

Ontrack Reference Guide 3.35.10

Damien Coraboeuf

Table of Contents

1. Installation	1
1.1. Prerequisites	1
1.2. Quick start	1
1.3. Postgres database	1
1.4. Installing using Docker	2
1.4.1. Overview	2
1.4.2. Basic deployment	2
1.4.3. Database persisted in volume	3
1.4.4. Database as H2 server	3
1.4.5. Docker Compose deployment	3
1.5. RPM installation	4
1.6. Debian installation	5
1.7. Standalone installation	6
1.8. Configuration	6
2. Basics	8
2.1. Managing projects	8
2.2. Managing branches	8
2.2.1. Managing the branches in the project page	8
2.2.2. Branch templating	8
2.2.3. Managing stale branches	9
2.2.4. Validation stamp filters	10
Using filters	10
Editing filters	11
Sharing	14
2.3. Managing validation stamps	15
2.3.1. Auto creation of validation stamps	15
Predefined validation stamps	16
Configuring projects	16
Auto creation of validation stamps	16
Auto creation of validation stamps when not predefined	16
2.3.2. Bulk update of validation stamps	17
2.4. Managing promotion levels	17
2.4.1. Auto promotion	17
2.4.2. Auto creation	17
Predefined promotion levels	18
Configuring projects	18
Auto creation of promotion levels	18
2.5. Managing the builds	18

2.5.1. Filtering the builds	19
Sharing filters	19
2.5.2. Build links	19
Definition of links	19
Decorations	20
Information	21
Querying	21
2.5.3. Run info	21
2.6. Managing validation runs	22
2.7. Properties	23
2.7.1. Message property	23
2.7.2. Meta information property	24
Input	24
Representation	24
Querying	24
3. Topics	26
3.1. Branch templates	26
3.1.1. Template Definition	26
3.1.2. Template Instances	26
3.1.3. Template expressions	28
Examples	28
3.2. Working with Git	28
3.2.1. Working with GitHub	28
General configuration	28
Project configuration	29
3.2.2. Working with GitLab	29
General configuration	29
Project configuration	30
3.2.3. Working with BitBucket	30
General configuration	30
Project configuration	30
3.2.4. Working with Subversion	31
Subversion configurations	31
Indexation	31
Project configuration	32
Branch configuration	32
Tag name	32
Tag pattern name	32
Revision name	33
Revision pattern	33
Build / tag synchronization	33

3.3. Change logs	33
3.3.1. Selection	33
3.3.2. Commits/revisions	34
3.3.3. Issues	35
4. File changes	36
5. Administration	38
5.1. Security	38
5.1.1. Concepts	38
5.1.2. Roles	38
Global roles	38
Project roles	39
5.1.3. Accounts	40
Built-in accounts	40
LDAP accounts	40
5.1.4. Account groups	40
5.1.5. General settings	40
5.1.6. Extending the security	41
5.2. LDAP setup	41
5.2.1. LDAP general setup	41
5.2.2. LDAP group mapping	42
5.3. Administration console	43
5.3.1. Managing running jobs	43
Filtering the jobs	43
Controlling the jobs	44
5.3.2. External connections	44
5.3.3. List of extensions	44
5.4. Application log messages	44
6. Integration	46
6.1. Integration with Jenkins	46
6.2. Monitoring	46
6.2.1. Health	46
6.2.2. Metrics	46
List of metrics	46
6.3. GraphQL support	48
6.4. Calling with Curl	48
6.5. Using the DSL	49
6.6. GraphiQL support	49
6.7. Extending the GraphQL schema	50
6.8. Encryption service	50
6.8.1. Selection of the confidential store	50
6.8.2. File confidential store	50

6.8.3. JDBC confidential store	50
6.8.4. Vault confidential store	51
6.8.5. Migrating encryption keys	51
6.8.6. Losing the encryption keys	52
6.8.7. Adding custom confidential store	52
6.9. Run info	52
6.9.1. Collection of run info	52
6.9.2. Displaying the run info	53
6.9.3. Exporting the run info	53
6.9.4. Exporting the run info to InfluxDB	53
7. DSL	55
7.1. DSL Usage	55
7.1.1. Embedded	55
7.1.2. Standalone shell	55
7.1.3. Connection	55
7.1.4. Retry mechanism	56
7.1.5. Calling the DSL	56
7.2. DSL Samples	57
7.2.1. DSL Security	57
Management of accounts	57
Account permissions	58
Management of account groups	58
Account group permissions	58
DSL LDAP mapping	59
7.2.2. DSL Images and documents	59
7.2.3. DSL Change logs	60
Getting the change log	60
Commits	61
Issues	61
Exporting the change log	62
File changes	62
7.2.4. DSL Branch template definitions	62
7.2.5. DSL SCM extensions	63
7.3. DSL Tool	63
7.4. DSL Reference	64
7.5. DSL Samples	64
8. Contributing	67
8.1. Branching strategy	67
8.2. Development	67
8.2.1. Environment set-up	67
Postgres set-up	67

8.2.2. Building locally	68
8.2.3. Launching the application from the command line	68
8.2.4. Launching the application in the IDE	68
8.2.5. Developing for the web	69
8.2.6. Running the tests	69
8.2.7. Integration with IDE	69
With Intellij	69
8.2.8. Delivery	69
Versioning	70
Deploying in production	70
8.2.9. Glossary	70
8.3. Architecture	71
8.3.1. Modules	71
8.3.2. UI	73
Resources	73
Resource decorators	73
8.3.3. Forms	73
Form object	74
Fields	74
Common field properties	74
help property	75
visibleIf property	75
text field	75
password field	76
memo field	76
email field	76
url field	76
namedEntries field	76
date field	77
yesno field	77
dateTime field	77
int field	77
selection field	77
multi-strings field	77
multi-selection field	77
multi-form field	78
Creating your custom fields	78
Form usage on the client	78
Fields rendering	78
8.3.4. Model	78
8.3.5. Model filtering	79

8.3.6. Jobs	79
Job architecture overview	79
Job registration	80
8.3.7. Encryption	81
8.3.8. Build filters	82
Usage	82
Implementation	83
8.3.9. Reference services	83
EntityDataStore	83
8.3.10. Technology	84
Client side	84
Server side	84
Layers	84
8.4. Testing	85
8.4.1. Running the unit and integration tests	85
8.4.2. Acceptance tests	85
On the command line	85
From the IDE	86
Standalone mode	86
8.4.3. Developing tests	87
Unit test context	87
Integration test context	87
8.5. Extending Ontrack	87
8.5.1. Preparing an extension	87
8.5.2. Extension ID	88
8.5.3. Coding an extension	89
8.5.4. Extension options	90
8.5.5. Writing tests for your extension	90
8.5.6. List of extension points	91
8.5.7. Running an extension	92
Using Gradle	92
8.5.8. Packaging an extension	92
8.5.9. Extension dependencies	92
8.5.10. Deploying an extension	92
Using the Docker image	93
Using the CentOS or Debian/Ubuntu package	93
In standalone mode	93
8.5.11. Extending properties	94
Java components	94
Web components	96
8.5.12. Extending decorators	97

Java components	97
Web components	98
8.5.13. Extending the user menu	99
Extension component	99
8.5.14. Extending pages	100
Extension menus	100
From the global user menu	100
From an entity page	100
Extension global settings	101
Extension page	101
Extension API	103
Extension API resource decorators	103
8.5.15. Extending event types	103
8.5.16. Extending GraphQL	104
Preparing the extension	104
Custom types	104
Root queries	106
Extra fields	107
Built-in scalar fields	108
Testing GraphQL	109
8.5.17. Extending cache	109
8.5.18. Extending metrics	110
8.5.19. Using Kotlin in extensions	111
8.5.20. Extending the settings	111
8.5.21. Extending the security	115
Adding functions	115
Adding roles	117
8.5.22. Extending confidential stores	119
9. Appendixes	120
9.1. Configuration properties	120
9.2. Deprecations and migration notes	121
9.2.1. Since 3.35	121
9.2.2. Since 2.28	121
9.2.3. Since 2.16	122
9.3. Roadmap	122
9.3.1. Use JPA / Hibernate for SQL queries	122
9.3.2. Using Neo4J as backend	122
9.3.3. Global DSL	122
9.3.4. Web hooks	122
9.4. Certificates	123
9.4.1. Registering a certificate in the JDK	123

9.4.2. Saving the certificate on MacOS	123
9.5. Metrics migration	123
9.6. Migration from H2 to Postgres	125
9.6.1. Prerequisites	125
9.6.2. Migration tool	125
9.6.3. Running the migration	125
9.6.4. Migration tool options	126
9.6.5. Secret files	126
9.7. Postgres and Flyway	126
9.8. Using production data	127
9.9. DSL Reference	128
9.9.1. Ontrack	128
9.9.2. AbstractProjectResource	136
9.9.3. AbstractResource	139
9.9.4. Account	140
9.9.5. AccountGroup	142
9.9.6. Admin	143
9.9.7. AuthenticationSource	147
9.9.8. Branch	147
9.9.9. Build	165
9.9.10. Config	172
9.9.11. Document	183
9.9.12. GroupMapping	184
9.9.13. LDAPSettings	185
9.9.14. PredefinedPromotionLevel	186
9.9.15. PredefinedValidationStamp	186
9.9.16. Project	187
9.9.17. ProjectEntityProperties	203
9.9.18. PromotionLevel	212
9.9.19. PromotionRun	216
9.9.20. SearchResult	217
9.9.21. ValidationRun	218
9.9.22. ValidationStamp	220

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 Postgres database.

1.2. Quick start

The fastest way to start Ontrack is to use [Docker Compose](#).

1.3. Postgres database

Unless you choose to deploy with [Docker Compose](#), you will need to have a Postgres database accessible by Ontrack.

Version 9.5 of Postgres has been tested successfully with Ontrack, but any later 9.x version should be OK as well.

No test with Postgres 10.x has been performed yet.

Ontrack will by default try to connect to `jdbc:postgresql://postgresql/ontrack`, using `ontrack / ontrack` as credentials.

Those parameters can be configured using normal Spring Boot JDBC [configuration](#), for example using arguments at startup:

```
--spring.datasource.url=jdbc:postgresql://localhost:5432/ontrack  
--spring.datasource.username=myuser  
--spring.datasource.password=password
```

See the [Spring Boot documentation](#) to see other and better ways to pass those [configuration](#) parameters.

1.4. Installing using Docker

Ontrack is distributed as a Docker image on the [Docker Hub](#), as [nemerosa/ontrack:3.35.10](#).

1.4.1. Overview

The Ontrack image exposes the ports [443](#) and [8080](#).

Two volumes are defined:

- [/var/ontrack/data](#) - contains some working files but also the log files.
- [/var/ontrack/conf](#) - contains the [configuration files](#) for Ontrack (see later).

Several modes of database setup can be done:

- [database persisted in volume](#)
- [using a H2 server database](#)

1.4.2. 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" \
...
```

Additional arguments to the Ontrack process, like [configuration arguments](#) passed on the command line, can use the [ONTRACK_ARGS](#) environment variable.

```
docker run \
...
--env "ONTRACK_ARGS=..."
...
```

1.4.3. Database persisted in volume

By default, the database JDBC URL is set to:

```
jdbc:h2:/var/ontrack/data/database/data;MODE=MYSQL;DB_CLOSE_ON_EXIT=FALSE;DEFFRAG_ALWAYS=TRUE
```



In order to put the database file on the host or in a data volume, share the `/var/ontrack/data` volume.



In this mode, the H2 database is NOT accessible while the Ontrack process runs. One will prefer to use the [H2 server mode](#).

1.4.4. Database as H2 server



This is the preferred mode when running in production mode.

One can run a H2 server by using Docker. For example:

```
docker container run -d --publish 9082:9082 --name h2 nemerosa/h2:latest
```



Running H2 as a Docker container is not at all required, and any other way is fine as long as the running database is accessible from the Ontrack process.

Then, just specify the H2 server address as argument. Using the database of the example above:

```
docker run \
...
--env "ONTRACK_DB_URL=jdbc:h2:tcp://h2:9082/ontrack;MODE=MYSQL"
...
```

1.4.5. Docker Compose deployment

Create the following file:

docker-compose.yml

```
version: "2.1"

services:
  # Ontrack container
  ontrack:
    image: nemerosa/ontrack:3
    environment:
      PROFILE: prod
    links:
      - "postgresql:postgresql"
    ports:
      - "8080:8080"

  # Postgresql database
  postgresql:
    image: postgres:9.5.2
    environment:
      POSTGRES_DB      : ontrack
      POSTGRES_USER    : ontrack
      POSTGRES_PASSWORD: ontrack
    ports:
      - "5432"
```

In the same directory, run:

```
docker-compose up -d
```

After some time, Ontrack becomes available at <http://localhost:8080>

1.5. 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

Directory	Description
/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 environment variables like `JAVA_OPTIONS` or `ONTRACK_DB_URL` (to use the H2 server mode). For example:

`/etc/default/ontrack`

```
JAVA_OPTIONS=-Xmx2048m
ONTRACK_DB_URL=jdbc:h2:tcp://h2:9082/ontrack;MODE=MYSQL
```

The `ONTRACK_ARGS` environment variable can be used to pass additional application parameters.

1.6. 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 environment variables like `JAVA_OPTIONS` or `ONTRACK_DB_URL` (to use the H2 server mode). For example:

[/etc/default/ontrack](#)

```
JAVA_OPTIONS=-Xmx2048m
ONTRACK_DB_URL=jdbc:h2:tcp://h2:9082/ontrack;MODE=MYSQL
```

The `ONTRACK_ARGS` environment variable can be used to pass additional application parameters.

1.7. 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`.

Options can be passed on the command line.



See the [Docker installation](#) section for information on how to connect to the database.

1.8. 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

Disabling branches after N (days)	60
Number of days of inactivity after a branch is disabled. 0 means that the branch won't ever be disabled automatically.	
Deleting branches after N (days) more	0
Number of days of inactivity after a branch is deleted, after it has been disabled automatically. 0 means that the branch won't ever be deleted automatically.	
Promotions to keep	x PRODUCTION
List of promotion levels which prevent a branch to be disabled or deleted	

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.

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

2.2.4. Validation stamp filters

When a branch defines many validation stamps, the view can become cluttered and not really useful any longer, because displaying too much information.

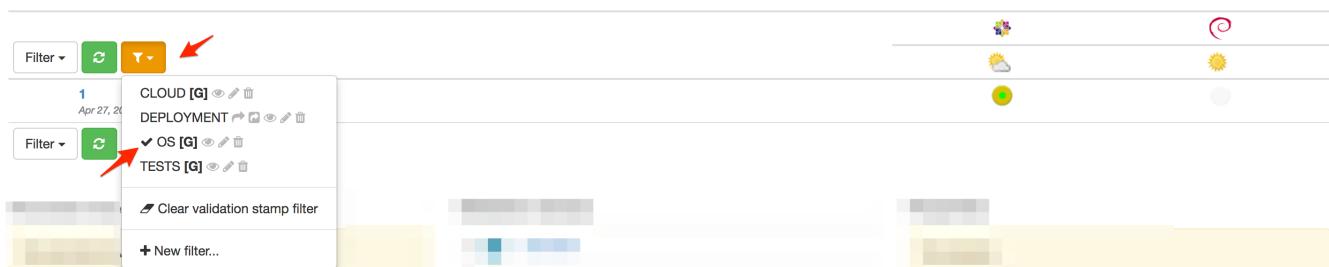
Validation stamp filters can be defined to restrict the view to a set of known validation stamps.

Using filters

Validation stamp filters can be selected in the branch view, just on the left of the list of validation stamp headers:



When a filter is selected, it is marked as such and only associated validation stamp columns are shown in the view:



The validation stamp filter menu is also marked in orange to indicate that a filter has been applied.

When the filter is applied, its name appears also in the URL. This can be used as a permalink:

[https://\[REDACTED\]/#/branch/1?vsFilter=OS](https://[REDACTED]/#/branch/1?vsFilter=OS)

You can remove the filter by selecting *Clear validation stamp filter* in the filter menu:



Editing filters



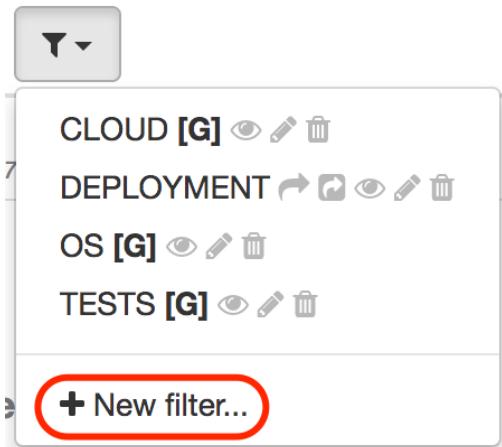
Only authorized users are allowed to edit the validation stamp filters for a branch.

A validation stamp filter is defined by:

- a name
- a list of validation stamp names to include

While it is possible to edit a filter using a dialog (see later), it is far easier to use the in-place editor.

Start by creating a validation stamp filter by selecting the *New Filter...* entry in the filter menu:



This displays a dialog to create the new filter:

Validation stamp filter

Name	DOCUMENTATION
<div style="text-align: right;">OK Cancel</div>	



Only the name is required and all current validation stamps filters are included by default.

When created, the filter can be directly edited in-place:

The following actions are possible:

- by clicking on the *Select none* button, no validation stamps is associated with the filter.
- by clicking on the *Select all* button, all validation stamps are associated with the filter.
- by clicking on the *Done with edition* button, the in-place edition stops and the normal display is resumed

You can also click on a validation stamp to remove it or to add it to the filter.

In case the validation stamp is associated with the filter, a *minus* icon appears close to its name. If it is not associated, the icon is dimmed and a plus icon appears:

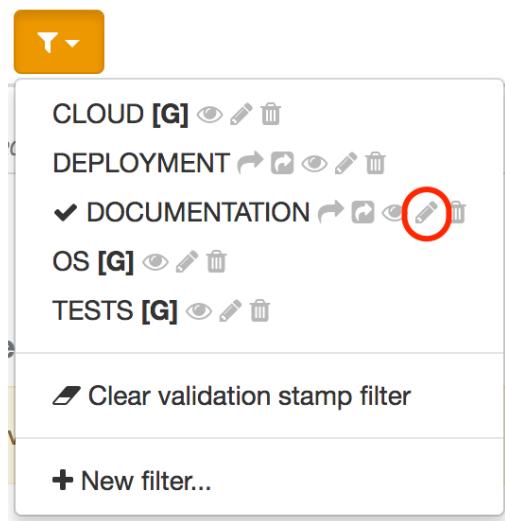
Note that you can also stop the edition by selecting the eye icon in the menu:

To start editing an existing filter, just click also on the eye icon close to its name:



Select any other filter, or removing the filter, will also stop the in-place edition.

To edit a filter directly, you can also select the pencil icon and edit the filter using a dialog:



This displays an edition dialog allowing to change the name and the list of validation stamps.



For a filter associated with a branch (see below, [sharing](#)), names can be selected among the validation stamps of the branch.



For a filter associated with a project, the list of validation stamps for *all* the branches is available.

For a *global* filter, names are no longer selected but must be edited.

Finally, to delete a filter, click on the trash icon:

CLOUD [G]
DEPLOYMENT
DOCUMENTATION [P]
OS [G]
TESTS [G]

+ New filter...



A confirmation will be asked before the deletion actually occurs.

Sharing

A filter is created by default at branch level and is only visible when the associated branch is displayed.

An authorized user can:

- share the filter at project level - in this case, the filter is available for all the branches of the project
- share the filter at global level - in this case, the filter is available for all projects and all branches

A filter shared at project level is shown with a [P] close to its name and a global filter with a [G]:

CLOUD [G]
DEPLOYMENT
✓ DOCUMENTATION [P]
OS [G]
TESTS [G]

✗ Clear validation stamp filter

+ New filter...

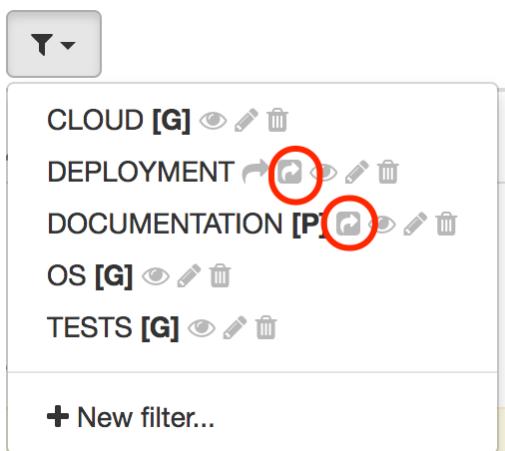
In the screenshot above:

- **DEPLOYMENT** is associated with the current branch
- **DOCUMENTATION** is associated with the project
- the other filters are global

To share a filter at project level, click on the share icon:



To share a filter at global level, click on the share icon:



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)

2.3.2. Bulk update of validation stamps

Validation stamps are attached to a [branch](#) but in reality, they are often duplicated in a project branches and among all the projects. Updating the description and the image of a validation stamp can fast become cumbersome.

The [predefined validation stamp](#) can mitigate but this won't solve the issue when validation stamps are created [automatically](#) even when not predefined.

In order to update all the validation stamps having the same name, across all branches and all projects, you can use the *Buld update* command in the validation stamp page:

Change image Update validation stamp Delete validation stamp Bulk update API Close

A confirmation will be asked and all the validation stamps having the same name, across all branches and all projects, will be updated with the same image and the same description.

A [predefined validation stamp](#) will also be updated or created.

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:

</> Change log ⏪ Previous build **Build links** ⏩ Promote ⏴ Validation run ⏵ Update build 🗑 Delete build </> API ✖ Close

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

Links

Project ontrack

Build db5eb4e

Project ontrack

Build 7bfc983

+ Add entry
?

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:

home ► test ►

master

Filter ▾

1

db5eb4e @ ontrack

7bfc98 @ ontrack

Jul 15, 2015 8:11 PM

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

Build 2

1 @ TEST

3 @ TEST

Information

The builds which are linked to a given build are displayed on the build page, in the information section:

Extra information

This build is used by...

[master-12345bdc @ test](#) [6.0.1 @ common-core](#)

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.

2.5.3. Run info

Builds can be associated with some [run info](#) which contains details about the source, the trigger and the duration of this build.

Information about the duration of the builds is shown just right of the build name in the branch page:

master

▼	3.25 May 5, 2018 11:37 AM - 2 minutes	
	3.24 May 5, 2018 11:37 AM	

or in the list of extensions in the build page:

Run information

- Source: [jenkins](#)
- Trigger: [user](#) [damien](#)
- Time: 2 minutes (139 s)

More details about run information at [Run info](#).

2.6. Managing validation runs

[Validation runs](#) associate a validation stamp to a build, and a status describing the validation: passed, failed, etc.

Additionally, a [run info](#) can be associated with the validation run to show information like a duration, a source or a trigger.

Validation runs can be created manually, but more often, they will be created automatically by a CI engine like [Jenkins](#).

Validation runs can be seen:

- in the branch overview - only their latest status is then visible
- in the branch overview, by clicking on a validation run (at the intersection of a validation stamp and a build), you can either create a new validation run (when there is none) or see the list of them and their statuses.

- in the build page
- in the validation stamp page

For one validation run, one can add comments and update the status to reflect (for example, to mention that a failure is under investigation).

2.7. Properties

All [entities](#) can be associated with properties.

2.7.1. Message property

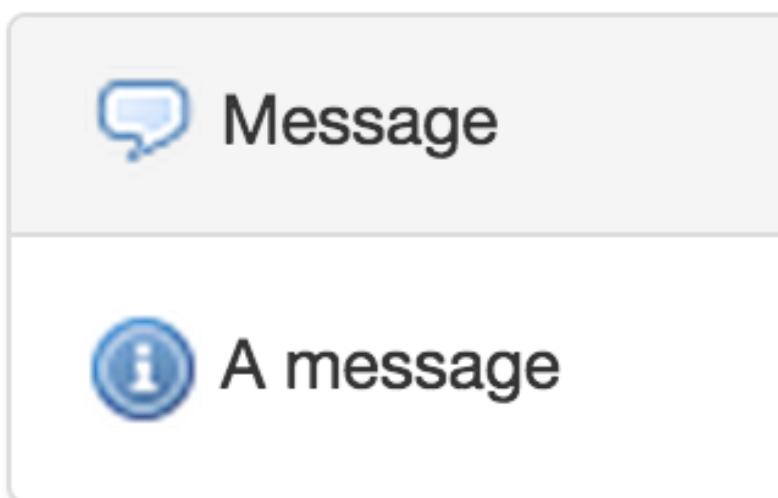
The *Message* property can be associated with any [entity](#) in Ontrack.

A message is the association of some *text* with a *type*: information, warning or error:

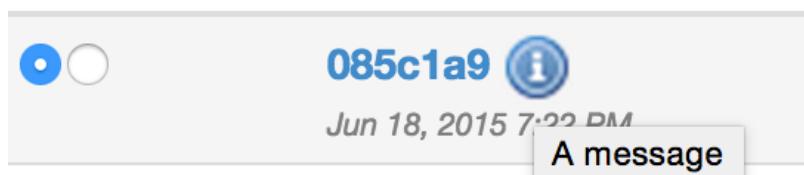
Message

Type	<input type="text" value="Info"/>
Type of message	
Text	<input type="text" value="A message"/>

The message is of course displayed in the list of properties:



i. and as a decoration:



2.7.2. Meta information property

Some arbitrary meta information properties can be associated with any [entity](#) in Ontrack, using a set of values for some names, and optionally some links and categories.

Input

Items	
Name	gradle
Value	2.4
Link	http://gradle.org
Category	dependency

Name	spring-boot
Value	1.2
Link	http://projects.spring.io/spring-boot
Category	dependency

[+ Add entry](#) [?](#)

Representation

The list of meta information properties is displayed in the list of properties:

Meta information		
dependency	gradle	2.4
dependency	spring-boot	1.2

Querying

The meta information 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:

- `name:` or `name:*` - all entities which contain a `name` meta information property
- `name:value` - all entities which contain a `name` meta information property with the exact `value`
- `name:val*` - all entities which contain a `name` meta information property whose value starts with `val`
- the `*` wildcard can be used in any place



Neither the link nor the category can be used for the search, only the name and the value.

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 nor 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
<code>override</code>	If set to <code>Yes</code> , the existing builds in Ontrack will be overridden by the tags in Subversion. If set to <code>No</code> (the default), the existing builds in Ontrack are never overridden and only new tags are taken into account.
<code>interval</code>	The frequency, in minutes, of the synchronization. If set to <code>0</code> , 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.

3.3.1. Selection

In the [branch view](#), you can select the two boundaries of the change log by:

- clicking on a build row
- clicking while holding the **shift** key on another build row

Two arrows indicate the current selection:

The screenshot shows a list of builds with their counts in brackets. The first build (4 [36]) has a red arrow pointing to it with the word "Click". The second build (2 [34]) also has a red arrow pointing to it with the word "Click". Between the two, the word "Shift" is written above the arrow pointing to build 2.

Build Number	Count	Date
5	[37]	May 10, 2017 8:22 PM
4	[36]	May 10, 2017 8:21 PM
3	[35]	May 10, 2017 7:55 PM
2	[34]	May 10, 2017 7:26 PM
1	[11]	Apr 27, 2017 7:43 PM



Note that by default the first and the last build of the current view are selected as 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](#)).

3.3.2. Commits/revisions

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

The screenshot shows a list of commits with their hash codes, descriptions, and timestamps. A red arrow points to the first commit (34e39ab). At the top right, there are buttons for 'Issues' and 'File changes'.

Commit Hash	Description	Author	Date
34e39ab	Unicity of the SVN config	Damien Coraboeuf	Jul 22, 2016 8:53 PM
b65cf37	#438 Forces builds on master	Damien Coraboeuf	Jul 22, 2016 7:41 PM
90a8d60	Merge branch 'master' into feature/441-configuration	Damien Coraboeuf	Jul 22, 2016 11:26 AM
b3ccf17d	Using a file repository for SVN integration tests - stabilisation	Damien Coraboeuf	Jul 21, 2016 11:34 AM
3b210a3	#441 Values in a file	Damien Coraboeuf	Jul 19, 2016 2:28 PM
faa83e5	#441 Value options as a map	Damien Coraboeuf	Jul 19, 2016 2:22 PM

3.3.3. Issues

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

Issues						Commits	File changes	Export
ID	State	Title	Milestone	Assignee	Updated	Labels		
#429	closed	Add a security general option to force authentication	2.22		Jun 24, 2016	feature		
#434	closed	Changing a BitBucket configuration does not update the indexing jobs	2.22		Jul 1, 2016	bug git job		
#437	closed	Configurable home page	2.23		Jul 7, 2016	feature performance		
#438	open	Using a slave for the build			Jul 20, 2016	delivery		
#439	closed	Integration tests for extensions	2.23	dcoraboeuf	Jul 11, 2016	enhancement extension		
#440	closed	Job scheduler useable for one-shot jobs	2.23	dcoraboeuf	Jul 12, 2016	enhancement		
#441	closed	Ontrack provisioning and configuration	2.24	dcoraboeuf	Jul 22, 2016	documentation feature		

Chapter 4. File changes

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

This screenshot shows the 'File changes' interface. At the top, there is a search bar with an 'ANT-like pattern' input field containing '.gitignore'. Below the search bar is a list of file changes, each with a small icon, a file name, and a download icon. The files listed are: .gitignore, buildSrc/build.gradle, buildSrc/src/main/groovy/net/nemerosa/ontrack/gradle/extension/OntrackExtension.groovy, buildSrc/src/main/groovy/net/nemerosa/ontrack/gradle/extension/OntrackExtensionPlugin.groovy, buildSrc/src/main/resources/extension/gulpfile.js, buildSrc/src/main/resources/extension/package.json, and docker/Dockerfile.

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:

This screenshot shows the 'File changes' interface with a red box highlighting the 'ANT-like pattern' input field. The field contains '**/*.java'. Below the input field is a list of Java files from the 'ontrack-extension-api' module, each with a small icon, a file name, and a download icon. The files listed are: ExtensionManager.java, ConfigurationHealthIndicator.java, ConfigurationNotFoundException.java, ConfigurationService.java, ConfigurationServiceListener.java, and ConfigurationValidationException.java.

For more complex selections, you can click 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

Create file change filter

This screenshot shows the 'Create file change filter' dialog box. It has two main sections: 'Name' and 'Filter(s)'. The 'Name' section contains a text input field with 'Main' and a descriptive note below it. The 'Filter(s)' section contains a text input field with '**/main/**' and a note below it. At the bottom right are 'OK' and 'Cancel' buttons.

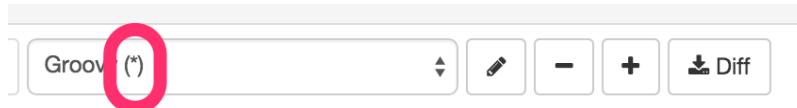
Name	Main
Name to use to save the filter.	
Filter(s)	**/main/**
List of ANT-like patterns (one per line).	

OK Cancel

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.

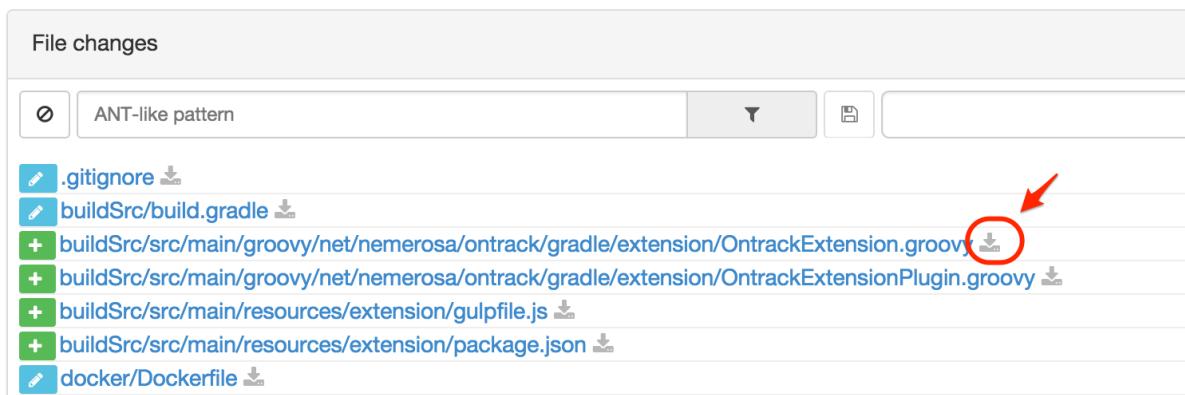
Finally, you can get the unified diff for the selected filter by clicking on the *Diff* button:



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*.

[Extensions](#) can contribute to built-in roles and functions - see [Extending the security](#) for details.

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 branches (creation / edition / deletion) and 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 going 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*.

5.1.6. Extending the security

[Extensions](#) can extend the security model beyond what is defined in the Ontrack core. See [Extending the security](#) for more details.

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:

Enable LDAP authentication Yes No

URL	<input type="text" value="https://ldap.nemerosa.com:636"/>
URL to the LDAP server. For example: https://ldap.nemerosa.com:636	
User	<input type="text" value="ldap_ontrack"/>
Name of the user used to connect to the LDAP server.	
Password	<input type="text" value="...."/>
Password of the user used to connect to the LDAP server.	
Search base	<input type="text" value="dc=nemerosa,dc=com"/>
Query to get the user. For example: dc=nemerosa,dc=com	
Search filter	<input type="text" value="(sAMAccountName={0})"/>
Filter on the user query. {0} will be replaced by the user name. For example: (sAMAccountName={0})	
Full name attribute	<input type="text" value="displayName"/>
Name of the attribute that contains the full name of the user. Defaults to cn	
Email attribute	<input type="text" value="mail"/>
Name of the attribute that contains the email of the user. Defaults to email	
Group attribute	<input type="text"/>
Name of the attribute that contains the groups the user belongs to. Defaults to memberOf)	
Group filter	<input type="text"/>
Name of the OU field used to filter groups a user belongs to (optional).	



The settings shown above are suitable to use with an Active 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](#)
- 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

Administration console											
Tools for the general management of ontrack											
Status		Jobs		Log							
Any status		Git		Any type							
ID	Category	Type	Description	State	Action	Schedule	Run count	Last duration	Error(s)	Last run	Next run
4	Git	Git build synchronisation	Branch 1 av & result			Every 30 minutes	1	a few seconds	-	3/4/16 7:42 PM	3/4/16 8:12 PM
2	Git	Git indexation	https://github.com/nemerosa/ontrack.git (github.com @ github)			Every 30 minutes	1	a few seconds	-	3/4/16 7:42 PM	3/4/16 8:12 PM
3	Git	Git indexation	https://github.com/nemerosa/ontrack.git (github.com @ github)			Every 30 minutes	1	a few seconds	-	3/4/16 7:42 PM	3/4/16 8:12 PM

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 screenshot shows the 'Log entries' page with the following interface elements:

- Header:** home > Log entries. A 'Close' button is in the top right corner.
- Page Title:** Log entries. Subtitle: List of application log messages.
- Navigation:** Buttons for 'Previous' and 'Next', a 'Refresh log' button, and a 'Delete all entries' button.
- Filtering:** Input fields for 'After' and 'Before' dates, 'Authentication' and 'Free text' filters, and 'Filter' and 'Reset filter' buttons.
- Table:** A table listing log entries. The columns are:
 - Level:** Shows 'ERROR' for all entries.
 - Timestamp:** Shows dates from Saturday, December 10, 2016, at 7:54:00 PM to 7:43:06 PM.
 - Type:** Shows '[svn][svn-indexation]' repeated for each entry.
 - Information:** Shows 'SVN indexation from latest for test' repeated for each entry.
 - Details:** Shows 'Details...' buttons for each entry.

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:

ERROR

Timestamp
Saturday, December 10, 2016 7:54:00 PM

Type
[svn][svn-indexation] SVN Indexation

Information
SVN indexation from latest for test2

Details

- job.key: test2
- job.progress:

Stacktrace

```
net.nemerosa.ontrack.extension.svn.client.SVNClientException: Problem while
accessing Subversion: org.tmatesoft.svn.core.SVNException: svn: E175002: unknown
host
svn: E175002: OPTIONS request failed on '/'
at
net.nemerosa.ontrack.extension.svn.client.SVNClientImpl.translateSVNException(SVN
ClientImpl.java:398)
at
```

OK

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



Since version 2.35 / 3.35, Ontrack uses the [Micrometer](#) framework to manage metrics, in order to allow a better integration with [Spring Boot 2](#).

See [Metrics migration](#) for information about the migration.

By default, Ontrack supports two external registries for metrics:

- InfluxDB
- Prometheus

The export to those engine is *disabled* by default and must be enabled explicitly.

For example, for Prometheus, you can use the `management.metrics.export.prometheus.enabled` property or the `MANAGEMENT_METRICS_EXPORT_PROMETHEUS_ENABLED` environment variable.

For the rest of the configuration, you have to consult the Spring Boot or Micrometer documentation.

List of metrics



The list of Ontrack specific metrics and their tags and values is available using the `/manage/ontrack_metrics` endpoint. Note that this endpoint needs authentication and some [administrator privileges](#).

General metrics:

- `ontrack_error` (counter) - number of error (the `type` tag contains the type of error)

Statistics about the `objects` stored by Ontrack:

- `ontrack_entity_project_total` (gauge) - total number of projects
- `ontrack_entity_branch_total` (gauge) - total number of branches
- `ontrack_entity_build_total` (gauge) - total number of builds
- `ontrack_entity_promotionLevel_total` (gauge) - total number of promotion levels
- `ontrack_entity_promotionRun_total` (gauge) - total number of promotion runs
- `ontrack_entity_validationStamp_total` (gauge) - total number of validation stamps
- `ontrack_entity_validationRun_total` (gauge) - total number of validation runs
- `ontrack_entity_validationRunStatus_total` (gauge) - total number of validation run statuses
- `ontrack_entity_property_total` (gauge) - total number of properties
- `ontrack_entity_event_total` (gauge) - total number of events

General metrics about `jobs`:

- `ontrack_job_count_total` (gauge) - total number of jobs
- `ontrack_job_running_total` (gauge) - total number of running jobs
- `ontrack_job_error_total` (gauge) - total number of jobs in error
- `ontrack_job_paused_total` (gauge) - total number of paused jobs
- `ontrack_job_disabled_total` (gauge) - total number of disabled jobs
- `ontrack_job_invalid_total` (gauge) - total number of invalid jobs
- `ontrack_job_error_count_total` (gauge) - total number of errors among all the jobs

Information about individual jobs:

- `ontrack_job_duration_ms` (timer) - duration of the execution of the job
- `ontrack_job_run_count` (counter) - number of times a job has run
- `ontrack_job_error_count` (counter) - number of errors for this job

They all have the following tags:



- `job-category` - category of the job
- `job-type` - type of the job
- `job-id` - ID of the job

Run information:

- `ontrack_run_build_time_seconds` (timer) - duration of a run for a build. It is associated with `project` and `branch` tags.
- `ontrack_run_validation_run_time_seconds` (timer) - duration of a run for a validation run. It is

associated with `project`, `branch`, `validation_stamp` and `status` tags.

More details at [Run info](#).

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:

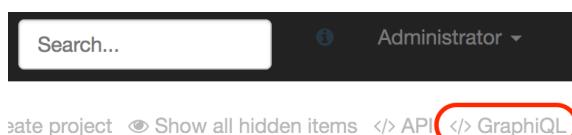
```
def result = ontrack.graphQLQuery(  
    '''{  
        projects (id: $projectId) {  
            id  
            branches {  
                id  
                name  
            }  
        }  
    }'''',  
    [  
        projectId: 10,  
    ]  
)  
  
assert result.errors != null && result.errors.empty  
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:

A screenshot of the GraphiQL IDE. On the left, there is a code editor window containing a GraphQL query:

```
1+ {  
2     projects {  
3         name  
4     }  
5 }
```

On the right, the results of the query are displayed in a JSON-like format:

```
+ {  
+   "errors": [],  
+   "data": {  
+     "projects": [  
+       {  
+         "name": "bitbucket"  
+       },  
+       {  
+         "name": "git"  
+       },  
+       {  
+         "name": "github"  
+       },  
+       {  
+         "name": "gitlab"  
+       },  
+       {  
+         "name": "svn"  
+       }  
+     ]  
+   }  
+ }
```



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.

6.8. Encryption service

Secrets used by Ontrack are [encrypted](#) using keys managed by a [ConfidentialStore](#).

Ontrack provides three types of storage:

- file based storage (default)
- [Vault](#) storage
- database storage

If needed, you can also create your own form of storage using [extensions](#).

6.8.1. Selection of the confidential store

The selection of the confidential store is done at startup time using the [ontrack.config.key-store configuration property](#).

It defaults to [file](#) (see below).

Additional configuration properties might be needed according to the type of store.

6.8.2. File confidential store

This is the default store but its selection can be made explicit by setting the [ontrack.config.key-store configuration property](#) to [file](#).

This store will store the keys in the [working directory](#) under the [security/secrets](#) subfolder.

A [master.key](#) file is used to encrypt the individual keys themselves, so two files will be typically present:

- [master.key](#)
- [net.nemerosa.ontrack.security.EncryptionServiceImpl.encryption](#)

6.8.3. JDBC confidential store

This store manages the keys directly in the Ontrack database. It can be selected by setting the [ontrack.config.key-store configuration property](#) to [jdbc](#).



This store is intrinsically insecure since it stores the keys in the same location where the secrets are themselves stored.

No further configuration is needed.

6.8.4. Vault confidential store

By setting the `ontrack.config.key-store` configuration property to `vault`, Ontrack will use `Vault` to store its encryption keys.

Following configuration properties are available to configure the connection to Vault:

Property	Default	Description
<code>ontrack.config.vault.uri</code>	<code>http://localhost:8200</code>	URI to the Vault end point
<code>ontrack.config.vault.token</code>	<code>test</code>	Token authentication
<code>ontrack.config.vault.prefix</code>	<code>secret/ontrack/key</code>	Path prefix for the storage of the keys WARNING: As of now, the support for Vault storage is experimental and is subject to change in later releases. In particular, the authentication mechanism might change.

6.8.5. Migrating encryption keys

In the event you want to migrate the encryption keys from one type of storage to another, follow this procedure.



In the procedure below, `ONTRACK_URL` designates the Ontrack URL and `ONTRACK_ADMIN_USER` the name of an Ontrack user which has the `ADMINISTRATOR` role.

Using the initial configuration for the store, start by exporting the key:

```
curl ${ONTRACK_URL}/admin/encryption \
--user ${ONTRACK_ADMIN_USER} \
--output ontrack.key
```

This command will export the encryption key into the local `ontrack/key` file.

Start Ontrack using the new configuration.



There might be errors at startup, when some `jobs` start to collect some data from the external applications. Those errors can be safely ignored for now.

Import the key file into Ontrack:

```
curl ${ONTRACK_URL}/admin/encryption \
--user ${ONTRACK_ADMIN_USER} \
-X PUT \
-H "Content-Type: text/plain" \
--data @ontrack.key
```

Restart Ontrack.

6.8.6. Losing the encryption keys

In case you lose the encryption keys, the consequence will be that the secrets stored by Ontrack won't be able to be decrypted. This will typically make the external applications your Ontrack instance connects to unreachable.

The only to fix this is to reenter the secrets.



Some pages might not display correctly if some applications are not reachable.

6.8.7. Adding custom confidential store

See [Extending confidential stores](#).

6.9. Run info

Builds and validation runs can be associated with some *run information* which contains:

- source of the information, like a Jenkins job
- trigger of the information, like a SCM change
- duration of the collection for the information (like the duration of a job)

6.9.1. Collection of run info

Run info can be attached to a build or a validation run using the REST API or the DSL of Ontrack.

This is typically done at CI engine level, where a solution like the [Ontrack Jenkins plugin](#) simplifies the operation.

When using the Jenkins pipeline as code, the `ontrackBuild` and `ontrackValidate` steps will do this automatically, so nothing to change. For example:

```

post {
    success {
        ontrackBuild project: "xxx", branch: "1.0", build: version
    }
}

```

When using the DSL, the run info must be specified explicitly. The Jenkins plugin provides a `jenkins.runInfo` binding which contains some run info ready to be passed:

```

ontrackScript script: """
def b = ontrack.build(...)
b.runInfo = jenkins.runInfo
"""

```

6.9.2. Displaying the run info

The run info is displayed in the branch overview and the build page for builds, and in the validation stamp and the validation run pages for the validation runs.

It is of course available through the REST API, GraphQL and the DSL.

6.9.3. Exporting the run info

While the run info is available from Ontrack, it can also be exported to other databases.

As of today, only InfluxDB is supported.

6.9.4. Exporting the run info to InfluxDB



This is an experimental feature. In the future, especially when migrating to Spring Boot 2.0, the configuration might change. The feature is very likely to stay though.

In order to export Ontrack run info as points into an InfluxDB database, following elements must be configured:

Property	Environment variable	Default	Description
<code>ontrack.influxdb.enabled</code>	<code>ONTRACK_INFLUXDB_ENABLED</code>	<code>false</code>	Enables the export of run info to InfluxDB
<code>ontrack.influxdb.uri</code>	<code>ONTRACK_INFLUXDB_URI</code>	<code>"http://localhost:8086"</code>	URI of the InfluxDB database
<code>ontrack.influxdb.username</code>	<code>ONTRACK_INFLUXDB_USERNAME</code>	<code>"root"</code>	User name to connect to the InfluxDB database
<code>ontrack.influxdb.password</code>	<code>ONTRACK_INFLUXDB_PASSWORD</code>	<code>"root"</code>	Password to connect to the InfluxDB database

Property	Environment variable	Default	Description
ontrack.influxdb.db	ONTRACK_INFLUXDB_DB	"ontrack"	Name of the InfluxDB database
ontrack.influxdb.create	ONTRACK_INFLUXDB_CREATE	true	If true , the database is created at startup
ontrack.influxdb.ssl.host-check	ONTRACK_INFLUXDB_SSL_HOST_CHECK	true	If false , disables host checking for certificates. This should not be used for a production system!
ontrack.influxdb.run-info	ONTRACK_INFLUXDB_RUN_INFO	true	If true , the run info is exported to InfluxDB (true by default)

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 [appendices](#).

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. Branching strategy

- development of features always goes to `feature/` branches
- the branches are created from:
 - the `master` branch for version 2
 - the `v3` branch for version 3
- new releases are created by branching from `develop` (V2) or `master` (V3), using `release/2.'` or `'release/3.` as a prefix
- pull requests must be made from the `master` branch

The versioning is automated using the [Gradle Versioning plug-in](#). No file needs to be updated to set the version.

8.2. Development

8.2.1. 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

Postgres set-up

Starting from release 3, Ontrack uses a Postgresql database for its backend.

By default, the application will try to access it at `jdbc:postgresql://localhost:5432/ontrack` using `ontrack / ontrack` as credentials.

You can of course set the database yourself, but the best way is to run the following Gradle command:

```
./gradlew devInit
```

This will use Docker in order to set up a Postgres database container, exposing the port **5432** on the host and named **postgresql**.

Following properties are available to personalize this setup:

Property	Default	Description
devPostgresPort	5432	Port the container must expose on the host
devPostgresName	postgresql	Name of the container to create

For example, to use the **5555** port:

```
./gradlew devInit -PdevPostgresPort=5555
```



This container is used only for your local development, and is not used for running the [integration tests](#), where another container is automatically created and destroyed.

8.2.2. Building locally

```
./gradlew clean build
```

To launch the integration tests or acceptance tests, see [Testing](#).

8.2.3. Launching the application from the command line

Just run:

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

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



The **dev** profile is activated by default and the working files and database file will be available under **ontrack-ui/work**.

8.2.4. 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>

8.2.5. 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.

8.2.6. Running the tests

See [Testing](#).

8.2.7. Integration with IDE

With IntelliJ

- install the Lombok plugin
- in [Build, Execution, Deployment > Compiler > Annotation Processors](#), check [Enable annotation processing](#)

8.2.8. 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.

8.2.9. 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

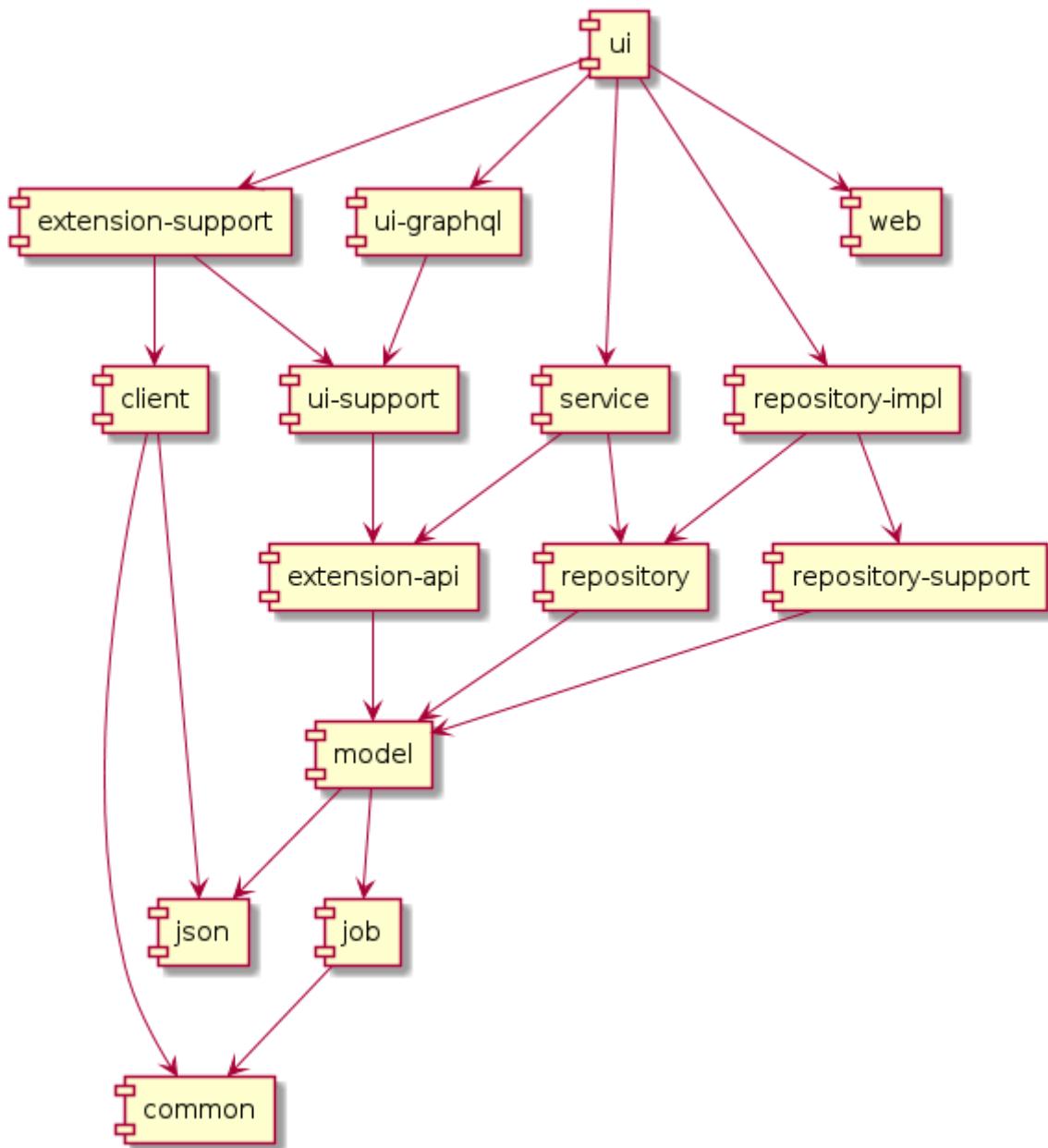
A resource is a model object decorated with links that allow the client side to interact with the API following the HATEOAS principle. More than just providing access to the model structure, those links reflect the actual use cases and the corresponding navigation. In particular, the links are driven 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.

8.3. Architecture

8.3.1. 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.

8.3.2. 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.

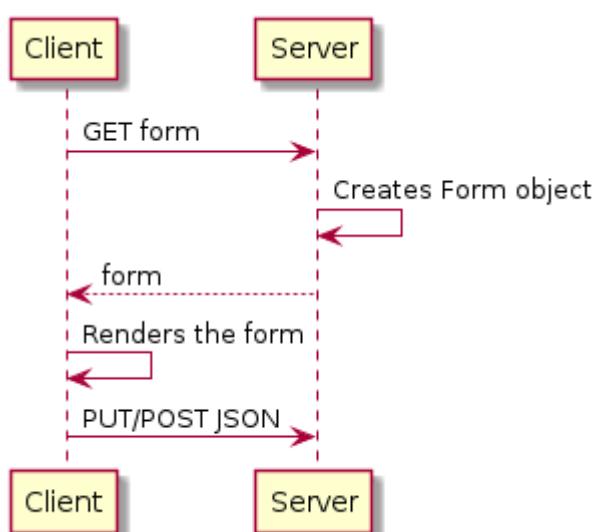
Resource decorators

TODO

8.3.3. 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
----------	--------	---------------	-------------

<code>name</code>	<code>of(...)</code>	<code>required</code>	Mapping
<code>label</code>	<code>label(...)</code>	<code>none</code>	Display name
<code>required</code>	<code>optional()</code>	<code>true</code>	Is the input required?
<code>readOnly</code>	<code>readOnly()</code>	<code>false</code>	Is the input read-only?
<code>validation</code>	<code>validation(...)</code>	<code>none</code>	Message to display if the field content is deemed invalid
<code>help</code>	<code>help(...)</code>	<code>none</code>	Help message to display for the field (see below for special syntax)
<code>visibleIf</code>	<code>visibleIf(...)</code>	<code>none</code>	Expression which defines if the field is displayed or not - see below for a detailed explanation
<code>value</code>	<code>value(...)</code>	<code>none</code>	Value to associate 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="text"/>` form field.

Name

Property	Method	Default value	Description
<code>length</code>	<code>length(...)</code>	<code>300</code>	Maximum length for the text
<code>regex</code>	<code>regex(...)</code>	<code>.*</code>	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:

List of links	Home	http://nemerosa.github.io/ontrack/	
	GitHub	https://github.com/nemerosa/ontrack	
Name	Link		

List of links associated with a name.

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.

Property	Method	Default value	Description
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

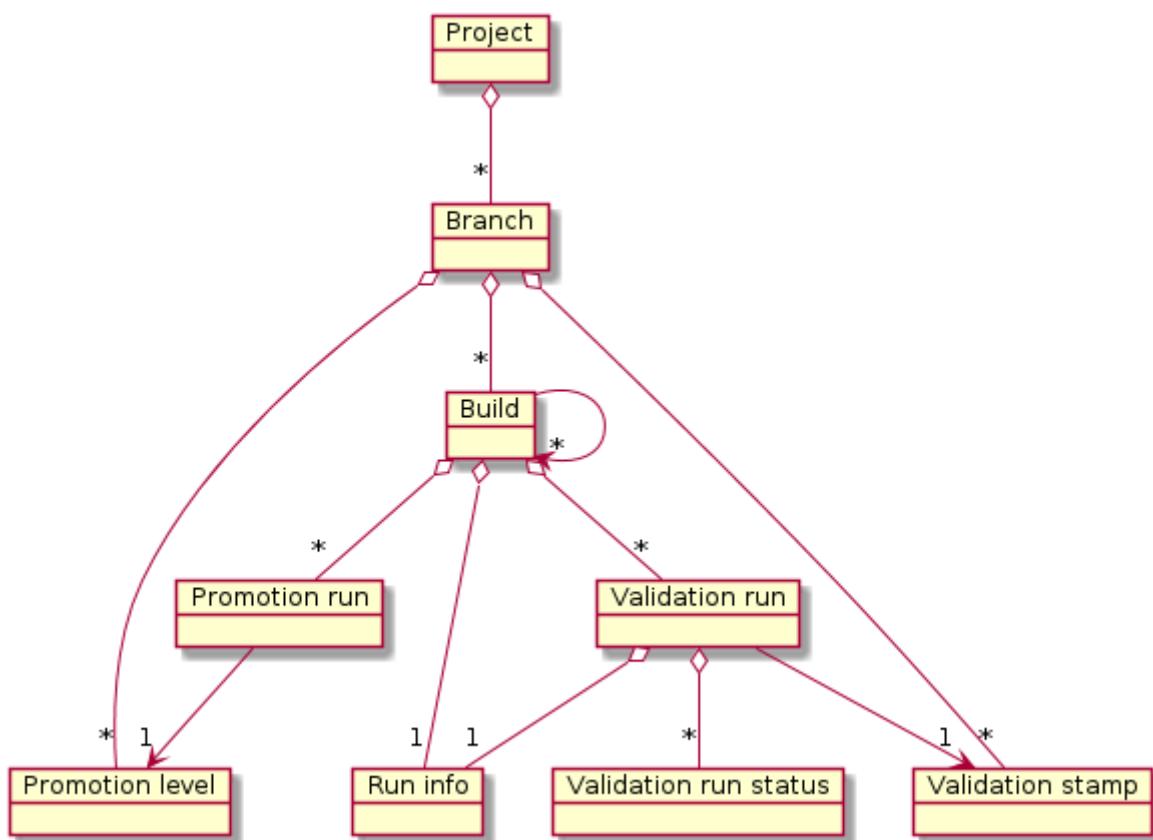
TODO

Fields rendering

TODO

8.3.4. 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.

Builds and validation runs can be attached to some "run info" which gives additional information like the duration of the build or the validation.

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

8.3.5. Model filtering

TODO

8.3.6. 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 = CATEGORY.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](#).

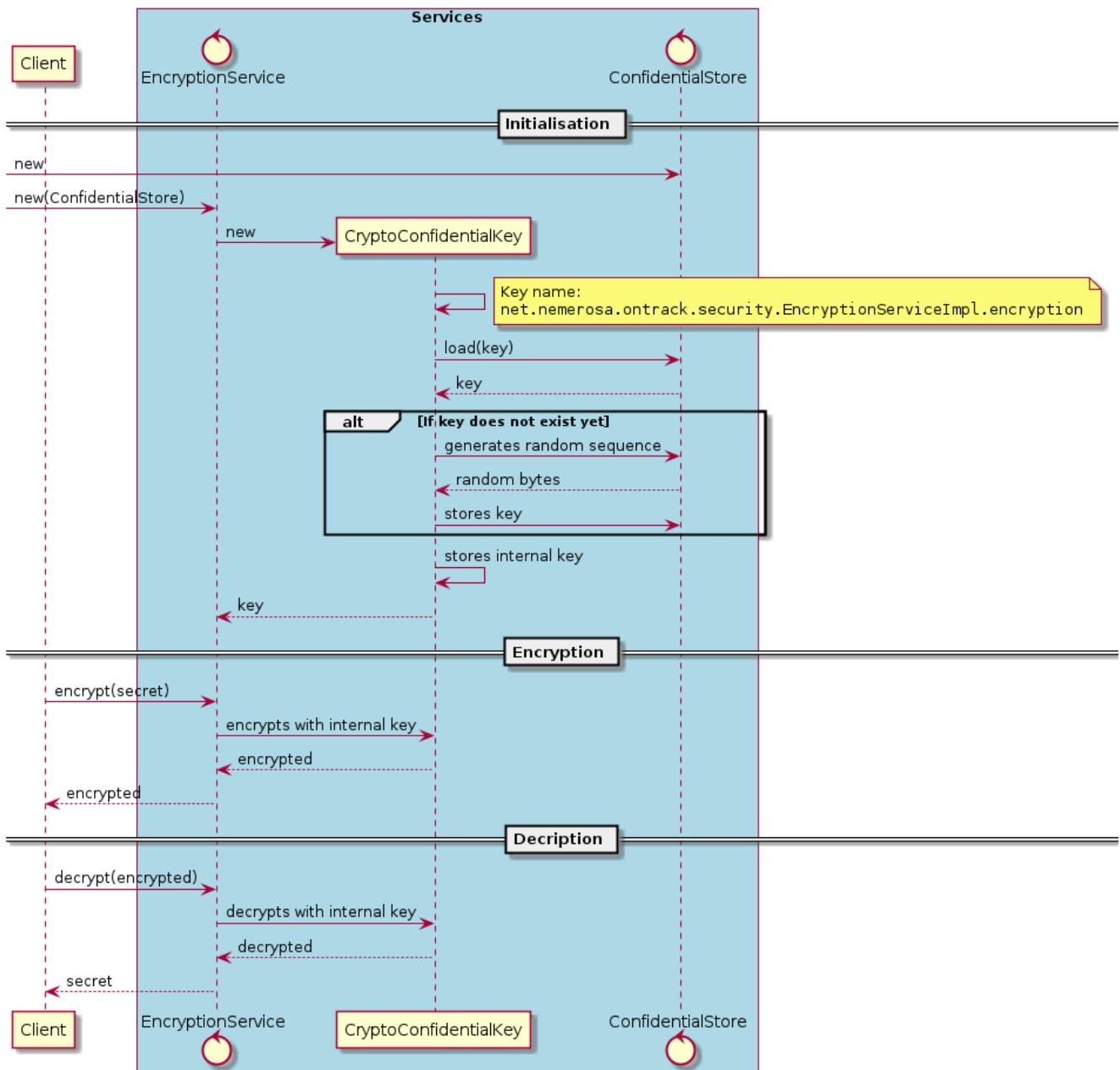
8.3.7. Encryption

Ontrack will store secrets, typically passwords and tokens, together with the configurations needed to connect to external applications: Git, Subversion, JIRA, etc.

In order to protect the integrity of those external applications, those secrets must be protected.

Ontrack does so by encrypting those secrets in the database, using the [AES128](#) algorithm. The [EncryptionService](#) is used for encryption.

The key needed for the encryption is stored and retrieved using a [ConfidentialStore](#) service.



See [Encryption service](#) for more details about using a confidential store.

8.3.8. 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.

8.3.9. 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
<code>StructureService</code>	Access to projects, branches and all entities
<code>SecurityService</code>	Checks the current context for authorizations
<code>PropertyService</code>	Access to the properties of the entities
<code>EntityDataService</code>	This service allows to store and retrieve arbitrary data with entities
<code>EntityDataStore</code>	This service allows to store audited and indexed data with entities . See EntityDataStore for more information.

`EntityDataStore`

The `EntityDataStore` is a service which allows [extensions](#) to store some data associated with [entities](#) with the following properties:

- data stored as JSON
- data always associated with an [entity](#)
- indexation by category and name, not necessarily unique
- grouping of data using a group ID
- unique generated numeric ID

- audit data - creation + updates

See the Javadoc for `net.nemerosa.ontrack.repository.support.store.EntityDataStore` for more details.

Example:

```
@Autowired  
EntityDataStore store;  
@Autowired  
SecurityService securityService;  
  
Branch branch = ...  
  
store.add(branch, "Category", "Name", securityService.getCurrentSignature(), null,  
json);
```

8.3.10. Technology

Client side

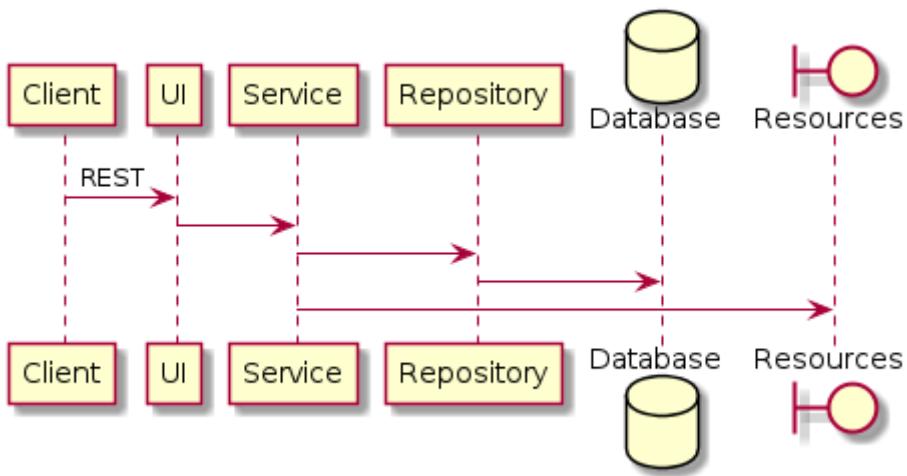
One page only, pure AJAX communication between the client and the server.

- 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



8.4. Testing

- *unit tests* are always run and should not access resources or 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.

8.4.1. 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.

8.4.2. 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 - defaults to <http://localhost:8080>
 - `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.acceptance.url=<url>` to specify the `<url>` where `ontrack` is deployed. It defaults to <http://localhost:8080>
- `--option.acceptance.admin=<password>` to specify the administrator password. It defaults to `admin`.
- `--option.acceptance.context=<context>` can be specified several time to define the context(s) the acceptance tests are running in (like `--option.acceptance.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, in `build/acceptance/ontrack-acceptance.xml`.

The directory can be changed using the `ontrack.acceptance.output-dir` argument or system property and defaults to `build/acceptance`.

The JUnit result file name can be changed using the `ontrack.acceptance.result-file-name` argument or system property and defaults to `ontrack-acceptance.xml`.

8.4.3. 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.

8.5. 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".

8.5.1. Preparing an extension

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



The use of [Kotlin](#) 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. [Kotlin](#) is very well supported.
- 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)

8.5.2. 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'  
}
```

8.5.3. 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)

8.5.4. 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)
        );
    }
}
```

8.5.5. 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())  
        );  
    }  
}
```

8.5.6. List of extension points

Ontrack provides the following extension points:

- [Properties](#) - allows to attach a property to an [entity](#)
- [Decorators](#) - allows to display a decoration for an [entity](#)
- [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.
- [Encryption key store](#) - allows to define a custom store for the [encryption keys](#).
- **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.

8.5.7. Running an extension



A Postgres database must be available to run an extension, since it is needed by Ontrack.

See the [development section](#) to see how quickly set it up.

Using Gradle

To run your extension using Gradle:

```
./gradlew ontrackRun
```

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.

8.5.8. Packaging an extension

Just run:

```
./gradlew clean build
```

The extension is available as a JAR (together with its transitive dependencies, see below) in `build/dist`.

8.5.9. Extension dependencies

If your extension depends on dependencies which are not brought by Ontrack, you have to collect them explicitly and put them in the same directory which contain your main JAR file.

The Ontrack plug-in provides an `ontrackPrepare` task which copies all dependencies (transitively) and the main JAR in the `build/dist` directory.

This task is called by the main `build` task.

8.5.10. 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>
# Overrides the extension directory, as to NOT use a volume
ENV EXTENSIONS_DIR /var/ontrack/your-extension
# Copies the extensions in the target volume
COPY extensions/*.jar /var/ontrack/your-extension/
```

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
```



When using a [child Dockerfile](#), the extension directory has to be customized because we cannot use the `VOLUME` in this case.

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>
```

8.5.11. 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 new MyProperty(
        replacementFunction.apply(value.getValue())
    );
}

```

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
```

8.5.12. 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, My.PropertyType.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>
```

8.5.13. 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.

8.5.14. 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](#) 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, My.PropertyType.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

8.5.15. 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()
);
```

8.5.16. 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 {
    testCompile "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. For example, `Map` and `Collection` types are not supported.

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.

8.5.17. Extending cache

Ontrack uses `Caffeine` to cache some data in memory to avoid reloading it from the database. The cache behaviour can be configured using `properties`.

Extensions can also use the Ontrack cache and make it configurable.

In order to declare one or several caches, just declare a `Component` which implements `CacheConfigExtension` and set the `Caffeine spec` string for each cache.

```

@Component
class MyCacheConfigExtension : CacheConfigExtension {
    override val caches: Map<String, String>
        get() = mapOf(
            "myCache" to "maximumSize=1000,expireAfterWrite=1h,recordStats"
        )
}

```



The cache statistics are available as [metrics](#) if the `recordStats` flag is set.

The cache thus declared become configurable through [external configuration](#). For example:

application.yml

```

ontrack:
    config:
        cache:
            specs:
                myCache: "maximumSize=2000,expireAfterWrite=1d,recordStats"

```

In order to use the cache in the code, you can just use the [Spring cache annotations](#). For example:

```

@Service
class MyServiceImpl: MyService {
    @Cacheable(cacheNames = "myCache")
    fun getValue(id: Int): MyObject = ...
}

```

8.5.18. Extending metrics



Starting from version 2.35, the metrics framework used by Ontrack has been migrated to [Micrometer](#). This is a breaking change - and the way metrics can be contributed to by extensions is totally different and some effort must be done in the migration.

In order for extensions to add their own metrics, they can interact directly with an inject `MeterRegistry` and then get gauges, timers, counters, etc.

Or they can create some `MeterBinder` beans to register some gauges at startup time.

Usually, migrating (monotonic) counters and timers will be straightforward:

```

val meterRegistry: MeterRegistry
meterRegistry.counter("...", tags).increment()
meterRegistry.timer("...", tags).record...

```

For gauge, you have to register them so that they can be call at any time by the meter registry:

```
val meterRegistry: MeterRegistry
meterRegistry.gauge("...", tags,
    sourceObject,
    { obj -> ... }
)
```

See the [Micrometer](#) documentation for more information on how to register metrics.

8.5.19. Using Kotlin in extensions

An [extension](#) can use [Kotlin](#) additionally to Java.

Just mention `kotlin()` in the Ontrack configuration in your `build.gradle` file:

build.gradle

```
ontrack {
    kotlin()
    ...
}
```

The Kotlin Gradle plug-in will be automatically applied and the Kotlin JVM for JRE8, with the same version than for Ontrack, will be added in `compileOnly` mode to your dependencies. Enjoy!

8.5.20. Extending the settings

An [extension](#) can add a entry in the list of global settings.

Start by creating an invariant class which contains the data to manage in the new settings.



in the sample below, we use some PuppetDB connection settings, which need a URL, a user name and a password.

```
@Data
public class PuppetDBSettings {
    private final String url;
    private final String username;
    private final String password;
}
```

The settings are managed in Ontrack by two distinct services:

- a manager - responsible for the edition of the settings
- a provider - responsible for retrieving the settings



as-of today, the service cannot be the same class.

To define the manager, extend the `AbstractSettingsManager` class and use your settings class as a parameter:

```
@Component
public class PuppetDBSettingsManager extends AbstractSettingsManager<PuppetDBSettings>
{

    private final SettingsRepository settingsRepository;
    private final EncryptionService encryptionService;

    @Autowired
    public PuppetDBSettingsManager(CachedSettingsService cachedSettingsService,
        SecurityService securityService, SettingsRepository settingsRepository,
        EncryptionService encryptionService) {
        super(PuppetDBSettings.class, cachedSettingsService, securityService);
        this.settingsRepository = settingsRepository;
        this.encryptionService = encryptionService;
    }

    @Override
    protected void doSaveSettings(PuppetDBSettings settings) {
        settingsRepository.setString(PuppetDBSettings.class, "url",
            settings.getUrl());
        settingsRepository.setString(PuppetDBSettings.class, "username",
            settings.getUsername());
        settingsRepository.setPassword(PuppetDBSettings.class, "password",
            settings.getPassword(), false, encryptionService::encrypt);
    }

    @Override
    protected Form getSettingsForm(PuppetDBSettings settings) {
        return Form.create()
            .with(
                Text.of("url")
                    .label("URL")
                    .help("URL to the PuppetDB server. For example:
http://puppetdb")
                    .value(settings.getUrl())
            )
            .with(
                Text.of("username")
                    .label("User")
                    .help("Name of the user used to connect to the
PuppetDB server.")
                    .optional()
                    .value(settings.getUsername())
            )
            .with(

```

```

        Password.of("password")
            .label("Password")
            .help("Password of the user used to connect to the
PuppetDB server.")
            .optional()
            .value("") // Password never sent to the client
    );
}

@Override
public String getId() {
    return "puppetdb";
}

@Override
public String getTitle() {
    return "PuppetDB settings";
}
}

```

To define the provided, implement the `AbstractSettingsManager` and use your settings class as a parameter:

```

@Component
public class PuppetDBSettingsProvider implements SettingsProvider<PuppetDBSettings> {

    private final SettingsRepository settingsRepository;
    private final EncryptionService encryptionService;

    @Autowired
    public PuppetDBSettingsProvider(SettingsRepository settingsRepository,
EncryptionService encryptionService) {
        this.settingsRepository = settingsRepository;
        this.encryptionService = encryptionService;
    }

    @Override
    public PuppetDBSettings getSettings() {
        return new PuppetDBSettings(
            settingsRepository.getString(PuppetDBSettings.class, "url", ""),
            settingsRepository.getString(PuppetDBSettings.class, "username", ""),
            settingsRepository.getPassword(PuppetDBSettings.class, "password", "",
encryptionService::decrypt)
        );
    }

    @Override
    public Class<PuppetDBSettings> getSettingsClass() {
        return PuppetDBSettings.class;
    }
}

```

That's all there is to do. Now, the new settings will automatically appear in the *Settings* page:

PuppetDB settings

The screenshot shows a form titled 'PuppetDB settings'. It contains three fields: 'URL' with a blurred value, 'User' set to 'admin', and 'Password' with three asterisks. Below the form is a blue 'Edit' button.

URL	[REDACTED]
User	admin
Password	***

Edit

and can be edited using the form defined above:

PuppetDB settings

URL	<input type="text" value="https://puppetdb-infra.clear2pay.com"/>
URL to the PuppetDB server. For example: https://puppetdb-infra.clear2pay.com	
User	<input type="text" value="admin"/>
Name of the user used to connect to the PuppetDB server.	
Password	<input type="password"/>
Password of the user used to connect to the PuppetDB server.	

8.5.21. Extending the security

The [security model](#) of Ontrack can be extended to fit for specific needs in extensions.

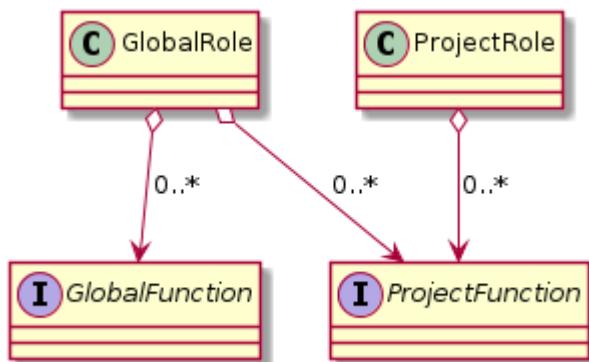
Adding functions

All authorizations in the code are granted through *functions*. We distinguish between:

- *global* functions about Ontrack in general
- *project* functions linked to a given project

[Global roles](#) are then linked to a number of global functions and project functions.

On the other hand, [project roles](#) can only be linked to project functions.



The association of *core* functions and *core* roles is fixed in the Ontrack core, but extensions can:

- define new global and project functions
- assign them to existing roles



For security reasons, extensions cannot associate *existing core* functions to roles.

In order to define a *global function*, just define an interface which extends [GlobalFunction](#):

```
public interface MyGlobalFunction extends GlobalFunction {}
```

Almost the same thing for a *project function*:

```
public interface MyProjectFunction extends ProjectFunction {}
```



No method is to be implemented.

Now, you can link those functions to existing roles by providing a **RoleContributor** component. In our example, we want to grant the global function and the project function to the **AUTOMATION** global role and the project function to the **PROJECT_OWNER** project role.

```
@Component
public MyRoleContributor implements RoleContributor {
    @Override
    public Map<String, List<Class<? extends GlobalFunction>>>
getGlobalFunctionContributionsForGlobalRoles() {
        return Collections.singletonMap(
            Roles.GLOBAL_AUTOMATION,
            Collections.singletonList(
                MyGlobalFunction.class
            )
        );
    }
    @Override
    public Map<String, List<Class<? extends ProjectFunction>>>
getProjectFunctionContributionsForGlobalRoles() {
        return Collections.singletonMap(
            Roles.GLOBAL_AUTOMATION,
            Collections.singletonList(
                MyProjectFunction.class
            )
        );
    }
    @Override
    public Map<String, List<Class<? extends ProjectFunction>>>
getProjectFunctionContributionsForProjectRoles() {
        return Collections.singletonMap(
            Roles.PROJECT_OWNER,
            Collections.singletonList(
                MyProjectFunction.class
            )
        );
    }
}
```



All available roles are listed in the **Roles** interface.

You can now check for those functions in your code by injecting the **SecurityService**:

```
private final SecurityService securityService;  
...  
if (securityService.isGlobalFunctionGranted(MyGlobalFunction.class)) {  
    ...  
}  
if (securityService.isProjectFunctionGranted(project, MyProjectFunction.class)) {  
    ...  
}
```

or:

```
private final SecurityService securityService;  
...  
securityService.checkGlobalFunction(MyGlobalFunction.class));  
securityService.checkProjectFunction(project, MyProjectFunction.class))
```



The project functions can be tested on a **Project** or any other **entity** which belongs to a project (branches, builds, etc.).

Adding roles

Both *global* and *project roles* can be added using the same **RoleContributor** extension class, by overriding the following methods:

```

@Component
public MyRoleContributor implements RoleContributor {
    @Override
    public List<RoleDefinition> getGlobalRoles() {
        return Collections.singletonList(
            new RoleDefinition(
                "MY_GLOBAL_ROLE",
                "My Global Role",
                "This is a new global role"
            )
        );
    }
    @Override
    public List<RoleDefinition> getProjectRoles() {
        return Collections.singletonList(
            new RoleDefinition(
                "MY_PROJECT_ROLE",
                "My Project Role",
                "This is a new project role"
            )
        );
    }
}

```

A new role can inherit from a built-in role:

i

```

@Override
public List<RoleDefinition> getProjectRoles() {
    return Collections.singletonList(
        new RoleDefinition(
            "MY_PROJECT_ROLE",
            "My Project Role",
            "This is a new project role",
            Roles.PROJECT_PARTICIPANT
        )
    );
}

```

In the previous example, the `MY_PROJECT_ROLE` will inherit from all functions of the built-in `PARTICIPANT` role.

Same principle applies for global roles.

Those roles becomes eligible for selection when managing accounts and groups.

Note that functions (built-in or contributed) can be associated to those new roles - see [Adding functions](#). By default, no function is associated to a contributed role.

8.5.22. Extending confidential stores

Extensions can define a custom [confidential store](#) used to store encryption keys.

Create a component which extends the [AbstractConfidentialStore](#) class:

```
@Component
@ConditionalOnProperty(name = OntrackConfigProperties.KEY_STORE, havingValue =
"custom")
public class CustomConfidentialStore extends AbstractConfidentialStore {

    public CustomConfidentialStore() {
        LoggerFactory.getLogger(CustomConfidentialStore.class).info(
            "[key-store] Using custom store"
        );
    }

    @Override
    public void store(String key, byte[] payload) throws IOException {
        // ...
        // Stores the key
    }

    @Override
    public byte[] load(String key) throws IOException {
        // ...
        // Retrieves the key or ...
        return null;
    }
}
```

Note the use of the [ConditionalOnProperty](#), which allows to select this store when the [ontrack.config.key-store](#) property is set to [custom](#).

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 delay greater than half
# the period of the job. Setting 0 would actually disable the scattering altogether.
ontrack.config.jobs.scattering-ratio = 1.0

# Confidential store for the encryption keys
ontrack.config.key-store = file

# Cache configuration
# Caffeine spec strings per cache type
# See http://static.javadoc.io/com.github.ben-manes/caffeine/2.6.0/com/github/benmanes/caffeine/cache/CaffeineSpec.html
# For example, for the 'properties' cache:
ontrack.config.cache.specs.properties =
maximumSize=1000,expireAfterWrite=1d,recordStats

```

9.2. Deprecations and migration notes

9.2.1. Since 3.35

`StructureService` deprecated method:

- The `getValidationRunsForBuildAndValidationStamp(net.nemerosa.ontrack.model.structure.ID, net.nemerosa.ontrack.model.structure.ID)` method is deprecated and should be replaced by `getValidationRunsForBuildAndValidationStamp(net.nemerosa.ontrack.model.structure.ID, net.nemerosa.ontrack.model.structure.ID, int, int)`.

GraphQL schema deprecations:

- `Build.linkedFrom` is now deprecated and must be replaced by either `uses` or `usedBy`

9.2.2. 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.

9.2.3. 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. Use JPA / Hibernate for SQL queries

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

9.3.2. 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

9.3.3. 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](#).

9.3.4. 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](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. Metrics migration

Since version 2.35 / 3.35, Ontrack uses the [Micrometer](#) framework to manage metrics, in order to allow a better integration with [Spring Boot 2](#).



This means that *old metrics are no longer supported* and that any tool / dashboard using the old keys must be adapted.

See also [Metrics](#).

Old metric key	New metric key	Tags
gauge.entity.project	ontrack_entity_project_total	-
gauge.entity.branch	ontrack_entity_branch_total	-
gauge.entity.build	ontrack_entity_build_total	-
gauge.entity.promotionLevel	ontrack_entity_promotionLevel_total	-
gauge.entity.promotionRun	ontrack_entity_promotionRun_total	-
gauge.entity.validationStamp	ontrack_entity_validationStamp_total	-
gauge.entity.validationRun	ontrack_entity_validationRun_total	-
gauge.entity.validationRunStatus	ontrack_entity_validationRunStatus_total	-
gauge.entity.property	ontrack_entity_property_total	-
gauge.entity.event	ontrack_entity_event_total	-
gauge.jobs	ontrack_job_count_total	-
gauge.jobs.running	ontrack_job_running_total	-
gauge.jobs.disabled	ontrack_job_disabled_total	-
gauge.jobs.error	ontrack_job_error_total	-
gauge.jobs.invalid	ontrack_job_invalid_total	-
gauge.jobs.paused	ontrack_job_paused_total	-
gauge.jobs.<category>	n/a	-
gauge.jobs.<category>.running	n/a	-
gauge.jobs.<category>.disabled	n/a	-
gauge.jobs.<category>.error	ontrack_job_error_count	category tag
gauge.jobs.<category>.invalid	n/a	-
gauge.jobs.<category>.paused	n/a	-
ontrack.job.count	n/a	-
ontrack.job.running	n/a	-
ontrack.job.disabled	n/a	-
ontrack.job.error	n/a	-
ontrack.job.invalid	n/a	-
ontrack.job.paused	n/a	-

Old metric key	New metric key	Tags
counter.error	ontrack_error	-
counter.error.<type>	ontrack_error	<type> as `type` tag



If a metric marked as *n/a* (not available) is still needed, either create an [extension](#) to add it or create an issue to have it added.

9.6. Migration from H2 to Postgres

Starting from version 3, Ontrack uses Postgres instead of H2 for its back-end.

9.6.1. Prerequisites

Before you actually start the migration, please make sure you have the following elements:

- a copy of the H2 database file to migrate (typically a `data.mv.db` file) and the associated credentials
- an access to the Postgres database to migrate to and the associated credentials
- a copy of the secret files



For a Docker installation, the database is located at `/var/ontrack/data/database/data.mv.db` and the secret files in `/var/ontrack/data/files/security/secrets`.



For a RPM or Debian installation, the database is located at `/usr/lib/ontrack/database/data.mv.db` and the secret files in `/usr/lib/ontrack/files/security/secrets`.

9.6.2. Migration tool

The migration tool is part of the Ontrack release and can be downloaded as [ontrack-postgresql-migration.jar](#) in the [GitHub release page](#).

9.6.3. Running the migration

Create a directory, called `#{MIGRATION}` in the rest of this section, and

- copy the `data.mv.db` database file in this directory
- copy the `ontrack-postgresql-migration.jar` migration tool in this directory

Run the following command:

```
java -jar ontrack-postgresql-migration.jar
```

By default, the tool will look for the H2 database file in the current directory, using `ontrack / ontrack`

as credentials.

By default, the tool will use a Postgres database located at `jdbc:postgresql://localhost:5432/ontrack`, using `ontrack / ontrack` as credentials.

You can change those default values using the configuration options below.

9.6.4. Migration tool options

Migration options can either be specified on the command line or by creating a local `application.properties` file.

Option	Default value	Description
<code>ontrack.migration.cleanup</code>	<code>false</code>	If set to <code>true</code> , all previous tables will be deleted . This forces a migration from scratch.
<code>ontrack.migration.skip-events</code>	<code>false</code>	If set to <code>true</code> , the events will not be migrated. This can speed up the migration but can have side effects since some entities will not be associated with their creation event.
<code>ontrack.migration.h2.url</code>	<code>jdbc:h2:/data;MODE=MYSQL;DB_CLOSE_ON_EXIT=FALSE;DEFRAK_AWALYS=TRUE</code>	JDBC URL to the H2 database
<code>ontrack.migration.h2.username</code>	<code>ontrack</code>	User used to connect to the H2 database
<code>ontrack.migration.h2.password</code>	<code>ontrack</code>	Password used to connect to the H2 database
<code>ontrack.migration.postgresql.url</code>	<code>jdbc:postgresql://localhost:5432/ontrack</code>	JDBC URL to the Postgres database
<code>ontrack.migration.postgresql.username</code>	<code>ontrack</code>	User used to connect to the Postgres database
<code>ontrack.migration.postgresql.password</code>	<code>ontrack</code>	Password used to connect to the Postgres database

9.6.5. Secret files

The `master.key` and `net.nemerosa.ontrack.security.EncryptionServiceImpl.encrypted` files must be copied and put in the [correct place](#).



For a development environment, put those files in `work/files/security/secrets`, relatively to the workspace root.

9.7. Postgres and Flyway

The creation and updates of the Ontrack schema in Postgres is managed using [Flyway](#).

The configuration is initialized in the `net.nemerosa.ontrack.repository.RepositoryConfig` class and SQL files are stored in `ontrack-database/src/main/resources/ontrack/sql`.



As of now, `extensions` cannot contribute to the schema.

The actual migration is processed using the `net.nemerosa.ontrack.service.support.StartupStrategy` component. Once the database has been upgraded, all `StartupService` implementations are started in their specified order.

9.8. Using production data

Some times, when `developing` Ontrack or working with `extensions`, it might be useful to copy the production data locally.

Make sure to have `pg_dump` and `psql` tools available in your working environment.



They are part of the Postgres installation but can be downloaded individually from the [download page](#).

To export the data from the *source* database in a local `ontrack.sql` file:

```
pg_dump --dbname ontrack --host <source-host> --port 5432 \
--username <source-user> > ontrack.sql
```

To import this data into the *target* database:

```
psql --dbname ontrack --host <target-host> --port 5432 \
--username <target-user> < ontrack.sql
```



the target database must be *empty* (no table, no sequences).

When the migration of data is done, do not forget to also copy the secret files and to put them in the correct location.



For a `Docker` installation, the secret files are in `/var/ontrack/data/files/security/secrets`.



For a `RPM` or `Debian` installation, the secret files are in `/usr/lib/ontrack/files/security/secrets`.



In the `development environment`, the secret files are in `work/files/security/secrets`, relatively to the workspace root..

9.9. DSL Reference

Class summary		
Class		
Ontrack	-	Methods
AbstractProjectResource		Methods
AbstractResource		Methods
Account		Methods
AccountGroup		Methods
Admin		Methods
AuthenticationSource		Methods
Branch	Properties	Methods
Build	Properties	Methods
Config		Methods
Document		Methods
GroupMapping		Methods
LDAPSettings		
PredefinedPromotionLevel		Methods
PredefinedValidationStamp		Methods
Project	Properties	Methods
ProjectEntityProperties		Methods
PromotionLevel	Properties	Methods
PromotionRun		Methods
SearchResult		Methods
ValidationRun		Methods
ValidationStamp		Methods

9.9.1. Ontrack

An Ontrack instance is usually bound to the `ontrack` identifier and is the root for all DSL calls.

Method summary	
Method	Description
<code>branch</code>	<code>Branch branch(String project, String branch)</code> Looks for a branch in a project. Fails if not found.

<code>build</code>	<code>Build build(String project, String branch, String build)</code>
	Looks for a build by name. Fails if not found.
<code>configure</code>	<code>Object configure(Closure closure)</code>
	Configures the general settings of Ontrack. See Config .
<code>delete</code>	<code>Object delete(String url)</code>
	Runs an arbitrary DELETE request for a relative path and returns JSON
<code>download</code>	<code>Document download(String url)</code>
	Downloads an arbitrary document using a relative path.
<code>findProject</code>	<code>Project findProject(String name)</code>
	Finds a project using its name. Returns null if not found.
<code>get</code>	<code>Object get(String url)</code>
	Runs an arbitrary GET request for a relative path and returns JSON
<code>getAdmin</code>	<code>Admin getAdmin()</code>
	Access to the administration of Ontrack
<code>getConfig</code>	<code>Config getConfig()</code>
	Access to the general configuration of Ontrack
<code>getProjects</code>	<code>List<Project> getProjects()</code>
	Gets the list of projects
<code>getVersion</code>	<code>String getVersion()</code>
	Gets the version of the remote Ontrack server
<code>graphQLQuery</code>	<code>Object graphQLQuery(String query, Map variables = [:])</code>
<code>post</code>	<code>Object post(String url, Object data)</code>
	Runs an arbitrary POST request for a relative path and some data, and returns JSON
<code>project</code>	<code>Project project(String name, String description = '', Closure closure)</code>
	Finds or creates a project, and configures it.

<code>project</code>	<code>Project project(String name, String description = '')</code> Finds or creates a project.
<code>promotionLevel</code>	<code>PromotionLevel promotionLevel(String project, String branch, String promotionLevel)</code> Looks for a promotion level by name. Fails if not found.
<code>put</code>	<code>Object put(String url, Object data)</code> Runs an arbitrary PUT request for a relative path and some data, and returns JSON
<code>search</code>	<code>List<SearchResult> search(String token)</code> Launches a global search based on a token.
<code>text</code>	<code>Object text(String url)</code> Runs an arbitrary GET request for a relative path and returns text
<code>upload</code>	<code>Object upload(String url, String name, Object o, String contentType)</code> Uploads some typed data on a relative path and returns some JSON
<code>upload</code>	<code>Object upload(String url, String name, Object o)</code> Uploads some arbitrary binary data on a relative path and returns some JSON. See upload .
<code>validationStamp</code>	<code>ValidationStamp validationStamp(String project, String branch, String validationStamp)</code> Looks for a validation stamp by name. Fails if not found.

branch

`Branch branch(String project, String branch)`

Looks for a branch in a project. Fails if not found.

See: [Branch](#)

build

`Build build(String project, String branch, String build)`

Looks for a build by name. Fails if not found.

See: [Build](#)

Sample:

```
def build = ontrack.build('prj', 'master', '1.0.0-12')
assert build.name == '1.0.0-12'
```

configure

`Object configure(Closure closure)`

Configures the general settings of Ontrack. See [Config](#).

delete

`Object delete(String url)`

Runs an arbitrary DELETE request for a relative path and returns JSON

download

`Document download(String url)`

Downloads an arbitrary document using a relative path.

See: [Document](#)

findProject

`Project findProject(String name)`

Finds a project using its name. Returns null if not found.

See: [Project](#)

Sample:

```
def project = ontrack.findProject('unexisting')
assert project == null
```

get

`Object get(String url)`

Runs an arbitrary GET request for a relative path and returns JSON

getAdmin

`Admin getAdmin()`

Access to the administration of Ontrack

See: [Admin](#)

getConfig

`Config getConfig()`

Access to the general configuration of Ontrack

See: [Config](#)

getProjects

`List<Project> getProjects()`

Gets the list of projects

See: [Project](#)

Sample:

```
def projects = ontrack.projects  
assert projects instanceof Collection
```

getVersion

`String getVersion()`

Gets the version of the remote Ontrack server

graphQLQuery

```
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.

post

```
Object post(String url, Object data)
```

Runs an arbitrary POST request for a relative path and some data, and returns JSON

project

```
Project project(String name, String description = '', Closure closure)
```

Finds or creates a project, and configures it.

See: [Project](#)

Sample:

```
def project = ontrack.project('my-project', 'My project') {  
    // Configuring the project  
}  
assert project.name == 'my-project'
```

project

```
Project project(String name, String description = '')
```

Finds or creates a project.

See: [Project](#)

Sample:

```
def project = ontrack.project('my-project', 'My project')  
assert project.name == 'my-project'
```

promotionLevel

```
PromotionLevel promotionLevel(String project, String branch, String promotionLevel)
```

Looks for a promotion level by name. Fails if not found.

See: [PromotionLevel](#)

put

```
Object put(String url, Object data)
```

Runs an arbitrary PUT request for a relative path and some data, and returns JSON

search

```
List<SearchResult> search(String token)
```

Launches a global search based on a token.

See: [SearchResult](#)

text

```
Object text(String url)
```

Runs an arbitrary GET request for a relative path and returns text

upload

```
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

upload

```
Object upload(String url, String name, Object o)
```

Uploads some arbitrary binary data on a relative path and returns some JSON. See [upload](#).

validationStamp

```
ValidationStamp validationStamp(String project, String branch, String validationStamp)
```

Looks for a validation stamp by name. Fails if not found.

See: [ValidationStamp](#)

9.9.2. AbstractProjectResource

See also: [AbstractResource](#)

Method summary	
Method	Description
<code>config</code>	<code>Object config(Closure closure)</code> Configures this entity.
<code>delete</code>	<code>Object delete()</code> Deletes this entity.
<code>getDecoration</code>	<code>Object getDecoration(String type)</code> Gets the data for the first decoration of a given type. If no decoration is available, returns null.
<code>getDecorations</code>	<code>List<> getDecorations(String type)</code> Returns the list of decoration data (JSON) for a given decoration type.
<code>getDecorations</code>	<code>List<> getDecorations()</code> Returns the list of decorations for this entity. Each item has a <code>decorationType</code> type name and <code>data</code> as JSON.
<code>getDescription</code>	<code>String getDescription()</code> Returns any description attached to this entity.
<code>getId</code>	<code>int getId()</code> Returns the numeric ID of this entity.
<code>getJenkinsJobDecoration</code>	<code>String getJenkinsJobDecoration()</code> Gets the Jenkins decoration for this entity.
<code>getMessageDecoration</code>	<code>Map<String> getMessageDecoration()</code> Gets any message for this entity
<code>getName</code>	<code>String getName()</code> Returns the name of this entity.
<code>getProperty</code>	<code>Object getProperty(String type, boolean required)</code>
<code>property</code>	<code>Object property(String type)</code>
<code>property</code>	<code>Object property(String type, Object data)</code> Sets the value for a property of this entity. Prefer using dedicated DSL methods.

config

`Object config(Closure closure)`

Configures this entity.

delete

`Object delete()`

Deletes this entity.

getDecoration

`Object getDecoration(String type)`

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

getDecorations

`List<> getDecorations(String type)`

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

getDecorations

`List<> getDecorations()`

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

getDescription

`String getDescription()`

Returns any description attached to this entity.

getId

`int getId()`

Returns the numeric ID of this entity.

getJenkinsJobDecoration

`String getJenkinsJobDecoration()`

Gets the Jenkins decoration for this entity.

getMessageDecoration

`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` if no message is associated with this entity.

Sample:

```
def project = ontrack.project('prj')
def branch = ontrack.branch('prj', 'test')
def build = ontrack.build('prj', 'test', '1')
assert project.messageDecoration == null
assert branch.messageDecoration == null
assert build.messageDecoration == null
project.config.message('Information', 'INFO')
assert project.messageDecoration == [type: 'INFO', text: 'Information']
branch.config.message('Warning', 'WARNING')
assert branch.messageDecoration == [type: 'WARNING', text: 'Warning']
build.config.message('Error', 'ERROR')
assert build.messageDecoration == [type: 'ERROR', text: 'Error']
```

getName

`String getName()`

Returns the name of this entity.

getProperty

`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
```

property

```
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')
```

property

```
Object property(String type, Object data)
```

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

9.9.3. AbstractResource

Method summary

Method	Description
<code>getNode</code>	<code>Object getNode()</code> Gets the internal JSON representation of this resource.
<code>getPage</code>	<code>String getPage()</code> Gets the Web page address for this resource.
<code>link</code>	<code>String link(String name)</code> Gets a link address.
<code>optionalLink</code>	<code>String optionalLink(String name)</code> Gets a link address if it exists.

getNode

```
Object getNode()
```

Gets the internal JSON representation of this resource.

getPage

`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')`.

link

`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'))
```

optionalLink

`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.

9.9.4. Account

Representation of a user account.

See also: [AbstractResource](#)

Method summary

Method	Description
<code>getAccountGroups</code>	<code>List<AccountGroup> getAccountGroups()</code> List of groups this account belongs to.

<code>getAuthenticationSource</code>	<code>AuthenticationSource getAuthenticationSource()</code> Source of the account: LDAP, built-in, ...
<code>getEmail</code>	<code>String getEmail()</code> Email for the account.
<code>getFullName</code>	<code>String getFullName()</code> Display name for the account.
<code>getId</code>	<code>int getId()</code> Unique ID for the account.
<code>getName</code>	<code>String getName()</code> User name, used for signing in.
<code>getRole</code>	<code>String getRole()</code> Role for the user: admin or not.

getAccountGroups

`List<AccountGroup> getAccountGroups()`

List of groups this account belongs to.

See: [AccountGroup](#)

getAuthenticationSource

`AuthenticationSource getAuthenticationSource()`

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

See: [AuthenticationSource](#)

getEmail

`String getEmail()`

Email for the account.

getFullName

`String getFullName()`

Display name for the account.

getId`int getId()`

Unique ID for the account.

getName`String getName()`

User name, used for signing in.

getRole`String getRole()`

Role for the user: admin or not.

9.9.5. AccountGroup

Account group. Just a name and a description.

See also: [AbstractResource](#)

Method summary

Method	Description
<code>getDescription</code>	<code>String getDescription()</code> Description of the group.
<code>getId</code>	<code>int getId()</code> Unique ID for the group.
<code>getName</code>	<code>String getName()</code> Name of the group. Unique.

getDescription`String getDescription()`

Description of the group.

getId`int getId()`

Unique ID for the group.

getName

`String getName()`

Name of the group. Unique.

9.9.6. Admin

Administration management

Method summary	
Method	Description
<code>account</code>	<code>Account account(String name, String fullName, String email, String password = '', List groupNames = [])</code> Creates or updates an account.
<code>accountGroup</code>	<code>AccountGroup accountGroup(String name, String description)</code> Creates or updates an account group.
<code>getAccountGlobalPermissions</code>	<code>List<Role> getAccountGlobalPermissions(String accountName)</code> Gets the list of global roles an account has. See Account permissions .
<code>getAccountGroupGlobalPermissions</code>	<code>List<Role> getAccountGroupGlobalPermissions(String groupName)</code> Gets the list of global roles an account group has. See Account group permissions .
<code>getAccountGroupProjectPermissions</code>	<code>List<Role> getAccountGroupProjectPermissions(String projectName, String groupName)</code> Gets the list of roles an account group has on a project. See Account group permissions .
<code>getAccountProjectPermissions</code>	<code>List<Role> getAccountProjectPermissions(String projectName, String accountName)</code> Gets the list of roles an account has on a project. See Account permissions .
<code>getAccounts</code>	<code>List<Account> getAccounts()</code> Returns the list of all accounts.
<code>getGroups</code>	<code>List<AccountGroup> getGroups()</code> Returns the list of all groups.

<code>getLdapMappings</code>	<code>List<GroupMapping> getLdapMappings()</code> Gets the list of LDAP mappings.
<code>ldapMapping</code>	<code>GroupMapping ldapMapping(String name, String groupName)</code> Creates or updates a LDAP mapping.
<code>setAccountGlobalPermission</code>	<code>void setAccountGlobalPermission(String accountName, String globalRole)</code> Sets a global role on an account. See Account permissions .
<code>setAccountGroupGlobalPermission</code>	<code>void setAccountGroupGlobalPermission(String groupName, String globalRole)</code> Sets a global role on an account group. See Account group permissions .
<code>setAccountGroupProjectPermission</code>	<code>void setAccountGroupProjectPermission(String projectName, String groupName, String projectRole)</code> Sets a project role on an account group. See Account group permissions .
<code>setAccountProjectPermission</code>	<code>void setAccountProjectPermission(String projectName, String accountName, String projectRole)</code> Sets a project role on an account. See Account permissions .

account

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

Creates or updates an account.

See: [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'
}
```

accountGroup

`List<Role> accountGroup(String name, String description)`

Creates or updates an account group.

See: [AccountGroup](#)

getAccountGlobalPermissions

`List<Role> getAccountGlobalPermissions(String accountName)`

Gets the list of global roles an account has. See [Account permissions](#).

getAccountGroupGlobalPermissions

`List<Role> getAccountGroupGlobalPermissions(String groupName)`

Gets the list of global roles an account group has. See [Account group permissions](#).

getAccountGroupProjectPermissions

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

Gets the list of roles an account group has on a project. See [Account group permissions](#).

getAccountProjectPermissions

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

Gets the list of roles an account has on a project. See [Account permissions](#).

getAccounts

`List<Account> getAccounts()`

Returns the list of all accounts.

See: [Account](#)

getGroups

`List<AccountGroup> getGroups()`

Returns the list of all groups.

See: [AccountGroup](#)

getLdapMappings

```
List<GroupMapping> getLdapMappings()
```

Gets the list of LDAP mappings.

See: [GroupMapping](#)

ldapMapping

```
GroupMapping ldapMapping(String name, String groupName)
```

Creates or updates a LDAP mapping.

See: [GroupMapping](#) 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'  
}
```

setAccountGlobalPermission

```
void setAccountGlobalPermission(String accountName, String globalRole)
```

Sets a global role on an account. See [Account permissions](#).

setAccountGroupGlobalPermission

```
void setAccountGroupGlobalPermission(String groupName, String globalRole)
```

Sets a global role on an account group. See [Account group permissions](#).

setAccountGroupProjectPermission

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

Sets a project role on an account group. See [Account group permissions](#).

setAccountProjectPermission

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

Sets a project role on an account. See [Account permissions](#).

9.9.7. AuthenticationSource

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

See also: [AbstractResource](#)

Method summary	
Method	Description
<code>getId</code>	<code>String getId()</code> Identifier for the source: ldap, password
<code>getName</code>	<code>String getName()</code> Display name for the source
<code>isAllowingPasswordChange</code>	<code>boolean isAllowingPasswordChange()</code> Does this source allow to change the password?

getId
<code>String getId()</code>
Identifier for the source: ldap, password

getName
<code>String getName()</code>
Display name for the source

isAllowingPasswordChange
<code>boolean isAllowingPasswordChange()</code>
Does this source allow to change the password?

9.9.8. Branch

See also: [AbstractProjectResource](#)

Go to the [methods](#)

Configuration properties	
See also: ProjectEntityProperties	
Configuration property summary	
Property	Description

<code>artifactorySync</code>	<code>Object artifactorySync(String configuration, String buildName, String buildNameFilter = '*', int interval = 0)</code>
<code>getArtifactorySync</code>	<code>Object getArtifactorySync()</code>
<code>getGitBranch</code>	<code>Object getGitBranch()</code>
<code>getSvn</code>	<code>Object getSvn()</code>
<code>getSvnSync</code>	<code>Object getSvnSync()</code>
<code>getSvnValidatorClosedIssues</code>	<code>Object getSvnValidatorClosedIssues()</code>
<code>gitBranch</code>	<code>Object gitBranch(String branch, Map params = [:])</code>
<code>svn</code>	<code>Object svn(Map params = [:])</code>
<code>svnSync</code>	<code>Object svnSync(int interval = 0, boolean override = false)</code>
<code>svnValidatorClosedIssues</code>	<code>Object svnValidatorClosedIssues(Collection closedStatuses)</code>

Configuration: svn

```
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 [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}'
```

Configuration: getSvn

Object `getSvn()`

See [svn](#)

Configuration: gitBranch

```
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

Configuration: getGitBranch

`Object getGitBranch()`

See [gitBranch](#)

```
    gitBranch 'release/1.0'.  
[
```

Configuration: svnValidatorClosedIssues

`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']
```

Configuration: getSvnValidatorClosedIssues

`Object getSvnValidatorClosedIssues()`

See [svnValidatorClosedIssues](#)

Configuration: svnSync

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
```

Configuration: getSvnSync

Object `getSvnSync()`

See [svnSync](#)

Configuration: artifactorySync

```
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.

Configuration: getArtifactorySync

```
Object getArtifactorySync()
```

See [artifactorySync](#)

Method summary

Method	Description
build	<pre>Build build(String name, String description = '', boolean getIfExists = false)</pre> <p>Creates a build for the branch</p>

<code>build</code>	<code>Build build(String name, String description = '', boolean getIfExists = false, Closure closure)</code> Creates a build for the branch and configures it using a closure. See build .
<code>call</code>	<code>Object call(Closure closure)</code> Configures the branch using a closure.
<code>disable</code>	<code>Object disable()</code> Disables the branch
<code>download</code>	<code>String download(String path)</code>
<code>enable</code>	<code>Object enable()</code> Enables the branch
<code>filter</code>	<code>List<Build> filter(String filterType, Map filterConfig)</code> Runs any filter and returns the list of corresponding builds.
<code>getConfig</code>	<code>BranchProperties getConfig()</code> Access to the branch properties
<code>getInstance</code>	<code>TemplateInstance getInstance()</code>
<code>getLastPromotedBuilds</code>	<code>List<Build> getLastPromotedBuilds()</code> Returns the last promoted builds.
<code>getProject</code>	<code>String getProject()</code> Returns the name of the project the branch belongs to.
<code>getPromotionLevels</code>	<code>List<PromotionLevel> getPromotionLevels()</code> Gets the list of promotion levels for this branch.
<code>getType</code>	<code>String getType()</code>
<code>getValidationStamps</code>	<code>List<ValidationStamp> getValidationStamps()</code> Gets the list of validation stamps for this branch.
<code>instance</code>	<code>Branch instance(String sourceName, Map params)</code> Creates or updates a new branch from this branch template. See DSL Branch template definitions .
<code>isDisabled</code>	<code>boolean isDisabled()</code> Gets the disabled state of the branch

<code>link</code>	<code>Object link(String templateName, boolean manual = true, Map parameters)</code>
<code>promotionLevel</code>	<code>PromotionLevel promotionLevel(String name, String description = '', boolean getIfExists = false)</code> Creates a promotion level for this branch.
<code>promotionLevel</code>	<code>PromotionLevel promotionLevel(String name, String description = '', boolean getIfExists = false, Closure closure)</code> Creates a promotion level for this branch and configures it using a closure.
<code>standardFilter</code>	<code>List<Build> standardFilter(Map filterConfig)</code> Returns a list of builds for the branch, filtered according to given criteria.
<code>sync</code>	<code>Object sync()</code> Synchronizes the branch template with its associated instances. Will fail if this branch is not a template.
<code>syncInstance</code>	<code>Object syncInstance()</code> Synchronises the branch instance with its associated template. Will fail if this branch is not a template instance.
<code>template</code>	<code>Object template(Closure closure)</code> Configure the branch as a template definition - see DSL Branch template definitions .
<code>unlink</code>	<code>Object unlink()</code>
<code>validationStamp</code>	<code>ValidationStamp validationStamp(String name, String description = '', boolean getIfExists = false)</code> Creates a validation stamp for this branch.
<code>validationStamp</code>	<code>ValidationStamp validationStamp(String name, String description = '', boolean getIfExists = false, Closure closure)</code> Creates a validation stamp for this branch and configures it using a closure.

build

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

Creates a build for the branch

See: [Build](#) 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`.

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](#).

See: [Build](#)

call

```
Object call(Closure closure)
```

Configures the branch using a closure.

disable

```
Object disable()
```

Disables the branch

download

```
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](#).

enable

```
Object enable()
```

Enables the branch

filter

```
List<Build> filter(String filterType, Map filterConfig)
```

Runs any filter and returns the list of corresponding builds.

See: [Build](#) This is a low level method and more specialised methods should instead be used like [standardFilter](#) and [getLastPromotedBuilds](#).

getConfig

```
BranchProperties getConfig()
```

Access to the branch properties

getInstance

```
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](#).

```
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']
```

getLastPromotedBuilds

`List<Build> getLastPromotedBuilds()`

Returns the last promoted builds.

See: [Build](#) For example, to get the last promoted build:

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

getProject

`String getProject()`

Returns the name of the project the branch belongs to.

Sample:

```
def branch = ontrack.branch('prj', 'master')
assert branch.project == 'prj'
```

getPromotionLevels

`List<PromotionLevel> getPromotionLevels()`

Gets the list of promotion levels for this branch.

See: [PromotionLevel](#)

getType

`String getType()`

Returns the type of the branch when it comes to [templating](#).

Possible values are:

- CLASSIC
- TEMPLATE_DEFINITION
- TEMPLATE_INSTANCE

getValidationStamps

```
List<ValidationStamp> getValidationStamps()
```

Gets the list of validation stamps for this branch.

See: [ValidationStamp](#)

instance

```
Branch instance(String sourceName, Map params)
```

Creates or updates a new branch from this branch template. See [DSL Branch template definitions](#).

isDisabled

```
boolean isDisabled()
```

Gets the disabled state of the branch

link

```
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 [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 promotionLevel(String name, String description = '', boolean getIfExists = false)
```

Creates a promotion level for this branch.

See: [PromotionLevel](#)

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](#)

standardFilter

`List<Build> standardFilter(Map filterConfig)`

Returns a list of builds for the branch, filtered according to given criteria.

See: [Build](#) 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
<code>count</code>	Maximum number of builds to be returned. Defaults to 10 .
<code>sincePromotionLevel</code>	Name of a promotion level . After reaching the first build having this promotion, no further build is returned.
<code>withPromotionLevel</code>	Name of a promotion level . Only builds having this promotion are returned.
<code>afterDate</code>	ISO 8601 date. Only builds having been created on or after this date are returned.
<code>beforeDate</code>	ISO 8601 date. Only builds having been created on or before this date are returned.
<code>sinceValidationStamp</code>	Name of a validation stamp . After reaching the first build having this validation (whatever the status), no further build is returned.
<code>sinceValidationStampStatus</code>	Refines the <code>sinceValidationStamp</code> criteria to check on the status of the validation.
<code>withValidationStamp</code>	Name of a validation stamp . Only builds having this validation (whatever the status) are returned.
<code>withValidationStampStatus</code>	Refines the <code>withValidationStamp</code> criteria to check on the status of the validation.
<code>withProperty</code>	Qualified name of a property (full class name of the PropertyType). Only builds having this property being set are returned.
<code>withPropertyValue</code>	Refines the <code>withProperty</code> criteria to check the property value. The way the value is matched with the actual value depends on the property.
<code>sinceProperty</code>	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.

sync

`Object sync()`

Synchronizes the branch template with its associated instances. Will fail if this branch is not a template.

`linkedFrom`

Selects builds which are *linked from* the build

syncInstance

`Object syncInstance()`

Synchronises the branch instance with its associated template. Will fail if this branch is not a template instance.

selected by the criteria. See [Build links](#) for the

template

`Object template(Closure closure)`

Configure the branch as a template definition - see [DSL Branch template definitions](#).

unlink

`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'
```

validationStamp

`ValidationStamp validationStamp(String name, String description = '', boolean getIfExists = false)`

Creates a validation stamp for this branch.

See: [ValidationStamp](#)

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](#)

9.9.9. Build

In order to get the change log between two builds, look at the documentation at [DSL Change logs](#).

See also: [AbstractProjectResource](#)

Go to the [methods](#)

Configuration properties

See also: [ProjectEntityProperties](#)

Configuration property summary

Property	Description
<code>getGitCommit</code>	<code>Object getGitCommit()</code> Gets the Git commit associated to this build.
<code>getJenkinsBuild</code>	<code>Object getJenkinsBuild()</code> Gets the Jenkins build property.
<code>getLabel</code>	<code>Object getLabel()</code>
<code>gitCommit</code>	<code>Object gitCommit(String commit)</code> Sets a Git commmit associated to this build.
<code>jenkinsBuild</code>	<code>Object jenkinsBuild(String configuration, String job, int buildNumber)</code> Associates a Jenkins build with this build.
<code>label</code>	<code>Object label(String name)</code>

Configuration: getLabel

`Object getLabel()`

See [label](#)

Configuration: label

```
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'
```

Configuration: jenkinsBuild

```
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](#).

Configuration: getJenkinsBuild

`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.

Configuration: gitCommit

`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'
```

Configuration: getGitCommit

`Object getGitCommit()`

Gets the Git commit associated to this build.

Method summary

Method	Description
<code>buildLink</code>	<code>Object buildLink(String project, String build)</code>
<code>call</code>	<code>Object call(Closure closure)</code> Configuration of the build in a closure.
<code>getBranch</code>	<code>String getBranch()</code> Gets the build branch name.
<code>getBuildLinks</code>	<code>List<Build> getBuildLinks()</code>
<code>getConfig</code>	<code>BuildProperties getConfig()</code>
<code>getProject</code>	<code>String getProject()</code> Gets the build project name.
<code>getPromotionRuns</code>	<code>List<PromotionRun> getPromotionRuns()</code> Gets the list of promotion runs for this build
<code>getRunInfo</code>	<code>RunInfo getRunInfo()</code> Gets the associated run info with this build, or <code>null</code> if none
<code>getSvnRevisionDecoration</code>	<code>Long getSvnRevisionDecoration()</code>
<code>getValidationRuns</code>	<code>List<ValidationRun> getValidationRuns()</code> Gets the list of validation runs for this build
<code>promote</code>	<code>PromotionRun promote(String promotion, Closure closure)</code> Promotes this build to the given promotion level and configures the created promotion run .
<code>promote</code>	<code>PromotionRun promote(String promotion)</code> Promotes this build to the given promotion level.
<code>setRunInfo</code>	<code>void setRunInfo(Map info)</code> Sets the run info for this build.
<code>validate</code>	<code>ValidationRun validate(String validationStamp, String validationStampStatus = 'PASSED', Closure closure)</code>
<code>validate</code>	<code>ValidationRun validate(String validationStamp, String validationStampStatus = 'PASSED')</code>

buildLink

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.buildLinks
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.

call

Object call(Closure closure)

Configuration of the build in a closure.

getBranch

String getBranch()

Gets the build branch name.

getBuildLinks

List<Build> getBuildLinks()

See [buildLink](#)

getConfig

`BuildProperties getConfig()`

getProject

`String getProject()`

Gets the build project name.

getPromotionRuns

`List<PromotionRun> getPromotionRuns()`

Gets the list of promotion runs for this build

See: [PromotionRun](#)

getRunInfo

`RunInfo getRunInfo()`

Gets the associated run info with this build, or `null` if none

The returned object has the following properties:

- `sourceType` - Type of source (like "jenkins")
- `sourceUri` - URI to the source of the run (like the URL to a Jenkins job)
- `triggerType` - Type of trigger (like "scm" or "user")
- `triggerData` - Data associated with the trigger (like a user ID or a commit)
- `runTime` - Time of the run (in seconds)

Example:

```
def build = ontrack.build('project', 'branch', '1')
def info = build.runInfo
assert info != null
assert info.runTime == 30
```

getSvnRevisionDecoration

```
Long getSvnRevisionDecoration()
```

Returns any [Subversion](#) revision attached to this build.

```
def build = ontrack.build('project', 'branch', '1')
Long revision = build.svnRevisionDecoration
assert revision != null
```

getValidationRuns

```
List<ValidationRun> getValidationRuns()
```

Gets the list of validation runs for this build

See: [ValidationRun](#)

promote

```
PromotionRun promote(String promotion, Closure closure)
```

Promotes this build to the given promotion level and configures the created [promotion run](#).

See: [PromotionRun](#)

promote

```
PromotionRun promote(String promotion)
```

Promotes this build to the given promotion level.

See: [PromotionRun](#)

setRunInfo

```
void setRunInfo(Map info)
```

Sets the run info for this build.

Accepted parameters are:

- `sourceType` - Type of source (like "jenkins")
- `sourceUri` - URI to the source of the run (like the URL to a Jenkins job)
- `triggerType` - Type of trigger (like "scm" or "user")
- `triggerData` - Data associated with the trigger (like a user ID or a commit)
- `runTime` - Time of the run (in seconds)

Example:

```
def build = ontrack.build('project', 'branch', '1')
build.runInfo triggerType: "user", triggerData: "damien", runTime: 45
```

validate

```
ValidationRun validate(String validationStamp, String validationStampStatus = 'PASSED', Closure closure)
```

See: [ValidationRun](#) Validates the build using the given validation stamp and status - possible values for the status are: PASSED, FAILED, DEFECTIVE, EXPLAINED, FIXED, INTERRUPTED, INVESTIGATING, WARNING

validate

```
ValidationRun validate(String validationStamp, String validationStampStatus = 'PASSED')
```

See: [ValidationRun](#) Validates the build using the given validation stamp and status - and configures the resulting validation run.

9.9.10. Config

General configuration of Ontrack.

Method summary

Method	Description
<code>artifactory</code>	<code>Object artifactory(String name, String url, String user = '', String password = '')</code> Creates or updates a Artifactory configuration.
<code>getArtifactory</code>	<code>List<String> getArtifactory()</code>
<code>getGit</code>	<code>List<String> getGit()</code>

<code>getGitHub</code>	<code>List<String> getGitHub()</code>
<code>getGitLab</code>	<code>List<String> getGitLab()</code>
<code>getGrantProjectViewToAll</code>	<code>boolean getGrantProjectViewToAll()</code> Checks if the projects are accessible in anonymous mode.
<code>getJenkins</code>	<code>List<String> getJenkins()</code>
<code>getJira</code>	<code>List<String> getJira()</code>
<code>getLdapSettings</code>	<code>LDAPSettings getLdapSettings()</code> Gets the global LDAP settings
<code>getPredefinedPromotionLevels</code>	<code>List<PredefinedPromotionLevel> getPredefinedPromotionLevels()</code> Gets the list of promotion levels. See autoPromotionLevel .
<code>getPredefinedValidationStamps</code>	<code>List<PredefinedValidationStamp> getPredefinedValidationStamps()</code> Gets the list of validation stamps. See autoValidationStamp .
<code>getStash</code>	<code>List<String> getStash()</code>
<code>getSvn</code>	<code>Object getSvn()</code>
<code>git</code>	<code>Object git(Map parameters, String name)</code> Creates or update a Git configuration
<code>github</code>	<code>Object github(String name)</code>
<code>github</code>	<code>Object github(Map parameters, String name)</code>
<code>gitLab</code>	<code>Object gitLab(Map parameters, String name)</code>
<code>jenkins</code>	<code>Object jenkins(String name, String url, String user = '', String password = '')</code> Creates or updates a Jenkins configuration.
<code>jira</code>	<code>Object jira(String name, String url, String user = '', String password = '')</code> Creates or updates a JIRA configuration.
<code>predefinedPromotionLevel</code>	<code>PredefinedPromotionLevel predefinedPromotionLevel(String name, String description = '', boolean getIfExists = false)</code> See autoPromotionLevel .
<code>predefinedPromotionLevel</code>	<code>PredefinedPromotionLevel predefinedPromotionLevel(String name, String description = '', boolean getIfExists = false, Closure closure)</code> See autoPromotionLevel .

<code>predefinedValidationStamp</code>	<code>PredefinedValidationStamp</code> <code>predefinedValidationStamp(String name, String description = '', boolean getIfExists = false)</code> <p>See autoValidationStamp.</p>
<code>predefinedValidationStamp</code>	<code>PredefinedValidationStamp</code> <code>predefinedValidationStamp(String name, String description = '', boolean getIfExists = false, Closure closure)</code> <p>See autoValidationStamp.</p>
<code>setGrantProjectViewToAll</code>	<code>Object setGrantProjectViewToAll(boolean grantProjectViewToAll)</code> <p>Sets if the projects are accessible in anonymous mode.</p>
<code>setLdapSettings</code>	<code>Object setLdapSettings(LDAPSettings settings)</code> <p>Sets the global LDAP settings</p>
<code>stash</code>	<code>Object stash(Map parameters, String name)</code> <p>Creates or updates a BitBucket configuration.</p>
<code>svn</code>	<code>Object svn(Map parameters, String name)</code> <p>Creates a or updates a Subversion configuration.</p>

artifactory

`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
```

getArtifactory

`List<String> getArtifactory()`

See [artifactory](#)

getGit

`List<String> getGit()`

See [git](#)

getGitHub

`List<String> getGitHub()`

See [gitHub](#)

getGitLab

`List<String> getGitLab()`

See [gitLab](#)

getGrantProjectViewToAll

`boolean getGrantProjectViewToAll()`

Checks if the projects are accessible in anonymous mode.

Sample:

```
ontrack.config.grantProjectViewToAll = false  
assert !ontrack.config.grantProjectViewToAll
```

getJenkins

`List<String> getJenkins()`

See [jenkins](#)

getJira

`List<String> getJira()`

See [jira](#)

getLdapSettings

`LDAPSettings getLdapSettings()`

Gets the global LDAP settings

See: [LDAPSettings](#)

getPredefinedPromotionLevels

`List<PredefinedPromotionLevel> getPredefinedPromotionLevels()`

Gets the list of promotion levels. See [autoPromotionLevel](#).

See: [PredefinedPromotionLevel](#)

getPredefinedValidationStamps

`List<PredefinedValidationStamp> getPredefinedValidationStamps()`

Gets the list of validation stamps. See [autoValidationStamp](#).

See: [PredefinedValidationStamp](#)

getStash

`List<String> getStash()`

See [stash](#)

getSvn

`Object getSvn()`

See [svn](#)

git

```
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
<code>remote</code>	the remote location
<code>user</code>	user used to connect to GitHub (optional)
<code>password</code>	password used to connect to GitHub (optional)
<code>commitLink</code>	Link to a commit, using <code>{commit}</code> as placeholder
<code>fileAtCommitLink</code>	Link to a file at a given commit, using <code>{commit}</code> and <code>{path}</code> as placeholders
<code>indexationInterval</code>	interval (in minutes) between each synchronisation (Ontrack maintains internally a clone of the GitHub repository)
<code>issueServiceConfigurationIdentifier</code>	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
```

gitHub

`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
<code>url</code>	the GitHub URL - is set by default to https://github.com if not filled in, allowing for using GitHub enterprise as well
<code>user</code>	user used to connect to GitHub (optional)
<code>password</code>	password used to connect to GitHub (optional)
<code>oauth2Token</code>	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'  
}
```

gitHub

`Object gitHub(Map parameters, String name)`

See [gitHub](#)

gitLab

```
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
<code>url</code>	the URL to the GitLab application, without any repository nor project reference
<code>user</code>	user used to connect to GitHub
<code>password</code>	Personal Access Token associated with the user
<code>ignoreSslCertificate</code>	Set to <code>true</code> if SSL checks must be disabled (optional, default to <code>false</code>)



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
```

jenkins

```
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
```

jira

```
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
```

predefinedPromotionLevel

```
PredefinedPromotionLevel predefinedPromotionLevel(String name, String description = '',  
boolean getIfExists = false)
```

See [autoPromotionLevel](#).

See: [PredefinedPromotionLevel](#)

predefinedPromotionLevel

```
PredefinedPromotionLevel predefinedPromotionLevel(String name, String description = '',  
boolean getIfExists = false, Closure closure)
```

See [autoPromotionLevel](#).

See: [PredefinedPromotionLevel](#)

predefinedValidationStamp

```
PredefinedValidationStamp predefinedValidationStamp(String name, String description = '',  
boolean getIfExists = false)
```

See [autoValidationStamp](#).

See: [PredefinedValidationStamp](#)

predefinedValidationStamp

```
PredefinedValidationStamp predefinedValidationStamp(String name, String description = '',  
boolean getIfExists = false, Closure closure)
```

See [autoValidationStamp](#).

See: [PredefinedValidationStamp](#)

setGrantProjectViewToAll

```
Object setGrantProjectViewToAll(boolean grantProjectViewToAll)
```

Sets if the projects are accessible in anonymous mode.

Sample:

```
ontrack.config.grantProjectViewToAll = true  
assert ontrack.config.grantProjectViewToAll
```

setLdapSettings

```
Object setLdapSettings(LDAPSettings settings)
```

Sets the global LDAP settings

stash

```
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
<code>url</code>	the URL of the Stash instance
<code>user</code> L user used to connect to Stash (optional)	<code>password</code>

Example:

```
ontrack.configure {
    stash 'MyStash', url: 'https://stask.example.com'
}
assert ontrack.config.stash.find { it == 'MyStash' } != null
```

svn

Object `svn(Map parameters, String name)`

Creates a or updates a Subversion configuration.

In order to create, update and access a Subversion configuration, use:

```
ontrack.configure {  
    svn 'myconfig', url: 'https://myhost/repo'  
}  
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
<code>url</code>	URL of the Subversion repository (required)
<code>user</code>	
<code>password</code>	
<code>tagFilterPattern</code>	none by default
<code>browserForPath</code>	none by default
<code>browserForRevision</code>	none by default
<code>browserForChange</code>	none by default
<code>indexationInterval</code>	0 by default
<code>indexationStart</code>	revision to start the indexation from (1 by default)
<code>issueServiceConfigurationIdentifier</code>	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'  
}
```

9.9.11. Document

Definition for a document, for upload and download methods. See also [DSL Images and documents](#).

Method summary

Method	Description
<code>getContent</code>	<code>byte[] getContent()</code> Returns the content of the document as an array of bytes.
<code>getType</code>	<code>String getType()</code> Returns the MIME type of the document.
<code>isEmpty</code>	<code>boolean isEmpty()</code> Returns true if the document is empty and has no content.

getContent
<code>byte[] getContent()</code>
Returns the content of the document as an array of bytes.

getType
<code>String getType()</code>
Returns the MIME type of the document.

isEmpty
<code>boolean isEmpty()</code>
Returns true if the document is empty and has no content.

9.9.12. GroupMapping

Mapping between a LDAP group and an account group.

See also: [AbstractResource](#)

Method summary	
Method	Description
<code>getGroupName</code>	<code>String getGroupName()</code> Name of the Ontrack account group.
<code>getName</code>	<code>String getName()</code> Name of the LDAP group.

getGroupName

`String getGroupName()`

Name of the Ontrack account group.

getName

`String getName()`

Name of the LDAP group.

9.9.13. LDAPSettings

LDAP settings parameters.

The LDAP settings are defined using the following values:

Parameter	Description
enabled	Set to <code>true</code> to actually enable the LDAP Authentication
url	URL to the LDAP end point. For example, <code>ldaps://ldap.company.com:636</code>
searchBase	DN for the search root, for example, <code>dc=company,dc=com</code>
searchFilter	Query to look for an account. <code>{0}</code> is replaced by the account name. For example, <code>(sAMAccountName={0})</code>
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 <code>cn</code>
emailAttribute	Attribute which contains the email of the account. Default to <code>email</code> .
groupAttribute	Multiple attribute name which contains the groups the account belong to. Defaults to <code>memberOf</code> .
groupFilter	When getting the list of groups for an account, filter this list using the <code>OU</code> 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',
]

```

9.9.14. PredefinedPromotionLevel

See also: [AbstractResource](#)

Method summary	
Method	Description
getImage	<p>Document getImage()</p> <p>Downloads the image for the promotion level. See DSL Images and documents.</p>
image	<p>Object image(Object o)</p> <p>Sets the image for this validation stamp (must be a PNG file). See DSL Images and documents.</p>

getImage

[Document getImage\(\)](#)

Downloads the image for the promotion level. See [DSL Images and documents](#).

See: [Document](#)

image

[Object image\(Object o\)](#)

Sets the image for this validation stamp (must be a PNG file). See [DSL Images and documents](#).

9.9.15. PredefinedValidationStamp

See also: [AbstractResource](#)

Method summary	
Method	Description

<code>getImage</code>	<code>Document getImage()</code>
	Downloads the image for the validation stamp. See DSL Images and documents .
<code>image</code>	<code>Object image(Object o)</code>
	Sets the image for this validation stamp (must be a PNG file). See DSL Images and documents .

getImage

`Document getImage()`

Downloads the image for the validation stamp. See [DSL Images and documents](#).

See: [Document](#)

image

`Object image(Object o)`

Sets the image for this validation stamp (must be a PNG file). See [DSL Images and documents](#).

9.9.16. 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](#)

Go to the [methods](#)

Configuration properties

See also: [ProjectEntityProperties](#)

Configuration property summary

Property	Description
<code>autoPromotionLevel</code>	<code>Object autoPromotionLevel(boolean autoCreate = true)</code>
<code>autoValidationStamp</code>	<code>Object autoValidationStamp(boolean autoCreate = true, boolean autoCreateIfNotPredefined = false)</code>

getAutoPromotionLevel	boolean getAutoPromotionLevel()
getAutoValidationStamp	boolean getAutoValidationStamp()
getGit	Object getGit()
getGitLab	Object getGitLab()
getJiraFollowLinks	List<String> getJiraFollowLinks()
getStale	Object getStale()
getStash	Object getStash()
getSvn	Object getSvn()
git	Object git(String name)
	Configures the project for Git.
github	Object github(Map parameters, String name)
	Configures the project for GitHub.
gitLab	Object gitLab(Map parameters, String name)
jiraFollowLinks	Object jiraFollowLinks(Collection linkNames)
jiraFollowLinks	Object jiraFollowLinks(String[] linkNames)
stale	Object stale(int disablingDuration = 0, int deletingDuration = 0, List promotionsToKeep = [])
	Setup of stale branches management.
stash	Object stash(String name, String project, String repository, int indexationInterval = 0, String issueServiceConfigurationIdentifier = '')
svn	Object svn(String name, String projectPath)
	Configures the project for Subversion.

Configuration: stale

`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 **PRODUCTION**:

```
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']
```

Configuration: gitLab

`Object gitLab(Map parameters, String name)`

Configuration: getGitLab

`Object getGitLab()`

See [gitLab](#)

Configuration: gitHub

`Object gitHub(Map parameters, String name)`

Configures the project for GitHub.

Configuration: stash

```
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'
```

Configuration: getStash

```
Object getStash()
```

See [stash](#)

Configuration: git

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'
```

Configuration: getGit

Object `getGit()`

See [git](#)

Configuration: svn

`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:

```
ontrack.project('project') {  
    config {  
        svn 'myconfig', '/project/trunk'  
    }  
}  
def cfg = ontrack.project('project').config.svn  
assert cfg.configuration.name == 'myconfig'  
assert cfg.projectPath == '/project/trunk'
```

Configuration: getSvn

`Object getSvn()`

See [svn](#)

Configuration: getStale

`Object getStale()`

See [stale](#)

Configuration: jiraFollowLinks

`Object jiraFollowLinks(Collection linkNames)`

See [jiraFollowLinks](#)

Configuration: jiraFollowLinks

`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']
```

Configuration: getJiraFollowLinks

`List<String> getJiraFollowLinks()`

See [jiraFollowLinks](#)

Configuration: autoValidationStamp

```
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 exist. 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](#)

Configuration: getAutoValidationStamp

boolean getAutoValidationStamp()

See [autoValidationStamp](#)

```
ontrack.config.predefinedValidationStamp('VS') {  
    image new File('my/image.png')  
}
```



You need Administrator rights to be able to update the predefined validation stamps.

Configuration: autoPromotionLevel

```
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.

Configuration: getAutoPromotionLevel

`boolean getAutoPromotionLevel()`

See [autoPromotionLevel](#)

Method summary

Method	Description
<code>branch</code>	<code>Branch branch(String name, String description = '', boolean getIfExists = false)</code> Retrieves or creates a branch for the project
<code>branch</code>	<code>Branch branch(String name, String description = '', boolean getIfExists = false, Closure closure)</code> Retrieves or creates a branch for the project, and then configures it.
<code>getBranches</code>	<code>List<Branch> getBranches()</code> Gets the list of branches for the project.
<code>getConfig</code>	<code>ProjectProperties getConfig()</code> Access to the project properties
<code>search</code>	<code>List<Build> search(Map form)</code> Searches for builds in the project.

branch

`Branch branch(String name, String description = '', boolean getIfExists = false)`

Retrieves or creates a branch for the project

See: [Branch](#) 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.

Sample:

```
def project = ontrack.project('test')
def branch = project.branch('new-branch')
```

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](#)

getBranches

```
List<Branch> getBranches()
```

Gets the list of branches for the project.

See: [Branch](#)

getConfig

```
ProjectProperties getConfig()
```

Access to the project properties

search

List<Build> search(Map form)

Searches for builds in the project.

See: [Build](#) 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 <code>buildName</code> 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 <code>PASSED</code>	none
property	Qualified name of a property that the build must have	none
propertyValue	Together with <code>property</code> , 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 <i>linked from</i> the build selected by the criteria. See Build links for the exact syntax.	none
linkedTo	Selects builds which are <i>linked to</i> the build selected by the criteria. See Build links for the exact syntax.	none

Example of build searches:

9.9.17. ProjectEntityProperties

Method summary

Method	Description
<code>getJenkinsBuild</code>	<code>Object getJenkinsBuild()</code>
<code>getJenkinsJob</code>	<code>Object getJenkinsJob()</code>
<code>getLinks</code>	<code>Map<String, String> getLinks()</code>
<code>getMessage</code>	<code>Object getMessage()</code>
<code>getMetaInfo</code>	<code>List<MetaInfo> getMetaInfo()</code>
<code>jenkinsBuild</code>	<code>Object jenkinsBuild(String configuration, String job, int build)</code>
<code>jenkinsJob</code>	<code>Object jenkinsJob(String configuration, String job)</code>
<code>links</code>	<code>Object links(Map links)</code>
<code>message</code>	<code>Object message(String text, String type = 'INFO')</code>
<code>metaInfo</code>	<code>Object metaInfo(Map map)</code>
<code>metaInfo</code>	<code>Object metaInfo(String name, String value, String link = null, String category = null)</code>
<code>property</code>	<code>Object property(String type, Map data)</code> Sets a property.
<code>property</code>	<code>Object property(String type, boolean required = true)</code>

getJenkinsBuild

`Object getJenkinsBuild()`

See [jenkinsBuild](#)

getJenkinsJob

`Object getJenkinsJob()`

See [jenkinsJob](#)

getLinks

`Map<String, String> getLinks()`

See [links](#)

getMessage

`Object getMessage()`

See [message](#)

getMetaInfo

`List<MetaInfo> getMetaInfo()`

See [metaInfo](#)

jenkinsBuild

```
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'

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'

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](#).

jenkinsJob

```
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:

links

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'
```

```
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](#).

message

```
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 [Message property](#) for more details about this property.

metaInfo

```
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 [Meta information property](#) for more details about this property.

metaInfo

```
Object metaInfo(String name, String value, String link = null, String category = null)
```

See [metaInfo](#)

property

```
Object property(String type, Map data)
```

Sets a property.

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
```

9.9.18. PromotionLevel

See also: [AbstractProjectResource](#)

Go to the [methods](#)

Configuration 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](#)

Configuration property summary

Property	Description
<code>autoPromotion</code>	<code>Object autoPromotion(String[] validationStamps)</code> Sets the validation stamps participating into the auto promotion.

<code>autoPromotion</code>	<code>Object autoPromotion(Collection validationStamps, String include = '', String exclude = '')</code>
	Sets the validation stamps participating into the auto promotion, and sets the include/exclude settings.

<code>getAutoPromotion</code>	<code>boolean getAutoPromotion()</code>
	Checks if the promotion level is set in auto promotion.

Configuration: `getAutoPromotion`

`boolean getAutoPromotion()`

Checks if the promotion level is set in auto promotion.

Configuration: `autoPromotion`

`Object autoPromotion(String[] validationStamps)`

Sets the validation stamps participating into the auto promotion.

Configuration: `autoPromotion`

`Object autoPromotion(Collection validationStamps, String include = '', String exclude = '')`

Sets the validation stamps participating into the auto promotion, and sets the include/exclude settings.

Method summary

Method	Description
<code>call</code>	<code>Object call(Closure closure)</code> Configuration of the promotion level with a closure.
<code>getAutoPromotionPropertyDecoration</code>	<code>Boolean getAutoPromotionPropertyDecoration()</code> Checks if this promotion level is set in auto decoration mode .
<code>getBranch</code>	<code>String getBranch()</code> Name of the associated branch.
<code>getConfig</code>	<code>PromotionLevelProperties getConfig()</code> Access to the promotion level properties

<code>getImage</code>	<code>Document getImage()</code>
	Gets the promotion level image (see DSL Images and documents)
<code>getProject</code>	<code>String getProject()</code>
	Name of the associated project.
<code>image</code>	<code>Object image(Object o)</code>
	Sets the promotion level image (see DSL Images and documents)
<code>image</code>	<code>Object image(Object o, String contentType)</code>
	Sets the promotion level image (see DSL Images and documents)

call

`Object call(Closure closure)`

Configuration of the promotion level with a closure.

getAutoPromotionPropertyDecoration

`Boolean getAutoPromotionPropertyDecoration()`

Checks if this promotion level is set in [auto decoration mode](#).

getBranch

`String getBranch()`

Name of the associated branch.

getConfig

`PromotionLevelProperties getConfig()`

Access to the promotion level properties

getImage

`Document getImage()`

Gets the promotion level image (see [DSL Images and documents](#))

See: [Document](#)

getProject

```
String getProject()
```

Name of the associated project.

image

```
Object image(Object o)
```

Sets the promotion level image (see [DSL Images and documents](#))

image

```
Object image(Object o, String contentType)
```

Sets the promotion level image (see [DSL Images and documents](#))

9.9.19. 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](#)

Method summary

Method	Description
<code>getPromotionLevel</code>	<code>Object getPromotionLevel()</code> Gets the associated promotion level (JSON)

getPromotionLevel

```
Object getPromotionLevel()
```

Gets the associated promotion level (JSON)

9.9.20. SearchResult

The **SearchResult** class is used for listing the results of a [search](#).

See also: [AbstractResource](#)

Sample:

```
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'
```

Method summary

Method	Description
getAccuracy	<code>int getAccuracy()</code> Gets a percentage of accuracy about the result.
getDescription	<code>String getDescription()</code> Gets a description for the search result.
getPage	<code>String getPage()</code> Gets the URI to display the search result details (Web).
getTitle	<code>String getTitle()</code> Gets the display name for the search result.
getUri	<code>String getUri()</code> Gets the URI to access the search result details (API).

[getAccuracy](#)

`int getAccuracy()`

Gets a percentage of accuracy about the result.

[getDescription](#)

`String getDescription()`

Gets a description for the search result.

getPage

`String getPage()`

Gets the URI to display the search result details (Web).

getTitle

`String getTitle()`

Gets the display name for the search result.

getUri

`String getUri()`

Gets the URI to access the search result details (API).

9.9.21. 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](#)

Method summary

Method	Description
<code>getRunInfo</code>	<code>RunInfo getRunInfo()</code> Gets the associated run info with this validation run or <code>null</code> if none
<code>getStatus</code>	<code>String getStatus()</code> Gets the status for this validation run.

<code>getValidationRunStatuses</code>	<code>Object getValidationRunStatuses()</code> Gets the list of statuses (JSON)
<code>getValidationStamp</code>	<code>Object getValidationStamp()</code> Gets the associated validation stamp (JSON)
<code>setRunInfo</code>	<code>void setRunInfo(Map info)</code> Sets the run info for this validation run.

getRunInfo

`RunInfo getRunInfo()`

Gets the associated run info with this validation run or `null` if none

The returned object has the following properties:

- `sourceType` - Type of source (like "jenkins")
- `sourceUri` - URI to the source of the run (like the URL to a Jenkins job)
- `triggerType` - Type of trigger (like "scm" or "user")
- `triggerData` - Data associated with the trigger (like a user ID or a commit)
- `runTime` - Time of the run (in seconds)

getStatus

`String getStatus()`

Gets the status for this validation run.

Possible values are:

- DEFECTIVE
- EXPLAINED
- FAILED
- FIXED
- INTERRUPTED
- INVESTIGATING
- PASSED
- WARNING

getValidationRunStatuses

`Object getValidationRunStatuses()`

Gets the list of statuses (JSON)

getValidationStamp

`Object getValidationStamp()`

Gets the associated validation stamp (JSON)

setRunInfo

`void setRunInfo(Map info)`

Sets the run info for this validation run.

Accepted parameters are:

- `sourceType` - Type of source (like "jenkins")
- `sourceUri` - URI to the source of the run (like the URL to a Jenkins job)
- `triggerType` - Type of trigger (like "scm" or "user")
- `triggerData` - Data associated with the trigger (like a user ID or a commit)
- `runTime` - Time of the run (in seconds)

9.9.22. ValidationStamp

See also: [AbstractProjectResource](#)

Method summary

Method	Description
<code>call</code>	<code>Object call(Closure closure)</code> Configuration of the promotion level with a closure.
<code>getBranch</code>	<code>String getBranch()</code> Name of the associated branch.
<code>getImage</code>	<code>Document getImage()</code> Gets the validation stamp image (see DSL Images and documents)
<code>getProject</code>	<code>String getProject()</code> Name of the associated project.

<code>getValidationStampWeatherDecoration</code>	<code>Object getValidationStampWeatherDecoration()</code>
<code>image</code>	Gets the validation stamp weather decoration. <code>Object image(Object o)</code>
<code>image</code>	Sets the validation stamp image (see DSL Images and documents) <code>Object image(Object o, String contentType)</code>
<code>image</code>	Sets the validation stamp image (see DSL Images and documents) <code>Object image(Object o, String contentType)</code>

call

`Object call(Closure closure)`

Configuration of the promotion level with a closure.

getBranch

`String getBranch()`

Name of the associated branch.

getImage

`Document getImage()`

Gets the validation stamp image (see [DSL Images and documents](#))

See: [Document](#)

getProject

`String getProject()`

Name of the associated project.

getValidationStampWeatherDecoration

```
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

image

```
Object image(Object o)
```

Sets the validation stamp image (see [DSL Images and documents](#))

image

```
Object image(Object o, String contentType)
```

Sets the validation stamp image (see [DSL Images and documents](#))