a generic type for all Accord Project parties, and is defined in the `org.accordproject.party` namespace;
- `Request` and `Response` which are generic types for responses and requests, and are defined in the `org.accordproject.runtime` namespace;
- `Duration` which is defined in the `org.accordproject.time` namespace.

### The Logic

The final part of the template is the `Ergo` tab of the `Logic` section, which describes the business logic.

![Advanced-Late-6](assets/advanced/late6.png)

Thanks to the `namespace` at the beginning of this file, the Ergo engine can know the definition for the `MiniLateDeliveryClause`, as well as the `LateRequest`, and `LateResponse` types defined in the `Model` tab.

To test the template execution, go to the `Request` tab in the `Logic` section. It should be already populated with a valid request. Press the `Trigger` button to trigger the clause.

![Advanced-Late-7](assets/advanced/late7.png)

Since the value of the `deliveredAt` parameter in the request is after the value of the `agreedDelivery` parameter in the request, this should return a new response which includes the calculated penalty.

Changing the date for the `deliveredAt` parameter in the request and triggering the contract again will result in a different penalty.

![Advanced-Late-8](assets/advanced/late8.png)

Note that the clause will return an error if it is called for a timely delivery.

![Advanced-Late-9](assets/advanced/late9.png)

## Add a Penalty Cap

We can now start building a more advanced clause. Let us first take a moment to notice that there is no limitation to the penalty resulting from a late delivery. Trigger the contract using the following request in the `Request` tab in `Logic`:
```json
{
  "$class": "org.accordproject.minilatedeliveryandpenalty.LateRequest",
  "agreedDelivery": "2019-04-10T12:00:00-05:00",
  "deliveredAt": "2019-04-20T03:24:00-05:00",
  "goodsValue": 200
}
```
The penalty should be rather low. Now send this other request:
```json
```

~~~
{
  "$class": "org.accordproject.minilatedeliveryandpenalty.LateRequest",
  "agreedDelivery": "2005-04-01T12:00:00-05:00",
  "deliveredAt": "2019-04-20T03:24:00-05:00",
  "goodsValue": 200
}
```
Notice that the penalty is now quite a large value. It is not unusual to cap a
penalty to a maximum amount. Let us now look at how to change the template to add
such a cap based on a percentage of the total value of the delivered goods.

### Update the Legal Text

To implement this, we first go to the `Grammar` tab in the `Text` section and add a
sentence indicating: `The total amount of penalty shall not, however, exceed
{{capPercentage}}% of the total value of the delayed goods.`

For convenience, you can copy-paste the new template text from here:
```tem
Late Delivery and Penalty.

In case of delayed delivery of Goods, {{seller}} shall pay to
{{buyer}} a penalty amounting to {{penaltyPercentage}}% of the total
value of the Goods for every {{penaltyDuration}} of delay. The total
amount of penalty shall not, however, exceed {{capPercentage}}% of the
total value of the delayed goods. If the delay is more than
{{maximumDelay}}, the Buyer is entitled to terminate this Contract.

```
This should immediately result in an error when parsing the contract text:

![Advanced-Late-10](assets/advanced/late10.png)

As explained in the error message, this is because the new template text uses a
variable `capPercentage` which has not been declared in the model.

### Update the Model

To define this new variable, go to the `Model` tab, and change the
`MiniLateDeliveryClause` type to include `o Double capPercentage`.

![Advanced-Late-11](assets/advanced/late11.png)

For convenience, you can copy-paste the new `MiniLateDeliveryClause` type from
here:
```ergo
asset MiniLateDeliveryClause extends Clause {
  --> Party buyer         // Party to the contract (buyer)
  --> Party seller        // Party to the contract (seller)
  o Duration penaltyDuration  // Length of time resulting in penalty
  o Double penaltyPercentage  // Penalty percentage
  o Double capPercentage      // Maximum penalty percentage
  o Duration maximumDelay     // Maximum delay before termination
}
```

This results in a new error, this time on the sample contract:

![Advanced-Late-12](assets/advanced/late12.png)
~~~

To fix it, we need to add that same line we added to the template, replacing the `capPercentage` by a value in the `Test Contract`: `The total amount of penalty shall not, however, exceed 52% of the total value of the delayed goods.`

For convenience, you can copy-paste the new test contract from here:
```md
Late Delivery and Penalty.

In case of delayed delivery of Goods, "Steve Seller" shall pay to
"Betty Buyer" a penalty amounting to 10.5% of the total
value of the Goods for every 2 days of delay. The total
amount of penalty shall not, however, exceed 52% of the
total value of the delayed goods. If the delay is more than
15 days, the Buyer is entitled to terminate this Contract.

```

Great, now the edited template should have no more errors, and the contract data should now include the value for the new `capPercentage` variable.

![Advanced-Late-13](assets/advanced/late13.png)

Note that the `Current Template` Tab indicates that the template has been changed.

### Update the Logic

At this point, executing the logic will still result in large penalties. This is because the logic does not take advantage of the new `capPercentage` variable. Edit the `logic.ergo` code to do so. After step `// 2. Penalty formula` in the logic, apply the penalty cap by adding some logic as follows:
```ergo
    // 3. Capped Penalty
    let cap = contract.capPercentage / 100.0 * request.goodsValue;

    let cappedPenalty =
      if penalty > cap
      then cap
      else penalty;

```
Do not forget to also change the value of the penalty in the returned `LateResponse` to use the new variable `cappedPenalty`:
```ergo
    // 5. Return the response
    return LateResponse{
      penalty: cappedPenalty,
      buyerMayTerminate: termination
    }
```
The logic should now look as follows:

![Advanced-Late-14](assets/advanced/late14.png)

### Run the new Logic

As a final test of the new template, you should try again to run the contract with a long delay in delivery. This should now result in a much smaller penalty, which is capped to 52% of the total value of the goods, or 104 USD.

![Advanced-Late-15](assets/advanced/late15.png)

:::tip
A full version of the template after those changes have been applied can be found as the [Mini Late Delivery And Penalty Capped](https://templates.accordproject.org/minilatedeliveryandpenalty-capped@0.6.0.html) in the Template Library.
:::

## Emit a Payment Obligation.

As a final extension to this template, we can modify it to emit a Payment Obligation. This first requires us to switch from a Clause template to a Contract template.

### Switch to a Contract Template

The first place to change is in the metadata for the template. This can be done easily with the `full contract` button in the `Current Template` tab. This will immediately result in an error indicating that the model does not contain an `Contract` type.

![Advanced-Late-16](assets/advanced/late16.png)

### Update the Model

To fix this, change the model to reflect that we are now editing a contract template, and change the type `AccordClause` to `AccordContract` in the type definition for the template variables:
```ergo
asset MiniLateDeliveryContract extends Contract {
  --> Party buyer         // Party to the contract (buyer)
  --> Party seller        // Party to the contract (seller)
  o Duration penaltyDuration  // Length of time resulting in penalty
  o Double penaltyPercentage  // Penalty percentage
  o Double capPercentage      // Maximum penalty percentage
  o Duration maximumDelay     // Maximum delay before termination
}
```

The next error is in the logic, since it still uses the old `MiniLateDeliveryClause` type which does not exist anymore.

### Update the Logic

The `Logic` error that occurs here is:
```bash
Compilation error (at file lib/logic.ergo line 19 col 31). Cannot find type with name 'MiniLateDeliveryClause'
contract MiniLateDelivery over MiniLateDeliveryClause {
                               ^^^^^^^^^^^^^^^^^^^^^^
```
Update the logic to use the the new `MiniLateDeliveryContract` type instead, as follows:
```ergo
contract MiniLateDelivery over MiniLateDeliveryContract {
```

The template should now be without errors.

### Add a Payment Obligation

Our final task is to emit a `PaymentObligation` to indicate that the buyer should pay the seller in the amount of the calculated penalty.

To do so, first import a couple of standard models: for the Cicero's [runtime model](https://models.accordproject.org/cicero/runtime.html) (which contains the definition of a `PaymentObligation`), and for the Accord Project's [money model](https://models.accordproject.org/money.html) (which contains the definition of a `MonetaryAmount`). The `import` statements at the top of your logic should look as follows:
```ergo
import org.accordproject.time.*
import org.accordproject.cicero.runtime.*
import org.accordproject.money.MonetaryAmount

```

Lastly, add a new step between steps `// 4.` and `// 5.` in the logic to emit a payment obligation in USD:
```ergo
    emit PaymentObligation{
      contract: contract,
      promisor: some(contract.seller),
      promisee: some(contract.buyer),
      deadline: none,
      amount: MonetaryAmount{ doubleValue: cappedPenalty, currencyCode: USD },
      description: contract.seller.partyId ++ " should pay penalty amount to " ++
contract.buyer.partyId
    };

```
That's it! You can observe in the `Request` tab that an `Obligation` is now being emitted. Try out adjusting values and continuing to send requests and getting responses and obligations.

![Advanced-Late-17](assets/advanced/late17.png)

:::tip
A full version of the template after those changes have been applied can be found as the [Mini-Late Delivery and Penalty Payment](https://templates.accordproject.org/minilatedeliveryandpenalty-payment@0.6.0.html) in the Template Library.
:::

---
id: version-0.22-tutorial-templates
title: Templates Deep Dive
original_id: tutorial-templates

In the [Getting Started](started-hello) section, we learned how to use the existing [helloworld@0.14.0.cta](https://templates.accordproject.org/archives/helloworld@0.14.0.cta) template archive. Here we take a look inside that archive to understand the structure of Accord Project templates.

## Unpack a Template Archive

A `.cta` archive is nothing more than a zip file containing the components of a template. Let's unzip that archive to see what is inside. First, create a directory in the place where you have downloaded that archive, then run the unzip command in a terminal:

```bash
$ mkdir helloworld
$ mv helloworld@0.14.0.cta helloworld
$ cd helloworld
$ unzip helloworld@0.14.0.cta
Archive:  helloworld@0.14.0.cta
 extracting: package.json
   creating: text/
 extracting: text/grammar.tem.md
 extracting: README.md
 extracting: text/sample.md
 extracting: request.json
   creating: model/
 extracting: model/@models.accordproject.org.time@0.2.0.cto
 extracting: model/@models.accordproject.org.accordproject.money@0.2.0.cto
 extracting: model/@models.accordproject.org.accordproject.contract.cto
 extracting: model/@models.accordproject.org.accordproject.runtime.cto
 extracting: model/@org.accordproject.ergo.options.cto
 extracting: model/model.cto
   creating: logic/
 extracting: logic/logic.ergo
```

## Template Components

Once you have unziped the archive, the directory should contain the following files and sub-directories:

```text
package.json
    Metadata for the template (name, version, description etc)

README.md
    A markdown file that describes the purpose and correct usage for the template

text/grammar.tem.md
    The default grammar for the template

text/sample.md
    A sample clause or contract text that is valid for the template

model/
    A collection of Concerto model files for the template. They define the Template
Model
    and models for the State, Request, Response, and Obligations used during
execution.

logic/
    A collection of Ergo files that implement the business logic for the template

test/
    A collection of unit tests for the template
```

state.json (optional)
    A sample valid state for the clause or contract

request.json (optional)
    A sample valid request to trigger execution for the template

In a nutshell, the template archive contains the three main components of the [Template Triangle](accordproject-concepts#what-is-a-template) in the corresponding directories (the natural language text of your clause or contract in the `text` directory, the data model in the `model` directory, and the contract logic in the `logic` directory). Additional files include metadata and samples which can be used to illustrate or test the template.

Let us look at each of those components.

### Template Text

#### Grammar

The file in `text/grammar.tem.md` contains the grammar for the template. It is natural language, with markup to indicate the variable(s) in your Clause or Contract.

```tem
Name of the person to greet: {{name}}.
Thank you!
```

In the `helloworld` template there is only one variable `name` which is indicated between `{{` and `}}`.

#### Sample Text

The file in `text/sample.md` contains a sample valid for that grammar.

```md
Name of the person to greet: "Fred Blogs".
Thank you!
```

### Template Model

The file in `model/model.cto` contains the data model for the template. This includes a description for each of the template variables, including what kind of variable it is (also called their [type](ref-glossary.html#components-of-data-models)).

Here is the model for the `helloworld` template:

```ergo
namespace org.accordproject.helloworld

import org.accordproject.contract.* from
https://models.accordproject.org/accordproject/contract.cto
import org.accordproject.runtime.* from
https://models.accordproject.org/accordproject/runtime.cto
```

~~~
transaction MyRequest extends Request {
  o String input
}

transaction MyResponse extends Response {
  o String output
}

/**
 * The template model
 */
asset HelloWorldClause extends Clause {
  /**
   * The name for the clause
   */
  o String name
}
```

The `HelloWorldClause` as well as the `Request` and `Response` are types which are
specified using the [Concerto modeling
language](https://github.com/accordproject/concerto).

The `HelloWorldClause` indicate that the template is for a Clause, and should have
a variable `name` of type `String` (i.e., text).

```ergo
asset HelloWorldClause extends Clause {
  o String name // variable 'name' is of type String
}
```

Types are always declared within a namespace (here `org.accordproject.helloworld`),
which provides a mechanism to disambiguate those types amongst multiple model
files.

### Template Logic

The file in `logic/logic.ergo` contains the executable logic. Each Ergo file is
identified by a namespace, and contains declarations (e.g., constants, functions,
contracts). Here is the Ergo logic for the `helloworld` template:

```ergo
namespace org.accordproject.helloworld

contract HelloWorld over TemplateModel {
  // Simple Clause
  clause greet(request : MyRequest) : MyResponse {
    return MyResponse{ output: "Hello " ++ contract.name ++ " " ++ request.input }
  }
}
```

This declares a single `HelloWorld` contract in the `org.accordproject.helloworld`
namespace, with one `greet` clause.

It also declares that this contract `HelloWorld` is parameterized over the given
`TemplateModel` found in the `models/model.cto` file.
~~~

The `greet` clause takes a request of type `MyRequest` as input and returns a response of type `MyResponse`.

The code for the `greet` clause returns a new `MyResponse` response with a single property `output` which is a string. That string is constructed using the string concatenation operator (`++`) in Ergo from the `name` in the contract (`contract.name`) and the input from the request (`request.input`).

## Use the Template

Even after you have unzipped the template archive, you can use that template from the directory directly, in the same way we did from the `.cta` archive in the [Getting Started](started-hello) section.

For instance you can use `cicero parse` or `cicero trigger` as follows:
```bash
$ cd helloworld
$ cicero parse
12:21:37 PM - INFO: Using current directory as template folder
12:21:37 PM - INFO: Loading a default text/sample.md file.
12:21:38 PM - INFO:
{
  "$class": "org.accordproject.helloworld.HelloWorldClause",
  "name": "Fred Blogs",
  "clauseId": "ca447073-242f-4721-a5b9-c5c14b57233d",
  "$identifier": "ca447073-242f-4721-a5b9-c5c14b57233d"
}
$ cicero trigger
12:21:54 PM - INFO: Using current directory as template folder
12:21:54 PM - INFO: Loading a default text/sample.md file.
12:21:54 PM - INFO: Loading a default request.json file.
12:21:55 PM - WARN: A state file was not provided, initializing state. Try the --
state flag or create a state.json in the root folder of your template.
12:21:55 PM - INFO:
{
  "clause": "helloworld@0.14.0-
4f8006ff0471176f2b5340500ba40c42adb180f26df50b747d8690c6dad79cfa",
  "request": {
    "$class": "org.accordproject.helloworld.MyRequest",
    "input": "Accord Project"
  },
  "response": {
    "$class": "org.accordproject.helloworld.MyResponse",
    "output": "Hello Fred Blogs Accord Project",
    "$timestamp": "2021-06-16T12:21:55.749-04:00"
  },
  "state": {
    "$class": "org.accordproject.runtime.State",
    "$identifier": "3fa15a55-d5db-491c-905a-7fcf5eb64d5f"
  },
  "emit": []
}
```

:::note
Remark that if your template directory contains a valid `sample.md` or valid `request.json`, Cicero will automatically detect those so you do not need to pass them using the `--sample` or `--request` options.
:::

---
id: version-0.23.0-accordproject
title: Overview
original_id: accordproject

## What is the Accord Project?

Accord Project is an open source, non-profit initiative aimed at transforming contract management and contract automation by digitizing contracts. It provides an open, standardized format for Smart Legal Contracts.

The Accord Project defines a notion of a legal template with associated computing logic which is expressive, open-source, and portable. Accord Project templates are similar to a clause or contract template in any document format, but they can be read, interpreted, and run by a computer.

## Why is the Accord Project relevant?

The Accord Project provides a universal format for smart legal contracts, and this format is embodied in a variety of open source projects that comprise the Accord Project technology stack. Input from businesses, lawyers and developers is crucial for the Accord Project.

### For Businesses

Contracting is undergoing a digital transformation driven by a need to deliver customer-centric legal and business solutions faster, and at lower cost. This imperative is fueling the adoption of a broad range of new technologies to improve the efficiency of drafting, managing, and executing legal contracting operations; the Accord Project is proud to be part of that movement.

The Accord Project provides a Smart Contract that does not depend on a blockchain, that can integrate text
and data and that can continue operating over its lifespan. The Accord Project smart contract can integrate with your technology platforms and become part of you digital infrastructure.

In addition, contributions from businesses are crucial for the development of the Accord Project. The expertise of stakeholders, such as business professionals and attorneys, is invaluable in improving the functionality and content of the Accord Project's codebase and specifications, to ensure that the templates meet real-world business requirements.

If this interests you, please visit our [Lifecycle and Industry Working Groups](https://www.accordproject.org/liwg) page for more information.

### For Lawyers

The Legal world is changing and Legal Tech is growing into a billion dollar industry. The modern lawyer has to be at home in the digital world. Law Schools now teach courses in coding for lawyers, computational law, blockchain and artificial intelligence. Legal Hackers is a world wide movement uniting lawyers across the world in a shared passion for law and technology. Lawyers need to move beyond the the written word on paper.

The template in an Accord Project Contract is pure legal text that can be drafted

by lawyers and interpreted by courts. An existing contract can easily be
transformed into a template by adding data points between curly braces that
represent the Concerto model and Ergo logic can be added as an integral part of the
contract. The template language is subject to judicial interpretation and the
Concerto model and Ergo logic can be interpreted by a computer creating a bridge
between the two worlds.

As a lawyer, contributing to the Accord Project would be a great opportunity to
learn about smart legal contracts. Through the Accord Project, you can understand
the foundations of open source technologies and learn how to develop smart
agreements.

If your organization wants to become a member of the Accord Project, please [join
our community](https://discord.com/invite/Zm99SKhhtA).

### For Developers

The Accord Project provides a universal format for smart legal contracts, and this
format is embodied in a variety of open source projects that comprise the Accord
Project technology stack. The Accord Project is an open source project and welcomes
contributions from anyone.

The Accord Project is developing tools including a [Visual Studio Code plugin]
(https://marketplace.visualstudio.com/items?itemName=accordproject.cicero-vscode-
extension), [React based web components](https://github.com/accordproject/web-
components) and a command line interface for working with Accord Project Contracts.
You can integrate contracts into existing applications, create new applications or
simply assist lawyers with developing applications with the Ergo language.

There is a welcoming community on Discord that is eager to help. [Join our
Community](https://discord.com/invite/Zm99SKhhtA)

## About this documentation

If you are new to Accord Project, you may want to first read about the notion of
[Smart Legal Contracts](accordproject-slc) and about [Accord Project Templates]
(accordproject-template). We also recommend taking the [Online Tour](accordproject-
tour).

To start using Accord Project templates, follow the [Install
Cicero](https://docs.accordproject.org/docs/next/started-installation.html)
instructions in the _Getting Started_ Section of the documentation.

You can find in-depth guides for the different components of a template in the
Template Guides part of the documentation:
- Learn how to write contract or template text in the [Markdown Text](markup-
preliminaries) Guide
- Learn how to design your data model in the [Concerto Model](model-concerto) Guide
- Learn how to write smart contract logic in the [Ergo Logic](logic-ergo) Guide

Finally, the documentation includes several step by step [Tutorials](tutorial-
templates) and some reference information (for APIs, command-line tools, etc.) can
be found in the [Reference Manual](ref-glossary).

---
id: version-0.23.0-markup-preliminaries

title: Preliminaries
original_id: markup-preliminaries

## Markdown & CommonMark

The text for Accord Project templates is written using markdown. It builds on the [CommonMark](https://commonmark.org) standard so that any CommonMark document is valid text for a template or contract.

As with other markup languages, CommonMark can express the document structure (e.g., headings, paragraphs, lists) and formatting useful for readability (e.g., italics, bold, quotations).

The main reference is the [CommonMark Specification](https://spec.commonmark.org/0.29/) but you can find an overview of CommonMark main features in the [CommonMark](markup-commonmark) Section of this guide.

## Accord Project Extensions

Accord Project uses two extensions to CommonMark: CiceroMark for the contract text, and TemplateMark for the template grammar.

### Lexical Conventions

Accord Project contract or template text is a string of `UTF-8` characters.

:::note
By convention, CiceroMark files have the `.md` extensions, and TemplateMark files have the `.tem.md` extension.
:::

The two sequences of characters `{{` and `}}` are reserved and used for the CiceroMark and TemplateMark extensions to CommonMark. There are three kinds of extensions:
1. Variables (written `{{variableName}}`) which may include an optional formatting (written `{{variableName as "FORMAT"}}`).
2. Formulas (written `{{% expression %}}`).
3. Blocks which may contain additional text or markdown. Blocks come in two flavors:
   1. Blocks corresponding to [markdown inline elements](https://spec.commonmark.org/0.29/#inlines) which may contain only other markdown inline elements (e.g., text, emphasis, links). Those have to be written on a single line as follows:
      ```
      {{#blockName variableName}} ... text or markdown ... {{/blockName}}
      ```
   2. Blocks corresponding to [markdown container elements](https://spec.commonmark.org/0.29/#container-blocks) which may contain arbitrary markdown elements (e.g., paragraphs, lists, headings). Those have to be written with each opening and closing tags on their own line as follows:
      ```
      {{#blockName variableName}}
      ... text or markdown ...
      {{/blockName}}
      ```

### CiceroMark

CiceroMark is used to express the natural language text for legal clauses or
contracts. It uses two specific extensions to CommonMark to facilitate contract
parsing:
1. Clauses within a contract can be identified using a `clause` block:
   ```
   {{#clause clauseName}}
   text of the clause
   {{/clause}}
   ```
2. The result of formulas within a contract or clause can be identified using:
   ```
   {{% result_of_the_formula %}}
   ```

For instance, the following CiceroMark for a loan between `John Smith` and `Jane
Doe` includes a title (`Loan agreement`) followed by some text, followed by a fixed
rate interest clause. The clause contains the terms for the loan and the result of
calculating the monthly payment.
```tem
# Loan agreement

This is a loan agreement between "John Smith" and "Jane Doe", which shall be
entered into
by the parties on January 21, 2021 - 3 PM, except in the event of a force majeure.

{{#clause fixedRate}}
## Fixed rate loan

This is a _fixed interest_ loan to the amount of £100,000.00
at the yearly interest rate of 2.5%
with a loan term of 15,
and monthly payments of {{%£667.00%}}
{{/clause}}
```

More information and examples can be found in the [CiceroMark](markup-ciceromark)
part of this guide.

### TemplateMark

TemplateMark is used to describe families of contracts or clauses with some
variable parts. It is based on CommonMark with several extensions to indicate those
variables parts:
1. _Variables_: e.g., `{{loanAmount}}` indicates the amount for a loan.
2. _Template Blocks_: e.g., `{{#if forceMajeure}}, except in the event of a force
majeure{{/if}}` indicates some optional text in the contract.
3. _Formulas_: e.g., `{{% monthlyPaymentFormula(loanAmount,rate,loanDuration) %}}`
calculates a monthly payment based on the `loanAmount`, `rate`, and `loanDuration`
variables.

For instance, the following TemplateMark for a loan between a `borrower` and a
`lender` includes a title (`Loan agreement`) followed by some text, followed by a
fixed rate interest clause. This template allows for either taking force majeure
into account or not, and calls into a formula to calculate the monthly payment.
```tem
# Loan agreement

This is a loan agreement between {{borrower}} and {{lender}}, which shall be
```

```
entered into
by the parties on {{date as "MMMM DD, YYYY - h A"}}{{#if forceMajeure}}, except in
the event of a force majeure{{/if}}.

{{#clause fixedRate}}
## Fixed rate loan

This is a _fixed interest_ loan to the amount of {{loanAmount as "K0,0.00"}}
at the yearly interest rate of {{rate}}%
with a loan term of {{loanDuration}},
and monthly payments of {{% monthlyPaymentFormula(loanAmount,rate,loanDuration) as
"K0,0.00" %}}
{{/clause}}
```

More information and examples can be found in the [TemplateMark](markup-
templatemark) part of this guide.

## Dingus

You can test your template or contract text using the [TemplateMark Dingus]
(https://templatemark-dingus.netlify.app), an online tool which lets you edit the
markdown and see it rendered as HTML, or as a document object model.

![TemplateMark Dingus](assets/dingus1.png)

You can select whether to parse your text as pure CommonMark (i.e., according to
the CommonMark specification), or with the CiceroMark or TemplateMark extensions.

![TemplateMark Dingus](assets/dingus2.png)

You can also inspect the HTML source, or the document object model (abstract syntax
tree or AST), even see a pdf rendering for your template.

![TemplateMark Dingus](assets/dingus3.png)

For instance, you can open the TemplateMark from the loan example on this page by
clicking [this link](https://templatemark-dingus.netlify.app/#md3=%7B%22source
%22%3A%22%23%20Loan%20agreement%5Cn%5CnThis%20is%20a%20loan%20agreement%20between
%20%7B%7Bborrower%7D%7D%20and%20%7B%7Blender%7D%7D%2C%20which%20shall%20be
%20entered%20into%5Cnby%20the%20parties%20on%20%7B%7Bdate%20as%20%5C%22MMMMM%20DD
%2C%20YYYY%20-%20hhA%5C%22%7D%7D%7B%7B%23if%20forceMajeure%7D%7D%2C%20except%20in
%20the%20event%20of%20a%20force%20majeure%7B%7B%2Fif%7D%7D.%5Cn%5Cn%7B%7B%23clause
%20fixedRate%7D%7D%5Cn%23%23%20Fixed%20rate%20loan%5Cn%5CnThis%20is%20a%20_fixed
%20interest_%20loan%20to%20the%20amount%20of%20%7B%7BloanAmount%20as%20%5C
%22K0%2C0.00%5C%22%7D%7D%5Cnat%20the%20yearly%20interest%20rate%20of%20%7B%7Brate
%7D%7D%25%5Cnwith%20a%20loan%20term%20of%20%7B%7BloanDuration%7D%7D%2C%5Cnand
%20monthly%20payments%20of%20%7B%7B%25%20monthlyPaymentFormula%28loanAmount%2Crate
%2CloanDuration%29%20as%20%5C%22K0%2C0.00%5C%22%20%25%7D%7D%5Cn%7B%7B%2Fclause%7D
%7D%5Cn%22%2C%22defaults%22%3A%7B%22templateMark%22%3Atrue%2C%22ciceroMark
%22%3Afalse%2C%22html%22%3Atrue%2C%22_highlight%22%3Atrue%2C%22_strict%22%3Afalse
%2C%22_view%22%3A%22html%22%7D%7D).

![TemplateMark Dingus](assets/dingus4.png)

---
id: version-0.23.0-model-classes

title: Classes
original_id: model-classes

## Concepts

Concepts are similar to class declarations in most object-oriented languages, in that they may have a super-type and a set of typed properties:

```js
abstract concept Animal {
  o DateTime dob
}

concept Dog extends Animal {
 o String breed
}
```

Concepts can be declared `abstract` if it should not be instantiated (must be subclassed).

## Identity

Concepts may optionally declare an identifying field, using either the `identified by` (explicitly named identity field) or `identified` (`$identifier` system identity field) syntax. Identifying fields must have type `String`.

`Person` below is defined to use the `email` property as its identifying field.

```
concept Person identified by email {
  o String email
  o String firstName
  o String lastName
}
```

While `Product` below will use `$identifier` as its identifying field.

```
concept Product identified {
  o String name
  o Double price
}
```

## Assets

An asset is a class declaration that has a single `String` property which acts as an identifier. You can use the `modelManager.getAssetDeclarations` API to look up all assets.

```js
asset Vehicle identified by vin {
  o String vin
}
```

Assets are implicitly `identified` if you do not specify your own identifing property. The property name is `$identifier`.

Assets are typically used in your models for the long-lived identifiable Things (or nouns) in the model: cars, orders, shipping containers, products, etc.

## Participants

Participants are class declarations that have a single `String` property acting as an identifier. You can use the `modelManager.getParticipantDeclarations` API to look up all participants.

```js
participant Customer identified by email {
  o String email
}
```

Participants are implicitly `identified` if you do not specify your own identifing property. The property name is `$identifier`.

Participants are typically used for the identifiable people or organizations in the model: person, customer, company, business, auditor, etc.

## Transactions

Transactions have an implicit `$timestamp` property with type `DateTime`. You can use the `modelManager.getTransactionDeclarations` API to look up all transactions.

```js
transaction Order {
}
```

Transactions are typically used in models for the identifiable business events or messages that are submitted by Participants to change the state of Assets: cart check out, change of address, identity verification, place order, etc.

## Events

Events are similar to Transactions in that they are also class declarations that have a `$timestamp` property. You can use the `modelManager.getEventDeclarations` API to look up all events.

```js
event LateDelivery {
}
```

Events are typically used in models for the identifiable business events or messages that are emitted by logic to signify that something of interest has occurred.

---
id: version-0.23.0-model-properties
title: Properties
original_id: model-properties

~~~
Class declarations contain properties. Each property has a type which can either be
a type defined in the same namespace, an imported type, or a primitive type.

### Primitive types

Concerto supports the following primitive types:

|Type | Description|
|--- | ---|
|`String` | a UTF8 encoded String.
|`Double` | a double precision 64 bit numeric value.
|`Integer` | a 32 bit signed whole number.
|`Long` | a 64 bit signed whole number.
|`DateTime` | an ISO 8601 & RFC 3339 compatible date or dateTime instance, with UTC
offset.
|`Boolean` | a Boolean value, either true or false.

:::note
Supported date & date-time formats for the `DateTime` primitive type:
- `YYYY-MM-DD`
- `YYYY-MM-DDTHH:mm:ssZ`
- `YYYY-MM-DDTHH:mm:ss±HH:mm`
- `YYYY-MM-DDTHH:mm:ss.SZ`
- `YYYY-MM-DDTHH:mm:ss.SSZ`
- `YYYY-MM-DDTHH:mm:ss.SSSZ`
- `YYYY-MM-DDTHH:mm:ss.S±HH:mm`
- `YYYY-MM-DDTHH:mm:ss.SS±HH:mm`
- `YYYY-MM-DDTHH:mm:ss.SSS±HH:mm`

Milliseconds will be truncated at 3 digits.

We guarantee to support values that are included by the ISO 8601-1:2019, RFC 3339
and HTML Living Standard specifications. However, other formats may be accepted
depending on your platforms, but are subject to change. During validation,
`DateTime` values will be normalized to the `YYYY-MM-DDTHH:mm:ss.SSSZ` format.
:::


### Meta Properties

|Property|Description|
|---|---|
|`[]` | declares that the property is an array|
|`optional` | declares that the property is not required for the instance to be
valid|
| `default` | declares a default value for the property, if no value is specified|
| `range` | declares a valid range for numeric properties|
| `regex` | declares a validation regex for string properties|

`String` fields may include an optional regular expression, which is used to
validate the contents of the field. Careful use of field validators allows Concerto
to perform rich data validation, leading to fewer errors and less boilerplate
application code.

The example below validates that a `String` variable starts with `abc`:

```
  o String myString regex=/abc.*/
~~~

```

`Double`, `Long` or `Integer` fields may include an optional range expression,
which is used to validate the contents of the field. Both the lower and upper bound
are optional, however at least one must be specified. The upper bound must be
greater than or equal to the lower bound.

```
  o Integer intLowerUpper range=[-1,1] // greater than or equal to -1 and less than
or equal to 1
  o Integer intLower range=[-1,] // greater than or equal to -1
  o Integer intUpper range=[,1] // less than or equal to 1

  o Long longLowerUpper range=[-1,1] // greater than or equal to -1 and less than
or equal to 1
  o Long longLower range=[-1,] // greater than or equal to -1
  o Long longUpper range=[,1] // less than or equal to 1

  o Double doubleLowerUpper range=[-1.0,1.0] // greater than or equal to -1 and
less than or equal to 1
  o Double doubleLower range=[-1.0,] // greater than or equal to -1
  o Double doubleUpper range=[,1.0] // less than or equal to 1
```

#### Example

```
asset Vehicle {
  o String model default="F150"
  o String make default="FORD"
  o Integer year default=2016 range=[1990,] optional // model year must be 1990 or
higher
  o String V5cID regex=/^[A-z][A-z][0-9]{7}/
}
```

---
id: version-0.23.0-model-vocabulary
title: Vocabulary
original_id: model-vocabulary

The Vocabulary module for Concerto optionally allows human-readable labels (Terms)
to be associated with model elements. Terms are stored within a locale specific
vocabulary YAML file associated with a Concerto namespace.

For example, a Concerto model that defines an enumeration with the values `RED`,
`GREEN`, `BLUE` can be associated with an English vocabulary with the terms "Red",
"Green", "Blue" and a French Vocabulary with terms "Rouge", "Vert", "Bleue".

The `VocabularyManager` class manages access to a set of Vocabulary files, and
includes logic to retrieve the most appropriate term for a requested locale.

### Example Model

```
namespace org.acme

~~~
enum Color {
    o RED
    o BLUE
    o GREEN
}

asset Vehicle identified by vin {
    o String vin
    o Color color
}

asset Truck extends Vehicle {
    o Double weight
}
```

Example Vocabulary Files

English - en

``` yaml
locale: en
namespace: org.acme
declarations:
  - Color: A color
  - Vehicle: A road vehicle
    properties:
      - vin: Vehicle Identification Number
      - model: Model of the vehicle
  - Truck: A vehicle capable of carrying cargo
    properties:
      - weight: The weight of the truck in KG
```

British English - en-gb

``` yaml
locale: en-gb
namespace: org.acme
declarations:
  - Truck: A lorry (a vehicle capable of carrying cargo)
  - Color: A colour
  - Milkfloat
```

French - fr

```yaml
locale: fr
namespace: org.acme
declarations:
  - Vehicle: Véhicule
    properties:
      - vin: Le numéro d'identification du véhicule (NIV)
```

Simplified Chinese zh-cn

```yaml
~~~

```
locale: zh-cn
namespace: org.acme
declarations:
  - Color: 颜色
    properties:
    - RED: 红色
    - GREEN: 绿色
    - BLUE: 蓝色
  - Vehicle: 车辆
    properties:
      - vin: 车辆识别代号
      - color: 颜色
```

As you can see in the vocabularies above, a vocabulary can supplement or override terms from a base vocabulary, as is the case of the `en-gb` vocabulary which redefines and adds terms specific to British English over the generic English `en` vocabulary.

## API Usage

Use the `VocabularyManager` classs to define new vocabularies, retrieve terms for a locale, or to validate
a vocabulary using a `ModelManager`.

### Adding a Vocabulary

Load the YAML file for the Vocabulary and add it to a `VocabularyManager`:

```

vocabularyManager = new VocabularyManager();
const enVocString = fs.readFileSync('./test/org.acme_en.voc', 'utf-8');
vocabularyManager.addVocabulary(enVocString);
```

### Retrieving a Term

Use the `getTerm` method on the `VocabularyManager` to retrieve a term for
a declaration or property within a namespace:

```

const term = vocabularyManager.getTerm('org.acme', 'en-gb', 'Color');
// term.should.equal('A colour');
```

```

const term = vocabularyManager.getTerm('org.acme', 'en-gb', 'Vehicle', 'vin');
// term.should.equal('Vehicle Identification Number');
```

### Resolve a Term using ModelManager Type Hierarchy

The `resolveTerm` method on the `VocabularyManager` may be used to lookup a term based on the type hierarchy defined by a `ModelManager`. In the example below, the property
`vin` is not defined on the `Truck` declaration but rather on the `Vehicle` super-type.

```
modelManager = new ModelManager();
const model = fs.readFileSync('./test/org.acme.cto', 'utf-8');
modelManager.addModelFile(model);
const term = vocabularyManager.resolveTerm(modelManager, 'org.acme', 'en-gb',
'Truck', 'vin');
// term.should.equal('Vehicle Identification Number');
```

### Validating a Vocabulary Manager

Use the `validate` method on the `VocabularyManager` to detect missing and redudant vocabulary
terms — comparing the terms in the `VocabularyManager` with the declarations in a `ModelManager`.
The return value from `validate` is an object containing information for the missing and additional terms.

> Note that allowing vocabularies to evolve independently of their associated namespace provides definition and translation workflow flexibility.

```
const result = vocabularyManager.validate(modelManager);
// result.missingVocabularies.length.should.equal(1);
// result.missingVocabularies[0].should.equal('org.accordproject');
// result.additionalVocabularies.length.should.equal(1);
// result.additionalVocabularies[0].getNamespace().should.equal('com.example');
//
result.vocabularies['org.acme/en'].additionalTerms.should.have.members(['Vehicle.mo
del']);
//
result.vocabularies['org.acme/en'].missingTerms.should.have.members(['Color.RED',
'Color.BLUE', 'Color.GREEN', 'Vehicle.color']);
// result.vocabularies['org.acme/en-
gb'].additionalTerms.should.have.members(['Milkfloat']);
// result.vocabularies['org.acme/fr'].missingTerms.should.have.members(['Color',
'Vehicle.color', 'Truck']);
// result.vocabularies['org.acme/fr'].additionalTerms.should.have.members([]);
// result.vocabularies['org.acme/zh-
cn'].missingTerms.should.have.members(['Truck']);
// result.vocabularies['org.acme/zh-cn'].additionalTerms.should.have.members([]);
```

Please refer to the [reference API](ref-concerto-api) for the `concerto-vocabulary` module for detailed API guidance.

---
id: version-0.23.0-ref-cicero-api
title: Cicero API
original_id: ref-cicero-api

## Modules

<dl>
<dt><a href="#module_cicero-engine">cicero-engine</a></dt>
<dd><p>Clause Engine</p>
</dd>
<dt><a href="#module_cicero-core">cicero-core</a></dt>

```
<dd><p>Cicero Core - defines the core data types for Cicero.</p>
</dd>
</dl>

## Classes

<dl>
<dt><a href="#Clause">Clause</a></dt>
<dd><p>A Clause is executable business logic, linked to a natural language (legally
enforceable) template.
A Clause must be constructed with a template and then prior to execution the data
for the clause must be set.
Set the data for the clause (an instance of the template model) by either calling
the setData method or by
calling the parse method and passing in natural language text that conforms to the
template grammar.</p>
</dd>
<dt><a href="#Contract">Contract</a></dt>
<dd><p>A Contract is executable business logic, linked to a natural language
(legally enforceable) template.
A Clause must be constructed with a template and then prior to execution the data
for the clause must be set.
Set the data for the clause (an instance of the template model) by either calling
the setData method or by
calling the parse method and passing in natural language text that conforms to the
template grammar.</p>
</dd>
<dt><a href="#Metadata">Metadata</a></dt>
<dd><p>Defines the metadata for a Template, including the name, version, README
markdown.</p>
</dd>
<dt><a href="#Template">Template</a></dt>
<dd><p>A template for a legal clause or contract. A Template has a template model,
request/response transaction types,
a template grammar (natural language for the template) as well as Ergo code for the
business logic of the
template.</p>
</dd>
<dt><a href="#TemplateInstance">TemplateInstance</a></dt>
<dd><p>A TemplateInstance is an instance of a Clause or Contract template. It is
executable business logic, linked to
a natural language (legally enforceable) template.
A TemplateInstance must be constructed with a template and then prior to execution
the data for the clause must be set.
Set the data for the TemplateInstance by either calling the setData method or by
calling the parse method and passing in natural language text that conforms to the
template grammar.</p>
</dd>
<dt><a href="#CompositeArchiveLoader">CompositeArchiveLoader</a></dt>
<dd><p>Manages a set of archive loaders, delegating to the first archive
loader that accepts a URL.</p>
</dd>
</dl>

## Functions

<dl>
<dt><a href="#isPNG">isPNG(buffer)</a> ⇒ <code>Boolean</code></dt>
<dd><p>Checks whether the file is PNG</p>
```

```
</dd>
<dt><a href="#getMimeType">getMimeType(buffer)</a> ⇒ <code>Object</code></dt>
<dd><p>Returns the mime-type of the file</p>
</dd>
</dl>

<a name="module_cicero-engine"></a>

## cicero-engine
Clause Engine


* [cicero-engine](#module_cicero-engine)
    * [.Engine](#module_cicero-engine.Engine)
        * [new Engine()](#new_module_cicero-engine.Engine_new)
        * [.trigger(clause, request, state, [currentTime], [utcOffset])]
(#module_cicero-engine.Engine+trigger) ⇒ <code>Promise</code>
        * [.invoke(clause, clauseName, params, state, [currentTime], [utcOffset])]
(#module_cicero-engine.Engine+invoke) ⇒ <code>Promise</code>
        * [.init(clause, [currentTime], [utcOffset], params)](#module_cicero-
engine.Engine+init) ⇒ <code>Promise</code>
        * [.getErgoEngine()](#module_cicero-engine.Engine+getErgoEngine) ⇒
<code>ErgoEngine</code>

<a name="module_cicero-engine.Engine"></a>

### cicero-engine.Engine
<p>
Engine class. Stateless execution of clauses against a request object, returning a
response to the caller.
</p>

**Kind**: static class of [<code>cicero-engine</code>](#module_cicero-engine)
**Access**: public

* [.Engine](#module_cicero-engine.Engine)
    * [new Engine()](#new_module_cicero-engine.Engine_new)
    * [.trigger(clause, request, state, [currentTime], [utcOffset])]
(#module_cicero-engine.Engine+trigger) ⇒ <code>Promise</code>
    * [.invoke(clause, clauseName, params, state, [currentTime], [utcOffset])]
(#module_cicero-engine.Engine+invoke) ⇒ <code>Promise</code>
    * [.init(clause, [currentTime], [utcOffset], params)](#module_cicero-
engine.Engine+init) ⇒ <code>Promise</code>
    * [.getErgoEngine()](#module_cicero-engine.Engine+getErgoEngine) ⇒
<code>ErgoEngine</code>

<a name="new_module_cicero-engine.Engine_new"></a>

#### new Engine()
Create the Engine.

<a name="module_cicero-engine.Engine+trigger"></a>

#### engine.trigger(clause, request, state, [currentTime], [utcOffset]) ⇒
<code>Promise</code>
Send a request to a clause for execution

**Kind**: instance method of [<code>Engine</code>](#module_cicero-engine.Engine)
**Returns**: <code>Promise</code> - a promise that resolves to a result for the
```

clause

Param	Type	Description
clause	[<code>Clause</code>](#Clause)	the clause
request	<code>object</code>	the request, a JS object that can be deserialized using the Composer serializer.
state	<code>object</code>	the contract state, a JS object that can be deserialized using the Composer serializer.
[currentTime]	<code>string</code>	the definition of 'now', defaults to current time
[utcOffset]	<code>number</code>	UTC Offset for this execution, defaults to local offset

<a name="module_cicero-engine.Engine+invoke"></a>

#### engine.invoke(clause, clauseName, params, state, [currentTime], [utcOffset]) ⇒ <code>Promise</code>
Invoke a specific clause for execution

Kind: instance method of [<code>Engine</code>](#module_cicero-engine.Engine)
Returns: <code>Promise</code> - a promise that resolves to a result for the clause

Param	Type	Description
clause	[<code>Clause</code>](#Clause)	the clause
clauseName	<code>string</code>	the clause name
params	<code>object</code>	the clause parameters, a JS object whose fields that can be deserialized using the Composer serializer.
state	<code>object</code>	the contract state, a JS object that can be deserialized using the Composer serializer.
[currentTime]	<code>string</code>	the definition of 'now', defaults to current time
[utcOffset]	<code>number</code>	UTC Offset for this execution, defaults to local offset

<a name="module_cicero-engine.Engine+init"></a>

#### engine.init(clause, [currentTime], [utcOffset], params) ⇒ <code>Promise</code>
Initialize a clause

Kind: instance method of [<code>Engine</code>](#module_cicero-engine.Engine)
Returns: <code>Promise</code> - a promise that resolves to a result for the clause initialization

Param	Type	Description
clause	[<code>Clause</code>](#Clause)	the clause
[currentTime]	<code>string</code>	the definition of 'now', defaults to current time
[utcOffset]	<code>number</code>	UTC Offset for this execution, defaults to local offset
params	<code>object</code>	the clause parameters, a JS object whose fields that can be deserialized using the Composer serializer.

<a name="module_cicero-engine.Engine+getErgoEngine"></a>

#### engine.getErgoEngine() ⇒ <code>ErgoEngine</code>

Provides access to the underlying Ergo engine.

Kind: instance method of [<code>Engine</code>](#module_cicero-engine.Engine)
Returns: <code>ErgoEngine</code> - the Ergo Engine for this Engine
<a name="module_cicero-core"></a>

## cicero-core
Cicero Core - defines the core data types for Cicero.

<a name="Clause"></a>

## Clause
A Clause is executable business logic, linked to a natural language (legally
enforceable) template.
A Clause must be constructed with a template and then prior to execution the data
for the clause must be set.
Set the data for the clause (an instance of the template model) by either calling
the setData method or by
calling the parse method and passing in natural language text that conforms to the
template grammar.

Kind: global class
Access: public
<a name="Contract"></a>

## Contract
A Contract is executable business logic, linked to a natural language (legally
enforceable) template.
A Clause must be constructed with a template and then prior to execution the data
for the clause must be set.
Set the data for the clause (an instance of the template model) by either calling
the setData method or by
calling the parse method and passing in natural language text that conforms to the
template grammar.

Kind: global class
Access: public
<a name="Metadata"></a>

## Metadata
Defines the metadata for a Template, including the name, version, README markdown.

Kind: global class
Access: public

* [Metadata](#Metadata)
    * [new Metadata(packageJson, readme, samples, request, logo)]
(#new_Metadata_new)
    * _instance_
        * [.getTemplateType()](#Metadata+getTemplateType) ⇒ <code>number</code>
        * [.getLogo()](#Metadata+getLogo) ⇒ <code>Buffer</code>
        * [.getAuthor()](#Metadata+getAuthor) ⇒ <code>*</code>
        * [.getRuntime()](#Metadata+getRuntime) ⇒ <code>string</code>
        * [.getCiceroVersion()](#Metadata+getCiceroVersion) ⇒ <code>string</code>
        * [.satisfiesCiceroVersion(version)](#Metadata+satisfiesCiceroVersion) ⇒
<code>string</code>
        * [.getSamples()](#Metadata+getSamples) ⇒ <code>object</code>
        * [.getRequest()](#Metadata+getRequest) ⇒ <code>object</code>
        * [.getSample(locale)](#Metadata+getSample) ⇒ <code>string</code>

```
        * [.getREADME()](#Metadata+getREADME) ⇒ <code>String</code>
        * [.getPackageJson()](#Metadata+getPackageJson) ⇒ <code>object</code>
        * [.getName()](#Metadata+getName) ⇒ <code>string</code>
        * [.getDisplayName()](#Metadata+getDisplayName) ⇒ <code>string</code>
        * [.getKeywords()](#Metadata+getKeywords) ⇒ <code>Array</code>
        * [.getDescription()](#Metadata+getDescription) ⇒ <code>string</code>
        * [.getVersion()](#Metadata+getVersion) ⇒ <code>string</code>
        * [.getIdentifier()](#Metadata+getIdentifier) ⇒ <code>string</code>
        * [.createTargetMetadata(runtimeName)](#Metadata+createTargetMetadata) ⇒
<code>object</code>
        * [.toJSON()](#Metadata+toJSON) ⇒ <code>object</code>
    * _static_
        * [.checkImage(buffer)](#Metadata.checkImage)
        * [.checkImageDimensions(buffer, mimeType)](#Metadata.checkImageDimensions)

<a name="new_Metadata_new"></a>

### new Metadata(packageJson, readme, samples, request, logo)
Create the Metadata.
<p>
<strong>Note: Only to be called by framework code. Applications should
retrieve instances from [Template](#Template)</strong>
</p>


| Param | Type | Description |
| --- | --- | --- |
| packageJson | <code>object</code> | the JS object for package.json (required) |
| readme | <code>String</code> | the README.md for the template (may be null) |
| samples | <code>object</code> | the sample markdown for the template in different
locales, |
| request | <code>object</code> | the JS object for the sample request |
| logo | <code>Buffer</code> | the bytes data for the image represented as an
object whose keys are the locales and whose values are the sample markdown. For
example:  {      default: 'default sample markdown',      en: 'sample text in
english',      fr: 'exemple de texte français'  } Locale keys (with the exception
of default) conform to the IETF Language Tag specification (BCP 47). THe `default`
key represents sample template text in a non-specified language, stored in a file
called `sample.md`. |

<a name="Metadata+getTemplateType"></a>

### metadata.getTemplateType() ⇒ <code>number</code>
Returns either a 0 (for a contract template), or 1 (for a clause template)

**Kind**: instance method of [<code>Metadata</code>](#Metadata)
**Returns**: <code>number</code> - the template type
<a name="Metadata+getLogo"></a>

### metadata.getLogo() ⇒ <code>Buffer</code>
Returns the logo at the root of the template

**Kind**: instance method of [<code>Metadata</code>](#Metadata)
**Returns**: <code>Buffer</code> - the bytes data of logo
<a name="Metadata+getAuthor"></a>

### metadata.getAuthor() ⇒ <code>\*</code>
Returns the author for this template.
```

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>*</code> - the author information
<a name="Metadata+getRuntime"></a>

### metadata.getRuntime() ⇒ <code>string</code>
Returns the name of the runtime target for this template, or null if this template
has not been compiled for a specific runtime.

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>string</code> - the name of the runtime
<a name="Metadata+getCiceroVersion"></a>

### metadata.getCiceroVersion() ⇒ <code>string</code>
Returns the version of Cicero that this template is compatible with.
i.e. which version of the runtime was this template built for?
The version string conforms to the semver definition

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>string</code> - the semantic version
<a name="Metadata+satisfiesCiceroVersion"></a>

### metadata.satisfiesCiceroVersion(version) ⇒ <code>string</code>
Only returns true if the current cicero version satisfies the target version of
this template

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>string</code> - the semantic version

Param	Type	Description
version	<code>string</code>	the cicero version to check against

<a name="Metadata+getSamples"></a>

### metadata.getSamples() ⇒ <code>object</code>
Returns the samples for this template.

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>object</code> - the sample files for the template
<a name="Metadata+getRequest"></a>

### metadata.getRequest() ⇒ <code>object</code>
Returns the sample request for this template.

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>object</code> - the sample request for the template
<a name="Metadata+getSample"></a>

### metadata.getSample(locale) ⇒ <code>string</code>
Returns the sample for this template in the given locale. This may be null.
If no locale is specified returns the default sample if it has been specified.

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>string</code> - the sample file for the template in the given
locale or null

Param	Type	Default	Description
locale	<code>string</code>	<code>null</code>	the IETF language code for the

language. |

<a name="Metadata+getREADME"></a>

### metadata.getREADME() ⇒ <code>String</code>
Returns the README.md for this template. This may be null if the template does not have a README.md

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>String</code> - the README.md file for the template or null
<a name="Metadata+getPackageJson"></a>

### metadata.getPackageJson() ⇒ <code>object</code>
Returns the package.json for this template.

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>object</code> - the Javascript object for package.json
<a name="Metadata+getName"></a>

### metadata.getName() ⇒ <code>string</code>
Returns the name for this template.

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>string</code> - the name of the template
<a name="Metadata+getDisplayName"></a>

### metadata.getDisplayName() ⇒ <code>string</code>
Returns the display name for this template.

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>string</code> - the display name of the template
<a name="Metadata+getKeywords"></a>

### metadata.getKeywords() ⇒ <code>Array</code>
Returns the keywords for this template.

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>Array</code> - the keywords of the template
<a name="Metadata+getDescription"></a>

### metadata.getDescription() ⇒ <code>string</code>
Returns the description for this template.

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>string</code> - the description of the template
<a name="Metadata+getVersion"></a>

### metadata.getVersion() ⇒ <code>string</code>
Returns the version for this template.

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>string</code> - the description of the template
<a name="Metadata+getIdentifier"></a>

### metadata.getIdentifier() ⇒ <code>string</code>
Returns the identifier for this template, formed from name@version.

Kind: instance method of [<code>Metadata</code>](#Metadata)
Returns: <code>string</code> - the identifier of the template

```
<a name="Metadata+createTargetMetadata"></a>

### metadata.createTargetMetadata(runtimeName) ⇒ <code>object</code>
Return new Metadata for a target runtime

**Kind**: instance method of [<code>Metadata</code>](#Metadata)
**Returns**: <code>object</code> - the new Metadata

| Param | Type | Description |
| --- | --- | --- |
| runtimeName | <code>string</code> | the target runtime name |

<a name="Metadata+toJSON"></a>

### metadata.toJSON() ⇒ <code>object</code>
Return the whole metadata content, for hashing

**Kind**: instance method of [<code>Metadata</code>](#Metadata)
**Returns**: <code>object</code> - the content of the metadata object
<a name="Metadata.checkImage"></a>

### Metadata.checkImage(buffer)
Check the buffer is a png file with the right size

**Kind**: static method of [<code>Metadata</code>](#Metadata)

| Param | Type | Description |
| --- | --- | --- |
| buffer | <code>Buffer</code> | the buffer object |

<a name="Metadata.checkImageDimensions"></a>

### Metadata.checkImageDimensions(buffer, mimeType)
Checks if dimensions for the image are correct.

**Kind**: static method of [<code>Metadata</code>](#Metadata)

| Param | Type | Description |
| --- | --- | --- |
| buffer | <code>Buffer</code> | the buffer object |
| mimeType | <code>string</code> | the mime type of the object |

<a name="Template"></a>

## *Template*
A template for a legal clause or contract. A Template has a template model,
request/response transaction types,
a template grammar (natural language for the template) as well as Ergo code for the
business logic of the
template.

**Kind**: global abstract class
**Access**: public

* *[Template](#Template)*
    * *[new Template(packageJson, readme, samples, request, logo, options,
authorSignature)](#new_Template_new)*
    * _instance_
        * *[.validate(options)](#Template+validate)*
        * *[.getTemplateModel()](#Template+getTemplateModel) ⇒
<code>ClassDeclaration</code>*
        * *[.getIdentifier()](#Template+getIdentifier) ⇒ <code>String</code>*
        * *[.getMetadata()](#Template+getMetadata) ⇒ [<code>Metadata</code>]
(#Metadata)*
        * *[.getName()](#Template+getName) ⇒ <code>String</code>*
        * *[.getDisplayName()](#Template+getDisplayName) ⇒ <code>string</code>*
        * *[.getVersion()](#Template+getVersion) ⇒ <code>String</code>*
        * *[.getDescription()](#Template+getDescription) ⇒ <code>String</code>*
        * *[.getHash()](#Template+getHash) ⇒ <code>string</code>*
        * *[.verifyTemplateSignature()](#Template+verifyTemplateSignature)*
        * *[.signTemplate(p12File, passphrase, timestamp)](#Template+signTemplate)*
        * *[.toArchive([language], [options])](#Template+toArchive) ⇒
<code>Promise.&lt;Buffer&gt;</code>*
        * *[.getParserManager()](#Template+getParserManager) ⇒
<code>ParserManager</code>*
        * *[.getLogicManager()](#Template+getLogicManager) ⇒
<code>LogicManager</code>*
        * *[.getIntrospector()](#Template+getIntrospector) ⇒
<code>Introspector</code>*
        * *[.getFactory()](#Template+getFactory) ⇒ <code>Factory</code>*
        * *[.getSerializer()](#Template+getSerializer) ⇒ <code>Serializer</code>*
        * *[.getRequestTypes()](#Template+getRequestTypes) ⇒ <code>Array</code>*
        * *[.getResponseTypes()](#Template+getResponseTypes) ⇒ <code>Array</code>*
        * *[.getEmitTypes()](#Template+getEmitTypes) ⇒ <code>Array</code>*
        * *[.getStateTypes()](#Template+getStateTypes) ⇒ <code>Array</code>*
        * *[.hasLogic()](#Template+hasLogic) ⇒ <code>boolean</code>*
    * _static_
        * *[.fromDirectory(path, [options])](#Template.fromDirectory) ⇒
[<code>Promise.&lt;Template&gt;</code>](#Template)*
        * *[.fromArchive(buffer, [options])](#Template.fromArchive) ⇒
[<code>Promise.&lt;Template&gt;</code>](#Template)*
        * *[.fromUrl(url, [options])](#Template.fromUrl) ⇒ <code>Promise</code>*
        * *[.instanceOf(classDeclaration, fqt)](#Template.instanceOf) ⇒
<code>boolean</code>*

<a name="new_Template_new"></a>

### *new Template(packageJson, readme, samples, request, logo, options,
authorSignature)*
Create the Template.
Note: Only to be called by framework code. Applications should
retrieve instances from [fromArchive](#Template.fromArchive) or [fromDirectory]
(#Template.fromDirectory).


| Param | Type | Description |
| --- | --- | --- |
| packageJson | <code>object</code> | the JS object for package.json |
| readme | <code>String</code> | the readme in markdown for the template (optional)
|
| samples | <code>object</code> | the sample text for the template in different
locales |
| request | <code>object</code> | the JS object for the sample request |
| logo | <code>Buffer</code> | the bytes data of logo |
| options | <code>Object</code> | e.g., { warnings: true } |
| authorSignature | <code>Object</code> | object containing template hash,
timestamp, author's certificate, signature |
```

<a name="Template+validate"></a>

### *template.validate(options)*
Verifies that the template is well formed.
Compiles the Ergo logic.
Throws an exception with the details of any validation errors.

Kind: instance method of [<code>Template</code>](#Template)

Param	Type	Description
options	<code>Object</code>	e.g., { verify: true }

<a name="Template+getTemplateModel"></a>

### *template.getTemplateModel() ⇒ <code>ClassDeclaration</code>*
Returns the template model for the template

Kind: instance method of [<code>Template</code>](#Template)
Returns: <code>ClassDeclaration</code> - the template model for the template
Throws:

- <code>Error</code> if no template model is found, or multiple template models are found

<a name="Template+getIdentifier"></a>

### *template.getIdentifier() ⇒ <code>String</code>*
Returns the identifier for this template

Kind: instance method of [<code>Template</code>](#Template)
Returns: <code>String</code> - the identifier of this template
<a name="Template+getMetadata"></a>

### *template.getMetadata() ⇒ [<code>Metadata</code>](#Metadata)*
Returns the metadata for this template

Kind: instance method of [<code>Template</code>](#Template)
Returns: [<code>Metadata</code>](#Metadata) - the metadata for this template
<a name="Template+getName"></a>

### *template.getName() ⇒ <code>String</code>*
Returns the name for this template

Kind: instance method of [<code>Template</code>](#Template)
Returns: <code>String</code> - the name of this template
<a name="Template+getDisplayName"></a>

### *template.getDisplayName() ⇒ <code>string</code>*
Returns the display name for this template.

Kind: instance method of [<code>Template</code>](#Template)
Returns: <code>string</code> - the display name of the template
<a name="Template+getVersion"></a>

### *template.getVersion() ⇒ <code>String</code>*
Returns the version for this template

Kind: instance method of [<code>Template</code>](#Template)

Returns: <code>String</code> - the version of this template. Use semver module to parse.
<a name="Template+getDescription"></a>

### *template.getDescription() ⇒ <code>String</code>*
Returns the description for this template

Kind: instance method of [<code>Template</code>](#Template)
Returns: <code>String</code> - the description of this template
<a name="Template+getHash"></a>

### *template.getHash() ⇒ <code>string</code>*
Gets a content based SHA-256 hash for this template. Hash
is based on the metadata for the template plus the contents of
all the models and all the script files.

Kind: instance method of [<code>Template</code>](#Template)
Returns: <code>string</code> - the SHA-256 hash in hex format
<a name="Template+verifyTemplateSignature"></a>

### *template.verifyTemplateSignature()*
verifies the signature stored in the template object using the template hash and timestamp

Kind: instance method of [<code>Template</code>](#Template)
<a name="Template+signTemplate"></a>

### *template.signTemplate(p12File, passphrase, timestamp)*
signs a string made up of template hash and time stamp using private key derived from the keystore

Kind: instance method of [<code>Template</code>](#Template)

Param	Type	Description
p12File	<code>String</code>	encoded string of p12 keystore file
passphrase	<code>String</code>	passphrase for the keystore file
timestamp	<code>Number</code>	timestamp of the moment of signature is done

<a name="Template+toArchive"></a>

### *template.toArchive([language], [options]) ⇒ <code>Promise.&lt;Buffer&gt;</code>*
Persists this template to a Cicero Template Archive (cta) file.

Kind: instance method of [<code>Template</code>](#Template)
Returns: <code>Promise.&lt;Buffer&gt;</code> - the zlib buffer

Param	Type	Description
[language]	<code>string</code>	target language for the archive (should be 'ergo')
[options]	<code>Object</code>	JSZip options and keystore object containing path and passphrase for the keystore

<a name="Template+getParserManager"></a>

### *template.getParserManager() ⇒ <code>ParserManager</code>*
Provides access to the parser manager for this template.

The parser manager can convert template data to and from
natural language text.

Kind: instance method of [<code>Template</code>](#Template)
Returns: <code>ParserManager</code> - the ParserManager for this template
<a name="Template+getLogicManager"></a>

### *template.getLogicManager() ⇒ <code>LogicManager</code>*
Provides access to the template logic for this template.
The template logic encapsulate the code necessary to
execute the clause or contract.

Kind: instance method of [<code>Template</code>](#Template)
Returns: <code>LogicManager</code> - the LogicManager for this template
<a name="Template+getIntrospector"></a>

### *template.getIntrospector() ⇒ <code>Introspector</code>*
Provides access to the Introspector for this template. The Introspector
is used to reflect on the types defined within this template.

Kind: instance method of [<code>Template</code>](#Template)
Returns: <code>Introspector</code> - the Introspector for this template
<a name="Template+getFactory"></a>

### *template.getFactory() ⇒ <code>Factory</code>*
Provides access to the Factory for this template. The Factory
is used to create the types defined in this template.

Kind: instance method of [<code>Template</code>](#Template)
Returns: <code>Factory</code> - the Factory for this template
<a name="Template+getSerializer"></a>

### *template.getSerializer() ⇒ <code>Serializer</code>*
Provides access to the Serializer for this template. The Serializer
is used to serialize instances of the types defined within this template.

Kind: instance method of [<code>Template</code>](#Template)
Returns: <code>Serializer</code> - the Serializer for this template
<a name="Template+getRequestTypes"></a>

### *template.getRequestTypes() ⇒ <code>Array</code>*
Provides a list of the input types that are accepted by this Template. Types use
the fully-qualified form.

Kind: instance method of [<code>Template</code>](#Template)
Returns: <code>Array</code> - a list of the request types
<a name="Template+getResponseTypes"></a>

### *template.getResponseTypes() ⇒ <code>Array</code>*
Provides a list of the response types that are returned by this Template. Types use
the fully-qualified form.

Kind: instance method of [<code>Template</code>](#Template)
Returns: <code>Array</code> - a list of the response types
<a name="Template+getEmitTypes"></a>

### *template.getEmitTypes() ⇒ <code>Array</code>*
Provides a list of the emit types that are emitted by this Template. Types use the
fully-qualified form.

Kind: instance method of [<code>Template</code>](#Template)
Returns: <code>Array</code> - a list of the emit types
<a name="Template+getStateTypes"></a>

### *template.getStateTypes() ⇒ <code>Array</code>*
Provides a list of the state types that are expected by this Template. Types use
the fully-qualified form.

Kind: instance method of [<code>Template</code>](#Template)
Returns: <code>Array</code> - a list of the state types
<a name="Template+hasLogic"></a>

### *template.hasLogic() ⇒ <code>boolean</code>*
Returns true if the template has logic, i.e. has more than one script file.

Kind: instance method of [<code>Template</code>](#Template)
Returns: <code>boolean</code> - true if the template has logic
<a name="Template.fromDirectory"></a>

### *Template.fromDirectory(path, [options]) ⇒
[<code>Promise.&lt;Template&gt;</code>](#Template)*
Builds a Template from the contents of a directory.
The directory must include a package.json in the root (used to specify
the name, version and description of the template).

Kind: static method of [<code>Template</code>](#Template)
Returns: [<code>Promise.&lt;Template&gt;</code>](#Template) - a Promise to the
instantiated template

Param	Type	Default	Description
path	<code>String</code>		to a local directory
[options]	<code>Object</code>	<code></code>	an optional set of options to
configure the instance.

<a name="Template.fromArchive"></a>

### *Template.fromArchive(buffer, [options]) ⇒
[<code>Promise.&lt;Template&gt;</code>](#Template)*
Create a template from an archive.

Kind: static method of [<code>Template</code>](#Template)
Returns: [<code>Promise.&lt;Template&gt;</code>](#Template) - a Promise to the
template

Param	Type	Default	Description
buffer	<code>Buffer</code>		the buffer to a Cicero Template Archive (cta)
file
[options]	<code>Object</code>	<code></code>	an optional set of options to
configure the instance.

<a name="Template.fromUrl"></a>

### *Template.fromUrl(url, [options]) ⇒ <code>Promise</code>*
Create a template from an URL.

Kind: static method of [<code>Template</code>](#Template)

```
**Returns**: <code>Promise</code> - a Promise to the template

| Param | Type | Default | Description |
| --- | --- | --- | --- |
| url | <code>String</code> |  | the URL to a Cicero Template Archive (cta) file |
| [options] | <code>Object</code> | <code></code> | an optional set of options to
configure the instance. |

<a name="Template.instanceOf"></a>

### *Template.instanceOf(classDeclaration, fqt) ⇒ <code>boolean</code>*
Check to see if a ClassDeclaration is an instance of the specified fully qualified
type name.

**Kind**: static method of [<code>Template</code>](#Template)
**Returns**: <code>boolean</code> - True if classDeclaration an instance of the
specified fully
qualified type name, false otherwise.
**Internal**:

| Param | Type | Description |
| --- | --- | --- |
| classDeclaration | <code>ClassDeclaration</code> | The class to test |
| fqt | <code>String</code> | The fully qualified type name. |

<a name="TemplateInstance"></a>

## *TemplateInstance*
A TemplateInstance is an instance of a Clause or Contract template. It is
executable business logic, linked to
a natural language (legally enforceable) template.
A TemplateInstance must be constructed with a template and then prior to execution
the data for the clause must be set.
Set the data for the TemplateInstance by either calling the setData method or by
calling the parse method and passing in natural language text that conforms to the
template grammar.

**Kind**: global abstract class
**Access**: public

* *[TemplateInstance](#TemplateInstance)*
    * *[new TemplateInstance(template)](#new_TemplateInstance_new)*
    * _instance_
        * *[.setData(data)](#TemplateInstance+setData)*
        * *[.getData()](#TemplateInstance+getData) ⇒ <code>object</code>*
        * *[.getEngine()](#TemplateInstance+getEngine) ⇒ <code>object</code>*
        * *[.getDataAsConcertoObject()](#TemplateInstance+getDataAsConcertoObject)
⇒ <code>object</code>*
        * *[.parse(input, [currentTime], [utcOffset], [fileName])]
(#TemplateInstance+parse)*
        * *[.draft([options], [currentTime], [utcOffset])](#TemplateInstance+draft)
⇒ <code>string</code>*
        * *[.formatCiceroMark(ciceroMarkParsed, options, format)]
(#TemplateInstance+formatCiceroMark) ⇒ <code>string</code>*
        * *[.getIdentifier()](#TemplateInstance+getIdentifier) ⇒
<code>String</code>*
        * *[.getTemplate()](#TemplateInstance+getTemplate) ⇒
[<code>Template</code>](#Template)*
        * *[.getLogicManager()](#TemplateInstance+getLogicManager) ⇒
```

```
<code>LogicManager</code>*
        * *[.toJSON()](#TemplateInstance+toJSON) ⇒ <code>object</code>*
    * _static_
        * *[.ciceroFormulaEval(logicManager, clauseId, ergoEngine, name)]
(#TemplateInstance.ciceroFormulaEval) ⇒ <code>\*</code>*
        * *[.rebuildParser(parserManager, logicManager, ergoEngine, templateName,
grammar)](#TemplateInstance.rebuildParser)*

<a name="new_TemplateInstance_new"></a>

### *new TemplateInstance(template)*
Create the Clause and link it to a Template.


| Param | Type | Description |
| --- | --- | --- |
| template | [<code>Template</code>](#Template) | the template for the clause |

<a name="TemplateInstance+setData"></a>

### *templateInstance.setData(data)*
Set the data for the clause

**Kind**: instance method of [<code>TemplateInstance</code>](#TemplateInstance)

| Param | Type | Description |
| --- | --- | --- |
| data | <code>object</code> | the data for the clause, must be an instance of the
template model for the clause's template. This should be a plain JS object and will
be deserialized and validated into the Concerto object before assignment. |

<a name="TemplateInstance+getData"></a>

### *templateInstance.getData() ⇒ <code>object</code>*
Get the data for the clause. This is a plain JS object. To retrieve the Concerto
object call getConcertoData().

**Kind**: instance method of [<code>TemplateInstance</code>](#TemplateInstance)
**Returns**: <code>object</code> - - the data for the clause, or null if it has not
been set
<a name="TemplateInstance+getEngine"></a>

### *templateInstance.getEngine() ⇒ <code>object</code>*
Get the current Ergo engine

**Kind**: instance method of [<code>TemplateInstance</code>](#TemplateInstance)
**Returns**: <code>object</code> - - the data for the clause, or null if it has not
been set
<a name="TemplateInstance+getDataAsConcertoObject"></a>

### *templateInstance.getDataAsConcertoObject() ⇒ <code>object</code>*
Get the data for the clause. This is a Concerto object. To retrieve the
plain JS object suitable for serialization call toJSON() and retrieve the `data`
property.

**Kind**: instance method of [<code>TemplateInstance</code>](#TemplateInstance)
**Returns**: <code>object</code> - - the data for the clause, or null if it has not
been set
<a name="TemplateInstance+parse"></a>
```

```
### *templateInstance.parse(input, [currentTime], [utcOffset], [fileName])*
Set the data for the clause by parsing natural language text.

**Kind**: instance method of [<code>TemplateInstance</code>](#TemplateInstance)

| Param | Type | Description |
| --- | --- | --- |
| input | <code>string</code> | the text for the clause |
| [currentTime] | <code>string</code> | the definition of 'now', defaults to
current time |
| [utcOffset] | <code>number</code> | UTC Offset for this execution, defaults to
local offset |
| [fileName] | <code>string</code> | the fileName for the text (optional) |

<a name="TemplateInstance+draft"></a>

### *templateInstance.draft([options], [currentTime], [utcOffset]) ⇒
<code>string</code>*
Generates the natural language text for a contract or clause clause; combining the
text from the template
and the instance data.

**Kind**: instance method of [<code>TemplateInstance</code>](#TemplateInstance)
**Returns**: <code>string</code> - the natural language text for the contract or
clause; created by combining the structure of
the template with the JSON data for the clause.

| Param | Type | Description |
| --- | --- | --- |
| [options] | <code>\*</code> | text generation options. |
| [currentTime] | <code>string</code> | the definition of 'now', defaults to
current time |
| [utcOffset] | <code>number</code> | UTC Offset for this execution, defaults to
local offset |

<a name="TemplateInstance+formatCiceroMark"></a>

### *templateInstance.formatCiceroMark(ciceroMarkParsed, options, format) ⇒
<code>string</code>*
Format CiceroMark

**Kind**: instance method of [<code>TemplateInstance</code>](#TemplateInstance)
**Returns**: <code>string</code> - the result of parsing and printing back the text

| Param | Type | Description |
| --- | --- | --- |
| ciceroMarkParsed | <code>object</code> | the parsed CiceroMark DOM |
| options | <code>object</code> | parameters to the formatting |
| format | <code>string</code> | to the text generation |

<a name="TemplateInstance+getIdentifier"></a>

### *templateInstance.getIdentifier() ⇒ <code>String</code>*
Returns the identifier for this clause. The identifier is the identifier of
the template plus '-' plus a hash of the data for the clause (if set).

**Kind**: instance method of [<code>TemplateInstance</code>](#TemplateInstance)
**Returns**: <code>String</code> - the identifier of this clause
```

<a name="TemplateInstance+getTemplate"></a>

### *templateInstance.getTemplate() ⇒ [<code>Template</code>](#Template)*
Returns the template for this clause

Kind: instance method of [<code>TemplateInstance</code>](#TemplateInstance)
Returns: [<code>Template</code>](#Template) - the template for this clause
<a name="TemplateInstance+getLogicManager"></a>

### *templateInstance.getLogicManager() ⇒ <code>LogicManager</code>*
Returns the template logic for this clause

Kind: instance method of [<code>TemplateInstance</code>](#TemplateInstance)
Returns: <code>LogicManager</code> - the template for this clause
<a name="TemplateInstance+toJSON"></a>

### *templateInstance.toJSON() ⇒ <code>object</code>*
Returns a JSON representation of the clause

Kind: instance method of [<code>TemplateInstance</code>](#TemplateInstance)
Returns: <code>object</code> - the JS object for serialization
<a name="TemplateInstance.ciceroFormulaEval"></a>

### *TemplateInstance.ciceroFormulaEval(logicManager, clauseId, ergoEngine, name) ⇒
<code>*</code>*
Constructs a function for formula evaluation based for this template instance

Kind: static method of [<code>TemplateInstance</code>](#TemplateInstance)
Returns: <code>*</code> - A function from formula code + input data to result

Param	Type	Description
logicManager	<code>*</code>	the logic manager
clauseId	<code>string</code>	this instance identifier
ergoEngine	<code>*</code>	the evaluation engine
name	<code>string</code>	the name of the formula

<a name="TemplateInstance.rebuildParser"></a>

### *TemplateInstance.rebuildParser(parserManager, logicManager, ergoEngine,
templateName, grammar)*
Utility to rebuild a parser when the grammar changes

Kind: static method of [<code>TemplateInstance</code>](#TemplateInstance)

Param	Type	Description
parserManager	<code>*</code>	the parser manager
logicManager	<code>*</code>	the logic manager
ergoEngine	<code>*</code>	the evaluation engine
templateName	<code>string</code>	this template name
grammar	<code>string</code>	the new grammar

<a name="CompositeArchiveLoader"></a>

## CompositeArchiveLoader
Manages a set of archive loaders, delegating to the first archive
loader that accepts a URL.

```
**Kind**: global class

* [CompositeArchiveLoader](#CompositeArchiveLoader)
    * [new CompositeArchiveLoader()](#new_CompositeArchiveLoader_new)
    * [.addArchiveLoader(archiveLoader)](#CompositeArchiveLoader+addArchiveLoader)
    * [.clearArchiveLoaders()](#CompositeArchiveLoader+clearArchiveLoaders)
    * *[.accepts(url)](#CompositeArchiveLoader+accepts) ⇒ <code>boolean</code>*
    * [.load(url, options)](#CompositeArchiveLoader+load) ⇒ <code>Promise</code>

<a name="new_CompositeArchiveLoader_new"></a>

### new CompositeArchiveLoader()
Create the CompositeArchiveLoader. Used to delegate to a set of ArchiveLoaders.

<a name="CompositeArchiveLoader+addArchiveLoader"></a>

### compositeArchiveLoader.addArchiveLoader(archiveLoader)
Adds a ArchiveLoader implemenetation to the ArchiveLoader

**Kind**: instance method of [<code>CompositeArchiveLoader</code>]
(#CompositeArchiveLoader)

| Param | Type | Description |
| --- | --- | --- |
| archiveLoader | <code>ArchiveLoader</code> | The archive to add to the
CompositeArchiveLoader |

<a name="CompositeArchiveLoader+clearArchiveLoaders"></a>

### compositeArchiveLoader.clearArchiveLoaders()
Remove all registered ArchiveLoaders

**Kind**: instance method of [<code>CompositeArchiveLoader</code>]
(#CompositeArchiveLoader)
<a name="CompositeArchiveLoader+accepts"></a>

### *compositeArchiveLoader.accepts(url) ⇒ <code>boolean</code>*
Returns true if this ArchiveLoader can process the URL

**Kind**: instance abstract method of [<code>CompositeArchiveLoader</code>]
(#CompositeArchiveLoader)
**Returns**: <code>boolean</code> - true if this ArchiveLoader accepts the URL

| Param | Type | Description |
| --- | --- | --- |
| url | <code>string</code> | the URL |

<a name="CompositeArchiveLoader+load"></a>

### compositeArchiveLoader.load(url, options) ⇒ <code>Promise</code>
Load a Archive from a URL and return it

**Kind**: instance method of [<code>CompositeArchiveLoader</code>]
(#CompositeArchiveLoader)
**Returns**: <code>Promise</code> - a promise to the Archive

| Param | Type | Description |
| --- | --- | --- |
| url | <code>string</code> | the url to get |
| options | <code>object</code> | additional options |

<a name="isPNG"></a>

## isPNG(buffer) ⇒ <code>Boolean</code>
Checks whether the file is PNG

**Kind**: global function
**Returns**: <code>Boolean</code> - whether the file in PNG

| Param | Type | Description |
| --- | --- | --- |
| buffer | <code>Buffer</code> | buffer of the file |

<a name="getMimeType"></a>

## getMimeType(buffer) ⇒ <code>Object</code>
Returns the mime-type of the file

**Kind**: global function
**Returns**: <code>Object</code> - the mime-type of the file

| Param | Type | Description |
| --- | --- | --- |
| buffer | <code>Buffer</code> | buffer of the file |


--------------------------------------------------------------------------------
---
id: version-0.23.0-ref-concerto-api
title: Concerto API
original_id: ref-concerto-api
---

## Modules

<dl>
<dt><a href="#module_concerto-core">concerto-core</a></dt>
<dd><p>Concerto core module. Concerto is a framework for defining domain
specific models.</p>
</dd>
<dt><a href="#module_concerto-cto">concerto-cto</a></dt>
<dd><p>Concerto CTO concrete syntax module. Concerto is a framework for defining
domain
specific models.</p>
</dd>
<dt><a href="#module_concerto-metamodel">concerto-metamodel</a></dt>
<dd><p>Concerto metamodel management. Concerto is a framework for defining domain
specific models.</p>
</dd>
<dt><a href="#module_concerto-tools">concerto-tools</a></dt>
<dd><p>Concerto Tools module.</p>
</dd>
<dt><a href="#module_concerto-util">concerto-util</a></dt>
<dd><p>Concerto utility module. Concerto is a framework for defining domain
specific models.</p>
</dd>
<dt><a href="#module_concerto-vocabulary">concerto-vocabulary</a></dt>
<dd><p>Concerto vocabulary module. Concerto is a framework for defining domain
```

```
specific models.</p>
</dd>
</dl>

## Classes

<dl>
<dt><a href="#AbstractPlugin">AbstractPlugin</a></dt>
<dd><p>Simple plug-in class for code-generation. This lists functions that can be
passed to extend the default code-generation behavior.</p>
</dd>
<dt><a href="#EmptyPlugin">EmptyPlugin</a></dt>
<dd><p>Simple plug-in class for code-generation. This lists functions that can be
passed to extend the default code-generation behavior.</p>
</dd>
</dl>

## Constants

<dl>
<dt><a href="#rootModelAst">rootModelAst</a> : <code>unknown</code></dt>
<dd></dd>
<dt><a href="#metaModelAst">metaModelAst</a> : <code>unknown</code></dt>
<dd><p>The metamodel itself, as an AST.</p>
</dd>
<dt><a href="#metaModelCto">metaModelCto</a></dt>
<dd><p>The metamodel itself, as a CTO string</p>
</dd>
<dt><a href="#levels">levels</a> : <code>Object</code></dt>
<dd><p>Default levels for the npm configuration.</p>
</dd>
<dt><a href="#colorMap">colorMap</a> : <code>Object</code></dt>
<dd><p>Default levels for the npm configuration.</p>
</dd>
</dl>

## Functions

<dl>
<dt><a href="#setCurrentTime">setCurrentTime([currentTime], [utcOffset])</a> ⇒
<code>object</code></dt>
<dd><p>Ensures there is a proper current time</p>
</dd>
<dt><a href="#newMetaModelManager">newMetaModelManager()</a> ⇒ <code>*</code></dt>
<dd><p>Create a metamodel manager (for validation against the metamodel)</p>
</dd>
<dt><a href="#validateMetaModel">validateMetaModel(input)</a> ⇒
<code>object</code></dt>
<dd><p>Validate metamodel instance against the metamodel</p>
</dd>
<dt><a href="#modelManagerFromMetaModel">modelManagerFromMetaModel(metaModel,
[validate])</a> ⇒ <code>object</code></dt>
<dd><p>Import metamodel to a model manager</p>
</dd>
<dt><a
href="#randomNumberInRangeWithPrecision">randomNumberInRangeWithPrecision(userMin,
userMax, precision, systemMin, systemMax)</a> ⇒ <code>number</code></dt>
<dd><p>Generate a random number within a given range with
a prescribed precision and inside a global range</p>
```

```
</dd>
<dt><a href="#updateModels">updateModels(models, newModel)</a> ⇒
<code>*</code></dt>
<dd><p>Update models with a new model</p>
</dd>
<dt><a href="#resolveExternal">resolveExternal(models, [options],
[fileDownloader])</a> ⇒ <code>Promise</code></dt>
<dd><p>Downloads all ModelFiles that are external dependencies and adds or
updates them in this ModelManager.</p>
</dd>
<dt><a href="#parse">parse(cto, [fileName])</a> ⇒ <code>object</code></dt>
<dd><p>Create decorator argument string from a metamodel</p>
</dd>
<dt><a href="#parseModels">parseModels(files)</a> ⇒ <code>*</code></dt>
<dd><p>Parses an array of model files</p>
</dd>
<dt><a href="#decoratorArgFromMetaModel">decoratorArgFromMetaModel(mm)</a> ⇒
<code>string</code></dt>
<dd><p>Create decorator argument string from a metamodel</p>
</dd>
<dt><a href="#decoratorFromMetaModel">decoratorFromMetaModel(mm)</a> ⇒
<code>string</code></dt>
<dd><p>Create decorator string from a metamodel</p>
</dd>
<dt><a href="#decoratorsFromMetaModel">decoratorsFromMetaModel(mm, prefix)</a> ⇒
<code>string</code></dt>
<dd><p>Create decorators string from a metamodel</p>
</dd>
<dt><a href="#propertyFromMetaModel">propertyFromMetaModel(mm)</a> ⇒
<code>string</code></dt>
<dd><p>Create a property string from a metamodel</p>
</dd>
<dt><a href="#declFromMetaModel">declFromMetaModel(mm)</a> ⇒
<code>string</code></dt>
<dd><p>Create a declaration string from a metamodel</p>
</dd>
<dt><a href="#toCTO">toCTO(metaModel)</a> ⇒ <code>string</code></dt>
<dd><p>Create a model string from a metamodel</p>
</dd>
<dt><a href="#findNamespace">findNamespace(priorModels, namespace)</a> ⇒
<code>*</code></dt>
<dd><p>Find the model for a given namespace</p>
</dd>
<dt><a href="#findDeclaration">findDeclaration(thisModel, name)</a> ⇒
<code>*</code></dt>
<dd><p>Find a declaration for a given name in a model</p>
</dd>
<dt><a href="#createNameTable">createNameTable(priorModels, metaModel)</a> ⇒
<code>object</code></dt>
<dd><p>Create a name resolution table</p>
</dd>
<dt><a href="#resolveName">resolveName(name, table)</a> ⇒ <code>string</code></dt>
<dd><p>Resolve a name using the name table</p>
</dd>
<dt><a href="#resolveTypeNames">resolveTypeNames(metaModel, table)</a> ⇒
<code>object</code></dt>
<dd><p>Name resolution for metamodel</p>
</dd>
<dt><a href="#resolveLocalNames">resolveLocalNames(priorModels, metaModel)</a> ⇒
```

```
<code>object</code></dt>
<dd><p>Resolve the namespace for names in the metamodel</p>
</dd>
<dt><a href="#resolveLocalNamesForAll">resolveLocalNamesForAll(allModels)</a> ⇒
<code>object</code></dt>
<dd><p>Resolve the namespace for names in the metamodel</p>
</dd>
<dt><a href="#inferModelFile">inferModelFile(defaultNamespace, defaultType,
schema)</a> ⇒ <code>string</code></dt>
<dd><p>Infers a Concerto model from a JSON Schema.</p>
</dd>
<dt><a href="#capitalizeFirstLetter">capitalizeFirstLetter(string)</a> ⇒
<code>string</code></dt>
<dd><p>Capitalize the first letter of a string</p>
</dd>
<dt><a href="#hashCode">hashCode(value)</a> ⇒ <code>number</code></dt>
<dd><p>Computes an integer hashcode value for a string</p>
</dd>
<dt><a href="#isObject">isObject(val)</a> ⇒ <code>Boolean</code></dt>
<dd><p>Returns true if val is an object</p>
</dd>
<dt><a href="#isBoolean">isBoolean(val)</a> ⇒ <code>Boolean</code></dt>
<dd><p>Returns true if val is a boolean</p>
</dd>
<dt><a href="#isNull">isNull(val)</a> ⇒ <code>Boolean</code></dt>
<dd><p>Returns true if val is null</p>
</dd>
<dt><a href="#isArray">isArray(val)</a> ⇒ <code>Boolean</code></dt>
<dd><p>Returns true if val is an array</p>
</dd>
<dt><a href="#isString">isString(val)</a> ⇒ <code>Boolean</code></dt>
<dd><p>Returns true if val is a string</p>
</dd>
<dt><a href="#isDateTime">isDateTime(val)</a> ⇒ <code>Boolean</code></dt>
<dd><p>Returns true if val is a date time</p>
</dd>
<dt><a href="#isInteger">isInteger(val)</a> ⇒ <code>Boolean</code></dt>
<dd><p>Returns true if val is an integer</p>
</dd>
<dt><a href="#isDouble">isDouble(val)</a> ⇒ <code>Boolean</code></dt>
<dd><p>Returns true if val is an integer</p>
</dd>
<dt><a href="#getType">getType(input)</a> ⇒ <code>string</code></dt>
<dd><p>Get the primitive Concerto type for an input</p>
</dd>
<dt><a href="#handleArray">handleArray(typeName, context, input)</a> ⇒
<code>object</code></dt>
<dd><p>Handles an array</p>
</dd>
<dt><a href="#handleType">handleType(name, context, input)</a> ⇒
<code>object</code></dt>
<dd><p>Handles an input type</p>
</dd>
<dt><a href="#removeDuplicateTypes">removeDuplicateTypes(context)</a></dt>
<dd><p>Detect duplicate types and remove them</p>
</dd>
<dt><a href="#inferModel">inferModel(namespace, rootTypeName, input)</a> ⇒
<code>string</code></dt>
<dd><p>Infers a Concerto model from a JSON instance.</p>
```

```
</dd>
<dt><a href="#labelToSentence">labelToSentence(labelName)</a> ⇒
<code>string</code></dt>
<dd><p>Inserts correct spacing and capitalization to a camelCase label</p>
</dd>
<dt><a href="#sentenceToLabel">sentenceToLabel(sentence)</a> ⇒
<code>string</code></dt>
<dd><p>Create a camelCase label from a sentence</p>
</dd>
<dt><a href="#writeModelsToFileSystem">writeModelsToFileSystem(files, path,
options)</a></dt>
<dd><p>Writes a set of model files to disk</p>
</dd>
<dt><a href="#camelCaseToSentence">camelCaseToSentence(text)</a> ⇒
<code>string</code></dt>
<dd><p>Converts a camel case string to a sentence</p>
</dd>
</dl>

<a name="module_concerto-core"></a>

## concerto-core
Concerto core module. Concerto is a framework for defining domain
specific models.


* [concerto-core](#module_concerto-core)
    * _static_
        * [.AstModelManager](#module_concerto-core.AstModelManager)
            * [new AstModelManager([options])](#new_module_concerto-
core.AstModelManager_new)
        * [.BaseModelManager](#module_concerto-core.BaseModelManager)
            * [new BaseModelManager([options], [processFile])]
(#new_module_concerto-core.BaseModelManager_new)
            * [.isModelManager()](#module_concerto-
core.BaseModelManager+isModelManager) ⇒ <code>boolean</code>
            * [.accept(visitor, parameters)](#module_concerto-
core.BaseModelManager+accept) ⇒ <code>Object</code>
            * [.validateModelFile(modelFile, [fileName])](#module_concerto-
core.BaseModelManager+validateModelFile)
            * [.addModelFile(modelFile, [cto], [fileName], [disableValidation])]
(#module_concerto-core.BaseModelManager+addModelFile) ⇒ <code>Object</code>
            * [.addModel(modelInput, [cto], [fileName], [disableValidation])]
(#module_concerto-core.BaseModelManager+addModel) ⇒ <code>Object</code>
            * [.updateModelFile(modelFile, [fileName], [disableValidation])]
(#module_concerto-core.BaseModelManager+updateModelFile) ⇒ <code>Object</code>
            * [.deleteModelFile(namespace)](#module_concerto-
core.BaseModelManager+deleteModelFile)
            * [.addModelFiles(modelFiles, [fileNames], [disableValidation])]
(#module_concerto-core.BaseModelManager+addModelFiles) ⇒
<code>Array.&lt;Object&gt;</code>
            * [.validateModelFiles()](#module_concerto-
core.BaseModelManager+validateModelFiles)
            * [.updateExternalModels([options], [fileDownloader])]
(#module_concerto-core.BaseModelManager+updateExternalModels) ⇒
<code>Promise</code>
            * [.writeModelsToFileSystem(path, [options])](#module_concerto-
core.BaseModelManager+writeModelsToFileSystem)
            * [.getModels([options])](#module_concerto-
core.BaseModelManager+getModels) ⇒ <code>Array.&lt;{name:string,
content:string}&gt;</code>
            * [.clearModelFiles()](#module_concerto-
core.BaseModelManager+clearModelFiles)
            * [.getModelFile(namespace)](#module_concerto-
core.BaseModelManager+getModelFile) ⇒ <code>ModelFile</code>
            * [.getNamespaces()](#module_concerto-
core.BaseModelManager+getNamespaces) ⇒ <code>Array.&lt;string&gt;</code>
            * [.getType(qualifiedName)](#module_concerto-
core.BaseModelManager+getType) ⇒ <code>ClassDeclaration</code>
            * [.getAssetDeclarations()](#module_concerto-
core.BaseModelManager+getAssetDeclarations) ⇒
<code>Array.&lt;AssetDeclaration&gt;</code>
            * [.getTransactionDeclarations()](#module_concerto-
core.BaseModelManager+getTransactionDeclarations) ⇒
<code>Array.&lt;TransactionDeclaration&gt;</code>
            * [.getEventDeclarations()](#module_concerto-
core.BaseModelManager+getEventDeclarations) ⇒
<code>Array.&lt;EventDeclaration&gt;</code>
            * [.getParticipantDeclarations()](#module_concerto-
core.BaseModelManager+getParticipantDeclarations) ⇒
<code>Array.&lt;ParticipantDeclaration&gt;</code>
            * [.getEnumDeclarations()](#module_concerto-
core.BaseModelManager+getEnumDeclarations) ⇒
<code>Array.&lt;EnumDeclaration&gt;</code>
            * [.getConceptDeclarations()](#module_concerto-
core.BaseModelManager+getConceptDeclarations) ⇒
<code>Array.&lt;ConceptDeclaration&gt;</code>
            * [.getFactory()](#module_concerto-core.BaseModelManager+getFactory) ⇒
<code>Factory</code>
            * [.getSerializer()](#module_concerto-
core.BaseModelManager+getSerializer) ⇒ <code>Serializer</code>
            * [.getDecoratorFactories()](#module_concerto-
core.BaseModelManager+getDecoratorFactories) ⇒
<code>Array.&lt;DecoratorFactory&gt;</code>
            * [.addDecoratorFactory(factory)](#module_concerto-
core.BaseModelManager+addDecoratorFactory)
            * [.derivesFrom(fqt1, fqt2)](#module_concerto-
core.BaseModelManager+derivesFrom) ⇒ <code>boolean</code>
            * [.resolveMetaModel(metaModel)](#module_concerto-
core.BaseModelManager+resolveMetaModel) ⇒ <code>object</code>
            * [.fromAst(ast)](#module_concerto-core.BaseModelManager+fromAst)
            * [.getAst([resolve])](#module_concerto-core.BaseModelManager+getAst) ⇒
<code>\*</code>
        * [.Concerto](#module_concerto-core.Concerto)
            * [new Concerto(modelManager)](#new_module_concerto-core.Concerto_new)
            * [.validate(obj, [options])](#module_concerto-core.Concerto+validate)
            * [.getModelManager()](#module_concerto-core.Concerto+getModelManager)
⇒ <code>\*</code>
            * [.isObject(obj)](#module_concerto-core.Concerto+isObject) ⇒
<code>boolean</code>
            * [.getTypeDeclaration(obj)](#module_concerto-
core.Concerto+getTypeDeclaration) ⇒ <code>\*</code>
            * [.getIdentifier(obj)](#module_concerto-core.Concerto+getIdentifier) ⇒
<code>string</code>
            * [.isIdentifiable(obj)](#module_concerto-core.Concerto+isIdentifiable)
⇒ <code>boolean</code>
            * [.isRelationship(obj)](#module_concerto-core.Concerto+isRelationship)
⇒ <code>boolean</code>
```

```
            * [.setIdentifier(obj, id)](#module_concerto-
core.Concerto+setIdentifier) ⇒ <code>\*</code>
            * [.getFullyQualifiedIdentifier(obj)](#module_concerto-
core.Concerto+getFullyQualifiedIdentifier) ⇒ <code>string</code>
            * [.toURI(obj)](#module_concerto-core.Concerto+toURI) ⇒
<code>string</code>
            * [.fromURI(uri)](#module_concerto-core.Concerto+fromURI) ⇒
<code>\*</code>
            * [.getType(obj)](#module_concerto-core.Concerto+getType) ⇒
<code>string</code>
            * [.getNamespace(obj)](#module_concerto-core.Concerto+getNamespace) ⇒
<code>string</code>
        * [.DecoratorManager](#module_concerto-core.DecoratorManager)
            * [.decorateModels(modelManager, decoratorCommandSet)]
(#module_concerto-core.DecoratorManager.decorateModels) ⇒ <code>ModelManager</code>
            * [.falsyOrEqual(test, value)](#module_concerto-
core.DecoratorManager.falsyOrEqual) ⇒ <code>Boolean</code>
            * [.applyDecorator(decorated, type, newDecorator)](#module_concerto-
core.DecoratorManager.applyDecorator)
            * [.executeCommand(namespace, declaration, command)](#module_concerto-
core.DecoratorManager.executeCommand)
        * [.Factory](#module_concerto-core.Factory)
            * [new Factory(modelManager)](#new_module_concerto-core.Factory_new)
            * _instance_
                * [.newResource(ns, type, [id], [options])](#module_concerto-
core.Factory+newResource) ⇒ <code>Resource</code>
                * [.newConcept(ns, type, [id], [options])](#module_concerto-
core.Factory+newConcept) ⇒ <code>Resource</code>
                * [.newRelationship(ns, type, id)](#module_concerto-
core.Factory+newRelationship) ⇒ <code>Relationship</code>
                * [.newTransaction(ns, type, [id], [options])](#module_concerto-
core.Factory+newTransaction) ⇒ <code>Resource</code>
                * [.newEvent(ns, type, [id], [options])](#module_concerto-
core.Factory+newEvent) ⇒ <code>Resource</code>
            * _static_
                * [.newId()](#module_concerto-core.Factory.newId) ⇒
<code>string</code>
        * [.AssetDeclaration](#module_concerto-core.AssetDeclaration) ⇐
<code>ClassDeclaration</code>
            * [new AssetDeclaration(modelFile, ast)](#new_module_concerto-
core.AssetDeclaration_new)
            * [.declarationKind()](#module_concerto-
core.AssetDeclaration+declarationKind) ⇒ <code>string</code>
        * *[.ClassDeclaration](#module_concerto-core.ClassDeclaration)*
            * *[new ClassDeclaration(modelFile, ast)](#new_module_concerto-
core.ClassDeclaration_new)*
            * *[._resolveSuperType()](#module_concerto-
core.ClassDeclaration+_resolveSuperType) ⇒ <code>ClassDeclaration</code>*
            * *[.validate()](#module_concerto-core.ClassDeclaration+validate)*
            * *[.isAbstract()](#module_concerto-core.ClassDeclaration+isAbstract) ⇒
<code>boolean</code>*
            * *[.getName()](#module_concerto-core.ClassDeclaration+getName) ⇒
<code>string</code>*
            * *[.getNamespace()](#module_concerto-
core.ClassDeclaration+getNamespace) ⇒ <code>string</code>*
            * *[.getFullyQualifiedName()](#module_concerto-
core.ClassDeclaration+getFullyQualifiedName) ⇒ <code>string</code>*
            * *[.isIdentified()](#module_concerto-
core.ClassDeclaration+isIdentified) ⇒ <code>Boolean</code>*
```

```
            * *[.isSystemIdentified()](#module_concerto-
core.ClassDeclaration+isSystemIdentified) ⇒ <code>Boolean</code>*
            * *[.isExplicitlyIdentified()](#module_concerto-
core.ClassDeclaration+isExplicitlyIdentified) ⇒ <code>Boolean</code>*
            * *[.getIdentifierFieldName()](#module_concerto-
core.ClassDeclaration+getIdentifierFieldName) ⇒ <code>string</code>*
            * *[.getOwnProperty(name)](#module_concerto-
core.ClassDeclaration+getOwnProperty) ⇒ <code>Property</code>*
            * *[.getOwnProperties()](#module_concerto-
core.ClassDeclaration+getOwnProperties) ⇒ <code>Array.&lt;Property&gt;</code>*
            * *[.getSuperType()](#module_concerto-
core.ClassDeclaration+getSuperType) ⇒ <code>string</code>*
            * *[.getSuperTypeDeclaration()](#module_concerto-
core.ClassDeclaration+getSuperTypeDeclaration) ⇒ <code>ClassDeclaration</code>*
            * *[.getAssignableClassDeclarations()](#module_concerto-
core.ClassDeclaration+getAssignableClassDeclarations) ⇒
<code>Array.&lt;ClassDeclaration&gt;</code>*
            * *[.getAllSuperTypeDeclarations()](#module_concerto-
core.ClassDeclaration+getAllSuperTypeDeclarations) ⇒
<code>Array.&lt;ClassDeclaration&gt;</code>*
            * *[.getProperty(name)](#module_concerto-
core.ClassDeclaration+getProperty) ⇒ <code>Property</code>*
            * *[.getProperties()](#module_concerto-
core.ClassDeclaration+getProperties) ⇒ <code>Array.&lt;Property&gt;</code>*
            * *[.getNestedProperty(propertyPath)](#module_concerto-
core.ClassDeclaration+getNestedProperty) ⇒ <code>Property</code>*
            * *[.toString()](#module_concerto-core.ClassDeclaration+toString) ⇒
<code>String</code>*
            * *[.isAsset()](#module_concerto-core.ClassDeclaration+isAsset) ⇒
<code>boolean</code>*
            * *[.isParticipant()](#module_concerto-
core.ClassDeclaration+isParticipant) ⇒ <code>boolean</code>*
            * *[.isTransaction()](#module_concerto-
core.ClassDeclaration+isTransaction) ⇒ <code>boolean</code>*
            * *[.isEvent()](#module_concerto-core.ClassDeclaration+isEvent) ⇒
<code>boolean</code>*
            * *[.isConcept()](#module_concerto-core.ClassDeclaration+isConcept) ⇒
<code>boolean</code>*
            * *[.isEnum()](#module_concerto-core.ClassDeclaration+isEnum) ⇒
<code>boolean</code>*
            * *[.isClassDeclaration()](#module_concerto-
core.ClassDeclaration+isClassDeclaration) ⇒ <code>boolean</code>*
        * [.ConceptDeclaration](#module_concerto-core.ConceptDeclaration) ⇐
<code>ClassDeclaration</code>
            * [new ConceptDeclaration(modelFile, ast)](#new_module_concerto-
core.ConceptDeclaration_new)
            * [.declarationKind()](#module_concerto-
core.ConceptDeclaration+declarationKind) ⇒ <code>string</code>
        * [.Decorator](#module_concerto-core.Decorator)
            * [new Decorator(parent, ast)](#new_module_concerto-core.Decorator_new)
            * [.getParent()](#module_concerto-core.Decorator+getParent) ⇒
<code>ClassDeclaration</code> \| <code>Property</code>
            * [.getName()](#module_concerto-core.Decorator+getName) ⇒
<code>string</code>
            * [.getArguments()](#module_concerto-core.Decorator+getArguments) ⇒
<code>Array.&lt;object&gt;</code>
        * [.DecoratorFactory](#module_concerto-core.DecoratorFactory)
            * *[.newDecorator(parent, ast)](#module_concerto-
core.DecoratorFactory+newDecorator) ⇒ <code>Decorator</code>*
```

```
        * [.EnumDeclaration](#module_concerto-core.EnumDeclaration) ⇐
<code>ClassDeclaration</code>
            * [new EnumDeclaration(modelFile, ast)](#new_module_concerto-
core.EnumDeclaration_new)
            * [.toString()](#module_concerto-core.EnumDeclaration+toString) ⇒
<code>String</code>
            * [.declarationKind()](#module_concerto-
core.EnumDeclaration+declarationKind) ⇒ <code>string</code>
        * [.EnumValueDeclaration](#module_concerto-core.EnumValueDeclaration) ⇐
<code>Property</code>
            * [new EnumValueDeclaration(parent, ast)](#new_module_concerto-
core.EnumValueDeclaration_new)
            * [.isEnumValue()](#module_concerto-
core.EnumValueDeclaration+isEnumValue) ⇒ <code>boolean</code>
        * [.EventDeclaration](#module_concerto-core.EventDeclaration) ⇐
<code>ClassDeclaration</code>
            * [new EventDeclaration(modelFile, ast)](#new_module_concerto-
core.EventDeclaration_new)
            * [.declarationKind()](#module_concerto-
core.EventDeclaration+declarationKind) ⇒ <code>string</code>
        * *[.IdentifiedDeclaration](#module_concerto-core.IdentifiedDeclaration) ⇐
<code>ClassDeclaration</code>*
            * *[new IdentifiedDeclaration(modelFile, ast)](#new_module_concerto-
core.IdentifiedDeclaration_new)*
        * [.IllegalModelException](#module_concerto-core.IllegalModelException) ⇐
<code>BaseFileException</code>
            * [new IllegalModelException(message, [modelFile], [fileLocation],
[component])](#new_module_concerto-core.IllegalModelException_new)
        * [.Introspector](#module_concerto-core.Introspector)
            * [new Introspector(modelManager)](#new_module_concerto-
core.Introspector_new)
            * [.getClassDeclarations()](#module_concerto-
core.Introspector+getClassDeclarations) ⇒
<code>Array.&lt;ClassDeclaration&gt;</code>
            * [.getClassDeclaration(fullyQualifiedTypeName)](#module_concerto-
core.Introspector+getClassDeclaration) ⇒ <code>ClassDeclaration</code>
        * [.ModelFile](#module_concerto-core.ModelFile)
            * [new ModelFile(modelManager, ast, [definitions], [fileName])]
(#new_module_concerto-core.ModelFile_new)
            * [.isModelFile()](#module_concerto-core.ModelFile+isModelFile) ⇒
<code>boolean</code>
            * [.isSystemModelFile()](#module_concerto-
core.ModelFile+isSystemModelFile) ⇒ <code>Boolean</code>
            * [.isExternal()](#module_concerto-core.ModelFile+isExternal) ⇒
<code>boolean</code>
            * [.getModelManager()](#module_concerto-core.ModelFile+getModelManager)
⇒ <code>ModelManager</code>
            * [.getImports()](#module_concerto-core.ModelFile+getImports) ⇒
<code>Array.&lt;string&gt;</code>
            * [.isDefined(type)](#module_concerto-core.ModelFile+isDefined) ⇒
<code>boolean</code>
            * [.getLocalType(type)](#module_concerto-core.ModelFile+getLocalType) ⇒
<code>ClassDeclaration</code>
            * [.getAssetDeclaration(name)](#module_concerto-
core.ModelFile+getAssetDeclaration) ⇒ <code>AssetDeclaration</code>
            * [.getTransactionDeclaration(name)](#module_concerto-
core.ModelFile+getTransactionDeclaration) ⇒ <code>TransactionDeclaration</code>
            * [.getEventDeclaration(name)](#module_concerto-
core.ModelFile+getEventDeclaration) ⇒ <code>EventDeclaration</code>
```

```
            * [.getParticipantDeclaration(name)](#module_concerto-
core.ModelFile+getParticipantDeclaration) ⇒ <code>ParticipantDeclaration</code>
            * [.getNamespace()](#module_concerto-core.ModelFile+getNamespace) ⇒
<code>string</code>
            * [.getName()](#module_concerto-core.ModelFile+getName) ⇒
<code>string</code>
            * [.getAssetDeclarations()](#module_concerto-
core.ModelFile+getAssetDeclarations) ⇒ <code>Array.&lt;AssetDeclaration&gt;</code>
            * [.getTransactionDeclarations()](#module_concerto-
core.ModelFile+getTransactionDeclarations) ⇒
<code>Array.&lt;TransactionDeclaration&gt;</code>
            * [.getEventDeclarations()](#module_concerto-
core.ModelFile+getEventDeclarations) ⇒ <code>Array.&lt;EventDeclaration&gt;</code>
            * [.getParticipantDeclarations()](#module_concerto-
core.ModelFile+getParticipantDeclarations) ⇒
<code>Array.&lt;ParticipantDeclaration&gt;</code>
            * [.getConceptDeclarations()](#module_concerto-
core.ModelFile+getConceptDeclarations) ⇒
<code>Array.&lt;ConceptDeclaration&gt;</code>
            * [.getEnumDeclarations()](#module_concerto-
core.ModelFile+getEnumDeclarations) ⇒ <code>Array.&lt;EnumDeclaration&gt;</code>
            * [.getDeclarations(type)](#module_concerto-
core.ModelFile+getDeclarations) ⇒ <code>Array.&lt;ClassDeclaration&gt;</code>
            * [.getAllDeclarations()](#module_concerto-
core.ModelFile+getAllDeclarations) ⇒ <code>Array.&lt;ClassDeclaration&gt;</code>
            * [.getDefinitions()](#module_concerto-core.ModelFile+getDefinitions) ⇒
<code>string</code>
            * [.getAst()](#module_concerto-core.ModelFile+getAst) ⇒
<code>object</code>
            * [.getConcertoVersion()](#module_concerto-
core.ModelFile+getConcertoVersion) ⇒ <code>string</code>
            * [.isCompatibleVersion()](#module_concerto-
core.ModelFile+isCompatibleVersion)
        * [.ParticipantDeclaration](#module_concerto-core.ParticipantDeclaration) ⇐
<code>ClassDeclaration</code>
            * [new ParticipantDeclaration(modelFile, ast)](#new_module_concerto-
core.ParticipantDeclaration_new)
            * [.declarationKind()](#module_concerto-
core.ParticipantDeclaration+declarationKind) ⇒ <code>string</code>
        * [.Property](#module_concerto-core.Property)
            * [new Property(parent, ast)](#new_module_concerto-core.Property_new)
            * [.getParent()](#module_concerto-core.Property+getParent) ⇒
<code>ClassDeclaration</code>
            * [.validate(classDecl)](#module_concerto-core.Property+validate)
            * [.getName()](#module_concerto-core.Property+getName) ⇒
<code>string</code>
            * [.getType()](#module_concerto-core.Property+getType) ⇒
<code>string</code>
            * [.isOptional()](#module_concerto-core.Property+isOptional) ⇒
<code>boolean</code>
            * [.getFullyQualifiedTypeName()](#module_concerto-
core.Property+getFullyQualifiedTypeName) ⇒ <code>string</code>
            * [.getFullyQualifiedName()](#module_concerto-
core.Property+getFullyQualifiedName) ⇒ <code>string</code>
            * [.getNamespace()](#module_concerto-core.Property+getNamespace) ⇒
<code>string</code>
            * [.isArray()](#module_concerto-core.Property+isArray) ⇒
<code>boolean</code>
            * [.isTypeEnum()](#module_concerto-core.Property+isTypeEnum) ⇒
```

```
<code>boolean</code>
            * [.isPrimitive()](#module_concerto-core.Property+isPrimitive) ⇒
<code>boolean</code>
        * [.RelationshipDeclaration](#module_concerto-core.RelationshipDeclaration)
⇐ <code>Property</code>
            * [new RelationshipDeclaration(parent, ast)](#new_module_concerto-
core.RelationshipDeclaration_new)
            * [.validate(classDecl)](#module_concerto-
core.RelationshipDeclaration+validate)
            * [.toString()](#module_concerto-core.RelationshipDeclaration+toString)
⇒ <code>String</code>
            * [.isRelationship()](#module_concerto-
core.RelationshipDeclaration+isRelationship) ⇒ <code>boolean</code>
        * [.TransactionDeclaration](#module_concerto-core.TransactionDeclaration) ⇐
<code>ClassDeclaration</code>
            * [new TransactionDeclaration(modelFile, ast)](#new_module_concerto-
core.TransactionDeclaration_new)
            * [.declarationKind()](#module_concerto-
core.TransactionDeclaration+declarationKind) ⇒ <code>string</code>
        * *[.Identifiable](#module_concerto-core.Identifiable) ⇐
<code>Typed</code>*
            * *[new Identifiable(modelManager, classDeclaration, ns, type, id,
timestamp)](#new_module_concerto-core.Identifiable_new)*
            * *[.getTimestamp()](#module_concerto-core.Identifiable+getTimestamp) ⇒
<code>string</code>*
            * *[.getIdentifier()](#module_concerto-core.Identifiable+getIdentifier)
⇒ <code>string</code>*
            * *[.setIdentifier(id)](#module_concerto-
core.Identifiable+setIdentifier)*
            * *[.getFullyQualifiedIdentifier()](#module_concerto-
core.Identifiable+getFullyQualifiedIdentifier) ⇒ <code>string</code>*
            * *[.toString()](#module_concerto-core.Identifiable+toString) ⇒
<code>String</code>*
            * *[.isRelationship()](#module_concerto-
core.Identifiable+isRelationship) ⇒ <code>boolean</code>*
            * *[.isResource()](#module_concerto-core.Identifiable+isResource) ⇒
<code>boolean</code>*
            * *[.toURI()](#module_concerto-core.Identifiable+toURI) ⇒
<code>String</code>*
        * [.Resource](#module_concerto-core.Resource) ⇐ <code>Identifiable</code>
            * [new Resource(modelManager, classDeclaration, ns, type, id,
timestamp)](#new_module_concerto-core.Resource_new)
            * [.toString()](#module_concerto-core.Resource+toString) ⇒
<code>String</code>
            * [.isResource()](#module_concerto-core.Resource+isResource) ⇒
<code>boolean</code>
            * [.isConcept()](#module_concerto-core.Resource+isConcept) ⇒
<code>boolean</code>
            * [.isIdentifiable()](#module_concerto-core.Resource+isIdentifiable) ⇒
<code>boolean</code>
            * [.toJSON()](#module_concerto-core.Resource+toJSON) ⇒
<code>Object</code>
        * *[.Typed](#module_concerto-core.Typed)*
            * *[new Typed(modelManager, classDeclaration, ns, type)]
(#new_module_concerto-core.Typed_new)*
            * *[.getType()](#module_concerto-core.Typed+getType) ⇒
<code>string</code>*
            * *[.getFullyQualifiedType()](#module_concerto-
core.Typed+getFullyQualifiedType) ⇒ <code>string</code>*
```

```
            * *[.getNamespace()](#module_concerto-core.Typed+getNamespace) ⇒
<code>string</code>*
            * *[.setPropertyValue(propName, value)](#module_concerto-
core.Typed+setPropertyValue)*
            * *[.addArrayValue(propName, value)](#module_concerto-
core.Typed+addArrayValue)*
            * *[.instanceOf(fqt)](#module_concerto-core.Typed+instanceOf) ⇒
<code>boolean</code>*
            * *[.toJSON()](#module_concerto-core.Typed+toJSON)*
        * [.ModelLoader](#module_concerto-core.ModelLoader)
            * [.loadModelManager(ctoFiles, options)](#module_concerto-
core.ModelLoader.loadModelManager) ⇒ <code>object</code>
            * [.loadModelManagerFromModelFiles(modelFiles, [fileNames], options)]
(#module_concerto-core.ModelLoader.loadModelManagerFromModelFiles) ⇒
<code>object</code>
        * [.ModelManager](#module_concerto-core.ModelManager)
            * [new ModelManager([options])](#new_module_concerto-
core.ModelManager_new)
            * [.addCTOModel(cto, [fileName], [disableValidation])]
(#module_concerto-core.ModelManager+addCTOModel) ⇒ <code>Object</code>
        * [.SecurityException](#module_concerto-core.SecurityException) ⇐
<code>BaseException</code>
            * [new SecurityException(message)](#new_module_concerto-
core.SecurityException_new)
        * [.Serializer](#module_concerto-core.Serializer)
            * [new Serializer(factory, modelManager, [options])]
(#new_module_concerto-core.Serializer_new)
            * [.setDefaultOptions(newDefaultOptions)](#module_concerto-
core.Serializer+setDefaultOptions)
            * [.toJSON(resource, [options])](#module_concerto-
core.Serializer+toJSON) ⇒ <code>Object</code>
            * [.fromJSON(jsonObject, [options])](#module_concerto-
core.Serializer+fromJSON) ⇒ <code>Resource</code>
        * [.TypeNotFoundException](#module_concerto-core.TypeNotFoundException) ⇐
<code>BaseException</code>
            * [new TypeNotFoundException(typeName, message, component)]
(#new_module_concerto-core.TypeNotFoundException_new)
            * [.getTypeName()](#module_concerto-
core.TypeNotFoundException+getTypeName) ⇒ <code>string</code>
        * [.BaseException](#module_concerto-core.BaseException) ⇐
<code>Error</code>
            * [new BaseException(message, component)](#new_module_concerto-
core.BaseException_new)
        * [.BaseFileException](#module_concerto-core.BaseFileException) ⇐
<code>BaseException</code>
            * [new BaseFileException(message, fileLocation, fullMessage,
[fileName], [component])](#new_module_concerto-core.BaseFileException_new)
            * [.getFileLocation()](#module_concerto-
core.BaseFileException+getFileLocation) ⇒ <code>string</code>
            * [.getShortMessage()](#module_concerto-
core.BaseFileException+getShortMessage) ⇒ <code>string</code>
            * [.getFileName()](#module_concerto-core.BaseFileException+getFileName)
⇒ <code>string</code>
        * [.FileDownloader](#module_concerto-core.FileDownloader)
            * [new FileDownloader(fileLoader, getExternalImports, concurrency)]
(#new_module_concerto-core.FileDownloader_new)
            * [.downloadExternalDependencies(files, [options])](#module_concerto-
core.FileDownloader+downloadExternalDependencies) ⇒ <code>Promise</code>
            * [.runJob(job, fileLoader)](#module_concerto-
```

core.FileDownloader+runJob) ⇒ <code>Promise</code>
        * [.TypedStack](#module_concerto-core.TypedStack)
            * [new TypedStack(resource)](#new_module_concerto-core.TypedStack_new)
            * [.push(obj, expectedType)](#module_concerto-core.TypedStack+push)
            * [.pop(expectedType)](#module_concerto-core.TypedStack+pop) ⇒
<code>Object</code>
            * [.peek(expectedType)](#module_concerto-core.TypedStack+peek) ⇒
<code>Object</code>
            * [.clear()](#module_concerto-core.TypedStack+clear)
    * _inner_
        * [~version](#module_concerto-core..version) : <code>Object</code>

<a name="module_concerto-core.AstModelManager"></a>

### concerto-core.AstModelManager
Manages the Concerto model files in AST format.

The structure of Resources (Assets, Transactions, Participants) is
modelled
in a set of Concerto files. The contents of these files are managed
by the ModelManager. Each Concerto file has a single namespace and
contains
a set of asset, transaction and participant type definitions.

Concerto applications load their Concerto files and then call the [addModelFile]
(ModelManager#addModelFile)
method to register the Concerto file(s) with the ModelManager.

Kind: static class of [<code>concerto-core</code>](#module_concerto-core)
<a name="new_module_concerto-core.AstModelManager_new"></a>

#### new AstModelManager([options])
Create the ModelManager.

Param	Type	Description
[options]	<code>object</code>	Serializer options

<a name="module_concerto-core.BaseModelManager"></a>

### concerto-core.BaseModelManager
Manages the Concerto model files.

The structure of Resources (Assets, Transactions, Participants) is
modelled
in a set of Concerto files. The contents of these files are managed
by the ModelManager. Each Concerto file has a single namespace and
contains
a set of asset, transaction and participant type definitions.

Concerto applications load their Concerto files and then call the [addModelFile]
(ModelManager#addModelFile)
method to register the Concerto file(s) with the ModelManager.

Use the Concerto class to validate instances.

Kind: static class of [<code>concerto-core</code>](#module_concerto-core)

```
* [.BaseModelManager](#module_concerto-core.BaseModelManager)
    * [new BaseModelManager([options], [processFile])](#new_module_concerto-
core.BaseModelManager_new)
    * [.isModelManager()](#module_concerto-core.BaseModelManager+isModelManager) ⇒
<code>boolean</code>
    * [.accept(visitor, parameters)](#module_concerto-core.BaseModelManager+accept)
⇒ <code>Object</code>
    * [.validateModelFile(modelFile, [fileName])](#module_concerto-
core.BaseModelManager+validateModelFile)
    * [.addModelFile(modelFile, [cto], [fileName], [disableValidation])]
(#module_concerto-core.BaseModelManager+addModelFile) ⇒ <code>Object</code>
    * [.addModel(modelInput, [cto], [fileName], [disableValidation])]
(#module_concerto-core.BaseModelManager+addModel) ⇒ <code>Object</code>
    * [.updateModelFile(modelFile, [fileName], [disableValidation])]
(#module_concerto-core.BaseModelManager+updateModelFile) ⇒ <code>Object</code>
    * [.deleteModelFile(namespace)](#module_concerto-
core.BaseModelManager+deleteModelFile)
    * [.addModelFiles(modelFiles, [fileNames], [disableValidation])]
(#module_concerto-core.BaseModelManager+addModelFiles) ⇒
<code>Array.&lt;Object&gt;</code>
    * [.validateModelFiles()](#module_concerto-
core.BaseModelManager+validateModelFiles)
    * [.updateExternalModels([options], [fileDownloader])](#module_concerto-
core.BaseModelManager+updateExternalModels) ⇒ <code>Promise</code>
    * [.writeModelsToFileSystem(path, [options])](#module_concerto-
core.BaseModelManager+writeModelsToFileSystem)
    * [.getModels([options])](#module_concerto-core.BaseModelManager+getModels) ⇒
<code>Array.&lt;{name:string, content:string}&gt;</code>
    * [.clearModelFiles()](#module_concerto-core.BaseModelManager+clearModelFiles)
    * [.getModelFile(namespace)](#module_concerto-
core.BaseModelManager+getModelFile) ⇒ <code>ModelFile</code>
    * [.getNamespaces()](#module_concerto-core.BaseModelManager+getNamespaces) ⇒
<code>Array.&lt;string&gt;</code>
    * [.getType(qualifiedName)](#module_concerto-core.BaseModelManager+getType) ⇒
<code>ClassDeclaration</code>
    * [.getAssetDeclarations()](#module_concerto-
core.BaseModelManager+getAssetDeclarations) ⇒
<code>Array.&lt;AssetDeclaration&gt;</code>
    * [.getTransactionDeclarations()](#module_concerto-
core.BaseModelManager+getTransactionDeclarations) ⇒
<code>Array.&lt;TransactionDeclaration&gt;</code>
    * [.getEventDeclarations()](#module_concerto-
core.BaseModelManager+getEventDeclarations) ⇒
<code>Array.&lt;EventDeclaration&gt;</code>
    * [.getParticipantDeclarations()](#module_concerto-
core.BaseModelManager+getParticipantDeclarations) ⇒
<code>Array.&lt;ParticipantDeclaration&gt;</code>
    * [.getEnumDeclarations()](#module_concerto-
core.BaseModelManager+getEnumDeclarations) ⇒
<code>Array.&lt;EnumDeclaration&gt;</code>
    * [.getConceptDeclarations()](#module_concerto-
core.BaseModelManager+getConceptDeclarations) ⇒
<code>Array.&lt;ConceptDeclaration&gt;</code>
    * [.getFactory()](#module_concerto-core.BaseModelManager+getFactory) ⇒
<code>Factory</code>
    * [.getSerializer()](#module_concerto-core.BaseModelManager+getSerializer) ⇒
<code>Serializer</code>
    * [.getDecoratorFactories()](#module_concerto-
core.BaseModelManager+getDecoratorFactories) ⇒
```

```
<code>Array.&lt;DecoratorFactory&gt;</code>
    * [.addDecoratorFactory(factory)](#module_concerto-
core.BaseModelManager+addDecoratorFactory)
    * [.derivesFrom(fqt1, fqt2)](#module_concerto-
core.BaseModelManager+derivesFrom) ⇒ <code>boolean</code>
    * [.resolveMetaModel(metaModel)](#module_concerto-
core.BaseModelManager+resolveMetaModel) ⇒ <code>object</code>
    * [.fromAst(ast)](#module_concerto-core.BaseModelManager+fromAst)
    * [.getAst([resolve])](#module_concerto-core.BaseModelManager+getAst) ⇒ <code>\
*</code>

<a name="new_module_concerto-core.BaseModelManager_new"></a>

#### new BaseModelManager([options], [processFile])
Create the ModelManager.


| Param | Type | Description |
| --- | --- | --- |
| [options] | <code>object</code> | Serializer options |
| [processFile] | <code>\*</code> | how to obtain a concerto AST from an input to
the model manager |

<a name="module_concerto-core.BaseModelManager+isModelManager"></a>

#### baseModelManager.isModelManager() ⇒ <code>boolean</code>
Returns true

**Kind**: instance method of [<code>BaseModelManager</code>](#module_concerto-
core.BaseModelManager)
**Returns**: <code>boolean</code> - true
<a name="module_concerto-core.BaseModelManager+accept"></a>

#### baseModelManager.accept(visitor, parameters) ⇒ <code>Object</code>
Visitor design pattern

**Kind**: instance method of [<code>BaseModelManager</code>](#module_concerto-
core.BaseModelManager)
**Returns**: <code>Object</code> - the result of visiting or null

| Param | Type | Description |
| --- | --- | --- |
| visitor | <code>Object</code> | the visitor |
| parameters | <code>Object</code> | the parameter |

<a name="module_concerto-core.BaseModelManager+validateModelFile"></a>

#### baseModelManager.validateModelFile(modelFile, [fileName])
Validates a Concerto file (as a string) to the ModelManager.
Concerto files have a single namespace.

Note that if there are dependencies between multiple files the files
must be added in dependency order, or the addModelFiles method can be
used to add a set of files irrespective of dependencies.

**Kind**: instance method of [<code>BaseModelManager</code>](#module_concerto-
core.BaseModelManager)
**Throws**:
```

- <code>IllegalModelException</code>

Param	Type	Description
modelFile	<code>string</code> | <code>ModelFile</code>	The Concerto file as a string
[fileName]	<code>string</code>	a file name to associate with the model file

<a name="module_concerto-core.BaseModelManager+addModelFile"></a>

#### baseModelManager.addModelFile(modelFile, [cto], [fileName], [disableValidation]) ⇒ <code>Object</code>
Adds a Concerto file (as an AST) to the ModelManager.
Concerto files have a single namespace. If a Concerto file with the
same namespace has already been added to the ModelManager then it
will be replaced.
Note that if there are dependencies between multiple files the files
must be added in dependency order, or the addModelFiles method can be
used to add a set of files irrespective of dependencies.

Kind: instance method of [<code>BaseModelManager</code>](#module_concerto-core.BaseModelManager)
Returns: <code>Object</code> - The newly added model file (internal).
Throws:

- <code>IllegalModelException</code>

Param	Type	Description
modelFile	<code>ModelFile</code>	Model as a ModelFile object
[cto]	<code>string</code>	an optional cto string
[fileName]	<code>string</code>	an optional file name to associate with the model file
[disableValidation]	<code>boolean</code>	If true then the model files are not validated

<a name="module_concerto-core.BaseModelManager+addModel"></a>

#### baseModelManager.addModel(modelInput, [cto], [fileName], [disableValidation]) ⇒ <code>Object</code>
Adds a model to the ModelManager.
Concerto files have a single namespace. If a Concerto file with the
same namespace has already been added to the ModelManager then it
will be replaced.
Note that if there are dependencies between multiple files the files
must be added in dependency order, or the addModel method can be
used to add a set of files irrespective of dependencies.

Kind: instance method of [<code>BaseModelManager</code>](#module_concerto-core.BaseModelManager)
Returns: <code>Object</code> - The newly added model file (internal).
Throws:

- <code>IllegalModelException</code>

| Param | Type | Description |

| --- | --- | --- |
modelInput	<code>*</code>	Model (as a string or object)
[cto]	<code>string</code>	an optional cto string
[fileName]	<code>string</code>	an optional file name to associate with the model file
[disableValidation]	<code>boolean</code>	If true then the model files are not validated

<a name="module_concerto-core.BaseModelManager+updateModelFile"></a>

#### baseModelManager.updateModelFile(modelFile, [fileName], [disableValidation]) ⇒ <code>Object</code>
Updates a Concerto file (as a string) on the ModelManager.
Concerto files have a single namespace. If a Concerto file with the
same namespace has already been added to the ModelManager then it
will be replaced.

Kind: instance method of [<code>BaseModelManager</code>](#module_concerto-core.BaseModelManager)
Returns: <code>Object</code> - The newly added model file (internal).
Throws:

- <code>IllegalModelException</code>

Param	Type	Description
modelFile	<code>string</code> | <code>ModelFile</code>	Model as a string or object
[fileName]	<code>string</code>	a file name to associate with the model file
[disableValidation]	<code>boolean</code>	If true then the model files are not validated

<a name="module_concerto-core.BaseModelManager+deleteModelFile"></a>

#### baseModelManager.deleteModelFile(namespace)
Remove the Concerto file for a given namespace

Kind: instance method of [<code>BaseModelManager</code>](#module_concerto-core.BaseModelManager)

Param	Type	Description
namespace	<code>string</code>	The namespace of the model file to delete.

<a name="module_concerto-core.BaseModelManager+addModelFiles"></a>

#### baseModelManager.addModelFiles(modelFiles, [fileNames], [disableValidation]) ⇒ <code>Array.&lt;Object&gt;</code>
Add a set of Concerto files to the model manager.

Kind: instance method of [<code>BaseModelManager</code>](#module_concerto-core.BaseModelManager)
Returns: <code>Array.&lt;Object&gt;</code> - The newly added model files (internal).

Param	Type	Description
modelFiles	<code>Array.&lt;string&gt;</code> |

<code>Array.&lt;ModelFile&gt;</code> | An array of models as strings or ModelFile
objects. |
| [fileNames] | <code>Array.&lt;string&gt;</code> | A array of file names to
associate with the model files |
| [disableValidation] | <code>boolean</code> | If true then the model files are not
validated |

<a name="module_concerto-core.BaseModelManager+validateModelFiles"></a>

#### baseModelManager.validateModelFiles()
Validates all models files in this model manager

Kind: instance method of [<code>BaseModelManager</code>](#module_concerto-
core.BaseModelManager)
<a name="module_concerto-core.BaseModelManager+updateExternalModels"></a>

#### baseModelManager.updateExternalModels([options], [fileDownloader]) ⇒
<code>Promise</code>
Downloads all ModelFiles that are external dependencies and adds or
updates them in this ModelManager.

Kind: instance method of [<code>BaseModelManager</code>](#module_concerto-
core.BaseModelManager)
Returns: <code>Promise</code> - a promise when the download and update
operation is completed.
Throws:

- <code>IllegalModelException</code> if the models fail validation

Param	Type	Description
[options]	<code>Object</code>	Options object passed to ModelFileLoaders
[fileDownloader]	<code>FileDownloader</code>	an optional FileDownloader

<a name="module_concerto-core.BaseModelManager+writeModelsToFileSystem"></a>

#### baseModelManager.writeModelsToFileSystem(path, [options])
Write all models in this model manager to the specified path in the file system

Kind: instance method of [<code>BaseModelManager</code>](#module_concerto-
core.BaseModelManager)

Param	Type	Description
path	<code>string</code>	to a local directory
[options]	<code>Object</code>	Options object
options.includeExternalModels	<code>boolean</code>	If true, external models
are written to the file system. Defaults to true

<a name="module_concerto-core.BaseModelManager+getModels"></a>

#### baseModelManager.getModels([options]) ⇒ <code>Array.&lt;{name:string,
content:string}&gt;</code>
Gets all the Concerto models

Kind: instance method of [<code>BaseModelManager</code>](#module_concerto-
core.BaseModelManager)
Returns: <code>Array.&lt;{name:string, content:string}&gt;</code> - the name

and content of each CTO file

Param	Type	Description
[options]	<code>Object</code>	Options object
options.includeExternalModels	<code>boolean</code>	If true, external models are written to the file system. Defaults to true

<a name="module_concerto-core.BaseModelManager+clearModelFiles"></a>

#### baseModelManager.clearModelFiles()
Remove all registered Concerto files

Kind: instance method of [<code>BaseModelManager</code>](#module_concerto-core.BaseModelManager)
<a name="module_concerto-core.BaseModelManager+getModelFile"></a>

#### baseModelManager.getModelFile(namespace) ⇒ <code>ModelFile</code>
Get the ModelFile associated with a namespace

Kind: instance method of [<code>BaseModelManager</code>](#module_concerto-core.BaseModelManager)
Returns: <code>ModelFile</code> - registered ModelFile for the namespace or null

Param	Type	Description
namespace	<code>string</code>	the namespace containing the ModelFile

<a name="module_concerto-core.BaseModelManager+getNamespaces"></a>

#### baseModelManager.getNamespaces() ⇒ <code>Array.&lt;string&gt;</code>
Get the namespaces registered with the ModelManager.

Kind: instance method of [<code>BaseModelManager</code>](#module_concerto-core.BaseModelManager)
Returns: <code>Array.&lt;string&gt;</code> - namespaces - the namespaces that have been registered.
<a name="module_concerto-core.BaseModelManager+getType"></a>

#### baseModelManager.getType(qualifiedName) ⇒ <code>ClassDeclaration</code>
Look up a type in all registered namespaces.

Kind: instance method of [<code>BaseModelManager</code>](#module_concerto-core.BaseModelManager)
Returns: <code>ClassDeclaration</code> - - the class declaration for the specified type.
Throws:

- <code>TypeNotFoundException</code> - if the type cannot be found or is a primitive type.

Param	Type	Description
qualifiedName	<code>string</code>	fully qualified type name.

<a name="module_concerto-core.BaseModelManager+getAssetDeclarations"></a>

#### baseModelManager.getAssetDeclarations() ⇒
<code>Array.&lt;AssetDeclaration&gt;</code>
Get the AssetDeclarations defined in this model manager

Kind: instance method of [<code>BaseModelManager</code>](#module_concerto-core.BaseModelManager)
Returns: <code>Array.&lt;AssetDeclaration&gt;</code> - the AssetDeclarations defined in the model manager
<a name="module_concerto-core.BaseModelManager+getTransactionDeclarations"></a>

#### baseModelManager.getTransactionDeclarations() ⇒
<code>Array.&lt;TransactionDeclaration&gt;</code>
Get the TransactionDeclarations defined in this model manager

Kind: instance method of [<code>BaseModelManager</code>](#module_concerto-core.BaseModelManager)
Returns: <code>Array.&lt;TransactionDeclaration&gt;</code> - the TransactionDeclarations defined in the model manager
<a name="module_concerto-core.BaseModelManager+getEventDeclarations"></a>

#### baseModelManager.getEventDeclarations() ⇒
<code>Array.&lt;EventDeclaration&gt;</code>
Get the EventDeclarations defined in this model manager

Kind: instance method of [<code>BaseModelManager</code>](#module_concerto-core.BaseModelManager)
Returns: <code>Array.&lt;EventDeclaration&gt;</code> - the EventDeclaration defined in the model manager
<a name="module_concerto-core.BaseModelManager+getParticipantDeclarations"></a>

#### baseModelManager.getParticipantDeclarations() ⇒
<code>Array.&lt;ParticipantDeclaration&gt;</code>
Get the ParticipantDeclarations defined in this model manager

Kind: instance method of [<code>BaseModelManager</code>](#module_concerto-core.BaseModelManager)
Returns: <code>Array.&lt;ParticipantDeclaration&gt;</code> - the ParticipantDeclaration defined in the model manager
<a name="module_concerto-core.BaseModelManager+getEnumDeclarations"></a>

#### baseModelManager.getEnumDeclarations() ⇒
<code>Array.&lt;EnumDeclaration&gt;</code>
Get the EnumDeclarations defined in this model manager

Kind: instance method of [<code>BaseModelManager</code>](#module_concerto-core.BaseModelManager)
Returns: <code>Array.&lt;EnumDeclaration&gt;</code> - the EnumDeclaration defined in the model manager
<a name="module_concerto-core.BaseModelManager+getConceptDeclarations"></a>

#### baseModelManager.getConceptDeclarations() ⇒
<code>Array.&lt;ConceptDeclaration&gt;</code>
Get the Concepts defined in this model manager

Kind: instance method of [<code>BaseModelManager</code>](#module_concerto-core.BaseModelManager)
Returns: <code>Array.&lt;ConceptDeclaration&gt;</code> - the ConceptDeclaration defined in the model manager
<a name="module_concerto-core.BaseModelManager+getFactory"></a>

#### baseModelManager.getFactory() ⇒ <code>Factory</code>
Get a factory for creating new instances of types defined in this model manager.

Kind: instance method of [<code>BaseModelManager</code>](#module_concerto-core.BaseModelManager)
Returns: <code>Factory</code> - A factory for creating new instances of types defined in this model manager.
<a name="module_concerto-core.BaseModelManager+getSerializer"></a>

#### baseModelManager.getSerializer() ⇒ <code>Serializer</code>
Get a serializer for serializing instances of types defined in this model manager.

Kind: instance method of [<code>BaseModelManager</code>](#module_concerto-core.BaseModelManager)
Returns: <code>Serializer</code> - A serializer for serializing instances of types defined in this model manager.
<a name="module_concerto-core.BaseModelManager+getDecoratorFactories"></a>

#### baseModelManager.getDecoratorFactories() ⇒ <code>Array.&lt;DecoratorFactory&gt;</code>
Get the decorator factories for this model manager.

Kind: instance method of [<code>BaseModelManager</code>](#module_concerto-core.BaseModelManager)
Returns: <code>Array.&lt;DecoratorFactory&gt;</code> - The decorator factories for this model manager.
<a name="module_concerto-core.BaseModelManager+addDecoratorFactory"></a>

#### baseModelManager.addDecoratorFactory(factory)
Add a decorator factory to this model manager.

Kind: instance method of [<code>BaseModelManager</code>](#module_concerto-core.BaseModelManager)

Param	Type	Description
factory	<code>DecoratorFactory</code>	The decorator factory to add to this model manager.

<a name="module_concerto-core.BaseModelManager+derivesFrom"></a>

#### baseModelManager.derivesFrom(fqt1, fqt2) ⇒ <code>boolean</code>
Checks if this fully qualified type name is derived from another.

Kind: instance method of [<code>BaseModelManager</code>](#module_concerto-core.BaseModelManager)
Returns: <code>boolean</code> - True if this instance is an instance of the specified fully
qualified type name, false otherwise.

Param	Type	Description
fqt1	<code>string</code>	The fully qualified type name to check.
fqt2	<code>string</code>	The fully qualified type name it is may be derived from.

<a name="module_concerto-core.BaseModelManager+resolveMetaModel"></a>

#### baseModelManager.resolveMetaModel(metaModel) ⇒ <code>object</code>
Resolve the namespace for names in the metamodel

Kind: instance method of [<code>BaseModelManager</code>](#module_concerto-core.BaseModelManager)
Returns: <code>object</code> - the resolved metamodel

Param	Type	Description
metaModel	<code>object</code>	the MetaModel

<a name="module_concerto-core.BaseModelManager+fromAst"></a>

#### baseModelManager.fromAst(ast)
Populates the model manager from a models metamodel AST

Kind: instance method of [<code>BaseModelManager</code>](#module_concerto-core.BaseModelManager)

Param	Type	Description
ast	<code>*</code>	the metamodel

<a name="module_concerto-core.BaseModelManager+getAst"></a>

#### baseModelManager.getAst([resolve]) ⇒ <code>*</code>
Get the full ast (metamodel instances) for a modelmanager

Kind: instance method of [<code>BaseModelManager</code>](#module_concerto-core.BaseModelManager)
Returns: <code>*</code> - the metamodel

Param	Type	Description
[resolve]	<code>boolean</code>	whether to resolve names

<a name="module_concerto-core.Concerto"></a>

### concerto-core.Concerto
Runtime API for Concerto.

Kind: static class of [<code>concerto-core</code>](#module_concerto-core)

* [.Concerto](#module_concerto-core.Concerto)
    * [new Concerto(modelManager)](#new_module_concerto-core.Concerto_new)
    * [.validate(obj, [options])](#module_concerto-core.Concerto+validate)
    * [.getModelManager()](#module_concerto-core.Concerto+getModelManager) ⇒ <code>*</code>
    * [.isObject(obj)](#module_concerto-core.Concerto+isObject) ⇒ <code>boolean</code>
    * [.getTypeDeclaration(obj)](#module_concerto-core.Concerto+getTypeDeclaration) ⇒ <code>*</code>
    * [.getIdentifier(obj)](#module_concerto-core.Concerto+getIdentifier) ⇒ <code>string</code>
    * [.isIdentifiable(obj)](#module_concerto-core.Concerto+isIdentifiable) ⇒ <code>boolean</code>
    * [.isRelationship(obj)](#module_concerto-core.Concerto+isRelationship) ⇒ <code>boolean</code>
    * [.setIdentifier(obj, id)](#module_concerto-core.Concerto+setIdentifier) ⇒

```
<code>\*</code>
    * [.getFullyQualifiedIdentifier(obj)](#module_concerto-
core.Concerto+getFullyQualifiedIdentifier) ⇒ <code>string</code>
    * [.toURI(obj)](#module_concerto-core.Concerto+toURI) ⇒ <code>string</code>
    * [.fromURI(uri)](#module_concerto-core.Concerto+fromURI) ⇒ <code>\*</code>
    * [.getType(obj)](#module_concerto-core.Concerto+getType) ⇒ <code>string</code>
    * [.getNamespace(obj)](#module_concerto-core.Concerto+getNamespace) ⇒
<code>string</code>

<a name="new_module_concerto-core.Concerto_new"></a>

#### new Concerto(modelManager)
Create a Concerto instance.


| Param | Type | Description |
| --- | --- | --- |
| modelManager | <code>\*</code> | The this.modelManager to use for validation etc.
|

<a name="module_concerto-core.Concerto+validate"></a>

#### concerto.validate(obj, [options])
Validates the instance against its model.

**Kind**: instance method of [<code>Concerto</code>](#module_concerto-
core.Concerto)
**Throws**:

- <code>Error</code> - if the instance if invalid with respect to the model


| Param | Type | Description |
| --- | --- | --- |
| obj | <code>\*</code> | the input object |
| [options] | <code>\*</code> | the validation options |

<a name="module_concerto-core.Concerto+getModelManager"></a>

#### concerto.getModelManager() ⇒ <code>\*</code>
Returns the model manager

**Kind**: instance method of [<code>Concerto</code>](#module_concerto-
core.Concerto)
**Returns**: <code>\*</code> - the model manager associated with this Concerto
class
<a name="module_concerto-core.Concerto+isObject"></a>

#### concerto.isObject(obj) ⇒ <code>boolean</code>
Returns true if the input object is a Concerto object

**Kind**: instance method of [<code>Concerto</code>](#module_concerto-
core.Concerto)
**Returns**: <code>boolean</code> - true if the object has a $class attribute

| Param | Type | Description |
| --- | --- | --- |
| obj | <code>\*</code> | the input object |
```

<a name="module_concerto-core.Concerto+getTypeDeclaration"></a>

#### concerto.getTypeDeclaration(obj) ⇒ <code>*</code>
Returns the ClassDeclaration for an object, or throws an exception

Kind: instance method of [<code>Concerto</code>](#module_concerto-core.Concerto)
Returns: <code>*</code> - the ClassDeclaration for the type
Throw: <code>Error</code> an error if the object does not have a $class attribute

Param	Type	Description
obj	<code>*</code>	the input object

<a name="module_concerto-core.Concerto+getIdentifier"></a>

#### concerto.getIdentifier(obj) ⇒ <code>string</code>
Gets the identifier for an object

Kind: instance method of [<code>Concerto</code>](#module_concerto-core.Concerto)
Returns: <code>string</code> - The identifier for this object

Param	Type	Description
obj	<code>*</code>	the input object

<a name="module_concerto-core.Concerto+isIdentifiable"></a>

#### concerto.isIdentifiable(obj) ⇒ <code>boolean</code>
Returns true if the object has an identifier

Kind: instance method of [<code>Concerto</code>](#module_concerto-core.Concerto)
Returns: <code>boolean</code> - is the object has been defined with an identifier in the model

Param	Type	Description
obj	<code>*</code>	the input object

<a name="module_concerto-core.Concerto+isRelationship"></a>

#### concerto.isRelationship(obj) ⇒ <code>boolean</code>
Returns true if the object is a relationship. Relationships are strings
of the form: 'resource:org.accordproject.Order#001' (a relationship)
to the 'Order' identifiable, with the id 001.

Kind: instance method of [<code>Concerto</code>](#module_concerto-core.Concerto)
Returns: <code>boolean</code> - true if the object is a relationship

Param	Type	Description
obj	<code>*</code>	the input object

<a name="module_concerto-core.Concerto+setIdentifier"></a>

```
#### concerto.setIdentifier(obj, id) ⇒ <code>\*</code>
Set the identifier for an object. This method does *not* mutate the
input object, use the return object.

**Kind**: instance method of [<code>Concerto</code>](#module_concerto-
core.Concerto)
**Returns**: <code>\*</code> - the input object with the identifier set

| Param | Type | Description |
| --- | --- | --- |
| obj | <code>\*</code> | the input object |
| id | <code>string</code> | the new identifier |

<a name="module_concerto-core.Concerto+getFullyQualifiedIdentifier"></a>

#### concerto.getFullyQualifiedIdentifier(obj) ⇒ <code>string</code>
Returns the fully qualified identifier for an object

**Kind**: instance method of [<code>Concerto</code>](#module_concerto-
core.Concerto)
**Returns**: <code>string</code> - the fully qualified identifier

| Param | Type | Description |
| --- | --- | --- |
| obj | <code>\*</code> | the input object |

<a name="module_concerto-core.Concerto+toURI"></a>

#### concerto.toURI(obj) ⇒ <code>string</code>
Returns a URI for an object

**Kind**: instance method of [<code>Concerto</code>](#module_concerto-
core.Concerto)
**Returns**: <code>string</code> - the URI for the object

| Param | Type | Description |
| --- | --- | --- |
| obj | <code>\*</code> | the input object |

<a name="module_concerto-core.Concerto+fromURI"></a>

#### concerto.fromURI(uri) ⇒ <code>\*</code>
Parses a resource URI into typeDeclaration and id components.

**Kind**: instance method of [<code>Concerto</code>](#module_concerto-
core.Concerto)
**Returns**: <code>\*</code> - an object with typeDeclaration and id attributes
**Throws**:

- <code>Error</code> if the URI is invalid or the type does not exist
in the model manager


| Param | Type | Description |
| --- | --- | --- |
| uri | <code>string</code> | the input URI |

<a name="module_concerto-core.Concerto+getType"></a>
```

```
#### concerto.getType(obj) ⇒ <code>string</code>
Returns the short type name

**Kind**: instance method of [<code>Concerto</code>](#module_concerto-
core.Concerto)
**Returns**: <code>string</code> - the short type name

| Param | Type | Description |
| --- | --- | --- |
| obj | <code>\*</code> | the input object |

<a name="module_concerto-core.Concerto+getNamespace"></a>

#### concerto.getNamespace(obj) ⇒ <code>string</code>
Returns the namespace for the object

**Kind**: instance method of [<code>Concerto</code>](#module_concerto-
core.Concerto)
**Returns**: <code>string</code> - the namespace

| Param | Type | Description |
| --- | --- | --- |
| obj | <code>\*</code> | the input object |

<a name="module_concerto-core.DecoratorManager"></a>

### concerto-core.DecoratorManager
Utility functions to work with
[DecoratorCommandSet](https://models.accordproject.org/concerto/decorators.cto)

**Kind**: static class of [<code>concerto-core</code>](#module_concerto-core)

* [.DecoratorManager](#module_concerto-core.DecoratorManager)
    * [.decorateModels(modelManager, decoratorCommandSet)](#module_concerto-
core.DecoratorManager.decorateModels) ⇒ <code>ModelManager</code>
    * [.falsyOrEqual(test, value)](#module_concerto-
core.DecoratorManager.falsyOrEqual) ⇒ <code>Boolean</code>
    * [.applyDecorator(decorated, type, newDecorator)](#module_concerto-
core.DecoratorManager.applyDecorator)
    * [.executeCommand(namespace, declaration, command)](#module_concerto-
core.DecoratorManager.executeCommand)

<a name="module_concerto-core.DecoratorManager.decorateModels"></a>

#### DecoratorManager.decorateModels(modelManager, decoratorCommandSet) ⇒
<code>ModelManager</code>
Applies all the decorator commands from the DecoratorCommandSet
to the ModelManager.

**Kind**: static method of [<code>DecoratorManager</code>](#module_concerto-
core.DecoratorManager)
**Returns**: <code>ModelManager</code> - a new model manager with the decorations
applied

| Param | Type | Description |
| --- | --- | --- |
| modelManager | <code>ModelManager</code> | the input model manager |
| decoratorCommandSet | <code>\*</code> | the DecoratorCommandSet object |
<a name="module_concerto-core.DecoratorManager.falsyOrEqual"></a>

#### DecoratorManager.falsyOrEqual(test, value) ⇒ <code>Boolean</code>
Compares two values. If the first argument is falsy
the function returns true.

**Kind**: static method of [<code>DecoratorManager</code>](#module_concerto-
core.DecoratorManager)
**Returns**: <code>Boolean</code> - true if the lhs is falsy or test === value

| Param | Type | Description |
| --- | --- | --- |
| test | <code>string</code> \| <code>null</code> | the value to test (lhs) |
| value | <code>string</code> | the value to compare (rhs) |

<a name="module_concerto-core.DecoratorManager.applyDecorator"></a>

#### DecoratorManager.applyDecorator(decorated, type, newDecorator)
Applies a decorator to a decorated model element.

**Kind**: static method of [<code>DecoratorManager</code>](#module_concerto-
core.DecoratorManager)

| Param | Type | Description |
| --- | --- | --- |
| decorated | <code>\*</code> | the type to apply the decorator to |
| type | <code>string</code> | the command type |
| newDecorator | <code>\*</code> | the decorator to add |

<a name="module_concerto-core.DecoratorManager.executeCommand"></a>

#### DecoratorManager.executeCommand(namespace, declaration, command)
Executes a Command against a ClassDeclaration, adding
decorators to the ClassDeclaration, or its properties, as required.

**Kind**: static method of [<code>DecoratorManager</code>](#module_concerto-
core.DecoratorManager)

| Param | Type | Description |
| --- | --- | --- |
| namespace | <code>string</code> | the namespace for the declaration |
| declaration | <code>\*</code> | the class declaration |
| command | <code>\*</code> | the Command object from the
org.accordproject.decoratorcommands model |

<a name="module_concerto-core.Factory"></a>

### concerto-core.Factory
Use the Factory to create instances of Resource: transactions, participants
and assets.

**Kind**: static class of [<code>concerto-core</code>](#module_concerto-core)

* [.Factory](#module_concerto-core.Factory)
    * [new Factory(modelManager)](#new_module_concerto-core.Factory_new)
    * _instance_
        * [.newResource(ns, type, [id], [options])](#module_concerto-
core.Factory+newResource) ⇒ <code>Resource</code>
        * [.newConcept(ns, type, [id], [options])](#module_concerto-
core.Factory+newConcept) ⇒ <code>Resource</code>
        * [.newRelationship(ns, type, id)](#module_concerto-
core.Factory+newRelationship) ⇒ <code>Relationship</code>
        * [.newTransaction(ns, type, [id], [options])](#module_concerto-
core.Factory+newTransaction) ⇒ <code>Resource</code>
        * [.newEvent(ns, type, [id], [options])](#module_concerto-
core.Factory+newEvent) ⇒ <code>Resource</code>
    * _static_
        * [.newId()](#module_concerto-core.Factory.newId) ⇒ <code>string</code>

<a name="new_module_concerto-core.Factory_new"></a>

#### new Factory(modelManager)
Create the factory.


| Param | Type | Description |
| --- | --- | --- |
| modelManager | <code>ModelManager</code> | The ModelManager to use for this
registry |

<a name="module_concerto-core.Factory+newResource"></a>

#### factory.newResource(ns, type, [id], [options]) ⇒ <code>Resource</code>
Create a new Resource with a given namespace, type name and id

**Kind**: instance method of [<code>Factory</code>](#module_concerto-core.Factory)
**Returns**: <code>Resource</code> - the new instance
**Throws**:

- <code>TypeNotFoundException</code> if the type is not registered with the
ModelManager


| Param | Type | Description |
| --- | --- | --- |
| ns | <code>String</code> | the namespace of the Resource |
| type | <code>String</code> | the type of the Resource |
| [id] | <code>String</code> | an optional string identifier |
| [options] | <code>Object</code> | an optional set of options |
| [options.disableValidation] | <code>boolean</code> | pass true if you want the
factory to return a [Resource](Resource) instead of a [ValidatedResource]
(ValidatedResource). Defaults to false. |
| [options.generate] | <code>String</code> | Pass one of: <dl>
<dt>sample</dt><dd>return a resource instance with generated sample data.</dd>
<dt>empty</dt><dd>return a resource instance with empty property values.</dd></dl>
|
| [options.includeOptionalFields] | <code>boolean</code> | if
<code>options.generate</code> is specified, whether optional fields should be
generated. |

<a name="module_concerto-core.Factory+newConcept"></a>

#### factory.newConcept(ns, type, [id], [options]) ⇒ <code>Resource</code>
Create a new Concept with a given namespace and type name

**Kind**: instance method of [<code>Factory</code>](#module_concerto-core.Factory)
**Returns**: <code>Resource</code> - the new instance
**Throws**:
```

- <code>TypeNotFoundException</code> if the type is not registered with the ModelManager

Param	Type	Description
ns	<code>String</code>	the namespace of the Concept
type	<code>String</code>	the type of the Concept
[id]	<code>String</code>	an optional string identifier
[options]	<code>Object</code>	an optional set of options
[options.disableValidation]	<code>boolean</code>	pass true if you want the factory to return a Concept instead of a ValidatedConcept. Defaults to false.
[options.generate]	<code>String</code>	Pass one of: <dl> <dt>sample</dt><dd>return a resource instance with generated sample data.</dd> <dt>empty</dt><dd>return a resource instance with empty property values.</dd></dl>
[options.includeOptionalFields]	<code>boolean</code>	if <code>options.generate</code> is specified, whether optional fields should be generated.

<a name="module_concerto-core.Factory+newRelationship"></a>

#### factory.newRelationship(ns, type, id) ⇒ <code>Relationship</code>
Create a new Relationship with a given namespace, type and identifier.
A relationship is a typed pointer to an instance. I.e the relationship
with `namespace = 'org.example'`, `type = 'Vehicle'` and `id = 'ABC' creates`
a pointer that points at an instance of org.example.Vehicle with the id
ABC.

Kind: instance method of [<code>Factory</code>](#module_concerto-core.Factory)
Returns: <code>Relationship</code> - - the new relationship instance
Throws:

- <code>TypeNotFoundException</code> if the type is not registered with the ModelManager

Param	Type	Description
ns	<code>String</code>	the namespace of the Resource
type	<code>String</code>	the type of the Resource
id	<code>String</code>	the identifier

<a name="module_concerto-core.Factory+newTransaction"></a>

#### factory.newTransaction(ns, type, [id], [options]) ⇒ <code>Resource</code>
Create a new transaction object. The identifier of the transaction is set to a
UUID.

Kind: instance method of [<code>Factory</code>](#module_concerto-core.Factory)
Returns: <code>Resource</code> - A resource for the new transaction.

Param	Type	Description
ns	<code>String</code>	the namespace of the transaction.
type	<code>String</code>	the type of the transaction.
[id]	<code>String</code>	an optional string identifier

```
| [options] | <code>Object</code> | an optional set of options |
| [options.generate] | <code>String</code> | Pass one of: <dl>
<dt>sample</dt><dd>return a resource instance with generated sample data.</dd>
<dt>empty</dt><dd>return a resource instance with empty property values.</dd></dl>
|
| [options.includeOptionalFields] | <code>boolean</code> | if
<code>options.generate</code> is specified, whether optional fields should be
generated. |

<a name="module_concerto-core.Factory+newEvent"></a>

#### factory.newEvent(ns, type, [id], [options]) ⇒ <code>Resource</code>
Create a new event object. The identifier of the event is
set to a UUID.

**Kind**: instance method of [<code>Factory</code>](#module_concerto-core.Factory)
**Returns**: <code>Resource</code> - A resource for the new event.

| Param | Type | Description |
| --- | --- | --- |
| ns | <code>String</code> | the namespace of the event. |
| type | <code>String</code> | the type of the event. |
| [id] | <code>String</code> | an optional string identifier |
| [options] | <code>Object</code> | an optional set of options |
| [options.generate] | <code>String</code> | Pass one of: <dl>
<dt>sample</dt><dd>return a resource instance with generated sample data.</dd>
<dt>empty</dt><dd>return a resource instance with empty property values.</dd></dl>
|
| [options.includeOptionalFields] | <code>boolean</code> | if
<code>options.generate</code> is specified, whether optional fields should be
generated. |

<a name="module_concerto-core.Factory.newId"></a>

#### Factory.newId() ⇒ <code>string</code>
Create a new ID for an object.

**Kind**: static method of [<code>Factory</code>](#module_concerto-core.Factory)
**Returns**: <code>string</code> - a new ID
<a name="module_concerto-core.AssetDeclaration"></a>

### concerto-core.AssetDeclaration ⇐ <code>ClassDeclaration</code>
AssetDeclaration defines the schema (aka model or class) for
an Asset. It extends ClassDeclaration which manages a set of
fields, a super-type and the specification of an
identifying field.

**Kind**: static class of [<code>concerto-core</code>](#module_concerto-core)
**Extends**: <code>ClassDeclaration</code>
**See**: See [ClassDeclaration](ClassDeclaration)

* [.AssetDeclaration](#module_concerto-core.AssetDeclaration) ⇐
<code>ClassDeclaration</code>
    * [new AssetDeclaration(modelFile, ast)](#new_module_concerto-
core.AssetDeclaration_new)
    * [.declarationKind()](#module_concerto-core.AssetDeclaration+declarationKind)
⇒ <code>string</code>

<a name="new_module_concerto-core.AssetDeclaration_new"></a>
```

```
#### new AssetDeclaration(modelFile, ast)
Create an AssetDeclaration.

**Throws**:

- <code>IllegalModelException</code>


| Param | Type | Description |
| --- | --- | --- |
| modelFile | <code>ModelFile</code> | the ModelFile for this class |
| ast | <code>Object</code> | The AST created by the parser |

<a name="module_concerto-core.AssetDeclaration+declarationKind"></a>

#### assetDeclaration.declarationKind() ⇒ <code>string</code>
Returns the kind of declaration

**Kind**: instance method of [<code>AssetDeclaration</code>](#module_concerto-
core.AssetDeclaration)
**Returns**: <code>string</code> - what kind of declaration this is
<a name="module_concerto-core.ClassDeclaration"></a>

### *concerto-core.ClassDeclaration*
ClassDeclaration defines the structure (model/schema) of composite data.
It is composed of a set of Properties, may have an identifying field, and may
have a super-type.
A ClassDeclaration is conceptually owned by a ModelFile which
defines all the classes that are part of a namespace.

**Kind**: static abstract class of [<code>concerto-core</code>](#module_concerto-
core)

* *[.ClassDeclaration](#module_concerto-core.ClassDeclaration)*
    * *[new ClassDeclaration(modelFile, ast)](#new_module_concerto-
core.ClassDeclaration_new)*
    * *[._resolveSuperType()](#module_concerto-
core.ClassDeclaration+_resolveSuperType) ⇒ <code>ClassDeclaration</code>*
    * *[.validate()](#module_concerto-core.ClassDeclaration+validate)*
    * *[.isAbstract()](#module_concerto-core.ClassDeclaration+isAbstract) ⇒
<code>boolean</code>*
    * *[.getName()](#module_concerto-core.ClassDeclaration+getName) ⇒
<code>string</code>*
    * *[.getNamespace()](#module_concerto-core.ClassDeclaration+getNamespace) ⇒
<code>string</code>*
    * *[.getFullyQualifiedName()](#module_concerto-
core.ClassDeclaration+getFullyQualifiedName) ⇒ <code>string</code>*
    * *[.isIdentified()](#module_concerto-core.ClassDeclaration+isIdentified) ⇒
<code>Boolean</code>*
    * *[.isSystemIdentified()](#module_concerto-
core.ClassDeclaration+isSystemIdentified) ⇒ <code>Boolean</code>*
    * *[.isExplicitlyIdentified()](#module_concerto-
core.ClassDeclaration+isExplicitlyIdentified) ⇒ <code>Boolean</code>*
    * *[.getIdentifierFieldName()](#module_concerto-
core.ClassDeclaration+getIdentifierFieldName) ⇒ <code>string</code>*
    * *[.getOwnProperty(name)](#module_concerto-
core.ClassDeclaration+getOwnProperty) ⇒ <code>Property</code>*
    * *[.getOwnProperties()](#module_concerto-
core.ClassDeclaration+getOwnProperties) ⇒ <code>Array.&lt;Property&gt;</code>*
    * *[.getSuperType()](#module_concerto-core.ClassDeclaration+getSuperType) ⇒
<code>string</code>*
    * *[.getSuperTypeDeclaration()](#module_concerto-
core.ClassDeclaration+getSuperTypeDeclaration) ⇒ <code>ClassDeclaration</code>*
    * *[.getAssignableClassDeclarations()](#module_concerto-
core.ClassDeclaration+getAssignableClassDeclarations) ⇒
<code>Array.&lt;ClassDeclaration&gt;</code>*
    * *[.getAllSuperTypeDeclarations()](#module_concerto-
core.ClassDeclaration+getAllSuperTypeDeclarations) ⇒
<code>Array.&lt;ClassDeclaration&gt;</code>*
    * *[.getProperty(name)](#module_concerto-core.ClassDeclaration+getProperty) ⇒
<code>Property</code>*
    * *[.getProperties()](#module_concerto-core.ClassDeclaration+getProperties) ⇒
<code>Array.&lt;Property&gt;</code>*
    * *[.getNestedProperty(propertyPath)](#module_concerto-
core.ClassDeclaration+getNestedProperty) ⇒ <code>Property</code>*
    * *[.toString()](#module_concerto-core.ClassDeclaration+toString) ⇒
<code>String</code>*
    * *[.isAsset()](#module_concerto-core.ClassDeclaration+isAsset) ⇒
<code>boolean</code>*
    * *[.isParticipant()](#module_concerto-core.ClassDeclaration+isParticipant) ⇒
<code>boolean</code>*
    * *[.isTransaction()](#module_concerto-core.ClassDeclaration+isTransaction) ⇒
<code>boolean</code>*
    * *[.isEvent()](#module_concerto-core.ClassDeclaration+isEvent) ⇒
<code>boolean</code>*
    * *[.isConcept()](#module_concerto-core.ClassDeclaration+isConcept) ⇒
<code>boolean</code>*
    * *[.isEnum()](#module_concerto-core.ClassDeclaration+isEnum) ⇒
<code>boolean</code>*
    * *[.isClassDeclaration()](#module_concerto-
core.ClassDeclaration+isClassDeclaration) ⇒ <code>boolean</code>*

<a name="new_module_concerto-core.ClassDeclaration_new"></a>

#### *new ClassDeclaration(modelFile, ast)*
Create a ClassDeclaration from an Abstract Syntax Tree. The AST is the
result of parsing.

**Throws**:

- <code>IllegalModelException</code>


| Param | Type | Description |
| --- | --- | --- |
| modelFile | <code>ModelFile</code> | the ModelFile for this class |
| ast | <code>Object</code> | the AST created by the parser |

<a name="module_concerto-core.ClassDeclaration+_resolveSuperType"></a>

#### *classDeclaration.\_resolveSuperType() ⇒ <code>ClassDeclaration</code>*
Resolve the super type on this class and store it as an internal property.

**Kind**: instance method of [<code>ClassDeclaration</code>](#module_concerto-
core.ClassDeclaration)
**Returns**: <code>ClassDeclaration</code> - The super type, or null if non
specified.
```

<a name="module_concerto-core.ClassDeclaration+validate"></a>

#### *classDeclaration.validate()*
Semantic validation of the structure of this class. Subclasses should
override this method to impose additional semantic constraints on the
contents/relations of fields.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Throws:

- <code>IllegalModelException</code>

Access: protected
<a name="module_concerto-core.ClassDeclaration+isAbstract"></a>

#### *classDeclaration.isAbstract() ⇒ <code>boolean</code>*
Returns true if this class is declared as abstract in the model file

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>boolean</code> - true if the class is abstract
<a name="module_concerto-core.ClassDeclaration+getName"></a>

#### *classDeclaration.getName() ⇒ <code>string</code>*
Returns the short name of a class. This name does not include the
namespace from the owning ModelFile.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>string</code> - the short name of this class
<a name="module_concerto-core.ClassDeclaration+getNamespace"></a>

#### *classDeclaration.getNamespace() ⇒ <code>string</code>*
Return the namespace of this class.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>string</code> - namespace - a namespace.
<a name="module_concerto-core.ClassDeclaration+getFullyQualifiedName"></a>

#### *classDeclaration.getFullyQualifiedName() ⇒ <code>string</code>*
Returns the fully qualified name of this class.
The name will include the namespace if present.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>string</code> - the fully-qualified name of this class
<a name="module_concerto-core.ClassDeclaration+isIdentified"></a>

#### *classDeclaration.isIdentified() ⇒ <code>Boolean</code>*
Returns true if this class declaration declares an identifying field
(system or explicit)

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>Boolean</code> - true if the class declaration includes an
identifier
<a name="module_concerto-core.ClassDeclaration+isSystemIdentified"></a>

#### *classDeclaration.isSystemIdentified() ⇒ <code>Boolean</code>*
Returns true if this class declaration declares a system identifier
$identifier

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>Boolean</code> - true if the class declaration includes a system identifier
<a name="module_concerto-core.ClassDeclaration+isExplicitlyIdentified"></a>

#### *classDeclaration.isExplicitlyIdentified() ⇒ <code>Boolean</code>*
Returns true if this class declaration declares an explicit identifier

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>Boolean</code> - true if the class declaration includes an explicit identifier
<a name="module_concerto-core.ClassDeclaration+getIdentifierFieldName"></a>

#### *classDeclaration.getIdentifierFieldName() ⇒ <code>string</code>*
Returns the name of the identifying field for this class. Note
that the identifying field may come from a super type.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>string</code> - the name of the id field for this class or null if it does not exist
<a name="module_concerto-core.ClassDeclaration+getOwnProperty"></a>

#### *classDeclaration.getOwnProperty(name) ⇒ <code>Property</code>*
Returns the field with a given name or null if it does not exist.
The field must be directly owned by this class -- the super-type is
not introspected.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>Property</code> - the field definition or null if it does not exist

Param	Type	Description
name	<code>string</code>	the name of the field

<a name="module_concerto-core.ClassDeclaration+getOwnProperties"></a>

#### *classDeclaration.getOwnProperties() ⇒ <code>Array.&lt;Property&gt;</code>*
Returns the fields directly defined by this class.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>Array.&lt;Property&gt;</code> - the array of fields
<a name="module_concerto-core.ClassDeclaration+getSuperType"></a>

#### *classDeclaration.getSuperType() ⇒ <code>string</code>*
Returns the FQN of the super type for this class or null if this
class does not have a super type.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-

core.ClassDeclaration)
Returns: <code>string</code> - the FQN name of the super type or null
<a name="module_concerto-core.ClassDeclaration+getSuperTypeDeclaration"></a>

#### *classDeclaration.getSuperTypeDeclaration() ⇒ <code>ClassDeclaration</code>*
Get the super type class declaration for this class.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>ClassDeclaration</code> - the super type declaration, or null if there is no super type.
<a name="module_concerto-core.ClassDeclaration+getAssignableClassDeclarations"></a>

#### *classDeclaration.getAssignableClassDeclarations() ⇒ <code>Array.&lt;ClassDeclaration&gt;</code>*
Get the class declarations for all subclasses of this class, including this class.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>Array.&lt;ClassDeclaration&gt;</code> - subclass declarations.
<a name="module_concerto-core.ClassDeclaration+getAllSuperTypeDeclarations"></a>

#### *classDeclaration.getAllSuperTypeDeclarations() ⇒ <code>Array.&lt;ClassDeclaration&gt;</code>*
Get all the super-type declarations for this type.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>Array.&lt;ClassDeclaration&gt;</code> - super-type declarations.
<a name="module_concerto-core.ClassDeclaration+getProperty"></a>

#### *classDeclaration.getProperty(name) ⇒ <code>Property</code>*
Returns the property with a given name or null if it does not exist.
Fields defined in super-types are also introspected.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>Property</code> - the field, or null if it does not exist

Param	Type	Description
name	<code>string</code>	the name of the field

<a name="module_concerto-core.ClassDeclaration+getProperties"></a>

#### *classDeclaration.getProperties() ⇒ <code>Array.&lt;Property&gt;</code>*
Returns the properties defined in this class and all super classes.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>Array.&lt;Property&gt;</code> - the array of fields
<a name="module_concerto-core.ClassDeclaration+getNestedProperty"></a>

#### *classDeclaration.getNestedProperty(propertyPath) ⇒ <code>Property</code>*
Get a nested property using a dotted property path

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>Property</code> - the property

Throws:

- <code>IllegalModelException</code> if the property path is invalid or the property does not exist

Param	Type	Description
propertyPath	<code>string</code>	The property name or name with nested structure e.g a.b.c

<a name="module_concerto-core.ClassDeclaration+toString"></a>

#### *classDeclaration.toString() ⇒ <code>String</code>*
Returns the string representation of this class

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>String</code> - the string representation of the class
<a name="module_concerto-core.ClassDeclaration+isAsset"></a>

#### *classDeclaration.isAsset() ⇒ <code>boolean</code>*
Returns true if this class is the definition of an asset.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>boolean</code> - true if the class is an asset
<a name="module_concerto-core.ClassDeclaration+isParticipant"></a>

#### *classDeclaration.isParticipant() ⇒ <code>boolean</code>*
Returns true if this class is the definition of a participant.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>boolean</code> - true if the class is an asset
<a name="module_concerto-core.ClassDeclaration+isTransaction"></a>

#### *classDeclaration.isTransaction() ⇒ <code>boolean</code>*
Returns true if this class is the definition of a transaction.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>boolean</code> - true if the class is an asset
<a name="module_concerto-core.ClassDeclaration+isEvent"></a>

#### *classDeclaration.isEvent() ⇒ <code>boolean</code>*
Returns true if this class is the definition of an event.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>boolean</code> - true if the class is an asset
<a name="module_concerto-core.ClassDeclaration+isConcept"></a>

#### *classDeclaration.isConcept() ⇒ <code>boolean</code>*
Returns true if this class is the definition of a concept.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>boolean</code> - true if the class is an asset

<a name="module_concerto-core.ClassDeclaration+isEnum"></a>

#### *classDeclaration.isEnum() ⇒ <code>boolean</code>*
Returns true if this class is the definition of a enum.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>boolean</code> - true if the class is an asset
<a name="module_concerto-core.ClassDeclaration+isClassDeclaration"></a>

#### *classDeclaration.isClassDeclaration() ⇒ <code>boolean</code>*
Returns true if this class is the definition of a enum.

Kind: instance method of [<code>ClassDeclaration</code>](#module_concerto-core.ClassDeclaration)
Returns: <code>boolean</code> - true if the class is an asset
<a name="module_concerto-core.ConceptDeclaration"></a>

### concerto-core.ConceptDeclaration ⇐ <code>ClassDeclaration</code>
ConceptDeclaration defines the schema (aka model or class) for
an Concept. It extends ClassDeclaration which manages a set of
fields, a super-type and the specification of an
identifying field.

Kind: static class of [<code>concerto-core</code>](#module_concerto-core)
Extends: <code>ClassDeclaration</code>
See: ClassDeclaration

* [.ConceptDeclaration](#module_concerto-core.ConceptDeclaration) ⇐ <code>ClassDeclaration</code>
    * [new ConceptDeclaration(modelFile, ast)](#new_module_concerto-core.ConceptDeclaration_new)
    * [.declarationKind()](#module_concerto-core.ConceptDeclaration+declarationKind) ⇒ <code>string</code>

<a name="new_module_concerto-core.ConceptDeclaration_new"></a>

#### new ConceptDeclaration(modelFile, ast)
Create a ConceptDeclaration.

Throws:

- <code>IllegalModelException</code>

Param	Type	Description
modelFile	<code>ModelFile</code>	the ModelFile for this class
ast	<code>Object</code>	The AST created by the parser

<a name="module_concerto-core.ConceptDeclaration+declarationKind"></a>

#### conceptDeclaration.declarationKind() ⇒ <code>string</code>
Returns the kind of declaration

Kind: instance method of [<code>ConceptDeclaration</code>](#module_concerto-core.ConceptDeclaration)
Returns: <code>string</code> - what kind of declaration this is
<a name="module_concerto-core.Decorator"></a>

```
### concerto-core.Decorator
Decorator encapsulates a decorator (annotation) on a class or property.

**Kind**: static class of [<code>concerto-core</code>](#module_concerto-core)

* [.Decorator](#module_concerto-core.Decorator)
    * [new Decorator(parent, ast)](#new_module_concerto-core.Decorator_new)
    * [.getParent()](#module_concerto-core.Decorator+getParent) ⇒
<code>ClassDeclaration</code> \| <code>Property</code>
    * [.getName()](#module_concerto-core.Decorator+getName) ⇒ <code>string</code>
    * [.getArguments()](#module_concerto-core.Decorator+getArguments) ⇒
<code>Array.&lt;object&gt;</code>

<a name="new_module_concerto-core.Decorator_new"></a>

#### new Decorator(parent, ast)
Create a Decorator.

**Throws**:

- <code>IllegalModelException</code>


| Param | Type | Description |
| --- | --- | --- |
| parent | <code>ClassDeclaration</code> \| <code>Property</code> | the owner of
this property |
| ast | <code>Object</code> | The AST created by the parser |

<a name="module_concerto-core.Decorator+getParent"></a>

#### decorator.getParent() ⇒ <code>ClassDeclaration</code> \| <code>Property</code>
Returns the owner of this property

**Kind**: instance method of [<code>Decorator</code>](#module_concerto-
core.Decorator)
**Returns**: <code>ClassDeclaration</code> \| <code>Property</code> - the parent
class or property declaration
<a name="module_concerto-core.Decorator+getName"></a>

#### decorator.getName() ⇒ <code>string</code>
Returns the name of a decorator

**Kind**: instance method of [<code>Decorator</code>](#module_concerto-
core.Decorator)
**Returns**: <code>string</code> - the name of this decorator
<a name="module_concerto-core.Decorator+getArguments"></a>

#### decorator.getArguments() ⇒ <code>Array.&lt;object&gt;</code>
Returns the arguments for this decorator

**Kind**: instance method of [<code>Decorator</code>](#module_concerto-
core.Decorator)
**Returns**: <code>Array.&lt;object&gt;</code> - the arguments for this decorator
<a name="module_concerto-core.DecoratorFactory"></a>

### concerto-core.DecoratorFactory
An interface for a class that processes a decorator and returns a specific
```

implementation class for that decorator.

Kind: static class of [<code>concerto-core</code>](#module_concerto-core)
<a name="module_concerto-core.DecoratorFactory+newDecorator"></a>

#### *decoratorFactory.newDecorator(parent, ast) ⇒ <code>Decorator</code>*
Process the decorator, and return a specific implementation class for that
decorator, or return null if this decorator is not handled by this processor.

Kind: instance abstract method of [<code>DecoratorFactory</code>]
(#module_concerto-core.DecoratorFactory)
Returns: <code>Decorator</code> - The decorator.

Param	Type	Description
parent	<code>ClassDeclaration</code> | <code>Property</code>	the owner of
this property
ast	<code>Object</code>	The AST created by the parser

<a name="module_concerto-core.EnumDeclaration"></a>

### concerto-core.EnumDeclaration ⇐ <code>ClassDeclaration</code>
EnumDeclaration defines an enumeration of static values.

Kind: static class of [<code>concerto-core</code>](#module_concerto-core)
Extends: <code>ClassDeclaration</code>
See: See ClassDeclaration

* [.EnumDeclaration](#module_concerto-core.EnumDeclaration) ⇐
<code>ClassDeclaration</code>
    * [new EnumDeclaration(modelFile, ast)](#new_module_concerto-
core.EnumDeclaration_new)
    * [.toString()](#module_concerto-core.EnumDeclaration+toString) ⇒
<code>String</code>
    * [.declarationKind()](#module_concerto-core.EnumDeclaration+declarationKind) ⇒
<code>string</code>

<a name="new_module_concerto-core.EnumDeclaration_new"></a>

#### new EnumDeclaration(modelFile, ast)
Create an EnumDeclaration.

Throws:

- <code>IllegalModelException</code>

Param	Type	Description
modelFile	<code>ModelFile</code>	the ModelFile for this class
ast	<code>Object</code>	The AST created by the parser

<a name="module_concerto-core.EnumDeclaration+toString"></a>

#### enumDeclaration.toString() ⇒ <code>String</code>
Returns the string representation of this class

Kind: instance method of [<code>EnumDeclaration</code>](#module_concerto-
core.EnumDeclaration)

```
**Returns**: <code>String</code> - the string representation of the class
<a name="module_concerto-core.EnumDeclaration+declarationKind"></a>

#### enumDeclaration.declarationKind() ⇒ <code>string</code>
Returns the kind of declaration

**Kind**: instance method of [<code>EnumDeclaration</code>](#module_concerto-
core.EnumDeclaration)
**Returns**: <code>string</code> - what kind of declaration this is
<a name="module_concerto-core.EnumValueDeclaration"></a>

### concerto-core.EnumValueDeclaration ⇐ <code>Property</code>
Class representing a value from a set of enumerated values

**Kind**: static class of [<code>concerto-core</code>](#module_concerto-core)
**Extends**: <code>Property</code>
**See**: See [Property](Property)

* [.EnumValueDeclaration](#module_concerto-core.EnumValueDeclaration) ⇐
<code>Property</code>
    * [new EnumValueDeclaration(parent, ast)](#new_module_concerto-
core.EnumValueDeclaration_new)
    * [.isEnumValue()](#module_concerto-core.EnumValueDeclaration+isEnumValue) ⇒
<code>boolean</code>

<a name="new_module_concerto-core.EnumValueDeclaration_new"></a>

#### new EnumValueDeclaration(parent, ast)
Create a EnumValueDeclaration.

**Throws**:

- <code>IllegalModelException</code>


| Param | Type | Description |
| --- | --- | --- |
| parent | <code>ClassDeclaration</code> | The owner of this property |
| ast | <code>Object</code> | The AST created by the parser |

<a name="module_concerto-core.EnumValueDeclaration+isEnumValue"></a>

#### enumValueDeclaration.isEnumValue() ⇒ <code>boolean</code>
Returns true if this class is the definition of a enum value.

**Kind**: instance method of [<code>EnumValueDeclaration</code>](#module_concerto-
core.EnumValueDeclaration)
**Returns**: <code>boolean</code> - true if the class is an enum value
<a name="module_concerto-core.EventDeclaration"></a>

### concerto-core.EventDeclaration ⇐ <code>ClassDeclaration</code>
Class representing the definition of an Event.

**Kind**: static class of [<code>concerto-core</code>](#module_concerto-core)
**Extends**: <code>ClassDeclaration</code>
**See**: See  [ClassDeclaration](ClassDeclaration)

* [.EventDeclaration](#module_concerto-core.EventDeclaration) ⇐
<code>ClassDeclaration</code>
```

```
    * [new EventDeclaration(modelFile, ast)](#new_module_concerto-
core.EventDeclaration_new)
    * [.declarationKind()](#module_concerto-core.EventDeclaration+declarationKind)
⇒ <code>string</code>

<a name="new_module_concerto-core.EventDeclaration_new"></a>

#### new EventDeclaration(modelFile, ast)
Create an EventDeclaration.

**Throws**:

- <code>IllegalModelException</code>


| Param | Type | Description |
| --- | --- | --- |
| modelFile | <code>ModelFile</code> | the ModelFile for this class |
| ast | <code>Object</code> | The AST created by the parser |

<a name="module_concerto-core.EventDeclaration+declarationKind"></a>

#### eventDeclaration.declarationKind() ⇒ <code>string</code>
Returns the kind of declaration

**Kind**: instance method of [<code>EventDeclaration</code>](#module_concerto-
core.EventDeclaration)
**Returns**: <code>string</code> - what kind of declaration this is
<a name="module_concerto-core.IdentifiedDeclaration"></a>

### *concerto-core.IdentifiedDeclaration ⇐ <code>ClassDeclaration</code>*
IdentifiedDeclaration

**Kind**: static abstract class of [<code>concerto-core</code>](#module_concerto-
core)
**Extends**: <code>ClassDeclaration</code>
**See**: See [ClassDeclaration](ClassDeclaration)
<a name="new_module_concerto-core.IdentifiedDeclaration_new"></a>

#### *new IdentifiedDeclaration(modelFile, ast)*
Create an IdentifiedDeclaration.

**Throws**:

- <code>IllegalModelException</code>


| Param | Type | Description |
| --- | --- | --- |
| modelFile | <code>ModelFile</code> | the ModelFile for this class |
| ast | <code>Object</code> | The AST created by the parser |

<a name="module_concerto-core.IllegalModelException"></a>

### concerto-core.IllegalModelException ⇐ <code>BaseFileException</code>
Exception throws when a composer file is semantically invalid

**Kind**: static class of [<code>concerto-core</code>](#module_concerto-core)
**Extends**: <code>BaseFileException</code>
```

```
**See**: See  [BaseFileException](BaseFileException)
<a name="new_module_concerto-core.IllegalModelException_new"></a>

#### new IllegalModelException(message, [modelFile], [fileLocation], [component])
Create an IllegalModelException.


| Param | Type | Description |
| --- | --- | --- |
| message | <code>string</code> | the message for the exception |
| [modelFile] | <code>ModelFile</code> | the modelfile associated with the
exception |
| [fileLocation] | <code>Object</code> | location details of the error within the
model file. |
| fileLocation.start.line | <code>number</code> | start line of the error location.
|
| fileLocation.start.column | <code>number</code> | start column of the error
location. |
| fileLocation.end.line | <code>number</code> | end line of the error location. |
| fileLocation.end.column | <code>number</code> | end column of the error location.
|
| [component] | <code>string</code> | the component which throws this error |

<a name="module_concerto-core.Introspector"></a>

### concerto-core.Introspector
Provides access to the structure of transactions, assets and participants.

**Kind**: static class of [<code>concerto-core</code>](#module_concerto-core)

* [.Introspector](#module_concerto-core.Introspector)
    * [new Introspector(modelManager)](#new_module_concerto-core.Introspector_new)
    * [.getClassDeclarations()](#module_concerto-
core.Introspector+getClassDeclarations) ⇒
<code>Array.&lt;ClassDeclaration&gt;</code>
    * [.getClassDeclaration(fullyQualifiedTypeName)](#module_concerto-
core.Introspector+getClassDeclaration) ⇒ <code>ClassDeclaration</code>

<a name="new_module_concerto-core.Introspector_new"></a>

#### new Introspector(modelManager)
Create the Introspector.


| Param | Type | Description |
| --- | --- | --- |
| modelManager | <code>ModelManager</code> | the ModelManager that backs this
Introspector |

<a name="module_concerto-core.Introspector+getClassDeclarations"></a>

#### introspector.getClassDeclarations() ⇒
<code>Array.&lt;ClassDeclaration&gt;</code>
Returns all the class declarations for the business network.

**Kind**: instance method of [<code>Introspector</code>](#module_concerto-
core.Introspector)
**Returns**: <code>Array.&lt;ClassDeclaration&gt;</code> - the array of class
declarations
```

```
<a name="module_concerto-core.Introspector+getClassDeclaration"></a>

#### introspector.getClassDeclaration(fullyQualifiedTypeName) ⇒
<code>ClassDeclaration</code>
Returns the class declaration with the given fully qualified name.
Throws an error if the class declaration does not exist.

**Kind**: instance method of [<code>Introspector</code>](#module_concerto-
core.Introspector)
**Returns**: <code>ClassDeclaration</code> - the class declaration
**Throws**:

- <code>Error</code> if the class declaration does not exist


| Param | Type | Description |
| --- | --- | --- |
| fullyQualifiedTypeName | <code>String</code> | the fully qualified name of the
type |

<a name="module_concerto-core.ModelFile"></a>

### concerto-core.ModelFile
Class representing a Model File. A Model File contains a single namespace
and a set of model elements: assets, transactions etc.

**Kind**: static class of [<code>concerto-core</code>](#module_concerto-core)

* [.ModelFile](#module_concerto-core.ModelFile)
    * [new ModelFile(modelManager, ast, [definitions], [fileName])]
(#new_module_concerto-core.ModelFile_new)
    * [.isModelFile()](#module_concerto-core.ModelFile+isModelFile) ⇒
<code>boolean</code>
    * [.isSystemModelFile()](#module_concerto-core.ModelFile+isSystemModelFile) ⇒
<code>Boolean</code>
    * [.isExternal()](#module_concerto-core.ModelFile+isExternal) ⇒
<code>boolean</code>
    * [.getModelManager()](#module_concerto-core.ModelFile+getModelManager) ⇒
<code>ModelManager</code>
    * [.getImports()](#module_concerto-core.ModelFile+getImports) ⇒
<code>Array.&lt;string&gt;</code>
    * [.isDefined(type)](#module_concerto-core.ModelFile+isDefined) ⇒
<code>boolean</code>
    * [.getLocalType(type)](#module_concerto-core.ModelFile+getLocalType) ⇒
<code>ClassDeclaration</code>
    * [.getAssetDeclaration(name)](#module_concerto-
core.ModelFile+getAssetDeclaration) ⇒ <code>AssetDeclaration</code>
    * [.getTransactionDeclaration(name)](#module_concerto-
core.ModelFile+getTransactionDeclaration) ⇒ <code>TransactionDeclaration</code>
    * [.getEventDeclaration(name)](#module_concerto-
core.ModelFile+getEventDeclaration) ⇒ <code>EventDeclaration</code>
    * [.getParticipantDeclaration(name)](#module_concerto-
core.ModelFile+getParticipantDeclaration) ⇒ <code>ParticipantDeclaration</code>
    * [.getNamespace()](#module_concerto-core.ModelFile+getNamespace) ⇒
<code>string</code>
    * [.getName()](#module_concerto-core.ModelFile+getName) ⇒ <code>string</code>
    * [.getAssetDeclarations()](#module_concerto-
core.ModelFile+getAssetDeclarations) ⇒ <code>Array.&lt;AssetDeclaration&gt;</code>
    * [.getTransactionDeclarations()](#module_concerto-
core.ModelFile+getTransactionDeclarations) ⇒
<code>Array.&lt;TransactionDeclaration&gt;</code>
    * [.getEventDeclarations()](#module_concerto-
core.ModelFile+getEventDeclarations) ⇒ <code>Array.&lt;EventDeclaration&gt;</code>
    * [.getParticipantDeclarations()](#module_concerto-
core.ModelFile+getParticipantDeclarations) ⇒
<code>Array.&lt;ParticipantDeclaration&gt;</code>
    * [.getConceptDeclarations()](#module_concerto-
core.ModelFile+getConceptDeclarations) ⇒
<code>Array.&lt;ConceptDeclaration&gt;</code>
    * [.getEnumDeclarations()](#module_concerto-core.ModelFile+getEnumDeclarations)
⇒ <code>Array.&lt;EnumDeclaration&gt;</code>
    * [.getDeclarations(type)](#module_concerto-core.ModelFile+getDeclarations) ⇒
<code>Array.&lt;ClassDeclaration&gt;</code>
    * [.getAllDeclarations()](#module_concerto-core.ModelFile+getAllDeclarations) ⇒
<code>Array.&lt;ClassDeclaration&gt;</code>
    * [.getDefinitions()](#module_concerto-core.ModelFile+getDefinitions) ⇒
<code>string</code>
    * [.getAst()](#module_concerto-core.ModelFile+getAst) ⇒ <code>object</code>
    * [.getConcertoVersion()](#module_concerto-core.ModelFile+getConcertoVersion) ⇒
<code>string</code>
    * [.isCompatibleVersion()](#module_concerto-core.ModelFile+isCompatibleVersion)

<a name="new_module_concerto-core.ModelFile_new"></a>

#### new ModelFile(modelManager, ast, [definitions], [fileName])
Create a ModelFile. This should only be called by framework code.
Use the ModelManager to manage ModelFiles.

**Throws**:

- <code>IllegalModelException</code>


| Param | Type | Description |
| --- | --- | --- |
| modelManager | <code>ModelManager</code> | the ModelManager that manages this
ModelFile |
| ast | <code>object</code> | The abstract syntax tree of the model as a JSON
object. |
| [definitions] | <code>string</code> | The optional CTO model as a string. |
| [fileName] | <code>string</code> | The optional filename for this modelfile |

<a name="module_concerto-core.ModelFile+isModelFile"></a>

#### modelFile.isModelFile() ⇒ <code>boolean</code>
Returns true

**Kind**: instance method of [<code>ModelFile</code>](#module_concerto-
core.ModelFile)
**Returns**: <code>boolean</code> - true
<a name="module_concerto-core.ModelFile+isSystemModelFile"></a>

#### modelFile.isSystemModelFile() ⇒ <code>Boolean</code>
Returns true if the ModelFile is a system namespace

**Kind**: instance method of [<code>ModelFile</code>](#module_concerto-
core.ModelFile)
**Returns**: <code>Boolean</code> - true if this is a system model file
```

```
<a name="module_concerto-core.ModelFile+isExternal"></a>

#### modelFile.isExternal() ⇒ <code>boolean</code>
Returns true if this ModelFile was downloaded from an external URI.

**Kind**: instance method of [<code>ModelFile</code>](#module_concerto-
core.ModelFile)
**Returns**: <code>boolean</code> - true iff this ModelFile was downloaded from an
external URI
<a name="module_concerto-core.ModelFile+getModelManager"></a>

#### modelFile.getModelManager() ⇒ <code>ModelManager</code>
Returns the ModelManager associated with this ModelFile

**Kind**: instance method of [<code>ModelFile</code>](#module_concerto-
core.ModelFile)
**Returns**: <code>ModelManager</code> - The ModelManager for this ModelFile
<a name="module_concerto-core.ModelFile+getImports"></a>

#### modelFile.getImports() ⇒ <code>Array.&lt;string&gt;</code>
Returns the types that have been imported into this ModelFile.

**Kind**: instance method of [<code>ModelFile</code>](#module_concerto-
core.ModelFile)
**Returns**: <code>Array.&lt;string&gt;</code> - The array of imports for this
ModelFile
<a name="module_concerto-core.ModelFile+isDefined"></a>

#### modelFile.isDefined(type) ⇒ <code>boolean</code>
Returns true if the type is defined in the model file

**Kind**: instance method of [<code>ModelFile</code>](#module_concerto-
core.ModelFile)
**Returns**: <code>boolean</code> - true if the type (asset or transaction) is
defined

| Param | Type | Description |
| --- | --- | --- |
| type | <code>string</code> | the name of the type |

<a name="module_concerto-core.ModelFile+getLocalType"></a>

#### modelFile.getLocalType(type) ⇒ <code>ClassDeclaration</code>
Returns the type with the specified name or null

**Kind**: instance method of [<code>ModelFile</code>](#module_concerto-
core.ModelFile)
**Returns**: <code>ClassDeclaration</code> - the ClassDeclaration, or null if the
type does not exist

| Param | Type | Description |
| --- | --- | --- |
| type | <code>string</code> | the short OR FQN name of the type |

<a name="module_concerto-core.ModelFile+getAssetDeclaration"></a>

#### modelFile.getAssetDeclaration(name) ⇒ <code>AssetDeclaration</code>
Get the AssetDeclarations defined in this ModelFile or null
```

Kind: instance method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)
Returns: <code>AssetDeclaration</code> - the AssetDeclaration with the given short name

Param	Type	Description
name	<code>string</code>	the name of the type

<a name="module_concerto-core.ModelFile+getTransactionDeclaration"></a>

#### modelFile.getTransactionDeclaration(name) ⇒ <code>TransactionDeclaration</code>
Get the TransactionDeclaration defined in this ModelFile or null

Kind: instance method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)
Returns: <code>TransactionDeclaration</code> - the TransactionDeclaration with the given short name

Param	Type	Description
name	<code>string</code>	the name of the type

<a name="module_concerto-core.ModelFile+getEventDeclaration"></a>

#### modelFile.getEventDeclaration(name) ⇒ <code>EventDeclaration</code>
Get the EventDeclaration defined in this ModelFile or null

Kind: instance method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)
Returns: <code>EventDeclaration</code> - the EventDeclaration with the given short name

Param	Type	Description
name	<code>string</code>	the name of the type

<a name="module_concerto-core.ModelFile+getParticipantDeclaration"></a>

#### modelFile.getParticipantDeclaration(name) ⇒ <code>ParticipantDeclaration</code>
Get the ParticipantDeclaration defined in this ModelFile or null

Kind: instance method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)
Returns: <code>ParticipantDeclaration</code> - the ParticipantDeclaration with the given short name

Param	Type	Description
name	<code>string</code>	the name of the type

<a name="module_concerto-core.ModelFile+getNamespace"></a>

#### modelFile.getNamespace() ⇒ <code>string</code>
Get the Namespace for this model file.

Kind: instance method of [<code>ModelFile</code>](#module_concerto-

```
core.ModelFile)
**Returns**: <code>string</code> - The Namespace for this model file
<a name="module_concerto-core.ModelFile+getName"></a>

#### modelFile.getName() ⇒ <code>string</code>
Get the filename for this model file. Note that this may be null.

**Kind**: instance method of [<code>ModelFile</code>](#module_concerto-
core.ModelFile)
**Returns**: <code>string</code> - The filename for this model file
<a name="module_concerto-core.ModelFile+getAssetDeclarations"></a>

#### modelFile.getAssetDeclarations() ⇒ <code>Array.&lt;AssetDeclaration&gt;</code>
Get the AssetDeclarations defined in this ModelFile

**Kind**: instance method of [<code>ModelFile</code>](#module_concerto-
core.ModelFile)
**Returns**: <code>Array.&lt;AssetDeclaration&gt;</code> - the AssetDeclarations
defined in the model file
<a name="module_concerto-core.ModelFile+getTransactionDeclarations"></a>

#### modelFile.getTransactionDeclarations() ⇒
<code>Array.&lt;TransactionDeclaration&gt;</code>
Get the TransactionDeclarations defined in this ModelFile

**Kind**: instance method of [<code>ModelFile</code>](#module_concerto-
core.ModelFile)
**Returns**: <code>Array.&lt;TransactionDeclaration&gt;</code> - the
TransactionDeclarations defined in the model file
<a name="module_concerto-core.ModelFile+getEventDeclarations"></a>

#### modelFile.getEventDeclarations() ⇒ <code>Array.&lt;EventDeclaration&gt;</code>
Get the EventDeclarations defined in this ModelFile

**Kind**: instance method of [<code>ModelFile</code>](#module_concerto-
core.ModelFile)
**Returns**: <code>Array.&lt;EventDeclaration&gt;</code> - the EventDeclarations
defined in the model file
<a name="module_concerto-core.ModelFile+getParticipantDeclarations"></a>

#### modelFile.getParticipantDeclarations() ⇒
<code>Array.&lt;ParticipantDeclaration&gt;</code>
Get the ParticipantDeclarations defined in this ModelFile

**Kind**: instance method of [<code>ModelFile</code>](#module_concerto-
core.ModelFile)
**Returns**: <code>Array.&lt;ParticipantDeclaration&gt;</code> - the
ParticipantDeclaration defined in the model file
<a name="module_concerto-core.ModelFile+getConceptDeclarations"></a>

#### modelFile.getConceptDeclarations() ⇒
<code>Array.&lt;ConceptDeclaration&gt;</code>
Get the ConceptDeclarations defined in this ModelFile

**Kind**: instance method of [<code>ModelFile</code>](#module_concerto-
core.ModelFile)
**Returns**: <code>Array.&lt;ConceptDeclaration&gt;</code> - the
ParticipantDeclaration defined in the model file
<a name="module_concerto-core.ModelFile+getEnumDeclarations"></a>
```

#### modelFile.getEnumDeclarations() ⇒ <code>Array.&lt;EnumDeclaration&gt;</code>
Get the EnumDeclarations defined in this ModelFile

Kind: instance method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)
Returns: <code>Array.&lt;EnumDeclaration&gt;</code> - the EnumDeclaration defined in the model file
<a name="module_concerto-core.ModelFile+getDeclarations"></a>

#### modelFile.getDeclarations(type) ⇒ <code>Array.&lt;ClassDeclaration&gt;</code>
Get the instances of a given type in this ModelFile

Kind: instance method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)
Returns: <code>Array.&lt;ClassDeclaration&gt;</code> - the ClassDeclaration defined in the model file

Param	Type	Description
type	<code>function</code>	the type of the declaration

<a name="module_concerto-core.ModelFile+getAllDeclarations"></a>

#### modelFile.getAllDeclarations() ⇒ <code>Array.&lt;ClassDeclaration&gt;</code>
Get all declarations in this ModelFile

Kind: instance method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)
Returns: <code>Array.&lt;ClassDeclaration&gt;</code> - the ClassDeclarations defined in the model file
<a name="module_concerto-core.ModelFile+getDefinitions"></a>

#### modelFile.getDefinitions() ⇒ <code>string</code>
Get the definitions for this model.

Kind: instance method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)
Returns: <code>string</code> - The definitions for this model.
<a name="module_concerto-core.ModelFile+getAst"></a>

#### modelFile.getAst() ⇒ <code>object</code>
Get the ast for this model.

Kind: instance method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)
Returns: <code>object</code> - The definitions for this model.
<a name="module_concerto-core.ModelFile+getConcertoVersion"></a>

#### modelFile.getConcertoVersion() ⇒ <code>string</code>
Get the expected concerto version

Kind: instance method of [<code>ModelFile</code>](#module_concerto-core.ModelFile)
Returns: <code>string</code> - The semver range for compatible concerto versions
<a name="module_concerto-core.ModelFile+isCompatibleVersion"></a>

#### modelFile.isCompatibleVersion()

```
Check whether this modelfile is compatible with the concerto version

**Kind**: instance method of [<code>ModelFile</code>](#module_concerto-
core.ModelFile)
<a name="module_concerto-core.ParticipantDeclaration"></a>

### concerto-core.ParticipantDeclaration ⇐ <code>ClassDeclaration</code>
Class representing the definition of a Participant.

**Kind**: static class of [<code>concerto-core</code>](#module_concerto-core)
**Extends**: <code>ClassDeclaration</code>
**See**: See  [ClassDeclaration](ClassDeclaration)

* [.ParticipantDeclaration](#module_concerto-core.ParticipantDeclaration) ⇐
<code>ClassDeclaration</code>
    * [new ParticipantDeclaration(modelFile, ast)](#new_module_concerto-
core.ParticipantDeclaration_new)
    * [.declarationKind()](#module_concerto-
core.ParticipantDeclaration+declarationKind) ⇒ <code>string</code>

<a name="new_module_concerto-core.ParticipantDeclaration_new"></a>

#### new ParticipantDeclaration(modelFile, ast)
Create an ParticipantDeclaration.

**Throws**:

- <code>IllegalModelException</code>


| Param | Type | Description |
| --- | --- | --- |
| modelFile | <code>ModelFile</code> | the ModelFile for this class |
| ast | <code>Object</code> | The AST created by the parser |

<a name="module_concerto-core.ParticipantDeclaration+declarationKind"></a>

#### participantDeclaration.declarationKind() ⇒ <code>string</code>
Returns the kind of declaration

**Kind**: instance method of [<code>ParticipantDeclaration</code>]
(#module_concerto-core.ParticipantDeclaration)
**Returns**: <code>string</code> - what kind of declaration this is
<a name="module_concerto-core.Property"></a>

### concerto-core.Property
Property representing an attribute of a class declaration,
either a Field or a Relationship.

**Kind**: static class of [<code>concerto-core</code>](#module_concerto-core)

* [.Property](#module_concerto-core.Property)
    * [new Property(parent, ast)](#new_module_concerto-core.Property_new)
    * [.getParent()](#module_concerto-core.Property+getParent) ⇒
<code>ClassDeclaration</code>
    * [.validate(classDecl)](#module_concerto-core.Property+validate)
    * [.getName()](#module_concerto-core.Property+getName) ⇒ <code>string</code>
    * [.getType()](#module_concerto-core.Property+getType) ⇒ <code>string</code>
    * [.isOptional()](#module_concerto-core.Property+isOptional) ⇒
```

```
<code>boolean</code>
    * [.getFullyQualifiedTypeName()](#module_concerto-
core.Property+getFullyQualifiedTypeName) ⇒ <code>string</code>
    * [.getFullyQualifiedName()](#module_concerto-
core.Property+getFullyQualifiedName) ⇒ <code>string</code>
    * [.getNamespace()](#module_concerto-core.Property+getNamespace) ⇒
<code>string</code>
    * [.isArray()](#module_concerto-core.Property+isArray) ⇒ <code>boolean</code>
    * [.isTypeEnum()](#module_concerto-core.Property+isTypeEnum) ⇒
<code>boolean</code>
    * [.isPrimitive()](#module_concerto-core.Property+isPrimitive) ⇒
<code>boolean</code>

<a name="new_module_concerto-core.Property_new"></a>

#### new Property(parent, ast)
Create a Property.

**Throws**:

- <code>IllegalModelException</code>


| Param | Type | Description |
| --- | --- | --- |
| parent | <code>ClassDeclaration</code> | the owner of this property |
| ast | <code>Object</code> | The AST created by the parser |

<a name="module_concerto-core.Property+getParent"></a>

#### property.getParent() ⇒ <code>ClassDeclaration</code>
Returns the owner of this property

**Kind**: instance method of [<code>Property</code>](#module_concerto-
core.Property)
**Returns**: <code>ClassDeclaration</code> - the parent class declaration
<a name="module_concerto-core.Property+validate"></a>

#### property.validate(classDecl)
Validate the property

**Kind**: instance method of [<code>Property</code>](#module_concerto-
core.Property)
**Throws**:

- <code>IllegalModelException</code>

**Access**: protected

| Param | Type | Description |
| --- | --- | --- |
| classDecl | <code>ClassDeclaration</code> | the class declaration of the property
|

<a name="module_concerto-core.Property+getName"></a>

#### property.getName() ⇒ <code>string</code>
Returns the name of a property
```

```
**Kind**: instance method of [<code>Property</code>](#module_concerto-
core.Property)
**Returns**: <code>string</code> - the name of this field
<a name="module_concerto-core.Property+getType"></a>

#### property.getType() ⇒ <code>string</code>
Returns the type of a property

**Kind**: instance method of [<code>Property</code>](#module_concerto-
core.Property)
**Returns**: <code>string</code> - the type of this field
<a name="module_concerto-core.Property+isOptional"></a>

#### property.isOptional() ⇒ <code>boolean</code>
Returns true if the field is optional

**Kind**: instance method of [<code>Property</code>](#module_concerto-
core.Property)
**Returns**: <code>boolean</code> - true if the field is optional
<a name="module_concerto-core.Property+getFullyQualifiedTypeName"></a>

#### property.getFullyQualifiedTypeName() ⇒ <code>string</code>
Returns the fully qualified type name of a property

**Kind**: instance method of [<code>Property</code>](#module_concerto-
core.Property)
**Returns**: <code>string</code> - the fully qualified type of this property
<a name="module_concerto-core.Property+getFullyQualifiedName"></a>

#### property.getFullyQualifiedName() ⇒ <code>string</code>
Returns the fully name of a property (ns + class name + property name)

**Kind**: instance method of [<code>Property</code>](#module_concerto-
core.Property)
**Returns**: <code>string</code> - the fully qualified name of this property
<a name="module_concerto-core.Property+getNamespace"></a>

#### property.getNamespace() ⇒ <code>string</code>
Returns the namespace of the parent of this property

**Kind**: instance method of [<code>Property</code>](#module_concerto-
core.Property)
**Returns**: <code>string</code> - the namespace of the parent of this property
<a name="module_concerto-core.Property+isArray"></a>

#### property.isArray() ⇒ <code>boolean</code>
Returns true if the field is declared as an array type

**Kind**: instance method of [<code>Property</code>](#module_concerto-
core.Property)
**Returns**: <code>boolean</code> - true if the property is an array type
<a name="module_concerto-core.Property+isTypeEnum"></a>

#### property.isTypeEnum() ⇒ <code>boolean</code>
Returns true if the field is declared as an enumerated value

**Kind**: instance method of [<code>Property</code>](#module_concerto-
core.Property)
**Returns**: <code>boolean</code> - true if the property is an enumerated value
```

```
<a name="module_concerto-core.Property+isPrimitive"></a>

#### property.isPrimitive() ⇒ <code>boolean</code>
Returns true if this property is a primitive type.

**Kind**: instance method of [<code>Property</code>](#module_concerto-
core.Property)
**Returns**: <code>boolean</code> - true if the property is a primitive type.
<a name="module_concerto-core.RelationshipDeclaration"></a>

### concerto-core.RelationshipDeclaration ⇐ <code>Property</code>
Class representing a relationship between model elements

**Kind**: static class of [<code>concerto-core</code>](#module_concerto-core)
**Extends**: <code>Property</code>
**See**: See  [Property](Property)

* [.RelationshipDeclaration](#module_concerto-core.RelationshipDeclaration) ⇐
<code>Property</code>
    * [new RelationshipDeclaration(parent, ast)](#new_module_concerto-
core.RelationshipDeclaration_new)
    * [.validate(classDecl)](#module_concerto-
core.RelationshipDeclaration+validate)
    * [.toString()](#module_concerto-core.RelationshipDeclaration+toString) ⇒
<code>String</code>
    * [.isRelationship()](#module_concerto-
core.RelationshipDeclaration+isRelationship) ⇒ <code>boolean</code>

<a name="new_module_concerto-core.RelationshipDeclaration_new"></a>

#### new RelationshipDeclaration(parent, ast)
Create a Relationship.

**Throws**:

- <code>IllegalModelException</code>


| Param | Type | Description |
| --- | --- | --- |
| parent | <code>ClassDeclaration</code> | The owner of this property |
| ast | <code>Object</code> | The AST created by the parser |

<a name="module_concerto-core.RelationshipDeclaration+validate"></a>

#### relationshipDeclaration.validate(classDecl)
Validate the property

**Kind**: instance method of [<code>RelationshipDeclaration</code>]
(#module_concerto-core.RelationshipDeclaration)
**Throws**:

- <code>IllegalModelException</code>

**Access**: protected

| Param | Type | Description |
| --- | --- | --- |
| classDecl | <code>ClassDeclaration</code> | the class declaration of the property
```

```
|

<a name="module_concerto-core.RelationshipDeclaration+toString"></a>

#### relationshipDeclaration.toString() ⇒ <code>String</code>
Returns a string representation of this property

**Kind**: instance method of [<code>RelationshipDeclaration</code>]
(#module_concerto-core.RelationshipDeclaration)
**Returns**: <code>String</code> - the string version of the property.
<a name="module_concerto-core.RelationshipDeclaration+isRelationship"></a>

#### relationshipDeclaration.isRelationship() ⇒ <code>boolean</code>
Returns true if this class is the definition of a relationship.

**Kind**: instance method of [<code>RelationshipDeclaration</code>]
(#module_concerto-core.RelationshipDeclaration)
**Returns**: <code>boolean</code> - true if the class is a relationship
<a name="module_concerto-core.TransactionDeclaration"></a>

### concerto-core.TransactionDeclaration ⇐ <code>ClassDeclaration</code>
Class representing the definition of an Transaction.

**Kind**: static class of [<code>concerto-core</code>](#module_concerto-core)
**Extends**: <code>ClassDeclaration</code>
**See**: See  [ClassDeclaration](ClassDeclaration)

* [.TransactionDeclaration](#module_concerto-core.TransactionDeclaration) ⇐
<code>ClassDeclaration</code>
    * [new TransactionDeclaration(modelFile, ast)](#new_module_concerto-
core.TransactionDeclaration_new)
    * [.declarationKind()](#module_concerto-
core.TransactionDeclaration+declarationKind) ⇒ <code>string</code>

<a name="new_module_concerto-core.TransactionDeclaration_new"></a>

#### new TransactionDeclaration(modelFile, ast)
Create an TransactionDeclaration.

**Throws**:

- <code>IllegalModelException</code>


| Param | Type | Description |
| --- | --- | --- |
| modelFile | <code>ModelFile</code> | the ModelFile for this class |
| ast | <code>Object</code> | The AST created by the parser |

<a name="module_concerto-core.TransactionDeclaration+declarationKind"></a>

#### transactionDeclaration.declarationKind() ⇒ <code>string</code>
Returns the kind of declaration

**Kind**: instance method of [<code>TransactionDeclaration</code>]
(#module_concerto-core.TransactionDeclaration)
**Returns**: <code>string</code> - what kind of declaration this is
<a name="module_concerto-core.Identifiable"></a>
```

```
### *concerto-core.Identifiable ⇐ <code>Typed</code>*
Identifiable is an entity with a namespace, type and an identifier.
Applications should retrieve instances from [Factory](Factory)
This class is abstract.

**Kind**: static abstract class of [<code>concerto-core</code>](#module_concerto-core)
**Extends**: <code>Typed</code>
**Access**: protected

* *[.Identifiable](#module_concerto-core.Identifiable) ⇐ <code>Typed</code>*
    * *[new Identifiable(modelManager, classDeclaration, ns, type, id, timestamp)](#new_module_concerto-core.Identifiable_new)*
    * *[.getTimestamp()](#module_concerto-core.Identifiable+getTimestamp) ⇒ <code>string</code>*
    * *[.getIdentifier()](#module_concerto-core.Identifiable+getIdentifier) ⇒ <code>string</code>*
    * *[.setIdentifier(id)](#module_concerto-core.Identifiable+setIdentifier)*
    * *[.getFullyQualifiedIdentifier()](#module_concerto-core.Identifiable+getFullyQualifiedIdentifier) ⇒ <code>string</code>*
    * *[.toString()](#module_concerto-core.Identifiable+toString) ⇒ <code>String</code>*
    * *[.isRelationship()](#module_concerto-core.Identifiable+isRelationship) ⇒ <code>boolean</code>*
    * *[.isResource()](#module_concerto-core.Identifiable+isResource) ⇒ <code>boolean</code>*
    * *[.toURI()](#module_concerto-core.Identifiable+toURI) ⇒ <code>String</code>*

<a name="new_module_concerto-core.Identifiable_new"></a>

#### *new Identifiable(modelManager, classDeclaration, ns, type, id, timestamp)*
Create an instance.
<p>
<strong>Note: Only to be called by framework code. Applications should
retrieve instances from [Factory](Factory)</strong>
</p>


| Param | Type | Description |
| --- | --- | --- |
| modelManager | <code>ModelManager</code> | The ModelManager for this instance |
| classDeclaration | <code>ClassDeclaration</code> | The class declaration for this instance. |
| ns | <code>string</code> | The namespace this instance. |
| type | <code>string</code> | The type this instance. |
| id | <code>string</code> | The identifier of this instance. |
| timestamp | <code>string</code> | The timestamp of this instance |

<a name="module_concerto-core.Identifiable+getTimestamp"></a>

#### *identifiable.getTimestamp() ⇒ <code>string</code>*
Get the timestamp of this instance

**Kind**: instance method of [<code>Identifiable</code>](#module_concerto-core.Identifiable)
**Returns**: <code>string</code> - The timestamp for this object
<a name="module_concerto-core.Identifiable+getIdentifier"></a>

#### *identifiable.getIdentifier() ⇒ <code>string</code>*
```

Get the identifier of this instance

Kind: instance method of [<code>Identifiable</code>](#module_concerto-
core.Identifiable)
Returns: <code>string</code> - The identifier for this object
<a name="module_concerto-core.Identifiable+setIdentifier"></a>

#### *identifiable.setIdentifier(id)*
Set the identifier of this instance

Kind: instance method of [<code>Identifiable</code>](#module_concerto-
core.Identifiable)

Param	Type	Description
id	<code>string</code>	the new identifier for this object

<a name="module_concerto-core.Identifiable+getFullyQualifiedIdentifier"></a>

#### *identifiable.getFullyQualifiedIdentifier() ⇒ <code>string</code>*
Get the fully qualified identifier of this instance.
(namespace '.' type '#' identifier).

Kind: instance method of [<code>Identifiable</code>](#module_concerto-
core.Identifiable)
Returns: <code>string</code> - the fully qualified identifier of this instance
<a name="module_concerto-core.Identifiable+toString"></a>

#### *identifiable.toString() ⇒ <code>String</code>*
Returns the string representation of this class

Kind: instance method of [<code>Identifiable</code>](#module_concerto-
core.Identifiable)
Returns: <code>String</code> - the string representation of the class
<a name="module_concerto-core.Identifiable+isRelationship"></a>

#### *identifiable.isRelationship() ⇒ <code>boolean</code>*
Determine if this identifiable is a relationship.

Kind: instance method of [<code>Identifiable</code>](#module_concerto-
core.Identifiable)
Returns: <code>boolean</code> - True if this identifiable is a relationship,
false if not.
<a name="module_concerto-core.Identifiable+isResource"></a>

#### *identifiable.isResource() ⇒ <code>boolean</code>*
Determine if this identifiable is a resource.

Kind: instance method of [<code>Identifiable</code>](#module_concerto-
core.Identifiable)
Returns: <code>boolean</code> - True if this identifiable is a resource,
false if not.
<a name="module_concerto-core.Identifiable+toURI"></a>

#### *identifiable.toURI() ⇒ <code>String</code>*
Returns a URI representation of a reference to this identifiable

Kind: instance method of [<code>Identifiable</code>](#module_concerto-
core.Identifiable)

Returns: <code>String</code> - the URI for the identifiable
<a name="module_concerto-core.Resource"></a>

### concerto-core.Resource ⇐ <code>Identifiable</code>
Resource is an instance that has a type. The type of the resource
specifies a set of properites (which themselves have types).

Type information in Concerto is used to validate the structure of
Resource instances and for serialization.

Resources are used in Concerto to represent Assets, Participants, Transactions and
other domain classes that can be serialized for long-term persistent storage.

Kind: static class of [<code>concerto-core</code>](#module_concerto-core)
Extends: <code>Identifiable</code>
Access: public
See: See Resource

* [.Resource](#module_concerto-core.Resource) ⇐ <code>Identifiable</code>
    * [new Resource(modelManager, classDeclaration, ns, type, id, timestamp)](#new_module_concerto-core.Resource_new)
    * [.toString()](#module_concerto-core.Resource+toString) ⇒ <code>String</code>
    * [.isResource()](#module_concerto-core.Resource+isResource) ⇒ <code>boolean</code>
    * [.isConcept()](#module_concerto-core.Resource+isConcept) ⇒ <code>boolean</code>
    * [.isIdentifiable()](#module_concerto-core.Resource+isIdentifiable) ⇒ <code>boolean</code>
    * [.toJSON()](#module_concerto-core.Resource+toJSON) ⇒ <code>Object</code>

<a name="new_module_concerto-core.Resource_new"></a>

#### new Resource(modelManager, classDeclaration, ns, type, id, timestamp)
This constructor should not be called directly.
<p>
<strong>Note: Only to be called by framework code. Applications should
retrieve instances from Factory</strong>
</p>

Param	Type	Description
modelManager	<code>ModelManager</code>	The ModelManager for this instance
classDeclaration	<code>ClassDeclaration</code>	The class declaration for this instance.
ns	<code>string</code>	The namespace this instance.
type	<code>string</code>	The type this instance.
id	<code>string</code>	The identifier of this instance.
timestamp	<code>string</code>	The timestamp of this instance

<a name="module_concerto-core.Resource+toString"></a>

#### resource.toString() ⇒ <code>String</code>
Returns the string representation of this class

Kind: instance method of [<code>Resource</code>](#module_concerto-core.Resource)

Returns: <code>String</code> - the string representation of the class
<a name="module_concerto-core.Resource+isResource"></a>

#### resource.isResource() ⇒ <code>boolean</code>
Determine if this identifiable is a resource.

Kind: instance method of [<code>Resource</code>](#module_concerto-core.Resource)
Returns: <code>boolean</code> - True if this identifiable is a resource, false if not.
<a name="module_concerto-core.Resource+isConcept"></a>

#### resource.isConcept() ⇒ <code>boolean</code>
Determine if this identifiable is a concept.

Kind: instance method of [<code>Resource</code>](#module_concerto-core.Resource)
Returns: <code>boolean</code> - True if this identifiable is a concept, false if not.
<a name="module_concerto-core.Resource+isIdentifiable"></a>

#### resource.isIdentifiable() ⇒ <code>boolean</code>
Determine if this object is identifiable.

Kind: instance method of [<code>Resource</code>](#module_concerto-core.Resource)
Returns: <code>boolean</code> - True if this object has an identifiying field false if not.
<a name="module_concerto-core.Resource+toJSON"></a>

#### resource.toJSON() ⇒ <code>Object</code>
Serialize this resource into a JavaScript object suitable for serialization to JSON,
using the default options for the serializer. If you need to set additional options
for the serializer, use the Serializer#toJSON method instead.

Kind: instance method of [<code>Resource</code>](#module_concerto-core.Resource)
Returns: <code>Object</code> - A JavaScript object suitable for serialization to JSON.
<a name="module_concerto-core.Typed"></a>

### *concerto-core.Typed*
Object is an instance with a namespace and a type.

This class is abstract.

Kind: static abstract class of [<code>concerto-core</code>](#module_concerto-core)
Access: protected

* *[.Typed](#module_concerto-core.Typed)*
    * *[new Typed(modelManager, classDeclaration, ns, type)](#new_module_concerto-core.Typed_new)*
    * *[.getType()](#module_concerto-core.Typed+getType) ⇒ <code>string</code>*
    * *[.getFullyQualifiedType()](#module_concerto-core.Typed+getFullyQualifiedType) ⇒ <code>string</code>*
    * *[.getNamespace()](#module_concerto-core.Typed+getNamespace) ⇒ <code>string</code>*

```
    * *[.setPropertyValue(propName, value)](#module_concerto-
core.Typed+setPropertyValue)*
    * *[.addArrayValue(propName, value)](#module_concerto-
core.Typed+addArrayValue)*
    * *[.instanceOf(fqt)](#module_concerto-core.Typed+instanceOf) ⇒
<code>boolean</code>*
    * *[.toJSON()](#module_concerto-core.Typed+toJSON)*

<a name="new_module_concerto-core.Typed_new"></a>

#### *new Typed(modelManager, classDeclaration, ns, type)*
Create an instance.
<p>
<strong>Note: Only to be called by framework code. Applications should
retrieve instances from [Factory](Factory)</strong>
</p>


| Param | Type | Description |
| --- | --- | --- |
| modelManager | <code>ModelManager</code> | The ModelManager for this instance |
| classDeclaration | <code>ClassDeclaration</code> | The class declaration for this
instance. |
| ns | <code>string</code> | The namespace this instance. |
| type | <code>string</code> | The type this instance. |

<a name="module_concerto-core.Typed+getType"></a>

#### *typed.getType() ⇒ <code>string</code>*
Get the type of the instance (a short name, not including namespace).

**Kind**: instance method of [<code>Typed</code>](#module_concerto-core.Typed)
**Returns**: <code>string</code> - The type of this object
<a name="module_concerto-core.Typed+getFullyQualifiedType"></a>

#### *typed.getFullyQualifiedType() ⇒ <code>string</code>*
Get the fully-qualified type name of the instance (including namespace).

**Kind**: instance method of [<code>Typed</code>](#module_concerto-core.Typed)
**Returns**: <code>string</code> - The fully-qualified type name of this object
<a name="module_concerto-core.Typed+getNamespace"></a>

#### *typed.getNamespace() ⇒ <code>string</code>*
Get the namespace of the instance.

**Kind**: instance method of [<code>Typed</code>](#module_concerto-core.Typed)
**Returns**: <code>string</code> - The namespace of this object
<a name="module_concerto-core.Typed+setPropertyValue"></a>

#### *typed.setPropertyValue(propName, value)*
Sets a property on this Resource

**Kind**: instance method of [<code>Typed</code>](#module_concerto-core.Typed)

| Param | Type | Description |
| --- | --- | --- |
| propName | <code>string</code> | the name of the field |
| value | <code>string</code> | the value of the property |
```

<a name="module_concerto-core.Typed+addArrayValue"></a>

#### *typed.addArrayValue(propName, value)*
Adds a value to an array property on this Resource

Kind: instance method of [<code>Typed</code>](#module_concerto-core.Typed)

Param	Type	Description
propName	<code>string</code>	the name of the field
value	<code>string</code>	the value of the property

<a name="module_concerto-core.Typed+instanceOf"></a>

#### *typed.instanceOf(fqt) ⇒ <code>boolean</code>*
Check to see if this instance is an instance of the specified fully qualified
type name.

Kind: instance method of [<code>Typed</code>](#module_concerto-core.Typed)
Returns: <code>boolean</code> - True if this instance is an instance of the specified fully
qualified type name, false otherwise.

Param	Type	Description
fqt	<code>String</code>	The fully qualified type name.

<a name="module_concerto-core.Typed+toJSON"></a>

#### *typed.toJSON()*
Overriden to prevent people accidentally converting a resource to JSON
without using the Serializer.

Kind: instance method of [<code>Typed</code>](#module_concerto-core.Typed)
Access: protected
<a name="module_concerto-core.ModelLoader"></a>

### concerto-core.ModelLoader
Create a ModelManager from model files, with an optional system model.

If a ctoFile is not provided, the Accord Project system model is used.

Kind: static class of [<code>concerto-core</code>](#module_concerto-core)

* [.ModelLoader](#module_concerto-core.ModelLoader)
    * [.loadModelManager(ctoFiles, options)](#module_concerto-core.ModelLoader.loadModelManager) ⇒ <code>object</code>
    * [.loadModelManagerFromModelFiles(modelFiles, [fileNames], options)](#module_concerto-core.ModelLoader.loadModelManagerFromModelFiles) ⇒ <code>object</code>

<a name="module_concerto-core.ModelLoader.loadModelManager"></a>

#### ModelLoader.loadModelManager(ctoFiles, options) ⇒ <code>object</code>
Load models in a new model manager

Kind: static method of [<code>ModelLoader</code>](#module_concerto-core.ModelLoader)
Returns: <code>object</code> - the model manager

Param	Type	Description
ctoFiles	<code>Array.&lt;string&gt;</code>	the CTO files (can be local file paths or URLs)
options	<code>object</code>	optional parameters
[options.offline]	<code>boolean</code>	do not resolve external models
[options.utcOffset]	<code>number</code>	UTC Offset for this execution

<a name="module_concerto-core.ModelLoader.loadModelManagerFromModelFiles"></a>

#### ModelLoader.loadModelManagerFromModelFiles(modelFiles, [fileNames], options) ⇒ <code>object</code>
Load system and models in a new model manager from model files objects

Kind: static method of [<code>ModelLoader</code>](#module_concerto-core.ModelLoader)
Returns: <code>object</code> - the model manager

Param	Type	Description
modelFiles	<code>Array.&lt;object&gt;</code>	An array of Concerto files as strings or ModelFile objects.
[fileNames]	<code>Array.&lt;string&gt;</code>	An optional array of file names to associate with the model files
options	<code>object</code>	optional parameters
[options.offline]	<code>boolean</code>	do not resolve external models
[options.utcOffset]	<code>number</code>	UTC Offset for this execution

<a name="module_concerto-core.ModelManager"></a>

### concerto-core.ModelManager
Manages the Concerto model files in CTO format.

The structure of Resources (Assets, Transactions, Participants) is modelled
in a set of Concerto files. The contents of these files are managed
by the ModelManager. Each Concerto file has a single namespace and contains
a set of asset, transaction and participant type definitions.

Concerto applications load their Concerto files and then call the [addModelFile](ModelManager#addModelFile)
method to register the Concerto file(s) with the ModelManager.

Kind: static class of [<code>concerto-core</code>](#module_concerto-core)

* [.ModelManager](#module_concerto-core.ModelManager)
    * [new ModelManager([options])](#new_module_concerto-core.ModelManager_new)
    * [.addCTOModel(cto, [fileName], [disableValidation])](#module_concerto-core.ModelManager+addCTOModel) ⇒ <code>Object</code>

<a name="new_module_concerto-core.ModelManager_new"></a>

#### new ModelManager([options])
Create the ModelManager.

| Param | Type | Description |

```
| --- | --- | --- |
| [options] | <code>object</code> | Serializer options |

<a name="module_concerto-core.ModelManager+addCTOModel"></a>

#### modelManager.addCTOModel(cto, [fileName], [disableValidation]) ⇒
<code>Object</code>
Adds a model in CTO format to the ModelManager.
This is a convenience function equivalent to `addModel` but useful since it avoids
having to copy the input CTO.

**Kind**: instance method of [<code>ModelManager</code>](#module_concerto-
core.ModelManager)
**Returns**: <code>Object</code> - The newly added model file (internal).
**Throws**:

- <code>IllegalModelException</code>


| Param | Type | Description |
| --- | --- | --- |
| cto | <code>string</code> | a cto string |
| [fileName] | <code>string</code> | an optional file name to associate with the
model file |
| [disableValidation] | <code>boolean</code> | If true then the model files are not
validated |

<a name="module_concerto-core.SecurityException"></a>

### concerto-core.SecurityException ⇐ <code>BaseException</code>
Class representing a security exception

**Kind**: static class of [<code>concerto-core</code>](#module_concerto-core)
**Extends**: <code>BaseException</code>
**See**: See [BaseException](BaseException)
<a name="new_module_concerto-core.SecurityException_new"></a>

#### new SecurityException(message)
Create the SecurityException.


| Param | Type | Description |
| --- | --- | --- |
| message | <code>string</code> | The exception message. |

<a name="module_concerto-core.Serializer"></a>

### concerto-core.Serializer
Serialize Resources instances to/from various formats for long-term storage
(e.g. on the blockchain).

**Kind**: static class of [<code>concerto-core</code>](#module_concerto-core)

* [.Serializer](#module_concerto-core.Serializer)
    * [new Serializer(factory, modelManager, [options])](#new_module_concerto-
core.Serializer_new)
    * [.setDefaultOptions(newDefaultOptions)](#module_concerto-
core.Serializer+setDefaultOptions)
    * [.toJSON(resource, [options])](#module_concerto-core.Serializer+toJSON) ⇒
```

```
<code>Object</code>
    * [.fromJSON(jsonObject, [options])](#module_concerto-core.Serializer+fromJSON)
⇒ <code>Resource</code>

<a name="new_module_concerto-core.Serializer_new"></a>

#### new Serializer(factory, modelManager, [options])
Create a Serializer.


| Param | Type | Description |
| --- | --- | --- |
| factory | <code>Factory</code> | The Factory to use to create instances |
| modelManager | <code>ModelManager</code> | The ModelManager to use for validation
etc. |
| [options] | <code>object</code> | Serializer options |

<a name="module_concerto-core.Serializer+setDefaultOptions"></a>

#### serializer.setDefaultOptions(newDefaultOptions)
Set the default options for the serializer.

**Kind**: instance method of [<code>Serializer</code>](#module_concerto-
core.Serializer)

| Param | Type | Description |
| --- | --- | --- |
| newDefaultOptions | <code>Object</code> | The new default options for the
serializer. |

<a name="module_concerto-core.Serializer+toJSON"></a>

#### serializer.toJSON(resource, [options]) ⇒ <code>Object</code>
<p>
Convert a [Resource](Resource) to a JavaScript object suitable for long-term
peristent storage.
</p>

**Kind**: instance method of [<code>Serializer</code>](#module_concerto-
core.Serializer)
**Returns**: <code>Object</code> - - The Javascript Object that represents the
resource
**Throws**:

- <code>Error</code> - throws an exception if resource is not an instance of
Resource or fails validation.


| Param | Type | Description |
| --- | --- | --- |
| resource | <code>Resource</code> | The instance to convert to JSON |
| [options] | <code>Object</code> | the optional serialization options. |
| [options.validate] | <code>boolean</code> | validate the structure of the
Resource with its model prior to serialization (default to true) |
| [options.convertResourcesToRelationships] | <code>boolean</code> | Convert
resources that are specified for relationship fields into relationships, false by
default. |
| [options.permitResourcesForRelationships] | <code>boolean</code> | Permit
resources in the place of relationships (serializing them as resources), false by
```

```
default. |
| [options.deduplicateResources] | <code>boolean</code> | Generate $id for
resources and if a resources appears multiple times in the object graph only the
first instance is serialized in full, subsequent instances are replaced with a
reference to the $id |
| [options.convertResourcesToId] | <code>boolean</code> | Convert resources that
are specified for relationship fields into their id, false by default. |
| [options.utcOffset] | <code>number</code> | UTC Offset for DateTime values. |

<a name="module_concerto-core.Serializer+fromJSON"></a>

#### serializer.fromJSON(jsonObject, [options]) ⇒ <code>Resource</code>
Create a [Resource](Resource) from a JavaScript Object representation.
The JavaScript Object should have been created by calling the
[toJSON](Serializer#toJSON) API.

The Resource is populated based on the JavaScript object.

**Kind**: instance method of [<code>Serializer</code>](#module_concerto-
core.Serializer)
**Returns**: <code>Resource</code> - The new populated resource

| Param | Type | Description |
| --- | --- | --- |
| jsonObject | <code>Object</code> | The JavaScript Object for a Resource |
| [options] | <code>Object</code> | the optional serialization options |
| options.acceptResourcesForRelationships | <code>boolean</code> | handle JSON
objects in the place of strings for relationships, defaults to false. |
| options.validate | <code>boolean</code> | validate the structure of the Resource
with its model prior to serialization (default to true) |
| [options.utcOffset] | <code>number</code> | UTC Offset for DateTime values. |

<a name="module_concerto-core.TypeNotFoundException"></a>

### concerto-core.TypeNotFoundException ⇐ <code>BaseException</code>
Error thrown when a Concerto type does not exist.

**Kind**: static class of [<code>concerto-core</code>](#module_concerto-core)
**Extends**: <code>BaseException</code>
**See**: see [BaseException](BaseException)

* [.TypeNotFoundException](#module_concerto-core.TypeNotFoundException) ⇐
<code>BaseException</code>
    * [new TypeNotFoundException(typeName, message, component)]
(#new_module_concerto-core.TypeNotFoundException_new)
    * [.getTypeName()](#module_concerto-core.TypeNotFoundException+getTypeName) ⇒
<code>string</code>

<a name="new_module_concerto-core.TypeNotFoundException_new"></a>

#### new TypeNotFoundException(typeName, message, component)
Constructor. If the optional 'message' argument is not supplied, it will be set to
a default value that
includes the type name.


| Param | Type | Description |
| --- | --- | --- |
| typeName | <code>string</code> | fully qualified type name. |
| message | <code>string</code> \| <code>undefined</code> | error message. |
| component | <code>string</code> | the optional component which throws this error
|

<a name="module_concerto-core.TypeNotFoundException+getTypeName"></a>

#### typeNotFoundException.getTypeName() ⇒ <code>string</code>
Get the name of the type that was not found.

**Kind**: instance method of [<code>TypeNotFoundException</code>](#module_concerto-
core.TypeNotFoundException)
**Returns**: <code>string</code> - fully qualified type name.
<a name="module_concerto-core.BaseException"></a>

### concerto-core.BaseException ⇐ <code>Error</code>
A base class for all Concerto exceptions

**Kind**: static class of [<code>concerto-core</code>](#module_concerto-core)
**Extends**: <code>Error</code>
<a name="new_module_concerto-core.BaseException_new"></a>

#### new BaseException(message, component)
Create the BaseException.


| Param | Type | Description |
| --- | --- | --- |
| message | <code>string</code> | The exception message. |
| component | <code>string</code> | The optional component which throws this error.
|

<a name="module_concerto-core.BaseFileException"></a>

### concerto-core.BaseFileException ⇐ <code>BaseException</code>
Exception throws when a Concerto file is semantically invalid

**Kind**: static class of [<code>concerto-core</code>](#module_concerto-core)
**Extends**: <code>BaseException</code>
**See**: [BaseException](BaseException)

* [.BaseFileException](#module_concerto-core.BaseFileException) ⇐
<code>BaseException</code>
    * [new BaseFileException(message, fileLocation, fullMessage, [fileName],
[component])](#new_module_concerto-core.BaseFileException_new)
    * [.getFileLocation()](#module_concerto-core.BaseFileException+getFileLocation)
⇒ <code>string</code>
    * [.getShortMessage()](#module_concerto-core.BaseFileException+getShortMessage)
⇒ <code>string</code>
    * [.getFileName()](#module_concerto-core.BaseFileException+getFileName) ⇒
<code>string</code>

<a name="new_module_concerto-core.BaseFileException_new"></a>

#### new BaseFileException(message, fileLocation, fullMessage, [fileName],
[component])
Create an BaseFileException


| Param | Type | Description |
| --- | --- | --- |
| message | <code>string</code> | the message for the exception |
| fileLocation | <code>string</code> | the optional file location associated with
the exception |
| fullMessage | <code>string</code> | the optional full message text |
| [fileName] | <code>string</code> | the file name |
| [component] | <code>string</code> | the component which throws this error |

<a name="module_concerto-core.BaseFileException+getFileLocation"></a>

#### baseFileException.getFileLocation() ⇒ <code>string</code>
Returns the file location associated with the exception or null

**Kind**: instance method of [<code>BaseFileException</code>](#module_concerto-
core.BaseFileException)
**Returns**: <code>string</code> - the optional location associated with the
exception
<a name="module_concerto-core.BaseFileException+getShortMessage"></a>

#### baseFileException.getShortMessage() ⇒ <code>string</code>
Returns the error message without the location of the error

**Kind**: instance method of [<code>BaseFileException</code>](#module_concerto-
core.BaseFileException)
**Returns**: <code>string</code> - the error message
<a name="module_concerto-core.BaseFileException+getFileName"></a>

#### baseFileException.getFileName() ⇒ <code>string</code>
Returns the fileName for the error

**Kind**: instance method of [<code>BaseFileException</code>](#module_concerto-
core.BaseFileException)
**Returns**: <code>string</code> - the file name or null
<a name="module_concerto-core.FileDownloader"></a>

### concerto-core.FileDownloader
Downloads the transitive closure of a set of model files.

**Kind**: static class of [<code>concerto-core</code>](#module_concerto-core)

* [.FileDownloader](#module_concerto-core.FileDownloader)
    * [new FileDownloader(fileLoader, getExternalImports, concurrency)]
(#new_module_concerto-core.FileDownloader_new)
    * [.downloadExternalDependencies(files, [options])](#module_concerto-
core.FileDownloader+downloadExternalDependencies) ⇒ <code>Promise</code>
    * [.runJob(job, fileLoader)](#module_concerto-core.FileDownloader+runJob) ⇒
<code>Promise</code>

<a name="new_module_concerto-core.FileDownloader_new"></a>

#### new FileDownloader(fileLoader, getExternalImports, concurrency)
Create a FileDownloader and bind to a FileLoader.


| Param | Type | Default | Description |
| --- | --- | --- | --- |
| fileLoader | <code>\*</code> |  | the loader to use to download model files |
| getExternalImports | <code>\*</code> |  | a function taking a file and returning
new files |
```

| concurrency | <code>Number</code> | <code>10</code> | the number of model files to download concurrently |

<a name="module_concerto-core.FileDownloader+downloadExternalDependencies"></a>

#### fileDownloader.downloadExternalDependencies(files, [options]) ⇒ <code>Promise</code>
Download all external dependencies for an array of model files

Kind: instance method of [<code>FileDownloader</code>](#module_concerto-core.FileDownloader)
Returns: <code>Promise</code> - a promise that resolves to Files[] for the external model files

Param	Type	Description
files	<code>Array.&lt;File&gt;</code>	the model files
[options]	<code>Object</code>	Options object passed to FileLoaders

<a name="module_concerto-core.FileDownloader+runJob"></a>

#### fileDownloader.runJob(job, fileLoader) ⇒ <code>Promise</code>
Execute a Job

Kind: instance method of [<code>FileDownloader</code>](#module_concerto-core.FileDownloader)
Returns: <code>Promise</code> - a promise to the job results

Param	Type	Description
job	<code>Object</code>	the job to execute
fileLoader	<code>Object</code>	the loader to use to download model files.

<a name="module_concerto-core.TypedStack"></a>

### concerto-core.TypedStack
Tracks a stack of typed instances. The type information is used to detect overflow / underflow bugs by the caller. It also performs basic sanity checking on push/pop to make detecting bugs easier.

Kind: static class of [<code>concerto-core</code>](#module_concerto-core)

* [.TypedStack](#module_concerto-core.TypedStack)
    * [new TypedStack(resource)](#new_module_concerto-core.TypedStack_new)
    * [.push(obj, expectedType)](#module_concerto-core.TypedStack+push)
    * [.pop(expectedType)](#module_concerto-core.TypedStack+pop) ⇒ <code>Object</code>
    * [.peek(expectedType)](#module_concerto-core.TypedStack+peek) ⇒ <code>Object</code>
    * [.clear()](#module_concerto-core.TypedStack+clear)

<a name="new_module_concerto-core.TypedStack_new"></a>

#### new TypedStack(resource)
Create the Stack with the resource at the head.

Param	Type	Description

| resource | <code>Object</code> | the resource to be put at the head of the stack
|

<a name="module_concerto-core.TypedStack+push"></a>

#### typedStack.push(obj, expectedType)
Push a new object.

Kind: instance method of [<code>TypedStack</code>](#module_concerto-
core.TypedStack)

Param	Type	Description
obj	<code>Object</code>	the object being visited
expectedType	<code>Object</code>	the expected type of the object being pushed

<a name="module_concerto-core.TypedStack+pop"></a>

#### typedStack.pop(expectedType) ⇒ <code>Object</code>
Push a new object.

Kind: instance method of [<code>TypedStack</code>](#module_concerto-
core.TypedStack)
Returns: <code>Object</code> - the result of pop

Param	Type	Description
expectedType	<code>Object</code>	the type that should be the result of pop

<a name="module_concerto-core.TypedStack+peek"></a>

#### typedStack.peek(expectedType) ⇒ <code>Object</code>
Peek the top of the stack

Kind: instance method of [<code>TypedStack</code>](#module_concerto-
core.TypedStack)
Returns: <code>Object</code> - the result of peek

Param	Type	Description
expectedType	<code>Object</code>	the type that should be the result of pop

<a name="module_concerto-core.TypedStack+clear"></a>

#### typedStack.clear()
Clears the stack

Kind: instance method of [<code>TypedStack</code>](#module_concerto-
core.TypedStack)
<a name="module_concerto-core..version"></a>

### concerto-core~version : <code>Object</code>
Kind: inner constant of [<code>concerto-core</code>](#module_concerto-core)
<a name="module_concerto-cto"></a>

## concerto-cto
Concerto CTO concrete syntax module. Concerto is a framework for defining domain
specific models.

```
<a name="module_concerto-metamodel"></a>

## concerto-metamodel
Concerto metamodel management. Concerto is a framework for defining domain
specific models.

<a name="module_concerto-tools"></a>

## concerto-tools
Concerto Tools module.

<a name="module_concerto-util"></a>

## concerto-util
Concerto utility module. Concerto is a framework for defining domain
specific models.

<a name="module_concerto-vocabulary"></a>

## concerto-vocabulary
Concerto vocabulary module. Concerto is a framework for defining domain
specific models.


* [concerto-vocabulary](#module_concerto-vocabulary)
    * [.Vocabulary](#module_concerto-vocabulary.Vocabulary)
        * [new Vocabulary(vocabularyManager, voc)](#new_module_concerto-
vocabulary.Vocabulary_new)
        * [.getNamespace()](#module_concerto-vocabulary.Vocabulary+getNamespace) ⇒
<code>string</code>
        * [.getLocale()](#module_concerto-vocabulary.Vocabulary+getLocale) ⇒
<code>string</code>
        * [.getIdentifier()](#module_concerto-vocabulary.Vocabulary+getIdentifier)
⇒ <code>string</code>
        * [.getTerms()](#module_concerto-vocabulary.Vocabulary+getTerms) ⇒
<code>Array</code>
        * [.getTerm(declarationName, [propertyName])](#module_concerto-
vocabulary.Vocabulary+getTerm) ⇒ <code>string</code>
        * [.validate(modelFile)](#module_concerto-vocabulary.Vocabulary+validate) ⇒
<code>\*</code>
        * [.toJSON()](#module_concerto-vocabulary.Vocabulary+toJSON) ⇒
<code>\*</code>
    * [.VocabularyManager](#module_concerto-vocabulary.VocabularyManager)
        * [new VocabularyManager([options])](#new_module_concerto-
vocabulary.VocabularyManager_new)
        * _instance_
            * [.clear()](#module_concerto-vocabulary.VocabularyManager+clear)
            * [.removeVocabulary(namespace, locale)](#module_concerto-
vocabulary.VocabularyManager+removeVocabulary)
            * [.addVocabulary(contents)](#module_concerto-
vocabulary.VocabularyManager+addVocabulary) ⇒ <code>Vocabulary</code>
            * [.getVocabulary(namespace, locale, [options])](#module_concerto-
vocabulary.VocabularyManager+getVocabulary) ⇒ <code>Vocabulary</code>
            * [.getVocabulariesForNamespace(namespace)](#module_concerto-
vocabulary.VocabularyManager+getVocabulariesForNamespace) ⇒
<code>Array.&lt;Vocabulary&gt;</code>
            * [.getVocabulariesForLocale(locale)](#module_concerto-
vocabulary.VocabularyManager+getVocabulariesForLocale) ⇒
```

<code>Array.&lt;Vocabulary&gt;</code>
            * [.resolveTerm(modelManager, namespace, locale, declarationName, [propertyName])](#module_concerto-vocabulary.VocabularyManager+resolveTerm) ⇒ <code>string</code>
            * [.getTerm(namespace, locale, declarationName, [propertyName])](#module_concerto-vocabulary.VocabularyManager+getTerm) ⇒ <code>string</code>
            * [.generateDecoratorCommands(modelManager, locale)](#module_concerto-vocabulary.VocabularyManager+generateDecoratorCommands) ⇒ <code>*</code>
            * [.validate(modelManager, locale)](#module_concerto-vocabulary.VocabularyManager+validate) ⇒ <code>*</code>
        * _static_
            * [.englishMissingTermGenerator(namespace, locale, declarationName, [propertyName])](#module_concerto-vocabulary.VocabularyManager.englishMissingTermGenerator) ⇒ <code>string</code>
            * [.findVocabulary(requestedLocale, vocabularies, [options])](#module_concerto-vocabulary.VocabularyManager.findVocabulary) ⇒ <code>Vocabulary</code>

<a name="module_concerto-vocabulary.Vocabulary"></a>

### concerto-vocabulary.Vocabulary
A vocabulary for a concerto model

Kind: static class of [<code>concerto-vocabulary</code>](#module_concerto-vocabulary)

* [.Vocabulary](#module_concerto-vocabulary.Vocabulary)
    * [new Vocabulary(vocabularyManager, voc)](#new_module_concerto-vocabulary.Vocabulary_new)
    * [.getNamespace()](#module_concerto-vocabulary.Vocabulary+getNamespace) ⇒ <code>string</code>
    * [.getLocale()](#module_concerto-vocabulary.Vocabulary+getLocale) ⇒ <code>string</code>
    * [.getIdentifier()](#module_concerto-vocabulary.Vocabulary+getIdentifier) ⇒ <code>string</code>
    * [.getTerms()](#module_concerto-vocabulary.Vocabulary+getTerms) ⇒ <code>Array</code>
    * [.getTerm(declarationName, [propertyName])](#module_concerto-vocabulary.Vocabulary+getTerm) ⇒ <code>string</code>
    * [.validate(modelFile)](#module_concerto-vocabulary.Vocabulary+validate) ⇒ <code>*</code>
    * [.toJSON()](#module_concerto-vocabulary.Vocabulary+toJSON) ⇒ <code>*</code>

<a name="new_module_concerto-vocabulary.Vocabulary_new"></a>

#### new Vocabulary(vocabularyManager, voc)
Create the Vocabulary

Param	Type	Description
vocabularyManager	<code>VocabularyManager</code>	the manager for this vocabulary
voc	<code>object</code>	the JSON representation of the vocabulary

<a name="module_concerto-vocabulary.Vocabulary+getNamespace"></a>

#### vocabulary.getNamespace() ⇒ <code>string</code>
Returns the namespace for the vocabulary

```
**Kind**: instance method of [<code>Vocabulary</code>](#module_concerto-
vocabulary.Vocabulary)
**Returns**: <code>string</code> - the namespace for this vocabulary
<a name="module_concerto-vocabulary.Vocabulary+getLocale"></a>

#### vocabulary.getLocale() ⇒ <code>string</code>
Returns the locale for the vocabulary

**Kind**: instance method of [<code>Vocabulary</code>](#module_concerto-
vocabulary.Vocabulary)
**Returns**: <code>string</code> - the locale for this vocabulary
<a name="module_concerto-vocabulary.Vocabulary+getIdentifier"></a>

#### vocabulary.getIdentifier() ⇒ <code>string</code>
Returns the identifier for the vocabulary, composed of the namespace plus the
locale

**Kind**: instance method of [<code>Vocabulary</code>](#module_concerto-
vocabulary.Vocabulary)
**Returns**: <code>string</code> - the identifier for this vocabulary
<a name="module_concerto-vocabulary.Vocabulary+getTerms"></a>

#### vocabulary.getTerms() ⇒ <code>Array</code>
Returns all the declarations for this vocabulary

**Kind**: instance method of [<code>Vocabulary</code>](#module_concerto-
vocabulary.Vocabulary)
**Returns**: <code>Array</code> - an array of objects
<a name="module_concerto-vocabulary.Vocabulary+getTerm"></a>

#### vocabulary.getTerm(declarationName, [propertyName]) ⇒ <code>string</code>
Gets the term for a concept, enum or property

**Kind**: instance method of [<code>Vocabulary</code>](#module_concerto-
vocabulary.Vocabulary)
**Returns**: <code>string</code> - the term or null if it does not exist

| Param | Type | Description |
| --- | --- | --- |
| declarationName | <code>string</code> | the name of a concept or enum |
| [propertyName] | <code>string</code> | the name of a property (optional) |

<a name="module_concerto-vocabulary.Vocabulary+validate"></a>

#### vocabulary.validate(modelFile) ⇒ <code>\*</code>
Validates a vocabulary against a ModelFile, returning errors for
missing and additional terms.

**Kind**: instance method of [<code>Vocabulary</code>](#module_concerto-
vocabulary.Vocabulary)
**Returns**: <code>\*</code> - an object with missingTerms and additionalTerms
properties

| Param | Type | Description |
| --- | --- | --- |
| modelFile | <code>ModelFile</code> | the model file for this vocabulary |

<a name="module_concerto-vocabulary.Vocabulary+toJSON"></a>
```

```
#### vocabulary.toJSON() ⇒ <code>\*</code>
Converts the object to JSON

**Kind**: instance method of [<code>Vocabulary</code>](#module_concerto-
vocabulary.Vocabulary)
**Returns**: <code>\*</code> - the contens of this vocabulary
<a name="module_concerto-vocabulary.VocabularyManager"></a>

### concerto-vocabulary.VocabularyManager
A vocabulary manager for concerto models. The vocabulary manager
stores and provides API access to a set of vocabulary files, where each file
is associated with a BCP-47 language tag and a Concerto namespace.

**Kind**: static class of [<code>concerto-vocabulary</code>](#module_concerto-
vocabulary)
**See**: https://datatracker.ietf.org/doc/html/rfc5646#section-2

* [.VocabularyManager](#module_concerto-vocabulary.VocabularyManager)
    * [new VocabularyManager([options])](#new_module_concerto-
vocabulary.VocabularyManager_new)
    * _instance_
        * [.clear()](#module_concerto-vocabulary.VocabularyManager+clear)
        * [.removeVocabulary(namespace, locale)](#module_concerto-
vocabulary.VocabularyManager+removeVocabulary)
        * [.addVocabulary(contents)](#module_concerto-
vocabulary.VocabularyManager+addVocabulary) ⇒ <code>Vocabulary</code>
        * [.getVocabulary(namespace, locale, [options])](#module_concerto-
vocabulary.VocabularyManager+getVocabulary) ⇒ <code>Vocabulary</code>
        * [.getVocabulariesForNamespace(namespace)](#module_concerto-
vocabulary.VocabularyManager+getVocabulariesForNamespace) ⇒
<code>Array.&lt;Vocabulary&gt;</code>
        * [.getVocabulariesForLocale(locale)](#module_concerto-
vocabulary.VocabularyManager+getVocabulariesForLocale) ⇒
<code>Array.&lt;Vocabulary&gt;</code>
        * [.resolveTerm(modelManager, namespace, locale, declarationName,
[propertyName])](#module_concerto-vocabulary.VocabularyManager+resolveTerm) ⇒
<code>string</code>
        * [.getTerm(namespace, locale, declarationName, [propertyName])]
(#module_concerto-vocabulary.VocabularyManager+getTerm) ⇒ <code>string</code>
        * [.generateDecoratorCommands(modelManager, locale)](#module_concerto-
vocabulary.VocabularyManager+generateDecoratorCommands) ⇒ <code>\*</code>
        * [.validate(modelManager, locale)](#module_concerto-
vocabulary.VocabularyManager+validate) ⇒ <code>\*</code>
    * _static_
        * [.englishMissingTermGenerator(namespace, locale, declarationName,
[propertyName])](#module_concerto-
vocabulary.VocabularyManager.englishMissingTermGenerator) ⇒ <code>string</code>
        * [.findVocabulary(requestedLocale, vocabularies, [options])]
(#module_concerto-vocabulary.VocabularyManager.findVocabulary) ⇒
<code>Vocabulary</code>

<a name="new_module_concerto-vocabulary.VocabularyManager_new"></a>

#### new VocabularyManager([options])
Create the VocabularyManager


| Param | Type | Description |
| --- | --- | --- |
| [options] | <code>\*</code> | options to configure vocabulary lookup |
| [options.missingTermGenerator] | <code>\*</code> | A function to call for missing
terms. The function should accept namespace, locale, declarationName, propertyName
as arguments |

<a name="module_concerto-vocabulary.VocabularyManager+clear"></a>

#### vocabularyManager.clear()
Removes all vocabularies

**Kind**: instance method of [<code>VocabularyManager</code>](#module_concerto-
vocabulary.VocabularyManager)
<a name="module_concerto-vocabulary.VocabularyManager+removeVocabulary"></a>

#### vocabularyManager.removeVocabulary(namespace, locale)
Removes a vocabulary from the vocabulary manager

**Kind**: instance method of [<code>VocabularyManager</code>](#module_concerto-
vocabulary.VocabularyManager)

| Param | Type | Description |
| --- | --- | --- |
| namespace | <code>string</code> | the namespace for the vocabulary |
| locale | <code>string</code> | the BCP-47 locale identifier |

<a name="module_concerto-vocabulary.VocabularyManager+addVocabulary"></a>

#### vocabularyManager.addVocabulary(contents) ⇒ <code>Vocabulary</code>
Adds a vocabulary to the vocabulary manager

**Kind**: instance method of [<code>VocabularyManager</code>](#module_concerto-
vocabulary.VocabularyManager)
**Returns**: <code>Vocabulary</code> - the vocabulary the was added

| Param | Type | Description |
| --- | --- | --- |
| contents | <code>string</code> | the YAML string for the vocabulary |

<a name="module_concerto-vocabulary.VocabularyManager+getVocabulary"></a>

#### vocabularyManager.getVocabulary(namespace, locale, [options]) ⇒
<code>Vocabulary</code>
Gets a vocabulary for a given namespace plus locale

**Kind**: instance method of [<code>VocabularyManager</code>](#module_concerto-
vocabulary.VocabularyManager)
**Returns**: <code>Vocabulary</code> - the vocabulary or null if no vocabulary
exists for the locale

| Param | Type | Description |
| --- | --- | --- |
| namespace | <code>string</code> | the namespace for the vocabulary |
| locale | <code>string</code> | the BCP-47 locale identifier |
| [options] | <code>\*</code> | options to configure vocabulary lookup |
| [options.localeMatcher] | <code>\*</code> | Pass 'lookup' to find a general
vocabulary, if available |

<a name="module_concerto-
```

```
vocabulary.VocabularyManager+getVocabulariesForNamespace"></a>

#### vocabularyManager.getVocabulariesForNamespace(namespace) ⇒
<code>Array.&lt;Vocabulary&gt;</code>
Gets all the vocabulary files for a given namespace

**Kind**: instance method of [<code>VocabularyManager</code>](#module_concerto-
vocabulary.VocabularyManager)
**Returns**: <code>Array.&lt;Vocabulary&gt;</code> - the array of vocabularies

| Param | Type | Description |
| --- | --- | --- |
| namespace | <code>string</code> | the namespace |

<a name="module_concerto-vocabulary.VocabularyManager+getVocabulariesForLocale"></
a>

#### vocabularyManager.getVocabulariesForLocale(locale) ⇒
<code>Array.&lt;Vocabulary&gt;</code>
Gets all the vocabulary files for a given locale

**Kind**: instance method of [<code>VocabularyManager</code>](#module_concerto-
vocabulary.VocabularyManager)
**Returns**: <code>Array.&lt;Vocabulary&gt;</code> - the array of vocabularies

| Param | Type | Description |
| --- | --- | --- |
| locale | <code>string</code> | the BCP-47 locale identifier |

<a name="module_concerto-vocabulary.VocabularyManager+resolveTerm"></a>

#### vocabularyManager.resolveTerm(modelManager, namespace, locale,
declarationName, [propertyName]) ⇒ <code>string</code>
Resolve the term for a property, looking up terms from a more general vocabulary
if required, and resolving properties using an object manager, allowing terms
defined
on super types to be automatically resolved.

**Kind**: instance method of [<code>VocabularyManager</code>](#module_concerto-
vocabulary.VocabularyManager)
**Returns**: <code>string</code> - the term or null if it does not exist

| Param | Type | Description |
| --- | --- | --- |
| modelManager | <code>ModelManager</code> | the model manager |
| namespace | <code>string</code> | the namespace |
| locale | <code>string</code> | the BCP-47 locale identifier |
| declarationName | <code>string</code> | the name of a concept or enum |
| [propertyName] | <code>string</code> | the name of a property (optional) |

<a name="module_concerto-vocabulary.VocabularyManager+getTerm"></a>

#### vocabularyManager.getTerm(namespace, locale, declarationName, [propertyName])
⇒ <code>string</code>
Gets the term for a concept, enum or property, looking up terms
from a more general vocabulary if required

**Kind**: instance method of [<code>VocabularyManager</code>](#module_concerto-
vocabulary.VocabularyManager)
```

Returns: <code>string</code> - the term or null if it does not exist

Param	Type	Description
namespace	<code>string</code>	the namespace
locale	<code>string</code>	the BCP-47 locale identifier
declarationName	<code>string</code>	the name of a concept or enum
[propertyName]	<code>string</code>	the name of a property (optional)

<a name="module_concerto-
vocabulary.VocabularyManager+generateDecoratorCommands"></a>

#### vocabularyManager.generateDecoratorCommands(modelManager, locale) ⇒ <code>\
*</code>
Creates a DecoractorCommandSet with @Term decorators
to decorate all model elements based on the vocabulary for a locale.
Pass the return value to the DecoratorManager.decorateModel to apply
the decorators to a ModelManager.

Kind: instance method of [<code>VocabularyManager</code>](#module_concerto-
vocabulary.VocabularyManager)
Returns: <code>*</code> - the decorator command set used to decorate the
model.

Param	Type	Description
modelManager	<code>ModelManager</code>	the Model Manager
locale	<code>string</code>	the BCP-47 locale identifier

<a name="module_concerto-vocabulary.VocabularyManager+validate"></a>

#### vocabularyManager.validate(modelManager, locale) ⇒ <code>*</code>
Validates the terms in the vocabulary against the namespaces and declarations
within a ModelManager

Kind: instance method of [<code>VocabularyManager</code>](#module_concerto-
vocabulary.VocabularyManager)
Returns: <code>*</code> - the result of validation

Param	Type	Description
modelManager	<code>ModelManager</code>	the Model Manager
locale	<code>string</code>	the BCP-47 locale identifier

<a name="module_concerto-
vocabulary.VocabularyManager.englishMissingTermGenerator"></a>

#### VocabularyManager.englishMissingTermGenerator(namespace, locale,
declarationName, [propertyName]) ⇒ <code>string</code>
Computes a term in English based on declaration and property name.

Kind: static method of [<code>VocabularyManager</code>](#module_concerto-
vocabulary.VocabularyManager)
Returns: <code>string</code> - the term or null if it does not exist

Param	Type	Description
namespace	<code>string</code>	the namespace
locale	<code>string</code>	the BCP-47 locale identifier

```
| declarationName | <code>string</code> | the name of a concept or enum |
| [propertyName] | <code>string</code> | the name of a property (optional) |

<a name="module_concerto-vocabulary.VocabularyManager.findVocabulary"></a>

#### VocabularyManager.findVocabulary(requestedLocale, vocabularies, [options]) ⇒
<code>Vocabulary</code>
Finds the vocabulary for a requested locale, removing language
identifiers from the locale until the locale matches, or if no
vocabulary is found, null is returned

**Kind**: static method of [<code>VocabularyManager</code>](#module_concerto-
vocabulary.VocabularyManager)
**Returns**: <code>Vocabulary</code> - the most specific vocabulary, or null

| Param | Type | Description |
| --- | --- | --- |
| requestedLocale | <code>string</code> | the BCP-47 locale identifier |
| vocabularies | <code>Array.&lt;Vocabulary&gt;</code> | the vocabularies to match
against |
| [options] | <code>\*</code> | options to configure vocabulary lookup |
| [options.localeMatcher] | <code>\*</code> | Pass 'lookup' to find a general
vocabulary, if available |

<a name="AbstractPlugin"></a>

## AbstractPlugin
Simple plug-in class for code-generation. This lists functions that can be passed
to extend the default code-generation behavior.

**Kind**: global class

* [AbstractPlugin](#AbstractPlugin)
    * [.addClassImports(clazz, parameters, options)]
(#AbstractPlugin+addClassImports)
    * [.addClassAnnotations(clazz, parameters, options)]
(#AbstractPlugin+addClassAnnotations)
    * [.addClassMethods(clazz, parameters, options)]
(#AbstractPlugin+addClassMethods)
    * [.addEnumAnnotations(enumDecl, parameters, options)]
(#AbstractPlugin+addEnumAnnotations)

<a name="AbstractPlugin+addClassImports"></a>

### abstractPlugin.addClassImports(clazz, parameters, options)
Additional imports to generate in classes

**Kind**: instance method of [<code>AbstractPlugin</code>](#AbstractPlugin)

| Param | Type | Description |
| --- | --- | --- |
| clazz | <code>ClassDeclaration</code> | the clazz being visited |
| parameters | <code>Object</code> | the parameter |
| options | <code>Object</code> | the visitor options |

<a name="AbstractPlugin+addClassAnnotations"></a>

### abstractPlugin.addClassAnnotations(clazz, parameters, options)
Additional annotations to generate in classes
```

Kind: instance method of [<code>AbstractPlugin</code>](#AbstractPlugin)

Param	Type	Description
clazz	<code>ClassDeclaration</code>	the clazz being visited
parameters	<code>Object</code>	the parameter
options	<code>Object</code>	the visitor options

<a name="AbstractPlugin+addClassMethods"></a>

### abstractPlugin.addClassMethods(clazz, parameters, options)
Additional methods to generate in classes

Kind: instance method of [<code>AbstractPlugin</code>](#AbstractPlugin)

Param	Type	Description
clazz	<code>ClassDeclaration</code>	the clazz being visited
parameters	<code>Object</code>	the parameter
options	<code>Object</code>	the visitor options

<a name="AbstractPlugin+addEnumAnnotations"></a>

### abstractPlugin.addEnumAnnotations(enumDecl, parameters, options)
Additional annotations to generate in enums

Kind: instance method of [<code>AbstractPlugin</code>](#AbstractPlugin)

Param	Type	Description
enumDecl	<code>EnumDeclaration</code>	the enum being visited
parameters	<code>Object</code>	the parameter
options	<code>Object</code>	the visitor options

<a name="EmptyPlugin"></a>

## EmptyPlugin
Simple plug-in class for code-generation. This lists functions that can be passed
to extend the default code-generation behavior.

Kind: global class

* [EmptyPlugin](#EmptyPlugin)
    * [.addClassImports(clazz, parameters)](#EmptyPlugin+addClassImports)
    * [.addClassAnnotations(clazz, parameters)](#EmptyPlugin+addClassAnnotations)
    * [.addClassMethods(clazz, parameters)](#EmptyPlugin+addClassMethods)
    * [.addEnumAnnotations(enumDecl, parameters)](#EmptyPlugin+addEnumAnnotations)

<a name="EmptyPlugin+addClassImports"></a>

### emptyPlugin.addClassImports(clazz, parameters)
Additional imports to generate in classes

Kind: instance method of [<code>EmptyPlugin</code>](#EmptyPlugin)

Param	Type	Description
clazz	<code>ClassDeclaration</code>	the clazz being visited

```
| parameters | <code>Object</code> | the parameter |

<a name="EmptyPlugin+addClassAnnotations"></a>

### emptyPlugin.addClassAnnotations(clazz, parameters)
Additional annotations to generate in classes

**Kind**: instance method of [<code>EmptyPlugin</code>](#EmptyPlugin)

| Param | Type | Description |
| --- | --- | --- |
| clazz | <code>ClassDeclaration</code> | the clazz being visited |
| parameters | <code>Object</code> | the parameter |

<a name="EmptyPlugin+addClassMethods"></a>

### emptyPlugin.addClassMethods(clazz, parameters)
Additional methods to generate in classes

**Kind**: instance method of [<code>EmptyPlugin</code>](#EmptyPlugin)

| Param | Type | Description |
| --- | --- | --- |
| clazz | <code>ClassDeclaration</code> | the clazz being visited |
| parameters | <code>Object</code> | the parameter |

<a name="EmptyPlugin+addEnumAnnotations"></a>

### emptyPlugin.addEnumAnnotations(enumDecl, parameters)
Additional annotations to generate in enums

**Kind**: instance method of [<code>EmptyPlugin</code>](#EmptyPlugin)

| Param | Type | Description |
| --- | --- | --- |
| enumDecl | <code>EnumDeclaration</code> | the enum being visited |
| parameters | <code>Object</code> | the parameter |

<a name="rootModelAst"></a>

## rootModelAst : <code>unknown</code>
**Kind**: global constant
<a name="metaModelAst"></a>

## metaModelAst : <code>unknown</code>
The metamodel itself, as an AST.

**Kind**: global constant
<a name="metaModelCto"></a>

## metaModelCto
The metamodel itself, as a CTO string

**Kind**: global constant
<a name="levels"></a>

## levels : <code>Object</code>
Default levels for the npm configuration.
```

```
**Kind**: global constant
<a name="colorMap"></a>

## colorMap : <code>Object</code>
Default levels for the npm configuration.

**Kind**: global constant
<a name="setCurrentTime"></a>

## setCurrentTime([currentTime], [utcOffset]) ⇒ <code>object</code>
Ensures there is a proper current time

**Kind**: global function
**Returns**: <code>object</code> - if valid, the dayjs object for the current time

| Param | Type | Description |
| --- | --- | --- |
| [currentTime] | <code>string</code> | the definition of 'now' |
| [utcOffset] | <code>number</code> | UTC Offset for this execution |

<a name="newMetaModelManager"></a>

## newMetaModelManager() ⇒ <code>\*</code>
Create a metamodel manager (for validation against the metamodel)

**Kind**: global function
**Returns**: <code>\*</code> - the metamodel manager
<a name="validateMetaModel"></a>

## validateMetaModel(input) ⇒ <code>object</code>
Validate metamodel instance against the metamodel

**Kind**: global function
**Returns**: <code>object</code> - the validated metamodel instance in JSON

| Param | Type | Description |
| --- | --- | --- |
| input | <code>object</code> | the metamodel instance in JSON |

<a name="modelManagerFromMetaModel"></a>

## modelManagerFromMetaModel(metaModel, [validate]) ⇒ <code>object</code>
Import metamodel to a model manager

**Kind**: global function
**Returns**: <code>object</code> - the metamodel for this model manager

| Param | Type | Default | Description |
| --- | --- | --- | --- |
| metaModel | <code>object</code> |  | the metamodel |
| [validate] | <code>boolean</code> | <code>true</code> | whether to perform
validation |

<a name="randomNumberInRangeWithPrecision"></a>

## randomNumberInRangeWithPrecision(userMin, userMax, precision, systemMin,
systemMax) ⇒ <code>number</code>
Generate a random number within a given range with
a prescribed precision and inside a global range
```

```
**Kind**: global function
**Returns**: <code>number</code> - a number

| Param | Type | Description |
| --- | --- | --- |
| userMin | <code>\*</code> | Lower bound on the range, inclusive. Defaults to
systemMin |
| userMax | <code>\*</code> | Upper bound on the range, inclusive. Defaults to
systemMax |
| precision | <code>\*</code> | The precision of values returned, e.g. a value of
`1` returns only whole numbers |
| systemMin | <code>\*</code> | Global minimum on the range, takes precidence over
the userMin |
| systemMax | <code>\*</code> | Global maximum on the range, takes precidence over
the userMax |

<a name="updateModels"></a>

## updateModels(models, newModel) ⇒ <code>\*</code>
Update models with a new model

**Kind**: global function
**Returns**: <code>\*</code> - the updated models

| Param | Type | Description |
| --- | --- | --- |
| models | <code>\*</code> | existing models |
| newModel | <code>\*</code> | new model |

<a name="resolveExternal"></a>

## resolveExternal(models, [options], [fileDownloader]) ⇒ <code>Promise</code>
Downloads all ModelFiles that are external dependencies and adds or
updates them in this ModelManager.

**Kind**: global function
**Returns**: <code>Promise</code> - a promise when the download and update
operation is completed.
**Throws**:

- <code>IllegalModelException</code> if the models fail validation


| Param | Type | Description |
| --- | --- | --- |
| models | <code>\*</code> | the AST for all the known models |
| [options] | <code>Object</code> | Options object passed to ModelFileLoaders |
| [fileDownloader] | <code>FileDownloader</code> | an optional FileDownloader |

<a name="parse"></a>

## parse(cto, [fileName]) ⇒ <code>object</code>
Create decorator argument string from a metamodel

**Kind**: global function
**Returns**: <code>object</code> - the string for the decorator argument

| Param | Type | Description |
| --- | --- | --- |
| cto | <code>string</code> | the Concerto string |
| [fileName] | <code>string</code> | an optional file name |

<a name="parseModels"></a>

## parseModels(files) ⇒ <code>\*</code>
Parses an array of model files

**Kind**: global function
**Returns**: <code>\*</code> - the AST / metamodel

| Param | Type | Description |
| --- | --- | --- |
| files | <code>Array.&lt;string&gt;</code> | array of cto files |

<a name="decoratorArgFromMetaModel"></a>

## decoratorArgFromMetaModel(mm) ⇒ <code>string</code>
Create decorator argument string from a metamodel

**Kind**: global function
**Returns**: <code>string</code> - the string for the decorator argument

| Param | Type | Description |
| --- | --- | --- |
| mm | <code>object</code> | the metamodel |

<a name="decoratorFromMetaModel"></a>

## decoratorFromMetaModel(mm) ⇒ <code>string</code>
Create decorator string from a metamodel

**Kind**: global function
**Returns**: <code>string</code> - the string for the decorator

| Param | Type | Description |
| --- | --- | --- |
| mm | <code>object</code> | the metamodel |

<a name="decoratorsFromMetaModel"></a>

## decoratorsFromMetaModel(mm, prefix) ⇒ <code>string</code>
Create decorators string from a metamodel

**Kind**: global function
**Returns**: <code>string</code> - the string for the decorators

| Param | Type | Description |
| --- | --- | --- |
| mm | <code>object</code> | the metamodel |
| prefix | <code>string</code> | indentation |

<a name="propertyFromMetaModel"></a>

## propertyFromMetaModel(mm) ⇒ <code>string</code>
Create a property string from a metamodel

**Kind**: global function
```

```
**Returns**: <code>string</code> - the string for that property

| Param | Type | Description |
| --- | --- | --- |
| mm | <code>object</code> | the metamodel |

<a name="declFromMetaModel"></a>

## declFromMetaModel(mm) ⇒ <code>string</code>
Create a declaration string from a metamodel

**Kind**: global function
**Returns**: <code>string</code> - the string for that declaration

| Param | Type | Description |
| --- | --- | --- |
| mm | <code>object</code> | the metamodel |

<a name="toCTO"></a>

## toCTO(metaModel) ⇒ <code>string</code>
Create a model string from a metamodel

**Kind**: global function
**Returns**: <code>string</code> - the string for that model

| Param | Type | Description |
| --- | --- | --- |
| metaModel | <code>object</code> | the metamodel |

<a name="findNamespace"></a>

## findNamespace(priorModels, namespace) ⇒ <code>\*</code>
Find the model for a given namespace

**Kind**: global function
**Returns**: <code>\*</code> - the model

| Param | Type | Description |
| --- | --- | --- |
| priorModels | <code>\*</code> | known models |
| namespace | <code>string</code> | the namespace |

<a name="findDeclaration"></a>

## findDeclaration(thisModel, name) ⇒ <code>\*</code>
Find a declaration for a given name in a model

**Kind**: global function
**Returns**: <code>\*</code> - the declaration

| Param | Type | Description |
| --- | --- | --- |
| thisModel | <code>\*</code> | the model |
| name | <code>string</code> | the declaration name |

<a name="createNameTable"></a>

## createNameTable(priorModels, metaModel) ⇒ <code>object</code>
```

Create a name resolution table

Kind: global function
Returns: <code>object</code> - mapping from a name to its namespace

Param	Type	Description
priorModels	<code>*</code>	known models
metaModel	<code>object</code>	the metamodel (JSON)

<a name="resolveName"></a>

## resolveName(name, table) ⇒ <code>string</code>
Resolve a name using the name table

Kind: global function
Returns: <code>string</code> - the namespace for that name

Param	Type	Description
name	<code>string</code>	the name of the type to resolve
table	<code>object</code>	the name table

<a name="resolveTypeNames"></a>

## resolveTypeNames(metaModel, table) ⇒ <code>object</code>
Name resolution for metamodel

Kind: global function
Returns: <code>object</code> - the metamodel with fully qualified names

Param	Type	Description
metaModel	<code>object</code>	the metamodel (JSON)
table	<code>object</code>	the name table

<a name="resolveLocalNames"></a>

## resolveLocalNames(priorModels, metaModel) ⇒ <code>object</code>
Resolve the namespace for names in the metamodel

Kind: global function
Returns: <code>object</code> - the resolved metamodel

Param	Type	Description
priorModels	<code>*</code>	known models
metaModel	<code>object</code>	the MetaModel

<a name="resolveLocalNamesForAll"></a>

## resolveLocalNamesForAll(allModels) ⇒ <code>object</code>
Resolve the namespace for names in the metamodel

Kind: global function
Returns: <code>object</code> - the resolved metamodel

Param	Type	Description

```
| allModels | <code>\*</code> | known models |

<a name="inferModelFile"></a>

## inferModelFile(defaultNamespace, defaultType, schema) ⇒ <code>string</code>
Infers a Concerto model from a JSON Schema.

**Kind**: global function
**Returns**: <code>string</code> - the Concerto model

| Param | Type | Description |
| --- | --- | --- |
| defaultNamespace | <code>string</code> | a fallback namespace to use for the
model if it can't be infered |
| defaultType | <code>string</code> | a fallback name for the root concept if it
can't be infered |
| schema | <code>object</code> | the input json object |

<a name="capitalizeFirstLetter"></a>

## capitalizeFirstLetter(string) ⇒ <code>string</code>
Capitalize the first letter of a string

**Kind**: global function
**Returns**: <code>string</code> - input with first letter capitalized

| Param | Type | Description |
| --- | --- | --- |
| string | <code>string</code> | the input string |

<a name="hashCode"></a>

## hashCode(value) ⇒ <code>number</code>
Computes an integer hashcode value for a string

**Kind**: global function
**Returns**: <code>number</code> - the hashcode

| Param | Type | Description |
| --- | --- | --- |
| value | <code>string</code> | the input string |

<a name="isObject"></a>

## isObject(val) ⇒ <code>Boolean</code>
Returns true if val is an object

**Kind**: global function
**Returns**: <code>Boolean</code> - true if val is an object

| Param | Type | Description |
| --- | --- | --- |
| val | <code>\*</code> | the value to test |

<a name="isBoolean"></a>

## isBoolean(val) ⇒ <code>Boolean</code>
Returns true if val is a boolean
```

```
**Kind**: global function
**Returns**: <code>Boolean</code> - true if val is a boolean

| Param | Type | Description |
| --- | --- | --- |
| val | <code>\*</code> | the value to test |

<a name="isNull"></a>

## isNull(val) ⇒ <code>Boolean</code>
Returns true if val is null

**Kind**: global function
**Returns**: <code>Boolean</code> - true if val is null

| Param | Type | Description |
| --- | --- | --- |
| val | <code>\*</code> | the value to test |

<a name="isArray"></a>

## isArray(val) ⇒ <code>Boolean</code>
Returns true if val is an array

**Kind**: global function
**Returns**: <code>Boolean</code> - true if val is an array

| Param | Type | Description |
| --- | --- | --- |
| val | <code>\*</code> | the value to test |

<a name="isString"></a>

## isString(val) ⇒ <code>Boolean</code>
Returns true if val is a string

**Kind**: global function
**Returns**: <code>Boolean</code> - true if val is a string

| Param | Type | Description |
| --- | --- | --- |
| val | <code>\*</code> | the value to test |

<a name="isDateTime"></a>

## isDateTime(val) ⇒ <code>Boolean</code>
Returns true if val is a date time

**Kind**: global function
**Returns**: <code>Boolean</code> - true if val is a string

| Param | Type | Description |
| --- | --- | --- |
| val | <code>\*</code> | the value to test |

<a name="isInteger"></a>

## isInteger(val) ⇒ <code>Boolean</code>
Returns true if val is an integer
```

Kind: global function
Returns: <code>Boolean</code> - true if val is a string

Param	Type	Description
val	<code>*</code>	the value to test

<a name="isDouble"></a>

## isDouble(val) ⇒ <code>Boolean</code>
Returns true if val is an integer

Kind: global function
Returns: <code>Boolean</code> - true if val is a string

Param	Type	Description
val	<code>*</code>	the value to test

<a name="getType"></a>

## getType(input) ⇒ <code>string</code>
Get the primitive Concerto type for an input

Kind: global function
Returns: <code>string</code> - the Concerto type

Param	Type	Description
input	<code>*</code>	the input object

<a name="handleArray"></a>

## handleArray(typeName, context, input) ⇒ <code>object</code>
Handles an array

Kind: global function
Returns: <code>object</code> - the type for the array

Param	Type	Description
typeName	<code>*</code>	the name of the type being processed
context	<code>*</code>	the processing context
input	<code>*</code>	the input object

<a name="handleType"></a>

## handleType(name, context, input) ⇒ <code>object</code>
Handles an input type

Kind: global function
Returns: <code>object</code> - an object for the type

Param	Type	Description
name	<code>*</code>	the name of the type being processed
context	<code>*</code>	the processing context
input	<code>*</code>	the input object

```
<a name="removeDuplicateTypes"></a>

## removeDuplicateTypes(context)
Detect duplicate types and remove them

**Kind**: global function

| Param | Type | Description |
| --- | --- | --- |
| context | <code>\*</code> | the context |

<a name="inferModel"></a>

## inferModel(namespace, rootTypeName, input) ⇒ <code>string</code>
Infers a Concerto model from a JSON instance.

**Kind**: global function
**Returns**: <code>string</code> - the Concerto model

| Param | Type | Description |
| --- | --- | --- |
| namespace | <code>string</code> | the namespace to use for the model |
| rootTypeName | <code>\*</code> | the name for the root concept |
| input | <code>\*</code> | the input json object |

<a name="labelToSentence"></a>

## labelToSentence(labelName) ⇒ <code>string</code>
Inserts correct spacing and capitalization to a camelCase label

**Kind**: global function
**Returns**: <code>string</code> - - The label text formatted for rendering

| Param | Type | Description |
| --- | --- | --- |
| labelName | <code>string</code> | the label text to be transformed |

<a name="sentenceToLabel"></a>

## sentenceToLabel(sentence) ⇒ <code>string</code>
Create a camelCase label from a sentence

**Kind**: global function
**Returns**: <code>string</code> - - The camelCase label

| Param | Type | Description |
| --- | --- | --- |
| sentence | <code>string</code> | the sentence |

<a name="writeModelsToFileSystem"></a>

## writeModelsToFileSystem(files, path, options)
Writes a set of model files to disk

**Kind**: global function

| Param | Type | Description |
| --- | --- | --- |
| files | <code>\*</code> | the set of files to write, with names and whether they
are external |
| path | <code>string</code> | a path to the directory where to write the files |
| options | <code>\*</code> | a set of options |

<a name="camelCaseToSentence"></a>

## camelCaseToSentence(text) ⇒ <code>string</code>
Converts a camel case string to a sentence

**Kind**: global function
**Returns**: <code>string</code> - modified string

| Param | Type | Description |
| --- | --- | --- |
| text | <code>string</code> | input |


--------------------------------------------------------------------------------
---
id: version-0.23.0-ref-ergo-api
title: Ergo API
original_id: ref-ergo-api
---

## Classes

<dl>
<dt><a href="#Commands">Commands</a></dt>
<dd><p>Utility class that implements the commands exposed by the Ergo CLI.</p>
</dd>
</dl>

## Functions

<dl>
<dt><a href="#getJson">getJson(input)</a> ⇒ <code>object</code></dt>
<dd><p>Load a file or JSON string</p>
</dd>
<dt><a href="#loadTemplate">loadTemplate(template, files)</a> ⇒
<code>Promise.&lt;LogicManager&gt;</code></dt>
<dd><p>Load a template from directory or files</p>
</dd>
<dt><a href="#fromDirectory">fromDirectory(path, [options])</a> ⇒
<code>Promise.&lt;LogicManager&gt;</code></dt>
<dd><p>Builds a LogicManager from a directory.</p>
</dd>
<dt><a href="#fromZip">fromZip(buffer, [options])</a> ⇒
<code>Promise.&lt;LogicManager&gt;</code></dt>
<dd><p>Builds a LogicManager from a Zip.</p>
</dd>
<dt><a href="#fromFiles">fromFiles(files, [options])</a> ⇒
<code>Promise.&lt;LogicManager&gt;</code></dt>
<dd><p>Builds a LogicManager from files.</p>
</dd>
<dt><a href="#validateContract">validateContract(modelManager, contract, utcOffset,
options)</a> ⇒ <code>object</code></dt>
<dd><p>Validate contract JSON</p>
</dd>
```

```
<dt><a href="#validateInput">validateInput(modelManager, input, utcOffset)</a> ⇒
<code>object</code></dt>
<dd><p>Validate input JSON</p>
</dd>
<dt><a href="#validateStandard">validateStandard(modelManager, input,
utcOffset)</a> ⇒ <code>object</code></dt>
<dd><p>Validate standard</p>
</dd>
<dt><a href="#validateInputRecord">validateInputRecord(modelManager, input,
utcOffset)</a> ⇒ <code>object</code></dt>
<dd><p>Validate input JSON record</p>
</dd>
<dt><a href="#validateOutput">validateOutput(modelManager, output, utcOffset)</a> ⇒
<code>object</code></dt>
<dd><p>Validate output JSON</p>
</dd>
<dt><a href="#validateOutputArray">validateOutputArray(modelManager, output,
utcOffset)</a> ⇒ <code>Array.&lt;object&gt;</code></dt>
<dd><p>Validate output JSON array</p>
</dd>
<dt><a href="#init">init(engine, logicManager, contractJson, currentTime,
utcOffset)</a> ⇒ <code>object</code></dt>
<dd><p>Invoke Ergo contract initialization</p>
</dd>
<dt><a href="#trigger">trigger(engine, logicManager, contractJson, stateJson,
currentTime, utcOffset, requestJson)</a> ⇒ <code>object</code></dt>
<dd><p>Trigger the Ergo contract with a request</p>
</dd>
<dt><a href="#resolveRootDir">resolveRootDir(parameters)</a> ⇒
<code>string</code></dt>
<dd><p>Resolve the root directory</p>
</dd>
<dt><a href="#compareComponent">compareComponent(expected, actual)</a></dt>
<dd><p>Compare actual and expected result components</p>
</dd>
<dt><a href="#compareSuccess">compareSuccess(expected, actual)</a></dt>
<dd><p>Compare actual result and expected result</p>
</dd>
</dl>

<a name="Commands"></a>

## Commands
Utility class that implements the commands exposed by the Ergo CLI.

**Kind**: global class

* [Commands](#Commands)
    * [.trigger(template, files, contractInput, stateInput, [currentTime],
[utcOffset], requestsInput, warnings)](#Commands.trigger) ⇒ <code>object</code>
    * [.invoke(template, files, clauseName, contractInput, stateInput,
[currentTime], [utcOffset], paramsInput, warnings)](#Commands.invoke) ⇒
<code>object</code>
    * [.initialize(template, files, contractInput, [currentTime], [utcOffset],
paramsInput, warnings)](#Commands.initialize) ⇒ <code>object</code>
    * [.parseCTOtoFileSync(ctoPath)](#Commands.parseCTOtoFileSync) ⇒
<code>string</code>
    * [.parseCTOtoFile(ctoPath)](#Commands.parseCTOtoFile) ⇒ <code>string</code>
```

<a name="Commands.trigger"></a>

### Commands.trigger(template, files, contractInput, stateInput, [currentTime], [utcOffset], requestsInput, warnings) ⇒ <code>object</code>
Send a request an Ergo contract

Kind: static method of [<code>Commands</code>](#Commands)
Returns: <code>object</code> - Promise to the result of execution

Param	Type	Description
template	<code>string</code>	template directory
files	<code>Array.&lt;string&gt;</code>	input files
contractInput	<code>string</code>	the contract data
stateInput	<code>string</code>	the contract state
[currentTime]	<code>string</code>	the definition of 'now', defaults to current time
[utcOffset]	<code>number</code>	UTC Offset for this execution, defaults to local offset
requestsInput	<code>Array.&lt;string&gt;</code>	the requests
warnings	<code>boolean</code>	whether to print warnings

<a name="Commands.invoke"></a>

### Commands.invoke(template, files, clauseName, contractInput, stateInput, [currentTime], [utcOffset], paramsInput, warnings) ⇒ <code>object</code>
Invoke an Ergo contract's clause

Kind: static method of [<code>Commands</code>](#Commands)
Returns: <code>object</code> - Promise to the result of invocation

Param	Type	Description
template	<code>string</code>	template directory
files	<code>Array.&lt;string&gt;</code>	input files
clauseName	<code>string</code>	the name of the clause to invoke
contractInput	<code>string</code>	the contract data
stateInput	<code>string</code>	the contract state
[currentTime]	<code>string</code>	the definition of 'now', defaults to current time
[utcOffset]	<code>number</code>	UTC Offset for this execution, defaults to local offset
paramsInput	<code>object</code>	the parameters for the clause
warnings	<code>boolean</code>	whether to print warnings

<a name="Commands.initialize"></a>

### Commands.initialize(template, files, contractInput, [currentTime], [utcOffset], paramsInput, warnings) ⇒ <code>object</code>
Invoke init for an Ergo contract

Kind: static method of [<code>Commands</code>](#Commands)
Returns: <code>object</code> - Promise to the result of execution

Param	Type	Description
template	<code>string</code>	template directory
files	<code>Array.&lt;string&gt;</code>	input files
contractInput	<code>string</code>	the contract data

```
| [currentTime] | <code>string</code> | the definition of 'now', defaults to
current time |
| [utcOffset] | <code>number</code> | UTC Offset for this execution, defaults to
local offset |
| paramsInput | <code>object</code> | the parameters for the clause |
| warnings | <code>boolean</code> | whether to print warnings |

<a name="Commands.parseCTOtoFileSync"></a>

### Commands.parseCTOtoFileSync(ctoPath) ⇒ <code>string</code>
Parse CTO to JSON File

**Kind**: static method of [<code>Commands</code>](#Commands)
**Returns**: <code>string</code> - The name of the generated CTOJ model file

| Param | Type | Description |
| --- | --- | --- |
| ctoPath | <code>string</code> | path to CTO model file |

<a name="Commands.parseCTOtoFile"></a>

### Commands.parseCTOtoFile(ctoPath) ⇒ <code>string</code>
Parse CTO to JSON File

**Kind**: static method of [<code>Commands</code>](#Commands)
**Returns**: <code>string</code> - The name of the generated CTOJ model file

| Param | Type | Description |
| --- | --- | --- |
| ctoPath | <code>string</code> | path to CTO model file |

<a name="getJson"></a>

## getJson(input) ⇒ <code>object</code>
Load a file or JSON string

**Kind**: global function
**Returns**: <code>object</code> - JSON object

| Param | Type | Description |
| --- | --- | --- |
| input | <code>object</code> | either a file name or a json string |

<a name="loadTemplate"></a>

## loadTemplate(template, files) ⇒ <code>Promise.&lt;LogicManager&gt;</code>
Load a template from directory or files

**Kind**: global function
**Returns**: <code>Promise.&lt;LogicManager&gt;</code> - a Promise to the
instantiated logicmanager

| Param | Type | Description |
| --- | --- | --- |
| template | <code>string</code> | template directory |
| files | <code>Array.&lt;string&gt;</code> | input files |

<a name="fromDirectory"></a>
```

## fromDirectory(path, [options]) ⇒ <code>Promise.&lt;LogicManager&gt;</code>
Builds a LogicManager from a directory.

Kind: global function
Returns: <code>Promise.&lt;LogicManager&gt;</code> - a Promise to the instantiated logicmanager

Param	Type	Description
path	<code>String</code>	to a local directory
[options]	<code>Object</code>	an optional set of options to configure the instance.

<a name="fromZip"></a>

## fromZip(buffer, [options]) ⇒ <code>Promise.&lt;LogicManager&gt;</code>
Builds a LogicManager from a Zip.

Kind: global function
Returns: <code>Promise.&lt;LogicManager&gt;</code> - a Promise to the instantiated logicmanager

Param	Type	Description
buffer	<code>Buffer</code>	the buffer to a Zip (zip) file
[options]	<code>Object</code>	an optional set of options to configure the instance.

<a name="fromFiles"></a>

## fromFiles(files, [options]) ⇒ <code>Promise.&lt;LogicManager&gt;</code>
Builds a LogicManager from files.

Kind: global function
Returns: <code>Promise.&lt;LogicManager&gt;</code> - a Promise to the instantiated logicmanager

Param	Type	Description
files	<code>Array.&lt;String&gt;</code>	file names
[options]	<code>Object</code>	an optional set of options to configure the instance.

<a name="validateContract"></a>

## validateContract(modelManager, contract, utcOffset, options) ⇒ <code>object</code>
Validate contract JSON

Kind: global function
Returns: <code>object</code> - the validated contract

Param	Type	Description
modelManager	<code>object</code>	the Concerto model manager
contract	<code>object</code>	the contract JSON
utcOffset	<code>number</code>	UTC Offset for DateTime values
options	<code>object</code>	parameters for contract variables validation

```
<a name="validateInput"></a>

## validateInput(modelManager, input, utcOffset) ⇒ <code>object</code>
Validate input JSON

**Kind**: global function
**Returns**: <code>object</code> - the validated input

| Param | Type | Description |
| --- | --- | --- |
| modelManager | <code>object</code> | the Concerto model manager |
| input | <code>object</code> | the input JSON |
| utcOffset | <code>number</code> | UTC Offset for DateTime values |

<a name="validateStandard"></a>

## validateStandard(modelManager, input, utcOffset) ⇒ <code>object</code>
Validate standard

**Kind**: global function
**Returns**: <code>object</code> - the validated input

| Param | Type | Description |
| --- | --- | --- |
| modelManager | <code>object</code> | the Concerto model manager |
| input | <code>object</code> | the input JSON |
| utcOffset | <code>number</code> | UTC Offset for DateTime values |

<a name="validateInputRecord"></a>

## validateInputRecord(modelManager, input, utcOffset) ⇒ <code>object</code>
Validate input JSON record

**Kind**: global function
**Returns**: <code>object</code> - the validated input

| Param | Type | Description |
| --- | --- | --- |
| modelManager | <code>object</code> | the Concerto model manager |
| input | <code>object</code> | the input JSON record |
| utcOffset | <code>number</code> | UTC Offset for DateTime values |

<a name="validateOutput"></a>

## validateOutput(modelManager, output, utcOffset) ⇒ <code>object</code>
Validate output JSON

**Kind**: global function
**Returns**: <code>object</code> - the validated output

| Param | Type | Description |
| --- | --- | --- |
| modelManager | <code>object</code> | the Concerto model manager |
| output | <code>object</code> | the output JSON |
| utcOffset | <code>number</code> | UTC Offset for DateTime values |

<a name="validateOutputArray"></a>

## validateOutputArray(modelManager, output, utcOffset) ⇒
```

```
<code>Array.&lt;object&gt;</code>
Validate output JSON array

**Kind**: global function
**Returns**: <code>Array.&lt;object&gt;</code> - the validated output array

| Param | Type | Description |
| --- | --- | --- |
| modelManager | <code>object</code> | the Concerto model manager |
| output | <code>\*</code> | the output JSON array |
| utcOffset | <code>number</code> | UTC Offset for DateTime values |

<a name="init"></a>

## init(engine, logicManager, contractJson, currentTime, utcOffset) ⇒
<code>object</code>
Invoke Ergo contract initialization

**Kind**: global function
**Returns**: <code>object</code> - Promise to the initial state of the contract

| Param | Type | Description |
| --- | --- | --- |
| engine | <code>object</code> | the execution engine |
| logicManager | <code>object</code> | the Template Logic |
| contractJson | <code>object</code> | contract data in JSON |
| currentTime | <code>string</code> | the definition of 'now' |
| utcOffset | <code>utcOffset</code> | UTC Offset for this execution |

<a name="trigger"></a>

## trigger(engine, logicManager, contractJson, stateJson, currentTime, utcOffset,
requestJson) ⇒ <code>object</code>
Trigger the Ergo contract with a request

**Kind**: global function
**Returns**: <code>object</code> - Promise to the response

| Param | Type | Description |
| --- | --- | --- |
| engine | <code>object</code> | the execution engine |
| logicManager | <code>object</code> | the Template Logic |
| contractJson | <code>object</code> | contract data in JSON |
| stateJson | <code>object</code> | state data in JSON |
| currentTime | <code>string</code> | the definition of 'now' |
| utcOffset | <code>utcOffset</code> | UTC Offset for this execution |
| requestJson | <code>object</code> | state data in JSON |

<a name="resolveRootDir"></a>

## resolveRootDir(parameters) ⇒ <code>string</code>
Resolve the root directory

**Kind**: global function
**Returns**: <code>string</code> - root directory used to resolve file names

| Param | Type | Description |
| --- | --- | --- |
| parameters | <code>string</code> | Cucumber's World parameters |
```

<a name="compareComponent"></a>

## compareComponent(expected, actual)
Compare actual and expected result components

Kind: global function

Param	Type	Description
expected	<code>string</code>	the expected component as specified in the test workload
actual	<code>string</code>	the actual component as returned by the engine

<a name="compareSuccess"></a>

## compareSuccess(expected, actual)
Compare actual result and expected result

Kind: global function

Param	Type	Description
expected	<code>string</code>	the expected successful result as specified in the test workload
actual	<code>string</code>	the successful result as returned by the engine

```
-----------------------------------------------------------------------------------
---
id: version-0.23.0-ref-migrate-concerto-1.0-2.0
title: Concerto 1.0 to 2.0
original_id: ref-migrate-concerto-1.0-2.0
---
```

Concerto `2.0` delivers fundamental improvements over previous releases, whilst
maintaining a high-degree (though not total!) of backwards compatibility with
`1.x`. In particular all of the `1.x` Concerto syntax remains valid in `2.0`.

> The release includes over 75 commits, and over 400 files changed. Thank you to
all the contributors!

:::note
We are currently in the process of migrating the Accord Project stack to Concero
v2.0. While the migration is underway you may see some components that still depend
upon Concerto v1.x.
:::

## Summary of Changes
- Update the Concerto metamodel to [version
0.3](https://models.accordproject.org/concerto/metamodel@0.3.0.html)
- Migrate the Concerto parser from pegjs (no longer maintained) to [peggy]
(https://peggyjs.org)
- Improvements to Typescript type definitions
- Fixes for JSON Schema generation
- Drop support for Node 12, adding support for Node 16
- Re-organize the code to make `concerto-core` independent of the CTO concrete
syntax, moving parsing and CTO generation into the new `concerto-cto` package.
- Add `concerto-util` package for common code

- Add `concerto-vocabulary` package, for managing localized terms for models
- Add `DecoratorManager` to allow decorations on model to be externalized and applied to models

## Summary of API Changes
- Added method `declarationKind()` to concept/asset etc to determine the type
- Removed the method `hasInstance` to perform instanceof checks
- `ModelFile.getAst` to return the metamodel for a model
- `ModelManager.addCTOModel` to add a model as a CTO string to a model manager
- `BaseModelManager` to manager models, independent of CTO syntax
- `BaseModelManager.getAst` to get metamodel for a set of models
- `BaseModelManager.fromAst` to create a ModelManager from a metamodel

---
id: version-0.30.0-markup-commonmark
title: CommonMark
original_id: markup-commonmark

The following CommonMark guide is non normative, but included for convenience. For a more detailed introduction we refer the reader the [CommonMark Webpage](https://commonmark.org/) and [Specification](https://spec.commonmark.org/0.29/).

## Formatting

### Italics

To italicize text, add one asterisk `*` or underscore `_` both before and after the relevant text.

##### Example

```md
_Donoghue v Stevenson_ is a landmark tort law case.
```
will be rendered as:

> _Donoghue v Stevenson_ is a landmark tort law case.

### Bold
To bold text, add two asterisks `**` or two underscores `__` both before and after the relevant text.

##### Example

```md
**Price** is defined in the Appendix.
```

will be rendered as:

> **Price** is defined in the Appendix.

### Bold and Italic
To bold _and_ italicize text, add `***` both before and after the relevant text.

~~~
##### Example

```md
***WARNING***: This product contains chemicals that may cause cancer.
```
will be rendered as:

> ***WARNING***: This product contains chemicals that may cause cancer.

## Paragraphs
To start a new paragraph, insert one or more blank lines. (In other words, all
paragraphs in markdown need to have one or more blank lines between them.)

##### Example

```md
This is the first paragraph.

This is the second paragraph.
This is not a third paragraph.
```
will be rendered as:

>This is the first paragraph.
>
>This is the second paragraph.
>This is not a third paragraph.


## Headings

### Using `#` (ATX Headings)

Level-1 through level-6 headings from are written with a `#` for each level.

#### Example

```md
# US Constitution
## Statutes enacted by Congress
### Rules promulgated by federal agencies
#### State constitution
##### Laws enacted by state legislature
###### Local laws and ordinances
```

will be rendered as:
> <h1>US Constitution</h1>
> <h2>Statutes enacted by Congress</h2>
> <h3>Rules promulgated by federal agencies</h3>
> <h4>State constitution</h4>
> <h5>Laws enacted by state legislature</h5>
> <h6>Local laws and ordinances</h6>

### Using `=` or `-` (Setext Headings)

Alternatively, headings with level 1 or 2 can be represented by using `=` and `-`
under the text of the heading.
~~~

~~~
#### Example

```md
Linux Foundation
================

Accord Project
--------------
```

will be rendered as:
> <h1>Linux Foundation</h1>
> <h2>Accord Project</h2>

## Lists

### Unordered Lists
To create an unordered list, use asterisks `*`, plus `+`, or hyphens `-` in the
beginning as list markers.

#### Example

```md
* Cicero
* Ergo
* Concerto
```

Will be rendered as:
>* Cicero
>* Ergo
>* Concerto

### Ordered Lists

To create an ordered list, use numbers followed by a period `.`.

#### Example

```md
1. One
2. Two
3. Three
```

will be rendered as:
>1. One
>2. Two
>3. Three

### Nested Lists

To create a list within another, indent each item in the sublist by four spaces.

#### Example
```md
1. Matters related to the business
    - enter into an agreement...
    - enter into any abnormal contracts...
~~~

~~~
2. Matters related to the assets
    - sell or otherwise dispose...
    - mortage, ...
```

will be rendered as:
>1. Matters related to the business
>    - enter into an agreement...
>    - enter into any abnormal contracts...
>2. Matters related to the assets
>    - sell or otherwise dispose...
>    - mortgage, ...

## Tables

To create a table, use pipes `|` to separate each column and use three or more
hyphens `---` for each column's header. For compatibility, you should not create a
table without a header and add also add a pipe on either end of a row.

#### Example

```md
| Header 1    | Header 2    |
| ----------- | ----------- |
| Column 1    | Column 2    |
```

will be rendered as

>| Header 1    | Header 2    |
>| ----------- | ----------- |
>| Column 1    | Column 2    |

It is not necessary to have identical cell widths for the whole table. The rendered
output will look the same irrespective of varying cell widths.

```md
| Header 1    | Header 2    |
| ---------| ---------|
| Column 1    | Column 2      |
```

will be rendered as

>| Header 1    | Header 2    |
>| ---------| ---------|
>| Column 1    | Column 2      |

### Formatting the Tables

A table can contain links, code (words or phrases in backticks (`) only) ,
formatted text (bold, italics) or images. However, adding lists, headings,
blockquotes, code blocks, horizontal rules or nested tables is not possible.

#### Example

```md
| Column1     | Column 2    |
| ----------- | ----------- |
~~~

~~~~
| text | ![ap_logo](https://docs.accordproject.org/docs/assets/020/template.png "AP
triangle")       |
| \`\`\`code block\`\`\`   | **Bold content**     |
| [link](http://clause.io) | *Italics* |
```
will be rendered as

>| Column1     | Column 2    |
>| ----------- | ----------- |
>| text | ![ap_logo](https://docs.accordproject.org/docs/assets/020/template.png
"AP triangle")       |
>| \`\`\`code block\`\`\`   | **Bold content**     |
>| [link](http://clause.io) | *Italics* |

## Horizontal Rule

A horizontal rule may be used to create a "thematic break" between paragraph-level
elements. In markdown, you can create a thematic break using either of the
following:

* `___`: three consecutive underscores
* `---`: three consecutive dashes
* `***`: three consecutive asterisks

#### Example

```md
___
---
***
```

Will be rendered as:
>___
>
>---
>
>***

## Escaping

Any markdown character that is used for a special purpose may be _escaped_ by
placing a backslash in front of it.

For instance avoid creating bold or italic when using `*` or `_` in a sentence,
place a backslash `\` in the front, like: `\*` or `\_`.

#### Example

```md
This is \_not\_ italics but _this_ is!
```
Will be rendered as:
> This is \_not\_ italics but _this_ is!


<!--References:
Commonmark official page and tutorial: https://commonmark.org/help/
OpenLaw Beginner's Guide: https://docs.openlaw.io/beginners-guide/
~~~~

~~~
Markdown cheatsheet: https://gist.github.com/jonschlinkert/5854601
Headings example:
http://www.nyc.gov/html/conflicts/downloads/pdf2/municipal_ethics_laws_ny_state/
introduction_to_american_law.pdf
-->

--------------------------------------------------------------------------------
---
id: version-0.30.0-model-api
title: Using the API
original_id: model-api
---

## Install the Core Library

To install the core model library in your project:
```
npm install @accordproject/concerto-core --save
```

Below are examples of API use.

## Validating JSON data using a Model

```js
const ModelManager = require('@accordproject/concerto-core').ModelManager;
const Concerto = require('@accordproject/concerto-core').Concerto;
const modelManager = new ModelManager();
modelManager.addCTOModel( `namespace org.acme.address
concept PostalAddress {
  o String streetAddress optional
  o String postalCode optional
  o String postOfficeBoxNumber optional
  o String addressRegion optional
  o String addressLocality optional
  o String addressCountry optional
}`, 'model.cto');

const postalAddress = {
    $class : 'org.acme.address.PostalAddress',
    streetAddress : '1 Maine Street'
};
const concerto = new Concerto(modelManager);
concerto.validate(postalAddress);
```

Now try validating this instance:

```
const postalAddress = {
    $class : 'org.acme.address.PostalAddress',
    missing : '1 Maine Street'
};
```

Validation should fail with the message:

```
Instance undefined has a property named missing which is not declared in
~~~

org.acme.address.PostalAddress
```

Runtime introspection of the model

You can use the Concerto `introspect` APIs to retrieve model information at runtime:

```
const typeDeclaration = concerto.getTypeDeclaration(postalAddress);
const fqn = typeDeclaration.getFullyQualifiedName();
console.log(fqn); // should equal 'org.acme.address.PostalAddress'

These APIs allow you to examine the declared properties, super types and meta-properies for a modelled type.
--------------------------------------------------------------------------------
---
id: version-0.30.0-model-namespaces
title: Namespaces
original_id: model-namespaces
---

Each Concerto file starts with the name and version of a single namespace. A Concerto namespace declares a set of *declarations*. A declaration is one of: enumeration, concept, asset, participant, transaction, event. All declarations within a single file belong to the same namespace.

```js
namespace org.acme@1.0.0 // declares version 1.0.0 of the org.acme namespace
```

### Imports

In order for one namespace to reference types defined in another namespace, the types must be imported for a version of a namespace.

## Simple

```js
import org.accordproject.address@1.0.0.PostalAddress // imports PostalAddress from version 1.0.0 of the org.accordproject.address namespace
```

## Multiple Imports

To import multiple types from the same namespace, use the `{}` syntax:

```js
import org.accordproject.address@1.0.0.{PostalAddress,Country} // imports PostalAddress and Country from version 1.0.0 of the org.accordproject.address namespace
```

## Importing from model published to a public URL

Import also can use the optional `from` declaration to import a model file that has been deployed to a URL.

```js
import org.accordproject.address@1.0.0.PostalAddress from
https://models.accordproject.org/address.cto
```

Imports using a `from` declaration can be downloaded into the model manager by
calling `modelManager.updateExternalModels`.

The Model Manager will resolve all imports to ensure that the set of declarations
that have been loaded are globally consistent.

## Strict:false mode

For backwards compatability, and when running with `strict:false` imports may
import types from unversioned namespaces, or may import all types in a namespace.

> Please migrate models to use versioned namespaces and imports as this capability
will be deprecated and removed in a future major release.

Imports can be either qualified or can use wildcards.

```js
import org.accordproject.address.PostalAddress // import a type from an unversioned
namespace
import org.accordproject.address.* // import all types from an unversioned
namespace
```

---
id: version-0.30.0-ref-concerto-api
title: Concerto API
original_id: ref-concerto-api

---
id: version-0.30.0-ref-concerto-cli
title: Command Line
original_id: ref-concerto-cli

Install the `@accordproject/concerto-cli` npm package to access the Concerto
command line interface (CLI). After installation you can use the `concerto` command
and its sub-commands as described below.

To install the Concerto CLI:
```
npm install -g @accordproject/concerto-cli
```

## Usage

```md
concerto <cmd> [args]

Commands:
```

  concerto validate   validate JSON against model files
  concerto compile   generate code for a target platform
  concerto get   save local copies of external model dependencies
  concerto parse   parse a cto string to a JSON syntax tree
  concerto print   print a JSON syntax tree to a cto string
  concerto version <release> modify the version of one or more model files
  concerto compare   compare two Concerto model files
  concerto infer   generate a concerto model from a source schema
  concerto generate <mode>   generate a sample JSON object for a concept

Options:
      --version Show version number   [boolean]
  -v, --verbose   [default: false]
      --help   Show help   [boolean]

## concerto validate
`concerto validate` lets you check whether a JSON sample is a valid instance of the given model.

```md
concerto validate

validate JSON against model files

Options:
      --version     Show version number                               [boolean]
  -v, --verbose                                                [default: false]
      --help        Show help                                         [boolean]
      --input       JSON to validate                                   [string]
      --model       array of concerto model files                       [array]
      --utcOffset   set UTC offset                                     [number]
      --offline     do not resolve external models    [boolean] [default: false]
      --functional  new validation API                [boolean] [default: false]
      --ergo        validation and emit for Ergo      [boolean] [default: false]
```

### Example
For example, using the `validate` command to check the sample `request.json` file from a [Late Delivery and Penalty](https://github.com/accordproject/cicero-template-library/tree/master/src/latedeliveryandpenalty) clause:

```
concerto validate --input request.json --model model/clause.cto
```

returns:

```json
{
  "$class":
"org.accordproject.latedeliveryandpenalty.LateDeliveryAndPenaltyRequest",
  "forceMajeure": false,
  "agreedDelivery": "2017-12-17T04:24:00.000-04:00",
  "goodsValue": 200,
  "$timestamp": "2021-06-17T09:41:54.207-04:00"
}
```

## concerto compile
`concerto compile` takes an array of local CTO files, downloads any external
dependencies (imports) and then converts all the model to the target format.

```md
concerto compile

generate code for a target platform

Options:
      --version            Show version number                         [boolean]
  -v, --verbose                                                 [default: false]
      --help               Show help                                   [boolean]
      --model              array of concerto model files   [array] [default: []]
      --offline            do not resolve external models
                                                      [boolean] [default: false]
      --target             target of the code generation
                                                [string] [default: "JSONSchema"]
      --output             output directory path [string] [default: "./output/"]
      --metamodel          Include the Concerto Metamodel in the output
                                                      [boolean] [default: false]
      --strict             Require versioned namespaces and imports
                                                      [boolean] [default: false]
      --useSystemTextJson  Compile for System.Text.Json library (`csharp` target
                           only)                      [boolean] [default: false]
      --useNewtonsoftJson  Compile for Newtonsoft.Json library (`csharp` target
                           only)                      [boolean] [default: false]
      --namespacePrefix    A prefix to add to all namespaces (`csharp` target
                           only)                                        [string]
      --pascalCase         Use PascalCase for generated identifier names
                                                       [boolean] [default: true]
```

At the moment, the available target formats are as follows:
- Go Lang: `concerto compile --model modelfile.cto --target Golang`
- JSONSchema: `concerto compile --model modelfile.cto --target JSONSchema`
- XMLSchema: `concerto compile --model modelfile.cto --target XMLSchema`
- Plant UML: `concerto compile --model modelfile.cto --target PlantUML`
- Typescript: `concerto compile --model modelfile.cto --target Typescript`
- Java: `concerto compile --model modelfile.cto --target Java`
- GraphQL: `concerto compile --model modelfile.cto --target GraphQL`
- CSharp: `concerto compile --model modelfile.cto --target CSharp`
- OData: `concerto compile --model modelfile.cto --target OData`
- Mermaid: `concerto compile --model modelfile.cto --target Mermaid`
- Markdown: `concerto compile --model modelfile.cto --target Markdown`

### Example
For example, using the `compile` command to export the `clause.cto` file from a
[Late Delivery and Penalty](https://github.com/accordproject/cicero-template-library/tree/master/src/latedeliveryandpenalty) clause into `Go Lang` format:

```md
cd ./model
concerto compile --model clause.cto --target Golang
```

returns:
```md
info: Compiled to Go in './output/'.
```

~~~~
```

## concerto get
`concerto get` allows you to resolve and download external models from a set of
local CTO files.

```md
concerto get

save local copies of external model dependencies

Options:
      --version  Show version number                                   [boolean]
  -v, --verbose                                                 [default: false]
      --help     Show help                                             [boolean]
      --model    array of concerto (cto) model files          [array] [required]
      --output   output directory path                  [string] [default: "./"]
```

### Example
For example, using the `get` command to get the external models in the `clause.cto`
file from a [Late Delivery and Penalty](https://github.com/accordproject/cicero-
template-library/tree/master/src/latedeliveryandpenalty) clause:

```md
concerto get --model clause.cto
```

returns:
```md
info: Loaded external models in './'.
```

## concerto parse
`concerto parse` allows you to parse a set of CTO models to their JSON
representation (metamodel).

```md
parse a cto string to a JSON syntax tree

Options:
      --version               Show version number                      [boolean]
  -v, --verbose                                                 [default: false]
      --help                  Show help                                [boolean]
      --model                 array of concerto model files   [array] [required]
      --resolve               resolve names to fully qualified names
                                                      [boolean] [default: false]
      --all                   import all models       [boolean] [default: false]
      --output                path to the output file                   [string]
      --excludeLineLocations  Exclude file line location metadata from metamodel
                              instance                [boolean] [default: false]
```

## concerto print
`concerto print` allows you to convert a model in JSON metamodel format to a CTO
string.

```md
concerto print
~~~~

~~~~
print a JSON syntax tree to a cto string

Options:
      --version  Show version number                                   [boolean]
  -v, --verbose                                                 [default: false]
      --help     Show help                                             [boolean]
      --input    the metamodel to export                     [string] [required]
      --output   path to the output file                                [string]
```

## concerto version
`concerto version` allows you to modify the version of one or more model files

```md
concerto version <release>

modify the version of one or more model files

Positionals:
  release  the new version, or a release to use when incrementing the existing
           version
    [string] [required] [choices: "keep", "major", "minor", "patch", "premajor",
                                           "preminor", "prepatch", "prerelease"]

Options:
      --version          Show version number                           [boolean]
  -v, --verbose                                                 [default: false]
      --help             Show help                                     [boolean]
      --model, --models  array of concerto model files        [array] [required]
      --prerelease       set the specified pre-release version          [string]
```

## concerto compare
`concerto compare` allows you to compare two model files

```md
concerto compare

compare two Concerto model files

Options:
      --version  Show version number                                   [boolean]
  -v, --verbose                                                 [default: false]
      --help     Show help                                             [boolean]
      --old      the old Concerto model file                 [string] [required]
      --new      the new Concerto model file                 [string] [required]
```

## concerto infer
`concerto infer` allows you to generate a Concerto model from a source schema such
as JSON Schema or an OpenAPI definition.

```md
concerto infer

generate a concerto model from a source schema

Options:
~~~~

```
      --version          Show version number                          [boolean]
  -v, --verbose                                                [default: false]
      --help             Show help                                    [boolean]
      --input            path to the input file          [string] [required]
      --output           path to the output file                     [string]
      --format           either `openapi` or `jsonSchema`
                                          [string] [default: "jsonSchema"]
      --namespace        The namespace for the output model
                                                         [string] [required]
      --typeName         The name of the root type[string] [default: "Root"]
      --capitalizeFirst  Capitalize the first character of type names
                                                [boolean] [default: false]
```

### Example
```console
concerto infer --namespace com.example.restapi --format openapi --input
example.swagger.json --output example.cto
```

## concerto generate
`concerto generate` allows you to generate a sample instance for a type in a model

```md
concerto generate <mode>

generate a sample JSON object for a concept

Positionals:
  mode  Generation mode. `empty` will generate a minimal example, `sample` will
        generate random values  [string] [required] [choices: "sample", "empty"]

Options:
      --version                 Show version number                   [boolean]
  -v, --verbose                                                [default: false]
      --help                    Show help                             [boolean]
      --model                   The file location of the source models
                                                          [array] [required]
      --concept                 The fully qualified name of the Concept type to
                                generate                    [string] [required]
      --includeOptionalFields   Include optional fields will be included in the
                                output              [boolean] [default: false]
      --metamodel               Include the Concerto Metamodel in the output
                                                    [boolean] [default: false]
      --strict                  Require versioned namespaces and imports
                                                    [boolean] [default: false]
```

---
id: version-0.30.0-ref-migrate-concerto-2.0-3.0
title: Concerto 2.0 to 3.0
original_id: ref-migrate-concerto-2.0-3.0

Concerto `3.0` delivers fundamental improvements over previous releases, whilst maintaining a high-degree (though not total!) of backwards compatibility with `2.x`. In particular all of the `2.x` Concerto syntax remains valid in `3.0`.

> View the [Concerto v3.0.0 release changelog](https://github.com/accordproject/concerto/releases/tag/v3.0.0) on GitHub. Thank you to all the contributors!

:::note
We are currently in the process of migrating the Accord Project stack to Concero v3.0. While the migration is underway you may see some components that still depend upon Concerto v2.x.
:::

## Summary of Changes

This new major version allows Concerto models to define an explicit version in a model file, (according to the [Semantic Versioning convention](https://semver.org)). Concerto models can also declare a dependency on an explicit version of another model. This makes it easier to govern changes to model definitions in large-scale deployments.

Version 3 also includes several new features:
- Compatibility Detection. Protect your users when your model changes by ensuring backwards compatibility
- Command Line Tool Enhancements. New concerto generate, concerto version, concerto compare commands.
- .NET Enhancements. Improved code generation for C#. .NET serialization and deserialization tools.

## Strict Mode

One of the major new features in Concerto v3 is the ability to version namespaces, and to import specific versions of a namespace. When Concerto is running in `strict: true` mode **only** versioned namespaces and versioned imports are permitted within CTO files. By default Concerto v3 uses `strict:false`, allowing it to be used with existing (unversioned) CTO files unchanged.

In `strict:true` mode importing all the types in a namespace is prohibited.

E.g.
```
namespace org.acme@1.0.0

import com.sample.model@1.0.0.* // in strict:true mode this is an error

concept Person {
}
```

It is recommended that you migrate your CTO files to use versioned namespaces. In the future the default setting for `strict` will be `true`, likely followed by deprecation of the `strict:false` behavior and eventual removal of support for unversioned namespaces. You have been warned!

## CLI Enhancements

The `concerto-cli` command line utility has been improved and extended, with support for checking semver compatability of models, incrementing namespace versions, generating code from models, and parser performance improvements.

Please see the updated [CLI documentation](ref-concerto-cli.md) for details.

## Core Enhancements

A new `InMemoryWriter` class is provided which adheres to the `FileWriter` interface, while storing files in memory. This makes it easier to integrate code generation into environments without access to a local filesystem.

### Security Enhancements

The regular expression (regex) engine used by core to validate string values is now pluggable, allowing integrators to protect themselves from recursive regular expressions, or other attacks.

## Tools Enhancements

Many improvemements to code generation have been delivered, including support for versioned namespaces, and much improved C# code generations. In addition, two new code generation targets are available:
- [Mermaid](https://mermaid-js.github.io), a textual format to represent class diagrams
- Markdown, a textual format to provide an overview of a model, including a nested Mermaid format diagram, as [supported by GitHub](https://github.blog/2022-02-14-include-diagrams-markdown-files-mermaid/).

## Analysis (New Feature!)

A new package `concerto-analysis` has been delivered, and exposed via the CLI, allowing users to compare two Concerto models, to detect new, updated, and removed model elements. The results of analysis can then be used to update the versions of namespaces, adhering to semantic versioning best practices.

---
id: version-0.30.1-accordproject-faq
title: FAQ
original_id: accordproject-faq

## Accord Project Frequently asked Questions

### What is a "Smart Contract" in the Accord Project?

A “smart” legal contract is a legally binding agreement that is digital and able to connect its terms and the performance of its obligations to external sources of data and software systems. The benefit is to enable a wide variety of efficiencies, automation, and real time visibility for lawyers, businesses, nonprofits, and government. The potential applications of smart legal contracts are limitless. Although the operation of smart legal contracts may be enhanced by using blockchain technology, a blockchain is not necessary, smart legal contracts can operate using traditional software systems without blockchain. A central goal of Accord Project technology is to be blockchain agnostic.

A smart legal contract consists of natural language text with certain parts (e.g. clauses, sections) of the agreement constructed as machine executable components. The libraries provided by the Accord Project enable a document to be:

* Structured as machine readable data objects; and
* Executed on, or integrated with, external systems (e.g. to initiate a payment or update an invoice)

While the Accord Project technology is targeted at the development of smart legal contracts, the open source codebase may also be used to develop other forms of machine-readable and executable documentation.

### How is an Accord Project "Smart Contract" different from "Smart Contracts" on the blockchain?

Accord Project Smart legal contracts should not be confused with so-called blockchain “smart contracts”, which are scripts that necesarily operate on a blockchain. On the blockchain a smart contract is often written in a specific language like solidity that executes and operates on the blockchain. It lives in a closed world. An Accord Project Smart Contract contains text based template that integrates with a data model and the Ergo language. The three components are integrated into a whole. Using Ergo an Accord Project Smart contract can communicate with other systems, it can send and receive data, it can perform calculations and it can interact with a blockchain.

### What benefits do Smart Legal Contracts provide?

Contractual agreements sit at the heart of any organization, governing relationships with employees, shareholders, customers, suppliers, financiers, and more. Yet contracts today are not capable of being efficiently managed as the valuable assets they are. Currently contracts exist as static text documents stored in cloud storage services, dated contract management systems, or even email inboxes. Often these documents are Word files or PDFs that can only be interpreted by humans. A smart legal contract, by contrast, can be interpreted by machines.

Smart Legal Contracts can be easily searched, analyzed, queried, and understood. By associating a data model to a contract, it is possible to extract a host of valuable data about a contract or draft a series of contracts from existing data points (i.e., variables and their values).

The data model is used to ensure that all of the necessary data is present in the contract, and that this data is valid. In addition, it provides the necessary structure to enable contracts to "come alive" by adding executable logic.

The result is a contract that is:

* Searchable
* Analyzable
* Real-time
* Integrated

Consequently, contracts are transformed from business liabilities in constant need of management to assets capable of providing real business intelligence and value. A Smart Contract contains a data model so that the data is part of the contract and not something held in an external system. The logical operations of the contract are also part of the contract. The contract can update itself and react to the outside world. Rather than being stored in filing cabinet it is a living breathing process.

### What is the Accord Project and what is its purpose?

The Accord Project is a non-profit, member-driven organization that builds open source code and documentation for smart legal contracts for use by transactional attorneys, business and finance professionals, and other contract users. Open source means that anyone can use and contribute to the code and documentation and use it in their own software applications and systems free of charge.

The purpose of the Accord Project is to establish and maintain a common and consistent legal and technical foundation for smart legal contracts. The Accord Project is organized into working groups focused on various use cases for Smart Contracts. The specific working groups are assisted by the Technology Working Group, which builds the underlying open source code and specifications to codify the knowledge of the transactional working groups. More details about the internal governance of the Accord Project are available [here](https://github.com/accordproject/governance).

### How can I get involved?

The Accord Project Community is developing several working groups focusing on different applications of smart contracts. The working groups have frequent calls and use the Accord Project's Discord group chat application (join by clicking [here](https://discord.com/invite/Zm99SKhhtA)) for discussion. The dates, dial-in instructions, and agendas for the working groups are all listed in the Project's public calendar and typically also in working group's respective Discord channels.

A primary purpose of the working groups is to develop a universally accessible and widely used open source library of modular, smart legal contracts, smart templates and models that reflect input from the community. Smart legal contract templates are built according to the Project's [Cicero Specification](https://github.com/accordproject/cicero).

Members can provide feedback into the templates and models relevant to a particular working group. You can immediately start contributing smart legal contract templates and models by using the Accord Project's [Template Studio](https://studio.accordproject.org/).

The Accord Project has developed an easy-to-use programming language for building and executing smart legal contracts called Ergo. The goals of Ergo are to be accessible and usable by non-technical professionals, portable across, and compatible with, a variety of environments such as SaaS platforms and different blockchains, and meeting security, safety, and other requirements.

You can use the Accord Project's [Template Studio](https://studio.accordproject.org/) to create and test your smart legal contracts.

```
--------------------------------------------------------------------------------
---
id: version-0.30.1-accordproject-slc
title: Smart Legal Contracts
original_id: accordproject-slc
---
```

A Smart Legal Contract is a human-readable _and_ machine-readable agreement that is digital, consisting of natural language and computable components.

The human-readable nature of the document ensures that signatories, lawyers, contracting parties and others are able to understand the contract.

The machine-readable nature of the document enables it to be interpreted and executed by computers, making the document "smart".

Contracts drafted with Accord Project can contain both traditional and machine-readable clauses. For example, a Smart Legal Contract may include a smart payment clause while all of the other provisions of the contract (Definitions, Jurisdiction clause, Force Majeure clause, ...) are being documented solely in regular natural language text.

A Smart Legal Contract is a general term to refer to two compatible, architectural forms of contract:
- Machine-Readable Contracts, which tie legal text to data
- Machine-Executable Contracts, which tie legal text to data and executable code

### Machine-Readable Contracts

By combining Text and a data, a clause or contract becomes machine-readable.

For instance, the clause below for a [fixed rate loan](https://templates.accordproject.org/fixed-interests-static@0.2.0.html) includes natural language text coupled with variables. Together, these variables refer to some data for the clause and correspond to the 'deal points':

```tem
## Fixed rate loan

This is a *fixed interest* loan to the amount of {{loanAmount}}
at the yearly interest rate of {{rate}}%
with a loan term of {{loanDuration}},
and monthly payments of {{monthlyPayment}}.
```

To make sense of the data, a _Data Model_, expressed in the Concerto schema language, defines the variables for the template and their associated Data Types:

```ergo
  o Double loanAmount     // loanAmount is a floating-point number
  o Double rate           // rate is a floating-point number
  o Integer loanDuration  // loanDuration is an integer
  o Double monthlyPayment // monthlyPayment is a floating-point number
```

The Data Types allow a computer to validate values inserted into each of the `{{variable}}` placeholders (e.g., `2.5` is a valid `{{rate}}` but `January` isn't). In other words, the Data Model lets a computer make sense of the structure of (and data in) the clause. To learn more about Data Types see [Concerto Modeling](https://concerto.accordproject.org/docs/intro).

The clause data (the 'deal points') can then be capture as a machine-readable representation:

```js
{
  "$class": "org.accordproject.interests.TemplateModel",
  "clauseId": "cec0a194-cd45-42f7-ab3e-7a673978602a",
```

~~~~
  "loanAmount": 100000.0,
  "rate": 2.5,
  "loanDuration": 15
  "monthlyPayment": 667.0
}
```

The values entered into the template text are associated with the name of the
variable e.g. `{{rate}} = 2.5%`. This provides the structure for understanding the
clause and its contents.

### Machine-Executable Contracts

By adding Logic to a machine-readable clause or contract in the form of expressions
- much like in a spreadsheet - the contract is able to execute operations based
upon data included in the contract.

For instance, the clause below is a variant of the earlier [fixed rate loan]
(https://templates.accordproject.org/fixed-interests@0.2.0.html). While it is
consistent with the previous one, the `{{monthlyPayment}}` variable is replaced
with an [Ergo](logic-ergo.md) expression
`monthlyPaymentFormula(loanAmount,rate,loanDuration)` which calculates the monthly
interest rate based upon the values of the other variables: `{{loanAmount}}`,
`{{rate}}`, and `{{loanDuration}}`.  To learn more about contract Logic see [Ergo
Logic](logic-ergo.md).

```tem
## Fixed rate loan

This is a *fixed interest* loan to the amount of {{loanAmount}}
at a yearly interest rate of {{rate}}%
with a loan term of {{loanDuration}},
and monthly payments of {{% monthlyPaymentFormula(loanAmount,rate,loanDuration)
%}}.
```

This is a simple example of the benefits of Machine-Executable contract, here
adding logic to ensure that the value of the `{{monthlyPayment}}` in the text is
always consistent with the other variables in the clause. In this example, we
display the contract text using the underlying [Markup](markup-preliminaries.md)
format, instead of the rich-text output that would be found in [editor tools]
(started-resources.md#ecosystem-tools) and PDF outputs.

More complex examples, (e.g., how to add post-signature logic which responds to
data sent to the contract or which triggers operations on external systems) can be
found in the rest of this documentation.


--------------------------------------------------------------------------------
---
id: version-0.30.1-accordproject-template
title: Accord Project Templates
original_id: accordproject-template
---

An Accord Project template ties legal text to computer code. It is composed of
three elements:

- **Template Text**: the natural language of the template
~~~~

- **Template Model**: the data model that backs the template, acting as a bridge between the text and the logic
- **Template Logic**: the executable business logic for the template

![Template](assets/020/template.png)

The three components (Text - Model - Logic) can also be intuitively understood as a **progression**, from _human-readable_ legal text to _machine-readable_ and _machine-executable_. When combined these three elements allow templates to be edited, validated, and then executed on any computer platform (on your own machine, on a Cloud platform, on Blockchain, etc).

> We use the computing term 'executed' here, which means run by a computer. This is distinct from the legal term 'executed', which usually refers to the process of signing an agreement.

## Template Text

![Template Text](assets/020/template_text.png)

The template text is the natural language of the clause or contract. It can include markup to indicate [variables](ref-glossary.md#variable) for that template.

The following shows the text of an **Acceptance of Delivery** clause.

```tem
## Acceptance of Delivery.

{{shipper}} will be deemed to have completed its delivery obligations
if in {{receiver}}'s opinion, the {{deliverable}} satisfies the
Acceptance Criteria, and {{receiver}} notifies {{shipper}} in writing
that it is accepting the {{deliverable}}.

## Inspection and Notice.

{{receiver}} will have {{businessDays}} Business Days to inspect and
evaluate the {{deliverable}} on the delivery date before notifying
{{shipper}} that it is either accepting or rejecting the
{{deliverable}}.

## Acceptance Criteria.

The 'Acceptance Criteria' are the specifications the {{deliverable}}
must meet for the {{shipper}} to comply with its requirements and
obligations under this agreement, detailed in {{attachment}}, attached
to this agreement.
```

The text is written in plain English, with variables between `{{` and `}}`. Variables allows template to be used in different agreements by replacing them with different values.

For instance, the following show the same **Acceptance of Delivery** clause where the `shipper` is `"Party A"`, the `receiver` is `"Party B"`, the `deliverable` is `"Widgets"`, etc.

```md
## Acceptance of Delivery.
```

~~~
"Party A" will be deemed to have completed its delivery obligations
if in "Party B"'s opinion, the "Widgets" satisfies the
Acceptance Criteria, and "Party B" notifies "Party A" in writing
that it is accepting the "Widgets".

## Inspection and Notice.

"Party B" will have 10 Business Days to inspect and
evaluate the "Widgets" on the delivery date before notifying
"Party A" that it is either accepting or rejecting the
"Widgets".

## Acceptance Criteria.

The "Acceptance Criteria" are the specifications the "Widgets"
must meet for the "Party A" to comply with its requirements and
obligations under this agreement, detailed in "Attachment X", attached
to this agreement.
```

### TemplateMark

TemplateMark is the markup format in which the text for Accord Project templates is
written. It defines notations (such as the `{{` and `}}` notation for variables
used in the **Acceptance of Delivery** clause) which allows a computer to make
sense of your templates.

It also provides the ability to specify the document structure (e.g., headings,
lists), to highlight certain terms (e.g., in bold or italics), to indicate text
which is optional in the agreement, and more.

_More information about the Accord Project markup can be found in the [Markdown
Text](markup-templatemark.md) Section of this documentation._

## Template Model

![Template Model](assets/020/template_model.png)

Unlike a standard document template (e.g., in Word or pdf), Accord Project
templates associate a _model_ to the natural language text. The model acts as a
bridge between the text and logic; it gives the users an overview of the
components, as well as the types of different components.

The model categorizes variables (is it a number, a monetary amount, a date, a
reference to a business or organization, etc.). This is crucial as it allows the
computer to make sense of the information contained in the template.

The following shows the model for the **Acceptance of Delivery** clause.

```ergo
/* The template model */
asset AcceptanceOfDeliveryClause extends AccordClause {

  /* the shipper of the goods*/
  --> Organization shipper

  /* the receiver of the goods */
  --> Organization receiver
~~~

~~~
  /* what we are delivering */
  o String deliverable

  /* how long does the receiver have to inspect the goods */
  o Integer businessDays

  /* additional information */
  o String attachment
}
```

Thanks to that model, the computer knows that the `shipper` variable (`"Party A"`
in the example) and the `receiver` variable (`"Party B"` in the example) are both
`Organization` types. The computer also knows that variable `businessDays` (`10` in
the example) is an `Integer` type; and that the variable `deliverable` (`"Widgets"`
in the example) is a `String` type, and can contain any text description.

> If you are unfamiliar with the different types of variables, or want a more
thorough explanation of what variables are, please refer to our [Glossary](ref-
glossary.md#data-models) for a more detailed explanation.

### Concerto

Concerto is the language which is used to write models in Accord Project templates.
Concerto offers modern modeling capabilities including support for primitive types
(numbers, dates, etc), nested or optional data structures, enumerations,
relationships, object-oriented style inheritance, and more.

_More information about Concerto can be found in the [Concerto
Model](https://concerto.accordproject.org/docs/intro) section of this
documentation._

## Template Logic

![Template Logic](assets/020/template_logic.png)

The combination of text and model already makes templates _machine-readable_, while
the logic makes it _machine-executable_.

### During Drafting

In the [Overview](accordproject.md) Section, we already saw how logic can be
embedded in the text of the template itself to automatically calculate a monthly
payment for a [fixed rate loan]():

```tem
## Fixed rate loan

This is a *fixed interest* loan to the amount of {{loanAmount}}
at a yearly interest rate of {{rate}}%
with a loan term of {{loanDuration}},
and monthly payments of {{% monthlyPaymentFormula(loanAmount,rate,loanDuration)
%}}.
```

This uses a `monthlyPaymentFormula` function which calculates the monthly payment
based on the other data points in the text:
```ergo
define function monthlyPaymentFormula(loanAmount: Double, rate: Double,
~~~

~~~~
loanDuration: Integer) : Double {
  let term = longToDouble(loanDuration * 12);       // Term in months
  if (rate = 0.0) then return (loanAmount / term)   // If the rate is 0
  else
    let monthlyRate = (rate / 12.0) / 100.0;        // Rate in months
    let monthlyPayment =                            // Payment calculation
      (monthlyRate * loanAmount)
      / (1.0 - ((1.0 + monthlyRate) ^ (-term)));
    return roundn(monthlyPayment, 0)                // Rounding
}
```
Each logic function has a _name_ (e.g., `monthlyPayment`), a _signature_ indicating
the parameters with their types (e.g., `loanAmount:Double`), and a _body_ which
performs the appropriate computation based on the parameters. The main payment
calculation is here based on the [standardized calculation used in the United
States](https://en.wikipedia.org/wiki/Mortgage_calculator#Monthly_payment_formula)
with `*` standing for multiplication, `/` for division, and `^` for exponentiation.

### After Signature

The logic can also be used to associate behavior to the template _after_ the
contract has been signed. This can be used for instance to specify what happens
when a delivery is received late, to check conditions for payment, determine if
there has been a breach of contract, etc.

The following shows post-signature logic for the **Acceptance of Delivery** clause.

```ergo
contract SupplyAgreement over SupplyAgreementModel {
  clause acceptanceofdelivery(request : InspectDeliverable) : InspectionResponse {

    let received = request.deliverableReceivedAt;
    enforce isBefore(received,now()) else
      throw ErgoErrorResponse{ message : "Transaction time is before the
deliverable date." }
    ;

    let status =
      if isAfter(now(), addDuration(received, Duration{ amount:
contract.businessDays, unit: ~org.accordproject.time.TemporalUnit.days}))
      then OUTSIDE_INSPECTION_PERIOD
      else if request.inspectionPassed
      then PASSED_TESTING
      else FAILED_TESTING
    ;
    return InspectionResponse{
      status : status,
      shipper : contract.shipper,
      receiver : contract.receiver
    }
  }
}
```

This logic describes what conditions must be met for a delivery to be accepted. It
checks whether the delivery has already been made; whether the acceptance is
timely, within the specified inspection date; and whether the inspection has passed
or not.
~~~~

### Ergo

Ergo is the programming language which is used to express contractual logic in templates. Ergo is specifically designed for legal agreements, and is intended to be accessible for those creating the corresponding prose for those computable legal contracts. Ergo expressions can also be embedded in the text for a template.

More information about Ergo can be found in the [Ergo Logic](logic-ergo.md) Section of this documentation.

## Cicero

The implementation for the Accord Project templates is called [Cicero](https://github.com/accordproject/cicero). It defines and can read the structure of templates, with natural language bound to a data model and logic. By doing this, Cicero allows users to create, validate and execute software templates which embody all three components in the template triangle above.

More information about how to install Cicero and get started with Accord Project templates can be found in the [Installation](started-installation.md) Section of this documentation.

Let's look at each component of the template triangle, starting with the text.

### What next?

Build your first smart legal contract templates, either [online](tutorial-studio.md) with Template Studio, or by [installing Cicero](started-installation.md).

Explore [sample templates](started-resources.md) and other resources in the rest of this documentation.

If some of technical words are unfamiliar, please consult the [Glossary](ref-glossary.md) for more detailed explanations.

```
---
id: version-0.30.1-accordproject
title: Overview
original_id: accordproject
---
```

## What is the Accord Project?

Accord Project is an open source, non-profit initiative aimed at transforming contract management and contract automation by digitizing contracts. It provides an open, standardized format for Smart Legal Contracts.

The Accord Project defines a notion of a legal template with associated computing logic which is expressive, open-source, and portable. Accord Project templates are similar to a clause or contract template in any document format, but they can be read, interpreted, and run by a computer.

## Why is the Accord Project relevant?

The Accord Project provides a universal format for smart legal contracts, and this format is embodied in a variety of open source projects that comprise the Accord

Project technology stack. Input from businesses, lawyers and developers is crucial for the Accord Project.

### For Businesses

Contracting is undergoing a digital transformation driven by a need to deliver customer-centric legal and business solutions faster, and at lower cost. This imperative is fueling the adoption of a broad range of new technologies to improve the efficiency of drafting, managing, and executing legal contracting operations; the Accord Project is proud to be part of that movement.

The Accord Project provides a Smart Contract that does not depend on a blockchain, that can integrate text
and data and that can continue operating over its lifespan. The Accord Project smart contract can integrate with your technology platforms and become part of you digital infrastructure.

In addition, contributions from businesses are crucial for the development of the Accord Project. The expertise of stakeholders, such as business professionals and attorneys, is invaluable in improving the functionality and content of the Accord Project's codebase and specifications, to ensure that the templates meet real-world business requirements.

If this interests you, please visit our [Lifecycle and Industry Working Groups](https://www.accordproject.org/liwg) page for more information.

### For Lawyers

The Legal world is changing and Legal Tech is growing into a billion dollar industry. The modern lawyer has to be at home in the digital world. Law Schools now teach courses in coding for lawyers, computational law, blockchain and artificial intelligence. Legal Hackers is a world wide movement uniting lawyers across the world in a shared passion for law and technology. Lawyers need to move beyond the the written word on paper.

The template in an Accord Project Contract is pure legal text that can be drafted by lawyers and interpreted by courts. An existing contract can easily be transformed into a template by adding data points between curly braces that represent the Concerto model and Ergo logic can be added as an integral part of the contract. The template language is subject to judicial interpretation and the Concerto model and Ergo logic can be interpreted by a computer creating a bridge between the two worlds.

As a lawyer, contributing to the Accord Project would be a great opportunity to learn about smart legal contracts. Through the Accord Project, you can understand the foundations of open source technologies and learn how to develop smart agreements.

If your organization wants to become a member of the Accord Project, please [join our community](https://discord.com/invite/Zm99SKhhtA).

### For Developers

The Accord Project provides a universal format for smart legal contracts, and this format is embodied in a variety of open source projects that comprise the Accord Project technology stack. The Accord Project is an open source project and welcomes contributions from anyone.

The Accord Project is developing tools including a [Visual Studio Code plugin] (https://marketplace.visualstudio.com/items?itemName=accordproject.cicero-vscode-extension), [React based web components](https://github.com/accordproject/web-components) and a command line interface for working with Accord Project Contracts. You can integrate contracts into existing applications, create new applications or simply assist lawyers with developing applications with the Ergo language.

There is a welcoming community on Discord that is eager to help. [Join our Community](https://discord.com/invite/Zm99SKhhtA)

## About this documentation

If you are new to Accord Project, you may want to first read about the notion of [Smart Legal Contracts](accordproject-slc.md) and about [Accord Project Templates](accordproject-template.md). We also recommend taking the [Online Tour](accordproject-tour.md).

To start using Accord Project templates, follow the [Install Cicero](https://docs.accordproject.org/docs/next/started-installation.html) instructions in the _Getting Started_ Section of the documentation.

You can find in-depth guides for the different components of a template in the _Template Guides_ part of the documentation:
- Learn how to write contract or template text in the [Markdown Text](markup-preliminaries.md) Guide
- Learn how to design your data model in the [Concerto Model](https://concerto.accordproject.org/docs/intro) Guide
- Learn how to write smart contract logic in the [Ergo Logic](logic-ergo.md) Guide

Finally, the documentation includes several step by step [Tutorials](tutorial-templates.md) and some reference information (for APIs, command-line tools, etc.) can be found in the [Reference Manual](ref-glossary.md).

---
id: version-0.30.1-ergo-tutorial
title: Ergo: A Tutorial
original_id: ergo-tutorial

## Overview of Accord

Cicero is an Open Source implementation of the Accord Project Template Specification. It defines the structure of natural language templates, bound to a data model, that can be executed using Ergo and request/response JSON messages. You can read the latest user documentation here: http://docs.accordproject.org.

In short, with the Accord Project you can take a classic contract, e.g. Word document and use Cicero to define natural language contract and clause templates that can be executed by an event driven computer program (aka Smart contract). For the tutorial, Cicero will be used to define natural language contract and clause templates. These clause templates handle the syllogistic language of contracts.

For example,
```md
 if the goods are more than [{DAYS}] late,
 then notify the supplier of the goods, with the message [{MESSAGE}].
```

DAYS and MESSAGE are variables

You can browse the library of Open Source Cicero contract and clause templates at: https://templates.accordproject.org.

So how goes the contract get executed? That is where Ergo comes in Ergo is a strongly-typed functional programming language designed to capture the legal intent of legal contracts and clauses. We will use Ergo to create the contract logic consisting of a contract class with executable embedded clauses. Note: prior to the emergence of Ergo, the Cicero JavaScript component was primary to the execution of code.

Ergo obviates the Cicero JavaScript component for the execution phase with a new more comprehensive language which we explore in this tutorial.

## Cicero

The Open Source Cicero project defines the format of clause and contract templates based on to the Cicero Template Specification. The templates are the link between the natural language of contracts usually composed in a Word document and the specification of a machine executable transaction. Cicero templates define the API by specifying request and response elements for the logic associated with functional transaction executed by Ergo.

Cicero templates are composed of two elements:
* Template Grammar (the natural language text for the template),
* Template Model (the data model that includes the variables contained within the template).
* The Logic (the executable business logic for the template) will be handled by Ergo.

When combined these three elements allow templates to be edited, analyzed, queried and executed.

## Setup Ergo Development environment

Before you can build Ergo, you must install and configure the following dependencies on your machine:

### Git

* Git: The [Github Guide to Installing Git][git-setup.md] is a good source of information.

### Node.js

* Node.js (LTS): We use Node to generate the documentation, run a development web server, run tests, and generate distributable files. Depending on your system, you can install Node either from source or as a pre-packaged bundle.
> Tip: Use nvm (or nvm-windows) to manage and install Node.js, This facilitates a version change of Node.js per project.
* Lerna: This is a tool which helps when handling multiple npm packages in the Ergo repository. To install:
```
npm install -g lerna@^3.15.0
```

### Visual Studio Code

Visual Studio Code is a lightweight but powerful source code editor which runs on your desktop and is available for Windows, macOS and Linux. It comes with built-in support for JavaScript and Node.js and has a rich ecosystem of extensions for other languages (such Ergo).

Follow the platform specific guides below:
See, https://code.visualstudio.com/docs/setup/
* macOS
* Linux
* Windows

#### Install Ergo VisualStudio Plugin

### Validate Development Environment and Toolset

Clone https://github.com/accordproject/ergo to your local machine

### Getting started

Install Ergo

The easiest way to install Ergo is as a Node.js package. Once you have Node.js installed on your machine, you can get the Ergo compiler and command-line using the Node.js package manager by typing the following in a terminal:
$ npm install -g @accordproject/ergo-cli@0.20

This will install the compiler itself (ergoc) and a command-line tool (ergo) to execute Ergo code. You can check that both have been installed and print the version number by typing the following in a terminal:
```sh
$ ergoc --version
$ ergo --version
```
Then, to get command line help:
```
$ ergoc --help
$ ergo execute --help
```
Compiling your first contract
```ergo
namespace org.accordproject.volumediscount

contract VolumeDiscount over VolumeDiscountContract {
   // Clause for volume discount
   clause volumediscount(request : VolumeDiscountRequest) : VolumeDiscountResponse
{
        if request.netAnnualChargeVolume < contract.firstVolume
        then return VolumeDiscountResponse{ discountRate: contract.firstRate }
        else if request.netAnnualChargeVolume < contract.secondVolume
        then return VolumeDiscountResponse{ discountRate: contract.secondRate }
        else return VolumeDiscountResponse{ discountRate: contract.thirdRate }
  }
}
```

To compile your first Ergo contract to JavaScript , within Visual Studio code
* Open the folder where you cloned https://github.com/accordproject/ergo
* Use View/Terminal to run the Ergo compiler:
```sh
$ ergoc ./examples/volumediscount/model.cto ./examples/volumediscount/logic.ergo
Compiling Ergo './examples/volumediscount/logic.ergo' -- creating
```

~~~~
'./examples/volumediscount/logic.js'
```

By default, Ergo compiles to JavaScript for execution. This may change in the
future to support other languages. The compiled code for the result in stored as
`./examples/volumediscount/logic.js`

### Execute a contract
To execute a contract, we pass the necessary parameters including the CTO, Ergo
files, the name of a contract and the json files containing request and contract
state
ergorun [ctos] [ergos] --contractname [file] --contract [file] --state [file] --
request [file]

So for example we use ergorun with :
```sh
$ ergorun ./examples/volumediscount/model.cto ./examples/volumediscount/logic.ergo
--contractname org.accordproject.volumediscount.VolumeDiscount
--contract ./examples/volumediscount/contract.json
--request ./examples/volumediscount/request.json
--state ./examples/volumediscount/state.json
```

Here contract.json contains the following values
```json
{
  "$class": "org.accordproject.volumediscount.VolumeDiscountContract",
  "parties": null,
  "contractId": "cr1",
  "firstVolume": 1,
  "secondVolume": 10,
  "firstRate": 3,
  "secondRate": 2.9,
  "thirdRate": 2.8
}
```

Request.json contains
```json
{
  "$class": "org.accordproject.volumediscount.VolumeDiscountRequest",
  "netAnnualChargeVolume": 10.4
}
```

logic.ergo contains:
```ergo
namespace org.accordproject.volumediscount

contract VolumeDiscount over VolumeDiscountContract {
  // Clause for volume discount
  clause volumediscount(request : VolumeDiscountRequest) : VolumeDiscountResponse {
    if request.netAnnualChargeVolume < contract.firstVolume
    then return VolumeDiscountResponse{ discountRate: contract.firstRate }
    else if request.netAnnualChargeVolume < contract.secondVolume
    then return VolumeDiscountResponse{ discountRate: contract.secondRate }
    else return VolumeDiscountResponse{ discountRate: contract.thirdRate }
  }
}
~~~~

~~~
```
~~~

Here netAnnualCharge Volume equals 10.4 which is not less than firstVolume and secondVolume which are equal to 1 and 10 respectively so the logic for the volumediscount clause returns thirdRate which equals 2.8

```
7:31:58 PM - info: Logging initialized. 2018-09-27T23:31:58.623Z
7:31:59 PM - info: {"response":
{"discountRate":2.8,"$class":"org.accordproject.volumediscount.VolumeDiscountRespon
se"},"state":
{"$class":"org.accordproject.cicero.contract.AccordContractState","stateId":"1"},"e
mit":[]}
```

PS D:\Users\jbambara\Github\ergo>

## Ergo Development

Create Template
Start with basic agreement in natural language and locate the variables
Here in the example see the bold
Volume-Based Card Acceptance Agreement [Abbreviated]
This Agreement is by and between ……….you agree to be bound by the Agreement.
Discount means an amount that we charge you for accepting the Card, which amount is:
(i) a percentage (Discount Rate) of the face amount of the Charge that you submit, or a flat per-
Transaction fee, or a combination of both; and/or
(ii) a Monthly Flat Fee (if you meet our requirements).

Transaction Processing and Payments. ………………… less all applicable deductions, rejections, and withholdings, which include:
………………………….

SETTLEMENT
a) Settlement Amount. Our agent will pay you according to your payment plan, ……………………..which include:
        (i) the Discount,
…………………………………………..
b) Discount. The Discount is determined according to the following table:

Annual Dollar Volume	Discount
Less than $1 million	3.00%
$1 million to $10 million	2.90%
Greater than $10 million	2.80%

Identify the request variables and contract instance variables
Codify the variables with $[{request}] or [{contract instance}]

Annual Dollar Volume	Discount
Less than $[{firstVolume}] million	[{firstRate}]%
$[{firstVolume}] million to $[{secondVolume}] million	[{secondRate}]%
Greater than $[{secondVolume}] million	[{thirdRate}]%

Create Model
Define the model asset which contains the contract instance variables and the transaction request and response. Defines the data model for the VolumeDiscount template. This defines the structure that the parser for the template generates from input source text. See model.cto below:

```
 namespace org.accordproject.volumediscount
import org.accordproject.cicero.contract.* from
https://models.accordproject.org/cicero/contract.cto
import org.accordproject.cicero.runtime.* from
https://models.accordproject.org/cicero/runtime.cto
asset VolumeDiscountContract extends AccordContract {
  o Double firstVolume
  o Double secondVolume
  o Double firstRate
  o Double secondRate
  o Double thirdRate
}
transaction VolumeDiscountRequest {
  o Double netAnnualChargeVolume
}
transaction VolumeDiscountResponse {
        o Double discountRate
}
```

Create Logic

The contract logic is accomplished by coding ERGO statements and expressions to consume the request and use contract instance variables to produce the desired response. In our example, request.netAnnualChargeVolume is tested against contract rates to produce the result.

```
namespace org.accordproject.volumediscount

define the contract
contract VolumeDiscount over VolumeDiscountContract {

define the contract clause and request : response

   clause volumediscount(request : VolumeDiscountRequest) : VolumeDiscountResponse
{

define the logic ; here we use if /then /else statement to test request parameter
against contract instance variable
 and return

        if request.netAnnualChargeVolume < contract.firstVolume
        then return VolumeDiscountResponse{ discountRate: contract.firstRate }
        else if request.netAnnualChargeVolume < contract.secondVolume
        then return VolumeDiscountResponse{ discountRate: contract.secondRate }
        else return VolumeDiscountResponse{ discountRate: contract.thirdRate }
  }
```

Ergo Language

As you have seen in this tutorial, Ergo is a domain-specific language (DSL) that captures the execution logic of legal contracts. In this simple example, you see that Ergo aims to have contracts and clauses as first-class elements of the language. To accommodate the maturation of distributed ledger implementations, Ergo will be blockchain neutral, i.e., the same contract logic can be executed either on and off chain on distributed ledger technologies like HyperLedger Fabric. Most importantly, Ergo is consistent with the Accord Protocol Template Specification.
Follow the links below to learn more about
Introduction to Ergo
Ergo Language Guide
Ergo Reference Guide

October 12, 2018

---
id: version-0.30.1-logic-advanced-expr
title: Advanced Expressions
original_id: logic-advanced-expr

## Match

### Match against Values

Match expressions allow to check an expression against multiple possible
values:

```ergo
    match fruitcode
      with 1 then "Apple"
      with 2 then "Apricot"
      else "Strange Fruit"
```

Match expressions can also be used to match against enumerated values:
```ergo
    match state
      with NY then "Empire State"
      with NJ then "Garden State"
      else "Far from home state"
```

### Match against Types

Match expressions can be used to match a value against a class type:.

```
define constant products = [
    Product{ id : "Blender" },
    Car{ id : "Batmobile", range: "Infinite" },
    Product{ id : "Cup" }
  ]

foreach p in products
return
  match p
    with let x : Car then "Car (" ++ x.id ++ ") with range " ++ x.range
    with let x : Product then "Product (" ++ x.id ++ ")"
    else "Not a product"
```
Should return the array `["Product (Blender)", "Car (Batmobile) with range
Infinite", "Product (Cup)"]`

## Foreach

Foreach expressions allow to apply an expression of every element in
an input array of values and returns a new array:

```ergo
  foreach x in [1.0,-2.0,3.0] return x + 1.0
```

Foreach expressions can have an optional condition of the values being
iterated over:

```ergo
  foreach x in [1.0,-2.0,3.0] where x > 0.0 return x + 1.0
```

Foreach expressions can iterate over multiple arrays. For example, the following
foreach expression returns all all [Pythagorean
triples](https://en.wikipedia.org/wiki/Pythagorean_triple):
```ergo
let nums = [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0];
foreach x in nums
foreach y in nums
foreach z in nums
where (x^2.0 + y^2.0 = z^2.0)
return {a: x, b: y, c: z}
```
and should return the array `[{a: 3.0, b: 4.0, c: 5.0}, {a: 4.0, b: 3.0, c: 5.0},
{a: 6.0, b: 8.0, c: 10.0}, {a: 8.0, b: 6.0, c: 10.0}]`.

## Template Literals

Template literals are similar to [String literals](logic-simple-expr.md#literal-
values) but with the ability to embed Ergo expressions. They are written with
between `` ` `` and may contains Ergo expressions inside `{{%` and `%}}`.

The following Ergo expressions illustrates the use of a template literal to
construct a String describing the content of a record.
```
let law101 = {
    name: "Law for developers",
    fee: 29.99
  };
`Course "{{% law101.name %}}" (Cost: {{% law101.fee %}})`
```
Should return the string literal `"Course \"Law for developers\" (Cost: 29.99)"`.

## Formatting

One can use template formatting using the `Expr as "FORMAT"` Ergo expression.
Supported formats are the same as those available in TemplateMark [Formatted
Variables](markup-templatemark.md#formatted-variables).

For instance:
```
let payment = MonetaryAmount{ currencyCode: USD, doubleValue: 1129.99 };
payment as "K0,0.00"
```
Should return the string literal `"$1,129.99"`.

---
id: version-0.30.1-logic-complex-type
title: Complex Values & Types
original_id: logic-complex-type

So far we only considered atomic values and types, such as string values or integers, which are not sufficient for most contracts. In Ergo, values and types are based on the [Concerto Modeling](https://concerto.accordproject.org/docs/intro) (often referred to as CTO models after the `.cto` file extension). This provides a rich vocabulary to define the parameters of your contract, the information associated to contract participants, the structure of contract obligation, etc.

In Ergo, you can either import an existing CTO model or declare types directly within your code. Let us look at the different kinds of types you can define and how to create values with those types.

## Arrays

Array types lets you define collections of values and are denoted with `[]` after the type of elements in that collection:

```ergo
    String[]                         // a String array
    Double[]                         // a Double array
```

You can write arrays as follows:
```ergo
    ["pear","apple","strawberries"]  // an array of String values
    [3.14,2.72,1.62]                 // an array of Double values
```

You can construct arrays using other expressions:
```ergo
    let pi = 3.14;
    let e = 2.72;
    let golden = 1.62;
    [pi,e,golden]
```

Ergo also provides functions to manipulate arrays as parts of its [standard library](ref-ergo-stdlib.md#functions-on-arrays). The following example uses the `sum` function to calculate the sum of all the elements in the `prettynumbers` array.
```ergo
    let pi = 3.14;
    let e = 2.72;
    let golden = 1.62;
    let prettynumbers : Double[] = [pi,e,golden];
    sum(prettynumbers)
```

You can access the element at a given position inside the array using an index:
```ergo
    let fruits = ["pear","apple","strawberries"];
    fruits[0]          // Returns: some("pear")
     let fruits = ["pear","apple","strawberries"];
    fruits[2]          // Returns: some("strawberries")
     let fruits = ["pear","apple","strawberries"];
    fruits[4]          // Returns: none
```

 Note that the index starts at `0` for the first element and that indexed-based access returns an optional value, since Ergo compiler cannot statically determine whether there will be an element at the corresponding index. You can learn more about how to handle optional values and types in the [Optionals](logic-complex-type.md#optionals) Section below.

## Classes

You can declare classes in the Concerto Modeling Language (concepts, transactions, events, participants or assets) by importing them from a CTO file or directly within your Ergo program:

```ergo
   define concept Seminar {
     name : String,
     fee : Double
   }
   define asset Product {
     id : String
   }
   define asset Car extends Product {
     range : String
   }
   define transaction Response {
     rate : Double,
     penalty : Double
   }
  define event PaymentObligation{
    amount : Double,
    description : String
  }
```

Once a class type has been defined, you can create an instance of that type using the class name along with the values for each fields:

```ergo
    Seminar{
      name: "Law for developers",
      fee: 29.99
    }
    Car{
      id: "Batmobile4156",
      range: "Infinite"
    }
```

> **TechNote:** When extending an existing class (e.g., `Car extends Product`), the sub-class includes the fields from the super-class. So `Car` includes the field `range` which is locally declared and the field `id` which is declared in `Product`.

You can access fields for values of a class type by using the `.` operator:
```ergo
    Seminar{
      name: "Law for developers",
      fee: 29.99
    }.fee                        // Returns 29.99
```

## Records

Sometimes it is convenient to declare a structure without having to declare it
first. You can do that using a record, which is similar to a class but without a
name attached to it:

```ergo
   {
     name : String,  // A record with a name of type String
     fee : Double    // and a fee of type Double
   }
```

You do not need to declare that record, and can directly write an instance of that
record as follows:

```ergo
    {
      name: "Law for developers",
      fee: 29.99
    }
```

> Typing `return { name: "Law for developers", fee: 29.99 }` in the [Ergo REPL]
(https://ergorepl.netlify.com), should answer `Response. {name: "Law for
developers", fee: 29.99} : {fee: Double, name: String}`.

You can access the field of a record using the `.` operator:
```ergo
    {
      name: "Law for developers",
      fee: 29.99
    }.fee                        // Returns 29.99
```
## Enums

Here is how to declare an enumerated type:

```ergo
define enum ProductType {
   DAIRY,
   BEEF,
   VEGETABLES
}
```

To create an instance of that enum:
```ergo
DAIRY
BEEF
```

## Optionals

An optional type can contain a value or not and is indicated with a `?`.

```ergo
Integer?           // An optional integer
PaymentObligation? // An optional payment obligation
```

~~~
Double[]?          // An optional array of doubles
```

An optional value can be either present, written `some(v)`, or absent, written
`none`.

```ergo
let i1 : Integer? = some(1); i1
let i2 : Integer? = none; i2
```

To operate on an optional type, you need to say what to do when the value is
present and what to do when the value is not present. The most general way to do
that is with a match expression:

This example matches a value which is present:
```ergo
match some(1)
with let? x then "I found " ++ toString(x) ++ " :-)"
else "I found nothing :-("
```
and should return `"I found 1 :-)"`.

While this example matches against a value which is absent:
```
match none
with let? x then "I found " ++ toString(x) ++ " :-)"
else "I found nothing :-("
```
and should return `"I found nothing :-("`.

More details on match expressions can be found in [Advanced Expressions](logic-
advanced-expr.md#match).

For conciseness, a few operators are also available on optional values. One can
give a default value when the optional is `none` using the operator `??`. For
instance:

```ergo
some(1) ?? 0       // Returns the integer 1
none ?? 0          // Returns the integer 0
```

You can also access the field inside an optional concept or an optional record
using the operator `?.`. For instance:

```ergo
some({a:1})?.a     // Returns the optional value: some(1)
none?.a            // Returns the optional value: none
```



--------------------------------------------------------------------------------
---
id: version-0.30.1-logic-ergo
title: Ergo Overview
original_id: logic-ergo
---
~~~

## Language Goals

Ergo aims to:
- have contracts and clauses as first-class elements of the language
- help legal-tech developers quickly and safely write computable legal contracts
- be modular, facilitating reuse of existing contract or clause logic
- ensure safe execution: the language should prevent run-time errors and non-terminating logic
- be blockchain neutral: the same contract logic can be executed either on and off chain on a variety of distributed ledger technologies
- be formally specified: the meaning of contracts should be well defined so it can be verified, and preserved during execution
- be consistent with the [Accord Project Templates](accordproject-template.md)

## Design Choices

To achieve those goals the design of Ergo is based on the following principles:

- Ergo contracts have a class-like structure with clauses akin to methods
- Ergo can handle types (concepts, transactions, etc) defined with the [Concerto Modeling Language](https://github.com/accordproject/concerto) (so called CML models), as mandated by the Accord Project Template Specification
- Ergo borrows from strongly-typed functional programming languages: clauses have a well-defined type signature (input and output), they are functions without side effects
- The compiler guarantees error-free execution for well-typed Ergo programs
- Clauses and functions are written in an expression language with limited expressiveness (it allows conditional and bounded iteration)
- Most of the compiler is written in Coq as a stepping stone for formal specification and verification

## Status

- The current implementation is considered *in development*, we welcome contributions (be it bug reports, suggestions for new features or improvements, or pull requests)
- The current compiler targets JavaScript (either standalone or for use in Cicero Templates and Hyperledger Fabric) and Java (experimental)

## This Guide

Ergo provides a simple expression language to describe computation. From those expressions, one can write functions, clauses, and then whole contract logic. This guide explains most of the Ergo concepts starting from simple expressions all the way to contracts.

Ergo is a _strongly typed_ language, which means it checks that the expressions you use are consistent (e.g., you can take the square root of `3.14` but not of `"pi!"`). The type system is here to help you write better and safer contract logic, but it also takes a little getting used to. This page also introduces Ergo types and how to work with them.

---
id: version-0.30.1-logic-simple-type
title: Introducing Types
original_id: logic-simple-type

We have so far talked about types only informally. When we wrote earlier:
```ergo
    "John Smith" // a String literal
    1            // an Integer literal
    ...
```
the comments mention that `"John Smith"` is of type `String`, and that `1` is of type `Integer`.

In reality, the Ergo compiler understands which types your expressions have and can detect whether those expressions apply to the right type(s) or not.

Ergo types are based on the [Concerto Modeling](https://concerto.accordproject.org/docs/intro) Language.

## Primitive types

The simplest of types are primitive types which describe the various kinds of literal values we saw in the previous section. Those primitive types are:

```ergo
    Boolean
    String
    Double
    Integer
    Long
    DateTime
```

:::note
The two primitive types `Integer` and `Long` are currently treated as the same type by the Ergo compiler.
:::

## Type errors

The Ergo compiler understand types and can detect type errors when you write expressions. For instance, if you write: `1.0 + 2.0 * 3.0`, the Ergo compiler knows that the expression is correct since all parameters for the operators `+` and `*` are of type `Double`, and it knows the result of that expression will be a `Double` as well.

If you write `1.0 + 2.0 * "some text"` the Ergo compiler will detect that `"some text"` is of type `String`, which is not of the right type for the operator `*` and return a type error.

> Typing `return 1.0 + 2.0 * "some text"` in the [Ergo REPL](https://ergorepl.netlify.com), should answer a type error:
> ```text
> Type error (at line 1 col 13). This operator received
> unexpected arguments of type Double  and String.
> return 1.0 + 2.0 * "some text"
>              ^^^^^^^^^^^^^^^^^
> ```

## Type annotations

~~~
In a let bindings, you can also use a _type annotation_ to indicate which type you
expect it to have.

```ergo
    let name : String = "John"; // declares and initialize a string variable
    name ++ " Smith"            // rest of the expression
```
or
```ergo
    let x : Double = 3.1416     // declares and initialize a double variable
    sqrt(x)                     // rest of the expression
```

This can be useful to document your code, or to remember what type you expect from
an expression.

The Ergo compiler will return a type error if the annotation is not consistent with
the expression that computes the value for that let binding. For instance, the
following will return a type error since `"pi!"` is not of type `Double`.

```ergo
    let x : Double = "pi!"; // TYPE ERROR: "pi!" is not a Double
    sqrt(x)
```

> Typing `return let x : Double = "pi!"; sqrt(x)` in the [Ergo
REPL](https://ergorepl.netlify.com), should answer a type error:
> ```text
> Type error (at line 1 col 7). The let type annotation Double for
> the name x does not match the actual type String.
> return let x : Double = "pi!"; sqrt(x)
>        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
> ```

This becomes particularly useful as your code becomes more complex. For instance
the following expression will also trigger a type error:

```ergo
    let rate = 3.5;
    let name : String =
      if rate > 0.0
      then 3.14         // TYPE ERROR: 3.14 is not a String
      else "John";
    name ++ " Smith"
```

Since not all the cases of the `if ... then ... else ...` expressions return a
value of type `String` which is the type annotation for the `name` variable.


--------------------------------------------------------------------------------
---
id: version-0.30.1-markup-preliminaries
title: Preliminaries
original_id: markup-preliminaries
---

## Markdown & CommonMark
~~~

The text for Accord Project templates is written using markdown. It builds on the
[CommonMark](https://commonmark.org) standard so that any CommonMark document is
valid text for a template or contract.

As with other markup languages, CommonMark can express the document structure
(e.g., headings, paragraphs, lists) and formatting useful for readability (e.g.,
italics, bold, quotations).

The main reference is the [CommonMark
Specification](https://spec.commonmark.org/0.29/) but you can find an overview of
CommonMark main features in the [CommonMark](markup-commonmark.md) Section of this
guide.

## Accord Project Extensions

Accord Project uses two extensions to CommonMark: CiceroMark for the contract text,
and TemplateMark for the template grammar.

### Lexical Conventions

Accord Project contract or template text is a string of `UTF-8` characters.

:::note
By convention, CiceroMark files have the `.md` extensions, and TemplateMark files
have the `.tem.md` extension.
:::

The two sequences of characters `{{` and `}}` are reserved and used for the
CiceroMark and TemplateMark extensions to CommonMark. There are three kinds of
extensions:
1. Variables (written `{{variableName}}`) which may include an optional formatting
(written `{{variableName as "FORMAT"}}`).
2. Formulas (written `{{% expression %}}`).
3. Blocks which may contain additional text or markdown. Blocks come in two
flavors:
   1. Blocks corresponding to [markdown inline
elements](https://spec.commonmark.org/0.29/#inlines) which may contain only other
markdown inline elements (e.g., text, emphasis, links). Those have to be written on
a single line as follows:
      ```
      {{#blockName variableName}} ... text or markdown ... {{/blockName}}
      ```
   2. Blocks corresponding to [markdown container
elements](https://spec.commonmark.org/0.29/#container-blocks) which may contain
arbitrary markdown elements (e.g., paragraphs, lists, headings). Those have to be
written with each opening and closing tags on their own line as follows:
      ```
      {{#blockName variableName}}
      ... text or markdown ...
      {{/blockName}}
      ```

### CiceroMark

CiceroMark is used to express the natural language text for legal clauses or
contracts. It uses two specific extensions to CommonMark to facilitate contract
parsing:
1. Clauses within a contract can be identified using a `clause` block:
   ```
   ```

~~~
   {{#clause clauseName}}
   text of the clause
   {{/clause}}
   ```
2. The result of formulas within a contract or clause can be identified using:
   ```
   {{% result_of_the_formula %}}
   ```

For instance, the following CiceroMark for a loan between `John Smith` and `Jane
Doe` includes a title (`Loan agreement`) followed by some text, followed by a fixed
rate interest clause. The clause contains the terms for the loan and the result of
calculating the monthly payment.
```tem
# Loan agreement

This is a loan agreement between "John Smith" and "Jane Doe", which shall be
entered into
by the parties on January 21, 2021 - 3 PM, except in the event of a force majeure.

{{#clause fixedRate}}
## Fixed rate loan

This is a _fixed interest_ loan to the amount of £100,000.00
at the yearly interest rate of 2.5%
with a loan term of 15,
and monthly payments of {{%£667.00%}}
{{/clause}}
```

More information and examples can be found in the [CiceroMark](markup-
ciceromark.md) part of this guide.

### TemplateMark

TemplateMark is used to describe families of contracts or clauses with some
variable parts. It is based on CommonMark with several extensions to indicate those
variables parts:
1. _Variables_: e.g., `{{loanAmount}}` indicates the amount for a loan.
2. _Template Blocks_: e.g., `{{#if forceMajeure}}, except in the event of a force
majeure{{/if}}` indicates some optional text in the contract.
3. _Formulas_: e.g., `{{% monthlyPaymentFormula(loanAmount,rate,loanDuration) %}}`
calculates a monthly payment based on the `loanAmount`, `rate`, and `loanDuration`
variables.

For instance, the following TemplateMark for a loan between a `borrower` and a
`lender` includes a title (`Loan agreement`) followed by some text, followed by a
fixed rate interest clause. This template allows for either taking force majeure
into account or not, and calls into a formula to calculate the monthly payment.
```tem
# Loan agreement

This is a loan agreement between {{borrower}} and {{lender}}, which shall be
entered into
by the parties on {{date as "MMMM DD, YYYY - h A"}}{{#if forceMajeure}}, except in
the event of a force majeure{{/if}}.

{{#clause fixedRate}}
## Fixed rate loan
~~~

This is a _fixed interest_ loan to the amount of {{loanAmount as "K0,0.00"}}
at the yearly interest rate of {{rate}}%
with a loan term of {{loanDuration}},
and monthly payments of {{% monthlyPaymentFormula(loanAmount,rate,loanDuration) as "K0,0.00" %}}
{{/clause}}
```

More information and examples can be found in the [TemplateMark](markup-templatemark.md) part of this guide.

Dingus

You can test your template or contract text using the [TemplateMark Dingus](https://templatemark-dingus.netlify.app), an online tool which lets you edit the markdown and see it rendered as HTML, or as a document object model.

![TemplateMark Dingus](assets/dingus1.png)

You can select whether to parse your text as pure CommonMark (i.e., according to the CommonMark specification), or with the CiceroMark or TemplateMark extensions.

![TemplateMark Dingus](assets/dingus2.png)

You can also inspect the HTML source, or the document object model (abstract syntax tree or AST), even see a pdf rendering for your template.

![TemplateMark Dingus](assets/dingus3.png)

For instance, you can open the TemplateMark from the loan example on this page by clicking [this link](https://templatemark-dingus.netlify.app/#md3=%7B%22source
%22%3A%22%23%20Loan%20agreement%5Cn%5CnThis%20is%20a%20loan%20agreement%20between
%20%7B%7Bborrower%7D%7D%20and%20%7B%7Blender%7D%7D%2C%20which%20shall%20be
%20entered%20into%5Cnby%20the%20parties%20on%20%7B%7Bdate%20as%20%5C%22MMMMM%20DD
%2C%20YYYY%20-%20hhA%5C%22%7D%7D%7B%7B%23if%20forceMajeure%7D%7D%2C%20except%20in
%20the%20event%20of%20a%20force%20majeure%7B%7B%2Fif%7D%7D.%5Cn%5Cn%7B%7B%23clause
%20fixedRate%7D%7D%5Cn%23%23%20Fixed%20rate%20loan%5Cn%5CnThis%20is%20a%20_fixed
%20interest_%20loan%20to%20the%20amount%20of%20%7B%7BloanAmount%20as%20%5C
%22K0%2C0.00%5C%22%7D%7D%5Cnat%20the%20yearly%20interest%20rate%20of%20%7B%7Brate
%7D%7D%25%5Cnwith%20a%20loan%20term%20of%20%7B%7BloanDuration%7D%7D%2C%5Cnand
%20monthly%20payments%20of%20%7B%7B%25%20monthlyPaymentFormula%28loanAmount%2Crate
%2CloanDuration%29%20as%20%5C%22K0%2C0.00%5C%22%20%25%7D%7D%5Cn%7B%7B%2Fclause%7D
%7D%5Cn%22%2C%22defaults%22%3A%7B%22templateMark%22%3Atrue%2C%22ciceroMark
%22%3Afalse%2C%22html%22%3Atrue%2C%22_highlight%22%3Atrue%2C%22_strict%22%3Afalse
%2C%22_view%22%3A%22html%22%7D%7D).

![TemplateMark Dingus](assets/dingus4.png)

---
id: version-0.30.1-markup-templatemark
title: TemplateMark
original_id: markup-templatemark

TemplateMark is an extension to CommonMark used to write the text in Accord Project templates. The extension includes new markdown for variables, inline and container
```

elements of the markdown and template formulas.

The kind of extension which can be used is based on the _type_ of the variable in
the [Concerto Model](https://concerto.accordproject.org/docs/intro) for your
template. For each type in your model differrent markdown elements apply: variable
markdown for atomic types in the model, list blocks for array types in the model,
optional blocks for optional types in the model, etc.

## Variables

Standard variables are written `{{variableName}}` where `variableName` is a
variable declared in the model.

The following example shows a template text with three variables (`buyer`,
`amount`, and `seller`):

```tem
Upon the signing of this Agreement, {{buyer}} shall pay {{amount}} to {{seller}}.
```

The way variables are handled (both during parsing and drafting) is based on their
type.

### String Variable

#### Description

If the variable `variableName` has type `String` in the model:
```ergo
o String variableName
```
The corresponding instance should contain text between quotes (`"`).

#### Examples

For example, consider the following model:

```ergo
asset Template extends AccordClause {
  o String buyer
  o String supplier
}
```

the following instance text:
```md
This Supply Sales Agreement is made between "Steve Supplier" and "Betty Byer".
```

matches the template:
```tem
This Supply Sales Agreement is made between {{supplier}} and {{buyer}}.
```

while the following instance texts do not match:
```md
This Supply Sales Agreement is made between 2019 and 2020.
```
or

```md
This Supply Sales Agreement is made between Steve Supplier and Betty Byer.
```

### Numeric Variable

#### Description

If the variable `variableName` has type `Double`, `Integer` or `Long` in the model:
```ergo
o Double variableName
o Integer variableName2
o Long variableName3
```
The corresponding instance should contain the corresponding number.

#### Examples

For example, consider the following model:

```ergo
asset Template extends AccordClause {
  o Double penaltyPercentage
}
```

the following instance text:
```md
The penalty amount is 10.5% of the total value of the Equipment whose delivery has
been delayed.
```

matches the template:
```tem
The penalty amount is {{penaltyPercentage}}% of the total value of the Equipment
whose delivery has been delayed.
```

while the following instance texts do not match:
```md
The penalty amount is ten% of the total value of the Equipment whose delivery has
been delayed.
```
or
```md
The penalty amount is "10.5"% of the total value of the Equipment whose delivery
has been delayed.
```

### Enum Variables

#### Description

If the variable `variableName` has an enumerated type:
```ergo
o EnumType variableName
```

The corresponding instance should contain a corresponding enumerated value without

~~~~
quotes.

#### Examples

For example, consider the following model:
```ergo
import org.accordproject.money.CurrencyCode from
https://models.accordproject.org/money.cto
asset Template extends AccordClause {
  o CurrencyCode currency
}

```
the following instance text:
```md
Monetary amounts in this contract are denominated in USD.
```

matches the template:
```tem
Monetary amounts in this contract are denominated in {{currency}}.
```

while the following instance texts do not match:
```md
Monetary amounts in this contract are denominated in "USD".
```
or
```md
Monetary amounts in this contract are denominated in $.
```

## Formatted Variables

Formatted variables are written `{{variableName as "FORMAT"}}` where `variableName`
is a variable declared in the model and the `FORMAT` is a type-dependent
description for the syntax of the variables in the contract.

The following example shows a template text with one variable with a format
`DD/MM/YYYY`.

```tem
The contract was signed on {{contractDate as "DD/MM/YYYY"}}.
```

### DateTime Variables

#### Description

If the variable `variableName` has type `DateTime`:
```ergo
o DateTime variableName
```
The corresponding instance should be a date and time, and can optionally be
formatted. The default format is `MM/DD/YYYY`, commonly used in the US.

#### DateTime Formats

The textual representation of a DateTime can be customized by including an optional
~~~~

format string using the `as` keyword directly in a template grammar. The following formatting tokens are supported:

Tokens are case-sensitive.

Input	Example	Description
`YYYY`	`2014`	4 or 2 digit year
`M`	`12`	1 or 2 digit month number
`MM`	`04`	2 digit month number
`MMM`	`Feb.`	Short month name
`MMMM`	`December`	Long month name
`D`	`3`	1 or 2 digit day of month
`DD`	`04`	2 digit day of month
`H`	`3`	24 hours (1 or 2 digits)
`HH`	`04`	24 hours (2 digits)
`h`	`1`	12 hours (1 or 2 digits)
`hh`	`02`	12 hours (2 digits)
`a`	`am` or `pm`	morning/afternoon (lowercase)
`A`	`AM` or `PM`	morning/afternoon (uppercase)
`mm`	`59`	2 digit minutes
`ss`	`34`	2 digit seconds
`SSS`	`002`	3 digit milliseconds
`Z`	`+01:00`	UTC offset

:::note
If `Z` is specified, it must occur as the last token in the format string.
:::

#### Examples

The format of the `contractDate` variable of type `DateTime` can be specified with the `DD/MM/YYYY` format, as is commonly used in Europe.

```tem
The contract was signed on {{contractDate as "DD/MM/YYYY"}}.
The contract was signed on 26/04/2019.
```

Other examples:

```tem
dateTimeProperty: {{dateTimeProperty as "D MMM YYYY HH:mm:ss.SSSZ"}}
dateTimeProperty: 1 Jan 2018 05:15:20.123+01:02
```

```tem
dateTimeProperty: {{dateTimeProperty as "D MMMM YYYY HH:mm:ss.SSSZ"}}
dateTimeProperty: 1 January 2018 05:15:20.123+01:02
```

```tem
dateTimeProperty: {{dateTimeProperty as "D-M-YYYY H mm:ss.SSSZ"}}
dateTimeProperty: 31-12-2019 2 59:01.001+01:01
```

```tem
dateTimeProperty: {{dateTimeProperty as "DD/MM/YYYY"}}
dateTimeProperty: 01/12/2018
```

```tem
dateTimeProperty: {{dateTimeProperty as "DD-MMM-YYYY H mm:ss.SSSZ"}}
dateTimeProperty: 04-Jan-2019 2 59:01.001+01:01
```

### Amount Variables

#### Description

If the variable `variableName` is of type `Integer`, `Long`, `Double` or
`MonetaryAmount`:
```ergo
o Integer integerVariable
o Long longVariable
o Double doubleVariable
o MonetaryAmount monetaryVariable
```

The corresponding instance should be a numeric value (with a currency code in the
case of monetary amounts), and can optionally be formatted.

#### Amount Formats

The textual representation of an amount can be customized by including an optional
format string using the `as` keyword directly in a template grammar. The following
formatting tokens are supported:

Tokens are case-sensitive.

Input	Example	Description	Type Supported
`0,0`	`3,100,200`	integer part with `,` separator	Integer,Long,Double,MonetaryAmount
`0 0`	`3 100 200`	integer part with ` ` separator	Integer,Long,Double,MonetaryAmount
`0,0.00`	`3,100,200.95`	decimal with two digits precision	Double,MonetaryAmount
`0 0,00`	`3 100 200,95`	decimal with two digits precision	Double,MonetaryAmount
`0,0.0000`	`3,100,200.95`	decimal with four digits precision	Double,MonetaryAmount
`CCC`	`USD`	currency code	MonetaryAmount
`K`	`$`	currency symbol	MonetaryAmount

The general format for the amount is `0{sep}0({sep}0+)?` where `{sep}` is a single
character (e.g., `,` or `.`). The first `{sep}` is used to separate every three
digits of the integer part. The second `{sep}` is used as a decimal point. And the
number of `0` after the second separator is used to indicate precision (number of
digits after the decimal point).

#### Examples

The following examples show formating for `Integer` or `Long` values.

```
The manuscript shall be completed within {{days as "0,0"}} days.
The manuscript shall be completed within 1,001 days.
```

```
The manuscript shall contain at most {{words as "0 0"}} words.
The manuscript shall contain at most 1 500 001 words.
```

The following examples show formatting for `Double` values.

```
The effective range of the device should be at least {{distance as "0,0.00mm"}}.
The effective range of the device should be at least 1,250,400.99mm.
```

```
The effective range of the device should be at least {{distance as "0 0,0000mm"}}.
The effective range of the device should be at least 1 250 400,9900mm.
```

The following examples show formatting for `MonetaryAmount` values.

```
The loan principal is {{principal as "0,0.00 CCC"}}.
The loan principal is 2,000,500,000.00 GBP.
```

```
The loan principal is {{principal as "K0,0.00"}}.
The loan principal is £2,000,500,000.00.
```

```
The loan principal is {{principal as "0 0,00 K"}}.
The loan principal is 2 000 500 000,00 €.
```

## Complex Types Variables

### Duration Types

#### Description

If the variable `variableName` has type `Duration`:
```ergo
import org.accordproject.time.Duration
o Duration variableName
```

The corresponding instance should contain the corresponding duration written with the amount as a number and the duration unit as literal text.

#### Examples

For example, consider the following model:

```ergo
asset Template extends AccordClause {
  o Duration termination
}
```

the following instance texts:
```md
If the delay is more than 15 days, the Buyer is entitled to terminate this
Contract.
```
and
```md
If the delay is more than 1 week, the Buyer is entitled to terminate this Contract.
```

both match the template:
```tem
If the delay is more than {{termination}}, the Buyer is entitled to terminate this
Contract.
```

while the following instance texts do not match:
```md
If the delay is more than a month, the Buyer is entitled to terminate this
Contract.
```
or
```md
If the delay is more than "two weeks", the Buyer is entitled to terminate this
Contract.
```

### Other Complex Types

#### Description

If the variable `variableName` has a complex type `ComplexType` (such as an
`asset`, a `concept`, etc.)
```ergo
o ComplexType variableName
```

The corresponding instance should contain all fields in the corresponding complex
type in the order they occur in the model, separated by a single white space
character.

#### Examples

For example, consider the following model:
```ergo
import org.accordproject.address.PostalAddress from
https://models.accordproject.org/address.cto
asset Template extends AccordClause {
  o PostalAddress address
}
```

the following instance text:

```md
Address of the supplier: "555 main street" "10290" "" "NY" "New York" "10001".
```

matches the template:
```tem
Address of the supplier: {{address}}.
```

Consider the following model:
```md
import org.accordproject.money.MonetaryAmount from
https://models.accordproject.org/money.cto
asset Template extends AccordClause {
  o MonetaryAmount amount
}
```

the following instance text:
```md
Total value of the goods: 50.0 USD.
```

matches the template:
```tem
Total value of the goods: {{amount}}.
```

## Inline Blocks

CiceroMark uses blocks to enable more advanced scenarios, to handle optional or
repeated text (e.g., lists), to change the variables in scope for a given section
of the text, etc. Inline blocks correspond to inline elements in the markdown.

Inline blocks always have the following syntactic structure:

```tem
{{#blockName variableName parameters}}...{{/blockName}}
```

where `blockName` indicates which kind of block it is (e.g., conditional block or
optional block), `variableName` indicates the template variable which is in scope
within the block. For certain blocks, additional `parameters` can be passed to
control the behavior of that block (e.g., the `join` block creates text from a list
with an optional separator).

### Conditional Blocks

Conditional blocks enables text which depends on a value of a `Boolean` variable in
your model:

```tem
{{#if forceMajeure}}This is a force majeure{{/if}}
```

Conditional blocks can also include an `else` branch to indicate that some other
text should be use when the value of the variable is `false`:

```tem
```

~~~
{{#if forceMajeure}}This is a force majeure{{else}}This is *not* a force
majeure{{/if}}
```

#### Examples

Drafting text with the first conditional block above using the following JSON data:
```json
{
  "$class": "org.accordproject.foo.Status",
  "forceMajeure": true
}
```

results in the following markdown text:

```md
This is a force majeure
```

Drafting text with this block using the following JSON data:
```json
{
  "$class": "org.accordproject.foo.Status",
  "forceMajeure": false
}
```

results in the following markdown text:

```md

```

### Optional Blocks

Optional blocks enable text which depends on the presence or absence of an
`optional` variable in your model:

```tem
{{#optional forceMajeure}}This applies except for Force Majeure cases in a
{{miles}} miles radius.{{/optional}}
```

Optional blocks can also include an `else` branch to indicate that some other text
should be use when the value of the variable is absent (`null` in the JSON data):

```tem
{{#optional forceMajeure}}This applies except for Force Majeure cases in a
{{miles}} miles radius.{{else}}This applies even in case a force
majeure.{{/optional}}
```

#### Examples

Drafting text with the second optional block above using the following JSON data:
```json
{
  "$class": "org.accordproject.foo.Status",
~~~

~~~~
  "forceMajeure": {
    "$class": "org.accordproject.foo.Distance",
    "miles": 250
  }
}
```

results in the following markdown text:

```md
This applies except for Force Majeure cases in a 250 miles radius.
```

Drafting text with this block using the following JSON data:
```json
{
  "$class": "org.accordproject.foo.Status",
  "forceMajeure": null
}
```

results in the following markdown text:

```md
This applies even in case a force majeure.
```

### With Blocks

A `with` block can be used to change variables that are in scope in a specific part
of a template grammar:

```tem
For the Tenant: {{#with tenant}}{{partyId}}, domiciled at {{address}}{{/with}}
For the Landlord: {{#with landlord}}{{partyId}}, domiciled at {{address}}{{/with}}
```

#### Example

Drafting text with this block using the following JSON data:
```json
{
  "$class": "org.accordproject.rentaldeposit.RentalDepositClause",
  "contractId": "31d817e2-d62a-4b70-b395-acd0d5da09f5",
  "tenant": {
    "$class": "org.accordproject.rentaldeposit.RentalParty",
    "partyId": "Michael",
    "address": "111, main street"
  }
  ...
}
```

results in the following markdown text:

```md
For the Tenant: "Michael", domiciled at "111, main street"
For the Landlord: "Parsa", domiciled at "222, chestnut road"
```
~~~~

### Join Blocks

A `join` block can be used to iterate over a variable containing an array of values, and can use an (optional) separator.

```tem
Discount applies to the following items: {{#join items separator=", "}}{{name}}
({{id}}){{/join}}.
```

#### Example

Drafting text with this block using the following JSON data:
```json
{
  "$class": "org.accordproject.sale.Order",
  "contractId": "31d817e2-d62a-4b70-b395-acd0d5da09f5",
  "items": [{
      "$class": "org.accordproject.slate.Item",
      "id": "111",
      "name": "Pineapple"
    },{
      "$class": "org.accordproject.slate.Item",
      "id": "222",
      "name": "Strawberries"
    },{
      "$class": "org.accordproject.slate.Item",
      "id": "333",
      "name": "Pomegranate"
    }
  ]
}
```

results in the following markdown text:

```md
Discount applies to the following items: Pineapple (111), Strawberries (222),
Pomegranate (333).
```

## Container Blocks

CiceroMark uses block expressions to enable more advanced scenarios, to handle optional or repeated text (e.g., lists), to change the variables in scope for a given section of the text, etc.

Container blocks always have the following syntactic structure:

```tem
{{#blockName variableName parameters}}
...
{{/blockName}}
```

where `blockName` indicates which kind of block it is (e.g., conditional block or list block), `variableName` indicates the template variable which is in scope within the block. For certain blocks, additional `parameters` can be passed to control the behavior of that block (e.g., the `join` block creates text from a list with an optional separator).

### Unordered Lists

```tem
{{#ulist rates}}
{{volumeAbove}}$ M<= Volume < {{volumeUpTo}}$ M : {{rate}}%
{{/ulist}}
```

#### Example

Drafting text with this block using the following JSON data:
```json
{
  "$class": "org.accordproject.volumediscountlist.VolumeDiscountContract",
  "contractId": "19243313-adc2-4ff1-aa41-993816ed2cdc",
  "rates": [
    {
      "$class": "org.accordproject.volumediscountlist.RateRange",
      "volumeUpTo": 1,
      "volumeAbove": 0,
      "rate": 3.1
    },
    {
      "$class": "org.accordproject.volumediscountlist.RateRange",
      "volumeUpTo": 10,
      "volumeAbove": 1,
      "rate": 3.1
    },
    {
      "$class": "org.accordproject.volumediscountlist.RateRange",
      "volumeUpTo": 50,
      "volumeAbove": 10,
      "rate": 2.9
    }
  ]
}
```

results in the following markdown text:

```md
- 0.0$ M <= Volume < 1.0$ M : 3.1%
- 1.0$ M <= Volume < 10.0$ M : 3.1%
- 10.0$ M <= Volume < 50.0$ M : 2.9%
```

### Ordered Lists

```tem
{{#olist rates}}
{{volumeAbove}}$ M <= Volume < {{volumeUpTo}}$ M : {{rate}}%
{{/olist}}
```

#### Example

~~~~
Drafting text with this block using the following JSON data:
```json
{
  "$class": "org.accordproject.volumediscountlist.VolumeDiscountContract",
  "contractId": "19243313-adc2-4ff1-aa41-993816ed2cdc",
  "rates": [
    {
      "$class": "org.accordproject.volumediscountlist.RateRange",
      "volumeUpTo": 1,
      "volumeAbove": 0,
      "rate": 3.1
    },
    {
      "$class": "org.accordproject.volumediscountlist.RateRange",
      "volumeUpTo": 10,
      "volumeAbove": 1,
      "rate": 3.1
    },
    {
      "$class": "org.accordproject.volumediscountlist.RateRange",
      "volumeUpTo": 50,
      "volumeAbove": 10,
      "rate": 2.9
    }
  ]
}
```

results in the following markdown text:
```md
1. 0.0$ M <= Volume < 1.0$ M : 3.1%
2. 1.0$ M <= Volume < 10.0$ M : 3.1%
3. 10.0$ M <= Volume < 50.0$ M : 2.9%
```

### Clause Blocks

Clause blocks can be used to include a clause template within a contract template:

```tem
Payment
-------
{{#clause payment}}
As consideration in full for the rights granted herein, Licensee shall pay Licensor
a one-time
fee in the amount of {{amountText}} ({{amount}}) upon execution of this Agreement,
payable as
follows: {{paymentProcedure}}.
{{/clause}}
```

#### Example

Drafting text with this block using the following JSON data:
```json
{
  "$class": "org.accordproject.copyrightlicense.CopyrightLicenseContract",
  "contractId": "944535e8-213c-4649-9e60-cc062cce24e8",
  ...
~~~~

~~~
  "paymentClause": {
    "$class": "org.accordproject.copyrightlicense.PaymentClause",
    "clauseId": "6c7611dc-410c-4134-a9ec-17fb6aad5607",
    "amountText": "one hundred US Dollars",
    "amount": {
      "$class": "org.accordproject.money.MonetaryAmount",
      "doubleValue": 100,
      "currencyCode": "USD"
    },
    "paymentProcedure": "bank transfer"
  }
}
```

results in the following markdown text:

```md
Payment
----

As consideration in full for the rights granted herein, Licensee shall pay Licensor
a one-time
fee in the amount of "one hundred US Dollars" (100.0 USD) upon execution of this
Agreement, payable as
follows: "bank transfer".

```

## Ergo Formulas

Ergo formulas in template text are essentially similar to Excel formulas, and
enable to create legal text dynamically, based on the other variables in your
contract. They are written `{{% ergoExpression %}}` where `ergoExpression` is any
valid [Ergo Expression](logic-ergo).

::: note
Formulas allow the template developer to generate arbitrary contract text from
other contract and clause variables. They therefore cannot be used to set a
template model variable during parsing. In other words formulas are evaluated when
drafting a contract but are ignored when parsing the contract text.
:::

### Evaluation Context

The context in which expressions within templates text are evaluated includes:
- The contract variables, which can be accessed using the variable name (or
`contract.variableName`)
- All constants or functions declared or imported in the main [Ergo module](logic-
module) for your template.

#### Fixed Interests Clause

For instance, let us look one more time at [fixed rate
loan](https://templates.accordproject.org/fixed-interests-static@0.2.0.html) clause
that was used previously:

```tem
## Fixed rate loan
~~~

This is a *fixed interest* loan to the amount of {{loanAmount}}
at the yearly interest rate of {{rate}}%
with a loan term of {{loanDuration}},
and monthly payments of {{% monthlyPaymentFormula(loanAmount,rate,loanDuration) %}}
```

The [`logic` directory](https://github.com/accordproject/cicero-template-library/
tree/master/src/fixed-interests/logic) for that template includes two Ergo modules:
```
./logic/interests.ergo  // Module containing the monthlyPaymentFormula function
./logic/logic.ergo      // Main module
```

A look inside the `logic.ergo` module shows the corresponding import, which ensures
the `monthlyPaymentFormula` function is also in scope in the text for the template:
```
namespace org.accordproject.interests

import org.accordproject.loan.interests.*

contract Interests over TemplateModel {
  ...
}

### Examples

Ergo provides a wide range of capabilities which you can use to construct the text
that should be included in the final clause or contract. Below are a few examples
for illustrations, but we encourage you to consult the [Ergo Logic](logic-ergo)
guide for a more comprehensive overview of Ergo.

#### Path expressions

The contents of complex values can be accessed using the `.` notation.

For instance the following template uses the `.` notation to access the first name
and last name of the contract author.

```tem
This contract was drafted by {{% author.name.firstName %}} {{% author.name.lastName
%}}
```

#### Built-in Functions

Ergo offers a number of pre-defined functions for a variety of primitive types.
Please consult the [Ergo Standard Library](ref-logic-stdlib) reference for the
complete list of built-in functions.

For instance the following template uses the `addPeriod` function to automatically
include the date at which a lease expires in the text of the contract:

```tem
This lease was signed on {{signatureDate}}, and is valid for a {{leaseTerm}}
period.
This lease will expire on {{% addPeriod(signatureDate, leaseTerm) %}}`
```

#### Iterators

Ergo's `foreach` expressions lets you iterate over collections of values.

For instance the following template uses a `foreach` expression combined with the `avg` built-in function to include the average product price in the text of the contract:

```tem
The average price of the products included in this purchase
order is {{% avg(foreach p in products return p.price) %}}.
```

#### Conditionals

Conditional expressions lets you include alternative text based on arbitrary conditions.

For instance, the following template uses a conditional expression to indicate the governing jurisdiction:

```tem
Each party hereby irrevocably agrees that process may be served on it in
any manner authorized by the Laws of {{%
    if address.country = US and getYear(now()) > 1959
    then "the State of " ++ address.state
    else "the Country of " ++ address.country
%}}
```

---
id: version-0.30.1-ref-migrate-0.13-0.20
title: Cicero 0.13 to 0.20
original_id: ref-migrate-0.13-0.20

Much has changed in the `0.20` release. This guide provides step-by-step instructions to port your Accord Project templates from version `0.13` or earlier to version `0.20`.

:::note
Before following those migration instructions, make sure to first install version `0.20` of Cicero, as described in the [Install Cicero](started-installation.md) Section of this documentation.
:::

## Metadata Changes

You will first need to update the `package.json` in your template. Remove the Ergo version which is now unnecessary, and change the Cicero version to `^0.20.0`.

#### Example

After those changes, the `accordproject` field in your `package.json` should look as follows (with the `template` field being either `clause` or `contract` depending on the template):
```js
```

...
    "accordproject": {
        "template": "clause",
        "cicero": "^0.20.0"
    }
...
```

Template Directory Changes

The layout of templates has changed to reflect the conceptual notion of Accord Project templates (as a triangle composed of text, model and logic). To migrate a template directory from version `0.13` or earlier to the new `0.20` layout:
1. Rename your `lib` directory to `logic`
2. Rename your `models` directory to `model`
3. Rename your `grammar` directory to `text`
4. Rename your template grammar from `text/template.tem` to `text/grammar.tem.md`
5. Rename your samples from `sample.txt` to `text/sample.md` (or generally any other `sample*.txt` files to `text/sample*.md`)

Example

Consider the [late delivery and penalty](https://templates.accordproject.org/latedeliveryandpenalty@0.14.1.html) clause. After applying those changes, the template directory should look as follows:
```
./.cucumber.js
./README.md
./package.json
./request-forcemajeure.json
./request.json
./state.json

./logic:
./logic/logic.ergo

./model:
./model/clause.cto

./test:
./test/logic.feature
./test/logic_default.feature

./text:
./text/grammar.tem.md
./text/sample-noforcemajeure.md
./text/sample.md
```

Text Changes

Both grammar and sample text for the templates has changed to support rich text annotations through CommonMark and a new syntax for variables. You can find complete information about the new syntax in the [CiceroMark](markup-cicero) Section of this documentation. For an existing template, you should apply the following changes.

Text Grammar Changes
```

1. Variables should be changed from `[{variableName}]` to `{{variableName}}`
2. Formatted variables should be changed to from `[{variableName as "FORMAT"}]` to `{{variableName as "FORMAT"}}`
3. Boolean variables should be changed to use the new block syntax, from `[{"This is a force majeure":?forceMajeure}]` to `{{#if forceMajeure}}This is a force majeure{{/if}}`
4. Nested clauses should be changed to use the new block syntax, from `[{#payment}]As consideration in full for the rights granted herein...[{/payment}]` to `{{#clause payment}}As consideration in full for the rights granted herein... {{/clause}}`

:::note
1. Template text is now interpreted as CommonMark which may lead to unexpected results if your text includes CommonMark characters or structure (e.g., `#` or `##` now become headings; `1.` or `-` now become lists). You should review both the grammar and samples so they follow the proper [CommonMark](https://commonmark.org) rules.
2. The new lexer reserves `{{` instead of reserving `[{` which means you should avoid using `{{` in your text unless for Accord Project variables.
:::

### Text Samples Changes

You should ensure that any changes to the grammar text is reflected in the samples. Any change in the grammar text outside of variables should be applied to the corresponding `sample.md` files as well.

:::tip
You can check that the samples and grammar are in agreement by using the `cicero parse` command.
:::

#### Example

Consider the text grammar for the [late delivery and penalty](https://templates.accordproject.org/latedeliveryandpenalty@0.14.1.html) clause:

```md
Late Delivery and Penalty.
In case of delayed delivery[{" except for Force Majeure cases,":? forceMajeure}]
[{seller}] (the Seller) shall pay to [{buyer}] (the Buyer) for every
[{penaltyDuration}]
of delay penalty amounting to [{penaltyPercentage}]% of the total value of the
Equipment
whose delivery has been delayed. Any fractional part of a [{fractionalPart}] is to
be
considered a full [{fractionalPart}]. The total amount of penalty shall not
however,
exceed [{capPercentage}]% of the total value of the Equipment involved in late
delivery.
If the delay is more than [{termination}], the Buyer is entitled to terminate this
Contract.
```

After applying the above rules to the code for the `0.13` version, and identifying the heading for the clause using the new markdown features, the grammar text becomes:

```tem
## Late Delivery and Penalty.

In case of delayed delivery{{#if forceMajeure}} except for Force Majeure
cases,{{/if}}
{{seller}} (the Seller) shall pay to {{buyer}} (the Buyer) for every
{{penaltyDuration}}
of delay penalty amounting to {{penaltyPercentage}}% of the total value of the
Equipment
whose delivery has been delayed. Any fractional part of a {{fractionalPart}} is to
be
considered a full {{fractionalPart}}. The total amount of penalty shall not
however,
exceed {{capPercentage}}% of the total value of the Equipment involved in late
delivery.
If the delay is more than {{termination}}, the Buyer is entitled to terminate this
Contract.
```

To make sure the `sample.md` file parses as well, the heading needs to be similarly
identified using markdown:
```md
## Late Delivery and Penalty.

In case of delayed delivery except for Force Majeure cases,
"Dan" (the Seller) shall pay to "Steve" (the Buyer) for every 2 days
of delay penalty amounting to 10.5% of the total value of the Equipment
whose delivery has been delayed. Any fractional part of a days is to be
considered a full days. The total amount of penalty shall not however,
exceed 55% of the total value of the Equipment involved in late delivery.
If the delay is more than 15 days, the Buyer is entitled to terminate this
Contract.
```

## Model Changes

There is no model changes required for this version.

## Logic Changes

Version `0.20` of Ergo has a few new features that are non backward compatible with
version `0.13`. Those may require you to change your template logic. The main non-
backward compatible feature is the new support for enumerated values.

### Enumerated Values

Enumerated values are now proper values with a proper enum type, and not based on
the type `String` anymore.

For instance, consider the enum declaration:
```js
enum Cardsuit {
  o CLUBS
  o DIAMONDS
  o HEARTS
  o SPADES
}
```

In version `0.13` or earlier the Ergo code would write `"CLUBS"` for an enum value and treat the type `Cardsuit` as if it was the type `String`.

As of version `0.20` Ergo writes `CLUBS` for that same enum value and the type `Cardsuit` is now distinct from the type `String`.

If you try to compile Ergo logic written for version `0.13` or earlier that features enumerated values, the compiler will likely throw type errors. You should apply the following changes:

1. Remove the quotes (`"`) around any enum values in your logic. E.g., `"USD"` should now be replaced by `USD` for monetary amounts;
3. If enum values are bound to variables with a type annotation, you should change the type annotation from `String` to the correct enum type. E.g., `let x : String = "DIAMONDS"; ...` should become `let x : Cardsuit = DIAMONDS; ...`;
3. If enum values are passed as parameters in clauses or functions, you should change the type annotation for that parameter from `String` to the correct enum type.
4. In a few cases the same enumerated value may be used in different enum types (e.g., `days` and `weeks` are used in both `TemporalUnit` and `PeriodUnit`). Those two values will now have different types. If you need to distinguish, you can use the fully qualified name for the enum value (e.g., `~org.accordproject.time.TemporalUnit.days` or `~org.accordproject.time.PeriodUnit.days`).

### Other Changes

1. `now` used to return the current time but is treated in `0.20` like any other variables. If your logic used the variable `now` without declaring it, this will raise a `Variable now not found` error. You should change your logic to use the `now()` function instead.

#### Example

Consider the Ergo logic for the [acceptance of delivery](https://templates.accordproject.org/acceptance-of-delivery@0.12.1.html) clause. Applying the above rules to the code for the `0.13` version:

```ergo
  clause acceptanceofdelivery(request : InspectDeliverable) : InspectionResponse {

    let received = request.deliverableReceivedAt;
    enforce isBefore(received,now) else
      throw ErgoErrorResponse{ message : "Transaction time is before the
deliverable date." }
    ;

    let dur =
      Duration{
        amount: contract.businessDays,
        unit: "days"
      };
    let status =
      if isAfter(now(), addDuration(received, dur))
      then "OUTSIDE_INSPECTION_PERIOD"
      else if request.inspectionPassed
      then "PASSED_TESTING"
      else "FAILED_TESTING"
    ;
    return InspectionResponse{
      status : status,
      shipper : contract.shipper,
      receiver : contract.receiver
    }
  }
```

results in the following new logic for the `0.20` version:

```ergo
  clause acceptanceofdelivery(request : InspectDeliverable) : InspectionResponse {

    let received = request.deliverableReceivedAt;
    enforce isBefore(received,now()) else                // changed to now()
      throw ErgoErrorResponse{ message : "Transaction time is before the
deliverable date." }
    ;

    let dur =
      Duration{
        amount: contract.businessDays,
        unit: ~org.accordproject.time.TemporalUnit.days  // enum value with fully
qualified name
      };
    let status =
      if isAfter(now(), addDuration(received, dur))      // changed to now()
      then OUTSIDE_INSPECTION_PERIOD                     // enum value has no
quotes
      else if request.inspectionPassed
      then PASSED_TESTING                                // enum value has no
quotes
      else FAILED_TESTING                                // enum value has no
quotes
    ;
    return InspectionResponse{
      status : status,
      shipper : contract.shipper,
      receiver : contract.receiver
    }
  }
```

## Command Line Changes

The Command Line interface for Cicero and Ergo has been completely overhauled for consistency. Release `0.20` also features new command line interfaces for Concerto and for the new `markdown-transform` project.

If you are familiar with the previous Accord Project command line interfaces (or if you have scripts relying on the previous version of the command line), here is a list of changes:

1. Ergo: A single new `ergo` command replaces both `ergoc` and `ergorun`
   - `ergoc` has been replaced by `ergo compile`
   - `ergorun execute` has been replaced by `ergo trigger`
   - `ergorun init` has been replaced by `ergo initialize`
   - All other `ergorun <command>` commands should use `ergo <command>` instead

2. Cicero:
   - The `cicero execute` command has been replaced by `cicero trigger`
   - The `cicero init` command has been replaced by `cicero initialize`
   - The `cicero generateText` command has been replaced by `cicero draft`
   - the `cicero generate` command has been replaced by `cicero compile`

Note that several options have been renamed for consistency as well. Some of the main option changes are:
1. `--out` and `--outputDirectory` have both been replaced by `--output`
2. `--format` has been replaced by `--target` in the new `cicero compile` command
3. `--contract` has been replaced by `--data` in all `ergo` commands

For more details on the new command line interface, please consult the corresponding [Cicero CLI](cicero-cli), [Concerto CLI](concerto-cli), [Ergo CLI](ergo-cli), and [Markus CLI](markus-cli) Sections in the reference manual.

## API Changes

A number of API changes may affect Node.js applications using Cicero or Ergo packages. The main API changes are:
1. Ergo:
   1. `@accordproject/ergo-engine` package
      - the `Engine.execute()` call has been renamed `Engine.trigger()`
2. Cicero:
   1. `@accordproject/cicero-core` package
      - the `TemplateInstance.generateText()` call has been renamed `TemplateInstance.draft` **and is now an `async` call**
      - the `Metadata.getErgoVersion()` call has been removed
   2. `@accordproject/cicero-engine` package
      - the `Engine.execute()` call has been renamed `Engine.trigger()`
      - the `Engine.generateText()` call has been renamed `Engine.draft()`

## Cicero Server Changes

Cicero server API has been changed to reflect the new underlying Cicero engine. Specifically:
1. The `execute` endpoint has been renamed `trigger`
2. The path to the sample has to include the `text` directory, so instead of `execute/templateName/sample.txt` it should use `trigger/templateName/text%2Fsample.md`

#### Example

Following the [README.md](https://github.com/accordproject/cicero/blob/master/packages/cicero-server/README.md) instructions, instead of calling:
```
curl -X POST --header 'Content-Type: application/json' --header 'Accept: application/json' http://localhost:6001/execute/latedeliveryandpenalty/sample.txt -d '{ "request": { "$class": "org.accordproject.latedeliveryandpenalty.LateDeliveryAndPenaltyRequest", "forceMajeure": false,"agreedDelivery": "December 17, 2017 03:24:00", "deliveredAt": null, "goodsValue": 200.00 }, "state": { "$class": "org.accordproject.cicero.contract.AccordContractState", "stateId" : "org.accordproject.cicero.contract.AccordContractState#1"}}'
```

You should call:
```
```

curl -X POST --header 'Content-Type: application/json' --header 'Accept: application/json' http://localhost:6001/trigger/latedeliveryandpenalty/sample.md -d '{ "request": { "$class": "org.accordproject.latedeliveryandpenalty.LateDeliveryAndPenaltyRequest", "forceMajeure": false,"agreedDelivery": "December 17, 2017 03:24:00", "deliveredAt": null, "goodsValue": 200.00 }, "state": { "$class": "org.accordproject.cicero.contract.AccordContractState", "stateId" : "org.accordproject.cicero.contract.AccordContractState#1"}}'

---
id: version-0.30.1-ref-migrate-0.20-0.21
title: Cicero 0.20 to 0.21
original_id: ref-migrate-0.20-0.21

The main change between the `0.20` release and the `0.21` release is the new markdown syntax and parser infrastructure based on [`markdown-it`](https://github.com/markdown-it/markdown-it). While most templates designed for `0.20` should still work on `0.21` some changes might be needed to the contract or template text to account for this new syntax.

:::note
Before following those migration instructions, make sure to first install version `0.21` of Cicero, as described in the [Install Cicero](started-installation.md) Section of this documentation.
:::

## Metadata Changes

You should only have to update the Cicero version in the `package.json` for your template to `^0.21.0`. Remember to also increment the version number for the template itself.

#### Example

After those changes, the `accordproject` field in your `package.json` should look as follows (with the `template` field being either `clause` or `contract` depending on the template):
```js
...
    "accordproject": {
        "template": "clause",
        "cicero": "^0.21.0"
    }
...
```

## Text Changes

Both the markdown for the grammar and sample text have been updated and consolidated in the `0.21` release and may require some adjustments. You can find complete information about the latest syntax in the [Markdown Text](markup-preliminaries) Section of this documentation. For an existing template, you should apply the following changes.

### Text Grammar Changes

~~~~
1. Clause or list blocks should have their opening and closing tags on a single
line terminated by whitespace. I.e., you should change occurrences of :
   ```
   {{#clause clauseName}}...clause text...{{/clauseName}}
   ```
   to
   ```
   {{#clause clauseName}}
   ...clause text...
   {{/clauseName}}
   ```
   and similarly for ordered and unordeded list blocks (`olist` and `ulist`).
2. Markdown container blocks are no longer supported inside inline blocks (`if`
`join` and `with` blocks).

:::tip
We recommend using the new [TemplateMark Dingus](https://templatemark-
dingus.netlify.app) to check that your template variables, blocks and formula are
properly identified following the new markdown parsing rules.
:::

### Text Samples Changes

1. Nested clause template should be now identified within contract templates using
clause blocks. I.e., if you use a `paymentClause`, you should change your text
from:
   ```
   ...negate the notices Licensor provides and requires hereunder.

   Payment. As consideration in full for the rights granted herein, Licensee shall
pay Licensor a one-time fee in the amount of "one hundred US Dollars" (100.0 USD)
upon execution of this Agreement, payable as follows: "bank transfer".

   General.
   ...
   ```
   to
   ```
   ...negate the notices Licensor provides and requires hereunder.

   {{#clause paymentClause}}
   Payment. As consideration in full for the rights granted herein, Licensee shall
pay Licensor a one-time fee in the amount of "one hundred US Dollars" (100.0 USD)
upon execution of this Agreement, payable as follows: "bank transfer".
   {{/clause}}

   General.
   ...
   ```
2. The text corresponding to formulas should be changed from `{{ ...text...}}` to
`{{% ...text... %}}`.

You should also ensure that any changes to the grammar text is reflected in the
samples. Any change in the grammar text outside of variables should be applied to
the corresponding `sample.md` files as well.

:::tip
You can check that the samples and grammar are in agreement by using the `cicero
~~~~

parse` command.
:::

## Model Changes

There should be no model changes required for this version.

## Logic Changes

There should be no logic changes required for this version.

## API Changes

A number of API changes may affect Node.js applications using Cicero or Ergo packages. The main API changes are:
1. Cicero:
   1. `@accordproject/cicero-core` package
      - the `ParserManager` class has been completely overhauled and moved to the `@accordproject/@markdown-template` package.

## CLI Changes

1. The `cicero draft --wrapVariables` option has been removed
2. The `ergo draft` command has been removed

## Cicero Server Changes

Cicero server API has been completely overhauled to match the more recent engine interface
1. The contract data is now passed as part of the REST POST request for the `trigger` endpoint

----
---
id: version-0.30.1-ref-migrate-0.21-0.22
title: Cicero 0.21 to 0.22
original_id: ref-migrate-0.21-0.22

The main change between the `0.21` release and the `0.22` release is the switch to version `1.0` of the Concerto modeling language and library. This change comes along with a complete revision for the Accord Project "base models" which define key types for: clause and contract data, parties, obligations and requests / responses. We encourage developers to get familiarized with the [new base models](https://github.com/accordproject/models/tree/master/src/accordproject) before switching to Cicero `0.22`.

:::note
Before following those migration instructions, make sure to first install version `0.22` of Cicero, as described in the [Install Cicero](started-installation.md) Section of this documentation.
:::

## Metadata Changes

You should only have to update the Cicero version in the `package.json` for your template to `^0.22.0`. Remember to also increment the version number for the template itself.

#### Example

After those changes, the `accordproject` field in your `package.json` should look
as follows (with the `template` field being either `clause` or `contract` depending
on the template):
```js
...
    "accordproject": {
        "template": "clause",
        "cicero": "^0.22.0"
    }
...
```

## Text Changes

There should be no text changes required for this version.

## Model Changes

Most templates will require changes to the model and should be re-written against
the new base Accord Project models. Most of the changes should be renaming for key
classes:

1. Contract and Clause data
   1. the `org.accordproject.cicero.contract.AccordContract` class is now
`org.accordproject.contract.Contract` found in
https://models.accordproject.org/accordproject/contract.cto
   2. the `org.accordproject.cicero.contract.AccordClause` class is now
`org.accordproject.contract.Clause` found in
https://models.accordproject.org/accordproject/contract.cto
2. Contract state and parties
   1. the `org.accordproject.cicero.contract.AccordState` class is now
`org.accordproject.runtime.State` found in
https://models.accordproject.org/accordproject/runtime.cto
   2. the `org.accordproject.cicero.contract.AccordParty` class is now
`org.accordproject.party.Party` found in
https://models.accordproject.org/accordproject/party.cto
3. Request and response
   1. the `org.accordproject.cicero.runtime.Request` class is now
`org.accordproject.runtime.Request` found in
https://models.accordproject.org/accordproject/runtime.cto
   2. the `org.accordproject.cicero.runtime.Response` class is now
`org.accordproject.runtime.Response` found in
https://models.accordproject.org/accordproject/runtime.cto
4. Predefined obligations have been moved to their own model file found in
https://models.accordproject.org/accordproject/obligation.cto

:::warning
Some of the properties in those base classes have changed, e.g., the contract state
no longer requires a `stateId`. As a result, corresponding changes to the contract
logic in Ergo or to the application code may be required.
:::

### Example

A typical change to a template model might look as follows, from:
```ergo
```

~~~
import org.accordproject.cicero.contract.* from
https://models.accordproject.org/cicero/contract.cto
import org.accordproject.cicero.runtime.* from
https://models.accordproject.org/cicero/runtime.cto

/**
 * Defines the data model for the Purchase Order Failure
 * template.
 */
asset PurchaseOrderFailure extends AccordContract {
  o AccordParty buyer
  ...
}
```
To:
```ergo
import org.accordproject.contract.* from
https://models.accordproject.org/accordproject/contract.cto
import org.accordproject.runtime.* from
https://models.accordproject.org/accordproject/runtime.cto
import org.accordproject.party.* from
https://models.accordproject.org/accordproject/party.cto
import org.accordproject.obligation.* from
https://models.accordproject.org/accordproject/obligation.cto

asset PurchaseOrderFailure extends Contract {
  --> Party buyer
  ...
}
```

## Logic Changes

Minimal changes to the contract logic should be required, however a few changes to
the base models may affect your Ergo code. Notably:
1. You should import the new Accord Project core models as needed
2. The contract state no longer requires a `stateId` field.
3. The base contract state has been moved to the runtime model, which may need to
be imported

## API Changes

A number of API changes may affect Node.js applications using Cicero or Ergo
packages. The main API changes are:
1. Additional `utcOffset` parameter.
   1. `@accordproject/cicero-core` package
      - the `TemplateInstance.parse` and `TemplateInstance.draft` calls take an
additional `utcOffset` parameter to specify the current timezone offset
   2. `@accordproject/cicero-engine` package
      - the `Engine.init`, `Engine.invoke` and `Engine.trigger` calls take an
additional `utcOffset` parameter to specify the current timezone offset
   3. `@accordproject/ergo-engine` package
      - the `Engine.init`, `Engine.invoke` and `Engine.trigger` calls take an
additional `utcOffset` parameter to specify the current timezone offset
2. New `es6` compilation target for Ergo.
   1. `@accordproject/ergo-compiler` package
      -  the `Compiler.compileToJavaScript` compilation target `cicero` has been
renamed to `es6`
   2. `@accordproject/cicero-core` package
~~~

      - the `Template.toArchive` compilation target `cicero` has been renamed to `es6`

## CLI Changes

1. Specific UTC timezone offset now needs to be passed using the new option `--utcOffset` option has been removed

## Cicero Server Changes

There should be no text changes required for this version.

---
id: version-0.30.1-ref-web-components-overview
title: Overview
original_id: ref-web-components-overview

Accord Project publishes [React](https://reactjs.org) user interface components for use in web applications. Please refer to the web-components project [on GitHub](https://github.com/accordproject/web-components) for detailed usage instructions.

You can preview these components in [the project's storybook](https://ap-web-components.netlify.app/).

## Contract Editor

The Contract Editor component provides a rich-text content editor for contract text with embedded clauses.

> Note that the contract editor does not currently support the full expressiveness of Cicero Templates. Please refer to the Limitations section for details.

### Limitations

The contract editor does not support templates which use the following CiceroMark features:

* Lists containing [nested lists](markup-commonmark.md#nested-lists)
* List blocks [list blocks](markup-commonmark.md#list-blocks)

## Error Logger

The Error Logger component is used to display structured error messages from the Contract Editor.

## Navigation

The Navigation component displays an outline view for a contract, allowing the user to quickly navigate between sections.

## Library

The Library component displays a vertical list of library item metadata, and allows the user to add an instance of a library item to a contract.

---

id: version-0.30.1-started-resources
title: Resources
original_id: started-resources

## Accord Project Resources

- The Main Web site includes latest news, links to working groups, organizational announcements, etc. : https://www.accordproject.org
- This Technical Documentation: https://docs.accordproject.org
- Recording of Working Group discussions, Tutorial Videos are available on Vimeo: https://vimeo.com/accordproject
- Join the [Accord Project Discord](https://discord.com/invite/Zm99SKhhtA) to get involved!

## User Content

Accord Project also maintains libraries containing open source, community-contributed content to help you when authoring your own templates:

- [Model Repository](https://models.accordproject.org/) : a repository of open source Concerto data models for use in templates
- [Template Library](https://templates.accordproject.org/) : a library of open source Clause and Contract templates for various legal domains (supply-chain, loans, intellectual property, etc.)

## Ecosystem & Tools

Accord Project is also developing tools to help with authoring, testing and running accord project templates.

### Editors

- [Template Studio](https://studio.accordproject.org/): a Web-based editor for Accord Project templates
- [VSCode Extension](https://marketplace.visualstudio.com/items?itemName=accordproject.cicero-vscode-extension): an Accord Project extension to the popular [Visual Studio Code](https://visualstudio.microsoft.com/) Editor
- [Emacs Mode](https://github.com/accordproject/ergo/tree/master/ergo.emacs): Emacs Major mode for Ergo (alpha)
- [VIM Plugin](https://github.com/accordproject/ergo/tree/master/ergo.vim): VIM plugin for Ergo (alpha)

### User Interface Components

- [Markdown Editor](https://github.com/accordproject/markdown-editor): a general purpose react component for markdown rendering and editing
- [Cicero UI](https://github.com/accordproject/cicero-ui): a library of react components for visualizing, creating and editing Accord Project templates

## Developers Resources

All the Accord Project technology is being developed as open source. The software packages are being actively maintained on [GitHub](https://github.com/accordproject) and we encourage organizations and individuals to contribute requirements, documentation, issues, new templates, and code.

Join us on the [#technology-wg Discord

channel](https://discord.com/invite/Zm99SKhhtA) for technical discussions and weekly updates.

### Cicero

- GitHub: https://github.com/accordproject/cicero
- Technical Questions and Answers on [Stack Overflow](https://stackoverflow.com/questions/tagged/cicero)

### Ergo

- GitHub: https://github.com/accordproject/ergo
- The [Ergo Language Guide](logic-ergo.md) is a good place to get started with Ergo.
