The Fast Track to daml

The TL;DR; for

This page's source is located here . Pull requests are welcome!

What is…?

Daml is an open-source smart contract language designed to build composable applications on an abstract ledger model.

Daml is a high level language that focuses on data privacy and authorization of distributed applications. These concepts are represented first class in the language.

By abstracting data privacy and authorization, Daml takes the burden off the programmer to think about concrete cryptographic primitives and lets her focus on workflow logic.

Daml is a statically typed functional language.

Applications specified in Daml can be deployed on a growing number of platforms including Amazon Aurora , VMWare Concord , R3 Corda , Hyperledger Fabric , Hyperledger Sawtooth , Project DABL and PostgreSQL .

The full documentation of Daml can be found here .

Concepts

`Party`	A party represents a person or legal entity (for example a bank). Parties can create contracts and exercise choices and are represented by the `Party` data type in Daml.
`Signatories, observers, and controllers`	Signatories, observers, and controllers are parties involved in actions taken on a contract, i.e., actions that are *exercised* on a contract. Signatories, observers, and controllers are therefore represented by the `Party` data type. They control who can read, create and archive a contract.
`Contract`	Contracts are created from blueprints called `templates` - this is the Daml code you write. Templates include: - contract data (e.g., date, description, parties involved etc.) - roles (signatory, observer) - choices and their respective controllers (*who* gets to do *what*) Every contract is a *template instance* stored as a row on the ledger. Contracts are immutable: once they are created on the ledger, the information in the contract cannot be changed. In order to “change” a contract you need to create a new one with the desired contract data.
`Choice`	A choice is something that a party can exercise (take action) on a contract. Choices give you a way to transform the data in a contract: while the contract itself is immutable, you can write a choice that archives the contract and creates a new version of it with the updated data. A choice can only be exercised by its controller and contains the `authorization` of all of the contract’s signatories as well as of the controller.
`Ledger`	The ledger represents the database where all contracts are recorded. More information on Daml Ledgers can be found here.

If you are interested you can find the detailed glossary here and a free online course here.

Command line tools

Install the daml assistant	`curl -sSL https://get.daml.com | sh -s <version>`
Create a new Daml project	`daml new <myproject>`
Create a new Daml/React full stack project	`daml new create-daml-app --template create-daml-app`
Start the IDE	`daml studio`
Build project	`daml build`
Build project, start the sandbox and JSON-API	`daml start`
Start the sandbox ledger (in wall-clock time-mode)	`daml sandbox`
Start the sandbox ledger (in static time-mode)	`daml sandbox --static-time`
Start the JSON-API server (requires a running ledger)	`daml json-api --ledger-host localhost --ledger-port 6865 --http-port 7575`
Upload a dar to the ledger	`daml ledger upload-dar <dar file>`
Run all test scripts and output test coverage report	`daml test --show-coverage --all --files Test.daml`
Project configuration file	`daml.yaml`

Basics

End-of-line comment	`let i = 1 -- This is a comment`
Delimited comment	`{- This is another comment -}`

Every Daml file starts with a module header like this:

```
daml 1.2
module Foo where
```

Types

Type annotation	`var : TypeName`
Builtin types	`Int, Decimal, Numeric n, Text, Bool, Party, Date, Time, RelTime`
Type synonym	`type MyInt = Int`
Lists	`type ListOfInts = [Int]`
Tuples	`type MyTuple = (Int, Text)`
Polymorphic types	`type MyType a b = [(a, b)]`

Data

Record	`data Record = Record { label1 : Int, label2 : Text}`
Product type	`data Product = Product Int Text`
Sum type	`data IntOrText = MyInt Int | MyText Text`
Record with type parameters	`data Record a b = Record {label1 : a, label2 : b}`
Deriving Show/Eq instances	`data Record = Record {label : Int} deriving (Show, Eq)`

Functions

Signature	`f : Text -> Text -> Text`
Definition	`f x y = x <> " " <> y`
Lambda definition	`\x y -> x <> y`
Polymorphic functions	`f : (Show a, Eq a) => a -> Text -> Text`
Function application	`f "hello" "world!"`
Partial application of functions	`salute : Text -> Text`
	`salute = f "Hello"`

Functions are first class members of Daml, in particular, functions can be arguments to functions

```
apply : (Text -> Text) -> Text -> Text
apply h x = h x

apply salute "John" -- "Hello John"
```

Contract Templates

Contract templates describe data that will be stored on the ledger. Templates determine who can read and write data; and by whom and how this data can be altered. A contract template is defined with the `template` keyword:

```
template MyData
  with
    i : Int
    party1 : Party
    party2 : Party
    dataKey : (Party, Text)
  where
    signatory party1
    observer  party2
    key dataKey : (Party, Text)
    maintainer key._1

    choice MyChoice : ()
    ...
```

`with` and `where` are keywords to structure the template.

`signatory`	Observes the contract and its evolution. Gives the signatory’s authority to all the defined contract updates in the contract choices.
`observer`	Observes the contract and its evolution.
`key`	A field of the contract data used as primary index of contracts defined by this template, see `Contract Keys`.
`maintainer`	A set of parties that guarantee uniqueness of contract keys of this template on the ledger, see `Contract Keys`.

Contract keys

Contract keys are unique and stable references to a contract that won’t change even if the contract id of that contract changes due to an update.

Contract keys are optional.

Contract keys have an associated set of key maintainer parties. These parties guarantee the uniquess of their maintained keys.

Contract keys are specified on a contract template with the `key` and `maintainer` keywords. If you specify a `key` you also have to specify its `maintainers`.

`key`	Can be any expression of the contract arguments that does *not* contain a contract id. It *must* include all maintainer parties specified in the `maintainer` field.
`maintainer`	Keys are unique for all specified maintainers. The maintainers need to be a projection of the expression specified with `key`.

Choices

The choices of a contract template specify the rules on how and by whom contract data can be changed.

```
(nonconsuming) choice NameOfChoice : ()
-- optional nonconsuming annotation, name and choice
return type
  with
    party1 : Party          -- choice arguments
    party2 : Party
    i : Int
  controller party1, party2 -- parties that can execute
this choice
    do                      -- the update that will be
executed
      assert (i == 42)
      create ...
      exercise ...
      return ()
```

Choices can be `consuming` or `nonconsuming`.

`consuming`	The default. The contract is consumed by this choice. Trying to exercise another choice on the same contract id will fail.
`nonconsuming`	The contract is not consumed by this choice and more choices can be exercised.

Updates

Updates specify the transactions that will be committed to the ledger. Updates are described within a `do` block:

```
do
  cid <- create NewContract with field1 = 1
                               , field2 = "hello world"
  let answer = 42
  exercise cid SomeChoice with choiceArgument = "123"
  return answer
```

`create`	create an instance of the given template on the ledger `create NameOfTemplate with exampleParameters`
`exercise`	exercise a choice on a given contract by contract id `exercise IdOfContract NameOfChoiceContract with choiceArgument1 = value1`
`exerciseByKey`	exercise a choice on a given contract by contract key `exerciseByKey @ContractType contractKey NameOfChoiceOnContract with choiceArgument1 = value1`
`fetch`	fetch the contract data from the ledger by contract id `fetchedContract <- fetch IdOfContract`
`fetchByKey`	fetch the contract id and data from the ledger by contract key `fetchedContract <- fetchByKey @ContractType contractKey`
`lookupByKey`	check whether a contract with the given key exists and if yes, return the contract id `fetchedContractId <- lookupByKey @ContractType contractKey`
`abort`	abort a transaction with an error message, the transaction will not be committed to the ledger `abort errorMessage`
`assert`	assert that a given predicate holds, otherwise fail the transaction `assert (condition == True)`
`getTime`	get the ledger effective time `currentTime <- getTime`
`return`	return a value from a `do` block `return 42`

`let`	bind a local variable or define a local function within the update `do` block
	`let createContract x = create NameOfContract with issuer = x; owner = x`
	`let answer = 42`

`this`	refers to the current contract data that contains this update in a choice `create NewContract with owner = this.owner`
`forA`	run a for loop of actions over a list `forA [alice, bob, charlie] $ \p -> create NewContract with owner = p`

Scripts

Daml script is a scripting language to run Daml commands against a ledger. For example:

```
module Test where

import Daml.Script

test = Script ()
test = do
  alice <- allocateParty "Alice"
  bob <- allocateParty "Bob"
  c <- submit alice $ createCmd NewContract with ...
  submit bob $ exerciseCmd c Accept with ...
```

Scripts are compiled like usual Daml code to a `dar` package with the `daml build` command.

Running a script	`daml script --dar example-0.0.1.dar --script-name ModuleName:scriptFunction --ledger-host localhost --ledger-port 6865`
Running a script with initial arguments given	`daml script --dar example-0.0.1.dar --input-file arguments_in_damllf_json.json --script-name ModuleName:scriptFunction --ledger-host localhost --ledger-port 6865`
Allocating a party on the ledger	`alice <- allocateParty "Alice"`
List all known parties on the ledger	`parties <- listKnownParties`
Query for a given contract template visible to a given party	`query @ExampleTemplate alice`
Create a new contract	`createCmd ExampleTemplate with ...`
Exercise a choice on a contract	`exerciseCmd contractId ChoiceName with ...`
Exercise a choice on a contract by contract key	`exerciseByKeyCmd contractKey ChoiceName with ...`
Create and then exercise a choice on the created contract	`createAndExerciseCmd (ExampleTemplate with ... ) (ChoiceName with ...)`
Pass time on the ledger (only applicable for a ledger running in **STATIC TIME MODE**, like the in-memory ledger of Daml Studio or `daml test`)	`passTime (hours 10)`
Set time on the ledger (only applicable for a ledger running in **STATIC TIME MODE**, like the in-memory ledger of Daml Studio or `daml test`)	`setTime (time (date 2007 Apr 5) 14 30 05)`

JavaScript/React API

Daml ledgers expose a unified API for interaction.

The following describes how to interact with a ledger using the TypeScript libraries `@daml/ledger`, `@daml/react` in a frontend build with React .

Import the libraries via:

```
import Ledger from @daml/ledger
import {useParty, ...} from @daml/react
```

React entry point:

```
import DamlLeddger from @daml/react

const App: React.FC = () => {
     <DamlLedger
      token: <your authentication token>
      httpBaseUrl?: <optional http base url>
      wsBaseUrl?: <optional websocket base url>
      party: <the logged in party>
    >
      <MainScreen />
    </DamlLedger>
};
```

Get the logged in party	`const party = useParty();`
	`...`
	`<h1> You're logged in as {party} </h1>`

Query the ledger	`const {contracts: queryResult, loading: isLoading, } = useQuery(ContractTemplate, () => ({field: value}), [dep1, dep2, ...])`
Query for contract keys	`const {contracts, loading} = useFetchByKey(ContractTemplate, () => key, [dep1, dep2, ...])`

Reload the query results	`reload = useReload();`
	`...`
	`onClick={() => reload()}`

Query the ledger, returns a refreshing stream	`const {contracts, loading}` = useStreamQuery(ContractTemplate, () => ({field: value}), [dep1, dep2, …]) `
Query for contract keys, returns a refreshing stream	`const {contracts, loading} = useStreamFetchByKey(ContractTemplate, () => key, [dep1, dep2, ...])`

Create a contract on the ledger	`const ledger = useLedger();`
	`const newContract = await ledger.create(ContractTemplate, arguments)`

Archive a contract on the ledger	`const ledger = useLedger();`
	`const archiveEvent = await ledger.archive(ContractTemplate, contractId)`

Exercise a contract choice on the ledger	`const ledger = useLedger();`
	`const [choiceReturnValue, events] = await ledger.exercise(ContractChoice, contractId, choiceArguments)`

DAML resources

- Official documentation
- The Daml code repository
- A Daml project template
- Read about how people are using Daml on the DAML Blog