Ontrack Reference Guide 2.29.8

Damien Coraboeuf

Table of Contents

Chapter 1. Installation

There are several ways to install Ontrack.

1.1. Prerequisites

Ontrack has been tested on different Linux variants (Ubuntu, Debian, CentOS) and should also work on Windows.

Ontrack relies on at least a JDK 8 build 25. More recent versions of the JDK8 are OK. However, no test has been done yet using early JDK 9 versions.

Ontrack runs fine with 512 Mb of memory. However, think of upgrading to 2 Gb of memory if you intend to host a lot of projects. See the different installation modes (Docker, RPM, etc.) to know how to setup the memory settings.

Ontrack stores its data in a local H2 database. This one can grow up to 500 Mb for big volumes (hundreds of projects).

1.2. Installing using Docker

Ontrack is distributed as a Docker image on the Docker Hub, as `nemerosa/ontrack:2.29.8`.

1.2.1. Overview

The Ontrack image exposes the ports `443` and `8080`.

Two volumes are defined:

- `/var/ontrack/data` - contains the data for Ontrack (files & database) but also the log files. This is typically provided through a data volume container.
- `/var/ontrack/conf` - contains the configuration files for Ontrack (see later).

1.2.2. Basic deployment

You can start Ontrack as a container and a shared database and configuration on the host using:

```
docker run --detach \
   --publish=8080:8080 \
   --volume=/var/ontrack/data:/var/ontrack/data \
   --volume=/var/ontrack/conf:/var/ontrack/conf \
   nemerosa/ontrack
```

The configuration files for Ontrack can be put on the host in `/var/ontrack/conf` and the database and working files will be available in `/var/ontrack/data`. The application will be available on port `8080` of the host.

Java options, like memory settings, can be passed to the Docker container using the `JAVA_OPTIONS` environment variable:

```
docker run \
   ...
   --env "JAVA_OPTIONS=-Xmx2048m" \
   ...
```

1.3. RPM installation

You can install Ontrack using a RPM file you can download from the releases page.

The RPM is continuously tested on CentOS 6.7 and CentOS 7.1.

To install Ontrack:

```
rpm -i ontrack.rpm
```

The following directories are created:

Directory	Description
`/opt/ontrack`	Binaries and scripts
`/usr/lib/ontrack`	Working and configuration directory
`/var/log/ontrack`	Logging directory

You can optionally create an `application.yml` configuration file in `/usr/lib/ontrack`. For example, to customise the port Ontrack is running on:

```
server:
  port: 9080
```

Ontrack is installed as a service using `/etc/init.d/ontrack`.

```
# Starting Ontrack
service ontrack start
# Status of Ontrack
service ontrack status
# Stopping Ontrack
service ontrack stop
```

To upgrade Ontrack:

```
# Stopping Ontrack
sudo service ontrack stop
# Updating
sudo rpm --upgrade ontrack.rpm
# Starting Ontrack
sudo service ontrack start
```

The optional `/etc/default/ontrack` file can be used to define the `JAVA_OPTIONS`, for example:

/etc/default/ontrack

```
JAVA_OPTIONS=-Xmx2048m
```

1.4. Debian installation

You can install Ontrack using a Debian file (`.deb`) you can download from the releases page.

To install Ontrack:

```
dpkg -i ontrack.deb
```

The following directories are created:

Directory	Description
`/opt/ontrack`	Binaries and scripts
`/usr/lib/ontrack`	Working and configuration directory
`/var/log/ontrack`	Logging directory

Ontrack is installed as a service using `/etc/init.d/ontrack`.

```
# Starting Ontrack
service ontrack start
# Status of Ontrack
service ontrack status
# Stopping Ontrack
service ontrack stop
```

The optional `/etc/default/ontrack` file can be used to define the `JAVA_OPTIONS`, for example:

/etc/default/ontrack

```
JAVA_OPTIONS=-Xmx2048m
```

1.5. Standalone installation

Ontrack can be downloaded as a JAR and started as a Spring Boot application.

Download the JAR from the Ontrack release page

Start it using `java -jar ontrack.jar` with the following options:

- `--spring.datasource.url=jdbc:h2:/var/ontrack/data/database/data`
- or `--spring.datasource.url=jdbc:h2:./database/data`
- and any other Java option, like memory settings: `-Xmx2048m`
- or configuration parameter like `--server.port=9999`

to specify the location of the H2 database files.

Options can also be specified in an `application.yml` file in the working directory.

For example:

application.yml

```
spring:
   datasource:
      url: "jdbc:h2:/var/ontrack/data/database/data"
```

1.6. Configuration

As a regular Spring Boot application, Ontrack can be configured using system properties and/or property files and/or YAML files. See the Spring Boot documentation for more details.

The way to provide a YAML `application.yml` configuration file or command line arguments will vary according to the installation (Docker, RPM, etc.). See the corresponding section above for more details.

For example, to setup the port Ontrack is running on, you can use the `server.port` property. Using a YAML file:

application.yml

```
server.port=9999
```

or the command line option:

```
--server.port=9999
```

See Configuration properties for the list of all available properties.

Chapter 2. Basics

2.1. Managing projects

2.2. Managing branches

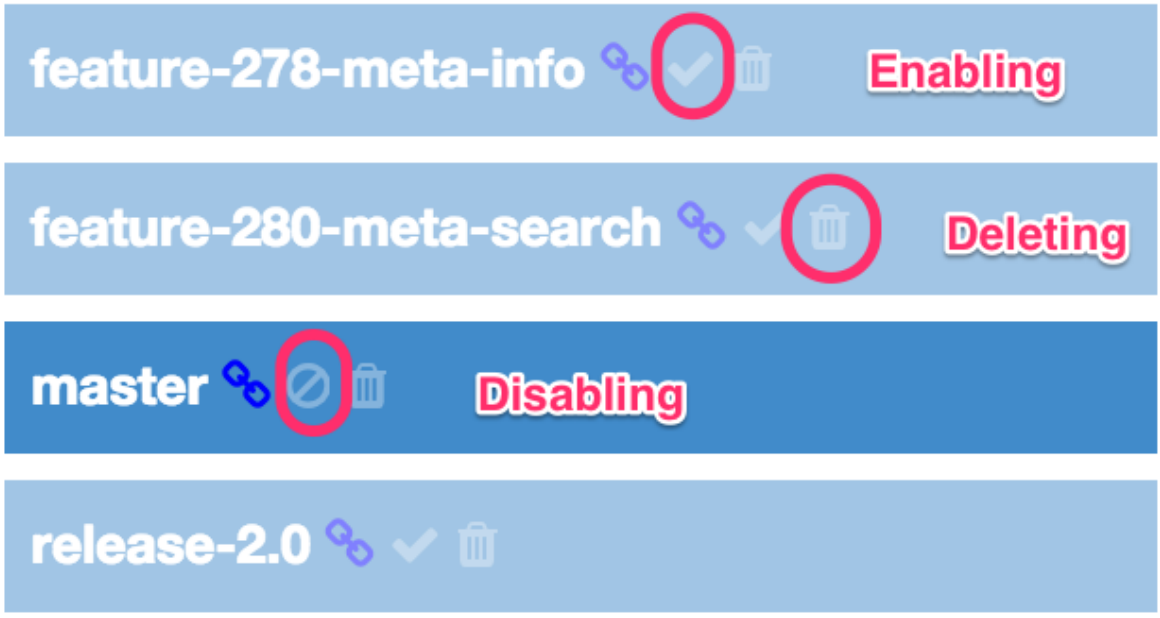
Several branches can be defined per project.

2.2.1. Managing the branches in the project page

If you click on the *Show all branches* button in the project page, you can display all the branches, including the ones being disabled and the templates.

According to your authorizations, the following commands will be displayed as icons just on the right of the branch name, following any other decoration:

- disabling the branch
- enabling the branch
- deleting the branch



This allows you to have a quick access to the management of the branches in a project. Only the deletion of a branch will prompt you about your decision.

2.2.2. Branch templating

In a context where branches are numerous, because the workflow you're working with implies the creation of many branches (feature branches, release branches, ...), each of them associated with its own pipeline, creating the branches by hand, even by cloning or copying them would be too much an effort.

Ontrack gives the possibility to create *branch templates* and to automatically create branches using this template according to a list of branches. This list of branches can either be static or provided by the SCM.

See Branch templates for details about using this feature.

2.2.3. Managing stale branches

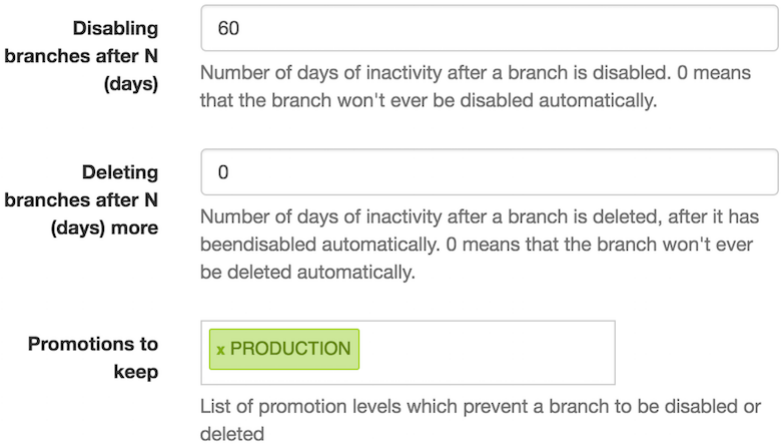
By default, Ontrack will keep all the branches of a project forever. This can lead to a big number of branches to be displayed.

You can configure a project to *disable* branches after a given number of days has elapsed since the last build, and then to *delete* them after an additional number of days has elapsed again.

To configure this:

- go to the project page
- select the *Stale branches* property and add it:

- set the number of days before disabling and the number of days before deleting



If the *disabling* days are set to 0, no branch will be ever disabled or deleted.

If the *deleting* days are set to 0, no branch will ever be deleted.

You can also set a list of promotion levels - a branch which is or has been promoted to such a promotion level will not be eligible for being disabled or deleted.

In the sample above, the stale branches will be disabled after 60 days (not shown any longer by default), and after again 300 days, they will be deleted (so after 360 days in total). Branches which have at least one build being promoted to `PRODUCTION` will not be deleted or disabled.s

Note that the *Stale branches* property can also be set programmatically using the [DSL](DSL Property Stale Branches).

2.3. Managing validation stamps

2.3.1. Auto creation of validation stamps

Creating the validation stamps for each branch, or making sure they are always up to date, can be a non trivial task. Having mechanisms like cloning or templates can help, but then one must still make sure the list of validation stamps in the template is up to date and than the template is regularly synchronized.

Another approach is to allow projects to create automatically the validation stamps on demand, whenever a build is validated. This must of course be authorised at project level and a list of predefined validation stamps must be maintained globally.

Predefined validation stamps

The management of predefined validation stamps is accessible to any *Administrator*, in his user menu.

He can create, edit and delete predefined validation stamps, and associate images with them.

Deleting a predefined validation stamp has no impact on the ones which were created from it in the branches. No link is kept between the validation stamps in the branches and the predefined ones.

Configuring projects

By default, a project does not authorise the automatic creation of a validation stamp. In case one attempts to validate a build using a non existing validation stamp, an error would be thrown.

In order to enable this feature on a project, add the *Auto validation stamps* property to the project and set *Auto creation* to *Yes*.

Disabling the auto creation can be done either by setting *Auto creation* to *No* or by removing the property altogether.

Auto creation of validation stamps

When the auto creation is enabled, build validations using a validation stamp name will follow this procedure:

- if the validation stamp is already defined in the branch, it is of course used
- if the validation stamp is predefined, it is used to create a new one on the branch and is then used
- in any other case, an error is displayed

The auto creation of validation stamps is available only through the DSL or through the API. It is not accessible through the GUI, where only the validation stamps of the branch can be selected for a build validation.

Auto creation of validation stamps when not predefined

You can also configure the project so that validation stamps are created on demand, *even when no predefined validation stamp is created*.

In this case:

- if the validation stamp is already defined in the branch, it is of course used
- if the validation stamp is predefined, it is used to create a new one on the branch and is then used
- in any other case, a new validation stamp is created in the branch, with the requested name (and with an empty image and a default description)

2.4. Managing promotion levels

2.4.1. Auto promotion

By default, a build is promoted explicitly, by associated a promotion with it.

By configuring an auto promotion, we allow a build to be automatically promoted whenever a given list of validations have passed on this build.

For example, if a build had passed integration tests on platforms A, B and C, we can imagine to promote automatically this build to a promotion level, without having to do it explicitly.

In order to configure the auto promotion, go to the promotion level and set the "Auto promotion" property. You then associate the list of validation stamps that must be `PASSED` on a build in order to get this build automatically promoted.

The list of validation stamps can be defined by:

- selecting a fixed list of validation stamps
- selecting the validation stamps based on their name, using `include` and `exclude` regular expressions

A validation stamp defined in the list is *always* taken into account in the auto promotion, whatever the values for the `include` and `exclude` regular promotions.

2.4.2. Auto creation

Creating the promotion levels for each branch, or making sure they are always up to date, can be a non trivial task. Having mechanisms like cloning or templating can help, but then one must still make sure the list of promotion levels in the template is up to date and than the template is

regularly synchronized.

Another approach is to allow projects to create automatically the promotion levels on demand, whenever a build is promoted. This must of course be authorized at project level and a list of predefined promotion levels must be maintained globally.

Predefined promotion levels

The management of predefined promotion levels is accessible to any *Administrator*, in his user menu.

He can create, edit and delete predefined promotion levels, and associate images with them.

Deleting a predefined promotion level has no impact on the ones which were created from it in the branches. No link is kept between the promotion levels in the branches and the predefined ones.

Configuring projects

By default, a project does not authorize the automatic creation of a promotion level. In case one attempts to validate a build using a non existing promotion level, an error would be thrown.

In order to enable this feature on a project, add the *Auto promotion levels* property to the project and set *Auto creation* to *Yes*.

Disabling the auto creation can be done either by setting *Auto creation* to *No* or by removing the property altogether.

Auto creation of promotion levels

When the auto creation is enabled, build promotions using a promotion level name will follow this procedure:

- if the promotion level is already defined in the branch, it is of course used
- if the promotion level is predefined, it is used to create a new one on the branch and is then used
- in any other case, an error is displayed

The auto creation of promotion levels is available only through the DSL or through the API. It is not accessible through the GUI, where only the promotion levels of the branch can be selected for a build promotion.

2.5. Managing the builds

The builds are displayed for a branch.

2.5.1. Filtering the builds

By default, only the 10 last builds of a branch are shown but you have the possibility to create *build filters* in order to change the list of displayed builds for a branch.

The management of filters is done using the *Filter* buttons at the top-left and bottom-left corners of the build list. Those buttons behave exactly the same way. They are not displayed if no build has ever been created for the branch.

Some filters, like *Last build per promotion*, are predefined, and you just have to select them to apply them.

You can create custom filters using the *build filter types* which are in the *New filter* section at the end of the *Filter* menu. You fill in the filter parameters and apply the filter by clicking on *OK*.

If you give your filter a name, this filter will be saved locally for the current branch and can be reused later on when using the same browser on the same machine account. If you are logged, you can save this filter for your account at *ontrack* level so you can reuse it from any workstation.

If the filter is not named, it will be applied all the same but won't be editable nor being able to be saved.

You can delete and edit any of your own filters.

You can disable any filtering by selection *Erase filter*. You would then return to the default: last 10 builds. Note that the saved filters are not impacted by this operation.

Sharing filters

By selecting the *Permalink* option in the *Filter* menu, you update your browser's URL to include information about the current selected filter. By copying this URL and send to another user, this other user will be able to apply the same filter than you, even if he did not create it in the first place.

Even anonymous (unnamed) filters can be shared this way.

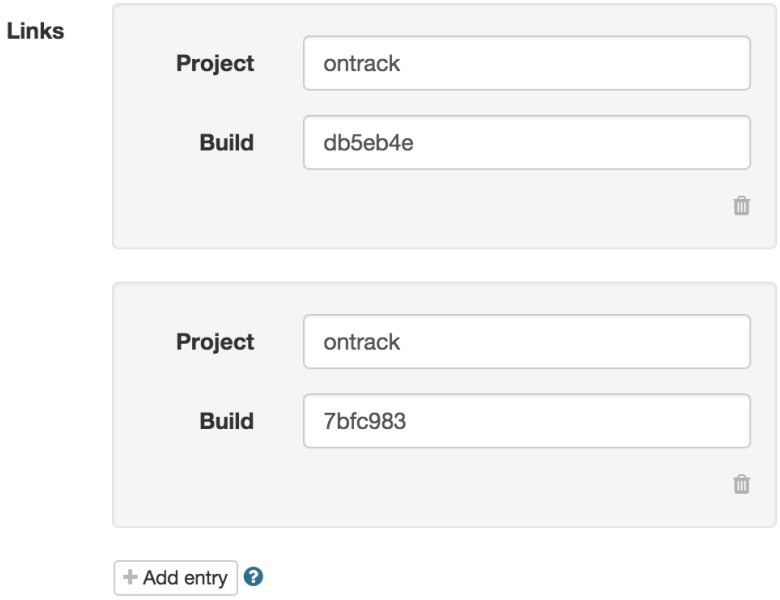
2.5.2. Build links

A build can be linked to other builds. This is particularly useful to represent dependencies between builds and projects.

Definition of links

If authorized, you'll see a *Build links* command at the top of the build page:

Clicking on this link will open a dialog which allows you to define the list of links:

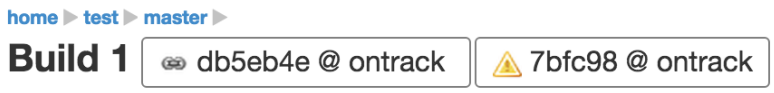


Note that:

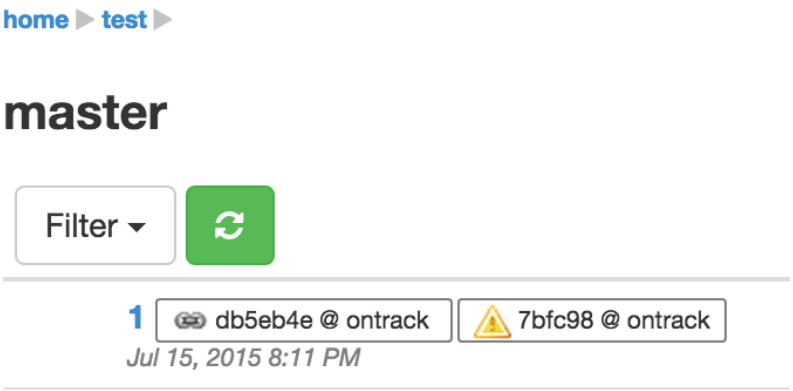
- usually, you'll probably edit those links in an automated process using the DSL
- you cannot define or see links to builds for which the project is not accessible to you

Decorations

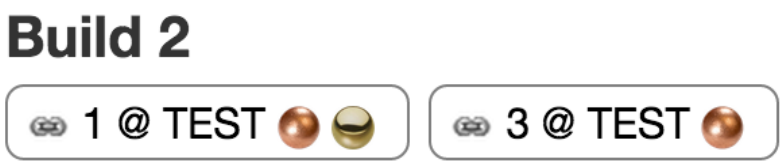
The build links are displayed as decorations in the build page header:



or in the list of builds:



In both cases, the decoration is clickable. If the target build has been promoted, the associated promotions will also be displayed.



Querying

The build links properties can be used for queries:

- in build filters
- in build searches
- in global searches

In all those cases, the syntax to find a match is:

- `project`, `project:` or `project:*` - all builds which contain a build link to the `project` project
- `project:build` - all builds which contain a link to the build `build` in the `project` project
- `project:build*` - all builds which contain a link to a build starting with `build` in the `project` project. The `*` wildcard can be used in any place.

2.6. Properties

All entities can be associated with properties.

include::property-message.adoc

include::property-meta.adoc

Chapter 3. Topics

3.1. Branch templates

In a context where branches are numerous, because the workflow you're working with implies the creation of many branches (feature branches, release branches, ...), each of them associated with its own pipeline, creating the branches by hand, even by cloning or copying them would be too much an effort.

Ontrack gives the possibility to create *branch templates* and to automatically create branches using this template according to a list of branches. This list of branches can either be static or provided by the SCM.

3.1.1. Template Definition

We distinguish between:

- the **branch template definition** - which defines a template for a group of branches
- the **branch template instances** - which are branches based on a template definition

There can be several template definitions per project, each with its own set of template instances.

A **template definition** is a branch:

- which is disabled (not visible by default)
- which has a special decoration for quick identification in the list of branches for a project
- which has a list of template parameters:
- names
- description

whose descriptions and property values use `${name}` expressions where name is a template parameter.

One can create a template definition from any branch following those rules:

- the user must be authorized to manage branch templates for a project
- the branch must not be already a *template instance*
- the branch must not have any existing build

3.1.2. Template Instances

A **template instance** is also a branch:

- which is linked to a template definition
- which has a set of name/values linked to the template parameters

- which has a special decoration for quick identification in the list of branches for a project
- it is a "normal branch" as far as the rest of Ontrack is concerned, but:
  - it cannot be edited
  - no property can be edited not deleted (they are linked to the template definition)

There are several ways to create template instances:

- from a definition, we can create one instance by providing:
  - a name for the instance
  - values for each template parameters
- we can define template synchronization settings linked to a template definition:
  - source of instance names - this is an extension point. This can be:
    - a list of names
    - a list of actual branches from a SCM, optionally filtered. The SCM information is taken from the project definition.
  - an interval of synchronization (manual or every x minutes)
    - a list of template expressions for each template parameter which define how to map an instance name into an actual parameter value (see below)

The actual creation of the instance is done using cloning and copy technics already in place in Ontrack. The replacement is done using the template parameters and their values (computed or not).

The manual creation of an instance follows the same rules than the creation of a branch. If the branch already exists, an error is thrown.

For automatic synchronization from a list of names (static or from a SCM):

- if a previously linked branch does not exist any longer, it is disabled (or deleted directly, according to some additional settings for the synchronization)
- if a branch already exists with the same name, but is not a template instance, a warning is emitted
- if a branch exists already, its descriptions and property values are synched again
- if a branch does not exist, it is created as usual

Reporting about the synchronization (like syncs, errors and warnings) are visible in the *Events* section, in the template definition or in the template instances.

The same synchronization principle applies to branch components: promotion levels, validation stamps and properties.

Finally, at a higher level, cloning a project would also clone the template definitions (not the

instances).

3.1.3. Template expressions

Those expressions are defined for the synchronization between template definitions and template instances. They bind a parameter name and a branch name to an actual parameter value.

A template expression is a string that contains references to the branch name using the `${…}` construct where the content is a Groovy expression where the `branchName` variable is bound to the branch name.

Note that those Groovy expression are executed in a *sand box* that prevent malicious code execution.

Examples

In a SVN context, we can bind the branch SVN configuration (branch location
tag pattern) this way, using simple replacements:

```
branchLocation: branchName -> /project/branches/${branchName}
tagPattern:     branchName -> /project/tags/{build:${branchName}*}
```

In a Jenkins context, we can bind the job name for a branch:

```
jobName: branchName -> PROJECT_${branchName.toUpperCase()}_BUILD
```

3.2. Working with Git

3.2.1. Working with GitHub

GitHub is an enterprise Git repository manager on the cloud or hosted in the premises.

When working with Git in Ontrack, one can configure a project to connect to a GitHub repository.

General configuration

The access to a GitHub instance must be configured.

1. as administrator, go to the *GitHub configurations* menu
2. click on *Create a configuration*
3. in the configuration dialog, enter the following parameters:
    - **Name** - unique name for the configuration
    - URL - URL to the GitHub instance. If left blank, it defaults to the https://github.com location
    - User & Password - credentials used to access GitHub - Ontrack only needs a read access to the repositories

- OAuth2 token - authentication can also be performed using an API token instead of using a user/password pair

The existing configurations can be updated and deleted.

Although it is possible to work with an anonymous user when accessing GitHub, this is not recommended. The rate of the API call will be limited and can lead to some errors.

Project configuration

The link between a project and a GitHub repository is defined by the *GitHub configuration* property:

- **Configuration** - selection of the GitHub configuration created before - this is used for the accesses
- **Repository** - GitHub repository, like `nemerosa/ontrack`
- Indexation interval - interval (in minutes) between each synchronisation (Ontrack maintains internally a clone of the GitHub repositories)
- Issue configuration - issue service. If not set or set to "GitHub issues", the issues of the repository will be used

Branches can be configured for Git independently.

3.2.2. Working with GitLab

GitLab unifies issues, code review, CI and CD into a single UI.

When working with Git in Ontrack, one can configure a project to connect to a GitLab repository.

General configuration

The access to a GitLab instance must be configured.

1. as administrator, go to the *GitLab configurations* menu
2. click on *Create a configuration*
3. in the configuration dialog, enter the following parameters:
   - **Name** - unique name for the configuration
   - **URL** - URL to the GitLab instance (not the repository, the GitLab server)
   - **User** & **Personal Access Token** - credentials used to access GitLab
   - Ignore SSL Certificate - select *Yes* if the SSL certificate for your GitLab instance cannot be trusted by default.

You cannot use the account's password - only Personal Access Tokens are supported.s

The existing configurations can be updated and deleted.

Project configuration

The link between a project and a GitLab repository is defined by the *GitLab configuration* property:

- **Configuration** - selection of the GitLab configuration created before - this is used for the access
- Issue configuration - select the source of issues for this project. This can be any ticketing system (like JIRA) or the built-in issue management for this GitLab project (displayed as "GitLab issues")
- **Repository** - repository name, like `nemerosa/ontrack`
- Indexation interval - how often, in minutes, must the content of this repository be synchronised with Ontrack. Use `0` to not automatically synchronize this repository (this can be done manually).

Branches can be configured for Git independently.

3.2.3. Working with BitBucket

BitBucket is an enterprise Git repository manager by Atlassian.

When working with Git in Ontrack, one can configure a project to connect to a Git repository defined in BitBucket in order to access to the change logs.

General configuration

The access to a BitBucket instance must be configured.

1. as administrator, go to the *BitBucket configurations* menu
2. click on *Create a configuration*
3. in the configuration dialog, enter the following parameters:
    - **Name** - unique name for the configuration
    - **URL** - URL to the Stash instance
    - User & Password - credentials used to access Stash - Ontrack only needs a read access to the repositories

The existing configurations can be updated and deleted.

Project configuration

The link between a project and a Stash repository is defined by the *Stash configuration* property:

- **Configuration** - selection of the Stash configuration created before - this is used for the access and the issues management
- **Project** - name of the *Stash project*
- **Repository** - name of the *Stash repository*
    - Indexation interval - interval (in minutes) between each synchronization (Ontrack

maintains internally a clone of the BitBucket repositories)

- Issue configuration - configured issue service to use when looking for issues in commits.

Branches can be configured for Git independently.

3.2.4. Working with Subversion

Ontrack allows you to configure projects and branches to work with Subversion in order to:

- get change logs
- search for issues linked to builds and promotions
- search for revisions

Subversion configurations

In order to be able to associate projects and branches with Subversion information, an administrator must first define one or several Subversion configurations.

As an administrator, go to the user menu and select *SVN configurations*.

In this page, you can create, update and delete Subversion configurations. Parameters for a Subversion configuration are:

- a name - it will be used for the association with projects
- a URL - Ontrack supports `svn`, `http` and `https` protocols - if the SSL certificate is not recognized by default, some additional configuration must be done at system level.

The URL must be the URL of the *repository*.

- a user and a password if the access to the repository requires authentication
- a tag filter pattern - optional, a regular expression which defines which tags must be indexed
- several URL used for browsing
- indexation interval in minutes (see below)
- indexation start - the revision where to start the indexation from
- issue configuration - issue service associated with this repository

Indexation

Ontrack works with Subversion by indexing some repository information locally, in order to avoid going over the network for each Subversion query.

This indexation is controlled by the parameters of the Subversion configuration: starting revision and interval. If this interval is set to 0, the indexation will have to be triggered manually.

Among the information being indexed, the copy of tags is performed and can be filtered if needed.

In order to access the indexation settings of a Subversion configuration, click on the *Indexation* link.

From the indexation dialog, you can:

- force the indexation from the latest indexed revision
- reindex a range of revisions
- erase all indexed information, and rerun it

The indexations run in background.

Project configuration

You can associate a project with a Subversion configuration by adding the *SVN configuration* property and selecting:

- a Subversion configuration using its name
- a reference path (typically to the `trunk`)

Like all paths in Subversion configurations of projects and branches, this is a *relative* path to the root of the repository. *Not* an absolute URL.

From then on, you can start configuring the branches of the project.

Branch configuration

You can associate a branch with Subversion by adding the *SVN configuration* property and selecting:

- a path to the branch
- a build revision link and its configuration if any

The path to the branch is *relative* to the URL of the SVN repository.

The build commit link defines how to associate a build and a location in Subversion (tag, revision, ...). This link works in both directions since we need also to find builds based on Subversion informations.

Build commit links are extension points - the following are available in Ontrack.

Tag name

The build name is considered a tag name in the `tags` folder for the branch. For example, if the branch path is `/projects/myproject/branches/1.1` then the tags folder is `/projects/myproject/tags` and build names will be looked for in this folder.

No configuration is needed.

Tag pattern name

The build name is considered a tag name in the `tags` folder but must follow a given pattern.

Revision name

The build name is always numeric and represent a revision on the branch path.

No configuration is needed.

Revision pattern

The build name has the branch revision in its name, using the `{revision}` token, following a given pattern. For example, if the pattern is `2.0.*-{revision}`, then all build names must start with `2.0.`, following by anything and suffixed by a revision number.

Build / tag synchronization

For branches whose builds are associated with tags, you have the option to enable a synchronization between the builds in Ontrack and the tags in the Subversion branch.

In the branch page, add the *SVN synchronisation* property and configure it:

Parameter	Description
`override`	If set to `Yes`, the existing builds in Ontrack will be overridden by the tags in Subversion. If set to `No` (the default), the existing builds in Ontrack are never overridden and only new tags are taken into account.
`interval`	The frequency, in minutes, of the synchronization. If set to `0`, the synchronization is not automated and must be triggered manually.

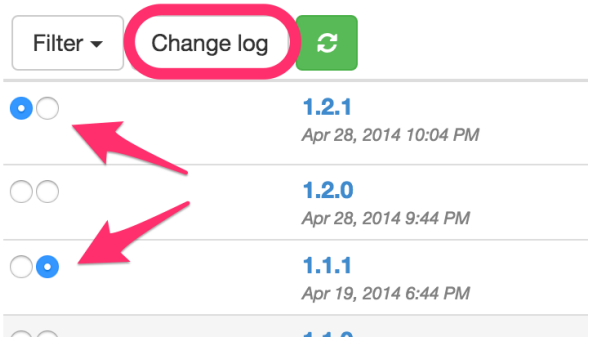
In order to disable globally the tag/build synchronization, without having to change manually all the configured branches, add the following entry in the Ontrack configuration file and restart Ontrack:

application.yml

```
ontrack:
   extension:
      svn:
         build-sync-disabled: true
```

3.3. Change logs

When working with Git or Subversion, you have the opportunity to get a change log between two builds of the same project. In the branch view, two radio buttons columns are available in the build list, which allow you to select the change log boundaries:



The *Change log* button displays the change log page, which contains four sections:

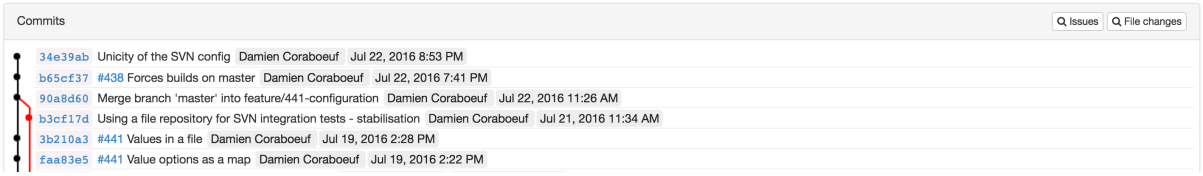
- general information about the two build boundaries
- the commits (for Git) or revision (for Subversion) section
- the issues section
- the file changes selection

Only the first section (build information) is always displayed - the three other ones are displayed only when you request them by clicking on one of the corresponding buttons or links.

Note that the issue section is available only if the corresponding SCM configuration (Git or Subversion) is associated with an issue server (like JIRA or GitHub).

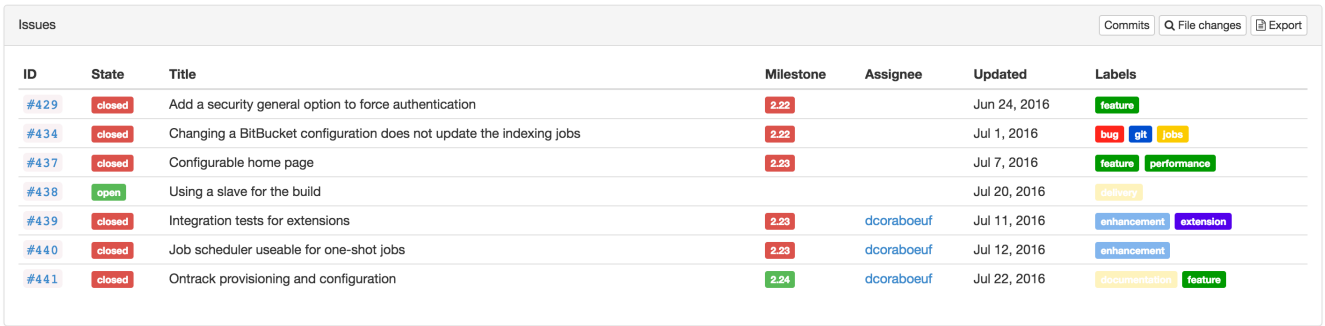
3.3.1. Commits/revisions

This section displays the changes between the two build boundaries. For Git, the associated commit graph is also displayed:



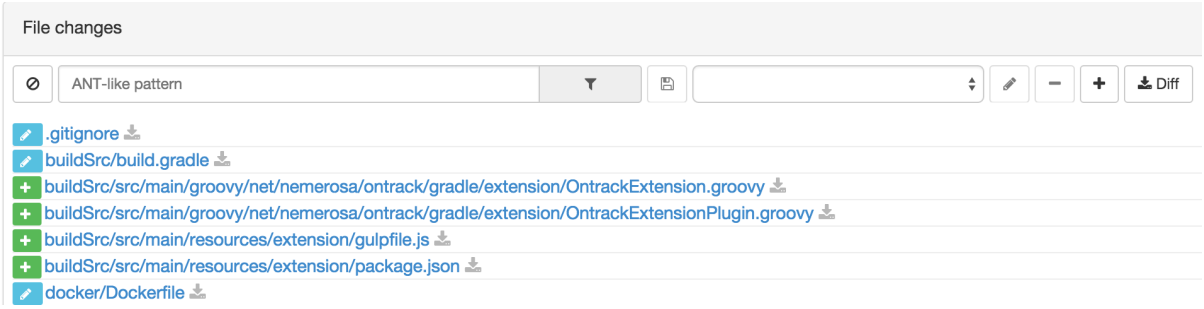
3.3.2. Issues

The list of issues associated with the commits between the two build boundaries is displayed here:



Chapter 4. File changes

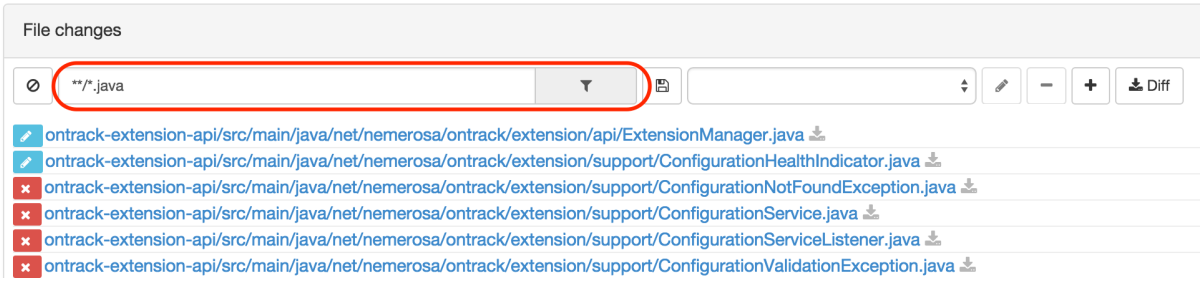
The list of file changes between the two build boundaries is displayed here:



Each file change is associated with the corresponding changes. This includes the list of revisions for Subversion.

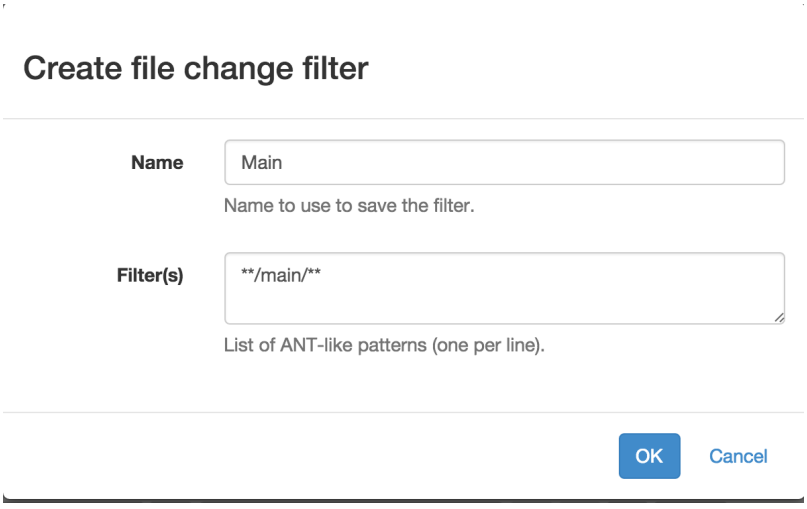
Additionally, you can define filters on the file changes, in order to have access to a list of files impacted by the change log.

By entering a ANT-like pattern, you can display the file paths which match:

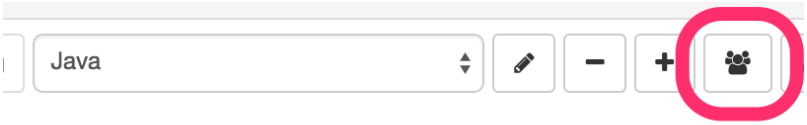


For more complex selections, you can clock on the *Edit* button and you'll have a dialog box which allows you to define:

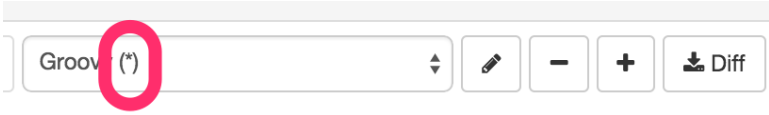
- a name for your filter
- a list of ANT-like patterns to match



If you are authorized, you can also save this filter for the project, allowing its selection by all users.



In the list of filters, you find the filters you have defined and the ones which have been shared for the whole project. The latter ones are marked with an asterisk (*):



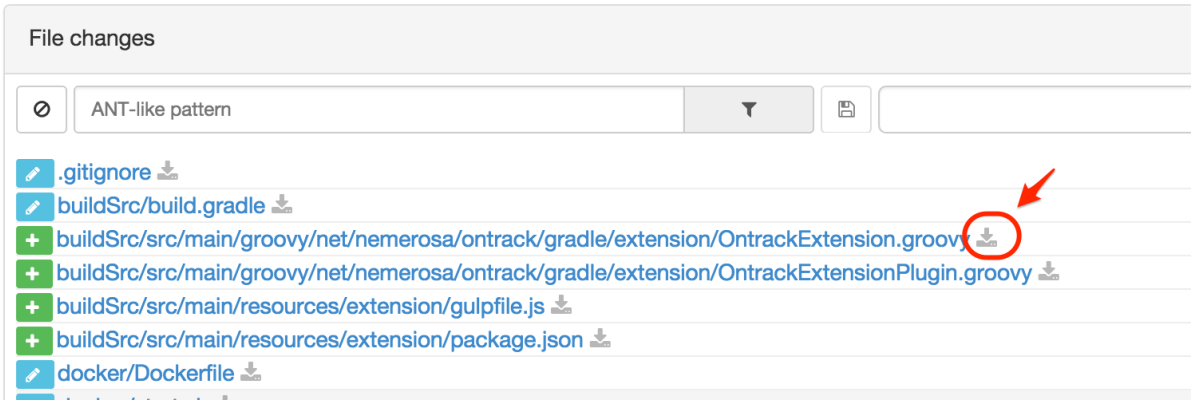
You can update and delete filters. Note that the shared filters won't be actually updated or deleted, unless you are authorized.

Finally, you can get the unified diff for the selected filter by clicking on the *Diff* button:

This will display a dialog with:

- the unified diff you can copy
- a permalink which allows you download the diff from another source

You can obtain a quick diff on one file by clicking on the icon at the right of a file in the change log:



Chapter 5. Administration

5.1. Security

The Ontrack security is based on accounts and account groups, and on authorizations granted to them.

5.1.1. Concepts

Each action in Ontrack is associated with an *authorisation function* and those functions are grouped together in *roles* which are granted to *accounts* and *account groups*.

An *account* can belong to several *account groups* and his set of final *authorisation functions* will be the aggregation of the rights given to the account and to the groups.

5.1.2. Roles

As of now, only roles can be assigned to groups and accounts, and the list of roles and their associated functions is defined by Ontrack itself.

Ontrack distinguishes between *global roles* and _project roles.

Global roles

An **ADMINISTRATOR** has access to all the functions of Ontrack, in all projects. At least such a role should be defined.

By default, right after installation, a default `admin` account is created with the `ADMINISTRATOR` role, having *admin* as password. This password should be changed as soon as possible.

A **CREATOR** can create any project and can, on all projects, configure them, create branches, manage branch templates, create promotion levels and validation stamps. This role should be attributed to service users in charge of automating the definition of projects and branches.

An **AUTOMATION** user can do the same things than a *CREATOR* but can, on all projects, additionally edit promotion levels and validation stamps, create builds, promote and validate them, synchronize branches with their template, manage account groups and project permissions. This role is suited for build and integration automation (CI).

A **CONTROLLER** can, on all projects, create builds, promote and validate them, synchronize branches with their template. It is suited for a basic CI need when the Ontrack structure already exists and does not need to be created.

A **READ_ONLY** can view all projects, but cannot perform any action on them.

The global roles can only be assigned by an *administrator*, in the *Account management* page, by going to the *Global permissions* command.

A *global permission* is created by associating:

- a *permission target* (an account or a group)
- a *global role*

Creation:

1. type the first letter of the account or the group you want to add a permission for
2. select the account or the group
3. select the role you want to give
4. click on *Submit*

Global permissions are created or deleted, not updated.

Project roles

A project **OWNER** can perform all operations on a project but to delete it.

A project **PARTICIPANT** has the right to see a project and to add comments in the validation runs (comment + status change).

A project **VALIDATION_MANAGER** can manage the validation stamps and create/edit the validation runs.

A project **PROMOTER** can create and delete promotion runs, can change the validation runs statuses.

A project **PROJECT_MANAGER** cumulates the functions of a PROMOTER and of a VALIDATION_MANAGER. He can additional manage the common build filters.

A project **READ_ONLY** user can view this project, but cannot perform any action on it.

Only project owners, automation users and administrators can grant rights in a project.

In the project page, select the *Permissions* command.

A *project permission* is created by associating:

- a *permission target* (an account or a group)
- a *project role*

Creation:

1. type the first letter of the account or the group you want to add a permission for
2. select the account or the group
3. select the role you want to give
4. click on *Submit*

Project permissions are created or deleted, not updated.

5.1.3. Accounts

Accounts are created with either:

- built-in authentication, with a password stored and encrypted in Ontrack itself
- LDAP setup

Built-in accounts

An *administrator* can create accounts. He must give them:

- a unique name
- a unique email
- a display name
- an initial password

Any user can change his own password by going to the *Change password* menu.

The *administrator* can give an account a list of global or project roles.

LDAP accounts

Accounts whose authentication is managed by the LDAP are not created directly but are instead created at first successful login.

As for the other types of accounts, the *administrator* can give them a list of global or project roles.

5.1.4. Account groups

An *administrator* can create groups using a name and a description, and assign them a list of global or project roles.

An account can be assigned to several groups.

If LDAP is enabled, some LDAP groups can be mapped to the account groups.

5.1.5. General settings

By default, all users (including anonymous ones) have access to all the projects, at least in read only mode.

You can disable this anonymous access by goint go to the *Settings* and click the *Edit* button in the *General* section. There you can set the *Grants project view to all* option to *No*.

5.2. LDAP setup

It is possible to enable authentication using a LDAP instance and to use the LDAP-defined groups to map them against Ontrack groups.

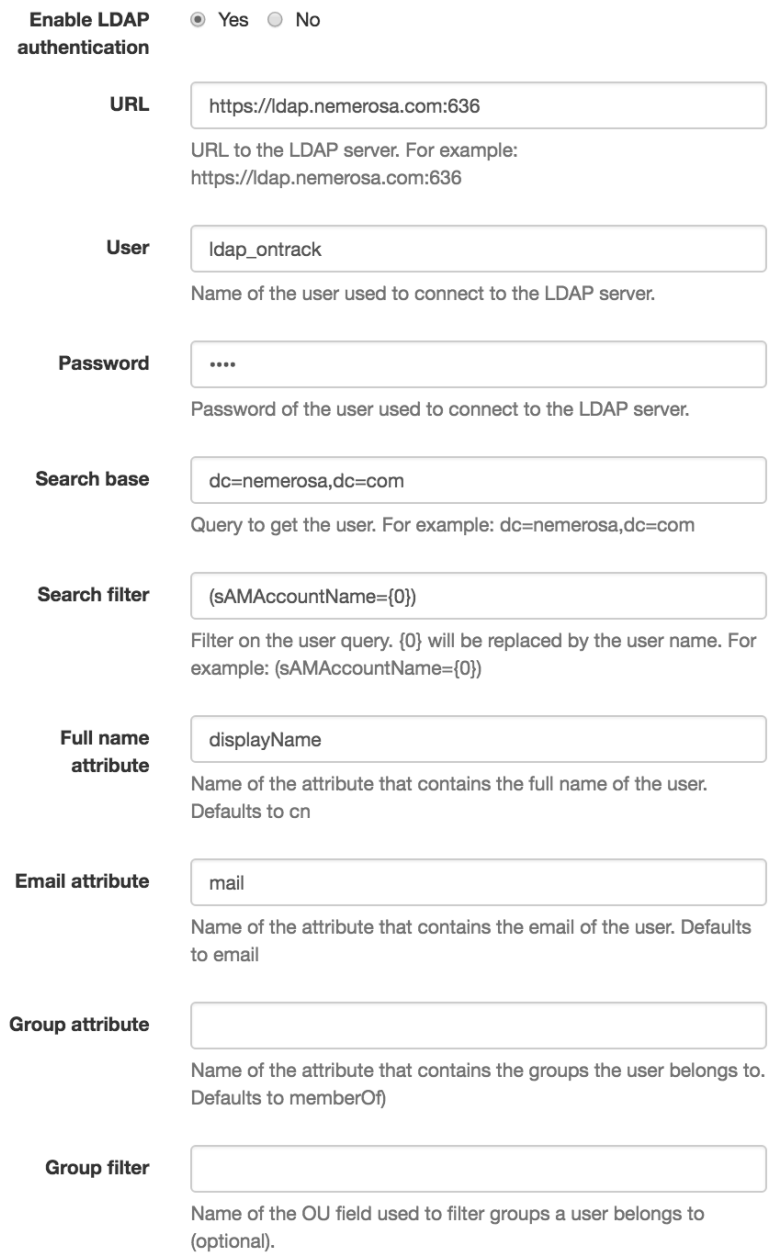
5.2.1. LDAP general setup

As an *administrator*, go to the *Settings* menu. In the *LDAP settings* section, click on *Edit* and fill the following parameters:

- *Enable LDAP authentication*: Yes
- *URL*: URL to your LDAP
- *User* and *Password*: credentials needed to access the LDAP
- *Search base*: query to get the user
- *Search filter*: filter on the user query
- *Full name attribute*: attribute which contains the full name, `cn` by default
- *Email attribute*: attribute which contains the email, `email` by default
- *Group attribute*: attribute which contains the list of groups a user belongs to, `memberOf` by default
- *Group filter*: optional, name of the OU field used to filter groups a user belongs to

As of version 2.14, the list of groups (indicated by the `memberOf` attribute or any other attribute defined by the *Group attribute* property) is not searched recursively and that only the direct groups are taken into account.

For example:



The settings shown above are suitable to use with an Activate Directory LDAP instance.

5.2.2. LDAP group mapping

A LDAP group a user belongs to can be used to map onto an Ontrack group.

As an *administrator*, go to the *Account management* menu and click on the *LDAP mapping* command.

This command is only available if the LDAP authentication has been enabled in the general settings.

To add a new mapping, click on *Create mapping* and enter:

- the *name* of the LDAP group you want to map

- the Ontrack *group* which must be mapped

For example, if you map the `ontrack_admin` LDAP group to an *Administrators* group in Ontrack, any user who belongs to *ontrack_admin* will automatically be assigned to the *Administrators* group when connecting.

This assignment based on mapping is dynamic only, and no information is stored about it in Ontrack.

Note that those LDAP mappings can be generated using the DSL.

Existing mappings can be updated and deleted.

5.3. Administration console

The *Administration console* is available to the *Administrators* only and is accessed through the user menu.

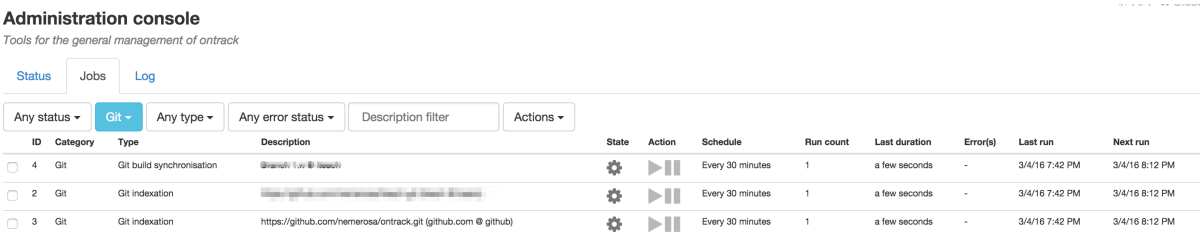
It allows an administrator to:

- manage the running [jobs](Architecture: jobs)
- see the state of the external connections
- see the list of extensions

5.3.1. Managing running jobs

The list of all registered jobs is visible to the administrator. From there, you can see:

- general informations about the jobs: name, description
- the run statistics



Filtering the jobs

The following filters are available:

- status
- *idle jobs*: jobs which are scheduled, but not running right now
- *running jobs*: jobs which are currently running
- *paused jobs*: jobs which are normally scheduled but which have been paused
- *disabled jobs*: jobs which are currently disabled

- *invalid jobs*: jobs which have been marked as invalid by the system (because their context is no longer applicable for example)
- category and type of the job
- error status - jobs whose last run raised an error
- description - filtering using a search token on the job description

Controlling the jobs

For one job, you can:

- force it to run now, if not already running or disabled
- pause it if it is a scheduled job
- resume it if it was paused
- remove it if it is an invalid job

You can also pause or resume all the jobs using the *Actions* menu. All jobs currently selected through the filter will be impacted.

The same *Actions* menu allows also to clear the current filter and to display all the jobs.

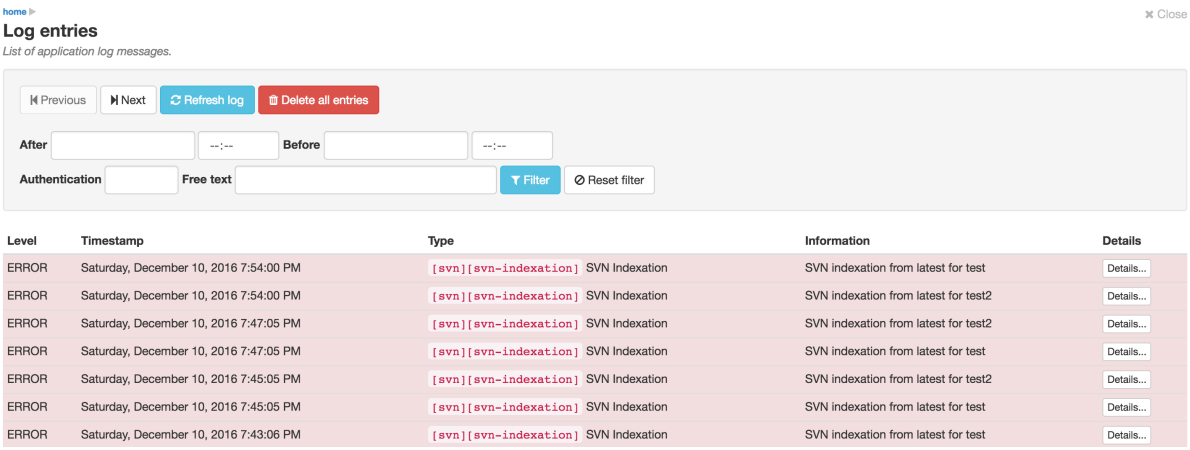
5.3.2. External connections

5.3.3. List of extensions

5.4. Application log messages

The list of application log messages is available to the *Administrators* only and is accessed through the user menu.

It allows an administrator to manage the error messages.



The log items are displayed from the most recent to the oldest. By default, only 20 items are displayed on a page. You can navigate from page to page by using the *Previous* and *Next* buttons.

You can filter the log entries you want to see by using the filter fields:

- after - only log entries created *after* this time will be displayed
- before - only log entries created *before* this time will be displayed
- authentication - you can enter the name of a user, and only errors having occurred to this user will be displayed.
- free text - this text will be searched in all other fields of the log message: details, information, type.

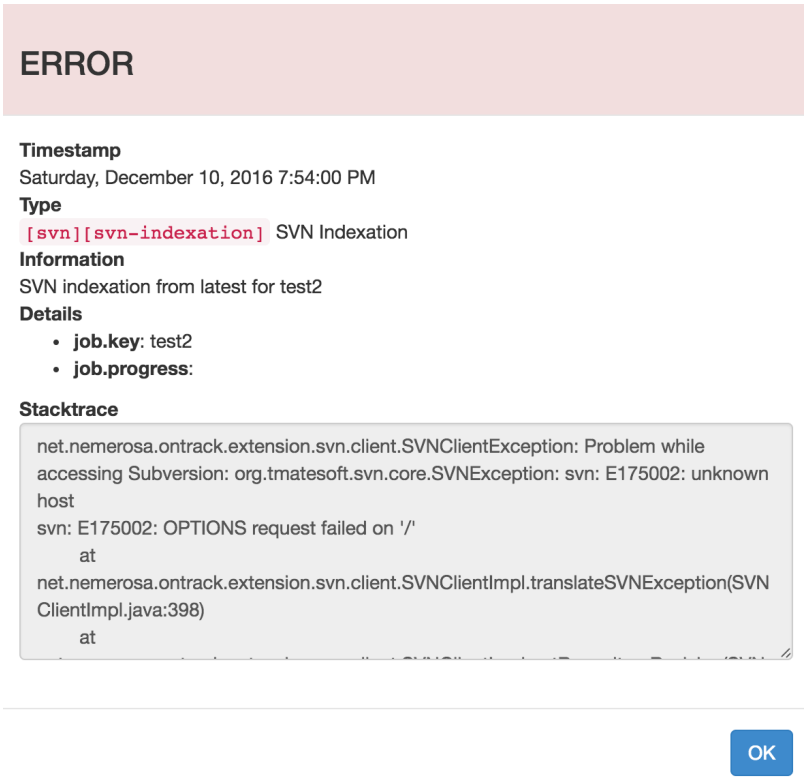
Click on the *Filter* button to activate the filter and on *Reset filter* to delete all fields.

You can refresh the log entries by clicking the *Refresh log* button.

Finally, you can remove *all* log entries (all of them, independently from the current filter) by clicking on the *Delete all entries* button. A confirmation will be asked.

Log entries are kept only for 7 days. This delay can be configured. See the documentation for more information.

You can click on the *Details...* button of a log entry to get more details about the error:



If available, the stack trace can be selected and copied (actually, like any other element of this dialog). Dismiss the dialog by clicking on the *OK* button.

Chapter 6. Integration

6.1. Integration with Jenkins

The best way to integration Ontrack with your Jenkins instance is to use the Ontrack plug-in.

Look at its documentation for details about its configuration and how to use it.

6.2. Monitoring

Ontrack is based on Spring Boot and exports metrics and health indicators that can be used to monitor the status of the applications.

6.2.1. Health

The `/manage/health` end point provides a JSON tree which indicates the status of all connected systems: JIRA, Jenkins, Subversion repositories, Git repositories, etc.

Note than an administrator can have access to this information as a dashboard in the *Admin console* (accessible through the user menu).

6.2.2. Metrics

The default Spring Boot metrics, plus specific ones for Ontrack, are accessible at the `/manage/metrics` end point.

Additionally, Ontrack can also export those metrics to a Graphite or InfluxDB back-end, to be displayed, for example, by Grafana.

Export to Graphite

The following options must be passed to Ontrack, either on the command line or in a local `application.yml` file:

- `ontrack.metrics.graphite.host` (required to enable Graphite export) host name or IP of the Graphite backend
- `ontrack.metrics.graphite.port` (defaults to `2003`) port of the Graphite backend
- `ontrack.metrics.graphite.period` (defaults to `60`) interval (in seconds) at which the data is sent to Graphite

For example, the following `application.yml` enables export to the `graphite` host:

```
ontrack:
   metrics:
      graphite:
         host: graphite
```

Export to InfluxDB

The following options must be passed to Ontrack, either on the command line or in a local `application.yml` file:

- `ontrack.metrics.influxdb.host` (required to enable InfluxDB export) host name or IP of the InfluxDB backend
- `ontrack.metrics.influxdb.port` (defaults to `8086`) port of the InfluxDB backend
- `ontrack.metrics.influxdb.user` (defaults to `root`) user used to connect to InfluxDB
- `ontrack.metrics.influxdb.password` (defaults to `root`) password used to connect to InfluxDB
- `ontrack.metrics.influxdb.database` (defaults to `ontrack`) name of the database in InfluxDB to send metrics to
- `ontrack.metrics.influxdb.period` (defaults to `60`) interval (in seconds) at which the data is sent to InfluxDB
- `ontrack.metrics.influxdb.retention-policy` (defaults to `autogen`) retention policy. Use `default` for InfluxDB version < 1.0.0

For example, the following `application.yml` enables export to the `influxdb` host:

```
ontrack:
   metrics:
      influxdb:
         host: influxdb
```

Grafana dashboard

You can download this sample dashboard as a starting point for defining an Ontrack monitoring system in Grafana:

1. download the `grafana.json` JSON file
2. import it in Grafana

List of metrics

Provided by Spring Boot:

- system metrics
- datasource metrics
- session metrics
- HTTP responses

Job metrics:

- `job-category.Category.*` - [timer metrics](https://dropwizard.github.io/metrics/3.1.0/manual/core/#timers) for the jobs in a given

category (for example: `ontrack.counter.job-category.SVNIndexLatest`)

- `job.*` - [timer metrics](https://dropwizard.github.io/metrics/3.1.0/manual/core/#timers) for all the jobs
- `gauge.jobs` - total number of jobs
- `gauge.jobs.running` - number of running jobs
- `gauge.jobs.disabled` - number of disabled jobs
- `gauge.jobs.error` - number of jobs in error
- `gauge.jobs.invalid` - number of invalid
- `gauge.jobs.paused` - number of invalid
- `gauge.jobs.<category>` - total number of jobs for a given category
- `gauge.jobs.<category>.running` - number of running jobs for a given category
- `gauge.jobs.<category>.disabled` - number of disabled jobs for a given category
- `gauge.jobs.<category>.error` - number of jobs in error for a given category
- `gauge.jobs.<category>.invalid` - number of invalid jobs for a given category
- `gauge.jobs.<category>.paused` - number of paused jobs for a given category

Tagged job metrics:

Only since Ontrack 2.27 and only in InfluxDB. Those metrics are associated with tags.

- `ontrack.job.count` - total number of jobs for a given job
- `ontrack.job.running` - number of running jobs for a given job
- `ontrack.job.disabled` - number of disabled jobs for a given job
- `ontrack.job.error` - number of jobs in error for a given job
- `ontrack.job.invalid` - number of invalid jobs for a given job
- `ontrack.job.paused` - number of paused jobs for a given job

Following tags are associated with each of those measurements:

- `category` of the job
- `type` of the job
- `jobId` of the job

Entity metrics:

- `gauge.entity.project` - number of projects
- `gauge.entity.branch` - number of branches
- `gauge.entity.build` - number of builds
- `gauge.entity.promotionLevel` - number of promotion levels

- `gauge.entity.promotionRun` - number of promotion runs
- `gauge.entity.validationStamp` - number of validation stamps
- `gauge.entity.validationRun` - number of validation runs
- `gauge.entity.validationRunStatus` - number of validation run statuses
- `gauge.entity.property` - number of properties attached to the entities
- `gauge.entity.event` - number of generated events

Application errors metrics:

- `counter.errors` - number of errors which have occurred in Ontrack - using a derivative of this value can help having an idea of the frequency of errors
- `counter.errors.*` - number of errors for a given category, like `GitService` or `UIErrorHandler`)

6.3. GraphQL support

Since version 2.29, Ontrack provides some support for GraphQL.

While most of the Ontrack model is covered, only the query mode is supported right now. Support for mutations might be integrated in later releases.

The GraphQL end point is available at the `/graphql` context path. For example, if Ontrack is available at http://localhost:8080, then the GraphQL end point is available at http://localhost:8080/graphql.

Ontrack supports all capabilities of GraphQL schema introspection.

Example of a GraphQL query, to get the list of branches for a project:

```
{
   projects (id: 10) {
      branches {
         id
         name
      }
   }
}
```

6.4. Calling with Curl

One basic way to integration with the GraphQL interface of Ontrack is to use Curl.

Given the following file:

query.json

```
{
   query: "{ projects (id: $projectId) { branches { id name }}}",
   variables: {
      projectId: 10
   }
}
```

You can `POST` this file to the Ontrack GraphQL end point, for example:

```
curl -X POST --user user http://localhost:8080/graphql --data @query.json
```

6.5. Using the DSL

The simplest way is to use the Ontrack DSL to run a query:

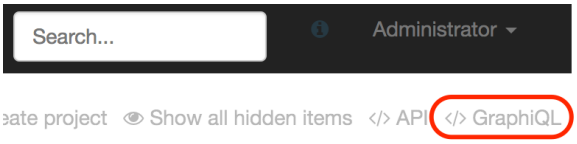
```
def result = ontrack.graphQLQuery(
   '''{
      projects (id: $projectId) {
         id
         branches {
            id
            name
         }
      }
   }''',
   [
      projectId: 10,
   ]
)

assert result.errors != null && result.errors.empty
assert result.data.projects.size() == 1
assert result.data.projects.get(0).id == 10
```

See the `graphQLQuery` documentation for more details.

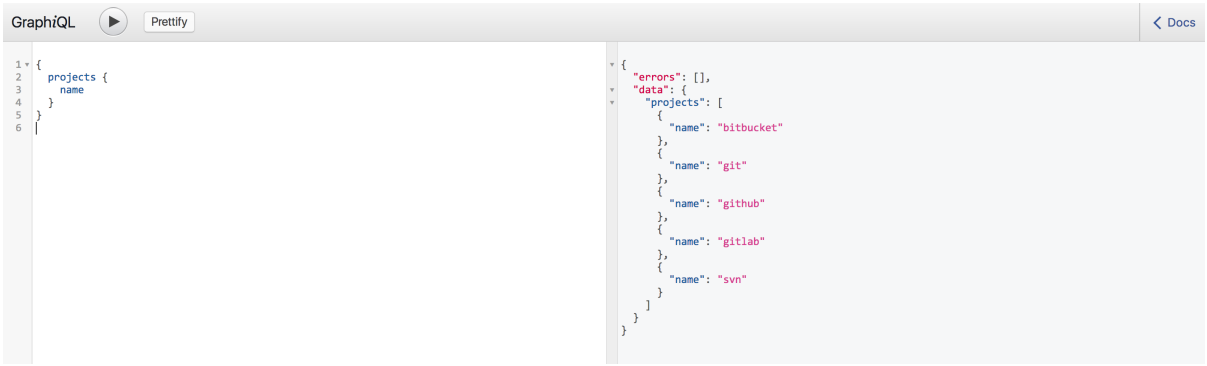
6.6. GraphiQL support

Ontrack supports GraphiQL and allows to experiment with Ontrack GraphQL queries directly in your browser.

You can access the GraphiQL IDE page by clicking on the *GraphiQL* command in the top right corner of the home page of Ontrack:



You can then type and experiment with your GraphQL queries:



The access rights used for your GraphQL queries are inherited from your Ontrack connection. Connect with your user in Ontrack before switching to the GraphiQL IDE. Login in from within GraphiQL is not supported yet.

6.7. Extending the GraphQL schema

The core Ontrack GraphQL query schema can be extended by custom extensions.

See Extending GraphQL for more information.

Chapter 7. DSL

Ontrack provides several ways of interaction:

- the graphical user interface (GUI)
- the REST API (UI - also used internally by the GUI)
- the Domain Specific Language (DSL)

Using the DSL, you can write script files which interact remotely with your Ontrack instance.

7.1. DSL Usage

In some cases, like when using the Ontrack Jenkins plug-in, you can just write some Ontrack DSL to use it, because the configuration would have been done for you.

In some other cases, you have to set-up the Ontrack DSL environment yourself.

7.1.1. Embedded

You can embed the Ontrack DSL in your own code by importing it.

Using Maven:

```
<dependencies>
   <groupId>net.nemerosa.ontrack</groupId>
   <artifactId>ontrack-dsl</artifactId>
   <version>{{ontrack-version}}</version>
</dependencies>
```

Using Gradle:

```
compile 'net.nemerosa.ontrack:ontrack-dsl:{{ontrack-version}}'
```

7.1.2. Standalone shell

See DSL Tool.

7.1.3. Connection

Before calling any DSL script, you have to configure an `Ontrack` instance which will connect to your remote Ontrack location:

```
import net.nemerosa.ontrack.dsl.*;

String url = "http://localhost:8080";
String user = "admin";
String password = "admin";

Ontrack ontrack = OntrackConnection.create(url)
   // Logging
   .logger(new OTHttpClientLogger() {
      public void trace(String message) {
         System.out.println(message);
      }
   })
   // Authentication
   .authenticate(user, password)
   // OK
   .build();
```

7.1.4. Retry mechanism

By default, if the remote Ontrack API cannot be reached, the calls will fail. You can enable a retry mechanism by defining a maximum number of retries and a delay between the retries (defaults to 10 seconds):

```
Ontrack ontrack = OntrackConnection.create(url)
    // ...
    // Max retries
    .maxTries(10)
    // Delay between retries (1 minute here)
    .retryDelaySeconds(60)
    // OK
    .build();
```

7.1.5. Calling the DSL

The Ontrack DSL is expressed through Groovy and can be called using the `GroovyShell`:

```
import groovy.lang.Binding;
import groovy.lang.GroovyShell;

Ontrack ontrack = ...

Map<String, Object> values = new HashMap<>();
values.put("ontrack", ontrack);
Binding binding = new Binding(values);

GroovyShell shell = new GroovyShell(binding);

Object shellResult = shell.evaluate(script);
```

7.2. DSL Samples

7.2.1. DSL Security

The DSL allows to manage the accounts and the account groups.

Management of accounts

To add or update a *built-in* account:

```
ontrack.admin.account(
   "dcoraboeuf",              // Name
   "Damien Coraboeuf",        // Display name
   "dcoraboeuf@nemerosa.net", // Email
   "my-secret-password",      // Password
   [                          // List of groups (optional)
      "Group1",
      "Group2"
   ]
)
```

To get the list of accounts:

```
def accounts = ontrack.admin.accounts
def account = accounts.find { it.name == 'dcoraboeuf' }
assert account != null
assert account.fullName == "Damien Coraboeuf"
assert account.email == "dcoraboeuf@nemerosa.net"
assert account.authenticationSource.allowingPasswordChange
assert account.authenticationSource.id == "password"
assert account.authenticationSource.name == "Built-in"
assert account.role == "USER"
assert account.accountGroups.length == 2
```

LDAP accounts cannot be created directly. See the documentation for more details.

Account permissions

To give a role to an account:

```
ontrack.admin.setAccountGlobalPermission(
   'dcoraboeuf', "ADMINISTRATOR
)
ontrack.project('PROJECT')
ontrack.admin.setAccountProjectPermission(
   'PROJECT', 'dcoraboeuf', "OWNER
)
```

To get the list of permissions for an account:

```
def permissions = ontrack.admin.getAccountProjectPermissions('PROJECT', 'dcoraboeuf')
assert permissions != null
assert permissions.size() == 1
assert permissions[0].id == 'OWNER'
assert permissions[0].name == 'Project owner'
```

Management of account groups

To add or update an account group:

```
ontrack.admin.accountGroup('Administrators', "Group of administrators")
```

To get the list of groups:

```
def groups = ontrack.admin.groups
def group = groups.find { it.name == 'Administrators' }
assert group.name == 'Administrators'
assert group.description == "Group of administrators"
```

Account group permissions

To give a role to an account group:

```
ontrack.admin.setAccountGroupGlobalPermission(
   'Administrators', "ADMINISTRATOR"
)
ontrack.project('PROJECT')
ontrack.admin.setAccountGroupProjectPermission(
   'PROJECT', 'Administrators', "OWNER"
)
```

To get the list of permissions for an account group:

```
def permissions = ontrack.admin.getAccountGroupProjectPermissions('PROJECT',
'Administrators')
assert permissions != null
assert permissions.size() == 1
assert permissions[0].id == 'OWNER'
assert permissions[0].name == 'Project owner'
```

DSL LDAP mapping

The LDAP mappings can be generated using the DSL.

To add or update a LDAP mapping:

```
ontrack.admin.ldapMapping 'ldapGroupName', 'groupName'
```

To get the list of LDAP mappings:

```
LDAPMapping mapping = ontrack.admin.ldapMappings[0]
assert mapping.name == 'ldapGroupName'
assert mapping.groupName == 'groupName'
```

7.2.2. DSL Images and documents

Some resources can be associated with images (like promotion levels and validation stamps) and some documents can be downloaded.

When uploading a document or an image, the DSL will accept any object (see below), optionally associated with a MIME content type (the content type is either read from the source object or defaults to `image/png`).

The object can be any of:

- a `URL` object - the MIME type and the binary content will be downloaded using the URL - the URL must be accessible anonymously
- a `File` object - the binary content is read from the file and the MIME type must be provided

- a valid URL string - same as an `URL` - see above
- a file path - same as a `File` - see above

For example:

```
ontrack.project('project') {
   branch('branch') {
      promotionLevel('COPPER', 'Copper promotion') {
         image '/path/to/local/file.png', 'image/png'
      }
   }
}
```

Document and image downloads return a `Document` object with has two properties:

- `content` - byte array
- `type` - MIME content type

For example, to store a promotion level's image into a file:

```
File file = ...
def promotionLevel = ontrack.promotionLevel('project', 'branch', 'COPPER')
file.bytes = promotionLevel.image.content
```

7.2.3. DSL Change logs

When a branch is configured for a SCM (Git, Subversion), a change log can be computed between two builds and following collections can be displayed:

- revisions or commits
- issues
- file changes

Change logs can also be computed between builds which belong to different branches, as long as they are in the same project. *This is only supported for Git, not for Subversion.*

Getting the change log

Given two builds, one gets access to the change log using:

```
def build1 = ontrack.build('proj', 'master', '1')
def build2 = ontrack.build('proj', 'master', '2')

def changelog = build1.getChangeLog(build2)
```

The returned change log might be `null` if the project and branches are not correctly configured.

On the returned `ChangeLog` object, one can access commits, issues and file changes.

Commits

The list of commits can be accessed using the `commits` property:

```
changeLog.commits.each {
    println "* ${it.shortId} ${it.message} (${it.author} at ${it.timestamp})"
}
```

Each item in the `commits` collection has the following properties:

- `id` - identifier, revision or commit hash
- `shortId` - short identifier, revision or abbreviated commit hash
- `author` - name of the committer
- `timestamp` - ISO date for the commit time
- `message` - raw message for the commit
- `formattedMessage` - HTML message with links to the issues
- `link` - link to the commit

This covers only the common attributes provided by Ontrack - additional properties are also available for a specific SCM.

Issues

The list of issues can be accessed using the `issues` property:

```
changeLog.issues.each {
    println "* ${it.displayKey} ${it.status} ${it.summary}"
}
```

Each item in the `issues` collection has the following properties:

- `key` - identifier, like `1`
- `displayKey` - display key (like `#1`)
- `summary` - short title for the issue
- `status` - status of the issue
- `url` - link to the issue

This covers only the common attributes provided by Ontrack - additional properties are also available for a specific issue service.

Exporting the change log

The change log can also be exported as text (HTML and Markdown are also available):

```
String text = changeLog.exportIssues(
        format: 'text',
        groups: [
                'Bugs'        : ['defect'],
                'Features'    : ['feature'],
                'Enhancements': ['enhancement'],
        ],
        exclude: ['design', 'delivery']
)
```

- `format` can be one of `text` (default), `html` or `markdown`
- `groups` allows to group issues per type. If not defined, no grouping is done
- `exclude` defines the types of issues to not include in the change log
- `altGroup` defaults to *Other* and is the name of the group where remaining issues do not fit.

File changes

The list of file changes can be accessed using the `files` property:

```
changeLog.files.each {
    println "* ${it.path} (${it.changeType})"
}
```

Each item in the `files` collection has the following properties:

- `path` - path changed
- `changeType` - nature of the change
- `changeTypes` - list of changes on this path

This covers only the common attributes provided by Ontrack - additional properties are also available for a specific SCM.

7.2.4. DSL Branch template definitions

Using the `template(Closure)` method on a branch, one can define the template definition for a branch.

For example:

```
template {
   parameter 'gitBranch', 'Name of the Git branch', 'release/${sourceName}'
   fixedSource '1.0', '1.1'
}
```

- `def parameter(String name, String description = '', String expression = '')` — defines a parameter for the template, with an optional expression based on a source name
- `def fixedSource(String… names)` — sets a synchronization source on the template, based on a fixed list of names

You can then use this branch definition in order to generate or update branches from it:

```
// Create a template
ontrack.branch('project', 'template') {
   template {
      parameter 'gitBranch', 'Name of the Git branch', 'release/${sourceName}'
   }
}
// Creates or updates the TEST instance
ontrack.branch('project', 'template').instance 'TEST', [
   gitBranch: 'my-branch'
]
```

7.2.5. DSL SCM extensions

If a SCM (Subversion or Git) is correctly configured on a branch, it is possible to download some files.

This is allowed only for the project owner.

For example, the following call:

```
def text = ontrack.branch('project', 'branch').download('folder/subfolder/path.txt')
```

will download the `folder/subfolder/path.txt` file from the corresponding SCM branch. A `OTNotFoundException` exception is thrown if the file cannot be found.

7.3. DSL Tool

Ontrack comes with an *Ontrack DSL Shell* tool that you can download from the releases page.

The `ontrack-dsl-shell.jar` is a fully executable JAR, published in GitHub release and in the Maven Central, and can be used to setup a running instance of Ontrack:

```
ontrack-dsl-shell.jar --url ... --user ... --password ... --file ...
```

You can display the full list options using `ontrack-dsl-shell.jar --help`.

The `--file` argument is the path to a file containing the Ontrack DSL to execute. If not set, or set to `-`, the DSL is taken from the standard input. For example:

```
cat project-list.groovy | ontrack-dsl-shell.jar --url https://ontrack.nemerosa.net
```

where `project-list.groovy` contains:

```
ontrack.projects*.name
```

This would return a JSON like:

```
[
    "iteach",
    "ontrack",
    "ontrack-jenkins",
    "versioning"
]
```

The tool always returns its response as JSON and its output can be pipelined with tools like `jq`. For example:

```
cat project-list.groovy | ontrack-dsl-shell.jar --url https://ontrack.nemerosa.net |
jq .
```

The JAR is a real executable, so there is no need to use `java -jar` on Unix like systems or MacOS.

7.4. DSL Reference

See the appendixes.

7.5. DSL Samples

Creating a build:

```
ontrack.branch('project', 'branch').build('1', 'Build 1')
```

Promoting a build:

```
ontrack.build('project', '1', '134').promote('COPPER')
```

Validating a build:

```
ontrack.build('project', '1', '134').validate('SMOKETEST', 'PASSED')
```

Getting the last promoted build:

```
def buildName = ontrack.branch('project', 'branch').lastPromotedBuilds[0].name
```

Getting the last build of a given promotion:

```
def branch = ontrack.branch('project', 'branch')
def builds = branch.standardFilter withPromotionLevel: 'BRONZE'
def buildName = builds[0].name
```

Configuring a whole branch:

```
ontrack.project('project') {
   branch('1.0') {
      promotionLevel 'COPPER', 'Copper promotion'
      promotionLevel 'BRONZE', 'Bronze promotion'
      validationStamp 'SMOKE', 'Smoke tests'
   }
}
```

Creating a branch template and an instance out of it:

```
// Branch template definition
ontrack.project(project) {
   config {
      gitHub 'ontrack'
   }
   branch('template') {
      promotionLevel 'COPPER', 'Copper promotion'
      promotionLevel 'BRONZE', 'Bronze promotion'
      validationStamp 'SMOKE', 'Smoke tests'
      // Git branch
      config {
         gitBranch '${gitBranch}'
      }
      // Template definition
      template {
         parameter 'gitBranch', 'Name of the Git branch'
      }
   }
}
// Creates a template instance
ontrack.branch(project, 'template').instance 'TEST', [
   gitBranch: 'feature/test'
]
```

Chapter 8. Contributing

Contributions to *Ontrack* are welcome!

1. Fork the GitHub project
2. Code your fixes and features
3. Create a pull request
4. Your pull requests, once tested successfully, will be integrated into the `master` branch, waiting for the next release

8.1. Development

8.1.1. Environment set-up

Following tools must be installed before you can start coding with Ontrack:

- JDK8u25 or better
- Docker 1.11 or more recent
- Docker Compose 1.6.2 or more recent

8.1.2. Building locally

```
./gradlew clean build
```

To launch the integration tests or acceptance tests, see Testing.

8.1.3. Launching the application from the command line

Just run:

```
./gradlew :ontrack-ui:bootRun
```

The application is then available at http://localhost:8080

8.1.4. Launching the application in the IDE

Prepare the Web resources by launching:

```
./gradlew dev
```

In order to launch the application, run the `net.nemerosa.ontrack.boot.Application` class with `--spring.profiles.active=dev` as argument.

The application is then available at http://localhost:8080

8.1.5. Developing for the web

If you develop on the web side, you can enable a LiveReload watch on the web resources:

```
./gradlew watch
```

Upon a change in the web resources, the browser page will be reloaded automatically.

8.1.6. Running the tests

See Testing.

8.1.7. Integration with IDE

With Intellij

- install the Lombok plugin
- in `Build, Execution, Deployment > Compiler > Annotation Processors`, check `Enable annotation processing`

8.1.8. Delivery

Official releases for Ontrack are available at:

- GitHub for the JAR, RPM and Debian packages
- Docker Hub for the Docker images

See the Installation documentation to know how to install them.

To create a package for delivery, just run:

```
./gradlew \
   clean \
   test \
   integrationTest \
   dockerLatest \
   build
```

This will create:

- a `ontrack-ui.jar`
- a `nemerosa/ontrack:latest` Docker image in your local registry

If you're not interested in having a Docker image, just omit the `dockerLatest` task.

Versioning

The version of the Ontrack project is computed automatically from the current SCM state, using the Gradle Versioning plug-in.

Deploying in production

See the Installation documentation.

8.1.9. Glossary

Form

Creation or update *links* can be accessed using the `GET` verb in order to get a form that allows the client to carry out the creation or update.

Such a form will give information about:

- the fields to be created/updated
- their format
- their validation rules
- their description
- their default or current values
- etc.

The GUI can use those forms in order to automatically (and optionally) display dialogs to the user. Since the model is responsible for the creation of those forms, this makes the GUI layer more resilient to the changes.

Link

In *resources*, links are attached to *model* objects, in order to implement a HATEOAS principle in the application interface.

HATEOAS does not rely exclusively on HTTP verbs since this would not allow a strong implementation of the actual use cases and possible navigations (which HATEOAS is all about).

For example, the "Project creation" link on the list of projects is *not* carried by the sole `POST` verb, but by a `_create` link. This link can be accessed through verbs:

- `OPTIONS` - list of allowed verbs
- `GET` - access to a form that allows to create the object
- `POST` or `PUT` for an update - actual creation (or update) of the object

Model

Representation of a concept in the application. This reflects the *ubiquitous language* used throughout the application, and is used in all layers. As POJO on server side, and JSON objects at

client side.

Repository

Model objects are persisted, retrieved and deleted through repository objects. Repositories act as a transparent persistence layer and hides the actual technology being used.

Resource

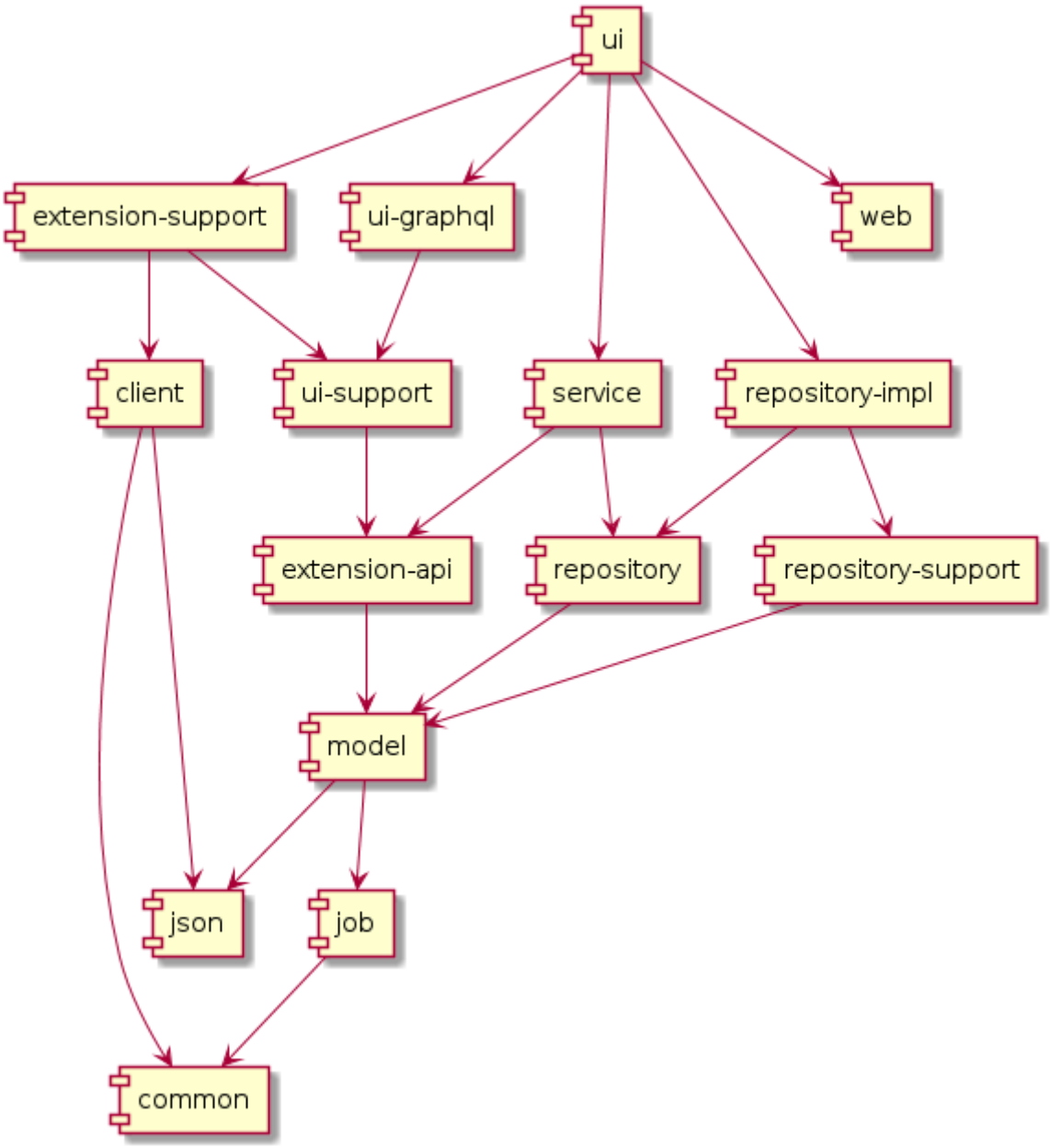
A resource is a model object decorated with links that allow the client side to interact with the API following the HATEOAS principle. More than just providing access to the model structure, those links reflect the actual use cases and the corresponding navigation. In particular, the links are driven by the authorizations (a "create" link not being there if the user is not authorized). See *Link* for more information.

Service

Services are used to provide interactions with the model.

8.2. Architecture

8.2.1. Modules



Not all modules nor links are shown here in order to keep some clarity. The Gradle build files in the source remain the main source of authority.

Modules are used in *ontrack* for two purposes:

- isolation
- distribution

We distinguish also between:

- core modules
- extension modules

Extension modules rely on the `extension-support` module to be compiled and tested. The link between the core modules and the extensions is done through the `extension-api` module, visible by the two worlds.

Modules like `common`, `json`, `tx` or `client` are purely utilitarian (actually, they could be extracted from `ontrack` itself).

The main core module is the `model` one, which defines both the API of the Ontrack services and the domain model.

8.2.2. UI

Resources

The UI is realized by REST controllers. They manipulate the *model* and get access to it through *services*.

In the end, the controllers return *model* objects that must be decorated by links in order to achieve Hateoas.

The controllers are not directly responsible for the decoration of the model objects as *resources* (model + links). This is instead the responsibility of the *resource decorators*.

The *model* objects are not returned as such, often their content needs to be filtered out. For example, when getting a list of branches for a project, we do not want each project to bring along its own copy of the project object. This is achieved using the *model filtering* technics.

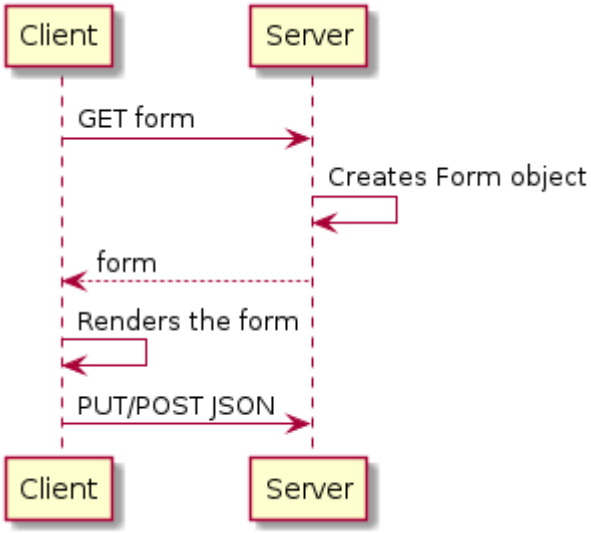
Resource decorators

TODO

8.2.3. Forms

Simple input forms do not need a lot of effort to design in Ontrack. They can be used directly in pages or in modal dialogs.

Server components (controllers, services, ...) are creating instances of the `Form` object and the client libraries (`service.form.js`) is responsible for their rendering:



Form object

The `Form` object is created by adding `Field`s into it using its `with` method:

```
import net.nemerosa.ontrack.model.form.Form;
public Form getMyForm() {
   return Form.create()
      .with(field1)
      .with(field2)
      ;
}
```

See the next section on how to create the field objects. The `Form` object contains utility methods for common fields:

```
Form.create()
   // `name` text field (40 chars max), with "Name" as a label
   // constrained by the `[A-Za-z0-9_\.\-]+` regular expression
   // Suitable for most name fields on the Ontrack model objects
   // (projects, branches, etc.)
   .name()
   // `password` password field (40 chars max) with "Password" as a label
   .password()
   // `description` memo field (500 chars max), optional, with "Description"
   // as a label
   .description()
   // `dateTime` date/time field (see below) with "Date/time" as a label
   .dateTime()
   // ...
   ;
```

In order to fill the fields with actual values, you can either use the `value(…)` method on the field object (see next section) or use the `fill(…)` method on the `Form` object.

```
Map<String, ?> data = ...
Form.create()
   // ...
   // Sets `value` as the value of the field with name "fieldName"
   .fill("fieldName", value)
   // Sets all values in the map using the key as the field name
   .fill(data)
```

Fields

Common field properties

Property	Method	Default value	Description

`name`	`of(…)`	*required*	Mapping
`label`	`label(…)`	*none*	Display name
`required`	`optional()`	`true`	Is the input required?
`readOnly`	`readOnly()`	`false`	Is the input read-only?
`validation`	`validation(…)`	*none*	Message to display is the field content is deemed invalid
`help`	`help(…)`	*none*	Help message to display for the field (see below for special syntax)
`visibleIf`	`visibleIf(…)`	*none*	Expression which defines if the field is displayed or not - see below for a detailed explanation
`value`	`value(…)`	*none*	Value to associated with the field

`help` property

TODO

`visibleIf` property

TODO

`text` field

The `text` field is a single line text entry field, mapped to the HTML `<input type="test"/>` form field.

Property	Method	Default value	Description
`length`	`length(…)`	`300`	Maximum length for the text
`regex`	`regex(…)`	`.*`	The text must comply with this regex in order to be valid

Example:

```
Form.create()
   .with(
      Text.of("name")
         .label("Name")
         .length(40)
         .regex("[A-Za-z0-9_\\.\\-]+")
         .validation("Name is required and must contain only alpha-numeric characters,
underscores, points or dashes.")
   )
```

`password` field

TODO

`memo` field

TODO

`email` field

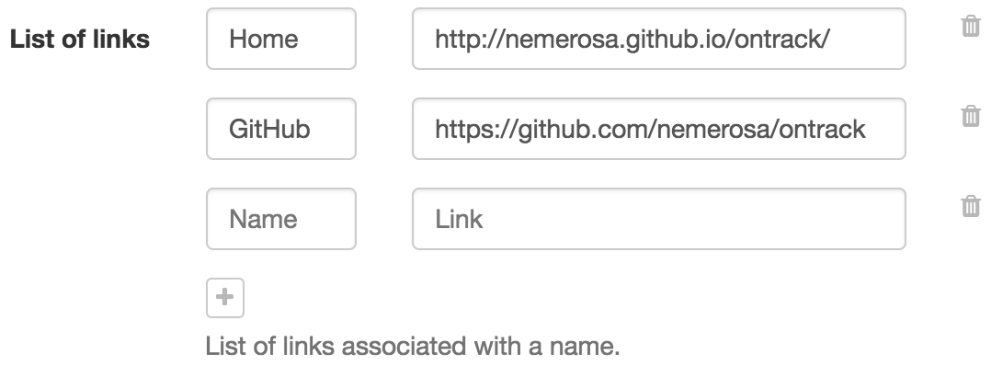
TODO

`url` field

TODO

`namedEntries` field

Multiple list of name/value fields:



The user can:

- add / remove entries in the list
- set a name and a value for each item
- the name might be optional - the value is not

Property	Method	Default value	Description
`nameLabel`	`nameLabel(…)`	"Name"	Label for the "name" input part of an entry.
`valueLabel`	`valueLabel(…)`	"Value"	Label for the "value" input part of an entry.
`nameRequired`	`nameOptional()`	`true`	If the name part is required.
`addText`	`addText(…)`	"Add an entry"	Label for the "add" button.

Example:

```
Form.create()
   .with(
      NamedEntries.of("links")
         .label("List of links")
         .nameLabel("Name")
         .valueLabel("Link")
         .nameOptional()
         .addText("Add a link")
         .help("List of links associated with a name.")
         .value(value != null ? value.getLinks() : Collections.emptyList())
   )
```

`date` field

TODO

`yesno` field

TODO

`dateTime` field

TODO

`int` field

TODO

`selection` field

TODO

`multi-strings` field

TODO

`multi-selection` field

TODO

`multi-form` field

TODO

Creating your custom fields

TODO

Form usage on the client

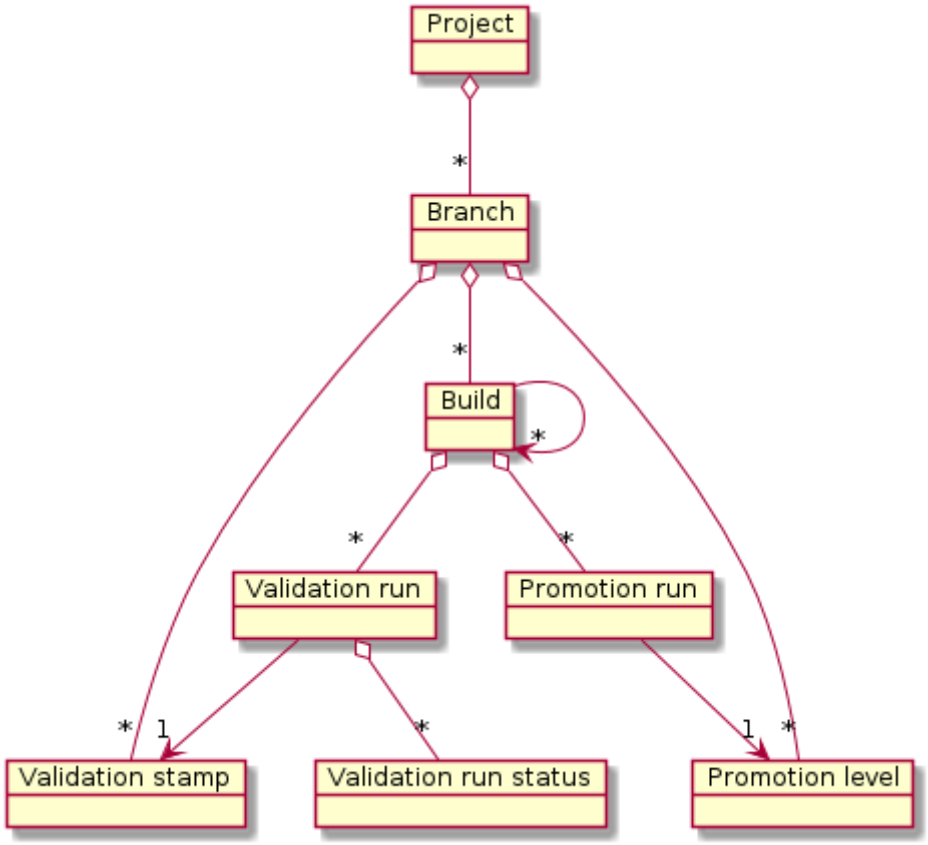
TODO

Fields rendering

TODO

8.2.4. Model

The root entity in Ontrack is the *project*.



Several *branches* can be attached to a *project*. *Builds* can be created within a branch and linked to other builds (same or other branches).

Promotion levels and *validation stamps* are attached to a *branch*:

a *promotion level* is used to define the *promotion* a given *build* has reached. A *promotion run* defines this association.

- a *validation stamp* is used to qualify some tests or other validations on a *build*. A *validation run* defines this association. There can be several runs per build and per validation stamp. A run itself has a sequence of statuses attached to it: passed, failed, investigated, etc.

Branches, *promotion levels* and *validation stamps* define the *static structure* of a *project*.

8.2.5. Model filtering

TODO

8.2.6. Jobs

Ontrack makes a heavy use of *jobs* in order to schedule regular activities, like:

- SCM indexation (for SVN for example)
- SCM/build synchronisations
- Branch templating synchronisation
- etc.

Services and extensions are responsible for providing Ontrack with the list of *jobs* they want to be executed. They do this by implementing the `JobProvider` interface that simply returns a collection of `JobRegistration`s to register at startup.

One component can also register a `JobOrchestratorSupplier`, which provides also a stream of `JobRegistration`s, but is more dynamic since the list of jobs to register will be determined regularly.

The *job scheduler* is in charge to collect all *registered jobs* and to run them all.

Job architecture overview

This section explains the underlying concepts behind running the jobs in Ontrack.

When a job is registered, it is associated with a schedule. This schedule is dynamic and can be changed with the time. For example, the indexation of a Git repository for a project is scheduled every 30 minutes, but then, is changed to 60 minutes. The job registration schedule is then changed to every 60 minutes.

A job provides the following key elements:

- a unique identifier: the *job key*
- a task to run, provided as a `JobRun` interface:

-

```
@FunctionalInterface
public interface JobRun {
    void run(JobRunListener runListener);
}
```

The task defined by the job can use the provided `JobRunListener` to provide feedback on the execution or to execution messages.

The job task is wrapped into a `Runnable` which is responsible to collect statistics about the job execution, like number of runs, durations, etc.

In the end, the `JobScheduler` can be associated with a `JobDecorator` to return another `Runnable` layer if needed.

The job scheduler is responsible to orchestrate the jobs. The list of jobs is maintained in memory using an index associating the job itself, its schedule and its current scheduled task (as a `ScheduledFuture`).

Job registration

A `JobRegistration` is the associated of a `Job` and of `Schedule` (run frequency for the job).

A `Schedule` can be built in several ways:

```
// Registration only, no schedule
Schedule.NONE
// Every 15 minutes, starting now
Schedule.everyMinutes(15)
// Every minute, starting now
Schedule.EVERY_MINUTE
// Every day, starting now
Schedule.EVERY_DAY
// Every 15 minutes, starting after 5 minutes
Schedule.everyMinutes(15).after(5)
```

see the `Schedule` class for more options.

By enabling the scattering options, one can control the schedule by adding a startup delay at the beginning of the job.

The `Job` interface must define the unique for the job. A key in unique within a type within a category.

Typically, the category and the type will be fixed (constants) while the key will depend on the job parameters and context. For example:

```
JobCategory CATEGORY = JobCategory.of("category").withName("My category");
JobType TYPE = CATEGORT.getType("type").withName("My type");
public JobKey getKey() {
   return TYPE.getKey("my-id")
}
```

The `Job` provides also a description, and the desired state of the job:

- disabled or not - might depend on the job parameters and context. For example, the job which synchronizes a branch instance with its template will be disable if the branch is disabled
- valid or not - when a job becomes invalid, it is not executed, and will be unregistered automatically. For example, a Subversion indexation job might become invalid if the associated repository configuration has been deleted.

Finally, of course, the job must provide the task to actually execute:

```
public JobRun getTask() {
   return (JobRunListener listener) -> ...
}
```

The task takes as parameter a `JobRunListener`.

All job tasks run with *administrator* privileges. *Job tasks* can throw runtime exceptions - they will be caught by the *job scheduler* and displayed in the administration console.

8.2.7. Build filters

The *build filters* are responsible for the filtering of *builds* when listing them for a *branch*.

Usage

By default, only the last 10 builds are shown for a branch, but a user can choose to create filters for a branch, and to select them.

The filters he creates are saved for later use: * locally, in its local browser storage * remotely, on the server, if he is connected

For a given branch, a filter is identified by a name. The list of available filters for a branch is composed of those stored locally and of those returned by the server. The later ones have priority when there is a name conflict.

Implementation

The `BuildFilter` interface defines how to use a filter. This filter takes as parameters:

- the current list of filtered builds

- the branch
- the build to filter

It returns two boolean results:

- is the build to be kept in the list?
- do we need to go on with looking for other builds?

The `BuildFilterService` is used to manage the build filters:

- by creating `BuildFilter` instances
- by managing `BuildFilterResource` instances

The service determines the type of `BuildFilter` by using its type, and uses injected `BuildFilterProvider`s to get an actual instance.

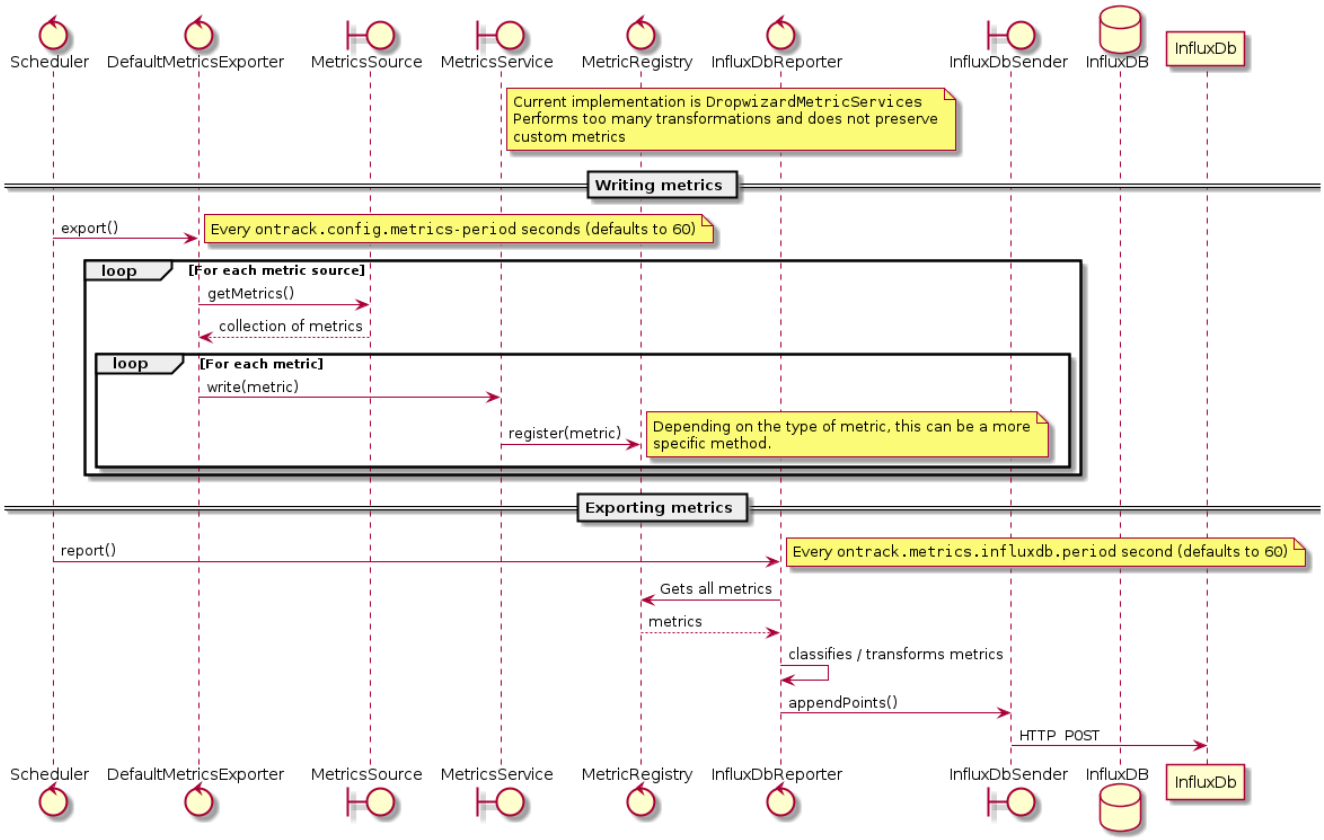
8.2.8. Reference services

This is a work in progress and this list is not exhaustive yet. In the meantime, the best source of information remains the source code...

Service	Description
`StructureService`	Access to projects, branches and all entities
`SecurityService`	Checks the current context for authorizations
`PropertyService`	Access to the properties of the entities
`EntityDataService`	This service allows to store and retrieve arbitrary data with entities

8.2.9. Metrics

Ontrack metrics are collected and exposed using the Spring Boot framework.



8.2.10. Technology

Client side

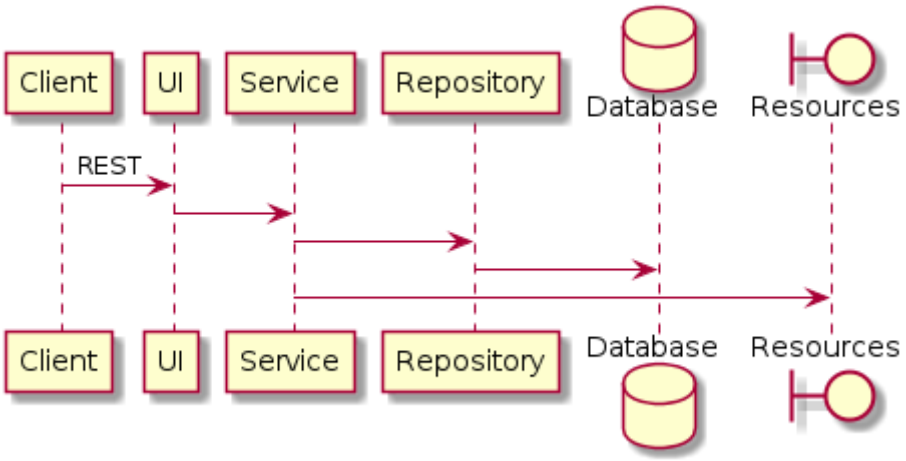
One page only, pure AJAX communication between the client and the server.

- AngularJS
- Angular UI Router
- Angular UI Bootstrap
- Bootstrap
- Less

Server side

- Spring Boot for the packaging & deployment
- Spring MVC for the REST API
- Spring for the IoC
- Spring Security & AOP for the security layer
- Plain JDBC for the data storage
- H2 in MySQL mode for the database engine

Layers



8.3. Testing

- *unit tests* are always run and should not access resources not load the application context (think: fast!)
- *integration tests* can access resources or load the application context, and run against a database
- *acceptance tests* are run against the deployed and running application.

8.3.1. Running the unit and integration tests

In order to run the *unit tests* only:

```
./gradlew test
```

In order to add the *integration tests*:

```
./gradlew integrationTest
```

From your IDE, you can launch both unit and integration tests using the default JUnit integration.

8.3.2. Acceptance tests

On the command line

This requires Docker & Docker Compose to be installed and correctly configured.

The application can be deployed on a local Docker container, running on SSL with a self signed certificate, and tested by running all the acceptance tests:

```
./gradlew ciAcceptanceTest
```

When using a Docker Machine, you will have to specify the Docker host:

```
./gradlew ciAcceptanceTest -PciMachine=`docker-machine ip build`
```

where `build` is the name of the Docker machine.

If the Docker container used for the tests must be kept, add the `-x ciStop` to the arguments.

To only deploy the application in a container without launching any test, you can also run `./gradlew ciStart`.

From the IDE

In order to develop or test acceptance tests, you might want to run them from your IDE.

1. Make sure you have a running application somewhere, either by launching it from your IDE (see Integration with IDE) or by running `ciStart` (see previous section).
2. Launch all, some or one test in the `ontrack-acceptance` module after having set the following system properties:
   - `ontrack.url` - the URL of the running application to test
   - `ontrack.disableSSL` - `true` if the server is running with a self signed certificate, and if you're using `https`

Standalone mode

For testing `ontrack` in real mode, with the application to test deployed on a remote machine, it is needed to be able to run acceptance tests in standalone mode, without having to check the code out and to build it.

Running the acceptance tests using the `ciAcceptanceTest` Gradle task remains the easiest way.

The acceptance tests are packaged as a standalone JAR, that contains all the dependencies.

To run the acceptance tests, you need a JDK8 and you have to run the JAR using:

```
java -jar ontrack-acceptance.jar <options>
```

The options are:

- `--option.url=<url>` to specify the `<url>` where `ontrack` is deployed. It defaults to http://localhost:8080
- `--option.admin=<password>` to specify the administrator password. It defaults to `admin`.
- `--option.context=<context>` can be specified several time to define the context(s) the acceptance tests are running in (like `--option.context=production` for example). According to the context, some tests can be excluded from the run.

The results of the tests is written as a JUnit XML file, `ciAcceptance.xml`.

8.3.3. Developing tests

Unit tests are JUnit tests whose class name ends with `Test`. Integration tests are JUnit tests whose class name ends with `IT`. Acceptance tests are JUnit tests whose class name starts with `ACC` and are located in the `ontrack-acceptance` module.

Unit test context

Unit tests do not need any application context or any database.

Integration test context

Integration tests will usually load an application context and connect to a Postgresql database.

For commodity, those tests will inherit from the `AbstractITTestSupport` class, and more specifically:

- from `AbstractRepositoryTestSupport` for JDBC repository integration tests
- from `AbstractServiceTestSupport` for service integration tests

Configuration for the integration tests is done in the `ITConfig` class.

8.4. Extending Ontrack

Ontrack allows extensions to contribute to the application, and actually, most of the core features, like Git change logs, are indeed extensions.

This page explains how to create your own extensions and to deploy them along Ontrack. The same coding principles apply also for coding core extensions and to package them in the main application.

Having the possibility to have external extensions in Ontrack is very new and the way to provide them is likely to change (a bit) in the next versions. In particular, the extension mechanism does not provide classpath isolation between the "plugins".

8.4.1. Preparing an extension

In order to create an extension, you have to create a Java project.

The use of Groovy is also possible.

Note that Ontrack needs at least a JDK8u65 to run.

Your extension needs to a Gradle project and have at least this minimal `build.gradle` file:

Maven might be supported in the future.

```
buildscript {
   repositories {
      mavenCentral()
      jcenter()
   }
   dependencies {
      classpath 'net.nemerosa.ontrack:ontrack-extension-plugin:{{ontrack-version}}'
   }
}
repositories {
   mavenCentral()
}
apply plugin: 'ontrack'
```

The `buildscript` section declares the version of Ontrack you're building your extension for. Both the `mavenCentral` and the `jcenter` repositories are needed to resolve the path for the `ontrack-extension-plugin` since the plugin is itself published in the Maven Central and some of its dependencies are in JCenter.

The repository declaration might be simplified in later versions.

The plug-in must declare the Maven Central as repository for the dependencies (Ontrack libraries are published in the Maven Central).

Finally, you can apply the `ontrack` plug-in. This one will:

- apply the `java` plug-in. If you want to use Groovy, you'll have to apply this plug-in yourself.
- add the `ontrack-extension-support` module to the `compile` configuration of your extension
- define some tasks used for running, testing and packaging your extension (see later)

8.4.2. Extension ID

Your extension must be associated with an identifier, which will be used throughout all the extension mechanism of Ontrack.

If the `name` of your extension project looks like `ontrack-extension-<xxx>`, the `xxx` will be ID of your extension. For example, in the `settings.gradle` file:

```
rootProject.name = 'ontrack-extension-myextension'
```

then `myextension` is your extension ID.

If for any reason, you do not want to use `ontrack-extension-` as a prefix for your extension name, you must specify it using the `ontrack` Gradle extension in the `build.gradle` file:

```
ontrack {
   id 'myextension'
}
```

8.4.3. Coding an extension

All your code must belong to a package starting with `net.nemerosa.ontrack` in order to be visible by the Ontrack application.

Typically, this should be like: `net.nemerosa.ontrack.extension.<id>` where `id` is the ID of your extension.

This limitation about the package name is likely to removed in future versions of Ontrack.

You now must declare your extension to Ontrack by creating an *extension feature* class:

```
package net.nemerosa.ontrack.extension.myextension;

import net.nemerosa.ontrack.extension.support.AbstractExtensionFeature;
import net.nemerosa.ontrack.model.extension.ExtensionFeatureOptions;
import org.springframework.stereotype.Component;

@Component
public class MyExtensionFeature extends AbstractExtensionFeature {
    public MyExtensionFeature() {
        super(
            "myextension",
            "My extension",
            "Sample extension for Ontrack",
            ExtensionFeatureOptions.DEFAULT
        );
    }
}
```

The `@Component` annotation makes this extension feature visible by Ontrack.

The arguments for the extension feature constructor are:

- the extension ID
- the display name
- a short description
- the extension options (see below)

8.4.4. Extension options

If your extension has some web components (templates, pages, etc.), it must declare this fact:

```
ExtensionFeatureOptions.DEFAULT.withGui(true)
```

If your extension depends on other extensions, it must declare them. For example, to depend on GitHub and Subversion extensions, first declare them as dependencies in the `build.gradle`:

```
ontrack {
   uses 'github'
   uses 'svn'
}
```

then, in your code:

```
@Component
public class MyExtensionFeature extends AbstractExtensionFeature {
    @Autowired
    public MyExtensionFeature(
      GitHubExtensionFeature gitHubExtensionFeature,
      SVNExtensionFeature svnExtensionFeature
    ) {
        super(
            "myextension",
            "My extension",
            "Sample extension for Ontrack",
            ExtensionFeatureOptions.DEFAULT
               .withDependency(gitHubExtensionFeature)
               .withDependency(svnExtensionFeature)
        );
    }
}
```

8.4.5. Writing tests for your extension

Additionally to creating unit tests for your extension, you can also write integration tests, which will run with the Ontrack runtime enabled.

This feature is only available starting from version 2.23.1.

When the `ontrack-extension-plugin` is applied to your code, it makes the `ontrack-it-utils` module available for the compilation of your tests.

In particular, this allows you to create integration tests which inherit from `AbstractServiceTestSupport`, to inject directly the Ontrack services you need and to use utility methods to create a test environment.

For example:

```
public MyTest extends AbstractServiceTestSupport {
   @Autowired
   private StructureService structureService
   @Test
   public void sample_test() {
      // Creates a project
      Project p = doCreateProject();
      // Can retrieve it by name...
      asUser().withView(p).execute(() ->
         assertTrue(structureService.findProjectByName(p.getName()).isPresent())
      );
   }
}
```

8.4.6. List of extension points

Ontrack provides the following extension points:

- Properties - allows to attach a property to an [entity](Model)
- Decorators - allows to display a decoration for an [entity](Model)
- User menu action - allows to add an entry in the connected user menu
- Settings - allows to add an entry in the global settings
- Metrics - allows to contribute to the metrics exported by Ontrack
- Event types - allows to define additional event types.
- GraphQL - allows contributions to the GraphQL Ontrack schema.
- **TODO** Entity action - allows to add an action for the page of an entity
- **TODO** Entity information - allows to add some information into the page of an entity
- **TODO** Search extension - provides a search end point for global text based searches
- **TODO** Issue service - provides support for a ticketing system
- **TODO** SCM service - provides support for a SCM system
- **TODO** Account management action - allows to add an action into the account management

Other topics:

- Creating pages
- **TODO** Creating services
- **TODO** Creating jobs

See Reference services for a list of the core Ontrack services.

8.4.7. Running an extension

Using Gradle

To run your extension using Gradle:

```
./gradlew ontrackRun
```

This does not work yet for Ontrack 3.0 (with Postgresql)

This will make the application available at http://localhost:8080

The `ontrackRun` Gradle task can be run directly from Intellij IDEA and if necessary in debug mode.

When running with Gradle in your IDE, if you edit some web resources and want your changes available in the browser, just rebuild your project (`Ctrl F9` in Intellij) and refresh your browser.

Using Docker

In order to deploy your extension in an Ontrack Docker container, start by building:

```
./gradlew clean build
```

then run an Ontrack container, configured to fetch its extension directory from your extension build directory:

```
docker run --detach \
   --volume `pwd`/build/libs:/var/ontrack/extensions \
   --publish 8080:8080 \
   nemerosa/ontrack:{{ontrack-version}}
```

See the documentation for the Docker options.

8.4.8. Packaging an extension

Just run:

```
./gradlew clean build
```

The extension is available as a JAR in `build/libs`.

8.4.9. Deploying an extension

Using the Docker image

The Ontrack Docker image uses the `/var/ontrack/extensions` volume to load extensions from. Bind this volume to your host or to a data container to start putting extensions in it. For example:

```
docker run --volume /extension/on/host:/var/ontrack/extensions ...
```

You can also create your own image. Create the following `Dockerfile`:

Dockerfile

```
# Base Ontrack image
FROM nemerosa/ontrack:<yourversion>
# Copies the extensions in the target volume
COPY extensions/*.jar /var/ontrack/extensions/
```

We assume here that your extensions are packaged in an `extensions` folder at the same level than your `Dockerfile`:

```
/-- Dockerfile
|-- extensions/
     |-- extension1.jar
     |-- extension2.jar
```

Using the CentOS or Debian/Ubuntu package

The RPM and Debian packages both use the `/usr/lib/ontrack/extensions` directory for the location of the extensions JAR files.

You can also create a RPM or Debian package which embeds both Ontrack and your extensions.

The means to achieve this depend on your build technology but the idea is the same in all cases:

Your package must:

- put the extension JAR files in `/usr/lib/ontrack/extensions`
- have a dependency on the `ontrack` package

In standalone mode

When running Ontrack directly, you have to set the `loader.path` to a directory containing the extensions JAR files:

```
java -Dloader.path=/path/to/extensions -jar ... <options>
```

8.4.10. Extending properties

Any entity in Ontrack can be associated with a set of properties. Extensions can contribute to create new ones.

A *property* is the association some Java components and a HTML template to render it on the screen.

Java components

First, a property must be associated with some data. Just create an invariant POJO like, for example:

```
package net.nemerosa.ontrack.extension.myextension;

import lombok.Data;

@Data
public class MyProperty {
    private final String value;
}
```

Note that Ontrack extensions can take benefit of using Lombok in order to reduce the typing. But this is not mandatory as all.

Then, you create the *property type* itself, by implementing the `PropertyType` interface or more easily by extending the `AbstractPropertyType` class. The parameter for this class is the data created above:

```
@Component
public class MyPropertyType extends AbstractPropertyType<MyProperty> {
}
```

The `@Component` notation registers the property type in Ontrack.

A property, or any extension is always associated with an extension feature and this one is typically injected:

```
@Autowired
public MyPropertyType(MyExtensionFeature extensionFeature) {
  super(extensionFeature);
}
```

Now, several methods need to be implemented:

- `getName` and `getDescription` return respectively a display name and a short description for the property
- `getSupportedEntityTypes` returns the set of entities the property can be applied to. For example,

if your property can be applied only on projects, you can return:

```
@Override
public Set<ProjectEntityType> getSupportedEntityTypes() {
  return EnumSet.of(ProjectEntityType.PROJECT);
}
```

- `canEdit` allows you to control who can create or edit the property for an entity. The `SecurityService` allows you to test the authorizations for the current user. For example, in this sample, we authorize the edition of our property only for users being granted to the project configuration:

```
@Override
public boolean canEdit(ProjectEntity entity, SecurityService securityService) {
  return securityService.isProjectFunctionGranted(entity, ProjectConfig.class);
}
```

- `canView` allows you to control who can view the property for an entity. Like for `canEdit`, the `SecurityService` is passed along, but you will typically return `true`:

```
@Override
public boolean canView(ProjectEntity entity, SecurityService securityService) {
  return true;
}
```

- `getEditionForm` returns the form being used to create or edit the property. Ontrack uses `Form` objects to generate automatically user forms on the client. See its Javadoc for more details. In our example, we only need a text box:

```
@Override
public Form getEditionForm(ProjectEntity entity, MyProperty value) {
  return Form.create()
          .with(
                  Text.of("value")
                          .label("My value")
                          .length(20)
                          .value(value !==== null ? value.getValue() : null)
          );
}
```

- the `fromClient` and `fromStorage` methods are used to parse back and forth the JSON into a property value. Typically:

```
@Override
public MyProperty fromClient(JsonNode node) {
  return fromStorage(node);
}

@Override
public MyProperty fromStorage(JsonNode node) {
  return parse(node, ProjectCategoryProperty.class);
}
```

- the `getSearchKey` is used to provide an indexed search value for the property:

```
@Override
public String getSearchKey(My value) {
  return value.getValue();
}
```

- finally, the `replaceValue` method is called when the property has to be cloned for another entity, using a replacement function for the text values:

```
@Override
public MyProperty replaceValue(MyProperty value, Function<String, String>
replacementFunction) {
  return replacementFunction.apply(value);
}
```

Web components

A HTML fragment (or template) must be created at:

```
src/main/resources
  \-- static
     \-- extension
         \-- myextension
            \-- property
               \-- net.nemerosa.ontrack.extension.myextension.MyPropertyType.tpl.html
```

Replace `myextension`, the package name and the property type accordingly of course.

The `tpl.html` will be used as a template on the client side and will have access to the `Property` object. Typically, only its `value` field, of the property data type, will be used.

The template is used the AngularJS template mechanism.

For example, to display the property as bold text in our sample:

```
<b>{{property.value.value}}</b>
```

The property must be associated with an icon, typically PNG, 24 x 24, located at:

```
src/main/resources
  \-- static
     \-- extension
         \-- myextension
            \-- property
               \-- net.nemerosa.ontrack.extension.myextension.MyPropertyType.png
```

8.4.11. Extending decorators

A decorator is responsible to display a decoration (icon, text, label, etc.) close to an entity name, in the entity page itself or in a list of those entities. Extensions can contribute to create new ones.

A *decorator* is the association some Java components and a HTML template to render it on the screen.

Java components

First, a decorator must be associated with some data. You can use any type, like a `String`, an `enum` or any other invariant POJO. In our sample, we'll take a `String`, which is the value of the `MyProperty` property described as example in Extending properties.

Then, you create the *decorator* itself, by implementing the `DecorationExtension` interface and extending the `AbstractExtension`. The parameter type is the decorator data defined above.

```
@Component
public class MyDecorator extends AbstractExtension implements
DecorationExtension<String> {
}
```

The `@Component` notation registers the decorator in Ontrack.

A decorator, or any extension is always associated with an extension feature and this one is typically injected. Other services can be injected at the same time. For example, our sample decorator needs to get a property on an entity so we inject the `PropertyService`:

```
private final PropertyService propertyService;
@Autowired
public MyDecorator(MyExtensionFeature extensionFeature, PropertyService
propertyService) {
   super(extensionFeature);
   this.propertyService = propertyService;
}
```

Now, several methods need to be implemented:

- `getScope` returns the set of entities the decorator can be applied to. For example, if your property can be applied only on projects, you can return:

```
@Override
public EnumSet<ProjectEntityType> getScope() {
   return EnumSet.of(ProjectEntityType.PROJECT);
}
```

- `getDecorations` returns the list of decorations for an entity. In our case, we want to return a decoration only if the project is associated with the `MyProperty` property and return its value as decoration data.

```
@Override
public List<Decoration<String>> getDecorations(ProjectEntity entity) {
  return propertyService.getProperty(entity, MyPropertyType.class).option()
          .map(p -> Collections.singletonList(
                  Decoration.of(
                          MyDecorator.this,
                          p.getValue()
                  )
          ))
          .orElse(Collections.emptyList());
}
```

Web components

A HTML fragment (or template) must be created at:

```
src/main/resources
  \-- static
     \-- extension
         \-- myextension
            \-- decoration
               \-- net.nemerosa.ontrack.extension.myextension.MyDecorator.tpl.html
```

Replace `myextension`, the package name and the decorator type accordingly of course.

The `tpl.html` will be used as a template on the client side and will have access to the `Decoration` object. Typically, only its `data` field, of the decoration data type, will be used.

The template is used the AngularJS template mechanism.

For example, to display the decoration data as bold text in our sample:

```
<!-- In this sample, `data` is a string -->
<b>{{decoration.data}}</b>
```

8.4.12. Extending the user menu

An extension can add a entry in the connected user menu, in order to point to an extension page.

Extension component

Define a component which extends `AbstractExtension` and implements `UserMenuExtension`:

```
package net.nemerosa.ontrack.extension.myextension;

@Component
public class MyUserMenuExtension extends AbstractExtension implements
UserMenuExtension {

    @Autowired
    public MyUserMenuExtension(MyExtensionFeature extensionFeature) {
        super(extensionFeature);
    }

    @Override
    public Action getAction() {
        return Action.of("my-user-menu", "My User Menu", "my-user-menu-page");
    }

    @Override
    public Class<? extends GlobalFunction> getGlobalFunction() {
        return ProjectList.class;
    }
}
```

In this sample, `my-user-menu-page` is the relative routing path to the page the user action must point to.

The `getGlobalFunction` method returns the function needed for authorizing the user menu to appear.

8.4.13. Extending pages

Extensions can also contribute to pages.

Extension menus

Extension pages must be accessible from a location:

- the global user menu

- an entity page

From the global user menu

TODO

From an entity page

In order for an extension to contribute to the menu of an entity page, you have to implement the `ProjectEntityActionExtension` interface and extend the `AbstractExtension`.

```
@Component
public class MyProjectActionExtension extends AbstractExtension implements
ProjectEntityActionExtension {
}
```

The `@Component` notation registers the extension in Ontrack.

An action extension, or any [extension](Extensions) is always associated with an extension feature and this one is typically injected. Other services can be injected at the same time. For example, our sample extension needs to get a property on an entity so we inject the `PropertyService`:

```
private final PropertyService propertyService;
@Autowired
public MyProjectActionExtension(MyExtensionFeature extensionFeature, PropertyService
propertyService) {
   super(extensionFeature);
   this.propertyService ==== propertyService;
}
```

The `getAction` method returns an optional `Action` for the entity. In our sample, we want to associate an action with entity if it is a project and if it has the `MyProperty` property being set:

```
@Override
public Optional<Action> getAction(ProjectEntity entity) {
   if (entity instanceof Project) {
       return propertyService.getProperty(entity, MyPropertyType.class).option()
               .map(p ->
                       Action.of(
                               "my-action",
                               "My action",
                               String.format("my-action/%d", entity.id())
                       )
               );
   } else {
       return Optional.empty();
   }
}
```

The returned `Action` object has the following properties:

- an `id`, uniquely identifying the target page in the extension
- a `name`, which will be used as display name for the menu entry
- a URI fragment, which will be used for getting to the extension end point (see later). Note that this URI fragment will be prepended by the extension path. So in our example, the final path for the `SAMPLE` project with id `12` would be: `extension/myextension/my-action/12`.

Extension global settings

TODO

Extension page

Before an extension can serve some web components, it must be declared as being GUI related. See the documentation to enable this (`ExtensionFeatureOptions.DEFAULT.withGui(true)`).

The extension must define an AngularJS module file at:

```
src/main/resources
  \-- static
     \-- extension
         \-- myextension
            \-- module.js
```

The `module.js` file name is fixed and is used by Ontrack to load the web components of your extension at startup.

This is an AngularJS (1.2.x) module file and can declare its configuration, its services, its controllers, etc. Ontrack uses `UI Router`(), version `0.2.11` for the routing of the pages, allowing a routing

declaration as module level.

For our example, we want to declare a page for displaying information for `extension/myextension/my-action/{project}` where `{project}` is the ID of one project:

```
angular.module('ontrack.extension.myextension', [
        'ot.service.core',
        'ot.service.structure'
    ])
    // Routing
    .config(function ($stateProvider) {
        $stateProvider.state('my-action', {
            url: '/extension/myextension/my-action/{project}',
            templateUrl: 'extension/myextension/my-action.tpl.html',
            controller: 'MyExtensionMyActionCtrl'
        });
    })
    // Controller
    .controller('MyExtensionMyActionCtrl', function ($scope, $stateParams, ot,
otStructureService) {
        var projectId ==== $stateParams.project;

        // View definition
        var view ==== ot.view();
        view.commands ==== [
            // Closing to the project
            ot.viewCloseCommand('/project/' + projectId)
        ];

        // Loads the project
        otStructureService.getProject(projectId).then(function (project) {
            // Breadcrumbs
            view.breadcrumbs ==== ot.projectBreadcrumbs(project);
            // Title
            view.title ==== "Project action for " + project.name;
            // Scope
            $scope.project ==== project;
        });
    })
;
```

The routing configuration declares that the end point at `/extension/myextension/my-action/{project}` will use the `extension/myextension/my-action.tpl.html` view and the `MyExtensionMyActionCtrl` controller defined below.

The `ot` and `otStructureService` are Ontrack Angular services, defined respectively by the `ot.service.core` and `ot.service.structure` modules.

The `MyExtensionMyActionCtrl` controller:

- gets the project ID from the state (URI) definition
- it defines an Ontrack view, and defines a close command to go back to the project page
- it then loads the project using the `otStructureService` service and upon loading completes some information into the view

Finally, we define a template at:

```
src/main/resources
  \-- static
     \-- extension
         \-- myextension
            \-- extension/myextension/my-action.tpl.html
```

which contains:

```
<ot-view>
    Action page for {{project.name}}.
</ot-view>
```

The `ot-view` is an Ontrack directive which does all the layout magic for you. You just have to provide the content.

Ontrack is using Bootstrap 3.x for the layout and basic styling, so you can start structuring your HTML with columns, rows, tables, etc. For example:

```
<ot-view>
   <div class="row">
      <div class="col-md-12">
         Action page for {{project.name}}.
      </div>
   </div>
</ot-view>
```

Extension API

TODO

Extension API resource decorators

TODO

8.4.14. Extending event types

Extensions can define additional event types which can then be used to add custom events to entities.

To register a custom event type:

```
@Autowired
public static final EventType CUSTOM_TYPE = SimpleEventType.of("custom-type", "My
custom event");
public MyExtension(..., EventFactory eventFactory) {
   super(extensionFeature);
   eventFactory.register(CUSTOM_TYPE);
}
```

Then, you can use it this way when you want to attach an event to, let's say, a build:

```
EventPostService eventPostService;
Build build;
...
eventPostService.post(
    Event.of(MyExtension.CUSTOM_TYPE).withBuild(build).get()
);
```

8.4.15. Extending GraphQL

Extensions can contribute to the Ontrack GraphQL core schema:

- custom types
- root queries
- additional fields in project entities

Preparing the extension

In your extension module, import the `ontrack-ui-graphql` module:

```
dependencies {
   compile "net.nemerosa.ontrack:ontrack-ui-graphql:${ontrackVersion}"
}
```

If you want to write integration tests for your GraphQL extension, you have to include the GraphQL testing utilities:

```
dependencies {
   compile "net.nemerosa.ontrack:ontrack-ui-graphql:${ontrackVersion}:tests"
}
```

Custom types

To define an extra type, you create a component which implements the `GQLType` interface:

```
@Component
public class PersonType implements GQLType {
    @Override
    public GraphQLObjectType getType() {
        return GraphQLObjectType.newObject()
                .name("Person")
                .field(f -> f.name("name")
                        .description("Name of the person")
                        .type(GraphQLString)
                )
                .build();
    }
}
```

See the graphql-java documentation for the description of the type construction.

You can use this component in other ones, like in queries, field definitions or other types, like shown below:

```
@Component
public class AccountType implements GQLType {

    private final PersonType personType;

    @Autowired
    public AccountType (PersonType personType) {
        this.personType = personType;
    }

    @Override
    public GraphQLObjectType getType() {
        return GraphQLObjectType.newObject()
                .name("Account")
                .field(f -> f.name("username")
                        .description("Account name")
                        .type(GraphQLString)
                )
                .field(f -> f.name("identity")
                        .description("Identity")
                        .type(personType.getType())
                )
                .build();
    }
}
```

You can also create GraphQL types dynamically by using introspection of your model classes.

Given the following model:

```
@Data
public class Person {
   private final String name;
}
@Data
public class Account {
   private final String username;
   private final Person identity;
}
```

You can generate the `Account` type by using:

```
@Override
public GraphQLObjectType getType() {
   return GraphQLBeanConverter.asObjectType(Account.class);
}
```

The `GraphQLBeanConverter.asObjectType` is still very **experimental** and its implementation is likely to change in the next versions of Ontrack.

Root queries

Your extension can contribute to the root query by creating a component implementing the `GQLRootQuery` interface:

```
@Component
public class AccountGraphQLRootQuery implements GQLRootQuery {

    private final AccountType accountType;

    @Autowired
    public AccountGraphQLRootQuery(AccountType accountType) {
        this.accountType = accountType;
    }

    @Override
    public GraphQLFieldDefinition getFieldDefinition() {
        return GraphQLFieldDefinition.newFieldDefinition()
                .name("accounts")
                .argument(a -> a.name("username")
                        .description("User name pattern")
                        .type(GraphQLString)
                )
                .type(accountType.getType())
                .dataFetcher(...)
                .build();
    }
}
```

This root query can then be used into your GraphQL queries:

```
{
   accounts(username: "admin*") {
      username
      identity {
         name
      }
   }
}
```

Extra fields

The Ontrack GraphQL extension mechanism allows contributions to the project entities like the projects, builds, etc.

For example, to contribute a `owner` field of type `Account` on the `Branch` project entity:

```
@Component
public class BranchOwnerGraphQLFieldContributor
   implements GQLProjectEntityFieldContributor {

   private final AccountType accountType;

   @Autowired
   public BranchOwnerGraphQLFieldContributor(AccountType accountType) {
      this.accountType = accountType;
   }

   @Override
   public List<GraphQLFieldDefinition> getFields(
         Class<? extends ProjectEntity> projectEntityClass,
         ProjectEntityType projectEntityType) {
     return Collections.singletonList(
             GraphQLFieldDefinition.newFieldDefinition()
                     .name("owner")
                     .type(accountType.getType())
                     .dataFetcher(GraphqlUtils.fetcher(
                        Branch.class,
                        (environment, branch) -> return ...
                     ))
                     .build()
     );
}
```

You can now use the `owner` field in your queries:

```
{
   branches(id: 1) {
      name
      project {
         name
      }
      owner {
         username
         identity {
            name
         }
      }
   }
}
```

Built-in scalar fields

The Ontrack GraphQL module adds the following scalar types, which you can use in your field or type definitions:

- `GQLScalarJSON.INSTANCE` - maps to a `JsonNode`
- `GQLScalarLocalDateTime.INSTANCE` - maps to a `LocalDateTime`

You can use them directly in your definitions:

```
...
.field(f -> f.name("content").type(GQLScalarJSON.INSTANCE))
.field(f -> f.name("timestamp").type(GQLScalarLocalDateTime.INSTANCE))
...
```

Testing GraphQL

In your tests, create a test class which extends `AbstractQLITSupport` and use the `run` method to execute a GraphQL query:

```
MyTestIT extends AbstractQLITSupport {
   @Test
   void my_test() {
     def p = doCreateProject()
     def data = run("""{
      projects(id: ${p.id}) {
         name
      }
     }""")
     assert data.projects.first().name == p.name
   }
}
```

While it is possible to run GraphQL tests in Java, it's easier to do using Groovy.

Chapter 9. Appendixes

9.1. Configuration properties

Ontrack uses the Spring Boot mechanism for its configuration. See the documentation on how to set those properties in your Ontrack installation.

All Spring Boot properties are available for configuration.

Additionally, Ontrack defines the following ones.

The names of the configuration properties are given for a `.properties` file format but you can configure them in YAML of course. They can also be provided as system properties or environment variables. See the Spring Boot documentation for more details.

This sample file is meant as a guide only. Do **not** copy/paste the entire content into your application; rather pick only the properties that you need.

When applicable, the default value is mentioned.

```
# =====================================================
# Ontrack properties
# =====================================================

# Maximum number of days to keep the log entries
ontrack.config.application-log-retention-days = 7

# Maximum number of errors to display as notification in the GUI
ontrack.config.application-log-info-max = 10

# Directory which contains all the working files of Ontrack
# It is usually set by the installation
ontrack.config.application-working-dir = work/files

# Metrics export period (in seconds)
ontrack.config.metrics-period

# Testing the configurations of external configurations
# Used only for internal testing, to disable the checks
# when creating external configurations
ontrack.config.configuration-test = true

# Number of threads to use to run the background jobs
ontrack.config.jobs.pool-size = 10

# Interval (in minutes) between each refresh of the job list
ontrack.config.jobs.orchestration = 2

# Set to true to not start any job at application startup
# The administrator can restore the scheduling jobs manually
ontrack.config.jobs.paused-at-startup = false

# Enabling the scattering of jobs
# When several jobs have the same schedule, this can create a peak of activity,
# potentially harmful for the performances of the application
# Enabling scattering allows jobs to be scheduled with an additional delay, computed
# as a fraction of the period.
ontrack.config.jobs.scattering = false

# Scattering ratio. Maximum fraction of the period to take into account for the
# scattering. For example, setting 0.5 would not add a dealy greater than half
# the period of the job. Setting 0 would actually disable the scattering altogether.
ontrack.config.jobs.scattering-ratio = 1.0
```

9.2. Deprecations and migration notes

9.2.1. Since 2.28

BitBucket global configurations are no longer associated with issue services, only project BitBucket configurations are. This is an alignment with the way the other SCM connections are working in Ontrack.

Upgrading to 2.28 performs an automated migration of the global configuration settings to the project ones.

9.2.2. Since 2.16

Support for custom branch and tags patterns in Subversion configurations has been **removed**. Ontrack now supports only standard Subversion structure: `project/trunk`, `project/branches` and `project/tags`. This has allowed a better flexibility in the association between builds and Subversion locations.

Association between builds and Subversion locations is now configured through a *build revision link* at branch level. The previous `buildPath` parameter is converted automatically to the appropriate type of link.

9.3. Roadmap

Here are big ideas for the future of Ontrack. No plan yet, just rough ideas or wish lists.

9.3.1. Switch to Postgresql for the database layer

Partial support in version 3.0.

This could be replaced altogether by the support for Neo4j.

A migration to Postgres must take the following items into account:

- definition of the schema
- integration with Liquibase or Flyway (dropping DBInit)
- definition of the different ways of working:
  - local development
    - with Docker
    - with a Postgres instance
  - build
  - docker
  - installation (RPM, Debian, manual)
- extensions (like `svn`) can contribute to the schema

- migration procedure (automated if possible)
- use of JSON columns for configurations, properties, etc.
- optimization of queries using JSON operators

9.3.2. Use JPA / Hibernate for SQL queries

- caching (no existent today)
- see impact on multi Ontrack cluster

9.3.3. Using Neo4J as backend

Ontrack basically stores its data as a graph, and Neo4J would be a perfect match for the storage.

Consider:

- migration
- search engine

9.3.4. Global DSL

The current Ontrack DSL can be used only remotely and cannot be run on the server.

We could design a DSL which can be run, either:

- remotely - interacting with the HTTP API
- in the server - interacting directly with the services

Additionally, the DSL should be extensible so that extensions can contribute to it, on the same model than the Jenkins Job DSL.

9.3.5. Web hooks

Have the possibility to register Webhooks in Ontrack in order to notify other applications about changes and events.

9.4. Certificates

Some resources (Jenkins servers, ticketing systems, SCM...) will be configured and accessed in Ontrack using the `https` protocol, possibly with certificates that are not accepted by default.

Ontrack does not offer any mechanism to accept such invalid certificates.

The running JDK has to be configured in order to accept those certificates.

9.4.1. Registering a certificate in the JDK

To register the certificate in your JDK:

```
sudo keytool -importcert \
    -keystore ${JAVA_HOME}/jre/lib/security/cacerts -storepass changeit \
    -alias ${CER_ALIAS} \
    -file ${CER_FILE}
```

To display its content:

```
sudo keytool -list \
    -keystore ${JAVA_HOME}/jre/lib/security/cacerts \
    -storepass changeit \
    -alias ${CER_ALIAS} \
    -v
```

See the complete documentation at http://docs.oracle.com/javase/8/docs/technotes/tools/unix/keytool.html.

9.4.2. Saving the certificate on MacOS

1. Open the Keychain Access utility (Applications → Utilities)
2. Select your certificate or key from the Certificates or Keys category
3. Choose File → Export items ...
4. In the Save As field, enter a ".cer" name for the exported item, and click Save.

You will be prompted to enter a new export password for the item.

9.5. DSL Reference

9.5.1. AbstractProjectResource

See also: AbstractResource

Object config(Closure closure)

Configures this entity.

String getDescription()

Returns any description attached to this entity.

Object property(String type)

Gets a required property on the project entity.

If the property does not exist or is not set, a `net.nemerosa.ontrack.dsl.PropertyNotFoundException` is thrown.

```
def project = ontrack.project('PRJ')
def value = project.property('com.my.PropertyType')
```

Object property(String type, Object data)

Sets the value for a property of this entity. Prefer using dedicated DSL methods.

Object getDecoration(String type)

Gets the data for the first decoration of a given type. If no decoration is available, returns null.

List<> getDecorations(String type)

Returns the list of decoration data (JSON) for a given decoration type.

List<> getDecorations()

Returns the list of decorations for this entity. Each item has a `decorationType` type name and `data` as JSON.

Map<String> getMessageDecoration()

Gets any message for this entity

The message is returned as a map containing:

- the `type` of message: `ERROR`, `WARNING` or `INFO`
- the actual `text` of the message

The returned map is `null` is no message is associated with this entity.

```
def project = ontrack.project('prj')
def branch = ontrack.branch('prj', 'test')
def build = ontrack.build('prj', 'test', '1')
// No decorations yet
assert project.messageDecoration == null
assert branch.messageDecoration == null
assert build.messageDecoration == null
// Project decorations
project.config.message('Information', 'INFO')
assert project.messageDecoration == [type: 'INFO', text: 'Information']
// Branch decorations
branch.config.message('Warning', 'WARNING')
assert branch.messageDecoration == [type: 'WARNING', text: 'Warning']
// Build decorations
build.config.message('Error', 'ERROR')
assert build.messageDecoration == [type: 'ERROR', text: 'Error']
```

String getJenkinsJobDecoration()

Gets the Jenkins decoration for this entity.

Object getProperty(String type, boolean required)

Gets a property on the project entity.

If `required` is set to `false` and if the property does not exist or is not set, `null` will be returned.

If `required` is set to `true` and if the property does not exist or is not set, a `net.nemerosa.ontrack.dsl.PropertyNotFoundException` is thrown.

```
def project = ontrack.project('PRJ')
def value = project.getProperty('com.my.UnsetPropertyType', false)
assert value == null
```

String getName()

Returns the name of this entity.

Object delete()

Deletes this entity.

int getId()

Returns the numeric ID of this entity.

9.5.2. AbstractResource

String link(String name)

Gets a link address.

The `name` of the link refers to a link which is embedded in the JSON representation of the resource. In the JSON, the link name is prefixed by `_` but the `name` does not have too.

For example, `link('create')` and `link('_create')` are equivalent.

If the link does not exist, an `ResourceMissingLinkException` error is thrown.

The value of the link can be called using the Ontrack methods:

```
def project = ontrack.projects[0]
def projectJson = ontrack.get(project.link('self'))
```

String optionalLink(String name)

Gets a link address if it exists.

This is the same than the `link` method, but returns `null` is the link does not exist instead of failing.

String getPage()

Gets the Web page address for this resource.

This method returns the URL of the Web page which displays this resource.

This is equivalent to calling `link('page')`.

Object getNode()

Gets the internal JSON representation of this resource.

9.5.3. Account

Representation of a user account.

See also: AbstractResource

String getFullName()

Display name for the account.

AuthenticationSource getAuthenticationSource()

Source of the account: LDAP, built-in, ...

See: AuthenticationSource

String getEmail()

Email for the account.

String getRole()

Role for the user: admin or not.

List<AccountGroup> getAccountGroups()

List of groups this account belongs to.

See: AccountGroup

String getName()

User name, used for signing in.

int getId()

Unique ID for the account.

9.5.4. AccountGroup

Account group. Just a name and a description.

See also: AbstractResource

String getDescription()

Description of the group.

String getName()

Name of the group. Unique.

int getId()

Unique ID for the group.

9.5.5. Admin

Administration management

List<Account> getAccounts()

Returns the list of all accounts.

See: Account

Account account(String name, String fullName, String email, String password = '', List groupNames = [])

Creates or updates an account.

This method is used only to create *built-in* accounts. There is no need to create LDAP-based accounts.

The `name`, `fullName` and `email` parameters are required and unique. The password is required only when creating a new account, not when updating it.

The list of groups the account must belong to is provided using the names of the groups.

```
ontrack.admin {
    account 'test', 'Test user', 'xxx'
}
```

See: Account

List<AccountGroup> getGroups()

Returns the list of all groups.

See: AccountGroup

AccountGroup accountGroup(String name, String description)

Creates or updates an account group.

See: AccountGroup

List<GroupMapping> getLdapMappings()

Gets the list of LDAP mappings.

See: GroupMapping

GroupMapping ldapMapping(String name, String groupName)

Creates or updates a LDAP mapping.

The `name` parameter is the name of the group in the LDAP.

The `groupName` parameter is the name of the group in Ontrack. It must exist.

```
ontrack.admin {
    accountGroup 'MyGroup', 'An Ontrack group'
    ldapMapping 'GroupInLDAP', 'MyGroup'
}
```

See: GroupMapping

void setAccountGlobalPermission(String accountName, String globalRole)

Sets a global role on an account. See Account permissions.

List<Role> getAccountGlobalPermissions(String accountName)

Gets the list of global roles an account has. See Account permissions.

void setAccountProjectPermission(String projectName, String accountName, String projectRole)

Sets a project role on an account. See Account permissions.

List<Role> getAccountProjectPermissions(String projectName, String accountName)

Gets the list of roles an account has on a project. See Account permissions.

void setAccountGroupGlobalPermission(String groupName, String globalRole)

Sets a global role on an account group. See Account group permissions.

List<Role> getAccountGroupGlobalPermissions(String groupName)

Gets the list of global roles an account group has. See Account group permissions.

void setAccountGroupProjectPermission(String projectName, String groupName, String projectRole)

Sets a project role on an account group. See Account group permissions.

List<Role> getAccountGroupProjectPermissions(String projectName, String groupName)

Gets the list of roles an account group has on a project. See Account group permissions.

9.5.6. AuthenticationSource

Authentication source for an account - indicates how the account is authenticated: LDAP, built-in, etc.

See also: AbstractResource

boolean isAllowingPasswordChange()

Does this source allow to change the password?

String getName()

Display name for the source

String getId()

Identifier for the source: ldap, password

9.5.7. Branch

See also: AbstractProjectResource

Properties

See also: ProjectEntityProperties

Object svn(Map params = [:])

To get the SVN branch configuration:

```
def getSvn()
```

To associate a branch with some Subversion properties:

```
def svn (Map<String, ?> params)
```

The project the branch belongs to must be configured for Subversion.

The parameters are:

- `branchPath` - required - the path to the branch, relative to the repository
- `link` - optional - type of build link:
  - `tag` for build name as tag (default)
  - `tagPattern` for build name as tag pattern
  - `revision` for build name as revision
  - `revisionPattern` for build name containing the revision
  - see the [Subversion](Working with Subversion) documentation for more details
- `data` - optional - configuration data for the link, typically `[pattern: '…']` for `tagPattern` and `revisionPattern` links

Example:

```
ontrack.project('project') {
   config {
       svn 'myconfig', '/project/trunk'
   }
   branch('mybranch') {
      config {
           svn branchPath: '/project/branches/mybranch',
               link: 'revisionPattern',
               data: [pattern: '2.0.*-{revision}']
       }
   }
}
def cfg = ontrack.branch('project', 'mybranch').config.svn
assert cfg.branchPath == '/project/branches/mybranch'
assert cfg.buildRevisionLink.id == 'revisionPattern'
assert cfg.buildRevisionLink.data.pattern == '11.8.4.*-{revision}'
```

Object getSvn()

See Object svn(Map params = [:]).

Object gitBranch(String branch, Map params = [:])

Defines the Git properties for an Ontrack branch. `branch` is the name of the Git branch. Possible parameters are:

- `buildCommitLink.id` & `buildCommitLink.data` are the definition of the link between the Ontrack builds and the Git commits. See the samples below.
- `override` - `true` or `false` - defines if the synchronisation overrides or not the existing builds

(defaults to `false`)

- `buildTagInterval` - interval in minutes between each synchronisation between the builds and Git branch. `0` (default) to disable. Note that not all build commit links allow for synchronisation

Examples:

- for a branch whose build names are long commits (the default):

```
ontrack.branch('project') {
   branch('1.0') {
      config {
         gitBranch 'release/1.0'
      }
   }
}
```

- for a branch whose build names are short commits:

```
ontrack.branch('project') {
   branch('1.0') {
      config {
         gitBranch 'release/1.0', [
            buildCommitLink: [
               id: 'commit',
               data: [
                  abbreviated: true
               ]
            ]
         ]
      }
   }
}
```

- for a branch whose build names are associated to Git tags following a given pattern

```
ontrack.branch('project') {
   branch('1.0') {
      config {
         gitBranch 'release/1.0', [
            buildCommitLink: [
               id: 'tagPattern',
               data: [
                  pattern: '1.0.*'
               ]
            ]
         ]
      }
   }
}
```

- for a branch whose build are associated to Git commits using a Git property:

```
ontrack.branch('project') {
   branch('1.0') {
      config {
         gitBranch 'release/1.0', [
            buildCommitLink: [
               id: 'git-commit-property'
            ]
         ]
      }
   }
}
```

The builds of this branch can be associated with a Git commit using the `build.config.gitCommit` method.

Object getGitBranch()

See Object gitBranch(String branch, Map params = [:]).

Object svnValidatorClosedIssues(Collection closedStatuses)

A SVN-enabled branch can be associated with a validator in order to validate if there are some anomalies for the issues in the change logs.

```
def svnValidatorClosedIssues(Collection<String> closedStatuses)
```

Sets the list of issues statuses which can raise warnings if one of the issues is present *after* the change log.

```
def getSvnValidatorClosedIssues()
```

Gets the list of statuses to look for.

Example:

```
ontrack.configure {
   svn 'myconfig', url: 'svn://localhost'
}
ontrack.project('project') {
   config {
       svn 'myconfig', '/project/trunk'
   }
   branch('test') {
       config {
           svn '/project/branches/mybranch', '/project/tags/{build:mybranch-*}'
           svnValidatorClosedIssues(['Closed'])
       }
   }
}
assert ontrack.branch('project',
'test').config.svnValidatorClosedIssues.closedStatuses == ['Closed']
```

Object getSvnValidatorClosedIssues()

See Object svnValidatorClosedIssues(Collection closedStatuses).

Object svnSync(int interval = 0, boolean override = false)

For a Subversion-enabled branch, an automated synchronisation can be set in order to regularly create builds from the list of tags in Subversion.

`def svnSync(int interval = 0, boolean override = false)`

Sets a synchronisation every `interval` minutes (`0` meaning no sync at all). The `override` flag is used to allow existing builds to be overridden.

Example:

```
ontrack.configure {
   svn 'myconfig', url: 'svn://localhost'
}
ontrack.project('project') {
   config {
       svn 'myconfig', '/project/trunk'
   }
   branch('test') {
       config {
           svn '/project/branches/mybranch', '/project/tags/{build:mybranch-*}'
           svnSync 30
       }
   }
}
def sync = ontrack.branch('project', 'test').config.svnSync
assert sync.override == false
assert sync.interval == 30
```

Object getSvnSync()

See Object svnSync(int interval = 0, boolean override = false).

Object artifactorySync(String configuration, String buildName, String buildNameFilter = '*', int interval = 0)

Branch builds can be synchronised with Artifactory:

```
def artifactorySync(String configuration, String buildName, String buildNameFilter = '*', int interval = 0)
```

and the corresponding configuration can be accessed:

```
def getArtifactorySync()
```

Example:

```
ontrack.configure {
   artifactory 'Artifactory', 'http://artifactory'
}
ontrack.project('project') {
   branch('test') {
       config {
           artifactorySync 'Artifactory', 'test', 'test-*', 30
       }
   }
}
def sync = ontrack.branch('project', 'test').config.artifactorySync
assert sync.configuration.name == 'Artifactory'
assert sync.buildName == 'test'
assert sync.buildNameFilter == 'test-*'
assert sync.interval == 30
```

See also Artifactory configuration to have access to the list of available configurations.

Object getArtifactorySync()

See Object artifactorySync(String configuration, String buildName, String buildNameFilter = '*', int interval = 0).

Object unlink()

Disconnects the branch template instance from its template:

```
assert ontrack.branch('project', 'test').type == 'TEMPLATE_INSTANCE'
ontrack.branch('project', 'test').unlink()
assert ontrack.branch('project', 'test').type == 'CLASSIC'
```

Object call(Closure closure)

Configures the branch using a closure.

Build build(String name, String description = '', boolean getIfExists = false)

Creates a build for the branch

For example,

```
def build = ontrack.branch('project', 'branch').build('123', '', true)
```

Settings the `getIfExists` parameter to `true` will return the build if it already exists, and will fail if set to `false`.

See: Build

Build build(String name, String description = '', boolean getIfExists = false, Closure closure)

Creates a build for the branch and configures it using a closure. See Build build(String name, String description = '', boolean getIfExists = false).

See: Build

Object template(Closure closure)

Configure the branch as a template definition - see DSL Branch template definitions.

Object link(String templateName, boolean manual = true, Map parameters)

Links a branch to an existing template. It will fail if the branch is already linked to a template or is a template definition itself. See Object unlink() to unlink a branch to its template.

The `templateName` is the name of the branch template, and the `parameters` map contains the values

needed to instantiate the template.

You can put the `manual` flag to `false` if the template definition does not need any parameter to create the instance.

PromotionLevel promotionLevel(String name, String description = '', boolean getIfExists = false)

Creates a promotion level for this branch.

See: PromotionLevel

PromotionLevel promotionLevel(String name, String description = '', boolean getIfExists = false, Closure closure)

Creates a promotion level for this branch and configures it using a closure.

See: PromotionLevel

ValidationStamp validationStamp(String name, String description = '', boolean getIfExists = false)

Creates a validation stamp for this branch.

See: ValidationStamp

ValidationStamp validationStamp(String name, String description = '', boolean getIfExists = false, Closure closure)

Creates a validation stamp for this branch and configures it using a closure.

See: ValidationStamp

BranchProperties getConfig()

Access to the branch properties

String download(String path)

Download a file from the branch SCM.

The branch *must* be associated with a SCM branch, for Git or Subversion. If not, the call will fail.

The `path` is relative to the root of the branch in the SCM and is always returned as text.

For security, only users allowed to configure the project will be able to download a file from the branch. See Security for the list of roles.

See also DSL SCM extensions.

String getProject()

Returns the name of the project the branch belongs to.

```
def branch = ontrack.branch('prj', 'master')
assert branch.project == 'prj'
```

List<Build> standardFilter(Map filterConfig)

Returns a list of builds for the branch, filtered according to given criteria.

For example, to get the last build of a given promotion:

```
def branch = ontrack.branch('project', 'branch')
def builds = branch.standardFilter count: 1, withPromotionLevel: 'BRONZE'
def buildName = builds[0].name
```

Builds are always returned from the most recent one to the oldest.

The `standardFilter` method accepts the following parameters:

Parameter	Description
`count`	Maximum number of builds to be returned. Defaults to `10`.
`sincePromotionLevel`	Name of a promotion level. After reaching the first build having this promotion, no further build is returned.
`withPromotionLevel`	Name of a promotion level. Only builds having this promotion are returned.
`afterDate`	ISO 8601 date. Only builds having been created on or after this date are returned.
`beforeDate`	ISO 8601 date. Only builds having been created on or before this date are returned.
`sinceValidationStamp`	Name of a validation stamp. After reaching the first build having this validation (whatever the status), no further build is returned.
`sinceValidationStampStatus`	Refines the `sinceValidationStamp` criteria to check on the status of the validation.
`withValidationStamp`	Name of a validation stamp. Only builds having this validation (whatever the status) are returned.
`withValidationStampStatus`	Refines the `withValidationStamp` criteria to check on the status of the validation.

Parameter	Description
`withProperty`	Qualified name of a property (full class name of the `PropertyType`). Only builds having this property being set are returned.
`withPropertyValue`	Refines the `withProperty` criteria to check the property value. The way the value is matched with the actual value depends on the property.
`sinceProperty`	Qualified name of a property (full class name of the `PropertyType`).After reaching the first build having this property being set, no further build is returned.
`sincePropertyValue`	Refines the `sinceProperty` criteria to check the property value. The way the value is matched with the actual value depends on the property.
`linkedFrom`	Selects builds which are *linked from* the build selected by the criteria. See Build links for the exact syntax.
`linkedFromPromotion`	The build must be linked FROM a build having this promotion (requires "Linked from")
`linkedTo`	Selects builds which are *linked to* the build selected by the criteria. See Build links for the exact syntax.
`linkedToPromotion`	The build must be linked TO a build having this promotion (requires "Linked to")

See: Build

List<Build> getLastPromotedBuilds()

Returns the last promoted builds.

For example, to get the last promoted build:

```
def buildName = ontrack.branch('project', 'branch').lastPromotedBuilds[0].name
```

See: Build

Object syncInstance()

Synchronises the branch instance with its associated template. Will fail if this branch is not a template instance.

List<PromotionLevel> getPromotionLevels()

Gets the list of promotion levels for this branch.

See: PromotionLevel

List<ValidationStamp> getValidationStamps()

Gets the list of validation stamps for this branch.

See: ValidationStamp

boolean isDisabled()

Gets the disabled state of the branch

TemplateInstance getInstance()

If the branch is a template instance, returns a `TemplateInstance` object which contains the list of this instance parameters as a `Map` in the `parameters` property. Otherwise returns `null`.

[source,groovy

```
ontrack.project('project') {
   branch('template') {
       template {
           parameter 'paramName', 'A parameter'
       }
   }
}
ontrack.branch(project, 'template').instance 'TEST', [
       paramName: 'paramValue'
]
def instance = ontrack.branch(project, 'TEST').instance
assert instance.parameters == [paramName: 'paramValue']
```

String getType()

Returns the type of the branch when it comes to templating.

Possible values are:

- CLASSIC
- TEMPLATE_DEFINITION
- TEMPLATE_INSTANCE

List<Build> filter(String filterType, Map filterConfig)

Runs any filter and returns the list of corresponding builds.

This is a low level method and more specialised methods should instead be used like `standardFilter` and `getLastPromotedBuilds`.

See: Build

Object sync()

Synchronizes the branch template with its associated instances. Will fail if this branch is not a template.

Object enable()

Enables the branch

Object disable()

Disables the branch

Branch instance(String sourceName, Map params)

Creates or updates a new branch from this branch template. See DSL Branch template definitions.

9.5.8. Build

In order to get the change log between two builds, look at the documentation at DSL Change logs.

See also: AbstractProjectResource

Properties

See also: ProjectEntityProperties

Object getLabel()

See Object label(String name).

Object label(String name)

A label or release can be attached to a build using:

```
def label(String name)
```

For example:

```
ontrack.build('project', 'branch', 'build').config {
   label 'RC1'
}
```

To get the label associated with a build:

```
def name = ontrack.build('project', 'branch', 'build').config.label
assert name == 'RC1'
```

Object gitCommit(String commit)

Sets a Git commmit associated to this build.

When working with Git, it is needed to associate a build with a commit. It can be done by using the build name itself as a commit indicator (full, short, or tag) or by putting the commit as a build property:

`def gitCommit(String commit)`

To get the commit back:

`def getGitCommit()`

Example:

```
ontrack.project('project') {
   branch('test') {
       build('1') {
           config {
               gitCommit 'adef13'
           }
       }
   }
}
assert ontrack.build('project', 'test', '1').config.gitCommit == 'adef13'
```

Object getGitCommit()

Gets the Git commit associated to this build.

Object jenkinsBuild(String configuration, String job, int buildNumber)

Associates a Jenkins build with this build.

The `configuration` parameter is the name of an existing Jenkins configuration.

The `job` parameter is the path to the Jenkins job. For a job `test` as the Jenkins root, it would be only `test` but for a job `test2` in a folder `parent2` which is itself in a folder `parent1` at the root, this would be `parent1/parent2/test2`.

The `buildNumber` is the number of the Jenkins build.

this link is created automatically when using the Ontrack Jenkins plug-in.

Object getJenkinsBuild()

Gets the Jenkins build property.

Returns an object describing the associated Jenkins build or `null` if none.

The returned object contains the following attributes:

- a `configuration` object, having itself a `name` and a `url` attribute
- a `name` - path to the job in Jenkins
- a `url` - absolute URL to the build page
- a `pathComponents` list of string - same than `path` but as list of separate components
- a `build` number

Even if a link to a Jenkins build is registered, this does not mean that the actual Jenkins build still actually exists.

ValidationRun validate(String validationStamp, String validationStampStatus = 'PASSED', Closure closure)

Validates the build using the given validation stamp and status - possible values for the status are: `PASSED`, `FAILED`, `DEFECTIVE`, `EXPLAINED`, `FIXED`, `INTERRUPTED`, `INVESTIGATING`, `WARNING`

See: ValidationRun

ValidationRun validate(String validationStamp, String validationStampStatus = 'PASSED')

Validates the build using the given validation stamp and status - and configures the resulting validation run.

See: ValidationRun

Object call(Closure closure)

Configuration of the build in a closure.

PromotionRun promote(String promotion, Closure closure)

Promotes this build to the given promotion level and configures the created promotion run.

See: PromotionRun

PromotionRun promote(String promotion)

Promotes this build to the given promotion level.

See: PromotionRun

List<PromotionRun> getPromotionRuns()

Gets the list of promotion runs for this build

See: PromotionRun

List<ValidationRun> getValidationRuns()

Gets the list of validation runs for this build

See: ValidationRun

Object buildLink(String project, String build)

A build can be linked to other builds.

To create links:

```
def build = ...
build.buildLink 'project1', '11.0'
build.buildLink 'project2', '2.3.1'
```

Several links can be attached to a build, by calling the `buildLink` method several times.

The target project and build *must* exist.

To get the list of linked builds:

```
def links = build.buildLlinks
assert links.size == 2
def link = links[0]
assert link.project = 'project1'
assert link.branch = '...'
assert link.name = '11.0'
assert link.page = '...' // URL to the build page
```

The `page` property of the link is a URL to the page of the link.

List<Build> getBuildLinks()

See Object buildLink(String project, String build).

Long getSvnRevisionDecoration()

Returns any Subversion revision attached to this build.

```
def build = ontrack.build('project', 'branch', '1')
Long revision = build.svnRevisionDecoration
assert revision != null
```

BuildProperties getConfig()

String getProject()

Gets the build project name.

String getBranch()

Gets the build branch name.

9.5.9. Config

General configuration of Ontrack.

List<String> getGit()

See Object git(Map parameters, String name).

Object svn(Map parameters, String name)

Creates a or updates a Subversion configuration.

In order to create or update a Subversion configuration, use:

```
def svn(Map<String, ?> parameters, String name)
```

To get the list of Subversion configuration *names*, use:

```
def getSvn()
```

Example:

```
// Create a Subversion configuration
ontrack.configure {
   svn 'myconfig', url: 'https://myhost/repo'
}
// Gets the list of Subversion configuration names
def names = ontrack.config.svn*.name
```

The configuration password is always returned blank.

Some parameters, like `url` are required. List of parameters is:

Parameter	Description
url	URL of the Subversion repository (required)
user	
password	
tagFilterPattern	none by default
browserForPath	none by default

Parameter	Description
browserForRevision	none by default
browserForChange	none by default
indexationInterval	0 by default
indexationStart	revision to start the indexation from (1 by default)
issueServiceConfigurationIdentifier	identifier for the issue service (optional) - see here

Example of issue link (with JIRA):

```
ontrack.configure {
   jira 'MyJIRAConfig', 'http://jira'
   svn 'myconfig', url: 'https://myhost/repo', issueServiceConfigurationIdentifier:
'jira//MyJIRAConfig'
}
```

Object getSvn()

See Object svn(Map parameters, String name).

Object jenkins(String name, String url, String user = '', String password = '')

Creates or updates a Jenkins configuration.

Access to Jenkins is done through the configurations:

```
def jenkins(String name, String url, String user = '', String password = '')
```

The list of Jenkins configurations is accessible:

```
List<String> getJenkins()
```

Example:

```
ontrack.configure {
   jenkins 'Jenkins', 'http://jenkins'
}
assert ontrack.config.jenkins.find { it == 'Jenkins' } != null
```

List<String> getJenkins()

See Object jenkins(String name, String url, String user = '', String password = '').

Object jira(String name, String url, String user = '', String password = '')

Creates or updates a JIRA configuration.

Access to JIRA is done through the configurations:

```
def jira(String name, String url, String user = '', String password = '')
```

The list of JIRA configurations is accessible:

```
List<String> getJira()
```

Example:

```
ontrack.configure {
   jira 'JIRA', 'http://jira'
}
assert ontrack.config.jira.find { it == 'JIRA' } != null
```

List<String> getJira()

See Object jira(String name, String url, String user = '', String password = '').

Object artifactory(String name, String url, String user = '', String password = '')

Creates or updates a Artifactory configuration.

Access to Artifactory is done through the configurations:

```
def artifactory(String name, String url, String user = '', String password = '')
```

The list of Artifactory configurations is accessible:

```
List<String> getArtifactory()
```

Example:

```
ontrack.configure {
   artifactory 'Artifactory', 'http://artifactory'
}
assert ontrack.config.artifactory.find { it == 'Artifactory' } != null
```

List<String> getArtifactory()

See Object artifactory(String name, String url, String user = '', String password = '').

List<PredefinedValidationStamp> getPredefinedValidationStamps()

Gets the list of validation stamps. See Object autoValidationStamp(boolean autoCreate = true, boolean autoCreateIfNotPredefined = false).

See: PredefinedValidationStamp

PredefinedValidationStamp predefinedValidationStamp(String name, String description = '', boolean getIfExists = false, Closure closure)

See Object autoValidationStamp(boolean autoCreate = true, boolean autoCreateIfNotPredefined = false).

See: PredefinedValidationStamp

PredefinedValidationStamp predefinedValidationStamp(String name, String description = '', boolean getIfExists = false)

See Object autoValidationStamp(boolean autoCreate = true, boolean autoCreateIfNotPredefined = false).

See: PredefinedValidationStamp

List<PredefinedPromotionLevel> getPredefinedPromotionLevels()

Gets the list of promotion levels. See Object autoPromotionLevel(boolean autoCreate = true).

See: PredefinedPromotionLevel

boolean getGrantProjectViewToAll()

Checks if the projects are accessible in anonymous mode.

```
ontrack.config.grantProjectViewToAll = false
assert !ontrack.config.grantProjectViewToAll
```

PredefinedPromotionLevel predefinedPromotionLevel(String name, String description = '', boolean getIfExists = false, Closure closure)

See Object autoPromotionLevel(boolean autoCreate = true).

See: PredefinedPromotionLevel

PredefinedPromotionLevel predefinedPromotionLevel(String name, String description = '', boolean getIfExists = false)

See Object autoPromotionLevel(boolean autoCreate = true).

See: PredefinedPromotionLevel

LDAPSettings getLdapSettings()

Gets the global LDAP settings

See: LDAPSettings

Object setLdapSettings(LDAPSettings settings)

Sets the global LDAP settings

Object setGrantProjectViewToAll(boolean grantProjectViewToAll)

Sets if the projects are accessible in anonymous mode.

```
ontrack.config.grantProjectViewToAll = true
assert ontrack.config.grantProjectViewToAll
```

Object gitLab(Map parameters, String name)

When working with GitLab, the access to the GitLab API must be configured.

```
def gitLab(Map<String, ?> parameters, String name)
```

The `name` is the identifier of the configuration - if it already exists, it will be updated.

The parameters are the following:

Parameter	Description
url	the URL to the GitLab application, without any repository nor project reference
user	user used to connect to GitHub
password	**Personal Access Token** associated with the user
ignoreSslCertificate	Set to `true` if SSL checks must be disabled (optional, default to `false`)

The `password` field must be a **Personal Access Token** - using the actual password is **not** supported.

Example:

```
ontrack.configure {
   gitLab 'AcmeGitLab', url: 'https://gitlab.acme.com', user: 'user', password:
'abcdef'
}
assert ontrack.config.gitLab.find { it == 'AcmeGitLab' } != null
```

List<String> getGitLab()

See Object gitLab(Map parameters, String name).

Object gitHub(Map parameters, String name)

See Object gitHub(String name).

Object gitHub(String name)

When working with GitHub, the access to the GitHub API must be configured.

```
def gitHub(Map<String, ?> parameters, String name)
```

The `name` is the identifier of the configuration - if it already exists, it will be updated.

The parameters are the following:

Parameter	Description
url	the GitHub URL - is set by default to `https://github.com` if not filled in, allowing for using GitHub enterprise as well
user	user used to connect to GitHub (optional)
password	password used to connect to GitHub (optional)
oauth2Token	OAuth token to use instead of a user/password (optional)

See Working with GitHub to know the meaning of those parameters.

Example:

```
ontrack.configure {
   gitHub 'github.com', oauth2Token: 'ABCDEF'
}
assert ontrack.config.gitHub.find { it == 'github.com' } != null
```

You can also configure an anonymous access to https://github.com (not recommended) by doing:

```
ontrack.configure {
   gitHub 'github.com'
}
```

List<String> getGitHub()

See Object gitHub(String name).

Object stash(Map parameters, String name)

Creates or updates a BitBucket configuration.

When working with BitBucket, the access to the BitBucket application must be configured.

`def stash(Map<String, ?> parameters, String name)`

The `name` is the identifier of the configuration - if it already exists, it will be updated.

The parameters are the following:

Parameter	Description
url	the URL of the Stash instance
user L user used to connect to Stash (optional)	password

Example:

```
ontrack.configure {
   stash 'MyStash', url: 'https://stask.example.com'
}
assert ontrack.config.stash.find { it == 'MyStash' } != null
```

List<String> getStash()

See Object stash(Map parameters, String name).

Object git(Map parameters, String name)

Creates or update a Git configuration

When working with Git, the access to the Git repositories must be configured.

`def git(Map<String, ?> parameters, String name)`

The `name` is the identifier of the configuration - if it already exists, it will be updated.

The parameters are the following:

Parameter	Description
remote	the remote location
user	user used to connect to GitHub (optional)
password	password used to connect to GitHub (optional)
commitLink	Link to a commit, using `{commit}` as placeholder
fileAtCommitLink	Link to a file at a given commit, using `{commit}` and `{path}` as placeholders
indexationInterval	interval (in minutes) between each synchronisation (Ontrack maintains internally a clone of the GitHub repository)
issueServiceConfigurationIdentifier	identifier for the linked issues (see example here)

See the documentation to know the meaning of those parameters.

Example:

```
ontrack.configure {
   git 'ontrack', remote: 'https://github.com/nemerosa/ontrack.git', user: 'test',
password: 'secret'
}
assert ontrack.config.git.find { it == 'ontrack' } != null
```

9.5.10. Document

Definition for a document, for upload and download methods. See also DSL Images and documents.

boolean isEmpty()

Returns true is the document is empty and has no content.

String getType()

Returns the MIME type of the document.

byte[] getContent()

Returns the content of the document as an array of bytes.

9.5.11. GroupMapping

Mapping between a LDAP group and an account group.

See also: AbstractResource

String getGroupName()

Name of the Ontrack account group.

String getName()

Name of the LDAP group.

9.5.12. LDAPSettings

LDAP settings parameters.

The LDAP settings are defined using the following values:

Parameter	Description
enabled	Set to `true` to actually enable the LDAP Authentication

Parameter	Description
url	URL to the LDAP end point. For example, `ldaps://ldap.company.com:636`
searchBase	DN for the search root, for example, `dc=company,dc=com`
searchFilter	Query to look for an account. `{0}` is replaced by the account name. For example, `(sAMAccountName={0})`
user	Service account user to connect to the LDAP
password	Password of the service account user to connect to the LDAP.
fullNameAttribute	Attribute which contains the display name of the account. Defaults to `cn`
emailAttribute	Attribute which contains the email of the account. Default to `email`.
groupAttribute	Multiple attribute name which contains the groups the account belong to. Defaults to `memberOf`.
groupFilter	When getting the list of groups for an account, filter this list using the `OU` attribute of the group. Defaults to blank (no filtering)

When *getting* the LDAP settings, the `password` field is always returned as an empty string.

For example, to set the LDAP settings:

```
ontrack.config.ldapSettings = [
        enabled     : true,
        url         : 'ldaps://ldap.company.com:636',
        searchBase  : 'dc=company,dc=com',
        searchFilter: '(sAMAccountName={0})',
        user        : 'service',
        password    : 'secret',
]
```

9.5.13. Ontrack

An Ontrack instance is usually bound to the `ontrack` identifier and is the root for all DSL calls.

Object text(String url)

Runs an arbitrary GET request for a relative path and returns text

Build build(String project, String branch, String build)

Looks for a build by name. Fails if not found.

```
def build = ontrack.build('prj', 'master', '1.0.0-12')
assert build.name == '1.0.0-12'
```

See: Build

Object post(String url, Object data)

Runs an arbitrary POST request for a relative path and some data, and returns JSON

List<Project> getProjects()

Gets the list of projects

```
def projects = ontrack.projects
assert projects instanceof Collection
```

See: Project

Project project(String name, String description = '', Closure closure)

Finds or creates a project, and configures it.

```
def project = ontrack.project('my-project', 'My project') {
    // Configuring the project
}
assert project.name == 'my-project'
```

See: Project

Project project(String name, String description = '')

Finds or creates a project.

```
def project = ontrack.project('my-project', 'My project')
assert project.name == 'my-project'
```

See: Project

Project findProject(String name)

Finds a project using its name. Returns null if not found.

```
def project = ontrack.findProject('unexisting')
assert project == null
```

See: Project

Branch branch(String project, String branch)

Looks for a branch in a project. Fails if not found.

See: Branch

PromotionLevel promotionLevel(String project, String branch, String promotionLevel)

Looks for a promotion level by name. Fails if not found.

See: PromotionLevel

ValidationStamp validationStamp(String project, String branch, String validationStamp)

Looks for a validation stamp by name. Fails if not found.

See: ValidationStamp

Object configure(Closure closure)

Configures the general settings of Ontrack. See Config.

Config getConfig()

Access to the general configuration of Ontrack

See: Config

Admin getAdmin()

Access to the administration of Ontrack

See: Admin

Object upload(String url, String name, Object o)

Uploads some arbitrary binary data on a relative path and returns some JSON. See Object upload(String url, String name, Object o, String contentType).

Object upload(String url, String name, Object o, String contentType)

Uploads some typed data on a relative path and returns some JSON

Creates a multi-part upload request where the `name` part contains the content of the given input `o` object. The content depends on the type of object:

- for a `net.nemerosa.ontrack.dsl.Document`, uploads this document - the `contentType` parameter is then ignored
- for a `File` or `URL`, the content of the file is uploaded, using the provided `contentType`
- for a `byte` array, the content of the array is uploaded, using the provided `contentType`
- for a `String`:
  - if starting with `classpath:`, the remainder of the string is used as a resource path (see `URL` above)
  - if a valid `URL`, we use the string as a URL - see above
  - if any other case, we consider the string to be the path to a file - see `File` above

Document download(String url)

Downloads an arbitrary document using a relative path.

See: Document

Object graphQLQuery(String query, Map variables = [:])

The `graphQLQuery` runs a GraphQL query against the Ontrack model.

It returns a JSON representation of an `ExecutionResult` object containing both the data and any error message.

```
def result = ontrack.graphQLQuery("""{
    projects(name: "P") {
        name
        branches {
            name
        }
    }
}""")

assert result.errors != null && result.errors.empty
assert result.data.projects.get(0).name == "P"
```

The `graphQLQuery` method accepts an additional `variables` map parameter to define any GraphQL variable used in the query. For example:

```
def result = ontrack.graphQLQuery("""{
    projects(id: $projectId) {
        name
        branches {
            name
        }
    }
}""", [projectId: 10])
```

See GraphQL support for more information about the GraphQL Ontrack integration.

Object get(String url)

Runs an arbitrary GET request for a relative path and returns JSON

Object put(String url, Object data)

Runs an arbitrary PUT request for a relative path and some data, and returns JSON

Object delete(String url)

Runs an arbitrary DELETE request for a relative path and returns JSON

List<SearchResult> search(String token)

Launches a global search based on a token.

See: SearchResult

9.5.14. PredefinedPromotionLevel

See also: AbstractResource

Document getImage()

Downloads the image for the promotion level. See DSL Images and documents.

See: Document

Object image(Object o)

Sets the image for this validation stamp (must be a PNG file). See DSL Images and documents.

9.5.15. PredefinedValidationStamp

See also: AbstractResource

Document getImage()

Downloads the image for the validation stamp. See DSL Images and documents.

See: Document

Object image(Object o)

Sets the image for this validation stamp (must be a PNG file). See DSL Images and documents.

9.5.16. Project

The project is the main entity of Ontrack.

```
// Getting a project
def project = ontrack.project('project')
project {
   // Creates a branch for the project
   branch('1.0')
}
```

See also: AbstractProjectResource

Properties

See also: ProjectEntityProperties

Object stale(int disablingDuration = 0, int deletingDuration = 0, List promotionsToKeep = [])

Setup of stale branches management.

Stale branches can be automatically disabled or even deleted.

To enable this property on a project:

```
ontrack.project('project').config {
   stale 15, 30
}

def property = ontrack.project('project').config.stale
assert property.disablingDuration == 15
assert property.deletingDuration == 30
assert property.promotionsToKeep == []
```

It is possible to make sure to keep branches which have been promoted to some levels. For example, if you want to keep branches which have been promoted to `PROMOTION`:

```
ontrack.project('project').config {
   stale 15, 30, ['PRODUCTION']
}

def property = ontrack.project('project').config.stale
assert property.disablingDuration == 15
assert property.deletingDuration == 30
assert property.promotionsToKeep == ['PRODUCTION']
```

Object getGit()

See Object git(String name).

Object svn(String name, String projectPath)

Configures the project for Subversion.

To associate a project with an existing Subversion configuration:

```
def svn (String name, String projectPath)
```

To get the SVN project configuration:

```
def getSvn()
```

Example:

```
// Associates a project with a 'myconfig' SVN configuration
ontrack.project('project') {
   config {
       svn 'myconfig', '/project/trunk'
   }
}
// Gets the SVN configuration
def cfg = ontrack.project('project').config.svn
assert cfg.configuration.name == 'myconfig'
assert cfg.projectPath == '/project/trunk'
```

Object getSvn()

See Object svn(String name, String projectPath).

Object gitLab(Map parameters, String name)

Object getGitLab()

See Object gitLab(Map parameters, String name).

Object gitHub(Map parameters, String name)

Configures the project for GitHub.

Object stash(String name, String project, String repository, int indexationInterval = 0, String issueServiceConfigurationIdentifier = '')

Associates the project with the BitBucket configuration with the given `name` and specifies the project in BitBucket and the repository.

Example:

```
ontrack.configure {
   jira 'MyJIRA', 'https://jira.example.com', 'user', 'password'
   stash 'MyStash', repository: 'https://stash.example.com', 'user', 'password'
}
ontrack.project('project') {
   config {
       stash 'MyStash', 'PROJECT', 'my-repo', 30, "jira//MyJIRA"
   }
}
assert ontrack.project('project').config.stash.configuration.name == 'MyStash'
assert ontrack.project('project').config.stash.project == 'PROJECT'
assert ontrack.project('project').config.stash.repository == 'my-repo'
assert ontrack.project('project').config.stash.indexationInterval == 30
assert ontrack.project('project').config.stash.issueServiceConfigurationIdentifier ==
"jira//MyJIRA"
assert ontrack.project('project').config.stash.repositoryUrl ==
'https://stash.example.com/projects/PROJECT/repos/my-repo'
```

Object getStash()

See Object stash(String name, String project, String repository, int indexationInterval = 0, String issueServiceConfigurationIdentifier = '').

Object git(String name)

Configures the project for Git.

Associates a project with a Git configuration.

```
def git(String name)
```

Gets the associated Git configuration:

```
def getGit()
```

Example:

```
ontrack.configure {
   git 'ontrack', remote: 'https://github.com/nemerosa/ontrack.git', user: 'test',
password: 'secret'
}
ontrack.project('project') {
   config {
       git 'ontrack'
   }
}
def cfg = ontrack.project('project').config.git
assert cfg.configuration.name == 'ontrack'
```

Object getStale()

See Object stale(int disablingDuration = 0, int deletingDuration = 0, List promotionsToKeep = []).

Object jiraFollowLinks(Collection linkNames)

See Object jiraFollowLinks(String[] linkNames).

Object jiraFollowLinks(String[] linkNames)

Links between JIRA issues can be followed when getting information about issues.

The links to follow can be configured at the project's level:

- `def jiraFollowLinks(String… linkNames)`
- `def jiraFollowLinks(Collection<String> linkNames)`

The list of links to follow is accessible through:

- `List<String> getJiraFollowLinks()`

Example:

```
ontrack.project('project') {
   config {
       jiraFollowLinks 'Clones', 'Depends'
   }
}
assert ontrack.project('project').config.jiraFollowLinks == ['Clones', 'Depends']
```

List<String> getJiraFollowLinks()

See Object jiraFollowLinks(String[] linkNames).

Object autoValidationStamp(boolean autoCreate = true, boolean autoCreateIfNotPredefined = false)

Validation stamps can be automatically created for a branch, from a list of predefined validation stamps, if the "Auto validation stamps" property is enabled on a project.

To enable this property on a project:

```
ontrack.project('project') {
   config {
      autoValidationStamp()
   }
}
```

or:

```
ontrack.project('project') {
   config {
      autoValidationStamp(true)
   }
}
```

You can also edit the property so that a validation stamp is created even when no predefined validation stamp does exit. In this case, the validation stamp will be created with the required name and without any image. To enable this feature:

```
ontrack.project('project') {
   config {
      autoValidationStamp(true, true)
   }
}
```

To get the value of this property:

```
boolean auto = ontrack.project('project').autoValidationStamp
```

The list of predefined validation stamps is accessible using:

```
def stamps = ontrack.config.predefinedValidationStamps
```

Each item contains the following properties:

- `id`
- `name`
- `description`

Its image is accessible through the `image` property.

In order to create/update predefined validation stamps, use the following method:

```
ontrack.config.predefinedValidationStamp('VS') {
   image new File('my/image.png')
}
```

You need Administrator rights to be able to update the predefined validation stamps.

boolean getAutoValidationStamp()

See Object autoValidationStamp(boolean autoCreate = true, boolean autoCreateIfNotPredefined = false).

Object autoPromotionLevel(boolean autoCreate = true)

Promotion levels can be automatically created for a branch, from a list of predefined promotion levels, if the "Auto promotion levels" property is enabled on a project.

To enable this property on a project:

```
ontrack.project('project') {
   config {
      autoPromotionLevel()
   }
}
```

or:

```
ontrack.project('project') {
   config {
      autoPromotionLevel(true)
   }
}
```

To get the value of this property:

```
boolean auto = ontrack.project('project').autoPromotionLevel
```

The list of predefined promotion levels is accessible using:

```
def stamps = ontrack.config.predefinedPromotionLevels
```

Each item contains the following properties:

- `id`
- `name`

- `description`

Its image is accessible through the `image` property.

In order to create/update predefined promotion levels, use the following method:

```
ontrack.config.predefinedPromotionLevel('VS') {
   image new File('my/image.png')
}
```

You need Administrator rights to be able to update the predefined promotion levels.

boolean getAutoPromotionLevel()

See Object autoPromotionLevel(boolean autoCreate = true).

Branch branch(String name, String description = '', boolean getIfExists = false)

Retrieves or creates a branch for the project

If the branch already exists, the `getIfExists` parameter is used:

- if `false` (default), an error is thrown
- if `true`, the existing branch is returned

If the branch does not exist, it is created.

```
def project = ontrack.project('test')
def branch = project.branch('new-branch')
```

See: Branch

Branch branch(String name, String description = '', boolean getIfExists = false, Closure closure)

Retrieves or creates a branch for the project, and then configures it.

See: Branch

ProjectProperties getConfig()

Access to the project properties

List<Branch> getBranches()

Gets the list of branches for the project.

See: Branch

List<Build> search(Map form)

Searches for builds in the project.

Possible options are:

Parameter	Description	Default
`maximumCount`	Maximum number of results to return	10
`branchName`	Regular expression for the branch a build belong to	none
`buildName`	Regular expression for the build name	none
`buildExactMatch`	Considers the `buildName` as being an exact match, and not a regular expression	`false`
`promotionName`	Name of a promotion level a build is promoted to	none
`validationStampName`	Name of a validation stamp a build has been validated to with status `PASSED`	??none
`property`	Qualified name of a property that the build must have	none
`propertyValue`	Together with `property`, refines the filter by checking the value of the build property. The way the value is matched with the actual value depends on the property	none
`linkedFrom`	Selects builds which are *linked from* the build selected by the criteria. See Build links for the exact syntax.	none
`linkedTo`	Selects builds which are *linked to* the build selected by the criteria. See Build links for the exact syntax.	none

Example of build searches:

```
def project = ontrack.project('project')
// List of last builds
def builds = project.search()
// Last build only
def build = project.search(maximumCount: 1)[0]
// Last build promoted to BRONZE
def build = project.search(promotionName: 'BRONZE', maximumCount: 1)[0]
// Creating a branch
ontrack.project('project') {
   branch 'MyBranch'
}
// Getting the list of branches
assert ontrack.project('project').branches.find { it.name == 'MyBranch' }
```

See: Build

9.5.17. PromotionLevel

See also: AbstractProjectResource

Properties

Builds can be auto promoted to a promotion level when this latter is configured to do so.

A promotion level is configured for auto promotion using:

```
ontrack.promotionLevel('project', 'branch', 'promotionLevel').config {
   autoPromotion 'VS1', 'VS2'
}
```

where `VS1`, `VS2` are the validation stamps which must be `PASSED` in order to promote a build automatically.

To get the list of validation stamps for the auto promotion of a promotion level:

```
def validationStamps = ontrack.promotionLevel('project', 'branch', 'promotionLevel')
   .config
   .autoPromotion
   .validationStamps
```

The validation stamps used to define an auto promotion can also be defined using regular expressions:

```
ontrack.promotionLevel('project', 'branch', 'promotionLevel').config {
   autoPromotion [], 'VS.*'
}
```

In this sample, all validation stamps whose name starts with `VS` will participate in the promotion.

You can also exclude validation stamps using their name:

```
ontrack.promotionLevel('project', 'branch', 'promotionLevel').config {
   autoPromotion [], 'VS.*', 'VS/.1'
}
```

In this sample, all validation stamps whose name starts with `VS` will participate in the promotion, but for the `VS.1` one.

See also: ProjectEntityProperties

Object autoPromotion(String[] validationStamps)

Sets the validation stamps participating into the auto promotion.

Object autoPromotion(Collection validationStamps, String include = '', String exclude = '')

Sets the validation stamps participating into the auto promotion, and sets the include/exclude settings.

boolean getAutoPromotion()

Checks if the promotion level is set in auto promotion.

Document getImage()

Gets the promotion level image (see DSL Images and documents)

See: Document

Object call(Closure closure)

Configuration of the promotion level with a closure.

PromotionLevelProperties getConfig()

Access to the promotion level properties

String getProject()

Name of the associated project.

String getBranch()

Name of the associated branch.

Object image(Object o, String contentType)

Sets the promotion level image (see DSL Images and documents)

Object image(Object o)

Sets the promotion level image (see DSL Images and documents)

Boolean getAutoPromotionPropertyDecoration()

Checks if this promotion level is set in auto decoration mode.

9.5.18. PromotionRun

You can get a promotion run by promoting a build:

```
def run = ontrack.build('project', 'branch', '1').promote('BRONZE')
assert run.promotionLevel.name == 'BRONZE'
```

or by getting the list of promotion runs for a build:

```
def runs = ontrack.build('project', 'branch', '1').promotionRuns
assert runs.size() == 1
assert runs[0].promotionLevel.name == 'BRONZE'
```

See also: AbstractProjectResource

Object getPromotionLevel()

Gets the associated promotion level (JSON)

9.5.19. SearchResult

The `SearchResult` class is used for listing the results of a search.

See also: AbstractResource

```
ontrack.project('prj')
def results = ontrack.search('prj')
assert results.size() == 1
assert results[0].title == 'Project prj'
assert results[0].page == 'https://host/#/project/1'
assert results[0].page == 'https://host/#/project/1'
```

String getDescription()

Gets a description for the search result.

String getPage()

Gets the URI to display the search result details (Web).

String getTitle()

Gets the display name for the search result.

String getUri()

Gets the URI to access the search result details (API).

int getAccuracy()

Gets a percentage of accuracy about the result.

9.5.20. ValidationRun

You can get a validation run by validating a build:

```
def run = ontrack.build(branch.project, branch.name, '2').validate('SMOKE', 'FAILED')
assert run.validationStamp.name == 'SMOKE'
assert run.validationRunStatuses[0].statusID.id == 'FAILED'
assert run.validationRunStatuses[0].statusID.name == 'Failed'
assert run.status == 'FAILED'
```

or by getting the list of validation runs for a build:

```
def runs = ontrack.build(branch.project, branch.name, '2').validationRuns
assert runs.size() == 1
assert runs[0].validationStamp.name == 'SMOKE'
assert runs[0].validationRunStatuses[0].statusID.id == 'FAILED'
assert runs[0].status == 'FAILED'
```

See also: AbstractProjectResource

String getStatus()

Gets the status for this validation run.

Possible values are:

- DEFECTIVE
- EXPLAINED
- FAILED
- FIXED
- INTERRUPTED
- INVESTIGATING
- PASSED
- WARNING

Object getValidationRunStatuses()

Gets the list of statuses (JSON)

Object getValidationStamp()

Gets the associated validation stamp (JSON)

9.5.21. ValidationStamp

See also: AbstractProjectResource

Document getImage()

Gets the validation stamp image (see DSL Images and documents)

See: Document

Object call(Closure closure)

Configuration of the promotion level with a closure.

String getProject()

Name of the associated project.

String getBranch()

Name of the associated branch.

Object image(Object o, String contentType)

Sets the validation stamp image (see DSL Images and documents)

Object image(Object o)

Sets the validation stamp image (see DSL Images and documents)

Object getValidationStampWeatherDecoration()

Gets the validation stamp weather decoration.

The "weather" of the validation stamp is the status of the last 4 builds having been validated for this validation stamp on the corresponding branch.

The returned object contains two attributes:

- `weather` which can have one of the following values:
  - `sunny`
  - `sunAndClouds`
  - `clouds`

    - `rain`
    - `storm`

- `text` - a display text for the weather type

9.5.22. ProjectEntityProperties

Object property(String type, Map data)

Sets a property.

Object property(String type, boolean required = true)

Gets a property on the project entity.

If `required` is set to `false` and if the property does not exist or is not set, `null` will be returned.

If `required` is set to `true` and if the property does not exist or is not set, a `net.nemerosa.ontrack.dsl.PropertyNotFoundException` is thrown.

```
def project = ontrack.project('PRJ')
def value = project.getProperty('com.my.UnsetPropertyType', false)
assert value == null
```

Object links(Map links)

Arbitrary named links can be associated with projects, branches, etc.

```
ontrack.project('project') {
   config {
       links 'project': 'http://project'
   }
   branch('test') {
       config {
           links 'branch': 'http://branch'
       }
       build('1') {
           config {
               links 'build': 'http://build'
           }
       }
   }
}
assert ontrack.project('project').config.links.project == 'http://project'
assert ontrack.branch('project', 'test').config.links.branch == 'http://branch'
assert ontrack.build('project', 'test', '1').config.links.build == 'http://build'
```

Map<String,String> getLinks()

See Object links(Map links).

Object metaInfo(String name, String value, String link = null, String category = null)

See Object metaInfo(Map map).

Object metaInfo(Map map)

Arbitrary meta information properties can be associated with any entity.

To set a list of meta information properties:

```
entity.config {
   metaInfo A: '1', B: '2'
}
```

This method does not allow to set links and will erase any previous meta information.

The following method is the preferred one:

```
ontrack.build('project', 'branch', '1').config {
   metaInfo 'name1', 'value1'
   metaInfo 'name2', 'value2', 'http://link2'
   metaInfo 'name3', 'value3', 'http://link3', 'Category'
}
```

This allows to keep any previous meta information property and to specify links if needed.

To get the meta information, for example on the previous build:

```
def list = ontrack.build('project', 'branch', '1').config.metaInfo
assert list.size() == 3
assert list[0].name = 'name1'
assert list[0].value = 'value1'
assert list[0].link = null
assert list[0].category = null
assert list[1].name = 'name2'
assert list[1].value = 'value2'
assert list[1].link = 'http://link2'
assert list[1].category = null
assert list[2].name = 'name3'
assert list[2].value = 'value3'
assert list[2].link = 'http://link3'
assert list[2].category = 'Category'
```

See [property-meta] for more details about this property.

List<MetaInfo> getMetaInfo()

See Object metaInfo(Map map).

Object jenkinsJob(String configuration, String job)

Projects, branches, promotion levels and validation stamps can have a reference to a Jenkins job:

```
def jenkinsJob(String configuration, String job)
```

or to get the job reference:

```
def getJenkinsJob()
```

Example:

```
ontrack.configure {
   jenkins 'Jenkins', 'http://jenkins'
}
ontrack.project('project') {
   config {
       jenkinsJob 'Jenkins', 'MyProject'
   }
   branch('test') {
       config {
           jenkinsJob 'Jenkins', 'MyBranch'
       }
       promotionLevel('COPPER') {
           config {
               jenkinsJob 'Jenkins', 'MyPromotion'
           }
       }
       validationStamp('TEST') {
           config {
               jenkinsJob 'Jenkins', 'MyValidation'
           }
       }
   }
}

def j = ontrack.project('project').config.jenkinsJob
assert j.configuration.name == 'Jenkins'
assert j.job == 'MyProject'
assert j.url == 'http://jenkins/job/MyProject'

j = ontrack.branch('project', 'test').config.jenkinsJob
assert j.configuration.name == 'Jenkins'
assert j.job == 'MyBranch'
assert j.url == 'http://jenkins/job/MyBranch'

j = ontrack.promotionLevel('project', 'test', 'COPPER').config.jenkinsJob
assert j.configuration.name == 'Jenkins'
assert j.job == 'MyPromotion'
assert j.url == 'http://jenkins/job/MyPromotion'

j = ontrack.validationStamp('project', 'test', 'TEST').config.jenkinsJob
assert j.configuration.name == 'Jenkins'
assert j.job == 'MyValidation'
assert j.url == 'http://jenkins/job/MyValidation'
```

Note that Jenkins folders are supported by giving the full job name. For example, the job name to give to the job in *A* > *B* > *C* would be `A/job/B/job/C` or even `A/B/C`.

See also the Jenkins build property.

Object getJenkinsJob()

See Object jenkinsJob(String configuration, String job).

Object jenkinsBuild(String configuration, String job, int build)

For builds, promotion runs and validation runs, it is possible to attach a reference to a Jenkins build:

```
def jenkinsBuild(String configuration, String job, int buildNumber)
```

or to get the build reference:

```
def getJenkinsBuild()
```

Example:

```
ontrack.configure {
   jenkins 'Jenkins', 'http://jenkins'
}
def name = uid('P')
def project = ontrack.project(name)
def branch = project.branch('test') {
   promotionLevel('COPPER')
   validationStamp('TEST')
}
def build = branch.build('1') {
   config {
       jenkinsBuild 'Jenkins', 'MyBuild', 1
   }
   promote('COPPER') {
       config {
           jenkinsBuild 'Jenkins', 'MyPromotion', 1
       }
   }
   validate('TEST') {
       config {
           jenkinsBuild 'Jenkins', 'MyValidation', 1
       }
   }
}

def j = ontrack.build(name, 'test', '1').config.jenkinsBuild
assert j.configuration.name == 'Jenkins'
assert j.job == 'MyBuild'
assert j.build == 1
assert j.url == 'http://jenkins/job/MyBuild/1'

// Promotion run build

j = ontrack.build(name, 'test', '1').promotionRuns[0].config.jenkinsBuild
assert j.configuration.name == 'Jenkins'
assert j.job == 'MyPromotion'
assert j.build == 1
assert j.url == 'http://jenkins/job/MyPromotion/1'

// Validation run build

j = ontrack.build(name, 'test', '1').validationRuns[0].config.jenkinsBuild
assert j.configuration.name == 'Jenkins'
assert j.job == 'MyValidation'
assert j.build == 1
assert j.url == 'http://jenkins/job/MyValidation/1'
```

Note that Jenkins folders are supported by giving the full job name. For example, the job name to give to the job in *A* > *B* > *C* would be `A/job/B/job/C` or even `A/B/C`.

See also the Jenkins job property.

Object getJenkinsBuild()

See Object jenkinsBuild(String configuration, String job, int build).

Object getMessage()

See Object message(String text, String type = 'INFO').

Object message(String text, String type = 'INFO')

An arbitrary message, together with a message type, can be associated with any entity.

To set the message on any entity:

```
entity.message(String text, String type = 'INFO')
```

Following types of messages are supported:

- `INFO`
- `WARNING`
- `ERROR`

For example, on a build:

```
ontrack.build('project', 'branch', '1').config {
   message 'My message', 'WARNING'
}
```

To get a message:

```
def msg = ontrack.build('project', 'branch', '1').config.message
assert msg.type == 'WARNING'
assert msg.text == 'My message'
```

See [property-message] for more details about this property.