jBPM Documentation

Version 6.1.0.Beta3

by *The JBoss jBPM team* [http://www.jboss.org/jbpm]


jBPM

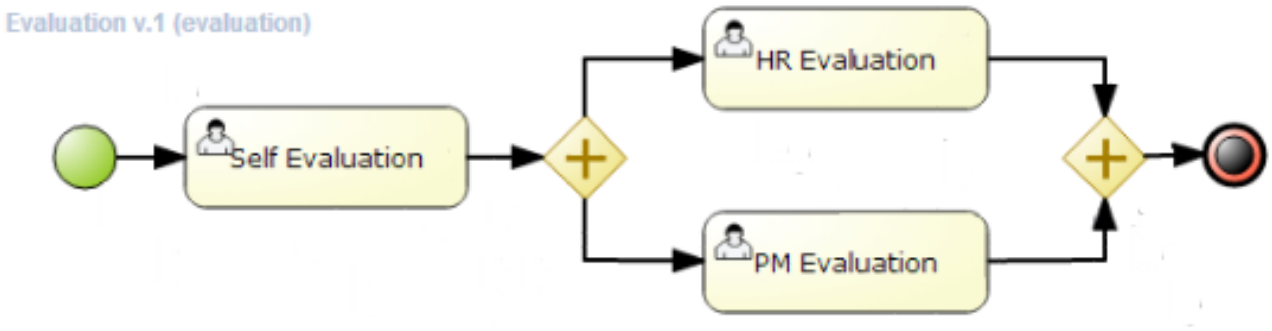
Part I. Getting Started

Introduction and getting started with jBPM

Chapter 1. Overview

1.1. What is jBPM?

jBPM is a flexible Business Process Management (BPM) Suite. It is light-weight, fully open-source (distributed under Apache license) and written in Java. It allows you to model, execute, and monitor business processes throughout their life cycle.



A business process allows you to model your business goals by describing the steps that need to be executed to achieve those goals, and the order of those goals are depicted using a flow chart. This process greatly improves the visibility and agility of your business logic. jBPM focuses on executable business processes, which are business processes that contain enough detail so they can actually be executed on a BPM engine. Executable business processes bridge the gap between business users and developers as they are higher-level and use domain-specific concepts that are understood by business users but can also be executed directly.

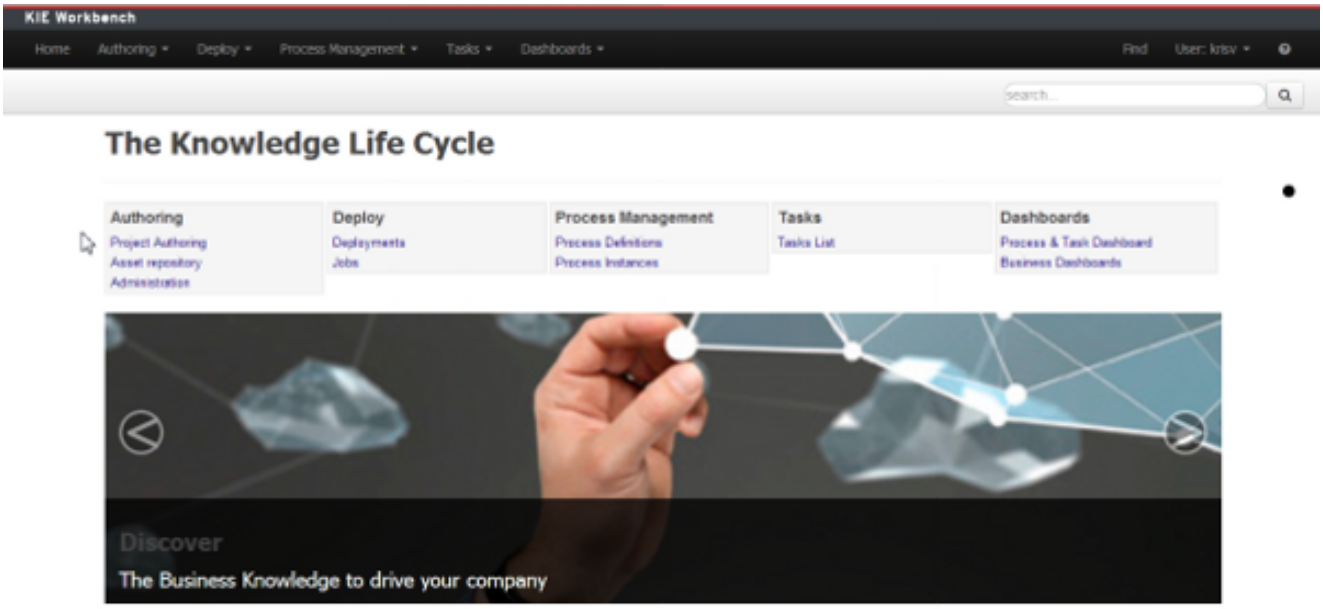
Business processes need to be supported throughout their entire life cycle: authoring, deployment, process management and task lists, and dashboards and reporting.

The core of jBPM is a light-weight, extensible workflow engine written in pure Java that allows you to execute business processes using the latest BPMN 2.0 specification. It can run in any Java environment, embedded in your application or as a service.

On top of the core engine, a lot of features and tools are offered to support business processes throughout their entire life cycle:

- Pluggable human task service based on WS-HumanTask for including tasks that need to be performed by human actors.
- Pluggable persistence and transactions (based on JPA / JTA).
- Web-based process designer to support the graphical creation and simulation of your business processes (drag and drop).
- Web-based data modeler and form modeler to support the creation of data models and process and task forms

- Web-based, customizable dashboards and reporting
- All combined in one web-based workbench, supporting the complete BPM life cycle:
  - Modeling and deployment - author your processes, rules, data models, forms and other assets
  - Execution - execute processes, tasks, rules and events on the core runtime engine
  - Runtime Management - work on assigned task, manage process instances, etc
  - Reporting - keep track of the execution using Business Activity Monitoring capabilities



- Eclipse-based developer tools to support the modeling, testing and debugging of processes
- Remote API to process engine as a service (REST, JMS, Remote Java API)
- Integration with Maven, Spring, OSGi, etc.

BPM creates the bridge between business analysts, developers and end users by offering process management features and tools in a way that both business users and developers like. Domain-specific nodes can be plugged into the palette, making the processes more easily understood by business users.

jBPM supports adaptive and dynamic processes that require flexibility to model complex, real-life situations that cannot easily be described using a rigid process. We bring control back to the end users by allowing them to control which parts of the process should be executed; this allows dynamic deviation from the process.

jBPM is not just an isolated process engine. Complex business logic can be modeled as a combination of business processes with business rules and complex event processing. jBPM can be combined with the Drools project to support one unified environment that integrates these paradigms where you model your business logic as a combination of processes, rules and events.

1.2. Overview

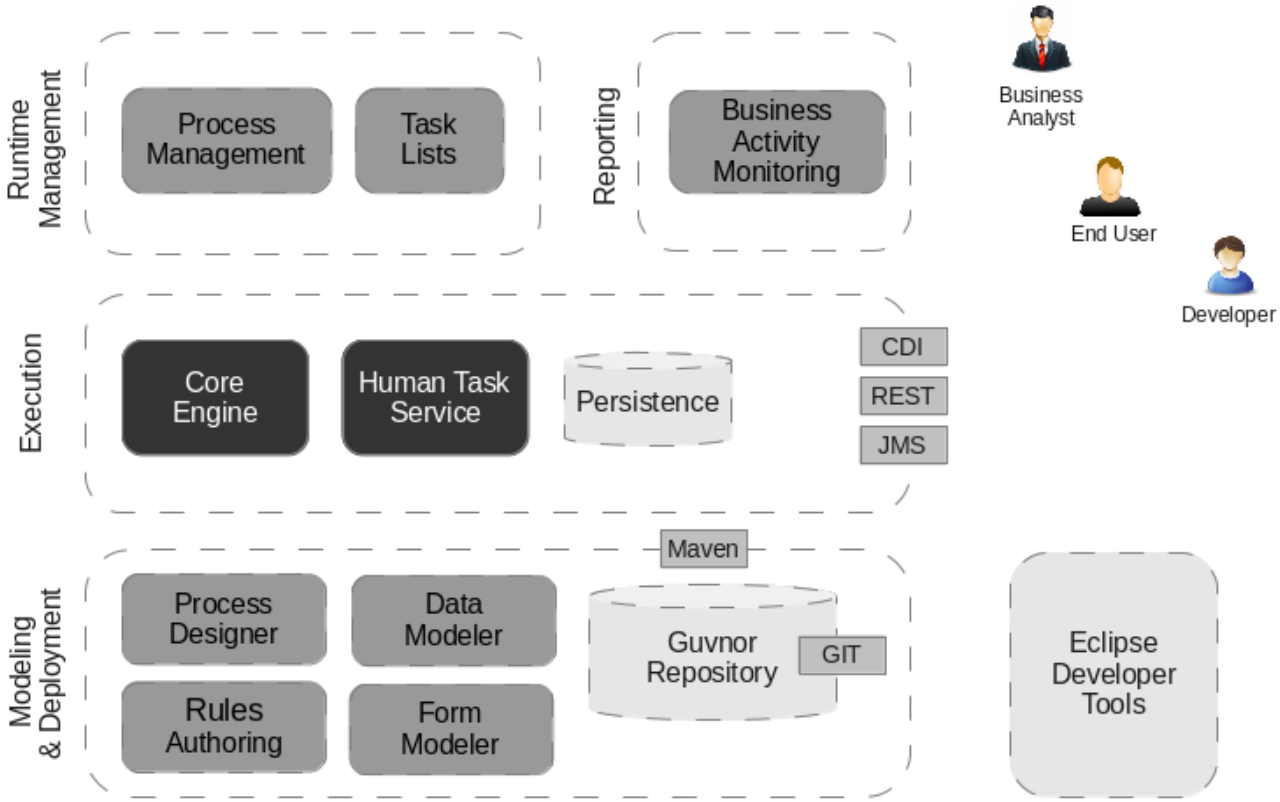


Figure 1.1.

This figure gives an overview of the different components of the jBPM project.

- The core engine is the heart of the project and allows you to execute business processes in a flexible manner. It is a pure Java component that you can choose to embed as part of your application or deploy it as a service and connect to it through the web-based UI or remote APIs.
  - An optional core service is the human task service that will take care of the human task life cycle if human actors participate in the process.
  - Another optional core service is runtime persistence; this will persist the state of all your process instances and log audit information about everything that is happening at runtime.
  - Applications can connect to the core engine by through its Java API or as a set of CDI services, but also remotely through a REST and JMS API.
- Web-based tools allows you to model, simulate and deploy your processes and other related artifacts (like data models, forms, rules, etc.):
  - The process designer allows business users to design and simulate business processes in a web-based environment.

    - The data modeler allows non-technical users to view, modify and create data models for use in your processes.
    - A web-based form modeler also allows you to create, generate or edit forms related to your processes (to start the process or to complete one of the user tasks).
    - Rule authoring allows you to specify different types of business rules (decision tables, guided rules, etc.) for combination with your processes.
    - All assets are stored and managed on the Guvnor repository (exposed through Git) and can be managed (versioning), built and deployed.
- The web-based management console allows business users to manage their runtime (manage business processes like start new processes, inspect running instances, etc.), to manage their task list and to perform Business Activity Monitoring (BAM) and see reports.
- The Eclipse-based developer tools are an extension to the Eclipse IDE, targeted towards developers, and allows you to create business processes using drag and drop, test and debug your processes, etc.

Each of the components are described in more detail below.

1.3. Core Engine

The core jBPM engine is the heart of the project. It's a light-weight workflow engine that executes your business processes. It can be embedded as part of your application or deployed as a service (possibly on the cloud). Its most important features are the following:

- Solid, stable core engine for executing your process instances.
- Native support for the latest BPMN 2.0 specification for modeling and executing business processes.
- Strong focus on performance and scalability.
- Light-weight (can be deployed on almost any device that supports a simple Java Runtime Environment; does not require any web container at all).
- (Optional) pluggable persistence with a default JPA implementation.
- Pluggable transaction support with a default JTA implementation.
- Implemented as a generic process engine, so it can be extended to support new node types or other process languages.
- Listeners to be notified of various events.
- Ability to migrate running process instances to a new version of their process definition

The core engine can also be integrated with a few other (independent) core services:

- The human task service can be used to manage human tasks when human actors need to participate in the process. It is fully pluggable and the default implementation is based on the WS-HumanTask specification and manages the life cycle of the tasks, task lists, task forms, and some more advanced features like escalation, delegation, rule-based assignments, etc.
- The history log can store all information about the execution of all the processes in the engine. This is necessary if you need access to historic information as runtime persistence only stores the current state of all active process instances. The history log can be used to store all current and historic states of active and completed process instances. It can be used to query for any information related to the execution of process instances, for monitoring, analysis, etc.

1.4. Process Designer

The web-based designer allows you to model your business processes in a web-based environment. It is targeted towards business users and offers a graphical editor for viewing and editing your business processes (using drag and drop), similar to the Eclipse plugin. It supports round-tripping between the Eclipse editor and the web-based designer. It also supports simulation of processes.

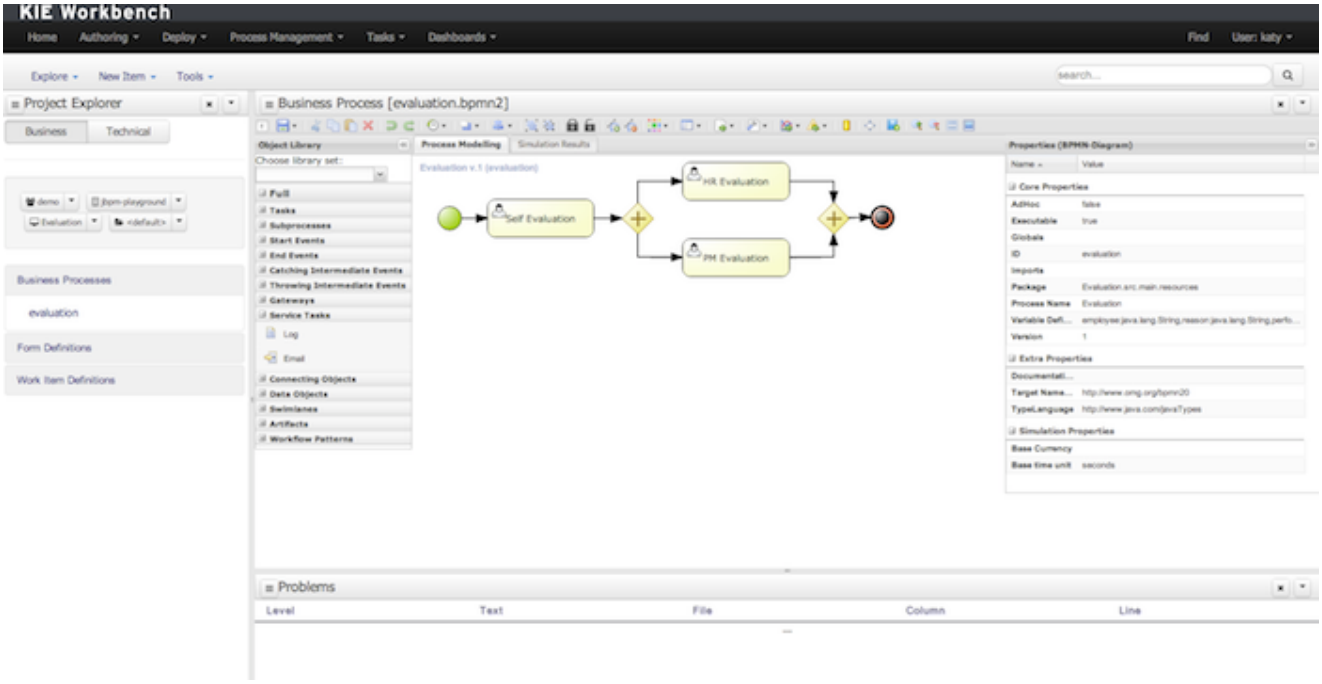


Figure 1.2. Web-based designer for creating BPMN2 processes

1.5. Data Modeler

Processes almost always have some kind of data to work with. The data modeler allows non-technical users to view, edit or create these data models.

Typically, a business process analyst or data analyst will capture the requirements for a process or application and turn these into a formal set of interrelated data structures. The new Data Modeler tool provides an easy, straightforward and visual aid for building both logical and physical data models, without the need for advanced development skills or explicit coding. The data modelers is transparently integrate into the workbench. Its main goals are to make data models into first class citizens in the process improvement cycle and allow for full process automation through the integrated use of data structures (and the forms that will be used to interact with them).

1.6. Form Modeler

The jBPM Form Modeler is a form engine and editor that enables users to create forms to capture and display information during process or task execution, without needing any coding or template markup skills.

It provides a WYSIWYG environment to model forms that it's easy to use for less technical users.

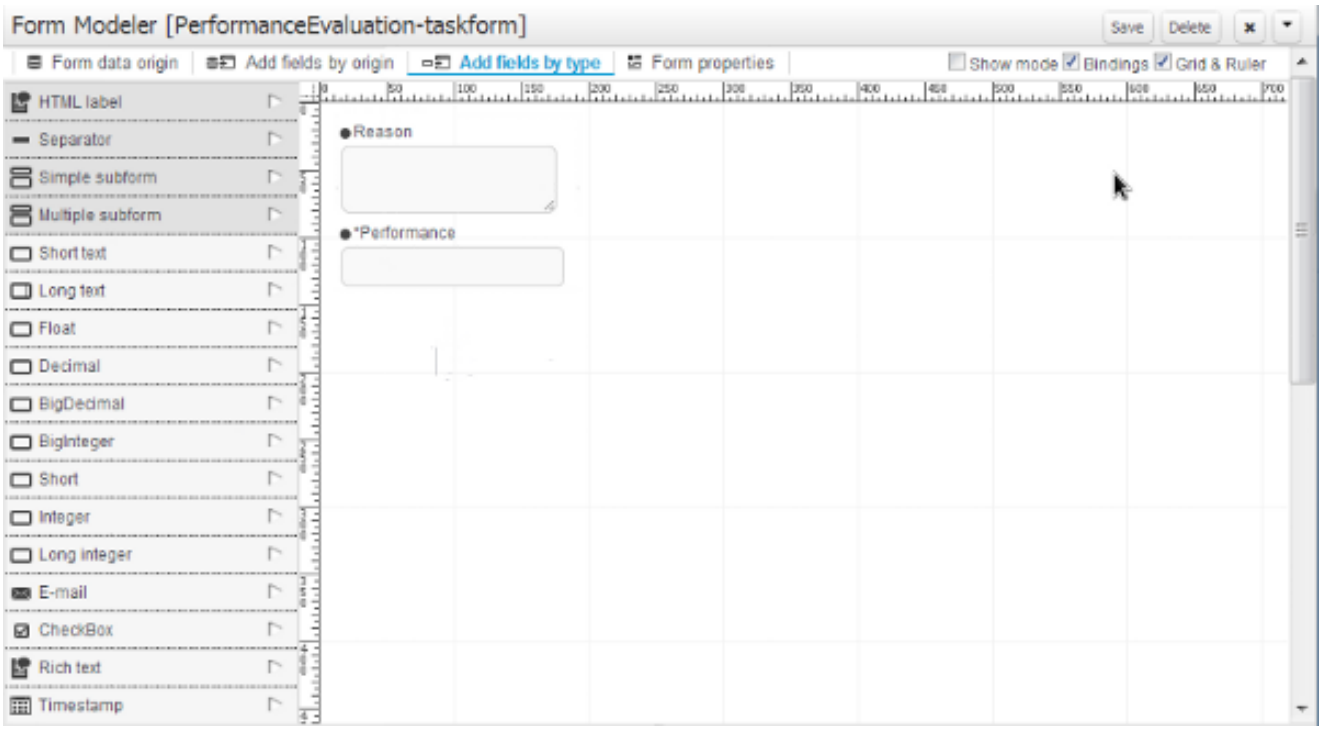


Figure 1.3. Form Modeler

Key features:

- Form Modeling WYSIWYG UI for forms
- Form autogeneration from data model / Java objects
- Data binding for Java objects

- Formula and expressions
- Customized forms layouts
- Forms embedding

The form modeler's user interfaces is aimed both at process analyst and developers for building and testing forms.

Developers or advanced used will also have some advanced features to customize form behavior and look&feel.

1.7. Process Instance and Task Management

Business processes can be managed through a web-based management console. It is targeted towards business users and its main features are the following:

- Process instance management: the ability to start new process instances, get a list of running process instances, visually inspect the state of a specific process instances.
- Human task management: being able to get a list of all your current tasks (either assigned to you or that you might be able to claim), and completing tasks on your task list (using customizable task forms).

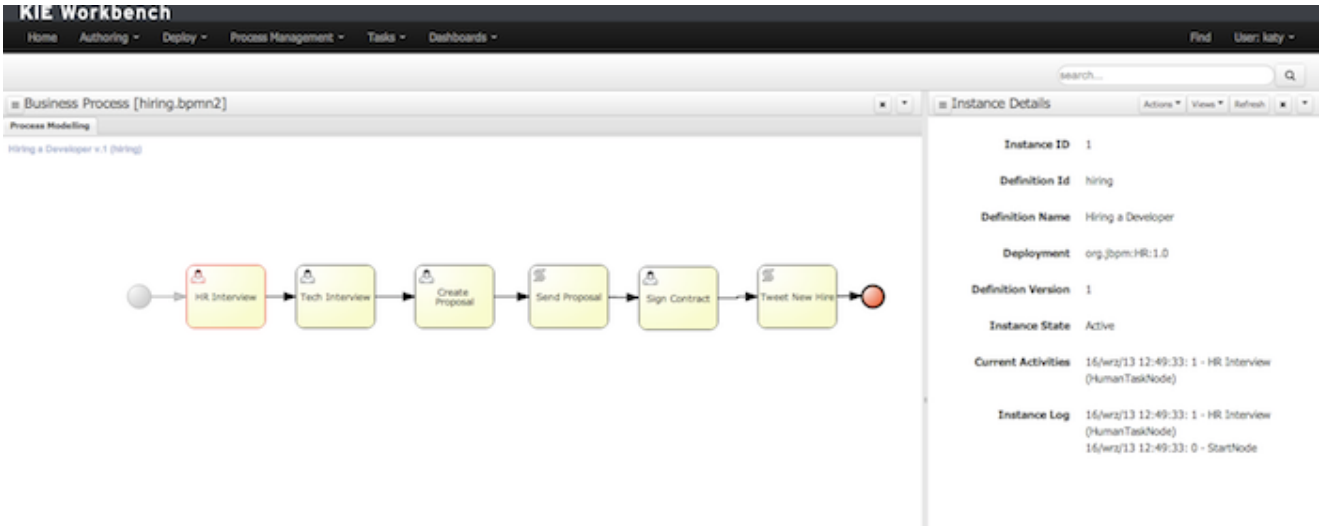


Figure 1.4. Managing your process instances

1.8. Business Activity Monitoring

As of version 6.0, jBPM comes with a full-featured BAM tooling which allows non-technical users to visually compose business dashboards. With this brand new module, to develop business activity monitoring and reporting solutions on top of jBPM has never been so easy!

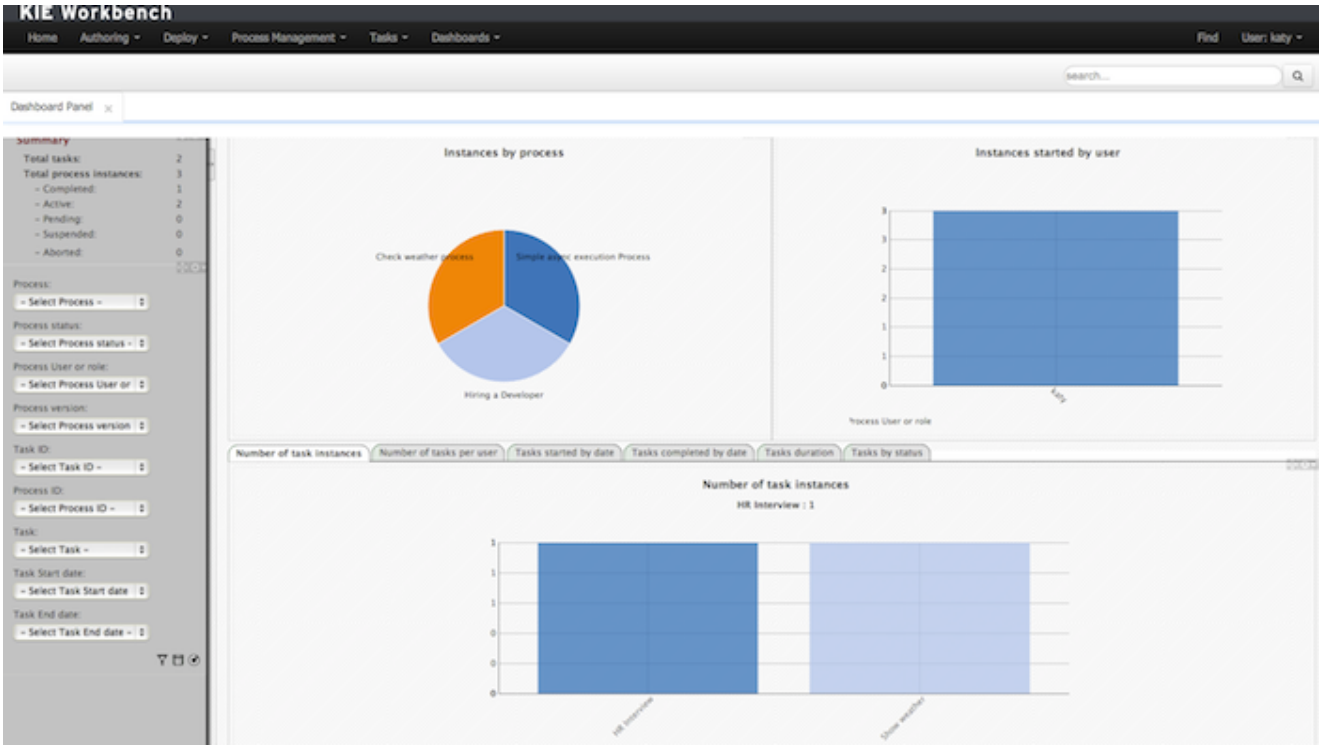


Figure 1.5. Business Activity Monitoring

Key features:

- Visual configuration of dashboards (Drag'n'drop).
- Graphical representation of KPIs (Key Performance Indicators).
- Configuration of interactive report tables.
- Data export to Excel and CSV format.
- Filtering and search, both in-memory or SQL based.
- Data extraction from external systems, through different protocols.
- Granular access control for different user profiles.
- Look'n'feel customization tools.
- Pluggable chart library architecture.
- Chart libraries provided: NVD3 & OFC2.

Target users:

- Managers / Business owners. Consumer of dashboards and reports.
- IT / System architects. Connectivity and data extraction.
- Analysts. Dashboard composition & configuration.

To get further information about the new and noteworthy BAM capabilities of jBPM please read the chapter *Business Activity Monitoring*.

1.9. Workbench

The workbench is the web-based application that combines all of the above web-based tools into one configurable solution.

It supports the following:

- A repository service to store your business processes and related artefacts, using a Git repository, which supports versioning, remote accessing (as a file system), and using REST services.
- A web-based user interface to manage your business processes, targeted towards business users; it also supports the visualization (and editing) of your artifacts (the web-based editors like designer, data and form modeler are integrated here), but also categorisation, build and deployment, etc..
- Collaboration features to have multiple actors (for example business users and developers) work together on the same project.

Workbench application covers complete life cycle of BPM projects starting at authoring phase, going through implementation, execution and monitoring.

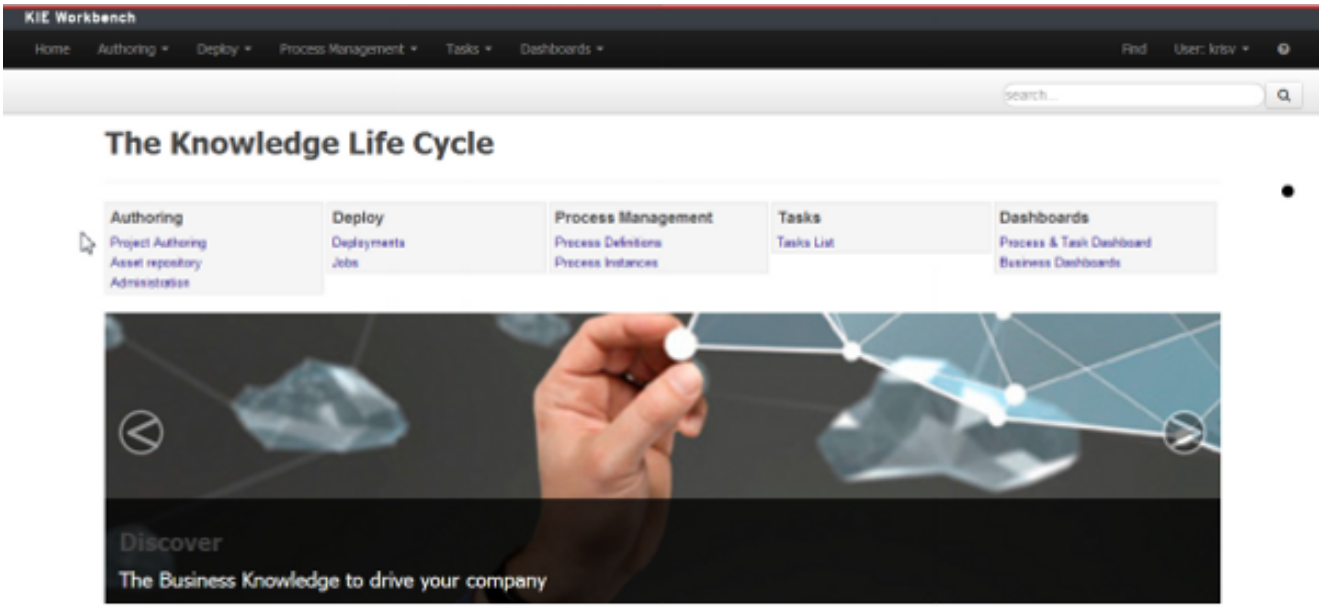


Figure 1.6. KIE workbench application

1.10. Eclipse Developer Tools

The Eclipse-based tools are a set of plugins to the Eclipse IDE and allow you to integrate your business processes in your development environment. It is targeted towards developers and has some wizards to get started, a graphical editor for creating your business processes (using drag and drop) and a lot of advanced testing and debugging capabilities.

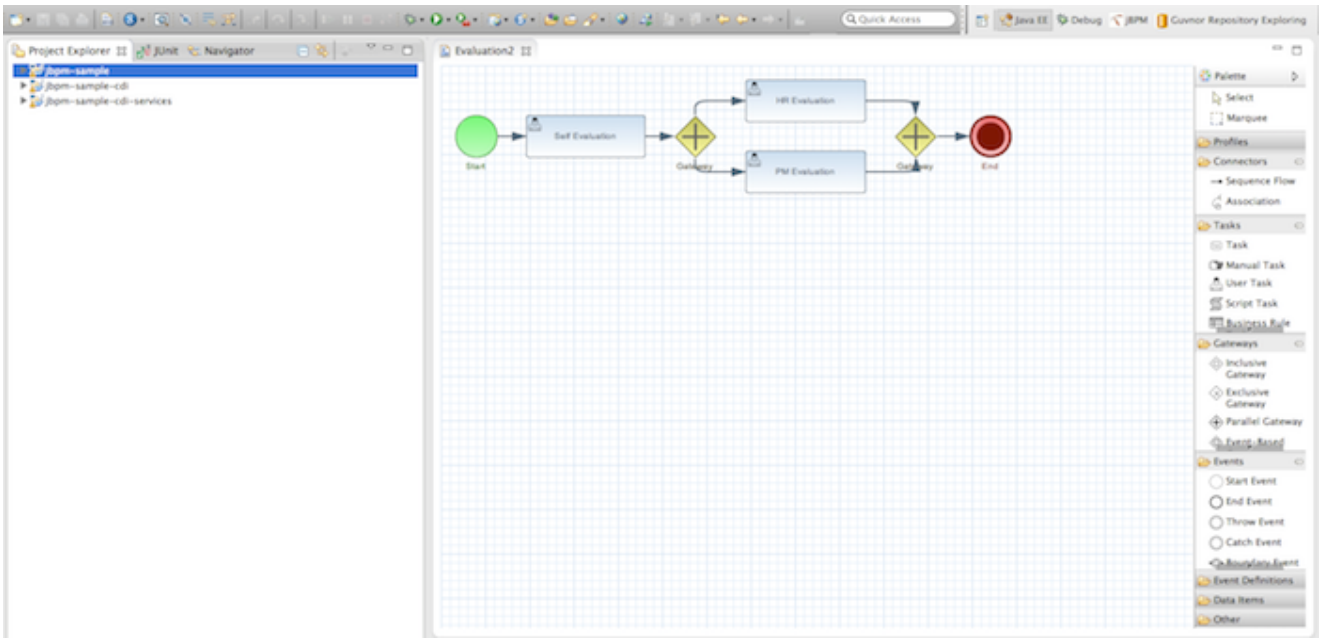


Figure 1.7. Eclipse editor for creating BPMN2 processes

It includes the following features:

- Wizard for creating a new jBPM project
- A graphical editor for BPMN 2.0 processes
- The ability to plug in your own domain-specific nodes
- Validation
- Runtime support (so you can select which version of jBPM you would like to use)
- Graphical debugging to see all running process instances of a selected session, to visualize the current state of one specific process instance, etc.

Chapter 2. Getting Started

2.1. Downloads

All releases can be downloaded from *SourceForge* [https://sourceforge.net/projects/jbpm/files/]. Select the version you want to download and then select which artifact you want:

- bin: all the jBPM binaries (JARs) and their dependencies
- src: the sources of the core components
- docs: the documentation
- examples: some jBPM examples, can be imported into Eclipse
- installer: the jbpm-installer, downloads and installs a demo setup of jBPM
- installer-full: the jbpm-installer, downloads and installs a demo setup of jBPM, already contains a number of dependencies prepackages (so they don't need to be downloaded separately)

2.2. Getting Started

If you like to take a quick tutorial that will guide you through most of the components using a simple example, take a look at the Installer chapter. This will teach you how to download and use the installer to create a demo setup, including most of the components. It uses a simple example to guide you through the most important features. Screencasts are available to help you out as well.

If you like to read more information first, the following chapters first focus on the core engine (API, BPMN 2.0, etc.). Further chapters will then describe the other components and other more complex topics like domain-specific processes, flexible processes, etc. After reading the core chapters, you should be able to jump to other chapters that you might find interesting.

You can also start playing around with some examples that are offered in a separate download. Check out the examples chapter to see how to start playing with these.

After reading through these chapters, you should be ready to start creating your own processes and integrate the engine with your application. These processes can be started from the installer or be started from scratch.

2.3. Community

Here are a lot of useful links part of the jBPM community:

- A feed of *blog entries* [http://planet.jboss.org/view/feed.seam?name=jbossjbpm] related to jBPM

- The *#jbossjbpm Twitter account* [http://twitter.com/jbossjbpm].
- A *user forum* [http://www.jboss.com/index.html?module=bb&op=viewforum&f=217] for asking questions and giving answers
- *A JIRA bug tracking system* [https://jira.jboss.org/jira/browse/JBPM] for bugs, feature requests and roadmap
- A *continuous build server* [https://hudson.jboss.org/hudson/job/jBPM/] for getting the *latest snapshots* [https://hudson.jboss.org/hudson/job/jBPM/lastSuccessfulBuild/artifact/jbpm-distribution/target/]

Please feel free to join us in our IRC channel at chat.freenode.net #jbpm. This is where most of the real-time discussion about the project takes place and where you can find most of the developers most of their time as well. Don't have an IRC client installed? Simply go to *http://webchat.freenode.net/*, input your desired nickname, and specify #jbpm. Then click login to join the fun.

2.4. Sources

2.4.1. License

The jBPM code itself is using the Apache License v2.0.

Some other components we integrate with have their own license:

- The new Eclipse BPMN2 plugin is Eclipse Public License (EPL) v1.0.
- The web-based designer is based on Oryx/Wapama and is MIT License
- The Drools project is Apache License v2.0.

2.4.2. Source code

jBPM now uses git for its source code version control system. The sources of the jBPM project can be found here (including all releases starting from jBPM 5.0-CR1):

https://github.com/droolsjbpm/jbpm

The source of some of the other components we integrate with can be found here:

- Other components related to the jBPM and Drools project can be found *here* [https://github.com/droolsjbpm].
- The new Eclipse BPMN2 plugin can be found *here* [https://git.eclipse.org/c/bpmn2-modeler/org.eclipse.bpmn2-modeler.git].

- The web-based designer can be found *here* [https://github.com/droolsjbpm/jbpm-designer]
- The kie workbench can be found *here* [https://github.com/droolsjbpm/kie-wb-distribution-wars] note this is an aggregate of other projects (drools-wb, jbpm-console-ng)

2.4.3. Building from source

If you're interested in building the source code, contributing, releasing, etc. make sure to read this *README* [https://github.com/droolsjbpm/droolsjbpm-build-bootstrap/blob/master/README.md].

2.5. Getting Involved

We are often asked "How do I get involved". Luckily the answer is simple, just write some code and submit it :) There are no hoops you have to jump through or secret handshakes. We have a very minimal "overhead" that we do request to allow for scalable project development. Below we provide a general overview of the tools and "workflow" we request, along with some general advice.

If you contribute some good work, don't forget to blog about it :)

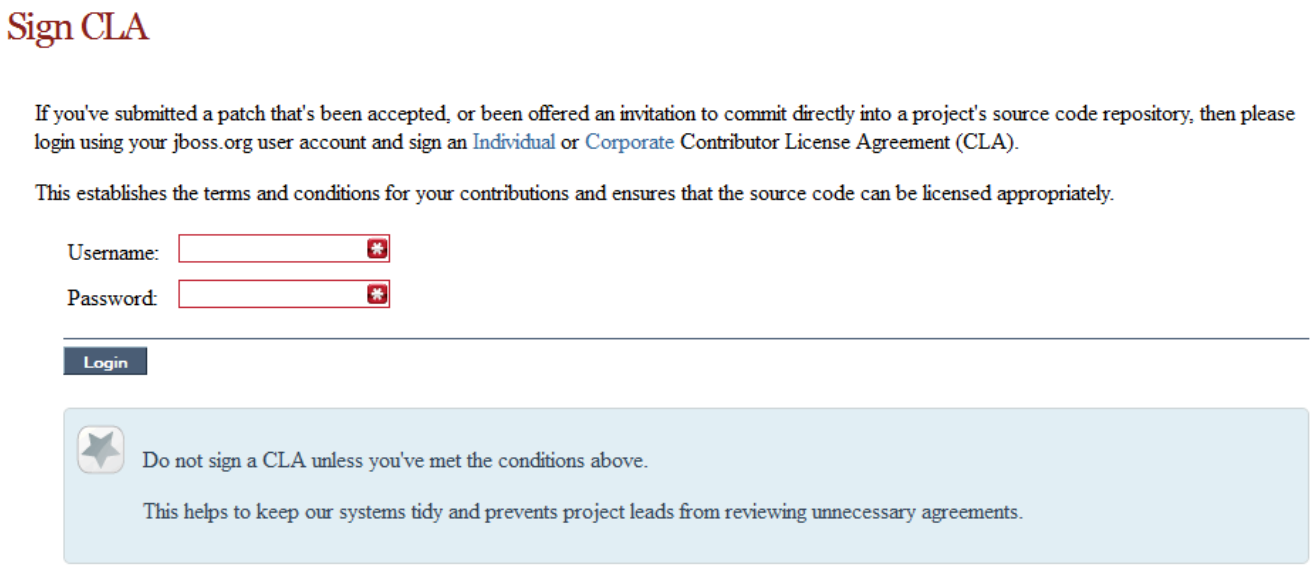
2.5.1. Sign up to jboss.org

Signing to jboss.org will give you access to the JBoss wiki, forums and JIRA. Go to *http://www.jboss.org/* and click "Register".

2.5.2. Sign the Contributor Agreement

The only form you need to sign is the contributor agreement, which is fully automated via the web. As the image below says "This establishes the terms and conditions for your contributions and ensures that source code can be licensed appropriately"

https://cla.jboss.org/



2.5.3. Submitting issues via JIRA

To be able to interact with the core development team you will need to use JIRA, the issue tracker. This ensures that all requests are logged and allocated to a release schedule and all discussions captured in one place. Bug reports, bug fixes, feature requests and feature submissions should all go here. General questions should be undertaken at the mailing lists.

Minor code submissions, like format or documentation fixes do not need an associated JIRA issue created.

https://issues.jboss.org/browse/JBRULES [???](Drools)

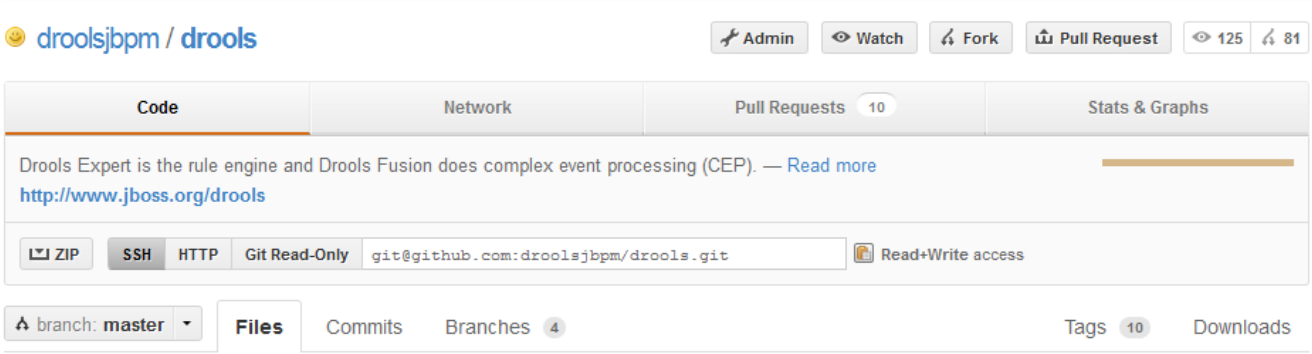
https://issues.jboss.org/browse/JBPM

https://issues.jboss.org/browse/GUVNOR

2.5.4. Fork GitHub

With the contributor agreement signed and your requests submitted to JIRA you should now be ready to code :) Create a GitHub account and fork any of the Drools, jBPM or Guvnor repositories. The fork will create a copy in your own GitHub space which you can work on at your own pace. If you make a mistake, don't worry blow it away and fork again. Note each GitHub repository provides you the clone (checkout) URL, GitHub will provide you URLs specific to your fork.

https://github.com/droolsjbpm



2.5.5. Writing Tests

When writing tests, try and keep them minimal and self contained. We prefer to keep the DRL fragments within the test, as it makes for quicker reviewing. If their are a large number of rules

then using a String is not practical so then by all means place them in separate DRL files instead to be loaded from the classpath. If your tests need to use a model, please try to use those that already exist for other unit tests; such as Person, Cheese or Order. If no classes exist that have the fields you need, try and update fields of existing classes before adding a new class.

There are a vast number of tests to look over to get an idea, MiscTest is a good place to start.

https://github.com/droolsjbpm/drools/blob/master/drools-compiler/src/test/java/org/drools/integrationtests/MiscTest.java [https://github.com/droolsjbpm]

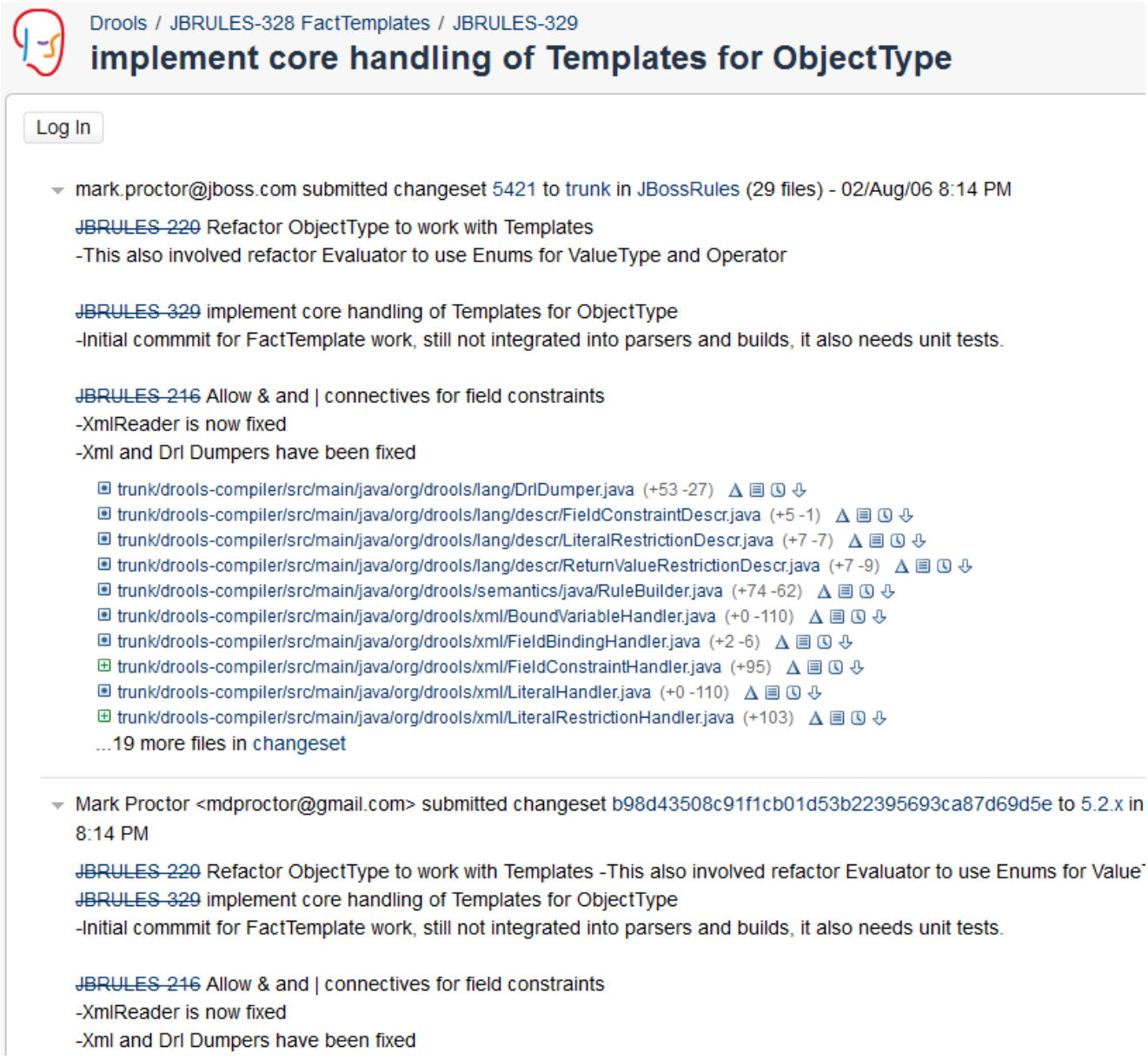
```
557     @Test
558     public void testEvalWithBigDecimal() throws Exception {
559         String str = "";
560         str += "package org.drools \n";
561         str += "import java.math.BigDecimal; \n";
562         str += "global java.util.List list \n";
563         str += "rule rule1 \n";
564         str += "    dialect \"java\" \n";
565         str += "when \n";
566         str += "    $bd : BigDecimal() \n";
567         str += "    eval( $bd.compareTo( BigDecimal.ZERO ) > 0 ) \n";
568         str += "then \n";
569         str += "    list.add( $bd ); \n";
570         str += "end \n";
571
572         KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();
573
574         kbuilder.add( ResourceFactory.newByteArrayResource( str.getBytes() ),
575                       ResourceType.DRL );
576
577         if ( kbuilder.hasErrors() ) {
578             logger.warn( kbuilder.getErrors().toString() );
579         }
580         assertFalse( kbuilder.hasErrors() );
581
582         KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
583         kbase.addKnowledgePackages( kbuilder.getKnowledgePackages() );
584
585         StatefulKnowledgeSession ksession = createKnowledgeSession(kbase);
586         List list = new ArrayList();
587         ksession.setGlobal( "list",
588                             list );
589         ksession.insert( new BigDecimal( 1.5 ) );
590
591         ksession.fireAllRules();
592
593         assertEquals( 1,
594                       list.size() );
595         assertEquals( new BigDecimal( 1.5 ),
596                       list.get( 0 ) );
597     }
598
```

2.5.6. Commit with Correct Conventions

When you commit, make sure you use the correct conventions. The commit must start with the JIRA issue id, such as JBRULES-220. This ensures the commits are cross referenced via JIRA, so we can see all commits for a given issue in the same place. After the id the title of the issue should come next. Then use a newline, indented with a dash, to provide additional information

related to this commit. Use an additional new line and dash for each separate point you wish to make. You may add additional JIRA cross references to the same commit, if it's appropriate. In general try to avoid combining unrelated issues in the same commit.

Don't forget to rebase your local fork from the original master and then push your commits back to your fork.

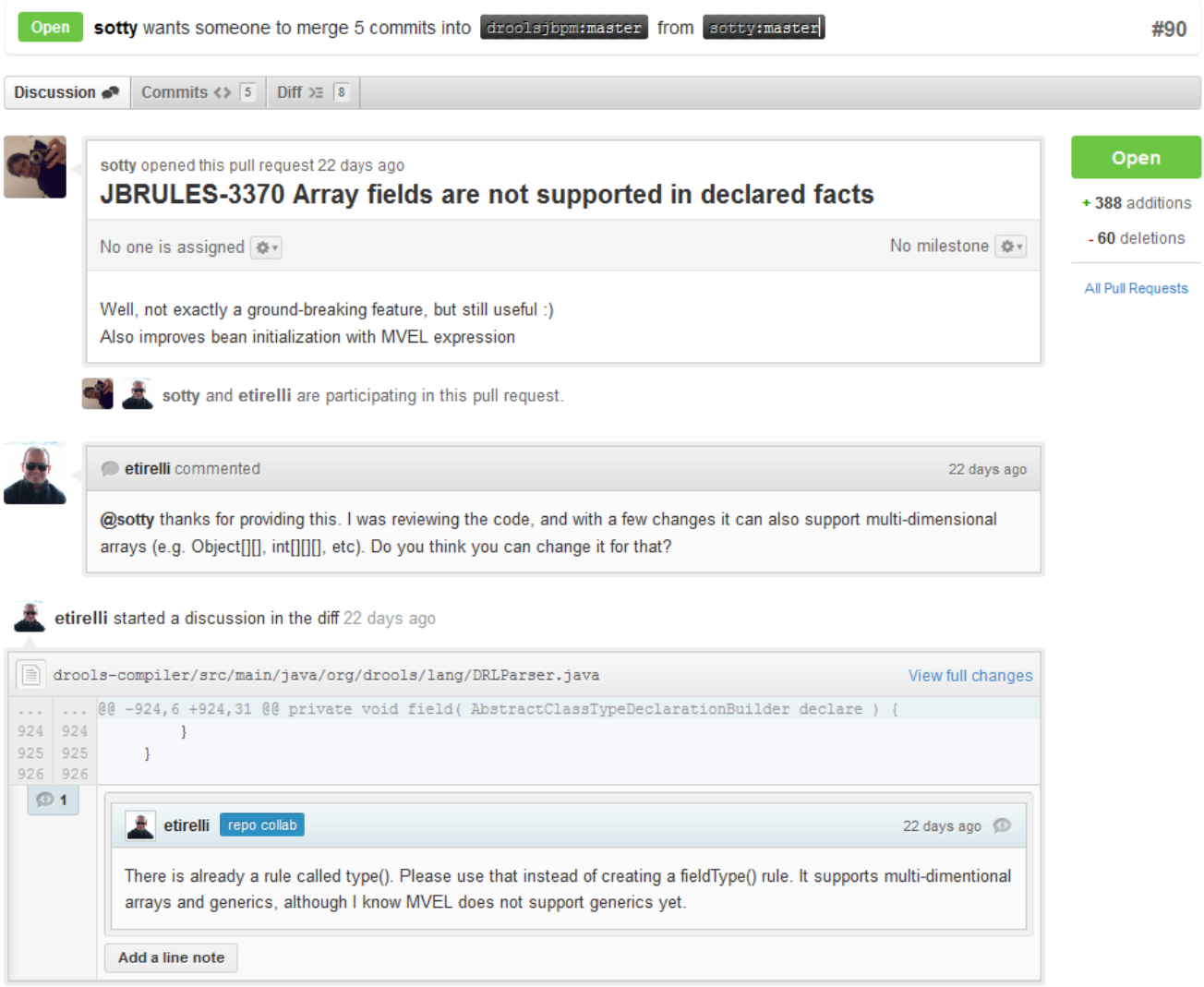


2.5.7. Submit Pull Requests

With your code rebased from original master and pushed to your personal GitHub area, you can now submit your work as a pull request. If you look at the top of the page in GitHub for your work area their will be a "Pull Request" button. Selecting this will then provide a gui to automate the submission of your pull request.

The pull request then goes into a queue for everyone to see and comment on. Below you can see a typical pull request. The pull requests allow for discussions and it shows all associated commits and the diffs for each commit. The discussions typically involve code reviews which provide helpful suggestions for improvements, and allows for us to leave inline comments on specific parts of the code. Don't be disheartened if we don't merge straight away, it can often take several revisions before we accept a pull request. Luckily GitHub makes it very trivial to go back to your code, do some more commits and then update your pull request to your latest and greatest.

It can take time for us to get round to responding to pull requests, so please be patient. Submitted tests that come with a fix will generally be applied quite quickly, where as just tests will often way until we get time to also submit that with a fix. Don't forget to rebase and resubmit your request from time to time, otherwise over time it will have merge conflicts and core developers will general ignore those.



2.6. What to do if I encounter problems or have questions?

You can always contact the jBPM community for assistance.

IRC: #jbpm at chat.freenode.net

jBPM User Forum [http://community.jboss.org/en/jbpm?view=discussions]

Chapter 3. jBPM Installer

3.1. Prerequisites

This script assumes you have Java JDK 1.6+ (set as JAVA_HOME), and Ant 1.7+ installed. If you don't, use the following links to download and install them:

Java: *http://java.sun.com/javase/downloads/index.jsp*

Ant: *http://ant.apache.org/bindownload.cgi*

Tip

To check whether Java and Ant are installed correctly, type the following commands inside a command prompt:

java -version

ant -version

This should return information about which version of Java and Ant you are currently using.

3.2. Downloading the Installer

First of all, you need to *download* [https://sourceforge.net/projects/jbpm/files/jBPM%206/] the installer and unzip it to your local file system. There are two versions

- full installer - which already contains a lot of the dependencies that are necessary during the installation
- minimal installer - which only contains the installer and will download all dependencies

In general, it is probably best to download the full installer: jBPM-{version}-installer-full.zip

You can also find the latest snapshot release here (only minimal installer) here:

https://hudson.jboss.org/jenkins/job/jBPM/lastSuccessfulBuild/artifact/jbpm-distribution/target/
[https://hudson.jboss.org/jenkins/job/jBPM/lastSuccessfulBuild/artifact/jbpm-distribution/target/]

3.3. Demo Setup

The easiest way to get started is to simply run the installation script to install the demo setup. The demo install will setup all the web tooling (on top of JBoss AS) and Eclipse tooling in a pre-

configured setup. Go into the jbpm-installer folder where you unzipped the installer and (from a command prompt) run:

```
ant install.demo
```

This will:

- Download JBoss AS
- Configure and deploy the web tooling
- Download Eclipse
- Install the Drools and jBPM Eclipse plugin
- Install the Eclipse BPMN 2.0 Modeler

Running this command could take a while (REALLY, not kidding, we are for example downloading an Eclipse installation, even if you downloaded the full installer).

Tip

The script always shows which file it is downloading (you could for example check whether it is still downloading by checking the whether the size of the file in question in the jbpm-installer/lib folder is still increasing). If you want to avoid downloading specific components (because you will not be using them or you already have them installed somewhere else), check below for running only specific parts of the demo or directing the installer to an already installed component.

Once the demo setup has finished, you can start playing with the various components by starting the demo setup:

```
ant start.demo
```

This will:

- Start H2 database server
- Start the JBoss AS
- Start Eclipse

Once everything is started, you can start playing with the Eclipse and web tooling, as explained in the following sections.

If you only want to try out the web tooling and do not wish to download and install the Eclipse tooling, you can use these alternative commands:

```
ant install.demo.noeclipse
ant start.demo.noeclipse
```

Similarly, if you only want to try out the Eclipse tooling and do not wish to download and install the web tooling, you can use these alternative commands:

```
ant install.demo.eclipse
ant start.demo.eclipse
```

Now continue with the 10-minute tutorials. Once you're done playing and you want to shut down the demo setup, you can use:

```
ant stop.demo
```

If at any point in time would like to start over with a clean demo setup - meaning all changes you did inside the web tooling and/or saved in the database will be lost, you can run the following command (after which you can run the installer again from scratch, note that this cannot be undone):

```
ant clean.demo
```

3.4. 10-Minute Tutorial using the Workbench

Open up the process management console:

http://localhost:8080/jbpm-console

Note

It could take a minute to start up the AS and web application. If the web page doesn't show up after a while, make sure you don't have a firewall blocking that port, or another application already using the port 8080. You can always take a look at the server log jbpm-installer/jboss-as-7.1.1.Final/standalone/log/server.log

Log in, using krisv / krisv as username / password.

Using a prebuilt Evaluation example, the *following screencast* [http://people.redhat.com/kverlaen/jbpm6F-installer-console.swf] gives an overview of how to manage your process instances. It shows you:

- How to build and deploy a process
- How to start a new process instance
- How to look up the current status of a running process instance
- How to look up your tasks
- How to complete a task
- How to generate reports to monitor your process execution

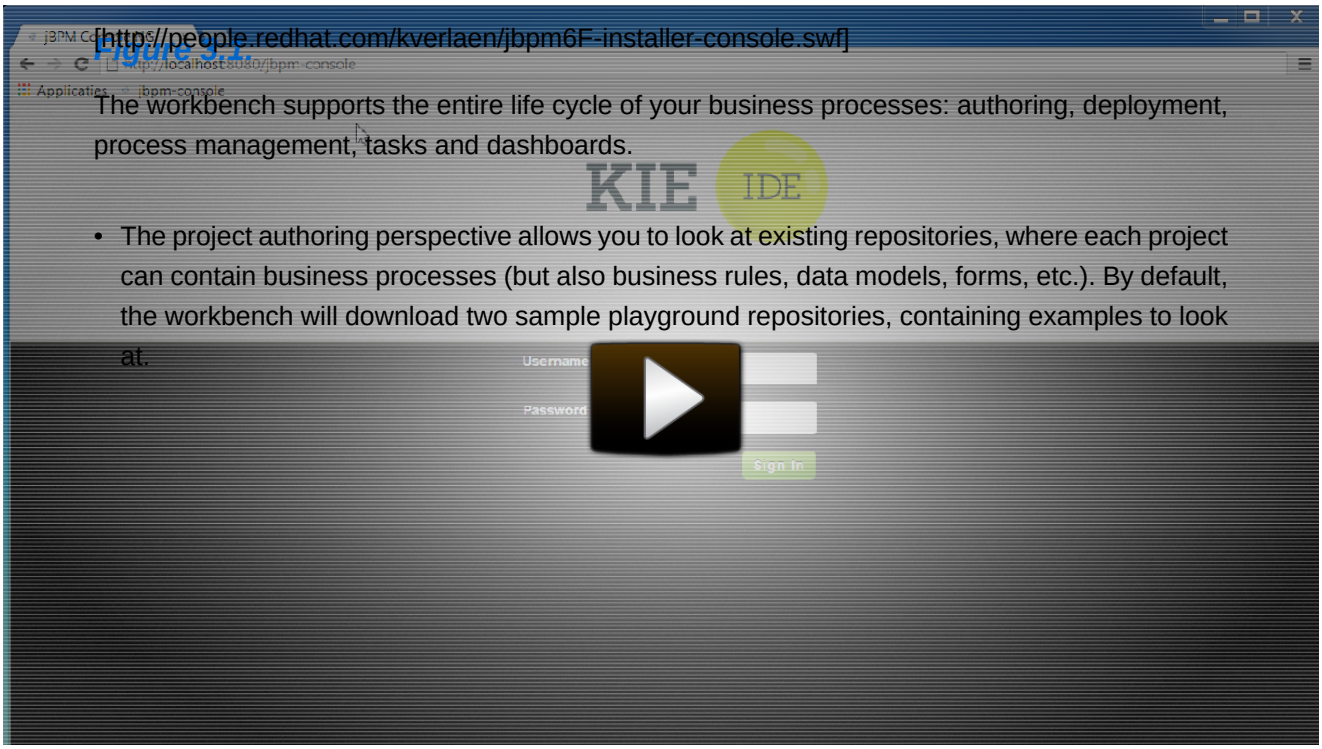


Figure 3.1: [http://people.redhat.com/kverlaen/jbpm6F-installer-console.swf]

The workbench supports the entire life cycle of your business processes: authoring, deployment, process management, tasks and dashboards.

- The project authoring perspective allows you to look at existing repositories, where each project can contain business processes (but also business rules, data models, forms, etc.). By default, the workbench will download two sample playground repositories, containing examples to look at.

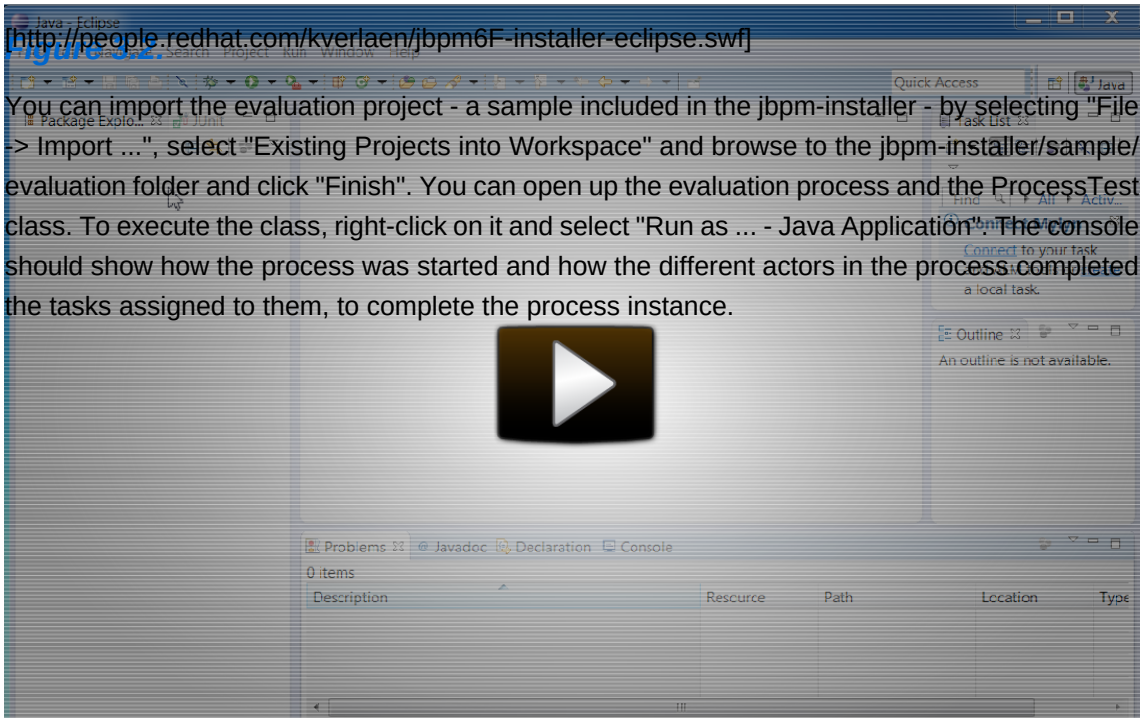
  - In this screencast, the Evaluation project inside the jbpm-playground repository is used.
- The project explorer shows all available artefacts:
  - evaluation: business process describing the evaluation process as a sequence of tasks
  - evaluation-taskform: process form to start the evaluation process
  - PerformanceEvaluation-taskform: task form to perform the evaluation tasks
- To make a process available for execution, you need to successfully build and deploy it first. To do so, open up the Project Editor (from the Tools menu) and click Build & Deploy.
- To manage your process definitions and instances, click on the "Process Management" menu option at the top menu bar an select one of available options depending on you interest:
  - Process Definitions - lists all available process definitions
  - Process Instances - lists all active process instances (allows to show completed, aborted as well by changing filter criteria)
- Process definitions panel allow you to start a new process instance by clicking on the "Play" button. The process form (as defined in the project) will be shown, where you need to fill in the necessary information to start the process. In this case, you need to fill the user you want to start an evaluation for (in this case use "krisv") and a reason for the request, after which you can complete the form. Some details about the process instance that was just started will be shown in the process instance details panel. From there you can access additional details:
  - Process model - to visualize current state of the process
  - Process variables - to see current values of process variables

  The process instance that you just started is first requiring a self-evaluation of the user and is waiting until the user has completed this task.
- To see the tasks that have been assigned to you, choose the "Tasks" menu option on the top bar and select "Task List" (you may need to click refresh to update your task view). The personal tasks table should show a "Performance Evaluation" task reserved for you. After starting the task, you can complete the task, which will open up the task form related to this task. You can fill in the necessary data and then complete the form and close the window. After completing the task, you could check the "Process Instances" once more to check the progress of your process instance. You should be able to see that the process is now waiting for your HR manager and project manager to also perform an evaluation. You could log in as "john" / "john" and "mary" / "mary" to complete these tasks.
- After starting and/or completing a few process instances and human tasks, you can generate a report of what has happened so far. Under "Dashboards", select "Process & Task Dashboard". This is a set of see predefined charts that allow users to spot what is going on in the system. Charts can be fully customized as well, as explained in the Business Activity Monitoring chapter.

3.5. 10-Minute Tutorial using Eclipse

The *following screencast* [http://people.redhat.com/kverlaen/jbpm6F-installer-eclipse.swf] gives an overview of how to use the Eclipse tooling. It shows you:

- How to import and execute the evaluation sample project
  - Import the evaluation project (included in the jbpm-installer)
  - Open the Evaluation.bpmn process
  - Open the com.sample.ProcessTest Java class
  - Execute the ProcessTest class to run the process
- How to create a new jBPM project (including sample process and JUnit test)



[http://people.redhat.com/kverlaen/jbpm6F-installer-eclipse.swf]

Figure 3.2.

You can import the evaluation project - a sample included in the jbpm-installer - by selecting "File -> Import ...", select "Existing Projects into Workspace" and browse to the jbpm-installer/sample/evaluation folder and click "Finish". You can open up the evaluation process and the ProcessTest class. To execute the class, right-click on it and select "Run as ... - Java Application". The console should show how the process was started and how the different actors in the process completed the tasks assigned to them, to complete the process instance.

You could also create a new project using the jBPM project wizard. The sample projects contain a process and an associated Java file to start the process. Select "File - New ... - Project ..." and under the "jBPM" category, select "jBPM project" and click "Next". Give the project a name and click "Next". You can choose from a simple HelloWorld example or a slightly more advanced example using persistence and human tasks. If you select the latter and click Finish, you should see a new project containing a "sample.bpmn" process and a "com.sample.ProcessTest" JUnit test class. You can open the BPMN2 process by double-clicking it. To execute the process, right-click on ProcessTest.java and select "Run As - Java Application".

3.6. Configuration

3.6.1. Playgrounds

The workbench by default brings two sample playground repositories (by cloning the jbpm-playground repository hosted on GitHub). In cases where this is not wanted (access to Internet might not be available or there might be a need to start with a completely clean installation of the workbench) this default behavior can be turned off. To do so, change the following system property in the start.jboss target to false in the build.xml:

```
-Dorg.kie.demo=false
```

Note that this will create a completely empty version of the workbench. To be able to start modeling processes, the following elements need to be created first:

- Organizational unit
- Repository (new or clone existing one)
- Project

3.6.2. Workbench Authentication

The workbench web application is using the "default" security domain for authenticating and authorizing users (as specified in the WEB-INF/jboss-web.xml inside the WARs).

The application server is configured by default to use properties files for specifying users. Note that this is for demo purposes only (as passwords and roles are stored in simple property files). The security domain is configured in the standalone.xml configuration file as follows:

```
<security-domain name="other" cache-type="default">
  <authentication>
    <login-module code="UsersRoles" flag="required">
      <module-option name="usersProperties" value="${jboss.server.config.dir}/
users.properties"/>
```

```
      <module-option name="rolesProperties" value="${jboss.server.config.dir}/
roles.properties"/>
    </login-module>
  </authentication>
</security-domain>
```

By default, these configuration files contain the following users:

Table 3.1. Default users

Name	Password	Workbench roles	Task roles
admin	admin	admin,analyst	
krisv	krisv	admin,analyst	
john	john	analyst	Accounting,PM
mary	mary	analyst	HR
sales-rep	sales-rep	analyst	sales
jack	jack	analyst	IT
katy	katy	analyst	HR
salaboy	salaboy	admin,analyst	IT,HR,Accounting

Authentication can be customized by editing the authentication and configuration files in the jbpm-installer/auth folder and/or by changing the standalone-*.xml files in the jbpm-installer folder. Note that you need to rerun the installer to make sure the modified files are copied and picked correctly.

3.6.3. Using your own database

3.6.3.1. Introduction

By default, the jbpm-installer uses an H2 database for persisting runtime data. In this section we will:

1. modify the persistence settings for runtime persistence of process instance state
2. test the startup with our new settings!

You will need a local instance of a database, in this case we will use MySQL.

First though, let's look at the persistence setup that jBPM uses. In the demo, and in general, there are following types of persistent entities used by jBPM:

- entities used for saving the actual session and process instance information - aka runtime data.
- entities used for logging and generating reports - aka audit log.
- entities used by the task service.

“persistent entities” in this context, are Java classes that represent information in the database.

3.6.3.2. Database setup

In the MySQL database used in this quickstart, create a single user:

- user/schema "jbpm" with password "jbpm" (which will be used to persist all entities)

If you end up using different names for your user/schemas, please make a note of where we insert "jbpm" in the configuration files.

If you want to try this quickstart with *another* database, a section at the end of this quickstart describes what you may need to modify.

3.6.3.3. Configuration

The following files define the persistence settings for the jbpm-installer demo:

- jbpm-installer/db/jbpm-persistence-JPA2.xml
- Application server configuration
  - standalone-*.xml

Tip

There are multiple standalone.xml files available (depending on whether you are using JBoss AS 7.1.1 or JBoss EAP 6.1.1 and whether you are running the normal or full profile). The full profile is required to use the JMS component for remote integration. Best practice is to update all standalone.xml files to have consistent setup but most important is to have standalone-full-as-7.1.1.Final.xml properly configured as this is used by default by the installer.

Do the following:

- Disable H2 default database and enable MySQL database in build.properties

```
# default is H2
# H2.version=1.3.168
# db.name=h2
# db.driver.jar.name=${db.name}.jar
#     db.driver.download.url=http://repo1.maven.org/maven2/com/h2database/h2/
${H2.version}/h2-${H2.version}.jar
#mysql
```

```
db.name=mysql
db.driver.module.prefix=com/mysql
db.driver.jar.name=${db.name}-connector-java.jar
db.driver.download.url=https://repository.jboss.org/nexus/service/local/
repositories/central/content/mysql/mysql-connector-java/5.1.18/mysql-
connector-java-5.1.18.jar
```

- `db/jbpm-persistence-JPA2.xml`:

  This is the JPA persistence file that defines the persistence settings used by jBPM for both the process engine information, the logging/BAM information and task service.

  In this file, you will have to change the name of the hibernate dialect used for your database.

  The original line is:

```
<property name="hibernate.dialect" value="org.hibernate.dialect.H2Dialect"/>
```

  In the case of a MySQL database, you need to change it to:

```
<property                                                name="hibernate.dialect"
 value="org.hibernate.dialect.MySQLDialect"/>
```

  For those of you who decided to use another database, a list of the available hibernate dialect classes can be found *here* [http://docs.jboss.org/hibernate/core/3.3/reference/en-US/html/session-configuration.html#configuration-optional-dialects].

- `standalone.xml`:

  This file is the configuration for the standalone JBoss application server. When the installer installs the demo, it copies these files to the `standalone/configuration` directory in the JBoss server directory.

  We need to change the datasource configuration in `standalone.xml` so that the jBPM process engine can use our MySQL database

  The original file contains the following lines:

```
<datasource  jndi-name="java:jboss/datasources/jbpmDS"  enabled="true"  use-
java-context="true" pool-name="H2DS">
   <connection-url>jdbc:h2:tcp://localhost/runtime/jbpm-demo</connection-url>
    <driver>h2</driver>
    <pool></pool>
```

```
    <security>
       <user-name>sa</user-name>
       <password></password>
    </security>
</datasource>
<drivers>
    <driver name="h2" module="com.h2database.h2">
       <xa-datasource-class>org.h2.jdbcx.JdbcDataSource</xa-datasource-class>
    </driver>
</drivers>
```

Change the lines to the following:

```
<datasource   jndi-name="java:jboss/datasources/jbpmDS"   pool-name="MySQLDS"
 enabled="true" use-java-context="true">
    <connection-url>jdbc:mysql://localhost:3306/jbpm</connection-url>
    <driver>mysql</driver>
    <pool></pool>
    <security>
       <user-name>jbpm</user-name>
       <password>jbpm</password>
    </security>
</datasource>
<drivers>
    <driver name="mysql" module="com.mysql">
       <xa-datasource-class>com.mysql.jdbc.jdbc2.optional.MysqlXADataSource</
xa-datasource-class>
    </driver>
</drivers>
```

- `Starting the demo`

  We've modified all the necessary files at this point. Now would be a good time to make sure your database is started up as well!

  The installer script copies this file into the jbpm-console WAR before the WAR is installed on the server. If you have already run the installer, it is recommended to stop the installer and clean it first using

```
ant stop.demo
```

  and

```
ant clean.demo
```

before continuing.

Run

```
ant install.demo
```

to (re)install the wars and copy the necessary configuration files. Once you've done that, (re)start the demo using

```
ant start.demo
```

.

- `Problems?`

  If this isn't working for you, please try the following:

  - Please double check the files you've modified: I *wrote* this, but still made mistakes when changing files!
  - Please make sure that you don't secretly have another (unmodified) instance of JBoss AS running.
  - If neither of those work (and you're using MySQL), please do then let us know.

3.6.3.4. Using a different database

If you decide to use a different database with this demo, you need to remember the following when going through the steps above:

- Change the JDBC URLs, usernames and passwords, and Hibernate dialect lines to match your database information in the configuration files mentioned above.
- In order to make sure your driver will be correctly installed in the JBoss AS 7 server, you can do one of two things. Both ways are explained *here* [https://community.jboss.org/wiki/DataSourceConfigurationinAS7].
  - *Install* [https://community.jboss.org/wiki/DataSourceConfigurationinAS7#Installing_a_JDBC_driver_as_a_module] the driver JAR as a *module*, which is what the install script does.

- *Otherwise, you can modify and install* [https://community.jboss.org/wiki/DataSourceConfigurationinAS7#Installing_a_JDBC_driver_as_a_deployment] the downloaded JAR as a *deployment*. In this case you will have to copy the JAR yourself to the `standalone/deployments` directory.

If you choose to install driver as JBoss module, please do the following:

- Disable default H2 driver properties

```
# default is H2
# H2.version=1.3.168
# db.name=h2
# db.driver.jar.name=${db.name}.jar
#   db.driver.download.url=http://repo1.maven.org/maven2/com/h2database/h2/
${H2.version}/h2-${H2.version}.jar
```

- Copy one of the example configs (mysql or postgresql)

```
#postgresql
db.name=postgresql
db.driver.module.prefix=org/postgresql
db.driver.jar.name=${db.name}-jdbc.jar
db.driver.download.url=https://repository.jboss.org/nexus/content/
repositories/thirdparty-uploads/postgresql/postgresql/9.1-902.jdbc4/
postgresql-9.1-902.jdbc4.jar
```

- Change the `db.name` property in `build.properties` to the name of the downloaded jdbc driver JAR you placed in `db/drivers`.
- Change the `<driver>` information in the `<datasource>` section of `standalone.xml` so that it refers to the name of your driver module (see next step). For example:

```
<driver>postgresql</driver>
```

- Further on in `standalone.xml` is the `<drivers>` section of the `<datasources>` (note the plural: drivers, datasources). We need to do the following with this file:
  - Change the name of the driver to match the name in the last step,
  - Give an appropriate name to the module,

- And fill in the correct name of the XA datasource class to use.

For example:

```
<drivers>
  <driver name="postgresql" module="org.postgresql">
      <xa-datasource-class>org.postgresql.xa.PGXADataSource</xa-datasource-
class>
  </driver>
</drivers>
```

- Change the `db.driver.module.prefix` property in `build.properties` to the same "value" you used for the module name in `standalone.xml`. In the example above, I used "`org.postgresql`" which means that I should then use `org/postgresql` for the `db.driver.module.prefix` property.

- Lastly, you'll have to create the `db/${db.name}_module.xml` file. As an example you can use db/mysql_module.xml, so just make a copy of it and:

  - Change the name of the *module* to match the `db.driver.module.prefix` property above

  - Change the name of the module resource to the name of the JDBC driver JAR that was downloaded.

  The top of the original file looks like this:

```
<module xmlns="urn:jboss:module:1.0" name="com.mysql">
   <resources>
     <resource-root path="mysql-connector-java.jar"/>
   </resources>
```

Change those lines to look like this, for example:

```
<module xmlns="urn:jboss:module:1.0" name="org.postgresql">
   <resources>
     <resource-root path="postgresql-9.1-902.jdbc4.jar"/>
   </resources>
```

3.6.4. jBPM database schema scripts (DDL scripts)

By default the demo setup makes use of Hibernate auto DDL generation capabilities to build up the complete database schema, including all tables, sequences, etc. This might not always be

welcomed (by your database administrator), and thus the installer provides DDL scripts for most popular databases.

Table 3.2. DDL scripts

Database name	Location
db2	jbpm-installer/db/ddl-scripts/db2
derby	jbpm-installer/db/ddl-scripts/derby
h2	jbpm-installer/db/ddl-scripts/h2
hsqldb	jbpm-installer/db/ddl-scripts/hsqldb
mysql5	jbpm-installer/db/ddl-scripts/mysql5
mysqlinnodb	jbpm-installer/db/ddl-scripts/mysqlinnodb
oracle	jbpm-installer/db/ddl-scripts/oracle
postgresql	jbpm-installer/db/ddl-scripts/postgresql
sqlserver	jbpm-installer/db/ddl-scripts/sqlserver
sqlserver2008	jbpm-installer/db/ddl-scripts/sqlserver2008

DDL scripts are provided for both jBPM and Quartz schemas although Quartz schema DDL script is only required when the timer service should be configured with Quartz database job store. See the section on timers for additional details.

This can be used to initially create the database schema, but it can also serve as the basis for any\ optimization that needs to be applied - such as indexes, etc.

3.6.5. jBPM installer script

jBPM installer ant script performs most of the work automatically and usually does not require additional attention but in case it does, here is a list of available targets that might be needed to perform some of the steps manually.

Table 3.3. jBPM installer available targets

Target	Description
clean.db	cleans up database used by jBPM demo (applies only to H2 database)
clean.demo	cleans up entire installation so new installation can be performed
clean.demo.noeclipse	same as clean.demo but does not remove Eclipse
clean.eclipse	removes Eclipse and its workspace
clean.generated.ddl	removes DDL scripts generated if any
clean.jboss	removes application server with all its deployments

Target	Description
clean.jboss.repository	removes repository content for demo setup (guvnor Maven repo, niogit, etc)
download.dashboard	downloads jBPM dashboard component (BAM)
download.db.driver	downloads DB driver configured in build.properties
download.ddl.dependencies	downloads all dependencies required to run DDL script generation tool
download.droolsjbpm.eclipse	downloads Drools and jBPM Eclipse plugin
download.eclipse	downloads Eclipse distribution
download.jboss	downloads JBoss Application Server
download.jBPM.bin	downloads jBPM binary distribution (jBPM libs and its dependencies)
download.jBPM.console	downloads jBPM console for JBoss AS
install.dashboard.into.jboss	installs jBPM dashboard into JBoss AS
install.db.files	installs DB driver as JBoss module
install.demo	installs complete demo environment
install.demo.eclipse	installs Eclipse with all jBPM plugins, no server installation
install.demo.noeclipse	similar to install.demo but skips Eclipse installation
install.dependencies	installs custom libraries (such as work item handlers, etc) into the jBPM console
install.droolsjbpm-eclipse.into.eclipse	installs droolsjbpm Eclipse plugin into Eclipse
install.eclipse	install Eclipse IDE
install.jboss	installs JBoss AS
install.jBPM-console.into.jboss	installs jBPM console application into JBoss AS

3.7. Frequently Asked Questions

Some common issues are explained below.

Q: What if the installer complains it cannot download component X?

A: Are you connected to the Internet? Do you have a firewall turned on? Do you require a proxy? It might be possible that one of the locations we're downloading the components from is temporarily offline. Try downloading the components manually (possibly from alternate locations) and put them in the jbpm-installer/lib folder.

Q: What if the installer complains it cannot extract / unzip a certain JAR/WAR/zip?

A: If your download failed while downloading a component, it is possible that the installer is trying to use an incomplete file. Try deleting the component in question from the jbpm-installer/lib folder and reinstall, so it will be downloaded again.

Q: What if I have been changing my installation (and it no longer works) and I want to start over again with a clean installation?

A: You can use ant clean.demo to remove all the installed components, so you end up with a fresh installation again.

Q: I sometimes see exceptions when trying to stop or restart certain services, what should I do?

A: If you see errors during shutdown, are you sure the services were still running? If you see exceptions during restart, are you sure the service you started earlier was successfully shutdown? Maybe try killing the services manually if necessary.

Q: Something seems to be going wrong when running Eclipse but I have no idea what. What can I do?

A: Always check the consoles for output like error messages or stack traces. You can also check the Eclipse Error Log for exceptions. Try adding an audit logger to your session to figure out what's happening at runtime, or try debugging your application.

Q: Something seems to be going wrong when running the a web-based application like the jbpm-console. What can I do?

A: You can check the server log for possible exceptions: jbpm-installer/jboss-as-{version}/standalone/log/server.log (for JBoss AS7).

For all other questions, try contacting the jBPM community as described in the Getting Started chapter.

Chapter 4. Examples

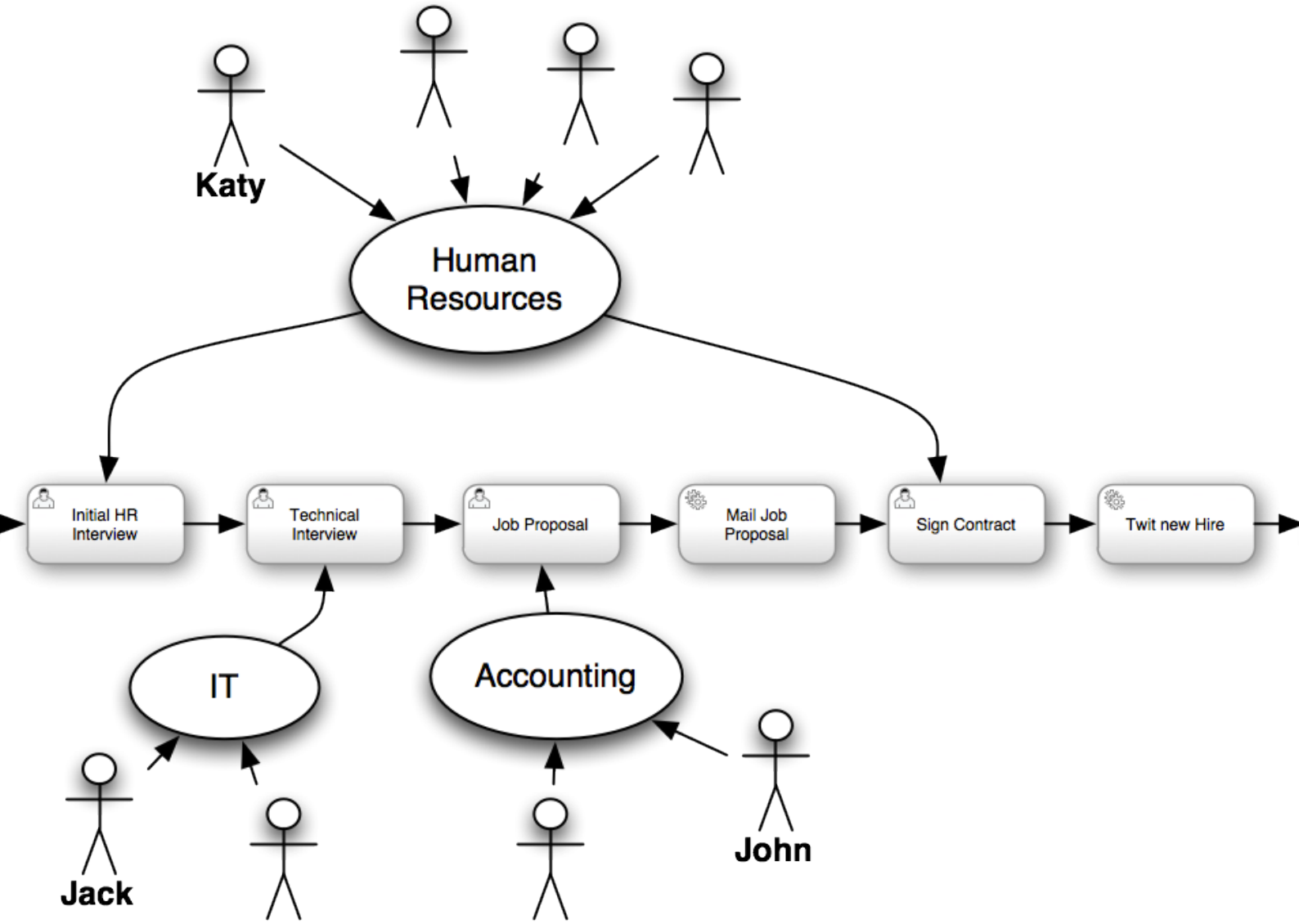
4.1. Introduction

The web-based workbench by default will install two sample repositories that contain various sample projects that help you getting started. This section shows different examples that can be found in the jbpm-playground repository (also available here: *https://github.com/droolsjbpm/jbpm-playground*). All these examples are high level and business oriented.

If you want to contribute with these examples please get in touch with any member of the jBPM/ Drools Team.

4.2. Human Resources Example

Let’s imagine for a second that you work for a Software company that works with several projects and from time to time the company wants to hire new developers. So, which employees, Departments and Systems are required to Hire a new Developer in your company? Trying to answering these questions will help you to define your business process. The following figure, represents how does this process works for Acme Inc. We can clearly see that three Departments are involved: Human Resources, IT and Accounting teams are involved. Inside our company we have Katy from the Human Resources Team, Jack on the IT team and John from the Accounting team involved. Notice that there are other people inside each team, but we will be using Katy, Jack and John to demonstrate how to execute the business process.



Notice that there are 6 activities defined inside this business process, 4 of them are User Tasks, which means that will be handled by people. The other two are Service Tasks, which means an interaction with another system will be required.

The process diagram is self explanatory, but just in case and to avoid confusions this is what is supposed to happen for each instance of the process that is started a particular candidate:

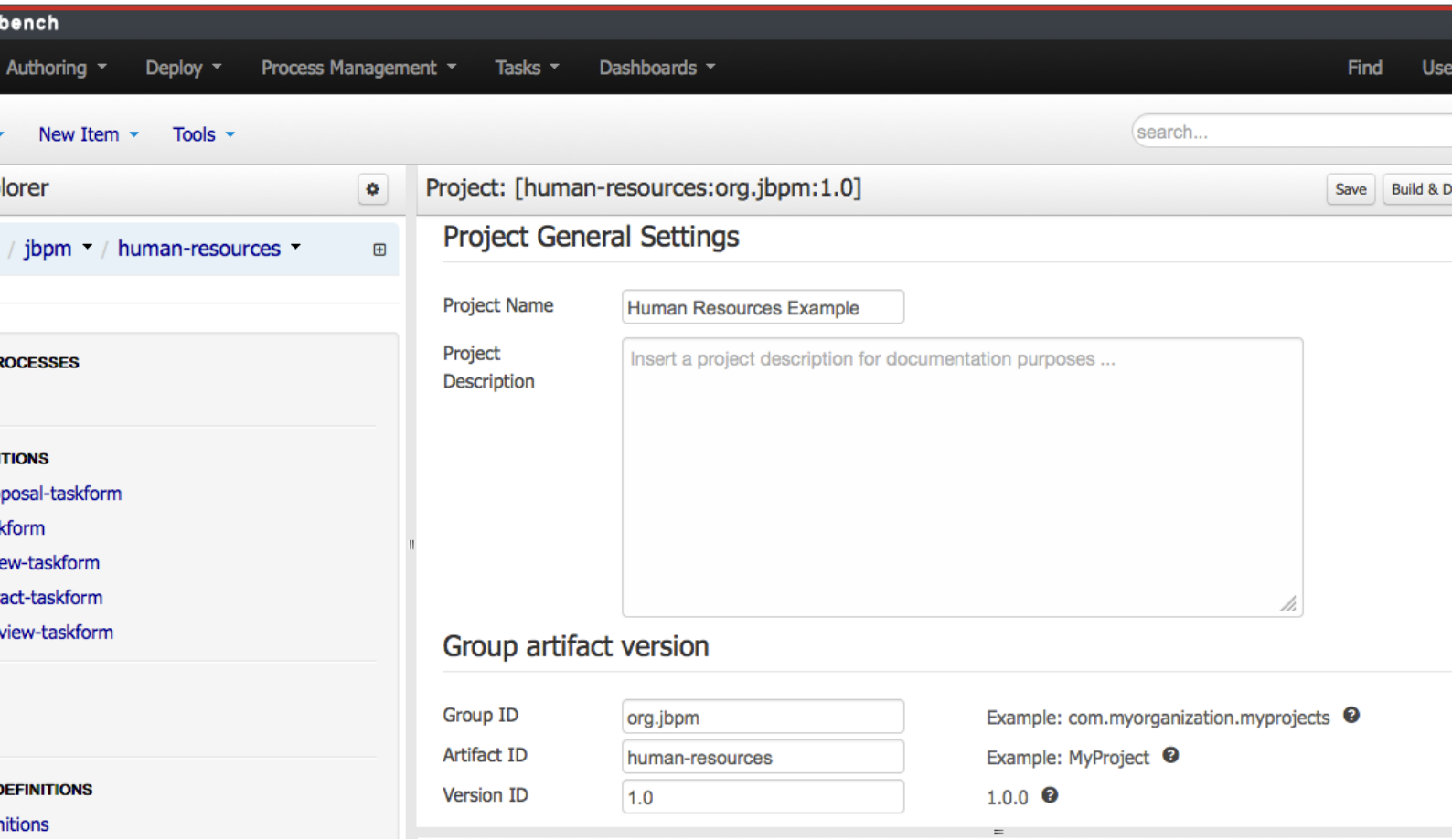
- The Human Resources Team perform the initial interview to the candidate to see if he/she fits the profile that the company is looking for.
- The IT Department perform a technical interview to evaluate the candidate skills and experience.

- Based on output of the Human Resources and IT teams, the accounting team create a Job Proposal which includes the yearly salary for the candidate. The proposal is created based on the output of both of the interviews (Human Resources and Technical).
- As soon as the proposal has being created it is automatically sent to the candidate via email.
- If the candidate accept the proposal, a new meeting is created with someone from the Human Resource team to sign the contract.
- If everything goes well, as soon as the process is notified that the candidate was hired, the system will automatically post a tweet about the new Hire using the Twitter service connector.

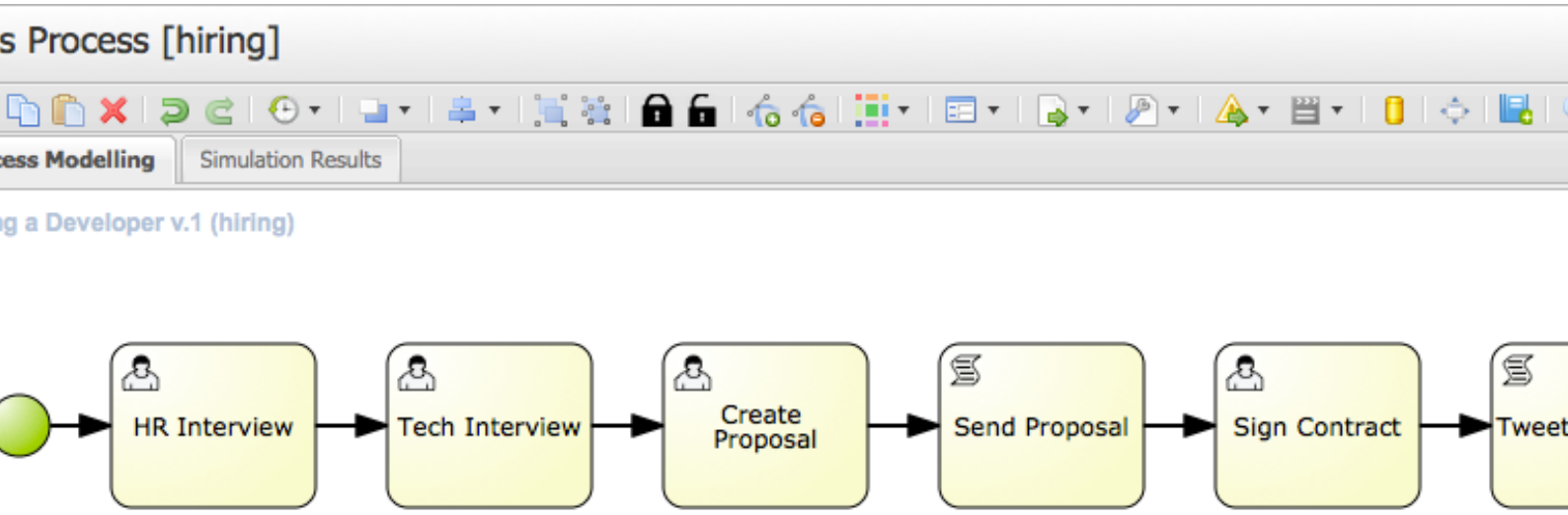
As you can see Jack, John and Katy will be performing the tasks for this example instance of the business process, but any person inside the company that have those Roles will be able to claim and interact with those tasks.

4.2.1. The KIE Project: human-resources

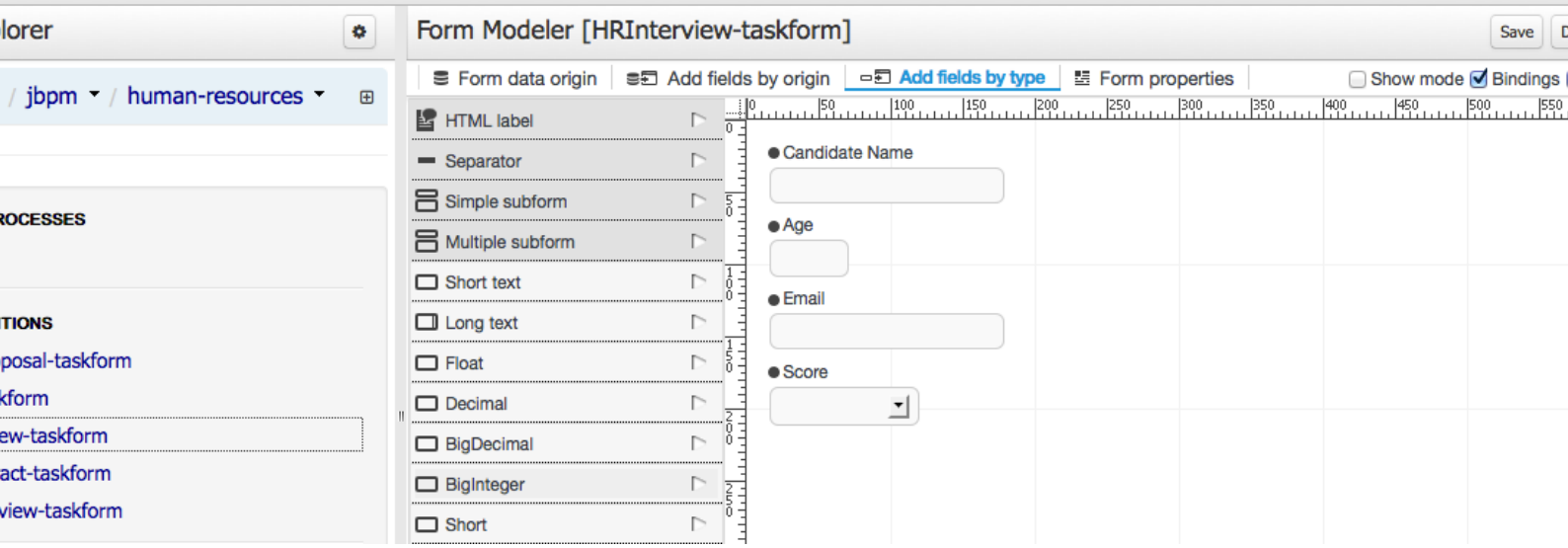
Let's take a look at the Project content in the Authoring Perspective:



As you can see it contains the **hiring.bpmn2** process and a set of forms for each human task. You can explore these knowledge assets by clicking on them. You will notice that different editors will open for different types of assets. If you click on the Business Process file you will be able to edit the process definition using the Process Designer:

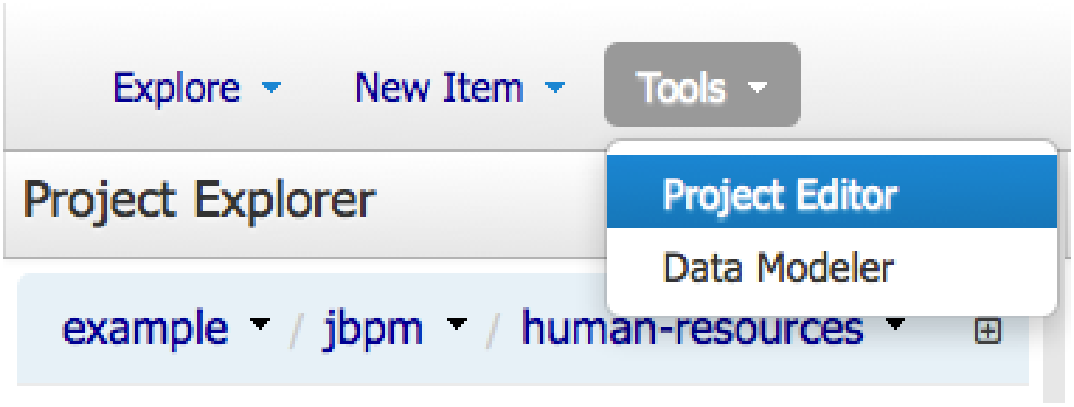


Feel free to inspect the forms as well. Notice that the Form Modeller will be opened and there you can edit the forms to fit your requirements.

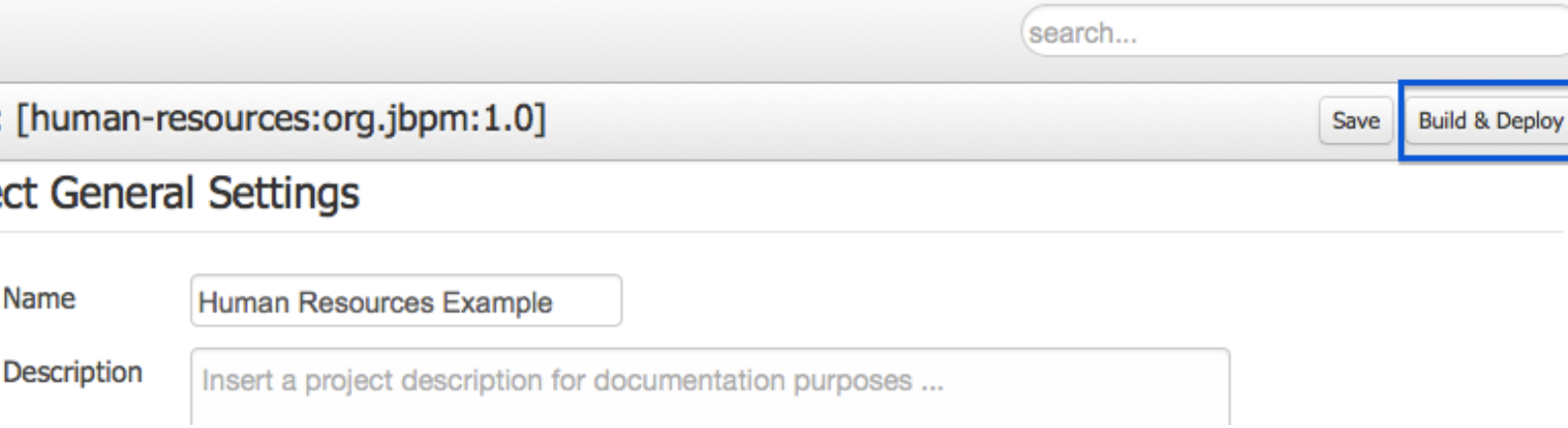


4.2.2. Building the Human Resources Example

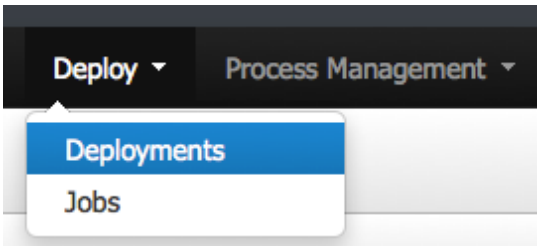
In order to build the Project so it gets available in the Process Definitions List you need to go to the Authoring Perspective and open the Project Editor panel:



Once you open the Project Editor, you will see on the top right corner of the panel the button called Build & Deploy. This button will allow you to create a new JAR artifact that will be deployed to the Runtime environment as a new Deployment Unit.

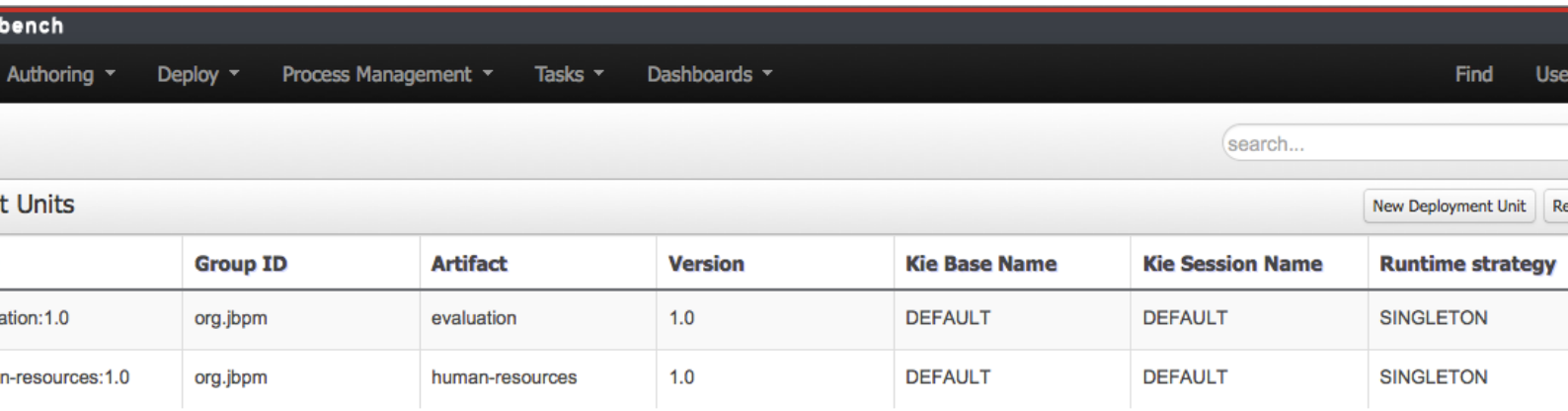


Once you get the visual notification that the project was built and deployed successfully you can go to the Deployments screen to verify that your artifact is there. In order to do that go to the top level menu called Deploy -> Deployments



In the Deployments screen you will find all the deployed units. By default when you Build & Deploy a project from the Project Editor, it will be automatically deployed using the default configurations. That is Singleton Strategy, the default KIE Base and the default KIE Session will be used.

If you want a more advanced deployment, that is selecting a different strategy or using non defaults KIE Base or KIE Session you will be able to undeploy and re-deploy your artifacts using their GAV and selecting non default options.

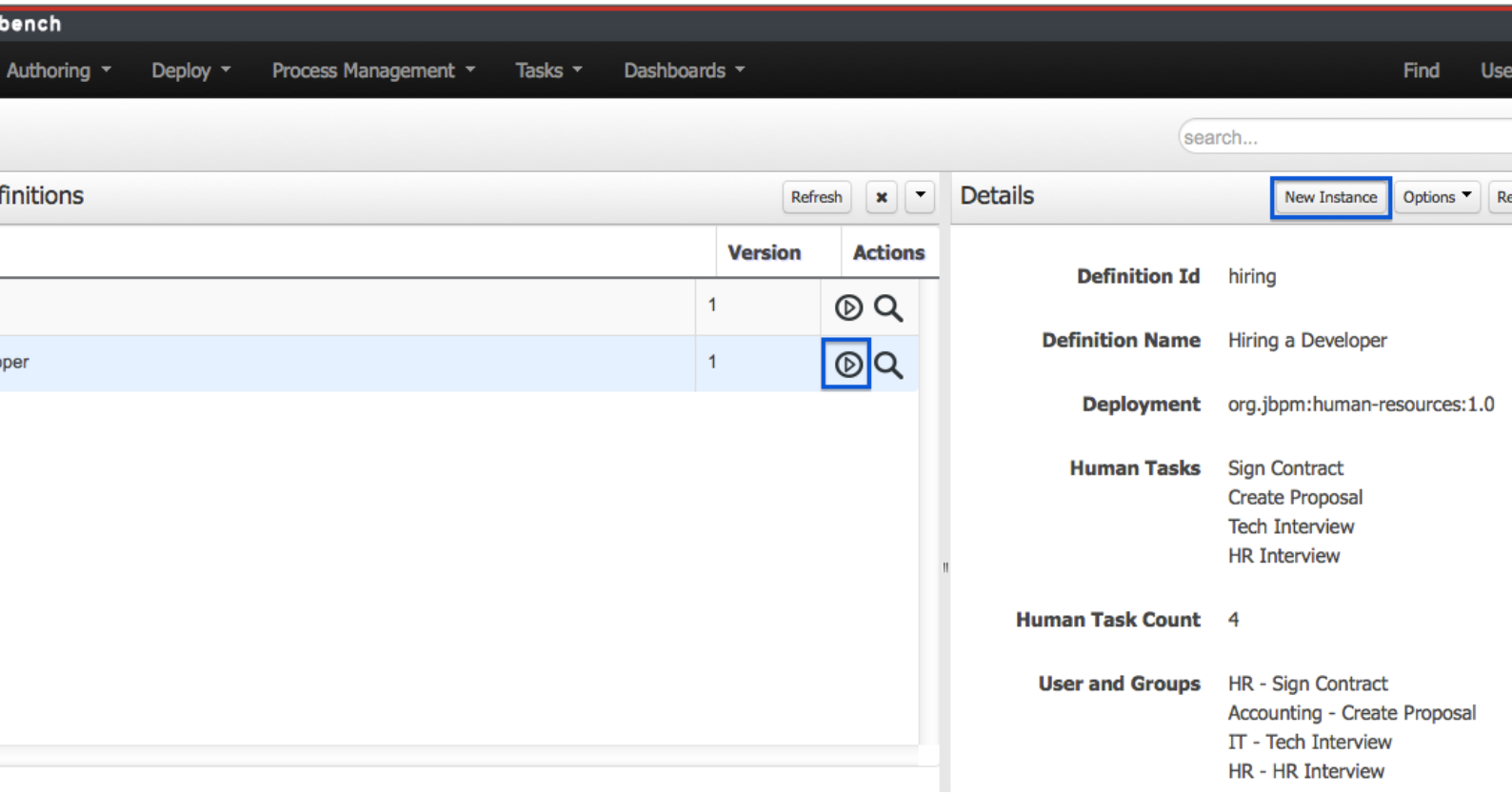


Once your artifact that contains the process definition is deployed, the Process Definition will be available under Process Management -> Process Definitions.

4.2.3. Create a new Process Instance

In order to create new Process Instances you need to go to Process Management -> Process Definitions.

Here you will find all the available process definitions in the runtime environment. If you want to add new process definitions look at the previous sections where it is explained how to build and deploy KIE Projects.



You can start process instances using any of the two options highlighted in the previous screen.

In order to create a new process instance most of the processes will require you to fill in some information and for that a form will be displayed. For this specific use case the name of the candidate that we are interviewing is required:

If we hit the big Start button, the new process instance will be created and the first task of the process will be create for the Human Resources Team. Depending on the assigned roles of the user that you are using to create the process instance you will be able to see the created task or not. In order to see the first task of the process we will need to logout tot he application and log in as someone from the Human Resources team.

After starting the process you can go to the Task -> Tasks section to interact with the created human tasks. Notice that in order to see the tasks in the task lists you will need to belong to some specific user Groups. For example the HR Interview task will be visitable for any member of the HR group, the Tech Interview will be visible by any member of the IT Group.

4.3. Examples zip

A zip file of examples can also be downloaded from the downloads page, containing various examples that can be opened in the Eclipse-based Developers Tools. Simply download and unzip the examples artefact and import into your Eclipse workspace.

Part II. jBPM Core

Using the jBPM Core Engine

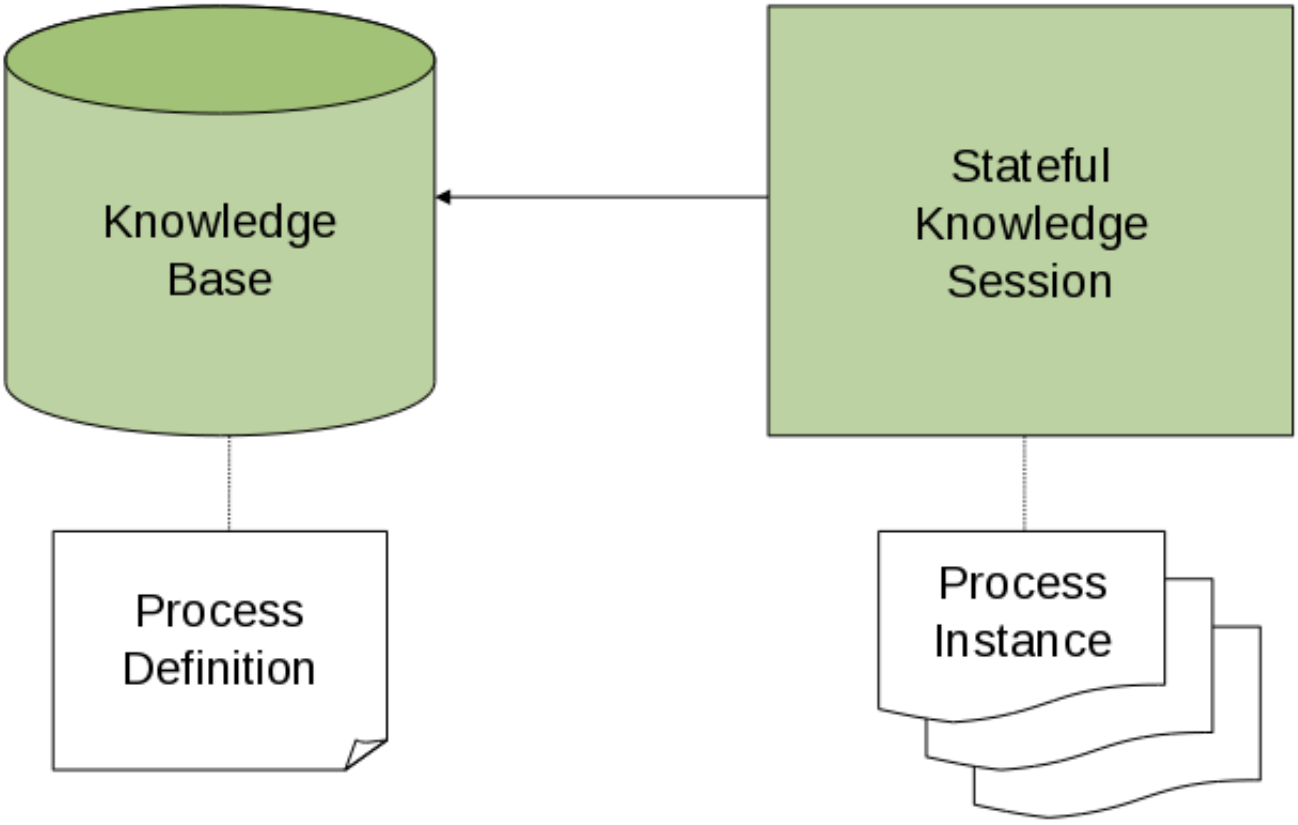
Chapter 5. Core Engine API

5.1. Overview

This chapter introduces the API you need to load processes and execute them. For more detail on how to define the processes themselves, check out the chapter on BPMN 2.0.

To interact with the process engine (for example, to start a process), you need to set up a session. This session will be used to communicate with the process engine. A session needs to have a reference to a knowledge base, which contains a reference to all the relevant process definitions. This knowledge base is used to look up the process definitions whenever necessary. To create a session, you first need to create a knowledge base, load all the necessary process definitions (this can be from various sources, like from classpath, file system or process repository) and then instantiate a session.

Once you have set up a session, you can use it to start executing processes. Whenever a process is started, a new process instance is created (for that process definition) that maintains the state of that specific instance of the process.



For example, imagine you are writing an application to process sales orders. You could then define one or more process definitions that define how the order should be processed. When starting up your application, you first need to create a knowledge base that contains those process definitions. You can then create a session based on this knowledge base so that, whenever a new sales order

comes in, a new process instance is started for that sales order. That process instance contains the state of the process for that specific sales request.

A knowledge base can be shared across sessions and usually is only created once, at the start of the application (as creating a knowledge base can be rather heavy-weight as it involves parsing and compiling the process definitions). Knowledge bases can be dynamically changed (so you can add or remove processes at runtime).

Sessions can be created based on a knowledge base and are used to execute processes and interact with the engine. You can create as many independent session as you need and creating a session is considered relatively lightweight. How many sessions you create is up to you. In general, most simple cases start out with creating one session that is then called from various places in your application. You could decide to create multiple sessions if for example you want to have multiple independent processing units (for example, if you want all processes from one customer to be completely independent from processes for another customer, you could create an independent session for each customer) or if you need multiple sessions for scalability reasons. If you don't know what to do, simply start by having one knowledge base that contains all your process definitions and create one session that you then use to execute all your processes.

The jBPM project has a clear separation between the API the users should be interacting with and the actual implementation classes. The public API exposes most of the features we believe "normal" users can safely use and should remain rather stable across releases. Expert users can still access internal classes but should be aware that they should know what they are doing and that the internal API might still change in the future.

As explained above, the jBPM API should thus be used to (1) create a knowledge base that contains your process definitions, and to (2) create a session to start new process instances, signal existing ones, register listeners, etc.

5.2. KieBase

The jBPM API allows you to first create a knowledge base. This knowledge base should include all your process definitions that might need to be executed by that session. To create a knowledge base, use a KieHelper to load processes from various resources (for example from the classpath or from the file system), and then create a new knowledge base from that helper. The following code snippet shows how to create a knowledge base consisting of only one process definition (using in this case a resource from the classpath).

```
KieHelper kieHelper = new KieHelper();
KieBase kieBase = kieHelper
                  .addResource(ResourceFactory.newClassPathResource("MyProcess.bpmn"))
                  .build();
```

The ResourceFactory has similar methods to load files from file system, from URL, InputStream, Reader, etc.

This is considered manual creation of knowledge base and while it is simple it is not recommended for real application development but more for try outs. Following you'll find recommended and much more powerful way of building knowledge base, knowledge session and more - RuntimeManager.

5.3. KieSession

Once you've loaded your knowledge base, you should create a session to interact with the engine. This session can then be used to start new processes, signal events, etc. The following code snippet shows how easy it is to create a session based on the previously created knowledge base, and to start a process (by id).

```
KieSession ksession = kbase.newKieSession();
ProcessInstance processInstance = ksession.startProcess("com.sample.MyProcess");
```

5.3.1. ProcessRuntime

The `ProcessRuntime` interface defines all the session methods for interacting with processes, as shown below.

```
/**
   * Start a new process instance.  The process (definition) that should
   * be used is referenced by the given process id.
   *
   * @param processId  The id of the process that should be started
   * @return the ProcessInstance that represents the instance of the process that was started
   */
  ProcessInstance startProcess(String processId);

  /**
   * Start a new process instance.  The process (definition) that should
   * be used is referenced by the given process id.  Parameters can be passed
   * to the process instance (as name-value pairs), and these will be set
   * as variables of the process instance.
   *
   * @param processId  the id of the process that should be started
   * @param parameters  the process variables that should be set when starting the process instance
   * @return the ProcessInstance that represents the instance of the process that was started
   */
  ProcessInstance startProcess(String processId,
                               Map<String, Object> parameters);

  /**
   * Signals the engine that an event has occurred. The type parameter defines
```

```
     * which type of event and the event parameter can contain additional information
     * related to the event.  All process instances that are listening to this type
     * of (external) event will be notified.  For performance reasons, this type of event
     * signaling should only be used if one process instance should be able to notify
     * other process instances. For internal event within one process instance, use the
     * signalEvent method that also include the processInstanceId of the process instance
     * in question.
     *
     * @param type the type of event
     * @param event the data associated with this event
     */
    void signalEvent(String type,
                     Object event);

    /**
     * Signals the process instance that an event has occurred. The type parameter defines
     * which type of event and the event parameter can contain additional information
     * related to the event.  All node instances inside the given process instance that
     * are listening to this type of (internal) event will be notified.  Note that the event
     * will only be processed inside the given process instance.  All other process instances
     * waiting for this type of event will not be notified.
     *
     * @param type the type of event
     * @param event the data associated with this event
     * @param processInstanceId the id of the process instance that should be signaled
     */
    void signalEvent(String type,
                     Object event,
                     long processInstanceId);

    /**
     * Returns a collection of currently active process instances.  Note that only process
     * instances that are currently loaded and active inside the engine will be returned.
     * When using persistence, it is likely not all running process instances will be loaded
     * as their state will be stored persistently.  It is recommended not to use this
     * method to collect information about the state of your process instances but to use
     * a history log for that purpose.
     *
     * @return a collection of process instances currently active in the session
     */
    Collection<ProcessInstance> getProcessInstances();

    /**
     * Returns the process instance with the given id.  Note that only active process instances
     * will be returned.  If a process instance has been completed already, this method will re
     * null.
     *
     * @param id the id of the process instance
     * @return the process instance with the given id or null if it cannot be found
```

```
     */
    ProcessInstance getProcessInstance(long processInstanceId);

    /**
     * Aborts the process instance with the given id.  If the process instance has been complet
     * (or aborted), or the process instance cannot be found, this method will throw an
     * IllegalArgumentException.
     *
     * @param id the id of the process instance
     */
    void abortProcessInstance(long processInstanceId);

    /**
     * Returns the WorkItemManager related to this session.  This can be used to
     * register new WorkItemHandlers or to complete (or abort) WorkItems.
     *
     * @return the WorkItemManager related to this session
     */
    WorkItemManager getWorkItemManager();
```

5.3.2. Event Listeners

The session provides methods for registering and removing listeners. A `ProcessEventListener` can be used to listen to process-related events, like starting or completing a process, entering and leaving a node, etc. Below, the different methods of the `ProcessEventListener` class are shown. An event object provides access to related information, like the process instance and node instance linked to the event. You can use this API to register your own event listeners.

```
public interface ProcessEventListener {

  void beforeProcessStarted( ProcessStartedEvent event );
  void afterProcessStarted( ProcessStartedEvent event );
  void beforeProcessCompleted( ProcessCompletedEvent event );
  void afterProcessCompleted( ProcessCompletedEvent event );
  void beforeNodeTriggered( ProcessNodeTriggeredEvent event );
  void afterNodeTriggered( ProcessNodeTriggeredEvent event );
  void beforeNodeLeft( ProcessNodeLeftEvent event );
  void afterNodeLeft( ProcessNodeLeftEvent event );
  void beforeVariableChanged(ProcessVariableChangedEvent event);
  void afterVariableChanged(ProcessVariableChangedEvent event);

}
```

A note about before and after events: these events typically act like a stack, which means that any events that occur as a direct result of the previous event, will occur between the before and the after of that event. For example, if a subsequent node is triggered as result of leaving a node, the

node triggered events will occur inbetween the beforeNodeLeftEvent and the afterNodeLeftEvent of the node that is left (as the triggering of the second node is a direct result of leaving the first node). Doing that allows us to derive cause relationships between events more easily. Similarly, all node triggered and node left events that are the direct result of starting a process will occur between the beforeProcessStarted and afterProcessStarted events. In general, if you just want to be notified when a particular event occurs, you should be looking at the before events only (as they occur immediately before the event actually occurs). When only looking at the after events, one might get the impression that the events are fired in the wrong order, but because the after events are triggered as a stack (after events will only fire when all events that were triggered as a result of this event have already fired). After events should only be used if you want to make sure that all processing related to this has ended (for example, when you want to be notified when starting of a particular process instance has ended.

Also note that not all nodes always generate node triggered and/or node left events. Depending on the type of node, some nodes might only generate node left events, others might only generate node triggered events. Catching intermediate events for example are not generating triggered events (they are only generating left events, as they are not really triggered by another node, rather activated from outside). Similarly, throwing intermediate events are not generating left events (they are only generating triggered events, as they are not really left, as they have no outgoing connection).

jBPM out-of-the-box provides a listener that can be used to create an audit log (either to the console or the a file on the file system). This audit log contains all the different events that occurred at runtime so it's easy to figure out what happened. Note that these loggers should only be used for debugging purposes. The following logger implementations are supported by default:

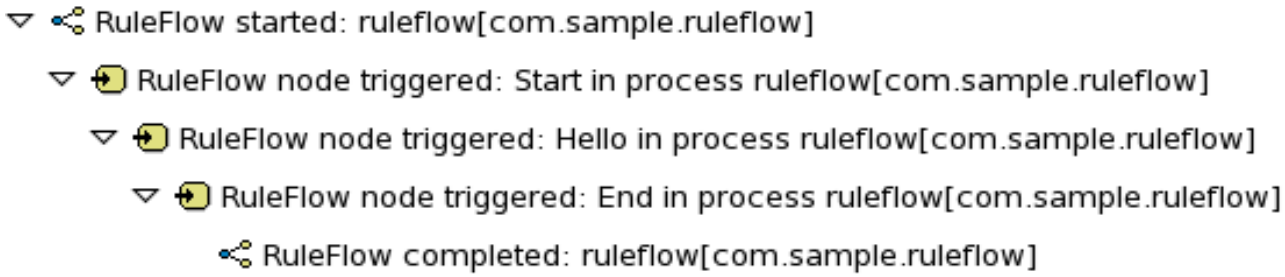
1. Console logger: This logger writes out all the events to the console.
2. File logger: This logger writes out all the events to a file using an XML representation. This log file might then be used in the IDE to generate a tree-based visualization of the events that occurred during execution.
3. Threaded file logger: Because a file logger writes the events to disk only when closing the logger or when the number of events in the logger reaches a predefined level, it cannot be used when debugging processes at runtime. A threaded file logger writes the events to a file after a specified time interval, making it possible to use the logger to visualize the progress in realtime, while debugging processes.

The `KnowledgeRuntimeLoggerFactory` lets you add a logger to your session, as shown below. When creating a console logger, the knowledge session for which the logger needs to be created must be passed as an argument. The file logger also requires the name of the log file to be created, and the threaded file logger requires the interval (in milliseconds) after which the events should be saved. You should always close the logger at the end of your application.

```
KnowledgeRuntimeLogger logger = KnowledgeRuntimeLoggerFactory.newFileLogger( ksession, "test"
```

```
// add invocations to the process engine here,
// e.g. ksession.startProcess(processId);
...
logger.close();
```

The log file that is created by the file-based loggers contains an XML-based overview of all the events that occurred at runtime. It can be opened in Eclipse, using the Audit View in the Drools Eclipse plugin, where the events are visualized as a tree. Events that occur between the before and after event are shown as children of that event. The following screenshot shows a simple example, where a process is started, resulting in the activation of the Start node, an Action node and an End node, after which the process was completed.



5.3.3. Correlation Keys

A common requirement when working with processes is ability to assign a given process instance some sort of business identifier that can be later on referenced without knowing the actual (generated) id of the process instance. To provide such capabilities, jBPM allows to use CorrelationKey that is composed of CorrelationProperties. CorrelationKey can have either single property describing it (which is in most cases) but it can be represented as multi valued properties set.

Correlation capabilities are provided as part of interface

```
CorrelationAwareProcessRuntime
```

that exposes following methods:

```
/**
 * Start a new process instance.  The process (definition) that should
* be used is referenced by the given process id.  Parameters can be passed
 * to the process instance (as name-value pairs), and these will be set
 * as variables of the process instance.
 *
 * @param processId  the id of the process that should be started
 * @param correlationKey custom correlation key that can be used to identify process insta
```

```
     * @param parameters  the process variables that should be set when starting the process i
     * @return the ProcessInstance that represents the instance of the process that was starte
     */
    ProcessInstance startProcess(String processId, CorrelationKey correlationKey, Map<String,

    /**
     * Creates a new process instance (but does not yet start it).  The process
     * (definition) that should be used is referenced by the given process id.
     * Parameters can be passed to the process instance (as name-value pairs),
     * and these will be set as variables of the process instance.  You should only
     * use this method if you need a reference to the process instance before actually
     * starting it.  Otherwise, use startProcess.
     *
     * @param processId  the id of the process that should be started
     * @param correlationKey custom correlation key that can be used to identify process insta
     * @param parameters  the process variables that should be set when creating the process i
     * @return the ProcessInstance that represents the instance of the process that was create
     */
    ProcessInstance createProcessInstance(String processId, CorrelationKey correlationKey, Ma

    /**
     * Returns the process instance with the given correlationKey.  Note that only active proc
     * will be returned.  If a process instance has been completed already, this method will r
     * null.
     *
     * @param correlationKey the custom correlation key assigned when process instance was cre
     * @return the process instance with the given id or null if it cannot be found
     */
    ProcessInstance getProcessInstance(CorrelationKey correlationKey);
```

Correlation is usually used with long running processes and thus require persistence to be enabled to be able to permanently store correlation information.

5.3.4. Threads

In the following text, we will refer to two types of "multi-threading": *logical* and *technical*. *Technical multi-threading* is what happens when multiple threads or processes are started on a computer, for example by a Java or C program. *Logical multi-threading* is what we see in a BPM process after the process reaches a parallel gateway, for example. From a functional standpoint, the original process will then split into two processes that are executed in a parallel fashion.

Of course, the jBPM engine supports logical multi-threading: for example, processes that include a parallel gateway. We've chosen to implement logical multi-threading using one thread: a jBPM process that includes logical multi-threading will only be executed in one technical thread. The main reason for doing this is that multiple (technical) threads need to be be able to communicate state information with each other if they are working on the same process. This requirement brings with it a number of complications. While it might seem that multi-threading would bring

performance benefits with it, the extra logic needed to make sure the different threads work together well means that this is not guaranteed. There is also the extra overhead incurred because we need to avoid race conditions and deadlocks.

In general, the jBPM engine executes actions in serial. For example, when the engine encounters a script task in a process, it will synchronously execute that script and wait for it to complete before continuing execution. Similarly, if a process encounters a parallel gateway, it will sequentially trigger each of the outgoing branches, one after the other. This is possible since execution is almost always instantaneous, meaning that it is extremely fast and produces almost no overhead. As a result, the user will usually not even notice this. Similarly, action scripts in a process are also synchronously executed, and the engine will wait for them to finish before continuing the process. For example, doing a `Thread.sleep(...)` as part of a script will not make the engine continue execution elsewhere but will block the engine thread during that period.

The same principle applies to service tasks. When a service task is reached in a process, the engine will also invoke the handler of this service synchronously. The engine will wait for the `completeWorkItem(...)` method to return before continuing execution. It is important that your service handler executes your service asynchronously if its execution is not instantaneous.

An example of this would be a service task that invokes an external service. Since the delay in invoking this service remotely and waiting for the results might be too long, it might be a good idea to invoke this service asynchronously. This means that the handler will only invoke the service and will notify the engine later when the results are available. In the mean time, the process engine then continues execution of the process.

Human tasks are a typical example of a service that needs to be invoked asynchronously, as we don't want the engine to wait until a human actor has responded to the request. The human task handler will only create a new task (on the task list of the assigned actor) when the human task node is triggered. The engine will then be able to continue execution on the rest of the process (if necessary) and the handler will notify the engine asynchronously when the user has completed the task.

5.4. RuntimeManager

5.4.1. Overview

RuntimeManager has been introduced to simplify and empower usage of knowledge API especially in context of processes. It provides configurable strategies that control actual runtime execution (how KieSessions are provided) and by default provides following:

- Singleton - runtime manager maintains single KieSession regardless of number of processes available
- Per Request - runtime manager delivers new KieSession for every request
- Per Process Instance - runtime manager maintains mapping between process instance and KieSession and always provides same KieSession whenever working with given process instance

Runtime Manager is primary responsible for managing and delivering instances of RuntimeEngine to the caller. In turn, RuntimeEngine encapsulates two the most important elements of jBPM engine:

- KieSession
- TaskService

Both of these components are already configured to work with each other smoothly without additional configuration from end user. No more need to register human task handler or keeping track if it's connected to the service or not.

```
public interface RuntimeManager {

    /**
     * Returns <code>RuntimeEngine</code> instance that is fully initialized:
     * <ul>
     *  <li>KiseSession is created or loaded depending on the strategy</li>
    * <li>TaskService is initialized and attached to ksession (via listener)</li>
     *  <li>WorkItemHandlers are initialized and registered on ksession</li>
*<li>EventListeners(process,agenda,workingmemory)areinitializedandaddedtoksession</
 li>
     * </ul>
@paramcontextheconcreteimplementationofthecontextthatissupportedbygiven<code>RuntimeManager</
 code>
     * @return instance of the <code>RuntimeEngine</code>
     */
    RuntimeEngine getRuntimeEngine(Context<?> context);

    /**
     * Unique identifier of the <code>RuntimeManager</code>
     * @return
     */
    String getIdentifier();

    /**
                                    *      Disposes      <code>RuntimeEngine</
 code> and notifies all listeners about that fact.
       * This method should always be used to dispose <code>RuntimeEngine</
 code> that is not needed
     * anymore. <br/>
     * ksession.dispose() shall never be used with RuntimeManager as it will break the internal
    * mechanisms of the manager responsible for clear and efficient disposal.<br/>
                 *   Dispose   is   not   needed   if   <code>RuntimeEngine</
 code> was obtained within active JTA transaction,
     * this means that when getRuntimeEngine method was invoked during active JTA transaction t
     * the runtime engine will happen automatically on transaction completion.
     * @param runtime
     */
```

```
    void disposeRuntimeEngine(RuntimeEngine runtime);

    /**
                                     *       Closes       <code>RuntimeManager</
code> and releases its resources. Shall always be called when
     * runtime manager is not needed any more. Otherwise it will still be active and operationa
     */
    void close();

}
```

RuntimeEngine interface provides the most important methods to get access to engine components:

```
public interface RuntimeEngine {

    /**
     * Returns <code>KieSession</code> configured for this <code>RuntimeEngine</
code>
     * @return
     */
    KieSession getKieSession();

    /**
    * Returns <code>TaskService</code> configured for this <code>RuntimeEngine</
code>
     * @return
     */
    TaskService getTaskService();
}
```

RuntimeManager will ensure that regardless of the strategy it will provide same capabilities when it comes to initialization and configuration of the RuntimeEngine. That means

- KieSession will be loaded with same factories (either in memory or JPA based)
- WorkItemHandlers will be registered on every KieSession (either loaded from db or newly created)
- Event listeners (Process, Agenda, WorkingMemory) will be registered on every KieSession (either loaded from db or newly created)
- TaskService will be configured with:
  - JTA transaction manager
  - same entity manager factory as for the KieSession

- UserGroupCallback from environment

On the other hand, RuntimeManager maintains the engine disposal as well by providing dedicated methods to dispose RuntimeEngine when it's no more needed to release any resources it might have acquired.

5.4.2. Strategies

Singleton strategy - instructs RuntimeManager to maintain single instance of RuntimeEngine (and in turn single instance of KieSession and TaskService). Access to the RuntimeEngine is synchronized and by that thread safe although it comes with a performance penalty due to synchronization. This strategy is similar to what was available by default in jBPM version 5.x and it's considered easiest strategy and recommended to start with.

It has following characteristics that are important to evaluate while considering it for given scenario:

- small memory footprint - single instance of runtime engine and task service
- simple and compact in design and usage
- good fit for low to medium load on process engine due to synchronized access
- due to single KieSession instance all state objects (such as facts) are directly visible to all process instances and vice versa
- not contextual - meaning when retrieving instances of RuntimeEngine from singleton RuntimeManager Context instance is not important and usually EmptyContext.get() is used although null argument is acceptable as well
- keeps track of id of KieSession used between RuntimeManager restarts to ensure it will use same session - this id is stored as serialized file on disc in temp location that depends on the environment can be one of following:
  - value given by jbpm.data.dir system property
  - value given by jboss.server.data.dir system property
  - value given by java.io.tmpdir system property

Per request strategy - instructs RuntimeManager to provide new instance of RuntimeEngine for every request. As request RuntimeManager will consider one or more invocations within single transaction. It must return same instance of RuntimeEngine within single transaction to ensure correctness of state as otherwise operation done in one call would not be visible in the other. This is sort of "stateless" strategy that provides only request scope state and once request is completed RuntimeEngine will be permanently destroyed - KieSession information will be removed from the database in case persistence was used.

It has following characteristics:

- completely isolated process engine and task service operations for every request

- completely stateless, storing facts makes sense only for the duration of the request
- good fit for high load, stateless processes (no facts or timers involved that shall be preserved between requests)
- KieSession is only available during life time of request and at the end is destroyed
- not contextual - meaning when retrieving instances of RuntimeEngine from per request RuntimeManager Context instance is not important and usually EmptyContext.get() is used although null argument is acceptable as well

Per process instance strategy - instructs RuntimeManager to maintain a strict relationship between KieSession and ProcessInstance. That means that KieSession will be available as long as the ProcessInstance that it belongs to is active. This strategy provides the most flexible approach to use advanced capabilities of the engine like rule evaluation in isolation (for given process instance only), maximum performance and reduction of potential bottlenecks intriduced by synchronization; and at the same time reduces number of KieSessions to the actual number of process instances rather than number of requests (in contrast to per request strategy).

It has following characteristics:

- most advanced strategy to provide isolation to given process instance only
- maintains strict relationship between KieSession and ProcessInstance to ensure it will always deliver same KieSession for given ProcessInstance
- merges life cycle of KieSession with ProcessInstance making both to be disposed on process instance completion (complete or abort)
- allows to maintain data (such as facts, timers) in scope of process instance - only process instance will have access to that data
- introduces bit of overhead due to need to look up and load KieSession for process instance
- validates usage of KieSession so it cannot be (ab)used for other process instances, in such a case exception is thrown
- is contextual - accepts following context instances:
  - EmptyContext or null - when starting process instance as there is no process instance id available yet
  - ProcessInstanceIdContext - used after process instance was created
  - CorrelationKeyContext - used as an alternative to ProcessInstanceIdContext to use custom (business) key instead of process instance id

5.4.3. Usage

Regular usage scenario for RuntimeManager is:

- At application startup
  - build RuntimeManager and keep it for entire life time of the application, it's thread safe and can be (or even should be) accessed concurrently
- At request
  - get RuntimeEngine from RuntimeManager using proper context instance dedicated to strategy of RuntimeManager
  - get KieSession and/or TaskService from RuntimeEngine
  - perform operations on KieSession and/or TaskService such as startProcess, completeTask, etc
  - once done with processing dispose RuntimeEngine using RuntimeManager.disposeRuntimeEngine method
- At application shutdown
  - close RuntimeManager

Note

When RuntimeEngine is obtained from RuntimeManager within an active JTA transaction, then there is no need to dispose RuntimeEngine at the end, as RuntimeManager will automatically dispose the RuntimeEngine on transaction completion (regardless of the completion status commit or rollback).

5.4.3.1. Example

Here is how you can build RuntimeManager and get RuntimeEngine (that encapsulates KieSession and TaskService) from it:

```
    // first configure environment that will be used by RuntimeManager
    RuntimeEnvironment environment = RuntimeEnvironmentBuilder.Factory.get()
    .newDefaultInMemoryBuilder()
                        .addAsset(ResourceFactory.newClassPathResource("BPMN2-
ScriptTask.bpmn2"), ResourceType.BPMN2)
    .get();

    // next create RuntimeManager - in this case singleton strategy is chosen
    RuntimeManager manager = RuntimeManagerFactory.Factory.get().newSingletonRuntimeManager(env

    // then get RuntimeEngine out of manager - using empty context as singleton
 does not keep track
```

```
    // of runtime engine as there is only one
    RuntimeEngine runtime = manager.getRuntimeEngine(EmptyContext.get());

    // get KieSession from runtime runtimeEngine - already initialized with all
 handlers, listeners, etc that were configured
    // on the environment
    KieSession ksession = runtimeEngine.getKieSession();

    // add invocations to the process engine here,
    // e.g. ksession.startProcess(processId);

    // and last dispose the runtime engine
    manager.disposeRuntimeEngine(runtimeEngine);
```

This example provides simplest (minimal) way of using *RuntimeManager* and *RuntimeEngine* although it provides few quite valuable information:

- KieSession will be in memory only - by using newDefaultInMemoryBuilder
- there will be single process available for execution - by adding it as an asset
- TaskService will be configured and attached to KieSession via LocalHTWorkItemHandler to support user task capabilities within processes

5.4.4. Configuration

The complexity of knowing when to create, dispose, register handlers, etc is taken away from the end user and moved to the runtime manager that knows when/how to perform such operations but still allows to have a fine grained control over this process by providing comprehensive configuration of the RuntimeEnvironment.

```
  public interface RuntimeEnvironment {

    /**
     * Returns <code>KieBase</code> that shall be used by the manager
     * @return
     */
    KieBase getKieBase();

    /**
  * KieSession environment that shall be used to create instances of <code>KieSession</
code>
     * @return
     */
```

```
    Environment getEnvironment();

    /**
 *KieSessionconfigurationthatshallbeusedtocreateinstancesof<code>KieSession</
code>
     * @return
     */
    KieSessionConfiguration getConfiguration();

    /**
     * Indicates if persistence shall be used for the KieSession instances
     * @return
     */
    boolean usePersistence();

    /**
       * Delivers concrete implementation of <code>RegisterableItemsFactory</
code> to obtain handlers and listeners
     * that shall be registered on instances of <code>KieSession</code>
     * @return
     */
    RegisterableItemsFactory getRegisterableItemsFactory();

    /**
          *  Delivers  concrete  implementation  of  <code>UserGroupCallback</
code> that shall be registered on instances
     * of <code>TaskService</code> for managing users and groups.
     * @return
     */
    UserGroupCallback getUserGroupCallback();

    /**
     * Delivers custom class loader that shall be used by the process engine and task service i
     * @return
     */
    ClassLoader getClassLoader();

    /**
     * Closes the environment allowing to close all depending components such as ksession facto
     */
    void close();
```

5.4.4.1. Building RuntimeEnvironment

While RuntimeEnvironment interface provides mostly access to data kept as part of the environment and will be used by the RuntimeManager, users should take advantage of builder style class that provides fluent API to configure RuntimeEnvironment with predefined settings.

```
public interface RuntimeEnvironmentBuilder {

    public RuntimeEnvironmentBuilder persistence(boolean persistenceEnabled);

    public RuntimeEnvironmentBuilder entityManagerFactory(Object emf);

   public RuntimeEnvironmentBuilder addAsset(Resource asset, ResourceType type);

    public RuntimeEnvironmentBuilder addEnvironmentEntry(String name, Object value);

   public RuntimeEnvironmentBuilder addConfiguration(String name, String value);

    public RuntimeEnvironmentBuilder knowledgeBase(KieBase kbase);

    public RuntimeEnvironmentBuilder userGroupCallback(UserGroupCallback callback);

    public RuntimeEnvironmentBuilder registerableItemsFactory(RegisterableItemsFactory factory)

    public RuntimeEnvironment get();

    public RuntimeEnvironmentBuilder classLoader(ClassLoader cl);

    public RuntimeEnvironmentBuilder schedulerService(Object globalScheduler);
```

Instances of the RuntimeEnvironmentBuilder can be obtained via RuntimeEnvironmentBuilderFactory that provides preconfigured sets of builder to simplify and help users to build the environment for the RuntimeManager.

```
public interface RuntimeEnvironmentBuilderFactory {

    /**
            *  Provides  completely  empty  <code>RuntimeEnvironmentBuilder</
code> instance that allows to manually
     * set all required components instead of relying on any defaults.
     * @return new instance of <code>RuntimeEnvironmentBuilder</code>
     */
    public RuntimeEnvironmentBuilder newEmptyBuilder();

    /**
       * Provides default configuration of <code>RuntimeEnvironmentBuilder</
code> that is based on:
     * <ul>
     *  <li>DefaultRuntimeEnvironment</li>
     * </ul>
```

```
             *  @return  new  instance  of  <code>RuntimeEnvironmentBuilder</
code> that is already preconfigured with defaults
     *
     * @see DefaultRuntimeEnvironment
     */
    public RuntimeEnvironmentBuilder newDefaultBuilder();

    /**
        * Provides default configuration of <code>RuntimeEnvironmentBuilder</
code> that is based on:
     * <ul>
     *  <li>DefaultRuntimeEnvironment</li>
     * </ul>
     * but it does not have persistence for process engine configured so it will only store pro
             *  @return  new  instance  of  <code>RuntimeEnvironmentBuilder</
code> that is already preconfigured with defaults
     *
     * @see DefaultRuntimeEnvironment
     */
    public RuntimeEnvironmentBuilder newDefaultInMemoryBuilder();

    /**
        * Provides default configuration of <code>RuntimeEnvironmentBuilder</
code> that is based on:
     * <ul>
     *  <li>DefaultRuntimeEnvironment</li>
     * </ul>
     * This one is tailored to works smoothly with kjars as the notion of kbase and ksessions
     * @param groupId group id of kjar
     * @param artifactId artifact id of kjar
     * @param version version number of kjar
             *  @return  new  instance  of  <code>RuntimeEnvironmentBuilder</
code> that is already preconfigured with defaults
     *
     * @see DefaultRuntimeEnvironment
     */
    public RuntimeEnvironmentBuilder newDefaultBuilder(String groupId, String artifactId, Strin

    /**
        * Provides default configuration of <code>RuntimeEnvironmentBuilder</
code> that is based on:
     * <ul>
     *  <li>DefaultRuntimeEnvironment</li>
     * </ul>
     * This one is tailored to works smoothly with kjars as the notion of kbase and ksessions
     * @param groupId group id of kjar
     * @param artifactId artifact id of kjar
     * @param version version number of kjar
     * @param kbaseName name of the kbase defined in kmodule.xml stored in kjar
```

```
    * @param ksessionName name of the ksession define in kmodule.xml stored in kjar
             *  @return  new  instance  of  <code>RuntimeEnvironmentBuilder</
code> that is already preconfigured with defaults
     *
     * @see DefaultRuntimeEnvironment
     */
    public RuntimeEnvironmentBuilder newDefaultBuilder(String groupId, String artifactId, Strin

    /**
        * Provides default configuration of <code>RuntimeEnvironmentBuilder</
code> that is based on:
     * <ul>
     *  <li>DefaultRuntimeEnvironment</li>
     * </ul>
     * This one is tailored to works smoothly with kjars as the notion of kbase and ksessions
     * @param releaseId <code>ReleaseId</code> that described the kjar
             *  @return  new  instance  of  <code>RuntimeEnvironmentBuilder</
code> that is already preconfigured with defaults
     *
     * @see DefaultRuntimeEnvironment
     */
    public RuntimeEnvironmentBuilder newDefaultBuilder(ReleaseId releaseId);

    /**
        * Provides default configuration of <code>RuntimeEnvironmentBuilder</
code> that is based on:
     * <ul>
     *  <li>DefaultRuntimeEnvironment</li>
     * </ul>
     * This one is tailored to works smoothly with kjars as the notion of kbase and ksessions
     * @param releaseId <code>ReleaseId</code> that described the kjar
     * @param kbaseName name of the kbase defined in kmodule.xml stored in kjar
    * @param ksessionName name of the ksession define in kmodule.xml stored in kjar
             *  @return  new  instance  of  <code>RuntimeEnvironmentBuilder</
code> that is already preconfigured with defaults
     *
     * @see DefaultRuntimeEnvironment
     */
    public RuntimeEnvironmentBuilder newDefaultBuilder(ReleaseId releaseId, String kbaseName, S

    /**
        * Provides default configuration of <code>RuntimeEnvironmentBuilder</
code> that is based on:
     * <ul>
     *  <li>DefaultRuntimeEnvironment</li>
     * </ul>
 *It relies on KieClasspathContainer that requires to have kmodule.xml present in META-
INF folder which
     * defines the kjar itself.
```

```
     * Expects to use default kbase and ksession from kmodule.
           *  @return  new  instance  of  <code>RuntimeEnvironmentBuilder</
code> that is already preconfigured with defaults
     *
     * @see DefaultRuntimeEnvironment
     */
    public RuntimeEnvironmentBuilder newClasspathKmoduleDefaultBuilder();

    /**
       * Provides default configuration of <code>RuntimeEnvironmentBuilder</
code> that is based on:
     * <ul>
     *  <li>DefaultRuntimeEnvironment</li>
     * </ul>
 *It relies on KieClasspathContainer that requires to have kmodule.xml present in META-
INF folder which
     * defines the kjar itself.
     * @param kbaseName name of the kbase defined in kmodule.xml
     * @param ksessionName name of the ksession define in kmodule.xml
           *  @return  new  instance  of  <code>RuntimeEnvironmentBuilder</
code> that is already preconfigured with defaults
     *
     * @see DefaultRuntimeEnvironment
     */
    public RuntimeEnvironmentBuilder newClasspathKmoduleDefaultBuilder(String kbaseName, String
```

Besides KieSession Runtime Manager provides access to TaskService too as integrated component of a RuntimeEngine that will always be configured and ready for communication between process engine and task service.

Since the default builder was used, it will already come with predefined set of elements that consists of:

- Persistence unit name will be set to org.jbpm.persistence.jpa (for both process engine and task service)
- Human Task handler will be automatically registered on KieSession
- JPA based history log event listener will be automatically registered on KieSession
- Event listener to trigger rule task evaluation (fireAllRules) will be automatically registered on KieSession

5.4.4.2. Registering handlers and listeners

To extend it with your own handlers or listeners a dedicated mechanism is provided that comes as implementation of RegisterableItemsFactory

```
    /**
     * Returns new instances of <code>WorkItemHandler</code> that will be registered on <code>RuntimeEngine</code>
     * @param runtime provides <code>RuntimeEngine</code> in case handler need to make use of it internally
     * @return map of handlers to be registered - in case of no handlers empty map shall be ret
     */
    Map<String, WorkItemHandler> getWorkItemHandlers(RuntimeEngine runtime);

    /**
     * Returns new instances of <code>ProcessEventListener</code> that will be registered on <code>RuntimeEngine</code>
     * @param runtime provides <code>RuntimeEngine</code> in case listeners need to make use of it internally
     * @return list of listeners to be registered - in case of no listeners empty list shall be
     */
    List<ProcessEventListener> getProcessEventListeners(RuntimeEngine runtime);

    /**
     * Returns new instances of <code>AgendaEventListener</code> that will be registered on <code>RuntimeEngine</code>
     * @param runtime provides <code>RuntimeEngine</code> in case listeners need to make use of it internally
     * @return list of listeners to be registered - in case of no listeners empty list shall be
     */
    List<AgendaEventListener> getAgendaEventListeners(RuntimeEngine runtime);

    /**
     * Returns new instances of <code>WorkingMemoryEventListener</code> that will be registered on <code>RuntimeEngine</code>
     * @param runtime provides <code>RuntimeEngine</code> in case listeners need to make use of it internally
     * @return list of listeners to be registered - in case of no listeners empty list shall be
     */
    List<WorkingMemoryEventListener> getWorkingMemoryEventListeners(RuntimeEngine runtime);
```

A best practice is to just extend those that come out of the box and just add your own. Extensions are not always needed as the default implementations of RegisterableItemsFactory provides possibility to define custom handlers and listeners. Following is a list of available implementations that might be useful (they are ordered in the hierarchy of inheritance):

- org.jbpm.runtime.manager.impl.SimpleRegisterableItemsFactory - simplest possible implementations that comes empty and is based on reflection to produce instances of handlers and listeners based on given class names

- org.jbpm.runtime.manager.impl.DefaultRegisterableItemsFactory - extension of the Simple implementation that introduces defaults described above and still provides same capabilities as Simple implementation
- org.jbpm.runtime.manager.impl.KModuleRegisterableItemsFactory - extension of default implementation that provides specific capabilities for kmodule and still provides same capabilities as Simple implementation
- org.jbpm.runtime.manager.impl.cdi.InjectableRegisterableItemsFactory - extension of default implementation that is tailored for CDI environments and provides CDI style approach to finding handlers and listeners via producers

Alternatively, simple (stateless or requiring only KieSession) work item handlers might be registered in the well known way - defined as part of CustomWorkItem.conf file that shall be placed on class path. To use this approach do following:

- create file "drools.session.conf" inside META-INF of the root of the class path, for web applications it will be WEB-INF/classes/META-INF
- add following line to drools.session.conf file "drools.workItemHandlers = CustomWorkItemHandlers.conf"
- create file "CustomWorkItemHandlers.conf" inside META-INF of the root of the class path, for web applications it will be WEB-INF/classes/META-INF
- define custom work item handlers in MVEL style inside CustomWorkItemHandlers.conf

```
[
  "Log": new org.jbpm.process.instance.impl.demo.SystemOutWorkItemHandler(),
                                          "WebService":                    new
 org.jbpm.process.workitem.webservice.WebServiceWorkItemHandler(ksession),
  "Rest": new org.jbpm.process.workitem.rest.RESTWorkItemHandler(),
                         "Service            Task"            :            new
 org.jbpm.process.workitem.bpmn2.ServiceTaskHandler(ksession)
]
```

And that's it, now all these work item handlers will be registered for any KieSession created by that application, regardless if it uses RuntimeManager or not.

5.4.4.2.1. Registering handlers and listeners in CDI environment

When using RuntimeManager in CDI environment there are dedicated interfaces that can be used to provide custom WorkItemHandlers and EventListeners to the RuntimeEngine.

```
public interface WorkItemHandlerProducer {

    /**
     * Returns map of (key = work item name, value work item handler instance) of work items
```

```
     * to be registered on KieSession
     * <br/>
     * Parameters that might be given are as follows:
     * <ul>
     *  <li>ksession</li>
     *  <li>taskService</li>
     *  <li>runtimeManager</li>
     * </ul>
     *
     * @param identifier - identifier of the owner - usually RuntimeManager that allows the pro
     * and provide valid instances for given owner
     * @param params - owner might provide some parameters, usually KieSession, TaskService, Ru
     * @return map of work item handler instances (recommendation is to always return new insta
     */
    Map<String, WorkItemHandler> getWorkItemHandlers(String identifier, Map<String, Object> par
}
```

Event listener producer shall be annotated with proper qualifier to indicate what type of listeners they provide, so pick one of following to indicate they type:

- @Process - for ProcessEventListener
- @Agenda - for AgendaEventListener
- @WorkingMemory - for WorkingMemoryEventListener

```
public interface EventListenerProducer<T> {

    /**
     * Returns list of instances for given (T) type of listeners
     * <br/>
     * Parameters that might be given are as follows:
     * <ul>
     *  <li>ksession</li>
     *  <li>taskService</li>
     *  <li>runtimeManager</li>
     * </ul>
     * @param identifier - identifier of the owner - usually RuntimeManager that allows the pro
     * and provide valid instances for given owner
     * @param params - owner might provide some parameters, usually KieSession, TaskService, Ru
     * @return list of listener instances (recommendation is to always return new instances whe
     */
    List<T> getEventListeners(String identifier, Map<String, Object>  params);
}
```

Implementations of these interfaces shall be packaged as bean archive (includes beans.xml inside META-INF) and placed on application classpath (e.g. WEB-INF/lib for web application). THat is

enough for CDI based RuntimeManager to discover them and register on every KieSession that is created or loaded from data store.

Some parameters are provided to the producers to allow handlers/listeners to be more stateful and be able to do more advanced things with the engine - like signal of the engine or process instance in case of an error. Thus all components are provided:

- KieSession
- TaskService
- RuntimeManager

Note

Whenever there is a need to interact with the process engine/task service from within handler or listener, recommended approach is to use RuntimeManager and retrieve RuntimeEngine (and then KieSession and/or TaskService) from it as that will ensure proper state managed according to strategy

In addition, some filtering can be applied based on identifier (that is given as argument to the methods) to decide if given RuntimeManager shall receive handlers/listeners or not.

5.5. Configuration

There are several control parameters available to alter engine default behavior. This allows to fine tune the execution for the environment needs and actual requirements. All of these parameters are set as JVM system properties, usually with -D when starting program e.g. application server.

Table 5.1. Control parameters

Name	Possible values	Default value	Description
jbpm.ut.jndi.lookup	String		Alternative JNDI name to be used when there is no access to the default one (java:comp/UserTransaction)
jbpm.enable.multi.con	true|false	false	Enables multiple incoming/outgoing sequence flows support for activities

Name	Possible values	Default value	Description
jbpm.business.calendar.properties	String	/ jbpm.business.calendar.properties	Allows to provide alternative classpath location of business calendar configuration file
jbpm.overdue.timer.delay	Long	2000	Specifies delay for overdue timers to allow proper initialization, in milliseconds
jbpm.process.name.comparator	String		Allows to provide alternative comparator class to empower start process by name feature, if not set NumberVersionComparator is used
jbpm.loop.level.disabled	true/false	true	Allows to enable or disable loop iteration tracking, to allow advanced loop support when using XOR gateways
org.kie.mail.session	String	mail/ jbpmMailSession	Allows to provide alternative JNDI name for mail session used by Task Deadlines
jbpm.usergroup.callback.properties	String	/ jbpm.usergroup.callback.properties	Allows to provide alternative classpath location for user group callback implementation (LDAP, DB)

Name	Possible values	Default value	Description
jbpm.user.group.mapping	String	${jboss.server.config.dir}/roles.properties	Allows to provide alternative location of roles.properties for JBossUserGroupCallbackImpl
jbpm.user.info.properties	String	/jbpm.user.info.properties	Allows to provide alternative classpath location of user info configuration (used by LDAPUserInfoImpl)
org.jbpm.ht.user.separator	String	,	Allows to provide alternative separator of actors and groups for user tasks, default is comma (,)
org.quartz.properties	String		Allows to provide location of the quartz config file to activate quartz based timer service
jbpm.data.dir	String	${jboss.server.data.dir} is available otherwise ${java.io.tmpdir}	Allows to provide location where data files produced by jbpm should be stored
org.kie.executor.pool.size	Integer	1	Allows to provide thread pool size for jbpm executor
org.kie.executor.retry.count	Integer	3	Allows to provide number of retries attempted in case of error by jbpm executor

Name	Possible values	Default value	Description	
org.kie.executor.interval	integer	3	Allows to provide frequency used to check for pending jobs by jbpm executor, in seconds	
org.kie.executor.disabled	boolean	true	Enables or disable jbpm executor	

Chapter 6. Processes

6.1. What is BPMN 2.0

Note

"The primary goal of BPMN is to provide a notation that is readily understandable by all business users, from the business analysts that create the initial drafts of the processes, to the technical developers responsible for implementing the technology that will perform those processes, and finally, to the business people who will manage and monitor those processes."

The Business Process Model and Notation (BPMN) 2.0 specification is an OMG specification that not only defines a standard on how to graphically represent a business process (like BPMN 1.x), but now also includes execution semantics for the elements defined, and an XML format on how to store (and share) process definitions.

jBPM6 allows you to execute processes defined using the BPMN 2.0 XML format. That means that you can use all the different jBPM6 tooling to model, execute, manage and monitor your business processes using the BPMN 2.0 format for specifying your executable business processes. Actually, the full BPMN 2.0 specification also includes details on how to represent things like choreographies and collaboration. The jBPM project however focuses on that part of the specification that can be used to specify executable processes.

Executable processes in BPMN consist of a different types of nodes being connected to each other using sequence flows. The BPMN 2.0 specification defines three main types of nodes:

- *Events*: They are used to model the occurrence of a particular event. This could be a start event (that is used to indicate the start of the process), end events (that define the end of the process, or of that subflow) and intermediate events (that indicate events that might occur during the execution of the process).
- *Activities*: These define the different actions that need to be performed during the execution of the process. Different types of tasks exist, depending on the type of activity you are trying to model (e.g. human task, service task, etc.) and activities could also be nested (using different types of sub-processes).
- *Gateways*: Can be used to define multiple paths in the process. Depending on the type of gateway, these might indicate parallel execution, choice, etc.

jBPM6 does not implement all elements and attributes as defined in the BPMN 2.0 specification. We do however support a significant subset, including the most common node types that can be used inside executable processes. This includes (almost) all elements and attributes as defined in the "Common Executable" subclass of the BPMN 2.0 specification, extended with some additional

elements and attributes we believe are valuable in that context as well. The full set of elements and attributes that are supported can be found below, but it includes elements like:

- *Flow objects*
  - Events
    - Start Event (None, Conditional, Signal, Message, Timer)
    - End Event (None, Terminate, Error, Escalation, Signal, Message, Compensation)
    - Intermediate Catch Event (Signal, Timer, Conditional, Message)
    - Intermediate Throw Event (None, Signal, Escalation, Message, Compensation)
    - Non-interrupting Boundary Event (Escalation, Signal, Timer, Conditional, Message)
    - Interrupting Boundary Event (Escalation, Error, Signal, Timer, Conditional, Message, Compensation)
  - Activities
    - Script Task
    - Task
    - Service Task
    - User Task
    - Business Rule Task
    - Manual Task
    - Send Task
    - Receive Task
    - Reusable Sub-Process (Call Activity)
    - Embedded Sub-Process
    - Event Sub-Process
    - Ad-Hoc Sub-Process
    - Data-Object
  - Gateways
    - Diverging
      - Exclusive

        - Inclusive
        - Parallel
        - Event-Based
      - Converging
        - Exclusive
        - Inclusive
        - Parallel
    - Lanes
- *Data*
    - Java type language
    - Process properties
    - Embedded Sub-Process properties
    - Activity properties
- *Connecting objects*
    - Sequence flow

For example, consider the following "Hello World" BPMN 2.0 process, which does nothing more that writing out a "Hello World" statement when the process is started.

An executable version of this process expressed using BPMN 2.0 XML would look something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions id="Definition"
             targetNamespace="http://www.example.org/MinimalExample"
             typeLanguage="http://www.java.com/javaTypes"
             expressionLanguage="http://www.mvel.org/2.0"
             xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
             xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
                 xs:schemaLocation="http://www.omg.org/spec/BPMN/20100524/MODEL
 BPMN20.xsd"
             xmlns:bpmndi="http://www.omg.org/spec/BPMN/20100524/DI"
             xmlns:dc="http://www.omg.org/spec/DD/20100524/DC"
             xmlns:di="http://www.omg.org/spec/DD/20100524/DI"
             xmlns:tns="http://www.jboss.org/drools">
```

```
<process processType="Private" isExecutable="true" id="com.sample.HelloWorld" name="Hello
   World" >

    <!-- nodes -->
    <startEvent id="_1" name="StartProcess" />
    <scriptTask id="_2" name="Hello" >
      <script>System.out.println("Hello World");</script>
    </scriptTask>
    <endEvent id="_3" name="EndProcess" >
        <terminateEventDefinition/>
    </endEvent>

    <!-- connections -->
    <sequenceFlow id="_1-_2" sourceRef="_1" targetRef="_2" />
    <sequenceFlow id="_2-_3" sourceRef="_2" targetRef="_3" />

  </process>

  <bpmndi:BPMNDiagram>
    <bpmndi:BPMNPlane bpmnElement="Minimal" >
      <bpmndi:BPMNShape bpmnElement="_1" >
        <dc:Bounds x="15" y="91" width="48" height="48" />
      </bpmndi:BPMNShape>
      <bpmndi:BPMNShape bpmnElement="_2" >
        <dc:Bounds x="95" y="88" width="83" height="48" />
      </bpmndi:BPMNShape>
      <bpmndi:BPMNShape bpmnElement="_3" >
        <dc:Bounds x="258" y="86" width="48" height="48" />
      </bpmndi:BPMNShape>
      <bpmndi:BPMNEdge bpmnElement="_1-_2" >
        <di:waypoint x="39" y="115" />
        <di:waypoint x="75" y="46" />
        <di:waypoint x="136" y="112" />
      </bpmndi:BPMNEdge>
      <bpmndi:BPMNEdge bpmnElement="_2-_3" >
        <di:waypoint x="136" y="112" />
        <di:waypoint x="240" y="240" />
        <di:waypoint x="282" y="110" />
      </bpmndi:BPMNEdge>
    </bpmndi:BPMNPlane>
  </bpmndi:BPMNDiagram>

</definitions>
```

To create your own process using BPMN 2.0 format, you can

- The jBPM Designer is an open-source web-based editor that supports the BPMN 2.0 format. We have embedded it into jbpm console for BPMN 2.0 process visualization and editing. You could use the Designer (either standalone or integrated) to create / edit BPMN 2.0 processes and then export them to BPMN 2.0 format or save them into repository and import them so they can be executed.
- A new BPMN2 Eclipse plugin is being created to support the full BPMN2 specification.
- You can always manually create your BPMN 2.0 process files by writing the XML directly. You can validate the syntax of your processes against the BPMN 2.0 XSD, or use the validator in the Eclipse plugin to check both syntax and completeness of your model.
- 

  Note

  Drools Eclipse Process editor has been deprecated in favor of BPMN2 Modeler for process modeling. It can still be used for limited number of supported elements but should be faced out as it is not being developed any more.

  Create a new Process file using the Drools Eclipse plugin wizard and in the last page of the wizard, make sure you select Drools 5.1 code compatibility. This will create a new process using the BPMN 2.0 XML format. Note however that this is not exactly a BPMN 2.0 editor, as it still uses different attributes names etc. It does however save the process using valid BPMN 2.0 syntax. Also note that the editor does not support all node types and attributes that are already supported in the execution engine.

The following code fragment shows you how to load a BPMN2 process into your knowledge base ...

```
private static KnowledgeBase createKnowledgeBase() throws Exception {
    KieHelper kieHelper = new KieHelper();
    KieBase kieBase = kieHelper
    .addResource(ResourceFactory.newClassPathResource("sample.bpmn2"))
    .build();

    return kieBase;
}
```

... and how to execute this process ...

```
KieBase kbase = createKnowledgeBase();
KieSession ksession = kbase.newKieSession();
ksession.startProcess("com.sample.HelloWorld");
```

For more detail, check out the chapter on the API and the basics.

6.2. Process

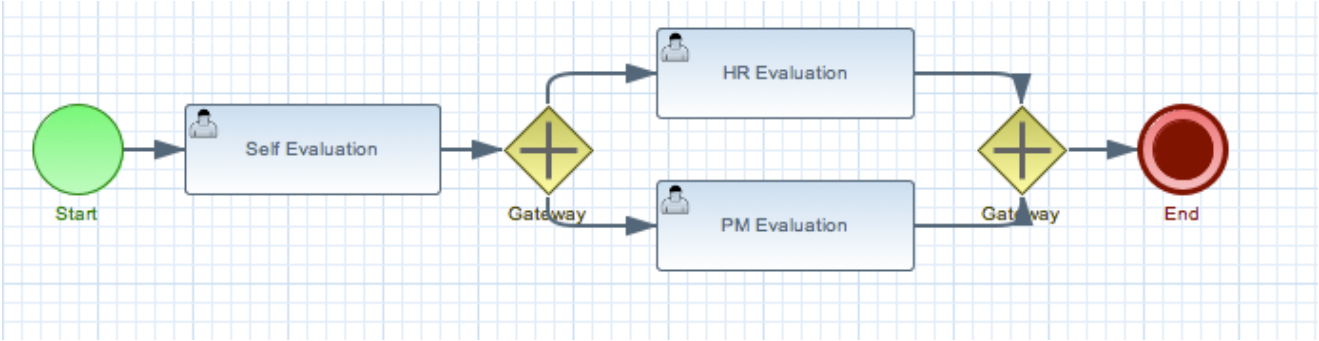


Figure 6.1.

A business process is a graph that describes the order in which a series of steps need to be executed, using a flow chart. A process consists of a collection of nodes that are linked to each other using connections. Each of the nodes represents one step in the overall process while the connections specify how to transition from one node to the other. A large selection of predefined node types have been defined. This chapter describes how to define such processes and use them in your application.

6.2.1. Creating a process

Processes can be created by using one of the following three methods:

1. Using the graphical process editor such as jBPM web designer or Eclipse BPMN2 modeler
2. As an XML file, according to the XML process format as defined in the XML Schema Definition in the BPMN 2.0 specification.
3. By directly creating a process using the Process API.

6.2.1.1. Using the graphical BPMN2 Editor

The graphical BPMN2 editor is an editor that allows you to create a process by dragging and dropping different nodes on a canvas and editing the properties of these nodes. The graphical BPMN2 modeler is an Eclipse plugin hosted on *eclipse.org* [http://www.eclipse.org/bpmn2-modeler/] that provides number of contributors where one of them is jBPM project. Once you have set up a jBPM project (see the installer for creating a working Eclipse environment where you can start), you can start adding processes. When in a project, launch the "New" wizard (use Ctrl+N) or right-click the directory you would like to put your process in and select "New", then "File". Give the file a name and the extension bpmn (e.g. MyProcess.bpmn). This will open up the process editor (you can safely ignore the warning that the file could not be read, this is just because the file is still empty).

First, ensure that you can see the Properties View down the bottom of the Eclipse window, as it will be necessary to fill in the different properties of the elements in your process. If you cannot see the properties view, open it using the menu "Window", then "Show View" and "Other...", and under the "General" folder select the Properties View.

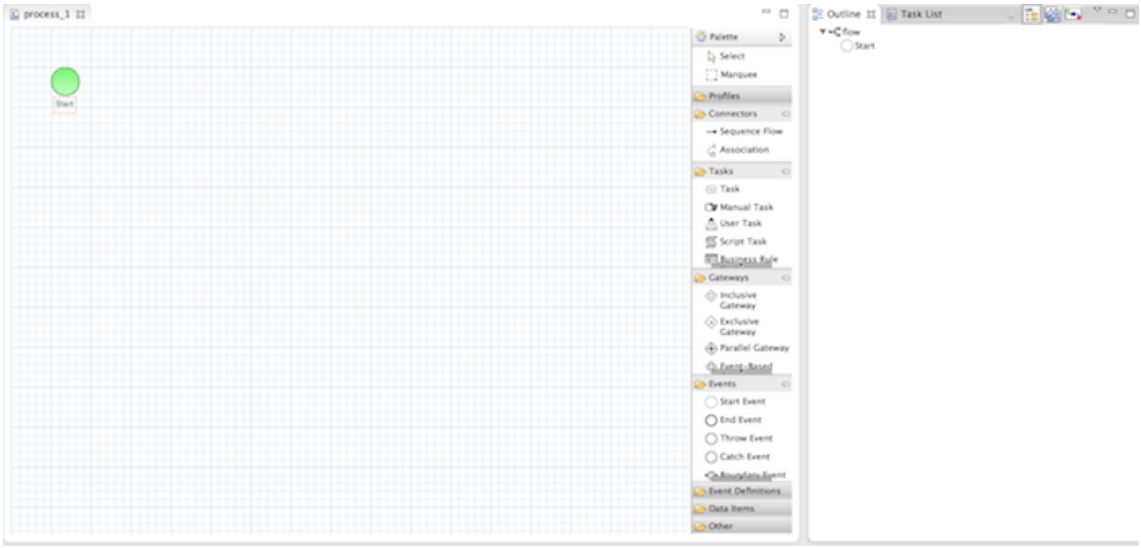


Figure 6.2. New process

The process editor consists of a palette, a canvas and an outline view. To add new elements to the canvas, select the element you would like to create in the palette and then add them to the canvas by clicking on the preferred location. For example, click on the "End Event" icon in the palette of the GUI. Clicking on an element in your process allows you to set the properties of that element. You can connect the nodes (as long as it is permitted by the different types of nodes) by using "Sequence Flow" from the palette.

You can keep adding nodes and connections to your process until it represents the business logic that you want to specify.

6.2.1.2. Defining processes using XML

It is also possible to specify processes using the underlying BPMN 2.0 XML directly. The syntax of these XML processes is defined using the BPMN 2.0 XML Schema Definition. For example, the following XML fragment shows a simple process that contains a sequence of a Start Event, a Script Task that prints "Hello World" to the console, and an End Event.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions id="Definition"
             targetNamespace="http://www.jboss.org/drools"
             typeLanguage="http://www.java.com/javaTypes"
             expressionLanguage="http://www.mvel.org/2.0"
             xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"Rule Task
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
                xsi:schemaLocation="http://www.omg.org/spec/BPMN/20100524/MODEL
  BPMN20.xsd"
             xmlns:g="http://www.jboss.org/drools/flow/gpd"
             xmlns:bpmndi="http://www.omg.org/spec/BPMN/20100524/DI"
             xmlns:dc="http://www.omg.org/spec/DD/20100524/DC"
             xmlns:di="http://www.omg.org/spec/DD/20100524/DI"
             xmlns:tns="http://www.jboss.org/drools">

<process processType="Private" isExecutable="true" id="com.sample.hello" name="Hello
  Process" >

    <!-- nodes -->
    <startEvent id="_1" name="Start" />
    <scriptTask id="_2" name="Hello" >
      <script>System.out.println("Hello World");</script>
    </scriptTask>
    <endEvent id="_3" name="End" >
        <terminateEventDefinition/>
    </endEvent>

    <!-- connections -->
    <sequenceFlow id="_1-_2" sourceRef="_1" targetRef="_2" />
    <sequenceFlow id="_2-_3" sourceRef="_2" targetRef="_3" />

  </process>

  <bpmndi:BPMNDiagram>
    <bpmndi:BPMNPlane bpmnElement="com.sample.hello" >
      <bpmndi:BPMNShape bpmnElement="_1" >
        <dc:Bounds x="16" y="16" width="48" height="48" />
      </bpmndi:BPMNShape>
      <bpmndi:BPMNShape bpmnElement="_2" >
        <dc:Bounds x="96" y="16" width="80" height="48" />
      </bpmndi:BPMNShape>
      <bpmndi:BPMNShape bpmnElement="_3" >
        <dc:Bounds x="208" y="16" width="48" height="48" />
      </bpmndi:BPMNShape>
      <bpmndi:BPMNEdge bpmnElement="_1-_2" >
        <di:waypoint x="40" y="40" />
        <di:waypoint x="136" y="40" />
      </bpmndi:BPMNEdge>
      <bpmndi:BPMNEdge bpmnElement="_2-_3" >
        <di:waypoint x="136" y="40" />
        <di:waypoint x="232" y="40" />
      </bpmndi:BPMNEdge>
    </bpmndi:BPMNPlane>
  </bpmndi:BPMNDiagram>
```

```
</definitions>
```

The process XML file consists of two parts, the top part (the "process" element) contains the definition of the different nodes and their properties, the lower part (the "BPMNDiagram" element) contains all graphical information, like the location of the nodes. The process XML consist of exactly one <process> element. This element contains parameters related to the process (its type, name, id and package name), and consists of three subsections: a header section (where process-level information like variables, globals, imports and lanes can be defined), a nodes section that defines each of the nodes in the process, and a connections section that contains the connections between all the nodes in the process. In the nodes section, there is a specific element for each node, defining the various parameters and, possibly, sub-elements for that node type.

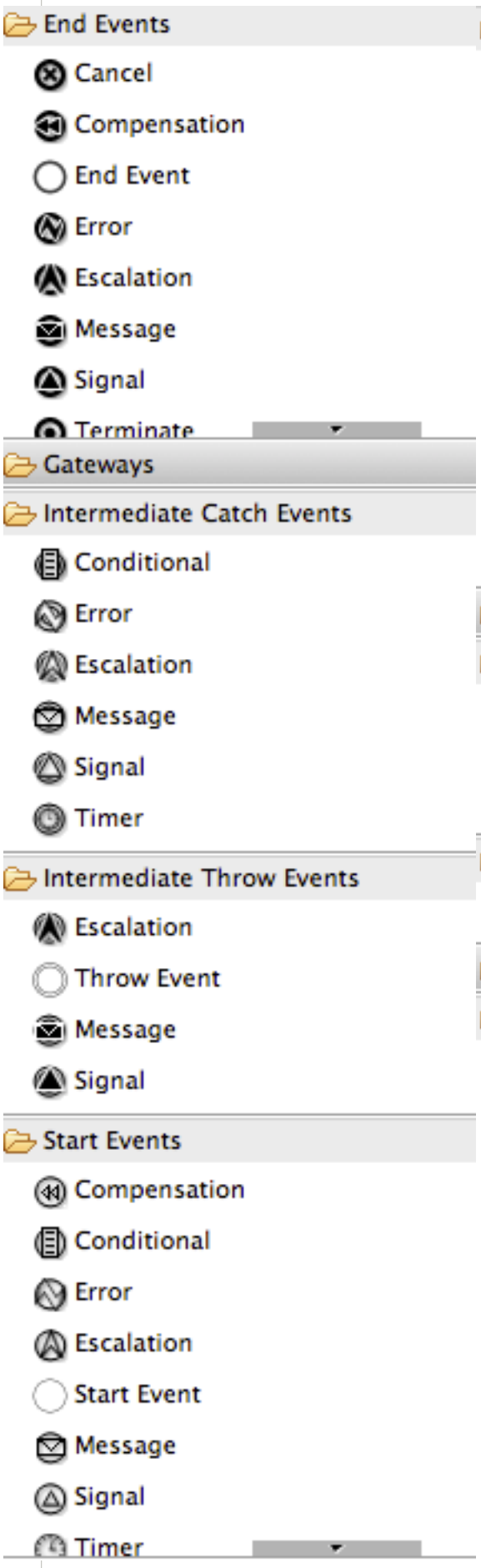


Figure 6.3. The different types of BPMN2 events

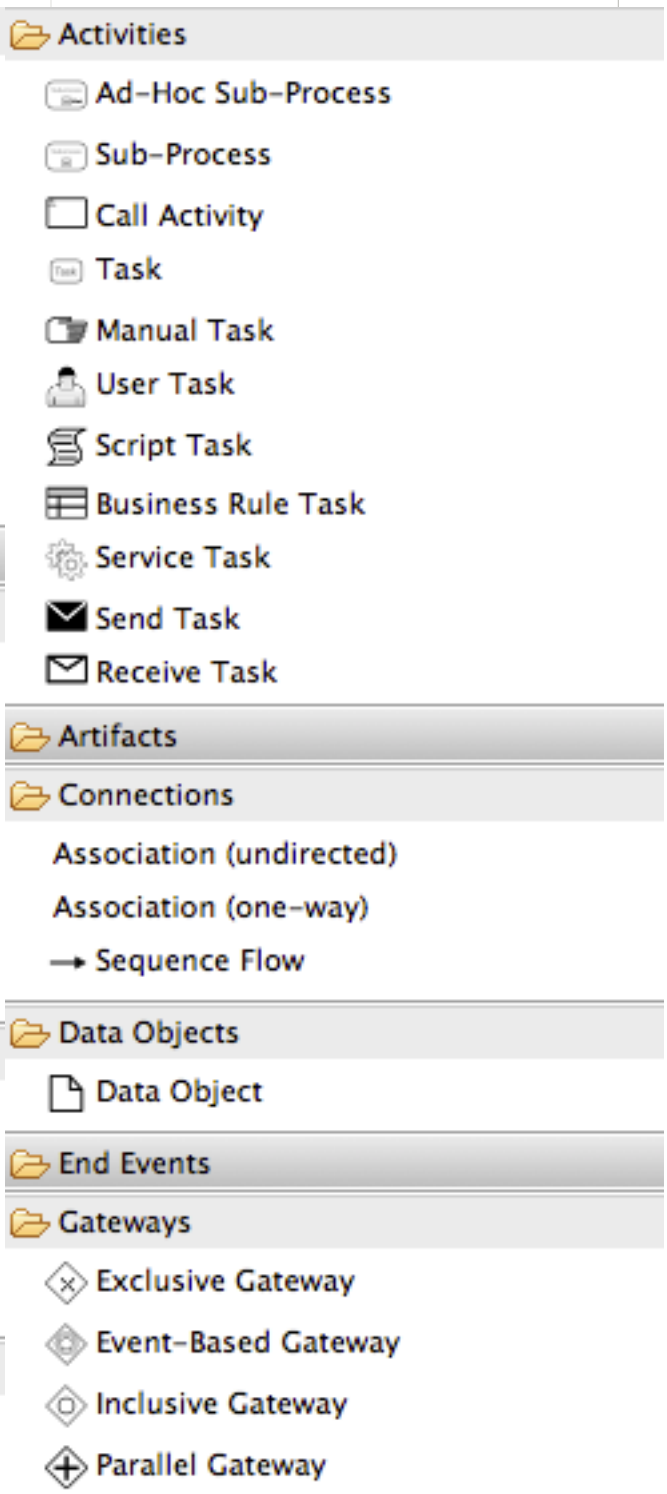


Figure 6.4. The different types of BPMN2 activities and gateways

6.2.1.3. Details: Process properties

A BPMN2 process is a flow chart where different types of nodes are linked using connections. The process itself exposes the following properties:

- *Id*: The unique id of the process.
- *Name*: The display name of the process.
- *Version*: The version number of the process.
- *Package*: The package (namespace) the process is defined in.

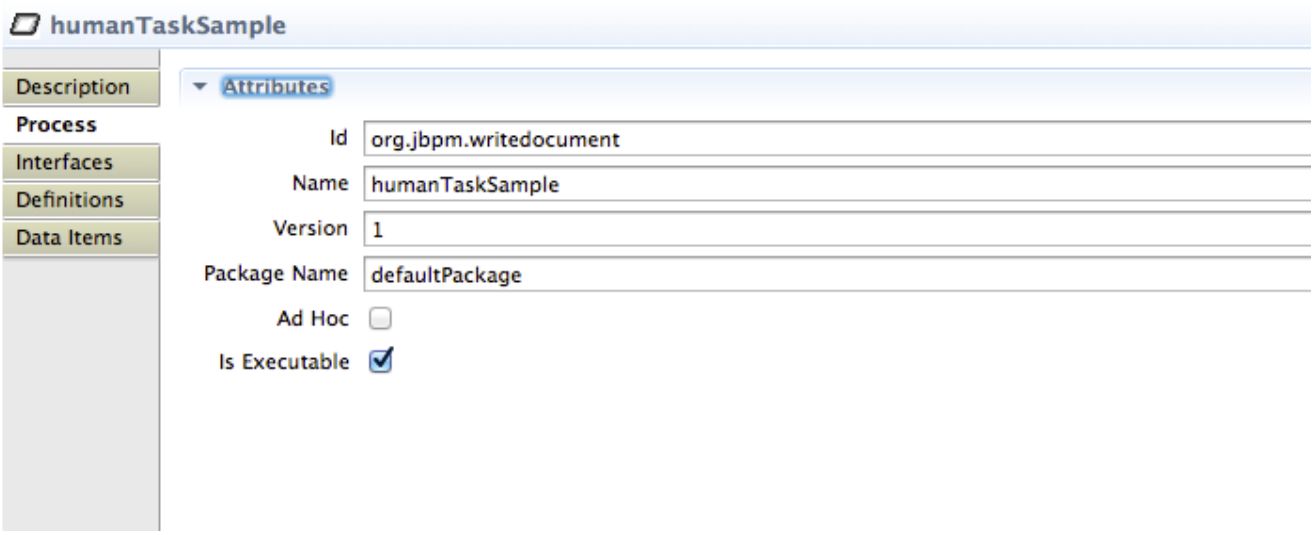


Figure 6.5. BPMN2 process properties

In addition to that following can be defined as well:

- *Variables*: Variables can be defined to store data during the execution of your process. See section "*???*" for details.
- *Swimlanes*: Specify the swimlanes used in this process for assigning human tasks. See chapter "*???*" for details.

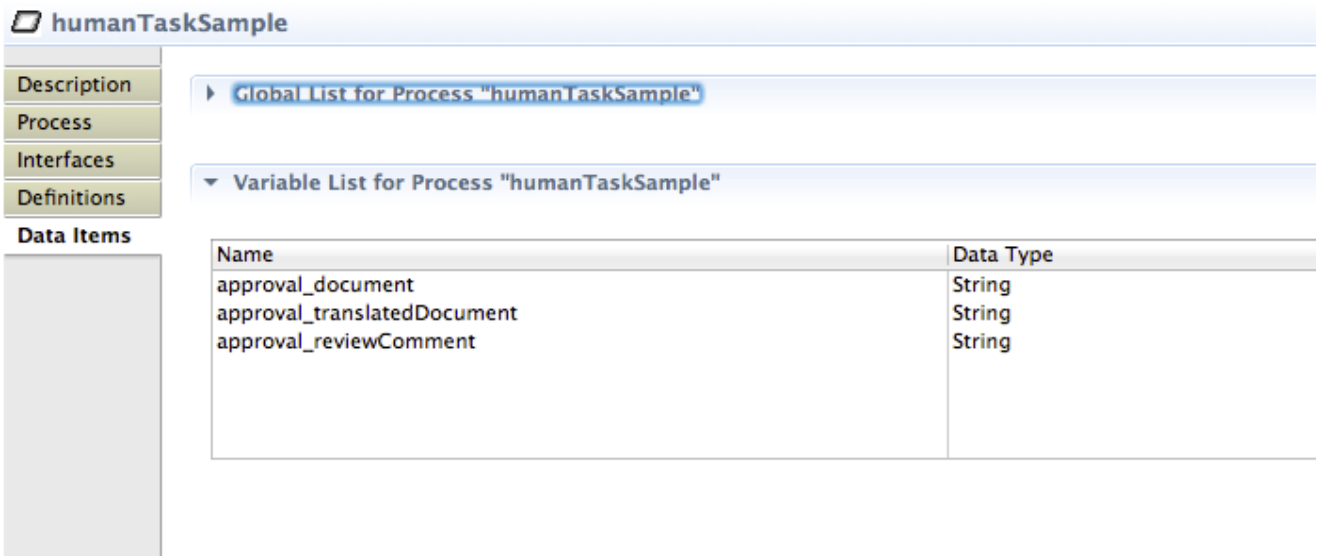


Figure 6.6. BPMN2 process variables

6.3. Activities

6.3.1. Script task

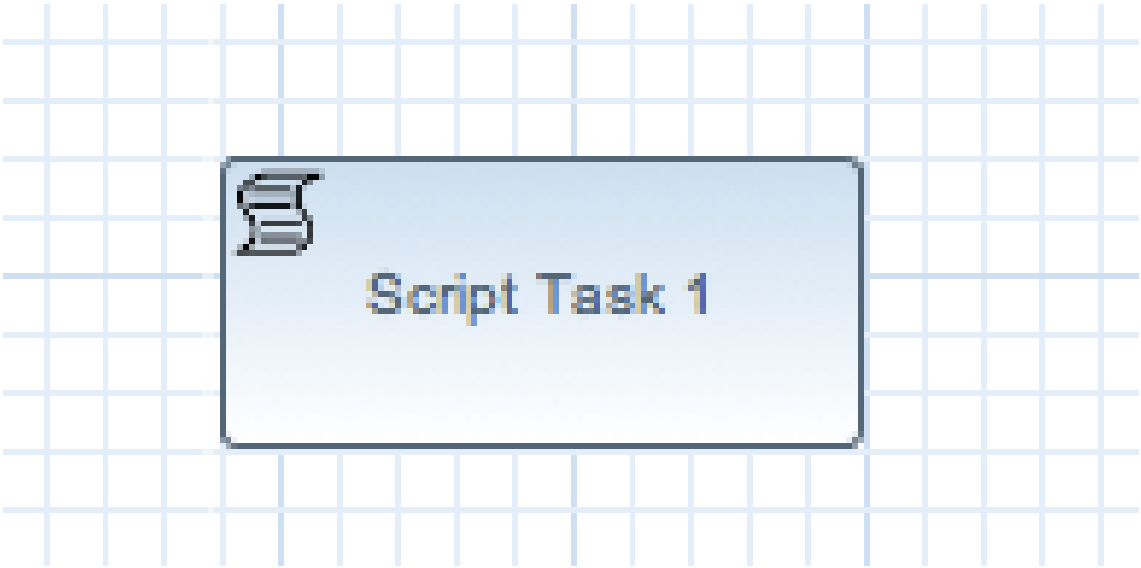


Figure 6.7. Script task

Represents a script that should be executed in this process. A Script Task should have one incoming connection and one outgoing connection. The associated action specifies what should be executed, the dialect used for coding the action (i.e., Java or MVEL), and the actual action code. This code can access any variables and globals. There is also a predefined variable `kcontext` that references the *`ProcessContext`* [http://docs.jboss.org/jbpm/v6.0.1/javadocs/org/kie/api/runtime/

process/ProcessContext.html] object (which can, for example, be used to access the current `ProcessInstance` or `NodeInstance`, and to get and set variables, or get access to the ksession using `kcontext.getKieRuntime()`). When a Script Task is reached in the process, it will execute the action and then continue with the next node. It contains the following properties:

- *Id*: The id of the node (which is unique within one node container).
- *Name*: The display name of the node.
- *Action*: The action script associated with this action node.

Note that you can write any valid Java code inside a script node. This basically allows you to do anything inside such a script node. There are some caveats however:

- When trying to create a higher-level business process, that should also be understood by business users, it is probably wise to avoid low-level implementation details inside the process, including inside these script tasks. A Script Task could still be used to quickly manipulate variables etc. but other concepts like a Service Task could be used to model more complex behaviour in a higher-level manner.
- Scripts should be immediate. They are using the engine thread to execute the script. Scripts that could take some time to execute should probably be modeled as an asynchronous Service Task.
- You should try to avoid contacting external services through a script node. Not only does this usually violate the first two caveats, it is also interacting with external services without the knowledge of the engine, which can be problematic, especially when using persistence and transactions. In general, it is probably wiser to model communication with an external service using a service task.
- Scripts should not throw exceptions. Runtime exceptions should be caught and for example managed inside the script or transformed into signals or errors that can then be handled inside the process.

6.3.2. Service task

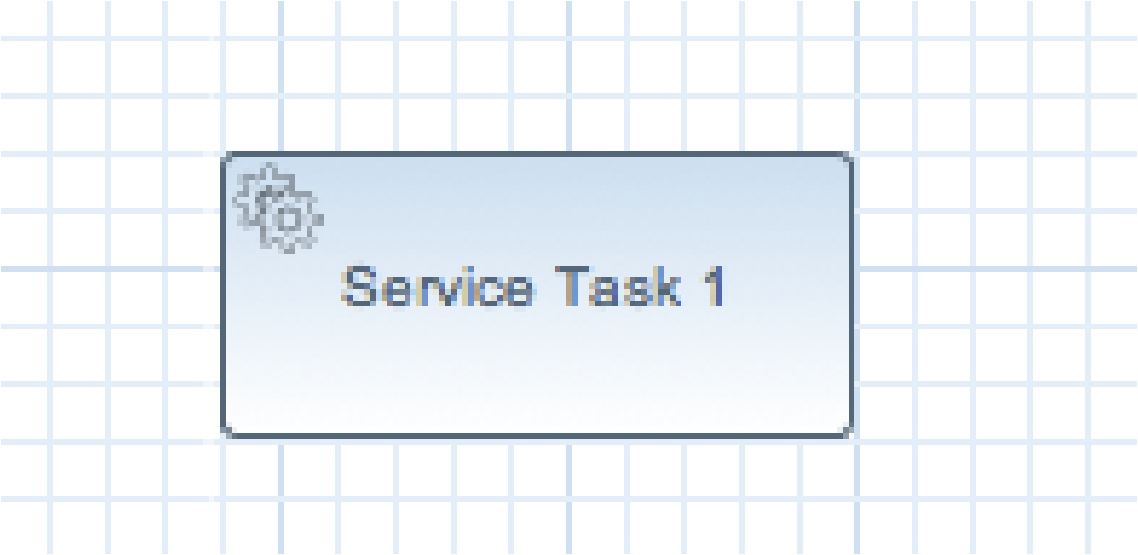


Figure 6.8. Service task

Represents an (abstract) unit of work that should be executed in this process. All work that is executed outside the process engine should be represented (in a declarative way) using a Service Task. Different types of services are predefined, e.g., sending an email, logging a message, etc. Users can define domain-specific services or work items, using a unique name and by defining the parameters (input) and results (output) that are associated with this type of work. Check the chapter on domain-specific processes for a detailed explanation and illustrative examples of how to define and use work items in your processes. When a Service Task is reached in the process, the associated work is executed. A Service Task should have one incoming connection and one outgoing connection.

- *Id*: The id of the node (which is unique within one node container).
- *Name*: The display name of the node.
- *Parameter mapping*: Allows copying the value of process variables to parameters of the work item. Upon creation of the work item, the values will be copied.
- *Result mapping*: Allows copying the value of result parameters of the work item to a process variable. Each type of work can define result parameters that will (potentially) be returned after the work item has been completed. A result mapping can be used to copy the value of the given result parameter to the given variable in this process. For example, the "FileFinder" work item returns a list of files that match the given search criteria within the result parameter `Files`. This list of files can then be bound to a process variable for use within the process. Upon completion of the work item, the values will be copied.
- *On-entry and on-exit actions*: Actions that are executed upon entry or exit of this node, respectively.

- *Additional parameters*: Each type of work item can define additional parameters that are relevant for that type of work. For example, the "Email" work item defines additional parameters such as `From`, `To`, `Subject` and `Body`. The user can either provide values for these parameters directly, or define a parameter mapping that will copy the value of the given variable in this process to the given parameter; if both are specified, the mapping will have precedence. Parameters of type `String` can use `#{expression}` to embed a value in the string. The value will be retrieved when creating the work item, and the substitution expression will be replaced by the result of calling `toString()` on the variable. The expression could simply be the name of a variable (in which case it resolves to the value of the variable), but more advanced MVEL expressions are possible as well, e.g., `#{person.name.firstname}`.

6.3.3. User task

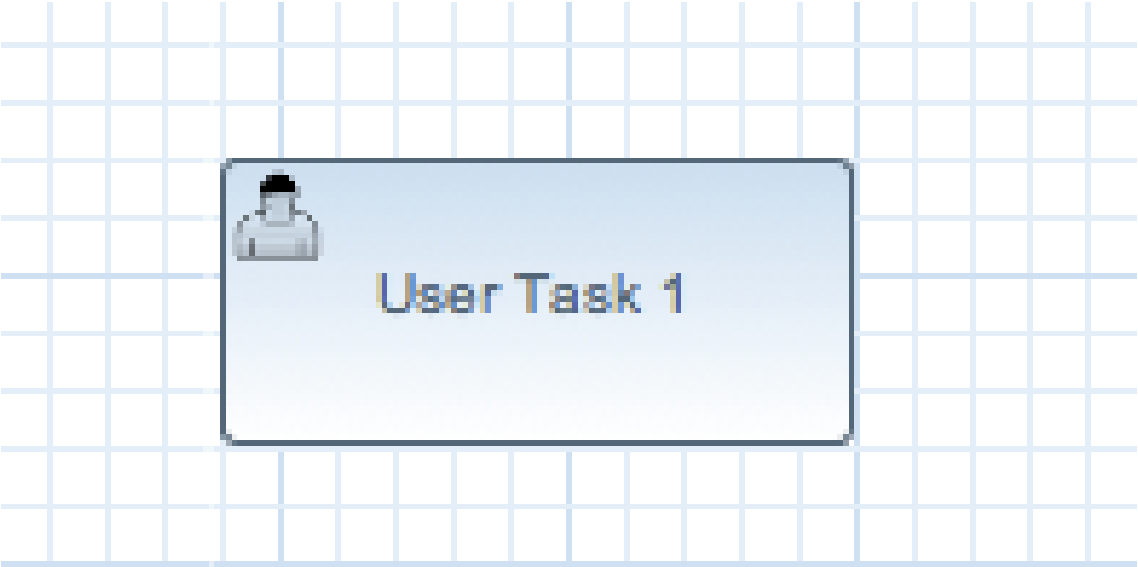


Figure 6.9. User task

Processes can also involve tasks that need to be executed by human actors. A User Task represents an atomic task to be executed by a human actor. It should have one incoming connection and one outgoing connection. User Tasks can be used in combination with Swimlanes to assign multiple human tasks to similar actors. Refer to the chapter on human tasks for more details. A User Task is actually nothing more than a specific type of service node (of type "Human Task"). A User Task contains the following properties:

- *Id*: The id of the node (which is unique within one node container).
- *Name*: The display name of the node.
- *TaskName*: The name of the human task.
- *Priority*: An integer indicating the priority of the human task.

- *Comment*: A comment associated with the human task.
- *ActorId*: The actor id that is responsible for executing the human task. A list of actor id's can be specified using a comma (',') as separator.
- *GroupId*: The group id that is responsible for executing the human task. A list of group id's can be specified using a comma (',') as separator.
- *Skippable*: Specifies whether the human task can be skipped, i.e., whether the actor may decide not to execute the task.
- *Content*: The data associated with this task.
- *Swimlane*: The swimlane this human task node is part of. Swimlanes make it easy to assign multiple human tasks to the same actor. See the human tasks chapter for more detail on how to use swimlanes.
- *On entry and on exit actions*: Action scripts that are executed upon entry and exit of this node, respectively.
- *Parameter mapping*: Allows copying the value of process variables to parameters of the human task. Upon creation of the human tasks, the values will be copied.
- *Result mapping*: Allows copying the value of result parameters of the human task to a process variable. Upon completion of the human task, the values will be copied. A human task has a result variable "Result" that contains the data returned by the human actor. The variable "ActorId" contains the id of the actor that actually executed the task.

A user task should define the type of task that needs to be executed (using properties like TaskName, Comment, etc.) and who needs to perform it (using either actorId or groupId). Note that if there is data related to this specific process instance that the end user needs when performing the task, this data should be passed as the content of the task. The task for example does not have access to process variables. Check out the chapter on human tasks to get more detail on how to pass data between human tasks and the process instance.

6.3.4. Reusable sub-process

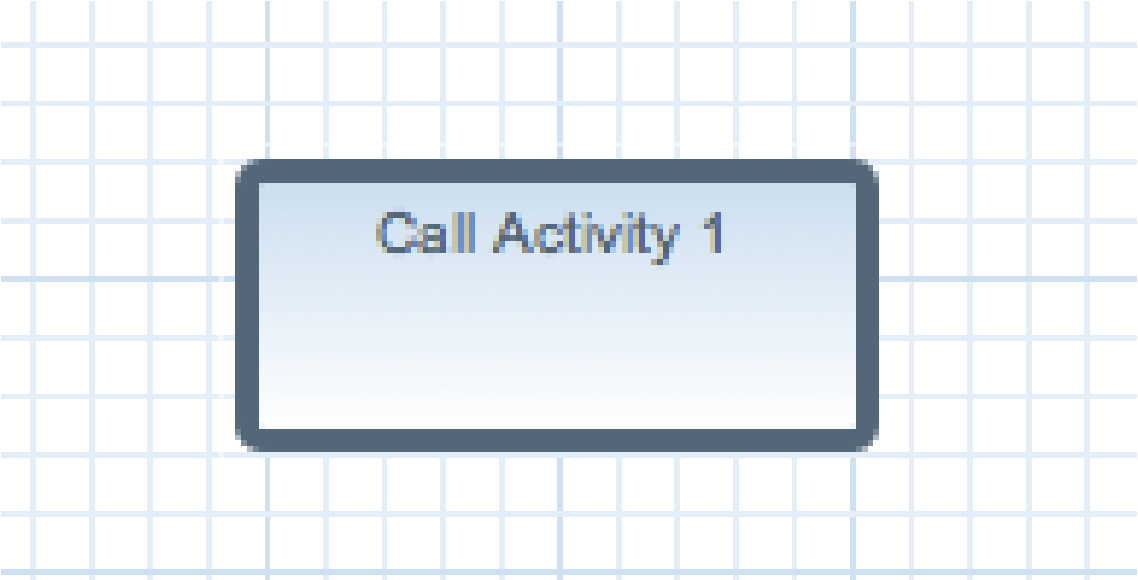


Figure 6.10. Reusable sub-process - Call activity

Represents the invocation of another process from within this process. A sub-process node should have one incoming connection and one outgoing connection. When a Reusable Sub-Process node is reached in the process, the engine will start the process with the given id. It contains the following properties:

- *Id*: The id of the node (which is unique within one node container).
- *Name*: The display name of the node.
- *ProcessId*: The id of the process that should be executed.
- *Wait for completion* (by default true): If this property is true, this sub-process node will only continue if the child process that was started has terminated its execution (completed or aborted); otherwise it will continue immediately after starting the subprocess (so it will not wait for its completion).
- *Independent* (by default true): If this property is true, the child process is started as an independent process, which means that the child process will not be terminated if this parent process is completed (or this sub-process node is canceled for some other reason); otherwise the active sub-process will be canceled on termination of the parent process (or cancellation of the sub-process node). Note that you can only set independent to "false" only when "Wait for completion" is set to true.
- *On-entry and on-exit actions*: Actions that are executed upon entry or exit of this node, respectively.

- *Parameter in/out mapping*: A sub-process node can also define in- and out-mappings for variables. The variables given in the "in" mapping will be used as parameters (with the associated parameter name) when starting the process. The variables of the child process that are defined for the "out" mappings will be copied to the variables of this process when the child process has been completed. Note that you can use "out" mappings only when "Wait for completion" is set to true.

6.3.5. Business rule task

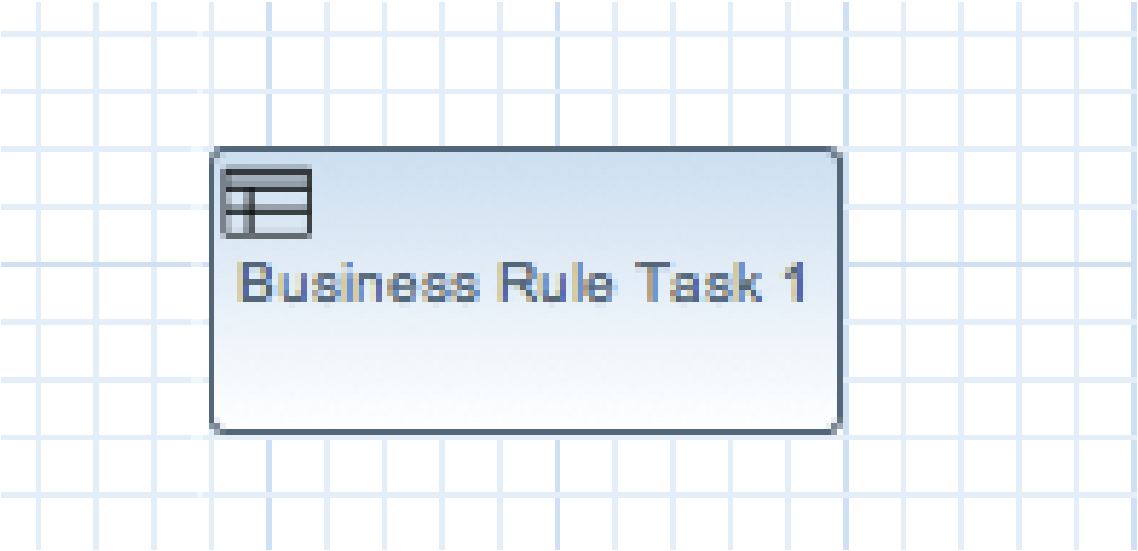


Figure 6.11. Business rule task

A Business Rule Task Represents a set of rules that need to be evaluated. The rules are evaluated when the node is reached. A Rule Task should have one incoming connection and one outgoing connection. Rules are defined in separate files using the Drools rule format. Rules can become part of a specific ruleflow group using the `ruleflow-group` attribute in the header of the rule.

When a Rule Task is reached in the process, the engine will start executing rules that are part of the corresponding ruleflow-group (if any). Execution will automatically continue to the next node if there are no more active rules in this ruleflow group. As a result, during the execution of a ruleflow group, new activations belonging to the currently active ruleflow group can be added to the Agenda due to changes made to the facts by the other rules. Note that the process will immediately continue with the next node if it encounters a ruleflow group where there are no active rules at that time.

If the ruleflow group was already active, the ruleflow group will remain active and execution will only continue if all active rules of the ruleflow group has been completed. It contains the following properties:

- *Id*: The id of the node (which is unique within one node container).

- *Name*: The display name of the node.
- *RuleFlowGroup*: The name of the ruleflow group that represents the set of rules of this RuleFlowGroup node.

6.3.6. Embedded sub-process

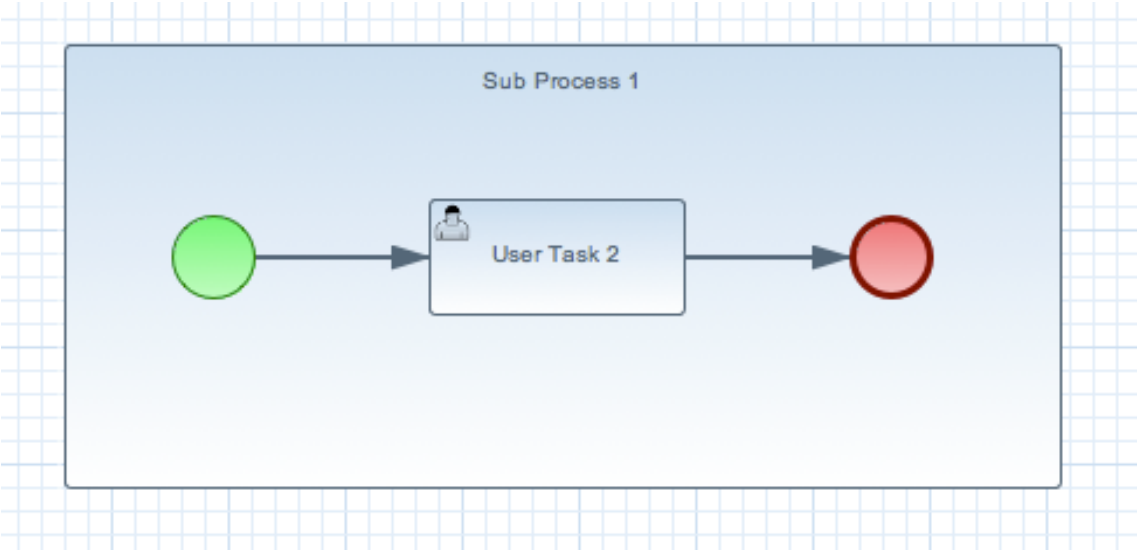


Figure 6.12. Embedded sub-process

A Sub-Process is a node that can contain other nodes so that it acts as a node container. This allows not only the embedding of a part of the process within such a sub-process node, but also the definition of additional variables that are accessible for all nodes inside this container. A sub-process should have one incoming connection and one outgoing connection. It should also contain one start node that defines where to start (inside the Sub-Process) when you reach the sub-process. It should also contain one or more end events. Note that, if you use a terminating event node inside a sub-process, you are terminating just that sub-process. A sub-process ends when there are no more active nodes inside the sub-process. It contains the following properties:

- *Id*: The id of the node (which is unique within one node container).
- *Name*: The display name of the node.
- *Variables*: Additional variables can be defined to store data during the execution of this node. See section “???” for details.

6.3.7. Multi-instance sub-process

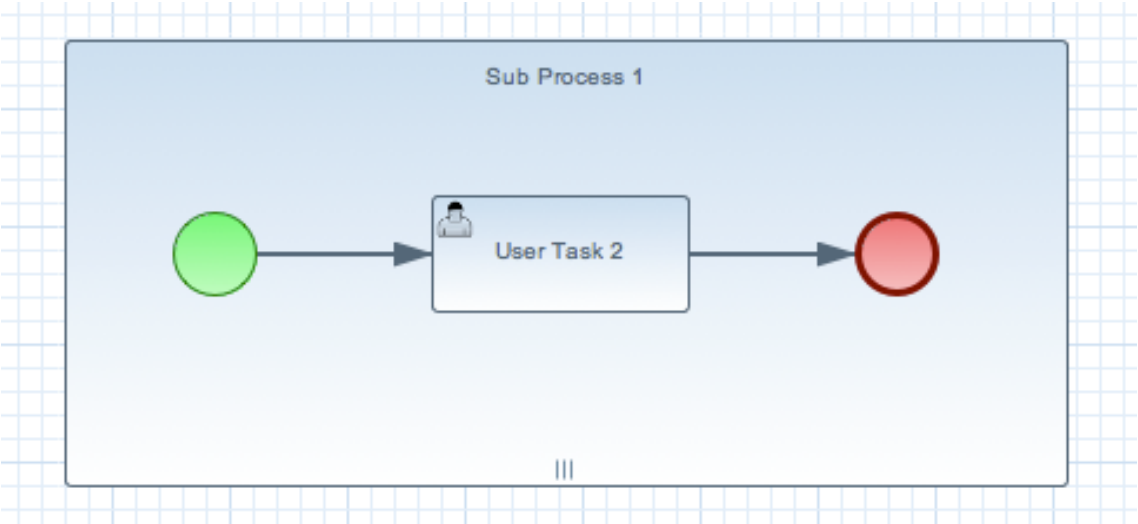


Figure 6.13. Multi-instance sub-process

A Multiple Instance sub-process is a special kind of sub-process that allows you to execute the contained process segment multiple times, once for each element in a collection. A multiple instance sub-process should have one incoming connection and one outgoing connection. It waits until the embedded process fragment is completed for each of the elements in the given collection before continuing. It contains the following properties:

- *Id*: The id of the node (which is unique within one node container).
- *Name*: The display name of the node.
- *CollectionExpression*: The name of a variable that represents the collection of elements that should be iterated over. The collection variable should be an array or of type `java.util.Collection`. If the collection expression evaluates to null or an empty collection, the multiple instances sub-process will be completed immediately and follow its outgoing connection.
- *VariableName*: The name of the variable to contain the current element from the collection. This gives nodes within the composite node access to the selected element.

6.4. Events

6.4.1. Start event

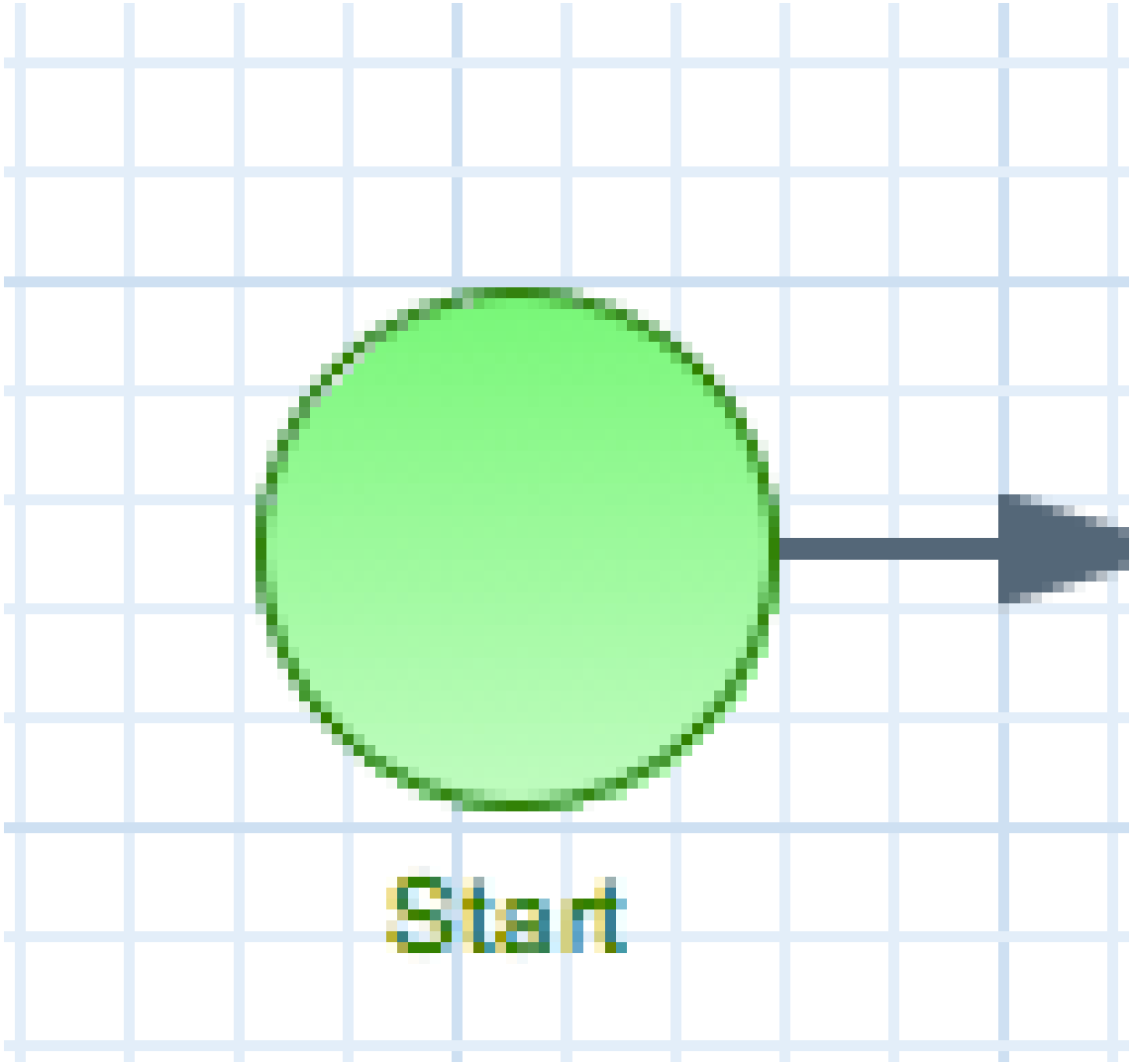


Figure 6.14. Start event

The start of the process. A process should have exactly one start node (none start node which does not have event definitions), which cannot have incoming connections and should have one outgoing connection. Whenever a process is started, execution will start at this node and automatically continue to the first node linked to this start event, and so on. It contains the following properties:

- *Id*: The id of the node (which is unique within one node container).
- *Name*: The display name of the node.

6.4.2. End events

6.4.2.1. End event

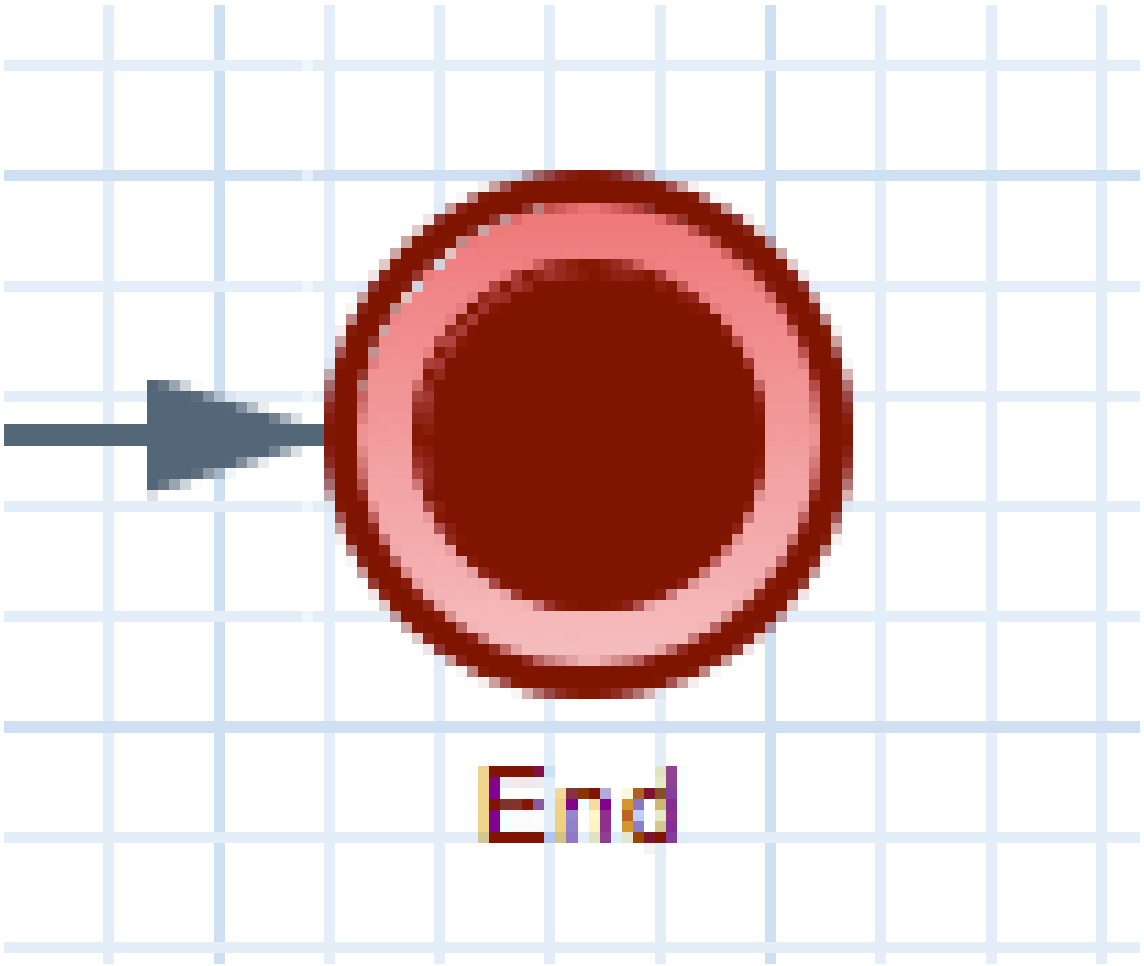


Figure 6.15. End event

The end of the process. A process should have one or more end events. The End Event should have one incoming connection and cannot have any outgoing connections. It contains the following properties:

- *Id*: The id of the node (which is unique within one node container).
- *Name*: The display name of the node.
- *Terminate*: An End Event can terminate the entire process or just the path. When a process instance is terminated, it means its state is set to completed and all other nodes that might still be active (on parallel paths) in this process instance are canceled. Non-terminating end events are simply end for this path (execution of this branch will end here), but other parallel paths can still continue. A process instance will automatically complete if there are no more active paths inside that process instance (for example, if a process instance reaches a non-terminating end node but there are no more active branches inside the process instance, the process instance

will be completed anyway). Terminating end events are visualized using a full circle inside the event node, non-terminating event nodes are empty. Note that, if you use a terminating event node inside a sub-process, you are terminating just that sub-process and top level continues.

6.4.2.2. Throwing error event

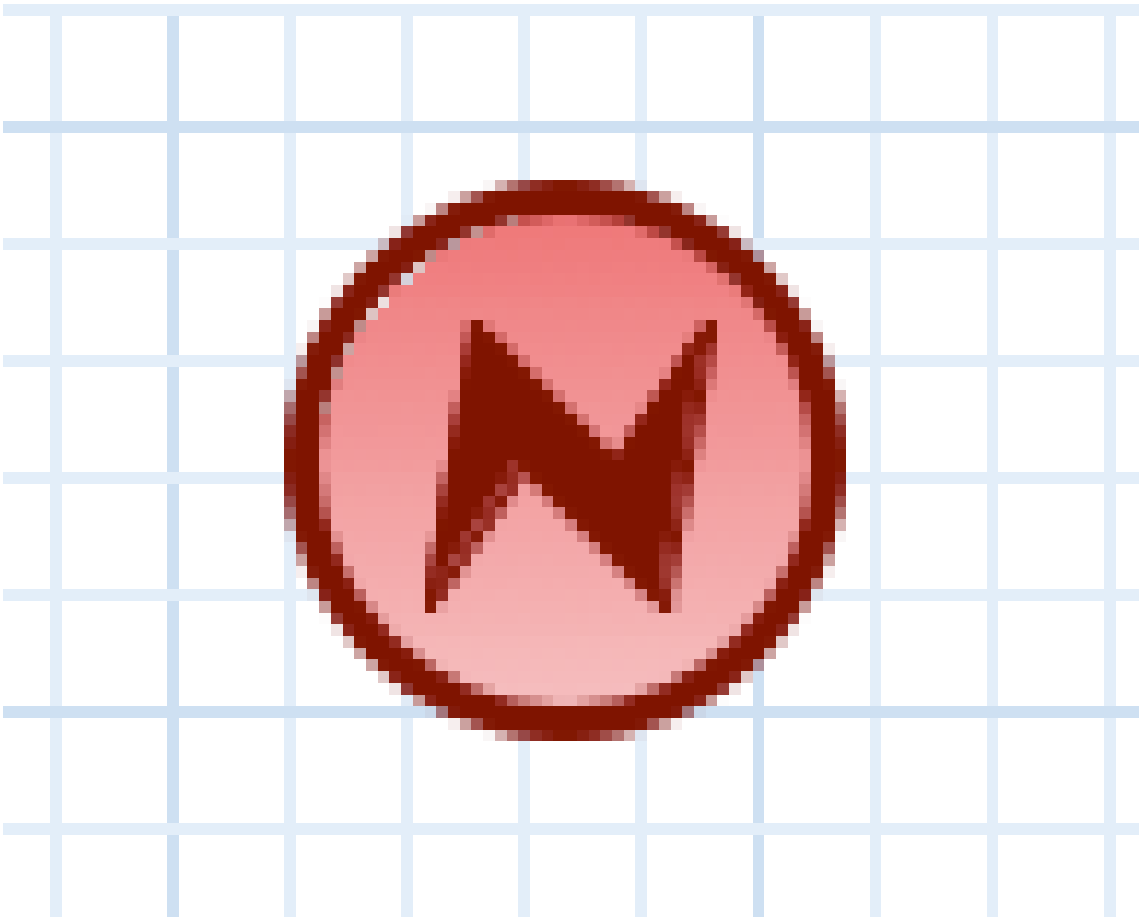


Figure 6.16. Throwing error event

An Error Event can be used to signal an exceptional condition in the process. It should have one incoming connection and no outgoing connections. When an Error Event is reached in the process, it will throw an error with the given name. The process will search for an appropriate error handler that is capable of handling this kind of fault. If no error handler is found, the process instance will be aborted. An Error Event contains the following properties:

- *Id*: The id of the node (which is unique within one node container).
- *Name*: The display name of the node.
- *FaultName*: The name of the fault. This name is used to search for appropriate exception handlers that are capable of handling this kind of fault.
- *FaultVariable*: The name of the variable that contains the data associated with this fault. This data is also passed on to the exception handler (if one is found).

Error handlers can be specified using boundary events.

6.4.3. Intermediate events

6.4.3.1. Catching timer event

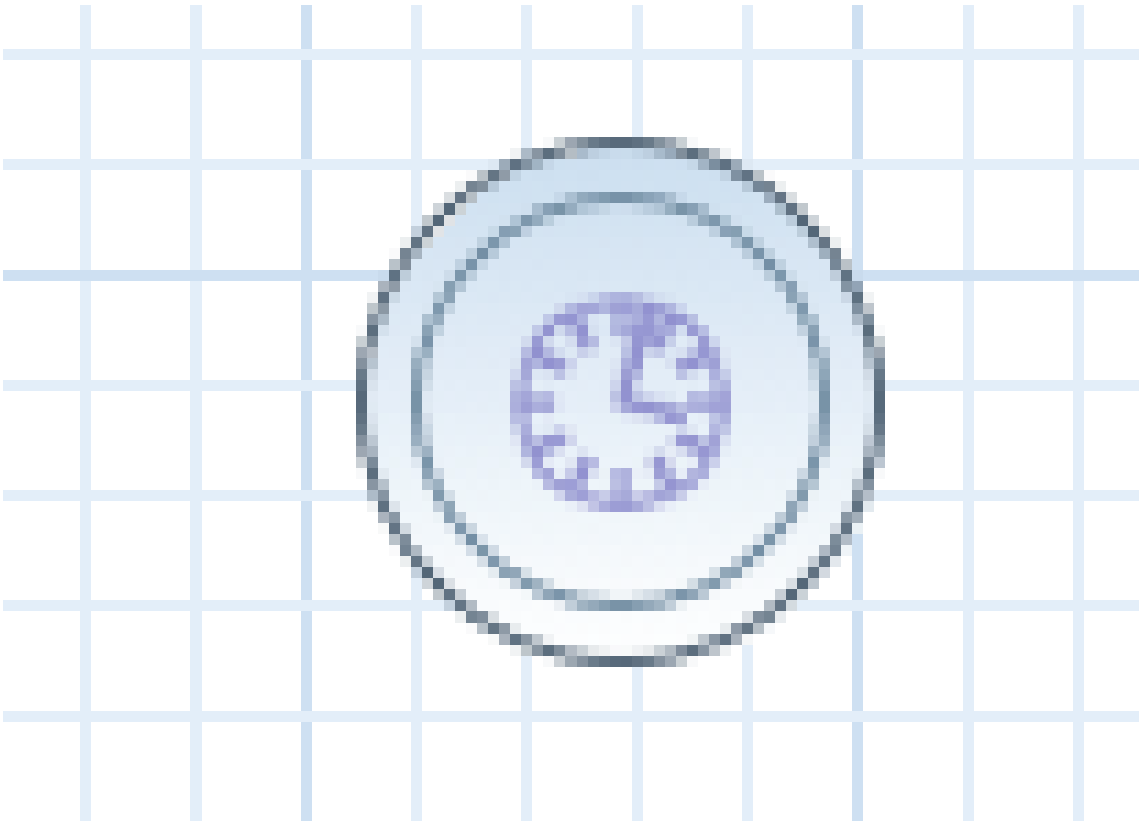


Figure 6.17. Catching timer event

Represents a timer that can trigger one or multiple times after a given period of time. A Timer Event should have one incoming connection and one outgoing connection. The timer delay specifies how long the timer should wait before triggering the first time. When a Timer Event is reached in the process, it will start the associated timer. The timer is canceled if the timer node is canceled (e.g., by completing or aborting the enclosing process instance). Consult the section “*???*” for more information. The Timer Event contains the following properties:

- *Id*: The id of the node (which is unique within one node container).
- *Name*: The display name of the node.
- *Timer delay*: The delay that the node should wait before triggering the first time. The expression should be of the form `[#d][#h][#m][#s][#[ms]]`. This allows you to specify the number of days, hours, minutes, seconds and milliseconds (which is the default if you don't specify anything). For example, the expression "1h" will wait one hour before triggering the timer. The expression could also use #{expr} to dynamically derive the delay based on some process

variable. Expr in this case could be a process variable, or a more complex expression based on a process variable (e.g. myVariable.getValue()).

- *Timer period*: The period between two subsequent triggers. If the period is 0, the timer should only be triggered once. The expression should be of the form `[#d][#h][#m][#s][#[ms]]`. You can specify the number of days, hours, minutes, seconds and milliseconds (which is the default if you don't specify anything). For example, the expression "1h" will wait one hour before triggering the timer again. The expression could also use #{expr} to dynamically derive the period based on some process variable. Expr in this case could be a process variable, or a more complex expression based on a process variable (e.g. myVariable.getValue()).

Timer events could also be specified as boundary events on sub-processes and tasks that are not automatic tasks like script task that have no wait state as timer will not have a change to fire before task completion.

6.4.3.2. Catching signal event

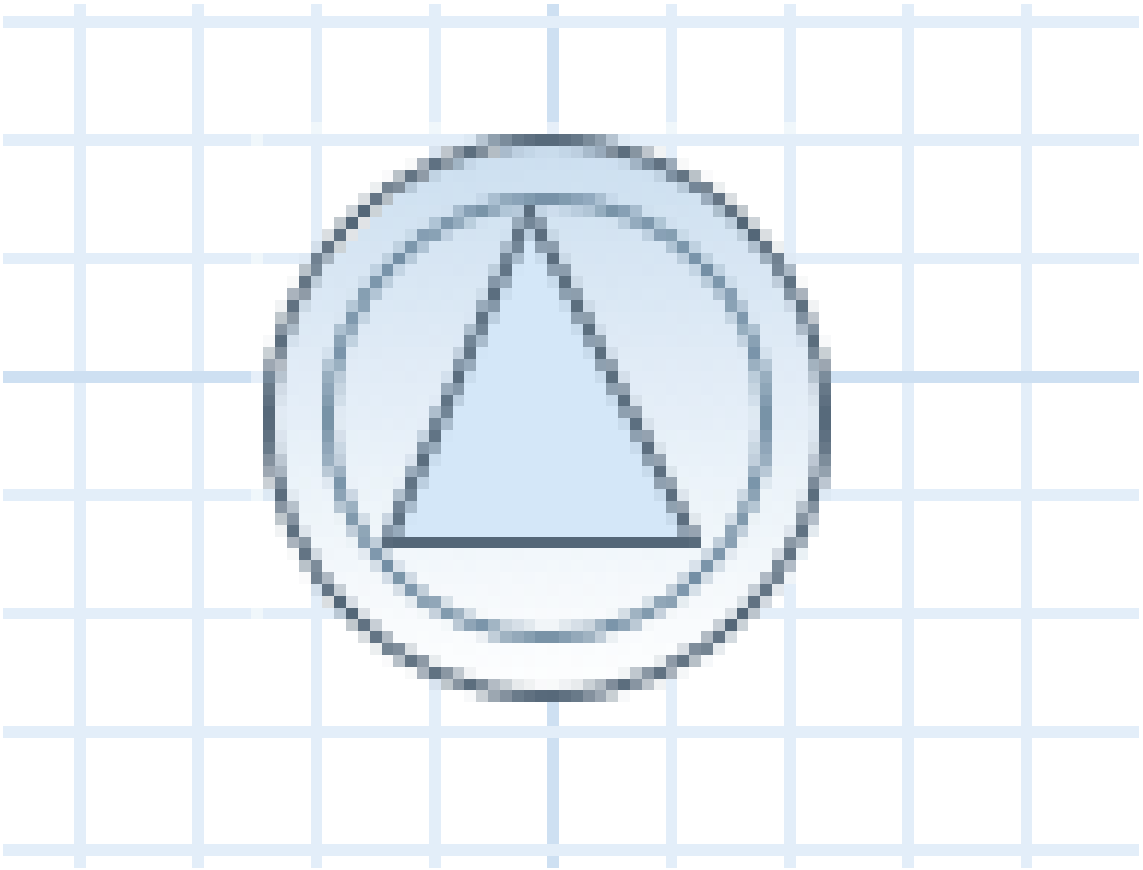


Figure 6.18. Catching signal event

A Signal Event can be used to respond to internal or external events during the execution of the process. A Signal Event should have one incoming connections and one outgoing connection. It specifies the type of event that is expected. Whenever that type of event is detected, the node connected to this event node will be triggered. It contains the following properties:

- *Id*: The id of the node (which is unique within one node container).
- *Name*: The display name of the node.
- *EventType*: The type of event that is expected.
- *VariableName*: The name of the variable that will contain the data associated with this event (if any) when this event occurs.

A process instance can be signaled that a specific event occurred using

```
ksession.signalEvent(eventType, data, processInstanceId)
```

This will trigger all (active) signal event nodes in the given process instance that are waiting for that event type. Data related to the event can be passed using the data parameter. If the event node specifies a variable name, this data will be copied to that variable when the event occurs.

It is also possible to use event nodes inside sub-processes. These event nodes will however only be active when the sub-process is active.

You can also generate a signal from inside a process instance. A script (in a script task or using on entry or on exit actions) can use

```
kcontext.getKieRuntime().signalEvent(eventType,                          data,
 kcontext.getProcessInstance().getId());
```

A throwing signal event could also be used to model the signaling of an event.

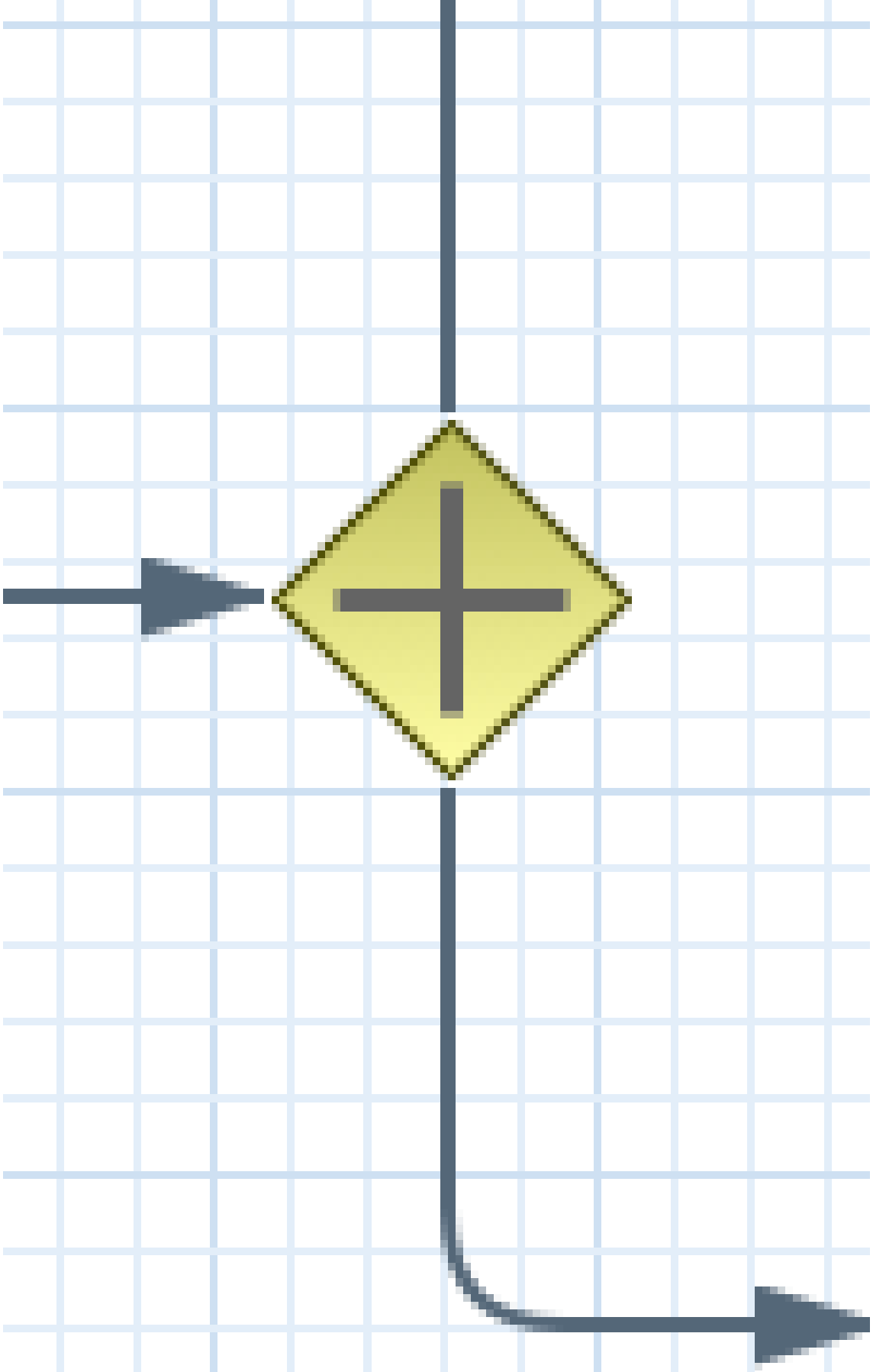


Figure 6.19. Diverging gateway

Allows you to create branches in your process. A Diverging Gateway should have one incoming connection and two or more outgoing connections. There are three types of gateway nodes currently supported:

- AND or parallel means that the control flow will continue in all outgoing connections simultaneously.
- XOR or exclusive means that exactly one of the outgoing connections will be chosen. The decision is made by evaluating the constraints that are linked to each of the outgoing connections. The constraint with the *lowest* priority number that evaluates to true is selected. Constraints can be specified using different dialects. Note that you should always make sure that at least one of the outgoing connections will evaluate to true at runtime (the engine will throw an exception at runtime if it cannot find at least one outgoing connection).
- OR or inclusive means that all outgoing connections whose condition evaluates to true are selected. Conditions are similar to the exclusive gateway, except that no priorities are taken into account. Note that you should make sure that at least one of the outgoing connections will evaluate to true at runtime because the engine will throw an exception at runtime if it cannot determine an outgoing connection.

It contains the following properties:

- *Id*: The id of the node (which is unique within one node container).
- *Name*: The display name of the node.
- *Type*: The type of the split node, i.e., AND, XOR or OR (see above).
- *Constraints*: The constraints linked to each of the outgoing connections (in case of an exclusive or inclusive gateway).

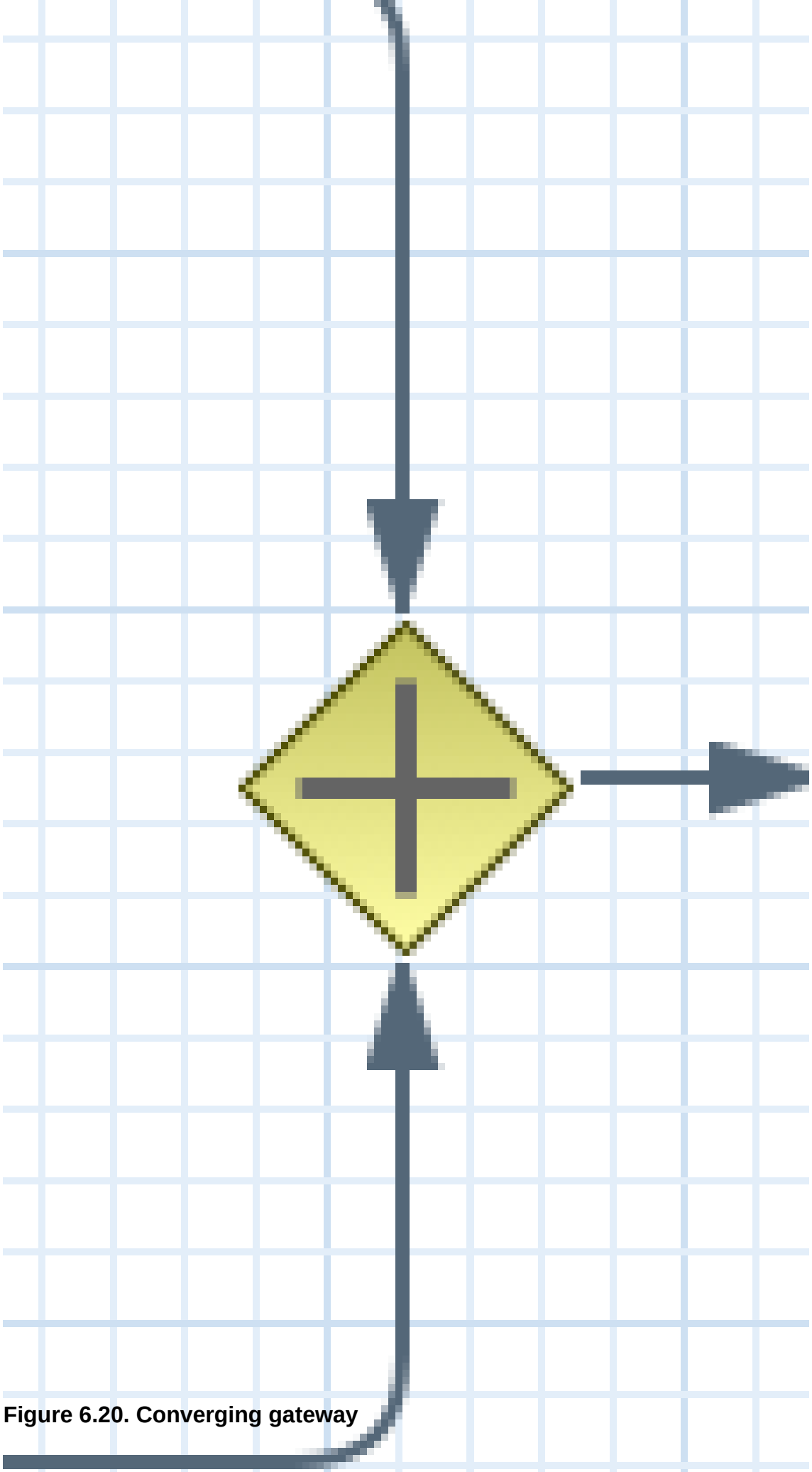


Figure 6.20. Converging gateway

Allows you to synchronize multiple branches. A Converging Gateway should have two or more incoming connections and one outgoing connection. There are three types of splits currently supported:

- AND or parallel means that is will wait until *all* incoming branches are completed before continuing.
- XOR or exclusive means that it continues as soon as *one* of its incoming branches has been completed. If it is triggered from more than one incoming connection, it will trigger the next node for each of those triggers.
- OR or inclusive means that it continues as soon as *all direct active paths* of its incoming branches has been completed. This is complex merge behaviour that is described in BPMN2 specification but in most cases it means that OR join will wait for all active flows that started in OR split. Some advanced cases (including other gateways in between or repeatable timers) will be causing different "direct active path" calculation.

It contains the following properties:

- *Id*: The id of the node (which is unique within one node container).
- *Name*: The display name of the node.
- *Type*: The type of the Join node, i.e. AND, OR or XOR.

6.6. Others

6.6.1. Variables

While the flow chart focuses on specifying the control flow of the process, it is usually also necessary to look at the process from a data perspective. Throughout the execution of a process, data can be retrieved, stored, passed on and used.

For storing runtime data, during the execution of the process, process variables can be used. A variable is defined by a name and a data type. This could be a basic data type, such as boolean, int, or String, or any kind of Object subclass (it must implement Serializable interface). Variables can be defined inside a variable *scope*. The top-level scope is the variable scope of the process itself. Subscopes can be defined using a Sub-Process. Variables that are defined in a subscope are only accessible for nodes within that scope.

Whenever a variable is accessed, the process will search for the appropriate variable scope that defines the variable. Nesting of variable scopes is allowed. A node will always search for a variable in its parent container. If the variable cannot be found, it will look in that one's parent container, and so on, until the process instance itself is reached. If the variable cannot be found, a read access yields null, and a write access produces an error message, with the process continuing its execution.

Variables can be used in various ways:

- Process-level variables can be set when starting a process by providing a map of parameters to the invocation of the `startProcess` method. These parameters will be set as variables on the process scope.
- Script actions can access variables directly, simply by using the name of the variable as a local parameter in their script. For example, if the process defines a variable of type "org.jbpm.Person" in the process, a script in the process could access this directly:

```
// call method on the process variable "person"
person.setAge(10);
```

Changing the value of a variable in a script can be done through the knowledge context:

```
kcontext.setVariable(variableName, value);
```

- Service tasks (and reusable sub-processes) can pass the value of process variables to the outside world (or another process instance) by mapping the variable to an outgoing parameter. For example, the parameter mapping of a service task could define that the value of the process variable x should be mapped to a task parameter y right before the service is being invoked. You can also inject the value of process variable into a hard-coded parameter String using `#{expression}`. For example, the description of a human task could be defined as `You need to contact person #{person.getName()}` (where person is a process variable), which will replace this expression by the actual name of the person when the service needs to be invoked. Similarly results of a service (or reusable sub-process) can also be copied back to a variable using a result mapping.
- Various other nodes can also access data. Event nodes for example can store the data associated to the event in a variable, etc. Check the properties of the different node types for more information.
- Process variables can be accessed also from the Java code of your application. It is done by casting of `ProcessInstance` to `WorkflowProcessInstance`. See the following example:

```
variable = ((WorkflowProcessInstance) processInstance).getVariable("variableName");
```

To list all the process variables see the following code snippet:

```
```

```
org.jbpm.process.instance.ProcessInstance processInstance = ...;
VariableScopeInstance variableScope = (VariableScopeInstance) processInstance.getContextInsta
Map<String, Object> variables = variableScope.getVariables();
```

Note that when you use persistence then you have to use a command based approach to get all process variables:

```
Map<String, Object> variables = ksession.execute(new GenericCommand<Map<String, Object>>() {
    public Map<String, Object> execute(Context context) {
        KieSession ksession = ((KnowledgeCommandContext) context).getStatefulKnowledgesession
        org.jbpm.process.instance.ProcessInstance processInstance = (org.jbpm.process.instanc
        VariableScopeInstance variableScope = (VariableScopeInstance) processInstance.getCont
        Map<String, Object> variables = variableScope.getVariables();
        return variables;
    }
});
```

Finally, processes (and rules) all have access to globals, i.e. globally defined variables and data in the Knowledge Session. Globals are directly accessible in actions just like variables. Globals need to be defined as part of the process before they can be used. You can for example define globals by clicking the globals button when specifying an action script in the Eclipse action property editor. You can also set the value of a global from the outside using `ksession.setGlobal(name, value)` or from inside process scripts using `kcontext.getKieRuntime().setGlobal(name,value);`.

6.6.2. Scripts

Action scripts can be used in different ways:

- Within a Script Task,
- As entry or exit actions, with a number of nodes.

Actions have access to globals and the variables that are defined for the process and the predefined variable `kcontext`. This variable is of type *`ProcessContext`* [http://docs.jboss.org/jbpm/v6.0.1/javadocs/org/kie/api/runtime/process/ProcessContext.html] and can be used for several tasks:

- Getting the current node instance (if applicable). The node instance could be queried for data, such as its name and type. You can also cancel the current node instance.

```
NodeInstance node = kcontext.getNodeInstance();
```

```
String name = node.getNodeName();
```

- Getting the current process instance. A process instance can be queried for data (name, id, processId, etc.), aborted or signaled an internal event.

```
ProcessInstance proc = kcontext.getProcessInstance();
proc.signalEvent( type, eventObject );
```

- Getting or setting the value of variables.
- Accessing the Knowledge Runtime allows you do things like starting a process, signaling (external) events, inserting data, etc.

jBPM currently supports two dialects, Java and MVEL. Java actions should be valid Java code. MVEL actions can use the business scripting language MVEL to express the action. MVEL accepts any valid Java code but additionally provides support for nested accesses of parameters (e.g., `person.name` instead of `person.getName()`), and many other scripting improvements. Thus, MVEL expressions are more convenient for the business user. For example, an action that prints out the name of the person in the "requester" variable of the process would look like this:

```
// Java dialect
System.out.println( person.getName() );

//  MVEL dialect
System.out.println( person.name );
```

6.6.3. Constraints

Constraints can be used in various locations in your processes, for example in a diverging gateway. jBPM supports two types of constraints:

- *Code constraints* are boolean expressions, evaluated directly whenever they are reached. We currently support two dialects for expressing these code constraints: Java and MVEL. Both Java and MVEL code constraints have direct access to the globals and variables defined in the process. Here is an example of a valid Java code constraint, `person` being a variable in the process:

```
return person.getAge() > 20;
```

  A similar example of a valid MVEL code constraint is:

```
return person.age > 20;
```

- *Rule constraints* are equals to normal Drools rule conditions. They use the Drools Rule Language syntax to express possibly complex constraints. These rules can, like any other rule, refer to data in the Working Memory. They can also refer to globals directly. Here is an example of a valid rule constraint:

```
Person( age > 20 )
```

  This tests for a person older than 20 being in the Working Memory.

Rule constraints do not have direct access to variables defined inside the process. It is however possible to refer to the current process instance inside a rule constraint, by adding the process instance to the Working Memory and matching for the process instance in your rule constraint. We have added special logic to make sure that a variable `processInstance` of type `WorkflowProcessInstance` will only match to the current process instance and not to other process instances in the Working Memory. Note that you are however responsible yourself to insert the process instance into the session and, possibly, to update it, for example, using Java code or an on-entry or on-exit or explicit action in your process. The following example of a rule constraint will search for a person with the same name as the value stored in the variable "name" of the process:

```
processInstance : WorkflowProcessInstance()
Person( name == ( processInstance.getVariable("name") ) )
# add more constraints here ...
```

6.6.4. Timers

Timers wait for a predefined amount of time, before triggering, once or repeatedly. They can be used to trigger certain logic after a certain period, or to repeat some action at regular intervals.

6.6.4.1. Configure timer with delay and period

A Timer node is set up with a delay and a period. The delay specifies the amount of time to wait after node activation before triggering the timer the first time. The period defines the time between subsequent trigger activations. A period of 0 results in a one-shot timer.

The (period and delay) expression should be of the form [#d][#h][#m][#s][#[ms]]. You can specify the amount of days, hours, minutes, seconds and milliseconds (which is the default if you don't specify anything). For example, the expression "1h" will wait one hour before triggering the timer (again).

6.6.4.2. Configure timer ISO-8601 date format

since version 6 timers can be configured with valid *ISO8601* [http://en.wikipedia.org/wiki/ISO_8601] date format that supports both one shot timers and repeatable timers. Timers can be defined as date and time representation, time duration or repeating intervals

- Date - 2013-12-24T20:00:00.000+02:00 - fires exactly at Christmas Eve at 8PM
- Duration - PT1S - fires once after 1 second
- Repeatable intervals - R/PT1S - fires every second, no limit, alternatively R5/PT1S will fire 5 times every second

6.6.4.3. Configure timer with process variables

In addition to two configuration options above timers can be specified using process variable that can consists of string representation of ether delay and period or ISO8601 date format. By specifying #{variable} engine will dynamically extract process variable and use it as timer expression.

The timer service is responsible for making sure that timers get triggered at the appropriate times. Timers can also be canceled, meaning that the timer will no longer be triggered.

Timers can be used in two ways inside a process:

- A Timer Event may be added to the process flow. Its activation starts the timer, and when it triggers, once or repeatedly, it activates the Timer node's successor. Subsequently, the outgoing connection of a timer with a positive period is triggered multiple times. Canceling a Timer node also cancels the associated timer, after which no more triggers will occur.
- Timers can be associated with a Sub-Process or tasks as a boundary event.

6.7. Process Fluent API

While it is recommended to define processes using the graphical editor or the underlying XML (to shield yourself from internal APIs), it is also possible to define a process using the Process API directly. The most important process model elements are defined in the packages `org.jbpm.workflow.core` and `org.jbpm.workflow.core.node`. A "fluent API" is provided that allows you to easily construct processes in a readable manner using factories. At the end, you can validate the process that you were constructing manually.

6.7.1. Example

This is a simple example of a basic process with a script task only:

```
RuleFlowProcessFactory factory =
    RuleFlowProcessFactory.createProcess("org.jbpm.HelloWorld");
factory
    // Header
    .name("HelloWorldProcess")
    .version("1.0")
    .packageName("org.jbpm")
    // Nodes
    .startNode(1).name("Start").done()
    .actionNode(2).name("Action")
        .action("java", "System.out.println(\"Hello World\");").done()
    .endNode(3).name("End").done()
    // Connections
    .connection(1, 2)
    .connection(2, 3);
RuleFlowProcess process = factory.validate().getProcess();
KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();
kbuilder.add(ResourceFactory.newByteArrayResource(
    XmlBPMNProcessDumper.INSTANCE.dump(process).getBytes()), ResourceType.BPMN2);
KnowledgeBase kbase = kbuilder.newKnowledgeBase();
StatefulKnowledgeSession ksession = kbase.newStatefulKnowledgeSession();
ksession.startProcess("org.jbpm.HelloWorld");
```

You can see that we start by calling the static `createProcess()` method from the `RuleFlowProcessFactory` class. This method creates a new process with the given id and returns the `RuleFlowProcessFactory` that can be used to create the process. A typical process consists of three parts. The header part comprises global elements like the name of the process, imports, variables, etc. The nodes section contains all the different nodes that are part of the process. The connections section finally links these nodes to each other to create a flow chart.

In this example, the header contains the name and the version of the process and the package name. After that, you can start adding nodes to the current process. If you have auto-completion you can see that you have different methods to create each of the supported node types at your disposal.

When you start adding nodes to the process, in this example by calling the `startNode()`, `actionNode()` and `endNode()` methods, you can see that these methods return a specific `NodeFactory`, that allows you to set the properties of that node. Once you have finished configuring that specific node, the `done()` method returns you to the current `RuleFlowProcessFactory` so you can add more nodes, if necessary.

When you are finished adding nodes, you must connect them by creating connections between them. This can be done by calling the method `connection`, which will link previously created nodes.

Finally, you can validate the generated process by calling the `validate()` method and retrieve the created `RuleFlowProcess` object.

6.8. Testing

Even though business processes aren't code (we even recommend you to make them as high-level as possible and to avoid adding implementation details), they also have a life cycle like other development artefacts. And since business processes can be updated dynamically, testing them (so that you don't break any use cases when doing a modification) is really important as well.

6.8.1. Unit testing

When unit testing your process, you test whether the process behaves as expected in specific use cases, for example test the output based on the existing input. To simplify unit testing, jBPM includes a helper class called **JbpmJUnitBaseTestCase** (in the jbpm-test module) that you can use to greatly simplify your JUnit testing, by offering:

- helper methods to create a new RuntimeManager and RuntimeEngine for a given (set of) process(es)
  - you can select whether you want to use persistence or not
- assert statements to check
  - the state of a process instance (active, completed, aborted)
  - which node instances are currently active
  - which nodes have been triggered (to check the path that has been followed)
  - get the value of variables

For example, consider the following "hello world" process containing a start event, a script task and an end event. The following JUnit test will create a new session, start the process and then verify whether the process instance completed successfully and whether these three nodes have been executed.

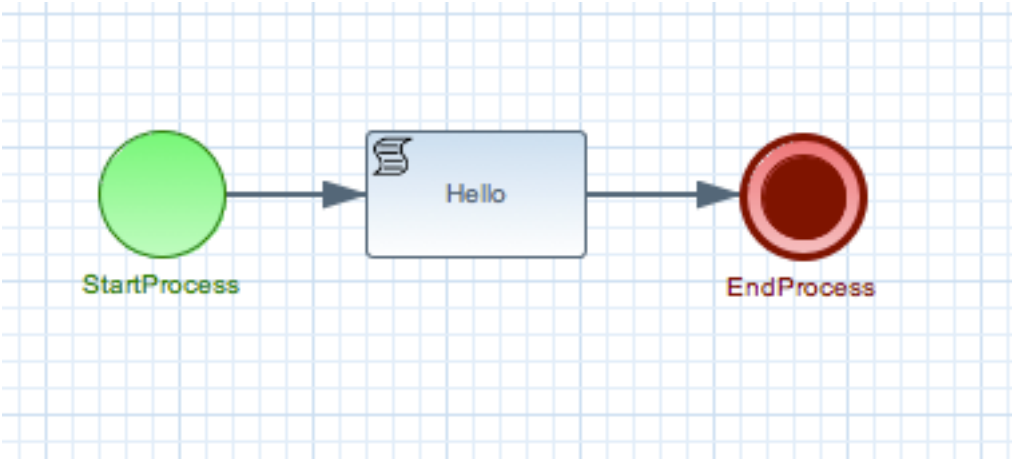


Figure 6.21.

```
public class ProcessPersistenceTest extends JbpmJUnitBaseTestCase {

    public ProcessPersistenceTest() {
        // setup data source, enable persistence
        super(true, true);
    }

    @Test
    public void testProcess() {
        // create runtime manager with single process - hello.bpmn
        createRuntimeManager("hello.bpmn");

        // take RuntimeManager to work with process engine
        RuntimeEngine runtimeEngine = getRuntimeEngine();

        // get access to KieSession instance
        KieSession ksession = runtimeEngine.getKieSession();

        // start process
        ProcessInstance processInstance = ksession.startProcess("com.sample.bpmn.hello");

        // check whether the process instance has completed successfully
        assertProcessInstanceCompleted(processInstance.getId(), ksession);

        // check what nodes have been triggered
        assertNodeTriggered(processInstance.getId(), "StartProcess", "Hello", "EndProcess");
    }
}
```

JbpmJUnitBaseTestCase acts as base test case class that shall be used for jBPM related tests. It provides four usage areas:

- JUnit life cycle methods
  - setUp: executed @Before and configures data source and EntityManagerFactory, cleans up Singleton's session id
  - tearDown: executed @After and clears out history, closes EntityManagerFactory and data source, disposes RuntimeEngines and RuntimeManager
- Knowledge Base and KnowledgeSession management methods
  - createRuntimeManager creates RuntimeManager for given set of assets and selected strategy
  - disposeRuntimeManager disposes RuntimeManager currently active in the scope of test
  - getRuntimeEngine creates new RuntimeEngine for given context

- Assertions
  - assertProcessInstanceCompleted
  - assertProcessInstanceAborted
  - assertProcessInstanceActive
  - assertNodeActive
  - assertNodeTriggered
  - assertProcessVarExists
  - assertNodeExists
  - assertVersionEquals
  - assertProcessNameEquals
- Helper methods
  - getDs - returns currently configured data source
  - getEmf - returns currently configured EntityManagerFactory
  - getTestWorkItemHandler - returns test work item handler that might be registered in addition to what is registered by default
  - clearHistory - clears history log
  - setupPoolingDataSource - sets up data source

JbpmJUnitBaseTestCase supports all three predefined RuntimeManager strategies as part of the unit testing. It's enough to specify which strategy shall be used whenever creating runtime manager as part of single test:

```
public class ProcessHumanTaskTest extends JbpmJUnitBaseTestCase {

    private static final Logger logger = LoggerFactory.getLogger(ProcessHumanTaskTest.class);

    public ProcessHumanTaskTest() {
        super(true, false);
    }

    @Test
    public void testProcessProcessInstanceStrategy() {
        RuntimeManager manager = createRuntimeManager(Strategy.PROCESS_INSTANCE, "manager", "hu
        RuntimeEngine runtimeEngine = getRuntimeEngine(ProcessInstanceIdContext.get());
        KieSession ksession = runtimeEngine.getKieSession();
        TaskService taskService = runtimeEngine.getTaskService();
```

```
        int ksessionID = ksession.getId();
        ProcessInstance processInstance = ksession.startProcess("com.sample.bpmn.hello");

        assertProcessInstanceActive(processInstance.getId(), ksession);
        assertNodeTriggered(processInstance.getId(), "Start", "Task 1");

        manager.disposeRuntimeEngine(runtimeEngine);
        runtimeEngine = getRuntimeEngine(ProcessInstanceIdContext.get(processInstance.getId()))

        ksession = runtimeEngine.getKieSession();
        taskService = runtimeEngine.getTaskService();

        assertEquals(ksessionID, ksession.getId());

        // let john execute Task 1
 List<TaskSummary>list=taskService.getTasksAssignedAsPotentialOwner("john","en-
UK");
        TaskSummary task = list.get(0);
        logger.info("John is executing task {}", task.getName());
        taskService.start(task.getId(), "john");
        taskService.complete(task.getId(), "john", null);

        assertNodeTriggered(processInstance.getId(), "Task 2");

        // let mary execute Task 2
        list = taskService.getTasksAssignedAsPotentialOwner("mary", "en-UK");
        task = list.get(0);
        logger.info("Mary is executing task {}", task.getName());
        taskService.start(task.getId(), "mary");
        taskService.complete(task.getId(), "mary", null);

        assertNodeTriggered(processInstance.getId(), "End");
        assertProcessInstanceCompleted(processInstance.getId(), ksession);
    }
}
```

Above is more complete example that uses PerProcessInstance runtime manager strategy and uses task service to deal with user tasks.

6.8.1.1. Testing integration with external services

Real-life business processes typically include the invocation of external services (like for example a human task service, an email server or your own domain-specific services). One of the advantages of our domain-specific process approach is that you can specify yourself how to actually execute your own domain-specific nodes, by registering a handler. And this handler can be different depending on your context, allowing you to use testing handlers for unit testing your process. When you are unit testing your business process, you can register test handlers that

then verify whether specific services are requested correctly, and provide test responses for those services. For example, imagine you have an email node or a human task as part of your process. When unit testing, you don't want to send out an actual email but rather test whether the email that is requested contains the correct information (for example the right to email, a personalized body, etc.).

A TestWorkItemHandler is provided by default that can be registered to collect all work items (a work item represents one unit of work, like for example sending one specific email or invoking one specific service and contains all the data related to that task) for a given type. This test handler can then be queried during unit testing to check whether specific work was actually requested during the execution of the process and that the data associated with the work was correct.

The following example describes how a process that sends out an email could be tested. This test case in particular will test whether an exception is raised when the email could not be sent (which is simulated by notifying the engine that the sending the email could not be completed). The test case uses a test handler that simply registers when an email was requested (and allows you to test the data related to the email like from, to, etc.). Once the engine has been notified the email could not be sent (using abortWorkItem(..)), the unit test verifies that the process handles this case successfully by logging this and generating an error, which aborts the process instance in this case.

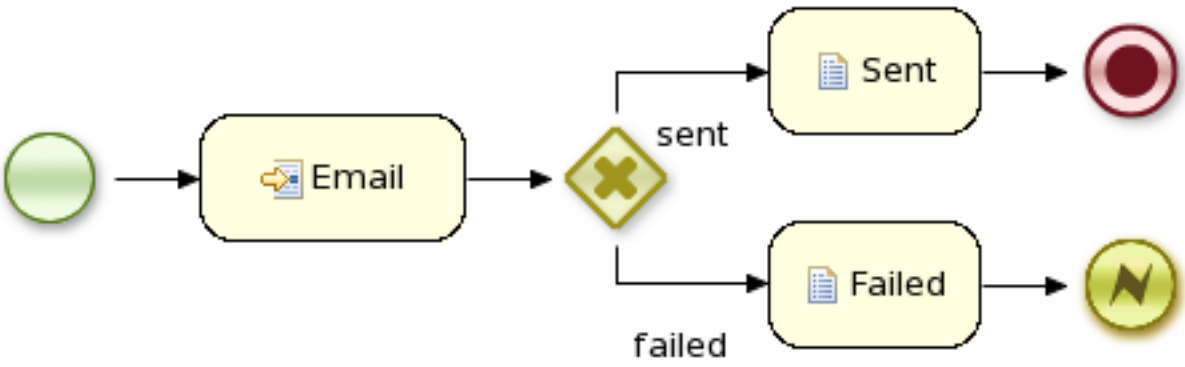


Figure 6.22.

```
public void testProcess2() {

    // create runtime manager with single process - hello.bpmn
    createRuntimeManager("sample-process.bpmn");
    // take RuntimeManager to work with process engine
    RuntimeEngine runtimeEngine = getRuntimeEngine();

    // get access to KieSession instance
    KieSession ksession = runtimeEngine.getKieSession();

    // register a test handler for "Email"
    TestWorkItemHandler testHandler = getTestWorkItemHandler();
```

```
   ksession.getWorkItemManager().registerWorkItemHandler("Email", testHandler);

    // start the process
    ProcessInstance processInstance = ksession.startProcess("com.sample.bpmn.hello2");

    assertProcessInstanceActive(processInstance.getId(), ksession);
    assertNodeTriggered(processInstance.getId(), "StartProcess", "Email");

    // check whether the email has been requested
    WorkItem workItem = testHandler.getWorkItem();
    assertNotNull(workItem);
    assertEquals("Email", workItem.getName());
    assertEquals("me@mail.com", workItem.getParameter("From"));
    assertEquals("you@mail.com", workItem.getParameter("To"));

    // notify the engine the email has been sent
    ksession.getWorkItemManager().abortWorkItem(workItem.getId());
    assertProcessInstanceAborted(processInstance.getId(), ksession);
    assertNodeTriggered(processInstance.getId(), "Gateway", "Failed", "Error");

}
```

6.8.1.2. Configuring persistence

You can configure whether you want to execute the JUnit tests using persistence or not. By default, the JUnit tests will use persistence, meaning that the state of all process instances will be stored in a (in-memory H2) database (which is started by the JUnit test during setup) and a history log will be used to check assertions related to execution history. When persistence is not used, process instances will only live in memory and an in-memory logger is used for history assertions.

Persistence (and setup of data source) is controlled by the super constructor and allows following

- default, no arg constructor - the most simple test case configuration (does NOT initialize data source and does NOT configure session persistence) - this is usually used for in memory process management, without human task interaction
- super(boolean, boolean) - allows to explicitly configure persistence and data source. This is the most common way of bootstrapping test cases for jBPM
  - super(true, false) - to execute with in memory process management with human tasks persistence
  - super(true, true) - to execute with persistent process management with human tasks persistence
- super(boolean, boolean, string) - same as super(boolean, boolean) but allows to use another persistence unit name than default (org.jbpm.persistence.jpa)

```
public class ProcessHumanTaskTest extends JbpmJUnitBaseTestCase {

    private static final Logger logger = LoggerFactory.getLogger(ProcessHumanTaskTest.class);

    public ProcessHumanTaskTest() {
         // configure this tests to not use persistence for process engine but
 still use it for human tasks
        super(true, false);
    }
}
```

Chapter 7. Human Tasks

7.1. Introduction

An important aspect of business processes is human task management. While some of the work performed in a process can be executed automatically, some tasks need to be executed by human actors.

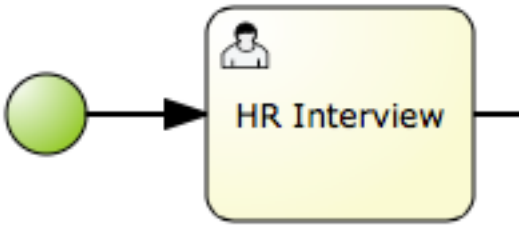
jBPM supports a special human task node inside processes for modeling this interaction with human users. This human task node allows process designers to define the properties related to the task that the human actor needs to execute, like for example the type of task, the actor(s), or the data associated with the task.

jBPM also includes a so-called human task service, a back-end service that manages the life cycle of these tasks at runtime. The jBPM implementation is based on the WS-HumanTask specification. Note however that this implementation is fully pluggable, meaning that users can integrate their own human task solution if necessary.

In order to have human actors participate in your processes, you first need to (1) include human task nodes inside your process to model the interaction with human actors, (2) integrate a task management component (like for example the WS-HumanTask based implementation provided by jBPM) and (3) have end users interact with a human task client to request their task list and claim and complete the tasks assigned to them. Each of these three elements will be discussed in more detail in the next sections.

7.2. Using User Tasks in our Processes

jBPM supports the use of human tasks inside processes using a special User Task node defined by the BPMN2 Specification(as shown in the figure above). A User Task node represents an atomic task that needs to be executed by a human actor.

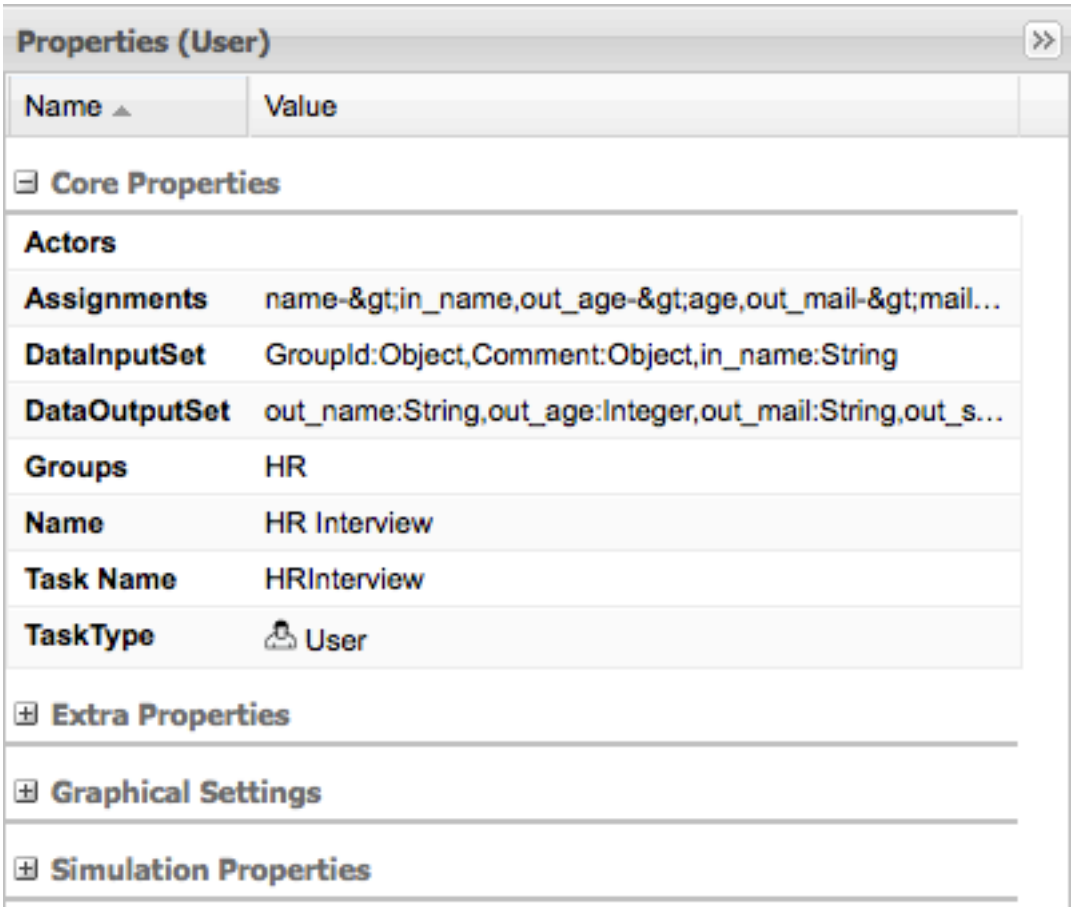


[Although jBPM has a special user task node for including human tasks inside a process, human tasks are considered the same as any other kind of external service that needs to be invoked and are therefore simply implemented as a domain-specific service. See the chapter on domain-specific processes to learn more about this.]

A User Task node contains the following core properties:

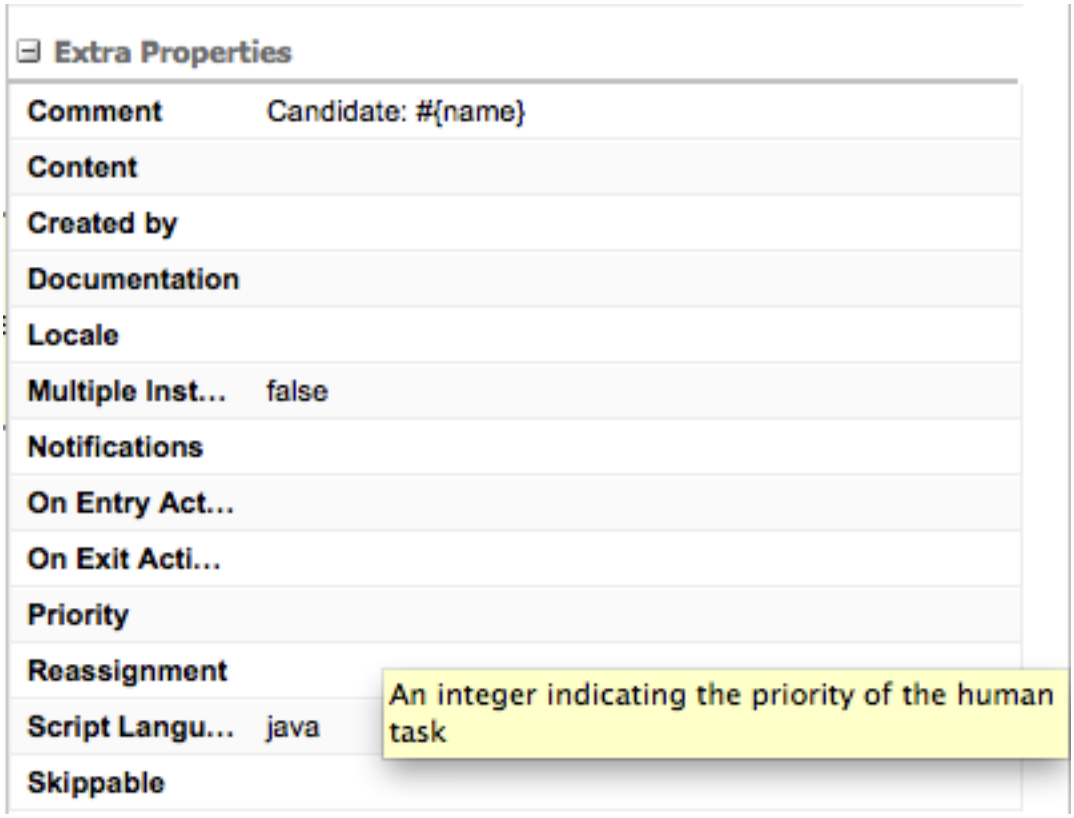
- Actors: The actors that are responsible for executing the human task. A list of actor id's can be specified using a comma (',') as separator.
- Group: The group id that is responsible for executing the human task. A list of group id's can be specified using a comma (',') as separator.
- Name: The display name of the node.
- TaskName: The name of the human task. This name is used to link the task to a Form. It also represent the internal name of the Task that can be used for other purposes.
- DataInputSet: all the input variables that the task will receive to work on. Usually you will be interested in copying variables from the scope of the process to the scope of the task. (Look at the data mappings section for an example)
- DataOutputSet: all the output variables that will be generated by the execution of the task. Here you specify all the name of the variables in the context of the task that you are interested to copy to the context of the process. (Look at the data mappings section for an example)
- Assignments: here you specify which process variable will be linked to each Data Input and Data Output mapping. (Look at the data mappings section for an example)

You can edit these variables in the properties view (see below) when selecting the User Task node.



A User Task node also contains the following extra properties:

- Comment: A comment associated with the human task. Here you can use expressions.
- Content: The data associated with this task.
- Priority: An integer indicating the priority of the human task.
- Skippable: Specifies whether the human task can be skipped, i.e., whether the actor may decide not to execute the task.
- On entry and on exit actions: Action scripts that are executed upon entry and exit of this node, respectively.



7.3. Data Mappings

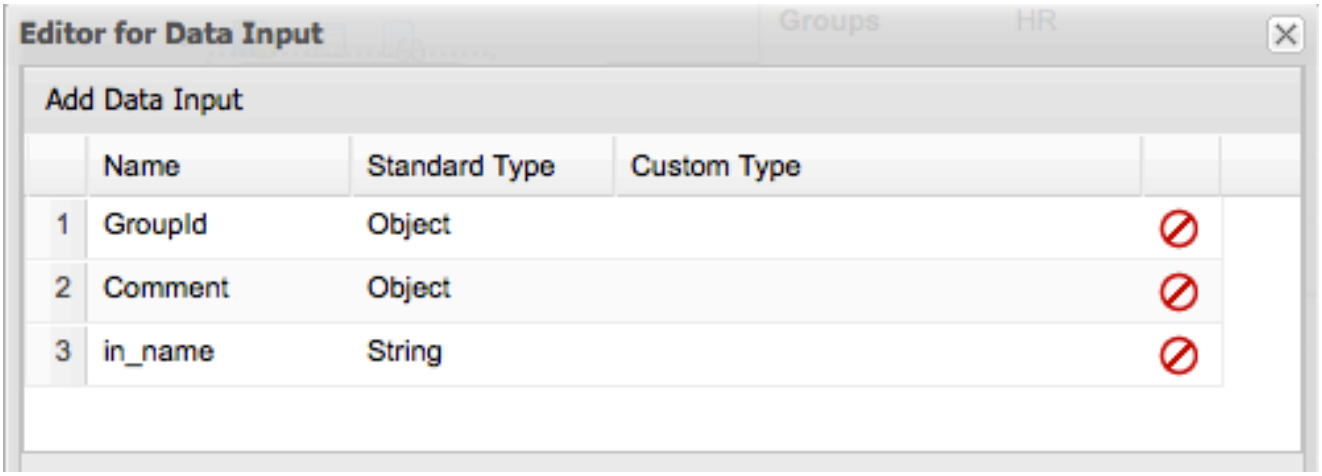
Human tasks typically present some data related to the task that needs to be performed to the actor that is executing the task and usually also request the actor to provide some result data related to the execution of the task. Task forms are typically used to present this data to the actor and request results.

The data that will be used by the Task needs to be specified when we define the User Task in our Process. In order to do that we need to define which data will be copied from the process context to the task context. Notice that the data is copied, so it can be modified inside the Task context but it will not affect the process variables unless we decide to copy back the value from the task to the process context.

Most of the times Forms are used to display data to the end user. Allowing them to generate/create new data that will be propagated to the process context to be used by future activities. In order

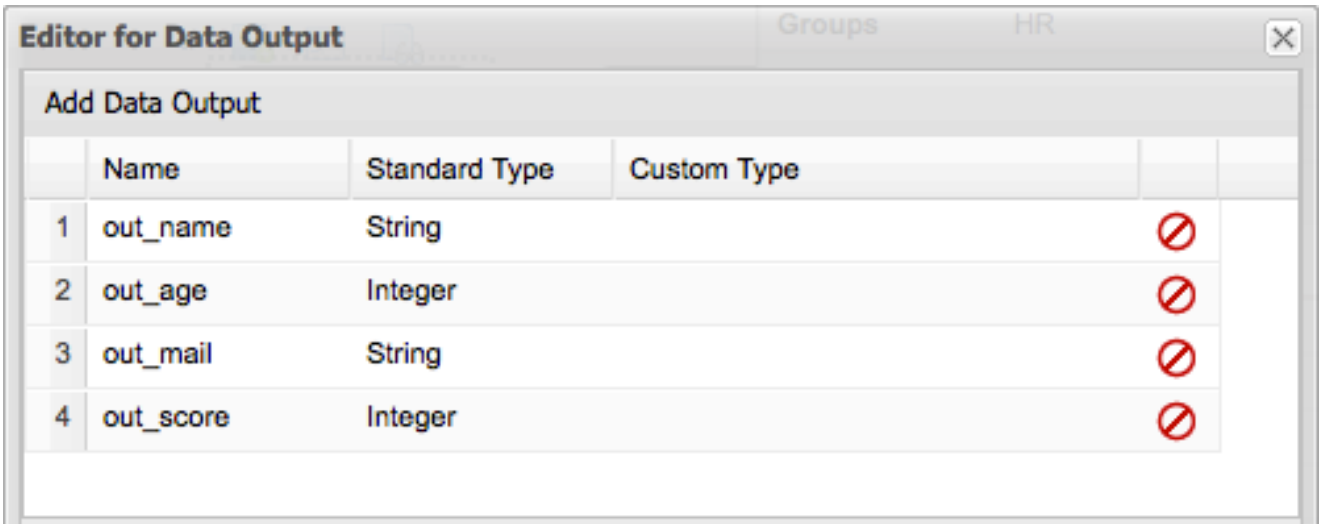
to decide how the information flow from the process to a particular task and from the task to the process we need to define which pieces of information will be automatically copied by the process engine. The following sections shows how to do these mappings by configuring the DataInputSet, DataOutputSet and the Assignments properties of a User Task.

Let's start defining the Task **DataInputSet**:

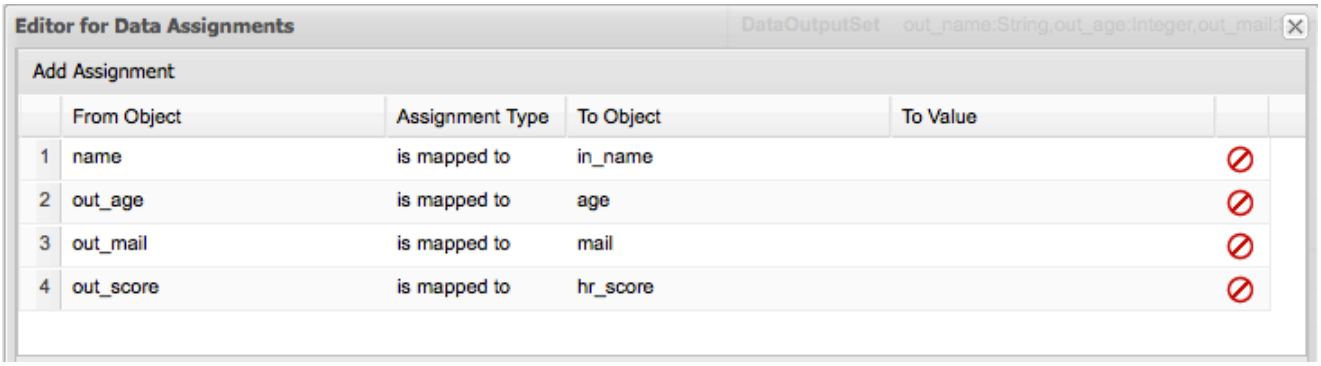


Both GroupId and Comment are automatically generated, so you don't need to worry about that. In this case the only user defined Data Input is called: **in_name**. This means that the task will be receiving information from the process context and internally this variable will be called in_name. The type is also specified here.

In the **Data Outputs** represent the data that will be generated by the tasks. In this case we have two variables of type String called: **out_name** and **out_mail** and two Integer variables called: **out_age** and **out_score** are defined. This means that inside the task context we will need to set the value to these variables.



Finally all the connections with the process context needs to be done in the **Data Assignments**. The main idea here is to define how Data Inputs and Data Outputs will be associated with process variables.



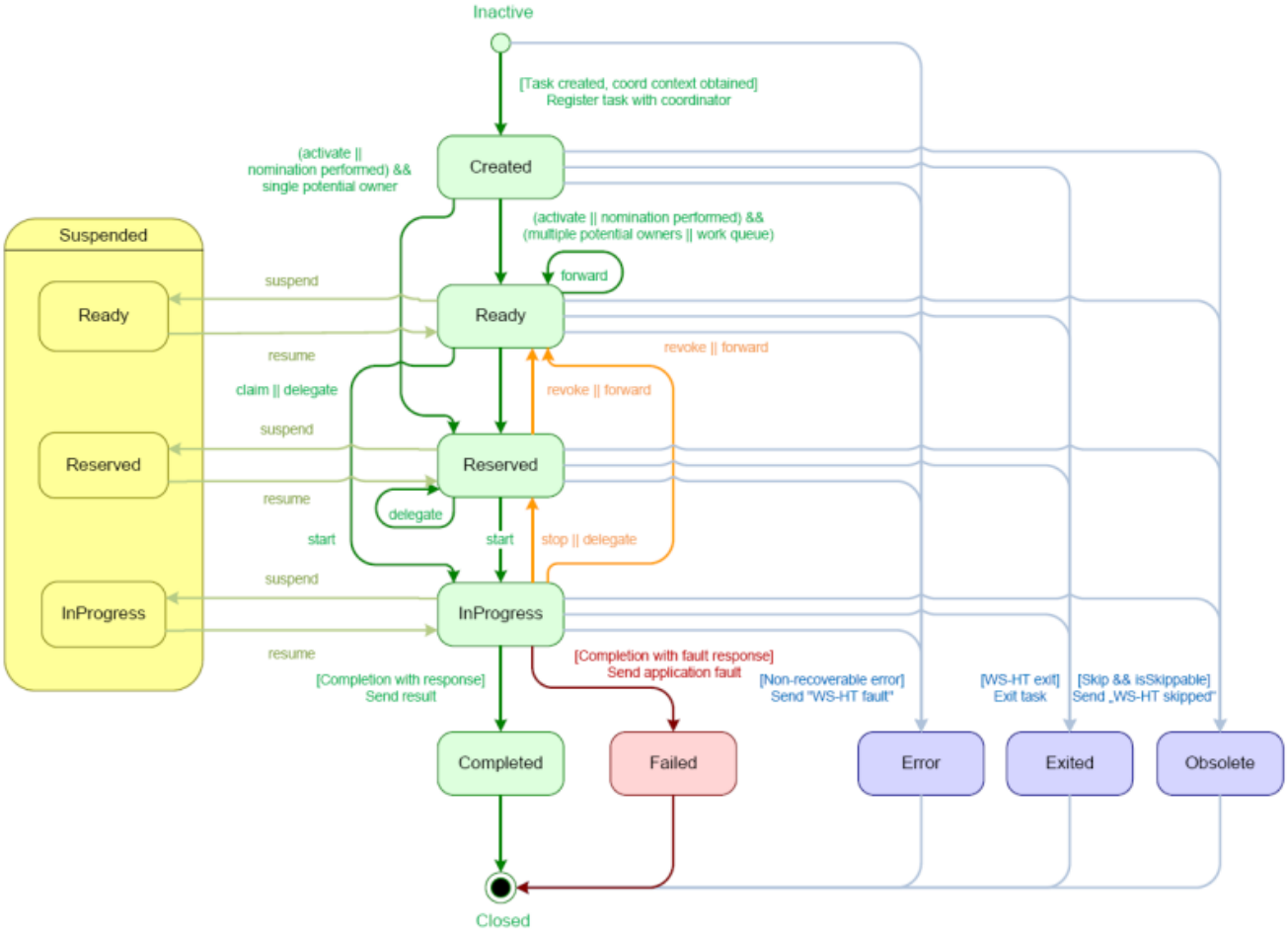
As shown in the previous screenshot, the assignments between the process variables (in this case (**name, age, mail and hr_score**)) and the Data Inputs and Outputs are done in the Data Assignments screen. Notice that the example uses a convention that makes it easy to know which is an internal Task variables (Data Input/Output) using the "**in**_" and "**out**_" prefix to the variable names. Using this convention you can quickly understand the Assignments screen. The first row maps the process variable called **name** to the data input called **in_name**. The second row maps the data output called **out_mail** to the process variable called **mail**, and so on.

These mappings at runtime will automatically copy the variables content from one context (process and task) to the other automatically for us.

7.4. Task Lifecycle

From the perspective of a process, when a user task node is encountered during the execution, a human task is created. The process will then only leave the user task node when the associated human task has been completed or aborted.

The human task itself usually has a complete life cycle itself as well. For details beyond what is described below, please check out the WS-HumanTask specification. The following diagram is from the WS-HumanTask specification and describes the human task life cycle.



A newly created task starts in the "Created" stage. Usually, it will then automatically become "Ready", after which the task will show up on the task list of all the actors that are allowed to execute the task. The task will stay "Ready" until one of these actors claims the task, indicating that he or she will be executing it.

When a user then eventually claims the task, the status will change to "Reserved". Note that a task that only has one potential (specific) actor will automatically be assigned to that actor upon creation of the task. When the user who has claimed the task starts executing it, the task status will change from "Reserved" to "InProgress".

Lastly, once the user has performed and completed the task, the task status will change to "Completed". In this step, the user can optionally specify the result data related to the task. If the task could not be completed, the user could also indicate this by using a fault response, possibly including fault data, in which case the status would change to "Failed".

While the life cycle explained above is the normal life cycle, the specification also describes a number of other life cycle methods, including:

- Delegating or forwarding a task, so that the task is assigned to another actor
- Revoking a task, so that it is no longer claimed by one specific actor but is (re)available to all actors allowed to take it

- Temporarly suspending and resuming a task
- Stopping a task in progress
- Skipping a task (if the task has been marked as skippable), in which case the task will not be executed

7.5. Task Permissions

Only users associated with a specific task are allowed to modify or retrieve information about the task. This allows users to create a jBPM workflow with multiple tasks and yet still be assured of both the confidentiality and integrity of the task status and information associated with a task.

Some task operations will end up throwing a `org.jbpm.services.task.exception.PermissionDeniedException` when used with information about an unauthorized user. For example, when a user is trying to directly modify the task (for example, by trying to claim or complete the task), the `PermissionDeniedException` will be thrown if that user does not have the correct role for that operation. Furthermore, a user will not be able to view or retrieve tasks that the user is not involved with, especially if this is via the jBPM Console or Kie Workbench applications.

7.5.1. Task Permissions Matrix

The permisions matrix below summarizes the actions that specific user roles are allowed to do. On the left side, possible operations are listed while user roles are listed across the top of the matrix.

The cells of the permissions matrix contain one of three possible characters, each of which indicate the user role permissions for that operation:

- a "+ indicates that the user role CAN do the specified operation
- a "-" indicates that the user role MAY NOT do the specified operation
- a "_" indicates that the user role MAY NOT do the specified operation, and that it is also not an operation that matches the user's role ("not applicable")

Furthermore, the following words or abbreviations in the table header refer to the following roles:

Table 7.1. Task roles in the permissions table

Word	Role	Description
Initiator	Task Initiator	The user who creates the task instance
Stakeholder	Task Stakeholder	The user involved in the task: this user can influence the progress of a task, by performing administrative actions on the task instance

Word	Role	Description
Potential	Potential Owner	The user who can claim the task before it has been claimed, or after it has been released or forward: only tasks that have the status "Ready" may be claimed; a potential owner becomes the actual owner of a task by claiming the task
Actual	Actual Owner	The user who has claimed the task and will progress the task to completion or failure
Administrator	Business Adminstrator	A "super user" who may modify the status or progress of a task at any point in a task's lifecycle

User roles are assigned to users by the definition of the task in the jBPM (BPMN2) process definition.

Permissions Matrices. The following matrix describes the authorizations for all operations which modify a task:

Table 7.2. Main operations permissions matrix

Operation \Role	Initiator	Stakeholder	Potential	Actual	Administrator
activate	+	+	_	_	+
claim	-	+	+	_	+
complete	-	+	_	+	+
delegate	+	+	+	+	+
fail	-	+	_	+	+
forward	+	+	+	+	+
nominate	+	+	+	+	+
release	+	+	+	+	+
remove	-	_	_	_	+
resume	+	+	+	+	+
skip	+	+	+	+	+
start	-	+	+	+	+
stop	-	+	_	+	+

Operation \Role	Initiator	Stakeholder	Potential	Actual	Administrator
suspend	+	+	+	+	+

The matrix below describes the authorizations used when retrieving task information. In short, it says that all users which have any role with regards to the specific task, are allowed to see the task. This applies to all operations that are used to retrieve any type of information about the task.

Table 7.3. Retrieval operations permissions matrix

Operation \Role	Initiator	Stakeholder	Potential	Actual	Administrator
get	+	+	+	+	+

7.6. Task Service and The Process Engine

As far as the jBPM engine is concerned, human tasks are similar to any other external service that needs to be invoked and are implemented as a domain-specific service. (For more on domain-specific services, see the chapter on them here.) Because a human task is an example of such a domain-specific service, the process itself only contains a high-level, abstract description of the human task to be executed and a work item handler that is responsible for binding this (abstract) task to a specific implementation.

Users can plug in any human task service implementation, such as the one that's provided by jBPM, or they may register their own implementation. In the next paragraphs, we will describe the human task service implementation provided by jBPM.

The jBPM project provides a default implementation of a human task service based on the WS-HumanTask specification. If you do not need to integrate jBPM with another existing implementation of a human task service, you can use this service. The jBPM implementation manages the life cycle of the tasks (creation, claiming, completion, etc.) and stores the state of all the tasks, task lists, and other associated information. It also supports features like internationalization, calendar integration, different types of assignments, delegation, escalation and deadlines. The code for the implementation itself can be found in the jbpm-human-task module.

The jBPM task service implementation is based on the WS-HumanTask (WS-HT) specification. This specification defines (in detail) the model of the tasks, the life cycle, and many other features. It is very comprehensive and the first version can be found here.

7.7. Task Service API

The human task service exposes a Java API for managing the life cycle of tasks. This allows clients to integrate (at a low level) with the human task service. Note that end users should probably not interact with this low-level API directly, but use one of the more user-friendly task clients

(see below) instead. These clients offer a graphical user interface to request task lists, claim and complete tasks, and manage tasks in general. The task clients listed below use the Java API to internally interact with the human task service. Of course, the low-level API is also available so that developers can use it in their code to interact with the human task service directly.

A task service (interface org.kie.api.task.TaskService) offers the following methods (among others) for managing the life cycle of human tasks:

```
              ...

              void start( long taskId, String userId );

              void stop( long taskId, String userId );

              void release( long taskId, String userId );

              void suspend( long taskId, String userId );

              void resume( long taskId, String userId );

              void skip( long taskId, String userId );

              void delegate(long taskId, String userId, String targetUserId);

                void complete( long taskId, String userId, Map<String, Object>
 results );

              ...
```

If you take a look at the method signatures you will notice that almost all of these methods take the following arguments:

- taskId: The id of the task that we are working with. This is usually extracted from the currently selected task in the user task list in the user interface.
- userId: The id of the user that is executing the action. This is usually the id of the user that is logged in into the application.

There is also an internal interface that you should check for more methods to interact with the Task Service, this interface is internal until it gets tested. Future version of the External (public) interface can include some of the methods proposed in the InternalTaskService interface. If you want to make use of the methods provided by this interface you need to manually cast to InternalTaskService. One method that can be useful from this interface is getTaskContent():

```
Map<String, Object> getTaskContent( long taskId );
```

This method saves you from doing all the boiler plate of getting the ContentMarshallerContext to unmarshall the serialized version of the task content. If you only want to use the stable/public API's you can just copy what this method does:

```
            Task taskById = taskQueryService.getTaskInstanceById(taskId);
                                                Content    contentById    =
taskContentService.getContentById(taskById.getTaskData().getDocumentContentId());
            ContentMarshallerContext context = getMarshallerContext(taskById);
                                               Object    unmarshalledObject    =

context.getEnvironment(), context.getClassloader());
            if (!(unmarshalledObject instanceof Map)) {
               throw new IllegalStateException(" The Task Content Needs to be
a Map in order to use this method and it was: "+unmarshalledObject.getClass());

            }
         Map<String, Object> content = (Map<String, Object>) unmarshalledObject;
            return content;
```

Because the content of the Task can be any Object, the previous method assume that you are storing a Map of objects to work. If you are storing other than a Map you should do the correspondent checks.

7.8. Interacting with the Task Service

In order to get access to the Task Service API it is recommended to let the Runtime Manager to make sure that everything is setup correctly. Look at the Runtime Manager section for more information. From the API perspective you should be doing something like this:

```
            ...
                                               RuntimeEngine    engine    =
runtimeManager.getRuntimeEngine(EmptyContext.get());
            KieSession kieSession = engine.getKieSession();
            // Start a process
          kieSession.startProcess("CustomersRelationship.customers", params);
            // Do Task Operations
            TaskService taskService = engine.getTaskService();
```

```
                            List<TaskSummary>  tasksAssignedAsPotentialOwner  =
taskService.getTasksAssignedAsPotentialOwner("mary", "en-UK");

              // Claim Task
              taskService.claim(taskSummary.getId(), "mary");
              // Start Task
              taskService.start(taskSummary.getId(), "mary");

              ...
```

If you use this approach, there is no need to register the Task Service with the Process Engine. The Runtime Manager will do that for you automatically. If you don't use the Runtime Manager, you will be responsible for setting the LocalHTWorkItemHandler in the session in order to get the Task Service notifying the Process Engine when a task is completed, or the Process Engine notifying that a task has been created.

In jBPM 6.x the Task Service runs locally to the Process and Rule Engine and for that reason multiple light clients can be created for different Process and Rule Engine's instances. All the clients will be sharing the same database (backend storage for the tasks).

Chapter 8. Persistence and Transactions

8.1. Process Instance State

jBPM allows the persistent storage of certain information. This chapter describes these different types of persistence, and how to configure them. An example of the information stored is the process runtime state. Storing the process runtime state is necessary in order to be able to continue execution of a process instance at any point, if something goes wrong. Also, the process definitions themselves, and the history information (logs of current and previous process states already) can also be persisted.

8.1.1. Runtime State

Whenever a process is started, a process instance is created, which represents the execution of the process in that specific context. For example, when executing a process that specifies how to process a sales order, one process instance is created for each sales request. The process instance represents the current execution state in that specific context, and contains all the information related to that process instance. Note that it only contains the (minimal) runtime state that is needed to continue the execution of that process instance at some later time, but it does not include information about the history of that process instance if that information is no longer needed in the process instance.

The runtime state of an executing process can be made persistent, for example, in a database. This allows to restore the state of execution of all running processes in case of unexpected failure, or to temporarily remove running instances from memory and restore them at some later time. jBPM allows you to plug in different persistence strategies. By default, if you do not configure the process engine otherwise, process instances are not made persistent.

If you configure the engine to use persistence, it will automatically store the runtime state into the database. You do not have to trigger persistence yourself, the engine will take care of this when persistence is enabled. Whenever you invoke the engine, it will make sure that any changes are stored at the end of that invocation, at so-called safe points. Whenever something goes wrong and you restore the engine from the database, you also should not reload the process instances and trigger them manually to resume execution, as process instances will automatically resume execution if they are triggered, like for example by a timer expiring, the completion of a task that was requested by that process instance, or a signal being sent to the process instance. The engine will automatically reload process instances on demand.

The runtime persistence data should in general be considered internal, meaning that you probably should not try to access these database tables directly and especially not try to modify these directly (as changing the runtime state of process instances without the engine knowing might have unexpected side-effects). In most cases where information about the current execution state

of process instances is required, the use of a history log is mostly recommended (see below). In some cases, it might still be useful to for example query the internal database tables directly, but you should only do this if you know what you are doing.

8.1.1.1. Binary Persistence

jBPM uses a binary persistence mechanism, otherwise known as marshalling, which converts the state of the process instance into a binary dataset. When you use persistence with jBPM, this mechanism is used to save or retrieve the process instance state from the database. The same mechanism is also applied to the session state and any work item states.

When the process instance state is persisted, two things happen:

- First, the process instance information is transformed into a binary blob. For performance reasons, a custom serialization mechanism is used and not normal Java serialization.
- This blob is then stored, alongside other metadata about this process instance. This metadata includes, among other things, the process instance id, process id, and the process start date.

Apart from the process instance state, the session itself can also store some state, such as the state of timer jobs, or the session data that any business rules would be evaluated over. This session state is stored separately as a binary blob, along with the id of the session and some metadata. You can always restore session state by reloading the session with the given id. The session id can be retrieved using `ksession.getId()`.

Note that the process instance binary datasets are usually relatively small, as they only contain the minimal execution state of the process instance. For a simple process instance, this usually contains one or a few node instances, i.e., any node that is currently executing, and any existing variable values.

As a result of jBPM using marshalling, the data model is both simple and small:

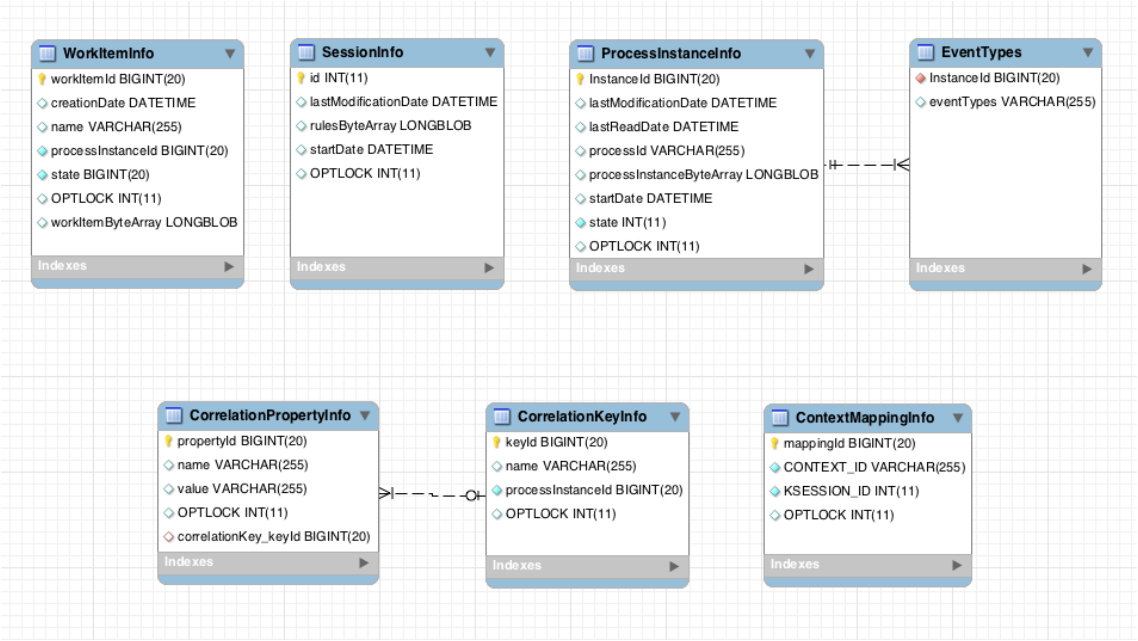


Figure 8.1. jBPM data model

[images/Chapter-Persistence/jbpm_schema.png]

The `sessioninfo` entity contains the state of the (knowledge) session in which the jBPM process instance is running.

Table 8.1. SessionInfo

Field	Description	Nullable
`id`	The primary key.	NOT NULL
`lastmodificationdate`	The last time that the entity was saved to the database	
`rulesbytearray`	The binary dataset containing the state of the session	NOT NULL
`startdate`	The start time of the session	
`optlock`	The version field that serves as its optimistic lock value	

The `processinstanceinfo` entity contains the state of the jBPM process instance.

Table 8.2. ProcessInstanceInfo

Field	Description	Nullable
`instanceid`	The primary key	NOT NULL
`lastmodificationdate`	The last time that the entity was saved to the database	

Field	Description	Nullable
`lastreaddate`	The last time that the entity was retrieved (read) from the database	
`processid`	The name (id) of the process	
`processinstancebytearray`	This is the binary dataset containing the state of the process instance	NOT NULL
`startdate`	The start time of the process	
`state`	An integer representing the state of the process instance	NOT NULL
`optlock`	The version field that serves as its optimistic lock value	

The `eventtypes` entity contains information about events that a process instance will undergo or has undergone.

Table 8.3. EventTypes

Field	Description	Nullable
`instanceid`	This references the `processinstanceinfo` primary key and there is a foreign key constraint on this column.	NOT NULL
`eventTypes`	A text field related to an event that the process has undergone.	

The `workiteminfo` entity contains the state of a work item.

Table 8.4. WorkItemInfo

Field	Description	Nullable
`workitemid`	The primary key	NOT NULL
`creationDate`	The name of the work item	
`name`	The name of the work item	
`processinstanceid`	The (primary key) id of the process: there is no foreign key constraint on this field.	NOT NULL
`state`	An integer representing the state of the work item	NOT NULL

Field	Description	Nullable
optlock	The version field that serves as its optimistic lock value	
workitembytearay	This is the binary dataset containing the state of the work item	NOT NULL

The CorrelationKeyInfo entity contains information about correlation keys assigned to given process instance - loose relationship as this table is considered optional used only when correlation capabilities are required.

Table 8.5. CorrelationKeyInfo

Field	Description	Nullable
keyid	The primary key	NOT NULL
name	assigned name of the correlation key	
processinstanceid	The id of the process instance which is assigned to this correlation key	NOT NULL
optlock	The version field that serves as its optimistic lock value	

The CorrelationPropertyInfo entity contains information about correlation properties for given correlation key that is assigned to given process instance.

Table 8.6. CorrelationPropertyInfo

Field	Description	Nullable
propertyid	The primary key	NOT NULL
name	The name of the property	
value	The value of the property	NOT NULL
optlock	The version field that serves as its optimistic lock value	
correlationKey-keyid	Foregin key to map to correlation key	NOT NULL

The ContextMappingInfo entity contains information about contextual information mapped to ksession. This is an internal part of RuntimeManager and can be considered optional when RuntimeManager is not used.

Table 8.7. ContextMappingInfo

Field	Description	Nullable
`mappingid`	The primary key	NOT NULL
`context_id`	Identifier of the context	NOT NULL
`ksession?id`	Identifier of the ksession mapped to this context	NOT NULL
`optlock`	The version field that serves as its optimistic lock value	

8.1.1.2. Safe Points

The state of a process instance is stored at so-called "safe points" during the execution of the process engine. Whenever a process instance is executing (for example when it started or continuing from a previous wait state, the engine executes the process instance until no more actions can be performed (meaning that the process instance either has completed (or was aborted), or that it has reached a wait state in all of its parallel paths). At that point, the engine has reached the next safe state, and the state of the process instance (and all other process instances that might have been affected) is stored persistently.

8.2. Audit Log

In many cases it will be useful (if not necessary) to store information *about* the execution of process instances, so that this information can be used afterwards. For example, sometimes we want to verify which actions have been executed for a particular process instance, or in general, we want to be able to monitor and analyze the efficiency of a particular process.

However, storing history information in the runtime database can result in the database rapidly increasing in size, not to mention the fact that monitoring and analysis queries might influence the performance of your runtime engine. This is why process execution history information can be stored separately.

This history log of execution information is created based on events that the process engine generates during execution. This is possible because the jBPM runtime engine provides a generic mechanism to listen to events. The necessary information can easily be extracted from these events and then persisted to a database. Filters can also be used to limit the scope of the logged information.

8.2.1. The jBPM Audit data model

The jbpm-audit module contains an event listener that stores process-related information in a database using JPA. The data model itself contains three entities, one for process instance information, one for node instance information, and one for (process) variable instance information.

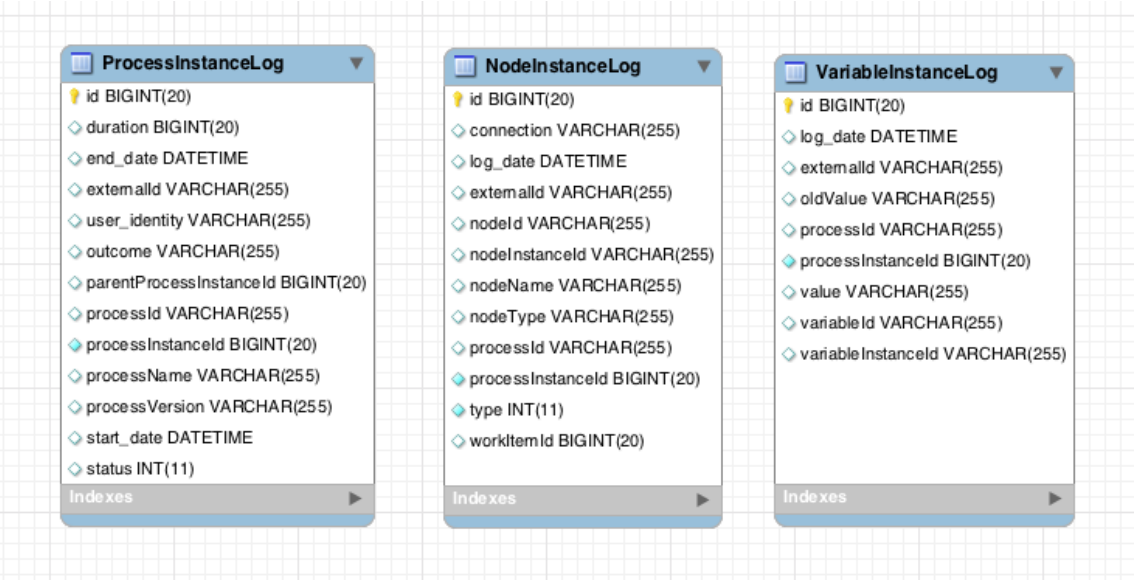


Figure 8.2. jBPM Audit data model

The `ProcessInstanceLog` table contains the basic log information about a process instance.

Table 8.8. ProcessInstanceLog

Field	Description	Nullable
`id`	The primary key and id of the log entity	NOT NULL
`duration`	Actual duration of this process instance since its start date	
`end_date`	When applicable, the end date of the process instance	
`externalId`	Optional external identifier used to correlate to some elements - e.g. deployment id	
`user_identity`	Optional identifier of the user who started the process instance	
`outcome`	The outcome of the process instance, for instance error code in case of process instance was finished with error event	
`parentProcessInstanceId`	The process instance id of the parent process instance if any	

Field	Description	Nullable
processid	The id of the process	
processinstanceid	The process instance id	NOT NULL
processname	The name of the process	
processversion	The version of the process	
start_date	The start date of the process instance	
status	The status of process instance that maps to process instance state	

The NodeInstanceLog table contains more information about which nodes were actually executed inside each process instance. Whenever a node instance is entered from one of its incoming connections or is exited through one of its outgoing connections, that information is stored in this table.

Table 8.9. NodeInstanceLog

Field	Description	Nullable
id	The primary key and id of the log entity	NOT NULL
connection	Actual identifier of the sequence flow that led to this node instance	
log_date	The date of the event	
externalId	Optional external identifier used to correlate to some elements - e.g. deployment id	
nodeid	The node id of the corresponding node in the process definition	
nodeinstanceid	The node instance id	
nodename	The name of the node	
nodetype	The type of the node	
processid	The id of the process that the process instance is executing	
processinstanceid	The process instance id	NOT NULL
type	The type of the event (0 = enter, 1 = exit)	NOT NULL

Field	Description	Nullable
`workItemId`	Optional - only for certain node types - The identifier of work item	

The `VariableInstanceLog` table contains information about changes in variable instances. The default is to only generate log entries when (after) a variable changes. It's also possible to log entries before the variable (value) changes.

Table 8.10. VariableInstanceLog

Field	Description	Nullable
`id`	The primary key and id of the log entity	NOT NULL
`externalId`	Optional external identifier used to correlate to some elements - e.g. deployment id	
`log_date`	The date of the event	
`processid`	The id of the process that the process instance is executing	
`processinstanceid`	The process instance id	NOT NULL
`oldvalue`	The previous value of the variable at the time that the log is made	
`value`	The value of the variable at the time that the log is made	
`variableid`	The variable id in the process definition	
`variableinstanceid`	The id of the variable instance	

8.2.2. Storing Process Events in a Database

To log process history information in a database like this, you need to register the logger on your session like this:

```
EntityManagerFactory emf = ...;
StatefulKnowledgeSession ksession = ...;
AbstractAuditLogger auditLogger = AuditLoggerFactory.newJPAInstance(emf);
ksession.addProcessEventListener(auditLogger);

// invoke methods one your session here
```

To specify the database where the information should be stored, modify the file `persistence.xml` file to include the audit log classes as well (ProcessInstanceLog, NodeInstanceLog and VariableInstanceLog), as shown below.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>


<persistence
  version="2.0"
        xsi:schemaLocation="http://java.sun.com/xml/ns/persistence    http://
java.sun.com/xml/ns/persistence/persistence_2_0.xsd
      http://java.sun.com/xml/ns/persistence/orm   http://java.sun.com/xml/ns/
persistence/orm_2_0.xsd"
  xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:orm="http://java.sun.com/xml/ns/persistence/orm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance>

  <persistence-unit name="org.jbpm.persistence.jpa" transaction-type="JTA">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <jta-data-source>jdbc/jbpm-ds</jta-data-source>
    <mapping-file>META-INF/JBPMorm.xml</mapping-file>
    <class>org.drools.persistence.info.SessionInfo</class>
    <class>org.jbpm.persistence.processinstance.ProcessInstanceInfo</class>
    <class>org.drools.persistence.info.WorkItemInfo</class>
    <class>org.jbpm.persistence.correlation.CorrelationKeyInfo</class>
    <class>org.jbpm.persistence.correlation.CorrelationPropertyInfo</class>
    <class>org.jbpm.runtime.manager.impl.jpa.ContextMappingInfo</class>

    <class>org.jbpm.process.audit.ProcessInstanceLog</class>
    <class>org.jbpm.process.audit.NodeInstanceLog</class>
    <class>org.jbpm.process.audit.VariableInstanceLog</class>

    <properties>
     <property name="hibernate.dialect" value="org.hibernate.dialect.H2Dialect"/
>
      <property name="hibernate.max_fetch_depth" value="3"/>
      <property name="hibernate.hbm2ddl.auto" value="update"/>
      <property name="hibernate.show_sql" value="true"/>
      <property name="hibernate.transaction.jta.platform"
      value="org.hibernate.service.jta.platform.internal.BitronixJtaPlatform"/>
    </properties>
  </persistence-unit>
</persistence>
```

All this information can easily be queried and used in a lot of different use cases, ranging from creating a history log for one specific process instance to analyzing the performance of all instances of a specific process.

This audit log should only be considered a default implementation. We don't know what information you need to store for analysis afterwards, and for performance reasons it is recommended to only store the relevant data. Depending on your use cases, you might define your own data model for storing the information you need, and use the process event listeners to extract that information.

8.2.3. Storing Process Events in a JMS queue for further processing

Process events are stored in the database synchronously and within the same transaction as actual process instance execution. That obviously takes some time especially in highly loaded systems and might have some impact on the database when both history log and runtime data are kept in the same database. To provide an alternative option for storing process events, a JMS based logger has been provided. It can be configured to submit messages to JMS queue instead of directly persisting them in the database. It can be configured to be transactional as well to avoid issues with inconsistent data in case of process engine transaction is rolled back.

```
ConnectionFactory factory = ...;
Queue queue = ...;
StatefulKnowledgeSession ksession = ...;
Map<String, Object> jmsProps = new HashMap<String, Object>();
jmsProps.put("jbpm.audit.jms.transacted", true);
jmsProps.put("jbpm.audit.jms.connection.factory", factory);
jmsProps.put("jbpm.audit.jms.queue", queue);
AbstractAuditLogger auditLogger = AuditLoggerFactory.newInstance(Type.JMS, session, jmsProps);
ksession.addProcessEventListener(auditLogger);

// invoke methods one your session here
```

This is just one of possible ways to configure JMS audit logger, see javadocs for AuditLoggerFactory for more details.

8.3. Transactions

The jBPM engine supports JTA transactions. It also supports local transactions *only* when using Spring. It does not support pure local transactions at the moment. For more information about using Spring to set up persistence, please see the Spring chapter in the Drools integration guide.

Whenever you do not provide transaction boundaries inside your application, the engine will automatically execute each method invocation on the engine in a separate transaction. If this

behavior is acceptable, you don't need to do anything else. You can, however, also specify the transaction boundaries yourself. This allows you, for example, to combine multiple commands into one transaction.

You need to register a transaction manager at the environment before using user-defined transactions. The following sample code uses the Bitronix transaction manager. Next, we use the Java Transaction API (JTA) to specify transaction boundaries, as shown below:

```
// create the entity manager factory and register it in the environment
EntityManagerFactory emf = Persistence.createEntityManagerFactory( "org.jbpm.persistence.jpa" )
Environment env = KnowledgeBaseFactory.newEnvironment();
env.set( EnvironmentName.ENTITY_MANAGER_FACTORY, emf );
env.set( EnvironmentName.TRANSACTION_MANAGER, TransactionManagerServices.getTransactionManager(

// create a new knowledge session that uses JPA to store the runtime state
StatefulKnowledgeSession ksession = JPAKnowledgeService.newStatefulKnowledgeSession( kbase, nul

// start the transaction
UserTransaction ut = (UserTransaction) new InitialContext().lookup( "java:comp/
UserTransaction" );
ut.begin();

// perform multiple commands inside one transaction
ksession.insert( new Person( "John Doe" ) );
ksession.startProcess( "MyProcess" );

// commit the transaction
ut.commit();
```

Note that, if you use Bitronix as the transaction manager, you should also add a simple `jndi.properties` file in you root classpath to register the Bitronix transaction manager in JNDI. If you are using the jbpm-test module, this is already included by default. If not, create a file named `jndi.properties` with the following content:

```
java.naming.factory.initial=bitronix.tm.jndi.BitronixInitialContextFactory
```

If you would like to use a different JTA transaction manager, you can change the `persistence.xml` file to use your own transaction manager. For example, when running inside JBoss Application Server v5.x or v7.x, you can use the JBoss transaction manager. You need to change the transaction manager property in `persistence.xml` to:

```
<property                                name="hibernate.transaction.jta.platform"
 value="org.hibernate.transaction.JBossTransactionManagerLookup" />
```

8.3.1. Container managed transaction

Special consideration need to be taken when embedding jBPM inside an application that executes in Container Managed Transaction (CMT) mode, for instance EJB beans. This especially applies to application servers that does not allow accessing UserTransaction instance from JNDI when being part of container managed transaction, e.g. WebSphere Application Server. Since default implementation of transaction manager in jBPM is based on UserTransaction to get transaction status which is used to decide if transaction should be started or not, in environments that prevent accessing UserTrancation it won't do its job. To secure proper execution in CMT environments a dedicated transaction manager implementation is provided:

```
org.jbpm.persistence.jta.ContainerManagedTransactionManager
```

This transaction manager expects that transaction is active and thus will always return ACTIVE when invoking getStatus method. Operations like begin, commit, rollback are no-op methods as transaction manager runs under managed transaction and can't affect it.

Note

To make sure that container is aware of any exceptions that happened during process instance execution, user needs to ensure that exceptions thrown by the engine are propagated up to the container to properly rollback transaction.

To configure this transaction manager following must be done:

- Insert transaction manager and persistence context manager into environment prior to creating/ loading session

```
Environment env = EnvironmentFactory.newEnvironment();
env.set(EnvironmentName.ENTITY_MANAGER_FACTORY, emf);
env.set(EnvironmentName.TRANSACTION_MANAGER,                                new
 ContainerManagedTransactionManager());
env.set(EnvironmentName.PERSISTENCE_CONTEXT_MANAGER,                        new
 JpaProcessPersistenceContextManager(env));
env.set(EnvironmentName.TASK_PERSISTENCE_CONTEXT_MANAGER,                   new
 JPATaskPersistenceContextManager(env));
```

```
```

- configure JPA provider (example hibernate and WebSphere)

```
<property                          name="hibernate.transaction.factory_class"
 value="org.hibernate.transaction.CMTTransactionFactory"/>
<property                           name="hibernate.transaction.jta.platform"
 value="org.hibernate.service.jta.platform.internal.WebSphereJtaPlatform"/>
```

With following configuration jBPM should run properly in CMT environment.

8.3.1.1. CMT dispose ksession command

Usually when running within container managed transaction disposing ksession directly will cause exceptions on transaction completion as there are some transaction synchronization registered by jBPM to clean up the state after invocation is finished. To overcome this problem specialized command has been provided `org.jbpm.persistence.jta.ContainerManagedTransactionDisposeCommand` which allows to simply execute this command instead of regular `ksession.dispose` which will ensure that ksession will be disposed at the transaction completion.

8.4. Configuration

By default, the engine does not save runtime data persistently. This means you can use the engine completely without persistence (so not even requiring an in memory database) if necessary, for example for performance reasons, or when you would like to manage persistence yourself. It is, however, possible to configure the engine to do use persistence by configuring it to do so. This usually requires adding the necessary dependencies, configuring a datasource and creating the engine with persistence configured.

8.4.1. Adding dependencies

You need to make sure the necessary dependencies are available in the classpath of your application if you want to user persistence. By default, persistence is based on the Java Persistence API (JPA) and can thus work with several persistence mechanisms. We are using Hibernate by default.

If you're using the Eclipse IDE and the jBPM Eclipse plugin, you should make sure the necessary JARs are added to your jBPM runtime directory. You don't really need to do anything (as the necessary dependencies should already be there) if you are using the jBPM runtime that is configured by default when using the jBPM installer, or if you downloaded and unzipped the jBPM runtime artifact (from the downloads) and pointed the jBPM plugin to that directory.

If you would like to manually add the necessary dependencies to your project, first of all, you need the JAR file `jbpm-persistence-jpa.jar`, as that contains code for saving the runtime state whenever necessary. Next, you also need various other dependencies, depending on the persistence solution and database you are using. For the default combination with Hibernate as the JPA persistence provider and using an H2 in-memory database and Bitronix for JTA-based transaction management, the following list of additional dependencies is needed:

- jbpm-persistence-jpa (org.jbpm)
- drools-persistence-jpa (org.drools)
- persistence-api (javax.persistence)
- hibernate-entitymanager (org.hibernate)
- hibernate-annotations (org.hibernate)
- hibernate-commons-annotations (org.hibernate)
- hibernate-core (org.hibernate)
- commons-collections (commons-collections)
- dom4j (dom4j)
- jta (javax.transaction)
- btm (org.codehaus.btm)
- javassist (javassist)
- slf4j-api (org.slf4j)
- slf4j-jdk14 (org.slf4j)
- h2 (com.h2database)
- jbpm-test (org.jbpm) for testing only, do not include it in the actual application

8.4.2. Manually configuring the engine to use persistence

You can use the `JPAKnowledgeService` to create your knowledge session. This is slightly more complex, but gives you full access to the underlying configurations. You can create a new knowledge session using `JPAKnowledgeService` based on a knowledge base, a knowledge session configuration (if necessary) and an environment. The environment needs to contain a reference to your Entity Manager Factory. For example:

```
// create the entity manager factory and register it in the environment
```

```
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory( "org.jbpm.persistence.jpa" );
Environment env = KnowledgeBaseFactory.newEnvironment();
env.set( EnvironmentName.ENTITY_MANAGER_FACTORY, emf );

// create a new knowledge session that uses JPA to store the runtime state
StatefulKnowledgeSession ksession = JPAKnowledgeService.newStatefulKnowledgeSession( kbase, nul
int sessionId = ksession.getId();

// invoke methods on your method here
ksession.startProcess( "MyProcess" );
ksession.dispose();
```

You can also use the `JPAKnowledgeService` to recreate a session based on a specific session id:

```
// recreate the session from database using the sessionId
ksession = JPAKnowledgeService.loadStatefulKnowledgeSession(sessionId, kbase, null, env );
```

Note that we only save the minimal state that is needed to continue execution of the process instance at some later point. This means, for example, that it does not contain information about already executed nodes if that information is no longer relevant, or that process instances that have been completed or aborted are removed from the database. If you want to search for history-related information, you should use the history log, as explained later.

You need to add a persistence configuration to your classpath to configure JPA to use Hibernate and the H2 database (or your own preference), called `persistence.xml` in the META-INF directory, as shown below. For more details on how to change this for your own configuration, we refer to the JPA and Hibernate documentation for more information.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<persistence
      version="2.0"
           xsi:schemaLocation="http://java.sun.com/xml/ns/persistence  http://
java.sun.com/xml/ns/persistence/persistence_2_0.xsd
        http://java.sun.com/xml/ns/persistence/orm  http://java.sun.com/xml/ns/
persistence/orm_2_0.xsd"
      xmlns="http://java.sun.com/xml/ns/persistence"
      xmlns:orm="http://java.sun.com/xml/ns/persistence/orm"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance>

  <persistence-unit name="org.jbpm.persistence.jpa" transaction-type="JTA">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <jta-data-source>jdbc/jbpm-ds</jta-data-source>
    <mapping-file>META-INF/JBPMorm.xml</mapping-file>
```

```
    <class>org.drools.persistence.info.SessionInfo</class>
    <class>org.jbpm.persistence.processinstance.ProcessInstanceInfo</class>
    <class>org.drools.persistence.info.WorkItemInfo</class>
    <class>org.jbpm.persistence.correlation.CorrelationKeyInfo</class>
    <class>org.jbpm.persistence.correlation.CorrelationPropertyInfo</class>
    <class>org.jbpm.runtime.manager.impl.jpa.ContextMappingInfo</class>

    <properties>
     <property name="hibernate.dialect" value="org.hibernate.dialect.H2Dialect"/
>
      <property name="hibernate.max_fetch_depth" value="3"/>
      <property name="hibernate.hbm2ddl.auto" value="update"/>
      <property name="hibernate.show_sql" value="true"/>
      <property name="hibernate.transaction.jta.platform"
       value="org.hibernate.service.jta.platform.internal.BitronixJtaPlatform"/
>
    </properties>
  </persistence-unit>
</persistence>
```

This configuration file refers to a data source called "jdbc/jbpm-ds". If you run your application in an application server (like for example JBoss AS), these containers typically allow you to easily set up data sources using some configuration (like for example dropping a datasource configuration file in the deploy directory). Please refer to your application server documentation to know how to do this.

For example, if you're deploying to JBoss Application Server v5.x, you can create a datasource by dropping a configuration file in the deploy directory, for example:

```
<?xml version="1.0" encoding="UTF-8"?>
<datasources>
  <local-tx-datasource>
    <jndi-name>jdbc/jbpm-ds</jndi-name>
    <connection-url>jdbc:h2:tcp://localhost/~/test</connection-url>
    <driver-class>org.h2.jdbcx.JdbcDataSource</driver-class>
    <user-name>sa</user-name>
    <password></password>
  </local-tx-datasource>
</datasources>
```

If you are however executing in a simple Java environment, you can use the `JBPMHelper` class to do this for you (see below for tests only) or the following code fragment could be used to set up a data source (where we are using the H2 in-memory database in combination with Bitronix in this case).

```
PoolingDataSource ds = new PoolingDataSource();
ds.setUniqueName("jdbc/jbpm-ds");
ds.setClassName("bitronix.tm.resource.jdbc.lrc.LrcXADataSource");
ds.setMaxPoolSize(3);
ds.setAllowLocalTransactions(true);
ds.getDriverProperties().put("user", "sa");
ds.getDriverProperties().put("password", "sasa");
ds.getDriverProperties().put("URL", "jdbc:h2:mem:jbpm-db");
ds.getDriverProperties().put("driverClassName", "org.h2.Driver");
ds.init();
```

8.4.3. Configuring the engine to use persistence using `JBPMHelper` - for tests only

You need to configure the jBPM engine to use persistence, usually simply by using the appropriate constructor when creating your session. There are various ways to create a session (as we have tried to make this as easy as possible for you and have several utility classes for you, depending for example if you are trying to write a process JUnit test).

The easiest way to do this is to use the `jbpm-test` module that allows you to easily create and test your processes. The `JBPMHelper` class has a method to create a session, and uses a configuration file to configure this session, like whether you want to use persistence, the datasource to use, etc. The helper class will then do all the setup and configuration for you.

To configure persistence, create a `jBPM.properties` file and configure the following properties (note that the example below are the default properties, using an H2 in-memory database with persistence enabled, if you are fine with all of these properties, you don't need to add new properties file, as it will then use these properties by default):

```
# for creating a datasource
persistence.datasource.name=jdbc/jbpm-ds
persistence.datasource.user=sa
persistence.datasource.password=
persistence.datasource.url=jdbc:h2:tcp://localhost/~/jbpm-db
persistence.datasource.driverClassName=org.h2.Driver

# for configuring persistence of the session
persistence.enabled=true
persistence.persistenceunit.name=org.jbpm.persistence.jpa
persistence.persistenceunit.dialect=org.hibernate.dialect.H2Dialect

# for configuring the human task service
taskservice.enabled=true
```

```
taskservice.datasource.name=org.jbpm.task
taskservice.usergroupcallback=org.jbpm.services.task.identity.JBossUserGroupCallbackImpl
taskservice.usergroupmapping=classpath:/usergroups.properties
```

If you want to use persistence, you must make sure that the datasource (that you specified in the `jBPM.properties` file) is initialized correctly. This means that the database itself must be up and running, and the datasource should be registered using the correct name. If you would like to use an H2 in-memory database (which is usually very easy to do some testing), you can use the `JBPMHelper` class to start up this database, using:

```
JBPMHelper.startH2Server();
```

To register the datasource (this is something you always need to do, even if you're not using H2 as your database, check below for more options on how to configure your datasource), use:

```
JBPMHelper.setupDataSource();
```

Next, you can use the `JBPMHelper` class to create your session (after creating your knowledge base, which is identical to the case when you are not using persistence):

```
StatefulKnowledgeSession ksession = JBPMHelper.newStatefulKnowledgeSession(kbase);
```

Once you have done that, you can just call methods on this ksession (like `startProcess`) and the engine will persist all runtime state in the created datasource.

You can also use the `JBPMHelper` class to recreate your session (by restoring its state from the database, by passing in the session id (that you can retrieve using `ksession.getId()`)):

```
StatefulKnowledgeSession ksession = JBPMHelper.loadStatefulKnowledgeSession(kbase, sessionId);
```

Part III. Workbench

How to use the web-based Workbench

Chapter 9. Workbench

9.1. Installation

9.1.1. War installation

From the workbench distribution zip, take the `kie-wb-*.war` that corresponds to your application server:

- `jboss-as7`: tailored for JBoss AS 7 (which is being renamed to WildFly in version 8)
- `eap-6`: tailored to JBoss EAP 6
- `tomcat7`: the generic war, works on Tomcat and Jetty

Note

The differences between these `war` files are superficial only, to allow out-of-the-box deployment. For example, some JARs might be excluded if the application server already supplies them.

To use the workbench on a different application server (WebSphere, WebLogic, ...), use the `tomcat7` war and tailor it to your application server's version.

9.1.2. Workbench data

The workbench stores its data, by default in the directory `$WORKING_DIRECTORY/.niogit`, for example `wildfly-8.0.0.Final/bin/.gitnio`, but it can be overridden with the *system property* `-Dorg.uberfire.nio.git.dir`.

Note

In production, make sure to back up the workbench data directory.

9.1.3. System properties

Here's a list of all system properties:

- **`org.uberfire.nio.git.dir`**: Location of the directory `.niogit`. Default: working directory
- **`org.uberfire.nio.git.daemon.enabled`**: Enables/disables git daemon. Default: `true`

- **`org.uberfire.nio.git.daemon.host`**: If git daemon enabled, uses this property as local host identifier. Default: `localhost`
- **`org.uberfire.nio.git.daemon.port`**: If git daemon enabled, uses this property as port number. Default: `9418`
- **`org.uberfire.nio.git.ssh.enabled`**: Enables/disables ssh daemon. Default: `true`
- **`org.uberfire.nio.git.ssh.host`**: If ssh daemon enabled, uses this property as local host identifier. Default: `localhost`
- **`org.uberfire.nio.git.ssh.port`**: If ssh daemon enabled, uses this property as port number. Default: `8001`
- **`org.uberfire.nio.git.ssh.cert.dir`**: Location of the directory `.security` where local certtificates will be stored. Default: working directory
- **`org.uberfire.metadata.index.dir`**: Place where Lucene `.index` folder will be stored. Default: working directory
- **`org.uberfire.cluster.id`**: Name of the helix cluster, for example: `kie-cluster`
- **`org.uberfire.cluster.zk`**: Connection string to zookeeper. This is of the form `host1:port1,host2:port2,host3:port3`, for example: `localhost:2188`
- **`org.uberfire.cluster.local.id`**: Unique id of the helix cluster node, note that '`:`' is replaced with '_', for example: `node1_12345`
- **`org.uberfire.cluster.vfs.lock`**: Name of the resource defined on helix cluster, for example: `kie-vfs`
- **`org.uberfire.cluster.autostart`**: Delays VFS clustering until the application is fully initialized to avoid conflicts when all cluster members create local clones. Default: `false`
- **`org.uberfire.sys.repo.monitor.disabled`**: Disable configuration monitor (do not disable unless you know what you're doing). Default: `false`
- **`org.uberfire.secure.key`**: Secret password used by password encryption. Default: `org.uberfire.admin`
- **`org.uberfire.secure.alg`**: Crypto algorithm used by password encryption. Default: `PBEWithMD5AndDES`
- **`org.guvnor.m2repo.dir`**: Place where Maven repository folder will be stored. Default: working-directory/repositories/kie
- **`org.kie.example.repositories`**: Folder from where demo repositories will be cloned. The demo repositories need to have been obtained and placed in this folder. Demo repositories can

be obtained from the kie-wb-6.1.0-SNAPSHOT-example-repositories.zip artifact. This System Property takes precedence over org.kie.demo and org.kie.example. Default: Not used.

- **`org.kie.demo`**: Enables external clone of a demo application from GitHub. This System Property takes precedence over org.kie.example. Default: `true`
- **`org.kie.example`**: Enables example structure composed by Repository, Organization Unit and Project. Default: `false`

To change one of these system properties in a WildFly or JBoss EAP cluster:

1. Edit the file `$JBOSS_HOME/domain/configuration/host.xml`.
2. Locate the XML elements `server` that belong to the `main-server-group` and add a system property, for example:

```
<system-properties>
  <property name="org.uberfire.nio.git.dir" value="..." boot-time="false"/>
  ...
</system-properties>
```

9.2. Quick Start

These steps help you get started with minimum of effort.

They should not be a substitute for reading the documentation in full.

9.2.1. Add repository

Create a new repository to hold your project by selecting the Administration Perspective.

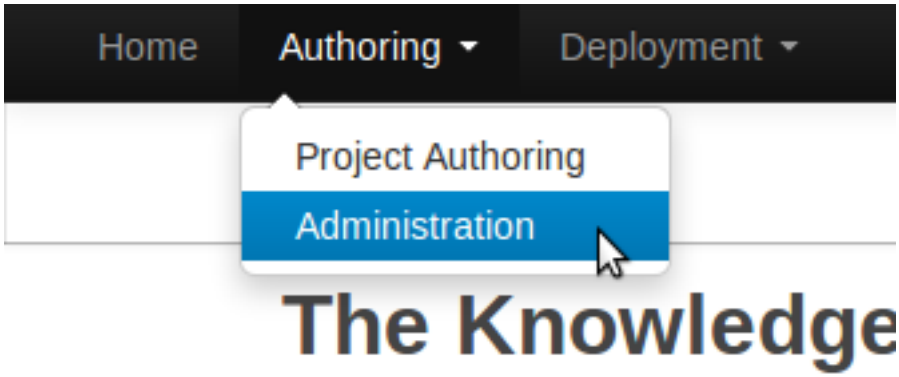


Figure 9.1. Selecting Administration perspective

Select the "New repository" option from the menu.

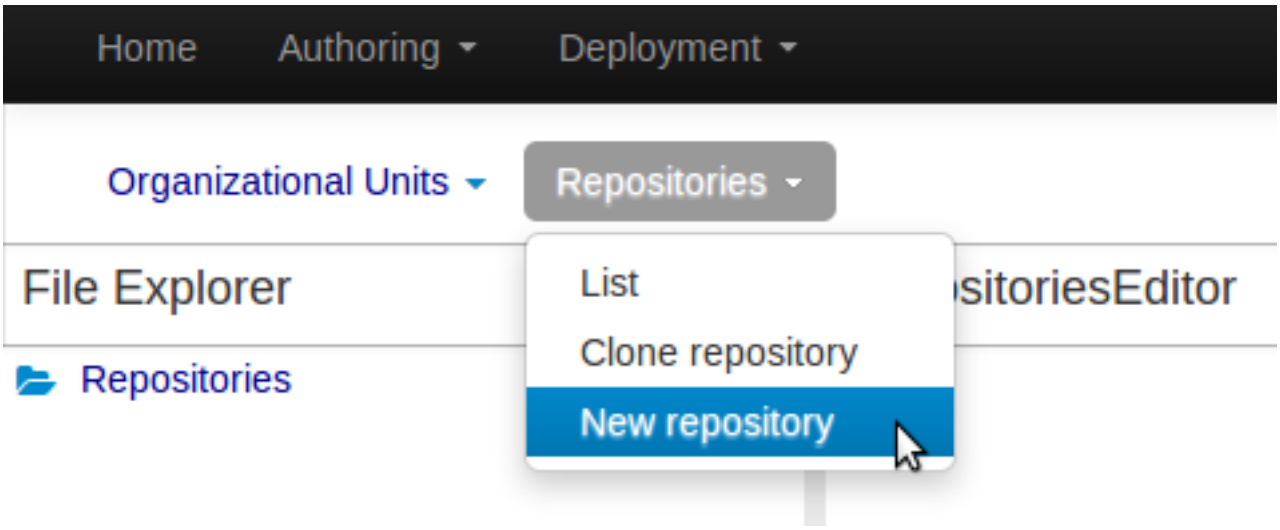


Figure 9.2. Creating new repository

Enter the required information.

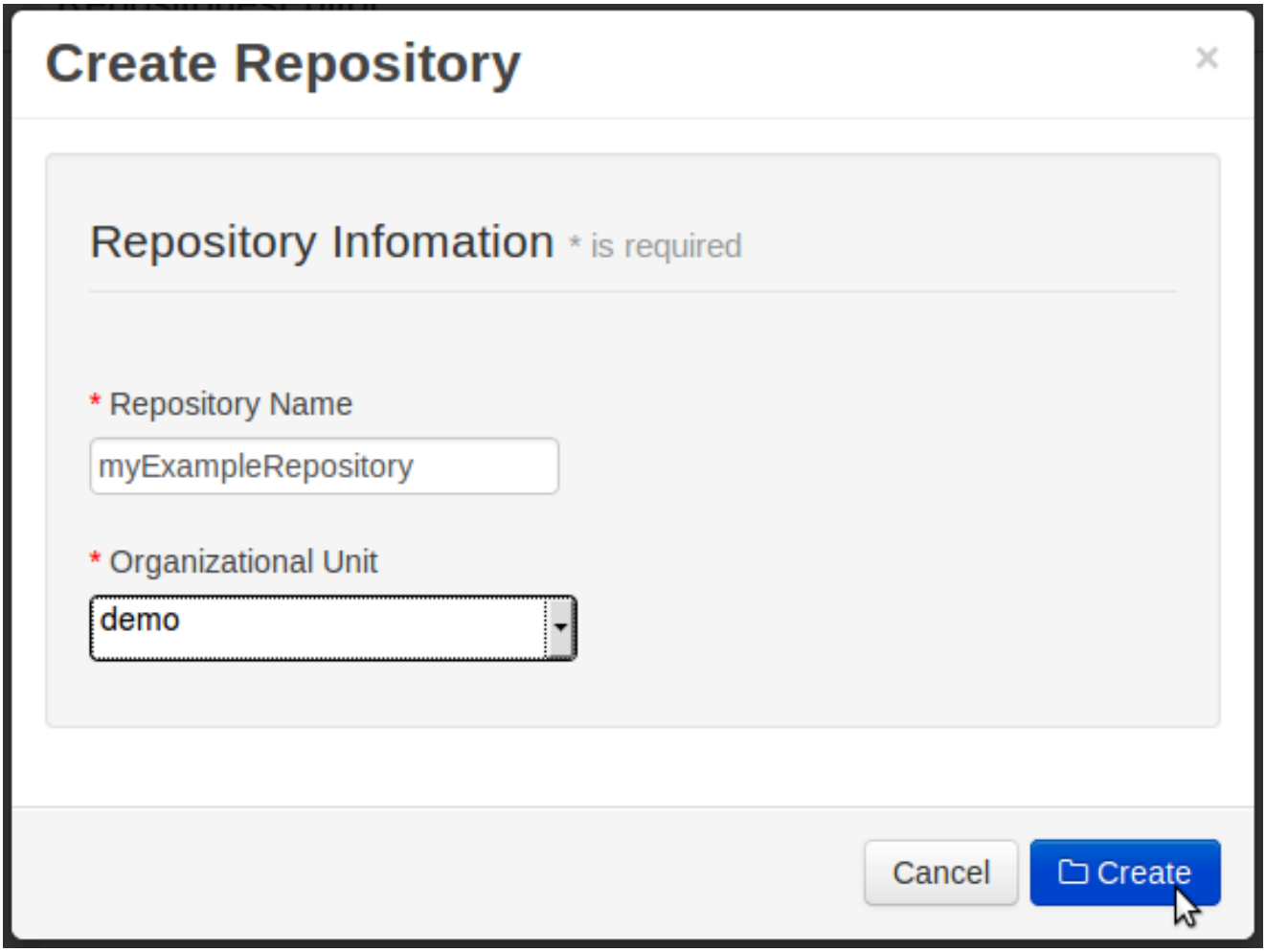


Figure 9.3. Entering repository information

9.2.2. Add project

Select the Authoring Perspective to create a new project.

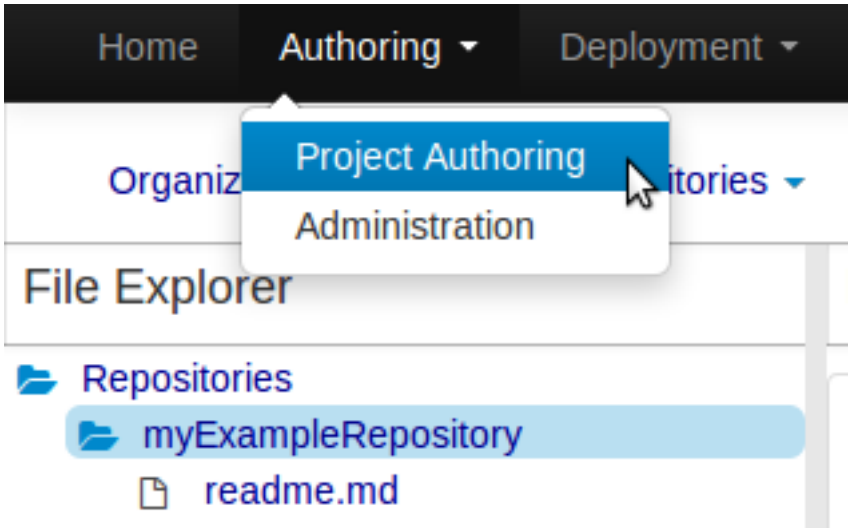


Figure 9.4. Selecting Authoring perspective

Select "Project" from the "New Item" menu.

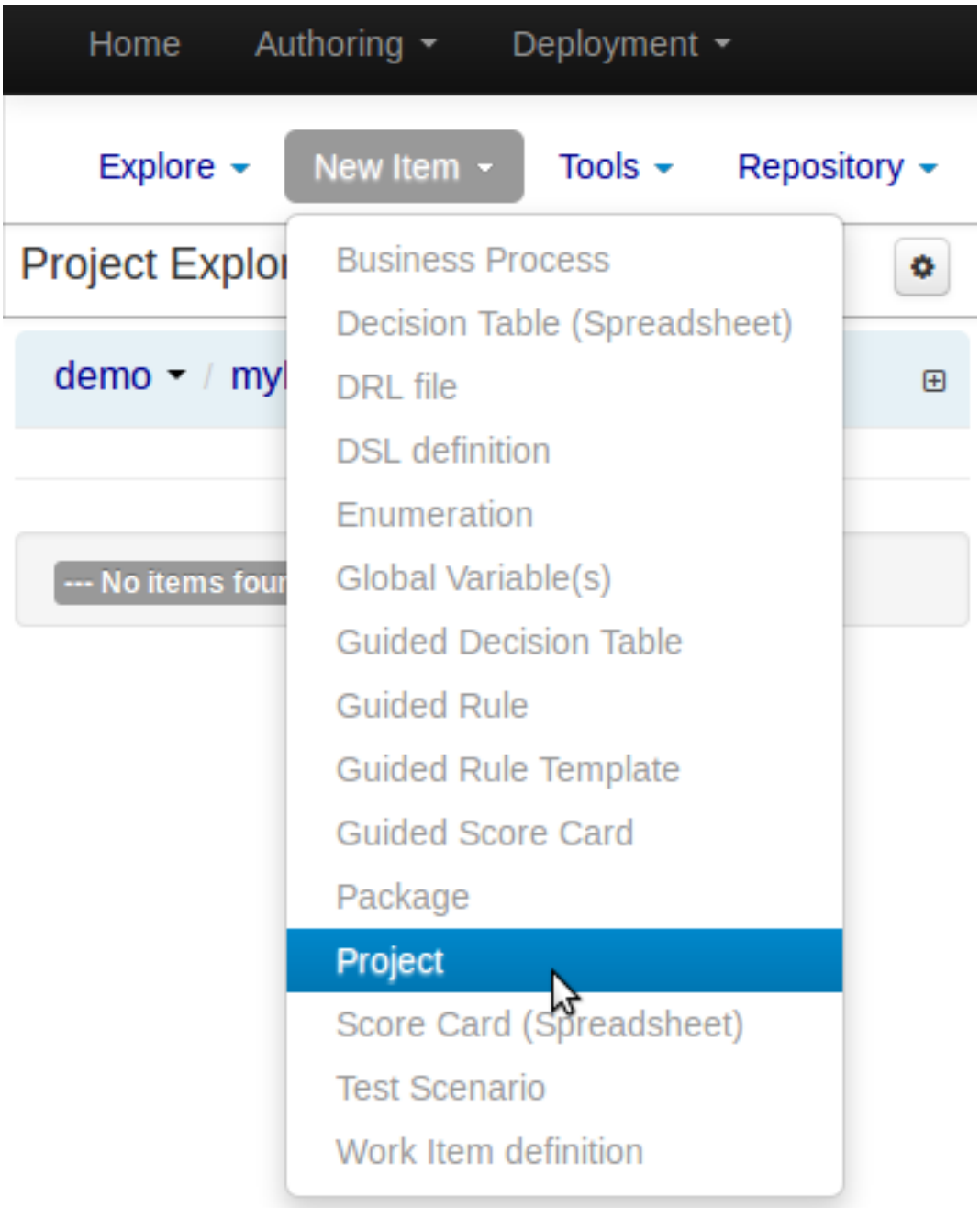


Figure 9.5. Creating new project

Enter a project name first.

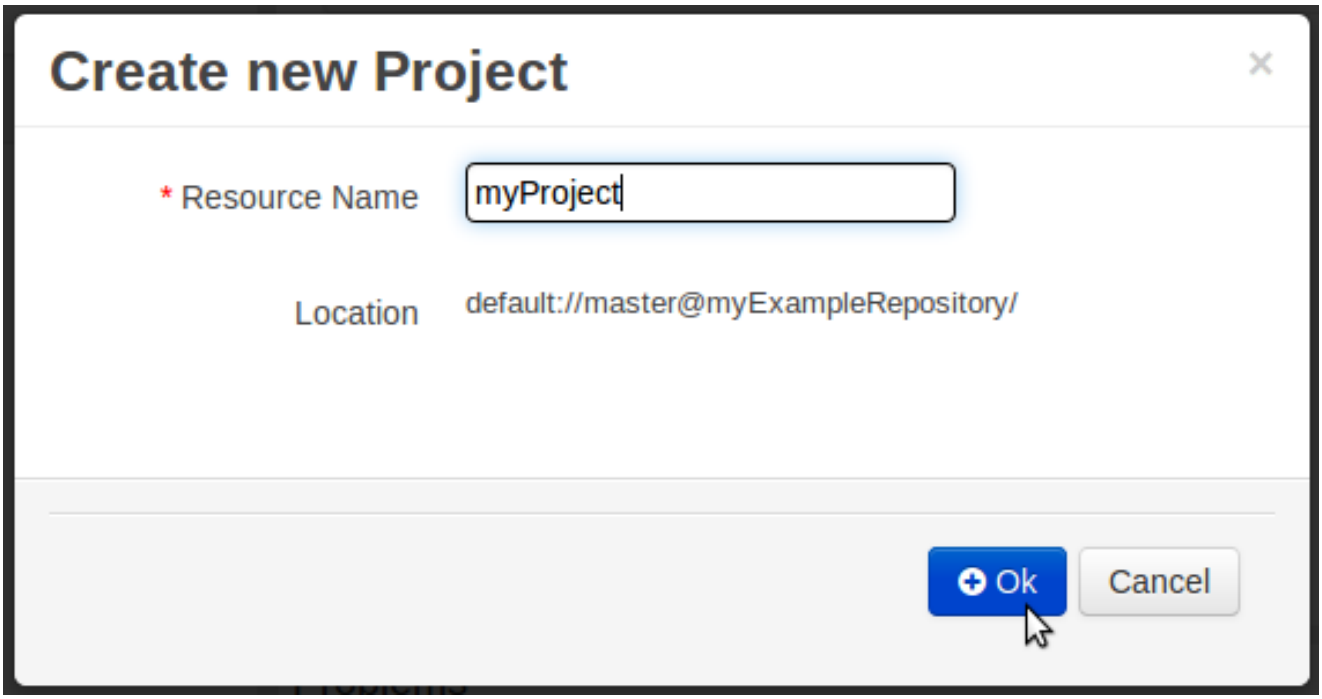


Figure 9.6. Entering project name

Enter the project details next.

- Group ID follows Maven conventions.
- Artifact ID is pre-populated from the project name.
- Version follows Maven conventions.

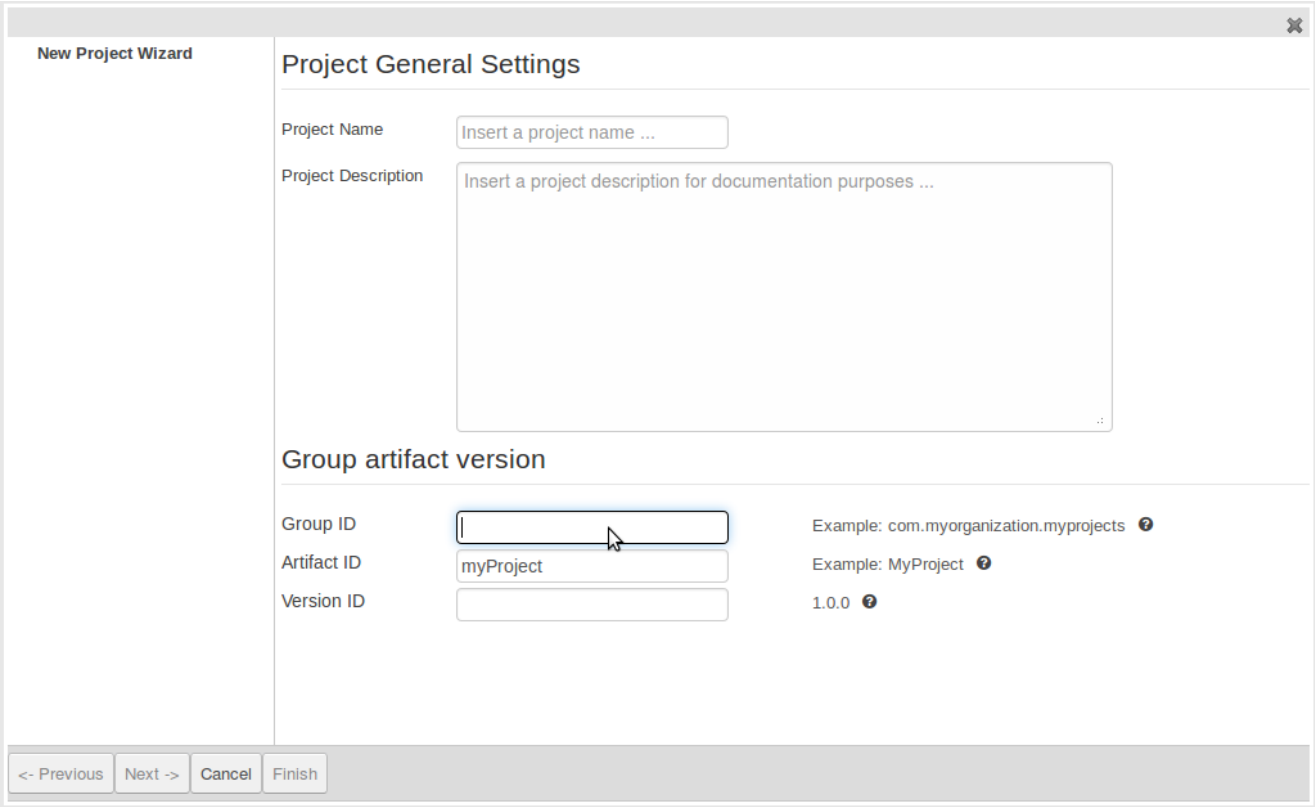


Figure 9.7. Entering project details

9.2.3. Define Data Model

After a project has been created you need to define Types to be used by your rules.

Select "Data Modeller" from the "Tools" menu.

Note

You can also use types contained in existing JARs.

Please consult the full documentation for details.

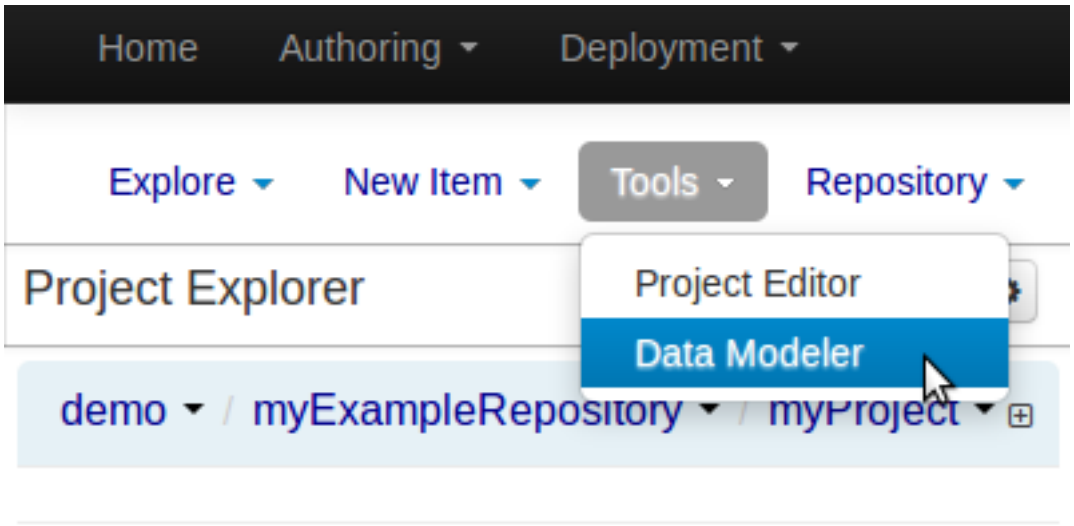


Figure 9.8. Selecting "Data Modeller"

Click on "Create" to create a new type.

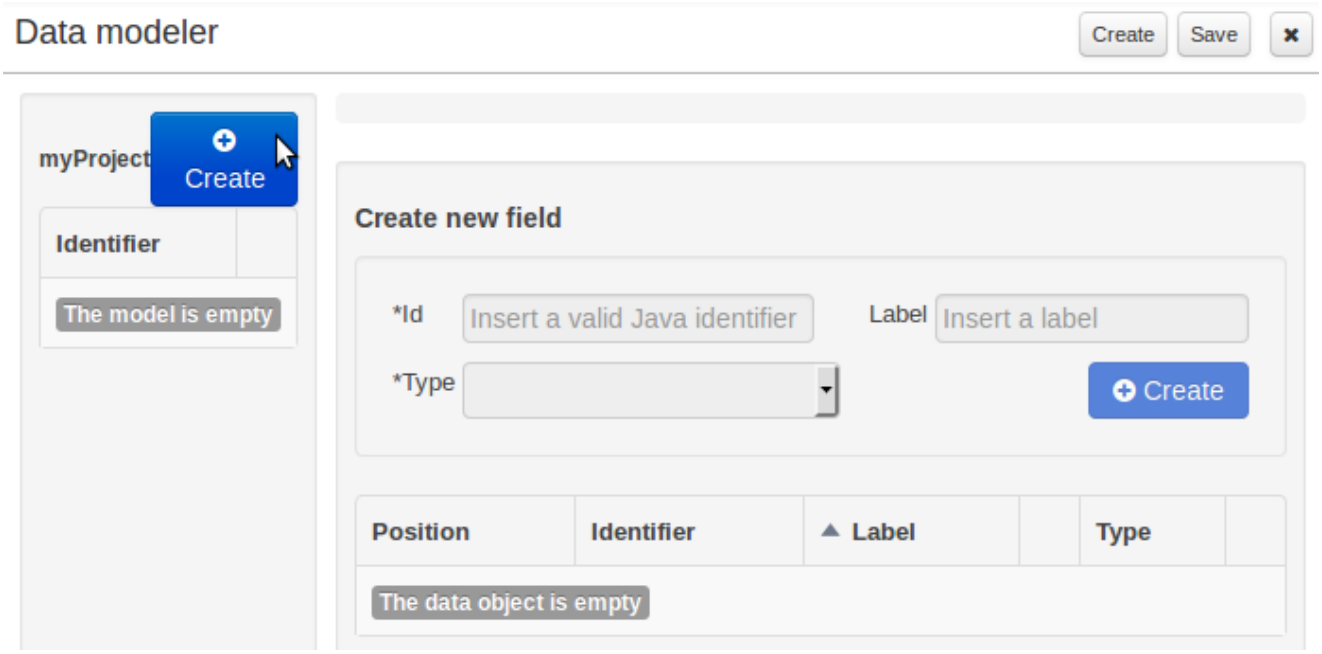


Figure 9.9. Selecting "Create" (type)

Enter the required details for the type.

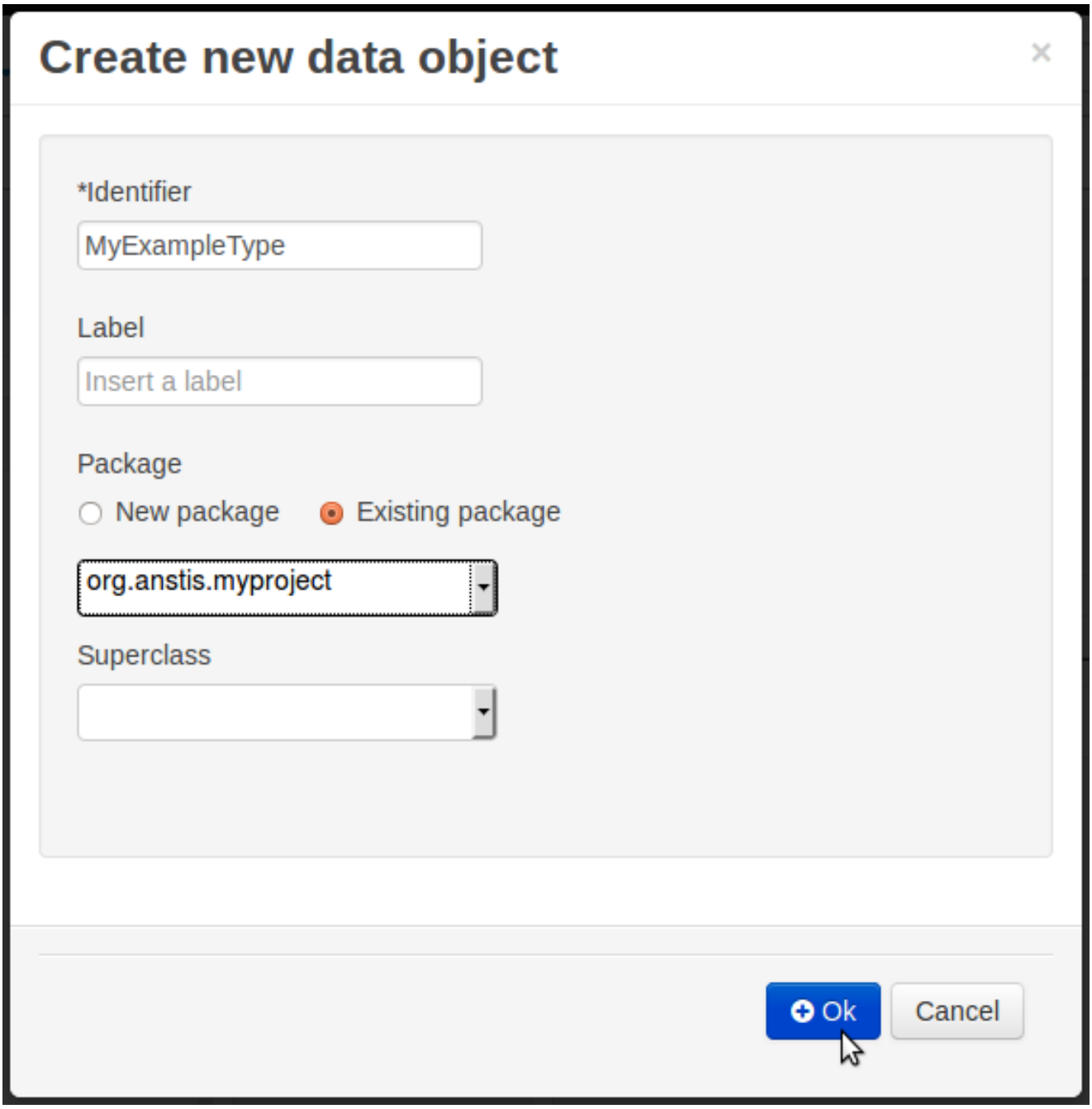


Figure 9.10. Entering required details

Click on "Create" to create a field for the type.

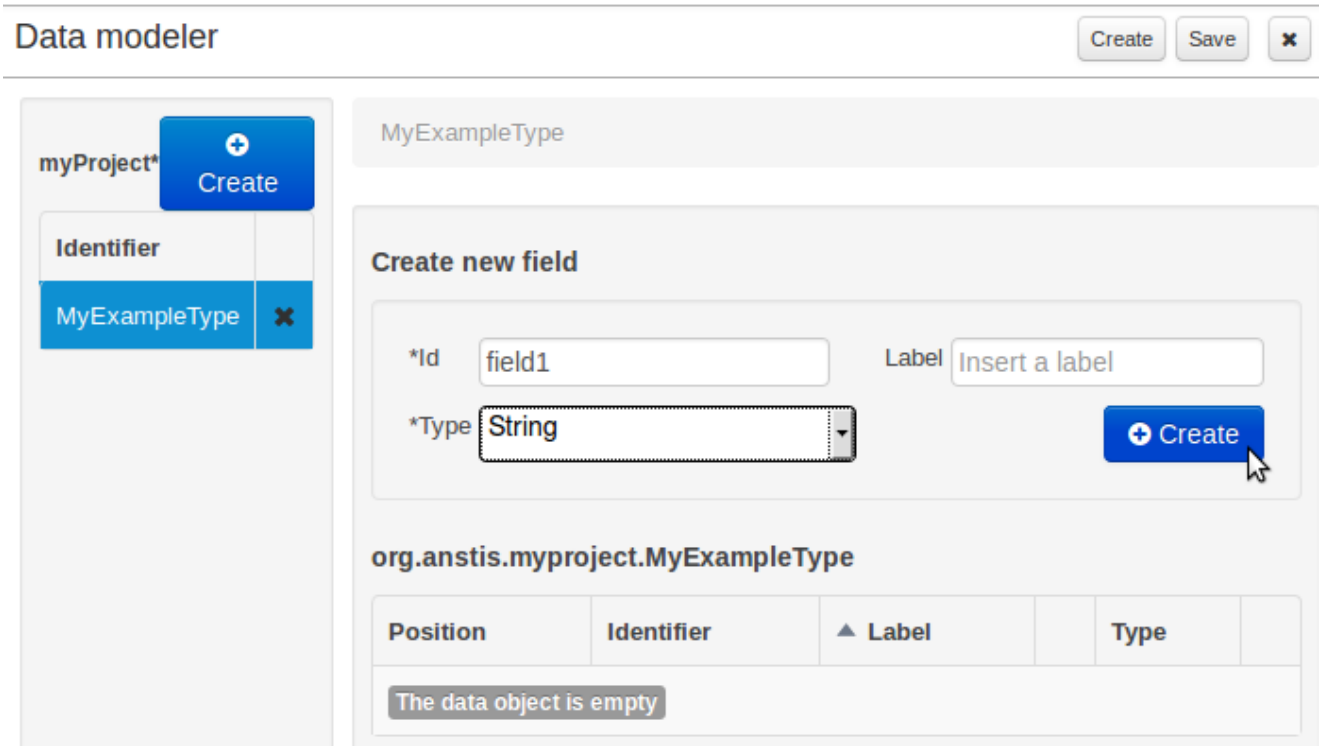


Figure 9.11. Selecting "Create" (field)

Click "Save" to create the model.

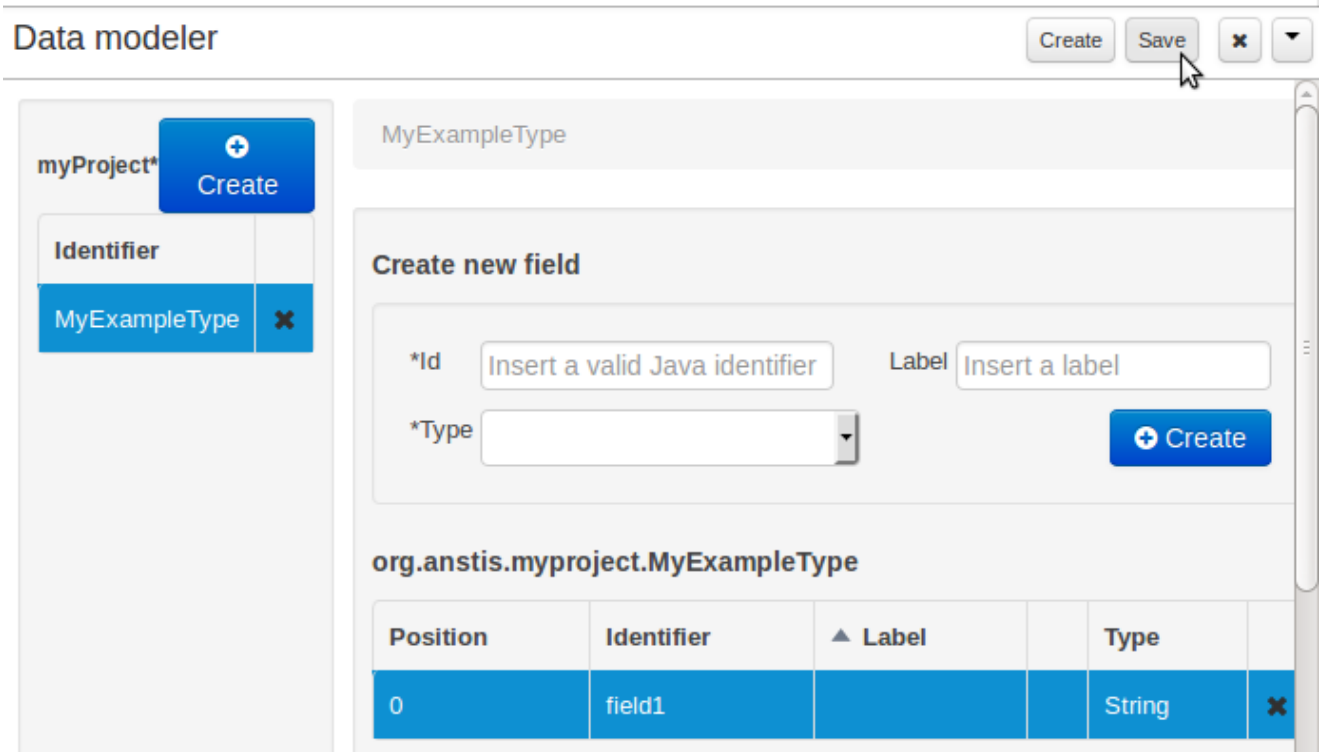


Figure 9.12. Clicking "Save"

9.2.4. Define Rule

Select "DRL file" (for example) from the "New Item" menu.

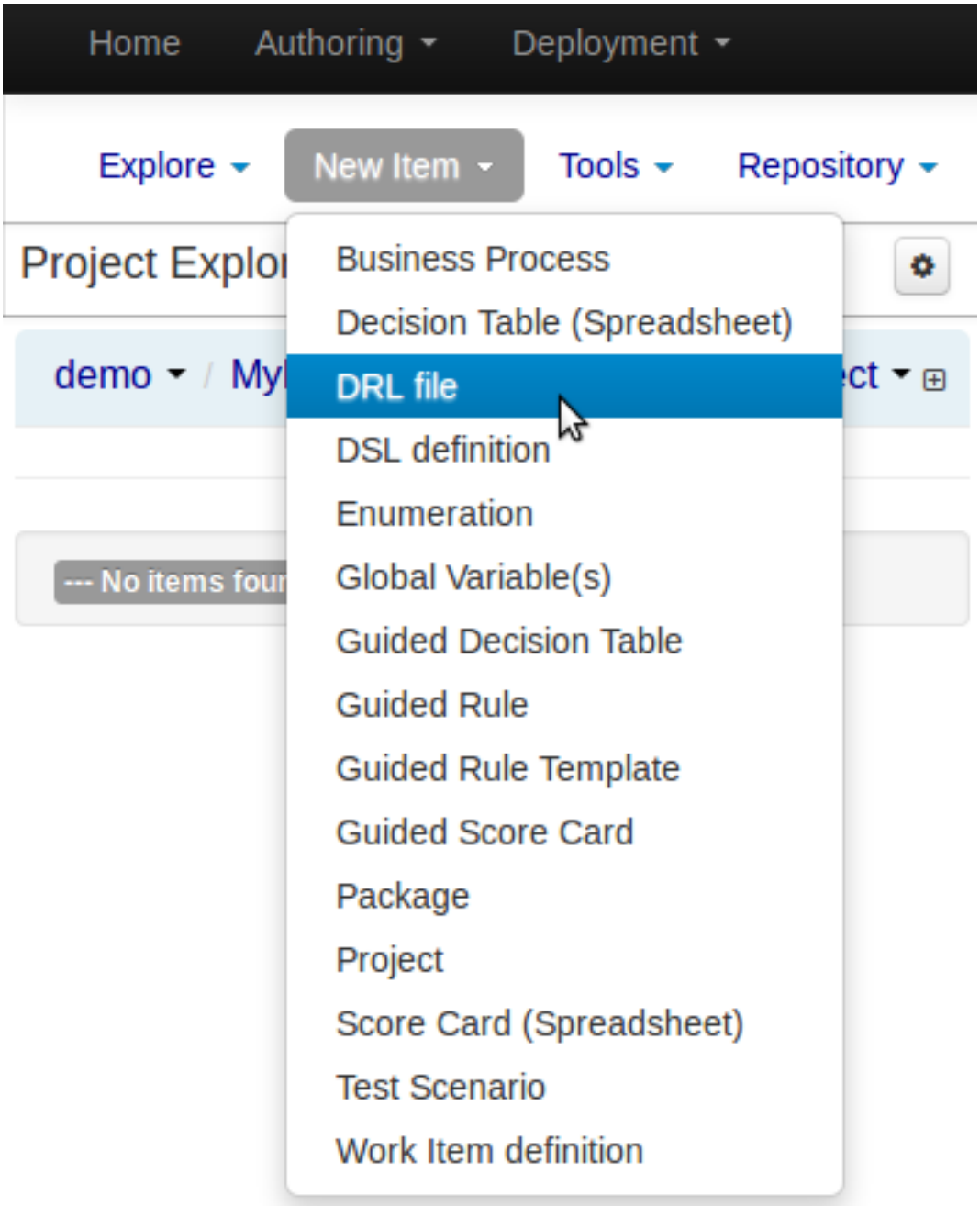


Figure 9.13. Selecting "DRL file" from the "New Item" menu

Enter a file name for the new rule.

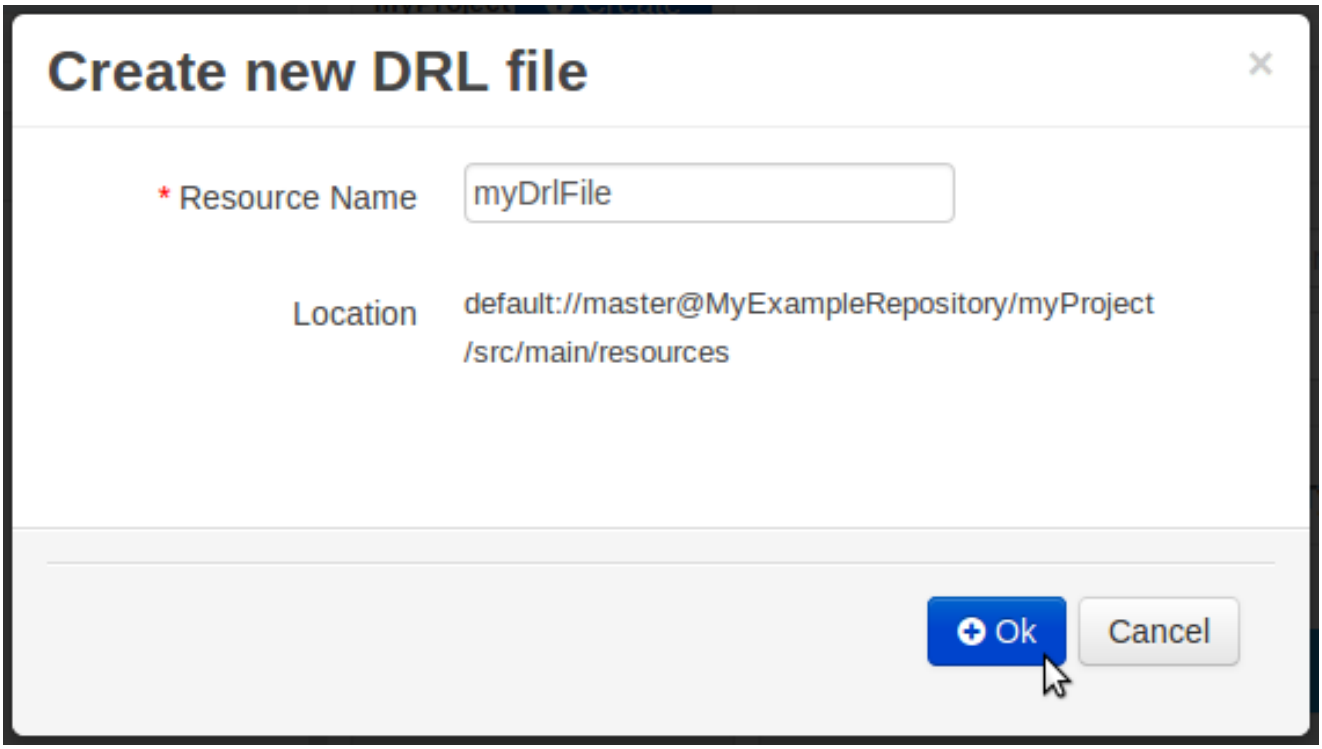


Figure 9.14. Entering file name for rule

Enter a definition for the rule.

The definition process differs from asset type to asset type.

The full documentation has details about the different editors.

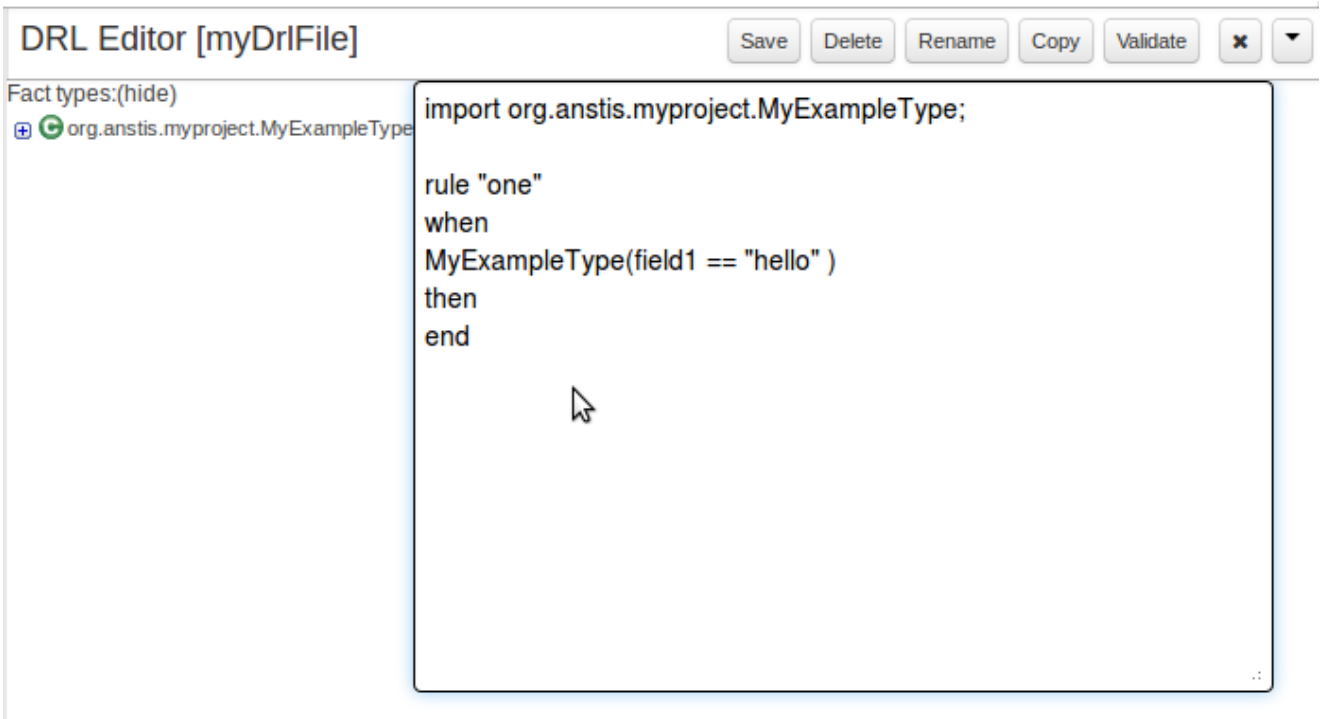


Figure 9.15. Defining a rule

Once the rule has been defined it will need to be saved.

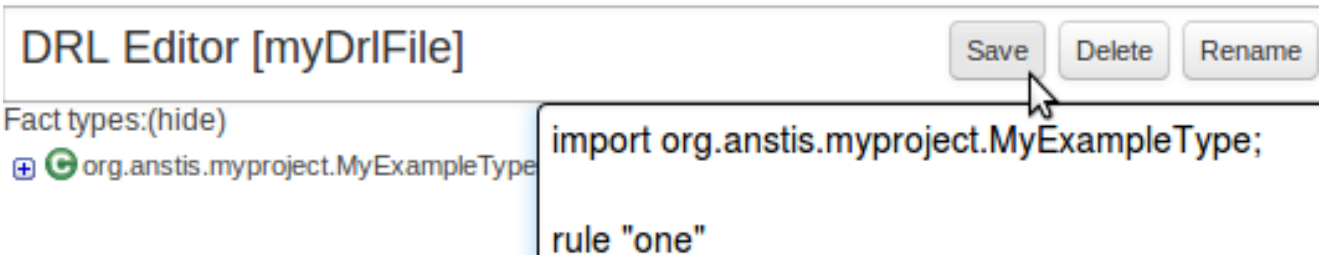


Figure 9.16. Saving the rule

9.2.5. Build and Deploy

Once rules have been defined within a project; the project can be built and deployed to the Workbench's Maven Artifact Repository.

To build a project select the "Project Editor" from the "Tools" menu.

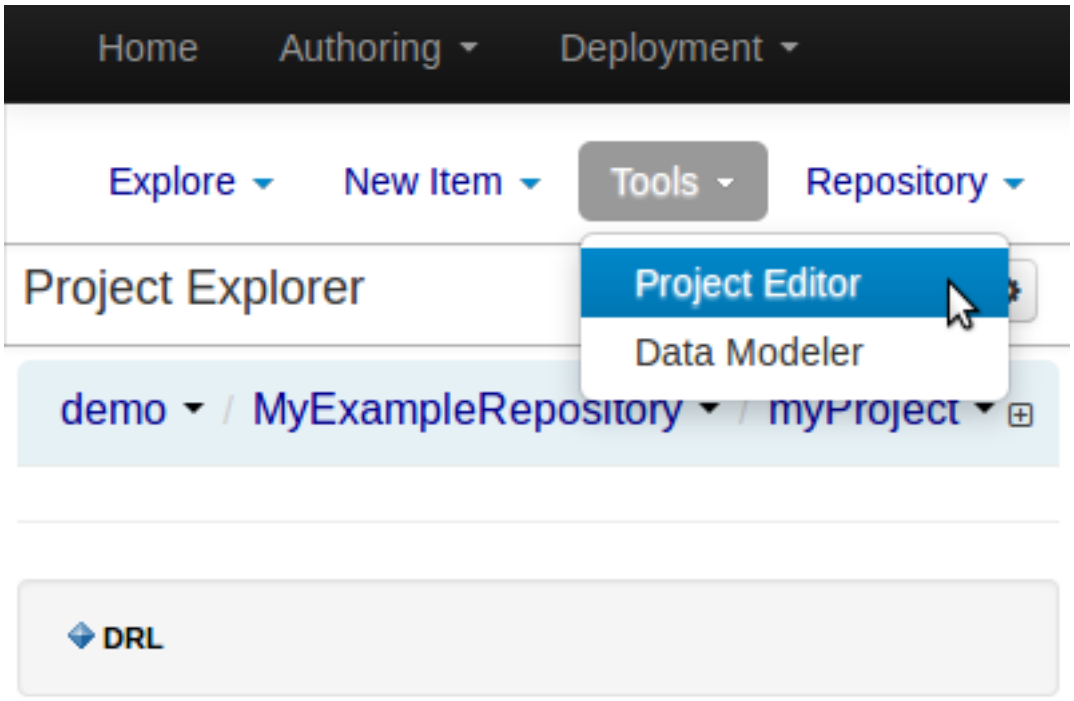


Figure 9.17. Selecting "Project Editor"

Click "Build and Deploy" to build the project and deploy it to the Workbench's Maven Artifact Repository.

When you select Build & Deploy the workbench will deploy to any repositories defined in the Dependency Management section of the pom in your workbench project. You can edit the pom.xml file associated with your workbench project under the Repository View of the project explorer. Details on dependency management in maven can be found here : http://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html

If there are errors during the build process they will be reported in the "Problems Panel".

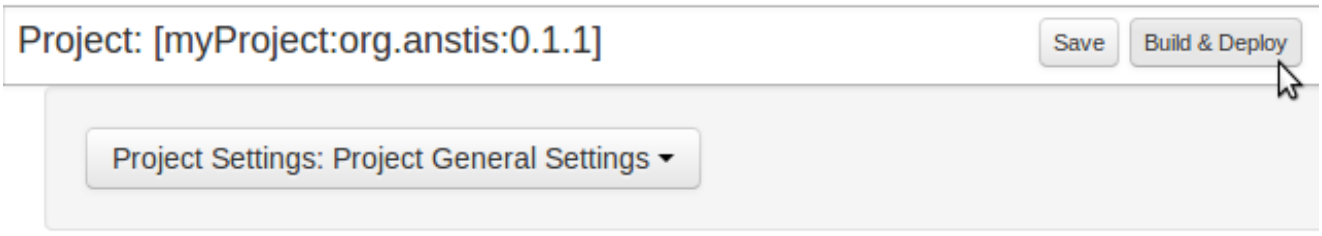


Figure 9.18. Building and deploying a project

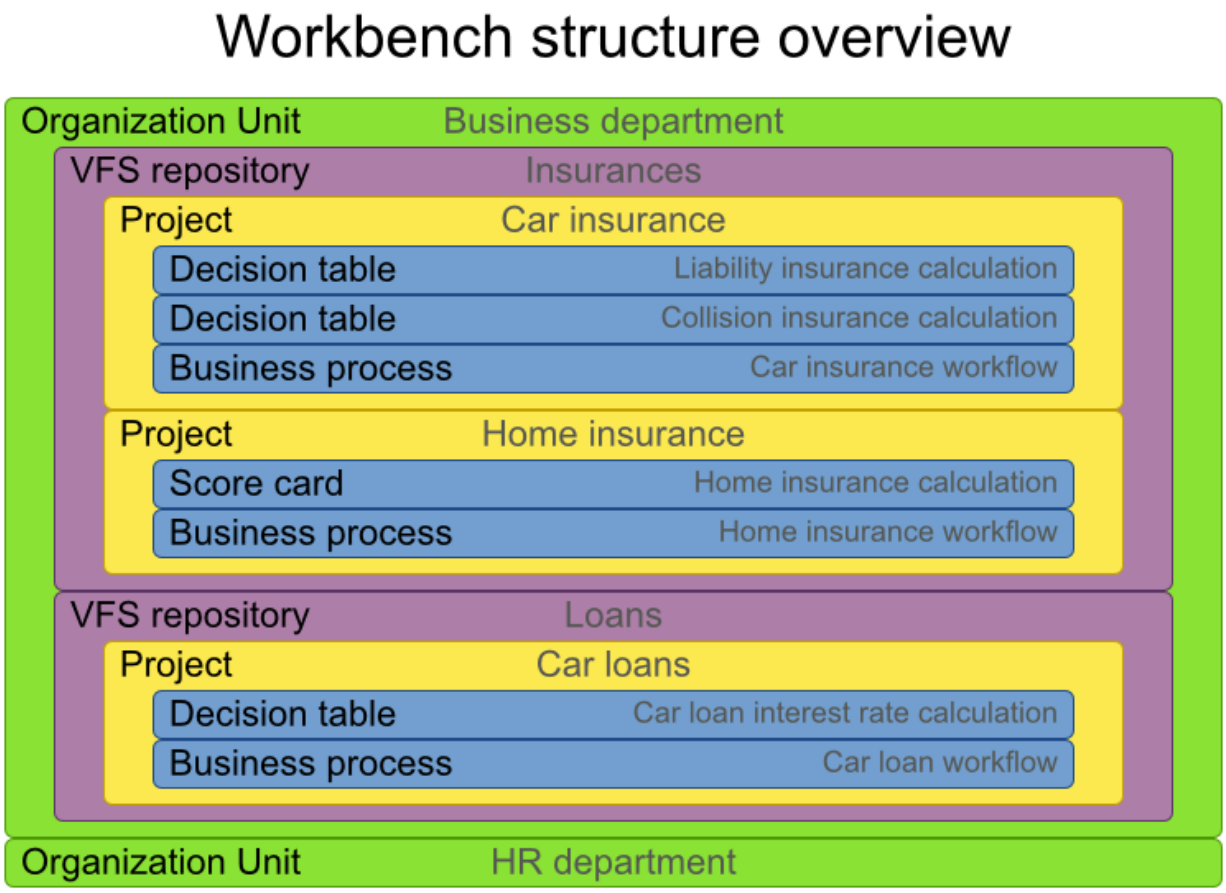
Now the project has been built and deployed; it can be referenced from your own projects as any other Maven Artifact.

The full documentation contains details about integrating projects with your own applications.

9.3. Administration

9.3.1. Administration overview

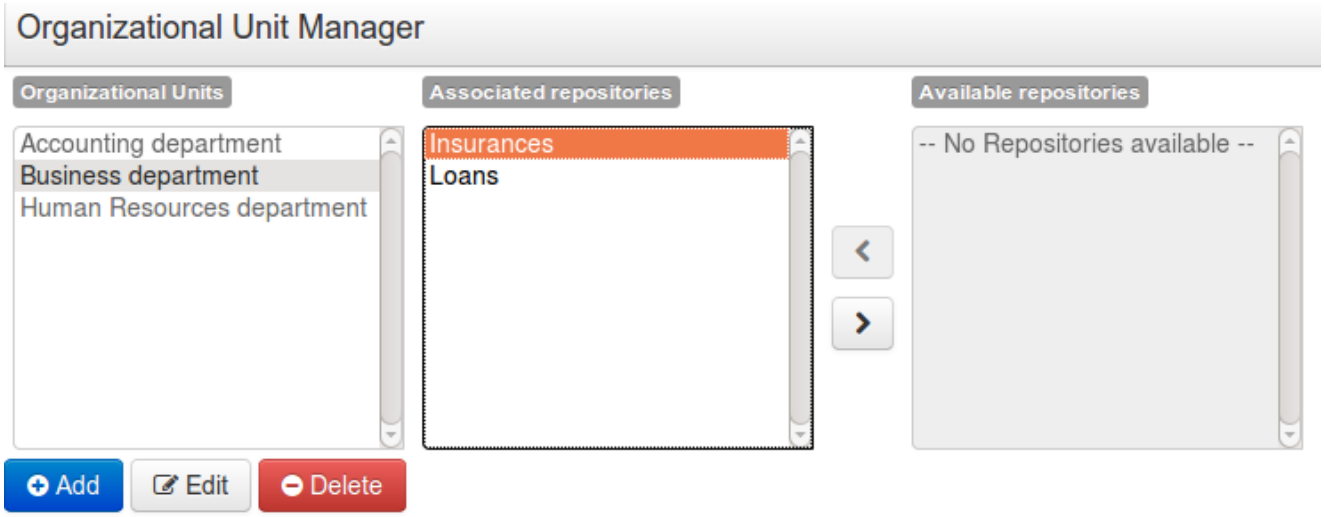
A workbench is structured with Organization Units, VFS repositories and projects:



9.3.2. Organizational unit

Organization units are useful to model departments and divisions.

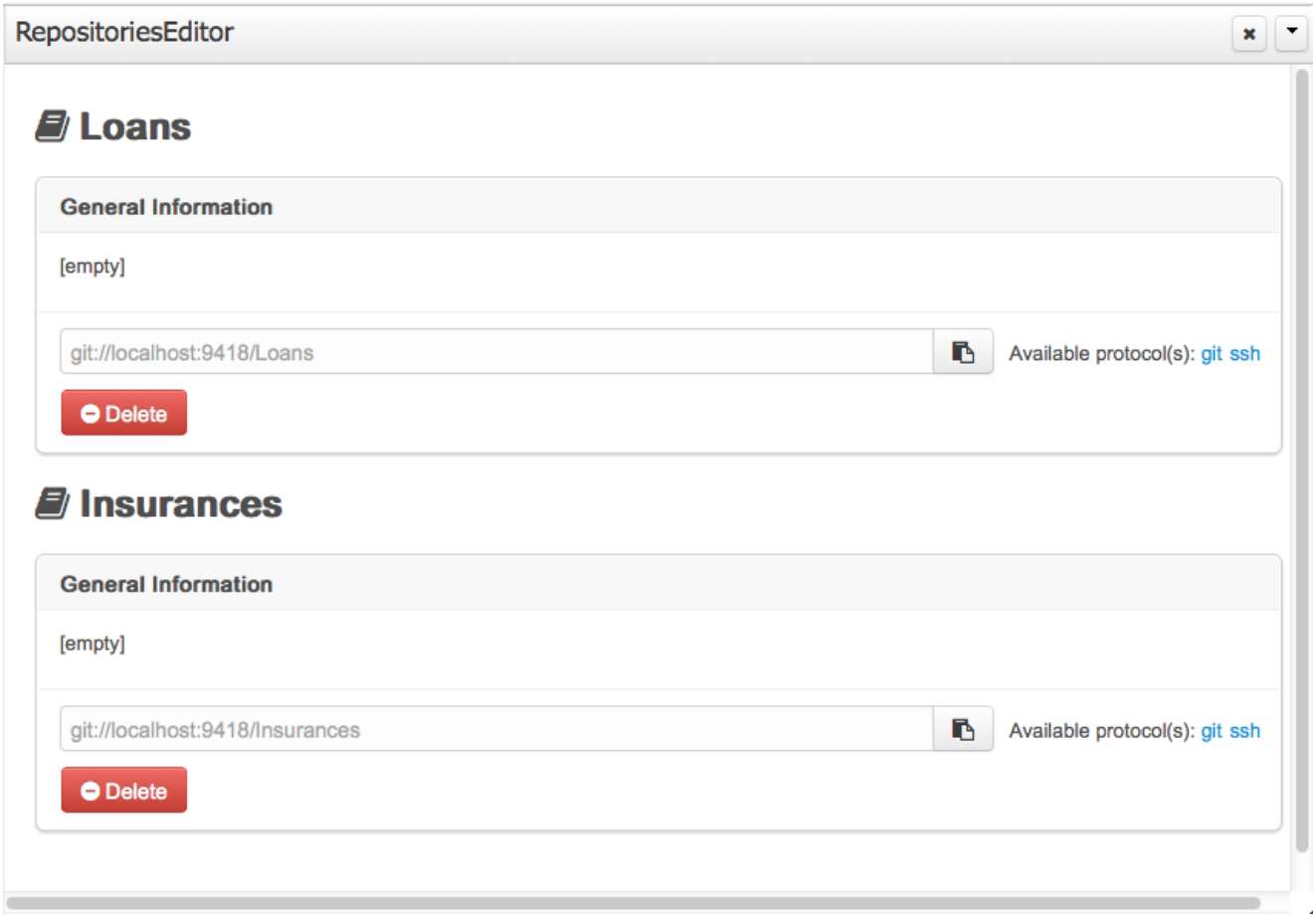
An organization unit can hold multiple repositories.



9.3.3. Repositories

Repositories are the place where assets are stored and each repository is organized by projects and belongs to a single organization unit.

Repositories are in fact a Virtual File System based storage, that by default uses GIT as backend. Such setup allows workbench to work with multiple backends and, in the same time, take full advantage of backend specifics features like in GIT case versioning, branching and even external access.



A new repository can be created from scratch or cloned from an existing repository.

One of the biggest advantage of using GIT as backend is the ability to clone a repository from external and use your preferred tools to edit and build your assets.

Note

It's important to follow Workbench structure: each project defined in a directory in repository root.

Warning

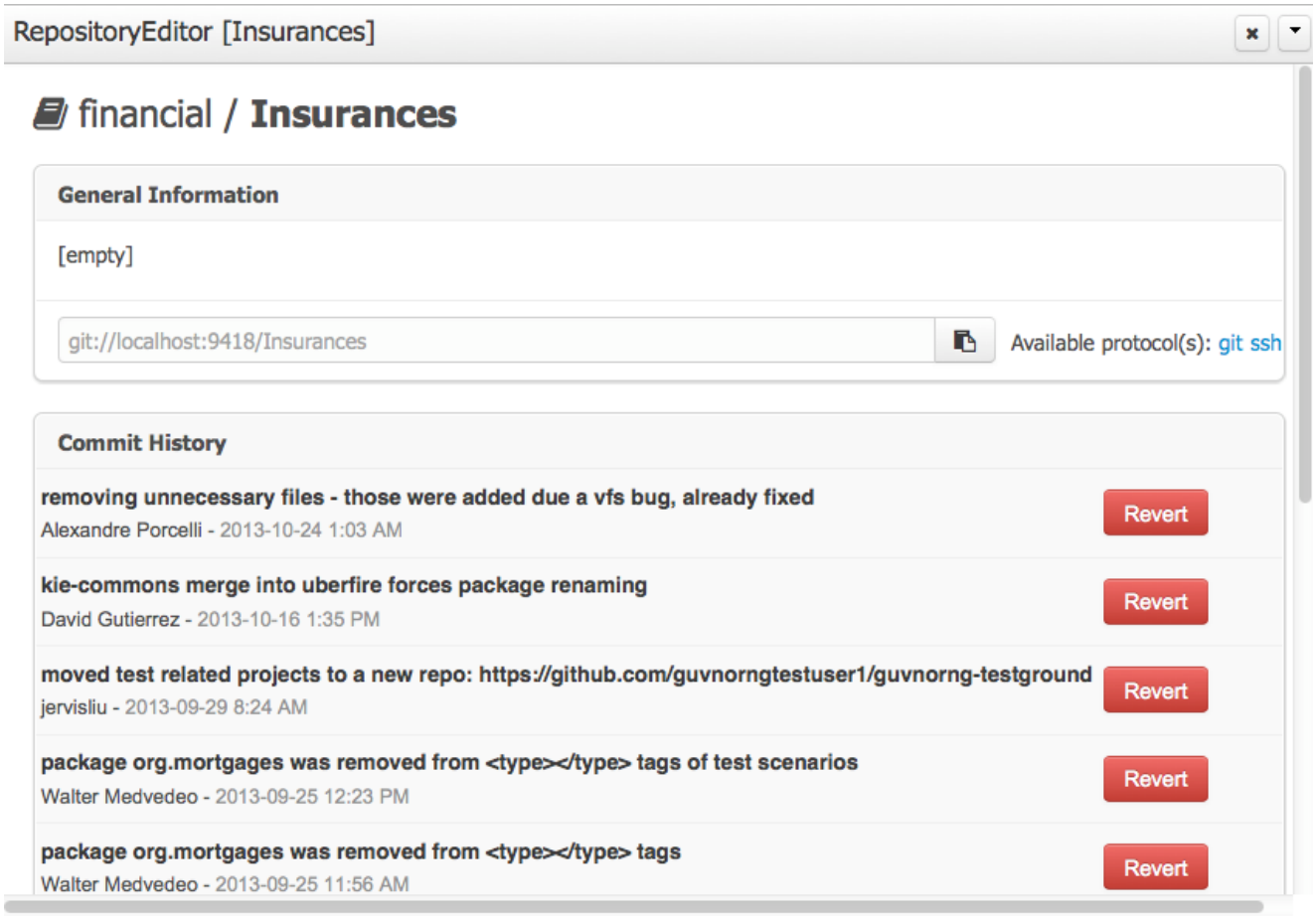
Workbench doesn't support multi projects

Warning

Never clone your repositories directly from *.niogit* directory. Use always the available protocol(s) displayed in repositories editor.

9.3.3.1. Repository Editor

One additional advantage to use GIT as backend is the possibility to revert your repository to a previous state. You can do it directly from the repository editor by browsing its commit history and clicking the **Revert** button.



9.4. Configuration

9.4.1. User management

The workbench authenticates its users against the application server's authentication and authorization (JAAS).

On JBoss EAP and WildFly, add a user with the script `$JBOSS_HOME/bin/add-user.sh` (or `.bat`):

```
$ ./add-user.sh
// Type: Application User
// Realm: empty (defaults to ApplicationRealm)
// Role: admin
```

There is no need to restart the application server.

9.4.2. Roles

The Workbench uses the following roles:

- admin
- analyst
- developer
- manager
- user

9.4.2.1. Admin

Administrates the BPMS system.

- Manages users
- Manages VFS Repositories
- Has full access to make any changes necessary

9.4.2.2. Developer

Developer can do almost everything admin can do, except clone repositories.

- Manages rules, models, process flows, forms and dashboards
- Manages the asset repository
- Can create, build and deploy projects
- Can use the JBDS connection to view processes

9.4.2.3. Analyst

Analyst is a weaker version of developer and does not have access to the asset repository or the ability to deploy projects.

9.4.2.4. Business user

Daily user of the system to take actions on business tasks that are required for the processes to continue forward. Works primarily with the task lists.

- Does process management

- Handles tasks and dashboards

9.4.2.5. Manager/Viewer-only User

Viewer of the system that is interested in statistics around the business processes and their performance, business indicators, and other reporting of the system and people who interact with the system.

- Only has access to dashboards

9.4.3. Restricting access to repositories

It is possible to restrict access to repositories using roles and organizational groups. To let an user access a repository.

The user either has to belong into a role that has access to the repository or to a role that belongs into an orgazinational group that has access to the repository. These restrictions can be managed with the command line config tool.

9.4.4. Command line config tool

Provides capabilities to manage the system repository from command line. System repository contains the data about general workbench settings: how editors behave, organizational groups, security and other settings that are not editable by the user. System repository exists in the .niogit folder, next to all the repositories that have been created or cloned into the workbench.

9.4.4.1. Config Tool Modes

- Online (default and recommended) - Connects to the Git repository on startup, using Git server provided by the KIE Workbench. All changes are made locally and published to upstream when:
  - "push-changes" command is explicitly executed
  - "exit" is used to close the tool
- Offline - Creates and manipulates system repository directly on the server (no discard option)

9.4.4.2. Available Commands

Table 9.1. Available Commands

exit	Publishes local changes, cleans up temporary directories and quits the command line tool
discard	Discards local changes without publishing them, cleans up temporary directories and quits the command line tool

help	Prints a list of available commands
list-repo	List available repositories
list-org-units	List available organizational units
list-deployment	List available deployments
create-org-unit	Creates new organizational unit
remove-org-unit	Removes existing organizational unit
add-deployment	Adds new deployment unit
remove-deployment	Removes existing deployment
create-repo	Creates new git repository
remove-repo	Removes existing repository (only from config)
add-repo-org-unit	Adds repository to the organizational unit
remove-repo-org-unit	Removes repository from the organizational unit
add-role-repo	Adds role(s) to repository
remove-role-repo	Removes role(s) from repository
add-role-org-unit	Adds role(s) to organizational unit
remove-role-org-unit	Removes role(s) from organizational unit
add-role-project	Adds role(s) to project
remove-role-project	Removes role(s) from project
push-changes	Pushes changes to upstream repository (only in online mode)

9.4.4.3. How to use

The tool can be found from kie-config-cli-${version}-dist.zip. Execute the kie-config-cli.sh script and by default it will start in online mode asking for a Git url to connect to (the default value is ssh://localhost/system). To connect to a remote server, replace the host and port with appropriate values, e.g. ssh://kie-wb-host/system.

```
./kie-config-cli.sh
```

To operate in offline mode, append the offline parameter to the kie-config-cli.sh command. This will change the behaviour and ask for a folder where the .niogit (system repository) is. If .niogit does not yet exist, the folder value can be left empty and a brand new setup is created.

```
./kie-config-cli.sh offline
```

9.5. Introduction

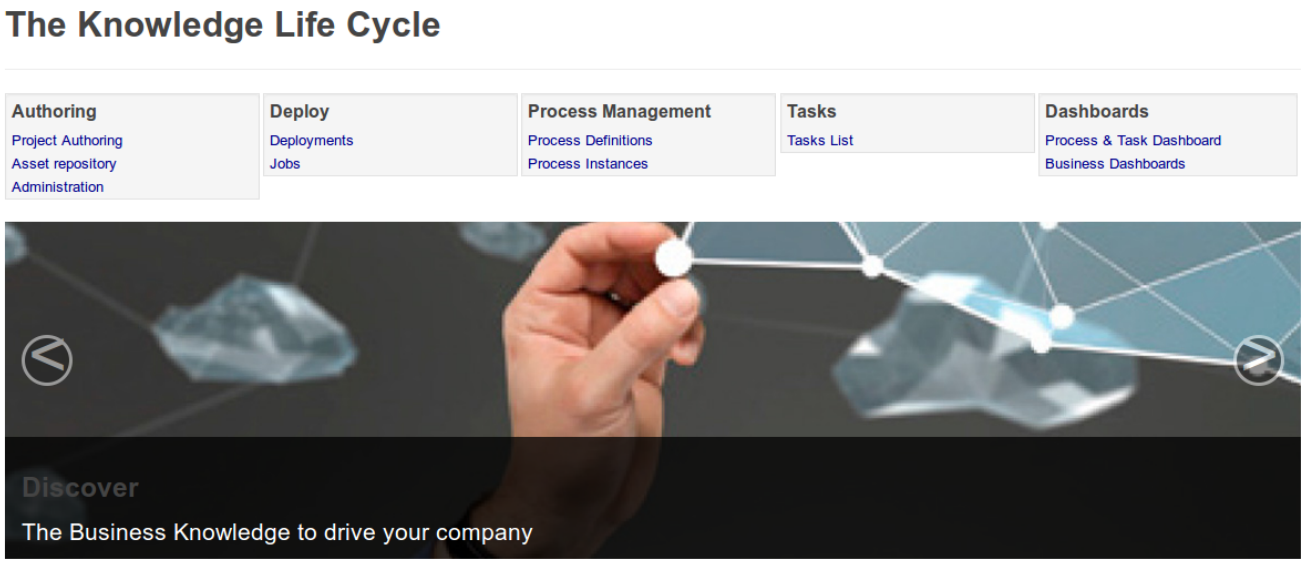
9.5.1. Log in and log out

Create a user with the role `admin` and log in with those credentials.

After successfully logging in, the account username is displayed at the top right. Click on it to review the roles of the current account.

9.5.2. Home screen

After logging in, the home screen shows. The actual content of the home screen depends on the workbench variant (Drools, jBPM, ...).



9.5.3. Workbench concepts

The Workbench is comprised of different logical entities:

- Part

  A Part is a screen or editor with which the user can interact to perform operations.

  Example Parts are "Project Explorer", "Project Editor", "Guided Rule Editor" etc. Parts can be repositioned.

- Panel

  A Panel is a container for one or more Parts.

  Panels can be resized.

- Perspective

A perspective is a logical grouping of related Panels and Parts.

The user can switch between perspectives by clicking on one of the top-level menu items; such as "Home", "Authoring", "Deploy" etc.

9.5.4. Initial layout

The Workbench consists of three main sections to begin; however its layout and content can be changed.

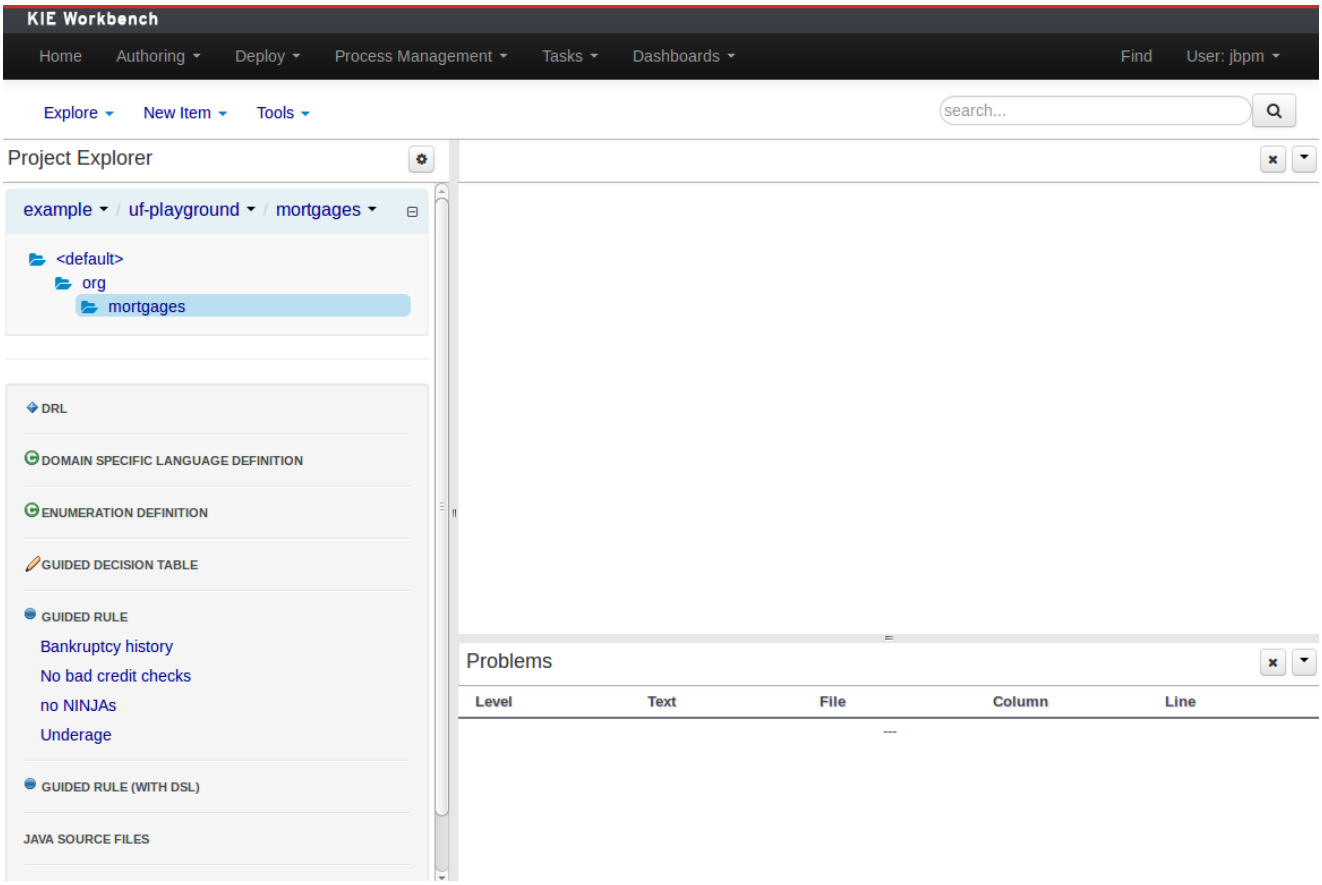


Figure 9.19. The Workbench

The initial Workbench shows the following components:-

- Project Explorer

  This provides the ability for the user to browse their configuration; of Organizational Units (in the above "example" is the Organizational Unit), Repositories (in the above "uf-playground" is the Repository) and Project (in the above "mortgages" is the Project).

- Problems

  This provides the user will real-time feedback about errors in the active Project.

- Empty space

  This empty space will contain an editor for assets selected from the Project Explorer.

  Other screens will also occupy this space by default; such as the Project Editor.

9.6. Changing the layout

The default layout may not be suitable for a user. Panels can therefore be either resized or repositioned.

This, for example, could be useful when running tests; as the test defintion and rule can be repositioned side-by-side.

9.6.1. Resizing

The following screenshot shows a Panel being resized.

Move the mouse pointer over the panel splitter (a grey horizontal or vertical line in between panels).

The cursor will changing indicating it is positioned correctly over the splitter. Press and hold the left mouse button and drag the splitter to the required position; then release the left mouse button.

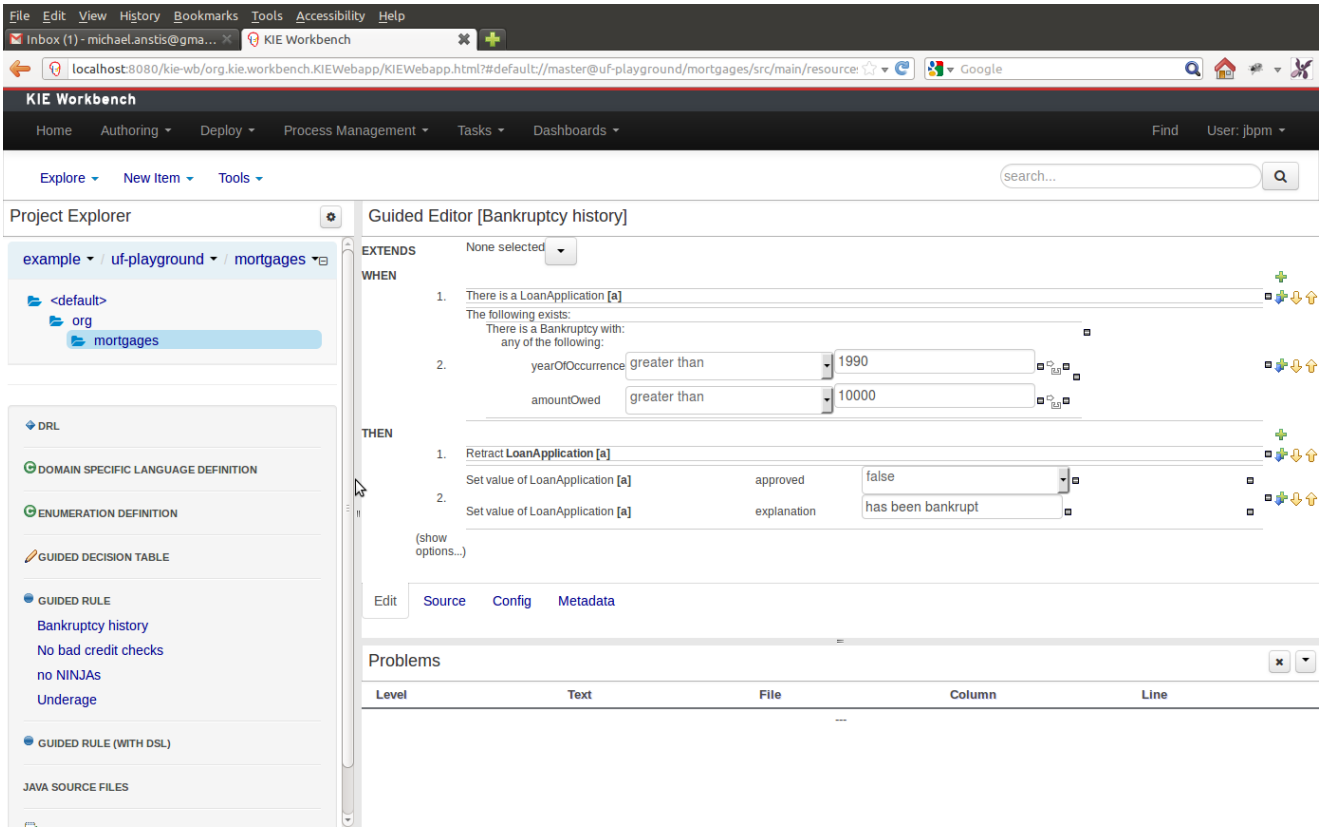


Figure 9.20. Resizing

9.6.2. Repositioning

The following screenshot shows a Panel being repositioned.

Move the mouse pointer over the Panel title ("Guided Editor [No bad credit checks]" in this example).

The cursor will change indicating it is positioned correctly over the Panel title. Press and hold the left mouse button. Drag the mouse to the required location. The target position is indicated with a pale blue rectangle. Different positions can be chosen by hovering the mouse pointer over the different blue arrows.

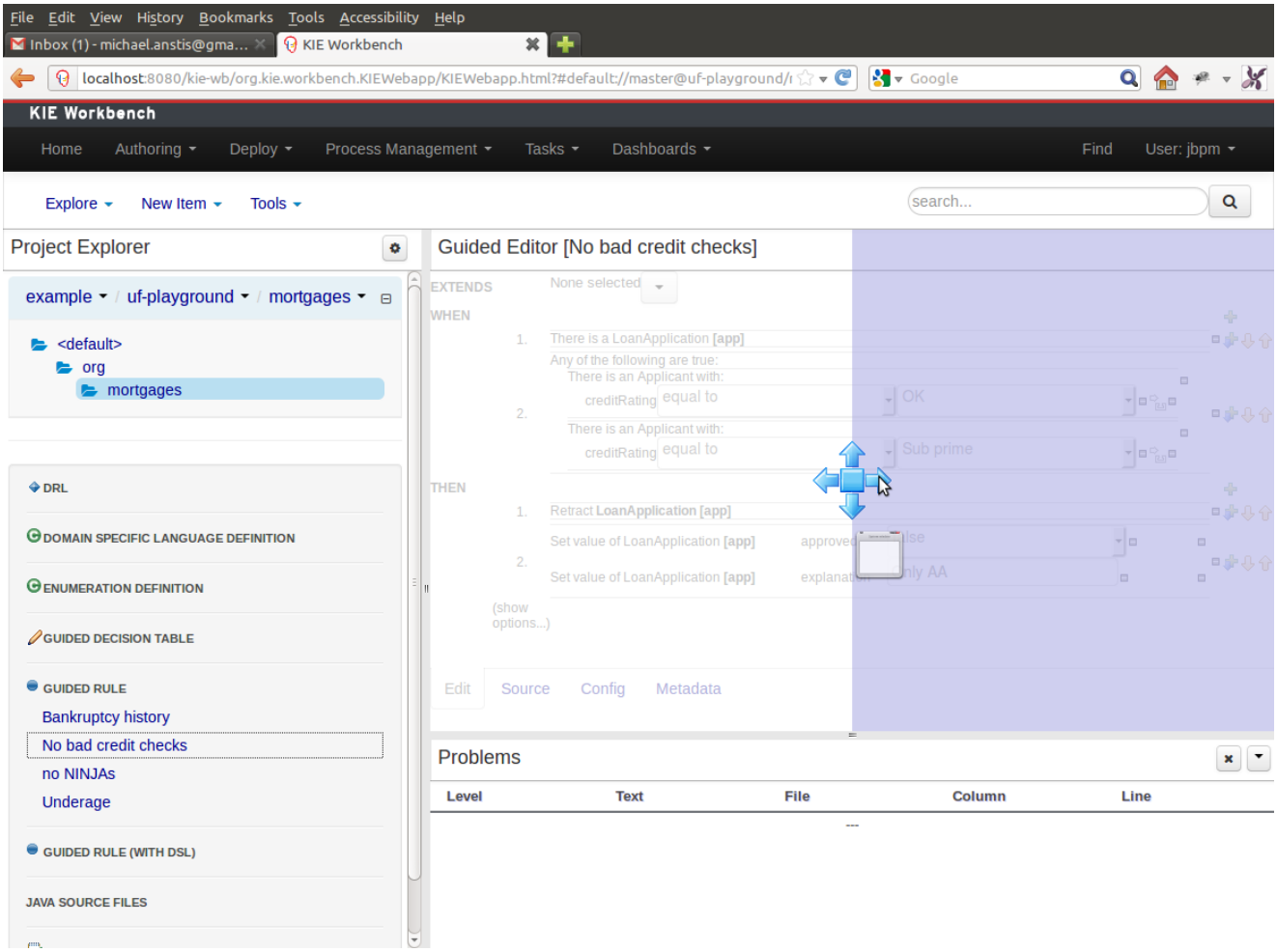


Figure 9.21. Repositioning - dragging

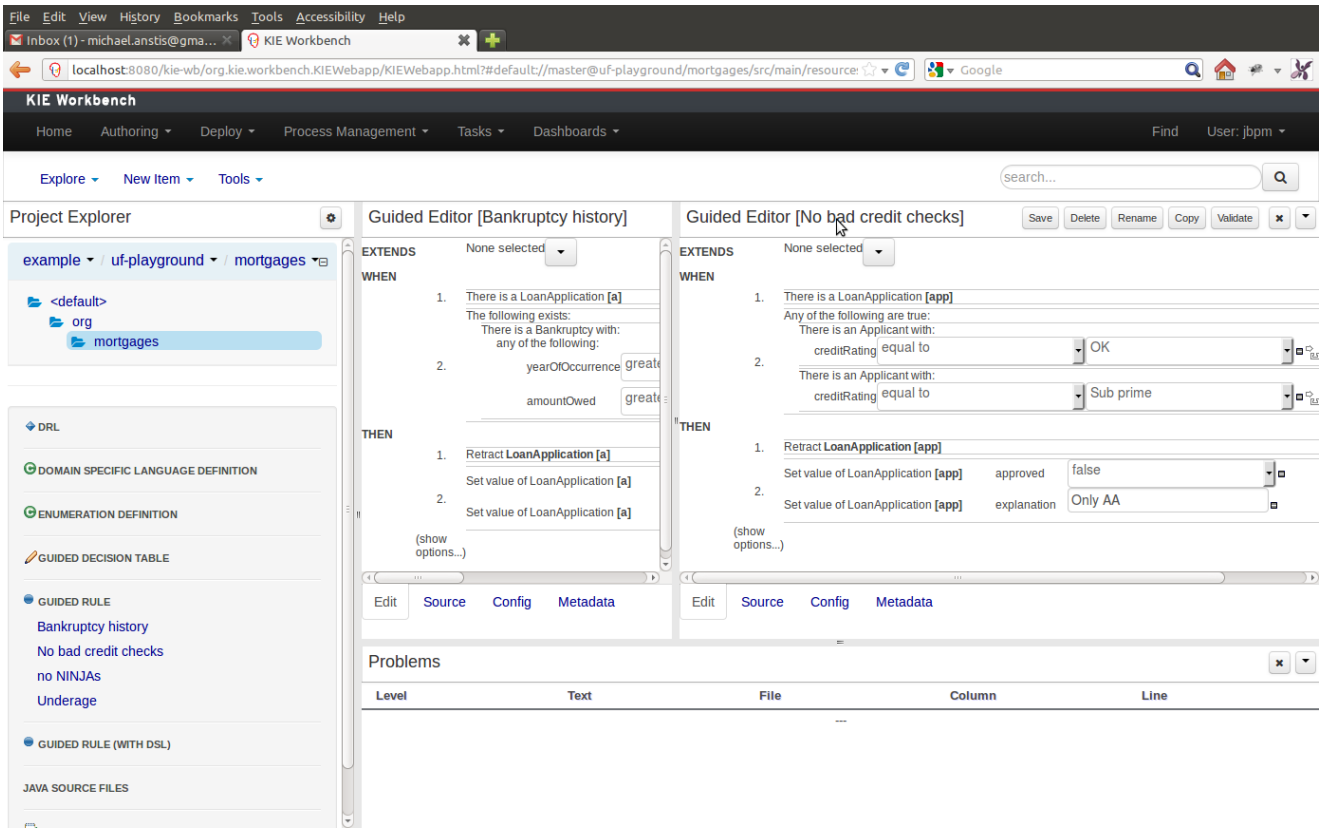


Figure 9.22. Repositioning - complete

9.7. Authoring

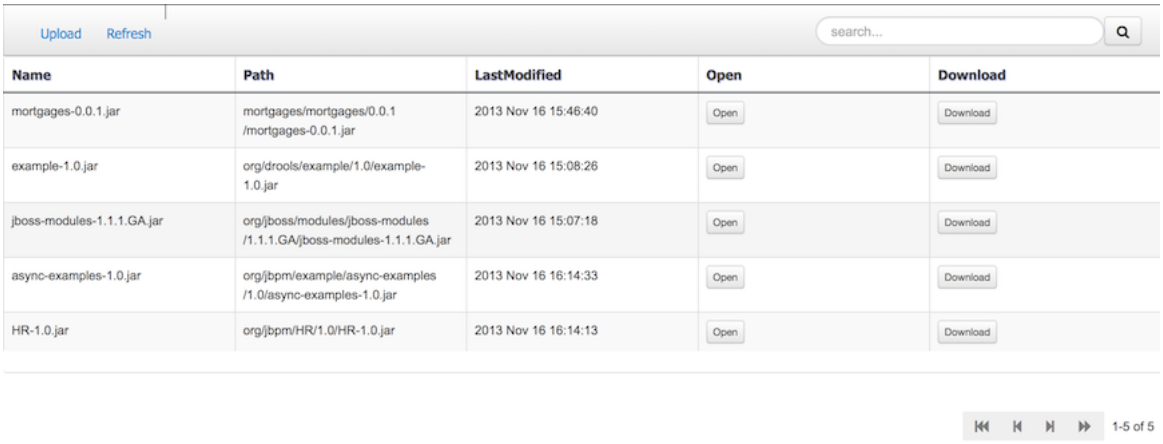
9.7.1. Artifact Repository

Projects often need external artifacts in their classpath in order to build, for example a domain model JARs. The artifact repository holds those artifacts.

The Artifact Repository is a full blown Maven repository. It follows the semantics of a Maven remote repository: all snapshots are timestamped. But it is often stored on the local hard drive.

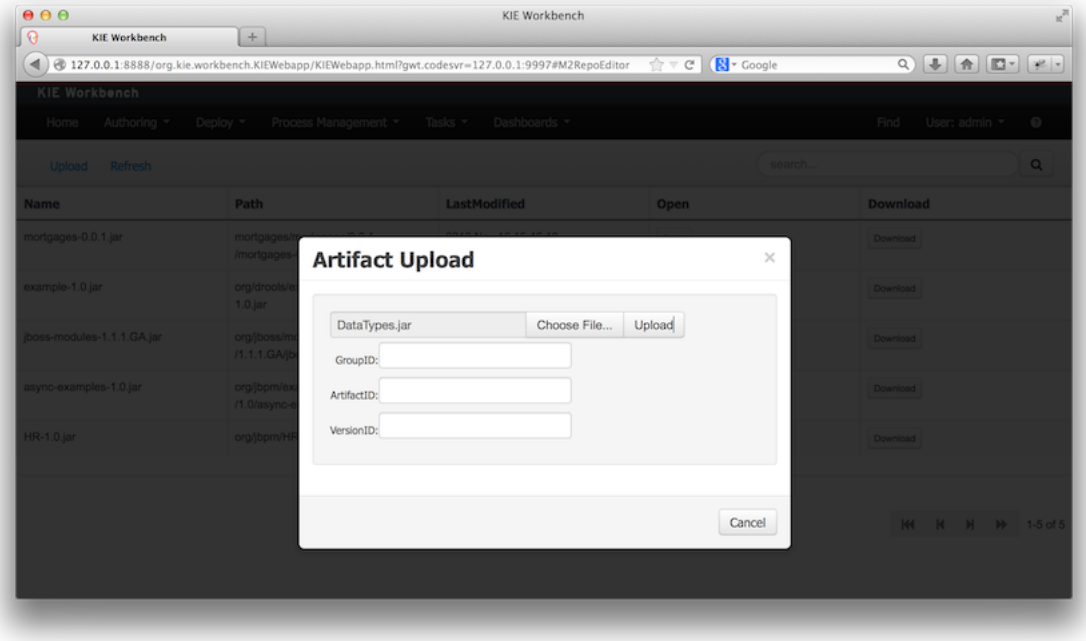
By default the artifact repository is stored under `$WORKING_DIRECTORY/repositories/kie`, but it can be overridden with the *system property* `-Dorg.guvnor.m2repo.dir`. There is only 1 Maven repository per installation.

The Artifact Repository screen shows a list of the artifacts in the Maven repository:



To add a new artifact to that Maven repository, either:

- Use the upload button and select a JAR. If the JAR contains a POM file under `META-INF/maven` (which every JAR build by Maven has), no further information is needed. Otherwise, a groupId, artifactId and version need be given too.



- Using Maven, `mvn deploy` to that Maven repository. Refresh the list to make it show up.

Note

This remote Maven repository is relatively simple. It does not support proxying, mirroring, ... like Nexus or Archiva.

9.7.2. Asset Editor

The Asset Editor is the principle component of Guvnor's User-Interface. It consists of two main views Edit and Metadata.

- The views
  - A : The editing area - exactly what form the editor takes depends on the Asset type.
  - B : This menu bar contains various actions for the Asset; such as Saving, Renaming, Copy etc.
  - C : Different views for asset content or asset information.
    - Edit shows the main editor for the asset
    - Source shows the asset in plain DRL. Note: This tab is only visible if the asset content can be generated into DRL.
    - Config contains the model imports used by the asset.
    - Metadata contains the metadata view for this editor. Explained in more detail below.

Figure 9.23. The Asset Editor - Edit tab

- Metadata
  - A : Meta data (from the "Dublin Core" standard):-

    "Title:" Name of the asset

    "Categories:" A deprecated feature for grouping the assets.

    "Last modified:" The last modified date.

    "By:" Who made the last change.

    "Note:" A comment made when the Asset was last updated (i.e. why a change was made)

    "Created on:" The date and time the Asset was created.

"Created by:" Who initially authored the Asset.

"Format:" The short format name of the type of Asset.

"URI:" URI to the asset inside the Git repository.

- B : Other miscellaneous meta data for the Asset.
- C : Version history of the Asset.
- D : Free-format documentation\description for the Asset. It is encouraged, but not mandatory, to record a description of the Asset before editing.
- E : Discussions regarding development of the Asset can be recorded here.

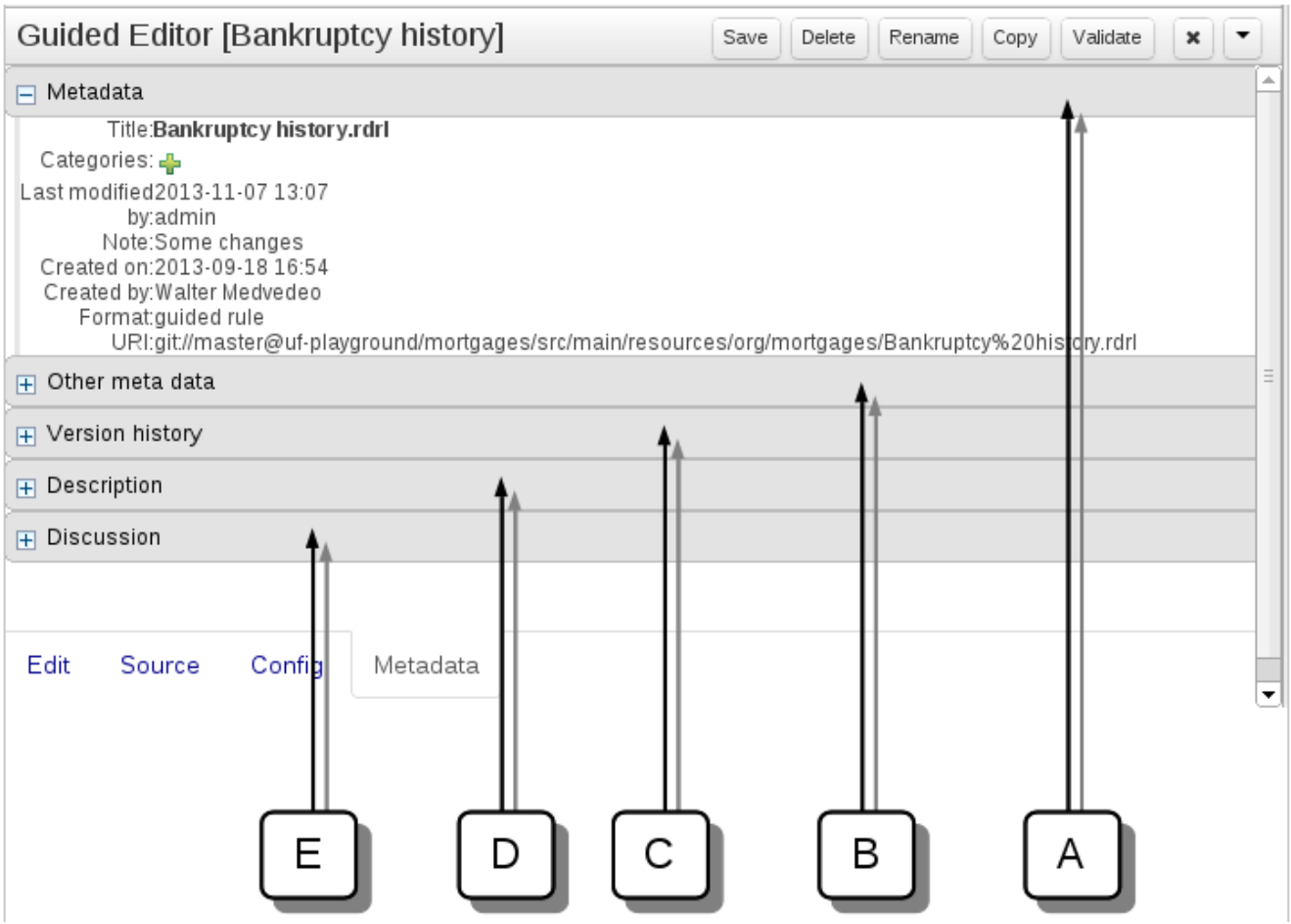


Figure 9.24. The Asset Editor - Attributes tab

Figure 9.25. The Asset Editor - Other meta data

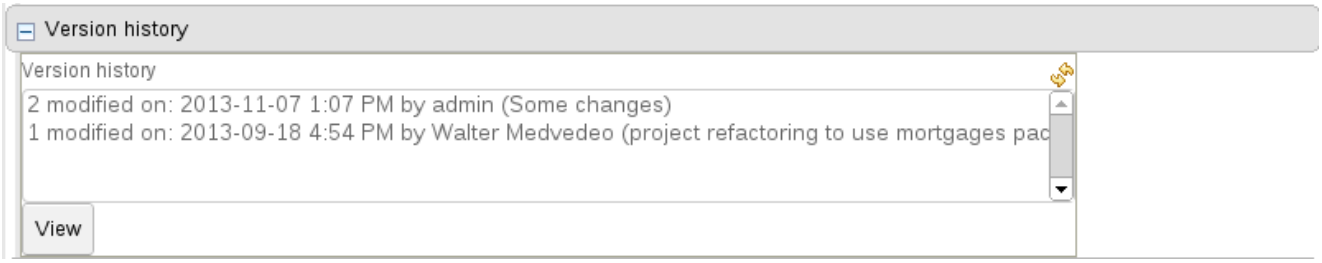


Figure 9.26. The Asset Editor - Version history

Figure 9.27. The Asset Editor - Description

Figure 9.28. The Asset Editor - Discussion

9.7.3. Project Explorer

The Project Explorer provides the ability to browse different Organizational Units, Repositories, Projects and their files.

9.7.3.1. Initial view

The initial view could be empty when first opened.

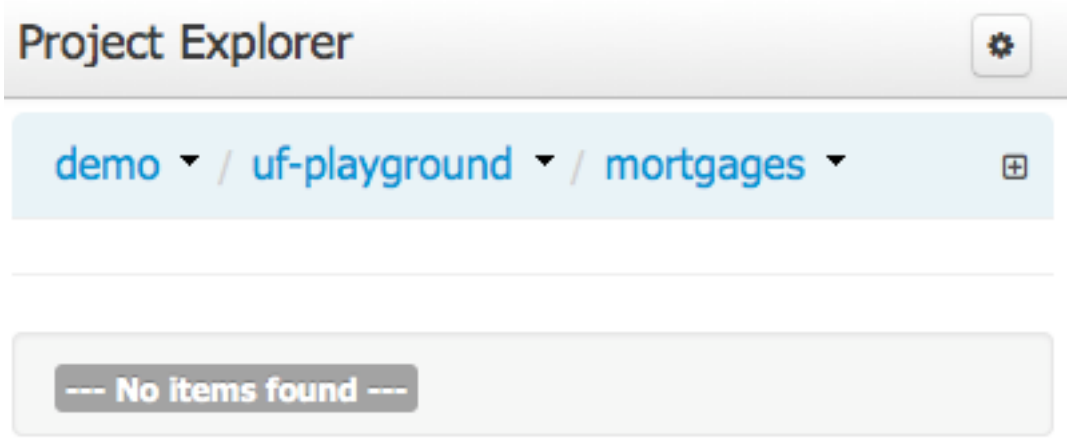


Figure 9.29. An empty initial view

The user may have to select an Organizational Unit, Repository and Project from the drop-down boxes.

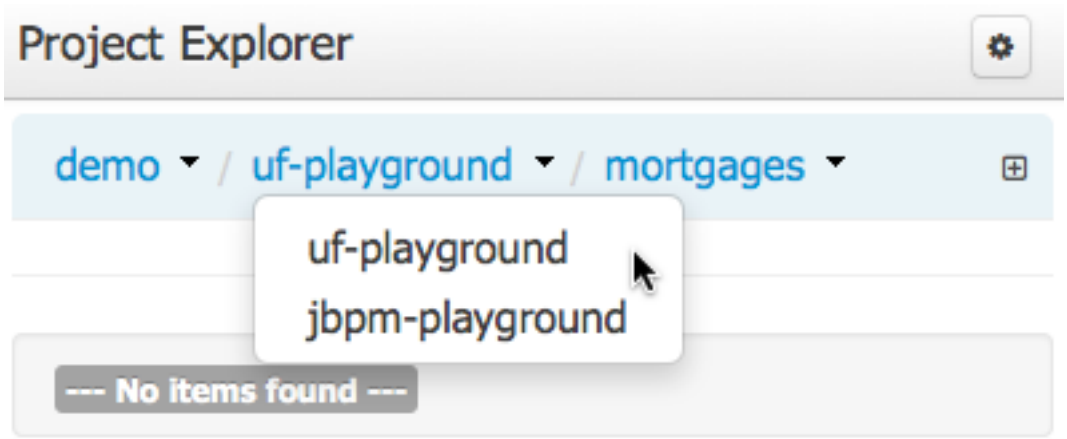


Figure 9.30. Selecting a repository

The default configuration hides Package details from view.

In order to reveal packages click on the icon as indicated in the following screen-shot.

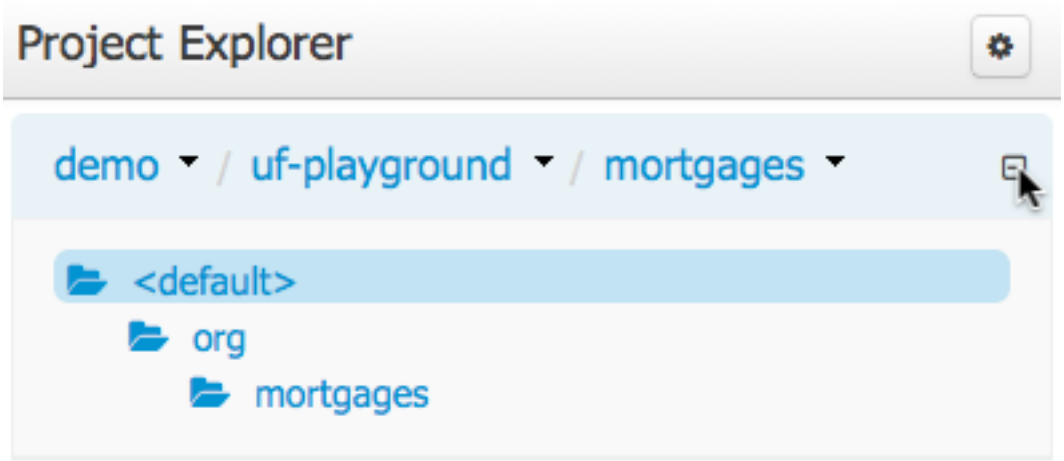


Figure 9.31. Showing packages

After a suitable combination of Organizational Unit, Repository, Project and Package have been selected the Project Explorer will show the contents. The exact combination of selections depends wholly on the structures defined within the Workbench installation and projects. Each section contains groups of related files.

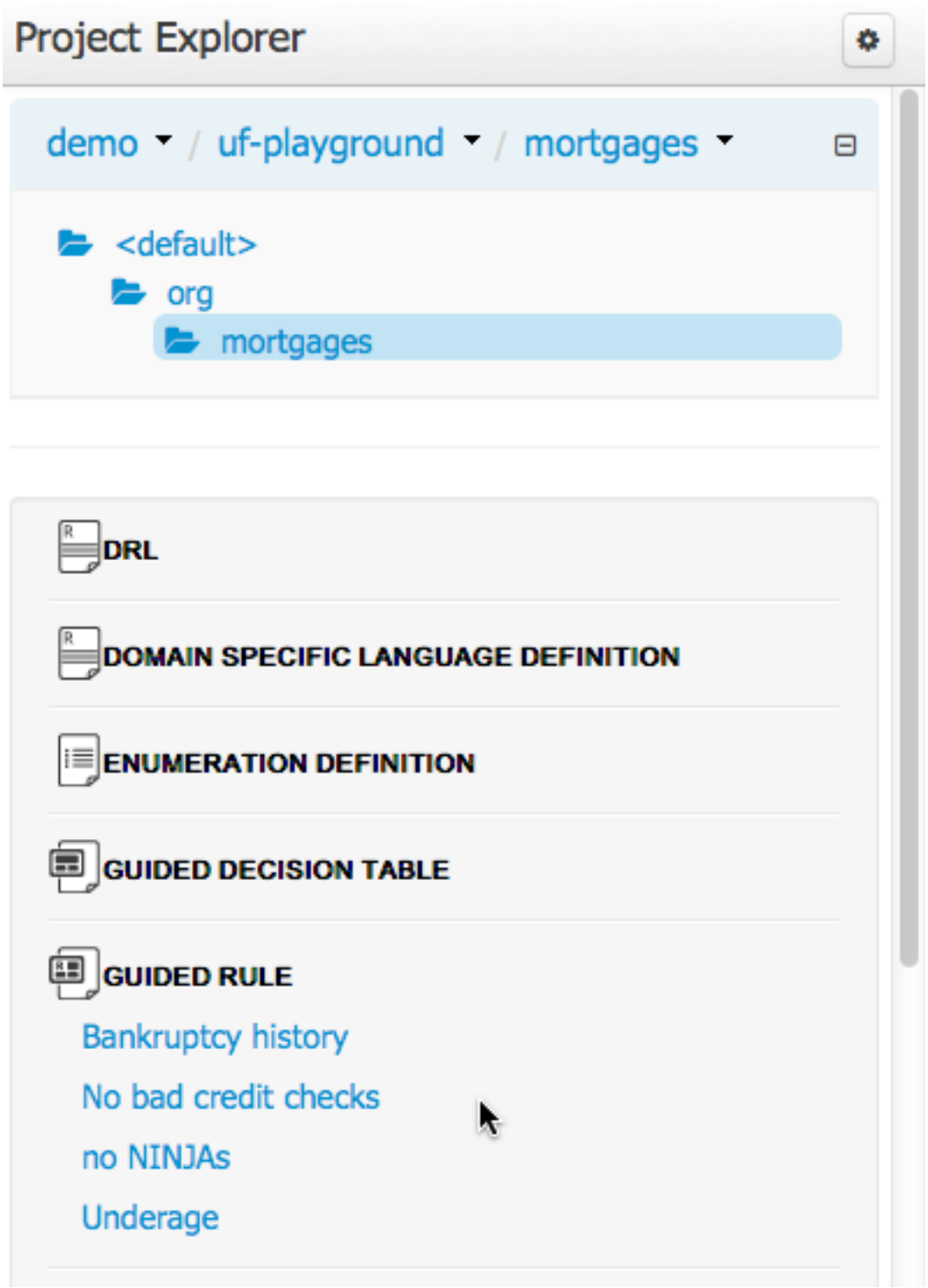


Figure 9.32. Expanded asset group

9.7.3.2. Different views

Project Explorer supports multiple views.

- Project View

A simplified view of the underlying project structure. Certain system files are hidden from view.

- Repository View

  A complete view of the underlying project structure including all files; either user-defined or system generated.

Views can be selected by clicking on the icon within the Project Explorer, as shown below.

Both Project and Repository Views can be further refined by selecting either "Show as Folders" or "Show as Links".

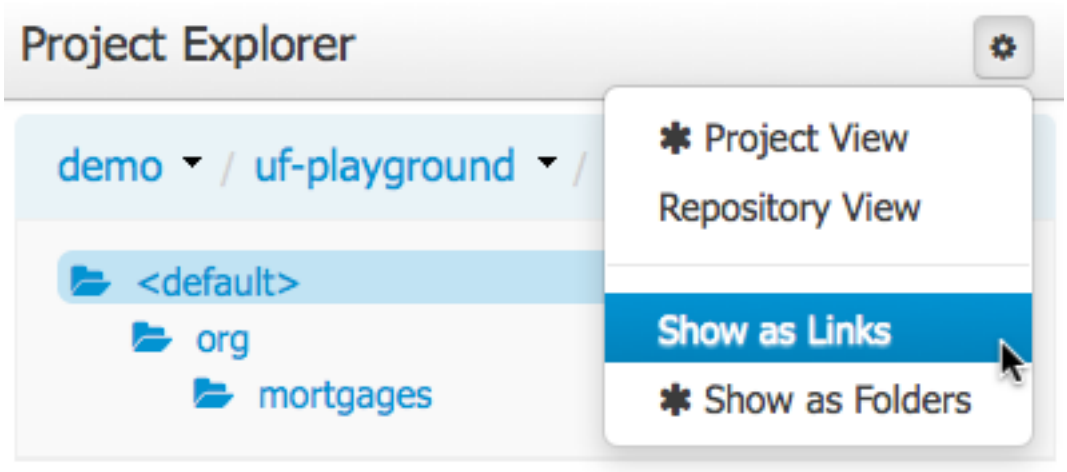


Figure 9.33. Switching view

9.7.3.2.1. Project View examples

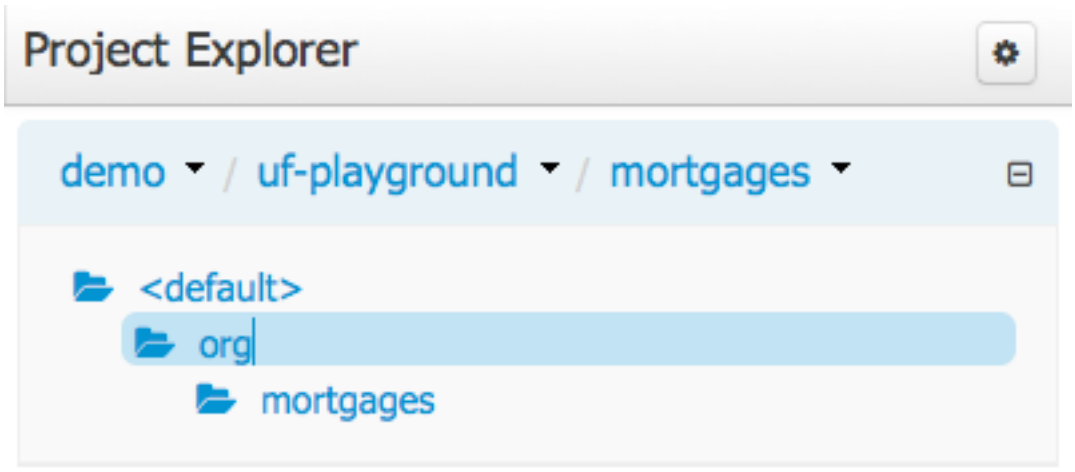


Figure 9.34. Project View - Folders

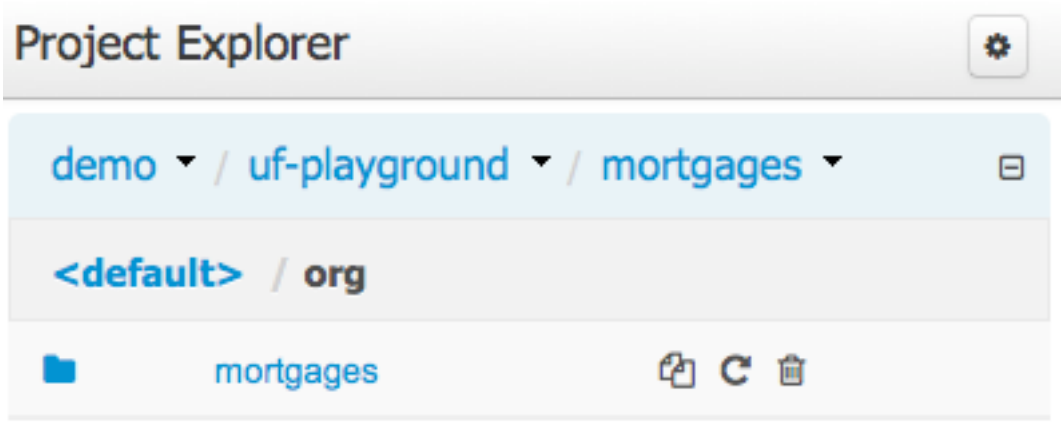


Figure 9.35. Project View - Links

9.7.3.2.2. Repository View examples

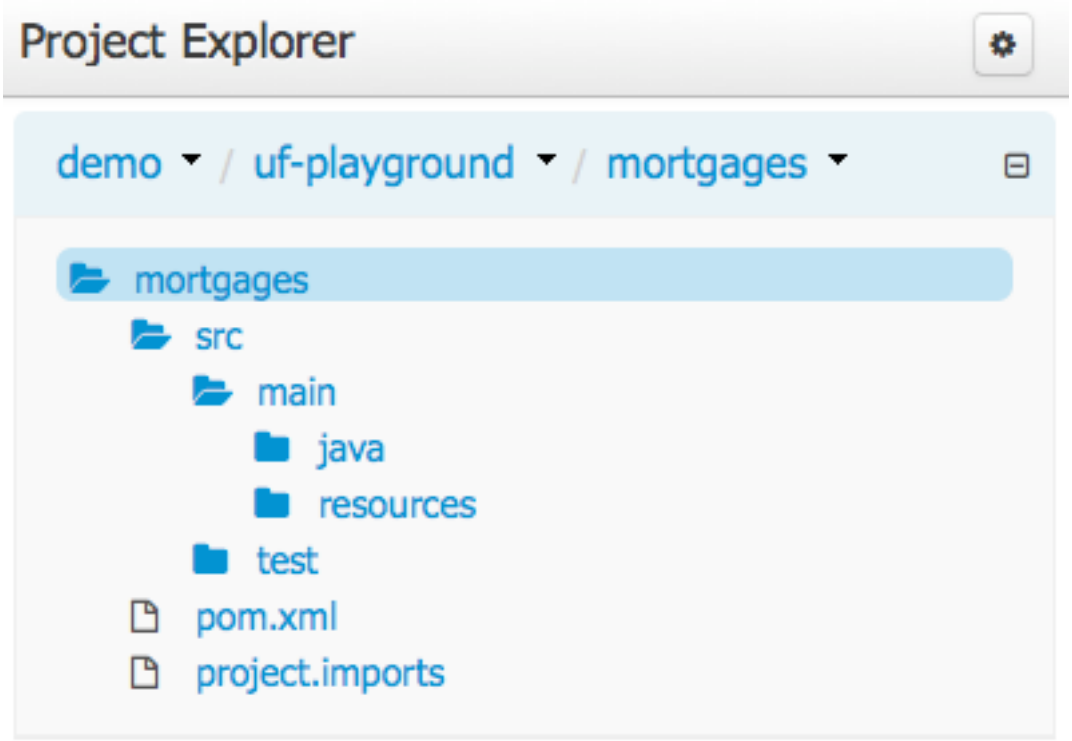


Figure 9.36. Repository View - Folders

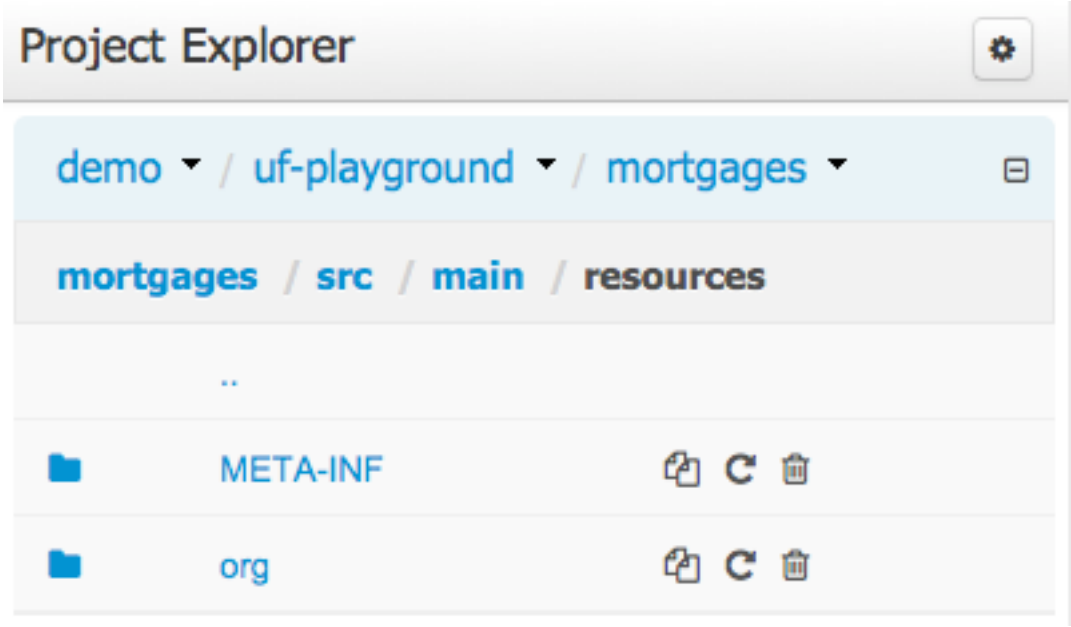


Figure 9.37. Repository View - Links

9.7.3.3. Copy, Rename and Delete Actions

Copy, rename and delete actions are available on *Links* mode, for packages (in of Project View) and for files and directories as well (in Repository View).

- A : Copy
- B : Rename
- C : Delete

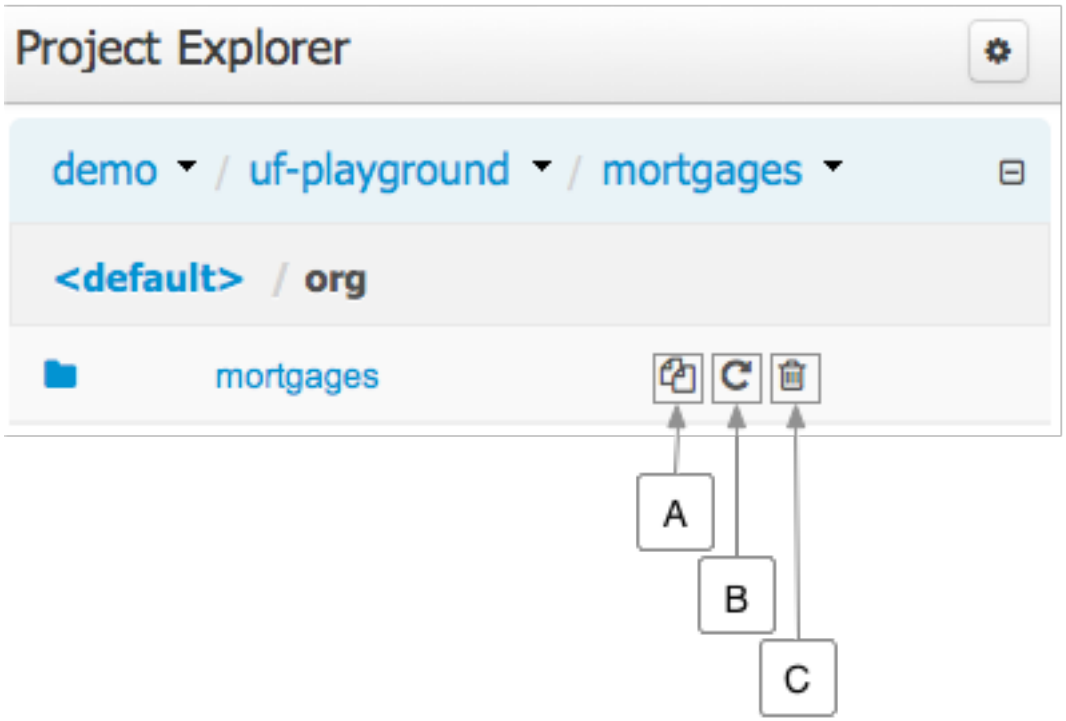


Figure 9.38. Project View - Package actions

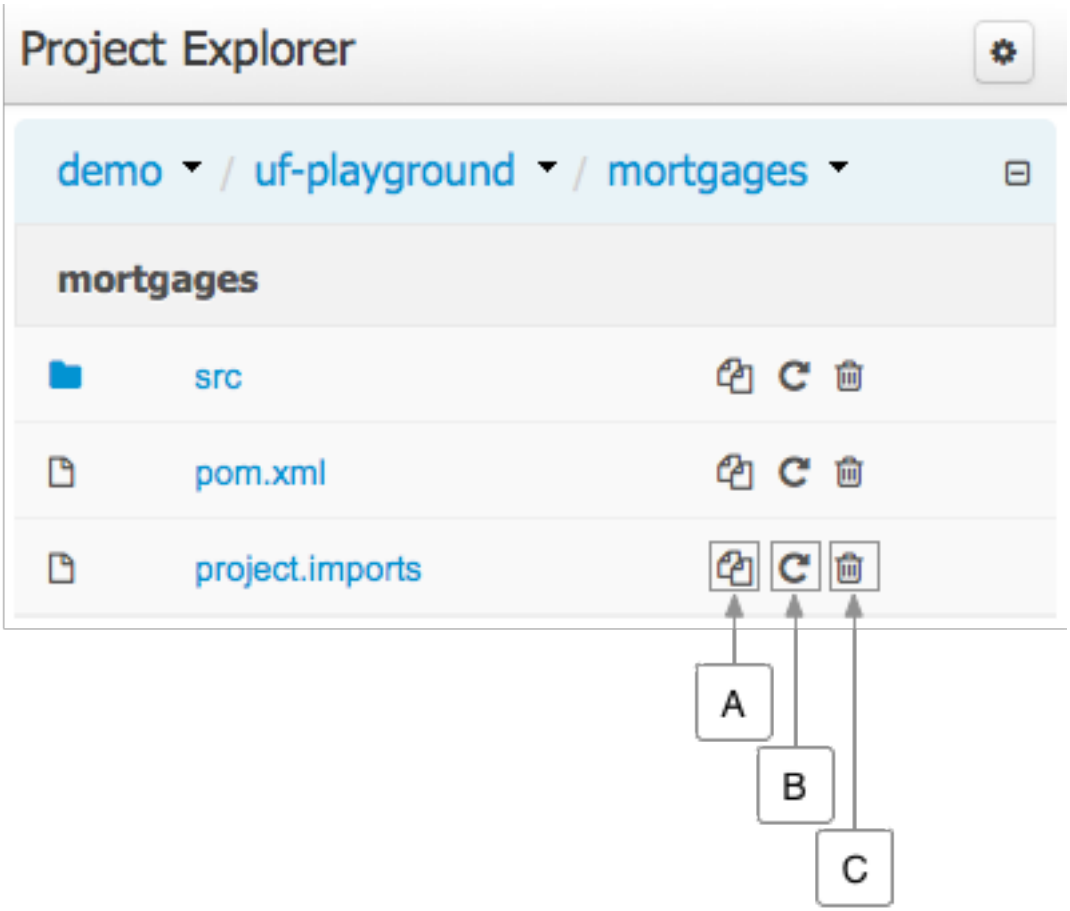


Figure 9.39. Repository View - Files and directories actions

Warning

Workbench roadmap includes a refactoring and an impact analyses tools, but currenctly doesn't have it. Until both tools are provided make sure that your changes (copy/rename/delete) on packages, files or directories doesn't have a major impact on your project.

In cases that your change had an unexcepcted impact, Workbench allows you to restore your repository using the *Repository editor*.

9.7.4. Project Editor

The Project Editor screen can be accessed from the Project menu. Project menu shows the settings for the currently active project.

Unlike most of the workbench editors, project editor edits more than one file. Showing everything that is needed for configuring the KIE project in one place.

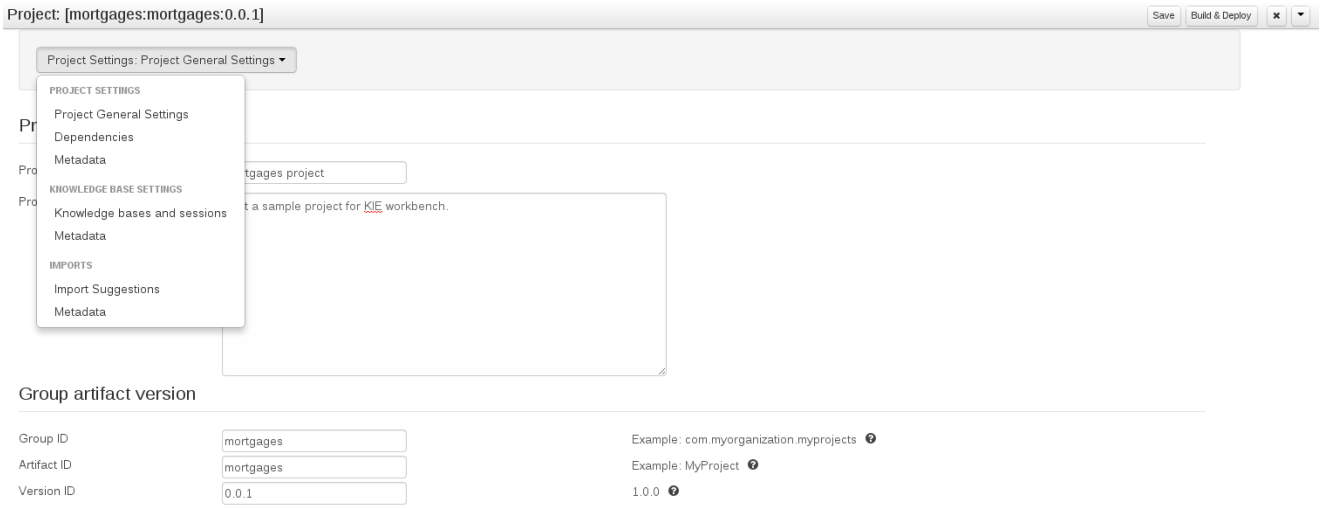


Figure 9.40. Project Screen and the different views

9.7.4.1. Build & Deploy

Build & Depoy builds the current project and deploys the KJAR into the workbench internal Maven repository.

9.7.4.2. Project Settings

Project Settings edits the pom.xml file used by Maven.

9.7.4.2.1. Project General Settings

General settings provide tools for project name and GAV-data (Group, Artifact, Version). GAV values are used as identifiers to differentiate projects and versions of the same project.

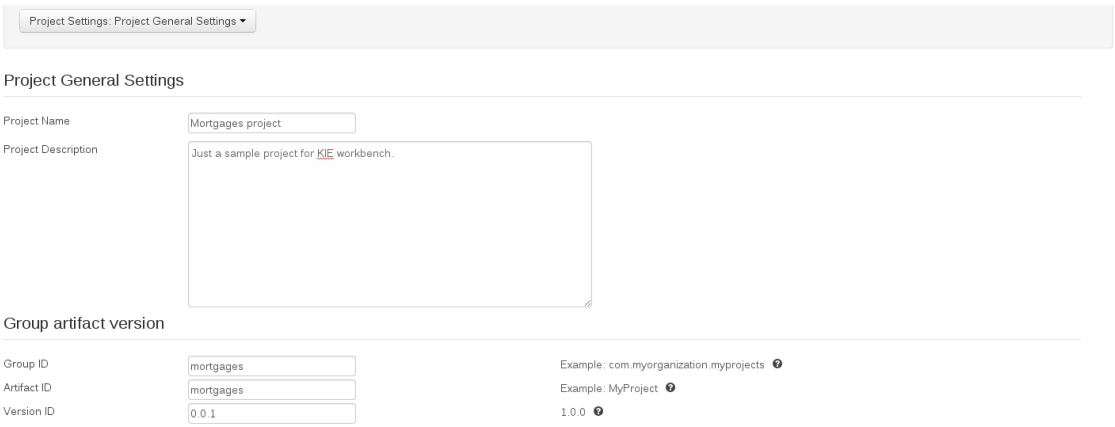


Figure 9.41. Project Settings

9.7.4.2.2. Dependencies

The project may have any number of either internal or external dependencies. Dependency is a project that has been built and deployed to a Maven repository. Internal dependencies are projects build and deployed in the same workbench as the project. External dependencies are retrieved from repositories outside of the current workbench. Each dependency uses the GAV-values to specify the project name and version that is used by the project.

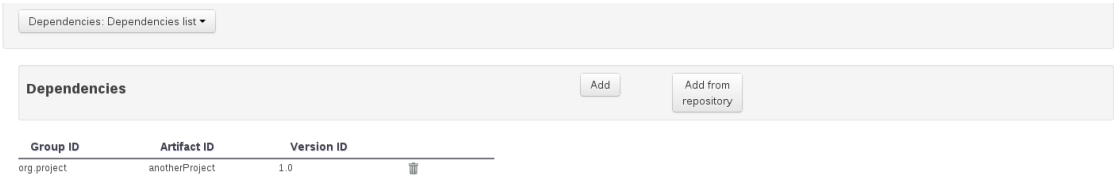


Figure 9.42. Dependencies

9.7.4.2.3. Metadata

Metadata for the pom.xml file.

9.7.4.3. Knowledge Base Settings

Knowledge Base Settings edits the kmodule.xml file used by Drools.

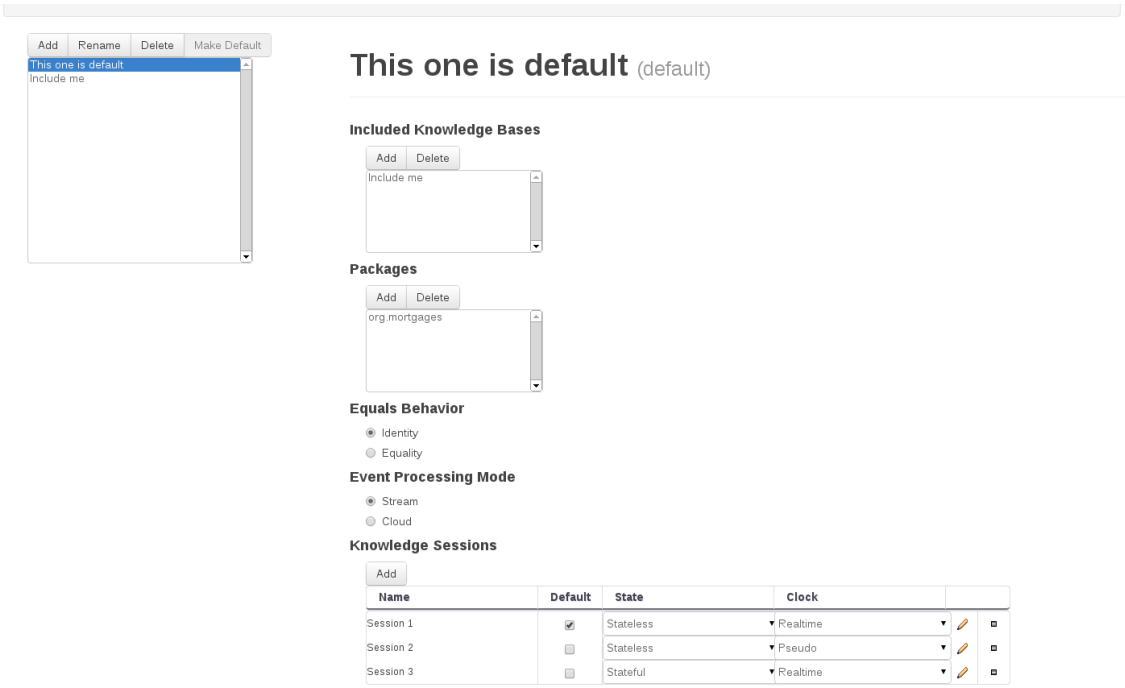


Figure 9.43. Knowledge Base Settings

Note

For more information about the Knowledge Base properties, check the Drools Expert documentation for kmodule.xml.

9.7.4.3.1. Knowledge bases and sessions

Knowledge bases and sessions lists the knowledge bases and the knowledge sessions specified for the project.

9.7.4.3.1.1. Knowledge base list

Lists all the knowledge bases by name. Only one knowledge base can be set as default.

9.7.4.3.1.2. Knowledge base properties

Knowledge base can include other knowledge bases. The models, rules and any other content in the included knowledge base will be visible and usable by the currently selected knowledge base.

Rules and models are stored in packages. The packages property specifies what packages are included into this knowledge base.

Equals behavior is explained in the Drools Expert part of the documentation.

Event processing mode is explained in the Drools Fusion part of the documentation.

9.7.4.3.1.3. Knowledge sessions

The table lists all the knowledge sessions in the selected knowledge base. There can be only one default of each type. The types are stateless and stateful. Clicking the pen-icon opens a popup that shows more properties for the knowledge session.

9.7.4.3.2. Metadata

Metadata for the kmodule.xml

9.7.4.4. Imports

Settings edits the project.imports file used by the workbench editors.

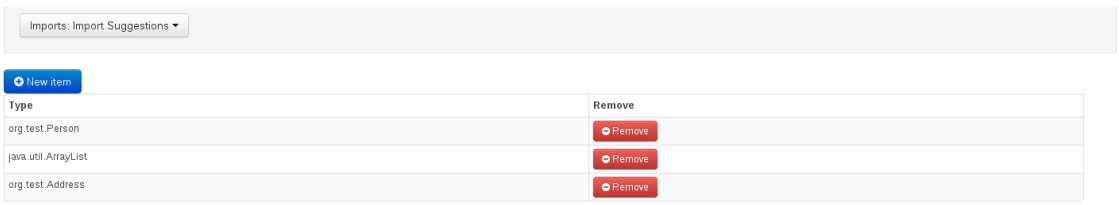


Figure 9.44. Imports

9.7.4.4.1. Import Suggestions

Import Suggestions lists imports that are used as suggestions when using the guided editors the workbench has. Making it easier to work with the workbench, as there is no need to type each import in each file that uses the import.

Note

Unlike in the previous version of Guvnor. The imports listed in the import suggestions are not automatically added into the knowledge base or into the packages of the workbench. Each import needs to be explicitly added into each file.

9.7.4.4.2. Metadata

Metadata for the project.imports file.

9.7.5. Validation

The Workbench provides a common and consistent service for users to understand whether files authored within the environment are valid.

9.7.5.1. Problem Panel

The Problems Panel shows real-time validation results of assets within a Project.

When a Project is selected from the Project Explorer the Problems Panel will refresh with validation results of the chosen Project.

When files are created, saved or deleted the Problems Panel content will update to show either new validation errors, or remove existing if a file was deleted.

Here an invalid DRL file has been created and saved.

The Problems Panel shows the validation errors.

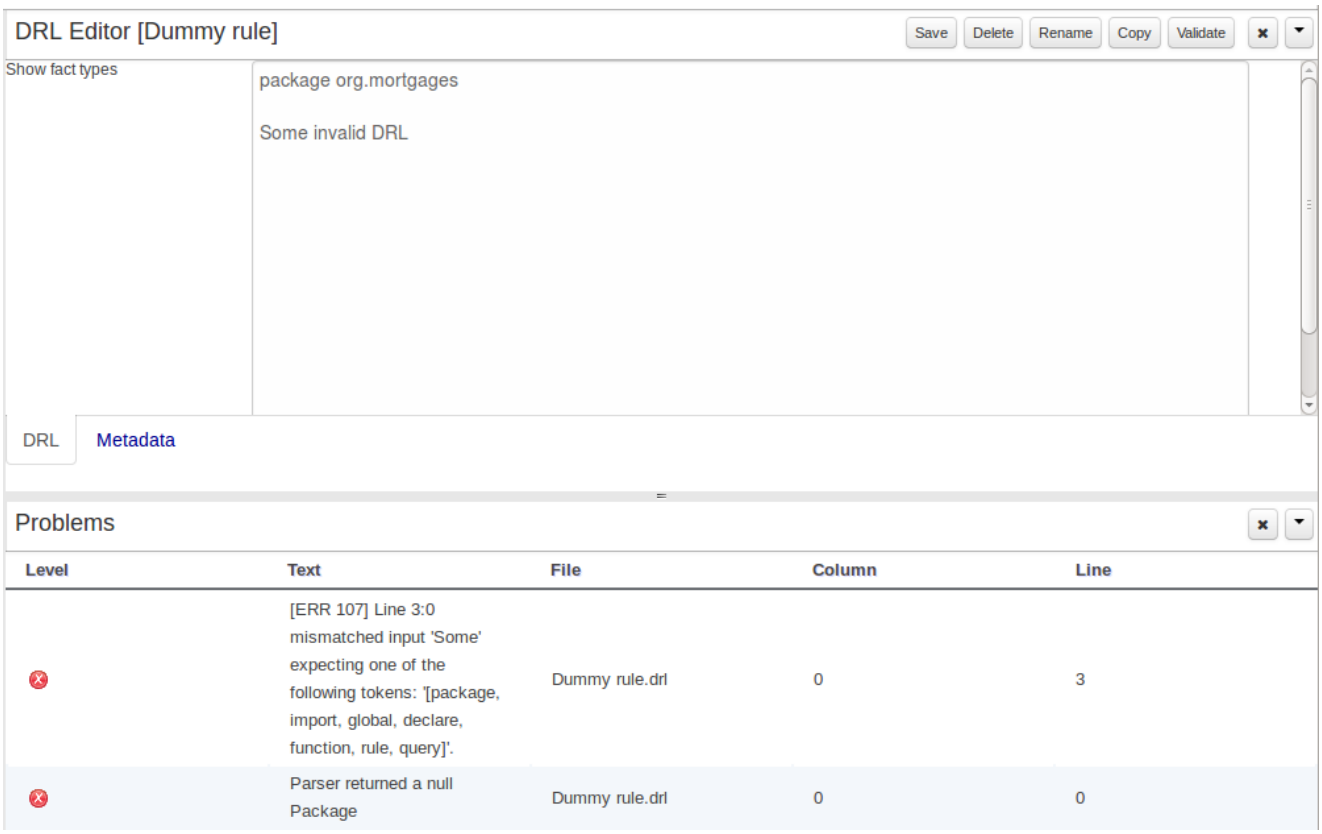


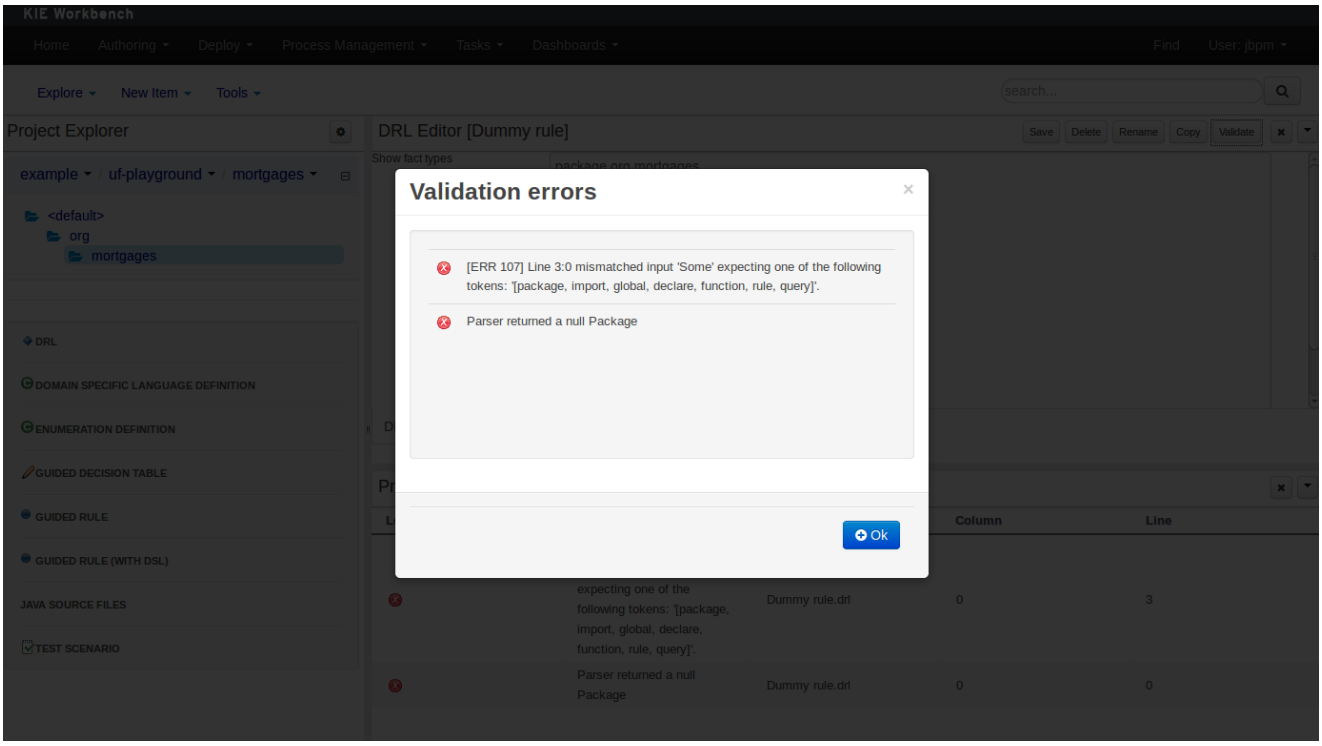
Figure 9.45. The Problems Panel

9.7.5.2. On demand validation

It is not always desirable to save a file in order to determine whether it is in a valid state.

All of the file editors provide the ability to validate the content before it is saved.

Clicking on the 'Validate' button shows validation errors, if any.



9.7.6. Data Modeller

9.7.6.1. First steps to create a data model

By default, a data model is always constrained to the context of a project. For the purpose of this tutorial, we will assume that a correctly configured project already exists.

To start the creation of a data model inside a project, take the following steps:

1. From the home panel, select the authoring perspective

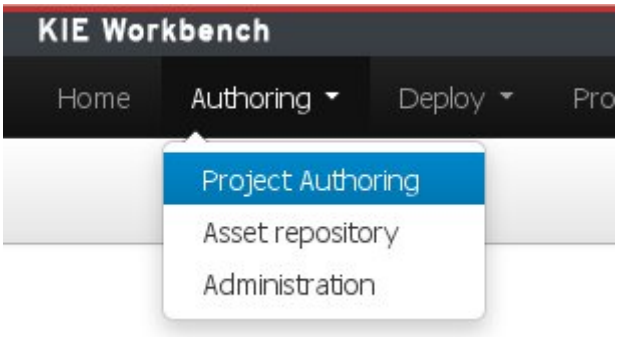


Figure 9.46. Go to authoring perspective

2. If not open already, start the Project Explorer panel

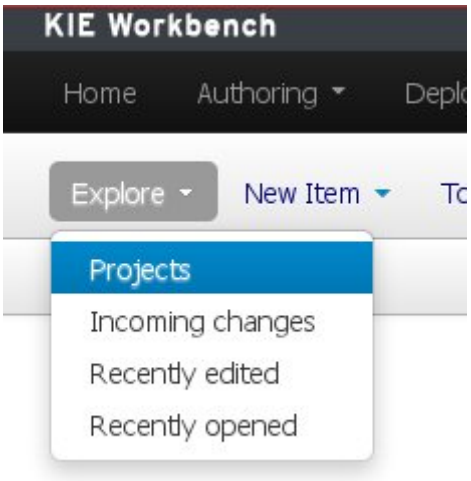


Figure 9.47. Open project explorer panel

3. From Project Explorer panel (the "Business" tab), select the organizational unit, repository, and the project the data model has to be created for. For this tutorial's example, the values "Tutorial", "Examples", and "Purchases" were respectively chosen

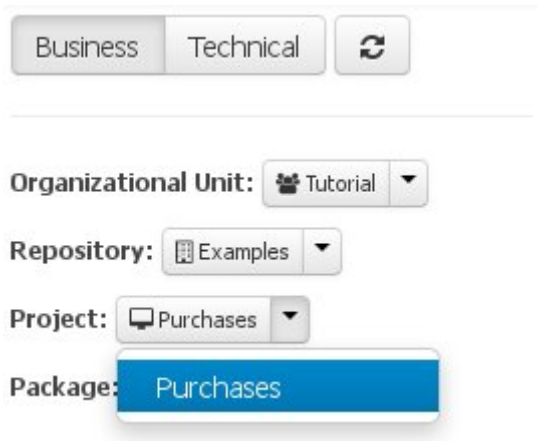


Figure 9.48. Choose project

4. Open the Data Modeller tool by clicking on the "Tools" authoring-menu entry, and selecting the "Data Modeller" option from the drop-down menu

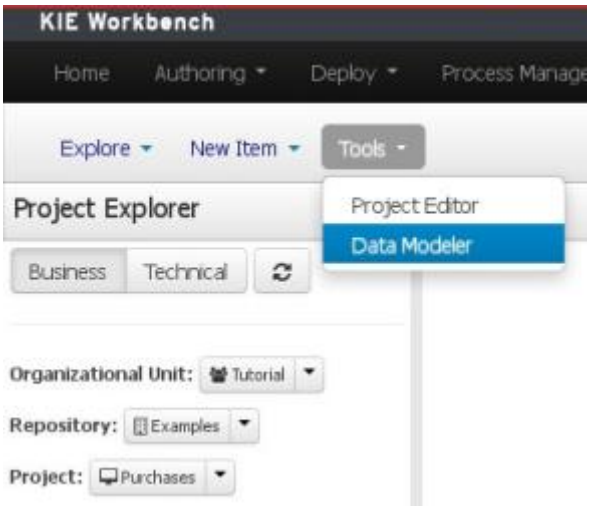


Figure 9.49. Open data modeller

This will start up the Data Modeller tool, which has the following general aspect:

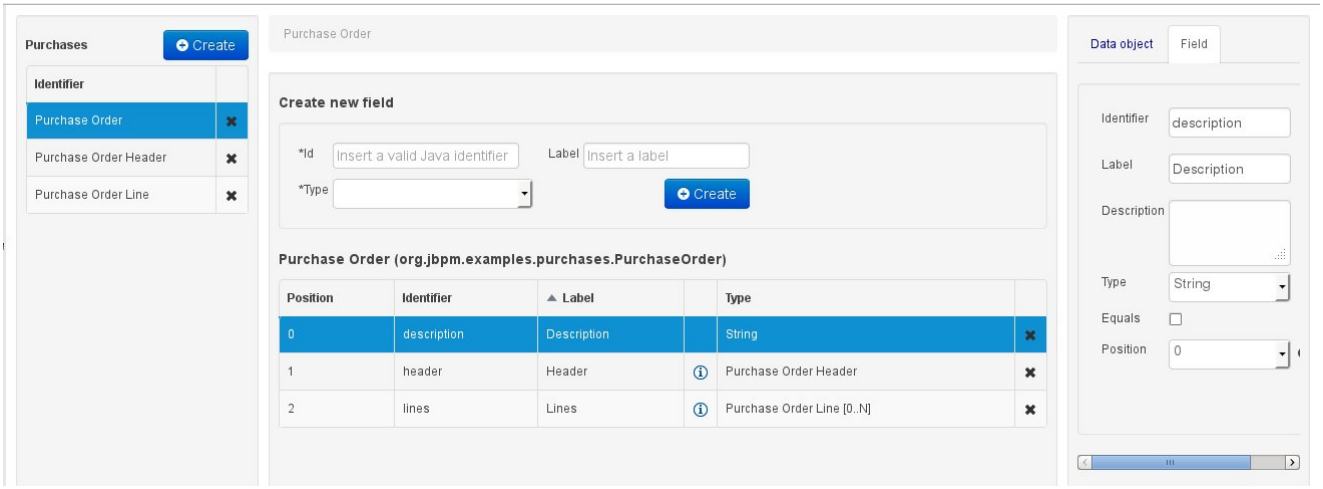


Figure 9.50. Data modeller overview

The Data Modeller panel is divided into the following sections:

- The leftmost "model browser" section, which shows a list of already existing data entities (if any are present, as in this example's case). Above the list the project's name and a button for new object creation are shown. Note that as soon as any changes are applied to the project, an '*' will be appended to the project's name to notify the user of the existence of non-persisted changes.

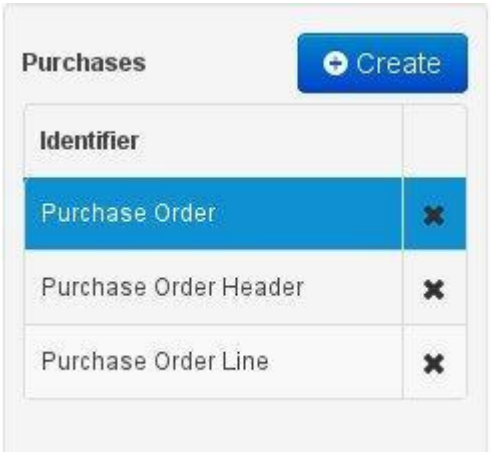


Figure 9.51. The data model browser

- The central section consists of three distinct parts:

  At the top, the "bread crumb widget": this is a navigational aid, which allows navigating back and forth through the data model, when accessing properties that themselves are model entities. The bread crumb trail shown in the image indicates that the object browser is currently visualizing the properties of an entity called "Purchase Order Line", which we accessed through another entity ("Purchase Order"), where it is defined as a field.

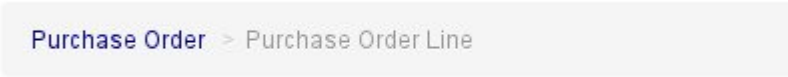


Figure 9.52. The bread crumb

the section beneath the bread crumb widget, is dedicated to the creation of new fields.

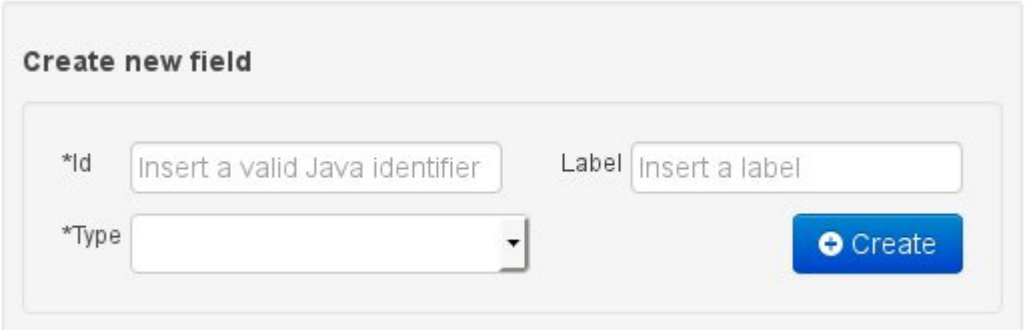


Figure 9.53. New field creation

the bottom section comprises the Entity's "field browser", which displays a list of the currently selected data object's (in the model browser) fields.

Figure 9.54. The entity field browser

- The "entity / field property editor". This is the rightmost section of the Data Modeller screen which visualizes a tabbed pane. The Data object tab allows the user to edit the properties of the currently selected entity in the model browser, whilst the Field tab enables edition of the properties of any of the currently selected object's fields.

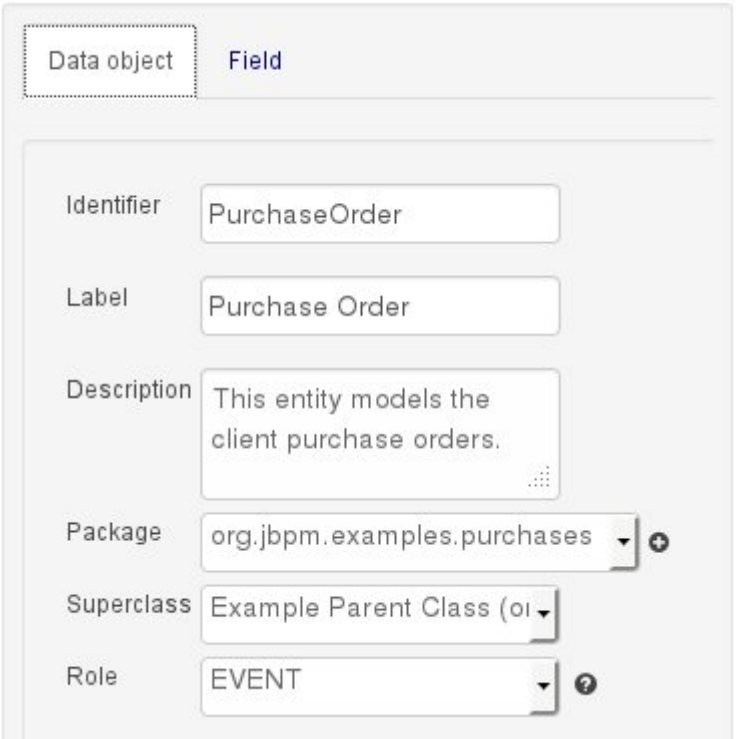


Figure 9.55. The entity/field property editor

9.7.6.2. Entities

A data model consists of data entities which are a logical representation of some real-world data. Such data entities have a fixed set of modeller (or application-owned) properties, such as its

internal identifier, a label, description, package etc. Besides those, an entity also has a variable set of user-defined fields, which are an abstraction of a real-world property of the type of data that this logical entity represents.

Creating a data entity can be achieved either by clicking the "Create" button in the model browser section (see fig. "The data model browser" above), or by clicking the one in the top data modeller menu:

Figure 9.56. Starting creation of an entity from the top menu

This will pop up the new object screen:

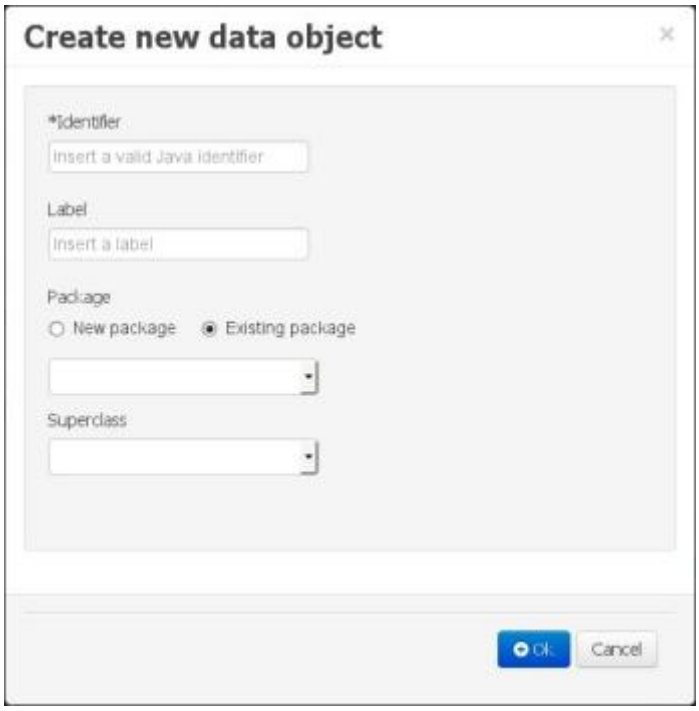


Figure 9.57. The new entity pop up screen

Some initial information needs to be provided before creating the new object:

- The object's internal identifier (mandatory). The value of this field must be unique per package, i.e. if the object's proposed identifier already exists in the selected package, an error message will be displayed.

- A label (optional): this field allows the user to define a user-friendly label for the data entity about to be created. This is purely conceptual info that has no further influence on how objects of this entity will be treated. If a label is defined, then this is how the entity will be displayed throughout the data modeller tool.
- A package (mandatory): a data entity must always be created within a package (or name space, in which this entity will be unique at a platform level). By default, the option for selecting an already existing package will be activated, in which case the corresponding drop-down shows all the packages that are currently defined. If a new package needs to be defined for this entity, then the "New package" option should be selected. In this case the new to be created package should be input into the corresponding text-field. The format for defining new packages is the same as the one for standard Java packages.
- A superclass (optional): this will indicate that this entity extends from another already existing one. Since the data modeller entities are translated into standard Java classes, indicating a superclass implies normal Java object extension at the generated-code level.

Once the user has provided at least the mandatory information, by pushing the "Ok" button at the bottom of the screen the new data entity will be created. It will be added to the model browser's entity listing.

It will also appear automatically selected, to make it easy for the user to complete the definition of the newly created entity, by completing the entity's properties in the Data Object Properties browser, or by adding new fields.

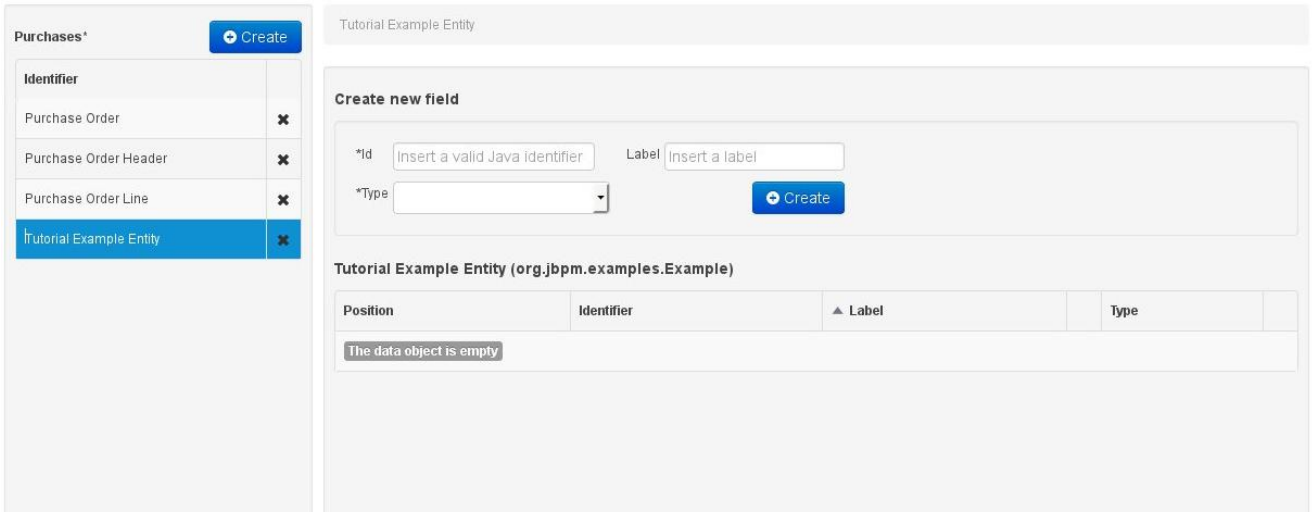


Figure 9.58. New entity has been created

Note

As can be seen in the above figure, after performing changes to the data model, the model name will appear with an '*' to alert the user of the existence of un-persisted changes to the model.

In the Data Modeller's object browsing section, an entity can be deleted by clicking upon the 'x' icon to the right of each entity. If an entity is being referenced from within another entity (as a field type), then the modeller tool will not allow it to be deleted, and an error message will appear on the screen.

9.7.6.3. Properties & relationships

Once the data entity has been created, it now has to be completed by adding user-defined properties to its definition. This can be achieved by providing the required information in the "Create new field" section (see fig. "New field creation"), and clicking on the "Create" button when finished. The following fields can (or must) be filled out:

- The field's internal identifier (mandatory). The value of this field must be unique per data entity, i.e. if the proposed identifier already exists within current entity, an error message will be displayed.
- A label (optional): as with the entity definition, the user can define a user-friendly label for the data entity field which is about to be created. This has no further implications on how fields from objects of this entity will be treated. If a label is defined, then this is how the field will be displayed throughout the data modeller tool.
- A field type (mandatory): each entity field needs to be assigned with a type.

  This type can be either of the following:

  1. A 'primitive' type: these include most of the object equivalents of the standard Java primitive types, such as Boolean, Short, Float, etc, as well as String, Date, BigDecimal and BigInteger.

Figure 9.59. Primitive field types

  2. An 'entity' type: any user defined entity automatically becomes a candidate to be defined as a field type of another entity, thus enabling the creation of relationships between entities. As

can be observed in the above figure, our recently defined 'Tutorial Example Entity' already appears in the types list and can be used as a field type, even for a field of itself. An entity type field can be created either in 'single' or in 'multiple' form, the latter implying that the field will be defined as a collection of this type, which will be indicated by the extension '[0..N]' in the type drop-down or in the entity fields table (as can be seen for the 'Lines' field of the 'Purchase Order' entity, for example).

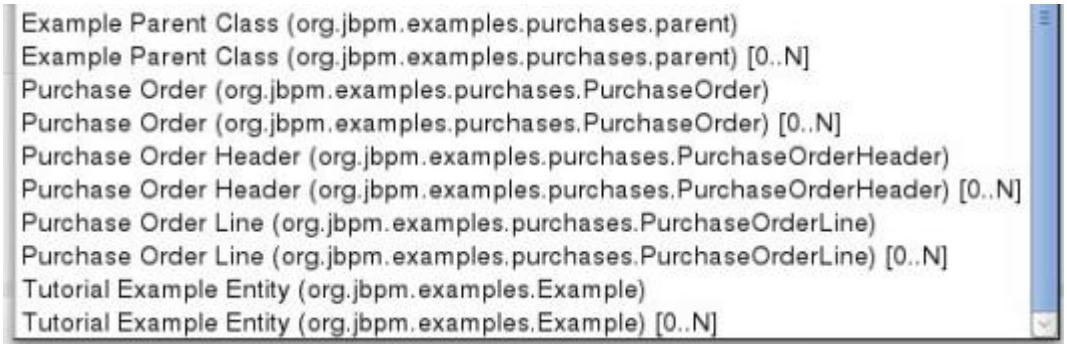


Figure 9.60. Entity field types

When finished introducing the initial information for a new field, clicking the 'Create' button will add the newly created field to the end of the entity's fields table below:

Figure 9.61. New field has been created

The new field will also automatically be selected in the entity's field list, and its properties will be shown in the Field tab of the Property editor. The latter facilitates completion of some additional properties of the new field by the user (see below).

At any time, any field (without restrictions) can be deleted from an entity definition by clicking on the corresponding 'x' icon in the entity's fields table.

9.7.6.4. Additional options

As stated before, both entities as well as entity fields require some of their initial properties to be set upon creation. These are by no means the only properties entities and fields have. Below we will give a detailed description of the additional entity and field properties.

9.7.6.4.1. Additional entity properties ("Data object tab")

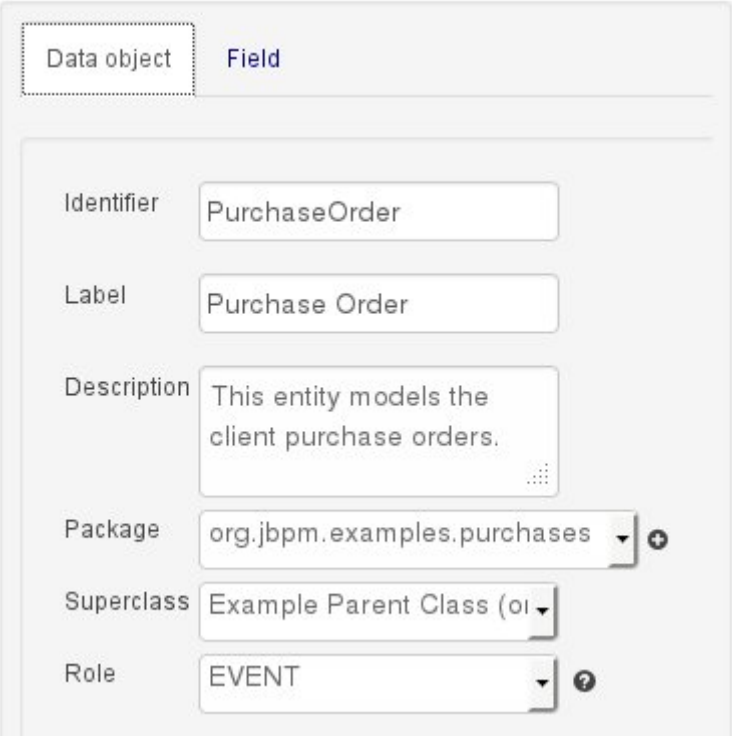


Figure 9.62. The entity's properties

- Description: this field allows the user to introduce some kind of description for the current entity, for documentation purposes only. As with the label property, this is conceptual information that will not influence the use or treatment of this entity or its instances in any way.
- Role: this property allows the assignment of a Role to the entity. The Role is a concept inherited from Drools Fusion, which for the time being only allows one possible value ("Event"). An entity that is designated with this value will be treated by the rules engine as an event type Fact (See Drools Fusion for more information on this matter).

9.7.6.4.2. Additional field properties ("Field tab")

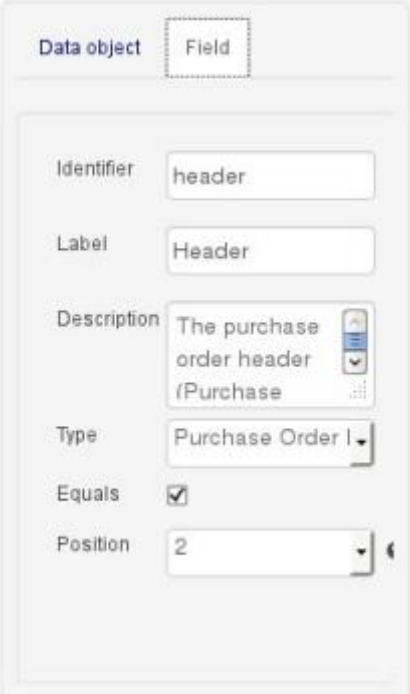


Figure 9.63. The entity's field properties

- Description: this field allows the user to introduce some kind of description for the current field, for documentation purposes only. As with the label property, this is conceptual information that will not influence the use or treatment of this entity or its instances in any way.
- Equals: checking this property for an entity field implies that it will be taken into account, at the code generation level, for the creation of both the equals() and hashCode() methods in the generated Java class. We will explain this in more detail in the following section.
- Position: this field requires a zero or positive integer. When set, this field will be interpreted by the Drools engine as a positional argument (see the section below and also the Drools documentation for more information on this subject).

9.7.6.5. Generate data model code.

The data model in itself is merely a visual tool that allows the user to define high-level data structures, for them to interact with the Drools Engine on the one hand, and the jBPM platform on the other. In order for this to become possible, these high-level visual structures have to be transformed into low-level artifacts that can effectively be consumed by these platforms. These artifacts are Java POJOs (Plain Old Java Objects), and they are generated every time the data model is saved, by pressing the "Save" button in the top Data Modeller Menu.

Figure 9.64. Save the data model from the top menu

At this time each entity that has been defined in the model will be translated into a Java class, according to the following transformation rules:

- The entity's identifier property will become the Java class's name. It therefore needs to be a valid Java identifier.
- The entity's package property becomes the Java class's package declaration.
- The entity's superclass property (if present) becomes the Java class's extension declaration.
- The entity's label and description properties will translate into the Java annotations "@org.kie.workbench.common.services.datamodeller.annotations.Label" and "@org.kie.workbench.common.services.datamodeller.annotations.Description", respectively. These annotations are merely a way of preserving the associated information, and as yet are not processed any further.
- The entity's role property (if present) will be translated into the "@org.kie.api.definition.type.Role" Java annotation, that *IS* interpreted by the application platform, in the sense that it marks this Java class as a Drools Event Fact-Type.

A standard Java default (or no parameter) constructor is generated, as well as a full parameter constructor, i.e. a constructor that accepts as parameters a value for each of the entity's user-defined fields.

The entity's user-defined fields are translated into Java class fields, each one of them with its own getter and setter method, according to the following transformation rules:

- The entity field's identifier will become the Java field identifier. It therefore needs to be a valid Java identifier.
- The entity field's type is directly translated into the Java class's field type. In case the entity field was declared to be multiple (i.e. '[0..N]'), then the generated field is of the "java.util.List" type.
- The equals property: when it is set for a specific field, then this class property will be annotated with the "@org.kie.api.definition.type.Key" annotation, which is interpreted by the Drools Engine, and it will 'participate' in the generated equals() method, which overwrites the equals() method of the Object class. The latter implies that if the field is a 'primitive' type, the equals method will simply compares its value with the value of the corresponding field in another

instance of the class. If the field is a sub-entity or a collection type, then the equals method will make a method-call to the equals method of the corresponding entity's Java class, or of the java.util.List standard Java class, respectively.

If the equals property is checked for *ANY* of the entity's user defined fields, then this also implies that in addition to the default generated constructors another constructor is generated, accepting as parameters all of the fields that were marked with Equals. Furthermore, generation of the equals() method also implies that also the Object class's hashCode() method is overwritten, in such a manner that it will call the hashCode() methods of the corresponding Java class types (be it 'primitive' or user-defined types) for all the fields that were marked with Equals in the Data Model.

- The position property: this field property is automatically set for all user-defined fields, starting from 0, and incrementing by 1 for each subsequent new field. However the user can freely changes the position among the fields. At code generation time this property is translated into the "@org.kie.api.definition.type.Position" annotation, which can be interpreted by the Drools Engine. Also, the established property order determines the order of the constructor parameters in the generated Java class.
- The entity's role property (if present) will be translated into the "@org.kie.api.definition.type.Role" Java annotation, that *IS* interpreted by the application platform, in the sense that it marks this Java class as a Drools Event Fact-Type.

As an example, the generated Java class code for the Purchase Order entity, corresponding to its definition as shown in the following figure purchase_example.jpg is visualized in the figure at the bottom of this chapter. Note that the two of the entity's fields, namely 'header' and 'lines' were marked with Equals, and have been assigned with the positions 2 and 1, respectively).

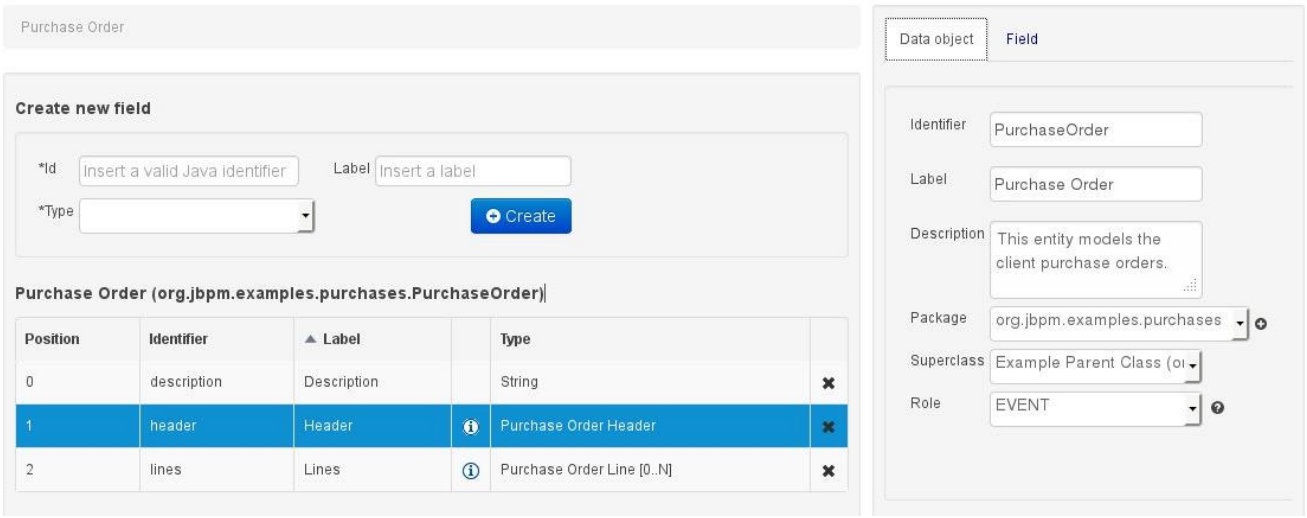


Figure 9.65. Purchase Order configuration

```
package org.jbpm.examples.purchases;

/**
 * This class was automatically generated by the data modeler tool.
 */
@org.kie.api.definition.type.Role(value =
org.kie.api.definition.type.Role.Type.EVENT)
@org.kie.workbench.common.services.datamodeller.annotations.Label(value =
"Purchase Order")
@org.kie.workbench.common.services.datamodeller.annotations.Description(value =
"This entity models the client purchase orders.")
public class PurchaseOrder extends org.jbpm.examples.purchases.parent
implements java.io.Serializable {

static final long serialVersionUID = 1L;

@org.kie.workbench.common.services.datamodeller.annotations.Label(value =
"Description")
@org.kie.api.definition.type.Position(value = 0)
@org.kie.workbench.common.services.datamodeller.annotations.Description(value =
"A description for this purchase order.")
private java.lang.String description;

@org.kie.workbench.common.services.datamodeller.annotations.Label(value =
"Lines")
@org.kie.api.definition.type.Position(value = 1)
@org.kie.workbench.common.services.datamodeller.annotations.Description(value =
"The purchase order items (collection of Purchase Order Line sub-entities).")
@org.kie.api.definition.type.Key
private java.util.List<org.jbpm.examples.purchases.PurchaseOrderLine> lines;

@org.kie.workbench.common.services.datamodeller.annotations.Label(value =
"Header")
@org.kie.api.definition.type.Position(value = 2)
@org.kie.workbench.common.services.datamodeller.annotations.Description(value =
"The purchase order header (Purchase Order Header sub-entity).")
@org.kie.api.definition.type.Key
private org.jbpm.examples.purchases.PurchaseOrderHeader header;

public PurchaseOrder() {}

public PurchaseOrder(
java.lang.String description,
java.util.List<org.jbpm.examples.purchases.PurchaseOrderLine> lines,
org.jbpm.examples.purchases.PurchaseOrderHeader header )
{
this.description = description;
this.lines = lines;
this.header = header;
```

```
    }

    public PurchaseOrder(
    java.util.List<org.jbpm.examples.purchases.PurchaseOrderLine> lines,
    org.jbpm.examples.purchases.PurchaseOrderHeader header )
    {
    this.lines = lines;
    this.header = header;
    }

    public java.lang.String getDescription() {
    return this.description;
    }

    public void setDescription( java.lang.String description ) {
    this.description = description;
    }

    public java.util.List<org.jbpm.examples.purchases.PurchaseOrderLine>
    getLines()
    {
    return this.lines;
    }

    public void setLines(
    java.util.List<org.jbpm.examples.purchases.PurchaseOrderLine> lines )
    {
    this.lines = lines;
    }

    public org.jbpm.examples.purchases.PurchaseOrderHeader getHeader() {
    return this.header;
    }

    public void setHeader( org.jbpm.examples.purchases.PurchaseOrderHeader
    header )
    {
    this.header = header;
    }

    @Override
    public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    org.jbpm.examples.purchases.PurchaseOrder that =
    (org.jbpm.examples.purchases.PurchaseOrder)o;
    if (lines != null ? !lines.equals(that.lines) : that.lines != null)
    return false;
    if (header != null ? !header.equals(that.header) : that.header != null)
```

```
    return false;
    return true;
    }

    @Override
    public int hashCode() {
    int result = 17;
    result = 13 * result + (lines != null ? lines.hashCode() : 0);
    result = 13 * result + (header != null ? header.hashCode() : 0);
    return result;
    }
    }
```

9.7.6.6. Using external models

Using an external model means the ability to use a set for already defined POJOs in current project context. In order to make those POJOs available a dependency to the given JAR should be added. Once the dependency has been added the external POJOs can be referenced from current project data model.

There are two ways to add a dependency to an external JAR file:

- Dependency to a JAR file already installed in current local M2 repository (typically associated the the user home).
- Dependency to a JAR file installed in current Kie Workbench/Drools Workbench "Guvnor M2 repository". (internal to the application)

9.7.6.6.1. Dependency to a JAR file in local M2 repository

To add a dependency to a JAR file in local M2 repository follow this steps.

9.7.6.6.1.1. Open the Project Editor for current project and select the Dependencies view.

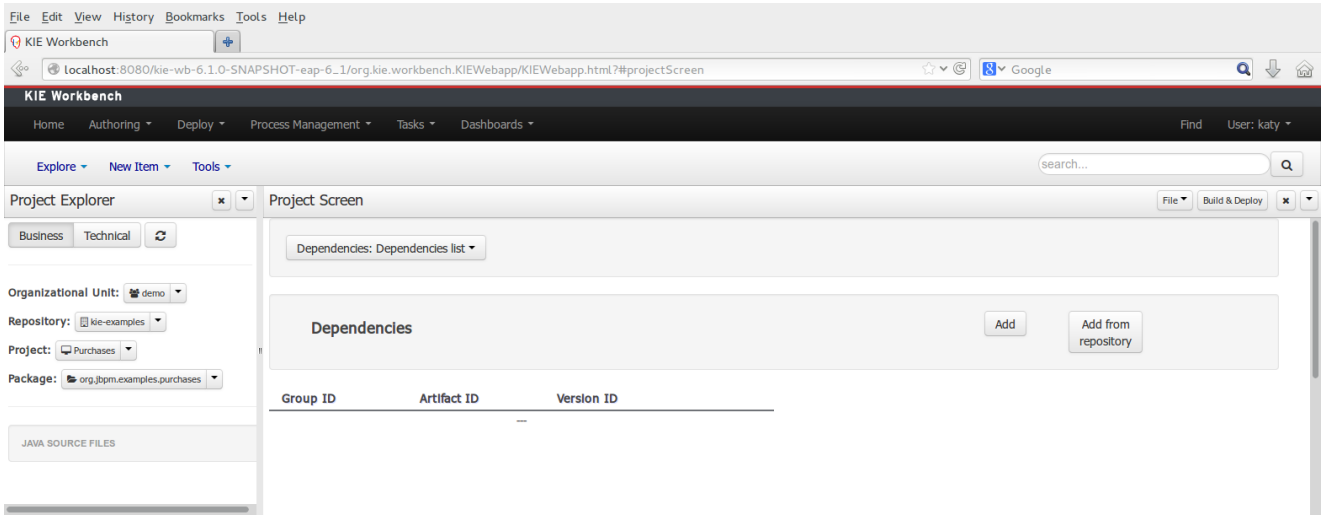


Figure 9.66. Project editor.

9.7.6.6.1.2. Click on the "Add" button to add a new dependency line.

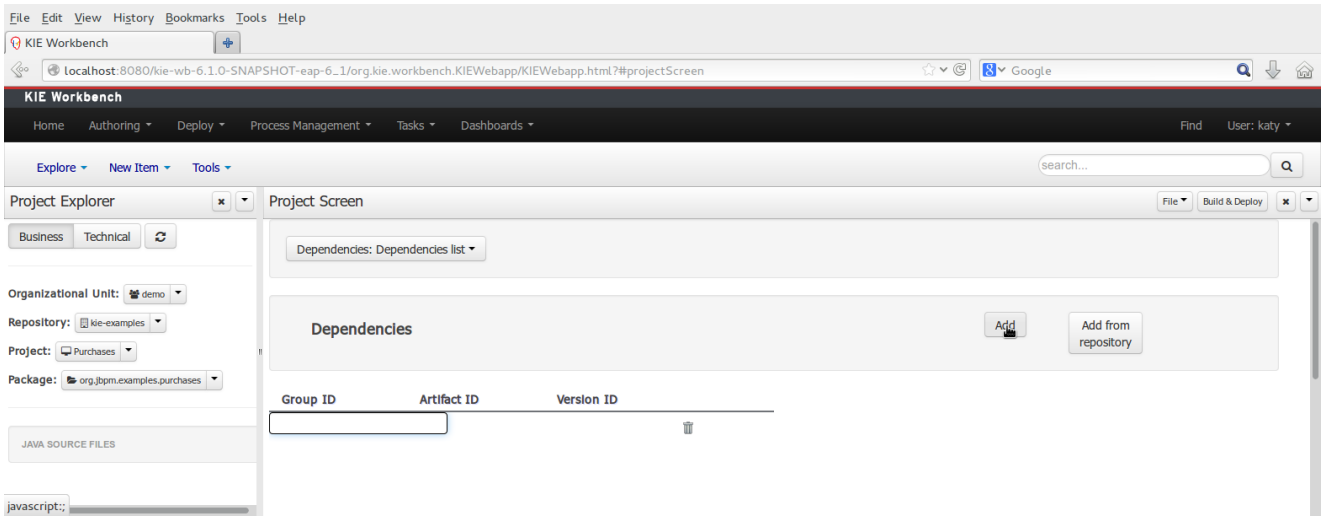


Figure 9.67. New dependency line.

9.7.6.6.1.3. Complete the GAV for the JAR file already installed in local M2 repository.

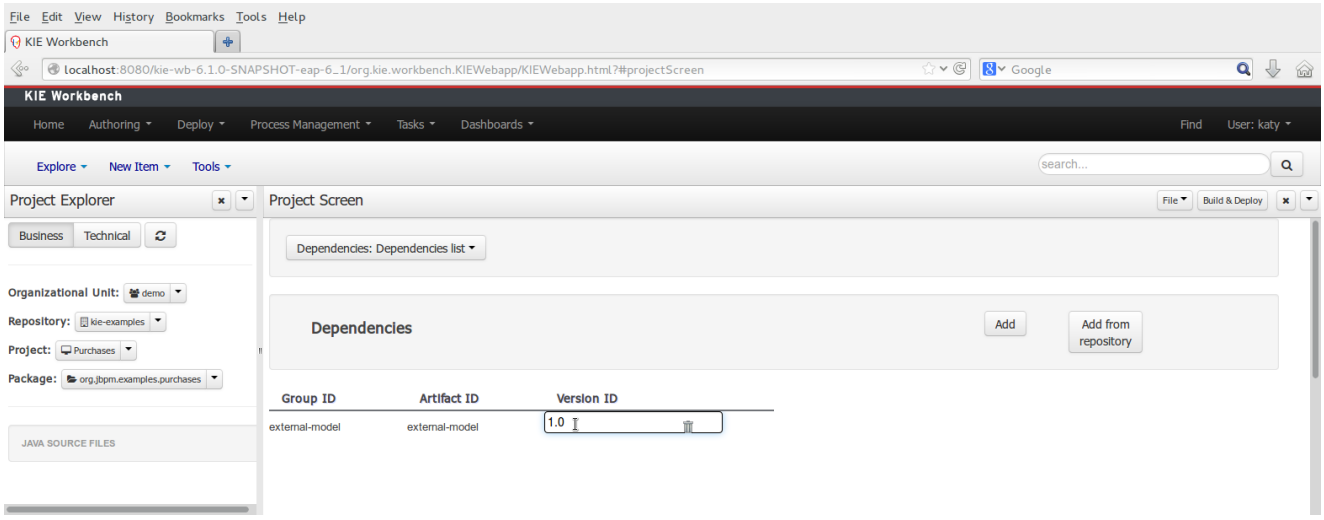


Figure 9.68. Dependency line edition.

9.7.6.6.1.4. Save the project to update its dependencies.

When project is saved the POJOs defined in the external file will be available.

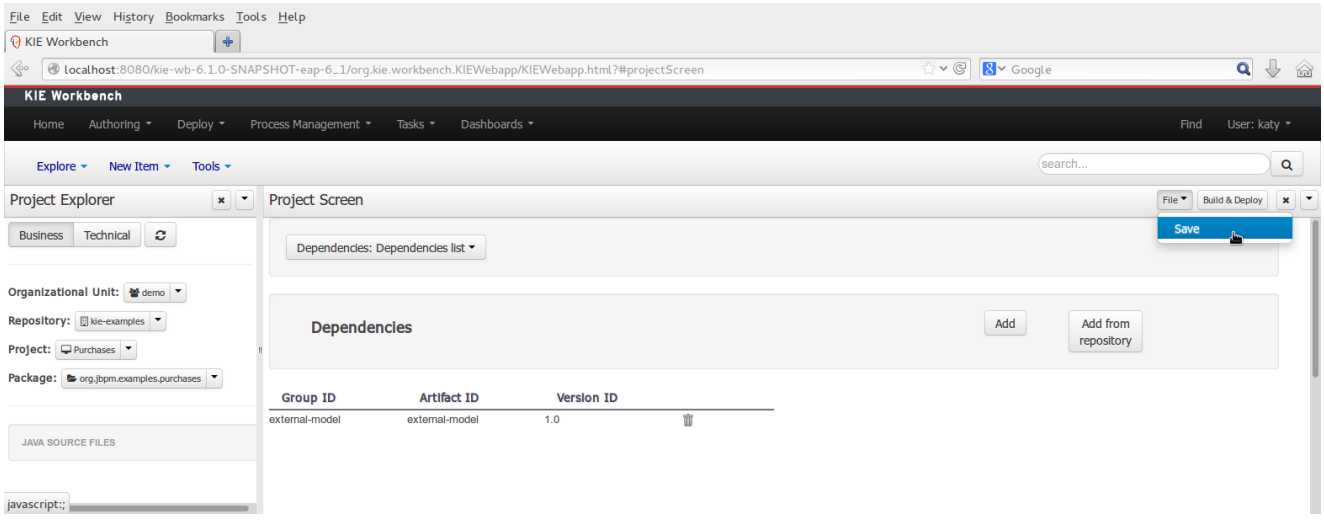


Figure 9.69. Save project.

9.7.6.6.2. Dependency to a JAR file in current "Guvnor M2 repository".

To add a dependency to a JAR file in current "Guvnor M2 repository" follow this steps.

9.7.6.6.2.1. Open the Maven Artifact Repository editor.

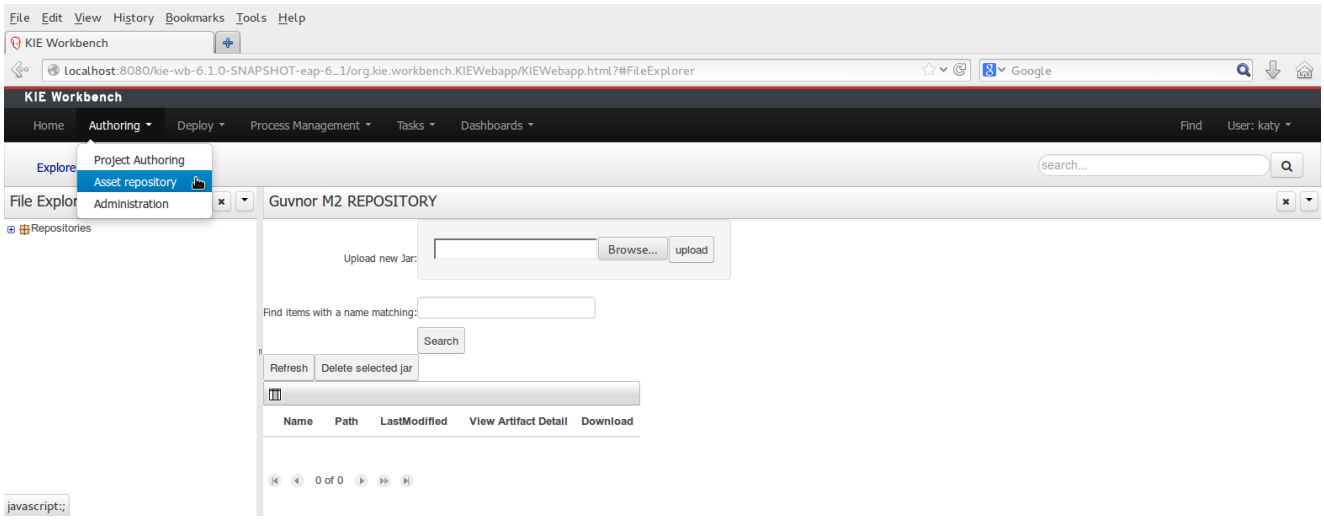


Figure 9.70. Guvnor M2 Repository editor.

9.7.6.6.2.2. Browse your local file system and select the JAR file to be uploaded using the Browse button.

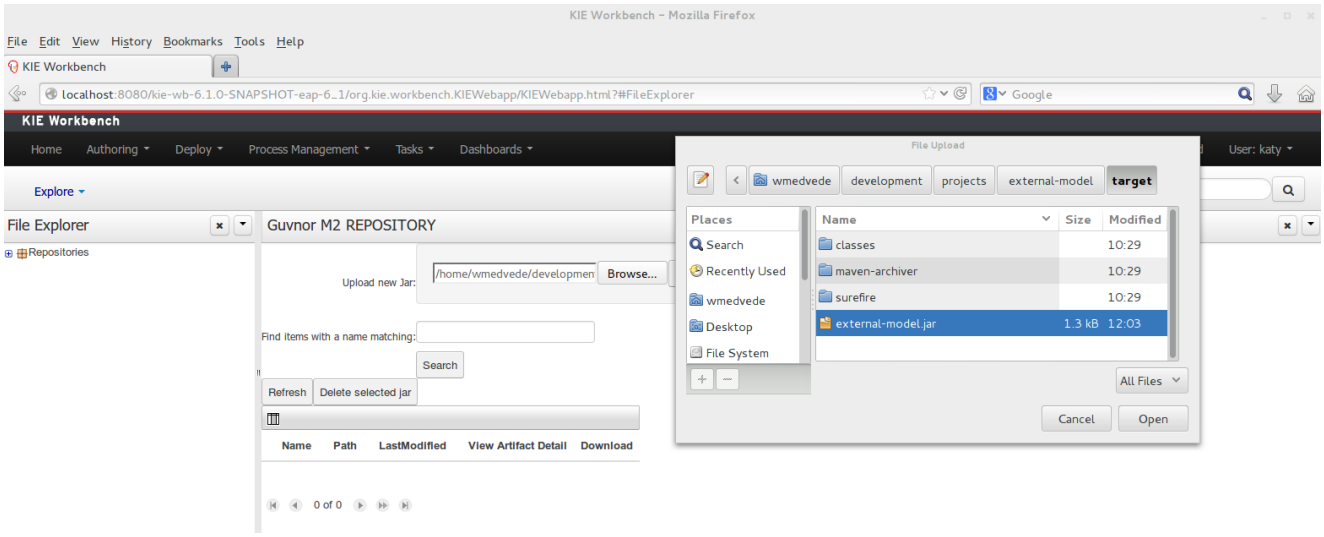


Figure 9.71. File browser.

9.7.6.6.2.3. Upload the file using the Upload button.

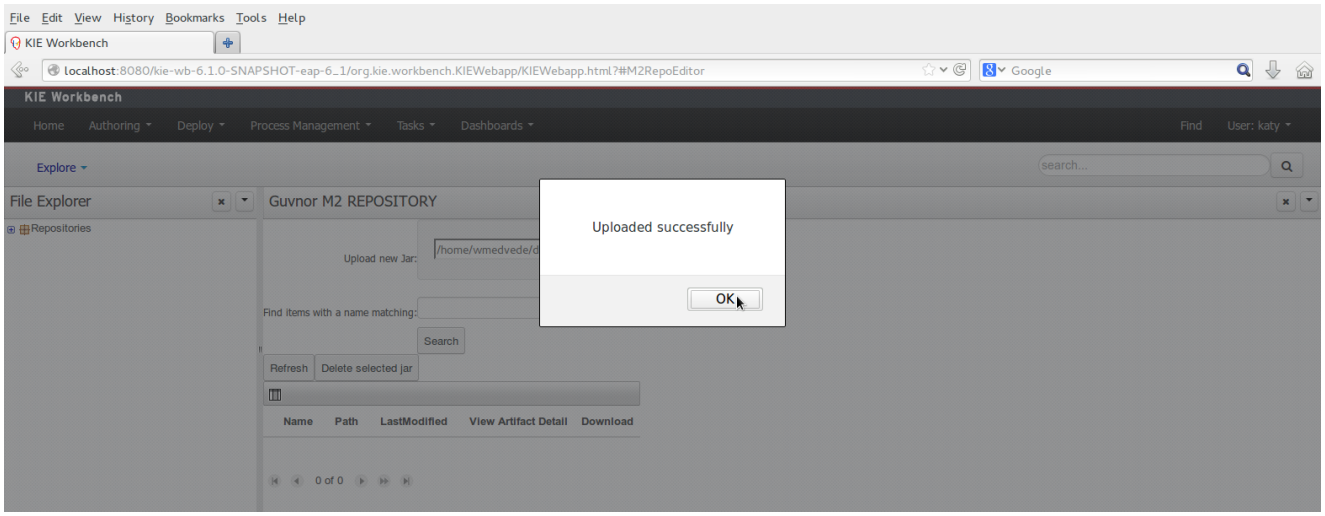


Figure 9.72. File upload success.

9.7.6.6.2.4. Guvnor M2 repository files.

Once the file has been loaded it will be displayed in the repository files list.

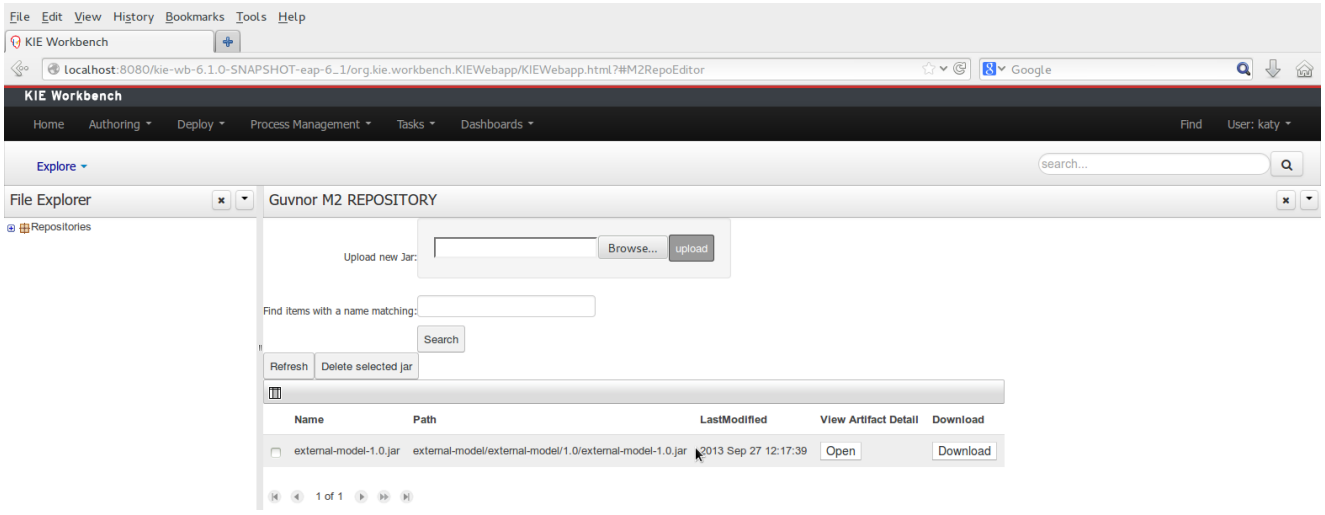


Figure 9.73. Files list.

9.7.6.6.2.5. Provide a GAV for the uploaded file (optional).

If the uploaded file is not a valid Maven JAR (don't have a pom.xml file) the system will prompt the user in order to provide a GAV for the file to be installed.

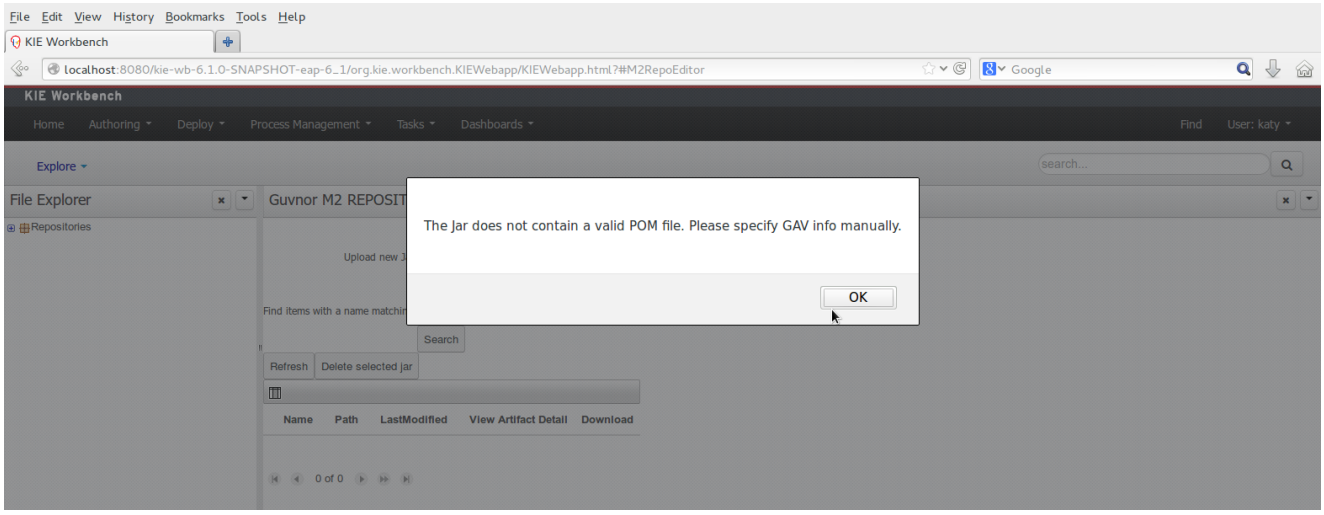


Figure 9.74. Not valid POM.

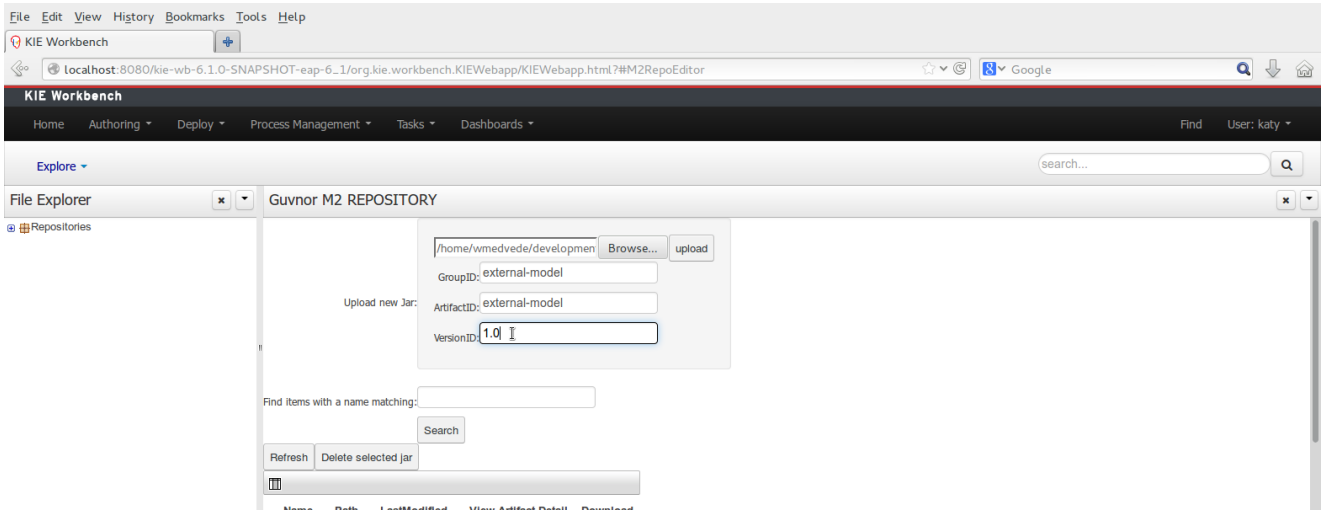


Figure 9.75. Enter GAV manually.

9.7.6.6.2.6. Add dependency from repository.

Open the project editor (see bellow) and click on the "Add from repository" button to open the JAR selector to see all the installed JAR files in current "Guvnor M2 repository". When the desired file is selected the project should be saved in order to make the new dependency available.

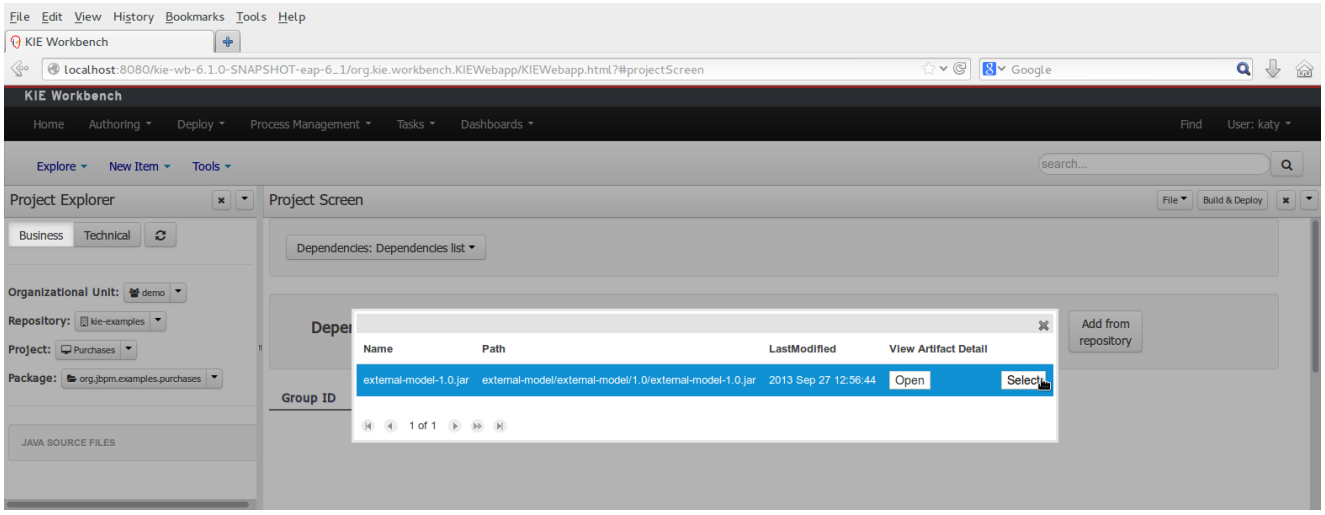


Figure 9.76. Select JAR from "Maven Artifact Repository".

9.7.6.6.3. Using the external objects

When a dependency to an external JAR has been set, the external POJOs can be used in the context of current project data model in the following ways:

- External POJOs can be extended by current model data objects.

- External POJOs can be used as field types for current model data objects.

The following screenshot shows how external objects are prefixed with the string " -ext- " in order to be quickly identified.

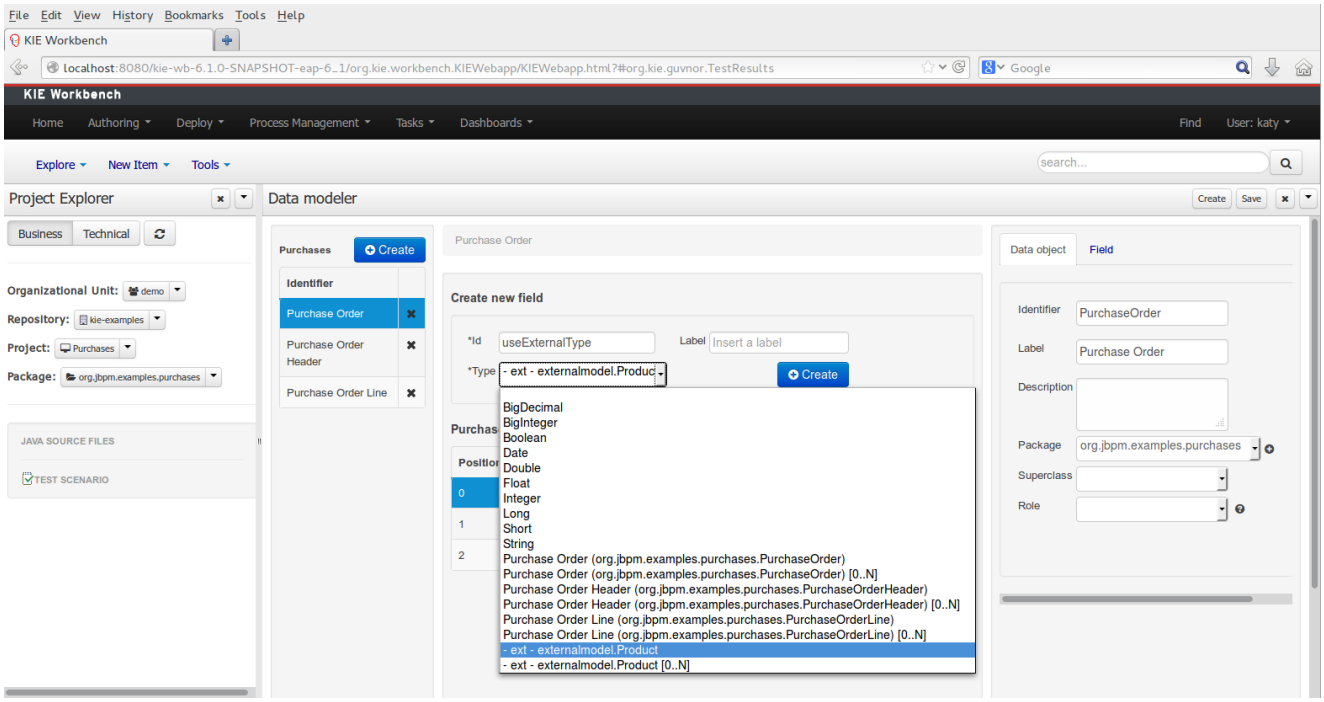


Figure 9.77. Identifying external objects.

9.7.6.7. External changes to models

It is possible to modify a project's assets externally, i.e. accessing them directly through the project's repository. While NOT a recommended practice, it is important to be aware of the implications this entails.

Caution

Performing changes to the data model outside of the context of the application is NOT recommended, and could lead to loss of information!

From an application context's perspective, we can basically identify two different scenarios:

9.7.6.7.1. No changes have been undertaken through the application

In this scenario the application user has basically just been navigating through the data model, without making any changes to it. Meanwhile, another user modifies the data model externally.

In this case, no immediate warning is issued to the application user. However, as soon as the user tries to make any kind of change, such as add or remove data objects or properties, or change any of the existing ones, the following pop-up will be shown:

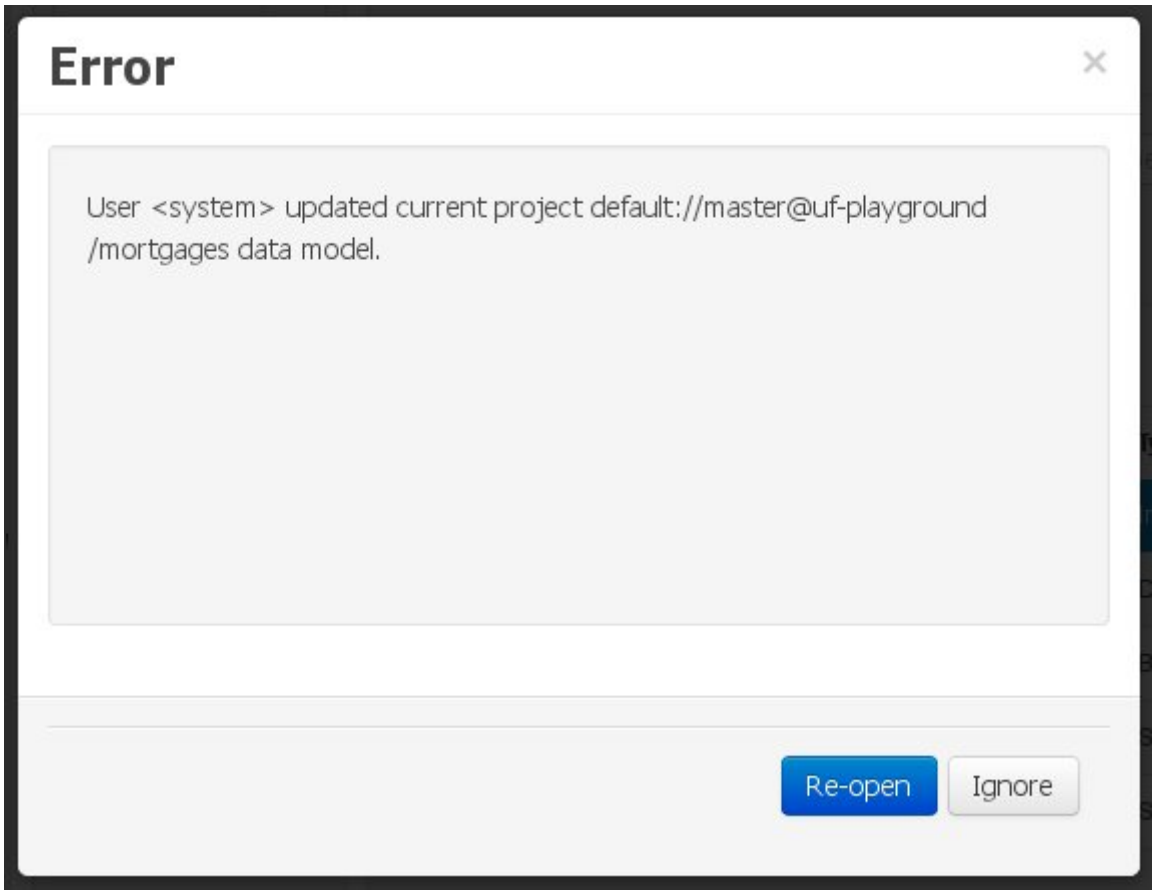


Figure 9.78. External changes warning

The user can choose to either:

- Re-open the data model, thus loading any external changes, and then perform the modification he was about to undertake, or
- Ignore any external changes, and go ahead with the modification to the model. In this case, when trying to persist these changes, another pop-up warning will be shown:

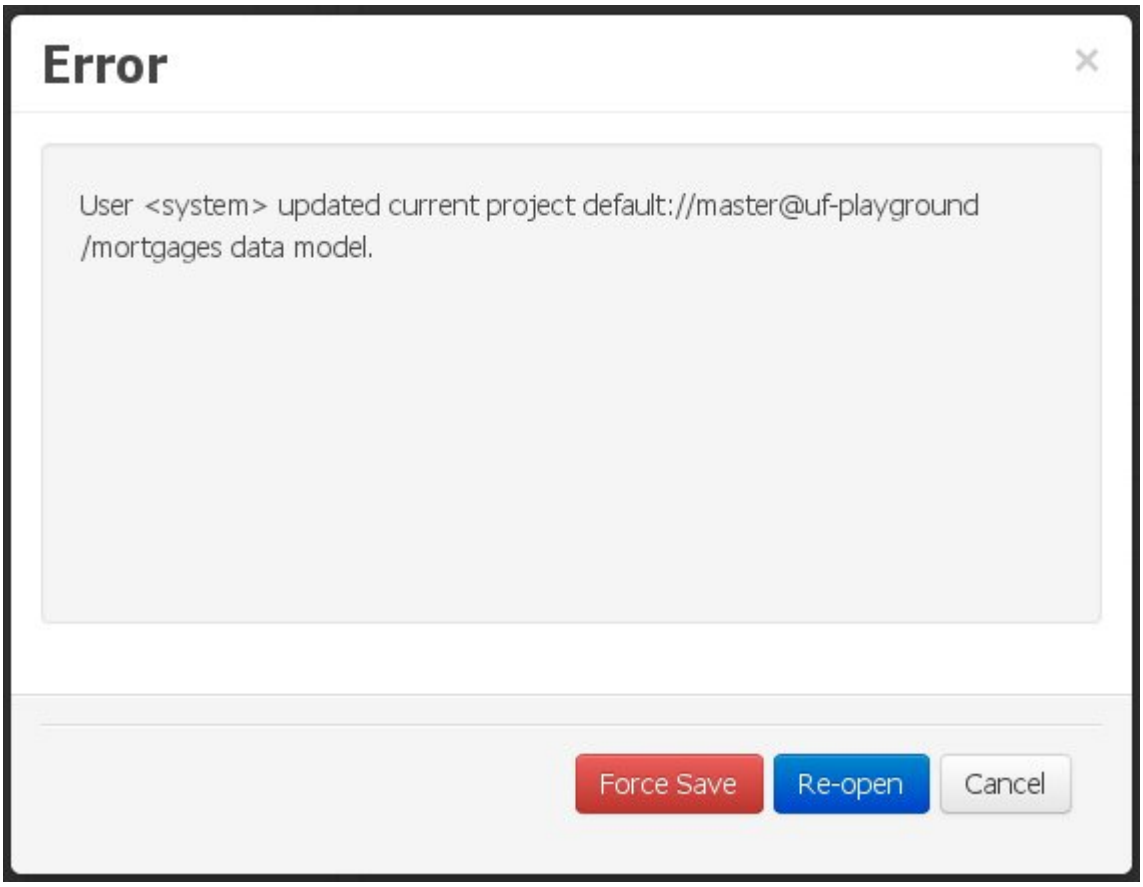


Figure 9.79. Force save / re-open

The "Force Save" option will effectively overwrite any external changes, while "Re-open" will discard any local changes and reload the model.

Warning

"Force Save" overwrites any external changes!

9.7.6.7.2. Changes have been undertaken through the application

The application user has made changes to the data model. Meanwhile, another user simultaneously modifies the data model from outside the application context.

In this alternative scenario, immediately after the external user commits his changes to the asset repository, a warning is issued to the application user:

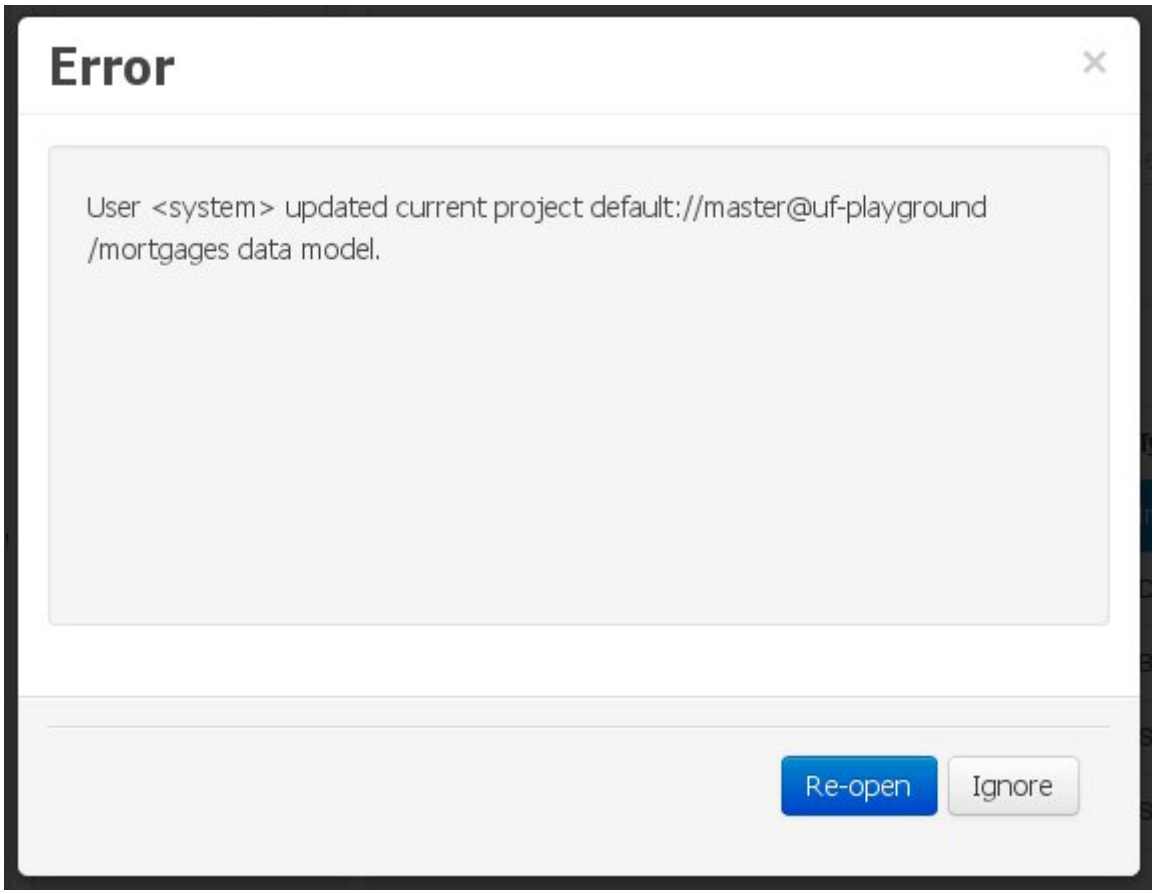


Figure 9.80. External changes warning

As with the previous scenario, the user can choose to either:

- Re-open the data model, thus losing any modifications that where made through the application, or
- Ignore any external changes, and continue working on the model.

  One of the following possibilities can now occur:

  - The user tries to persist the changes he made to the model by clicking the "Save" button in the data modeller top level menu. This leads to the following warning message:

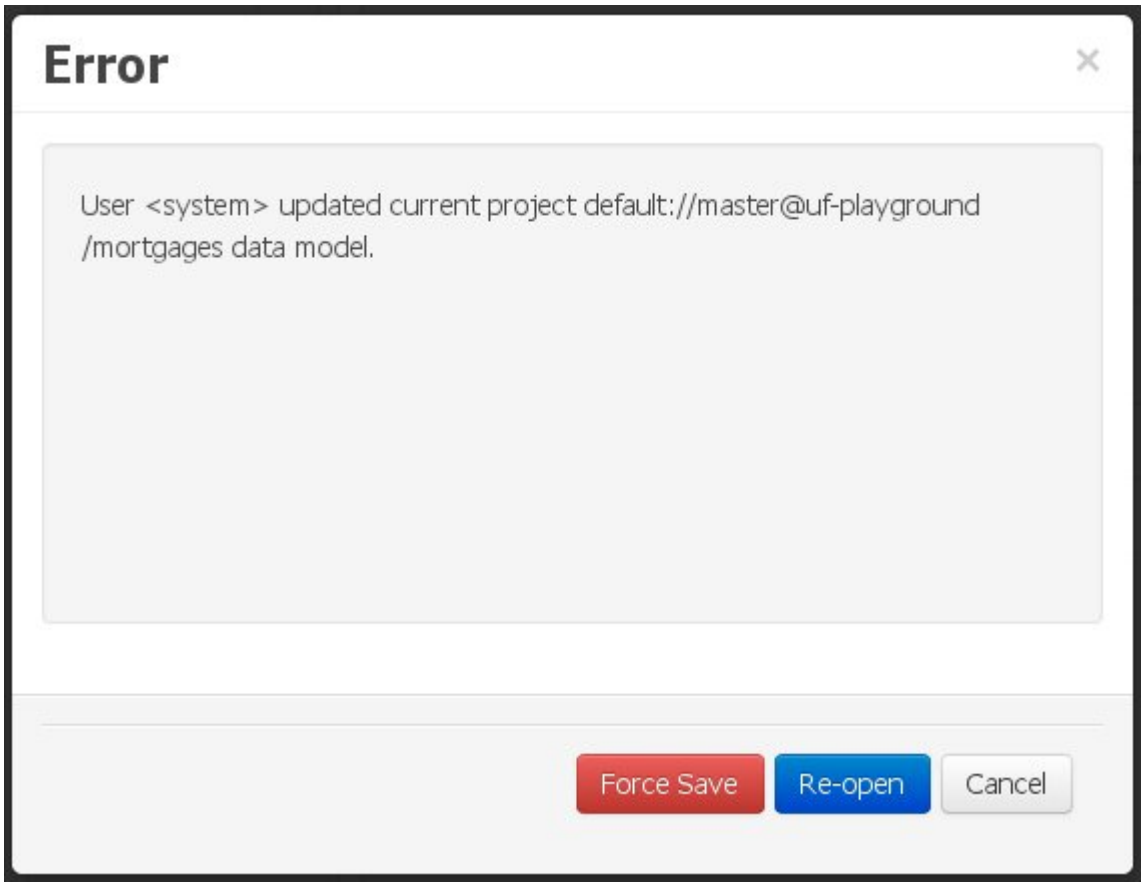


Figure 9.81. Force save / re-open

The "Force Save" option will effectively overwrite any external changes, while "Re-open" will discard any local changes and reload the model.

- The user switches to another project. In this case he will be warned of the existence of non-persisted local changes through the following warning message:

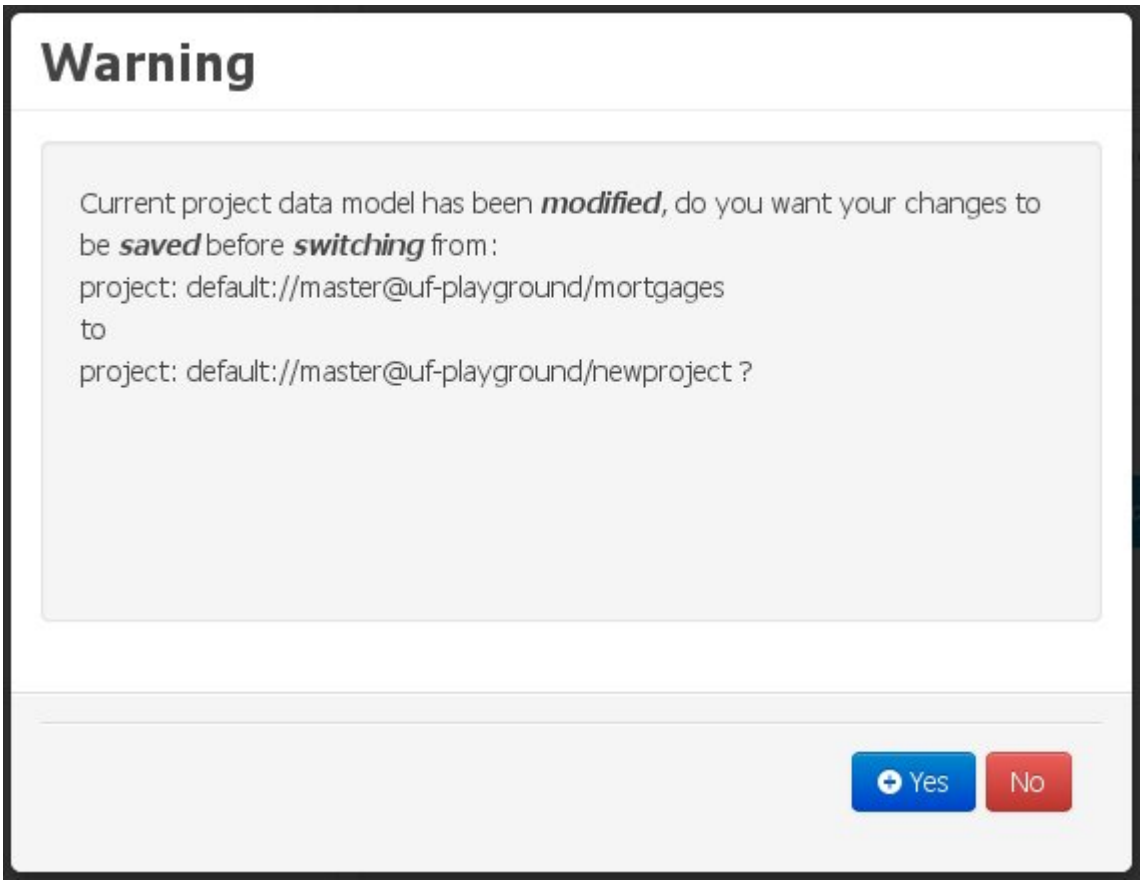


Figure 9.82. Project switch warning

If the user chooses to persist the local changes, then another pop-up message will point out the existence of the changes that were made externally:

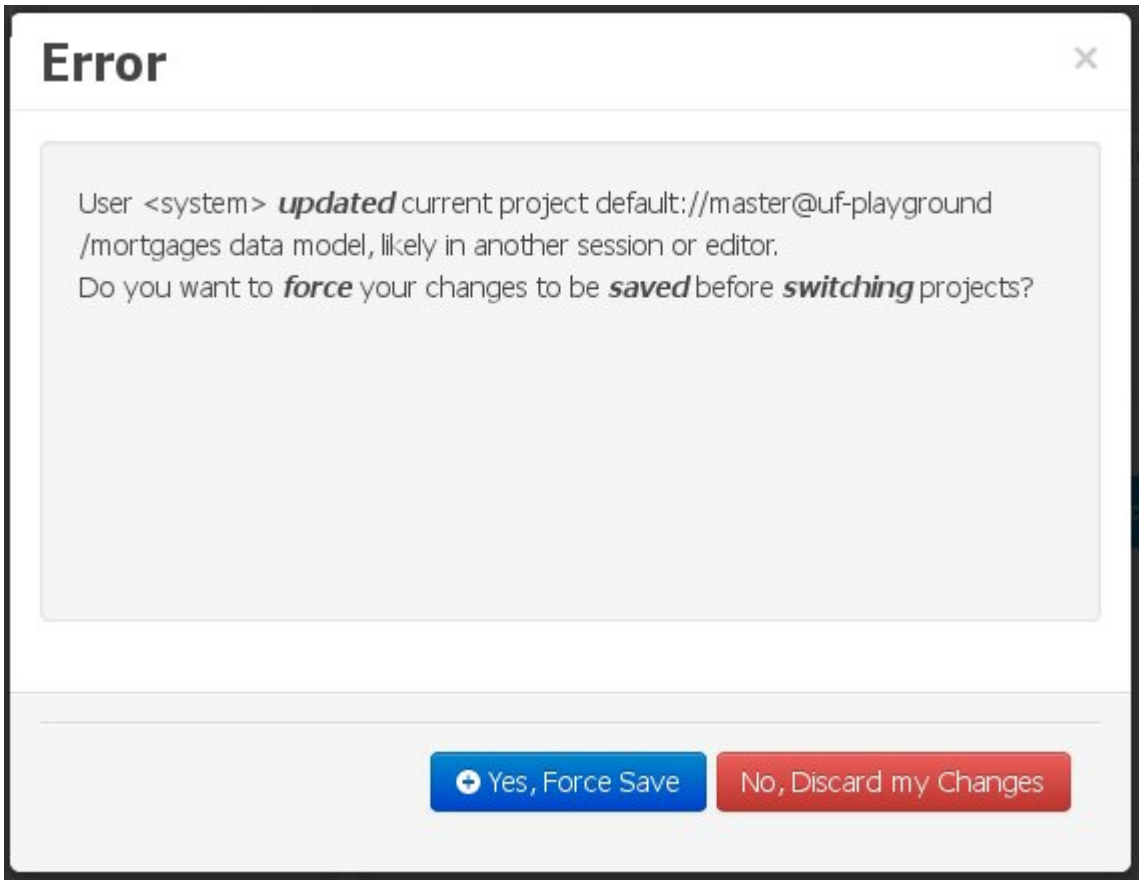


Figure 9.83. Project switch external changes warning

The "Yes, Force Save" option will effectively overwrite any external changes, while "No, Discard my Changes" will switch to the other project, discarding any local changes.

9.7.7. Categories Editor

Categories allow assets to be labelled (or tagged) with any number of categories that you define. Assets can belong to any number of categories. In the below diagram, you can see this can in effect create a folder/explorer like view of categories. The names can be anything you want, and are defined by the Workbench administrator (you can also remove/add new categories).

Note

Categories do not have the same role in the current release of the Workbench as they had in prior versions (up to and including 5.5). Projects can no longer be built using a selector to include assets that are labelled with certain Categories. Categories are therefore considered a deprecated feature.

9.7.7.1. Launching the Categories Editor

The Categories Editor is available from the Repository menu on the Authoring Perspective.

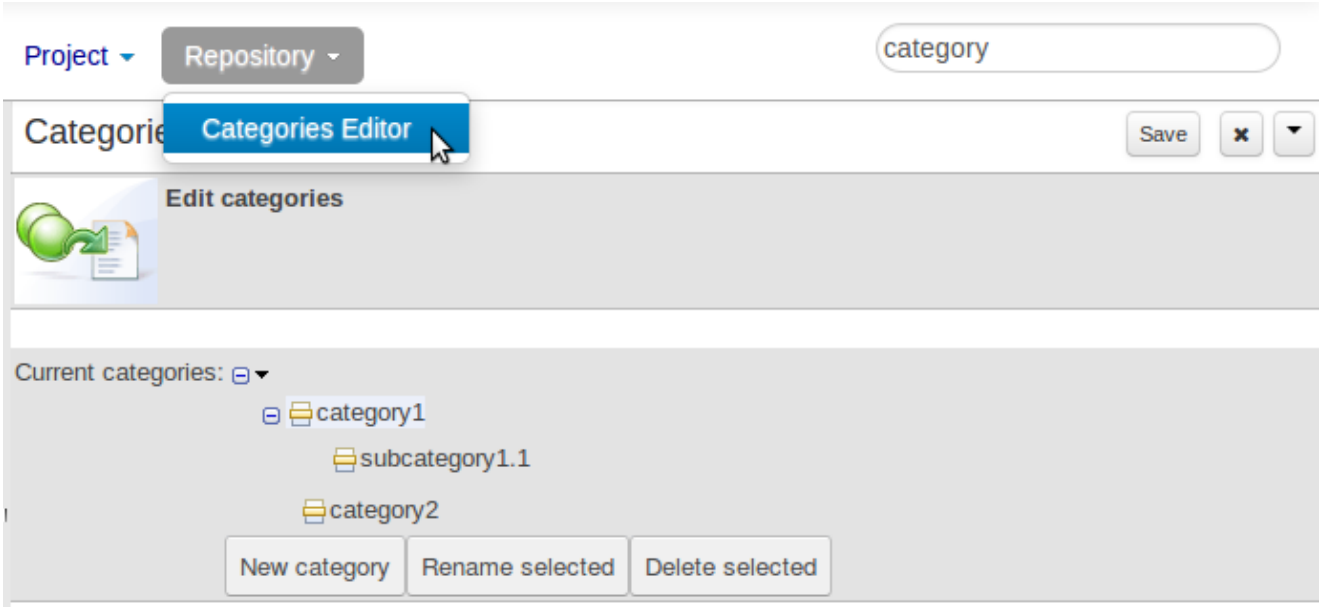


Figure 9.84. Launching Categories Editor

9.7.7.2. Managing Categories

The below view shows the administration screen for setting up categories (there) are no categories in the system by default. As the categories can be hierarchical you chose the "parent" category that you want to create a sub-category for. From here categories can also be removed (but only if they are not in use by any current versions of assets).

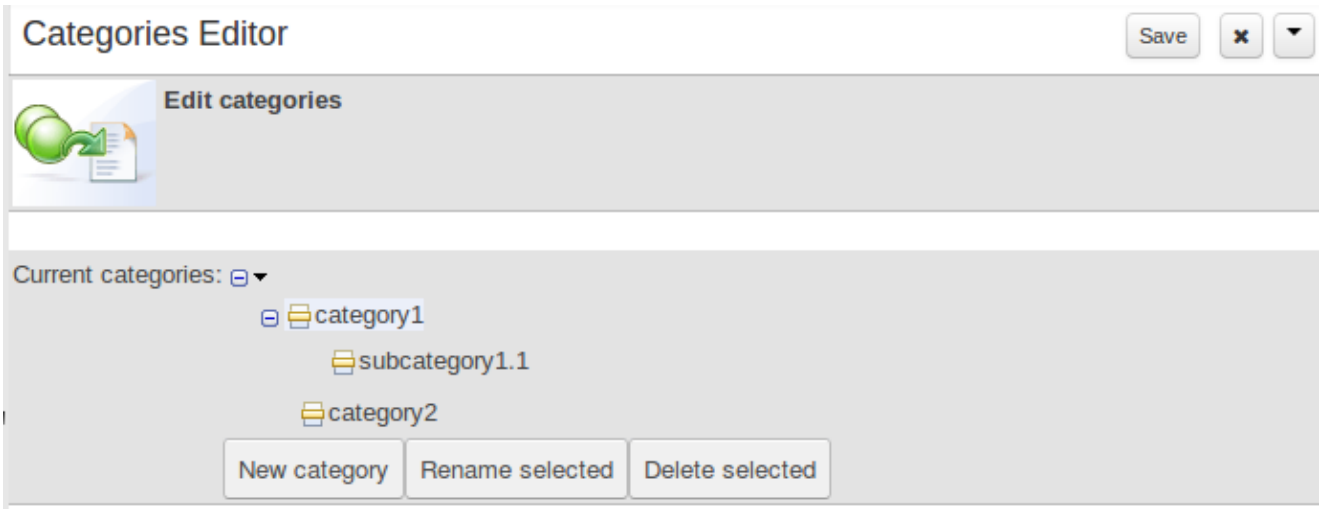


Figure 9.85. Managing categories

Generally categories are created with meaningful name that match the area of the business the rule applies to (if the rule applies to multiple areas, multiple categories can be attached).

9.7.7.3. Adding Categories to assets

Assets can be assigned Categories using the MetaData tab on the assets' editor.

When you open an asset to view or edit, it will show a list of categories that it currently belongs to If you make a change (remove or add a category) you will need to save the asset - this will create a new item in the version history. Changing the categories of a rule has no effect on its execution.

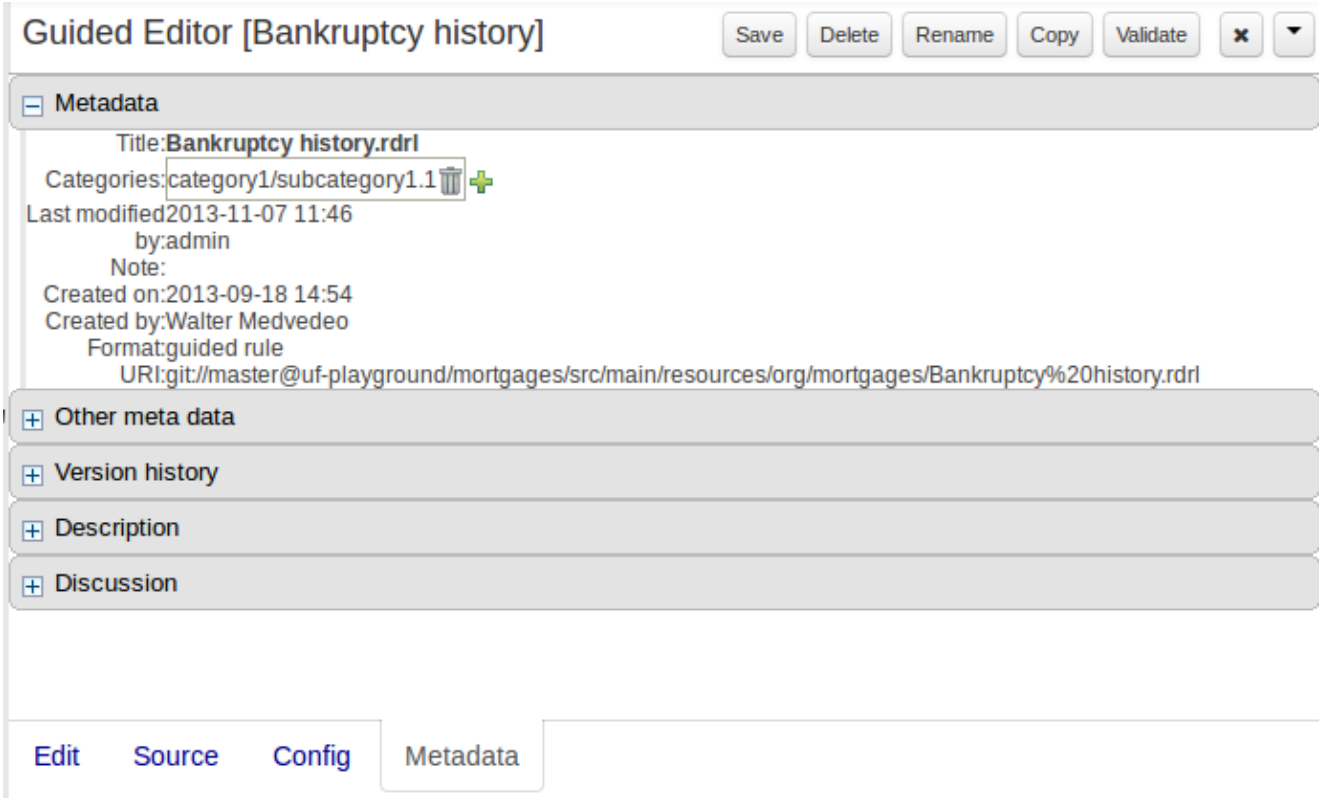


Figure 9.86. Adding Categories to an asset

9.8. Embedding Workbench In Your Application

As we already know, Workbench provides a set of editors to author assets in different formats. According to asset's format a specialized editor is used.

One additional feature provided by Workbench is the ability to embed it in your own (Web) Applications thru it's **standalone** mode. So, if you want to edit rules, processes, decision tables, etc... in your own applications without switch to Workbench, you can.

In order to embed Workbench in your application all you'll need is the Workbench application deployed and running in a web/application server and, from within your own web applications, an iframe with proper HTTP query parameters as described in the following table.

Table 9.2. HTTP query parameters for standalone mode

Parameter Name	Explanation	Allow multiple values	Example
standalone	With just the presence of this parameter workbench will switch to standalone mode.	no	(none)
path	Path to the asset to be edited. Note that asset should already exists.	no	git://master@uf-playground/todo.md
perspective	Reference to an existing perspective name.	no	org.guvnor.m2repo.client.perspectives.Guv
header	Defines the name of the header that should be displayed (useful for context menu headers).	yes	ComplementNavArea

Note

Path and Perspective parameters are mutual exclusive, so can't be used together.

Chapter 10. Workbench Integration

10.1. REST

REST API calls to Knowledge Store allow you to manage the Knowledge Store content and manipulate the static data in the repositories of the Knowledge Store. The calls are asynchronous, that is, they continue their execution after the call was performed as a job. The job ID is returned by every calls to allow after the REST API call was performed to request the job status and verify whether the job finished successfully. Parameters of these calls are provided in the form of JSON entities.

When using Java code to interface with the REST API, the classes used in POST operations or otherwise returned by various operations can be found in the `(org.kie.workbench.services:)kie-wb-common-services` JAR. All of the classes mentioned below can be found in the `org.kie.workbench.common.services.shared.rest` package in that JAR.

10.1.1. Job calls

Every Knowledge Store REST call returns its job ID after it was sent. This is necessary as the calls are asynchronous and you need to be able to reference the job to check its status as it goes through its lifecycle. During its lifecycle, a job can have the following statuses:

- `ACCEPTED`: the job was accepted and is being processed
- `BAD_REQUEST`: the request was not accepted as it contained incorrect content
- `RESOURCE_NOT_EXIST`: the requested resource (path) does not exist
- `DUPLICATE_RESOURCE`: the resource already exists
- `SERVER_ERROR`: an error on the server occurred
- `SUCCESS`: the job finished successfully
- `FAIL`: the job failed
- `DENIED`: the job was denied
- `GONE`: the job ID could not be found

  A job can be GONE in the following cases:

  - The job was explicitly removed
  - The job finished and has been deleted from the status cache (the job is removed from status cache after the cache has reached its maximum capacity)

- The job never existed

The following `job` calls are provided:

[GET] `/jobs/{jobID}`
Returns the job status

Returns a `JobResult` instance

Example 10.1. An example (formatted) response body to the get job call on a repository clone request

```
"{
  "status":"SUCCESS",
  "jodId":"1377770574783-27",
   "result":"Alias: testInstallAndDeployProject, Scheme: git, Uri: git://
testInstallAndDeployProject",
  "lastModified":1377770578194,"detailedResult":null
}"
```

[DELETE] `/jobs/{jobID}`
Removes the job: If the job is not yet being processed, this will remove the job from the job queue. However, this will not cancel or stop an ongoing job

Returns a `JobResult` instance

10.1.2. Repository calls

Repository calls are calls to the Knowledge Store that allow you to manage its Git repositories and their projects.

The following `repositories` calls are provided:

[GET] `/repositories`
Gets information about the repositories in the Knowledge Store

Returns a `Collection<Map<String, String>>` or `Collection<RepositoryRequest>` instance, depending on the JSON serialization library being used. The keys used in the `Map<String, String>` instance match the fields in the `RepositoryRequest` class

Example 10.2. An example (formatted) response body to the get repositories call

```
[
```

```
  {
    "name":"wb-assets",
    "description":"generic assets",
    "userName":null,
    "password":null,
    "requestType":null,
    "gitURL":"git://bpms-assets"
 },
 {
   "name":"loanProject",
   "description":"Loan processes and rules",
   "userName":null,
   "password":null,
   "requestType":null,
   "gitURL":"git://loansProject"
 }
]
```

[POST] `/repositories`

Creates a new empty repository or a new repository cloned from an existing (git) repository

Consumes a `RepositoryRequest` instance

Returns a `CreateOrCloneRepositoryRequest` instance

Example 10.3. An example (formatted) response body to the create repositories call

```
{
  "name":"new-project-repo",
  "description":"repo for my new project",
  "userName":null,"password":null,
  "requestType":"new",
  "gitURL":null
}
```

[DELETE] `/repositories/{repositoryName}`

Removes the repository from the Knowledge Store

Returns a `RemoveRepositoryRequest` instance

[POST] `/repositories/{repositoryName}/projects/`

Creates a project in the repository

Consumes an `Entity` instance

Returns a `CreateProjectRequest` instance

Example 10.4. An example (formatted) request body that defines the project to be created

```
{
  "name":"myProject",
  "description": "my project"
}
```

10.1.3. Organizational unit calls

Organizational unit calls are calls to the Knowledge Store that allow you to manage its organizational units, so as to organize the connected Git repositories.

The following `organizationalUnits` calls are provided:

[POST] `/organizationalunits`
: Creates an organizational unit in the Knowledge Store

  Consumes an `OrganizationalUnit` instance

  Returns a `CreateOrganizationalUnitRequest` instance

Example 10.5. An example (formatted) request body defining a new organizational unit to be created

```
{
  "name":"testgroup",
  "description":"",
  "owner":"tester",
  "repositories":["testGroupRepository"]
}
```

[POST] `/organizationalunits/`*`{organizationalUnitName}`*`/repositories/`*`{repositoryName}`*
: Adds the repository to the organizational unit

  Returns a `AddRepositoryToOrganizationalUnitRequest` instance

[DELETE] `/organizationalunits/`*`{organizationalUnitName}`*`/repositories/`*`{repositoryName}`*
: Removes the repository from the organizational unit

  Returns a `RemoveRepositoryFromOrganizationalUnitRequest` instance

10.1.4. Maven calls

Maven calls are calls to a Project in the Knowledge Store that allow you compile and deploy the Project resources.

The following `maven` calls are provided:

[POST] `/repositories/{repositoryName}/projects/{projectName}/maven/compile`
Compiles the project (equivalent to `mvn compile`)

Consumes a `BuildConfig` instance. While this must be supplied, it's not needed for the operation and may be left blank.

Returns a `CompileProjectRequest` instance

[POST] `/repositories/{repositoryName}/projects/{projectName}/maven/install`
Installs the project (equivalent to `mvn install`)

Consumes a `BuildConfig` instance. While this must be supplied, it's not needed for the operation and may be left blank.

Returns a `InstallProjectRequest` instance

[POST] `/repositories/{repositoryName}/projects/{projectName}/maven/test`
Compiles the project runs a test as part of compilation

Consumes a `BuildConfig` instance

Returns a `TestProjectRequest` instance

[POST] `/repositories/{repositoryName}/projects/{projectName}/maven/deploy`
Deploys the project (equivalent to `mvn deploy`)

Consumes a `BuildConfig` instance. While this must be supplied, it's not needed for the operation and may be left blank.

Returns a `DeployProjectRequest` instance

10.1.5. REST summary

The URL templates in the table below are relative the following URL:

- `http://server:port/business-central/rest`

Table 10.1. Knowledge Store REST calls

URL Template	Type	Description
/jobs/{jobID}	GET	return the job status

URL Template	Type	Description
/jobs/{jobID}	DELETE	Remove the job
/organizationalunits	GET	return a list of organizational units
/organizationalunits	POST	create an organizational unit in the Knowledge Store described by the JSON `OrganizationalUnit` entity
/organizationalunits/{organizationalUnitName}/repositories/{repositoryName}	POST	add a repository to an organizational unit
/organizationalunits/{organizationalUnitName}/repositories/{repositoryName}	DELETE	Remove a repository from an organizational unit
/repositories/	POST	add the repository to the organizational unit described by the JSON `RepositoryReqest` entity
/repositories	GET	return the repositories in the Knowledge Store
/repositories/{repositoryName}	DELETE	Remove the repository from the Knowledge Store
/repositories/	POST	create or clone the repository defined by the JSON `RepositoryRequest` entity
/repositories/{repositoryName}/projects/	POST	create the project defined by the JSON entity in the repository
/repositories/{repositoryName}/projects/{projectName}/maven/compile/	POST	compile the project
/repositories/{repositoryName}/projects/{projectName}/maven/install	POST	install the project
/repositories/{repositoryName}/projects/{projectName}/maven/test/	POST	compile the project and run tests as part of compilation
/repositories/{repositoryName}/projects/{projectName}/maven/deploy/	POST	deploy the project

Chapter 11. Workbench High Availability

11.1.1. VFS clustering

The *VFS repositories* (usually git repositories) stores all the assets (such as rules, decision tables, process definitions, forms, etc). If that VFS resides on each local server, then it must be kept in sync between all servers of a cluster.

Use *Apache Zookeeper* [http://zookeeper.apache.org/] and *Apache Helix* [http://helix.incubator.apache.org/] to accomplish this. Zookeeper glues all the parts together. Helix is the cluster management component that registers all cluster details (nodes, resources and the cluster itself). Uberfire (on top of which Workbench is build) uses those 2 components to provide VFS clustering.

To create a VFS cluster:

1. Download *Apache Zookeeper* [http://zookeeper.apache.org/] and *Apache Helix* [http://helix.incubator.apache.org/].
2. Install both:
   a. Unzip Zookeeper into a directory (`$ZOOKEEPER_HOME`).
   b. In `$ZOOKEEPER_HOME`, copy `zoo_sample.conf` to `zoo.conf`
   c. Edit `zoo.conf`. Adjust the settings if needed. Usually only these 2 properties are relevant:

```
# the directory where the snapshot is stored.
dataDir=/tmp/zookeeper
# the port at which the clients will connect
clientPort=2181
```

   d. Unzip Helix into a directory (`$HELIX_HOME`).
3. Configure the cluster in Zookeeper:
   a. Go to its `bin` directory:

```
$ cd $ZOOKEEPER_HOME/bin
```

   b. Start the Zookeeper server:

```
$ sudo ./zkServer.sh start
```

If the server fails to start, verify that the `dataDir` (as specified in `zoo.conf`) is accessible.

c. To review Zookeeper's activities, open `zookeeper.out`:

```
$ cat $ZOOKEEPER_HOME/bin/zookeeper.out
```

4. Configure the cluster in Helix:

a. Go to its `bin` directory:

```
$ cd $HELIX_HOME/bin
```

b. Create the cluster:

```
$ ./helix-admin.sh --zkSvr localhost:2181 --addCluster kie-cluster
```

The `zkSvr` value must match the used Zookeeper server. The cluster name (`kie-cluster`) can be changed as needed.

c. Add nodes to the cluster:

```
# Node 1
$   ./helix-admin.sh   --zkSvr   localhost:2181   --addNode   kie-cluster
 nodeOne:12345
# Node 2
$   ./helix-admin.sh   --zkSvr   localhost:2181   --addNode   kie-cluster
 nodeTwo:12346
...
```

Usually the number of nodes a in cluster equal the number of application servers in the cluster. The node names (`nodeOne:12345` , ...) can be changed as needed.

Note

`nodeOne:12345` is the unique identifier of the node, which will be referenced later on when configuring application servers. It is not a host and port number, but instead it is used to uniquely identify the logical node.

d. Add resources to the cluster:

```
$ ./helix-admin.sh --zkSvr localhost:2181 --addResource kie-cluster vfs-repo 1 LeaderStandby AUTO_REBALANCE
```

The resource name (`vfs-repo`) can be changed as needed.

e. Rebalance the cluster to initialize it:

```
$ ./helix-admin.sh --zkSvr localhost:2181 --rebalance kie-cluster vfs-repo 2
```

f. Start the Helix controller to manage the cluster:

```
$  ./run-helix-controller.sh --zkSvr localhost:2181 --cluster kie-cluster 2>&1 > /tmp/controller.log &
```

5. Configure the security domain correctly on the application server. For example on WildFly and JBoss EAP:

a. Edit the file `$JBOSS_HOME/domain/configuration/domain.xml`.

For simplicity sake, presume we use the default domain configuration which uses the profile `full` that defines two server nodes as part of `main-server-group`.

b. Locate the profile `full` and add a new security domain by copying the other security domain already defined there by default:

```
<security-domain name="kie-ide" cache-type="default">
    <authentication>
         <login-module code="Remoting" flag="optional">
            <module-option name="password-stacking" value="useFirstPass"/>
         </login-module>
         <login-module code="RealmDirect" flag="required">
            <module-option name="password-stacking" value="useFirstPass"/>
         </login-module>
```

```
    </authentication>
</security-domain>
```

Important

The security-domain name is a magic value.

6. Configure the *system properties* for the cluster on the application server. For example on WildFly and JBoss EAP:

   a. Edit the file `$JBOSS_HOME/domain/configuration/host.xml`.

   b. Locate the XML elements `server` that belong to the `main-server-group` and add the necessary system property.

   For example for nodeOne:

```
<system-properties>
  <property name="jboss.node.name" value="nodeOne" boot-time="false"/>
  <property name="org.uberfire.nio.git.dir" value="/tmp/kie/nodeone" boot-
time="false"/>
     <property   name="org.uberfire.metadata.index.dir"   value="/tmp/kie/
nodeone" boot-time="false"/>
    <property  name="org.uberfire.cluster.id"  value="kie-cluster"  boot-
time="false"/>
   <property  name="org.uberfire.cluster.zk"  value="localhost:2181"  boot-
time="false"/>
 <property name="org.uberfire.cluster.local.id" value="nodeOne_12345" boot-
time="false"/>
   <property  name="org.uberfire.cluster.vfs.lock"  value="vfs-repo"  boot-
time="false"/>
  <!-- If you're running both nodes on the same machine: -->
   <property  name="org.uberfire.nio.git.daemon.port"  value="9418"  boot-
time="false"/>
</system-properties>
```

   And for nodeTwo:

```
<system-properties>
  <property name="jboss.node.name" value="nodeTwo" boot-time="false"/>
  <property name="org.uberfire.nio.git.dir" value="/tmp/kie/nodetwo" boot-
time="false"/>
     <property   name="org.uberfire.metadata.index.dir"   value="/tmp/kie/
nodetwo" boot-time="false"/>
```

```
    <property name="org.uberfire.cluster.id" value="kie-cluster" boot-
time="false"/>
   <property name="org.uberfire.cluster.zk" value="localhost:2181" boot-
time="false"/>
 <property name="org.uberfire.cluster.local.id" value="nodeTwo_12346" boot-
time="false"/>
   <property name="org.uberfire.cluster.vfs.lock" value="vfs-repo" boot-
time="false"/>
  <!-- If you're running both nodes on the same machine: -->
    <property name="org.uberfire.nio.git.daemon.port" value="9419" boot-
time="false"/>
</system-properties>
```

Make sure the cluster, node and resource names match those configured in Helix.

11.1.2. jBPM clustering

In addition to the information above, jBPM clustering requires additional configuration. See *this blog post* [http://mswiderski.blogspot.com.br/2013/06/clustering-in-jbpm-v6.html] to configure the database etc correctly.

Chapter 12. Designer

Designer is a graphical web-based BPMN2 editor. It allows users to model and simulate executable BPMN2 processes. The main goal of Designe is to provide intuitive means to both technical and non-technical users to quickly create their executable business processes. This chapter intends to describe all feature Designer offers currently.

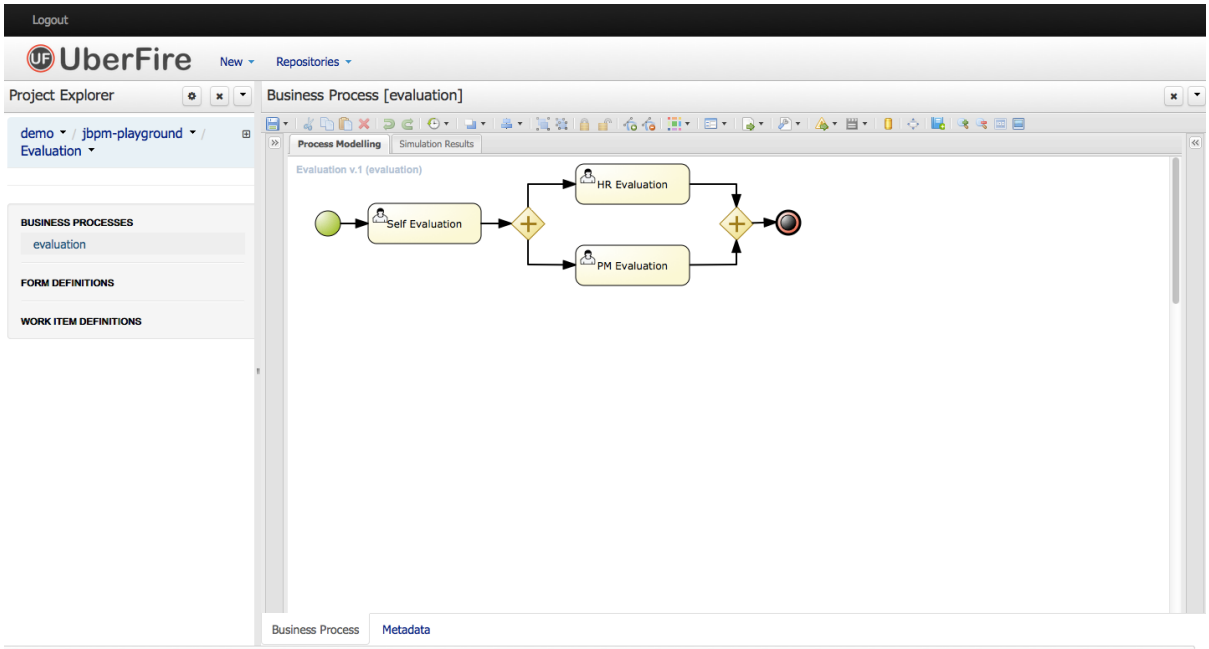


Figure 12.1. Designer

Designer targets the following business process modelling scenarios:

- View and/or edit existing BPMN2 processes: Designer allows you to open existing BPMN2 processes (for example created using the BPMN2 Eclipse editor or any other tooling that exports BPMN2 XML).
- Create fully executable BPMN2 processes: A user can create a new BPMN2 process in the Designer and use the editing capabilities (drag and drop and filling in properties in the properties panel) to fill in the details. This for example allows business users to create complete business processes all inside a a browser. The integration with Drools Guvnor allows for your business processes as wells as other business assets such as business rules, process forms/images, etc. to be stored and versioned inside a content repository.
- View and/or edit Human Task forms during process modelling (using the in-line form editor or the Form Modeller).
- Simulate your business process models. Busines Process Simulation is based on the BPSIM 1.0 specification.

Designer supports all BPMN2 elements that are also supported by jBPM as well as all jBPM-specific BPMN2 extension elements and attributes.

12.1. Designer UI Explained

Designer UI is composed of a number of sections as shown below:

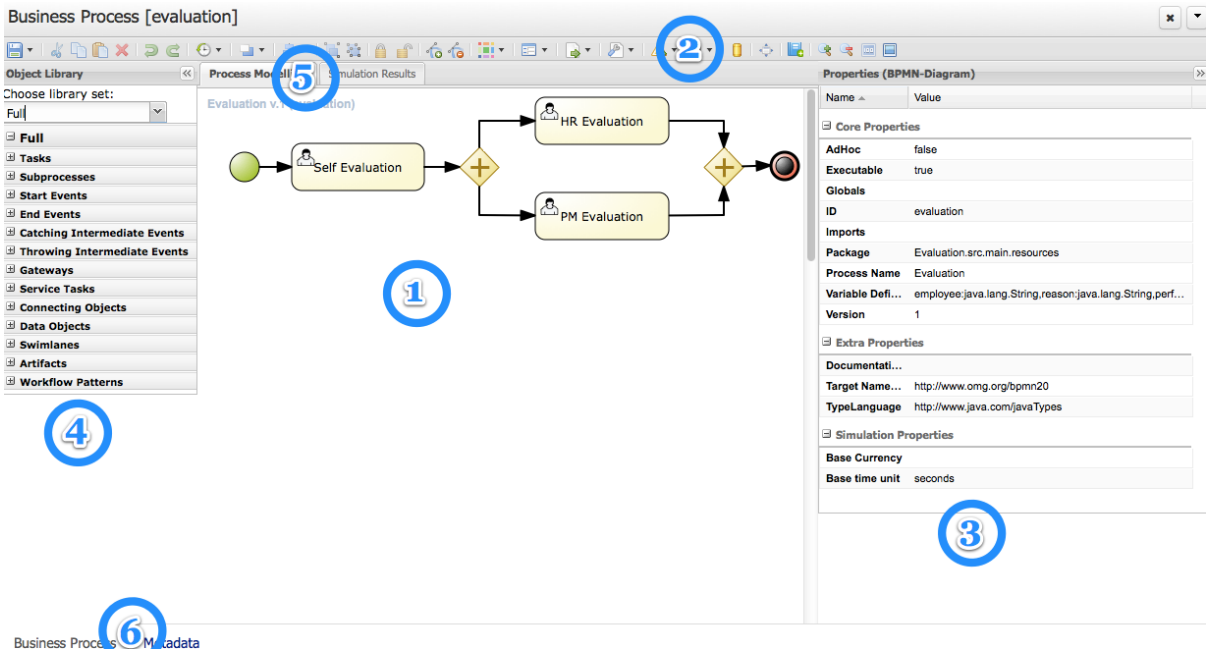


Figure 12.2. Designer sections

- (1) Modelling Canvas - this is your process drawing board. After dropping different shapes onto the canvas, you can move them around, connect them, etc. Clicking on a shape on the canvas allows you to set its properties in the expandable Properties Window (3) (as well as create connecting shapes and morph the shape into other shapes).
- (2) Toolbar - the toolbar contains a vast number of functions offered by Designer (described later). These includes operations that can be performed on shapes present on the Canvas. Individual operations are disabled or enabled depending on what is selected. For example, if no shapes are selected, the Cut/Paste/Delete operations are disabled, and become enabled once you select a shape. Hovering over the icons in the Toolbar displays the description text of the operation.
- (3) Properties Panel - this expandable section on the right side of Designer allows you to set both process and shape properties. It is divided in four sections, namely "Core properties", and "Extra Properties, "Graphical Settings", and "Simulation Properties" are is expandable. When clicking on a shape in the Canvas, this panel is reloaded to show properties specific to the shape type. If you click on the canvas itself (not on a shape) the section shows your general process properties.

- (4) Object Repository Panel - the expandable section on the left side of Designer shows the jBPM BPMN2 (default) shape repository tree. It includes all shapes of the jBPM BPMN2 stencil set which can be used to assemble your processes. If you expand each section sub-group you can see the BPMN2 elements that can be placed onto the Designer Canvas (1) by dragging and dropping the shape onto it.
- (5) View Tabs - currently Designer offers functionality tabs for Process Modelling and Simulation. Process Modelling is the default tab. When users run process simulation, its results are presented in the Simulation tab.
- (6) Info Tabls - On the bottom Designer shows two different Info tabs. The Business Process tab includes the process modeling while the Metadata tab displays the process metadata such as created by and last modified information.

12.2. Getting started with Modelling

The Object Repository panel provide means for users to select and drag/drop BPMN2 shapes onto the modelling canvas. Shapes are divided into sections as shown below:

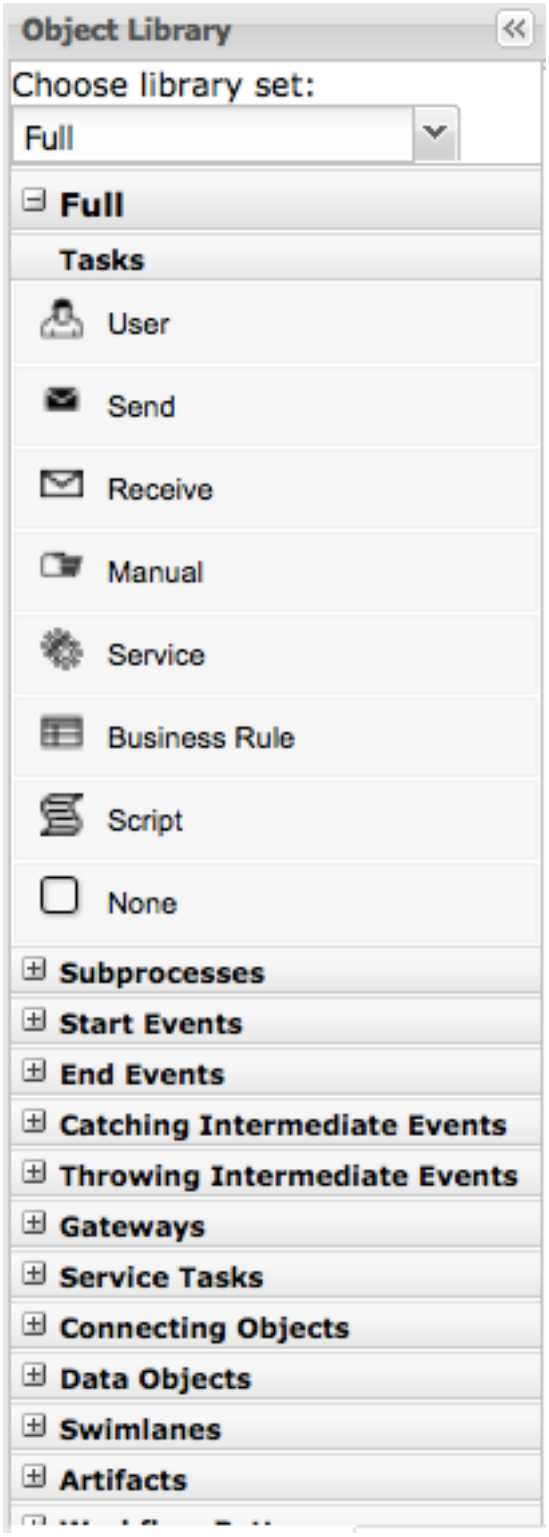


Figure 12.3. Object Repository

Once a shape is dropped onto the canvas users have a much faster way of continuing modelling without having to go back to the Object Repository panel. This is realized through the shape morphing menu which is presented when a shape on the drawing canvas is clicked on. This menu

allows users to either select a connecting shape (next shape) or morph the selected node into another node type. In addition this menu includes means to store the shape name as a dictionary item (explained later), view the specific BPMN2 code of the selected shape, as well as create/edit the task form (in the case of user tasks only).

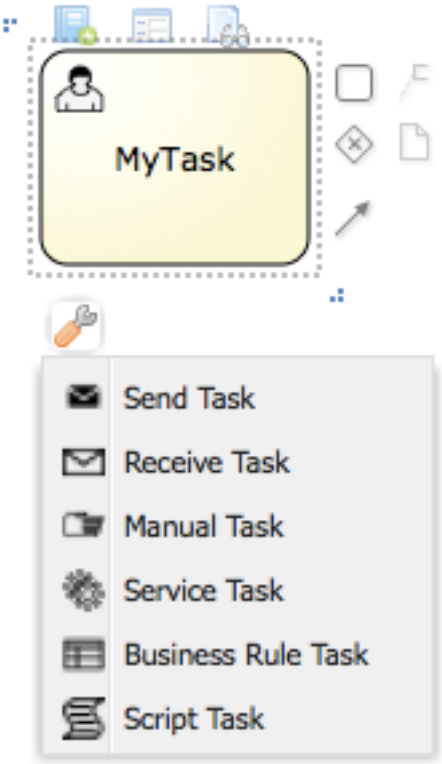


Figure 12.4. Morphing Menu for shapes

When connecting shapes Designer applies connection rules that follow the BPMN2 specification. The connection shapes presented in the morphing menu only show shapes that are allowed to be connections. Similarly same rules are applied when dropping a shape from the Object Library from the canvas and trying to connect an existing shape to it. Additional connection rules for boundary events are also available (explained later) and applied when for example moving an intermediate event node onto the edge of a task node.

Users can give names to every shape on the drawing canvas. This is done by double-clicking onto the shape as shown below.

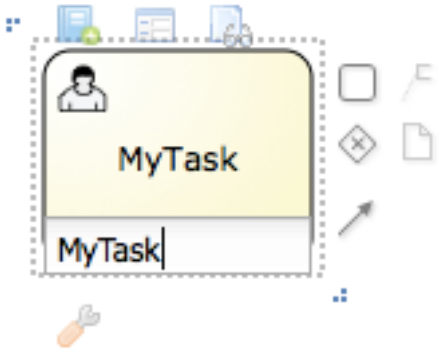


Figure 12.5. Naming a shape

The name of a shape can be pulled from the Process Dictionary. If terms are set up in the dictionary, auto-complete can be used for the node names:

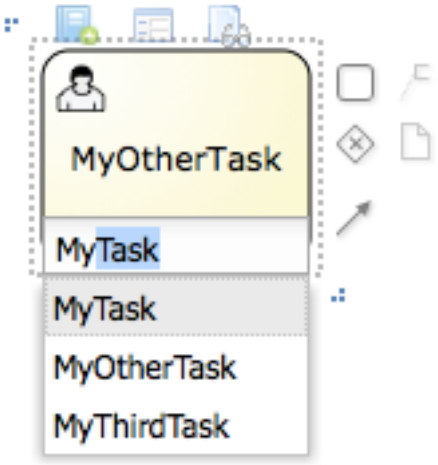


Figure 12.6. Name auto-completion from dictionary

Designer also shows three buttons on top of a clicked shape as shown below.

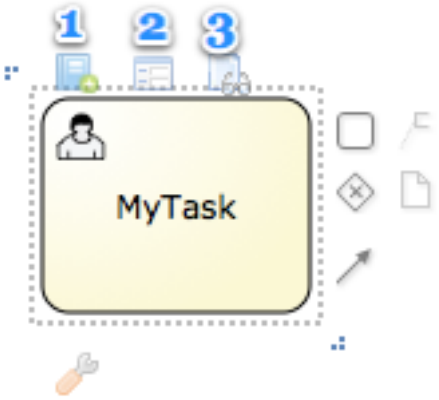


Figure 12.7. Extra in-line options

These include:

- (1) Add To Dictionary - this option allows users to add the name of the task to the Process Dictionary (explained in more details later)
- (2) Edit Task Form - allows users to create/edit the Task Form. This option is only available for User Tasks
- (3) View shape sources - shows the BPMN2 for this particular shape only.

The section should get you started with creating simple business process models by dragging/dropping BPMN2 shapes onto the drawing canvas. Next sections will dive deeper into many other aspects of Designer.

12.3. Designer Toolbar

The Designer toolbar contains many different functions which can be used during process modelling.

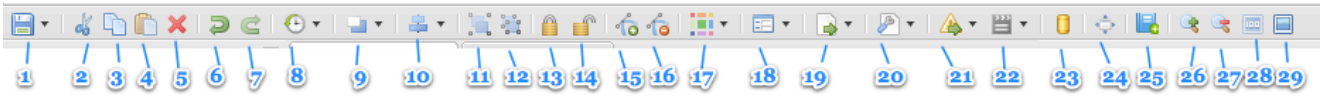


Figure 12.8. Toolbar Buttons

We will now go through each of the buttons in the Designer Toolbar and give a brief overview of what it does.

(1) Save - allows users to save, copy, rename and delete the business process model. In addition users can turn on auto-save which will automatically save the business process within a defined time interval.

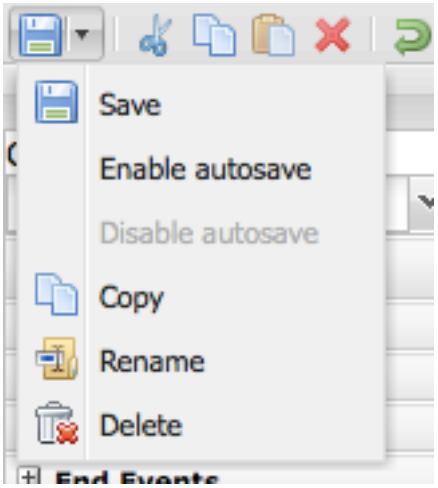


Figure 12.9. Save Button

(2) Cut - enabled when a portion of the model is selected.

(3) Copy - enabled when a portion of the model is selected.

(4) Paste - paste the copied portion of the model onto the drawing board.

(5) Delete - enabled when there is a portion of the model is selected and removes it.

(6, 7) Undo/Redo - undo the last performed operation on the drawing canvas.

(8) Local History - local history allows continuous storage of your business process onto your browsers internal storage. Stored version of the business process can persist internet outages or browser crashes so your work will not be lost. This feature is disabled by default and must be enabled by users. Once local history has been enabled users are able to view all previously stored snapshots of their business model, clear local history, configure the snapshot interval, or disable local history. Note that local history will only take a snapshot of your business process on the set storing interval if there were some changes done in the model. If at the end of the snapshot interval Designer detects that there were no changes since the last local history save, no new snapshot will be created.

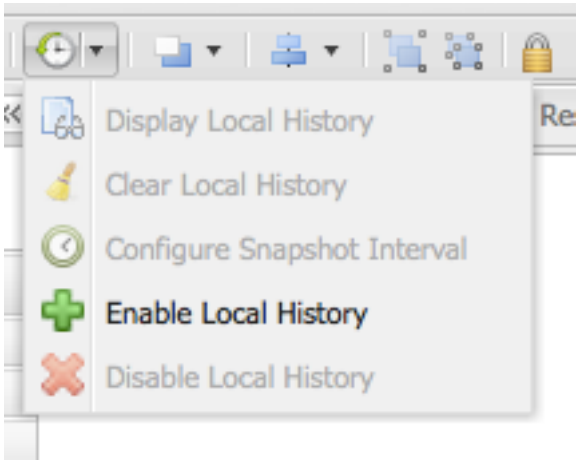


Figure 12.10. Local History

The Local History results screen allows users to select a stored snapshot of the model and view its process image, and restore it back onto their drawing board.

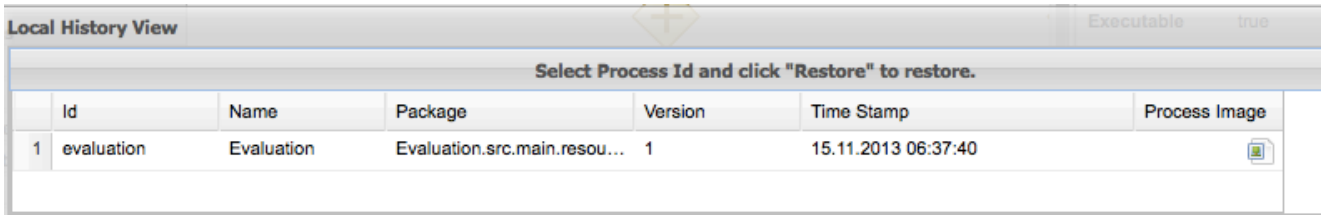


Figure 12.11. Local History Sample Results

(9) Object positioning - allows users to position one or more nodes in the business. Note that at last one shape must be selected first, otherwise these options are disable. Contains options "Bring to Front", "Bring to back", "Bring forward", and "Bring Backward"

(10) Alignment: enabled when a portion of the model is selected. Includes options "Align Bottom", "Align Middle", "Align Top", "Align Left", "Align Center", "Align Right", and "Align Same Size".

(11, 12) Group and Ungroup - allows grouping and ungrouping of selected shapes on the drawing board.

(13, 14) Locking and Unlocking - allows parts of the business model to be locked and unlocked. Locked parts of the model cannot be edited (visual display and properties are both locked). Locked nodes are displayed in a light blue color. This feature fosters collaboration of process modelling by allowing users to set parts of their model as "completed" and preventing any further changes to that portion. Other parts of the model can continue to be edited.

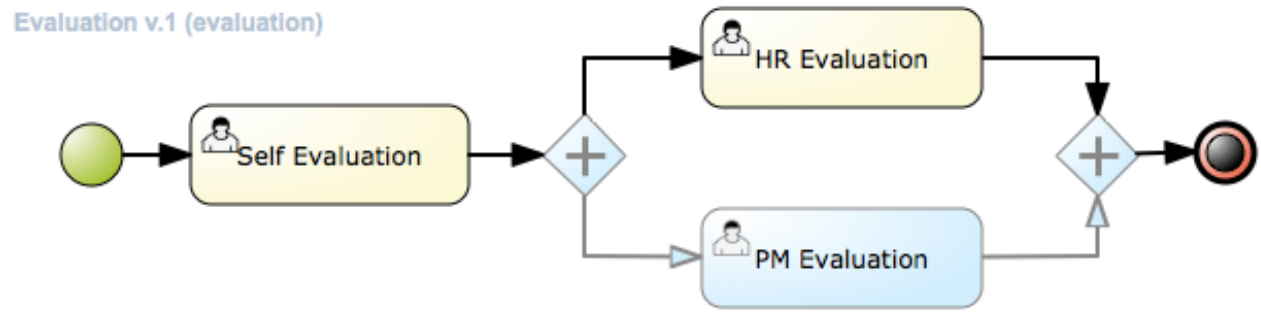


Figure 12.12. Locked Nodes

(15, 16) Add/Remove Docker - this allows users to add or remove Dockers, or edge points, to sequence flows in the model. Enables when a sequence flow (connector) is selected. It allows users to create very customized connection points from one shape to another. Users can add and remove as many dockers as they would like on a single sequence flow.

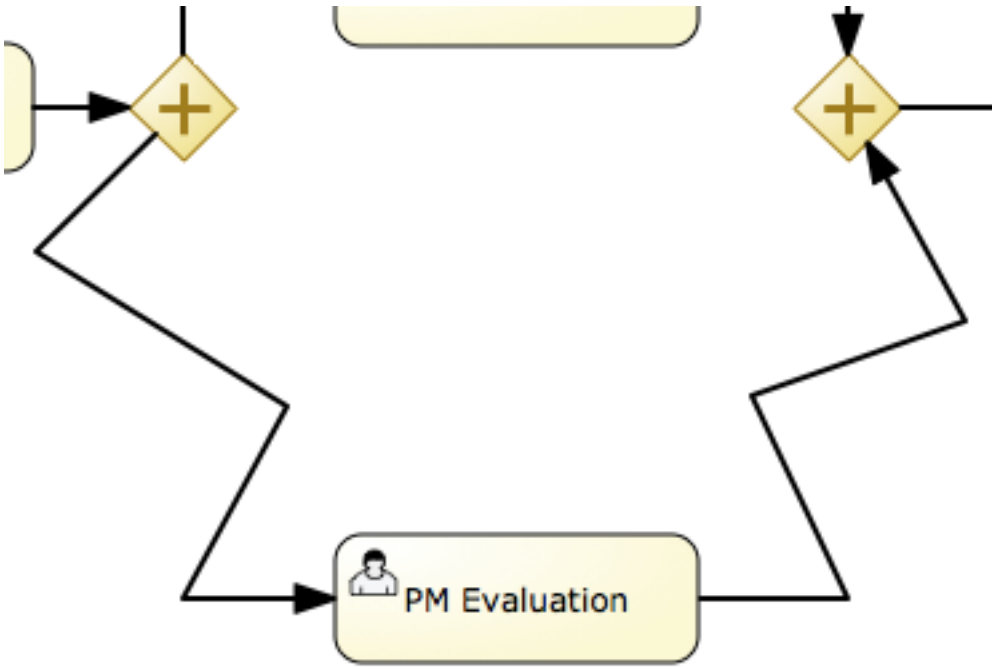


Figure 12.13. Adding dockers to a sequence flow

(17) Color Themes - Colors are a big part or process modelling as they help with expressing intent as well as help allowing visually impaired users to better view the model. Designer provides two default color themes out of the box named "jBPM" and "High Contrast". The jBPM theme is the default theme used for all new business processes created. Users can switch color themes and the changes will be applied to all nodes that are currently on the model, as well as any new shapes added. Users have the ability to add new custom color themes by adding their own definitions in the Designer themes.json file. Color theme selection is persisted over browser close or possible crash/internet loss.

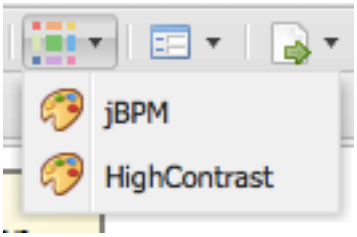


Figure 12.14. Color Themes selection

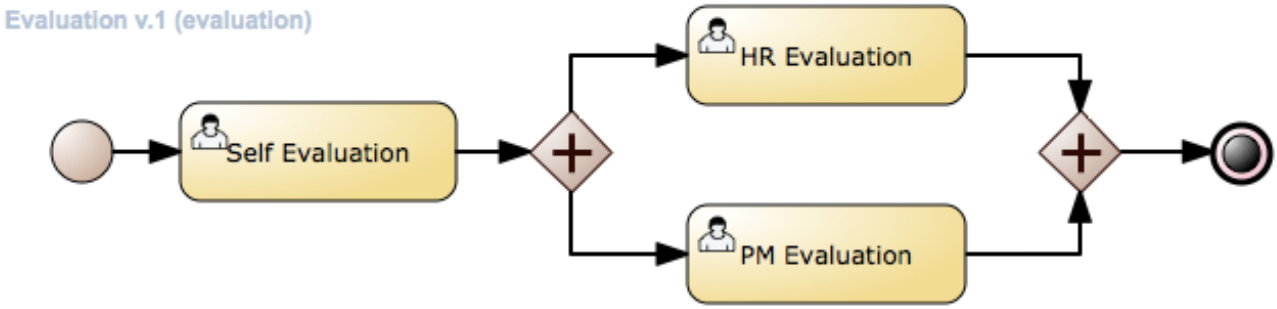


Figure 12.15. Switching to High Contrast Color Theme

(18) Process and Task forms - here users have the ability to generate/edit process and task forms. When no user task is selected the default enabled options are "Edit Process Form" and "Generate all Forms". Generate all forms will apply the current model information such as process variables, data objects, and the user tasks data input/output parameters and associations to generate default executable input forms. Upon editing a process and task form, users have the choice between two form editors, the jBPM Form Modeler, and the Designer in-line meta editor. The Designer meta editor is targeted more to technical users as it is text based with the ability for live preview. When the user selects an user task in the model, the "Edit Task Form" and "Generate Task Form" options are enabled which allow users to edit the particular task form, or choose to apply the same generation logic to create a task form for the selected task only. Users have the ability to extend the default form generation templates in designer to create fully customized templates. Node that in the case of the Designer meta editor for forms, generating forms will overwrite existing forms for the process and user tasks. In the case of Form Modeler form generation, a merging algorithm is applied when generating.

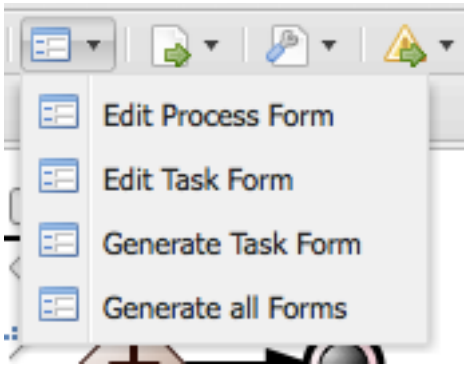


Figure 12.16. Form generation selection

When selecting a task, users have the ability to edit the selected tasks form via the form button shown above the user task node.

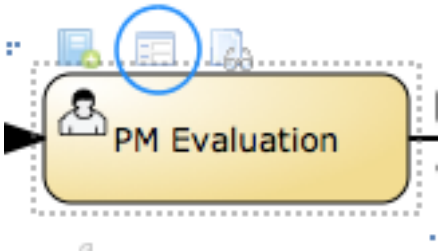


Figure 12.17. In-line task editing

When editing forms, users are asked to choose between the Form Modeler and the Designer in-line meta editor. If the user selects Form Modeler the form is shown in a new asset tab separately from Designer. Designer meta editor is in-line and part of the Designer application.

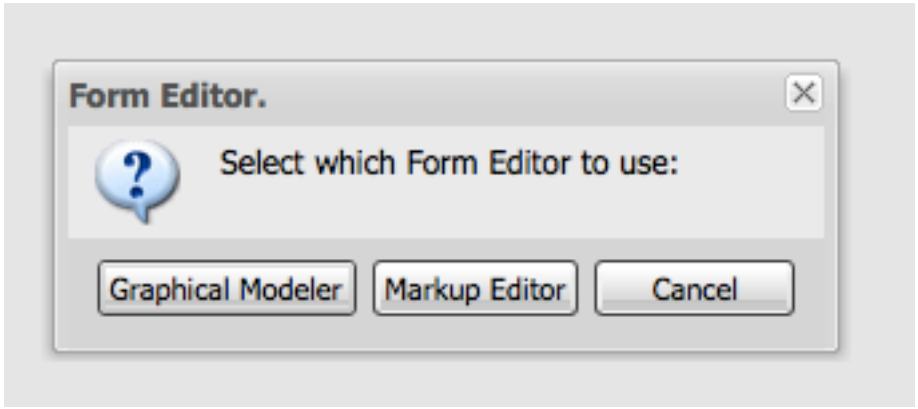


Figure 12.18. Form Editor Selection

The Designer in-line meta form editor is a powerful text-based editor with a live preview feature as well as auto-completion on process variables and user task data inputs/outputs.

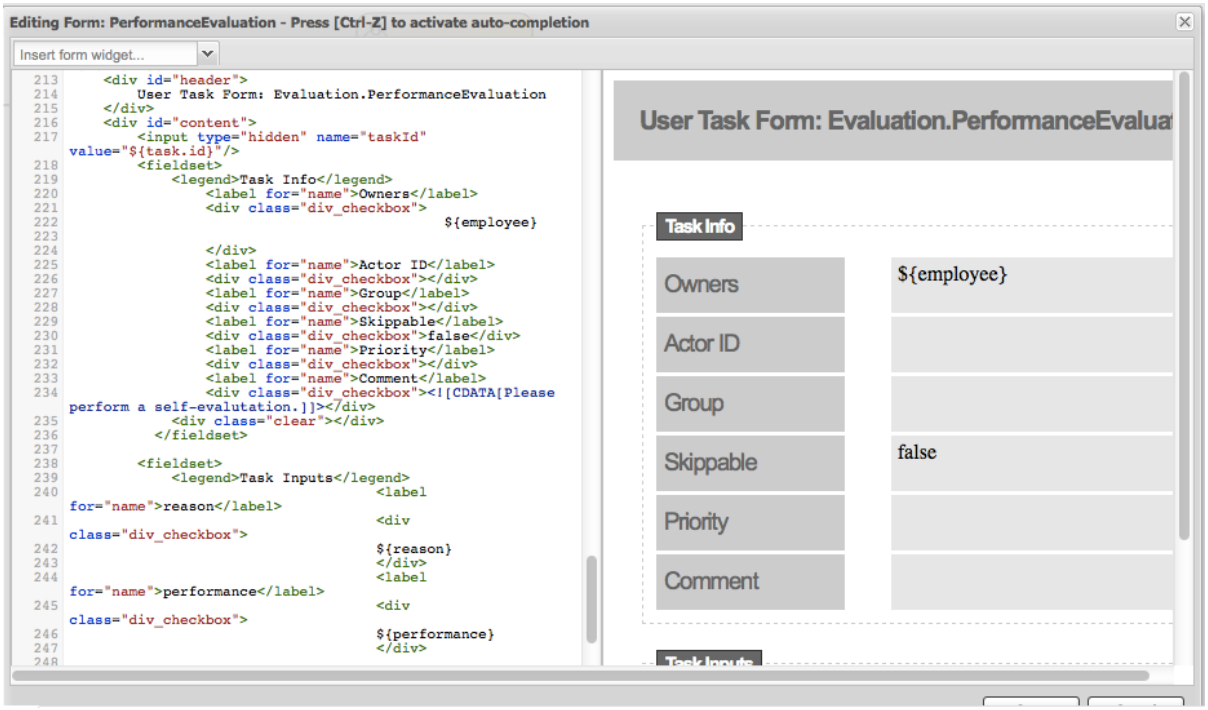


Figure 12.19. Designer in-line form meta editor with live-preview

(19) Process Information Sharing - this section includes many functions that help with sharing information of your model. These include:

- Share process image - generates a stand-alone HTML image tag which contains a Base64 encoded image source of the current model on the canvas. This link can be shared to team members or other parties and embedded in any HTML content or email that allows HTML content embedding.
- Share process PDF - generates a stand-alone HTML object tag which contains a Base64 encoded PDF source of the current model on the canvas. This can similarly be shared and embedded in any HTML content.
- Download process PNG - generates a PNG image of the current process on the drawing board which users can download and share.
- Download process PDF - generates a PDF of the current process on the drawing board which can be downloaded and shared.
- View Process Sources - displays the current process sources in various formats, namely BPMN2, JSON, SVG, and ERDF. Also has the option to download the BPMN2 sources.

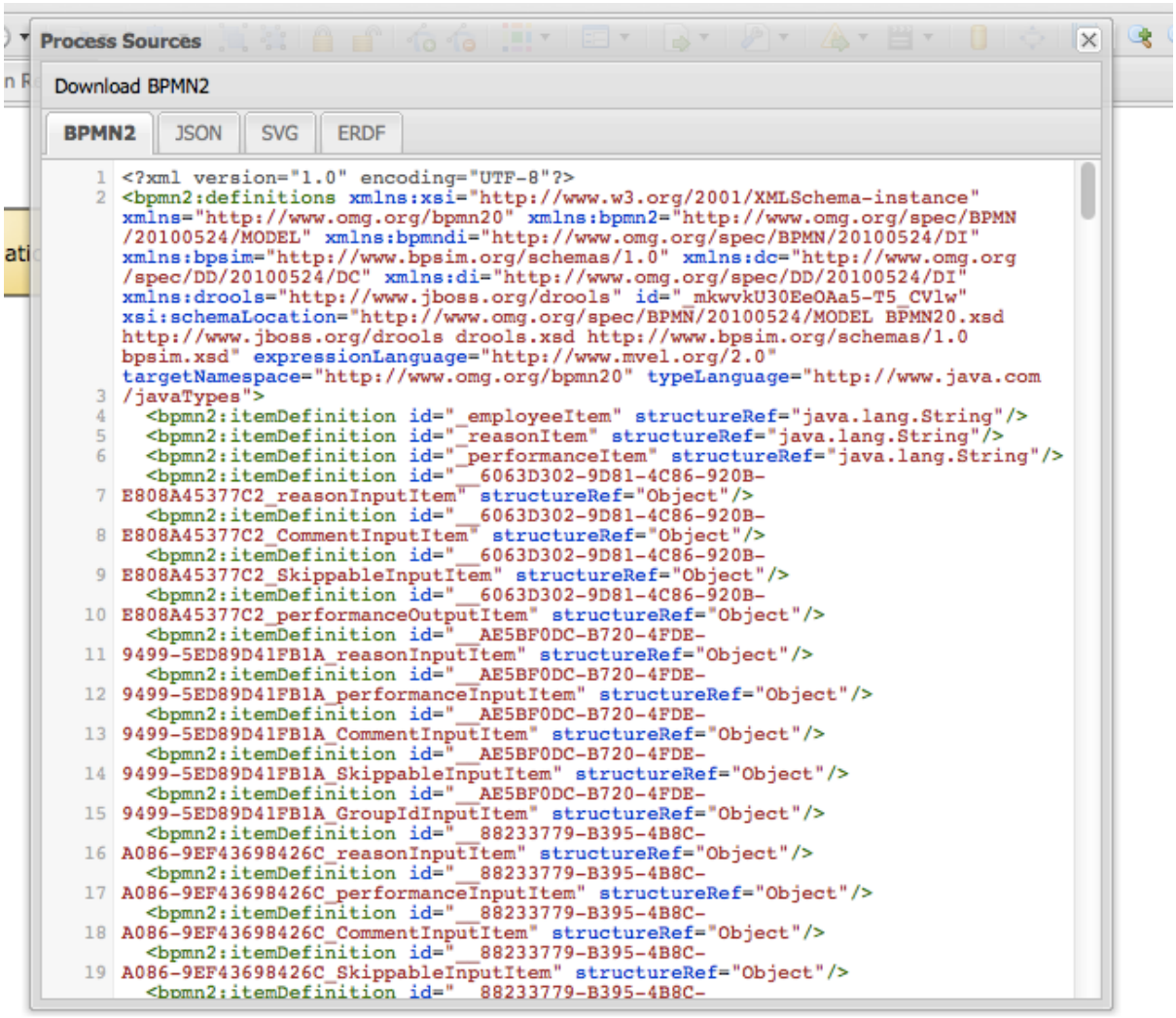


Figure 12.20. Process Sources View

(20) Extra tooling - this section allows users to import their existing BPMN2 processes into designer as well as be able to migrate their old jPDL based processes to BPMN2. For BPMN2 or JSON imports users can choose to add the import ontop of the existing model on the drawing board or choose to replace the current one with the import.

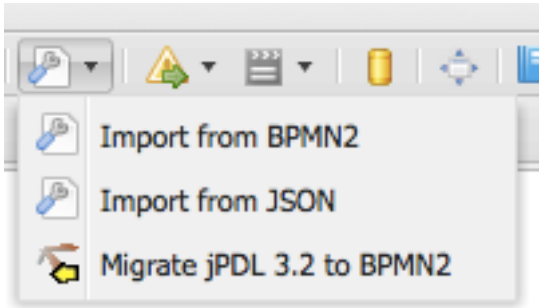


Figure 12.21. Extra tooling section

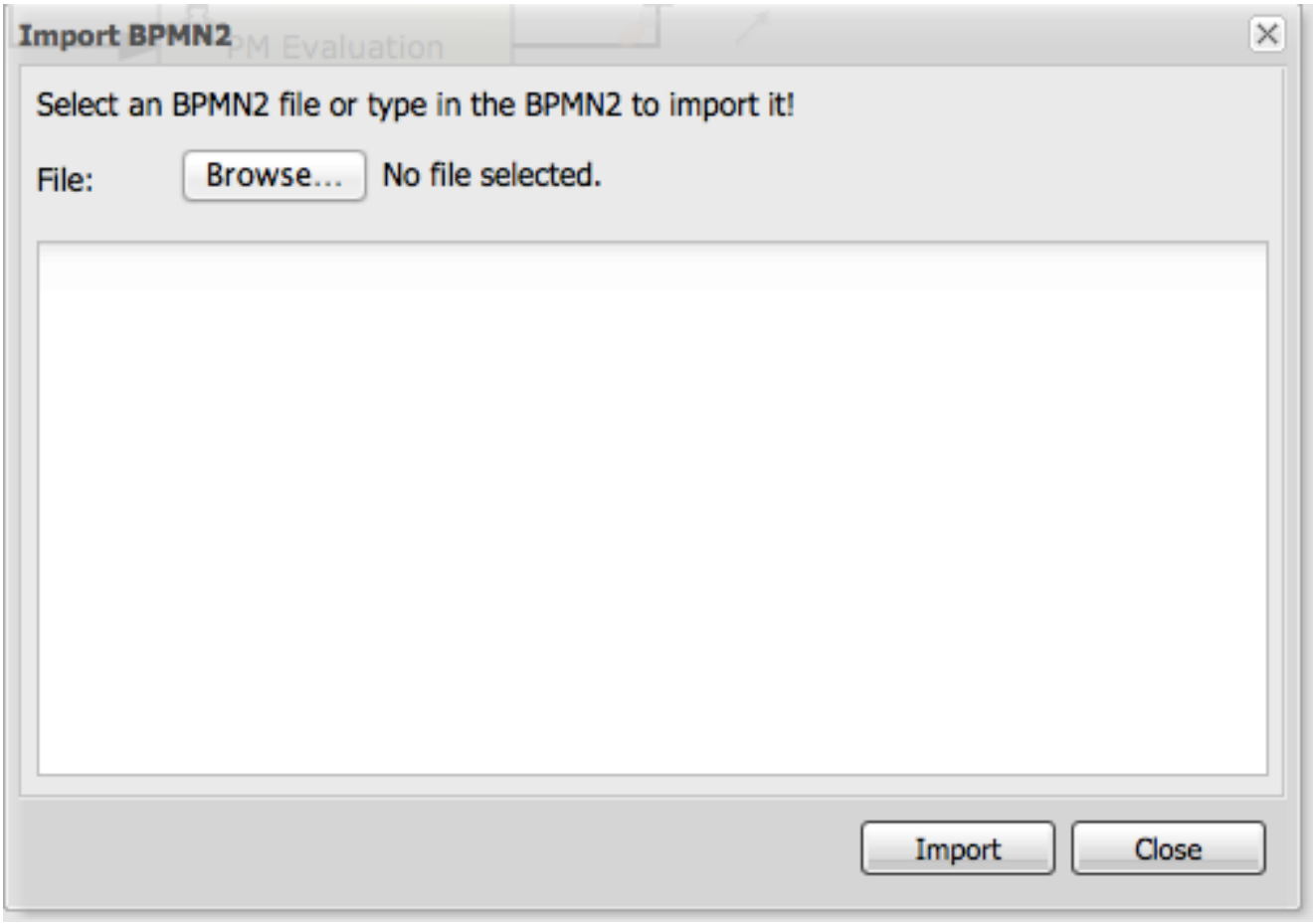


Figure 12.22. Import existing BPMN2 panel

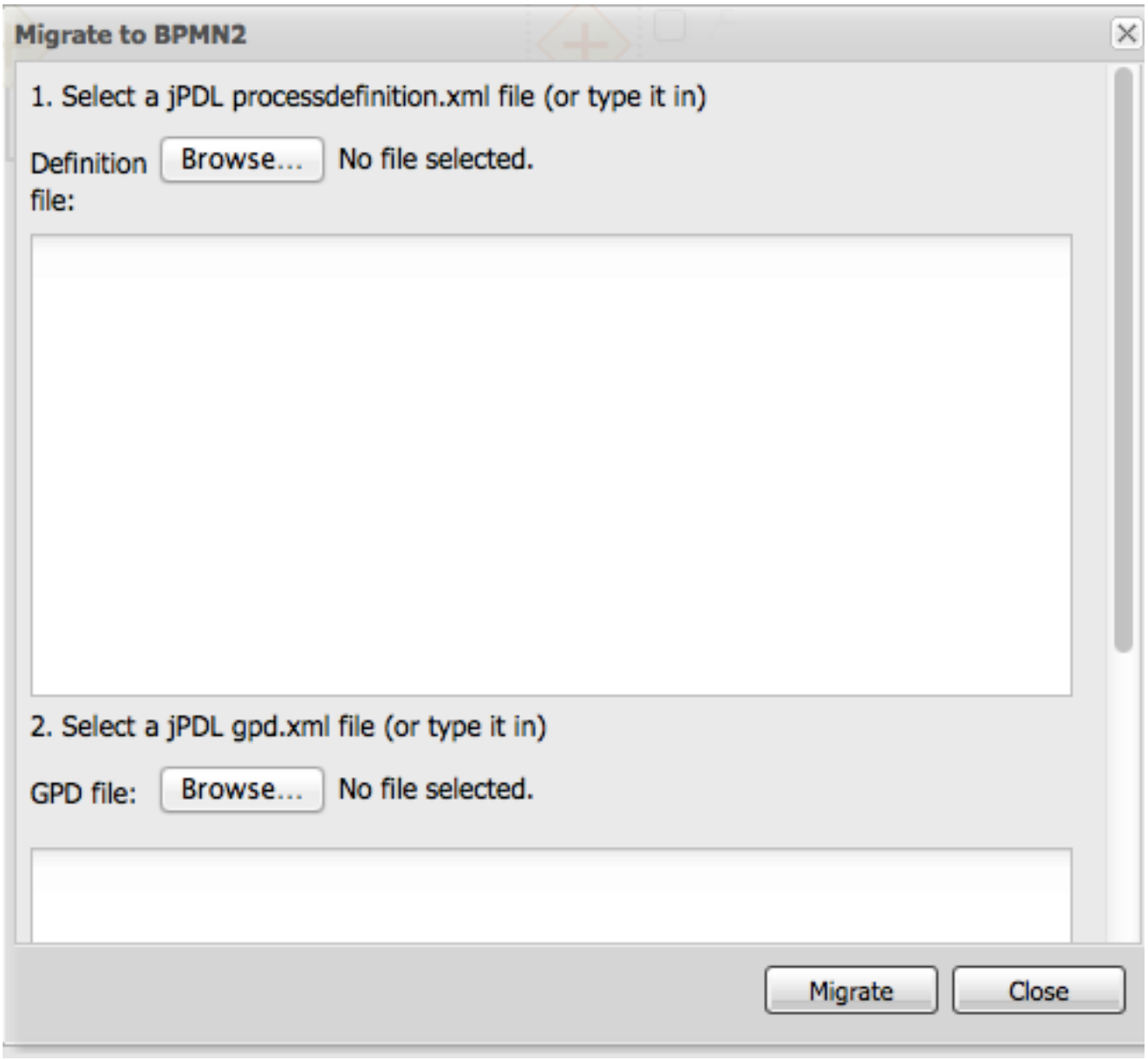


Figure 12.23. Process Migration panel

(21) Visual Validation - Designer includes over 100 validation checks and this list is growing. It allows users to view validation issues in real-time as they are modelling their business process. Users can enable visual validation, disable it, as well as view all validation issues at once. If Visual Validation is turned on, Designer with set the shape border of shapes that do not pass validation to red color. Users can then click on that particular shape to view the validation issues for that particular shape only. Alternatively "View All Issues" present a combined list of all validation errors currently found. Note that you do not have to periodically save your business process in order for validation to update. It will do so on its own short intervals during modelling. Users can extend the list of validation issues to include their own types of validation on certain elements of their business model.

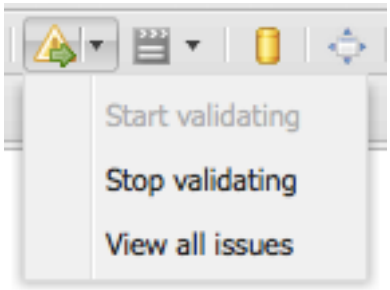


Figure 12.24. Visual Validation Toolbar

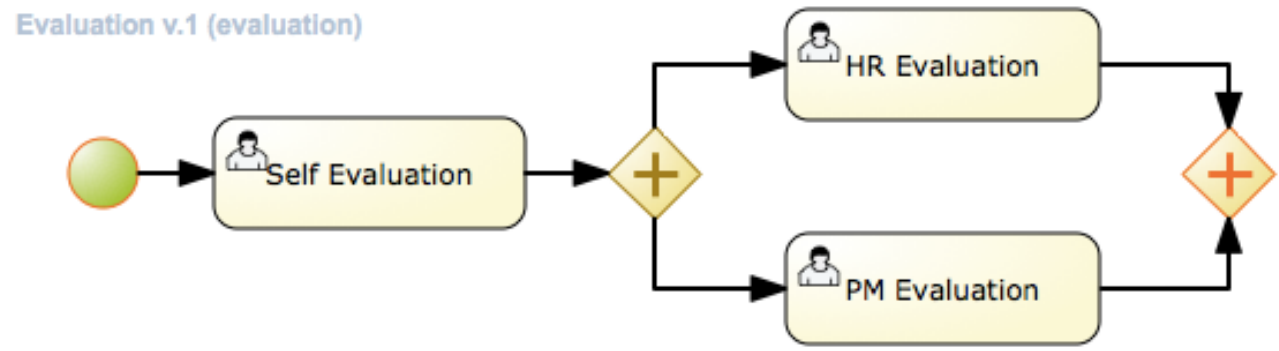


Figure 12.25. Shapes with validation errors displayed with red border

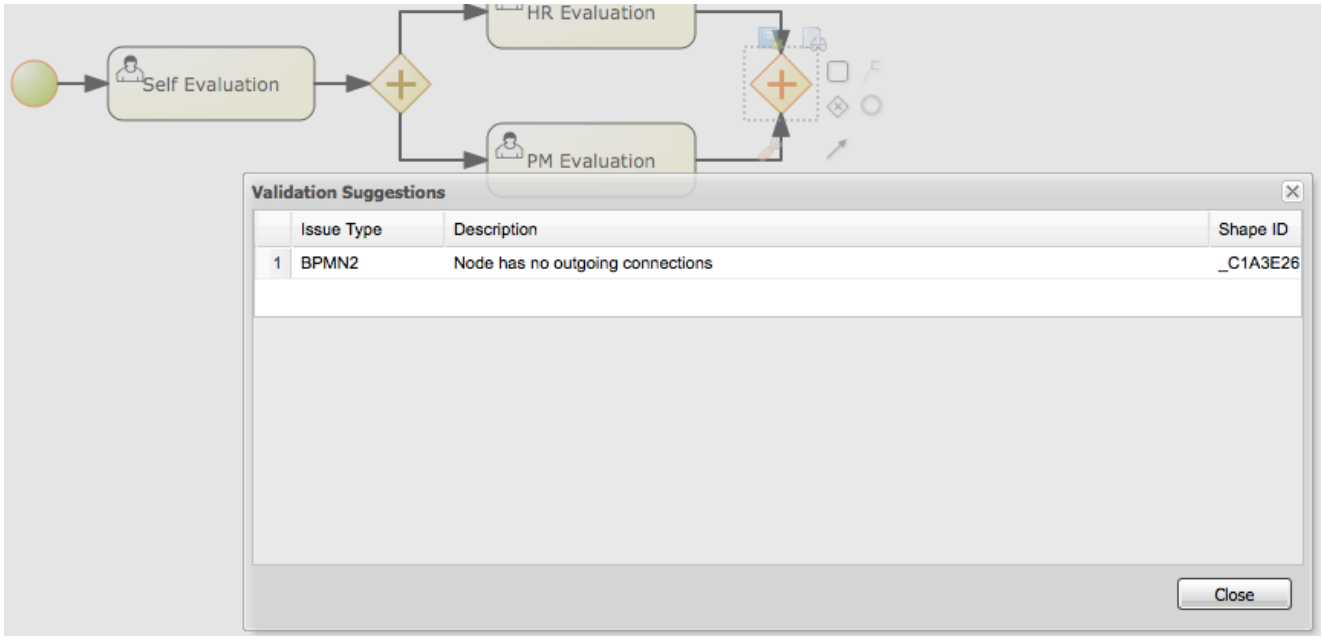


Figure 12.26. Single shape validation issues display

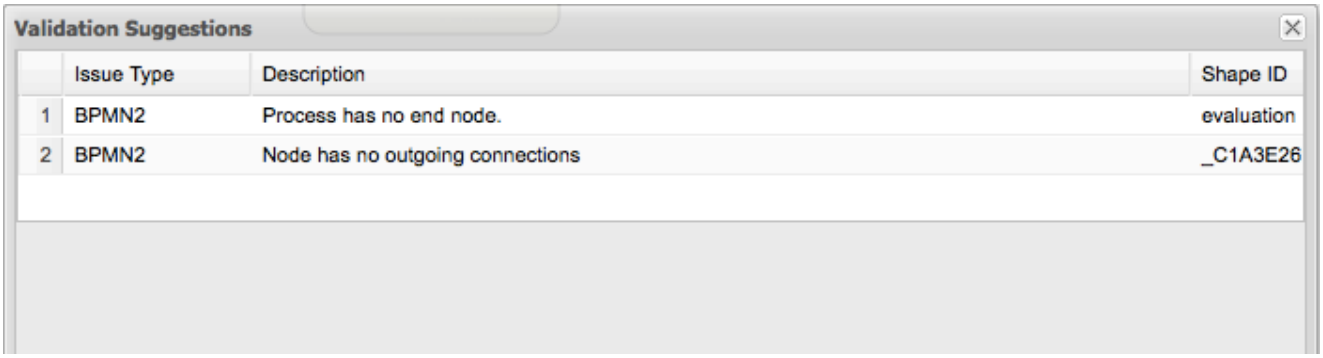


Figure 12.27. View all issues validation display

(22) Process Simulation - Business Process Simulation deals with statistical analysis of process models over time. It's main goals include

- Pre-execution and post-execution optimization
- Reducing the risk of change in business processes
- Predict business process performance
- Foster continuous improvements of performance, quality and resource utilization of business processes

Designer includes a powerful simulation engine which is based on jBPM and Drools and a graphical user interface to view and interpret simulation results. In addition users are able to view all process paths included in their current model on the drawing board. Designer Process Simulation is based on the BPSim 1.0 specification. Details of Process Simulation capabilities in Designer are can be found in its Simulation documentation chapter. Here we just give a brief overview of all features it contains.

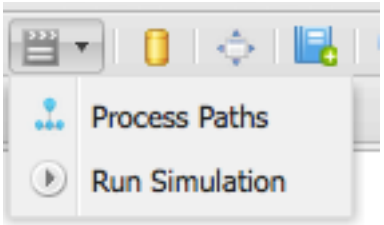


Figure 12.28. Simulation tooling section

When selecting Process Paths, the simulation engine find all possible paths in the business model. Users can choose certain found paths and choose to display them. The chosen path is marked with given colors as shown below.

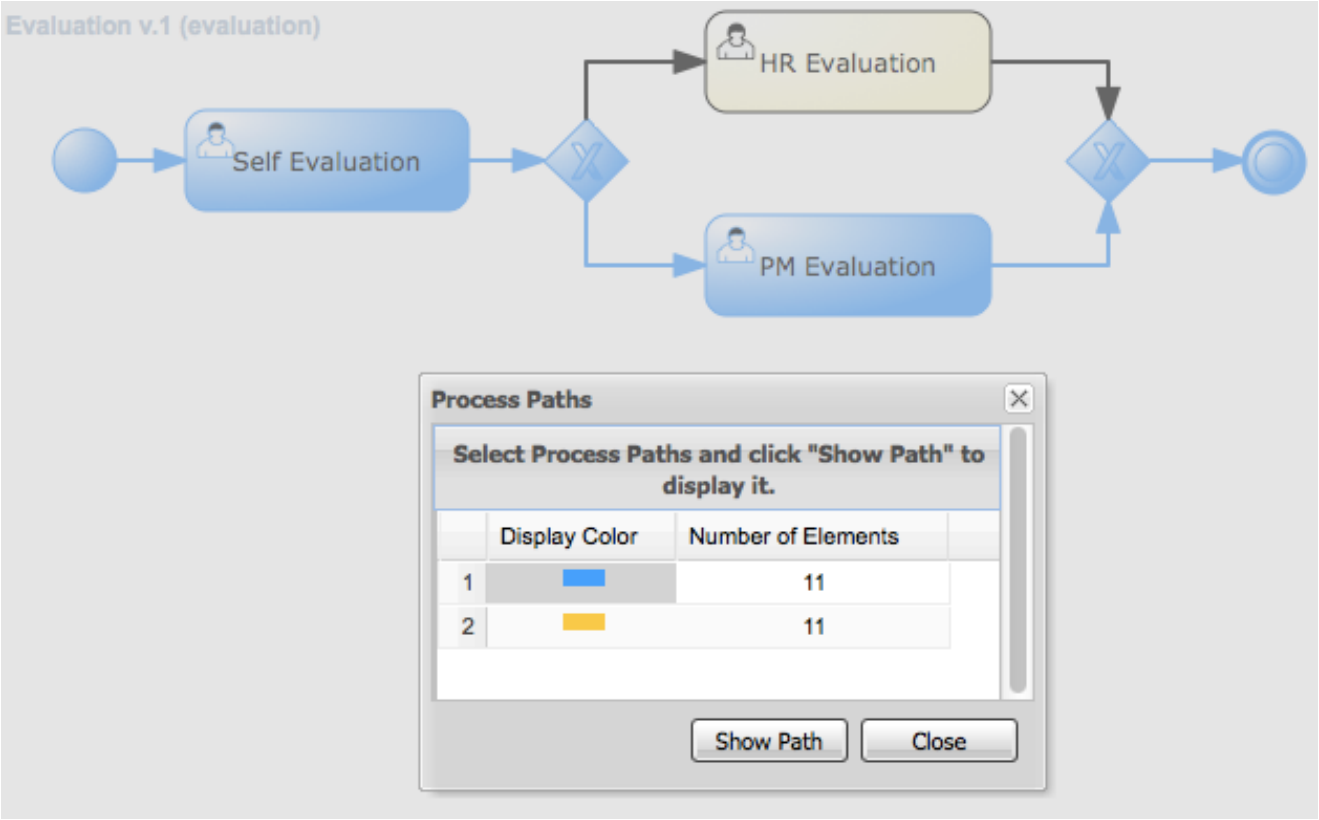


Figure 12.29. View all issues validation display

When selecting "Run Simulation", users have to enter in simulation runtime properties. These include the number of instances of this business process to simulate and the interval time and units. This interval is the time in-between consecutive simulation.

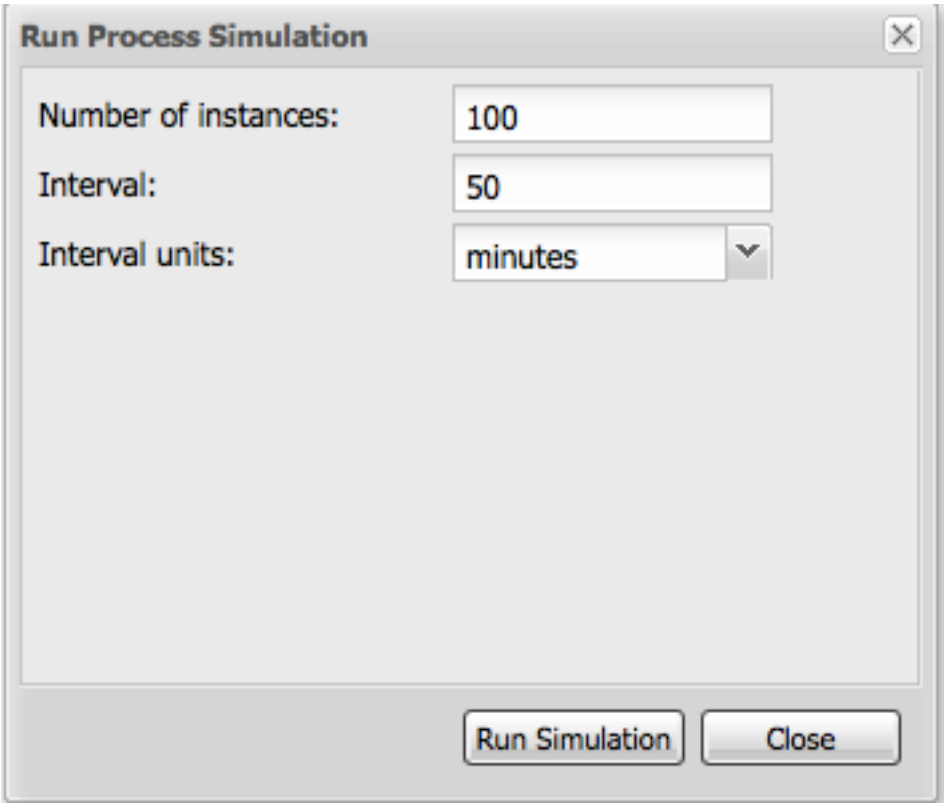


Figure 12.30. Simulation runtime properties

Each shape on the drawing board includes Simulation properties (properties panel) where users can set numerous simulation properties for that particular shape. More info on each of these properties can be found in the Simulation chapter of the documentation. Designer pre-sets some defaults for new processes, which allows business processes to be simulated by default without any modifications of these properties. Note however that the results of the default settings may not be optimal or targeted for the users particular needs.

Figure 12.31. Simulation properties for shapes

Once the simulation runtime has completed, users are shown the simulation results in the "Simulation Results" tab of Designer. The results default to the process results. Users can switch to

results for each particular shape in their business process to see more specific detauls. In addition, the results contain process paths simulation results for each path in the business process.

Figure 12.32. Sample simulation results

Designer simulation presents the users with many different chart types. These include:

- Process results: Execution times, Activity instances, Total cost
- Human Task results: Execution times, Resource Utilization, Resource Cost
- All other nodes: Execution times
- Process Paths: Path Execution

The below image shows a number of possible chart types users can view after process simulation has completed.

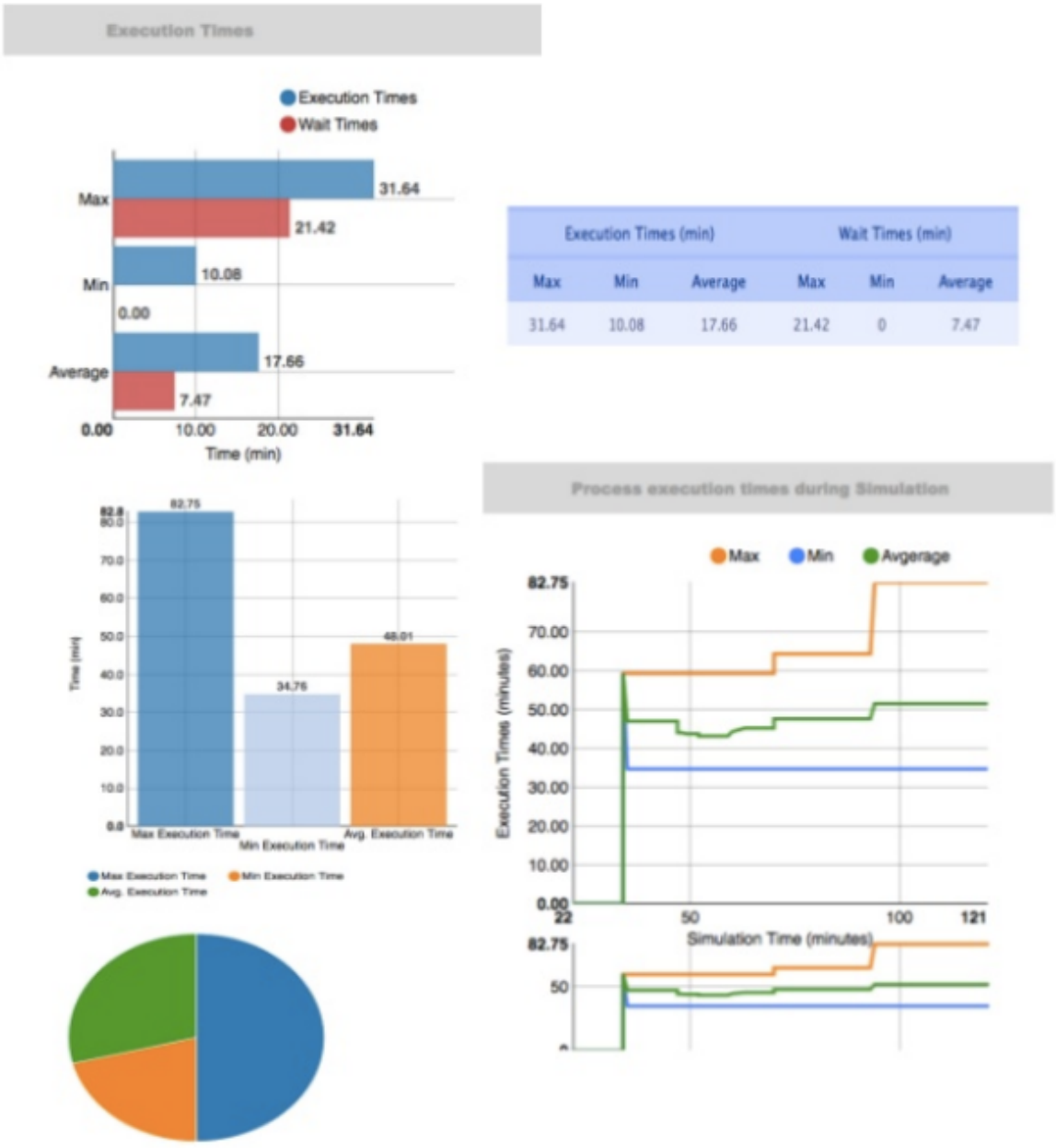


Figure 12.33. Types of simulation results charts

In addition to the chart results, Designer simulation also offers a full timeline display that includes all details of what happened during simulation. This timeline allows users to navigate through each event that happened during process simulation and select a particular node to display results at that particular point in time.

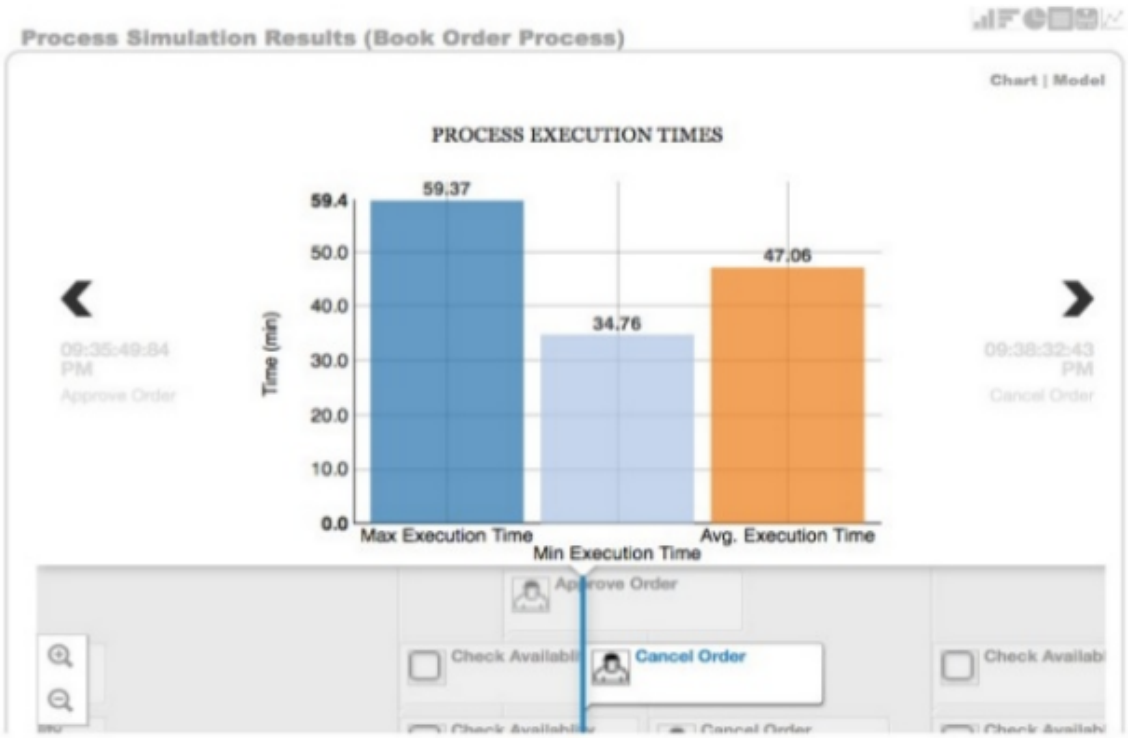


Figure 12.34. Simulation timeline

The simulation timeline can be switched to the Model view. This view displays the process model with the currently selected node in the timeline highlighted. The highlighted node displays the simulation results at that particular point in time of the simulation.

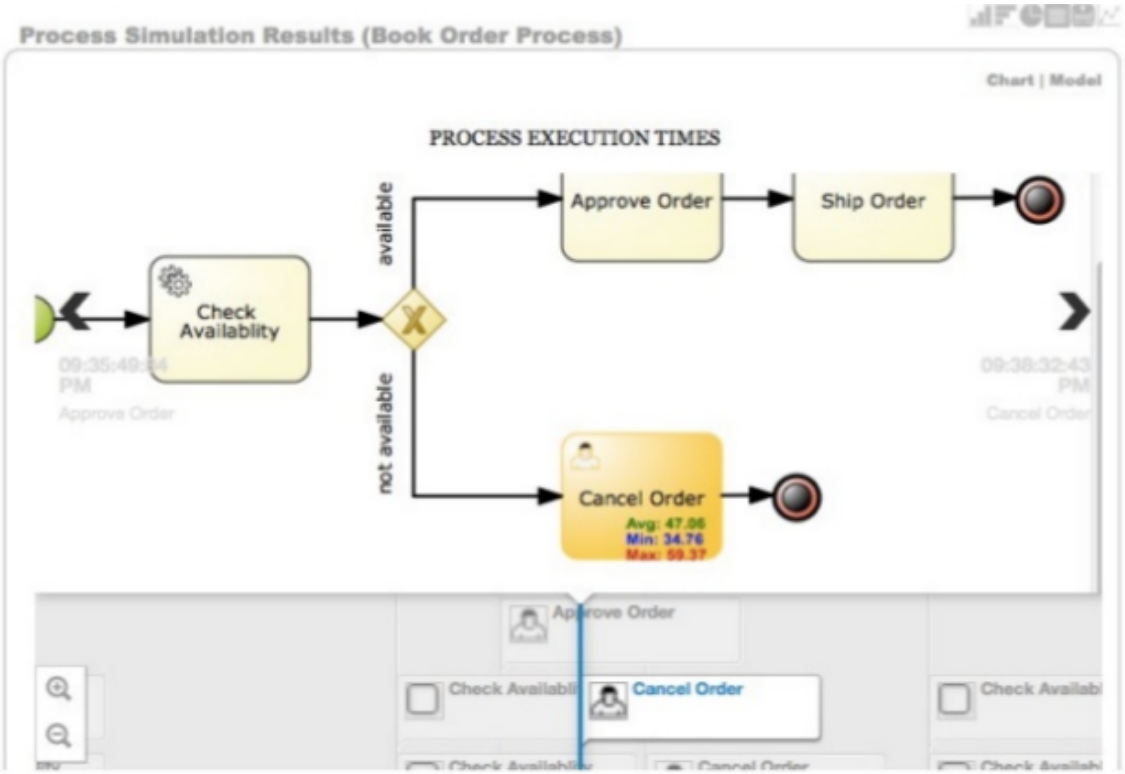


Figure 12.35. Simulation timeline model view

Path execution results shows a chart displaying the chosen path as well as path instance execution details.

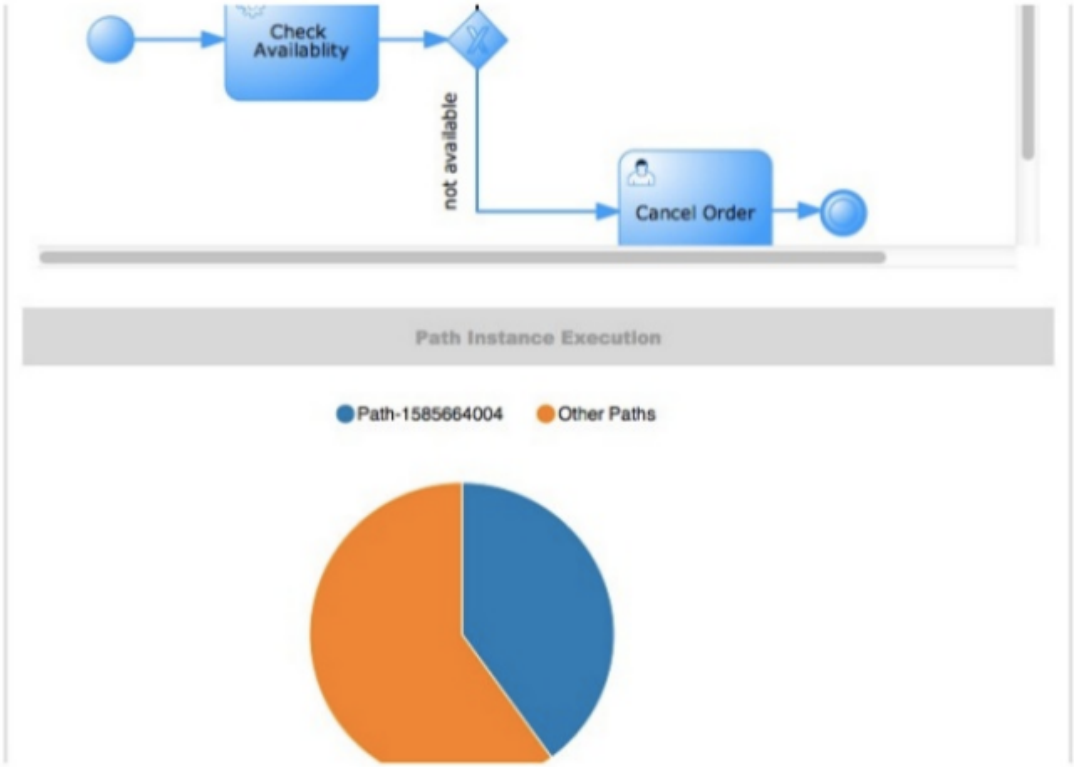


Figure 12.36. Path execution details

(23) Service Repository - this feature allows users to connect to an existing service tasks repository to install service tasks into their list of available shapes. Mode default of this can be found in the Service Repository chapter of the documentation. Users have to enter the URL to the existing service repository and then can install the available service nodes by double-clicking on a particular results row.

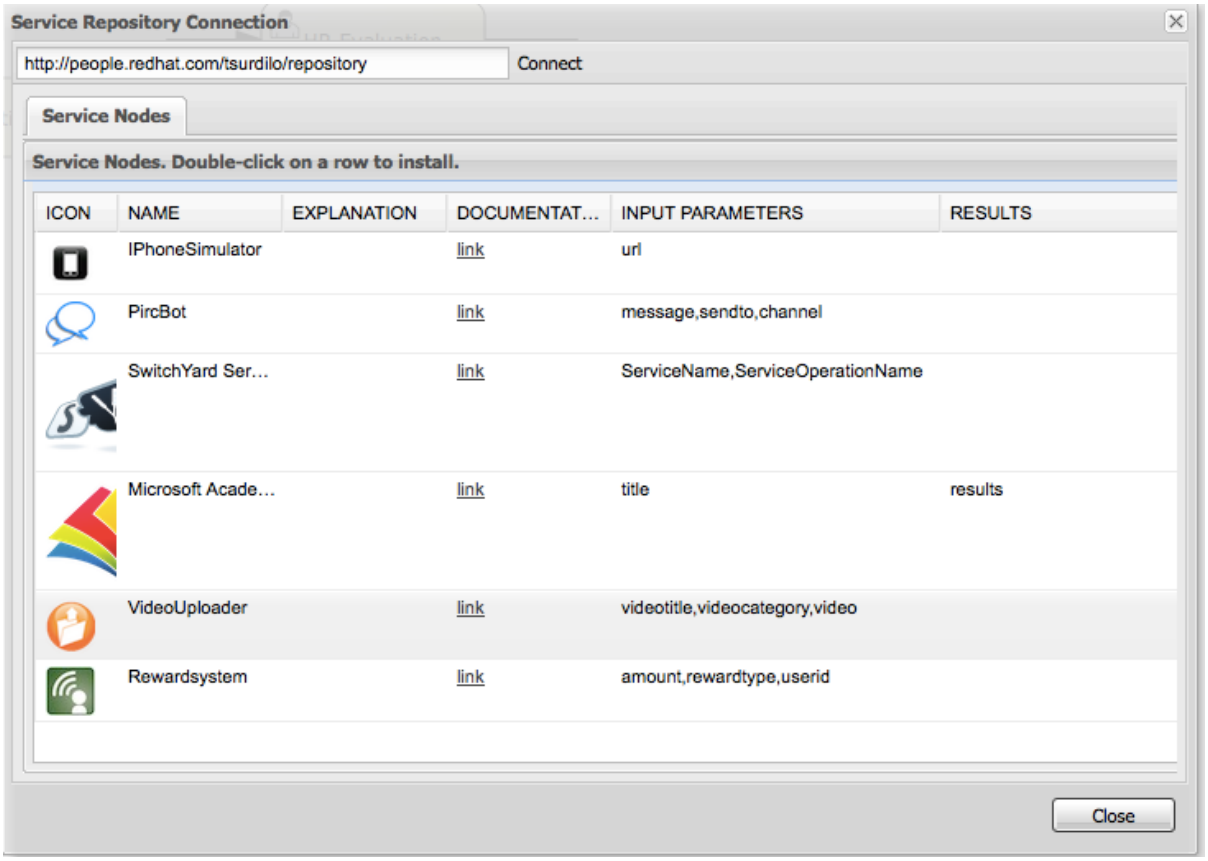


Figure 12.37. Service Repository installation view

(24) Full screen Modev - allows users to place the drawing board of Designer into full-screen mode. This can help with better visualizing larger business processes without having to scroll. Note that this feature is possible only if your browser has full screen mode capabilities. If it does not designer will show a message stating this to the user.

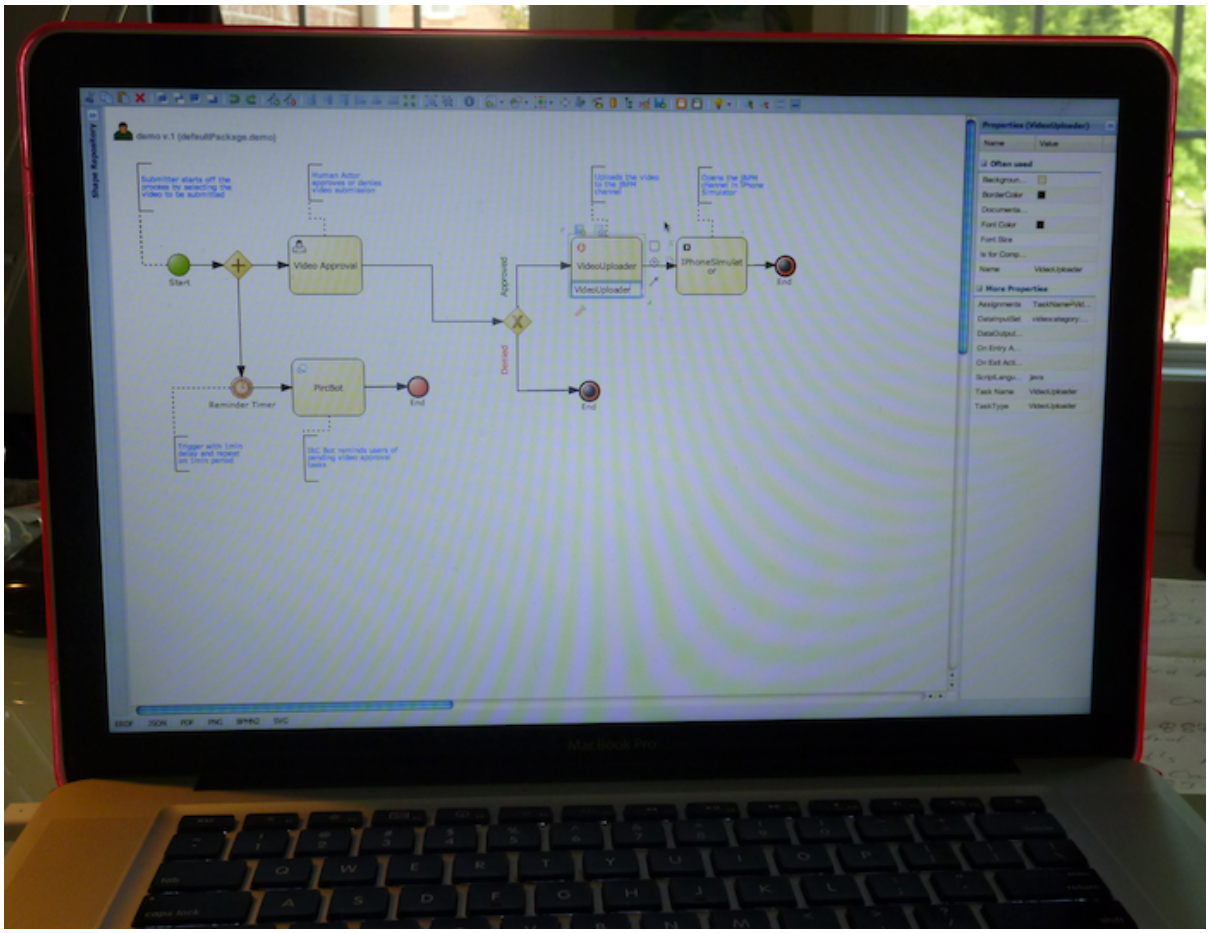


Figure 12.38. Full Screen Mode

(25) Process Dictionary - Designer Dictionary Editor allows users to create their own dictionary entries or harvest from process documentation or business requirement documents. Process Dictionary entries can be used as auto-completion for shape names. This will be expanded in the future versions to allow mapping of node patters to specific dictionary entries as well. Users can add entries to the dictionary in the Dictioanry Editor or from the selected shapes directly.

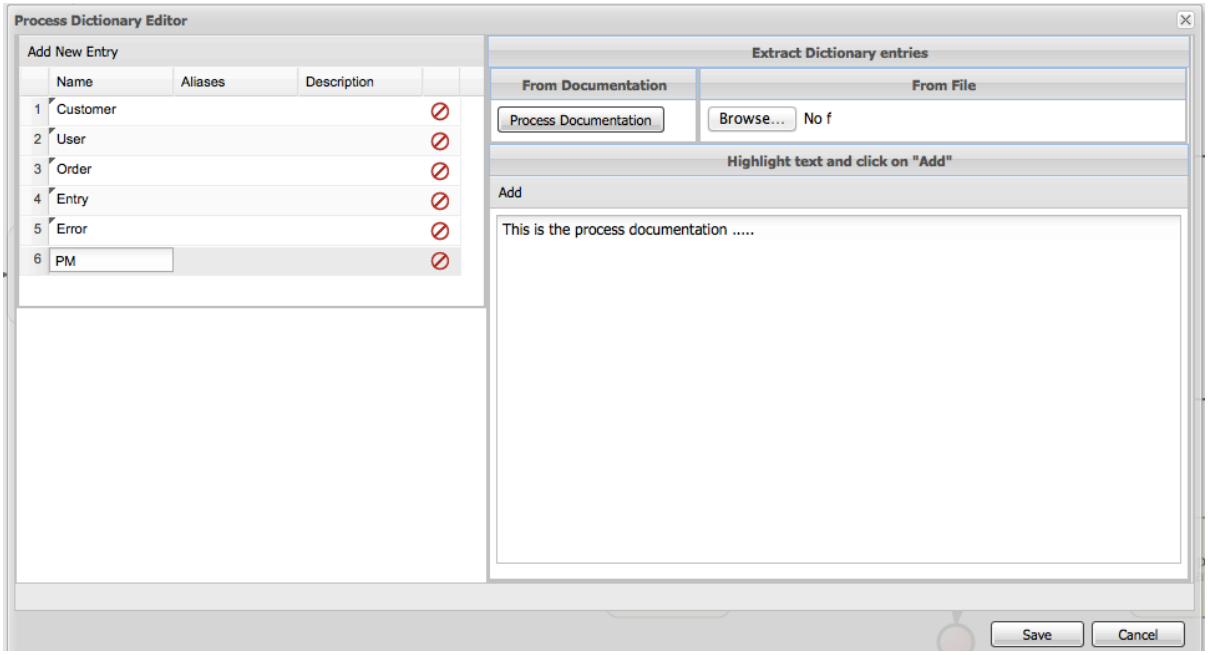


Figure 12.39. Process Dictionary entry screen

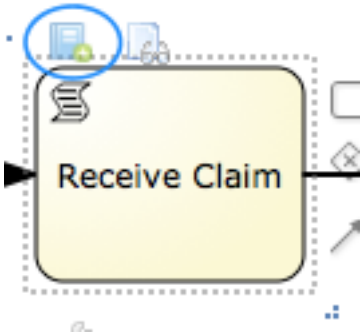


Figure 12.40. Addint to process dictionary from selected shape

(26, 27, 28, 29) Zooming - zooming allows users to zoom in/out of the model, zoom in/out back to the original setting as well as zoom the process model on the drawing board to fit the currently dimensions of the drawing board.

Chapter 13. Form Modeler

This chapter intends to describe in a simple ways all the steps required to create a process with human tasks, generate and modify the forms for these tasks and execute them. It will provide initial guidance to perform all initial steps, but it will not provide a full description of all available features.

Given that forms are going to be used in tasks, it's possible to generate forms automatically from process variables and task definitions. These forms can be later be modified by using the form editor. In runtime, forms will receive data from process variables, display it to the user and capture his input, and then finally updating process variables again with the new values.

The following example will show all the steps to follow to create a form for the 'Create order' task in the process below.

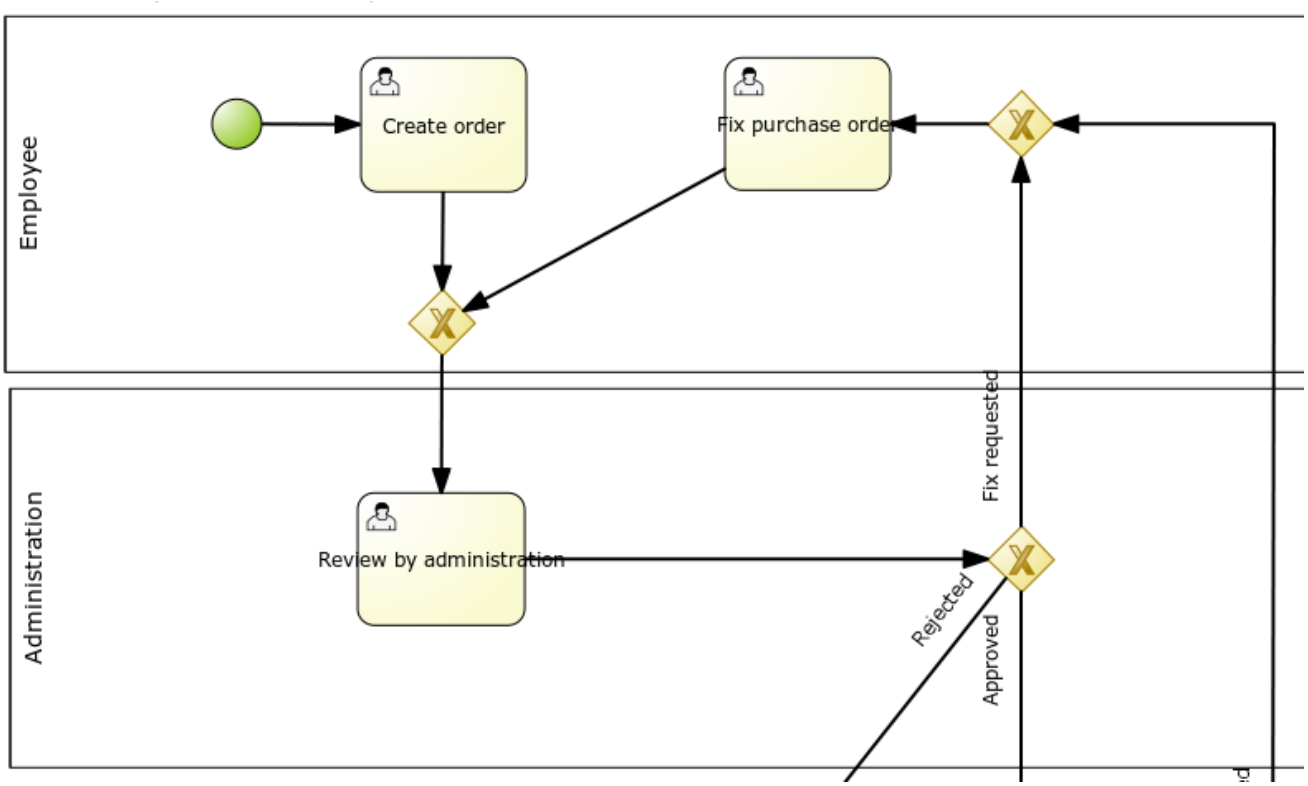


Figure 13.1. Process example

This form must look like the following in execution:

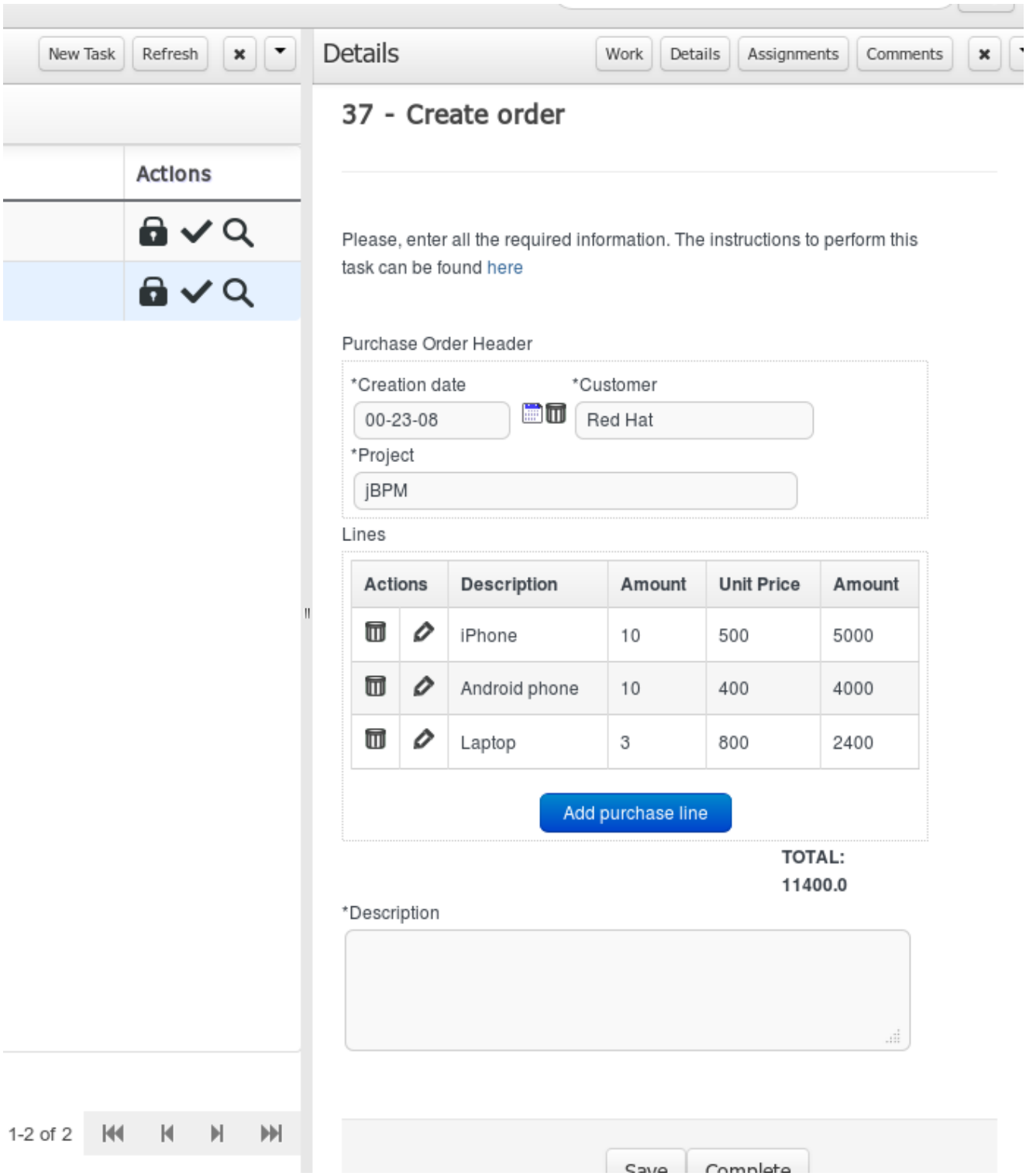


Figure 13.2. Process example

13.1. Configure process and human tasks

To hold values capture by forms, process variables can be created. These variables can be of a simple type like 'String' or a complex type. These complex types can be defined by using the Data Modeler tool, or be just regular POJOs (Plain Java Objects) created with any Java IDE.

In this example, we define a variable 'po' of type 'org.jbpm.examples.purchases.PurchaseOrder', defined with the Data Modeler tool.

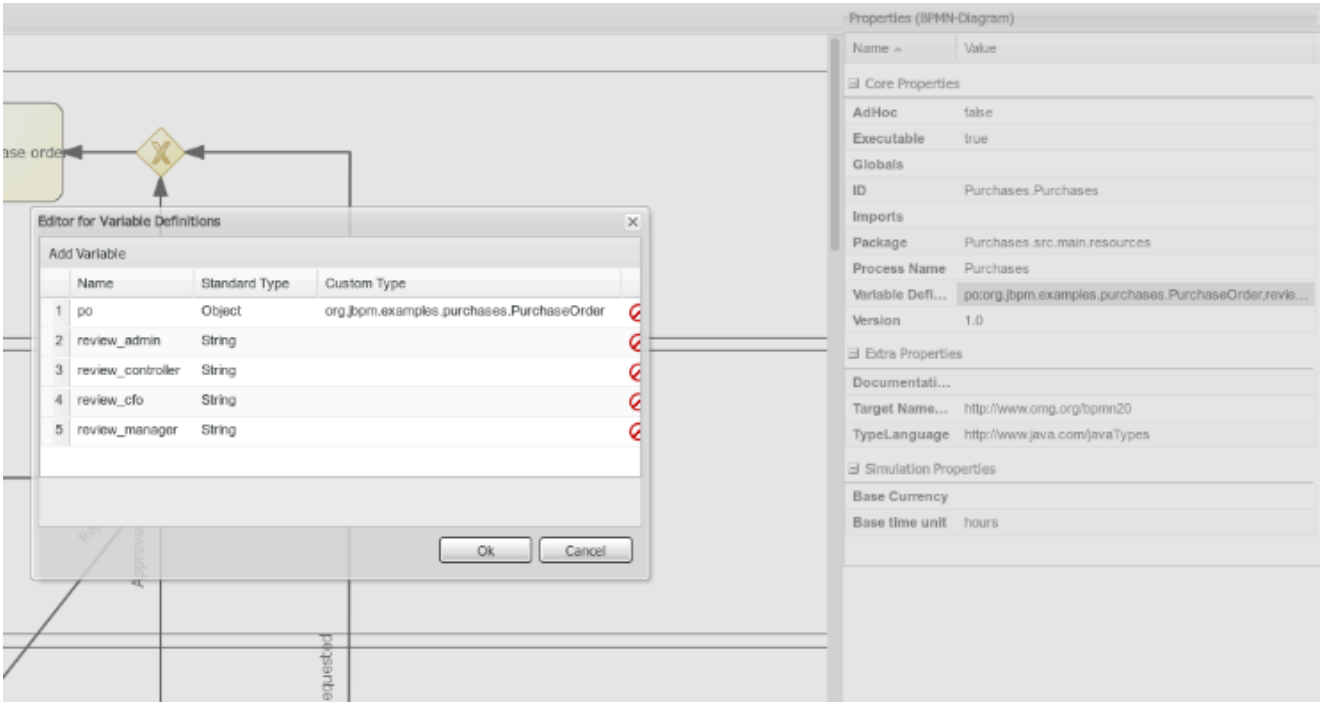


Figure 13.3. Process variable definition

This variable is declared in the 'variables definition' property for the process.

After that, we must configure which variables are set as input parameters to the task, which ones will receive the response back from the form and establish the mappings. This is done by setting the 'DataInputSet', 'DataOutputSet' and 'Assignments' properties for any human task. See screenshots below for details.

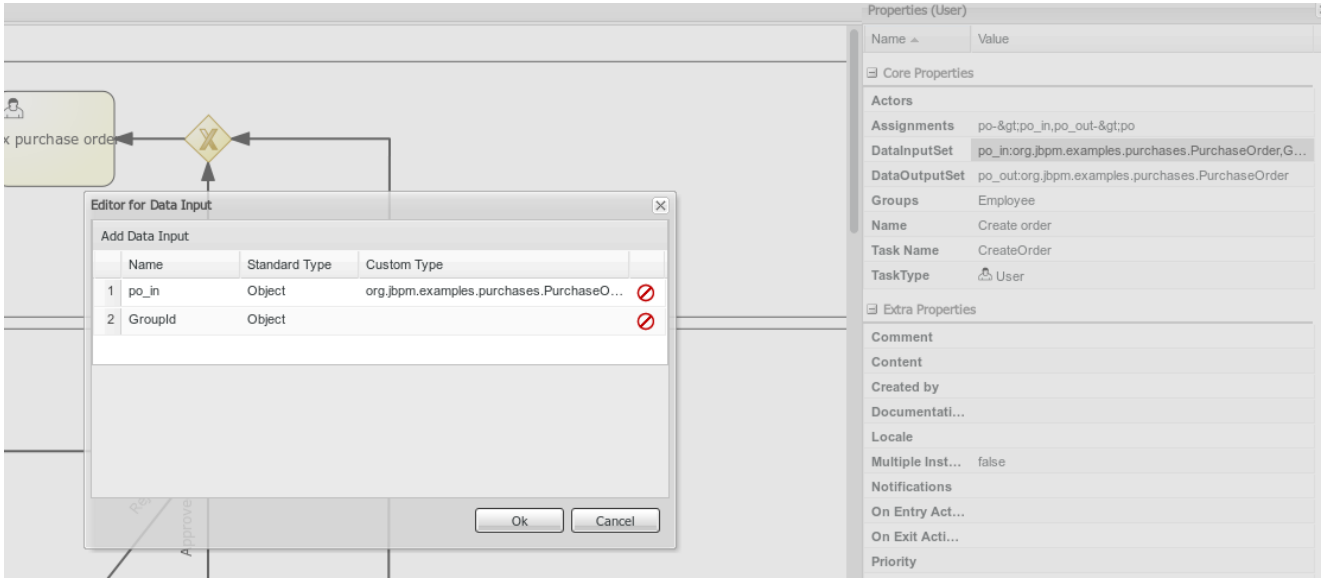


Figure 13.4. Data input variable definition

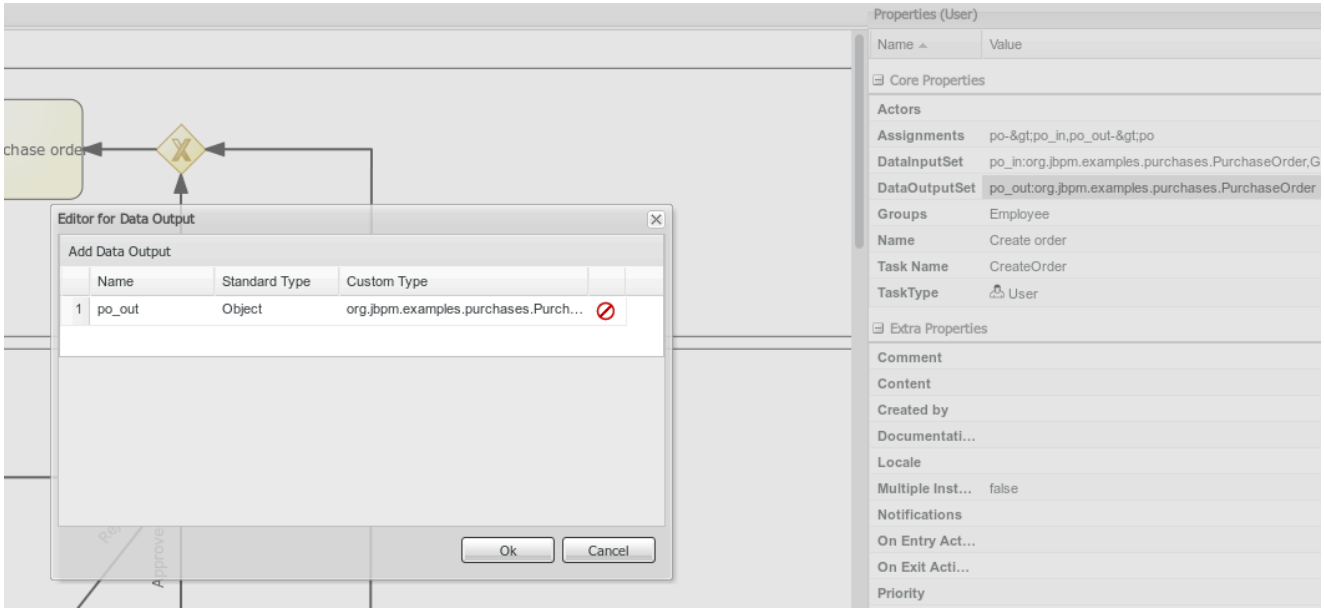


Figure 13.5. Data output variable definition

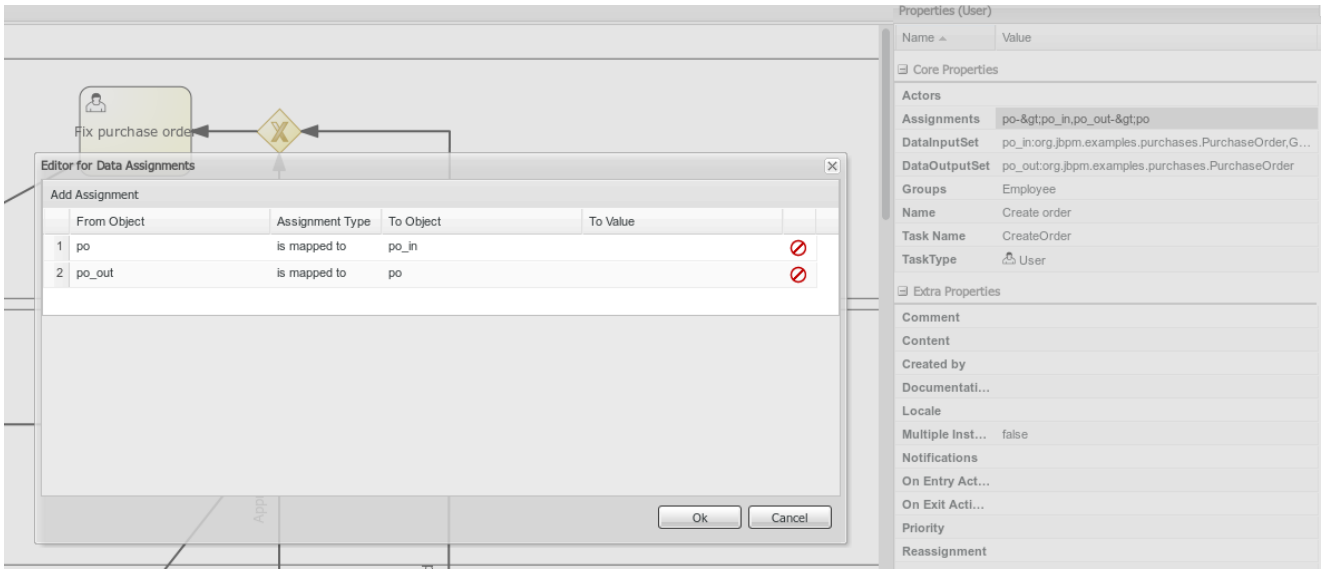


Figure 13.6. Variable mapping definition

13.2. Generate forms from task definitions

The Process Designer module provides some functionality to generate the forms automatically from task and variable definitions, as well as easily open the right form from the modeler.

This is done with the following menu option.

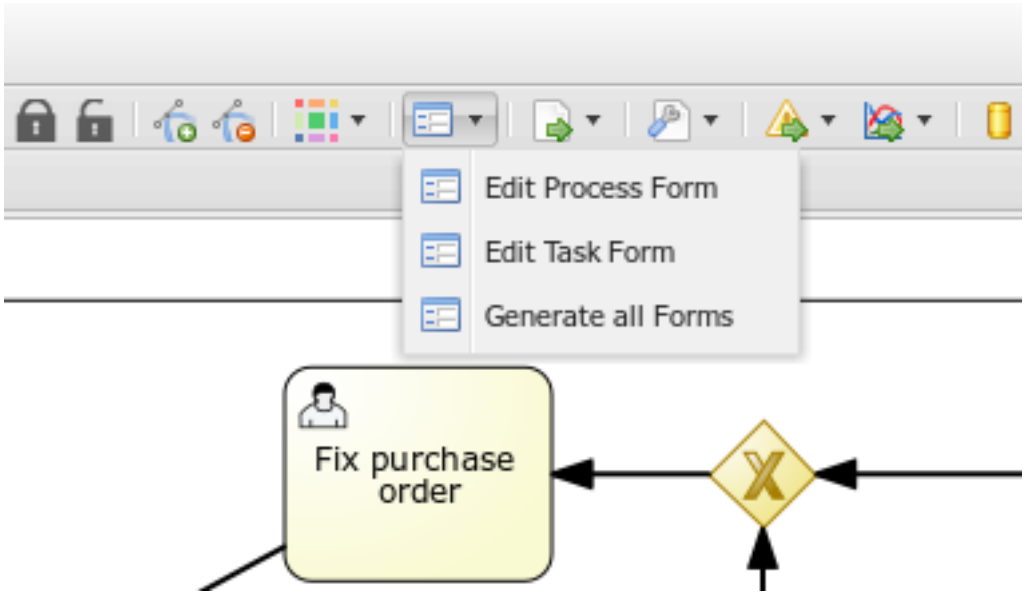


Figure 13.7. Form automatic generation

You can also click on the icon on top of task to open the form directly.

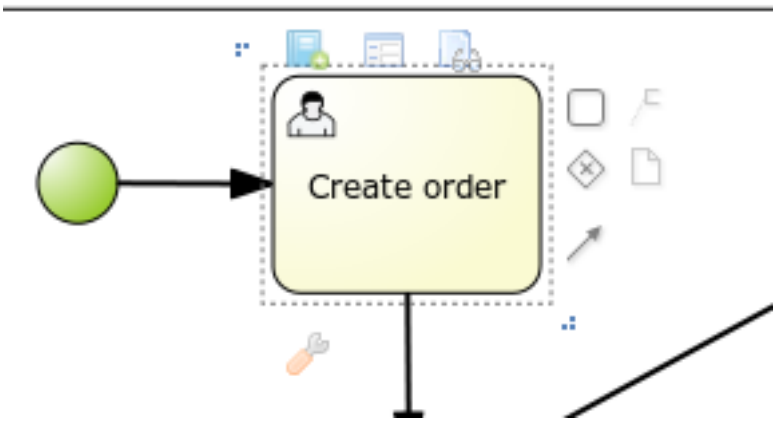


Figure 13.8. Access to form edition

Forms are related to tasks by following a naming convention. If a form with a name formName-taskform is defined in the same package as the process, then this form is used by the human task engine to display and capture information from user.

Also, if a form named ProcessId-task form is created, it will be used as the initial form when starting this process.

For example, for our process the following forms would be generated.

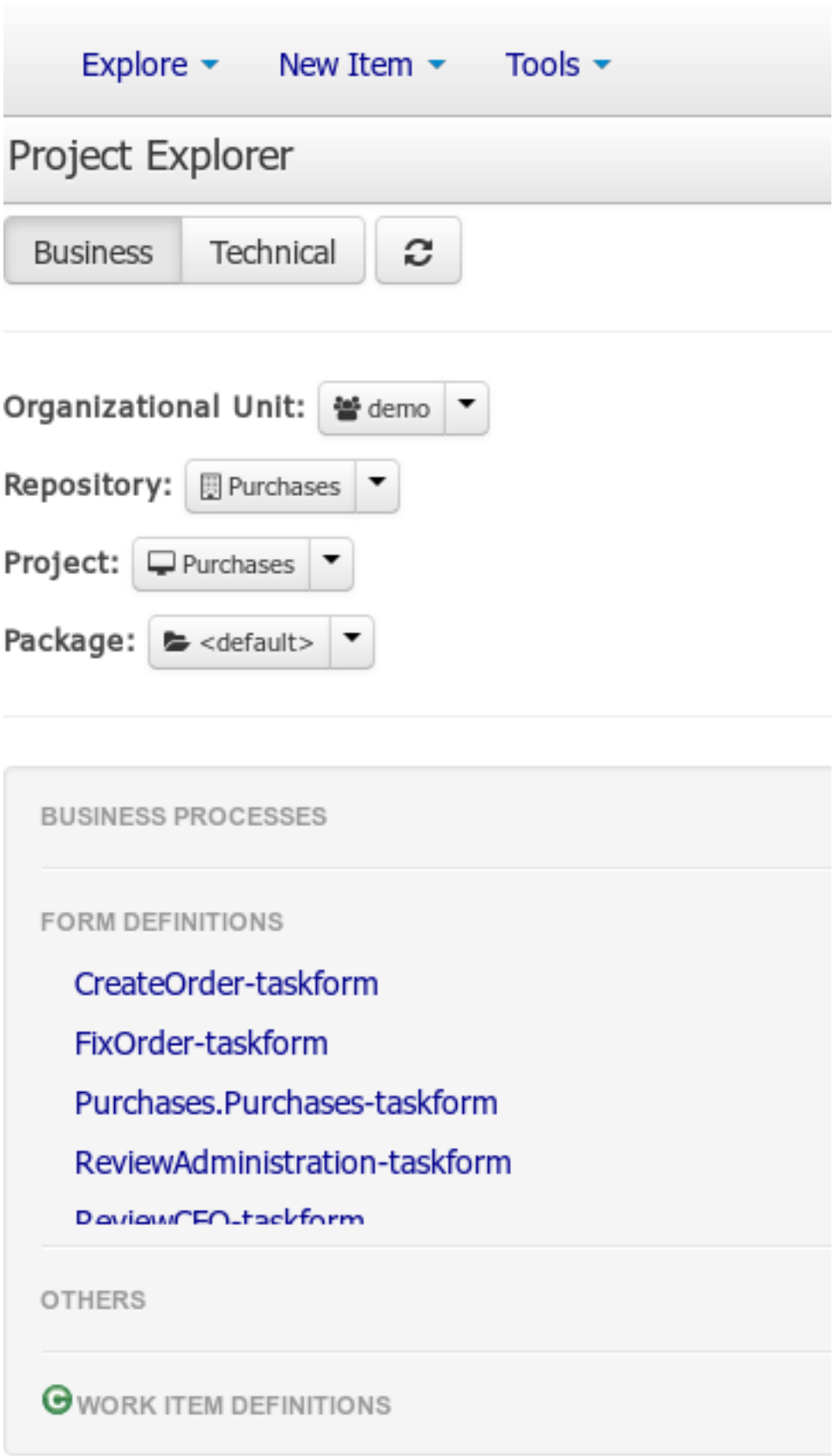


Figure 13.9. Access to form edition

13.3. Edit forms

Once the forms have been generated, you can start editing them. There are several artifacts that are generated in the previous process, but also can be created manually.

13.3.1. Form generated description

When the form has been generated automatically, this tab contain the process variables as data origins. This allow bind form fields with them, this relation it's linked creating data bindings.

A data binding define how task inputs will be mapped to form variables, and when the form is validated and submitted, how the values will update the task outputs.

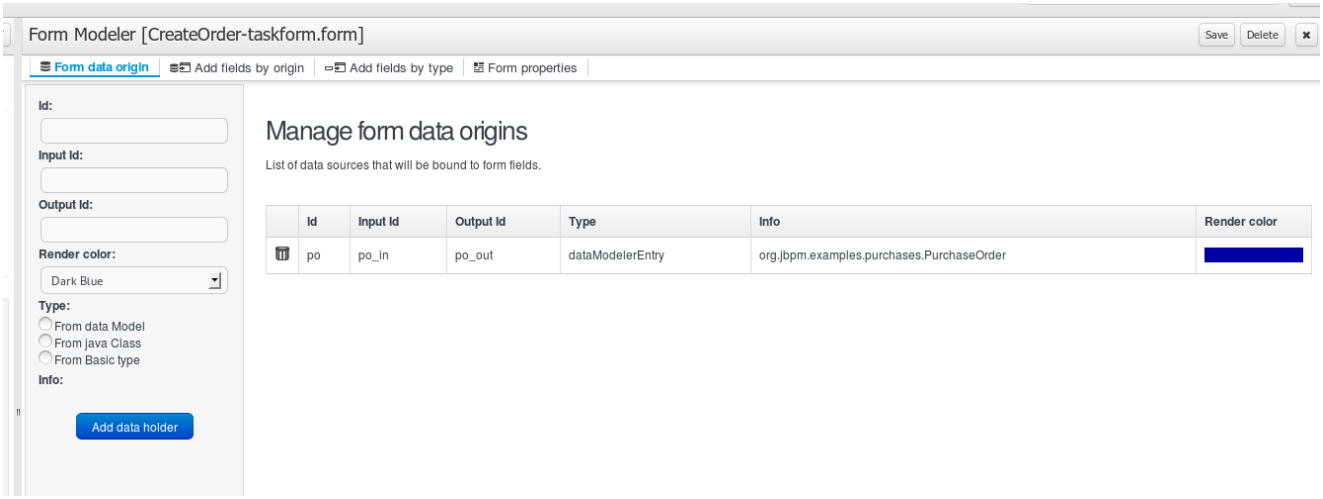


Figure 13.10. Generated form

For example, for this process, the following bindings are generated. Notice that the identifiers are automatically generated. You can have as many data origins as required, and can use a different colour to identify it.

In automatic form generation, a data origin is created for each process variable. The generated form have a field for each data origin bindable item (view FieldTypes) and this automatic fields have the binding defined too.

When these fields are displayed in editor the color of the data origin is shown over the field to make easy view if the field is correctly bound and the data origin implied.

13.3.2. Customizing form

We can change the way the form is displayed to the user in the task list. Next, we will show different levels of customization that will allow change it

13.3.2.1. Moving fields

The fields may be placed in different regions of the form. To move a field the user can access the contextual menu of the field and select 'Move field'.

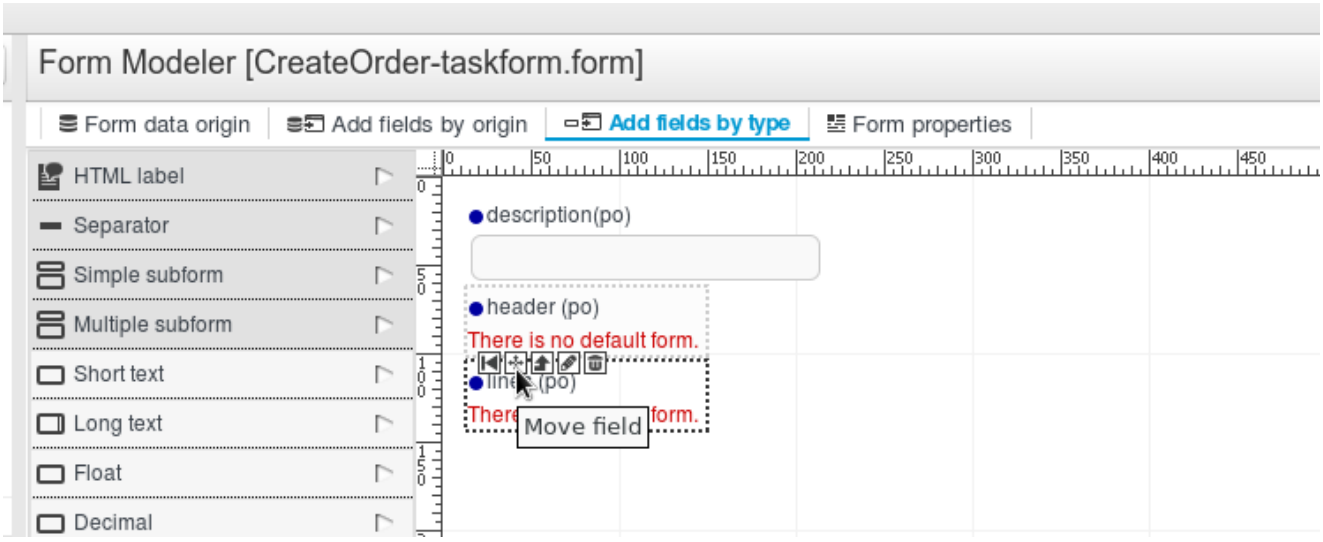


Figure 13.11. Move field option

This will display the different regions of the form where you can place it.

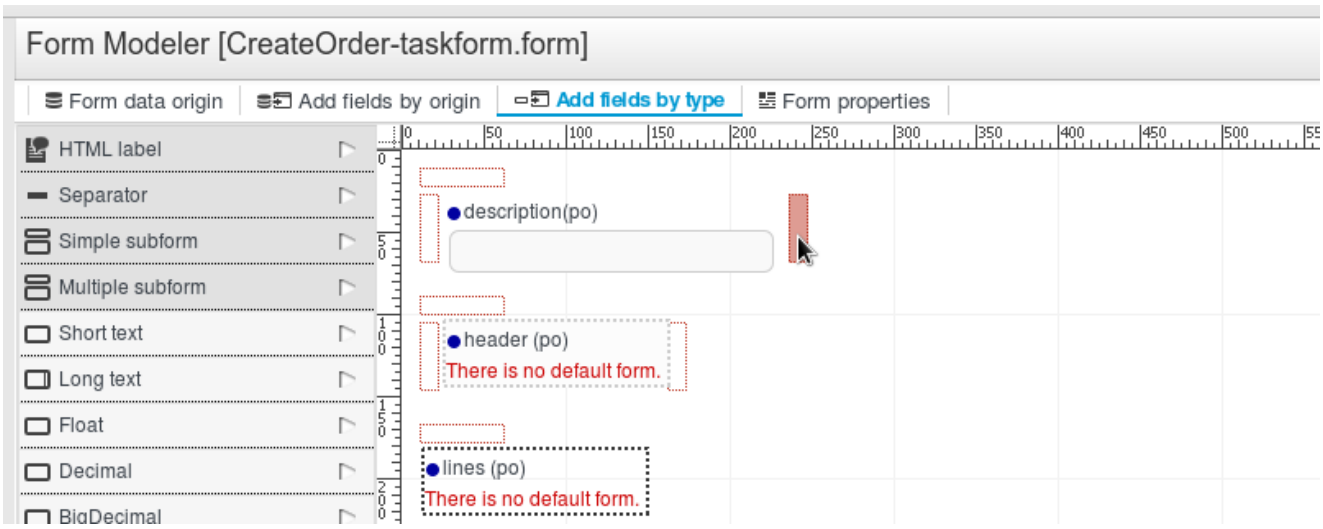


Figure 13.12. Destination areas to move the field

A field can be moved to the first or the last region with the contextual icons for that purpose.

13.3.2.2. Adding new fields

You can add fields to forms either by its origin or by selecting one type of form field.

Let's see what has been created automatically for this purchase order form.

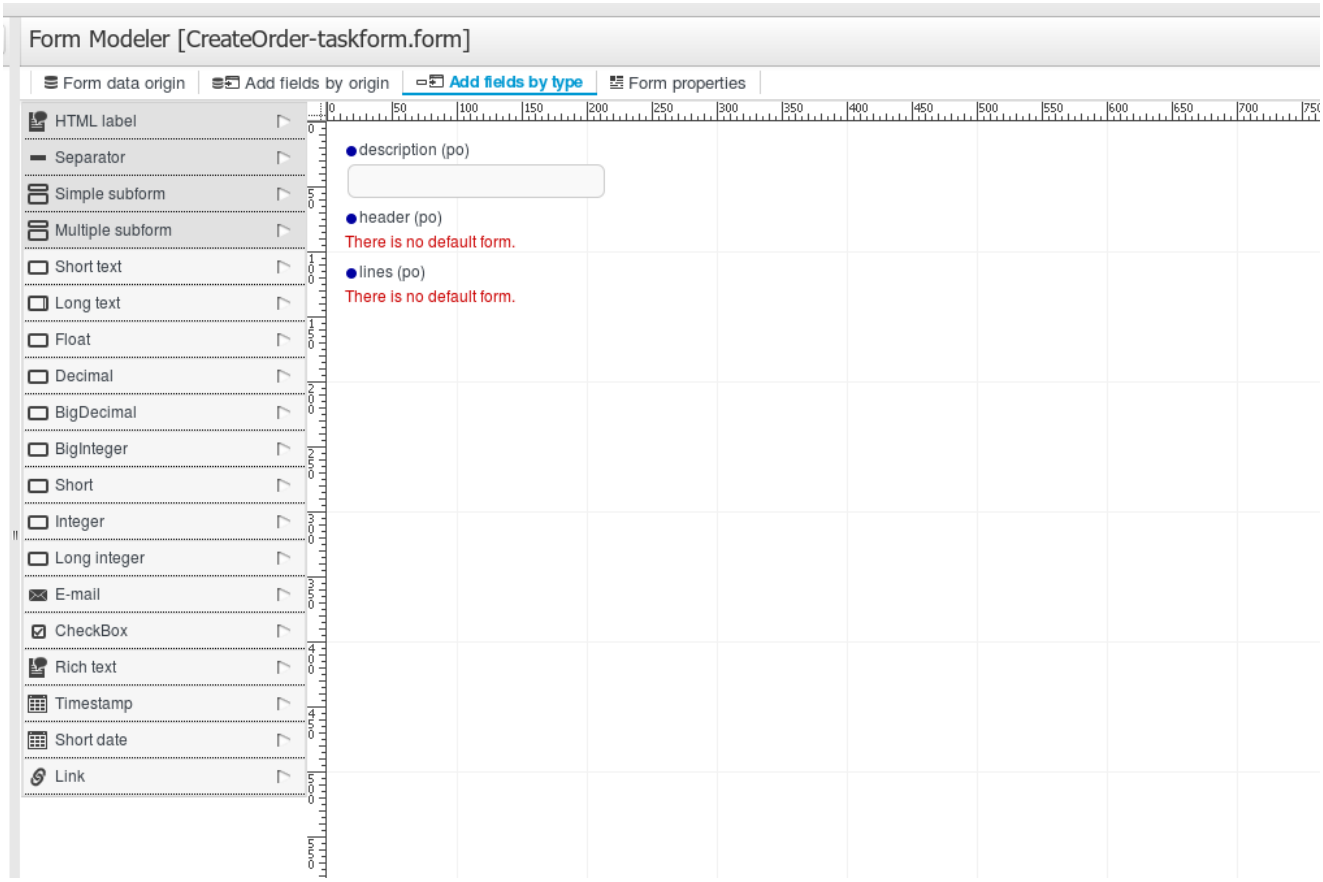


Figure 13.13. Form properties have been added by default, but are not still configured

- Add fields by origin: this tab allows you to add fields to the form based on the data origins defined. These fields will have the correct configuration on the "Input binding expression" and "Output binding expression" properties, so when the form is submitted, the fields values will be stored in the corresponding Data Origin.

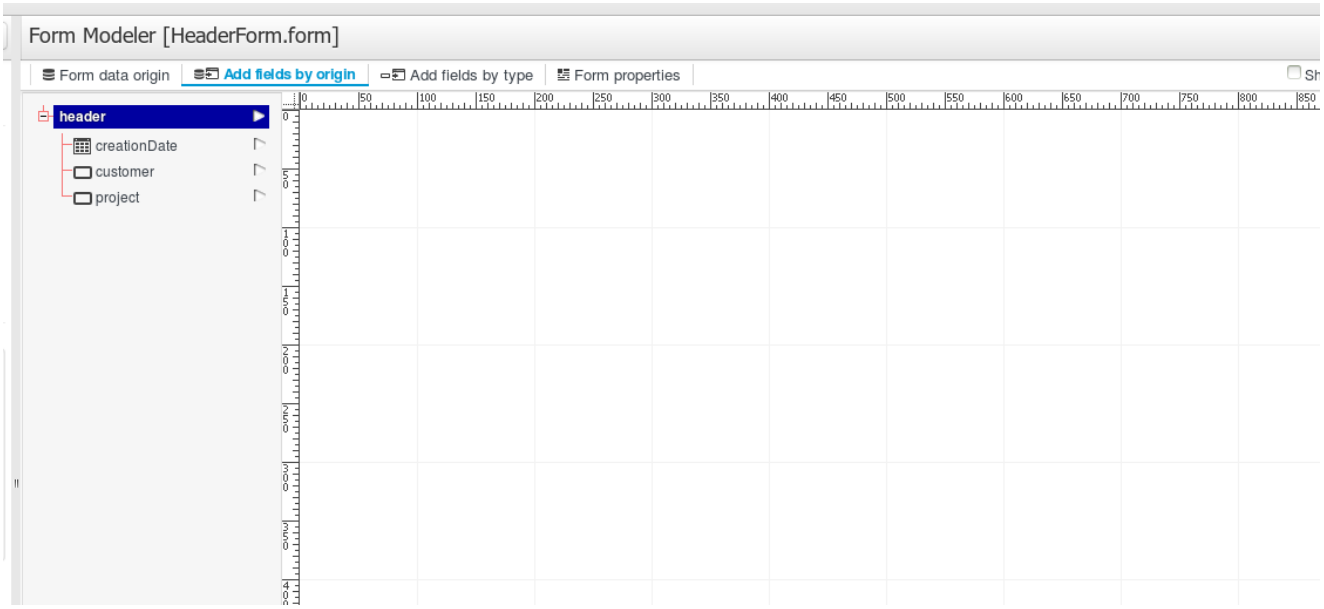


Figure 13.14. Add field by origin

- Add fields by type: this tab allows you to freely add fields to the form from the Field Types palette on the Form Modeler. These fields won't be storing their values on any Data Origin until they have a correct configuration on the "Input binding expression" and "Output binding expression" properties.

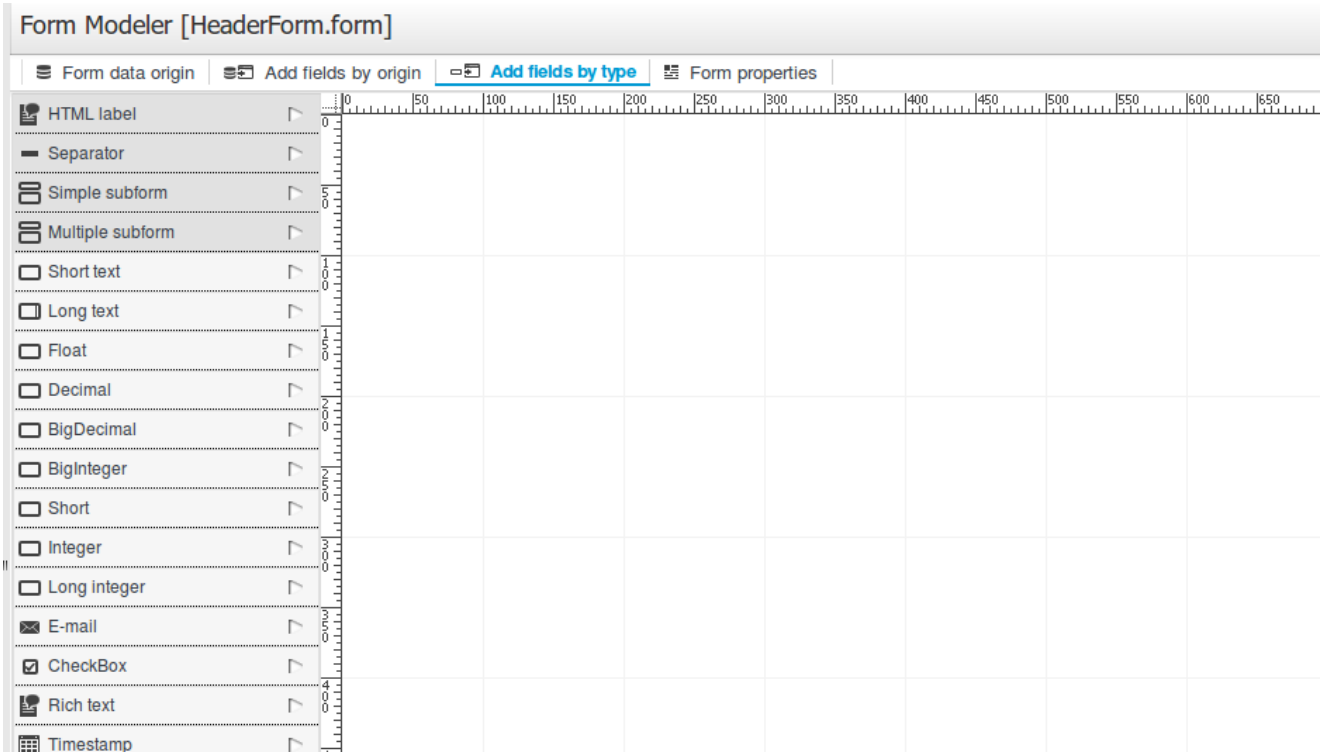


Figure 13.15. Add field by type

To see a complete list of the available field types go to *Field types section*.

Notice the data model 'po' of type 'org.jbpm.examples.purchases.PurchaseOrder' is composed of three properties.

- Simple: property of type text (description). We will adjust the view settings.
- Complex: property of type object (header).
- Complex: property of type array of objects (lines)

Now all these properties had to be configured.

13.3.2.3. Field configuration

Each field can be configured to enhance performance in the form. There are a group of common properties, that we call ‘Generic field properties’ and a group of specific properties that depends on the field type.

13.3.2.3.1. Generic field properties

There are a group of properties that are common to all field types. We will detail them below:

Table 13.1.

Field type	Can change the field type to other compatible field types
Field Name	Will be used as identifier in formulas calculation
Label	The text that will be shown as field label
Error message	When something goes wrong with the field, like validations,.. this message will be displayed
Label ccs class	Allows enter a class css to apply in label visualization
Label css style	to enter directly the style to apply to the label.
Help text	The text introduced is displayed as alt attribute to help to the user in data introduction
Style class	Allows enter a class css to apply in field visualization
Css style	to enter directly the style to apply to the label.
Read Only	When this check is on, the field will be used only for read

Input binding expression	This expression defines the link between field and process task input variable. It will be used in runtime to set the field value with that task input variable data.
Output binding expression	This expression defines the link between field and process task output variable. It will be used in runtime to set that task output variable.

13.3.2.3.2. Specific field properties

Let's explain the specific properties of each field type:

- Short Text (java.lang.String)
  - Compatible field type: Long text, E-mail, Rich text
  - Specific properties
    - Size: input text length.
    - MaxLength: Maximum number of characters allowed.
    - Required: Indicates if it’s mandatory to fill this field.
    - Show HTML: indicates whether the contents of the field is interpreted as HTML in show mode.
    - Formula. to enter expressions that will be evaluated to set the field value. These expressions are described in *Formula & expression section* .
    - Range value. A range formula allows you to let you specify the values that the user can select from an specific field. These expressions are described in *Formula & expression section*
    - Pattern. Allow introduce an expression to specify the validation of the field. In case that the field value introduced hasn’t match the expression, and error is thrown and the error message has to be shown.
    - Default Value formula. Expression to set the field default value.
- Long Text (java.lang.String)
  - Compatible field type: Long text, E-mail, Rich text
  - Specific properties
    - Size: input text length.
    - MaxLength: Maximum number of characters allowed.

- Required: Indicates if it’s mandatory to fill this field.
- Height: The number or rows to show at text area.
- Formula. to enter expressions that will be evaluated to set the field value. These expressions are described in *Formula & expression section* .
- Range value. A range formula allows you to let you specify the values that the user can select from an specific field. These expressions are described in *Formula & expression section*
- Pattern. Allow introduce an expression to specify the validation of the field. In case that the field value introduced hasn’t match the expression, and error is thrown and the error message has to be shown.
- Default Value formula. Expression to set the field default value.

- Float (java.lang.Float)
  - Specific properties
    - Size: input text length.
    - MaxLength: Maximum number of characters allowed.
    - Required: Indicates if it’s mandatory to fill this field.
    - Formula. to enter expressions that will be evaluated to set the field value. These expressions are described in *Formula & expression section* .
    - Range value. A range formula allows you to let you specify the values that the user can select from an specific field. These expressions are described in *Formula & expression section*
    - Pattern. Allow introduce an expression to specify how the Float value has to be displayed. The pattern allowed is show in section pattern in *http://docs.oracle.com/javase/6/docs/api/java/text/DecimalFormat.html* [http://docs.oracle.com/javase/6/docs/api/java/text/DecimalFormat.html]
    - Default Value formula. Expression to set the field default value.

- Decimal (java.lang.Double)
  - Specific properties
    - Size: input text length.
    - MaxLength: Maximum number of characters allowed.
    - Required: Indicates if it’s mandatory to fill this field.

    - Formula. Used to enter expressions that will be evaluated to set the field value. These expressions are described in *Formula & expression section*.
    - Range value. A range formula allows you to let you specify the values that the user can select from an specific field. These expressions are described in *Formula & expression section* .
    - Pattern. Allow introduce an expression to specify how the Double value has to be displayed. The pattern allowed is show in section pattern in *http://docs.oracle.com/javase/6/docs/api/java/text/DecimalFormat.html* [http://docs.oracle.com/javase/6/docs/api/java/text/DecimalFormat.html]
    - Default Value formula. Expression to set the field default value.

- BigDecimal (java.math.BigDecimal)
    - Specific properties
        - Size: input text length.
        - MaxLength: Maximum number of characters allowed.
        - Required: Indicates if it’s mandatory to fill this field.
        - Formula. Used to enter expressions that will be evaluated to set the field value. These expressions are described in *Formula & expression section*.
        - Range value. A range formula allows you to let you specify the values that the user can select from an specific field. These expressions are described in *Formula & expression section* .
        - Pattern. Allow introduce an expression to specify how the BigDecimal value has to be displayed. The pattern allowed is show in section pattern in *http://docs.oracle.com/javase/6/docs/api/java/text/DecimalFormat.html* [http://docs.oracle.com/javase/6/docs/api/java/text/DecimalFormat.html]
        - Default Value formula. Expression to set the field default value.
- Big integer (java.math.BigInteger)
    - Specific properties
        - Size: input text length.
        - MaxLength: Maximum number of characters allowed.
        - Required: Indicates if it’s mandatory to fill this field.
        - Formula. Used to enter expressions that will be evaluated to set the field value. These expressions are described in *Formula & expression section*.

    - Range value. A range formula allows you to let you specify the values that the user can select from an specific field. These expressions are described in *Formula & expression section* .
    - Default Value formula. Expression to set the field default value.
- Short (java.lang.Short)
    - Specific properties
        - Size: input text length.
        - MaxLength: Maximum number of characters allowed.
        - Required: Indicates if it's mandatory to fill this field.
        - Formula. Used to enter expressions that will be evaluated to set the field value. These expressions are described in *Formula & expression section*.
        - Range value. A range formula allows you to let you specify the values that the user can select from an specific field. These expressions are described in *Formula & expression section* .
        - Default Value formula. Expression to set the field default value.
- Integer (java.lang.Integer)
    - Specific properties
        - Size: input text length.
        - MaxLength: Maximum number of characters allowed.
        - Required: Indicates if it's mandatory to fill this field.
        - Formula. Used to enter expressions that will be evaluated to set the field value. These expressions are described in *Formula & expression section*.
        - Range value. A range formula allows you to let you specify the values that the user can select from an specific field. These expressions are described in *Formula & expression section* .
        - Default Value formula. Expression to set the field default value.
- Long Integer (java.lang.Long)
    - Specific properties
        - Size: input text length.
        - MaxLength: Maximum number of characters allowed.

    - Required: Indicates if it's mandatory to fill this field.
    - Formula. Used to enter expressions that will be evaluated to set the field value. These expressions are described in *Formula & expression section*.
    - Range value. A range formula allows you to let you specify the values that the user can select from an specific field. These expressions are described in *Formula & expression section* .
    - Default Value formula. Expression to set the field default value.
- E-mail (java.lang.String)
    - Compatible field type: Short text, Long text, Rich text
    - Specific properties
        - Size: input text length.
        - MaxLength: Maximum number of characters allowed.
        - Required: Indicates if it's mandatory to fill this field.
        - Default Value formula. Expression to set the field default value.
- Checkbox (java.lang.Boolean)
    - Specific properties
        - Required: Indicates if it's mandatory to fill this field.
        - Default Value formula. Expression to set the field default value.
- Rich text: (java.lang.String)
    - Compatible field type: Short text, Long text, E-mail
    - Specific properties
        - Size: input text length.
        - MaxLength: Maximum number of characters allowed.
        - Required: Indicates if it's mandatory to fill this field.
        - Height: The number or rows to show at text area.
        - Default Value formula. Expression to set the field default value.
- Timestamp (java.util.Date)
    - Compatible field type: Short date

- Specific properties
  - Size: input text length.
  - Required: Indicates if it’s mandatory to fill this field.
  - Formula. to enter expressions that will be evaluated to set the field value. These expressions are described in *Formula & expression section* .
  - Default Value formula. Expression to set the field default value.
- Short date (java.util.Date)
  - Compatible field type: Timestamp
  - Specific properties
    - Size: input text length.
    - Required: Indicates if it’s mandatory to fill this field.
    - Formula. to enter expressions that will be evaluated to set the field value. These expressions are described in *Formula & expression section* .
    - Default Value formula. Expression to set the field default value.
- Simple subform (Object)
  - For more details see section*Simple Object (Subform field Type)*.

    Specific properties
    - Default form. Show the list of available forms to select what one will be displayed to show the object.
- Multiple subform (Multiple Object)
  - For more details see section*Arrays of objects.(Multiple subform field Type)*.

    Specific properties
    - Default form. Show the list of available forms to select what one will be displayed to show the object when no other form is configured with an specific purpose.
    - Preview form. If a form is specified, it will be used to show the item details
    - Table form. If a form is specified, it will be used to show the table columns when the item list is showed
    - New item text. Text to show at New Item button
    - Add item text. Text to show at Add Item button

- Cancel text. Text to show at Cancel button
- Allow remove Items. If this check is selected, the form allow remove items in table view.
- Allow edit items. If this check is selected, the form allow edit items in table view.
- Allow preview items. If this check is selected, the form allow preview items in table view.
- Hide creation button. Check to not show the creation button
- Expanded. If is checked, when a new item is being added, the field display the table with the existing items and the creation form at same time
- Allow data enter in table mode. Allow modify data in table view directly.

13.3.2.3.3. Complex Fields Configuration

There are two types of complex fields: fields representing an object, and fields representing an object array.

Once the field is added to the form, either automatically or manually, it must be configured so that the form had to know how to display the objects that will contain in execution time.

Next we describe how can be the configuration process:

- The first thing to do is define how the contained object will be displayed. This is done creating a form that represents the object.
- In case of the object array, you can define a form to show in preview(edition), or to show when table is shown

Once the form to represent the object, the parent form has to be configured to use them in the parent Subform or Multiple subform.

Below we will describe how the setup would be:

13.3.2.3.3.1. Simple Object (Subform field Type)

One possible way of setting the value for an object property is by using an existing form, and embedding this form into the parent. This is called subform.

In this example, the Purchase Order header data is held in an object. Therefore, we must create a form to enter all the purchase order header data and link it from the parent task form.

We will follow the steps:

1. Create new form.

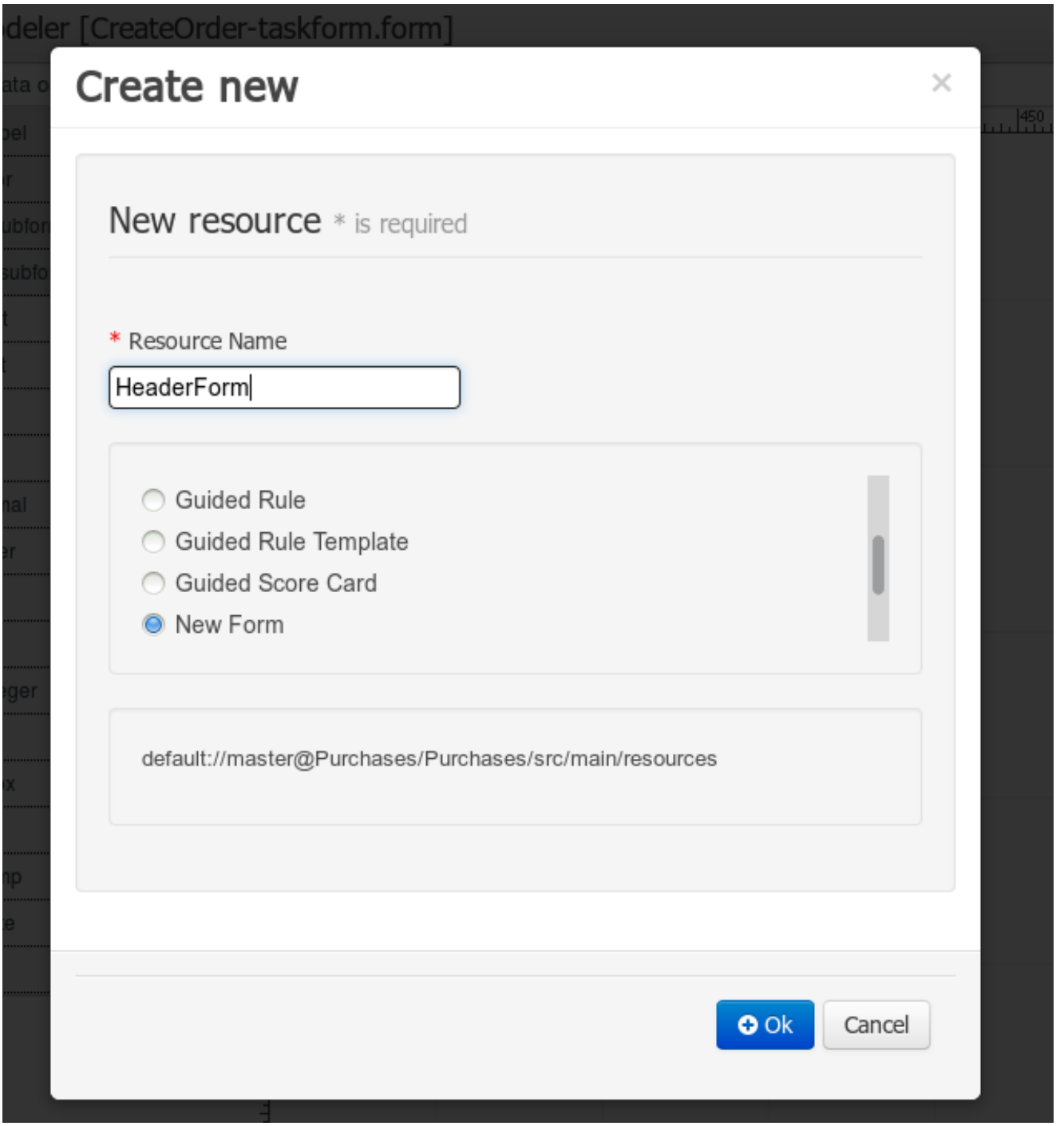


Figure 13.16. Create new form

2. Create new data origin, selecting the type of the purchase order header.

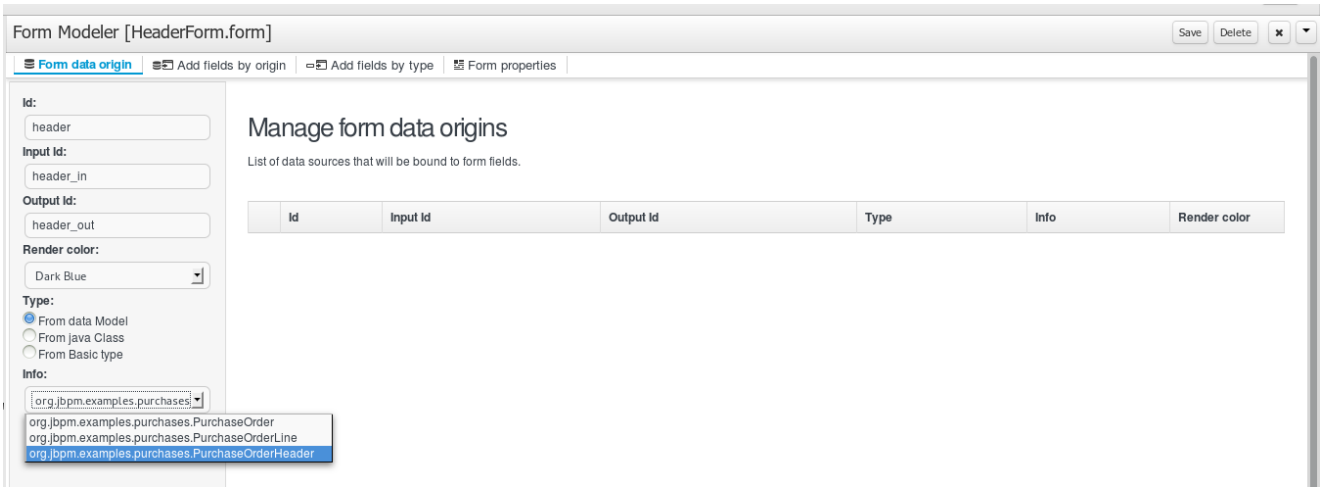


Figure 13.17. Create new data origin

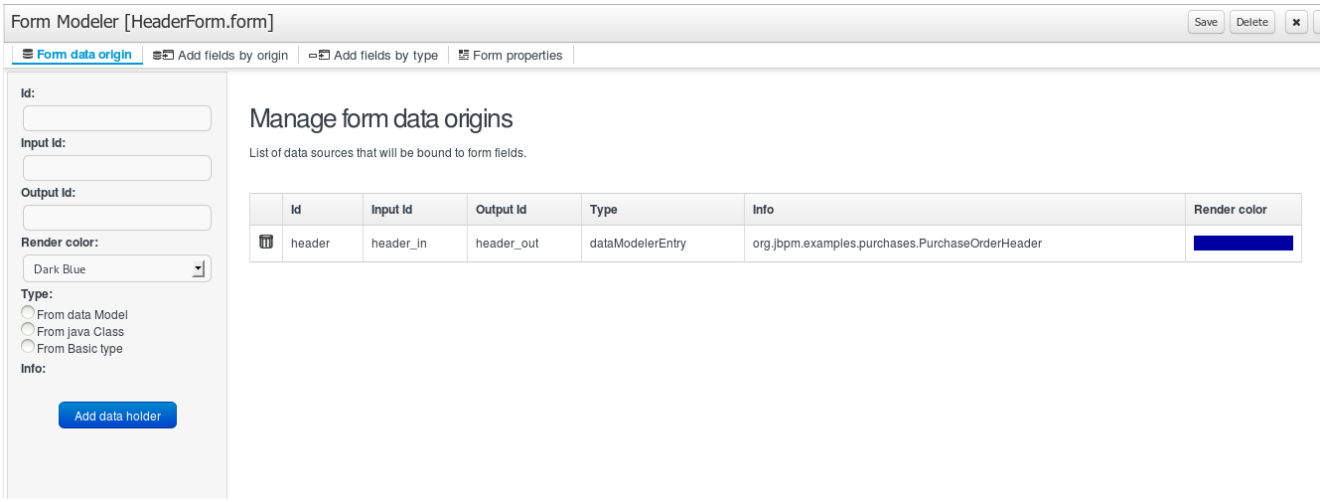


Figure 13.18. Data origin

3. Add fields by origin. All the properties are shown, and can be added to the form, either one by one or all of them at once.

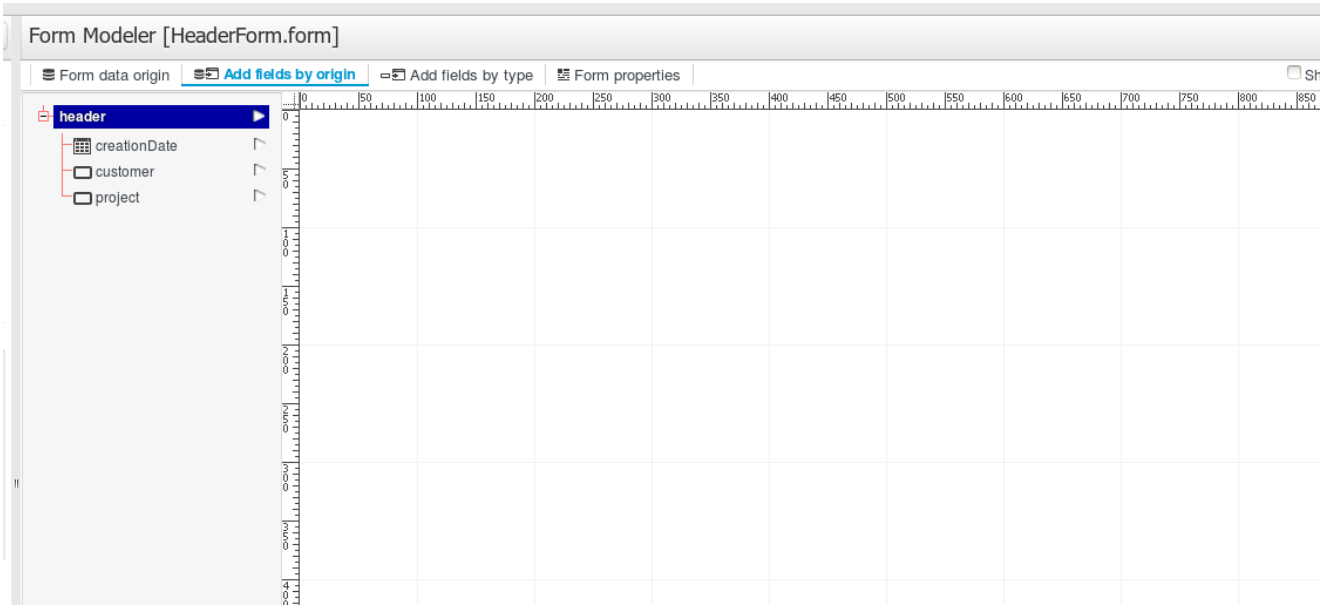


Figure 13.19. Add fields by origin

All the properties have been added to the form, and now we can edit each of them and move them around.

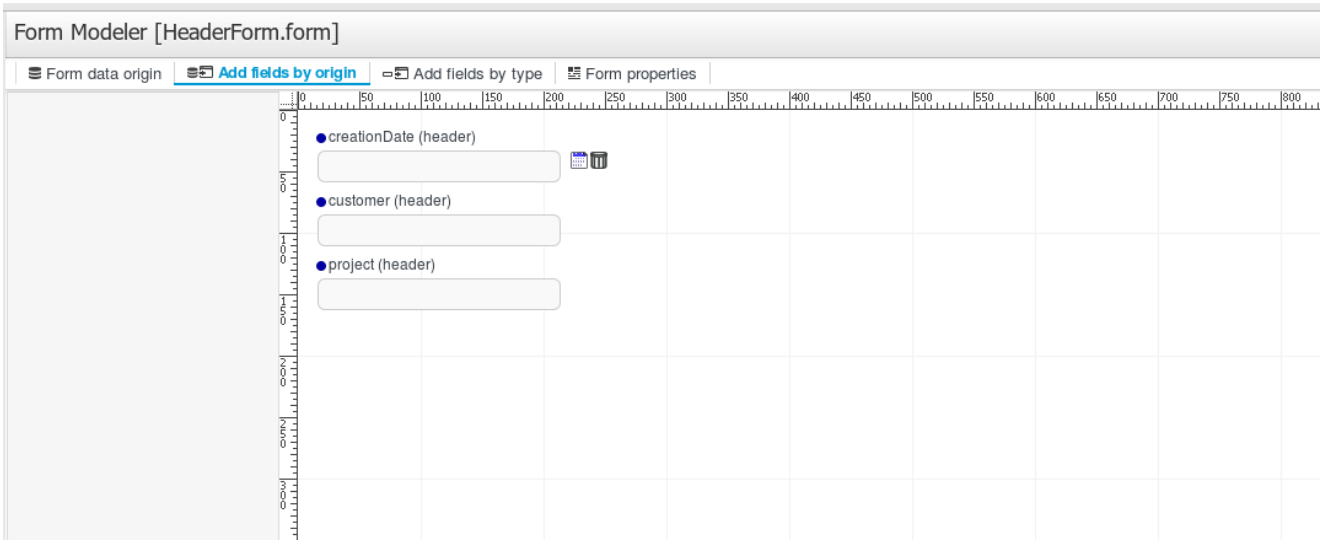


Figure 13.20. All data origin fields added

4. Configure the fields and customize form.

5. Once the form has been saved, open the initial parent form and set the field property 'Default form'.

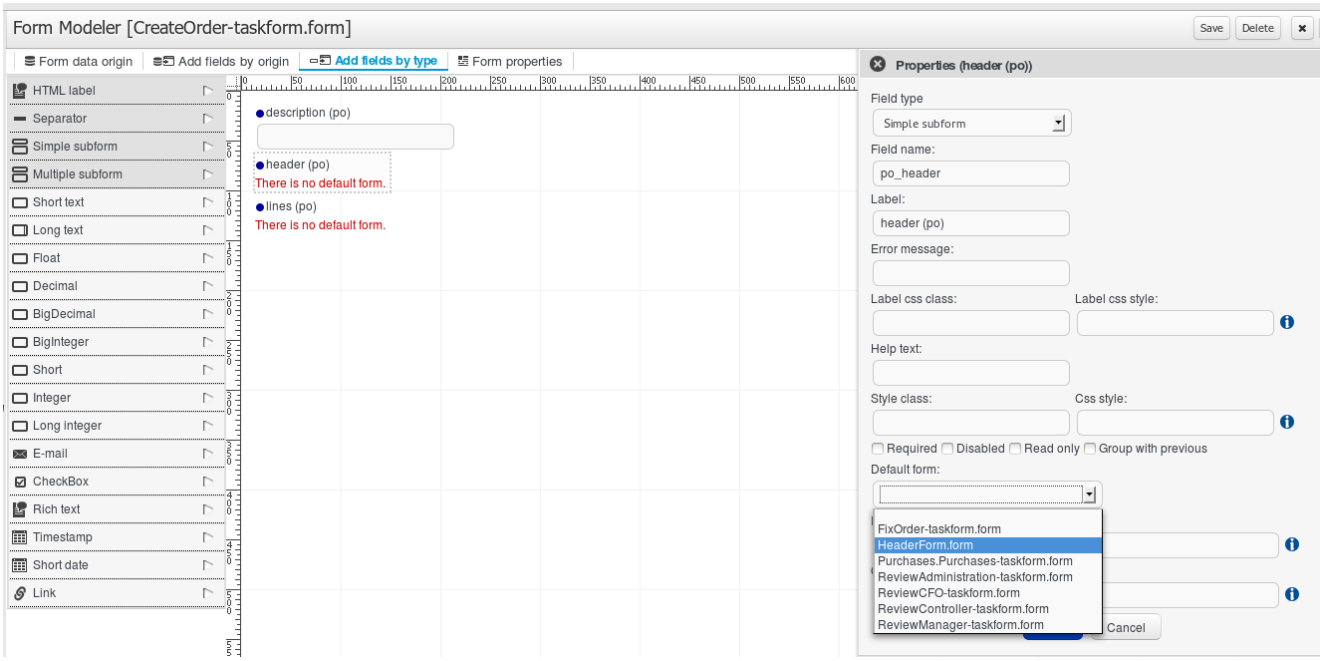


Figure 13.21. Configure the parent form

This will insert the subform inside the parent form, and will be shown as below:

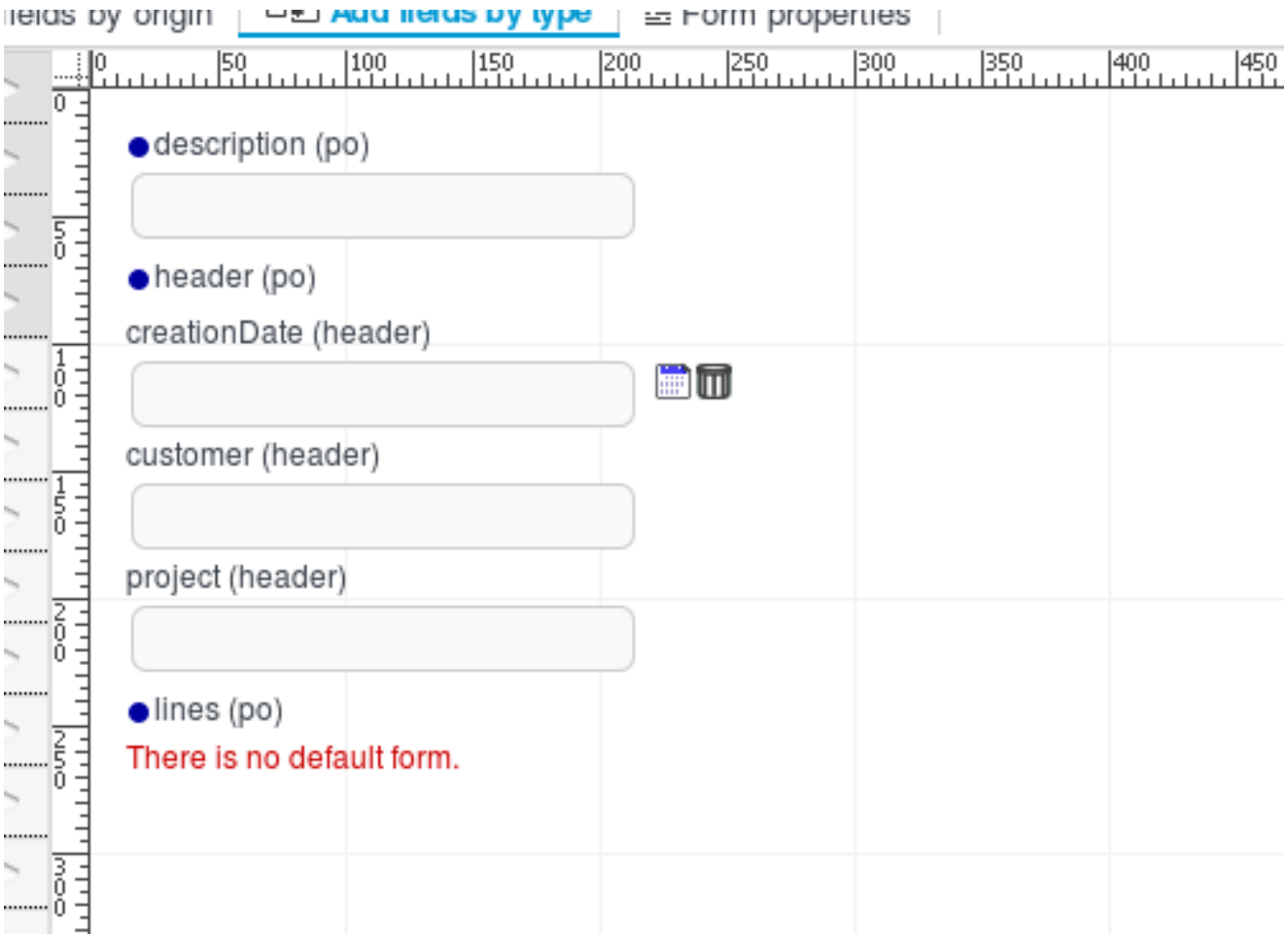


Figure 13.22. Parent form visualization after subform configuration

13.3.2.3.3.2. Arrays of objects.(Multiple subform field Type)

Now, we want to be able to create, edit and remove purchase order lines, by displaying a table with all the values and being able to capture information through a form. This will be done as follows:

Create a form that will hold and capture the information for each line's value (description, amount, unitPrice and total), following the same steps as above. This will be done as follows:

1. Create new form.

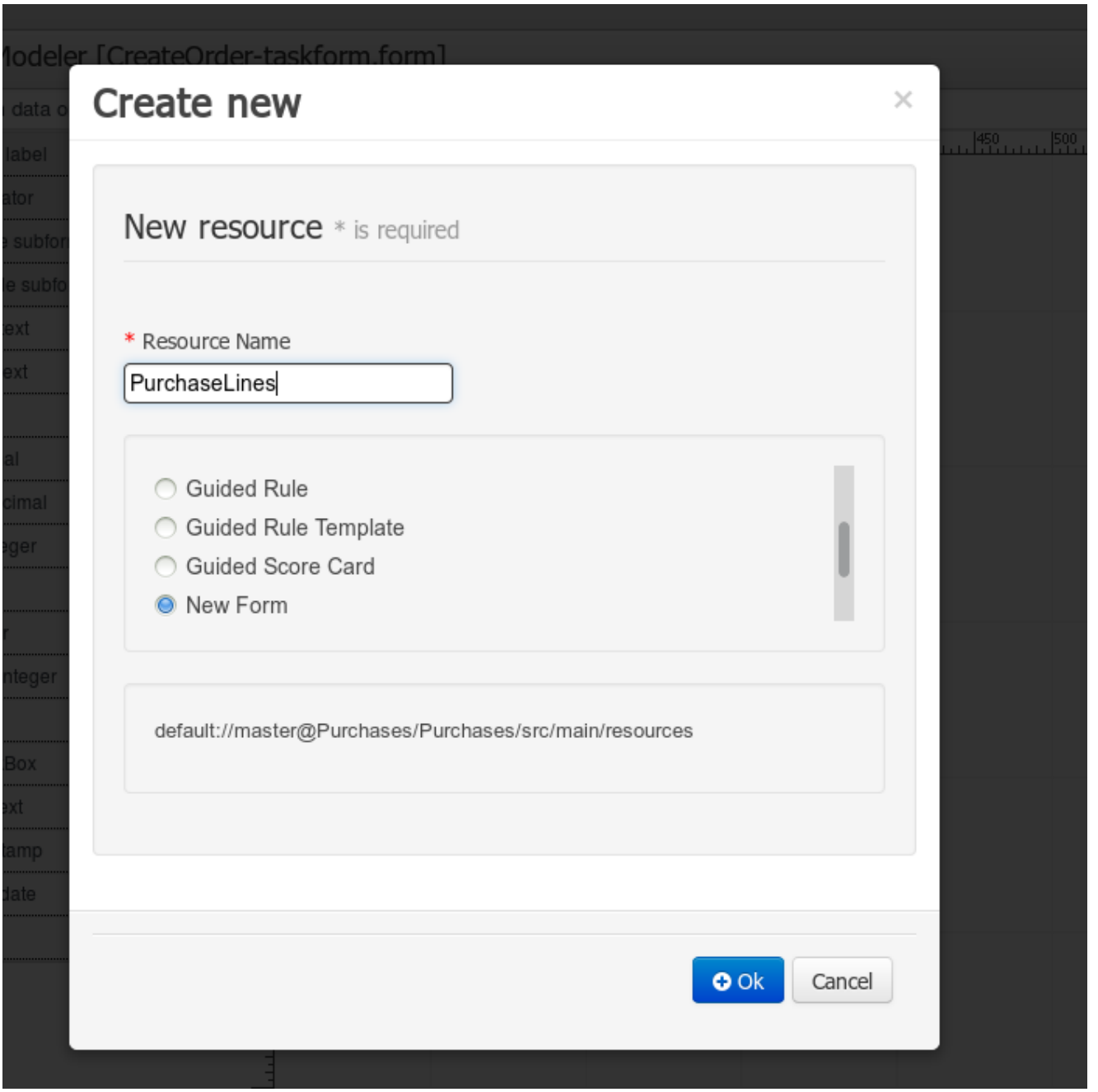


Figure 13.23. Create new form

2. Create new data origin.

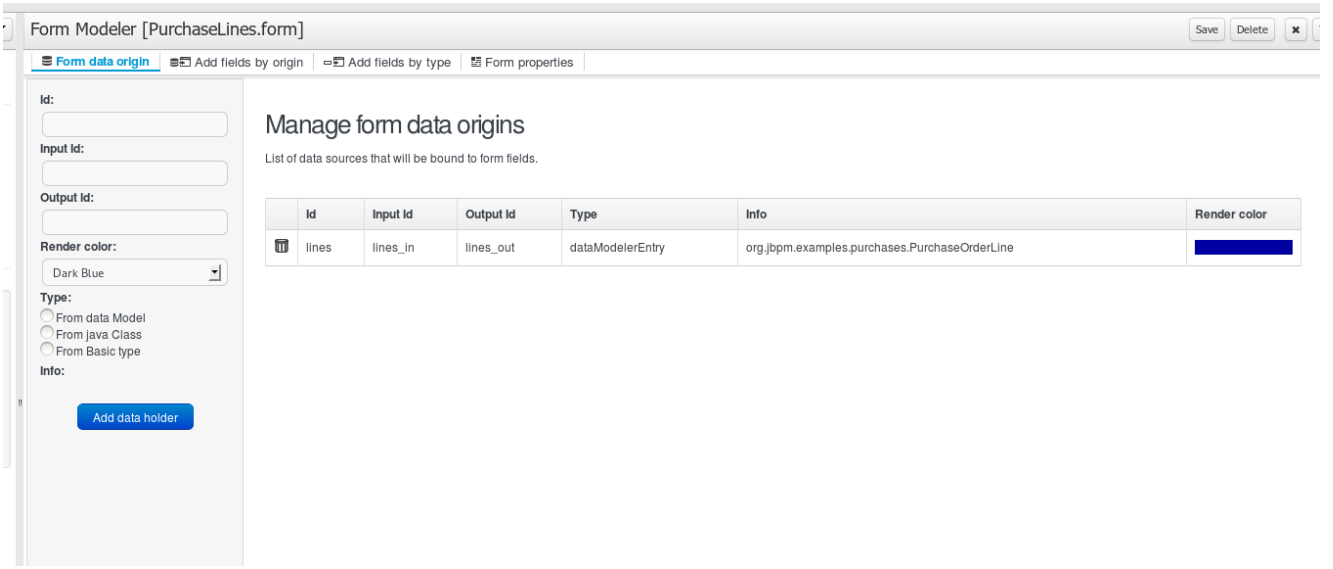


Figure 13.24. Create new data origin

3. Add fields by origin. All the properties are shown, and can be added to the form, either one by one or all of them at once.

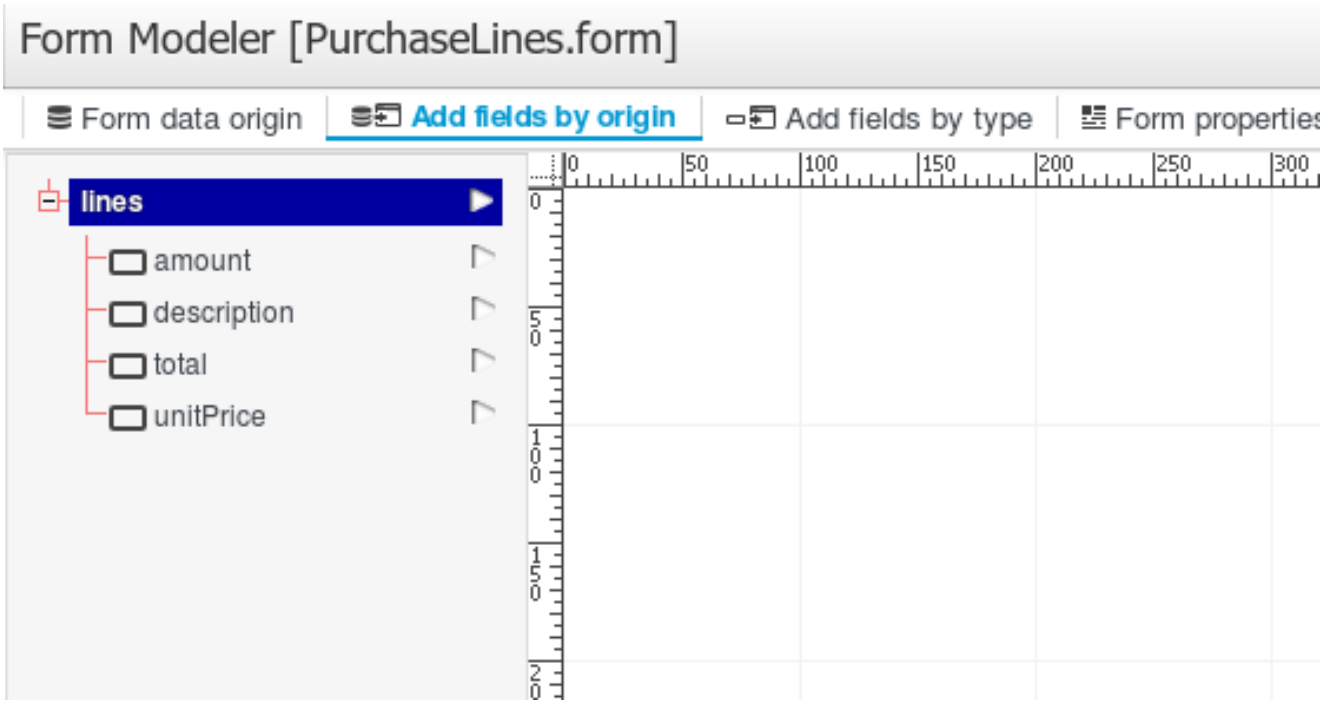


Figure 13.25. Configure the parent form

4. Customize form. Change display options to improve the form visualization

5. Configure the fields. After creating the basic form structure, we can use a formula to calculate automatically the total field. This formulas and expressions are described in *Formula & expression section*.

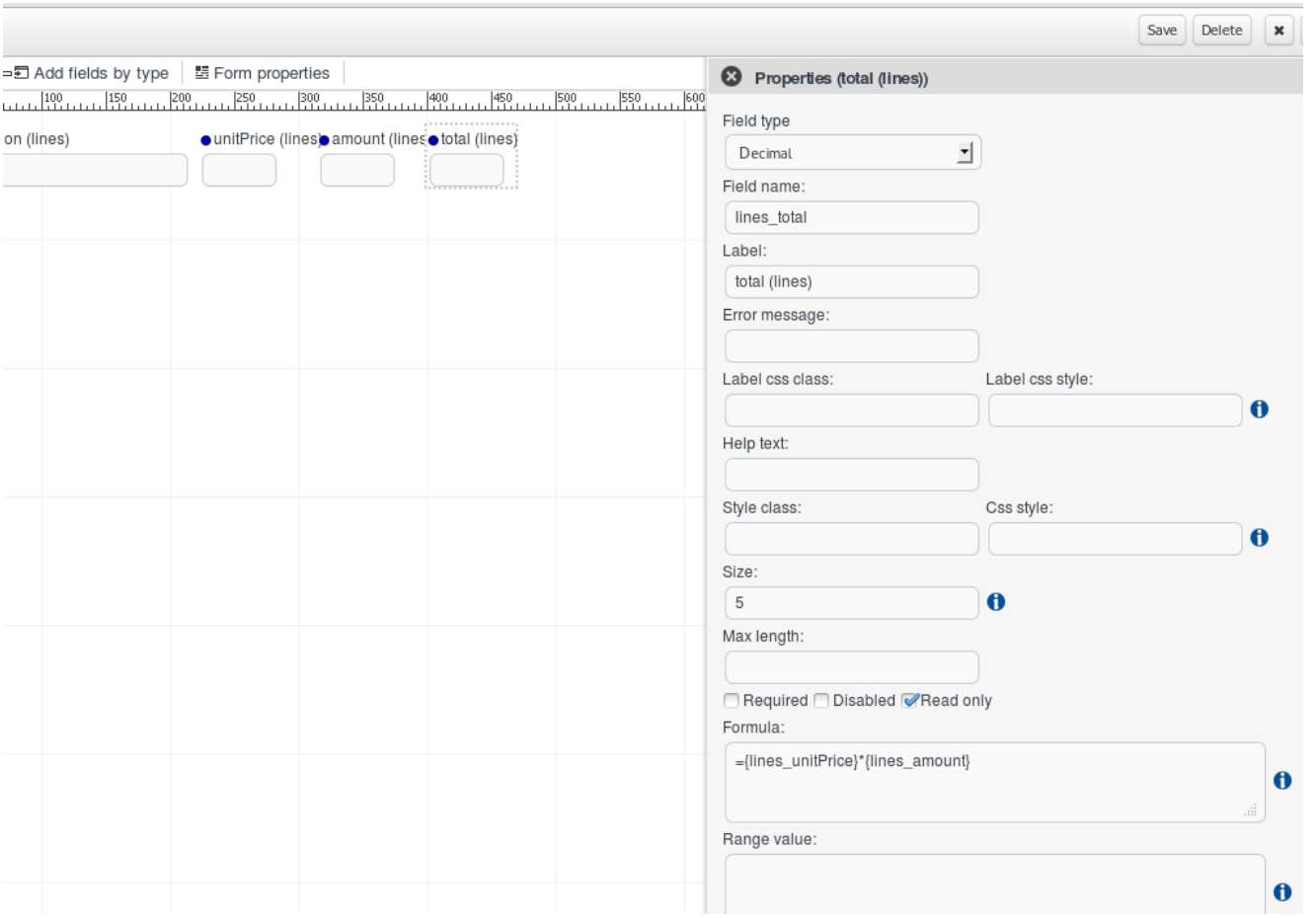


Figure 13.26. Configuring formulas

6. Finally, we save the lines form and go back to the parent form and configure all the lines properties.

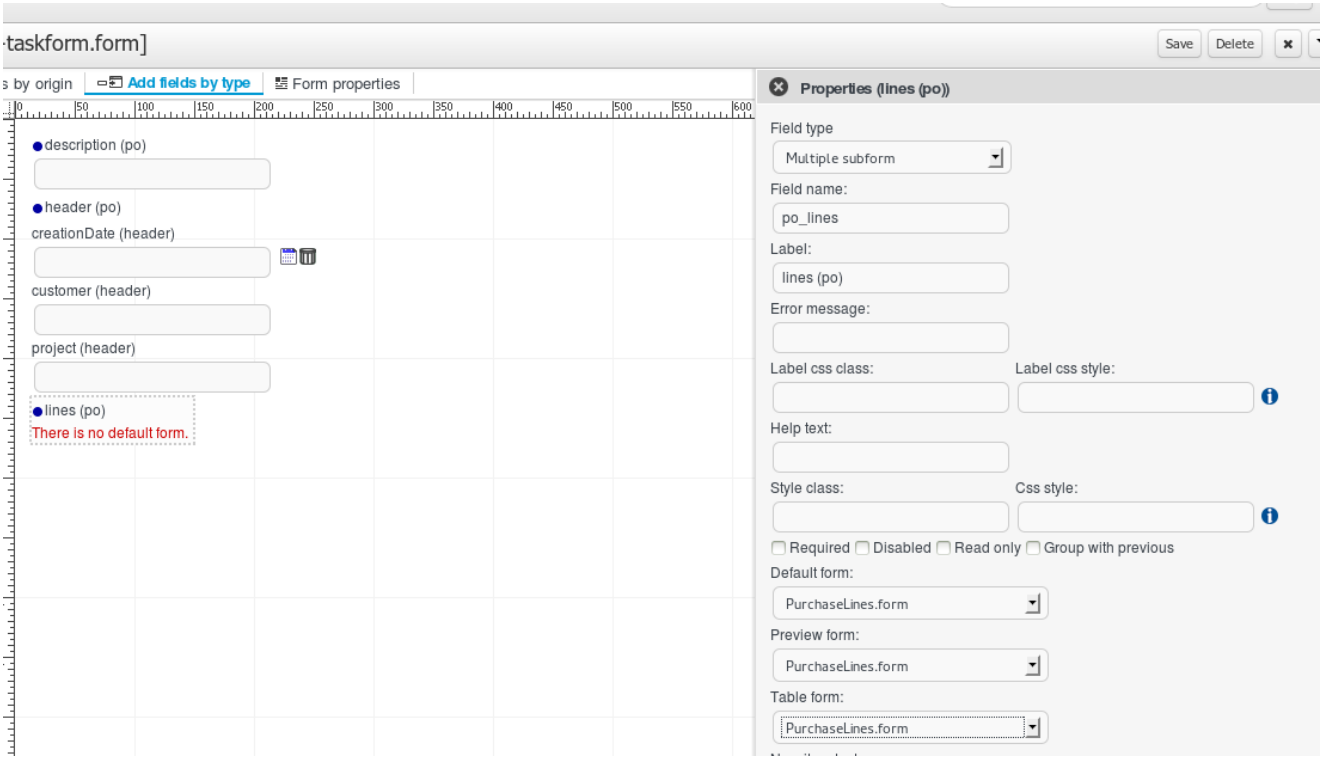


Figure 13.27. Configure the parent form

13.3.2.3.4. Formulas

Form Modeler provides a Formula Engine that you can use to automatically calculate field values. That Formula engine supports Java and XPATH expressions to access the form fields values. Let's see some examples.

- Setting a Default value formula

  Imagine that you have a form that contains a date field "Creation date" that has to be set by default with the current date. To do that you should edit the field properties and set a Default value formula like:

```
=new java.util.Date();
```

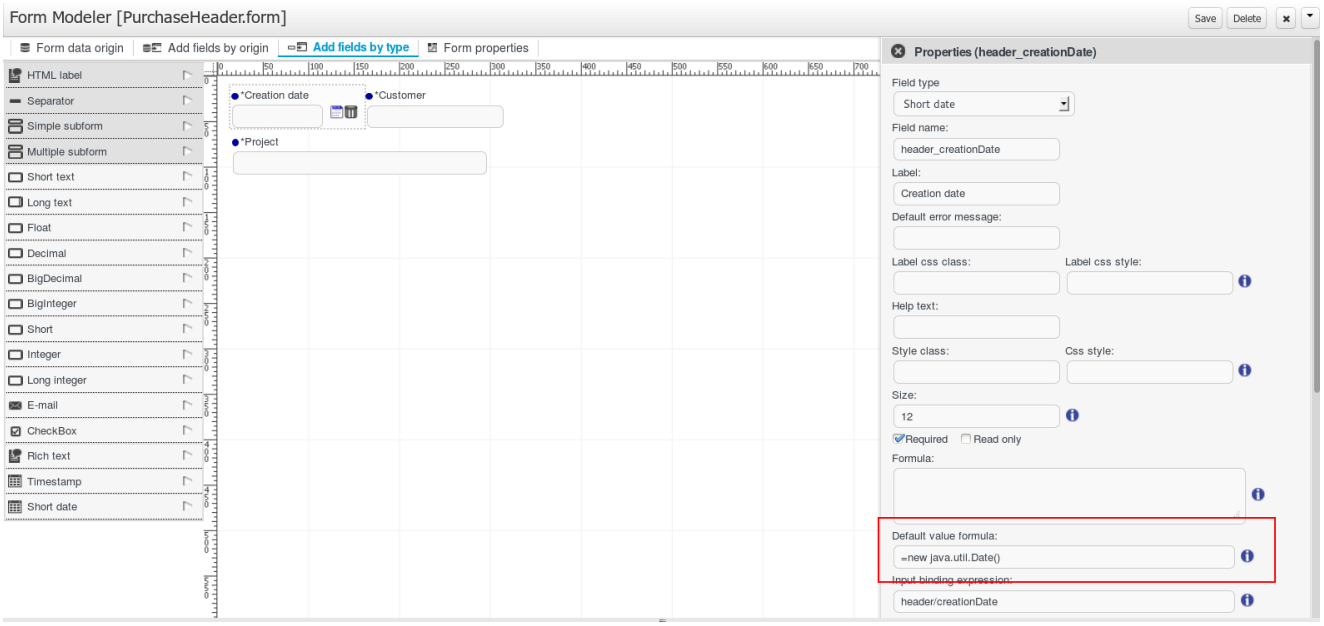


Figure 13.28. Setting default value formula

After setting a Default formula value on a field properties, when the form is rendered by the first time the field will have the specified value.

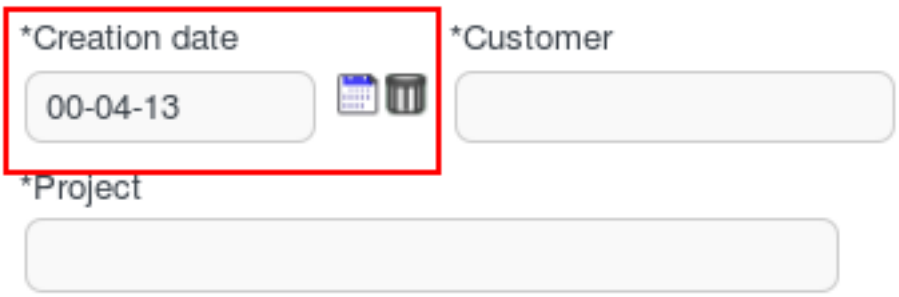


Figure 13.29. Rendering field with default formula

As you can see, you can use a default formula any expression that return a value supported for the field.

- Setting a Formula

The formula engine allows you to calculate formulas that depend on other Field values using XPATH expressions to refer to fields values like {a_field_nane}, standard operators (+, -, *, /, %...) to operate with them or calls to Java Functions for more complex operations.

To start let's see how you can create a formula to calculate the line_total of a Purchase Order Line. Look at the image below and look at the formula on the line_total properties.

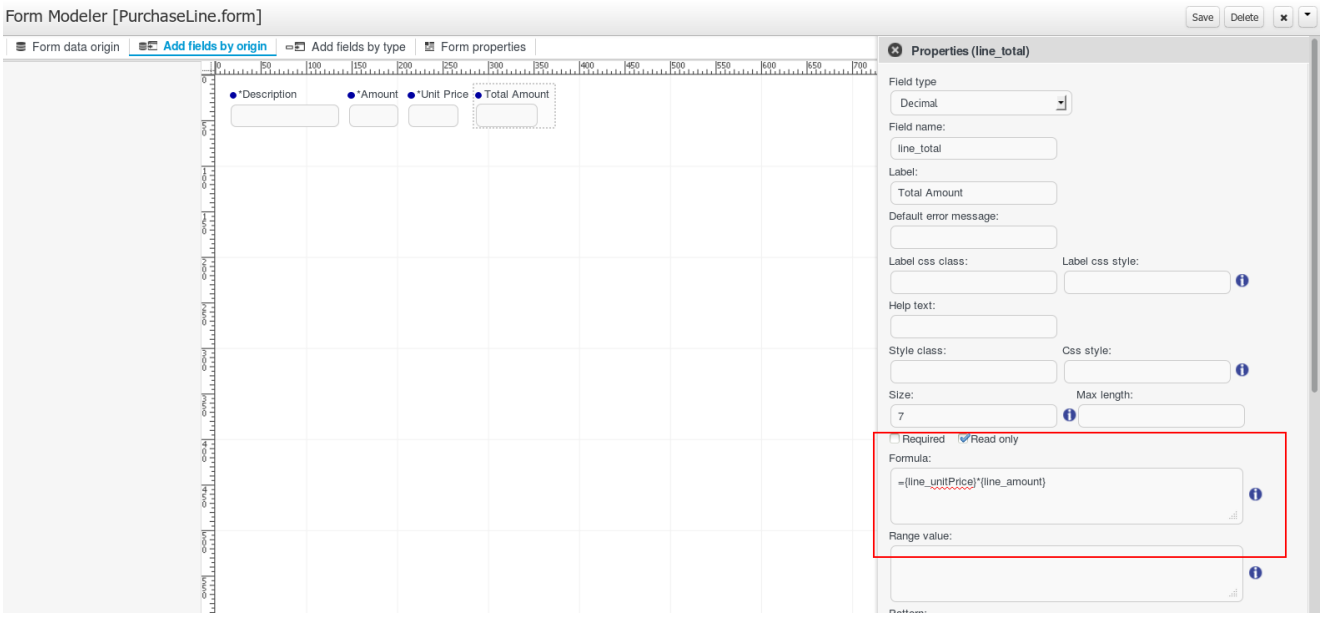


Figure 13.30. Rendering field with default formula

With this expression:

```
={line_unitPrice}*{line_amount}
```

we're forcing the Total of the line value to be the result of the the Unit price multiplied by the Amount, so when the user fills the Amount and Unit Price fields automatically the Total Amount field value is going to be calculated and filled with the operation result:

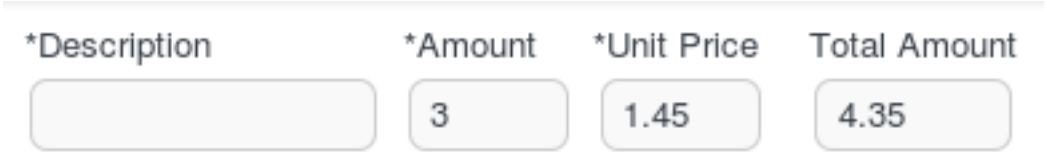


Figure 13.31. Rendering field with default formula result

Is possible to create formulas to operate with values stored in subforms using expressions like

```
={a_field/a_subform_field}
```

Look at the next image to see how it works:

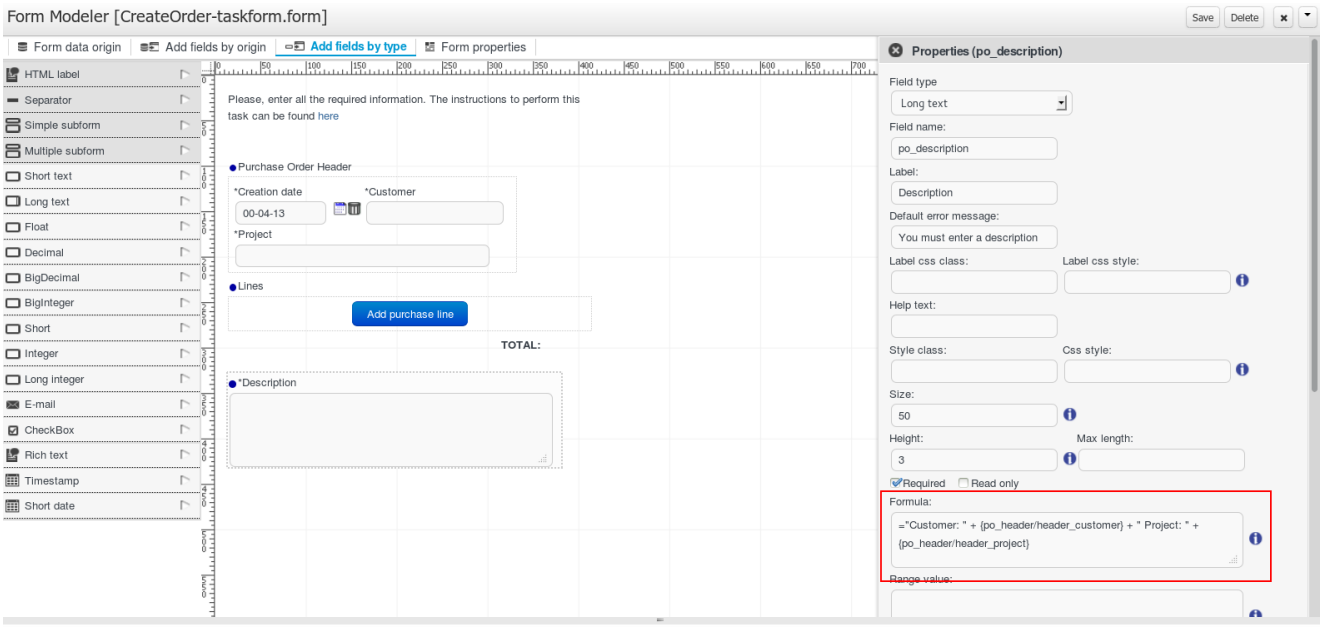


Figure 13.32.

This form has a subform field called po_header that is showing a form with the fields header_creationDate, header_customer and header_project. We want the Description field on our parent form to show some information from the header. Look at the Description field properties formula.

```
="Customer: " + {po_header/header_customer} + " Project: " + {po_header/
header_project}
```

This formula returns a text when the fields header_customer and header_projects are filled on the child form, so from now the parent form will be filled like this:

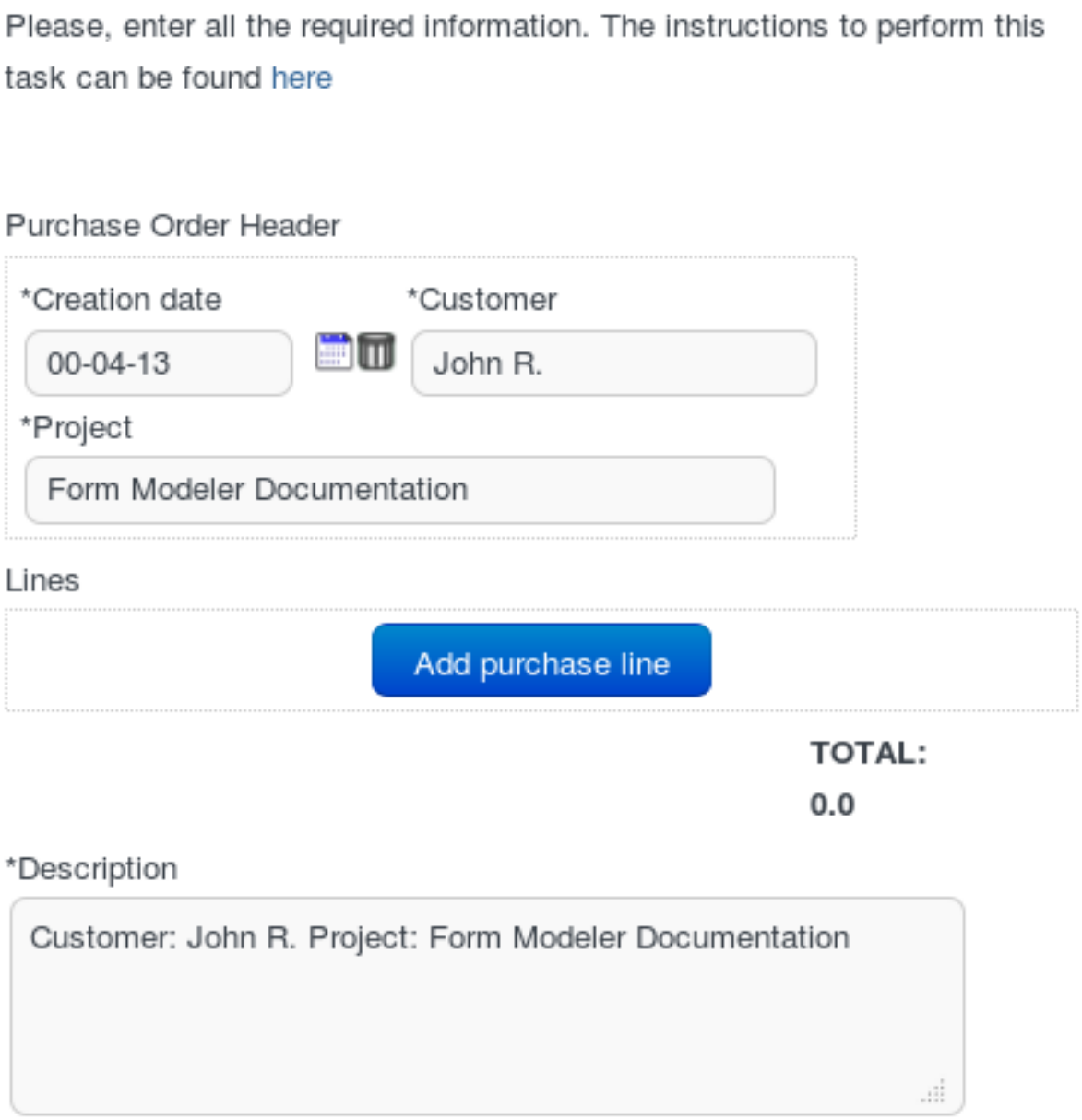


Figure 13.33.

Ok, you've seen how to create formulas that access to a subform fields values, now we are going to see how to work with values stored in Multiple Subforms. Imagine that we have a Purchase Order Line form that contains a multiple subform of Purchase Order Lines, and we want to calculate the total amount of the lines created. Look at the image below and how the TOTAL field is configured.

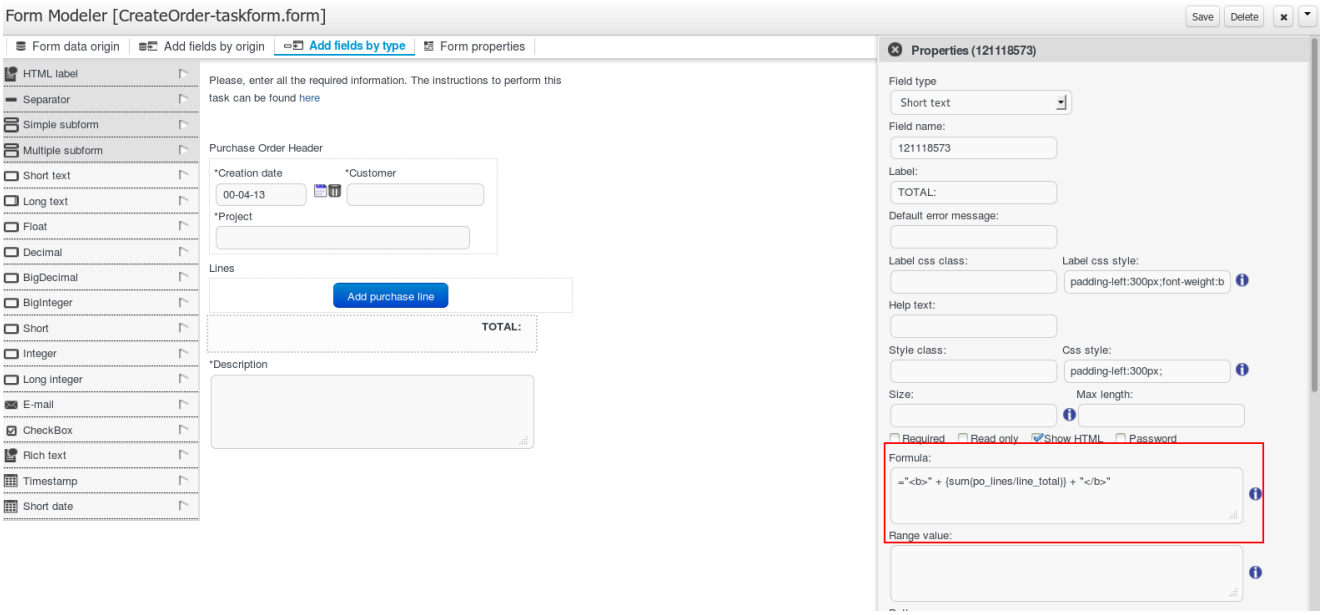


Figure 13.34.

On the formula expression:

```
="
<b>" + {sum(po_lines/line_total)} + "</b>
"
```

we are using the XPATH function sum() that is going to summarize the totals of all the lines. So after creating some Lines the form will look like this:

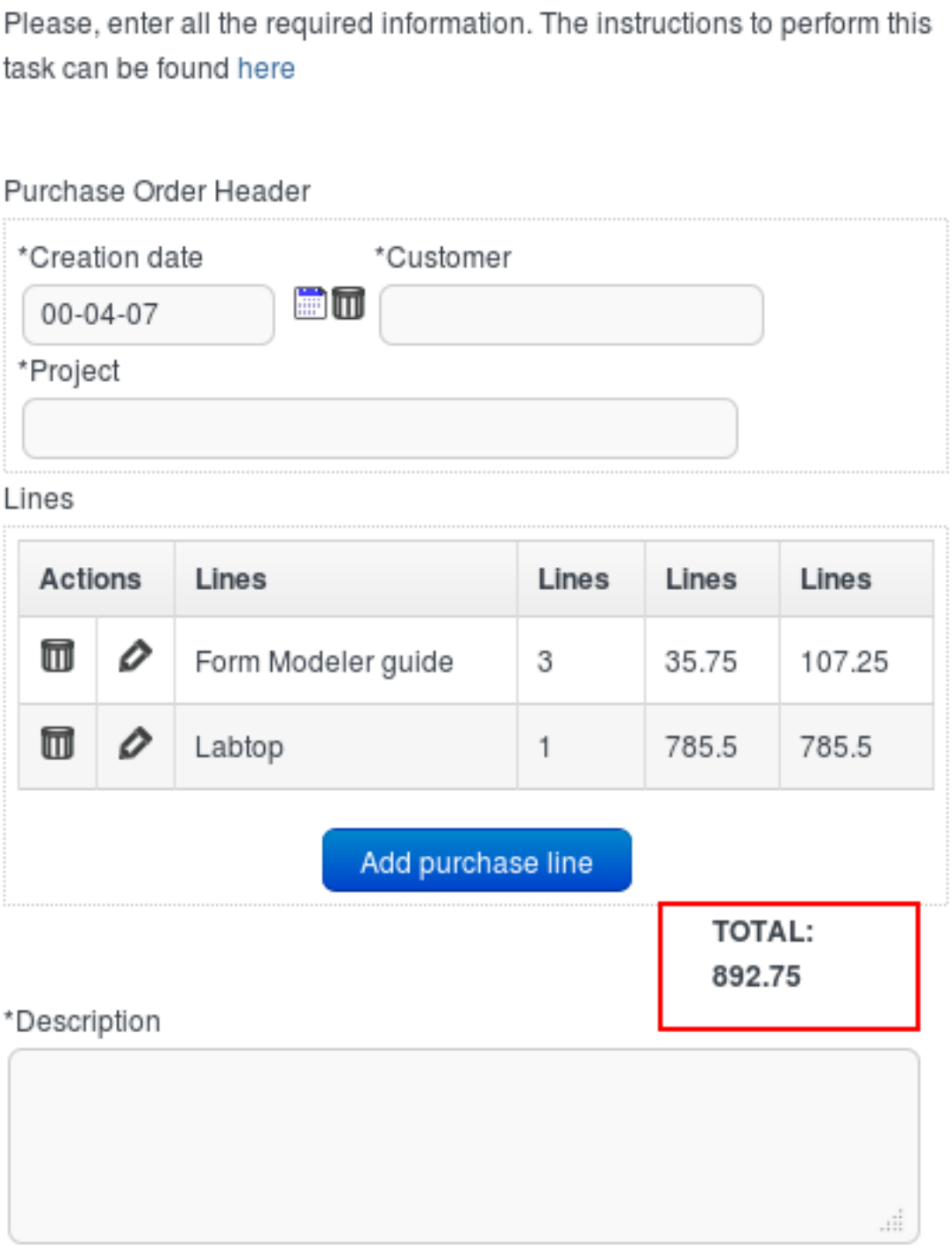


Figure 13.35.

Note that the line_total child field corresponds with the field line_total field on then form selected as a Default Form selected on the Lines field configuration.

On this sample we are using the sum() XPATH function to calculate the total of the Purchase Order, but XPATH provides a lot of possibilities to select values from a set of children and also a lot functions to summarize values (sum, count, avg...). For more information about XPATH you can take a look at *http://www.w3schools.com/xpath/*

- Setting a Range Formula

A range formula allows you to let you specify the values that the user can select from an specific field, showing it like a select box. It can be used on all simple types except Dates and Checkboxes.

To see how it works look the next image and look at the Review Status field configuration.

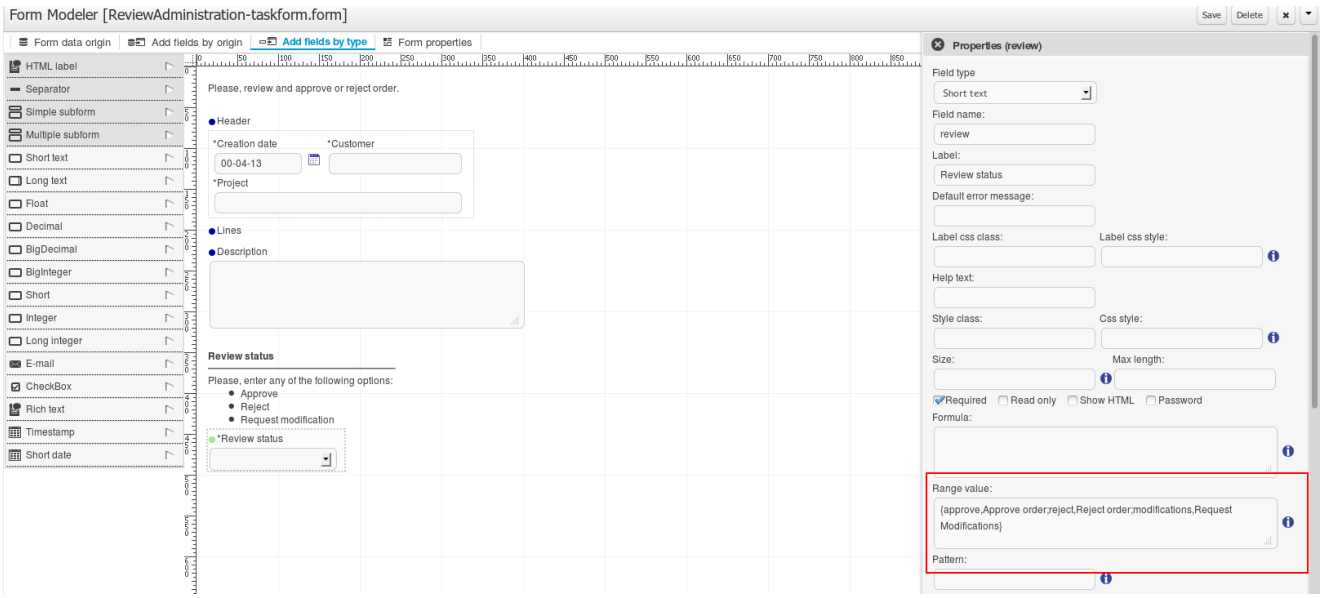


Figure 13.36. Setting default value formula

As you can see that field is being shown as a select box and it has a range formula that specifies the values like this:

```
{approve,Approve order;reject,Reject order;modifications,Request
 Modifications}
```

This expression is defining 3 duos of value/"text to show" separated with the character ',' and each of this duos is separated from each other other with the ';' character. So due this formula the resulting select box will show:

Table 13.2.

Value stored in input	Text shown on Select Box
approve	Approve order
reject	Reject order
modifications	Request Modifications

13.3.2.4. Customizing form layout

When you need an extra customization level and have more control over the HTML that is displayed. The form modeler provides the ability to edit the HTML directly.

To use this functionality, the user have to specify that in the ‘Form properties’ tab, 'Custom form layout' option and save.

Now the form is displayed with the custom HTML. To access this HTML editing, we click on the icon 'Edit'

The HTML editor is displayed; the HTML code will define how the form has to be shown. In this editor the user can directly create the HTML i locate the fields and labels with the syntax described below:

$field{fieldName} for field identified fieldName

$label{fieldName} for field identified fieldName label

These expressions will be replaced by the field or label rendering when the form will be shown.

Form modeler also provides two ways to help in the form HTML creation.

- 'Insert form elements'

  Two select: one for the fields and another for the labels. Clicking on that, the field or label text is added to HTML. These selects only show the form fields haven’t been added yet.

- 'Generate template based on'

  This functionality generates the HTML using all fields (default, alignment fields or Not aligned) depending on the selected value and overwrite the HTML.

13.3.3. Field types

There are three types of field types that you can use to model your form:

- Simple types

  These field types are used to represent simple properties like texts, numeric, dates, etc. The supported Field types are:

Table 13.3. Field types

Name	Description	Java Type	Default on generated forms
Short Text	Simple input to enter short texts.	java.lang.String	yes
Long Text	Text area to enter long text.	java.lang.String	no
Rich Text	HTMLEditor to enter formatted texts .	java.lang.String	no
Email	Simple input to enter short text with email pattern.	java.lang.String	no
Float	Input to enter short decimals.	java.lang.Float	yes
Decimal	Input to enter number with decimals.	java.lang.Double	yes
BigDecimal	Input to enter big decimal numbers.	java.math.BigDecimal	yes
BigInteger	Input to enter big integers.	java.math.BigInteger	yes
Short	Input to enter short integers	java.lang.Short	yes
Integer	Input to enter integers.	java.lang.Integer	yes
Long Integer	Input to enter long integers	java.lang.Long	yes
Checkbox	Checkbox to enter true/false values	java.lang.Boolean	yes
Timestamp	Input to enter date & time values	java.util.Date	yes
Short Date	Input to enter date values.	java.util.Date	no

- Complex types

  These field types are made to deal with properties that are Java Objects instead of basic types. These field types need extra forms to be created in order to show and write values onto the specified Java Object/s

Table 13.4. Complex types

Name	Description	Java Type	Default on generated forms
Simple subform	Renders the a form, it is used to deal with 1:1 relationships.	java.lang.Object	yes
Multiple subform	This field type is used to deal with 1:N relationships. It allows to create, edit and delete a set child Objects.Text area to enter long text.	java.util.List	yes

- Decorators

  Decorators are a type of field types that don't store data in the Object shown on the form. They can be used with aesthetic purpose

Table 13.5. Decorators

Name	Description
HTML label	Allows the user to create HTML code that will be rendered in the form
Separator	Renders an HTML separator

13.3.3.1. Custom Field Types

Is possible to extend the platform to add Custom Field Types that make a specific field (of any type) on the form to look and behave totally different than the standard platform fields. On this section we will take a look on how to create them and how to configure them.

13.3.3.1.1. How to create Custom Field Types

Basically a Custom Field Type is a Java class that implements the *org.jbpm.formModeler.core.fieldTypes.CustomFieldType* interface and is packaged inside inside a JAR file that is placed on the Application Server classpath or inside the application WAR.

Lets take a look at*org.jbpm.formModeler.core.fieldTypes.CustomFieldType*:

```
package org.jbpm.formModeler.core.fieldTypes;

import java.util.Locale;
```

```
import java.util.Map;

/**
 * Definition interface for custom fields
 */
public interface CustomFieldType {
    /**
     * This method returns a text definition for the custom type. This text will be shown on t
     * @param locale The current user locale
     * @return A String that describes the field type on the specified locale.
     */
    public String getDescription(Locale locale);

    /**
     * This method returns a string that contains the HTML code that will be used to show the
     * shown on screen
     * @param value The current field value
     * @param fieldName The field name
     * @param namespace The unique id for the rendered form, it should be used to generate ide
     * @param required Determines if the field is required or not
     * @param readonly Determines if the field must be shown on read only mode
     * @param params A list of configuration params that can be set on the field configuration
     * @return The HTML that will be used to show the field value
     */
    public String getShowHTML(Object value, String fieldName, String namespace, boolean requi

    /**
     * This method returns a String that contains the HTML code that will show the input view
     * @param value The current field value
     * @param fieldName The field name
     * @param namespace The unique id for the rendered form, it should be used to generate ide
     * @param required Determines if the field is required or not
     * @param readonly Determines if the field must be shown on read only mode
     * @param params A list of configuration params that can be set on the field configuration
    * @return The HTML code that will be used to show the input view of the field.
     */
    public String getInputHTML(Object value, String fieldName, String namespace, boolean requ

    /**
    * This method is used to obtain the field value from the submitted values.
     * @param requestParameters A Map containing the request parameters for the submitted form
     * @param requestFiles A Map containing the java.io.Files uploaded on the request
     * @param fieldName The field name
     * @param namespace The unique id for the rendered form, it should be used to generate ide
     * @param previousValue The previous value of the current field
     * @param required Determines if the field is required or not
     * @param readonly Determines if the field must be shown on read only mode
     * @param params A list of configuration params that can be set on the field configuration
     * @return The value of the field based on the submitted form values.
```

```
        */
        public Object getValue(Map requestParameters, Map requestFiles, String fieldName, String
        }
```

As you can see this Interface defines the methods that determines how the field has to be shown on the screen for when the form is shown on insert(getInputHTML(...)) or readonly (getShowHTML(...)) mode. It also provides the method (getValue(...)) that reads the needed parameters from the request and to obtain the correct field value. Te returned value type must match with the type of the field added on the form. So (for example) you can create a File input that uploads a file to a server folder and saves a String with the storage path as the field value, so on your forms you can turn all the text compatible fields (Short Text, Long Text, Rich Text and Email) on Input File.

To see ho can it be done look at the example on *https://github.com/droolsjbpm/jbpm-form-modeler/tree/master/jbpm-form-modeler-sample-custom-types/jbpm-form-modeler-custom-file-type*.

Please note that this is just a sample and it only should be used with learning purposes.

13.3.3.1.2. Configuring and using Custom Field Types

Now let's see how to use and configure and use a Custom Field type. Following the example on the previous chapter, we have created a File Input type and we have it already installed on our application. So now we are going to create a new form and add a Short Text property and turn it into a File Input and edit the field properties changing the Field Type from *Short text* to*Custom field*.

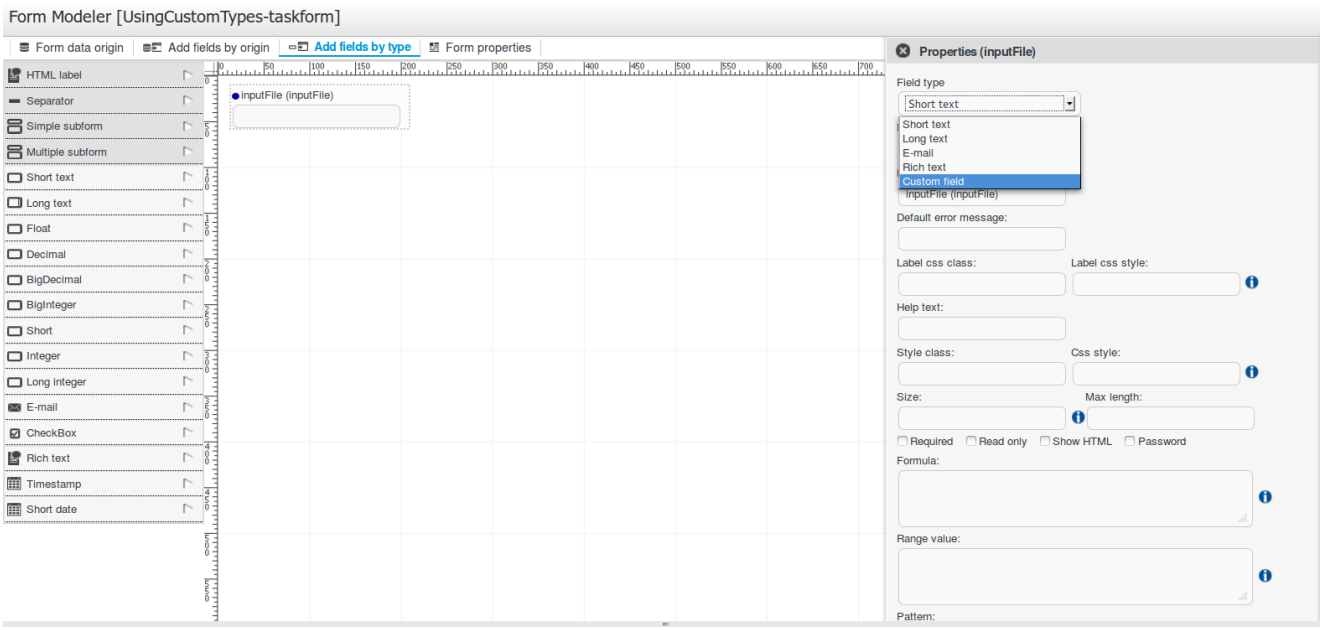


Figure 13.37. Changing a field type to*Custom field*

After changing the field type a new set of properties will appear:

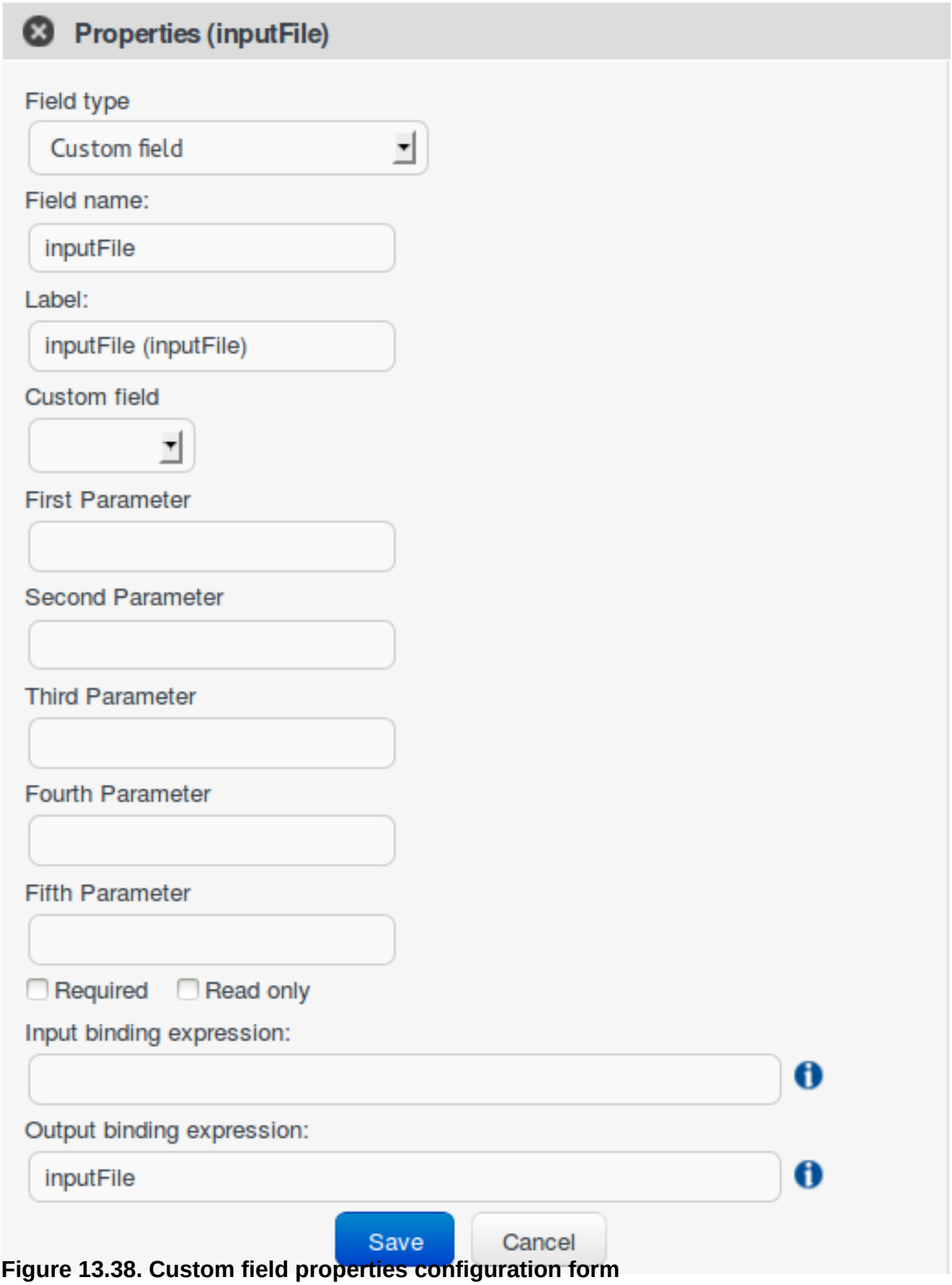


Figure 13.38. Custom field properties configuration form

Table 13.6. Custom field properties

Property	Description
Field type	Can change the field type to other compatible field types
Field Name	Will be used as identifier in formulas calculation
Label	The text that will be shown as field label
Custom field	A list containing all the Custom Field Types available on the platform
First parameter	A String parameter that can be user to pass custom configuration needed by the Custom Field Type implementation
Second parameter	A String parameter that can be user to pass custom configuration needed by the Custom Field Type implementation
Third parameter	A String parameter that can be user to pass custom configuration needed by the Custom Field Type implementation
Fourth parameter	A String parameter that can be user to pass custom configuration needed by the Custom Field Type implementation
Fifth parameter	A String parameter that can be user to pass custom configuration needed by the Custom Field Type implementation
Required	Indicates if it’s mandatory to fill this field.
Read Only	When this check is on, the field will be used only for read
Input binding expression	This expression defines the link between field and process task input variable. It will be used in runtime to set the field value with that task input variable data.
Output binding expression	This expression defines the link between field and process task output variable. It will be used in runtime to set that task output variable.

So opening the Custom field select box we'll be able to select the *File Input* from the available custom types:

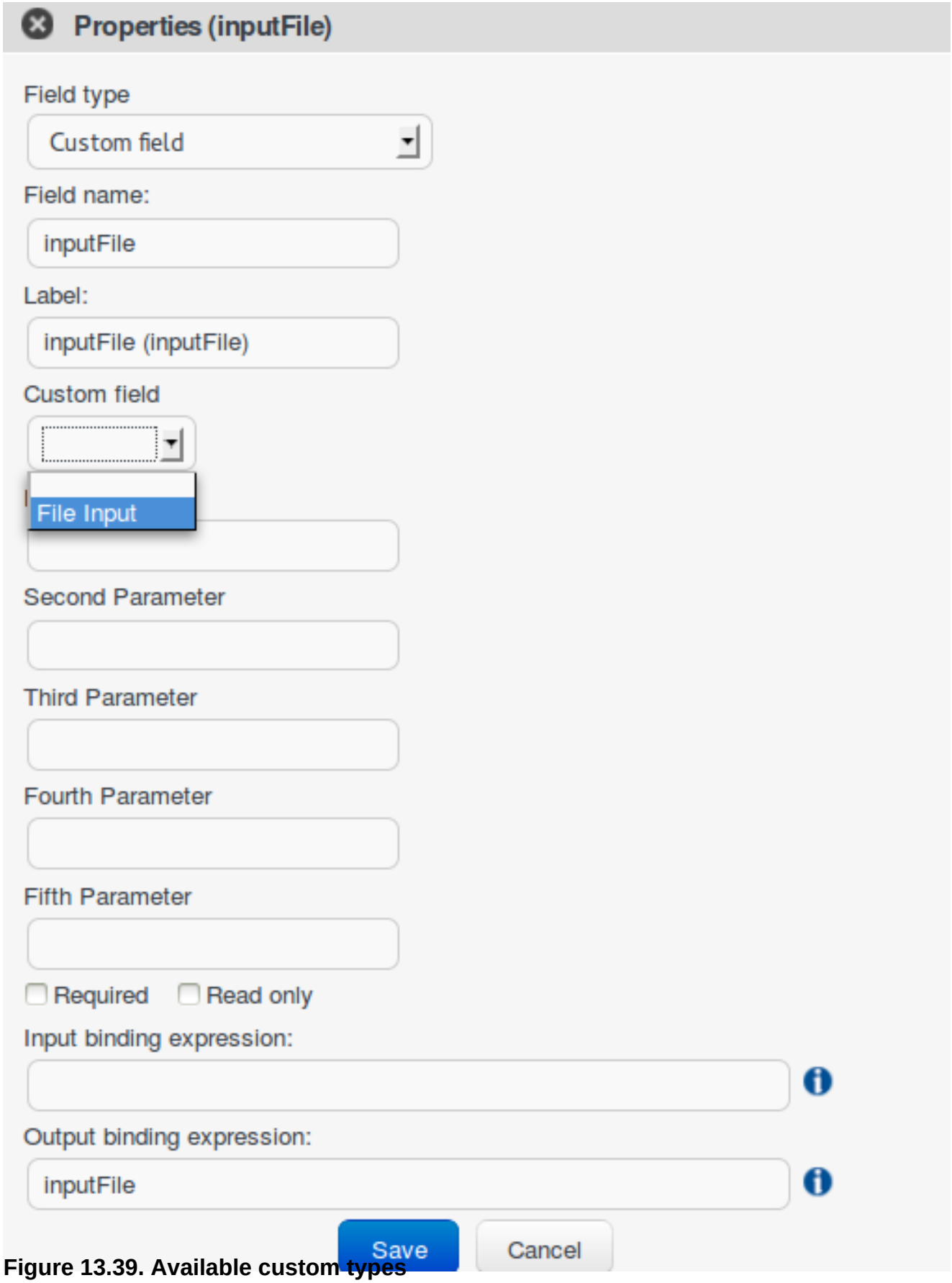


Figure 13.39. Available custom types

After selecting the *File Input* type on the list and saving the field properties the form will look like:

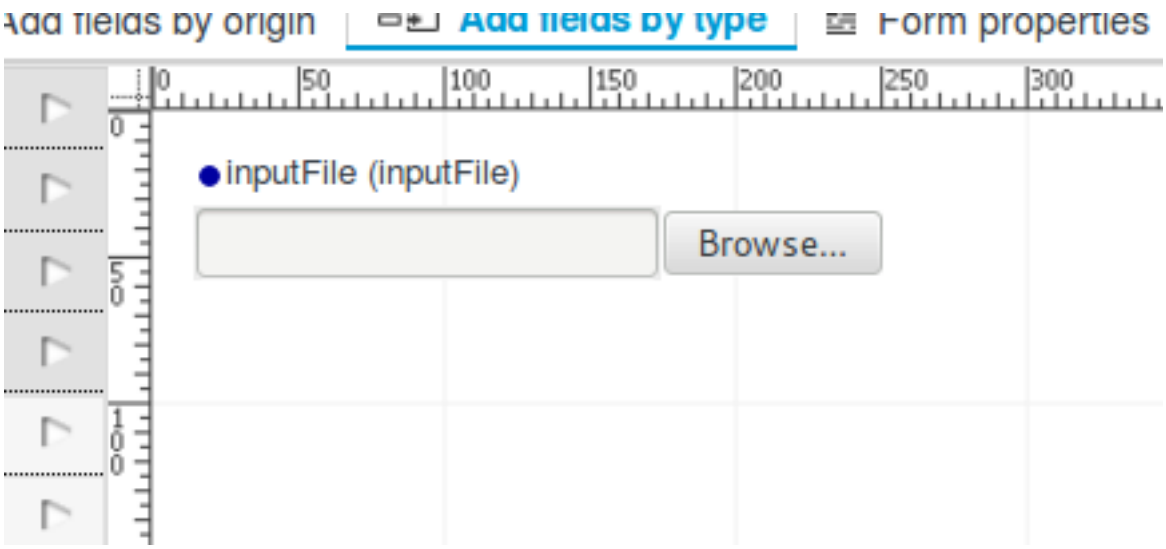


Figure 13.40. Custom type display in a form

If we build a simple process and configure a *Short text* to be shown as the sample*File Input*, if we build the project on runtime the field will behave uploading the chosen files to the server and allowing the user to download it like this:

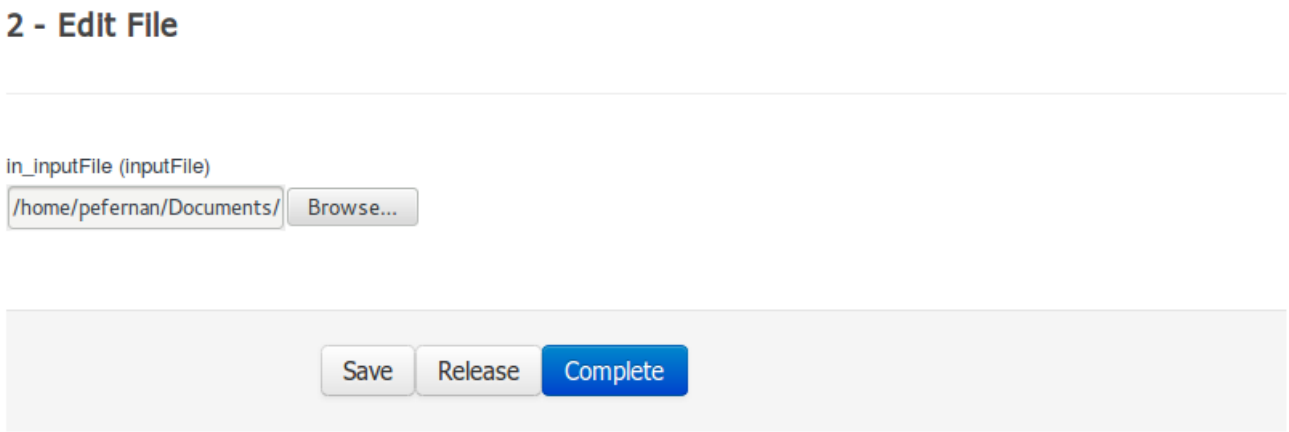


Figure 13.41. Choosing the file to upload

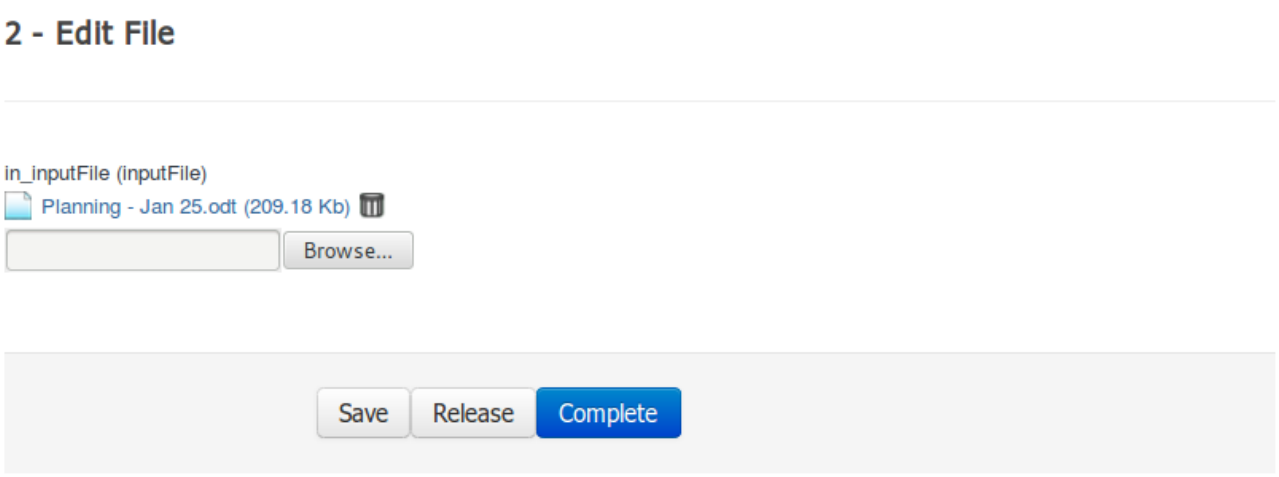


Figure 13.42. File uploaded, showing the download link

If we take a look at what's the process variable value, we'll see that is storing a String with the file path stored in server.

Figure 13.43. Process variable storing custom type results

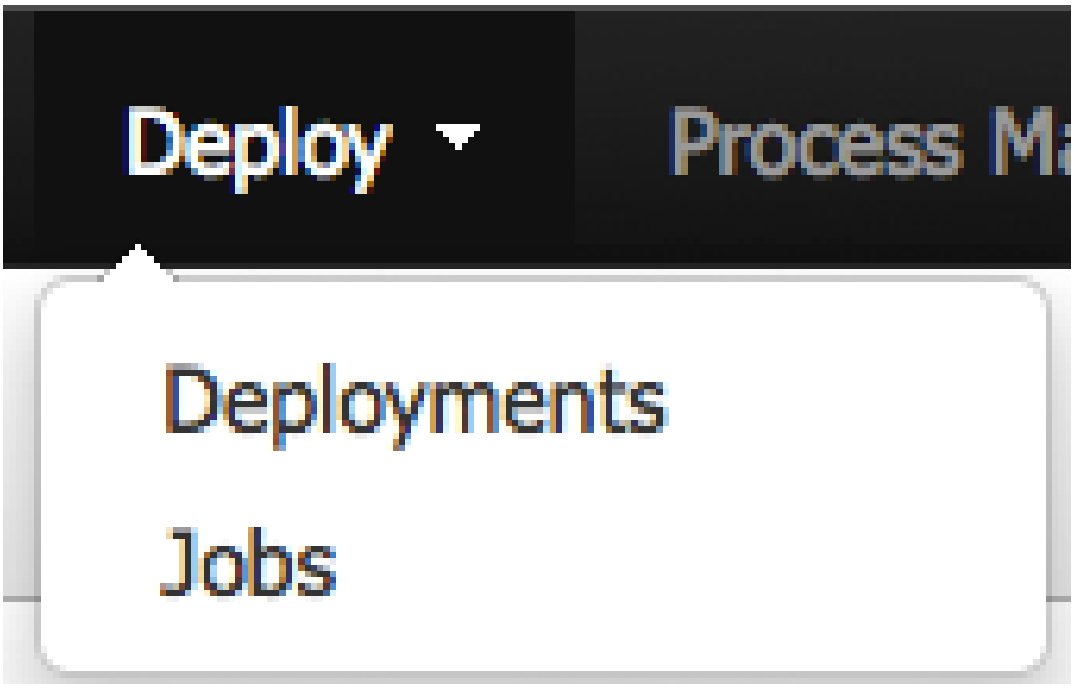
Chapter 14. Runtime Management

14.1. Deployments

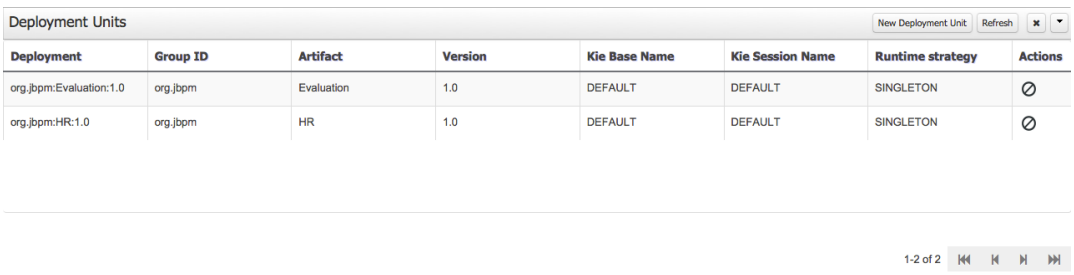
This chapter introduces the Deployment administration screen. Technical users will be able to check which deployment units are deployed into the platform and available to use. You can find the source code of these screens here: https://github.com/droolsjbpm/jbpm-console-ng/tree/master/jbpm-console-ng-business-domain

14.1.1. Deployment Units List

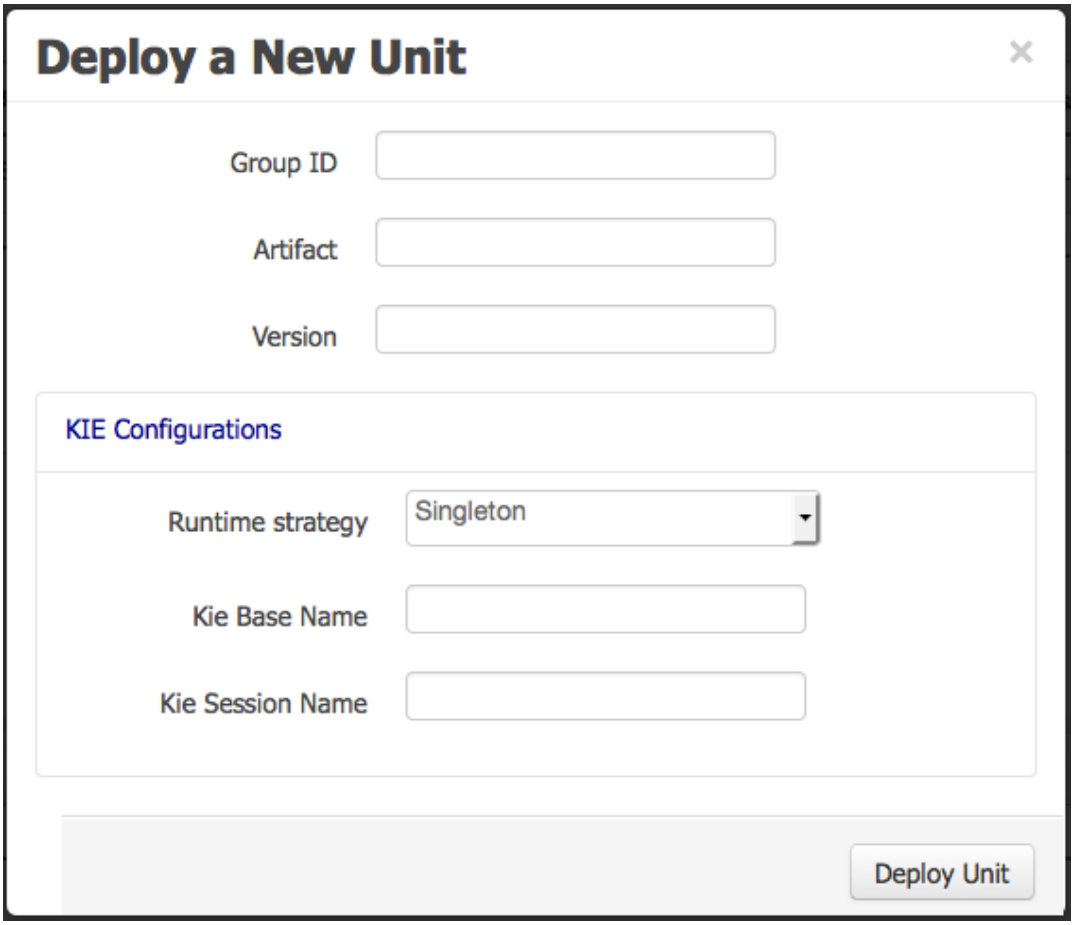
You can access to the Deployment Units List under the Runtime menu (TODO: fix image and menu name)



The Deployment Unit list shows all the Deployment Units deployed into the platform that are already enabled to be used. Each deployment unit can contain multiple business processes and business rules. By default the list is populated by Building and Deploying a KIE Module using the Project Editor Screen. When you Build and Deploy a



You also have the option to deploy custom Deployment Units with other options different from the defaults. When a KIE Project is deployed, by default the "DEFAULT" KIE Base and KIE Sessions are used and the SINGLETON Strategy is used. You can select a different KIE Base and KIE Session using the New Deployment Unit.



14.2. Jobs

TBD

Chapter 15. Process and Task Management

15.1. Process Management

This chapter describes the screens related with the creation and management of process definitions and process instances.

Once you have modelled and configured all the technical details to run a process definition your process definition will appear in the Process Definitions List. Once you have the process in the Process Definition List, you can start new instances of it. The following sections describes the features provided by each of these screens. You can find these screens under the Process Management Menu, in the jBPM Console NG or in Kie Workbench.

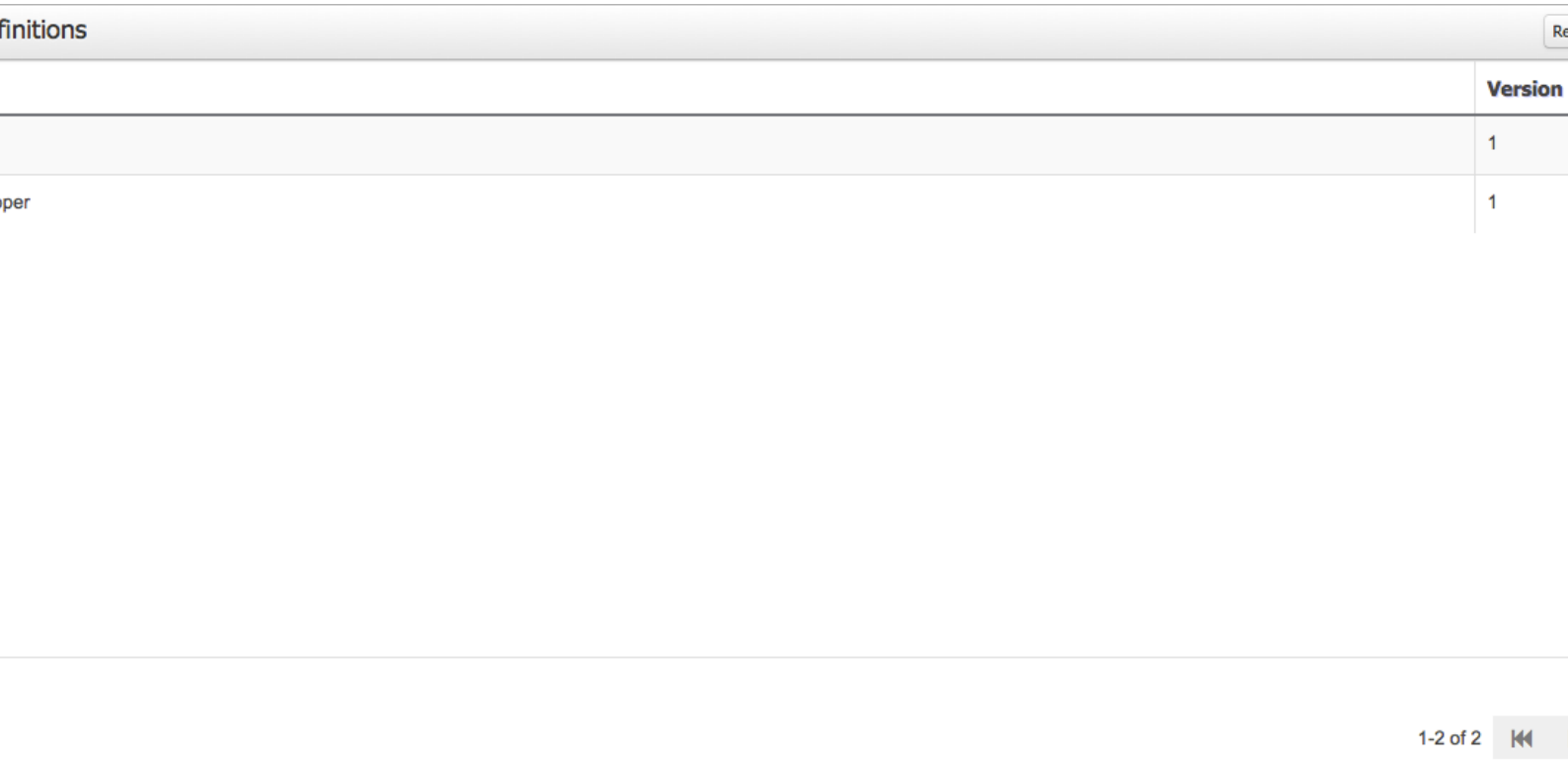
You can find the source code for this module here: https://github.com/droolsjbpm/jbpm-console-ng/tree/master/jbpm-console-ng-process-runtime

15.1.1. Process Definitions

The process definition section is composed by two main screens: the Process Definition Lists and the Process Definition Details.

15.1.1.1. The Process Definition List

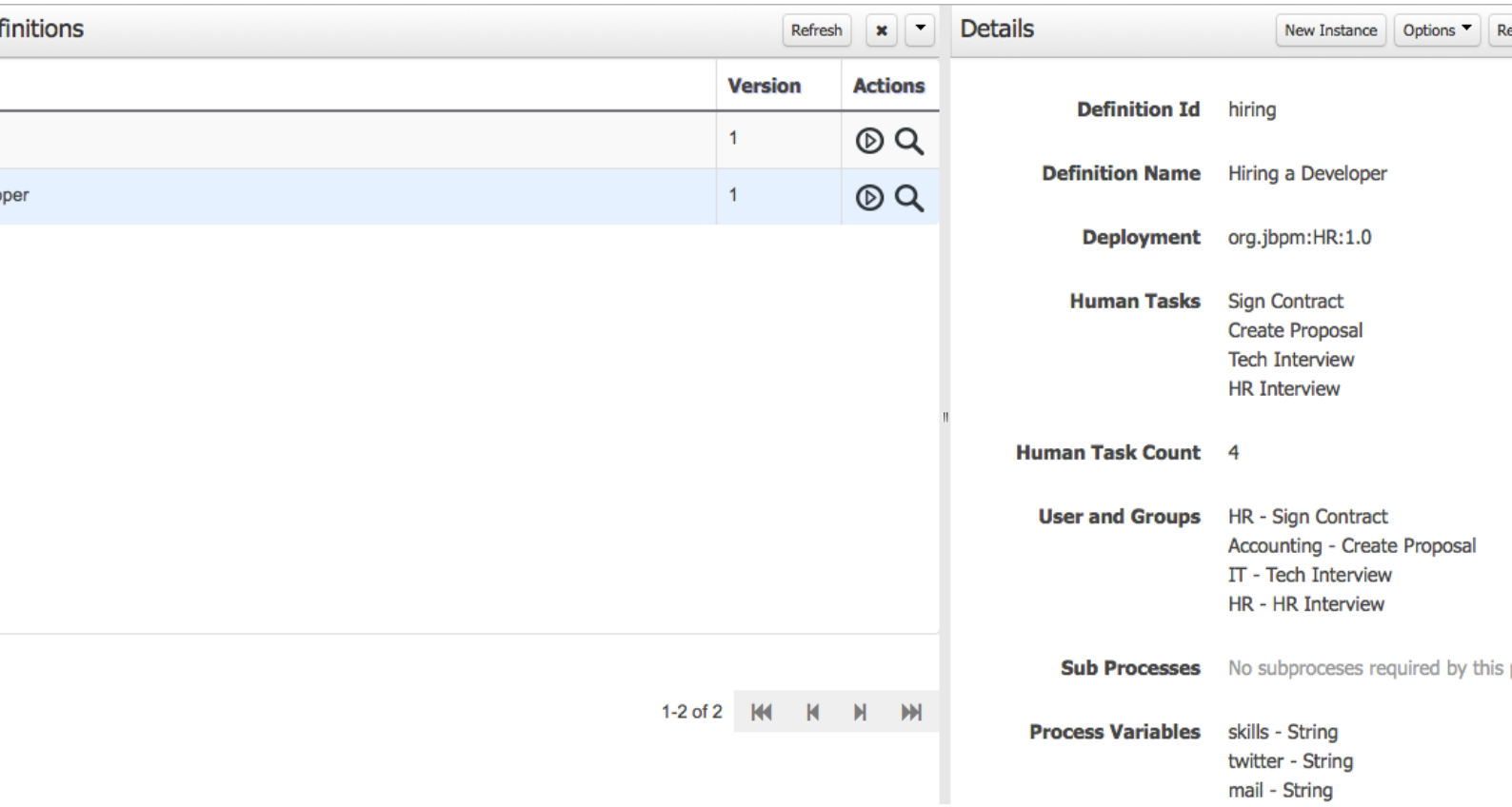
The process definition list shows all the available process definitions that were deployed into the platform. Look at the Deployments section for more information about how to check all the deployment units available.



You can click in the list rows to access to the details of the process definition.

15.1.1.2. The Process Definition Details

The process definition details shows all the available information about the process definition. You can consider this screen as a brief about the process model. You can quickly see if there is a Sub Process associated with it, or how many users and groups are participating in the selected definition.



Notice that you can View the Process Model (Read Only mode) using the Options Menu in the top bar. You can also look at all the process instances for the selected process definition by going to Options -> View Process Instances.

15.1.1.3. Creating Process Instances

You can create new Process Instances from the Process Definition List or from the Process Definition Detail view.

When you want to create a Process Instance usually a Form will be presented to introduce the information required by the process to be started. Once you complete the form and click into the Start Process button, the instance will be created and the details of the Process Instance will be displayed on top of the Process Definition Details.

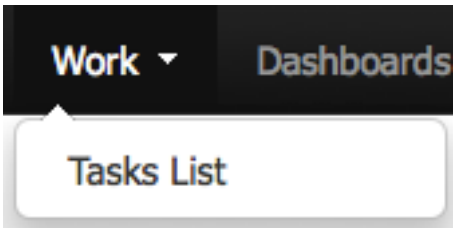
15.2. Tasks

This chapter introduces the Task Management screens and the its integration with the Form Modeller component to allow users to work on their assigned tasks. You can find the source code of these screens here: *https://github.com/droolsjbpm/jbpm-console-ng/tree/master/jbpm-console-ng-human-tasks* [https://github.com/droolsjbpm/jbpm-console-ng/tree/master/jbpm-console-ng-human-tasks] . At the end of this section you will find a technical description about how to customize these views.

15.2.1. Task List

Every user with access to the platform will have access to its personal task list where tasks assigned to him/her will be displayed. Each user will be able to create its own personal tasks or work on tasks that were create as a result of a business process execution.

You can access to the Task List under the **Work** main menu:

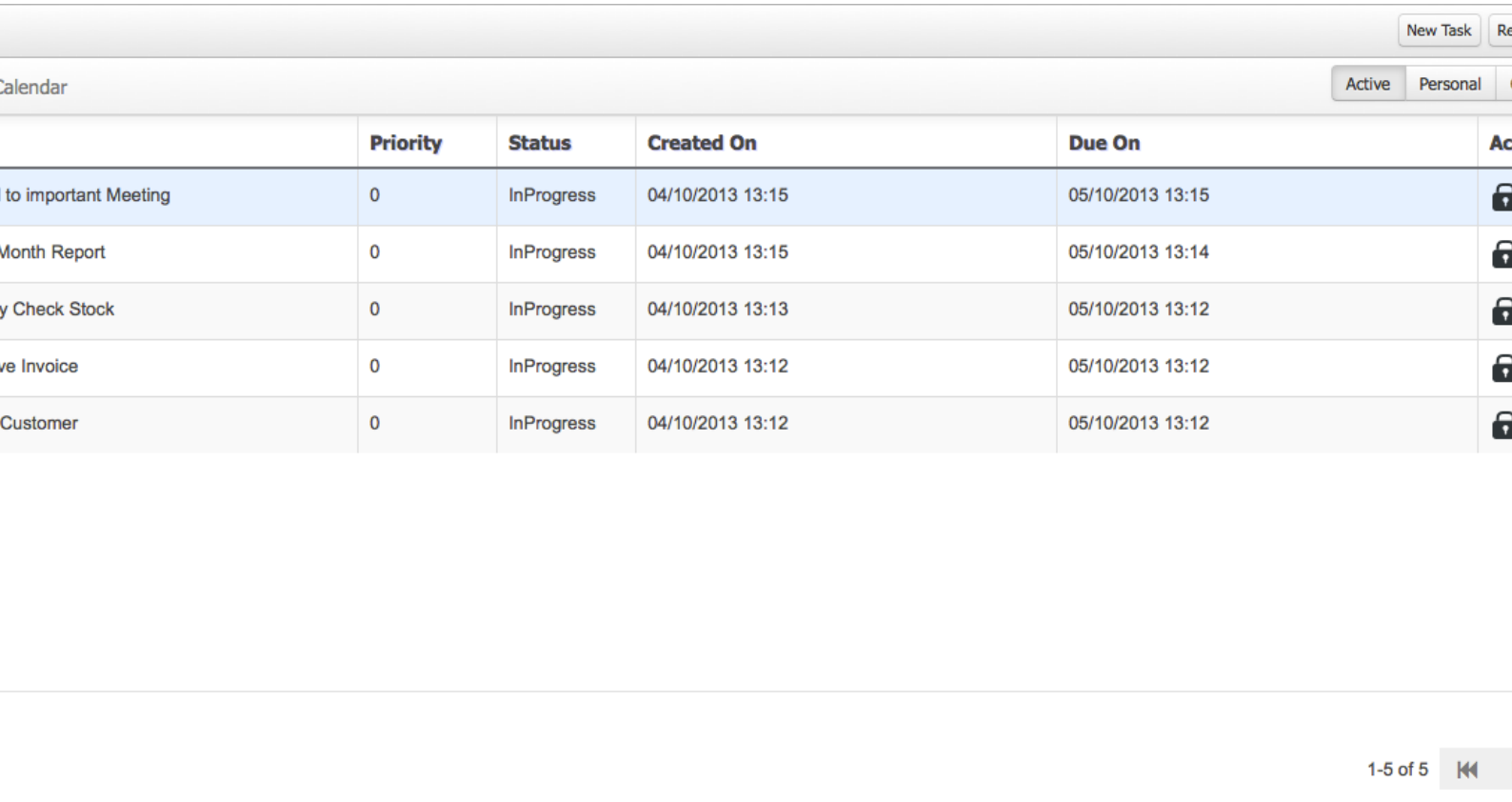


15.2.1.1. Task List (Personal and Group Tasks)

Pending tasks can be displayed using different metaphors depending on what the user is interested on. We are currently providing two different views explained in the sections below: **Grid** and **Calendar** View.

15.2.1.1.1. Task List (Grid View)

If you are interested in having a tabular view of all the pending tasks for a specific person or group you can use the **Grid** View. The list will show all the pending tasks ordered by the columns presented. You can change the default ordering clicking on the column header. In future version more advanced filters will be provided and the search mechanism will be improved to look for task internal data. This view offer a more traditional BPM Task List view.



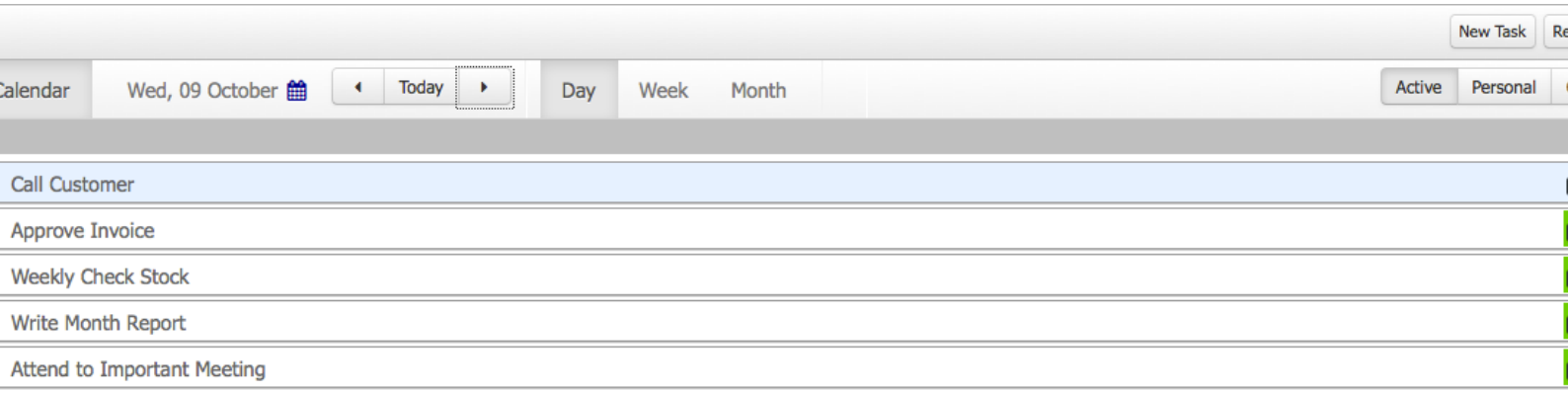
With this current version you can filter based on the tasks status:

- Active: all the Active tasks that user can work on. That means Personal and Group Tasks.
- Personal: all the personal tasks that already belong to the user.
- Group: all the group tasks that needs to be claimed by the user in order to start working on them.
- All: show all the tasks no matter the status. It will show completed tasks as well with the exception of completed tasks that belongs to a process that is already finished. In such cases the tasks are cleaned up after the process is completed and for that reason they will not be displayed.

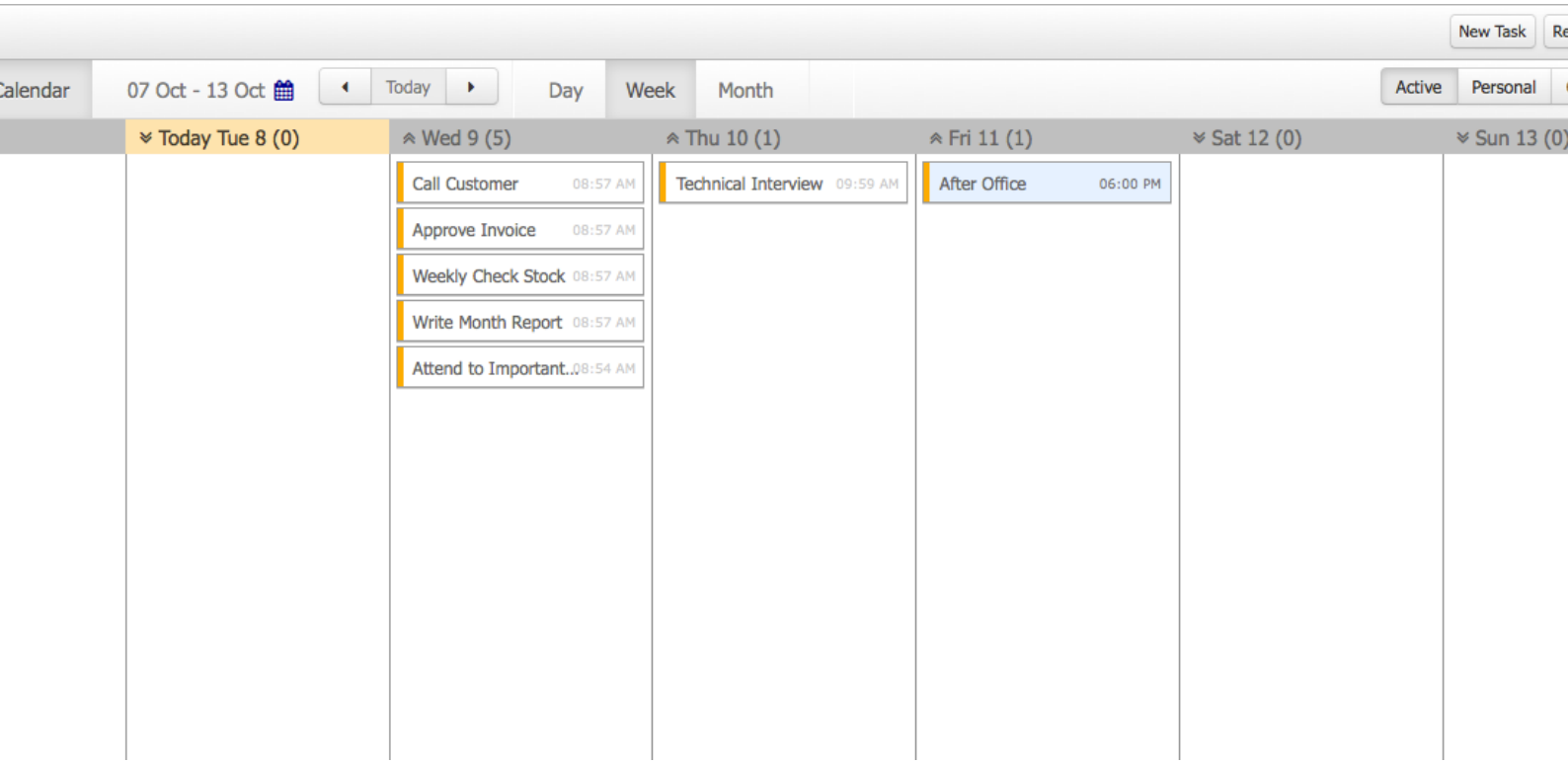
15.2.1.1.2. Task List (Calendar View)

If you want a more time oriented view of your pending tasks you can use the Calendar View. This view arrange the tasks based on the Task Due Date. You can switch between three different ranges: Day, Week or Month.

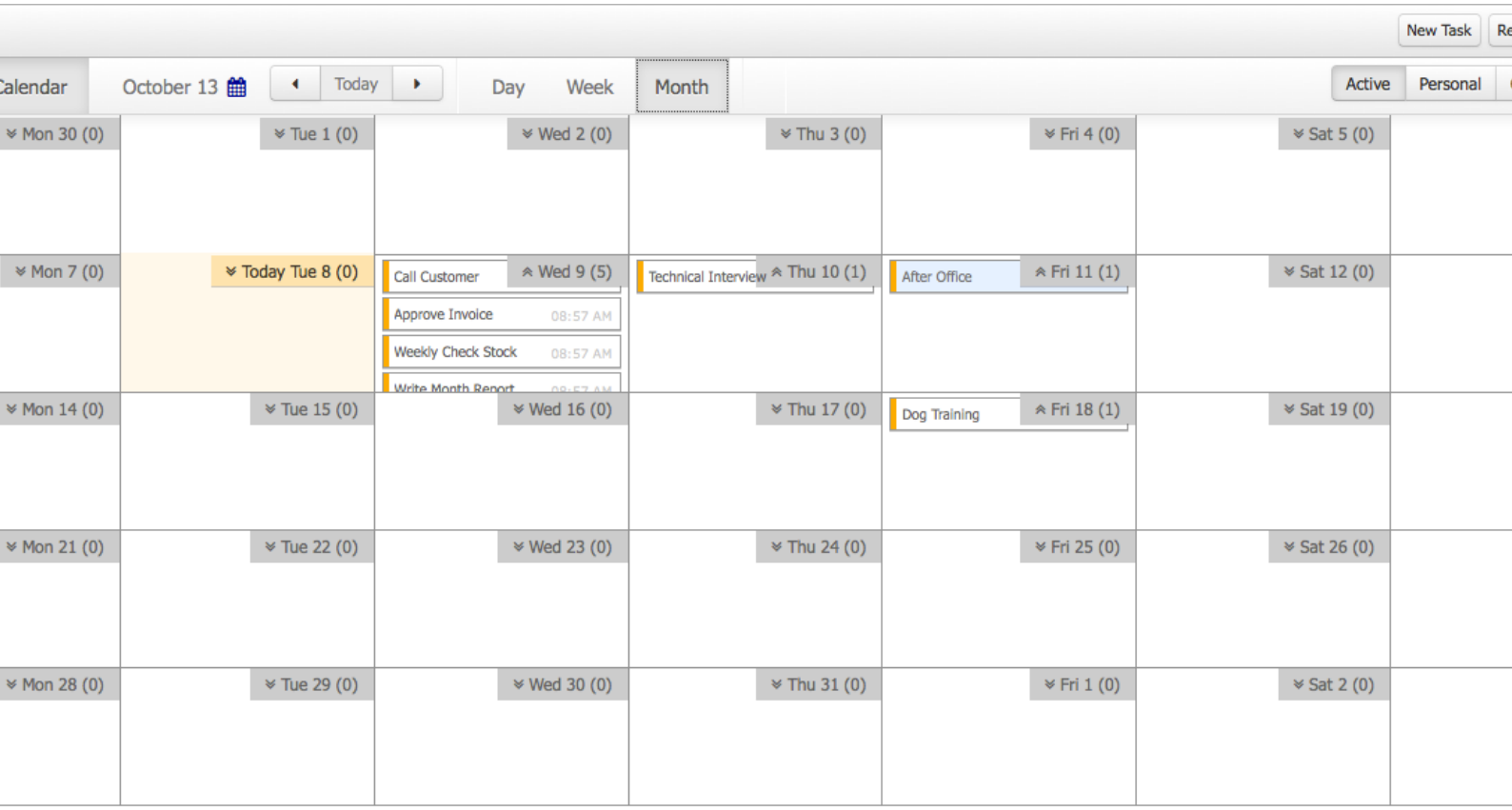
The Day view shows all the tasks that Due Today. Notice that you can change the selected date using the calendar or using the Next and Previous button. The Today button will be enabled when you are in a different day than today, and when you click it it will return the selection to the current date.



The Week view shows all the tasks pending for the current week. You can change the selected week using the calendar or the Next and Previous button. If you click on the Today button, you will be moved to the week the current week.

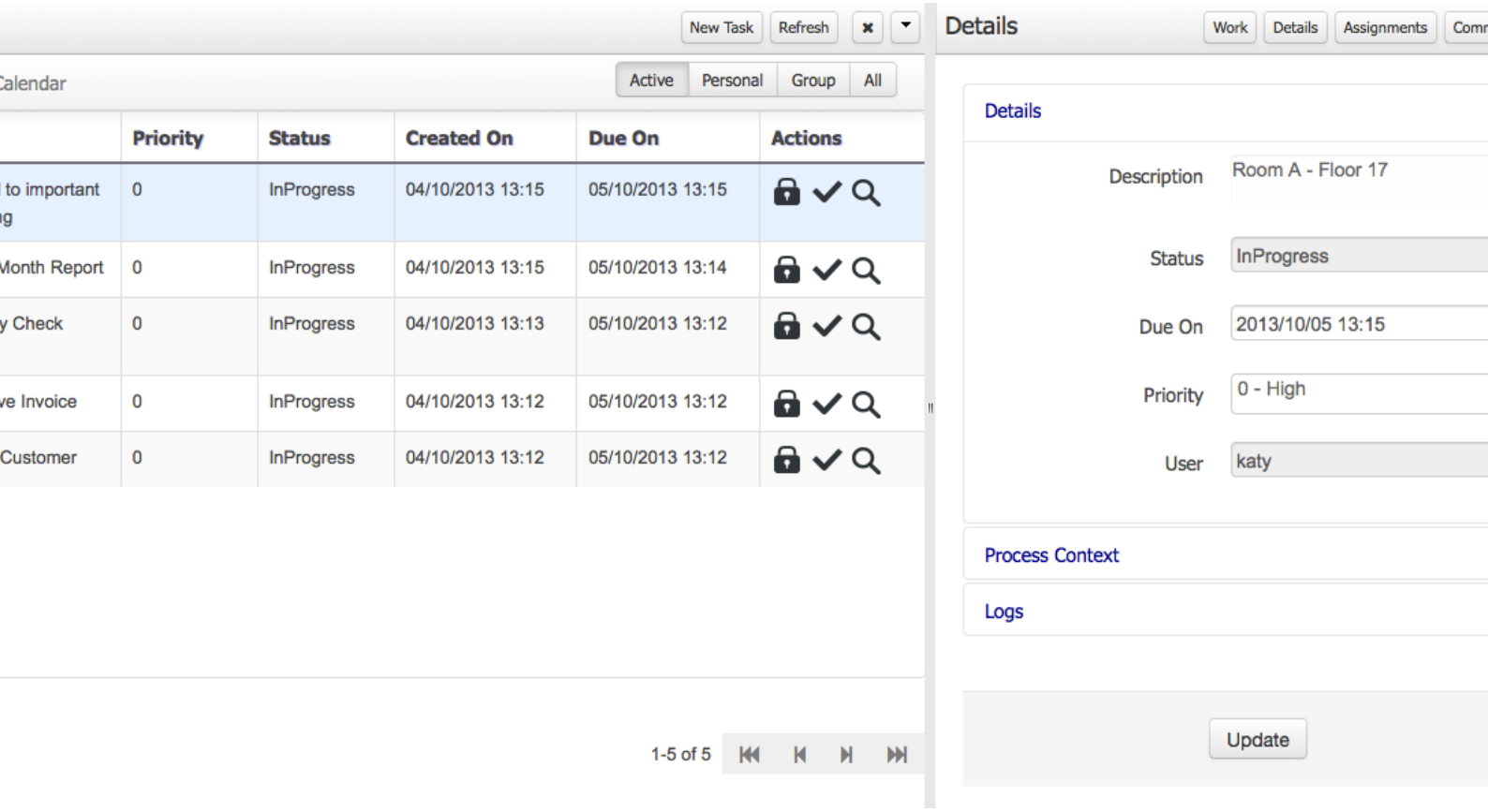


The Month view shows all the tasks that due on the selected month. You can change the month using the calendar or the Next and Previous button. If you click on the Today button the calendar will show all the tasks that due in the current month.

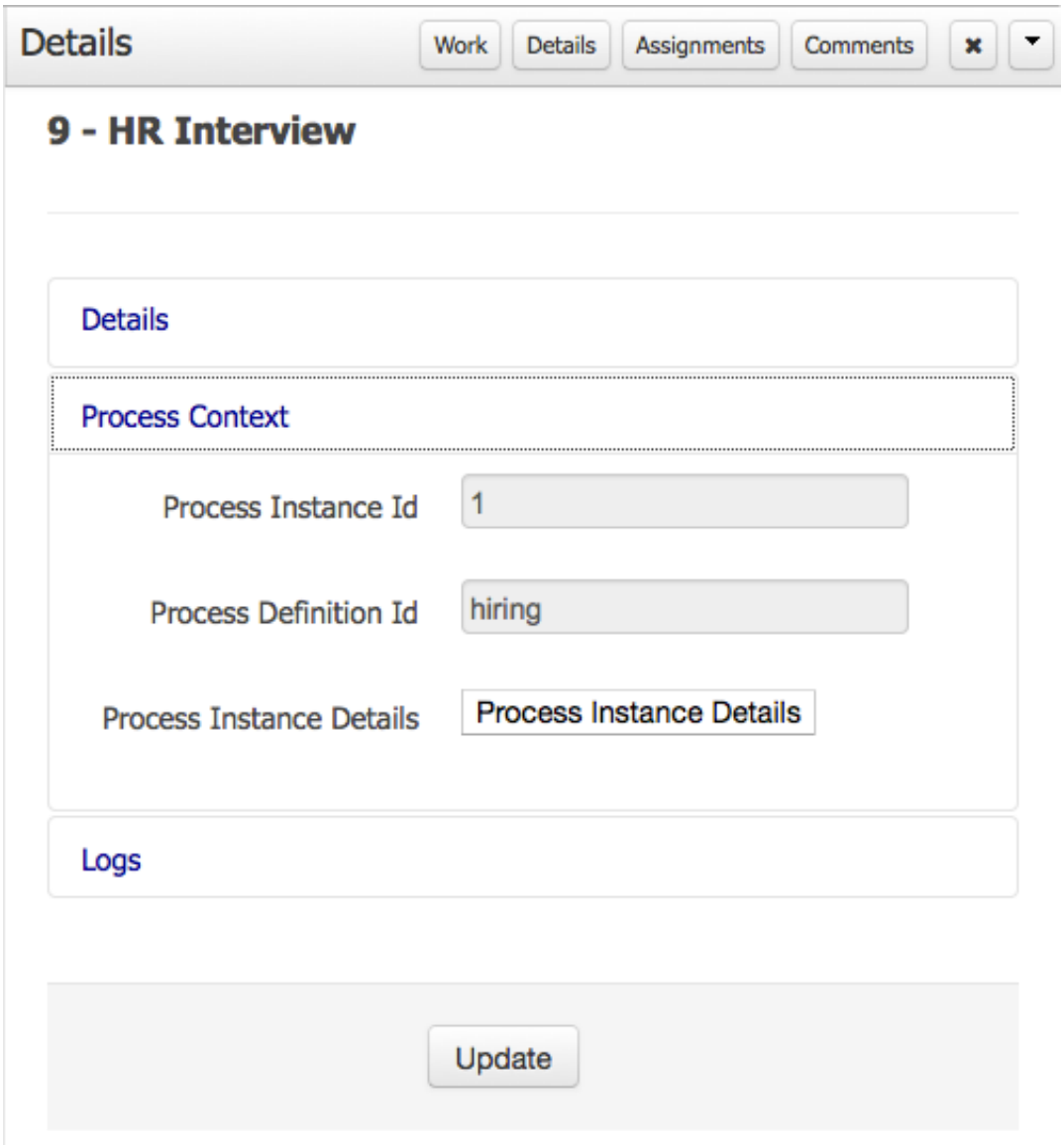


15.2.1.2. Task Details

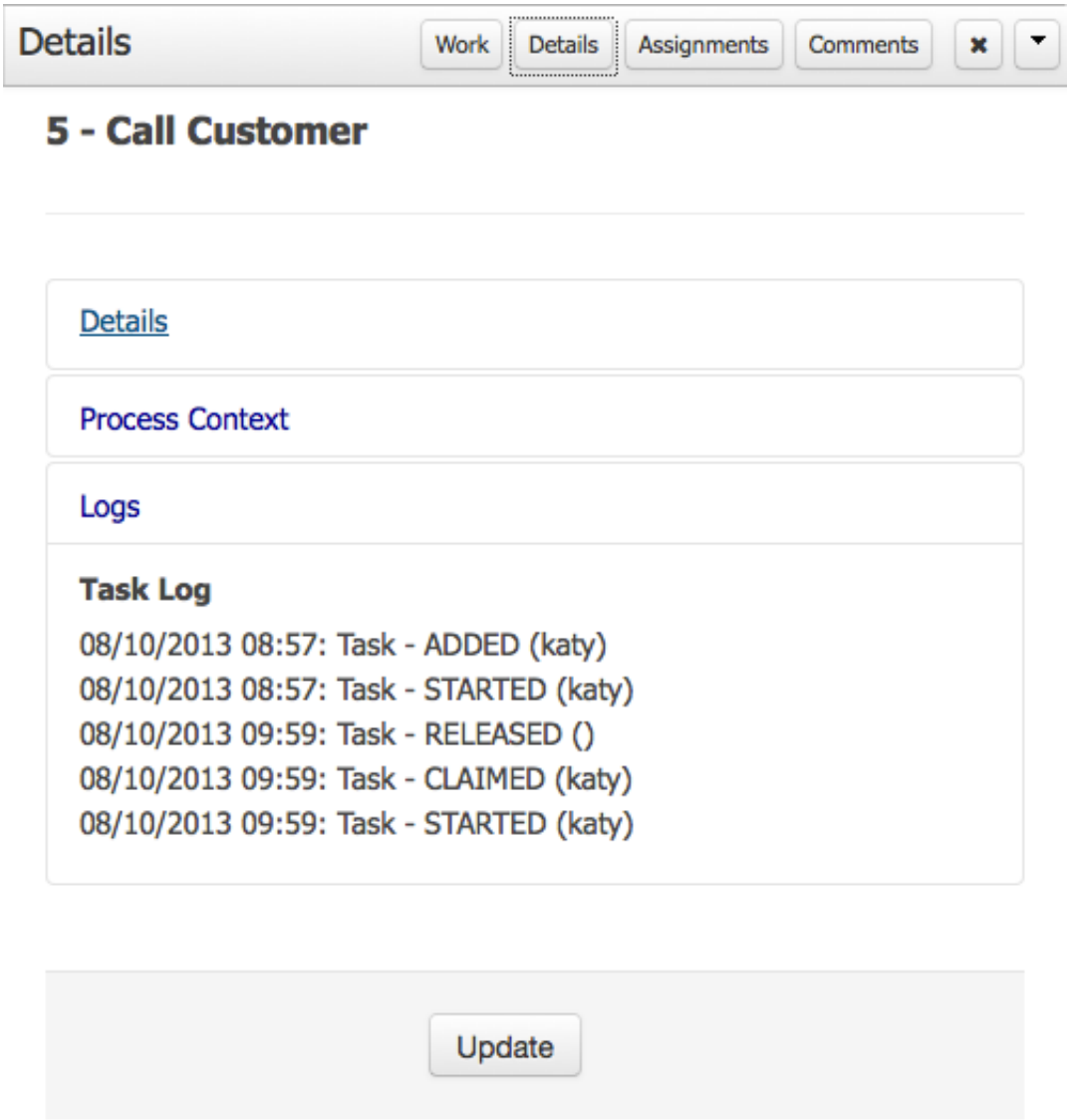
You can access to the Task Details by clicking in a task row (in both Grid and Calendar Views). The details associated with a task can be changed, like for example the Due Date, the Priority or the task description.



You can also view the Process Context for a specific task. If the task was created by a Business Process, you will have access to see the Process Instance status that has created it.

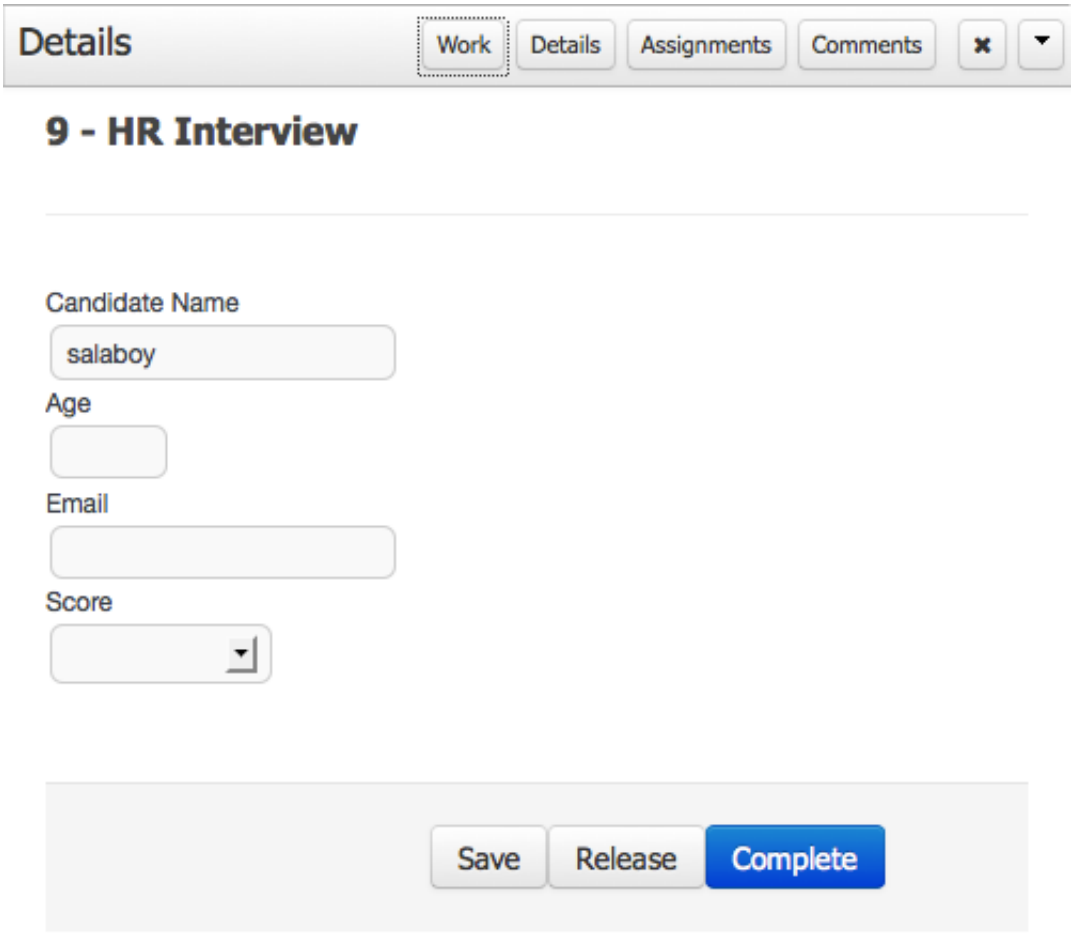


Finally you can see the Task Log, which allows you to see all the operations that has been executed on the task since its creation.



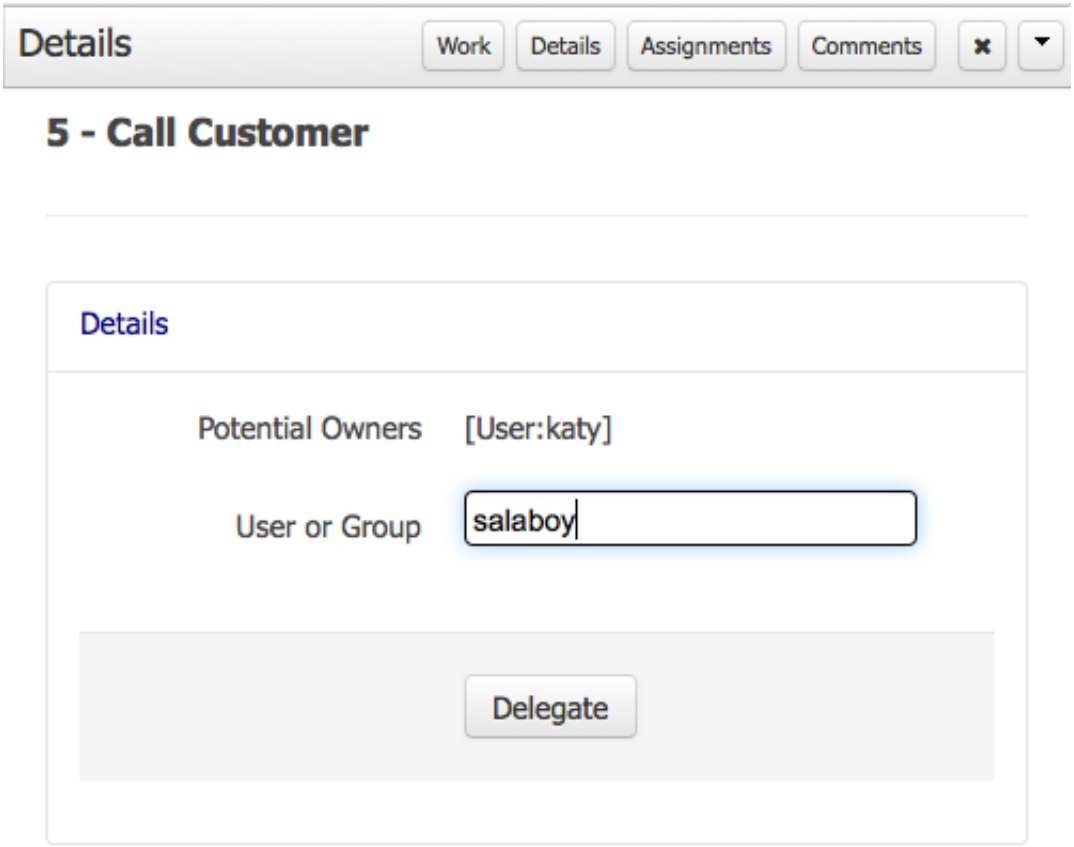
15.2.1.3. Work on a Task

Tasks can have associated a Form to store data. If tasks are part of a Business Process, usually some data needs to be collected and propagated to the business process for further usage. For that reason, tasks has to provide a way to gather and store data. Forms can be created for specific tasks using the Form Modeller. If no form is provided a dynamic form will be created based on the information that the task needs to handle. If a task is created as an ad-hoc task (not related with any process) there will be no such information to generate a form and only basic actions will be provided.



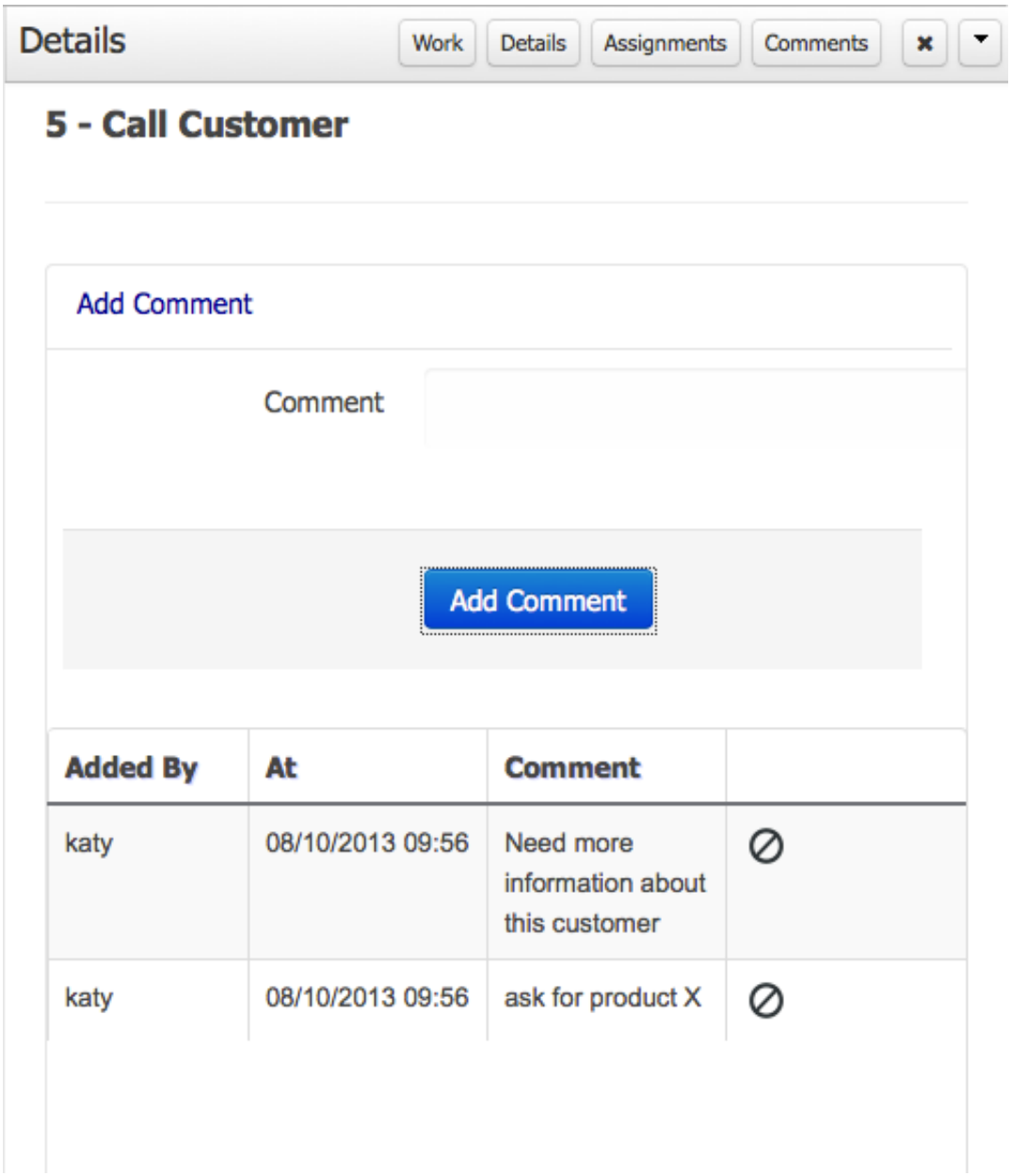
15.2.1.4. Task Assignments

You can Delegate tasks to different people when you are not able to work on them.



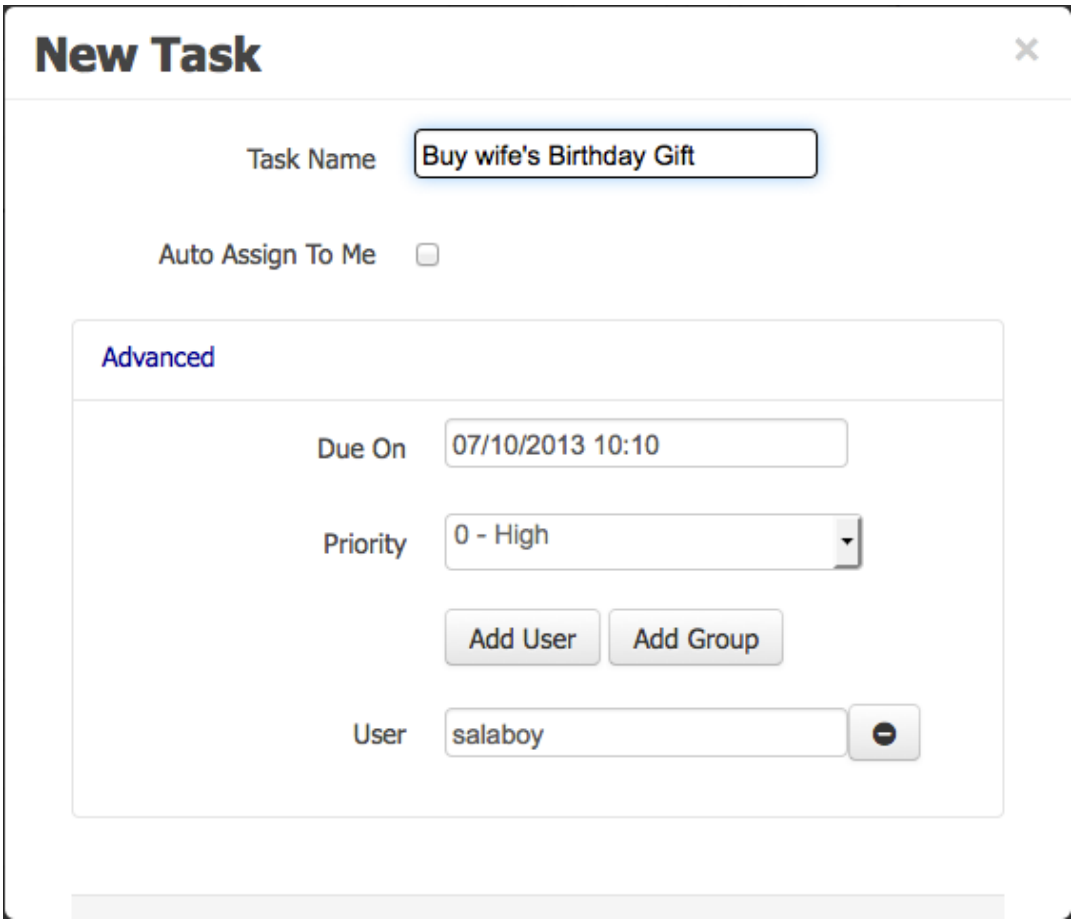
15.2.1.5. Task Comments

You can add comments to your tasks to keep track of the progress or to keep information related to the task. Notice that if you delegate the task other users can also add comments helping on the collaboration to complete the task.



15.2.2. New Task (Ad-Hoc Task)

As mentioned in the introduction a User can create their own tasks, which will not be associated with any Business Process. These tasks can be used to keep track of your personal list of TO DOs. You can also create tasks and assign them to different people in your team or group.


New Task
Task Name
Buy wife's Birthday Gift
Auto Assign To Me
Advanced
Due On
07/10/2013 10:10
Priority
0 - High
Add User
Add Group
User
salaboy

Chapter 16. Business Activity Monitoring

16.1. Overview

Imagine you are developing a BPM solution which mixes process with business data. Imagine also you need some forms to be used within processes in order to let the users enter data. Moreover, you'll likely want to have some kind of dashboards to display metrics and key performance indicators in order to quickly assess how your processes are doing. So far so good.

jBPM brings you all the ingredients you need to develop end-to-end business process solutions. The jBPM's BAM module (also known as Dashboard Builder or just Dashbuilder) allows for composing custom business dashboards mixing data coming from heterogeneous sources of information. The module is now fully integrated into KIE workbench. A new specific section for dealing with dashboards has been added and it can be accessed either from the home page or from the menu bar, as shown in the next figure.

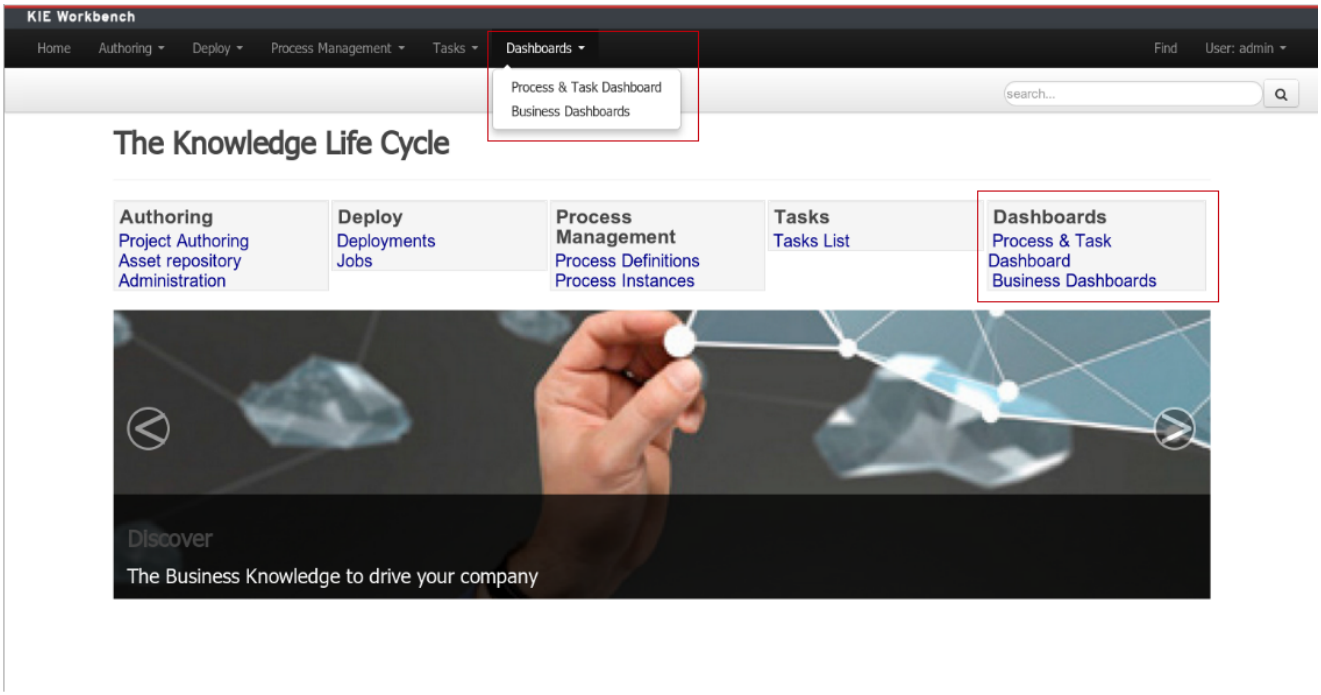


Figure 16.1. BAM menu options in the KIE Workbench home page

In the figure, Within the highlighted sections, there exists two options:

- **Business Dashboards:** This option is intended to give users access to the generic dashboard tooling either to compose new dashboards or just to consume existing ones.

- **Process & Task Dashboard:** It opens up the Process Dashboard perspective which contains several performance indicators related to the jBPM execution engine.

16.2. Business Dashboards

BPM solutions are not only made up with processes, rules or forms but also with data belonging to the customer business domain. Such data is handled in the forms, the rules and, of course, the dashboards that are part of the solution. Usually, dashboards feed with data coming from several sources of information, from business domain entities persisted into relational databases to data hold in legacy systems. In order to cope with this kind of scenarios a generic highly customizable dashboard tooling is needed.

It's obviously expected that a customer building a BPM solution want to track how its processes are performing. To do so the customer need a monitoring and reporting tool. This is the main reason why the Dashbuilder project has been included as a core module of the jBPM ecosystem. Notice also that Dashbuilder, as an independent project, is not only used by jBPM but also by many other projects like, for example, JBoss Teiid a data virtualization system that allows applications to use data from multiple, heterogeneous data stores.

Note

Please, read the Dashbuilder book in order to get detailed information about how to build custom dashboards.

An example of dashboard is the Sales Dashboard which comes built-in any installation of Dashbuilder. Two screenshots below:

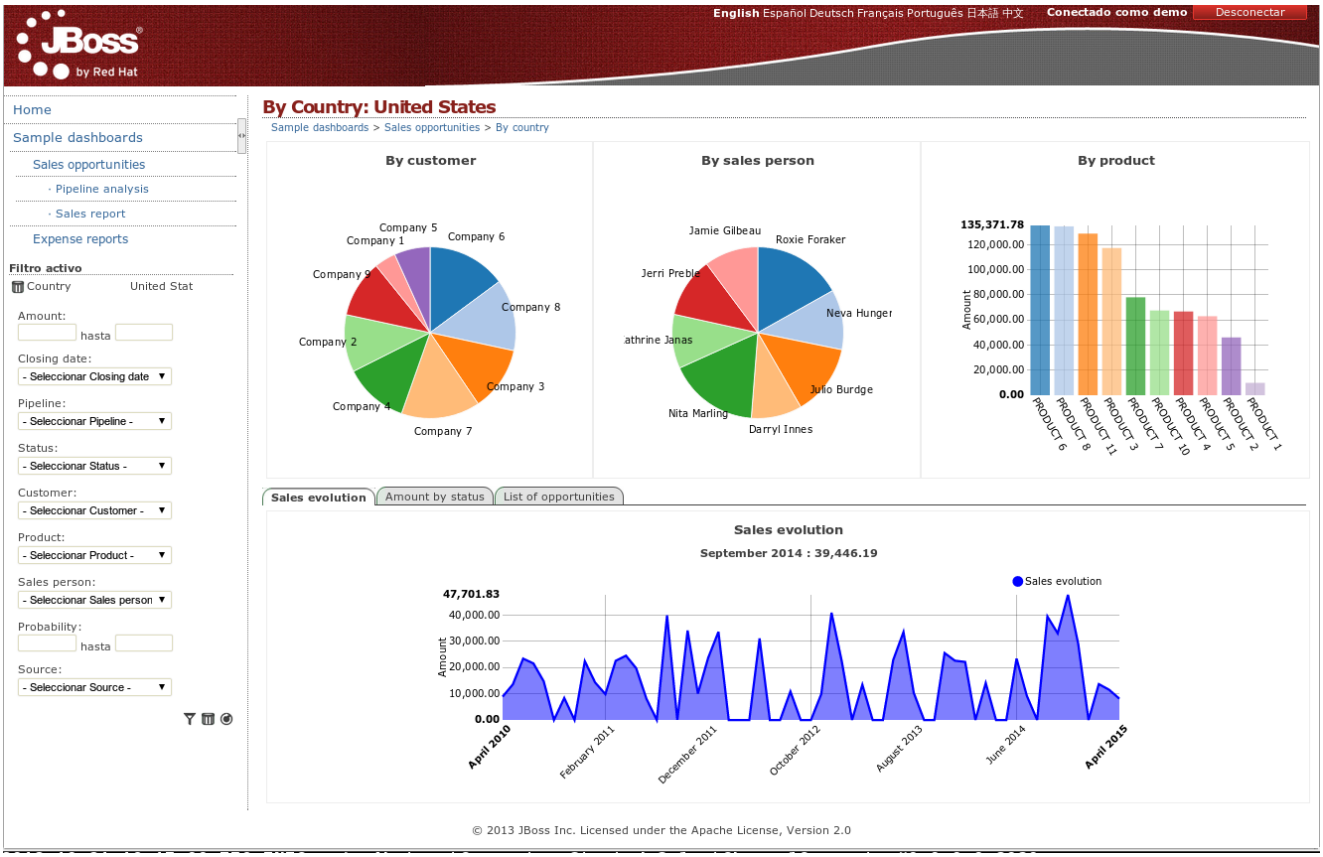


Figure 16.2. Sales opportunities by country

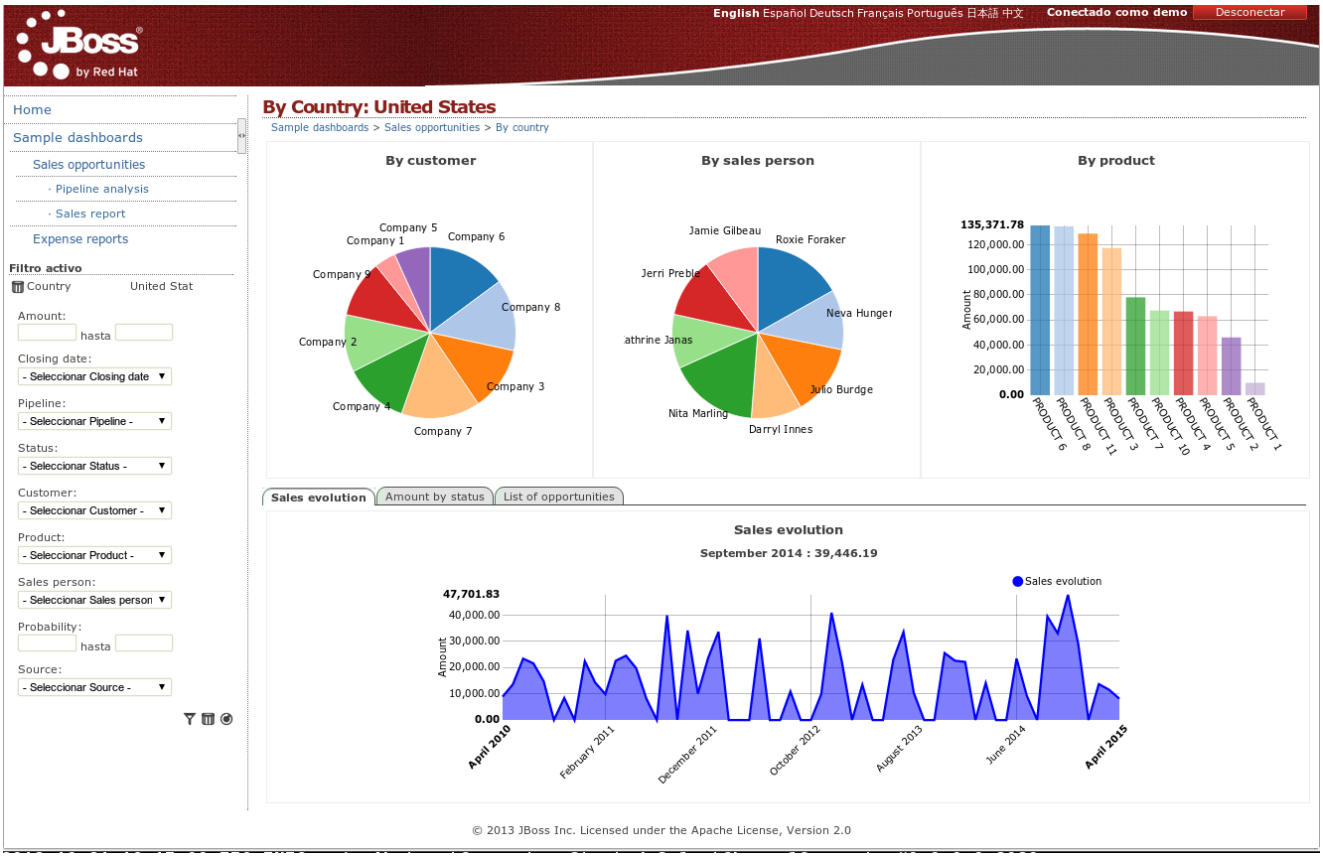


Figure 16.3. Sales opportunities report table

16.3. Process Dashboard

The jBPM Process Dashboard is an specific use case of a dashboard feed from data coming from a relational database via SQL queries. In this case, the database tables consumed are: *processinstancelog* and *bamtasksummary* both belonging to the jBPM engine.

From the data provider perspective there exists 3 data providers in charge of retrieving the data needed by all the key performance indicators of the jBPM Process Dashboard. These data provides are all defined in the Dashbuilder tooling data provider management screen.

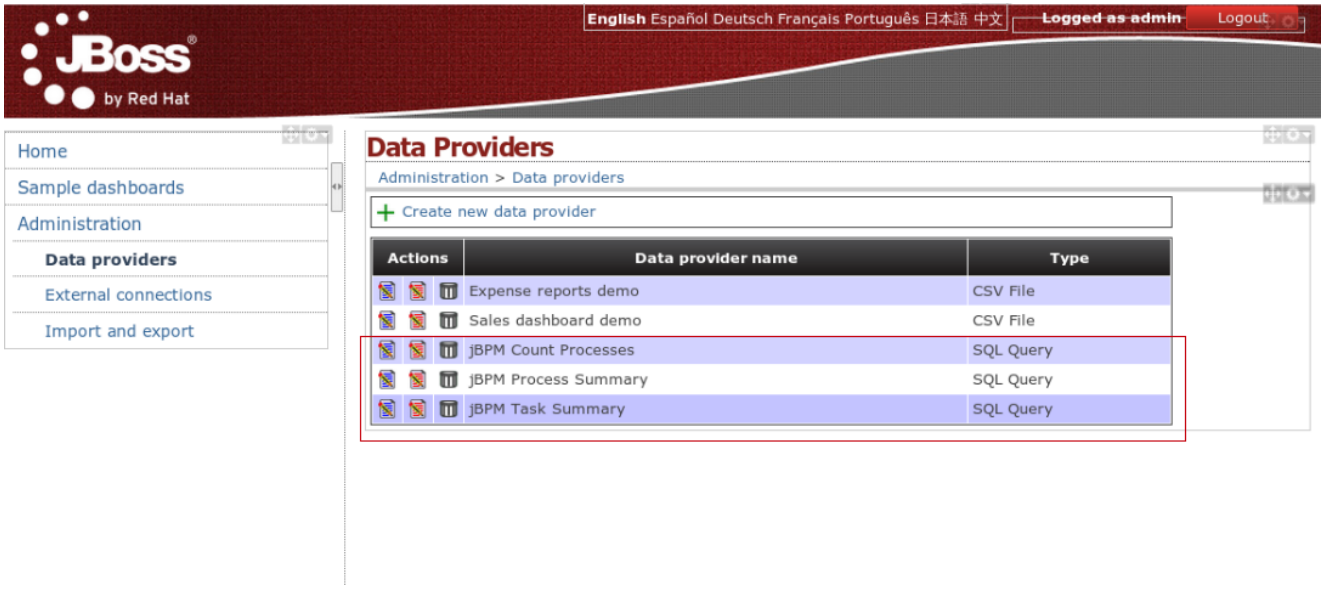


Figure 16.4. jBPM Process Dashboard data providers

- **jBPM Count Processes:** Retrieves the total number of process instances grouped by status.

```
        select total.processname, ifnull(total.instances,0) total,
        ifnull(active.instances_act,0) active,
        ifnull(completed.instances_compl,0) completed,
        ifnull(pending.instances_pend,0) pending,
        ifnull(suspended.instances_susp,0) suspended,
        ifnull(aborted.instances_abrt,0) aborted
        from
         (select pi.processinstanceid as pId, pi.processname as processname,
count(*) as instances
        from processinstancelog pi group by pi.processinstanceid,processname)
as total
        left outer join
        (select pi.processinstanceid as pId, count(*) as instances_act
        from processinstancelog pi
        where pi.status=1 group by pi.processinstanceid) as active
        on (total.pId=active.pId)
        left outer join
        (select pi.processinstanceid as pId, count(*) as instances_compl
        from processinstancelog pi
        where pi.status=2 group by pi.processinstanceid) as completed
        on (total.pId=completed.pId)
        left outer join
        (select pi.processinstanceid as pId, count(*) as instances_pend
        from processinstancelog pi
        where pi.status=0 group by pi.processinstanceid) as pending
        on (total.pId=pending.pId)
```

```
        left outer join
        (select pi.processinstanceid as pId, count(*) as instances_susp
        from processinstancelog pi
        where pi.status=4 group by pi.processinstanceid) as suspended
        on (total.pId=suspended.pId)
        left outer join
        (select pi.processinstanceid as pId, count(*) as instances_abrt
        from processinstancelog pi
        where pi.status=3 group by pi.processinstanceid) as aborted
        on (total.pId=aborted.pId)
        where {sql_condition, optional, processname, processname}
        order by processname
```

- **jBPM Process Summary:** Retrieves data from all the process instances.

```
        select processinstanceid,
        processname,
        status,
        start_date,
        end_date,
        user_identity,
        processversion,
        duration
        from processinstancelog
```

- **jBPM Task Summary:** Retrieves data from all the process tasks.

```
        select ts.taskid,
        ts.processinstanceid,
        ps.processname,
        ps.processversion,
        ts.taskname,
        ts.createddate,
        ts.enddate,
        ts.userid,
        ts.duration,
        ts.status
        from bamtasksummary ts
                                left    join    processinstancelog    ps    on
  (ts.processinstanceid=ps.processinstanceid)
```

From the end user perspective, the jBPM Process Dashboard has been designed to consume the data from the data providers defined above. It has been also designed has a panel fully integrated into the KIE Workbench environment as shown in the next figure:

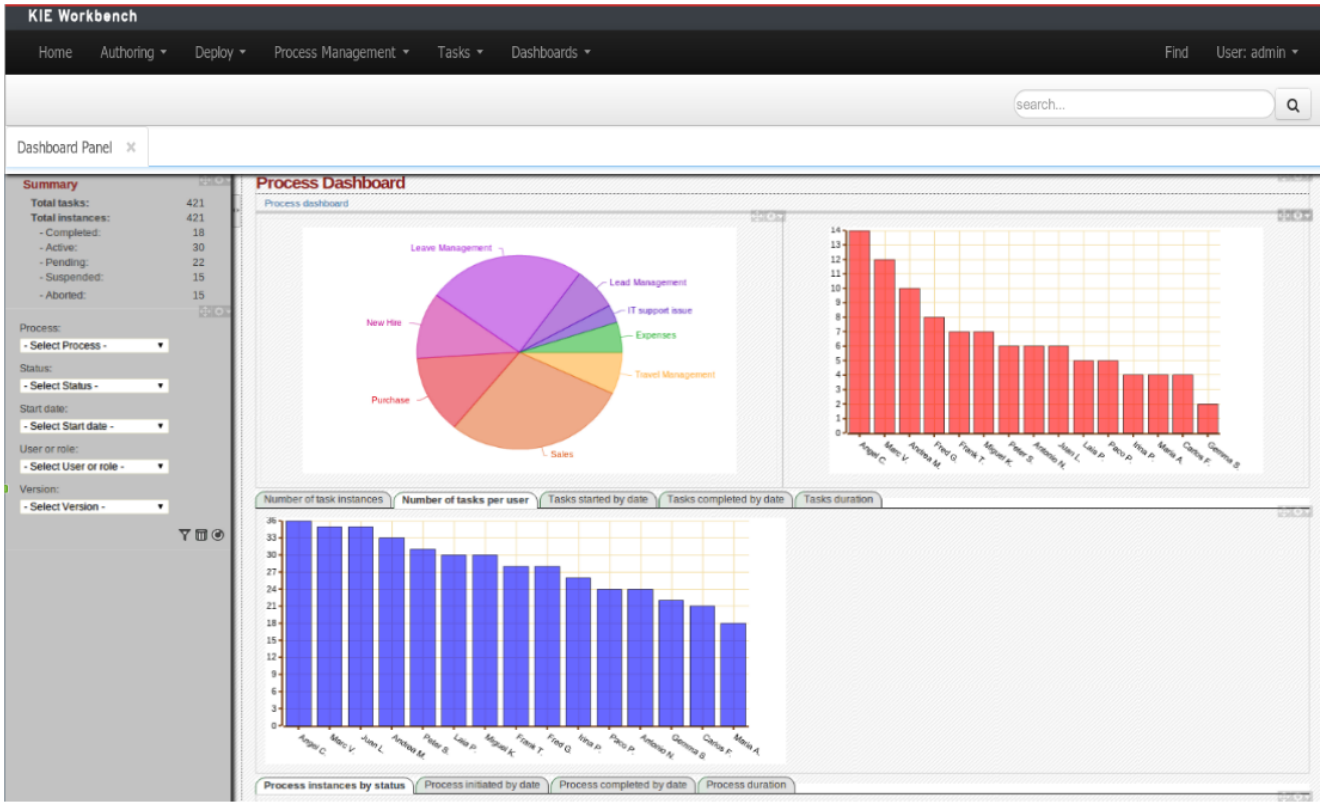


Figure 16.5. jBPM Process Dashboard populated with data coming from running process instances

The dashboard itself is composed by two views or pages:

- Global main view: containing metrics about all the processes.

Table 16.1. jBPM Process Dashboard: Global KPIs

Key Performance Indicator	Data provider
Total number of instances by process	jBPM Count Processes
Instances started by user	jBPM Process Summary
Total number of tasks by user/group	jBPM Task Summary
Number of tasks started by date	jBPM Task Summary
Number of tasks completed by date	jBPM Task Summary
Overall tasks duration (average, min. and max.)	jBPM Task Summary

Key Performance Indicator	Data provider
Number of tasks by task status	jBPM Task Summary
Number of process instances by status	jBPM Process Summary
Number of process instances started by date	jBPM Process Summary
Number of process instances completed by date	jBPM Process Summary
Overall process instances duration (average, min. and max.)	jBPM Process Summary

- Process detailed view: containing metrics about an specific process. To get into this view a process must be selected from the global view. Once a process is selected, a drill-down request is carried out by the system and the process specific view is set as the current screen.

Table 16.2. jBPM Process Dashboard: Process specific KPIs

Key Performance Indicator	Data provider
Total number of process instances by status	jBPM Count Processes
Total number of tasks by process version	jBPM Task Summary
Total number of tasks by user/group	jBPM Task Summary
Number of process tasks started by date	jBPM Task Summary
Number of process tasks completed by date	jBPM Task Summary
Overall tasks duration (average, min. and max.)	jBPM Task Summary
Number of tasks by task status	jBPM Task Summary
Number of process instances by status	jBPM Process Summary
Number of process instances started by date	jBPM Process Summary
Number of process instances completed by date	jBPM Process Summary
Overall process instances duration (average, min. and max.)	jBPM Process Summary

Note

Notice, those are generic metrics not tied to any specific business process. Nonetheless, it's worth to mention that it would be very easy for customers to modify, extend or adapt this generic dashboard for custom needs. A customer could take the jBPM Process Dashboard as the base template for building a custom dashboard which mixes data coming from the jBPM engine plus data coming from its own business domain.

Chapter 17. Remote API

The workbench contains an execution server (for executing processes and tasks), which also allows you to invoke various process and task related operations through a remote API. As a result, you can setup your process engine "as a service" and integrate this into your applications easily by doing remote requests and/or sending the necessary triggers to the execution server whenever necessary (without the need to embed or manage this as part of your application).

Both a REST and JMS based service are available (which you can use directly), and a Java remote client allows you to invoke these operations using the existing KieSession and TaskService interfaces (you also use for local interaction), making remote integration as easy as if you were interacting with a local process engine.

17.1. Remote Java API

The Remote Java API provides `KieSession`, `TaskService` and `AuditLogService` interfaces to the JMS and REST APIs.

The interface implementations provided by the Remote Java API take care of the underlying logic needed to communicate with the JMS or REST APIs. In other words, these implementations will allow you to interact with a remote workbench instance (i.e. KIE workbench or the jBPM Console) via known interfaces such as the `KieSession` or `TaskService` interface, without having to deal with the underlying transport and serialization details.

The Remote Java API provides clients, not "instances"

While the `KieSession`, `TaskSerivce` and `AuditLogService` instances provided by the Remote Java API may "look" and "feel" like local instances of the same interfaces, please make sure to remember that these instances are only wrappers around a REST or jMS client that interacts with a remote REST or JMS API.

This means that if a requested operation fails on the *server*, the Remote Java API client instance on the *client* side will throw a `RuntimeException` indicating that the REST call failed. This is different from the behaviour of a "real" (or local) instance of a `KieSession`, `TaskSerivce` and `AuditLogService` instance because the exception the local instances will throw will relate to how the operation failed. Also, while local instances require different handling (such as having to dispose of a `KieSession`), client instances provided by the Remote Java API hold no state and thus do not require any special handling.

Lastly, operations on a Remote Java API client instance that would normally throw other exceptions (such as the `TaskService.claim(taskId, userId)` operation when called by a user who is not a potential owner), will now throw a `RuntimeException` instead when the requested operation fails on the *server*.

The first step in interacting with the remote runtime is to create either the `RemoteRestRuntimeFactory` or `RemoteJmsRuntimeEngineFactory`, both of which are instances of the `RemoteRuntimeEngineFactory` interface.

The configuration for the Remote Java API is done when creating the `RemoteRuntimeEngineFactory` instance: there are a number of different constructors for both the JMS and REST impelementations that allow the configuration of such things as the base URL of the REST API, JMS queue location or timeout while waiting for responses.

Once the factory instances have been created, there are a couple of methods that can then be used to instantiate the client instance that you want to use:

Remote Java API Methods

`RemoteRuntimeEngine RemoteRuntimeEngineFactory.newRuntimeEngine()`
: This method instantiates a new `RemoteRuntimeEngine` (client) instance.

`RemoteRuntimeEngineFactory.addExtraJaxbClasses(Collection<Class<?>> extraJaxbClasses )`
: This method adds extra classes to the classpath available to the serialization mechanisms.

  When passing instances of user-defined classes in a Remote Java API call, it's important to have added the classes via this method first so that the class instances can be serialized correctly.

`KieSession RemoteRuntimeEngine.getKieSession()`
: This method instantiates a new (client) `KieSession` instance.

`TaskService RemoteRuntimeEngine.getTaskService()`
: This method instantiates a new (client) `TaskService` instance.

`AuditLogService RemoteRuntimeEngine.getAuditLogService()`
: This method instantiates a new (client) `AuditLogService` instance.

17.1.1. The REST Remote Java RuntimeEngine Factory

Oops!

The `RemoteRestRuntimeFactory` should indeed have been called the `RemoteRestRuntime`*`Engine`*`Factory`! Sometimes it's the easiest mistakes which are the hardest to catch. This will be corrected in future releases.

The `RemoteRestRuntimeFactory` has 4 constructor methods available. Besides an "everything" constructor method which provides all of the options provided below, there are also the following 3 constructors:

RemoteRestRuntimeFactory constructor method parameters

Simple constructor

Table 17.1. Simple `RemoteRestRuntimeFactory` constructor parameters

Name	Type	Description
`deploymentId`	`java.lang.String`	This is the name (id) of the deployment the `RuntimeEngine` should interact with.
`baseUrl`	`java.net.URL`	This is the URL of the deployed jbpm-console, kie-wb or BPMS instance. For example: `http://127.0.0.1:8080/jbpm-console/`
`username`	`java.lang.String`	This is the user name needed to access the JMS queues.
`password`	`java.lang.String`	This is the password needed to access the JMS queues.

Constructor with timeout parameter

Table 17.2. `RemoteRestRuntimeFactory` constructor parameters with (response message) timeout parameter

Name	Type	Description
`deploymentId`	`java.lang.String`	This is the name (id) of the deployment the `RuntimeEngine` should interact with.
`baseUrl`	`java.net.URL`	This is the URL of the deployed jbpm-console, kie-wb or BPMS instance. For example: `http://127.0.0.1:8080/jbpm-console/`
`username`	`java.lang.String`	This is the user name needed to access the REST API.

Name	Type	Description
`password`	`java.lang.String`	This is the password needed to access the REST API.
`timeoutInSeconds`	`int`	This maximum number of seconds to wait when waiting for a response from the server.

Constructor with form-based authorization parameter

Table 17.3. `RemoteRestRuntimeFactory` constructor parameters with form-based authorization parameter

Name	Type	Description
`deploymentId`	`java.lang.String`	This is the name (id) of the deployment the `RuntimeEngine` should interact with.
`baseUrl`	`java.net.URL`	This is the URL of the deployed jbpm-console, kie-wb or BPMS instance. For example: `http://127.0.0.1:8080/jbpm-console/`
`username`	`java.lang.String`	This is the user name needed to access the REST API.
`password`	`java.lang.String`	This is the password needed to access the REST API.
`useFormBasedAuth`	`boolean`	Whether or not to use form-based authentication when making a REST call. Form-based authentication will be necessary on tomcat instances.

17.1.1.1. Example usage

The following example illustrates how the Remote Java API can be used with the REST API.

```
public void startProcessAndHandleTaskViaRestRemoteJavaAPI(URL instanceUrl, String deploymentId,
```

```
  // the serverRestUrl should contain a URL similar to "http://localhost:8080/
jbpm-console/"

   // Setup the factory class with the necessarry information to communicate
 with the REST services
  RemoteRuntimeEngineFactory restSessionFactory
    = new RemoteRestRuntimeFactory(deploymentId, instanceUrl, user, password);

  // Create KieSession and TaskService instances and use them
  RemoteRuntimeEngine engine = restSessionFactory.newRuntimeEngine();
  KieSession ksession = engine.getKieSession();
  TaskService taskService = engine.getTaskService();

   // Each opertion on a KieSession, TaskService or AuditLogService (client)
 instance
   // sends a request for the operation to the server side and waits for the
 response
   // If something goes wrong on the server side, the client will throw an
 exception.
  ProcessInstance processInstance
    = ksession.startProcess("com.burns.reactor.maintenance.cycle");
  long procId = processInstance.getId();

  String taskUserId = user;
  taskService = engine.getTaskService();
 List<TaskSummary> tasks = taskService.getTasksAssignedAsPotentialOwner(user, "en-
UK");

  long taskId = -1;
  for (TaskSummary task : tasks) {
    if (task.getProcessInstanceId() == procId) {
      taskId = task.getId();
    }
  }

  if (taskId == -1) {
     throw new IllegalStateException("Unable to find task for " + user + " in
 process instance " + procId);
  }

  taskService.start(taskId, taskUserId);
}
```

17.1.2. The JMS Remote Java RuntimeEngine Factory

The Remote JMS Java RuntimeEngine works precisely the same as the REST variant, except that it has different constructors.

The `RemoteJmsRuntimeEngineFactory` constructors can be grouped into 3 types. The list below specifies the main arguments to each group type.

1. The URL of the execution server instance is given
2. The JMS remote access objects (such as the `ConnectionFactory` and `Queue`) are given.
3. A remote `InitialContext` instance (created using JNDI) from the server is given.

Configuration using the server `URL`. Configuration using only the URL of the server where the jBPM Console or Kie Workbench instance is running, is the most straightforward. However, this is only possible when the jBPM console or KIE Workbench instance is running on Wildfly or JBoss AS, or JBoss EAP.

The following table describes the parameters used when using an `InitialContext` to configure a `RemoteJmsRuntimeEngineFactory` instance:

Table 17.4. RemoteJmsRuntimeFactory constructor arguments

Name	Type	Description
`deploymentId`	`java.lang.String`	This is the name (id) of the deployment the `RuntimeEngine` should interact with.
`serverURL`	`java.net.URL`	This is the URL of the (JBoss) server instance..
`username`	`java.lang.String`	This is the user name needed to access the JMS queues and JNDI InitialContext instance.
`password`	`java.lang.String`	This is the password needed to access the JMS queues and JNDI InitialContext instance.

Configuration using an `InitialContext` instance. When configuring the `RemoteJmsRuntimeEngineFactory` with an `InitialContext` instance as a parameter, it's necessary to retrieve the (remote) `InitialContext` instance first from the remote server. The following code illustrates how to do this.

Important

The code below illustrates how this can be done with a *JBoss AS 7 or EAP 6* server instance. Similar code is used in the constructor above.

However, regardless of which application server you use, it is necessary to include in your classpath the class specified as the `INITIAL_CONTEXT_FACTORY` (see below). For JBoss AS 7 and EAP 6, the artifact (jar) containing this class is the `org.jboss:jboss-remote-naming` artifact (jar), version `1.0.5.Final` or higher. Depending on the version of AS 7 or EAP 6 that you use, this version may vary.

If you are using a *different* application server, please see your specific application server documentation for the parameters and artifacts necessary to create an `InitialContextFactory` instance or otherwise get a remote `InitialContext` instance (via JNDI) from the application server instance.

```
public void startProcessAndTaskViaJmsRemoteJavaAPI(String serverHostName, String deploymentId,
  // Setup remote JMS runtime engine factory
  InitialContext remoteInitialContext
    = getRemoteInitialContext(serverHostName, user, password);
  int maxTimeoutSecs = 5;
  RemoteJmsRuntimeEngineFactory remoteJmsFactory
    = new RemoteJmsRuntimeEngineFactory(deploymentId, remoteInitialContext, user, password, max

  // Interface with JMS api
  RuntimeEngine engine = remoteJmsFactory.newRuntimeEngine();
  KieSession ksession = engine.getKieSession();
  ProcessInstance processInstance = ksession.startProcess("com.burns.reactor.maintenance.cycle"
  long procId = processInstance.getId();
  TaskService taskService = engine.getTaskService();
  List<Long> tasks = taskService.getTasksByProcessInstanceId(procId);
  taskService.start(tasks.get(0), user);
}

private static InitialContext getRemoteInitialContext(String jbossServerHostName, String user,
  // Configure the (JBoss AS 7/EAP 6) InitialContextFactory
  Properties initialProps = new Properties();
  initialProps.setProperty(InitialContext.INITIAL_CONTEXT_FACTORY, "org.jboss.naming.remote.cli
  initialProps.setProperty(InitialContext.PROVIDER_URL, "remote://"+ jbossServerHostName + ":44
  initialProps.setProperty(InitialContext.SECURITY_PRINCIPAL, user);
  initialProps.setProperty(InitialContext.SECURITY_CREDENTIALS, password);

  for (Object keyObj : initialProps.keySet()) {
    String key = (String) keyObj;
    System.setProperty(key, (String) initialProps.get(key));
  }

  // Create the remote InitialContext instance
  try {
    return new InitialContext(initialProps);
  } catch (NamingException e) {
```

```
                    throw     new     RuntimeException("Unable     to     create
 " + InitialContext.class.getSimpleName(), e);
  }
}
```

The following table describes the parameters used when using an `InitialContext` to configure a `RemoteJmsRuntimeEngineFactory` instance:

Table 17.5. RemoteJmsRuntimeFactory constructor arguments

Name	Type	Description
`deploymentId`	`java.lang.String`	This is the name (id) of the deployment the `RuntimeEngine` should interact with.
`initialContext`	`javax.naming.InitialContext`	This is a remote `InitialContext` instance from which the (`javax.jms`) `ConnectionFactory` and `Queue` instances can be retrieved.
`username`	`java.lang.String`	This is the user name needed to access the JMS queues.
`password`	`java.lang.String`	This is the password needed to access the JMS queues.
`timeoutSeconds`	`int`	This maximum number of seconds to wait when waiting for a response from the server.

Configuration using `ConnectionFactory` and `Queue` instance parameters. Some users may have direct access to a `ConnectionFactory` and the `Queue` instances needed to interact with the JMS API. In this case, they can use the `RemoteJmsRuntimeEngineFactory` constructor that uses the following arguments:

Table 17.6. RemoteJmsRuntimeEngineFactory constructor arguments

Name	Type	Description
`deploymentId`	`java.lang.String`	This is the name (id) of the deployment the `RuntimeEngine` should interact with.
`connectionFactory`	`javax.jms.ConnectionFactory`	This is a `ConnectionFactory` instance used to connect

Name	Type	Description
		to the `ksessionQueue` or `taskQueue`.
`ksessionQueue`	`javax.jms.Queue`	This is an instance of the `Queue` for requests relating to the process instance.
`taskQueue`	`javax.jms.Queue`	This is an instance of the `Queue` for requests relating to task service usage.
`responseQueue`	`javax.jms.Queue`	This is an instance of the `Queue` used to receive responses.
`username`	`java.lang.String`	This is the user name needed to access the JMS queues (in your application server configuration).
`password`	`java.lang.String`	This is the password needed to access the JMS queues (in your application server configuration).
`timeoutSeconds`	`int`	This maximum number of seconds to wait when waiting for a response from the server.

17.1.3. Supported methods

As mentioned above, the Remote Java API provides client-like instances of the `RuntimeEngine`, `KieSession`, `TaskService` and `AuditLogService` interfaces.

This means that while many of the methods in those interfaces are available, some are not. The following tables lists the methods which are available. Methods not listed in the below, will throw an `UnsupportedOperationException` explaining that the called method is not available.

Table 17.7. Available process-related `KieSession` methods

Returns	Method signature	Description
`void`	`abortProcessInstance(long processInstanceId)`	Abort the process instance
`ProcessInstance`	`getProcessInstance(long processInstanceId)`	Return the process instance

Returns	Method signature	Description
`ProcessInstance`	`getProcessInstance(long processInstanceId, boolean readOnly)`	Return the process instance
`List<ProcessInstance>`	`getProcessInstances()`	Return all (active) process instances
`void`	`signalEvent(String signal, Object event)`	Signal all (active) process instances
`void`	`signalEvent(String signal, Object event, long processInstanceId)`	Signal the process instance
`ProcessInstance`	`startProcess(String processId, CorrelationKey correlationKey, Map<String, Object> parameters)`	Start a new process and return the process instance (if the process instance has not immediately completed)
`ProcessInstance`	`startProcess(String processId, Map<String, Object> parameters);`	Start a new process and return the process instance (if the process instance has not immediately completed)

Table 17.8. Available rules-related `KieSession` methods

Returns	Method signature	Description
`Long`	`getFactCount()`	Return the total fact count
`Object`	`getGlobal(String identifier)`	Return a global fact
`Integer`	`getId()`	Return the id of the `KieSession`
`void`	`setGlobal(String identifier, Object value)`	Set a global fact
`void`	`fireAllRules()`	Fire all rules

Table 17.9. Available `WorkItemManager` methods

Returns	Method signature	Description
`void`	`abortWorkItem(long workItemId)`	Abort the work item
`void`	`completeWorkItem(long workItemId, Map<String, Object> results)`	Complete the work item
`WorkItem`	`getWorkItem(long workItemId)`	Return the work item

Table 17.10. Available task operation `TaskService` methods

Returns	Method signature	Description
`Long`	`addTask(Task task, Map<String, Object> params)`	Add a new task
`void`	`activate(long taskId, String userId)`	Activate a task
`void`	`claim(long taskId, String userId)`	Claim a task
`void`	`claim(long taskId, String userId, List<String> groupIds)`	Claim a task
`void`	`claimNextAvailable(String userId, String language)`	Claim the next available task for a user
`void`	`claimNextAvailable(String userId, List<String> groupIds, String language)`	Claim the next available task for a user
`void`	`complete(long taskId, String userId, Map<String, Object> data)`	Complete a task
`void`	`delegate(long taskId, String userId, String targetUserId)`	Delegate a task
`void`	`exit(long taskId, String userId)`	Exit a task

Returns	Method signature	Description
void	`fail(long taskId, String userId, Map<String, Object> faultData)`	Fail a task
void	`forward(long taskId, String userId, String targetEntityId)`	Forward a task
void	`nominate(long taskId, String userId, List<OrganizationalEntity> potentialOwners)`	Nominate a task
void	`release(long taskId, String userId)`	Release a task
void	`remove(long taskId, String userId)`	Remove a task
void	`resume(long taskId, String userId)`	Resume a task
void	`skip(long taskId, String userId)`	Skip a task
void	`start(long taskId, String userId)`	Start a task
void	`stop(long taskId, String userId)`	Stop a task
void	`suspend(long taskId, String userId)`	Suspend a task

Table 17.11. Available task retrieval and query `TaskService` methods

Returns	Method signature
Task	`getTaskByWorkItemId(long workItemId)`
Task	`getTaskById(long taskId)`
List<TaskSummary>	`getTasksAssignedAsBusinessAdministrator(String userId, String language)`
List<TaskSummary>	`getTasksAssignedAsPotentialOwner(String userId, String language)`
List<TaskSummary>	`getTasksAssignedAsPotentialOwnerByStatus(String userId,`

Returns	Method signature
	List<Status>gt; status, String language)
List<TaskSummary>	getTasksOwned(String userId, String language)
List<TaskSummary>	getTasksOwnedByStatus(String userId, List<Status> status, String language)
List<TaskSummary>	getTasksByStatusByProcessInstanceId(long processInstanceId, List<Status> status, String language)
List<TaskSummary>	getTasksByProcessInstanceId(long processInstanceId)
List<TaskSummary>	getTasksByVariousFields(List<Long> workItemIds, List<Long> taskIds, List<Long> procInstIds, List<String> busAdmins, List<String> potOwners, List<String> taskOwners, List<Status> status, boolean union)
List<TaskSummary>	getTasksByVariousFields(Map <String, List<?>> parameters, boolean union)
Content	getContentById(long contentId)
Attachment	getAttachmentById(long attachId)

Table 17.12. Available `AuditLogService` methods

Returns	Method signature
List<ProcessInstanceLog>	findProcessInstances()
List<ProcessInstanceLog>	findProcessInstances(String processId)
List<ProcessInstanceLog>	findActiveProcessInstances(String processId)
ProcessInstanceLog	findProcessInstance(long processInstanceId)
List<ProcessInstanceLog>	findSubProcessInstances(long processInstanceId)
List<NodeInstanceLog>	findNodeInstances(long processInstanceId)
List<NodeInstanceLog>	findNodeInstances(long processInstanceId, String nodeId)
List<VariableInstanceLog>	findVariableInstances(long processInstanceId)
List<VariableInstance	findVariableInstances(long processInstanceId, String variableId)
List<VariableInstance	findVariableInstancesByName(String variableId,

Returns	Method signature
	`boolean onlyActiveProcesses)`
List<VariableInstance	`findVariableInstancesByNameAndValue(String variableId, String value, boolean onlyActiveProcesses)`
void	`clear()`

17.2. REST

REST API calls to the execution server allow you to remotely manage processes and tasks and retrieve various dynamic information from the execution server. The majority of the calls are synchronous, which means that the call will only finish once the requested operation has completed on the server. The exceptions to this are the deployment `POST` calls, which will return the status of the request while the actual operation requested will asynchronously execute.

When using Java code to interface with the REST API, the classes used in POST operations or otherwise returned by various operations can be found in the `(org.kie.remote:)kie-services-client` JAR.

17.2.1. Runtime calls

This section lists REST calls that interface with

The *deploymentId* component of the REST calls below must conform to the following regular expression:

- `[\w\.-]+(:[\w\.-]+){2,2}(:[\w\.-]*){0,2}`

For information, see the *Deployment calls* section.

17.2.1.1. Process calls

[POST] `/runtime/{deploymentId}/process/{processDefId}/start`

- Starts a process.
- Returns a `JaxbProcessInstanceResponse` instance, that contains basic information about the process instance.
- The *prodessDefId*component of the URL must conform to the following regex: `[_a-zA-Z0-9-:\.]+`
- This operation takes *map query parameters* (see above), which will be used as parameters for the process instance.

[GET] `/runtime/{deploymentId}/process/instance/{procInstId}`

- Does a (read only) retrieval of the process instance. This operation will fail (code 400) if the process instance has been completed.
- Returns a `JaxbProcessInstanceResponse` instance.
- The *procInstId* component of the URL must conform to the following regex: `[0-9]+`

[POST] `/runtime/{deploymentId}/process/instance/{procInstId+}/abort`

- Aborts the process instance.
- Returns a `JaxbGenericResponse` indicating whether or not the operation has succeeded.
- The *procInstId* component of the URL must conform to the following regex: `[0-9]+`

[POST] `/runtime/{deploymentId}/process/instance/{procInstId}/signal`

- Signals the process instance.
- Returns a `JaxbGenericResponse` indicating whether or not the operation has succeeded.
- The *procInstId* component of the URL must conform to the following regex: `[0-9]+`
- This operation takes a `signal` and a `event` query parameter.
  - The `signal` parameter value is used as the name of the signal. This parameter is required.
  - The `event` parameter value is used as the value of the event. This value may use the *number query parameter* syntax described above.

[GET] `/runtime/{deploymentId}/process/instance/{procInstId}/variables`

- Gets the list of process variables in a process instance.
- Returns a `JaxbVariablesResponse`
- The *procInstId* component of the URL must conform to the following regex: `[0-9]+`

[POST] `/runtime/{deploymentId}/signal`

- Signals the `KieSession`
- Returns a `JaxbGenericResponse` indicating whether or not the operation has succeeded.
- The *procInstId* component of the URL must conform to the following regex: `[0-9]+`
- This operation takes a `signal` and a `event` query parameter.
  - The `signal` parameter value is used as the name of the signal. This parameter is required.

- The `event` parameter value is used as the value of the event. This value may use the *number query parameter* syntax described above.

[GET] `/runtime/{deploymentId}/workitem/{workItemId}`

- Gets a `WorkItem` instance
- Returns a `JaxbWorkItem` instance
- The *workItemId* component of the URL must conform to the following regex: `[0-9]+`

[POST] `/runtime/{deploymentId}/workitem/{workItemId}/complete`

- Completes a `WorkItem`
- Returns a `JaxbGenericResponse` indicating whether or not the operation has succeeded
- The *workItemId* component of the URL must conform to the following regex: `[0-9]+`
- This operation takes *map query parameters*, which are used as input to signify the results for completion of the work item.

[POST] `/runtime/{deploymentId}/workitem/{workItemId: [0-9-]+}/abort`

- Aborts a `WorkItem`
- Returns a `JaxbGenericResponse` indicating whether or not the operation has succeeded
- The *workItemId* component of the URL must conform to the following regex: `[0-9]+`

17.2.1.2. Process calls "with variables"

[POST] `/runtime/{deploymentId}/withvars/process/{processDefId}/start`

- Starts a process and retrieves the list of variables associated with the process instance
- Returns a `JaxbProcessInstanceWithVariablesResponse` that contains:
  - Information about the process instance (with the same fields and behaviour as the `JaxbProcessInstanceResponse`
  - A key-value list of the variables available in the process instance.
- The *processDefId* component of the URL must conform to the following regex: `[_a-zA-Z0-9-:\.]+`

[POST] `/runtime/{deploymentId}/withvars/process/instance/{procInstId}`

- Starts a process and retrieves the list of variables associated with the process instance
- Returns a `JaxbProcessInstanceWithVariablesResponse` (see the above REST call)

- The *processInstId* component of the URL must conform to the following regex: `[0-9]+`

[POST] `/runtime/{deploymentId}/withvars/process/instance/{procInstId}/signal`

- Signals a process instance and retrieves the list of variables associated it
- Returns a `JaxbProcessInstanceWithVariablesResponse` (see above)
- The *processInstId* component of the URL must conform to the following regex: `[0-9]+`
- This operation takes a `signal` and a `event` query parameter.
  - The `signal` parameter value is used as the name of the signal. This parameter is required.
  - The `event` parameter value is used as the value of the event. This value may use the *number query parameter* syntax described above.

17.2.2. History calls

Note

Between the 6.0.0.Final and 6.0.1.Final releases, the History REST calls were udpated and fixed in order to make them both more robust and accessible. While the REST calls that were provided with 6.0.0.Final are still available in 6.0.1.Final, they will be removed in a future release.

17.2.2.1. Calls available as of 6.0.1.Final

[POST] `/history/clear`

- Cleans (deletes) all history logs

[GET] `/history/instances`

- Gets a list of `ProcessInstanceLog` instances
- Returns a `JaxbHistoryLogList` instance that contains a list of `JaxbProcessInstanceLog` instances
- This operation responds to pagination parameters

[GET] `/history/instance/{procInstId}`

- Gets the `ProcessInstanceLog` instance associated with the specified process instance
- Returns a `JaxbHistoryLogList` instance that contains a `JaxbProcessInstanceLog` instance

- The *processInstId* component of the URL must conform to the following regex: `[0-9]+`
- This operation responds to pagination parameters

[GET] `/history/instance/{procInstId}/child`

- Gets a list of `ProcessInstanceLog` instances associated with any child/sub-processes associated with the specified process instance
- Returns a `JaxbHistoryLogList` instance that contains a list of `JaxbProcessInstanceLog` instances
- The *processInstId* component of the URL must conform to the following regex: `[0-9]+`
- This operation responds to pagination parameters

[GET] `/history/instance/{procInstId}/node`

- Gets a list of `NodeInstanceLog` instances associated with the specified process instance
- Returns a `JaxbHistoryLogList` instance that contains a list of `JaxbNodeInstanceLog` instances
- The *processInstId* component of the URL must conform to the following regex: `[0-9]+`
- This operation responds to pagination parameters

[GET]`/history/instance/{procInstId}/variable`

- Gets a list of `VariableInstanceLog` instances associated with the specified process instance
- Returns a `JaxbHistoryLogList` instance that contains a list of `JaxbVariableInstanceLog` instances
- The *processInstId* component of the URL must conform to the following regex: `[0-9]+`
- This operation responds to pagination parameters

[GET] `/history/instance/{procInstId}/node/{nodeId}`

- Gets a list of `NodeInstanceLog` instances associated with the specified process instance that have the given (node) id
- Returns a `JaxbHistoryLogList` instance that contains a list of `JaxbNodeInstanceLog` instances
- The *processInstId* component of the URL must conform to the following regex: `[0-9]+`
- The *nodeId* component of the URL must conform to the following regex: `[a-zA-Z0-9-:\.]+`
- This operation responds to pagination parameters

[GET] `/history/instance/{procInstId}/variable/{varId}`

- Gets a list of `VariableInstanceLog` instances associated with the specified process instance that have the given (variable) id
- Returns a `JaxbHistoryLogList` instance that contains a list of `JaxbVariableInstanceLog` instances
- The *processInstId* component of the URL must conform to the following regex: `[0-9]+`
- The *varId* component of the URL must conform to the following regex: `[a-zA-Z0-9-:\.]+`
- This operation responds to pagination parameters

[GET] `/history/process/{processDefId}`

- Gets a list of `ProcessInstanceLog` instances associated with the specified process definition
- Returns a `JaxbHistoryLogList` instance that contains a list of `JaxbProcessInstanceLog` instances
- The *processDefId* component of the URL must conform to the following regex: `[_a-zA-Z0-9-:\.]+`
- This operation responds to pagination parameters

17.2.2.1.1. History calls that search by variable

[GET] `/history/variable/{varId}`

- Gets a list of `VariableInstanceLog`instances associated with the specified variable id
- Returns a `JaxbHistoryLogList` instance that contains a list of `JaxbVariableInstanceLog` instances
- The *varId* component of the URL must conform to the following regex: `[a-zA-Z0-9-:\.]+`
- This operation responds to pagination parameters

[GET] `/history/variable/{varId}/value/{value}`

- Gets a list of `VariableInstanceLog` instances associated with the specified variable id that contain the value specified
- Returns a `JaxbHistoryLogList` instance that contains a list of `JaxbVariableInstanceLog` instances
- Both the *varId* and *value* components of the URL must conform to the following regex: `[a-zA-Z0-9-:\.]+`

- This operation responds to pagination parameters

[GET] `/history/variable/{varId}/instances`

- Gets a list of `ProcessInstance` instances that contain the variable specified by the given variable id.
- Returns a `JaxbProcessInstanceListResponse` instance that contains a list of `JaxbProcessInstanceResponse` instances
- The *varId* component of the URL must conform to the following regex: `[a-zA-Z0-9-:\.]+`
- This operation responds to pagination parameters

[GET] `/history/variable/{varId}/value/{value}/instances`

- Gets a list of `ProcessInstance` instances that contain the variable specified by the given variable id which contains the (variable) value specified
- Returns a `JaxbProcessInstanceListResponse` instance that contains a list of `JaxbProcessInstanceResponse` instances
- Both the *varId* and *value* components of the URL must conform to the following regex: `[a-zA-Z0-9-:\.]+`
- This operation responds to pagination parameters

17.2.2.2. Deprecated history calls available in 6.0.0.Final

Rest calls that contain "`rest/runtime/{deploymentId}/history`" have been deprecated: the same functionality provided by these calls can be found in the history REST calls described in the previous sections.

Important

If you're using the 6.0.0.Final release, the following applies to the History REST calls:

The history calls in 6.0.0.Final are dependent on a deployment being available to call them. This is because the History REST calls in 6.0.0.Final needed the persistence framework of a deployment in order to be executed. This means that history REST calls listed below may sometimes fail when used with a deployment unit that uses a `PER_REQUEST` or `PER_PROCESS_INSTANCE` strategy (i.e. when the deployment is no longer available).

[POST] `/runtime/{deploymentId}/history/clear`

- Cleans (deletes) all history logs

[GET] `/runtime/{deploymentId}/history/instances`

- Gets a list of `ProcessInstanceLog` instances
- Returns a `JaxbHistoryLogList` instance that contains a list of `JaxbProcessInstanceLog` instances
- This operation responds to pagination parameters

[GET] `/runtime/{deploymentId}/history/instance/{procInstId}`

- Gets the `ProcessInstanceLog` instance associated with the specified process instance
- Returns a `JaxbHistoryLogList` instance that contains a `JaxbProcessInstanceLog` instance
- The *processInstId* component of the URL must conform to the following regex: `[0-9]+`
- This operation responds to pagination parameters

[GET] `/runtime/{deploymentId}/history/instance/{procInstId}/child`

- Gets a list of `ProcessInstanceLog` instances associated with any child/sub-processes associated with the specified process instance
- Returns a `JaxbHistoryLogList` instance that contains a list of `JaxbProcessInstanceLog` instances
- The *processInstId* component of the URL must conform to the following regex: `[0-9]+`
- This operation responds to pagination parameters

[GET] `/runtime/{deploymentId}/history/instance/{procInstId}/node`

- Gets a list of `NodeInstanceLog`instances associated with the specified process instance
- Returns a `JaxbHistoryLogList` instance that contains a list of `JaxbNodeInstanceLog` instances
- The *processInstId* component of the URL must conform to the following regex: `[0-9]+`
- This operation responds to pagination parameters

[GET] `/runtime/{deploymentId}/history/instance/{procInstId}/variable`

- Gets a list of `VariableInstanceLog` instances associated with the specified process instance
- Returns a `JaxbHistoryLogList` instance that contains a list of `JaxbVariableInstanceLog` instances
- The *processInstId* component of the URL must conform to the following regex: `[0-9]+`

- This operation responds to pagination parameters

[GET] `/runtime/{deploymentId}/history/instance/{procInstId}/node/{nodeId}`

- Gets a list of `NodeInstanceLog` instances associated with the specified process instance that have the given (node) id
- Returns a `JaxbHistoryLogList` instance that contains a list of `JaxbNodeInstanceLog` instances
- The *processInstId* component of the URL must conform to the following regex:`[0-9]+`
- The *nodeId* component of the URL must conform to the following regex: `[a-zA-Z0-9-:\.]+`
- This operation responds to pagination parameters

[GET] `/runtime/{deploymentId}/history/instance/{procInstId}/variable/{varId}`

- Gets a list of `VariableInstanceLog` instances associated with the specified process instance that have the given (variable) id
- Returns a `JaxbHistoryLogList` instance that contains a list of `JaxbVariableInstanceLog` instances
- The *processInstId* component of the URL must conform to the following regex: `[0-9]+`
- The *varId* component of the URL must conform to the following regex: `[a-zA-Z0-9-:\.]+`
- This operation responds to pagination parameters

[GET] `/runtime/{deploymentId}/history/process/{processDefId}`

- Gets a list of `ProcessInstanceLog` instances associated with the specified process definition
- Returns a `JaxbHistoryLogList` instance that contains a list of `JaxbProcessInstanceLog` instances
- The *processDefId* component of the URL must conform to the following regex: `[_a-zA-Z0-9-:\.]+`
- This operation responds to pagination parameters

17.2.2.2.1. History calls that search by variable

[GET] `/runtime/{deploymentId}/history/variable/{varId}`

- Gets a list of `VariableInstanceLog` instances associated with the specified variable id
- Returns a `JaxbHistoryLogList` instance that contains a list of `JaxbVariableInstanceLog` instances

- The *varId* component of the URL must conform to the following regex: `[a-zA-Z0-9-:\.]+`
- This operation responds to pagination parameters

[GET] `/runtime/{deploymentId}/history/variable/{varId}/value/{value}`

- Gets a list of `VariableInstanceLog` instances associated with the specified variable id that contain the value specified
- Returns a `JaxbHistoryLogList` instance that contains a list of `JaxbVariableInstanceLog` instances
- Both the *varId* and *value* components of the URL must conform to the following regex: `[a-zA-Z0-9-:\.]+`
- This operation responds to pagination parameters

[GET] `/runtime/{deploymentId}/history/variable/{varId}/instances`

- Gets a list of `ProcessInstance` instances that contain the variable specified by the given variable id.
- Returns a `JaxbProcessInstanceListResponse` instance that contains a list of `JaxbProcessInstanceResponse` instances
- The *varId* component of the URL must conform to the following regex: `[a-zA-Z0-9-:\.]+`
- This operation responds to pagination parameters

[GET] `/runtime/{deploymentId}/history/variable/{varId}/value/{value}/instances`

- Gets a list of `ProcessInstance` instances that contain the variable specified by the given variable id which contains the (variable) value specified
- Returns a `JaxbProcessInstanceListResponse` instance that contains a list of `JaxbProcessInstanceResponse` instances
- Both the *varId* and *value* components of the URL must conform to the following regex: `[a-zA-Z0-9-:\.]+`
- This operation responds to pagination parameters

17.2.3. Task calls

The following section describes the three different types of task calls:

- Task REST operations that mirror the `TaskService` interface, allowing the user to interact with the remote `TaskService` instance
- The Task query REST operation, that allows users to query for `Task` instances
- Other Task REST operations that retrieve information

Task operation authorizations. Task REST operations use the user information (used to authorize and authenticate the HTTP call) to check whether or not the requested operations can happen. This also applies to REST calls that retrieve information, such as the task query operation. REST calls that request information will only return information about tasks that the user is allowed to see.

With regards to retrieving information, only users associated with a task may retrieve information about the task. However, the authorizations of progress and other modifications of task information are more complex. See the *Task Permissions* section in the *Task Service* documentation for more infomration.

Note

Given that many users have expressed the wish for a "super-task-user" that can execute task REST operations on all tasks, regardless of the users associated with the task, there are now plans to implement that feature. However, for the 6.0.x releases, this feature is not available.

17.2.3.1. Task operation calls

All of the task operation calls described in this section use the user (id) used in the REST basic authorization as input for the user parameter in the specific call.

Some of the operations take an optional `lanaguage` query parameter. If this parameter is not given as a element of the URL itself, the default value of "`en-UK`" is used.

The *taskId* component of the REST calls below must conform to the following regex:

- `[0-9]+`

[POST] `/task/{taskId}/activate`

- Activates a task
- Returns a `JaxbGenericResponse` with the status of the operation

[POST] `/task/{taskId}/claim`

- Claims a task
- Returns a `JaxbGenericResponse` with the status of the operation

[POST] `/task/{taskId}/claimnextavailable`

- Claims the next available task
- Returns a `JaxbGenericResponse`with the status of the operation

- Takes an optional `language`query parameter.

[POST] `/task/{taskId}/complete`

- Completes a task
- Returns a `JaxbGenericResponse` with the status of the operation
- Takes map query parameters, which are the "results" input for the complete operation

[POST] `/task/{taskId}/delegate`

- Delegates a task
- Returns a `JaxbGenericResponse` with the status of the operation
- Requires a `targetId` query parameter, which identifies the user or group to which the task is delegated

[POST] `/task/{taskId}/exit`

- Exits a task
- Returns a `JaxbGenericResponse` with the status of the operation

[POST] `/task/{taskId}/fail`

- Fails a task
- Returns a `JaxbGenericResponse` with the status of the operation

[POST] `/task/{taskId}/forward`

- Delegates a task
- Returns a `JaxbGenericResponse` with the status of the operation
- Requires a `targetId` query parameter, which identifies the user or group to which the task is forwarded

[POST] `/task/{taskId}/nominate`

- Nominates a task
- Returns a `JaxbGenericResponse` with the status of the operation
- Requires at least one of either the `user` or `group` query parameter, which identify the user(s) or group(s) that are nominated for the task

[POST] `/task/{taskId}/release`

- Releases a task
- Returns a `JaxbGenericResponse` with the status of the operation

[POST] `/task/{taskId}/resume`

- Resumes a task
- Returns a `JaxbGenericResponse` with the status of the operation

[POST] `/task/{taskId}/skip`

- Skips a task
- Returns a `JaxbGenericResponse` with the status of the operation

[POST] `/task/{taskId}/start`

- Starts a task
- Returns a `JaxbGenericResponse` with the status of the operation

[POST] `/task/{taskId}/stop`

- Stops a task
- Returns a `JaxbGenericResponse` with the status of the operation

[POST] `/task/{taskId}/suspend`

- Suspends a task
- Returns a `JaxbGenericResponse` with the status of the operation

17.2.3.2. Task query call

[GET] `/task/query`

The `/task/query` operation queries all non-archived tasks based on the parameters given.

- Queries the available non-archived tasks
- Returns a `JaxbTaskSummaryListResponse` with a list of `TaskSummaryImpl` instances.
- Takes the following (case-*in*sensitive) query parameters listed below:
  - `businessAdministrator`
    - Specifies that the returned tasks should have the business administrator identified by this parameter
    - This parameter may be repeated
  - `potentialOwner`
    - Specifies that the returned tasks should have the potential owner identified by this parameter
    - This parameter may be repeated

- `processInstanceId`
  - Specifies that the returned tasks should be associated with the process instance identified by this parameter
  - This parameter may be repeated
- `status`
  - Specifies that the returned tasks should have the status identified by this parameter
  - This parameter may be repeated
- `taskId`
  - Specifies that the returned tasks should have the (task) id identified by this parameter
  - This parameter may be repeated
- `taskOwner`
  - Specifies that the returned tasks should have the task owner (initiator) identified by this parameter
  - This parameter may be repeated
- `workItemId`
  - Specifies that the returned tasks should be associated with the work item identified by this parameter
  - This parameter may be repeated
- `language`
  - Specifies the language that the returned tasks should be associated with
  - This parameter may be repeated
- `union`
  - This specifies whether the query should query the union or intersection of the parameters. See below for more info.
  - This parameter must only be passed *once*

Example 17.1. Query usage

This call retrieves the task summaries of all tasks that have a work item id of 3, 4, *or* 5. If you specify the *same* parameter multiple times, the query will select tasks that match *any* of that parameter.

- `http://server:port/rest/task/query?workItemId=3&workItemId=4&workItemId=5`

The next call will retrieve any task summaries for which the task id is 27 *and* for which the work item id is 11. Specifying *different* parameters will result in a set of tasks that match *both* (*all*) parameters.

- `http://server:port/rest/task/query?workItemId=11&taskId=27`

The next call will retrieve any task summaries for which the task id is 27 *or* the work item id is 11. While these are different parameters, the `union` parameter is being used here so that the union of the two queries (the work item id query and the task id query) is returned.

- `http://server:port/rest/task/query?workItemId=11&taskId=27&union=true`

The next call will retrieve any task summaries for which the status is `` `Created` `` *and* the potential owner of the task is `Bob`. Note that the letter case for the status parameter value is case-*in*sensitve.

- `http://server:port/rest/task/query?status=creAted&potentialOwner=Bob`

The next call will return any task summaries for which the status is `` `Created` `` *and* the potential owner of the task is `bob`. Note that the potential owner parameter is case-*sensitive*. `bob` is not the same user id as `Bob`!

- `http://server:port/rest/task/query?status=created&potentialOwner=bob`

The next call will return the *intersection* of the set of task summaries for which the process instance is 201, the potential owner is `bob` and for which the status is `` `Created` `` *or* `` `Ready` ``.

- `http://server:port/rest/task/query?status=created&status=ready&potentialOwner=bob&processInstanceId=201`

That means that the task summaries that have the following characteristics would be included:

- process instance id 201, potential owner `bob`, status `Ready`
- process instance id 201, potential owner `bob`, status `Created`

And that following task summaries will *not* be included:

- process instance id *183*, potential owner `bob`, status `Created`
- process instance id 201, potential owner `*mary*`, status `Ready`
- process instance id 201, potential owner `bob`, status `*Complete*`

17.2.3.3. Other Task calls

[GET] `/task/{taskId}/content`

- Gets the task content from a task identified by the given task id
- Returns a `JaxbContent` with the content of the task
- The *taskId* component of the URL must conform to the following regex: `[0-9]+`

[GET] `/task/content/{contentId}`

- Gets the task content from a task identified by the given content id
- Returns a `JaxbContent` with the content of the task
- The *contentId* component of the URL must conform to the following regex: `[0-9]+`

17.2.4. Deployment calls

The calls described in this section allow users to manage deployments. Deployments are in fact `KieModule` JARs which can be deployed or undeployed, either via the UI or via the REST calls described below. Configuration options, such as the runtime strategy, should be specified when deploying the deployment: the configuration of a deployment can not be changed after it has already been deployed.

The above deploymentId regular expression describes an expression that contains the following elements, separated from eachother by a `:` character:

1. The group id
2. The artifact id
3. The version
4. The (optional) kbase id
5. The (optional) ksession id

In a more formal sense, the *deploymentId* component of the REST calls below must conform to the following regex:

- `[\w\.-]+(:[\w\.-]+){2,2}(:[\w\.-]*){0,2}`

This regular expression is explained as follows:

- The `[\w\.-]` element, which occurs 3 times in the above regex, refers to a character set that can contain the following character sets:

  ```
  [A-Z]          [0-9]          .
  [a-z]          _              -
  ```

- This `[\w\.-]` element occurs at least 3 times and at most 5 times, separated by a ':' character each time.

Example 17.2. Accepted `deploymentId`'s

- `com.wonka:choco-maker:67.190`

These example `deploymentId`'s contain the optional *kbase* and *ksession* id groups.

- `com.wonka:choco-maker:67.190:oompaBase`
- `com.wonka:choco-maker:67.190:oompaLoompaBase:gloopSession`

17.2.4.1. Asynchrous deployment calls

There are 2 operations that can be used to modify the status of a deployment:

- `/deployments/{deploymentId}/deploy`
- `/deployments/{deploymentId}/undeploydeploy`

These `POST` deployment calls are both *asynchronous*, which means that the information returned by the `POST` request does not reflect the eventual final status of the operation itself.

Important

As noted above, both the `/deploy` and `/undeploy` operations are *asynchronous* REST operations. Successfull requests to these URLs will return the status `202` upon the request completion. RFC 2616 defines the `202` status as meaning that "the request has been accepted for processing, but the processing has not been completed."

This means the following:

1. While the request may have been accepted "successfully", the operation itself (deploying or undeploying the deployment unit) may actually fail.
2. Furthermore, information about deployments, such as that retrieved by calling the `GET` operations described below, are *snapshots* and the information (including the status of the deployment unit) may have changed by the time the user client receives the answer to the `GET` request.

17.2.4.2. Additional deployment call details

In addition to the asynchronous nature of the `POST` deploy and undeploy calls described above, the following information is also important to know:

- The deploy and undeploy operations can fail if one of the following is true:
  1. An identical job has already been submitted to the queue and has not yet completed.
  2. The amount of (deploy/undeploy) jobs submitted but not yet processed exceeds the deploy/undeploy job queue size.

17.2.4.2.1. The deploy/undeploy job queue

Requests for the deployment or undeployment of deployment units are added as they are received to a single queue and processed in a serial fashion.

The reason for this is that the increased complexity necessary to manage and safeguard concurrent deploy and undeploy operation requests possibly involving the same deployment unit is far more costly than the benefit of having these requests complete safely in a concurrent environment, especially given that deploy and undeploy requests will be far less utilized than parts of the remote API involving actions on the deployment units themselves.

The size of this queue can be determined at startup by the parameter described below. Otherwise, the deployment job queue size defaults to 100.

The following system properties can be specified in order to change the size of the deploy/undeploy job queue:

Table 17.13. System properties that influence REST deployment call behavior

Parameter name	Description
`org.kie.remote.rest.deployment.job.queue.size`	Specifies the size of the deploy/undeploy job queue. Must be numerical.

17.2.4.3. Deployment call details

[GET] `/deployment/`

- Returns a list of all the available deployed instances in a `JaxbDeploymentUnitList` instance

[GET] `/deployment/ {deploymentId}`

- Returns a `JaxbDeploymentUnit` instance containing th e information (including the configuration) of the deployment unit.
  - This operation will fail when the URL uses a *deployementId* that refers to a deployment unit that does not exist or for which the deployment has not yet been completed.
  - This operation may succeed for deployment units for which an undeploy operation request has not yet completed.

[POST] `/deployment/ {deploymentId} /deploy`

- Deploys the deployment unit referenced by the *deploymentId*
- Returns a `JaxbDeploymentJobResult` instance with the status of the *request*

- Takes a `strategy` query parameter:
  - This parameter describes the runtime strategy used for the deployment.
  - This parameter takes the following (case- *in* sensitive) values:
    - `SINGLETON`
    - `PER_REQUEST`
    - `PER_PROCESS_INSTANCE`
  - The default runtime strategy used for a deployment is `SINGLETON` .
- The deploy operation is an *asynchronous* operation. The status of the deployment can be retrieved using the `GET` calls described above.
- The request can fail for the reasons described

[POST] `/deployment/ {deploymentId} /undeploy`

- Undeploys the deployment unit referenced by the *deploymentId*
- Returns a `JaxbDeploymentJobResult` instance with the status of the *request*
- The undeploy operation is an *asynchronous* operation. The status of the deployment can be retrieved using the `GET` calls described above.

17.2.5. Execute calls

While there is a `/runtime/{id}/execute` and a `task/execute` method, both will take all types of commands. This is possible because execute takes a JaxbCommandsRequest object, which contains a list of `(org.kie.api.command.)Command` objects. The `JaxbCommandsRequest` has fields to store the proper `deploymentId` and `processInstanceId` information.

Of course, if you send a request with a command that needs this information (`deploymentId`, for example) and don't fill the `deploymentId` in, the request will fail.

17.2.5.1. Execution call details

[POST] `/task/execute`

- Executes a `Command` , assumed to be related to tasks.
- Returns a `JaxbCommandResponse` implementation with the result of the operation

[POST] `/runtime/ {deploymentId} /execute`

- Executes a `Command` , assumed to be related to business processes or the knowledge session.

- Returns a `JaxbCommandResponse` implementation with the result of the operation

17.2.5.2. Commands accepted

Runtime commands.

AbortWorkItemCommand	GetProcessInstancesCommand	GetGlobalCommand
CompleteWorkItemCommand	SetProcessInstanceVariablesCommand	GetIdCommand
GetWorkItemCommand	SignalEventCommand	SetGlobalCommand
AbortProcessInstanceCommand	StartCorrelatedProcessCommand	DeleteCommand
GetProcessIdsCommand	StartProcessCommand	FireAllRulesCommand
GetProcessInstanceByCorrelationKeyCommand	GetVariableCommand	InsertObjectCommand
GetProcessInstanceCommand	GetFactCountCommand	UpdateCommand

Task commands.

ActivateTaskCommand	FailTaskCommand	GetTasksOwnedCommand
AddTaskCommand	ForwardTaskCommand	NominateTaskCommand
CancelDeadlineCommand	GetAttachmentCommand	ProcessSubTaskCommand
ClaimNextAvailableTaskCommand	GetContentCommand	ReleaseTaskCommand
ClaimTaskCommand	GetTaskAssignedAsBusinessAdminCommand	ResumeTaskCommand
CompleteTaskCommand	GetTaskAssignedAsPotentialOwnerCommand	SkipTaskCommand
CompositeCommand	GetTaskByWorkItemIdCommand	StartTaskCommand
DelegateTaskCommand	GetTaskCommand	StopTaskCommand
ExecuteTaskRulesCommand	GetTasksByProcessInstanceIdCommand	SuspendTaskCommand
ExitTaskCommand	GetTasksByStatusByProcessInstanceIdCommand	

Task commands.

ClearHistoryLogsCommand	FindProcessInstanceCommand	FindSubProcessInstancesCommand
FindActiveProcessInstancesCommand	FindProcessInstancesCommand	FindVariableInstancesByNameCommand
FindNodeInstancesCommand	FindSubProcessInstancesCommand	FindVariableInstancesCommand

17.2.6. Additional Information

17.2.6.1. Serialization: JAXB or JSON

Except for the *Execute calls*, all other REST calls described below can use either JAXB or JSON.

All REST calls, unless otherwise specified, will use JAXB serialization.

When using JSON, make sure to add the JSON media type (`"application/json"`) to the `ACCEPT` header of your REST call.

17.2.6.2. Sending and receiving user class instances

Sometimes, users may wish to pass instances of their own classes as parameters to commands sent in a REST request or JMS message. In order to do this, there are a number of requirements.

1. The user-defined class satisfy the following in order to be property serialized and deserialized by the JMS API:

   - The user-defined class must be correctly annotated with JAXB annotations, including the following:

     - The user-defined class must be annotated with a `javax.xml.bind.annotation.XmlRootElement` annotation with a non-empty `name` value

     - All fields or getter/setter methods must be annotated with a `javax.xml.bind.annotation.XmlElement` or `javax.xml.bind.annotation.XmlAttribute` annotations.

     Furthermore, the following usage of JAXB annotations is recommended:

     - Annotate the user-defined class with a `javax.xml.bind.annotation.XmlAccessorType` annotation specifying that fields should be used, (`javax.xml.bind.annotation.XmlAccessType.FIELD`). This also means that you should annotate the fields (instead of the getter or setter methods) with `@XmlElement` or `@XmlAttribute` annotations.

     - Fields annotated with `@XmlElement` or `@XmlAttribute` annotations should also be annotated with `javax.xml.bind.annotation.XmlSchemaType` annotations specifying the type of the field, even if the fields contain primitive values.

     - Use objects to store primitive values. For example, use the `java.lang.Integer` class for storing an integer value, and not the `int` class. This way it will always be obvious if the field is storing a value.

   - The user-defined class definition must implement a no-arg constructor.

   - Any fields in the user-defined class must either be object primitives (such as a `Long` or `String`) or otherwise be objects that satisfy the first 2 requiremends in this list (correct usage of JAXB annotations and a no-arg constructor).

2. The class definition must be included in the deployment jar of the deployment that the JMS message content is meant for.

3. The sender must set a “deploymentId” string property on the JMS bytes message to the name of the deploymentId. This property is necessary in order to be able to load the proper classes from the deployment itself before deserializing the message on the server side.

Retrieving process variables

While *submitting* an instance of a user-defined class is possible via both the JMS and REST API's, *retrieving* an instance of the process variable is only possible via the REST API.

17.2.6.3. Including the deployment id

When interacting with the Remote API, users may want to pass instances of their own classes as parameters to certain operations. As mThis will only be possible if the KJar for a deployment includes these classes.

REST calls that involve the `TaskService` (e.g. that start with `/task`..), often do not contain any information about the associated deployment. In that case, an extra query parameter will have to be added to the REST call so that the server can find the appropriate deployment with the class (definition) and correctly deserialize the information passed with the call.

For these REST calls which do not contain the deployment id, you'll need to add the following parameter:

Table 17.14. Deployment id query parameter

Parameter name	Description
`deploymentId`	Value (must match the regex `[a-zA-Z0-9-:\.]+`) specifies the deployment which contains the user-defined class(es) needed to correctly deserialze information passed in the call

17.2.6.4. Pagination

Some of the REST calls below return lists of information. The results of these operations can be *paginated* , which means that the lists can be split up and returned according to the parameters sent by the user.

For example, if the REST call parameters indicate that *page 2* with *page size 10* should be returned for the results, then results 10 to (and including) 19 will be returned.

The first page is always page 1 (as opposed to page "0").

Table 17.15. Pagination query parameter syntax

Parameter name	Description
`page`	The page number requested. The default value is 1.
`p`	Synonym for the above `page` parameter.
`pageSize`	The number of elements per page to return. The default value is 10.
`s`	Synonym for the above `pageSize` parameter.

If both a "long" pagination parameter and its synonym are used, then only the value from the "long" variant is used. For example, if the `page` is given with a value of 11 and the `p` parameter

is given with a value of 37, then the value of the `page` parameter, *11* , will be used and the `p` parameter will be ignored.

For the following operations, pagination is *always* used. See above for the default values used.

Table 17.16. REST operations using pagination

REST call URL	Short Description
`/runtime/{deploymentId}/history/instance`	Returns a list of `ProcessInstanceLog` instances
`runtime/{deploymentId}/history/instance/{procInstid}`	Returns a list of `ProcessInstanceLog` instances
`/runtime/{deploymentId}/history/instance/{procInstId}/child`	Returns a list of `ProcessInstanceLog` instances
`/runtime/{deploymentId}/history/instance/{procInstId}/node`	Returns a list of `NodeInstanceLog` instances
`/runtime/{deploymentId}/history/instance/{procInstId}/node/{nodeId}`	Returns a list of `NodeInstanceLog` instances
`/runtime/{deploymentId}/history/instance/{procInstId}/variable`	Returns a list of `VariableInstanceLog` instances
`/runtime/{deploymentId}/history/instance/{procInstId}/variable/{varId}`	Returns a list of `VariableInstanceLog` instances
`/runtime/{deploymentId}/history/variable/{variableId}`	Returns a list of `VariableInstanceLog` instances
`/runtime/{deploymentId}/history/variable/{variableId}/instances`	Returns a list of `ProcessInstance` instances
`/runtime/{deploymentId}/history/variable/{variableId}/value/{value}`	Returns a list of `VariableInstanceLog` instances
`/runtime/{deploymentId}/history/variable/{variableId}/value/{value}/instances`	Returns a list of `ProcessInstance` instances
`/runtime/{deploymentId}/history/process/{procDefId}`	Returns a list of `ProcessInstanceLog` instances
`/task/query`	Returns a list of `TaskSummaryImpl` instances

17.2.6.5. Map query parameters

If you're triggering an operation with a REST API call that would normally (e.g. when interacting the same operation on a local `KieSession` or `TaskService` instance) take an instance of a `java.util.Map` as one of its parameters, you can submit key-value pairs to the operation to simulate this behaviour by passing a query parameter whose name starts with `map_` .

Example 17.3.

If you pass the query parameter `map_kEy=vAlue` in a REST call, then the `Map` that's passed to the actual underlying `KieSession` or `TaskService` operation will contain this (`String, String`) key value pair: `"kEy" => "vAlue"` .You could pass this parameter like so:

```
http://localhost:8080/kie-wb/rest/runtime/myproject/process/
wonka.factory.loompa.hire/start?map_kEy=vAlue
```

Map query parameters also use the object query parameter syntax described below, so the following query parameter, `map_total=5000` will be translated into a key-value pair in a map where the key is the String "total" and the value is a Long with the value of 5000. For example:

```
http://localhost:8080/kie-wb/rest/runtime/myproject/process/
wonka.factory.oompa.chocolate/start?map_total=5000
```

The following operations take query map parameters:

- `/runtime/{deploymentId}/process/{processDefId}/start`
- `/runtime/{deploymentId}/workitem/{processItemId}/complete`
- `/runtime/{deploymentId}/withvars/process/{processDefId}/start`
- `/task/{taskId}/complete`
- `/task/{taskId}/fail`

17.2.6.6. Number query parameters

While REST calls obviously only take strings as query parameters, using the following notation for query parameters will mean that the string is translated into a different type of object when the value of the string is used in the actual operation:

Table 17.17. Number query parameter syntax

Regex syntax	Type
`\d+`	`Long`
`\d+i`	`Integer`
`\d+l`	`Long`

17.2.6.7. Runtime strategies

The REST calls allow access to the underlying deployments, regardless of whether these deployments use the `Singleton` , `Per-Process-Instance` or `Per-Request` strategies.

While there's enough information in the URL in order to access deployments that use the `Singleton` , or `Per-Request` strategies, that's not always the case with the `Per-Process-Instance` runtimes because the REST operation will obviously need the process instance id in order to identify the deployment.

Therefore, for REST calls for which the URL does not contain the process instance id, you'll need to add the following parameter:

Table 17.18. Per-Process-Instance runtime query parameter

Parameter name	Description
`runtimeProcInstId`	Value (must match the regex `[0-9]+`) specifies the process instance id that identifies the underlying `Per-Process-Instance` deployment Will have no effect if the underlying deployment uses the `Singleton` or `Per-Request` strategy

17.2.7. REST summary

The URL templates in the table below are relative the following URL:

- `http://server:port/business-central/rest`

Table 17.19. runtime REST calls

URL Template	Type	Description
/runtime/{deploymentId}/process/{procDefID}/start	POST	start a process instance based on the Process definition (accepts query map parameters)
/runtime/{deploymentId}/process/instance/{procInstanceID}	GET	return a process instance details
/runtime/{deploymentId}/process/instance/{procInstanceID}/abort	POST	abort the process instance
/runtime/{deploymentId}/process/instance/{procInstanceID}/signal	POST	send a signal event to process instance (accepts query map parameters)
/runtime/{deploymentId}/process/instance/{procInstanceID}/variable/{varId}	GET	return a variable from a process instance
/runtime/{deploymentId}/signal/{signalCode}	POST	send a signal event to deployment
/runtime/{deploymentId}/workitem/{workItemID}/complete	POST	complete a work item (accepts query map parameters)
/runtime/{deploymentId}/workitem/{workItemID}/abort	POST	abort a work item

URL Template	Type	Description
/runtime/{deploymentId}/withvars/process/{procDefinitionID}/start	POST	start a process instance and return the process instance with its variables Note that even if a passed variable is not defined in the underlying process definition, it is created and initialized with the passed value.
/runtime/{deploymentId}/withvars/process/instance/{procInstanceID}/	GET	return a process instance with its variables
/runtime/{deploymentId}/withvars/process/instance/{procInstanceID}/signal	POST	send a signal event to the process instance (accepts query map parameters) The following query parameters are accepted: • The `signal` parameter specifies the name of the signal to be sent • The `event` parameter specifies the (optional) value of the signal to be sent

Table 17.20. task REST calls

URL Template	Type	Description
/task/query	GET	return a TaskSummary list
/task/content/{contentID}	GET	returns the content of a task
/task/{taskID}	GET	return the task
/task/{taskID}/activate	POST	activate the task
/task/{taskID}/claim	POST	claim the task
/task/{taskID}/claimnextavailable	POST	claim the next available task
/task/{taskID}/complete	POST	complete the task (accepts query map paramaters)

URL Template	Type	Description
/task/{taskID}/delegate	POST	delegate the task
/task/{taskID}/exit	POST	exit the task
/task/{taskID}/fail	POST	fail the task
/task/{taskID}/forward	POST	forward the task
/task/{taskID}/nominate	POST	nominate the task
/task/{taskID}/release	POST	release the task
/task/{taskID}/resume	POST	resume the task (after suspending)
/task/{taskID}/skip	POST	skip the task
/task/{taskID}/start	POST	start the task
/task/{taskID}/stop	POST	stop the task
/task/{taskID}/suspend	POST	suspend the task
/task/{taskID}/content	GET	returns the content of a task

Table 17.21. history REST calls

URL Template	Type	Description
/history/clear/	POST	delete all process, node and history records
/history/instances	GET	return the list of all process instance history records
/history/instance/{procInstId}	GET	return a list of process instance history records for a process instance
/history/instance/{procInstId}/child	GET	return a list of process instance history records for the subprocesses of the process instance
/history/instance/{procInstId}/node	GET	return a list of node history records for a process instance
/history/instance/{procInstId}/node/{nodeId}	GET	return a list of node history records for a node in a process instance

URL Template	Type	Description
/history/instance/{procInstId}/variable	GET	return a list of variable history records for a process instance
/history/instance/{procInstId}/variable/{variableId}	GET	return a list of variable history records for a variable in a process instance
/history/process/{procDefId}	GET	return a list of process instance history records for process instances using a given process definition
/history/variable/{varId}	GET	return a list of variable history records for a variable
/history/variable/{varId}/instances	GET	return a list of process instance history records for process instances that contain a variable with the given variable id
/history/variable/{varId}/value/{value}	GET	return a list of variable history records for variable(s) with the given variable id and given value
/history/variable/{varId}/value/{value}/instances	GET	return a list of process instance history records for process instances with the specified variable that contains the specified variable value

Table 17.22. deployment REST calls

URL Template	Type	Description
/deployments	GET	return a list of (deployed) deployments
/deployment/{deploymentId}	GET	return the status and information about the deployment
/deployment/{deploymentId}/deploy	POST	submit a request to deploy a deployment

URL Template	Type	Description
/deployment/{deploymentId}/undeploy	POST	submit a request to undeploy a deployment

17.3. JMS

The Java Message Service (JMS) is an API that allows Java Enterprise components to communicate with each other asynchronously and reliably.

Operations on the runtime engine and tasks can be done via the JMS API exposed by the jBPM console and KIE workbench. However, it's not possible to manage deployments or the knowledge base via this JMS API.

Unlike the REST API, it is possible to send a batch of commands to the JMS API that will all be processed in one request after which the responses to the commands will be collected and return in one response message.

17.3.1. JMS Queue setup

When the Workbench is deployed on the JBoss AS or EAP server, it automatically creates 3 queues:

- `jms/queue/KIE.SESSION`
- `jms/queue/KIE.TASK`
- `jms/queue/KIE.RESPONSE`

The `KIE.SESSION` and `KIE.TASK` queues should be used to send request messages to the JMS API. Command response messages will be then placed on the `KIE.RESPONSE` queues. Command request messages that involve starting and managing business processes should be sent to the `KIE.SESSION` and command request messages that involve managing human tasks, should be sent to the `KIE.TASK` queue.

Although there are 2 different input queues, `KIE.SESSION` and `KIE.TASK`, this is only in order to provide multiple input queues so as to optimize processing: command request messages will be processed in the same manner regardless of which queue they're sent to. However, in some cases, users may send many more requests involving human tasks than requests involving business processes, but then not want the processing of business process-related request messages to be delayed by the human task messages. By sending the appropriate command request messages to the appropriate queues, this problem can be avoided.

The term "*command request message*" used above refers to a JMS byte message that contains a serialized `JaxbCommandsRequest` object. At the moment, only XML serialization (as opposed to, JSON or protobuf, for example) is supported.

17.3.2. Using the remote Java API

While it is possible to interact with a BPMS or KIE workbench server instance by sending and processing JMS messages that you create yourself, *it will always be easier to use the remote Java API* that's supplied by the `kie-services-client` jar.

For more information about how to use the remote Java API to interact with the JMS API of a server instance, see the *Remote Java API* section.

17.3.3. Serialization issues

The JMS API accepts `ByteMessage` instances that contain serialized `JaxbCommandsRequest` objects. These `JaxbCommandsRequest` instances can be filled with multiple command objects. In this way, it's possible to send a batch of commands for processing to the JMS API.

When users wish to include their own classes with requests, there a number of requirements that must be met for the user-defined classes. For more information about these requirements, see the *Sending and receiving user class instances* section in the REST API documentation.

17.3.4. Example JMS usage

The following is a rather long example that shows how to use the JMS API. The numbers ("callouts") along the side of the example refer to notes below that explain particular parts of the example. It's supplied for those advanced users that do not wish to use the jBPM Remote Java API.

The jBPM Remote Java API, described here, will otherwise take care of all of the logic shown below.

```
// normal java imports skipped

import org.drools.core.command.runtime.process.StartProcessCommand;
import org.jbpm.services.task.commands.GetTaskAssignedAsPotentialOwnerCommand;
import org.kie.api.command.Command;
import org.kie.api.runtime.process.ProcessInstance;
import org.kie.api.task.model.TaskSummary;

import org.kie.services.client.api.command.exception.RemoteCommunicationExcepti(1)on;
import org.kie.services.client.serialization.JaxbSerializationProvider;
import org.kie.services.client.serialization.SerializationConstants;
import org.kie.services.client.serialization.SerializationException;
import org.kie.services.client.serialization.jaxb.impl.JaxbCommandResponse;
import org.kie.services.client.serialization.jaxb.impl.JaxbCommandsRequest;
import org.kie.services.client.serialization.jaxb.impl.JaxbCommandsResponse;
import org.kie.services.client.serialization.jaxb.rest.JaxbExceptionResponse;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
```

```
public class DocumentationJmsExamples {

  protected static final Logger logger = LoggerFactory.getLogger(DocumentationJmsExamples.class

  public void sendAndReceiveJmsMessage() {

    String USER = "charlie";
    String PASSWORD = "ch0c0licious";

    String DEPLOYMENT_ID = "test-project";
    String PROCESS_ID_1 = "oompa-processing";
    URL serverUrl;
    try {
      serverUrl = new URL("http://localhost:8080/jbpm-console/");
    } catch (MalformedURLException murle) {
      logger.error("Malformed URL for the server instance!", murle);
      return;
    }

    // Create JaxbCommandsRequest instance and add commands
    Command<?> cmd = new StartProcessCommand(PROCESS_ID_1);
    int oompaProcessingResultIndex = 0;                                                    5
   JaxbCommandsRequest req = new JaxbCommandsRequest(DEPLOYMENT_ID, cmd);                  2
    req.getCommands().add(new GetTaskAssignedAsPotentialOwnerCommand(USER, "en-
UK"));
    int loompaMonitoringResultIndex = 1;                                                   5

    // Get JNDI context from server
    InitialContext context = getRemoteJbossInitialContext(serverUrl, USER, PASSWORD);

    // Create JMS connection
    ConnectionFactory connectionFactory;
    try {
              connectionFactory  =  (ConnectionFactory)  context.lookup("jms/
RemoteConnectionFactory");
    } catch (NamingException ne) {
     throw new RuntimeException("Unable to lookup JMS connection factory.", ne);
    }

    // Setup queues
    Queue sendQueue, responseQueue;
    try {
      sendQueue = (Queue) context.lookup("jms/queue/KIE.SESSION");
      responseQueue = (Queue) context.lookup("jms/queue/KIE.RESPONSE");
    } catch (NamingException ne) {
     throw new RuntimeException("Unable to lookup send or response queue", ne);
    }
```

```
    // Send command request
     Long processInstanceId = null; // needed if you're doing an operation on
a PER_PROCESS_INSTANCE deployment
    String humanTaskUser = USER;
    JaxbCommandsResponse cmdResponse = sendJmsCommands(
        DEPLOYMENT_ID, processInstanceId, humanTaskUser, req,
        connectionFactory, sendQueue, responseQueue,
        USER, PASSWORD, 5);

    // Retrieve results
    ProcessInstance oompaProcInst = null;
    List<TaskSummary> charliesTasks = null;
   for (JaxbCommandResponse<?> response : cmdResponse.getResponses()) {        6
     if (response instanceof JaxbExceptionResponse) {
       // something went wrong on the server side
       JaxbExceptionResponse exceptionResponse = (JaxbExceptionResponse) response;
       throw new RuntimeException(exceptionResponse.getMessage());
     }

     if (response.getIndex() == oompaProcessingResultIndex) {                  5
       oompaProcInst = (ProcessInstance) response.getResult();                 6
     } else if (response.getIndex() == loompaMonitoringResultIndex) {          5
       charliesTasks = (List<TaskSummary>) response.getResult();               6
     }
    }
  }

  private JaxbCommandsResponse sendJmsCommands(String deploymentId, Long processInstanceId, Str
      ConnectionFactory factory, Queue sendQueue, Queue responseQueue, String jmsUser, String j
    req.setProcessInstanceId(processInstanceId);
    req.setUser(user);

    Connection connection = null;
    Session session = null;
    String corrId = UUID.randomUUID().toString();
    String selector = "JMSCorrelationID = '" + corrId + "'";
    JaxbCommandsResponse cmdResponses = null;
    try {

      // setup
      MessageProducer producer;
      MessageConsumer consumer;
      try {
        if (jmsPassword != null) {
          connection = factory.createConnection(jmsUser, jmsPassword);
        } else {
          connection = factory.createConnection();
        }
```

```
        session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);

        producer = session.createProducer(sendQueue);
        consumer = session.createConsumer(responseQueue, selector);

        connection.start();
      } catch (JMSException jmse) {
            throw new RemoteCommunicationException("Unable to setup a JMS
connection.", jmse);
      }

      JaxbSerializationProvider serializationProvider = new JaxbSerializationProvider();
      // if necessary, add user-created classes here:
                              //     xmlSerializer.addJaxbClasses(MyType.class,
AnotherJaxbAnnotatedType.class);

      // Create msg
      BytesMessage msg;
      try {
        msg = session.createBytesMessage();

        // set properties
        msg.setJMSCorrelationID(corrId);
        msg.setIntProperty(SerializationConstants.SERIALIZATION_TYPE_PROPERTY_NAME, JaxbSeria
        Collection<Class<?>> extraJaxbClasses = serializationProvider.getExtraJaxbClasses();
        if (!extraJaxbClasses.isEmpty()) {
          String extraJaxbClassesPropertyValue = JaxbSerializationProvider
              .classSetToCommaSeperatedString(extraJaxbClasses);
          msg.setStringProperty(SerializationConstants.EXTRA_JAXB_CLASSES_PROPERTY_NAME, extraJ
          msg.setStringProperty(SerializationConstants.DEPLOYMENT_ID_PROPERTY_NAME, deploymentI
        }

        // serialize request
        String xmlStr = serializationProvider.serialize(req);
        msg.writeUTF(xmlStr);
      } catch (JMSException jmse) {
          throw new RemoteCommunicationException("Unable to create and fill a
JMS message.", jmse);
      } catch (SerializationException se) {
            throw new RemoteCommunicationException("Unable to deserialze JMS
message.", se.getCause());
      }

      // send
      try {
        producer.send(msg);
      } catch (JMSException jmse) {
```

```
            throw new RemoteCommunicationException("Unable to send a JMS
message.", jmse);
     }

     // receive
     Message response;
     try {
       response = consumer.receive(timeout);
     } catch (JMSException jmse) {
        throw new RemoteCommunicationException("Unable to receive or retrieve
the JMS response.", jmse);
     }

     if (response == null) {
       logger.warn("Response is empty, leaving");
       return null;
     }
     // extract response
     assert response != null : "Response is empty.";
     try {
       String xmlStr = ((BytesMessage) response).readUTF();
       cmdResponses = (JaxbCommandsResponse) serializationProvider.deserialize(xmlStr);
     } catch (JMSException jmse) {
               throw new RemoteCommunicationException("Unable to extract
" + JaxbCommandsResponse.class.getSimpleName()
           + " instance from JMS response.", jmse);
     } catch (SerializationException se) {
               throw new RemoteCommunicationException("Unable to extract
" + JaxbCommandsResponse.class.getSimpleName()
           + " instance from JMS response.", se.getCause());
     }
     assert cmdResponses != null : "Jaxb Cmd Response was null!";
   } finally {
     if (connection != null) {
       try {
         connection.close();
         session.close();
       } catch (JMSException jmse) {
         logger.warn("Unable to close connection or session!", jmse);
       }
     }
   }
   return cmdResponses;
 }

 private InitialContext getRemoteJbossInitialContext(URL url, String user, String password) {
   Properties initialProps = new Properties();
   initialProps.setProperty(InitialContext.INITIAL_CONTEXT_FACTORY, "org.jboss.naming.remote.c
   String jbossServerHostName = url.getHost();
```

```
    initialProps.setProperty(InitialContext.PROVIDER_URL, "remote://"+ jbossServerHostName + ":
    initialProps.setProperty(InitialContext.SECURITY_PRINCIPAL, user);
    initialProps.setProperty(InitialContext.SECURITY_CREDENTIALS, password);

    for (Object keyObj : initialProps.keySet()) {
      String key = (String) keyObj;
      System.setProperty(key, (String) initialProps.get(key));
    }
    try {
      return new InitialContext(initialProps);
    } catch (NamingException e) {
                throw   new   RemoteCommunicationException("Unable   to   create
 " + InitialContext.class.getSimpleName(), e);
    }
  }
}
```

1. These classes can all be found in the `kie-services-client` and the `kie-services-jaxb` JAR.
2. The `JaxbCommandsRequest` instance is the "holder" object in which you can place all of the commands you want to execute in a particular request. By using the `JaxbCommandsRequest.getCommands()` method, you can retrieve the list of commands in order to add more commands to the request.

   A deployment id is required for command request messages that deal with business processes. Command request messages that only contain human task-related commands do not require a deployment id.
3. Note that the JMS message sent to the remote JMS API *must* be constructed as follows:

   - It must be a JMS byte message.
   - It must have a filled JMS Correlation ID property.
   - It must have an int property with the name of "serialization" set to an acceptable value (only 0 at the moment).
   - It must contain a serialized instance of a `JaxbCommandsRequest`, added to the message as a UTF string
4. The same serialization mechanism used to serialize the request message will be used to serialize the response message.
5. In order to match the response to a command, to the initial command, use the `index` field of the returned `JaxbCommandResponse` instances. This `index` field will match the index of the initial command. Because not all commands will return a result, it's possible to send 3 commands with a command request message, and then receive a command response message that only includes one `JaxbCommandResponse` message with an `index` value of 1. That 1 then identifies it as the response to the second command.

6 Since many of the results returned by various commands are not serializable, the jBPM JMS Remote API converts these results into JAXB equivalents, all of which implement the `JaxbCommandResponse` interface. The `JaxbCommandResponse.getResult()` method then returns the JAXB equivalent to the actual result, which will conform to the interface of the result.

For example, in the code above, the `StartProcessCommand` returns a `ProcessInstance`. In order to return this object to the requester, the `ProcessInstance` is converted to a `JaxbProcessInstanceResponse` and then added as a `JaxbCommandResponse` to the command response message. The same applies to the `List<TaskSummary>` that's returned by the `GetTaskAssignedAsPotentialOwnerCommand`.

However, not all methods that can be called on a normal `ProcessInstance` can be called on the `JaxbProcessInstanceResponse` because the `JaxbProcessInstanceResponse` is simply a representation of a `ProcessInstance` object. This applies to various other command response as well. In particular, methods which require an active (backing) `KieSession`, such as `ProcessInstance.getProess()` or `ProcessInstance.signalEvent(String type, Object event)` will throw an `UnsupportedOperationException`.

Part IV. Eclipse

How to use the Eclipse-based tooling

Chapter 18. jBPM Eclipse Plugin

18.1. jBPM Eclipse Plugin

The jBPM Eclipse plugin provides developers (and very technical users) with an environment to edit and test processes, and integrate it deeply with their applications. It provides the following features (on top of the Eclipse IDE):

- Wizards for creation of
  - a jBPM project
  - a BPMN2 process
- jBPM Perspective (showing the most commonly used views in a predefined layout)

18.1.1. Installation

The jBPM installer is capable of downloading and installing an Eclipse installation, including the Drools and jBPM Eclipse plugin (with a full jBPM runtime preconfigured) and the Eclipse BPMN2 Modeler.

Tip

Using the jBPM installer is definitely the recommended starting point for most users.

You can however also download and install the jBPM Eclipse Plugin manually. To do so, you need to:

- Download Eclipse (Kepler recommended, but older versions like Indigo or Juno should also still work)
- Start Eclipse
- Select "Install New Software ..." from the Help menu. Add the Drools and jBPM update site *http://downloads.jboss.org/jbpm/release/6.0.1.Final/updatesite/* [http://downloads.jboss.org/jbpm/release/6.0.1.Final/updatesite/]. You should see the plugins as shown below. Note that you can also download and unzip the Drools and jBPM update site to your local file system and use that as local update site instead.

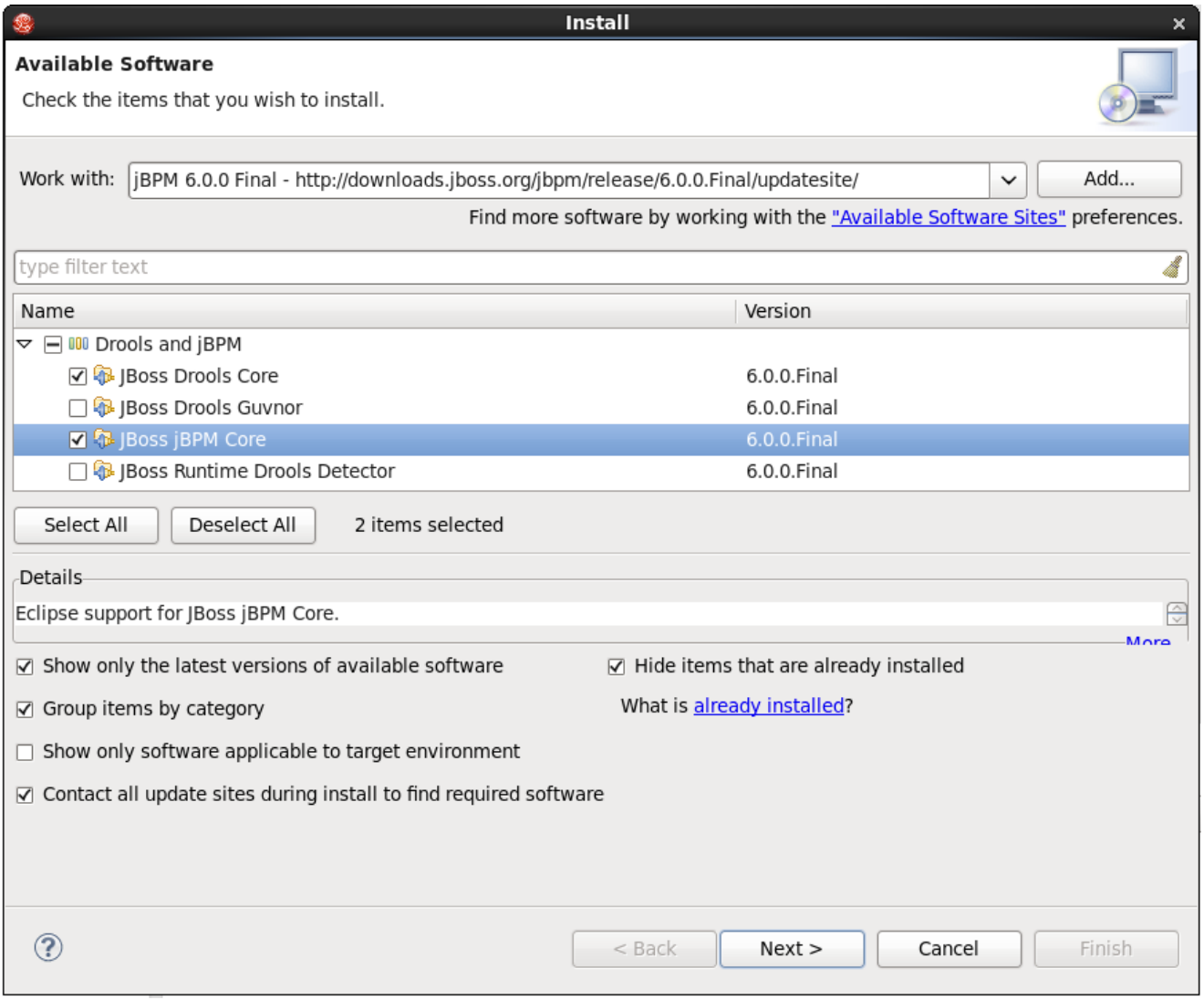


Figure 18.1.

Select the JBoss jBPM Core and JBoss Drools Core plugins and click "Next >". Click "Next >" again after reviewing your selecting, accept the terms of the license agreement and click "Finish" to download and install the plugins. If you get a warning about installing software that contains unsigned content, click OK. After successful installation, Eclipse should ask you to restart, click Yes.

- The plugin should now be installed. To check, check if you can for example see the new jBPM Project wizard: under the "File" menu, select "New Project ..." and there you should be able to see "New jBPM Project" under the jBPM category.
- Register a jBPM runtime to get started, see the section on jBPM runtimes in this chapter for more information.

Note that, when doing a manual install, you still need to manually install the Eclipse BPMN 2.0 Modeler plugin as well. Check out the chapter on the Eclipse BPMN 2.0 Modeler on how to do that.

18.1.2. jBPM Project Wizard

The aim of the new project wizard is to set up an executable sample project to start using processes immediately. This will set up a basic structure, the classpath, sample process and a test case to get you started. To create a new jBPM project, in the "File" menu select "New" and then "Project ..." and under the jBPM category, select "jBPM Project". A dialog as shown below should pop up.

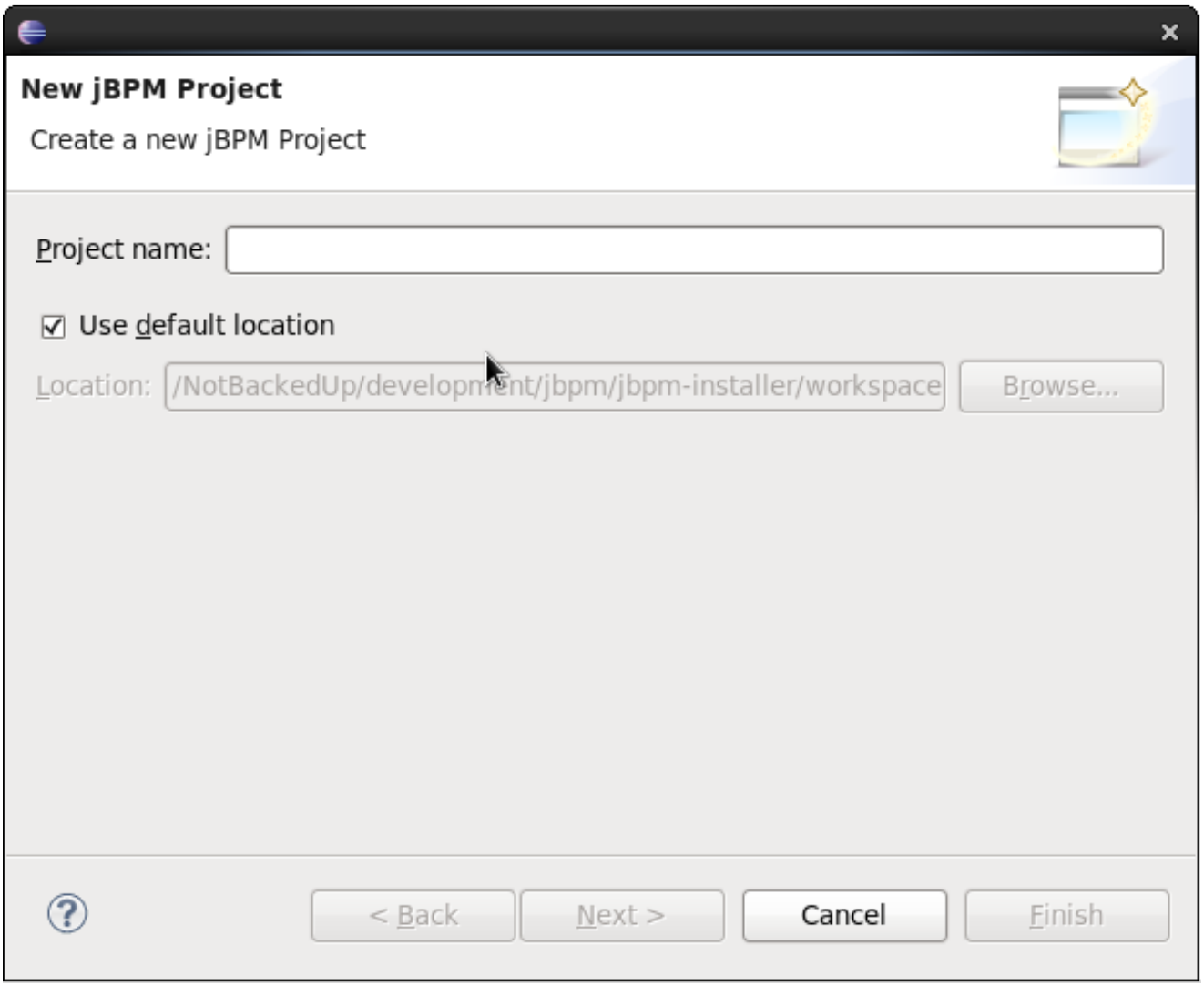


Figure 18.2.

Fill in a name for your project and if necessary change the location where this project should be located (by default Eclipse will generate it inside your Eclipse workspace folder) and click "Next >".

Now you can optionally include a sample process in your project to get started. You can select to either use a simple "Hello World" process, a slightly more advanced process including human tasks and persistence or simply an empty project. You can also select to include a JUnit test class that you can use to test your process. These can serve as a starting point, and will give you something executable almost immediately, which you can then modify to your needs.

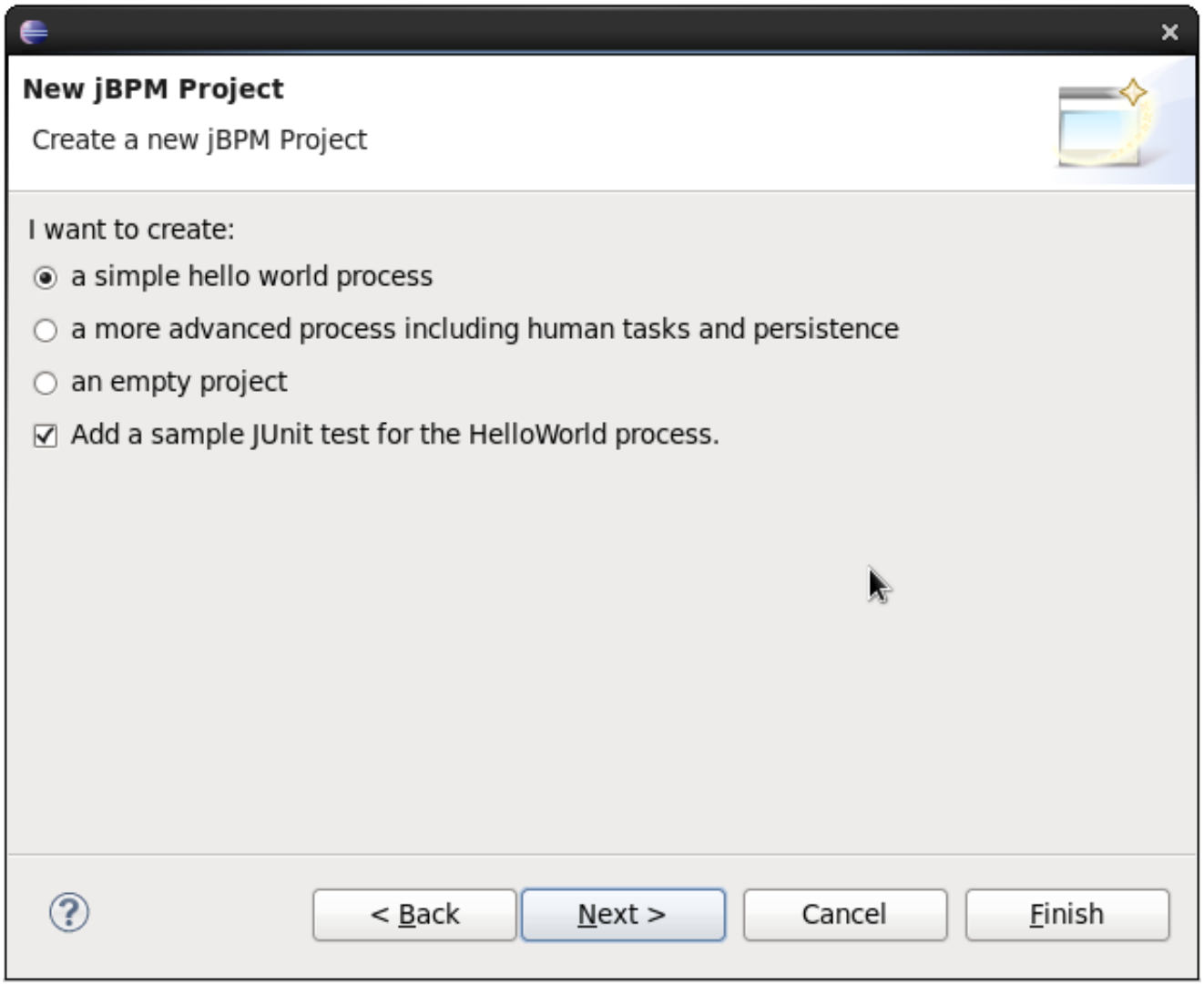


Figure 18.3.

Finally, the last page in the wizard allows you select a jBPM runtime, as shown below. You can either use the default runtime (as configured for you workspace, in your workspace preferences), or you can select a specific runtime for this project. For more information about runtimes and how to create them, see the section on jBPM runtimes in this chapter.

You can also select which version of jBPM you want to generate sample code for. By default it will generate an example using the latest jBPM 6.x API, but you could also generate examples using the old jBPM 5.x API. Note that you yourself are responsible for making sure that the code you generate can be understood by the runtime (for example, if you create an example using jBPM6 API but select a jBPM5 runtime, your sample will not compile). Also note that, if you want to execute a jBPM5 example on jBPM6, you will need to have the knowledge-api JAR inside your jBPM6 runtime, as this is responsible for the backwards compatibility of the jBPM5 API in jBPM6.

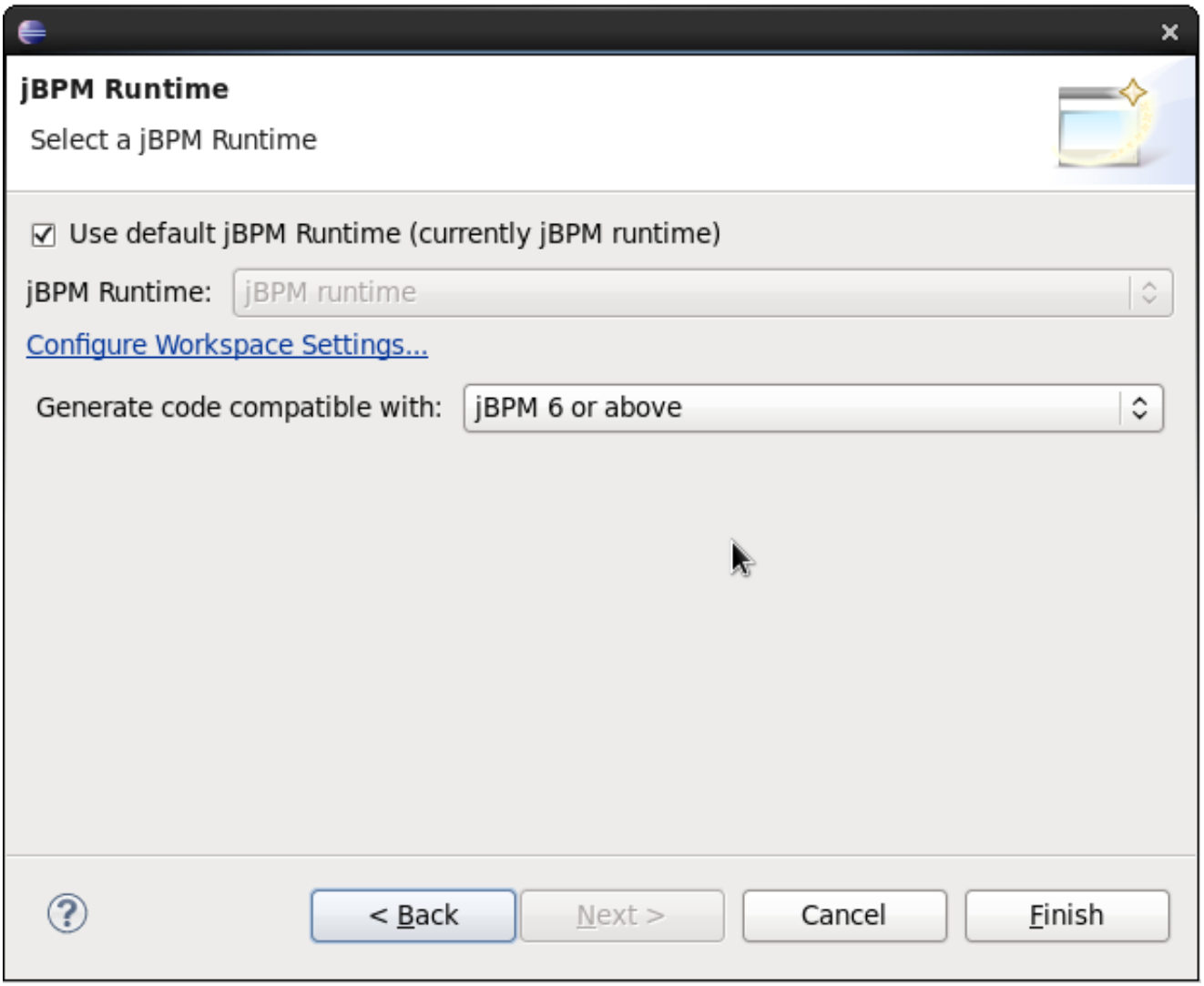


Figure 18.4.

When you selected the simple 'hello world' example, the result is shown below. Feel free to experiment with the plug-in at this point.

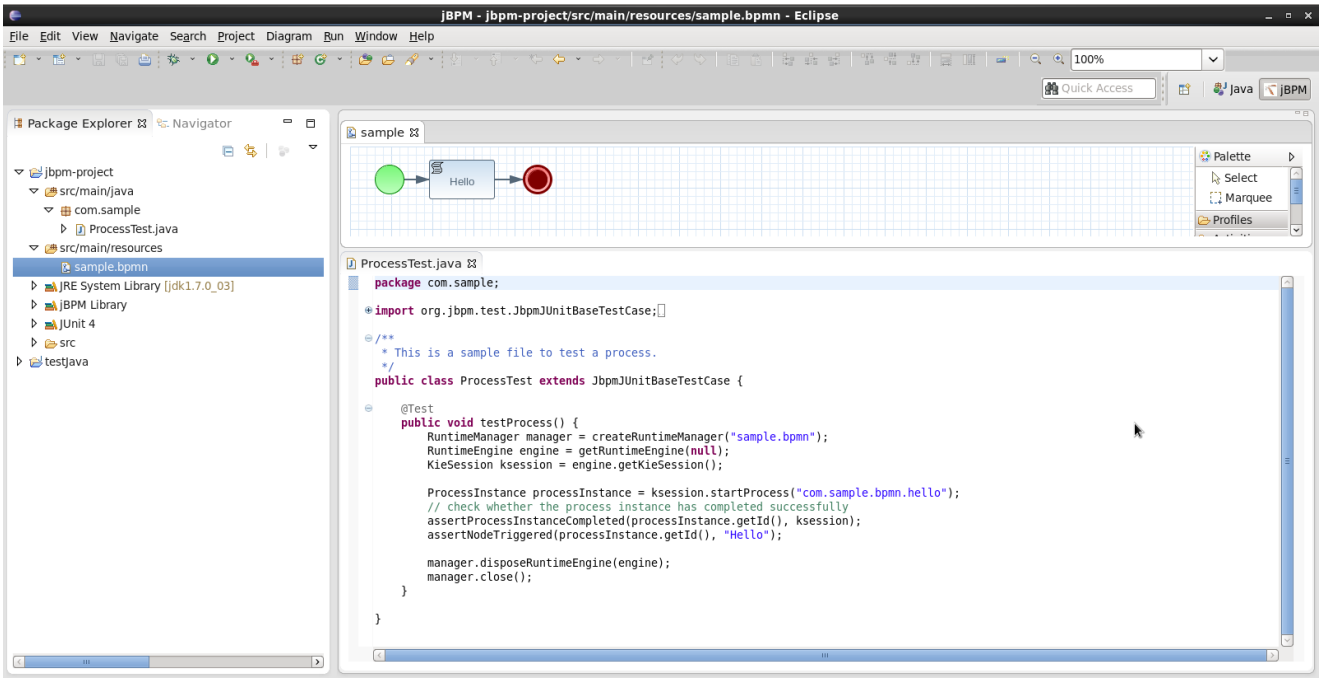


Figure 18.5. New jBPM project artifacts

The newly created project contains an example process file (sample.bpmn) in the src/main/resources directory and an example Java file (ProcessTest.java) that can be used to test the process in a jBPM engine. You'll find this in the folder src/main/java, in the com.sample package. All the other JARs that are necessary during execution are also added to the classpath in a custom classpath container called jBPM Library.

You can also convert an existing Java project to a jBPM project by selecting the "Convert to jBPM Project" action. Right-click the project you want to convert and under the "Configure" category (at the bottom) select "Convert to jBPM Project". This will add the jBPM Library to your project's classpath.

18.1.3. New BPMN2 Process Wizard

You can create a new process simply as an empty text file with extension ".bpmn", or use the "New BPMN2 Process" wizard to do so. To create a new process, in the "File" menu select "New" and then "Other ..." and under the jBPM category, select "BPMN2 Process" and click "Next >". In the next dialog, you should select the folder where the process should be created (for example the src/main/resources folder of your project) and a name for the process. Clicking "Finish" should create your new process (by default it should only contain one start node) and open it so you can start editing it.

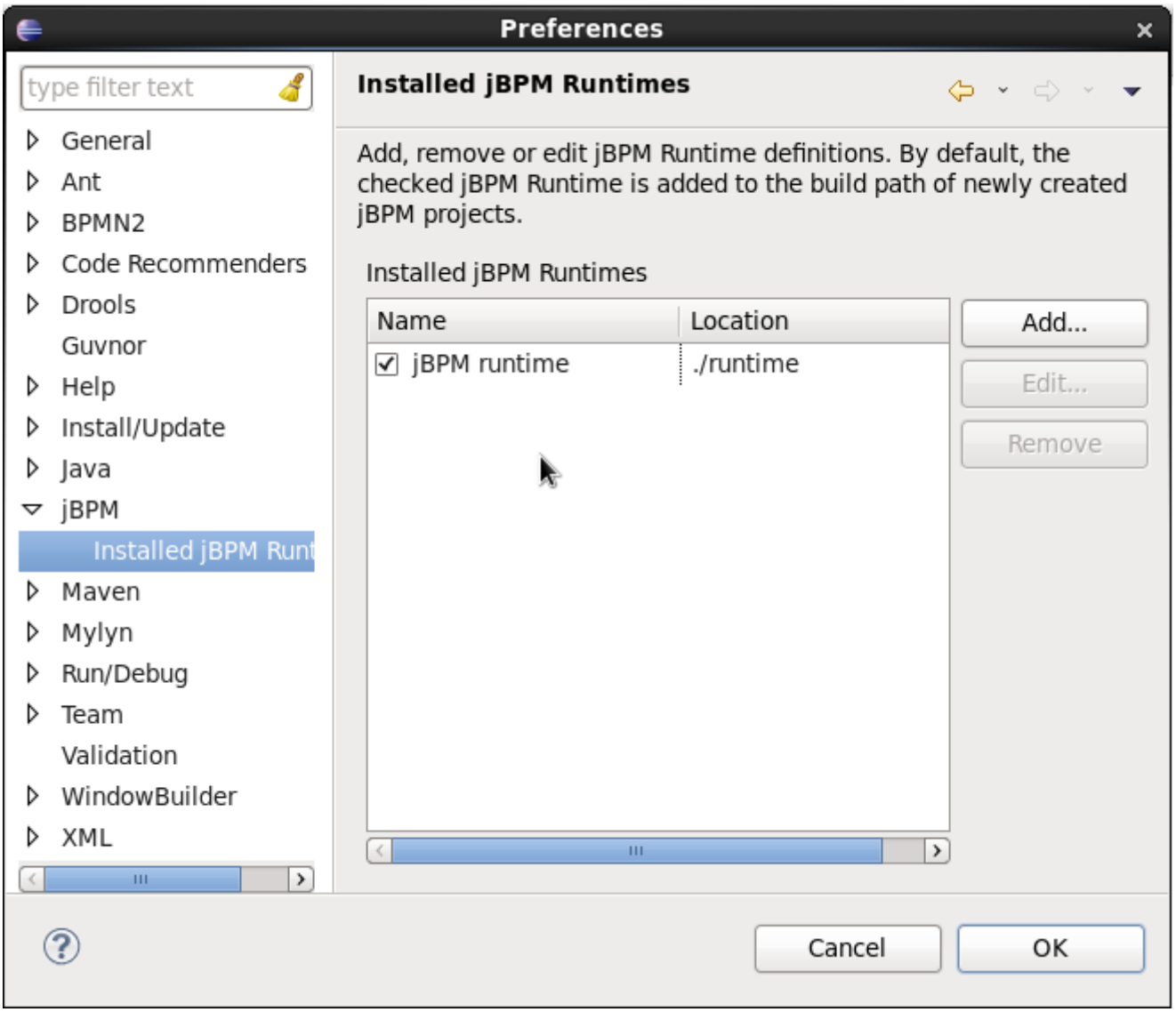
18.1.4. jBPM Runtime

A jBPM runtime is a collection of JAR files that represent one specific release of the jBPM project JARs. To create a runtime, download the binary distribution of the version of jBPM you want to

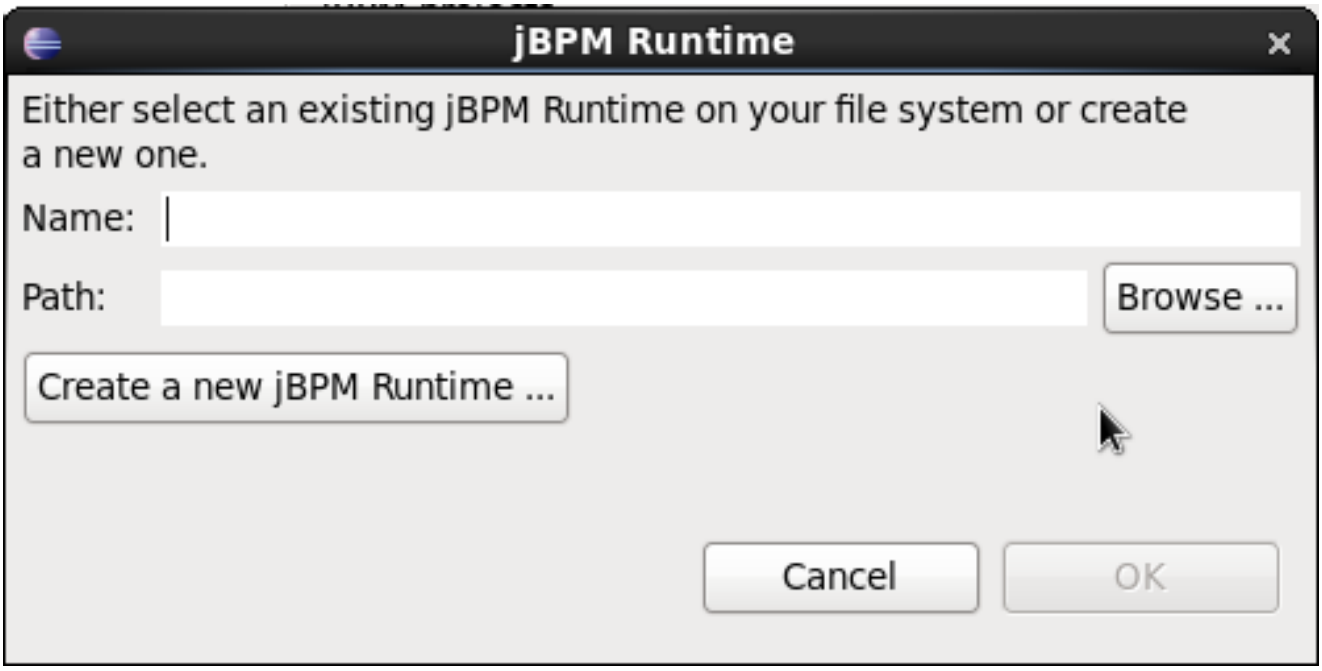
use and unzip on your local file system. You must then point the IDE to the release of your choice by selecting the folder where these JARs are located. If you want to create a new runtime based on the latest jBPM project JARs included in the plugin itself, you can also easily do that. You are required to specify a default jBPM runtime for your Eclipse workspace, but each individual project can override the default and select the appropriate runtime for that project specifically.

18.1.4.1. Defining a jBPM Runtime

To define one or more jBPM runtimes using the Eclipse preferences view you open up your Preferences, by selecting the "Preferences" menu item in the menu "Window". A "Preferences" dialog should show all your settings. On the left side of this dialog, under the jBPM category, select "Installed jBPM runtimes". The panel on the right should then show the currently defined jBPM runtimes. For example, if you used the jBPM Installer, it should look like the figure below.



To define a new jBPM runtime, click on the "Add" button. A dialog such as the one shown below should pop up, asking for the name of your runtime and the location on your file system where it can be found.



In general, you have two options:

1. If you simply want to use the default JAR files as included in the jBPM Eclipse plugin, you can create a new jBPM runtime automatically by clicking the "Create a new jBPM Runtime ..." button. A file browser will show up, asking you to select the folder on your file system where you want this runtime to be created. The plugin will then automatically copy all required dependencies to the specified folder. Make sure to select a unique name for the newly created runtime and click "OK" to register this runtime.

   Tip

   Note that creating a jBPM runtime from the default JAR files as included in the jBPM Eclipse plugin is only recommended to get you started the first time and for very simple use cases. The runtime that is created this way only contains the minimal set of JARs, and therefore doesn't support a significant set of features, including for example persistence. Make sure to create a full runtime (using the second approach) for real development.

2. If you want to use one specific release of the jBPM project, you should create a folder on your file system that contains all the necessary jBPM libraries and dependencies (for example by downloading the binary distribution and unzipping it on your local file system). Instead of creating a new jBPM runtime as explained above, give your runtime a unique name and click the "Browse ..." button to select the location of this folder containing all the required JARs. Click "OK" to register this runtime.

After clicking the OK button, the runtime should show up in your table of installed jBPM runtimes, as shown below. Click on the checkbox in front of one of the installed runtimes to make it the

default jBPM runtime. The default jBPM runtime will be used as the runtime of all your new jBPM projects (in case you didn't select a project-specific runtime).

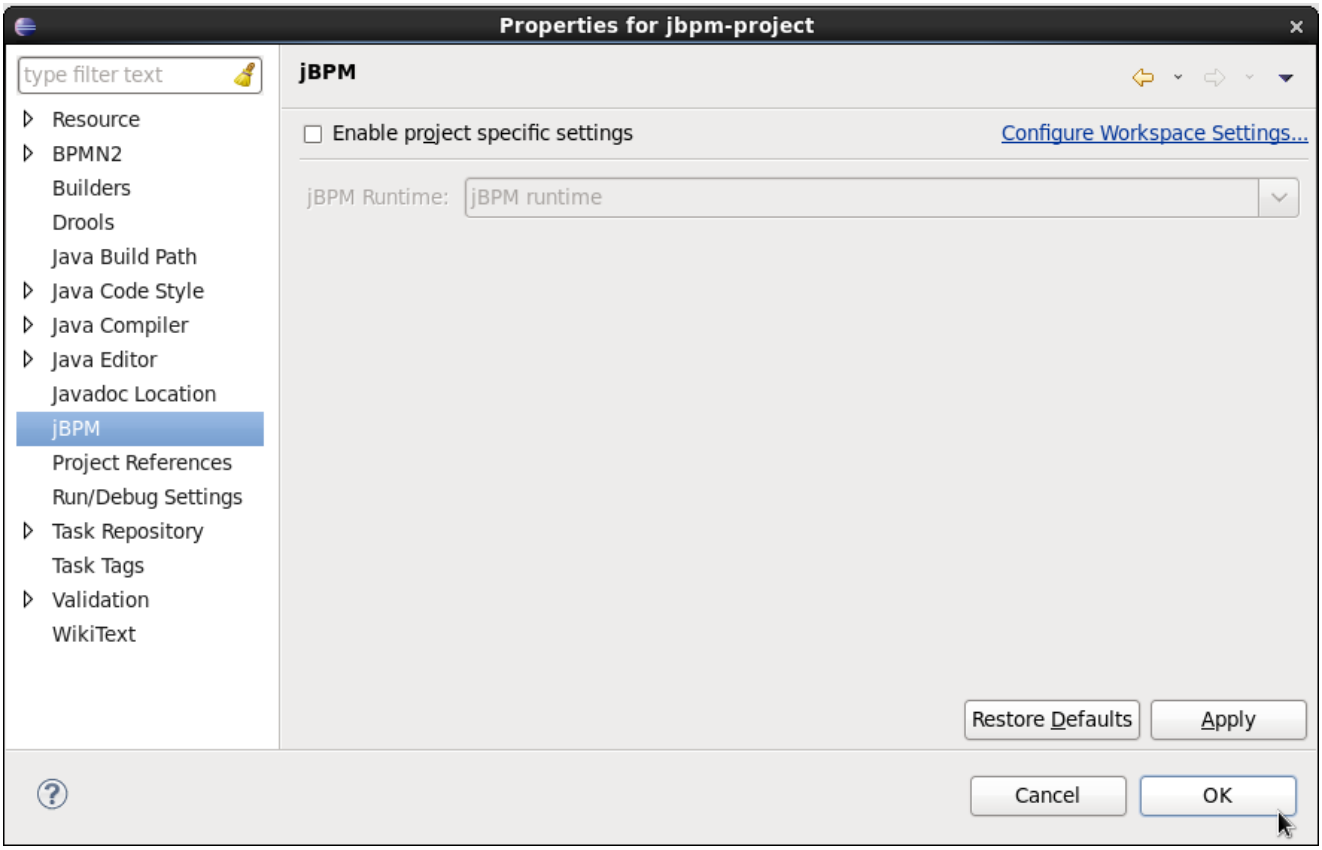
You can add as many jBPM runtimes as you need. Note that you will need to restart Eclipse if you changed the default runtime and you want to make sure that all the projects that are using the default runtime update their classpath accordingly.

18.1.4.2. Selecting a runtime for your jBPM project

Whenever you create a jBPM project (using the New jBPM Project wizard or by converting an existing Java project to a jBPM project), the plugin will automatically add all the required JARs to the classpath of your project.

When creating a new jBPM project, the plugin will automatically use the default Drools runtime for that project, unless you specify a project-specific one. You can do this in the final step of the New jBPM Project wizard, as shown below, by deselecting the "Use default Drools runtime" checkbox and selecting the appropriate runtime in the drop-down box. If you click the "Configure workspace settings ..." link, the workspace preferences showing the currently installed jBPM runtimes will be opened, so you can add new runtimes there.

You can change the runtime of a jBPM project at any time by opening the project properties and selecting the jBPM category, as shown below. Mark the "Enable project specific settings" checkbox and select the appropriate runtime from the drop-down box. If you click the "Configure workspace settings ..." link, the workspace preferences showing the currently installed jBPM runtimes will be opened, so you can add new runtimes there. If you deselect the "Enable project specific settings" checkbox, it will use the default runtime as defined in your global workspace preferences.



18.1.5. jBPM Maven Project Wizard

The aim of the new Maven project wizard is to set up an executable sample project to start using processes immediately (but not as normal Java project with all jBPM dependencies added using a jBPM library but by using Maven (and thus a pom.xml) to define your project's properties and dependencies. This wizard will set up a Maven project using a pom.xml, and include a sample process and Java class to execute it. To create a new jBPM Maven project, in the "File" menu select "New" and then "Project ..." and under the jBPM category, select "jBPM Project (Maven)". Give your project a name and click finish. The result should be as shown below.

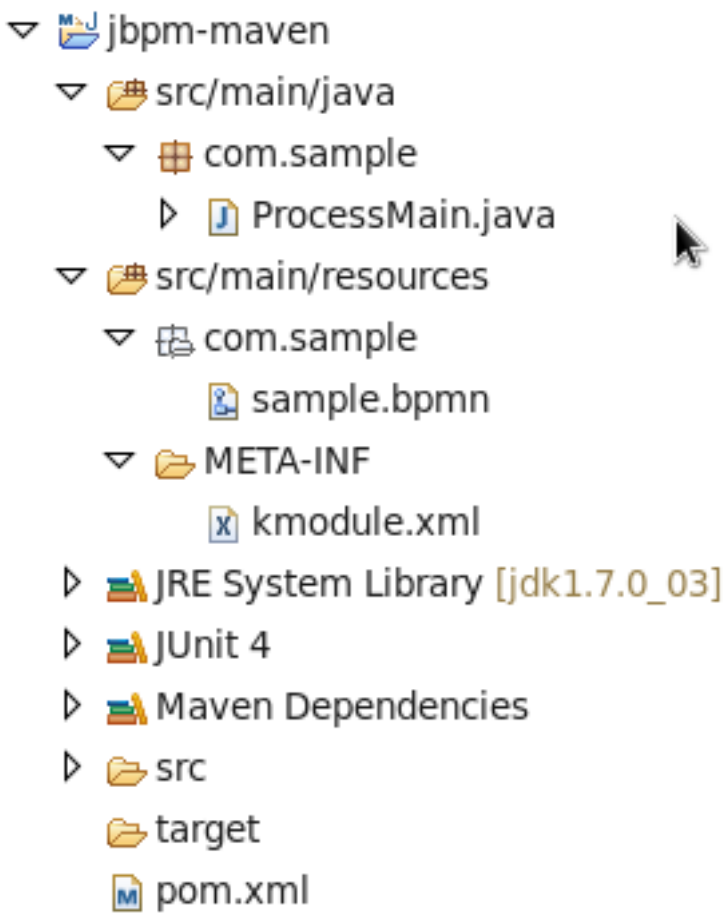


Figure 18.6.

The pom.xml that is generated for your project contains the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xsi:schemaLocation="http://maven.apache.org/POM/4.0.0  http://
maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>

  <groupId>com.sample</groupId>
  <artifactId>jbpm-example</artifactId>
  <version>1.0.0-SNAPSHOT</version>

  <name>jBPM :: Sample Maven Project</name>
  <description>A sample jBPM Maven project</description>

  <properties>
    <jbpm.version>6.0.0.Final</jbpm.version>
```

```
  </properties>

  <repositories>
    <repository>
      <id>jboss-public-repository-group</id>
      <name>JBoss Public Repository Group</name>
      <url>http://repository.jboss.org/nexus/content/groups/public/</url>
      <releases>
        <enabled>true</enabled>
        <updatePolicy>never</updatePolicy>
      </releases>
      <snapshots>
        <enabled>true</enabled>
        <updatePolicy>daily</updatePolicy>
      </snapshots>
    </repository>
  </repositories>

  <dependencies>
    <dependency>
      <groupId>org.jbpm</groupId>
      <artifactId>jbpm-test</artifactId>
      <version>${jbpm.version}</version>
    </dependency>
  </dependencies>
</project>
```

In the properties section, you can specify which version of jBPM you would like to use (by default it uses 6.0.0.Final). It adds the JBoss Nexus Maven repository (where all the jBPM JARs and their dependencies are located) to your project and configures the dependencies.

Note

By default, only the jbpm-test JAR is specified as a dependency, as this has transitive dependencies to almost all of the core dependencies you will need. You are free to update the dependencies section however to include only the dependencies you need.

The project also contains a sample process, under src/main/resources, in the com.sample package, and a kmodule.xml configuration file under the META-INF folder. The kmodule.xml defines which resources (processes, rules, etc.) are to be loaded as part of your project. In this case, it is defining a kbase called "kbase" that will load all the resources in the com.sample folder:

```
<kmodule xmlns="http://jboss.org/kie/6.0.0/kmodule">
  <kbase name="kbase" packages="com.sample"/>
```

```
</kmodule>
```

Finally, it also contains a Java class that can be used to execute the sample process. It will first create a kbase called "kbase" (by inspecting the kmodule.xml file and thus loading the `sample.bpmn` process) and then use a `RuntimeManager` to get access to a `KieSession` and `TaskService`. In this case, it is used to start a process and then complete the tasks created by this process one by one.

18.1.6. Drools Eclipse plugin

The Drools Eclipse Plugin, which is bundled as part of the same Eclipse Update Site as the jBPM Eclipse Plugin, provides similar features for creating and editing business rules, and execute them using the Drools engine. This for example allows you to create and edit .drl files containing business rules. You can combine your processes and rules inside one project and execute them together on the same KieSession.

18.2. Debugging

This section describes how to debug processes using the jBPM Eclipse plugin. This means that the current state of your running processes can be inspected and visualized during the execution. Note that we currently don't allow you to put breakpoints on the nodes within a process directly. You can however put breakpoints inside any Java code you might have (i.e. your application code that is invoking the engine or invoked by the engine, listeners, etc.) or inside rules (that could be evaluated in the context of a process). At these breakpoints, you can then inspect the internal state of all your process instances.

When debugging the application, you can use the following debug views to track the execution of the process:

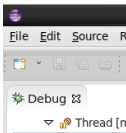
1. The process instances view, showing all running process instances (and their state). When double-clicking a process instance, the process instance view visually shows the current state of that process instance at that point in time.
2. The audit view, showing the audit log (note that you should probably use a threaded file logger if you want to session to save the audit event to the file system on regular intervals, so the audit view can be update to show the latest state).
3. The global data view, showing the globals.
4. Other views related to rule execution like the working memory view (showing the contents (data) in the working memory related to rule execution), the agenda view (showing all activated rules), etc.

18.2.1. The Process Instances View

The process instances view shows the process instances currently running in the selected ksession. To be able to use the process instances view, first open the Process Instances view

(Window - Show View - Other ... and under the Drools category select Process Instances and Process Instance). Tip: it might be useful to drag the Process Instance view to the Outline View and slightly enlarge it, as shown in the screenshot below, so you can see both the Process Instances and Process Instance views at the same time.

Next, use a (regular) Java breakpoint to stop your application at a specific point (for example right after starting a new process instance). In the Debug perspective, select the ksession you would like to inspect, and the Process Instances view should show the process instances that are currently active inside that ksession. For example, the screenshot below shows one running process instance (with id "1"). When double-clicking a process instance, the process instance viewer will graphically show the progress of that process instance. An example where the process instance is waiting for a human actor to perform "Task 1" is shown below.



Note

The process instances view shows the process instances currently active inside the selected ksession. Note that, when using persistence, process instances are not kept in memory inside the ksession, as they are stored in the database as soon as the command completes. Therefore, you will not be able to use the Process Instances view when using persistence. For example, when executing a JUnit test using the JbpmJUnitBaseTestCase, make sure to call "super(true, false);" in the constructor to create a runtime manager that is not using persistence.

Tip

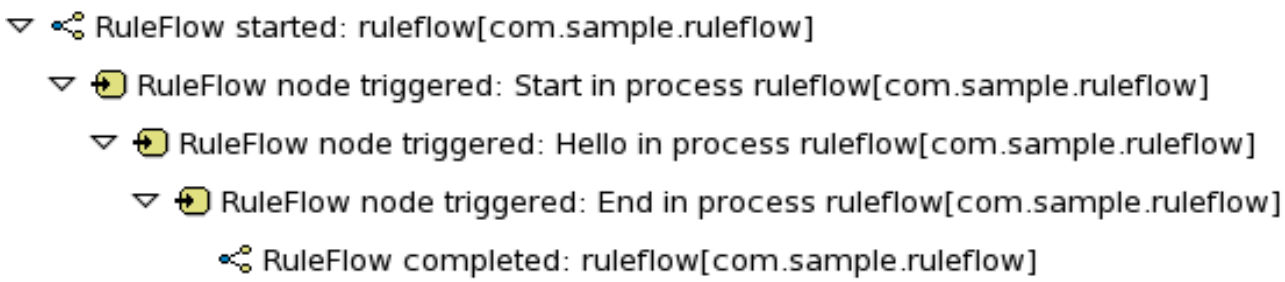
When you double-click a process instance in the process instances view and the process instance view complains that it cannot find the process, this means that the plugin wasn't able to find the process definition of the selected process instance in the cache of parsed process definitions. To solve this, simply change the process definition in question and save again (so it will be parsed) or rebuild the project that contains the process definition in question.

18.2.2. The Audit View

The audit view can be used to show the all the events inside an audit log in a tree-based manner. An audit log is an XML-based log file which contains a log of all the events that occurred while executing a specific ksession. To create a logger, use KieServices to create a new logger and attach it to a ksession. Be sure to close the logger after usage.

```
KieRuntimeLogger logger = KieServices.Factory.get().getLoggers()
    .newThreadedFileLogger(ksession, "mylogfile", 1000);
// do something with the ksession here
logger.close();
```

To be able to use the Audit View, first open it (Window - Show View - Other ... and under the Drools category select Audit). To open up a log file in the audit view, open the selected log file in the audit view (using the "Open Log" action in the top right corner), or simply drag and drop the log file from the Package Explorer or Navigator into the audit view. A tree-based view is generated based on the data inside the audit log. An event is shown as a subnode of another event if the child event is caused by (a direct consequence of) the parent event. An example is shown below.



Tip

Note that the file-based logger will only save the events on close (or when a certain threshold is reached). If you want to make sure the events are saved on a regular interval (for example during debugging), make sure to use a threaded file logger,

so the audit view can be update to show the latest state. When creating a threaded file logger, you can specify the interval after which events should be saved to the file (in milliseconds).

18.3. Synchronizing with Workbench Repositories

From Eclipse, you can synchronize your local workspace with one or more repositories that are managed inside the workbench application. This enables collaboration between developers using Eclipse and users of the web-based workbench (business analysts or end users for example). Synchronization between the workbench repositories and your local version of these projects is done using Git (a popular distributed source code version control system).

When creating and executing processes inside Eclipse, you are creating them on your local file system. You can however also import an existing repository from the Workbench, apply changes and push these changes back into the Workbench repositories. We are using existing Git tools for this. Note that this section will describe how to do this using the EGit tooling (Eclipse Tooling for Git which comes by default with most versions of Eclipse), but feel free to use your preferred Git tool instead.

Note

This section is not intended to explain what Git is, or how to use EGit, in detail. If you don't have any experience with Git and/or EGit, it might be recommended to read up on them first if necessary.

18.3.1. Importing a workbench repository

To import an existing repository from the workbench, you can use the EGit import wizard. In the File menu, select "Import ..." and in the Git category, select "Projects from Git" and click "Next >". This should open a new dialog where you should select the location of the repository you would like to import. Since we are connecting to a repository that is managed by the workbench application, select "URI" and click "Next >" once more.

Use the following URI to connect to your workbench repositories:

```
ssh://<hostname>:8001/<repository_name>
```

For example, if you are running the workbench application on your local host (for example by using the jbpm-installer), and you want to import the jbpm-playground repo, use the following URI:

```
ssh://localhost:8001/jbpm-playground
```

Note that you can change the port that is used by the server to provide ssh access to the git repository if necessary, using the system property `org.uberfire.nio.git.ssh.port`

Fill in the URI of the repository you would like to import, as for example shown below, and click "Next >".

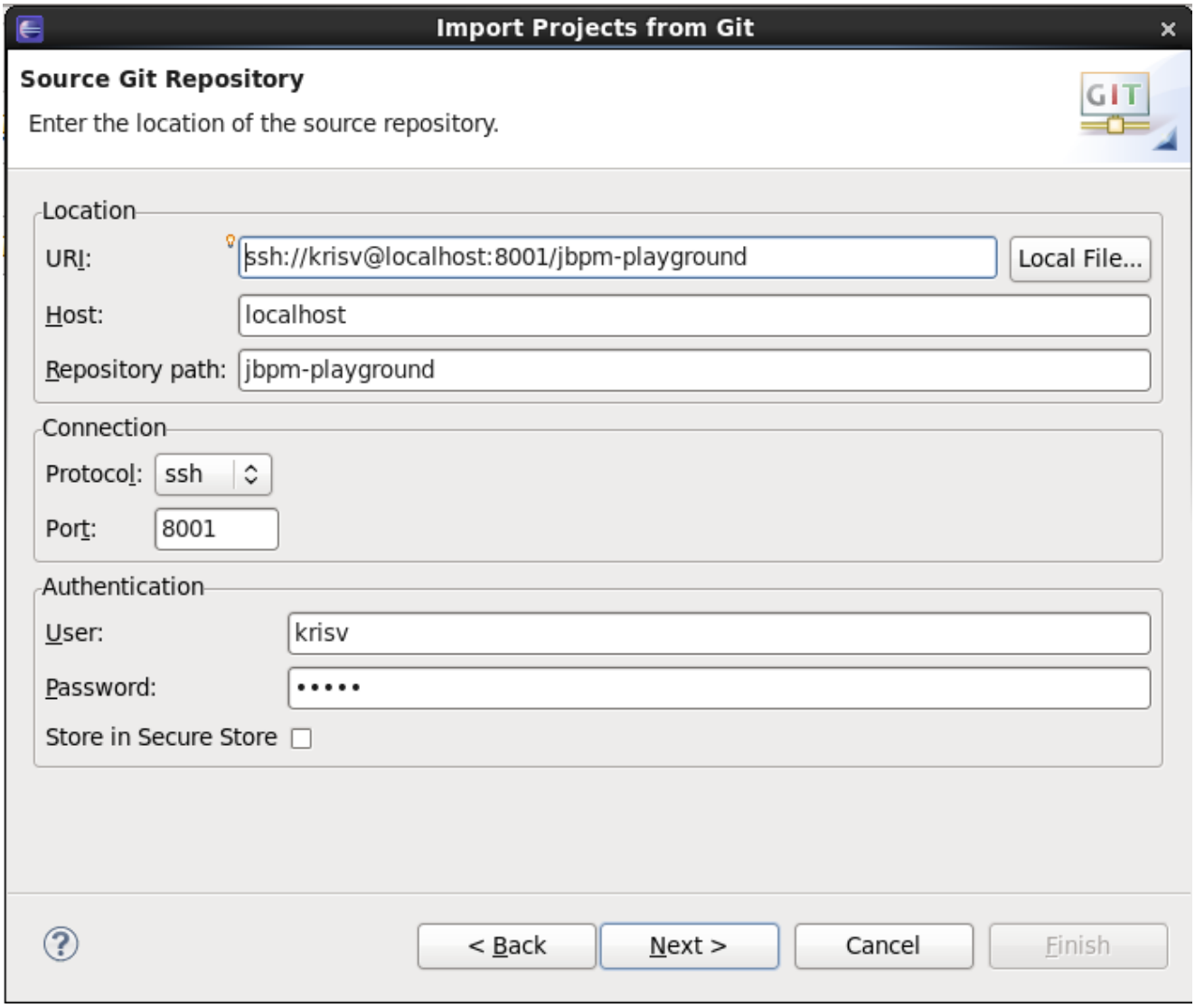


Figure 18.7.

You will be asked to select which branch you would like to import. Select the master branch and click "Next >" again.

Finally, you need to specify where on your local file system you would like this repository to be created. Fill in the directory (you can use the Browse button to select the folder in question, and if necessary you can create a new folder there as well) and click "Next >". This will now download the repository to the folder you just selected.

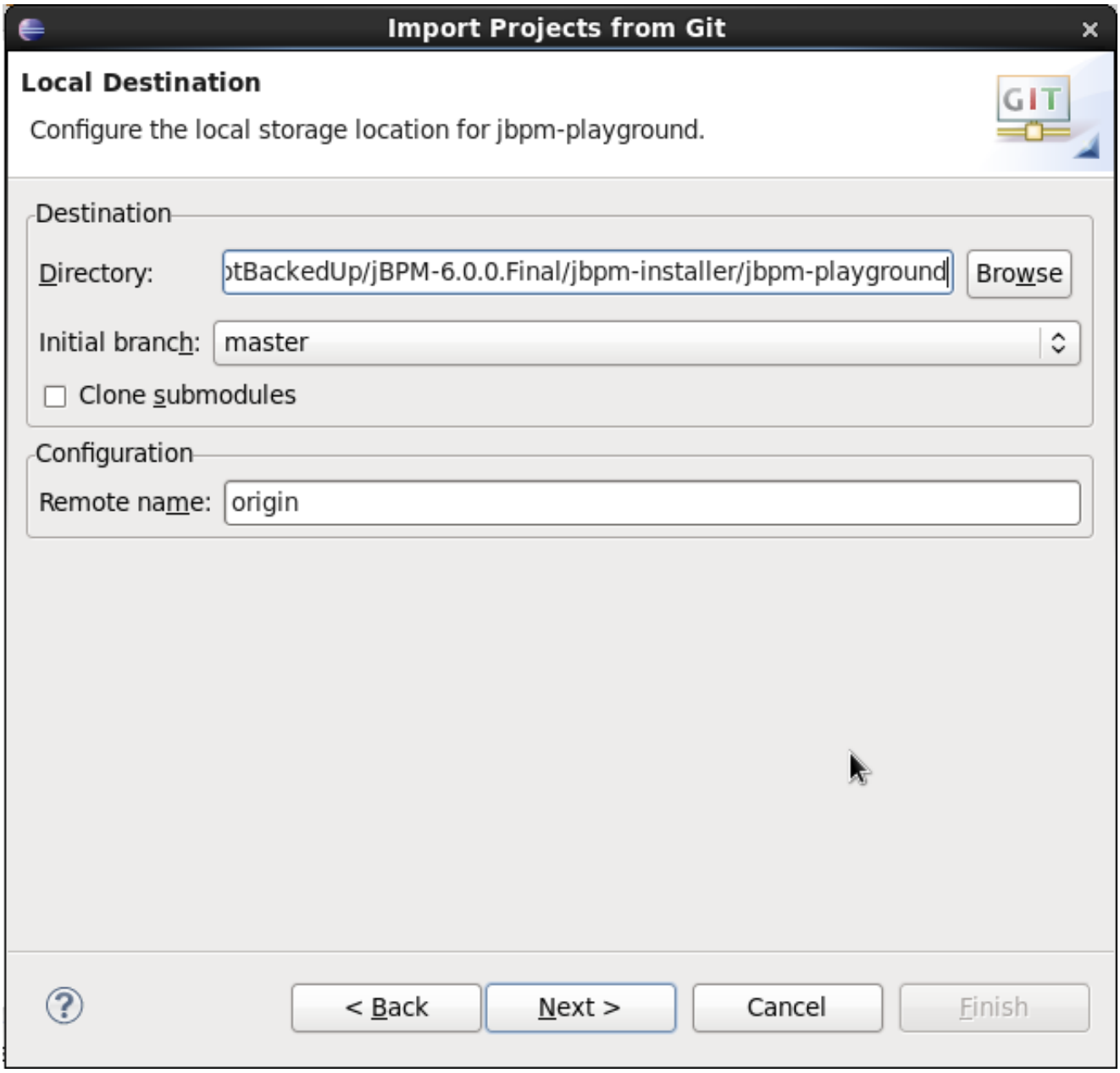


Figure 18.8.

You still need to import the repository you just downloaded as a project in your Eclipse workspace. Select "Import as general project" and after clicking "Next >", give it a name and click "Finish". After doing so, your workspace should now contain your repository, and you should be able to browse, open and edit the various assets inside.

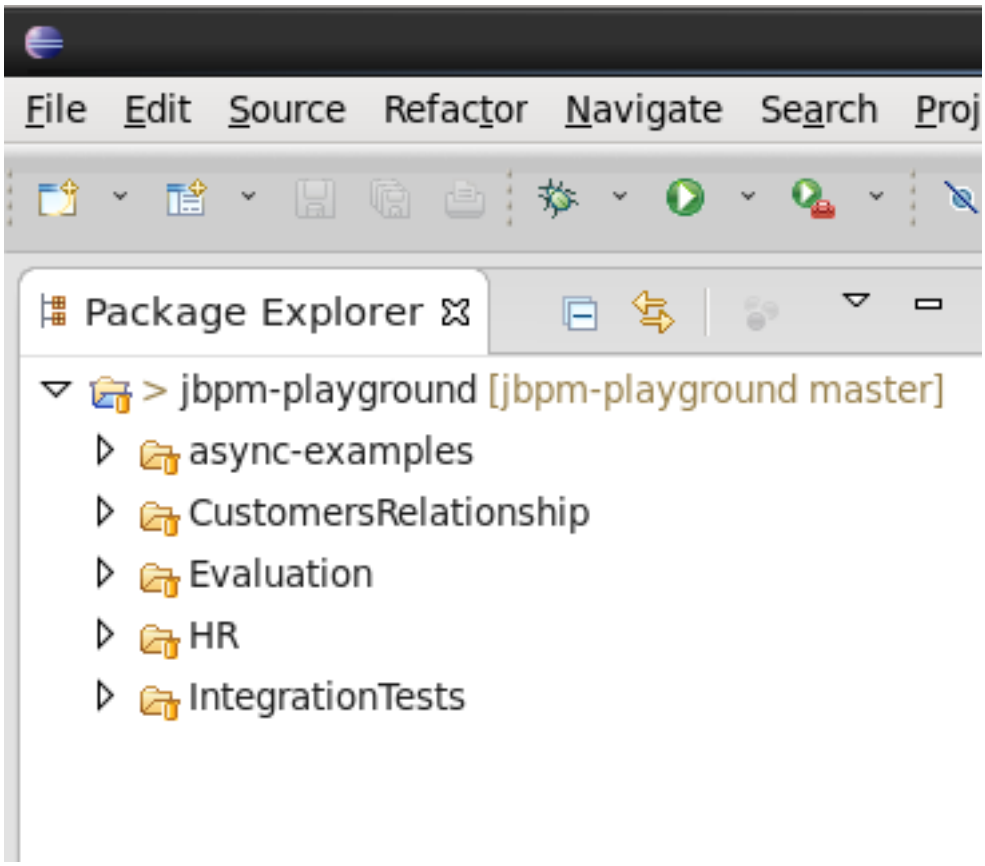


Figure 18.9.

18.3.2. Committing changes to the workbench

You can commit and push changes (you do locally) back to the workbench repositories. To commit changes, right-click on your repository project and select "Team -> Commit ...". A new dialog pops up, showing all the changes you have on your local file system. Select the files you want to commit (if you double-click them, you can get an overview of the changes you did for that file), provide an appropriate commit message and click "Commit".

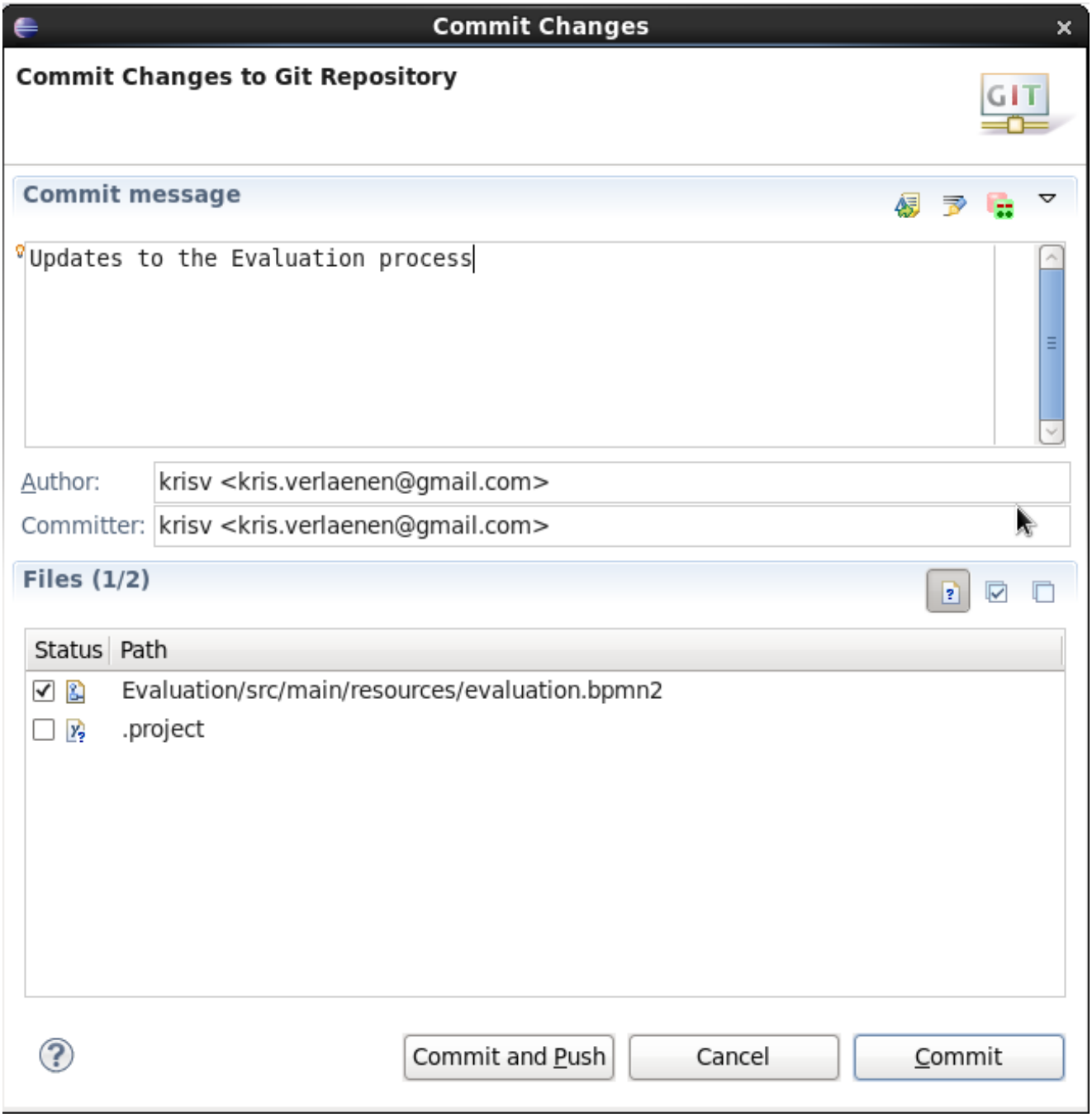


Figure 18.10.

Once you've committed your change to your local git, you still need to push it to the workbench repository. Right-click your project again, and select "Team -> Push to Upstream".

Note

You are only allowed to push changes upstream if your local version includes all recent changes (otherwise you might be overriding someone else's changes).

You might be forced to update (and if necessary resolve conflicts) before you are allowed to commit any changes.

18.3.3. Updating from to the workbench

To retrieve the latest changes from the workbench repository, right-click your repository project and select "Team -> Fetch from Upstream". This will fetch all changes from the workbench repository, but not yet apply them to your local version. Now right-click your project again and select "Team -> Merge ...". In the dialog that pops up next, you need to select "origin/master" branch (under Remote Tracking) to indicate that you want to merge in all changes from the original repository in the workbench, and click "Merge".

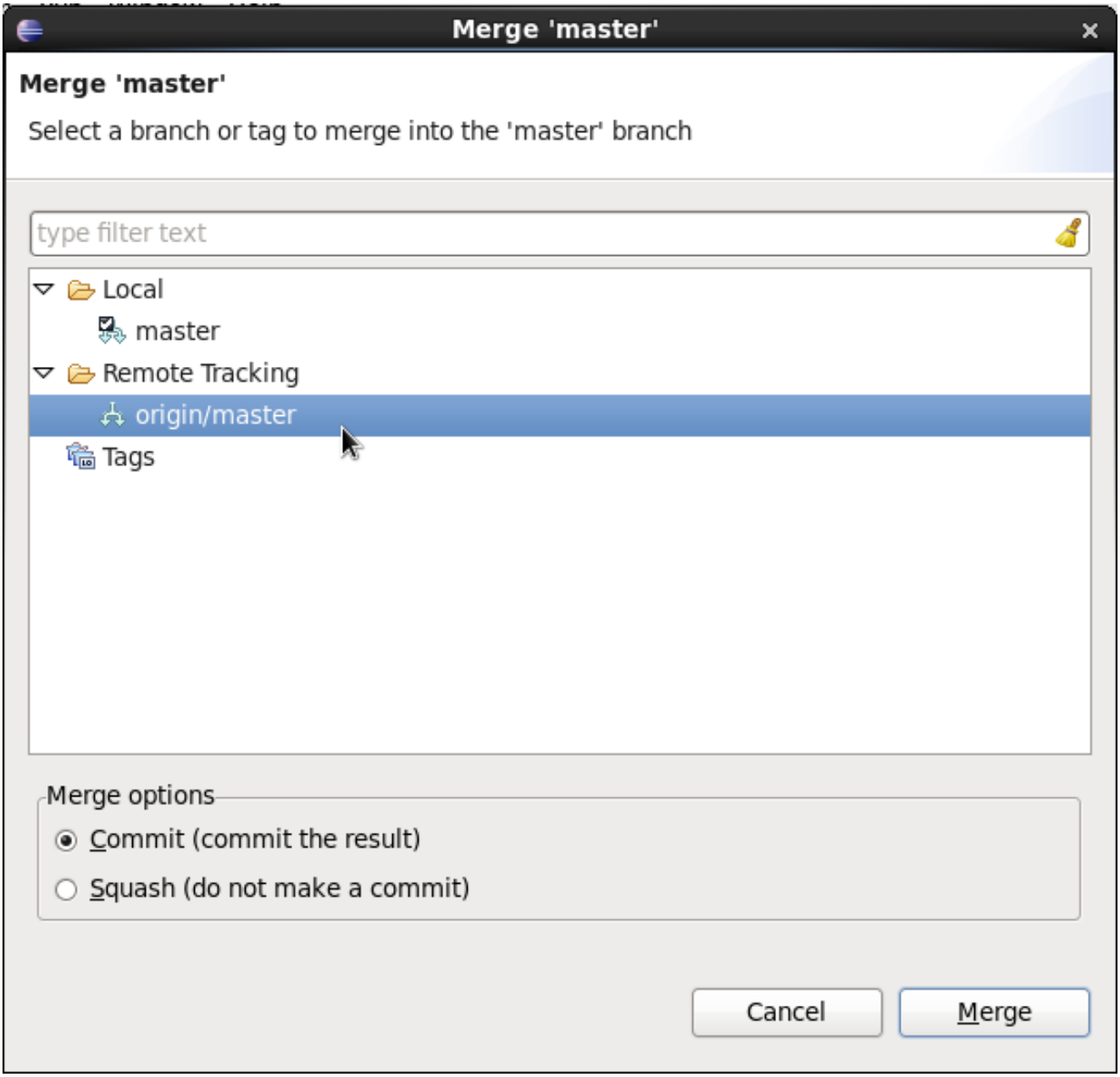


Figure 18.11.

Note

It is possible that you have committed and/or conflicting changes in your local version, you might have to resolve these conflicts and commit the merge results before you will be able to complete the merge successfully. It is recommended to update regularly, before you start updating a file locally, to avoid merge conflicts being detected when trying to commit changes.

18.3.4. Working on individual projects

When you import a repository, it will download all the projects that are inside that repository. It is however useful to mount one specific project as a separate Java project in Eclipse. When you do this, Eclipse will be able to interpret the information in the project pom.xml file (that you created in the workbench), download and include any dependencies you specified, compile any Java classes you have in your project (that you for example created with the data modeler), etc.

To do so, right-click on one of the projects in your repository project and select "Import ..." and under the Maven category, select "Existing Maven Projects" (as shown below) and click Next.

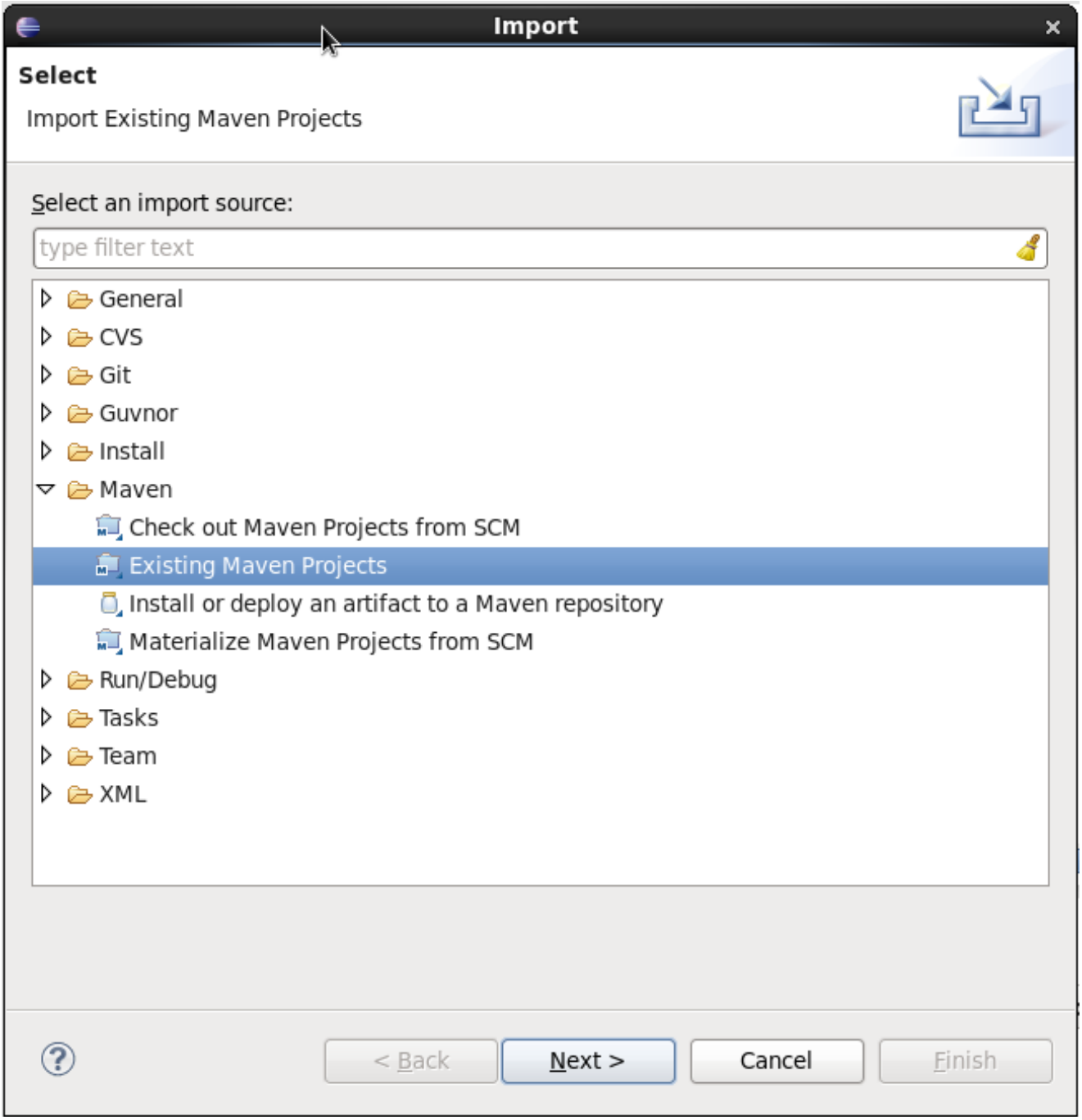


Figure 18.12.

In the next page, you should see the pom.xml of the project you selected. Click Finish.

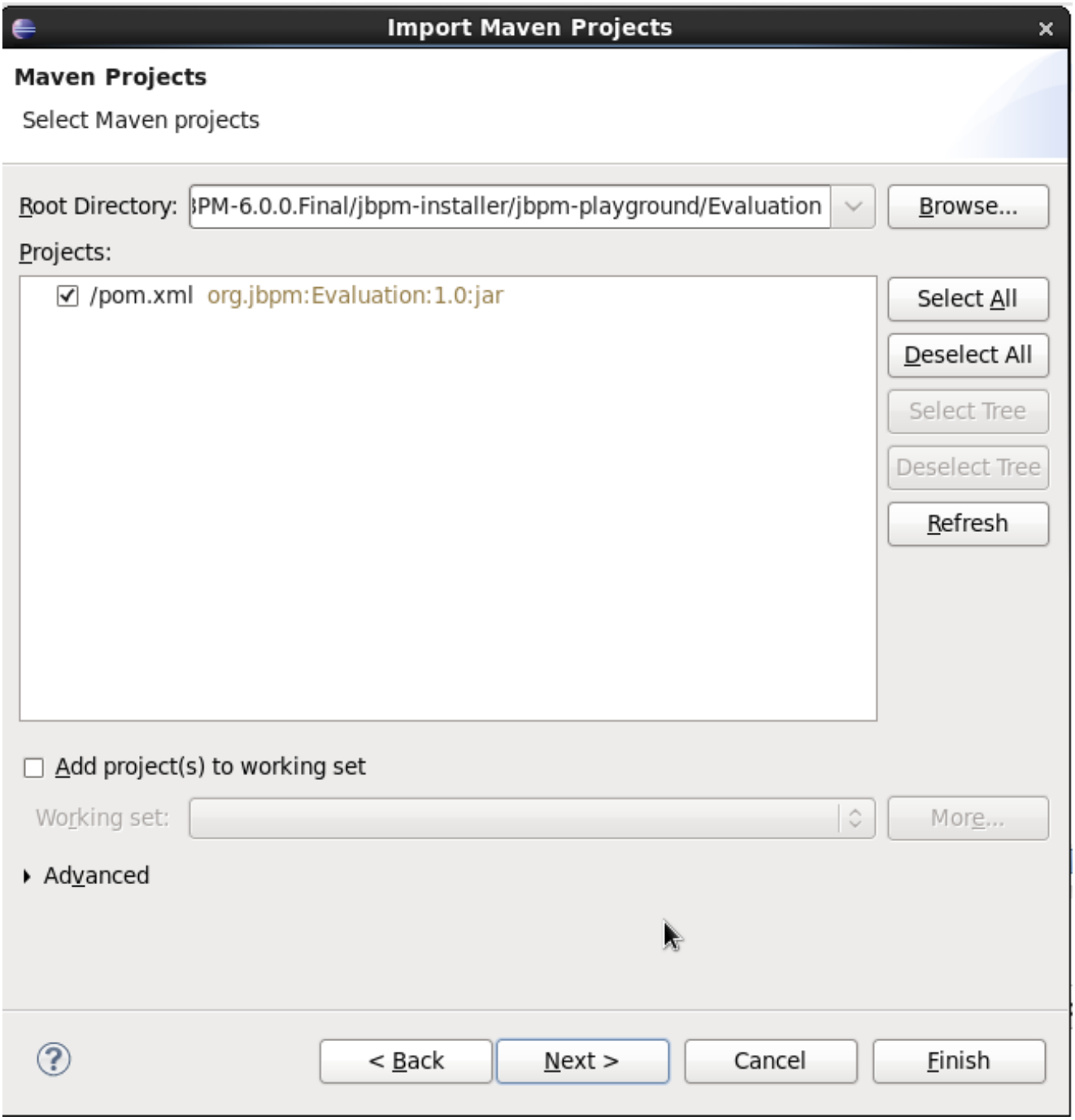


Figure 18.13.

If your project requires some of the jBPM libraries to correctly compile and/or execute any Java classes in your project (for example if you have test classes in your project that start up a jBPM engine and execute some tests for your project, or if you are using the data modeler, which will add some annotations to the generated Java classes), you still need to add the jBPM libraries to the classpath of your project. To do so, simply convert your project into a jBPM project, which will add the jBPM library to your project's classpath. Right-click your project and select "Configure ->

Convert to jBPM Project". Your project should now have a jBPM Library added to its classpath (it might be necessary to clean your project to pick up this change and recompile all Java classes).

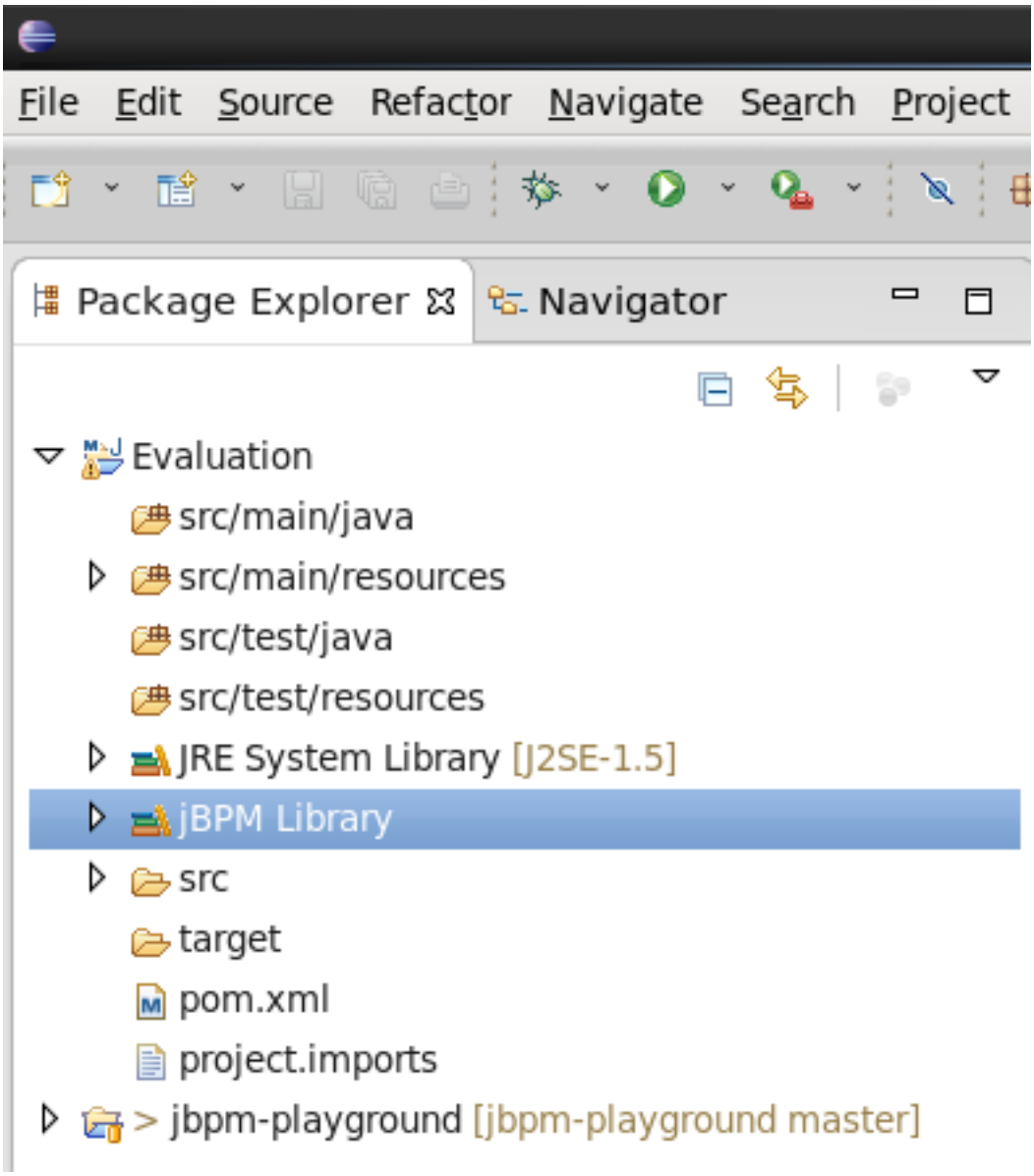


Figure 18.14.

Chapter 19. Eclipse BPMN 2.0 Modeler

19.1. Overview

The Eclipse BPMN 2.0 Modeler allows you to specify business processes, choreographies, etc. using the BPMN 2.0 XML syntax (including BPMNDI for the graphical information). The editor itself is based on the Eclipse Graphiti framework and the Eclipse BPMN 2.0 EMF meta-model.

Features:

- It supports almost all BPMN 2.0 process constructs and attributes (including lanes and pools, annotations and all the BPMN2 node types).
- Added additional support for the few custom attributes that jBPM introduces using a special jBPM Target Runtime.
- Allows you to configure which elements and attributes you want use when modeling processes (so we can limit the constructs for example to the subset currently supported by jBPM, which is a profile supported by default, or even more if you like).

The BPMN2 Modeler project is being developed at eclipse.org, sponsored by Red Hat/JBoss. Red Hat understands the benefits of developing software in the community, and therefore, the Eclipse BPMN 2.0 Modeler was developed not just for the jBPM project only, but it can be used in a much broader context and is fully spec compliant. jBPM-specific features are developed as part of a separate jBPM Target Runtime. We welcome other organizations in contributing to this modeler as well and (re)using the generic functionality and/or defining their own target runtime if necessary. Not only is this a good thing for the community, but it also leaves the path open for the jBPM suite to evolve as new features are requested by customers.

Many thanks go out to the people at Codehoop that did a great job in creating a first version of this editor.

19.2. Installation

The jBPM installer is capable of downloading and installing an Eclipse installation, including the Eclipse BPMN2 Modeler and the Drools and jBPM Eclipse plugin (with a full jBPM runtime preconfigured).

Tip

Using the jBPM installer is definitely the recommended starting point for most users.

You can however also download and install the jBPM Eclipse Plugin manually. To do so, you need Eclipse 3.6.2 (Helios) or newer. To install, startup Eclipse and install the Eclipse BPMN 2.0 Modeler from the following update site (from menu Help -> Install new software and then add the update site in question by clicking the Add button, filling in a name and the correct URL as shown below). It will automatically download all other dependencies as well (e.g. Graphiti etc.)

Eclipse 3.6 (Helios): *http://download.eclipse.org/bpmn2-modeler/updates/helios*

Eclipse 3.7 - 4.2.1 (Indigo - Juno): *http://download.eclipse.org/bpmn2-modeler/updates/juno*

Eclipse 4.3 (Kepler): *http://download.eclipse.org/bpmn2-modeler/updates/kepler*

The project is hosted at eclipse.org and open for anyone to contribute. The project home page can he found here:

http://http://eclipse.org/bpmn2-modeler/

Sources are available here (using Eclipse Public License v1.0):

https://git.eclipse.org/c/bpmn2-modeler/org.eclipse.bpmn2-modeler.git

A community forum for posting questions and exchanging ideas is also available here:

http://www.eclipse.org/forums/

A Bugzilla bug tracking system is available for reporting new bugs, or checking the status of existing bugs, here:

https://bugs.eclipse.org/bugs/buglist.cgi?product=BPMN2Modeler

19.3. Documentation

The Eclipse BPMN 2.0 Modeler documentation is available at:

http://eclipse.org/bpmn2-modeler/documentation.php

It contains various screencasts but also a full user guide, describing all its features in detail:

http://eclipse.org/bpmn2-modeler/documentation/UserGuide-v1.0.pdf

Here are some screenshots of the editor in action.

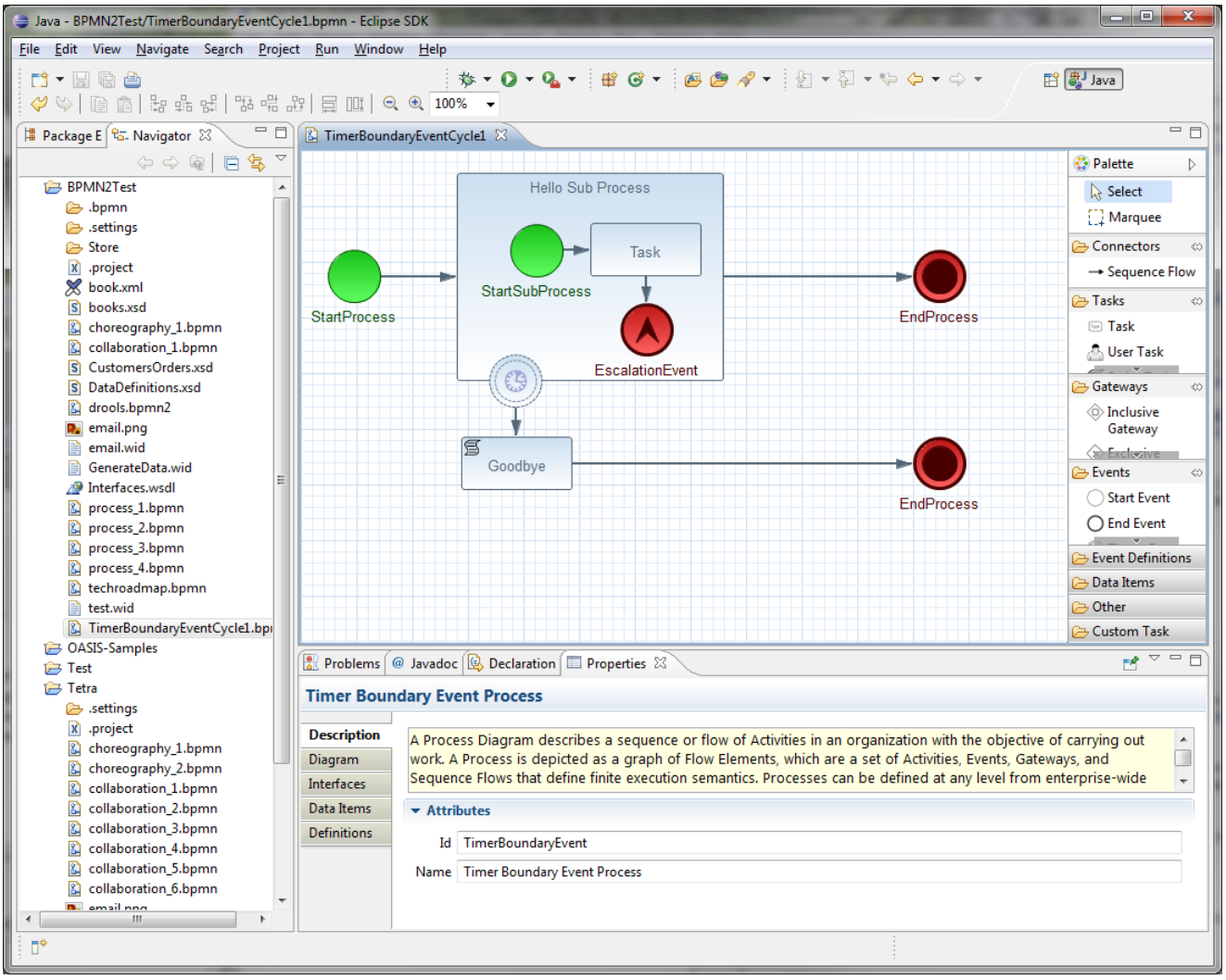


Figure 19.1.

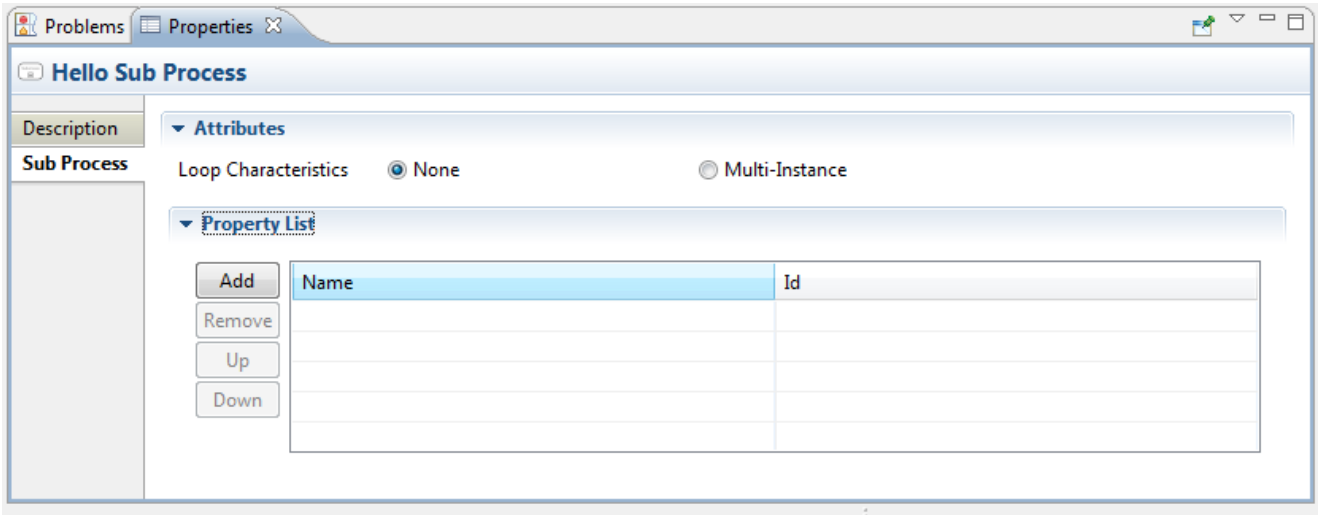


Figure 19.2.

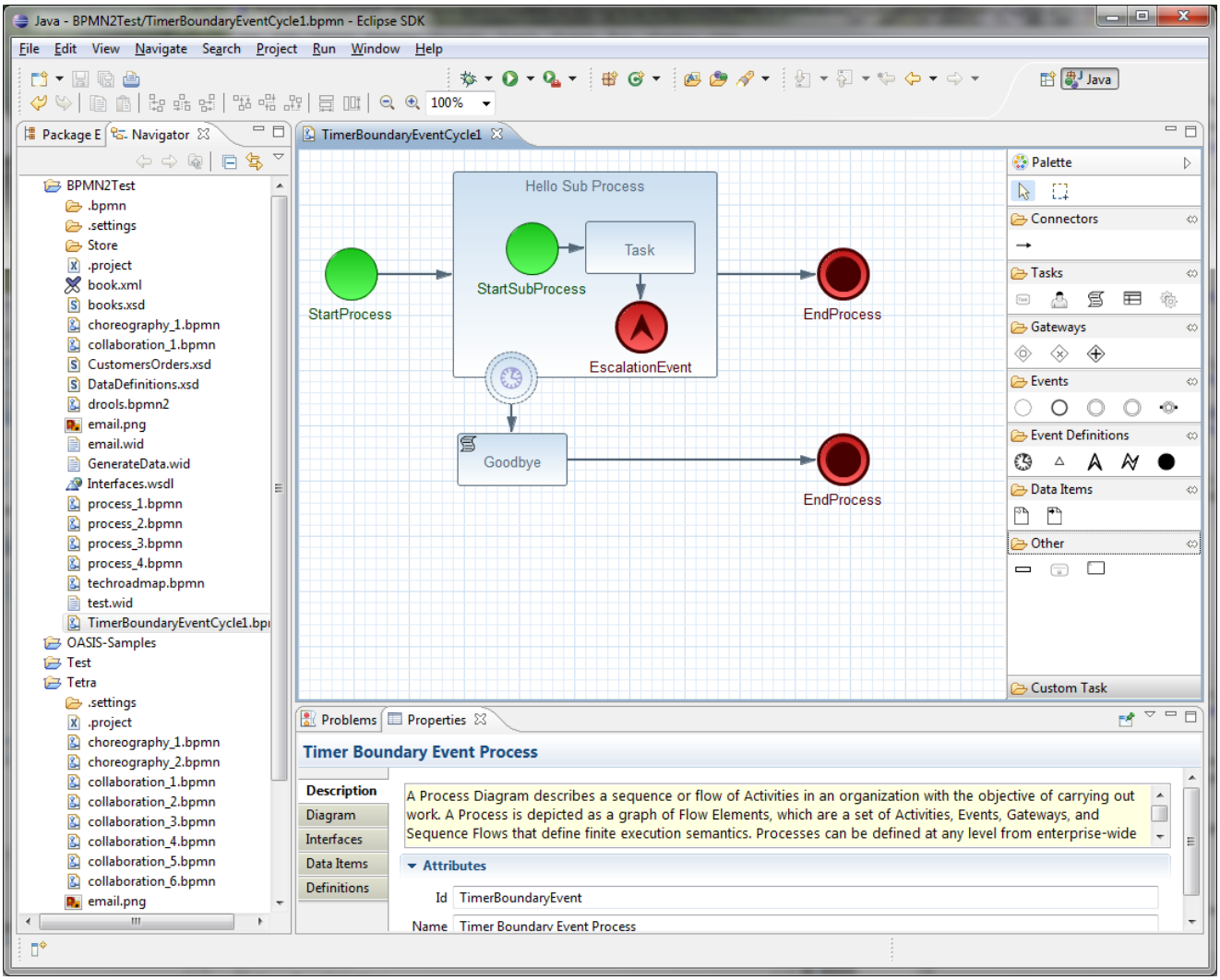

Java - BPMN2Test/TimerBoundaryEventCycle1.bpmn - Eclipse SDK
File Edit View Navigate Search Project Run Window Help
Package E
Navigator
BPMN2Test
.bpmn
.settings
Store
.project
book.xml
books.xsd
choreography_1.bpmn
collaboration_1.bpmn
CustomersOrders.xsd
DataDefinitions.xsd
drools.bpmn2
email.png
email.wid
GenerateData.wid
Interfaces.wsdl
process_1.bpmn
process_2.bpmn
process_3.bpmn
process_4.bpmn
techroadmap.bpmn
test.wid
TimerBoundaryEventCycle1.bpı
OASIS-Samples
Test
Tetra
.settings
.project
choreography_1.bpmn
choreography_2.bpmn
collaboration_1.bpmn
collaboration_2.bpmn
collaboration_3.bpmn
collaboration_4.bpmn
collaboration_5.bpmn
collaboration_6.bpmn
email.png
TimerBoundaryEventCycle1
Hello Sub Process
StartProcess
StartSubProcess
Task
EscalationEvent
EndProcess
Goodbye
EndProcess
Palette
Connectors
Tasks
Gateways
Events
Event Definitions
Data Items
Other
Custom Task
Problems
Properties
Timer Boundary Event Process
Description
Diagram
Interfaces
Data Items
Definitions
A Process Diagram describes a sequence or flow of Activities in an organization with the objective of carrying out work. A Process is depicted as a graph of Flow Elements, which are a set of Activities, Events, Gateways, and Sequence Flows that define finite execution semantics. Processes can be defined at any level from enterprise-wide
Attributes
Id
TimerBoundaryEvent
Name
Timer Boundary Event Process

Figure 19.3.

Part V. Integration

Integrating jBPM with other technologies, frameworks, etc.

Chapter 20. Integration

20.1. Maven

Apache Maven is used by jBPM for two main purposes:

- as deployment units that gets installed into runtime environment for execution
- as dependency management tool for building systems based on jBPM - embedding jBPM into application

20.1.1. Maven artifacts as deployment units

Since version 6, jBPM provides simplified and complete deployment mechanism that is based entirely on Apache Maven artifacts. These artifacts also known as **kjars** are simple JAR files that include a descriptor for KIE system to produce KieBase and KieSession. Descriptor of the kjar is represented as XML file named kmodule.xml and it can be:

- empty to apply all defaults
- custom configuration of KieBase and KieSession

```
<kmodule  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns="http://
jboss.org/kie/6.0.0/kmodule">
</kmodule>
```

Empty kmodule.xml that provides all defaults for the kjar:

- single default KieBase that
  - contains all assets from all packages
  - event processing mode set to - cloud
  - equality behaviour set to - identity
  - declarative agenda is disabled
  - scope set to - ApplicationScope - valid for CDI integrations only
- single default stateless KieSession that
  - is bound to above (single, default) KieBase

- clock type is set to - real time
- scope set to - ApplicationScope - valid for CDI integrations only

- single default stateful KieSession that
  - is bound to above (single, default) KieBase
  - clock type is set to - real time
  - scope set to - ApplicationScope - valid for CDI integrations only

All these and more can be configured manually via kmodule.xml when defaults are not enough. The complete set of elements can be found in *xsd schema* [https://github.com/droolsjbpm/droolsjbpm-knowledge/blob/6.0.x/kie-api/src/main/resources/org/kie/api/kmodule.xsd] of kmodule.xml.

```
<kmodule xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xmlns="http://jboss.org/kie/6.0.0/kmodule">
  <kbase name="defaultKieBase" default="true" eventProcessingMode="cloud" equalsBehavior="ident

  <ksession name="defaultKieSession" type="stateful" default="true" clockType="realtime" scope=
        <workItemHandlers>
            <workItemHandler name="CustomTask" type="FQCN_OF_HANDLER" />
        </workItemHandlers>
        <listeners>
            <listener type="FQCN_OF_EVENT_LISTENER" />
        </listeners>
    </ksession>

faultStatelessKieSession type="realtime" scope="javax.enterprise.context.ApplicationScoped"/
>
  </kbase>
</kmodule>
```

As illustrated in the listing above the kmodule.xml provides flexible way of instructing the runtime engine on what should be configured and how. The example above does not present all available options, but these are the most common when working with processes.

Note

Important to note is that when using RuntimeManager, KieSession instances are created by the RuntimeManager instead of by KieContainer but kmodule.xml (or model in general) is aways used as a base of the construction process. KieBase although is always taken from KieContainer.

Kjars are represented same way as any other Maven artifact - by **G**roup **A**rtifact **V**ersion which is then represented as ReleaseId in KIE API. This the the only thing required to deploy kjar into runtime environment such as KIE Workbeanch.

20.1.2. Use Maven for dependency management

When building systems that embed jBPM as workflow engine the simplest way is to configure all dependencies required by jBPM via Apache Maven. jBPM provides set of BOMs (Bill Of Material) to simplify what artifacts needs to be declared. Common way to start with integration of custom application and jBPM is to define dependency management:

```
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <drools.version>6.0.0.Final</drools.version>
  <jbpm.version>6.0.0.Final</jbpm.version>
  <hibernate.version>4.2.0.Final</hibernate.version>
  <hibernate.core.version>4.2.0.Final</hibernate.core.version>
  <slf4j.version>1.6.4</slf4j.version>
  <jboss.javaee.version>1.0.0.Final</jboss.javaee.version>
  <logback.version>1.0.9</logback.version>
  <h2.version>1.3.161</h2.version>
  <btm.version>2.1.4</btm.version>
  <junit.version>4.8.1</junit.version>
</properties>
<dependencyManagement>
  <dependencies>
    <!-- define drools BOM -->
    <dependency>
      <groupId>org.drools</groupId>
      <artifactId>drools-bom</artifactId>
      <type>pom</type>
      <version>${drools.version}</version>
      <scope>import</scope>
    </dependency>
    <!-- define drools BOM -->
    <dependency>
      <groupId>org.jbpm</groupId>
      <artifactId>jbpm-bom</artifactId>
      <type>pom</type>
      <version>${jbpm.version}</version>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

Above should be declared in top level pom.xml so all modules that need to use KIE (drools and jBPM) API can access it.

Next, module(s) that would operate on KIE API should declare following dependencies:

```
<dependency>
  <groupId>org.jbpm</groupId>
  <artifactId>jbpm-flow</artifactId>
</dependency>
<dependency>
  <groupId>org.jbpm</groupId>
  <artifactId>jbpm-flow-builder</artifactId>
</dependency>
<dependency>
  <groupId>org.jbpm</groupId>
  <artifactId>jbpm-bpmn2</artifactId>
</dependency>
<dependency>
  <groupId>org.jbpm</groupId>
  <artifactId>jbpm-persistence-jpa</artifactId>
</dependency>
<dependency>
  <groupId>org.jbpm</groupId>
  <artifactId>jbpm-human-task-core</artifactId>
</dependency>
<dependency>
  <groupId>org.jbpm</groupId>
  <artifactId>jbpm-runtime-manager</artifactId>
</dependency>
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-api</artifactId>
  <version>${slf4j.version}</version>
</dependency>
```

Above are the main runtime dependencies, regardless of where the application is deployed (application server, servlet container, standalone app). A good practice is to test the workflow components to ensure they work properly before actual deployment and thus following test dependencies should be defined:

```
<!-- test dependencies -->
<dependency>
  <groupId>org.jbpm</groupId>
  <artifactId>jbpm-shared-services</artifactId>
  <classifier>btm</classifier>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>ch.qos.logback</groupId>
```

```
      <artifactId>logback-classic</artifactId>
      <version>${logback.version}</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>${junit.version}</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-entitymanager</artifactId>
      <version>${hibernate.version}</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-core</artifactId>
      <version>${hibernate.core.version}</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>com.h2database</groupId>
      <artifactId>h2</artifactId>
      <version>${h2.version}</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.codehaus.btm</groupId>
      <artifactId>btm</artifactId>
      <version>${btm.version}</version>
      <scope>test</scope>
    </dependency>
```

Last but not least, define the JBoss Maven repository for artifacts resolution:

```
  <repositories>
    <repository>
      <id>jboss-public-repository-group</id>
      <name>JBoss Public Repository Group</name>
      <url>http://repository.jboss.org/nexus/content/groups/public/</url>
      <releases>
        <updatePolicy>never</updatePolicy>
      </releases>
      <snapshots>
        <updatePolicy>daily</updatePolicy>
```

```
      </snapshots>
    </repository>
  </repositories>
```

That should allow to configure jBPM in your application and provide access to KIE API to operate on processes, rules, events.

20.2. CDI

20.2.1. Overview

jBPM 6 comes with out of the box integration with CDI (Contexts and Dependency Injection). Although most of the API can be used in CDI world there are some dedicated modules that are designed especially for CDI containers. The most important one is jbpm-kie-services that provides high level services that shall be used in most of the cases were CDI is available for jBPM integration. It provides following set of services:

- DeploymentService
- FormProviderService
- RuntimeDataService
- BPMN2DataService

These services are first class citizens for CDI world so they are available for injection in any other CDI bean.

20.2.1.1. DeploymentService

Service responsible for deploying DeploymentUnits into runtime environment. By deploying given deployment unit becomes ready for execution and has RuntimeManager created for it.DeploymentService can next be used to retrieve:

- RuntimeManager instance for given deployment id
- DeployedUnit that represents complete deployment process for given deployment id
- list of all deployed units known to the deployment service

Deployment service stores the deployed units only in memory and thus in case of a need to restore all previously deployed units, component that uses deployment service needs to store that information itself. Common places for such a store are database, file system, repository of some sort, etc. Deployment service will fire CDI events on deployment and undeployment to allow application components to react real time to these events to be able to store deployments or remove them from the store when they are undeployed.

- DeploymentEvent with qualifier @Deploy will be fired on deployment
- DeploymentEvent with qualifier @Undeploy will be fired on undeployment

use CDI observer mechanism to get notification on above events. First to save deployments in the store of your choice:

```
    public void saveDeployment(@Observes @Deploy DeploymentEvent event) {
        // store deployed unit info for further needs
        DeployedUnit deployedUnit = event.getDeployedUnit();
    }
```

next to remove it when it was undeployed

```
    public void removeDeployment(@Observes @Undeploy DeploymentEvent event) {
        // remove deployment with id event.getDeploymentId()
    }
```

Due to the fact that there might be several implementation of DeploymentService use of qualifiers is needed to instruct CDI container which one shall be injected. jBPM comes with two out of the box:

- @Kjar - KmoduleDeploymentService that is tailored to work with KmoduleDeploymentUnits that are small descriptor on top of a kjar - recommended to use in most of the cases
- @Vfs - VFSDeploymentDService that allows to deploy assets directly from VFS (Virtual File System) that is provided by *UberFire framework* [http://droolsjbpm.github.io/uberfire/]. Due to that fact VFSDeploymentService and VFSDeploymentUnit are not bundled with jbpm core modules but with jbpm-console-ng modules.

The general practice is that every implementation of DeploymentService should come with dedicated implementation of DeploymentUnit as these two provided out of the box.

20.2.1.2. FormProviderService

FormProviderService provides access to form representations usually displayed on UI for both process forms and user task forms. It is built on concept of isolated FormProviders that can provide different capabilities and be backed by different technologies. FormProvider interface describes contract for the implementations

```
public interface FormProvider {

    int getPriority();

    String render(String name, ProcessDesc process, Map<String, Object> renderContext);

    String render(String name, Task task, ProcessDesc process, Map<String, Object> renderContex
}
```

Implementations of FormProvider interface should always define priority as this is the main driver for the FormProviderService to ask for the content of the form of a given provider. FormProviderService will collect all available providers and iterate over them asking for the form content (rendered) in their priority order. The lower the number the higher priority it gets during evaluation, e.g. provider with priority 5 will be evaluated before provider with priority 10. FormProviderService will iterate over available providers as long as one delivers the content. In the worse case scenario, simple text based forms will be returned.

jBPM comes with following FormProviders out of the box:

- Fremarker based implementation to support jbpm version 5 process and task forms - priority 3
- Default forms provider, considered last resort if none of the other providers deliver content this one will always provide simplest possible forms - lowest priority (1000)
- when form modeler is used there is additional FormProvider available to deliver forms modeled in that tool - priority 2

20.2.1.3. RuntimeDataService

RuntimeDataService provides access to actual data that is availabe on runtime such as

- available processes to be executed - with various filters
- active process instances - with various filters
- process instance history
- process instance variables
- active and completed nodes of process instance

Default implementation of RuntimeDataService is observing deployment events and index all deployed processes to expose them to the calling components. So whatever gets deployed RuntimeDataService will be aware of that.

20.2.1.4. BPMN2DataService

Service that provides access to process details stored as part of BPMN2 XML.

Note

Before using any method that provides information, **findProcessId** must be invoked to populate repository with process information taken from BPMN2 content.

BPMN2DataService provides access to following data:

- overall description of process for given process definition

- collection of all user tasks found in the process definition
- information about defined inputs for user task node
- information about defined outputs for user task node
- ids of reusable processes (call activity) defined within given process definition
- information about process variables defined within given process definition
- information about all organizational entities (users and groups) included in the process definition. Depending on the actual process definition the returned values for users and groups can contain
  - actual user or group name
  - process variable that will be used to get actual user or group name on runtime e.g. #{manager}

20.2.2. Configuring CDI integration

To make use of jbpm-kie-services in your system you'll need to provide some beans for the out of the box services to satisfy all dependencies they have. There are several beans that depends on actual scenario

- entity manager and entity manager factory
- user group callback for human tasks
- identity provider to pass authenticated user information to the services

When running in JEE environment like an JBoss Application Server following producer bean should satisfy all requirements of the jbpm-kie-services

```
public class EnvironmentProducer {

    @PersistenceUnit(unitName = "org.jbpm.domain")
    private EntityManagerFactory emf;

    @Inject
    @Selectable
    private UserGroupCallback userGroupCallback;

    @Produces
    public EntityManagerFactory getEntityManagerFactory() {
        return this.emf;
    }

    @Produces
    @RequestScoped
    public EntityManager getEntityManager() {
        EntityManager em = emf.createEntityManager();
```

```
        return em;
    }

    public void close(@Disposes EntityManager em) {
        em.close();
    }

    @Produces
    public UserGroupCallback produceSelectedUserGroupCalback() {
        return userGroupCallback;
    }

    @Produces
    public IdentityProvider produceIdentityProvider {
        return new IdentityProvider() {
             // implement IdentityProvider
        };
    }
}
```

Then beans.xml for the application should enable proper alternative for user group callback (that will be taken based on @Selectable qualifier)

```
<beans       xmlns="http://java.sun.com/xml/ns/javaee"       xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://docs.jboss.org/
cdi/beans_1_0.xsd">

  <alternatives>
    <class>org.jbpm.kie.services.cdi.producer.JAASUserGroupInfoProducer</class>
  </alternatives>

</beans>
```

Note

org.jbpm.kie.services.cdi.producer.JAASUserGroupInfoProducer is just an example here which usually is the good fit for JBoss Application Server to reuse security settings on application server regardless of what it actually is (LDAP, DB, etc). Check Human Task section for more alternatives for UserGroupCallback.

Optionally there can be several other producers provided to deliver:

- WorkItemHandlers

- Process, Agenda, WorkingMemory event listeners

These components can be provided by implementing following interfaces

```
/**
 * Allows to provide custom implementations to deliver WorkItem name and WorkItemHandler instan
 * for the runtime.
 * <br/>
 * It will be invoked by RegisterableItemsFactory implementation (especially InjectableRegister
 * in CDI world) for every KieSession. Recommendation is to always produce new instances to avo
 * results.
 *
 */
public interface WorkItemHandlerProducer {

    /**
     * Returns map of (key = work item name, value work item handler instance) of work items
     * to be registered on KieSession
     * <br/>
     * Parameters that might be given are as follows:
     * <ul>
     *  <li>ksession</li>
     *  <li>taskService</li>
     *  <li>runtimeManager</li>
     * </ul>
     *
     * @param identifier - identifier of the owner - usually RuntimeManager that allows the pro
     * and provide valid instances for given owner
     * @param params - owner might provide some parameters, usually KieSession, TaskService, Ru
     * @return map of work item handler instances (recommendation is to always return new insta
     */
    Map<String, WorkItemHandler> getWorkItemHandlers(String identifier, Map<String, Object> par
}
```

and

```
/**
 * Allows do define custom producers for know EventListeners. Intention of this is that there m
 * implementations that might provide different listener instance based on the context they are
 * <br/>
 * It will be invoked by RegisterableItemsFactory implementation (especially InjectableRegister
 * in CDI world) for every KieSession. Recommendation is to always produce new instances to avo
 * results.
 *
 * @param <T> type of the event listener - ProcessEventListener, AgendaEventListener, WorkingMe
 */
public interface EventListenerProducer<T> {
```

```
    /**
     * Returns list of instances for given (T) type of listeners
     * <br/>
     * Parameters that might be given are as follows:
     * <ul>
     *  <li>ksession</li>
     *  <li>taskService</li>
     *  <li>runtimeManager</li>
     * </ul>
     * @param identifier - identifier of the owner - usually RuntimeManager that allows the pro
     * and provide valid instances for given owner
     * @param params - owner might provide some parameters, usually KieSession, TaskService, Ru
     * @return list of listener instances (recommendation is to always return new instances whe
     */
    List<T> getEventListeners(String identifier, Map<String, Object>  params);
}
```

Beans implementing these two interfaces will be collected on runtime and consulted when building KieSession by RuntimeManager. See RuntimeManager section for more details on this.

20.2.3. RuntimeManager as CDI bean

RuntimeManager itself can be injected as CDI bean into any other CDI bean within the application. It has then requirement to get RungimeEnvironment properly produces to allow RuntimeManager to be correctly initialized. RuntimeManager comes with three predefined strategies and each of them gets CDI qualifier so it can be referenced:

- @Singleton
- @PerRequest
- @PerProcessInstance

Producer that was defined in Configuration section should be now enhanced with producer methods to provide RuntimeEnvironment

```
public class EnvironmentProducer {

    @PersistenceUnit(unitName = "org.jbpm.domain")
    private EntityManagerFactory emf;

    @Inject
    @Selectable
    private UserGroupCallback userGroupCallback;

    @Inject
    private BeanManager beanManager;
```

```
    @Produces
    public EntityManagerFactory getEntityManagerFactory() {
        return this.emf;
    }

    @Produces
    @RequestScoped
    public EntityManager getEntityManager() {
        EntityManager em = emf.createEntityManager();
        return em;
    }

    public void close(@Disposes EntityManager em) {
        em.close();
    }

    @Produces
    public UserGroupCallback produceSelectedUserGroupCalback() {
        return userGroupCallback;
    }

    @Produces
    public IdentityProvider produceIdentityProvider {
        return new IdentityProvider() {
             // implement IdentityProvider
        };
    }

    @Produces
    @Singleton
    @PerRequest
    @PerProcessInstance
    public RuntimeEnvironment produceEnvironment(EntityManagerFactory emf) {

       RuntimeEnvironment environment = RuntimeEnvironmentBuilder.Factory.get()
                .newDefaultBuilder()
                .entityManagerFactory(emf)
                .userGroupCallback(getUserGroupCallback())
                .registerableItemsFactory(InjectableRegisterableItemsFactory.getFactory(beanMar
                        .addAsset(ResourceFactory.newClassPathResource("BPMN2-
ScriptTask.bpmn2"), ResourceType.BPMN2)
                        .addAsset(ResourceFactory.newClassPathResource("BPMN2-
UserTask.bpmn2"), ResourceType.BPMN2)
                .get();
        return environment;
    }
}
```

In this example single producer method is capable of providing RuntimeEnvironment for all strategies of RuntimeManager by specifying all qualifiers on the method level.

Once complete producer is available, RuntimeManager can be injected into application's CDi bean

```
public class ProcessEngine {

    @Inject
    @Singleton
    private RuntimeManager singletonManager;

    public void startProcess() {

        RuntimeEngine runtime = singletonManager.getRuntimeEngine(EmptyContext.get());
        KieSession ksession = runtime.getKieSession();

        ProcessInstance processInstance = ksession.startProcess("UserTask");

        singletonManager.disposeRuntimeEngine(runtime);
    }
}
```

That's all what needs to be configured to make use of CDI power with jBPM.

Note

An obvious limitation of injecting directly RuntimeManager via CDI is that there might be only one RuntimeManager in the application. That in some case can be desired and that's why there is such option. In general recommended approach is to make use of DeploymentService whenever there is a need to have many RuntimeManagers active within application.

As an alternative to DeploymentService, RuntimeManagerFactory can be injected and then RuntimeManager instance can be created manually by the application. In such case EnvironmentProducer stays same as for DeploymentService and following is an example of simple ProcessEngine bean

```
public class ProcessEngine {

    @Inject
    private RuntimeManagerFactory managerFactory;

    @Inject
    private EntityManagerFactory emf;
```

```
    @Inject
    private BeanManager beanManager;

    public void startProcess() {
       RuntimeEnvironment environment = RuntimeEnvironmentBuilder.Factory.get()
                .newDefaultBuilder()
                .entityManagerFactory(emf)
                        .addAsset(ResourceFactory.newClassPathResource("BPMN2-
ScriptTask.bpmn2"), ResourceType.BPMN2)
                        .addAsset(ResourceFactory.newClassPathResource("BPMN2-
UserTask.bpmn2"), ResourceType.BPMN2)
                .registerableItemsFactory(InjectableRegisterableItemsFactory.getFactory(beanMar
                .get();

        RuntimeManager manager = managerFactory.newSingletonRuntimeManager(environment);
        RuntimeEngine runtime = manager.getRuntimeEngine(EmptyContext.get());
        KieSession ksession = runtime.getKieSession();

        ProcessInstance processInstance = ksession.startProcess("UserTask");

        manager.disposeRuntimeEngine(runtime);
        manager.close();
    }

}
```

20.3. OSGi

All core jBPM JARs (and core dependencies) are OSGi-enabled. That means that they contain MANIFEST.MF files (in the META-INF directory) that describe their dependencies etc. These manifest files are automatically generated by the build. You can plug these JARs directly into an OSGi environment.

OSGi is a dynamic module system for declarative services. So what does that mean? Each JAR in OSGi is called a bundle and has its own Classloader. Each bundle specifies the packages it exports (makes publicly available) and which packages it imports (external dependencies). OSGi will use this information to wire the classloaders of different bundles together; the key distinction is you don't specify what bundle you depend on, or have a single monolithic classpath, instead you specify your package import and version and OSGi attempts to satisfy this from available bundles.

It also supports side by side versioning, so you can have multiple versions of a bundle installed and it'll wire up the correct one. Further to this Bundles can register services for other bundles to use. These services need initialisation, which can cause ordering problems - how do you make sure you don't consume a service before its registered? OSGi has a number of features to help with service composition and ordering. The two main ones are the programmatic ServiceTracker

and the XML based Declarative Services. There are also other projects that help with this; Spring DM, iPOJO, Gravity.

The following jBPM JARs are OGSi-enabled:

- jbpm-flow
- jbpm-flow-builder
- jbpm-bpmn2

Part VI. Advanced Topics

Some more advanced topics

Chapter 21. Domain-specific Processes

21.1. Introduction

jBPM provides the ability to create and use domain-specific task nodes in your business processes. This simplifies development when you're creating business processes that contain tasks dealing with other technical systems.

When using jBPM, we call these domain-specific task nodes "*custom work items*" or (custom) "*service nodes*". There are two separate aspects to creating and using custom work items:

- Adding a node with a custom work item to a process definition using the Eclipse editor or jBPM designer.
- Creating a custom *work item handler* that the jBPM engine will use when executing the custom work item in a running process.

With regards to a BPMN2 process, custom work items are certain types of `<task>` nodes. In most cases, custom work items are `<task>` nodes in a BPMN2 process definition, although they can also be used with certain other task type nodes such as, among others, `<serviceTask>` or `<sendTask>` nodes.

Tip

When creating custom work items, it's important to separate the data associated with the work item, from how the work item should be handled. In other words, separate the *what* from the *how*. That means that custom work items should be:

- declarative (what, not how)
- high-level (no code)

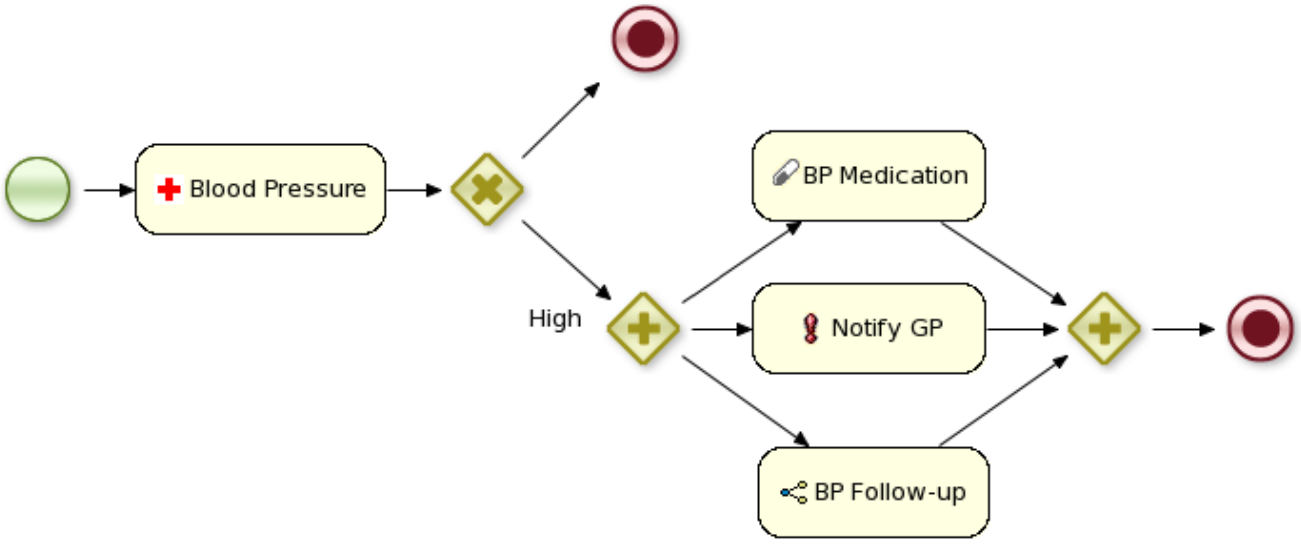
On the other hand, custom work item *handlers*, which are Java classes, should be:

- procedural (how, not what)
- low-level (because it's code!)

Work item handlers should almost never contain any data.

Users can thus easily define their own set of domain-specific service nodes and integrate them with the process language. For example, the next figure shows an example of a healthcare-

related BPMN2 process. The process includes domain-specific service nodes for measuring blood pressure, prescribing medication, notifying care providers and following-up on the patient.



21.2. Overview

Before moving on to an example, this section explains what custom work items and custom work item handlers are.

21.2.1. Work Item Definitions

In short, we use the term *custom work item* when we're describing a node in your process that represents a domain-specific task and as such, contains extra properties and is handled by a `WorkItemHandler` implementation.

Because it's a domain-specific *task*, that means that a *custom work item* is equivalent to a `<task>` or `<task>`-type node in BPMN2. However, a *`WorkItem`* is also Java class instance that's used when a `WorkItemHandler` instance is called to complete the task or work item.

Depending on the BPMN2 editor you're using, you can create a custom work item definition in one of two ways:

- If you're using *Designer*, then this means creating a MVEL based definition and adding the definition in Designer itself. A description of this can be found in the *???* section in the *???* chapter. Once this is done, a new service node will appear on the BPMN 2.0 palette.
- If you're using the *Eclipse BPMN 2.0 modeler plugin* (which can be found *here* [http://eclipse.org/bpmn2-modeler/]), then you'll can modify the BPMN2 `<task>` or `<task>`-type element to work with `WorkItemHandler` implementations. See the *???* section in the *???* chapter.

21.2.2. Work Item Handlers

A *work item handler* is a Java class used to execute (or abort) work items. That also means that the class implements the `org.kie.runtime.instance.WorkItemHandler` interface. While

jBPM provides some custom `WorkItemHandler` instances (listed below), a Java developer with a minimal knowledge of jBPM can easily create a new work item handler class with its own custom business logic.

Among others, jBPM offers the following `WorkItemHandler` implementations:

- In the `jbpm-bpmn2` module, `org.jbpm.bpmn2.handler` package:
  - ReceiveTaskHandler (for use with BPMN element `<receiveTask>`)
  - SendTaskHandler (for use with BPMN element `<sendTask>`)
  - ServiceTaskHandler (for use with BPMN element `<serviceTask>`)
- In the `jbpm-workitems` module, in various packages under the `org.jbpm.process.workitem` package:
  - ArchiveWorkItemHandler

There are a many more `WorkItemHandler` implementations present in the `jbpm-workitems` module. If you're looking for specific integration logic with Twitter, for example, we recommend you take a look at the classes made available there.

In general, a `WorkItemHandler`'s `.executeWorkItem(...)` and `.abortWorkItem(...)` methods will do the following:

1. Extract information about the task being executed (or aborted) from the `WorkItem` instance
2. Execute the necessary business logic. This might be mean interacting with a web service, database, or other technical component.
3. Inform the process engine that the work item has been completed (or aborted) by calling one of the following two methods on the `WorkItemManager` instance passed to the method:

```
WorkItemManager.completeWorkItem(long workItemId, Map<String, Object> results)
WorkItemManager.abortWorkItem(long workItemId)
```

In order to make sure that your custom work item handler is used for a particular process instance, it's necessary to register the work item handler before starting the process. This makes the engine aware of your `WorkItemHandler` so that the engine can use it for the proper node. For example:

```
ksession.getWorkItemManager().registerWorkItemHandler("Notification",
    new NotificationWorkItemHandler());
```

The `ksession` variable above is a `StatefulKnowledgeSession` (and also a `KieSession`) instance. The example code above comes from the example that we will go through in the next session.

Tip

You can use different work item handlers for the same process depending on the system on which it runs: by registering different work item handlers on different systems, you can customize how a custom work item is processed on a particular system. You can also substitute mock `WorkItemHandler` instances when testing.

21.3. Example: Notifications

Let's start by showing you how to include a simple work item for sending notifications. A work item is defined by a unique name and includes additional parameters that describe the work in more detail. Work items can also return information after they have been executed, specified as results.

Our notification work item could be defined using a work definition with four parameters and no results. For example:

- Name: "Notification"
- Parameters:
  - From [String type]
  - To [String type]
  - Message [String type]
  - Priority [String type]

21.3.1. The Notification Work Item Definition

21.3.1.1. Creating the work item definition

In our example we will create a MVEL work item definition that defines a "Notification" work item. Using MVEL is the default way to This file will be placed in the project classpath in a directory called `META-INF`. The work item configuration file for this example, `MyWorkDefinitions.wid`, will look like this:

```
import org.drools.core.process.core.datatype.impl.type.StringDataType;
[
  // the Notification work item
  [
    "name" : "Notification",
    "parameters" : [
```

```
      "Message" : new StringDataType(),
      "From" : new StringDataType(),
      "To" : new StringDataType(),
      "Priority" : new StringDataType(),
    ],
    "displayName" : "Notification",
    "icon" : "icons/notification.gif"
  ]
]
```

The project directory structure could then look something like this:

```
project/src/main/resources/META-INF/MyWorkDefinitions.wid
```

We also want to *add* a specific icon to be used in the process editor with the work item. To add this, you will need `.gif` or `.png` images with a pixel size of 16x16. We put them in a directory outside of the `META-INF` directory, for example, here:

```
project/src/main/resources/icons/notification.gif
```

21.3.1.2. Registering the work definition

The jBPM Eclipse editor uses the configuration mechanisms supplied by Drools to register work item definition files. That means adding a `drools.workDefinitions` property to the `drools.rulebase.conf` file in the `META-INF`.

The `drools.workDefinitions` property represents a list of files containing work item definitions, separated using spaces. If you want to *exclude* all other work item definitions and only use your definition, you could use the following:

```
drools.workDefinitions = MyWorkDefinitions.wid
```

However, if you only want to add the newly created node definition to the existing palette nodes, you can define the `drools.workDefinitions` property as follows:

```
drools.workDefinitions = MyWorkDefinitions.wid WorkDefinitions.conf
```

We recommended that you use the extension `.wid` for your own definitions of domain specific nodes. The `.conf` extension used with the default definition file, `WorkDefinitions.conf`, for backward compatibility reasons.

21.3.1.3. Using your new work item in your processes

We've created our work item definition and configured it, so now we can start using it in our processes. The process editor contains a separate section in the palette where the different service nodes that have been defined for the project appear.

Using drag and drop, a notification node can be created inside your process. The properties can be filled in using the properties view.

Besides any custom properties, the following three properties are available for all work items:

1. `Parameter Mapping`: Allows you to map the value of a variable in the process to a parameter of the work item. This allows you to customize the work item based on the current state of

the actual process instance (for example, the priority of the notification could be dependent of some process-specific information).

2. `Result Mapping`: Allows you to map a result (returned once a work item has been executed) to a variable of the process. This allows you to use results in the remainder of the process.

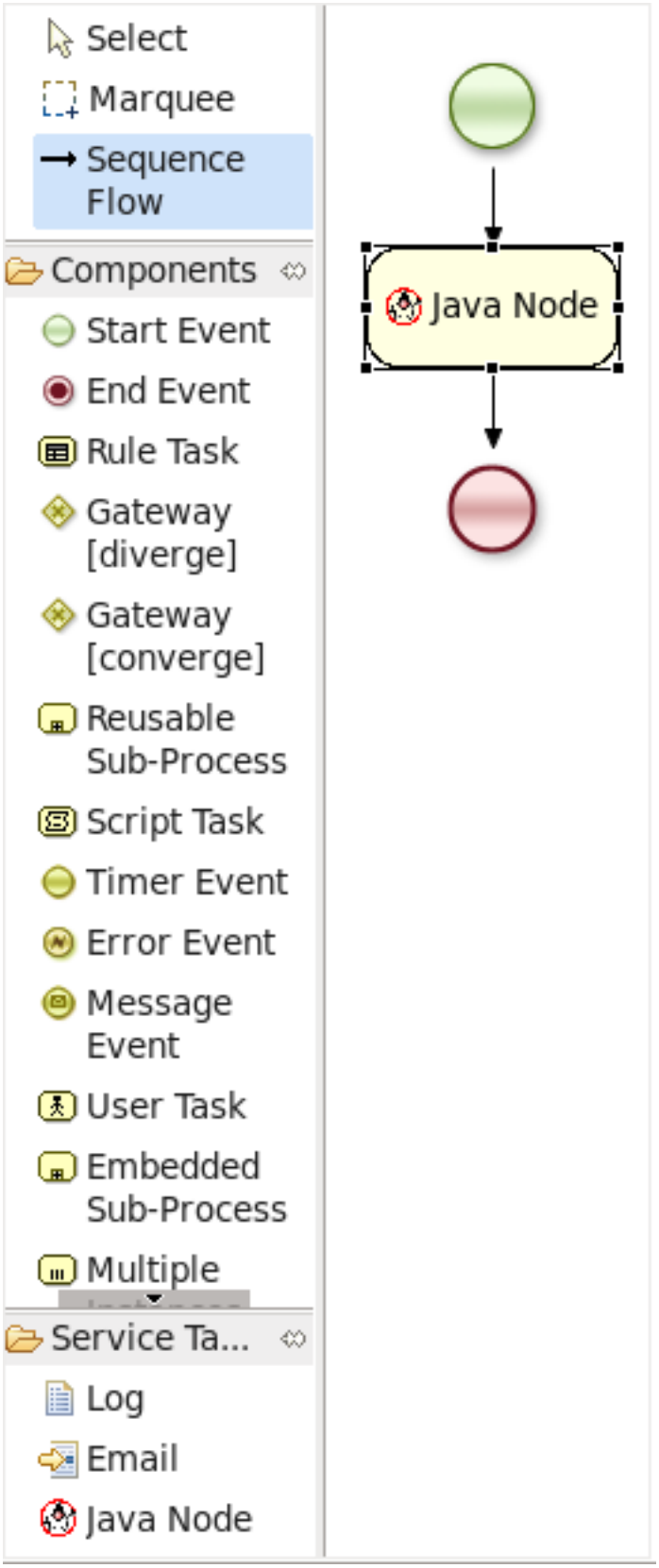
3. `Wait for completion`: By default, the process waits until the requested work item has been completed before continuing with the process. It is also possible to continue immediately after the work item has been requested (and not waiting for the results) by setting `wait for completion` to false.

Here is an example that creates a domain specific node to execute Java, asking for the class and method parameters. It includes a custom `java.gif` icon and consists of the following files and resulting screenshot:

```
import org.drools.core.process.core.datatype.impl.type.StringDataType;
[
  // the Java Node work item located in:
  // project/src/main/resources/META-INF/JavaNodeDefinition.wid
  [
    "name" : "JavaNode",
    "parameters" : [
      "class" : new StringDataType(),
      "method" : new StringDataType(),
    ],
    "displayName" : "Java Node",
    "icon" : "icons/java.gif"
  ]
]
```

```
// located in: project/src/main/resources/META-INF/drools.rulebase.conf
drools.workDefinitions = JavaNodeDefinition.wid WorkDefinitions.conf

// icon for java.gif located in:
// project/src/main/resources/icons/java.gif
```


Select
Marquee
Sequence Flow
Components
Start Event
End Event
Rule Task
Gateway [diverge]
Gateway [converge]
Reusable Sub-Process
Script Task
Timer Event
Error Event
Message Event
User Task
Embedded Sub-Process
Multiple
Service Ta...
Log
Email
Java Node
Java Node

21.3.2. The `NotificationWorkItemHandler`

21.3.2.1. Creating a new work item handler

Once we've created our `Notification` work item definition (see the sections above), we can then create a custom implementation of a *work item handler* that will contain the logic to send the notification.

In order to execute our *Notification* work items, we first create a `NotificationWorkItemHandler` that implements the `WorkItemHandler` interface:

```
package com.sample;

import org.kie.api.runtime.process.WorkItem;
import org.kie.api.runtime.process.WorkItemHandler;
import org.kie.api.runtime.process.WorkItemManager;

public class NotificationWorkItemHandler implements WorkItemHandler {

  public void executeWorkItem(WorkItem workItem, WorkItemManager manager) {
    // extract parameters
    String from = (String) workItem.getParameter("From");
    String to = (String) workItem.getParameter("To");
    String message = (String) workItem.getParameter("Message");
    String priority = (String) workItem.getParameter("Priority");

    // send email
   EmailService service = ServiceRegistry.getInstance().getEmailService();
    service.sendEmail(from, to, "Notification", message);

    // notify manager that work item has been completed
    manager.completeWorkItem(workItem.getId(), null);
  }

  public void abortWorkItem(WorkItem workItem, WorkItemManager manager) {
    // Do nothing, notifications cannot be aborted
  }

}
```

1. The `ServiceRegistry` class is simply a made-up class that we're using for this example. In your own `WorkItemHandler` implementations, the code containing your domain-specific logic would go here.
2. Notifying the *`WorkItemManager`* instance when your a work item has been completed is crucial. For many synchronous actions, like sending an email in this

case, the `WorkItemHandler` implementation will notify the `WorkItemManager` in the `executeWorkItem(...)` method.

This `WorkItemHandler` sends a notification as an email and then notifies the WorkItemManager that the work item has been completed.

Note that not all work items can be completed directly. In cases where executing a work item takes some time, execution can continue *asynchronously* and the work item manager can be notified later.

In these situations, it might also be possible that a work item is *aborted* before it has been completed. The `WorkItemHandler.abortWorkItem(...)` method can be used to specify how to abort such work items.

Tip

Remember, if the `WorkItemManager` is not notified about the completion, the process engine will never be notified that your service node has completed.

21.3.2.2. Registering the work item handler

`WorkItemHandler` instances need to be registered with the `WorkItemManager` in order to be used. In this case, we need to register an instance of our `NotificationWorkItemHandler` in order to use it with our process containing a `Notification` work item. We can do that like this:

```
StatefulKnowledgeSession ksession = kbase.newStatefulKnowledgeSession();
ksession.getWorkItemManager().registerWorkItemHandler(
  "Notification",                                                    (1)
  new NotificationWorkItemHandler()                                  (2)
);
```

1. This is the drools name of the `<task>` (or other task type) node. See below for an example.
2. This is the instance of our custom work item handler instance!

If we were to look at the BPMN2 syntax for our process with the `Notification` process, we would see something like the following example. Note the use of the `tns:taskName` attribute in the `<task>` node. This is necessary for the `WorkItemManager` to be able to see which `WorkItemHandler` instance should be used with which task or work item.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions id="Definition"
```

```
            xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
               xs:schemaLocation="http://www.omg.org/spec/BPMN/20100524/MODEL
 BPMN20.xsd"
...
            xmlns:tns="http://www.jboss.org/drools">

...

    <process  isExecutable="true"  id="myCustomProcess"  name="Domain-Specific
 Process" >

...

    <task id="_5" name="Notification Task" tns:taskName="Notification" >

...
```

Tip

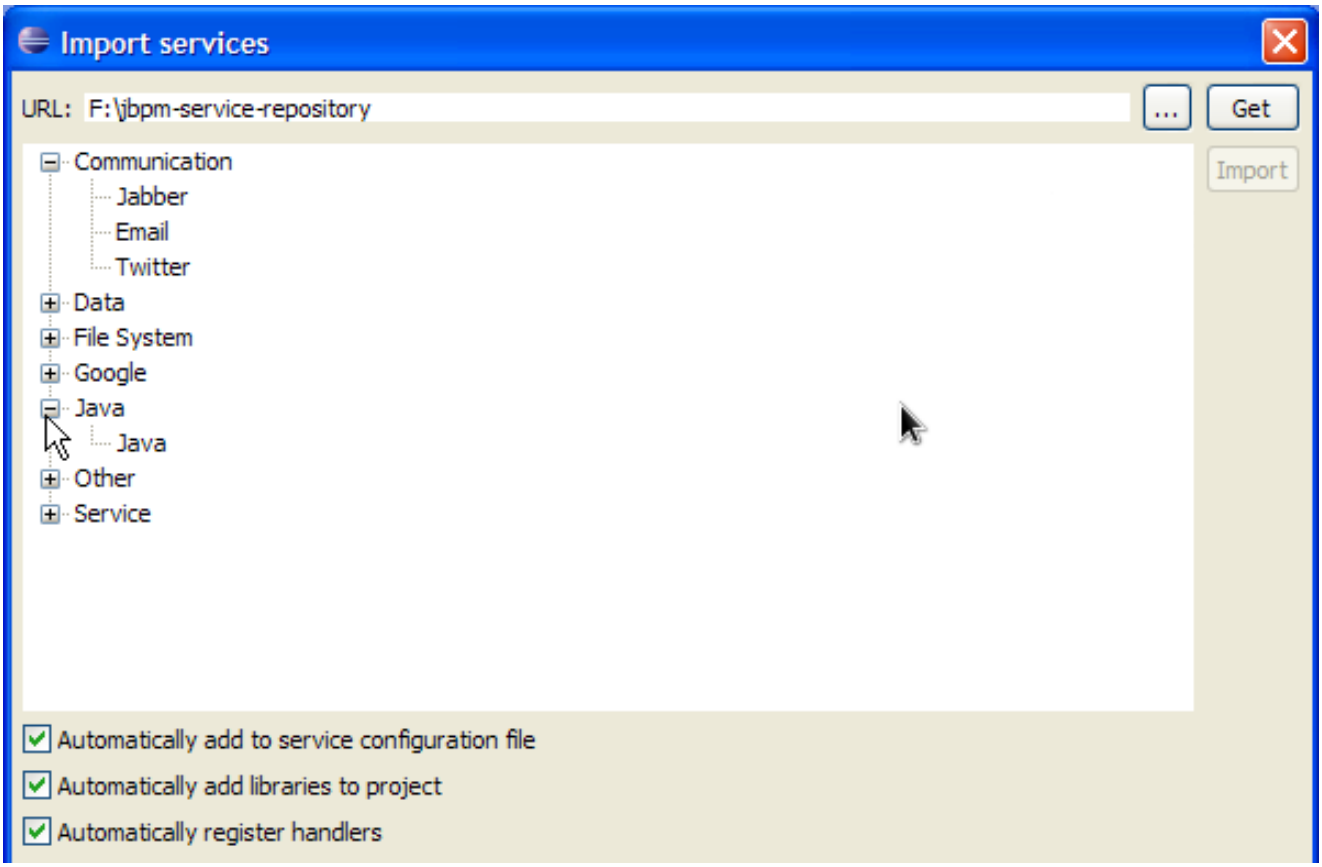
Different work item handlers could be used depending on the context. For example, during testing or simulation, it might not be necessary to actually execute the work items. In this case specialized dummy work item handlers could be used during testing.

21.4. Service Repository

A lot of these domain-specific services are generic, and can be reused by a lot of different users. Think for example about integration with Twitter, doing file system operations or sending email. Once such a domain-specific service has been created, you might want to make it available to other users so they can easily import and start using it.

A service repository allows you to import services by browsing the repository looking for services you might need and importing these services into your workspace. These will then automatically be added to your palette and you can start using them in your processes. You can also import additional artefacts like for example an icon, any dependencies you might need, a default handler that will be used to execute the service (although you're always free to override the default, for example for testing), etc.

To browse the repository, open the wizard to import services, point it to the right location (this could be to a directory in your file system but also a public or private URL) and select the services you would like to import. For example, in Eclipse, right-click your project that contains your processes and select "Configure ... -> Import jBPM services ...". This will open up a repository browser. In the URL field, fill in the URL of your repository (see below for the URL of the public jBPM repository that hosts some common service implementations out-of-the-box), or use the "..." button to browse to a folder on your file system. Click the Get button to retrieve the contents of that repository.



Select the service you would like to import and then click the Import button. Note that the Eclipse wizard allows you to define whether you would like to automatically configure the service (so it shows up in the palette of your processes), whether you would also like to download any dependencies that might be needed for executing the service and/or whether you would like to automatically register the default handler, so make sure to mark the right checkboxes before importing your service (if you are unsure what to do, leaving all check boxes marked is probably best).

After importing your service, (re)open your process diagram and the new service should show up in your palette and you can start using it in your process. Note that most services also include documentation on how to use them (e.g. what the different input and output parameters are) when you select them browsing the service repository.

Click on the image below to see a screencast where we import the Twitter service in a new jBPM project and create a simple process with it that sends an actual tweet. Note that you need the necessary Twitter keys and secrets to be able to programmatically send tweets to your Twitter account. How to create these is explained *here* [http://docs.jboss.org/jbpm/v6.0/repository/Twitter/], but once you have these, you can just drop them in your project using a simple configuration file.

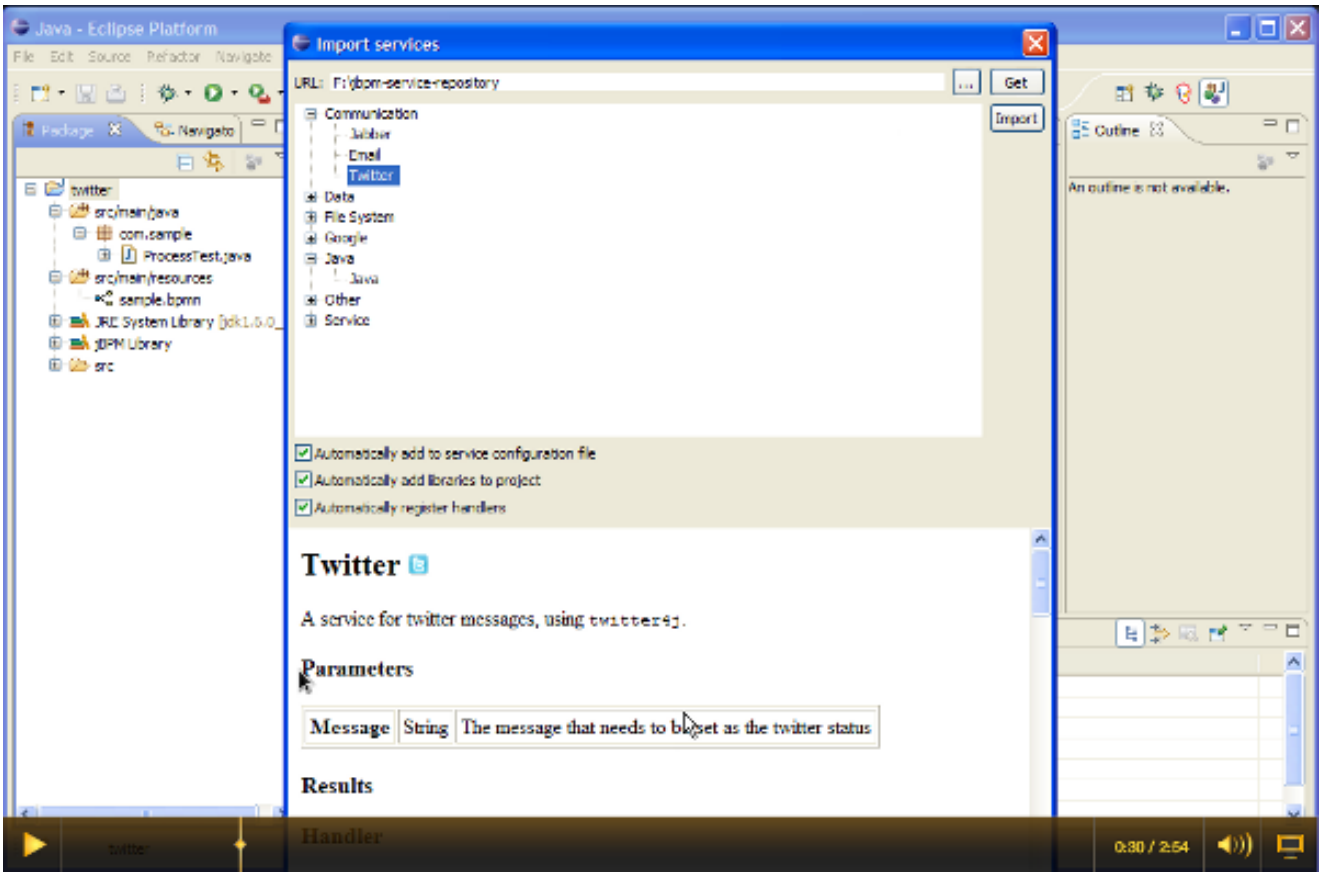


Figure 21.1.

[http://people.redhat.com/kverlaen/twitter-repository.swf]

21.4.1. Public jBPM service repository

We are building a public service repository that contains predefined services that people can use out-of-the-box if they want to:

http://docs.jboss.org/jbpm/v6.0/repository/

This repository contains some integrations for common services like Twitter integration or file system operations that you can import. Simply point the import wizard to this URL to start browsing the repository.

If you have an implementation of a common service that you would like to contribute to the community, do not hesitate to contact someone from the development team. We are always looking for contributions to extend our repository.

21.4.2. Setting up your own service repository

You can set up your own service repository and add your own services by creating a configuration file that contains the necessary information (this is an extended version of the normal work

definition configuration file as described earlier in this chapter) and putting the necessary files (like an icon, dependencies, documentation, etc.) in the right folders.

The extended configuration file contains the normal properties (like name, parameters, results and icon), with some additional ones. For example, the following extended configuration file describes the Twitter integration service (as shown in the screencast above):

```
import org.drools.core.process.core.datatype.impl.type.StringDataType;
[
  [
    "name" : "Twitter",
    "description" : "Send a Twitter message",
    "parameters" : [
      "Message" : new StringDataType()
    ],
    "displayName" : "Twitter",
                                              "eclipse:customEditor"           :
 "org.drools.eclipse.flow.common.editor.editpart.work.SampleCustomEditor",
    "icon" : "twitter.gif",
    "category" : "Communication",
    "defaultHandler" : "org.jbpm.process.workitem.twitter.TwitterHandler",
    "documentation" : "index.html",
    "dependencies" : [
      "file:./lib/jbpm-twitter.jar",
      "file:./lib/twitter4j-core-2.2.2.jar"
    ]
  ]
]
```

- The icon property should refer to a file with the given file name in the same folder as the extended configuration file (so it can be downloaded by the import wizard and used in the process diagrams). Icons should be 16x16 GIF files.
- The category property defines the category this service should be placed under when browsing the repository.
- The defaultHandler property defines the default handler implementation (i.e. the Java class that implements the `WorkItemHandler` interface and can be used to execute the service). This can automatically be registered as the handler for that service when importing the service from the repository.
- The documentation property defines a documentation file that describes what the service does and how it works. This property should refer to a HTML file with the given name in the same folder as the extended configuration file (so it can be shown by the import wizard when browsing the repository).

- The dependencies property defines additional dependencies that are necessary to execute this service. This usually includes the handler implementation JAR, but could also include additional external dependencies. These dependencies should also be located on the repository on the given location (relative to the folder where the extended configuration file is located), so they can be downloaded by the import wizard when importing the service.

The root of your repository should also contain an `index.conf` file that references all the folders that should be processed when searching for services on the repository. Each of those folders should then contain:

- An extended configuration file with the same name as the folder (e.g. `Twitter.conf`)
- The icon as references in the configuration file
- The documentation as references in the configuration file
- The dependencies as references in the configuration file (for example in a lib folder)

You can create your own hierarchical structure, because if one of those folders also contains an `index.conf` file, that will be used to scan additional sub-folders. Note that the hierarchical structure of the repository is not shown when browsing the repository using the import wizard, as the category property in the configuration file is used for that.

Chapter 22. Exception Management

22.1. Overview

This chapter will describe how to deal with unexpected behavior in your business processes using both BPMN2 and technical mechanisms.

The first section will explain Technical Exceptions: we'll go through an example that uses both BPMN2 and `WorkItemHandler` implementations in order to isolate and handle exceptions caused by a technical component. We will also explain how to modify the example to suit other use cases.

The second section will define and explain the types of (BPMN2) exceptions that can happen or be used in a business process.

22.2. Introduction

What happens to a business process when something unexpected happens during the process? Most of the time, when creating and designing a new process definition, the first step is to describe the *normative* or desirable behaviour. However, a process definition that only describes all of the normal tasks and their execution order is incomplete.

The next step is to think about what might go *wrong* when the business process is run. What would happen if any of the human or technical actors in the process do *not* respond in unexpected ways? Will any of the technical systems that the process interacts with return unexpected results -- or not return any results at all?

Deviations from the normative or "happy" flow of a business process are called *exceptions*. In some cases, exceptions might not be that unusual, such as trying to debit an empty bank account. However, some processes might contain many complex situations involving exceptions, all of which must be handled correctly.

Note

The rest of chapter assumes that you know how to create custom `<task>` nodes and how to implement and register `WorkItemHandler` implementations. More information about these topics can be found in the *Domain-specific Processes* chapter.

22.3.1. Technical Exceptions

Technical exceptions happen when a technical component of a business process acts in an unexpected way. When using Java based systems, this often results in a literal Java Exception being thrown by the system.

Technical components used in a process can fail in a way that can not be described using BPMN2. In this case, it's important to handle these exceptions in expected ways.

The following types of code might throw exceptions:

- Any code that is present in the process definition itself
- Any code that is executed during a process and is not part of jBPM
- Any code that interacts with a technical component outside of the process engine

However, those are somewhat abstract definitions. We can narrow down the places at which an exception might be thrown. Technical exceptions can occur at the following points:

1. Code present in `<scriptTask>` nodes or in the jbpm-specific `<onEntry>` and `<onExit>` elements
2. Code executed in `WorkItemHandlers` associated with `<task>` and task-type nodes

It is *much easier* to ensure correct exception handling for `<task>` and other task-type nodes that use `WorkItemHandler` implementations, than for code executed directly in a `<scriptTask>`.

Exceptions thrown by `<scriptTask>` can cause the process to fail in an unrecoverable fashion. While there are certain things that you can do to contain the damage, a process that has failed in this way can not be restarted or otherwise recovered. This also applies for other nodes in a process definition that contain script code in the node definition, such as the `<onEntry>` and `<onExit>` elements.

When jBPM engine does throw an exception generated by the code in a `<scriptTask>` the exception thrown is a special Java exception called the `WorkflowRuntimeException` that contains information about the process.

Warning

Again, exceptions generated by a `<scriptTask>` node (and other nodes containing script code) will leave the process *unrecoverable*. In fact, often, the code that starts the process itself will end up throwing the exception generated by the business process, without returning a reference to the process instance.

For this reason, it's important to limit the scope of the code in these nodes to operations dealing with process variables. Using a `<scriptTask>` to interact with a different technical component, such as a database or web service has *significant risks* because any exceptions thrown will corrupt or abort the process.

`<task>` nodes, `<serviceTask>` nodes and the rest of the `task`-type nodes are explicitly meant for interacting with other systems -- not `<scriptTask>` nodes! Use `<task>`-type nodes to interact with other technical components.

22.3.1.1. Handling exceptions in `WorkItemHandler` instances

`WorkItemHandler` classes are used when your process interacts with other technical systems. For an introduction to them and how to use them in processes, please see the *Domain-specific Processes* chapter.

While you can build exception handling into your own `WorkItemhandler` implementations, there are also two "handler decorator" classes that you can use to *wrap* a `WorkItemhandler` implementation.

These two wrapper classes include logic that is executed when an exception is thrown during the execution (or abortion) of a work item.

Table 22.1. Exception Handling `WorkItemHandler` wrapper classes

Decorator classes in the `org.jbpm.bpmn2.handler` package	Description
`SignallingTaskHandlerDecorator`	This class wraps an existing `WorkItemHandler` implementation. When the `.executeWorkItem(...)` or `.abortWorkItem(...)` methods of the original `WorkItemHandler` instance throw an exception, the `SignallingTaskHandlerDecorator` will catch the exception and signal the process instance using a configurable event type. The exception thrown will be passed as part of the event. This functionality can be used to signal an *Event SubProcess* defined in the process definition.
`LoggingTaskHandlerDecorator`	This class reacts to all exceptions thrown by the `.executeWorkItem(...)` or `.abortWorkItem(...)` `WorkItemHandler` methods by logging the errors. It also saves any exceptions thrown so to an internal list so that they can be retrieved later for inspection or further logging. Lastly, the content and format of the message logged upon an exception are configurable.

While the two classes described above should cover most cases involving exception handling, a Java developer with some experience with jBPM should be able to create a `WorkItemHandler` that executes custom code upon an exception.

If you do decide to write a custom `WorkItemHandler` that includes exception handling logic, keep the following checklist in mind:

1. Are you catching all possible exceptions that you want to (and no more, or less)?
2. Are you making sure to either complete or abort the work item after an exception has been caught? If not, are there mechanisms to retry the process later? Or are incomplete process instances acceptable?
3. >
   What other actions should be taken when an exception is caught? Do you want to simply log the exception, or is it also important to interact with other technical systems? Do you want to trigger a (BPMN2) subprocess that will handle the exception?

Important

When you use the `WorkItemManager` to signal that the work item has been completed or aborted, make sure to do that *after you've sent any signals* to the process instance. Depending on how you've defined your process, calling `WorkItemManager.completeWorkItem(...)` or `WorkItemManager.abortWorkItem(...)` will trigger the completion of the process instance. This is because the these methods trigger the jBPM process engine to continue the process flow.

In the next section, we'll describe an example that uses the `SignallingTaskHandlerDecorator` to signal an *event subprocess* when a work item handler throws an exception.

22.3.2. Technical Exception Examples

22.3.2.1. Example: service task handlers

We'll go through one example in this section, and then look quickly at how you can change it to get the behavior you want. The example involves an `<error>` event that's caught by an *(Error) Event SubProcess*.

When an *Error Event* is thrown, the containing process will be interrupted. This means that after the process flow attached to the error event has executed, the following will happen:

1. process execution will stop, and no other parts of the process will execute
2. the process instance will end up in an aborted state (instead of completed)

The example we'll go through contains an `<error>`, but at the end of the section, we'll show how you can change the process to use a `<signal>` instead.

Tip

The code and BPMN2 process definition shown in the next section are available in the `jbpm-examples` module. See the `org.jbpm.examples.exceptions.ExceptionHandlingErrorExample` class for the Java code. The BPMN2 process definition is available in the `exceptions/ExceptionHandlingWithError.bpmn2` file in the `src/main/resources` directory of the `jbpm-examples` module.

22.3.2.1.1. BPMN2 configuration

Let's look at the BPMN2 process definition first. Besides the definition of the process, the BPMN2 elements defined before the actual process definition are also important. Here's an image of the BPMN2 process that we'll be using in the example:

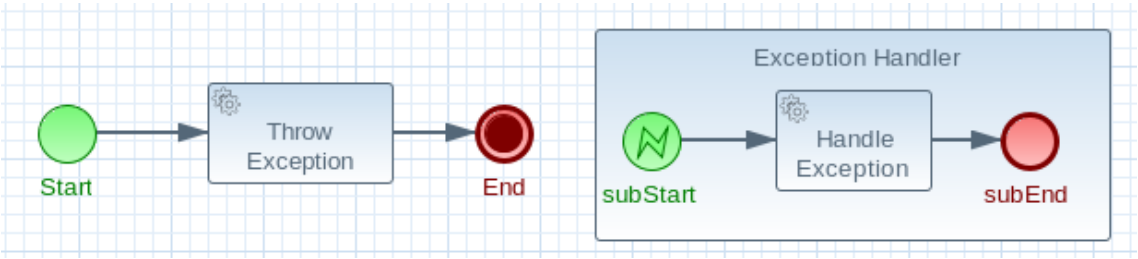


Figure 22.1.

The BPMN2 process fragment below is part of the process shown above, and contains some notes on the different BPMN2 elements.

Note

If you're viewing this on a web browser, you may need to widen or narrow your browser window in order to see the "callout" or note numbers on the right hand side of the code.

```
<itemDefinition id="_stringItem" structureRef="java.lang.String"/>          (1)
<message id="_message" itemRef="_stringItem"/>                              (2)

                          <interface              id="_serviceInterface"
name="org.jbpm.examples.exceptions.service.ExceptionService">
  <operation id="_serviceOperation" name="throwException">
    <inMessageRef>_message</inMessageRef>                                   (2)
  </operation>
</interface>
```

```
 <error id="_exception" errorCode="code" structureRef="_exceptionItem"/>      3

                              <itemDefinition              id="_exceptionItem"
structureRef="org.kie.api.runtime.process4.WorkItem"/>
 <message id="_exceptionMessage" itemRef="_exceptionItem"/>                   4

                         <interface            id="_handlingServiceInterface"
name="org.jbpm.examples.exceptions.service.ExceptionService">
   <operation id="_handlingServiceOperation" name="handleException">
     <inMessageRef>_exceptionMessage</inMessageRef>                           4
   </operation>
 </interface>

    <process  id="ProcessWithExceptionHandlingError"  name="Service  Process"
isExecutable="true" processType="Private">
   <!-- properties -->
   <property id="serviceInputItem" itemSubjectRef="_stringItem"/>             1
   <property id="exceptionInputItem" itemSubjectRef="_exceptionItem"/>        4

   <!-- main process -->
   <startEvent id="_1" name="Start" />
       <serviceTask  id="_2"  name="Throw  Exception"  implementation="Other"
operationRef="_serviceOperation">

   <!-- rest of the serviceTask element and process definition... -->

   <subProcess id="_X" name="Exception Handler" triggeredByEvent="true" >
     <startEvent id="_X-1" name="subStart">
       <dataOutput id="_X-1_Output" name="event"/>
       <dataOutputAssociation>
         <sourceRef>_X-1_Output</sourceRef>
         <targetRef>exceptionInputItem</targetRef>                            4
       </dataOutputAssociation>
       <errorEventDefinition id="_X-1_ED_1" errorRef="_exception" />          3
     </startEvent>

     <!-- rest of the subprocess definition... -->

   </subProcess>

 </process>
```

1 This `<itemDefinition>` element defines a data structure that we then use in the `serviceInputItem` property in the process.

2. This `<message>` element (1rst reference) defines a *message* that has a String as its content (as defined by the `<itemDefintion>` element on line above). The `<interface>` element below it refers to it (2nd reference) in order to define what type of content the service (defined by the `<interface>`) expects.
3. This `<error>` element (1rst reference) defines an error for use later in the process: an *Event SubProcess* is defined that is triggered by this *error* (2nd reference). The content of the error is defined by the `<itemDefintion>` element defined below the `<error>` element.
4. This `<itemDefintion>` element (1rst reference) defines an item that contains a `WorkItem` instance. The `<message>` element (2nd reference) then defines a *message* that uses this *item definition* to define its content. The `<interface>` element below that refers to the `<message>` definition (3rd reference) in order to define the type of content that the service expects.

   In the process itself, a `<property>` element (4th reference) is defined as having the content defined by the initial `<itemDefintion>`. This is helpful because it means that the *Event SubProcess* can then store the *error* it receives in that property (5th reference).

Caution

When you're using a `<serviceTask>` to call a Java class, make sure to double check the class name in your BPMN2 definition! A small typo there can cost you time later when you're trying to figure out what went wrong.

22.3.2.1.2. `SignallingTaskHandlerDecorator` and `WorkItemHandler` configuration

Now that BPMN2 process definition is (hopefully) a little clearer, we can look at how to set up jBPM to take advantage of the above BPMN2.

In the (BPMN2) process definition above, we define two different `<serviceTask>` activities. The `org.jbpm.bpmn2.handler.ServiceTaskHandler` class is the default task handler class used for `<serviceTask>` tasks. If you don't specify a `WorkItemHandler` implementation for a `<serviceTask>`, the `ServiceTaskHandler` class will be used.

In the code below, you'll see that we actually wrap or decorate the `ServiceTaskHandler` class with a `SignallingTaskHandlerDecorator` instance. We do this in order to define the what happens when the `ServiceTaskHandler` throws an exception.

In this case, the `ServiceTaskHandler` will throw an exception because it's configured to call the `ExceptionService.throwException` method, which throws an exception. (See the `_handlingServiceInterface` `<interface>` element in the BPMN2.)

In the code below, we also configure which (error) event is sent to the process instance by the `SignallingTaskHandlerDecorator` instance. The `SignallingTaskHandlerDecorator` does this when an exception is thrown in a *task*. In this case, since we've defined an `<error>` with the *error code* “code” in the BPMN2, we set the signal to `Error-code`.

Important

When signalling the jBPM process engine with an event of some sort, you should keep in mind the rules for signalling process events.

- Error events can be signalled by sending an "Error-" + <the `errorCode` attribute value> value to the session.
- Signal events can be signalled by sending the name of the signal to the session.

```
import java.util.HashMap;
import java.util.Map;

import org.jbpm.bpmn2.handler.ServiceTaskHandler;
import org.jbpm.bpmn2.handler.SignallingTaskHandlerDecorator;
import org.jbpm.examples.exceptions.service.ExceptionService;
import org.kie.api.KieBase;
import org.kie.api.io.ResourceType;
import org.kie.api.runtime.KieSession;
import org.kie.api.runtime.process.ProcessInstance;
import org.kie.internal.builder.KnowledgeBuilder;
import org.kie.internal.builder.KnowledgeBuilderFactory;
import org.kie.internal.io.ResourceFactory;

public class ExceptionHandlingErrorExample {

    public static final void main(String[] args) {
        runExample();
    }

    public static ProcessInstance runExample() {
        KieSession ksession = createKieSession();

        String eventType = "Error-code";                                    (1)
        SignallingTaskHandlerDecorator signallingTaskWrapper                (2)
            = new SignallingTaskHandlerDecorator(ServiceTaskHandler.class, eventType);
        signallingTaskWrapper.setWorkItemExceptionParameterName(ExceptionServic(3)e.exceptionPar
                ksession.getWorkItemManager().registerWorkItemHandler("Service
 Task", signallingTaskWrapper);

        Map<String, Object> params = new HashMap<String, Object>();
        params.put("serviceInputItem", "Input to Original Service");
        ProcessInstance processInstance = ksession.startProcess("ProcessWithExceptionHandlingEr
```

```
        return processInstance;
    }

    private static KieSession createKieSession() {
      KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();
                kbuilder.add(ResourceFactory.newClassPathResource("exceptions/
ExceptionHandlingWithError.bpmn2"), ResourceType.BPMN2);
        KieBase kbase = kbuilder.newKnowledgeBase();
        return kbase.newKieSession();
    }
```

1. Here we define the name of the event that will be sent to the process instance if the wrapped `WorkItemHandler` implementation throws an exception. The `eventType` string is used when instantiating the `SignallingTaskHandlerDecorator` class.
2. Then we construct an instance of the `SignallingTaskHandlerDecorator` class. In this case, we simply give it the *class name* of the `WorkItemHandler` implementation class to instantiate, but another constructor is available that we can pass an *instance* of a `WorkItemHandler` implementation to (necessary if the `WorkItemHandler` implementation does not have a no-argument constructor).
3. When an exception is thrown by the wrapped `WorkItemHandler`, the `SignallingTaskHandlerDecorator` saves it as a parameter in the `WorkItem` instance with a parameter name that we configure the `SignallingTaskHandlerDecorator` to give it (see the code below for the `ExceptionService`).

22.3.2.1.3. `ExceptionService` setup and configuration

In the BPMN2 process definition above, a service interface is defined that references the `ExceptionService` class:

```
<interface id="_handlingServiceInterface" name="org.jbpm.examples.exceptions.service.ExceptionS
    <operation id="_handlingServiceOperation" name="handleException">
```

In order to fill in the blanks a little bit, the code for the `ExceptionService` class has been included below. In general, you can specify any Java class with the default or an other no-argument constructor and have it executed during a `<serviceTask>`

```

public class ExceptionService {

  public static String exceptionParameterName = "my.exception.parameter.name";

  public void handleException(WorkItem workItem) {
```

```
            System.out.println(  "Handling  exception  caused  by  work  item
 '" + workItem.getName() + "' (id: " + workItem.getId() + ")");

    Map<String, Object> params = workItem.getParameters();
    Throwable throwable = (Throwable) params.get(exceptionParameterName);
    throwable.printStackTrace();
  }

  public String throwException(String message) {
      throw new RuntimeException("Service failed with input: " + message );
  }

  public static void setExceptionParameterName(String exceptionParam) {
      exceptionParameterName = exceptionParam;
  }

}
```

22.3.2.1.4. Changing the example to use a `<signal>`

In the example above, the thrown Error Event interrupts the process: no other flows or activities are executed once the Error Event has been thrown.

However, when a *Signal Event* is processed, the process will continue after the *Signal Event SubProcess* (or whatever other activities that the Signal Event triggers) has been executed. Furthermore, this implies that the the process will *not* end up in an aborted state, unlike a process that throws an Error Event.

In the process above, we use the `<error>` element in order to be able to use an Error Event:

```
<error id="_exception" errorCode="code" structureRef="_exceptionItem"/>
```

When we want to use a Signal Event instead, we remove that line and use a `<signal>` element:

```
<signal id="exception-signal" structureRef="_exceptionItem"/>
```

However, we must also change all references to the "`_exception`" `<error>` so that they now refer to the "`exception-signal`" `<signal>`.

That means that the `<errorEventDefintion>` element in the `<startEvent>`,

```
<errorEventDefinition id="_X-1_ED_1" errorRef="_exception" />
```

must be changed to a `<signalEventDefintion>` which would like like this:

```
<signalEventDefinition id="_X-1_ED_1" signalRef="exception-signal"/>
```

In short, we have to make the following changes to the `<startEvent>` in the Event SubProcess:

1. It will now contain a `<signalEventDefintion>` instead of a `<errorEventDefintion>`
2. The `errorRef` attribute in the `<erroEventDefintion>` is now a `signalRef` attribute in the `<signalEventDefintion>`.
3. The `id` attribute in the `signalRef` is of course now the id of the `<signal>` element. Before it was id of `<error>` element.
4. Lastly, when we signal the process in the Java code, we do not signal "`Error-code`" but simply "`exception-signal`", the `id` of the `<signal>` element.

22.3.2.2. Example: logging exceptions thrown by bad `<scriptTask>` nodes

In this section, we'll briefly describe what's possible when dealing with `<scriptTask>` nodes that throw exceptions, and then quickly go through an example (also available in the `jbpm-examples` module) that illustrates this.

22.3.2.2.1. Introduction

If you're reading this, then you probably already have a problem: you're either expecting to run into this problem because there are scripts in your process definition that might throw an exception, or you're already running a process instance with scripts that are causing a problem.

Unfortunately, if you're running into this problem, then there is not much you can do. The only thing that you *can* do is retrieve more information about exactly what's causing the problem. Luckily, when a `<scriptTask>` node causes an exception, the exception is then wrapped in a `WorkflowRuntimeException`.

What type of information is available? The `WorkflowRuntimeException` instance will contain the information outlined in the following table. All of the fields listed are available via the normal `get*` methods.

Table 22.2. Information contained in `WorkflowRuntimeException` instances.

Field name	Type	Description
`processInstanceId`	`long`	The id of the `ProcessInstance` instance in

Field name	Type	Description
		which the exception occurred. This `ProcessInstance` may not exist anymore or be available in the database if using persistence!
`processId`	`String`	The id of the process definition that was used to start the process (i.e. "`ExceptionScriptTask`" in `ksession.startProcess("ExceptionScriptTas`)
`nodeId`	`long`	The value of the (BPMN2) id attribute of the node that threw the exception.
`nodeName`	`String`	The value of the (BPMN2) name attribute of the node that threw the exception.
`variables`	`Map<String, Object>`	The map containing the variables in the process instance (*experimental*).
`message`	`String`	The short message indicating what went wrong.
`cause`	`Throwable`	The original exception that was thrown.

22.3.2.2.2. Example: Exceptions thrown by a `<scriptTask>`.

The following code illustrates how to extract extra information from a process instance that throws a `WorkflowRuntimeException` exception instance.

```
import org.jbpm.workflow.instance.WorkflowRuntimeException;
import org.kie.api.KieBase;
import org.kie.api.io.ResourceType;
import org.kie.api.runtime.KieSession;
import org.kie.api.runtime.process.ProcessInstance;
import org.kie.internal.builder.KnowledgeBuilder;
import org.kie.internal.builder.KnowledgeBuilderFactory;
import org.kie.internal.io.ResourceFactory;
```

```
public class ScriptTaskExceptionExample {

    public static final void main(String[] args) {
        runExample();
    }

    public static void runExample() {
        KieSession ksession = createKieSession();
        Map<String, Object> params = new HashMap<String, Object>();
        String varName = "var1";
        params.put( varName , "valueOne" );
        try {
            ProcessInstance processInstance = ksession.startProcess("ExceptionScriptTask", para
        } catch( WorkflowRuntimeException wfre ) {
            String msg = "An exception happened in "
                    + "process instance [" + wfre.getProcessInstanceId()
                    + "] of process [" + wfre.getProcessId()
                    + "] in node [id: " + wfre.getNodeId()
                    + ", name: " + wfre.getNodeName()
                            + "] and variable " + varName + " had the value
 [" + wfre.getVariables().get(varName)
                    + "]";
            System.out.println(msg);
        }
    }

    private static KieSession createKieSession() {
      KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();
                kbuilder.add(ResourceFactory.newClassPathResource("exceptions/
ScriptTaskException.bpmn2"), ResourceType.BPMN2);
        KieBase kbase = kbuilder.newKnowledgeBase();
        return kbase.newKieSession();
    }

}
```

22.4.1. Business Exceptions

Business Exceptions are exceptions that are designed and managed in the BPMN2 specification of a business process. In other words, Business Exceptions are exceptions which happen at the process or workflow level, and are not related to the technical components.

Many of the elements in BPMN2 related to Business Exceptions are related to *Compensation* and *Business Transactions*. Compensation, in particular, is complexer than many other parts of the BPMN2 specification.

Full support for *compensation* and *business transactions* is expected with the release of jBPM 6.1 or 6.2. Once that has been implemented, this section will contain more information about using those BPMN2 features with jBPM.

22.4.1.1. Business Exceptions elements in BPMN2

The following attempts to briefly describe Compensation and Business Transaction related elements in BPMN2. For more complete information about these elements and their uses, see the BPMN2 specification, Bruce Silver's book `BPMN Method and Style` or any of the other available books about the use of BPMN2.

Table 22.3. BPMN2 Exception Handling Elements

BPMN2 Element types	Description
Errors	Error Events can be used to signal when a process has encountered an unexpected situation: signalling an error is often called *throwing* an error. Boundary Error Events in a different part of the process can then be used to *catch* the error and initiate a sequence of activities to handle the exception. Errors themselves can be extended with extra information that is passed from the throwing to catching event. This is done with the use of an Item Definition.
Compensation	Exception handling activities *associated* with the normal activities in a Business Transaction are triggered by *Compensation Events*. There are 3 types of compensation events: Intermediate (a.k.a. Boundary) (catch) events, Start (catch) events, and Intermediate or End (throw) events. Compensation Boundary (catch) events may only be attached to activities (e.g. tasks) that could cause an exception. These Boundary events are then *associated* (not linked!) with a Task that will be executed if the Boundary event catches a (thrown) Compensation signal.

BPMN2 Element types	Description
	Start (catch) events are used when defining an *Compensation Event SubProcess*, which requires them in order to be able to catch a (thrown) Compensation signal. Compensation Intermediate and End events are used in order to throw Compensation Events. These events often follow decision nodes that determine whether the workflow executed up to that point has succeeded. If not, the path including the Intermediate or End Event is chosen in order to trigger Compensatoin for the activities that did not succeed.

BPMN2 contains a number of constructs to model exceptions in business processes. There are several advantages to doing exception handling at the business process level (as opposed to handling it with code):

- *Transparency*
  - Being able to quickly see what happens in exceptional situations means that the results and performance of a process is more easily monitored and measured.
  - It also increases how easily a process can be implemented as well as how maintainable a process definition is.
- *Business Logic Isolation*
  - Again, the idea behind using a business process is to isolate the business logic from the technical code. This simplifies the complexity of the system and increases how quickly you can create new business processes and change existing ones.
  - Implementing exception handling at a technical level often takes more time because it's often complexer and specific to a system.

22.4.1.2. Designing a workflow with Business Exceptions

Where are business exceptions likely to occur? There is academic research on this, but some possible examples are:

- When an interaction with an external party or 3rd party system does not go as planned
- When you can not fully check the the input data in your process (like a client's address information, for example)

- In general, if there are parts of your process that are particularly dependent on one of the following, a business exception will be a good idea:
  - Company policy or policy governing certain (in-house) procedures
  - Laws governing the business process (such as age requirements, for example)

Chapter 23. Flexible Processes

Case management and its relation to BPM is a hot topic nowadays. There definitely seems to be a growing need amongst end users for more flexible and adaptive business processes, without ending up with overly complex solutions. Everyone seems to agree that using a process-centric approach only in many cases leads to complex solutions that are hard to maintain. The "knowledge workers" no longer want to be locked into rigid processes but wants to have the power and flexibility to regain more control over the process themselves.

The term case management is often used in that context. Without trying to give a precise definition of what it might or might not mean, as this has been a hot topic for discussion, it refers to the basic idea that many applications in the real world cannot really be described completely from start to finish (including all possible paths, deviations, exceptions, etc.). Case management takes a different approach: instead of trying to model what should happen from start to finish, let's give the end user the flexibility to decide what should happen at runtime. In its most extreme form for example, case management doesn't even require any process definition at all. Whenever a new case comes in, the end user can decide what to do next based on all the case data.

A typical example can be found in healthcare (clinical decision support to be more precise), where care plans can be used to describe how patients should be treated in specific circumstances, but people like general practitioners still need to have the flexibility to add additional steps and deviate from the proposed plan, as each case is unique. And there are similar examples in claim management, help desk support, etc.

So, should we just throw away our BPM system then? No! Even at its most extreme form (where we don't model any process up front), you still need a lot of the other features a BPM system (usually) provides: there still is a clear need for audit logs, monitoring, coordinating various services, human interaction (e.g. using task forms), analysis, etc. And, more importantly, many cases are somewhere in between, or might even evolve from case management to more structured business process over time (when we for example try to extract common approaches from many cases). If we can offer flexibility as part of our processes, can't we let the users decide how and where they would like to apply it?

Let me give you two examples that show how you can add more and more flexibility to your processes. The first example shows a care plan that shows the tasks that should be performed when a patient has high blood pressure. While a large part of the process is still well-structured, the general practitioner can decide himself which tasks should be performed as part of the sub-process. And he also has the ability to add new tasks during that period, tasks that were not defined as part of the process, or repeat tasks multiple times, etc. The process uses an ad-hoc sub-process to model this kind of flexibility, possibly augmented with rules or event processing to help in deciding which fragments to execute.

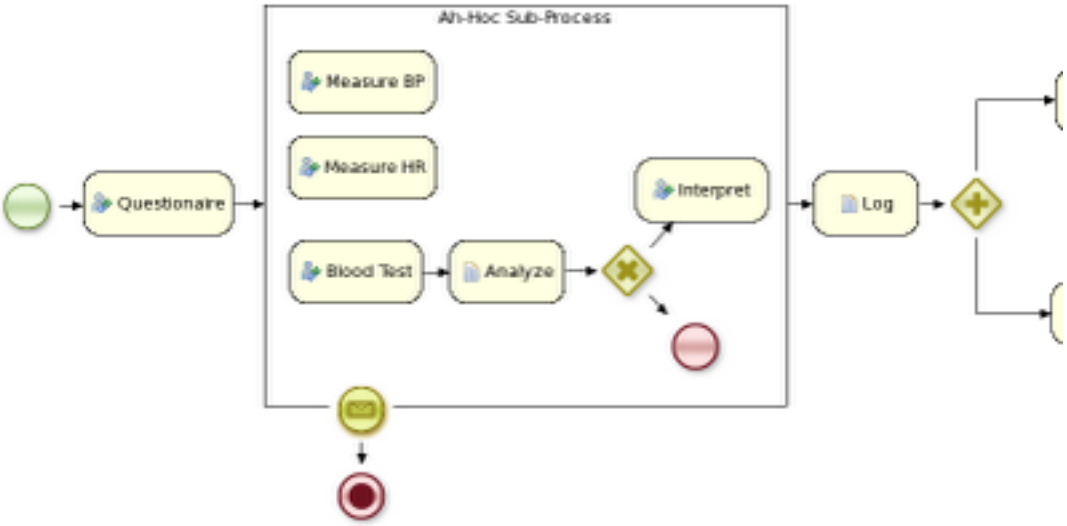


Figure 23.1. Healthcare: high blood pressure

The second example actually goes a lot further than that. In this example, an internet provider could define how cases about internet connectivity problems will be handled by the internet provider. There are a number of actions the case worker can select from, but those are simply small process fragments. The case worker is responsible for selecting what to do next and can even add new tasks dynamically. As you can see, there is not process from start to finish anymore, but the user is responsible for selecting which process fragments to execute.

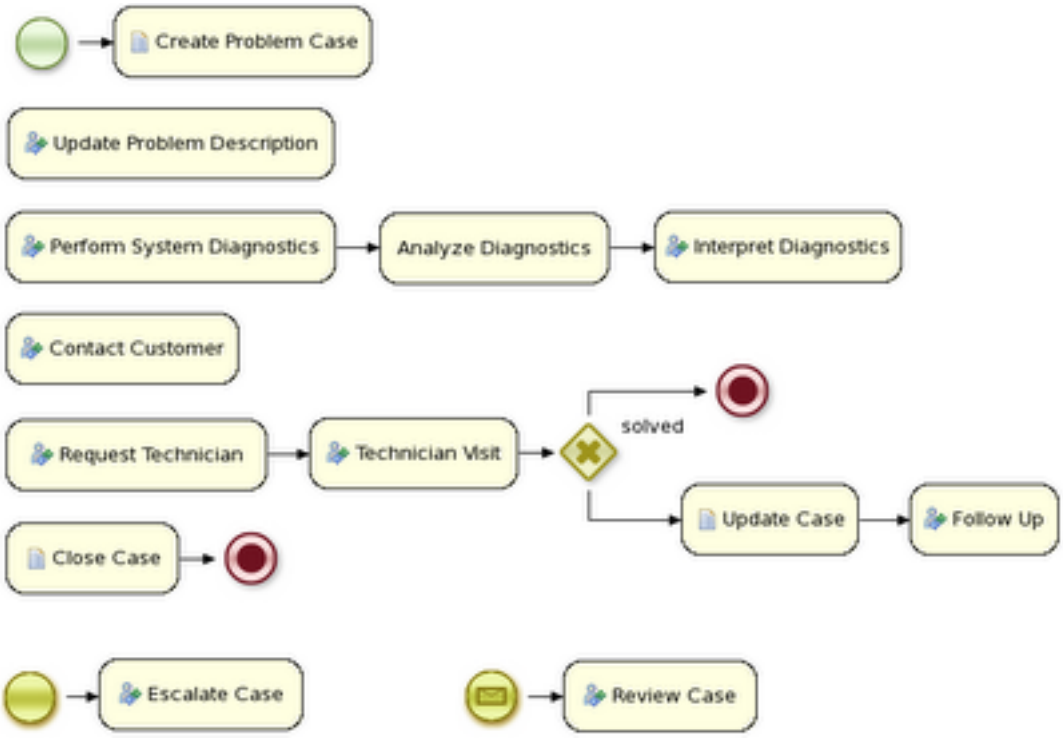


Figure 23.2. Telecom: process fragments

And in its most extreme form, we even allow you to create case instances without a process definition, where what needs to be performed is selected purely at runtime. This however doesn't mean you can't figure out anymore what 's actually happening. For example, meetings can be very ad hoc and dynamic, but we usually want a log of what was actually discussed. The following screenshot shows how our regular audit view can still be used in this case, and the end user could then for example get a lot more info about what actually happened by looking at the data associated with each of those steps. And maybe, over time, we can even automate part of that by using a semi-structured process.

Figure 23.3. Audit log for dynamic case

Chapter 24. Concurrency and asynchronous execution

24.1. Concurrency

In the following text, we will refer to two types of "multi-threading": logical and technical. Technical multi-threading is what happens when multiple threads or processes are started on a computer, for example by a Java or C program. Logical multi-threading is what we see in a BPM process after the process reaches a parallel gateway, for example. From a functional standpoint, the original process will then split into two processes that are executed in a parallel fashion.

Of course, the jBPM engine supports logical multi-threading: for example, processes that include a parallel gateway. We've chosen to implement logical multi-threading using one thread: a jBPM process that includes logical multi-threading will only be executed in one technical thread. The main reason for doing this is that multiple (technical) threads need to be be able to communicate state information with each other if they are working on the same process. This requirement brings with it a number of complications. While it might seem that multi-threading would bring performance benefits with it, the extra logic needed to make sure the different threads work together well means that this is not guaranteed. There is also the extra overhead incurred because we need to avoid race conditions and deadlocks.

24.1.1. Engine execution

In general, the jBPM engine executes actions in serial. For example, when the engine encounters a script task in a process, it will synchronously execute that script and wait for it to complete before continuing execution. Similarly, if a process encounters a parallel gateway, it will sequentially trigger each of the outgoing branches, one after the other. This is possible since execution is almost always instantaneous, meaning that it is extremely fast and produces almost no overhead. As a result, the user will usually not even notice this. Similarly, action scripts in a process are also synchronously executed, and the engine will wait for them to finish before continuing the process. For example, doing a Thread.sleep(...) as part of a script will not make the engine continue execution elsewhere but will block the engine thread during that period.

The same principle applies to service tasks. When a service task is reached in a process, the engine will also invoke the handler of this service synchronously. The engine will wait for the completeWorkItem(...) method to return before continuing execution. It is important that your service handler executes your service asynchronously if its execution is not instantaneous.

An example of this would be a service task that invokes an external service. Since the delay in invoking this service remotely and waiting for the results might be too long, it might be a good idea to invoke this service asynchronously. This means that the handler will only invoke the service and will notify the engine later when the results are available. In the mean time, the process engine then continues execution of the process.

Human tasks are a typical example of a service that needs to be invoked asynchronously, as we don't want the engine to wait until a human actor has responded to the request. The human task handler will only create a new task (on the task list of the assigned actor) when the human task node is triggered. The engine will then be able to continue execution on the rest of the process (if necessary) and the handler will notify the engine asynchronously when the user has completed the task.

24.1.2. Multiple knowledge sessions and persistence

The simplest way to run multiple processes is to run them all using one knowledge session. However, there are cases in which it's necessary to run multiple processes in different knowledge sessions, even in different (technical) threads. Both are supported by jBPM.

When we add persistence (using a database, for example) to a situation in which we have multiple knowledge sessions (and processes), there is a guideline that users should be aware of. The following paragraphs explain why this guideline is important to follow.

Tip

Please make sure to use a database that allows row-level locks as well as table-level locks.

For example, a user could have a situation in which there are 2 (or more) threads running, each with its own knowledge session instance. On each thread, jBPM processes are being started using the local knowledge session instance.

In this use case, a race condition exists in which both thread A and thread B will have coincidentally simultaneously finished a process. At this point, because persistence is being used, both thread A and B will be committing changes to the database. If row-level locks are not possible, then the following situation can occur:

- Thread A has a lock on the ProcessInstanceInfo table, having just committed a change to that table.
- Thread A wants a lock on the SessionInfo table in order to commit a change there.
- Thread B has the opposite situation: it has a lock on the SessionInfo table, having just committed a change there.
- Thread B wants a lock on the ProcessInstanceInfo table, even though Thread A already has a lock on it.

This is a deadlock situation which the database and application will not be able to solve. However, if row-level locks are possible (and enabled!!) in the database (and tables used), then this situation will not occur.

24.2. Asynchronous execution

24.2.1. Asynchronous handlers

How can we implement an asynchronous service handler? To start with, this depends on the technology you're using. If you're only using Java, you could execute the actual service in a new thread:

```
public class MyServiceTaskHandler implements WorkItemHandler {


  public void executeWorkItem(WorkItem workItem, WorkItemManager manager) {
    new Thread(new Runnable() {
      public void run() {
        // Do the heavy lifting here ...
      }
    }).start();
  }

  public void abortWorkItem(WorkItem workItem, WorkItemManager manager) {
  }
}
```

It's advisable to have your handler contact a service that executes the business operation, instead of having it perform the actual work. If anything goes wrong with a business operation, it doesn't affect your process. The loose coupling that this provides also gives you greater flexibility in reusing services and developing them.

For example, you can have your human task handler simply invoke the human task service to add a task there. To implement an asynchronous handler, you usually have to simply do an asynchronous invocation of this service. This usually depends on the technology you use to do the communication, but this might be as simple as asynchronously invoking a web service, or sending a JMS message to the external service.

24.2.2. jbpm executor

In version 6, jBPM introduces new component called jbpm executor which provides quite advanced features for asynchronous execution. It delivers generic environment for background execution of commands. Commands are nothing more than business logic encapsulated within simple interface. It does not have any process runtime related information, that means no need to complete work items, or anything of that sort. It purely focuses on the business logic to be executed. It receives data via CommandContext and returns results of the execution with ExecutionResults.

Before looking into details on jBPM support for asynchronous execution let's look at what are the common requirements for such execution:

- allows asynchronous execution of given piece of business logic
- allows to retry in case of resources are temporarily unavailable e.g. external system interaction
- allows to handle errors in case all retries have been attempted
- provides cancellation option
- provides history log of execution

When confronting these requirements with the "simple async handler" (executed as separate thread) you can directly notice that all of these would need to be implemented all over again by different systems. Due to that a common, generic component has been provided out of the box to simplify and empower usage.

jBPM executor operates on commands, which are essential piece of code that is going to be executed as background job.

```
/**
 * Executor's Command are dedicated to contain purely business logic that should be executed.
 * It should not have any reference to underlying process engine and should not be concerned
 * with any process runtime related logic such us completing work item, sending signals, etc.
 * <br/>
 * Information that are taken from process will be delivered as part of data instance of
                            *                            <code>CommandContext</
code>. Depending on the execution context that data can vary but
 * in most of the cases following will be given:
 * <ul>
 *  <li></li>
 *  <li>businessKey - usually unique identifier of the caller</li>
    *        <li>callbacks    -    FQCN    of    the    <code>CommandCollback</
code> that shall be used on command completion</li>
 * </ul>
 * When executed as part of the process (work item handler) additional data can be expected:
 * <ul>
 * <li>workItem - the actual work item that is being executed with all its parameters</
li>
 *  <li>processInstanceId - id of the process instance that triggered this work</
li>
 *  <li>deploymentId - if given process instance is part of an active deployment</
li>
 * </ul>
 * Important note about implementations is that it shall always be possible to be initialized w
 * as executor service is an async component so it will initialize the command on demand using
 * In case there is a heavy logic on initialization it should be placed in another service impl
 * can be looked up from within command.
 */
public interface Command {
```

```
    /**
     * Executed this command's logic.
     * @param ctx - contextual data given by the executor service
     * @return returns any results in case of successful execution
    * @throws Exception in case execution failed and shall be retried if possible
     */
    public ExecutionResults execute(CommandContext ctx) throws Exception;
}
```

Looking at the interface above, there is no specific integration with the jBPM runtime engine, it's decoupled from it to put main focus on the actual logic that shall be executed as part of that command rather to worry about integration with process engine. This design promotes reuse of already existing logic by simply wrapping it with Command implementation.

Input data is transferred from process engine to command via CommandContext. It acts purely as data transfer object and puts single requirement on the data it holds - all objects must be serializable.

```
/**
 * Data holder for any contextual data that shall be given to the command upon execution.
 * Important note that every object that is added to the data container must be serializable
 * meaning it must implement <code>java.io.Seriazliable</code>
 *
 */
public class CommandContext implements Serializable {

    private static final long serialVersionUID = -1440017934399413860L;
    private Map<String, Object> data;

    public CommandContext() {
        data  = new HashMap<String, Object>();
    }

    public CommandContext(Map<String, Object> data) {
        this.data = data;
    }

    public void setData(Map<String, Object> data) {
        this.data = data;
    }

    public Map<String, Object> getData() {
        return data;
    }

    public Object getData(String key) {
        return data.get(key);
```

```
    }

    public void setData(String key, Object value) {
        data.put(key, value);
    }

    public Set<String> keySet() {
        return data.keySet();
    }

    @Override
    public String toString() {
        return "CommandContext{" + "data=" + data + '}';
    }
}
```

Next outcome is provided to process engine via ExecutionResults, which is very similar in nature to the CommandContext and acts as data transfer object.

```
/**
 * Data holder for command's result data. Whatever command produces should be placed in
 * this results so they can be later on referenced by name by the requester - e.g. process inst
 *
 */
public class ExecutionResults implements Serializable {

    private static final long serialVersionUID = -1738336024526084091L;
    private Map<String, Object> data = new HashMap<String, Object>();

    public ExecutionResults() {
    }

    public void setData(Map<String, Object> data) {
        this.data = data;
    }

    public Map<String, Object> getData() {
        return data;
    }

    public Object getData(String key) {
        return data.get(key);
    }

    public void setData(String key, Object value) {
        data.put(key, value);
    }
```

```
    public Set<String> keySet() {
        return data.keySet();
    }

    @Override
    public String toString() {
        return "ExecutionResults{" + "data=" + data + '}';
    }


}
```

Executor covers all requirements listed above and provides user interface as part of jbpm console and kie workbench (kie-wb) applications.

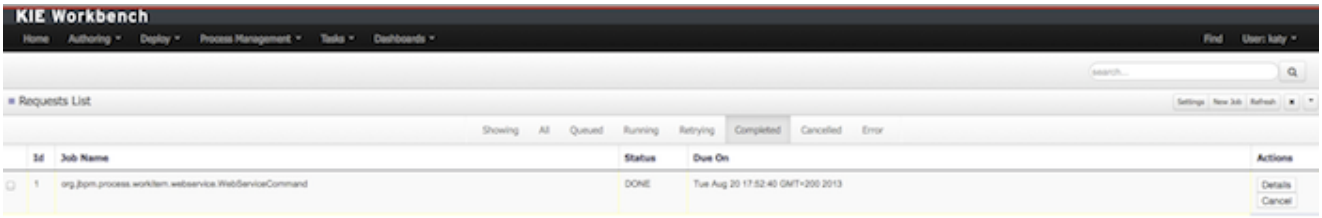


Figure 24.1.

Above screenshot illustrates history view of executor's job queue. As can be seen on it there are several options available:

- view details of the job
- cancel given job
- create new job

24.2.2.1. WorkItemHandler backed with jbpm executor

jBPM (again in version 6) provides an out of the box async work item handler that is backed by the jbpm executor. So by default all features that executor delivers will be available for background execution within process instance. AsyncWorkItemHandler can be configured in two ways:

- as generic handler that expects to get the command name as part of work item parameters
- as specific handler for given type of work item - for example web service

Option 1 is by default configured for jbpm console and kie-wb web applications and is registered under **async** name in every ksession that is bootstrapped within the applications. So whenever

there is a need to execute some logic asynchronously following needs to be done at modeling time (using jbpm web designer):

- specify async as TaskName property
- create data input called CommandClass
- assign fully qualified class name for the CommandClass data input

Next follow regular way to complete process modeling. Note that all data inputs will be transferred to executor so they must be serializable.

Second option allows to register different instances of AsyncWorkItemHandler for different work items. Since it's registered for dedicated work item most likely the command will be dedicated to that work item as well. If so CommandClass can be specified on registration time instead of requiring it to be set as work item parameters. To register such handlers for jbpm console or kie-wb additional class is required to inform what shall be registered. A CDI bean that implements WorkItemHandlerProducer interface needs to be provided and placed on the application classpath so CDI container will be able to find it. Then at modeling time TaskName property needs to be aligned with those used at registration time.

24.2.2.2. Configuration

jbpm executor is configurable to allow fine tuning of its environment. In general jbpm executor runs as a thread pool that periodically checks for waiting jobs and executes them when needed. Configuration of jbpm executor is done via system properties:

- org.kie.executor.disabled = true|false - allows to completely disable executor component
- org.kie.executor.pool.size = Integer - allows to specify thread pool size where default it 1
- org.kie.executor.retry.count = Integer - allows to specify number of retries in case of errors while running a job
- org.kie.executor.interval = Integer - allows to specify interval (in seconds) that executor will use while checking for waiting jobs where default is 3 seconds

Chapter 25. Release Notes

25.1. New and Noteworthy in KIE API 6.0.0

25.1.1. New KIE name

KIE is the new umbrella name used to group together our related projects; as the family continues to grow. KIE is also used for the generic parts of unified API; such as building, deploying and loading. This replaces the droolsjbpm and knowledge keywords that would have been used before.

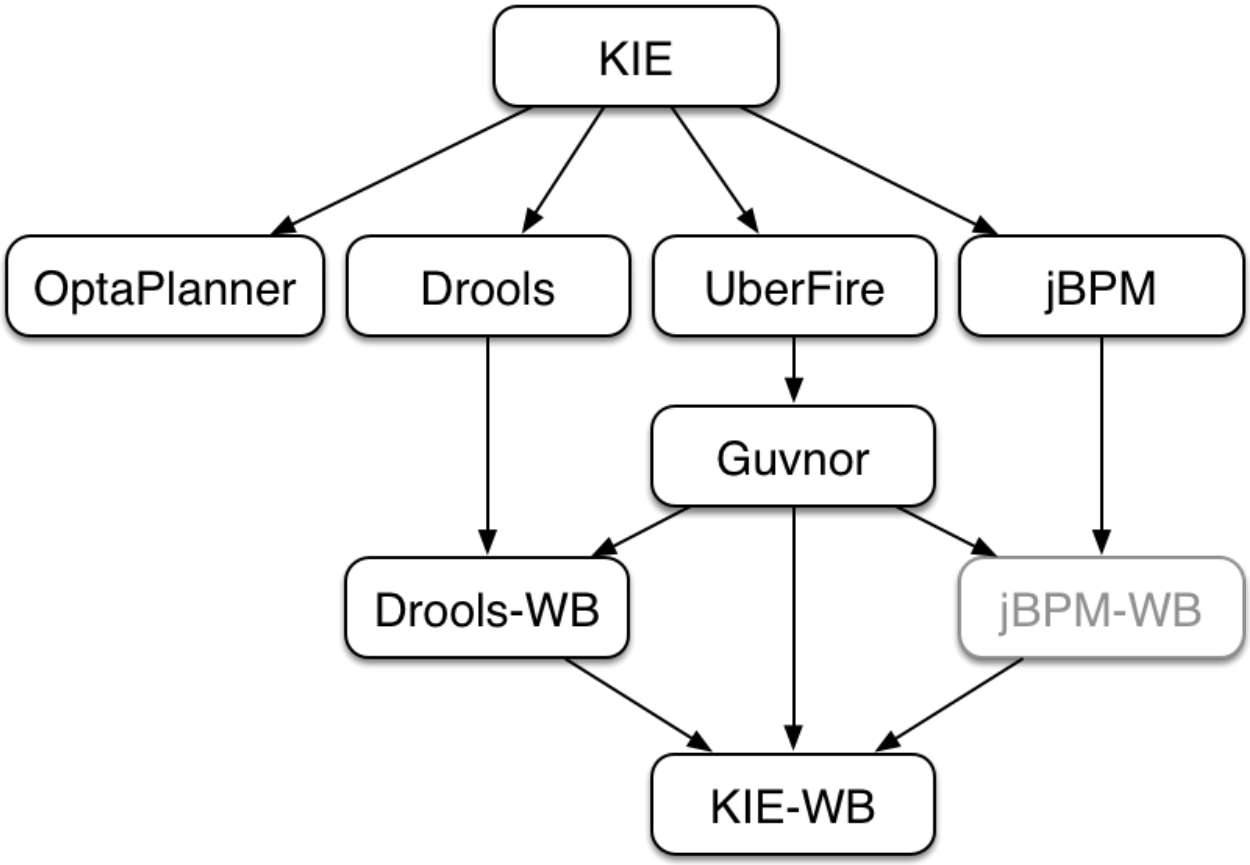


Figure 25.1. KIE Anatomy

25.1.2. Maven aligned projects and modules and Maven Deployment

One of the biggest complaints during the 5.x series was the lack of defined methodology for deployment. The mechanism used by Drools and jBPM was very flexible, but it was too flexible. A big focus for 6.0 was streamlining the build, deploy and loading (utilization) aspects of the system. Building and deploying activities are now aligned with Maven and Maven repositories.

The utilization for loading rules and processess is now convention and configuration oriented, instead of programmatic, with sane defaults to minimise the configuration.

Projects can be built with Maven and installed to the local M2_REPO or remote Maven repositories. Maven is then used to declare and build the classpath of dependencies, for KIE to access.

25.1.3. Configuration and convention based projects

The 'kmodule.xml' provides declarative configuration for KIE projects. Conventions and defaults are used to reduce the amount of configuration needed.

Example 25.1. Declare KieBases and KieSessions

```
<kmodule xmlns="http://jboss.org/kie/6.0.0/kmodule">
  <kbase name="kbase1" packages="org.mypackages>
    <ksession name="ksession1"/>
  </kbase>
</kmodule>
```

Example 25.2. Utilize the KieSession

```
KieServices ks = KieServices.Factory.get();
KieContainer kContainer = ks.getKieClasspathContainer();

KieSession kSession = kContainer.newKieSession("ksession1");
kSession.insert(new Message("Dave", "Hello, HAL. Do you read me, HAL?"));
kSession.fireAllRules();
```

25.1.4. KieBase Inclusion

It is possible to include all the KIE artifacts belonging to a KieBase into a second KieBase. This means that the second KieBase, in addition to all the rules, function and processes directly defined into it, will also contain the ones created in the included KieBase. This inclusion can be done declaratively in the kmodule.xml file

Example 25.3. Including a KieBase into another declaratively

```
<kmodule xmlns="http://jboss.org/kie/6.0.0/kmodule">
  <kbase name="kbase2" includes="kbase1">
    <ksession name="ksession2"/>
  </kbase>
</kmodule>
```

or programmatically using the `KieModuleModel`.

Example 25.4. Including a KieBase into another programmatically

```
KieModuleModel kmodule = KieServices.Factory.get().newKieModuleModel();
KieBaseModel kieBaseModel1 = kmodule.newKieBaseModel("KBase2").addInclude("KBase1");
```

25.1.5. KieModules, KieContainer and KIE-CI

Any Maven produced JAR with a 'kmodule.xml' in it is considered a KieModule. This can be loaded from the classpath or dynamically at runtime from a Resource location. If the kie-ci dependency is on the classpath it embeds Maven and all resolving is done automatically using Maven and can access local or remote repositories. Settings.xml is obeyed for Maven configuration.

The KieContainer provides a runtime to utilize the KieModule, versioning is built in throughout, via Maven. Kie-ci will create a classpath dynamically from all the Maven declared dependencies for the artifact being loaded. Maven LATEST, SNAPSHOT, RELEASE and version ranges are supported.

Example 25.5. Utilize and Run - Java

```
KieServices ks = KieServices.Factory.get();
KieContainer kContainer = ks.newKieContainer(
        ks.newReleaseId("org.mygroup", "myartefact", "1.0") );

KieSession kSession = kContainer.newKieSession("ksession1");
kSession.insert(new Message("Dave", "Hello, HAL. Do you read me, HAL?"));
kSession.fireAllRules();
```

KieContainers can be dynamically updated to a specific version, and resolved through Maven if KIE-CI is on the classpath. For stateful KieSessions the existing sessions are incrementally updated.

Example 25.6. Dynamically Update - Java

```
KieContainer kContainer.updateToVersion(
                ks.newReleaseId("org.mygroup", "myartefact", "1.1") );
```

25.1.6. KieScanner

The `KieScanner` is a Maven-oriented replacement of the KnowledgeAgent present in Drools 5. It continuously monitors your Maven repository to check if a new release of a Kie project has

been installed and if so, deploys it in the `KieContainer` wrapping that project. The use of the `KieScanner` requires kie-ci.jar to be on the classpath.

A `KieScanner` can be registered on a `KieContainer` as in the following example.

Example 25.7. Registering and starting a KieScanner on a KieContainer

```
KieServices kieServices = KieServices.Factory.get();
ReleaseId releaseId = kieServices.newReleaseId( "org.acme", "myartifact", "1.0-SNAPSHOT" );
KieContainer kContainer = kieServices.newKieContainer( releaseId );
KieScanner kScanner = kieServices.newKieScanner( kContainer );

// Start the KieScanner polling the Maven repository every 10 seconds
kScanner.start( 10000L );
```

In this example the `KieScanner` is configured to run with a fixed time interval, but it is also possible to run it on demand by invoking the `scanNow()` method on it. If the `KieScanner` finds, in the Maven repository, an updated version of the Kie project used by that `KieContainer` it automatically downloads the new version and triggers an incremental build of the new project. From this moment all the new `KieBase`s and `KieSession`s created from that `KieContainer` will use the new project version.

25.1.7. Hierarchical ClassLoader

The CompositeClassLoader is no longer used; as it was a constant source of performance problems and bugs. Traditional hierarchical classloaders are now used. The root classloader is at the KieContext level, with one child ClassLoader per namespace. This makes it cleaner to add and remove rules, but there can now be no referencing between namespaces in DRL files; i.e. functions can only be used by the namespaces that declared them. The recommendation is to use static Java methods in your project, which is visible to all namespaces; but those cannot (like other classes on the root KieContainer ClassLoader) be dynamically updated.

25.1.8. Legacy API Adapter

The 5.x API for building and running with Drools and jBPM is still available through Maven dependency "knowledge-api-legacy5-adapter". Because the nature of deployment has significantly changed in 6.0, it was not possible to provide an adapter bridge for the KnowledgeAgent. If any other methods are missing or problematic, please open a JIRA, and we'll fix for 6.1

25.1.9. KIE Documentation

While a lot of new documentation has been added for working with the new KIE API, the entire documentation has not yet been brought up to date. For this reason there will be continued

references to old terminologies. Apologies in advance, and thank you for your patience. We hope those in the community will work with us to get the documentation updated throughout, for 6.1

25.2. New and Noteworthy in jBPM 6.0.0

25.2.1. KIE API

A new public API has been created for interacting with the core engine (shared between jBPM and Drools). This not only handles runtime operations to start processes, etc. but also instantiating sessions, registering listeners, configuration, etc.

New APIs were added in various areas, like for example the TaskService interface was moved to the public API, the new RuntimeManager was introduced and a lot of related interfaces and classes were added as well.

For backwards compatibility with v5, a knowledge-api JAR has been constructed, that implements the old v5 knowledge-api interfaces on top of the v6 engine. Make sure to include this JAR in your classpath if you want to keep using the v5 API.

25.2.2. jBPM Core Engine

The execution engine itself has (mostly) remained the same, although we've done various improvements in the following areas:

- RuntimeManager: instantiating a ksession (and an associated task service) has been simplified significantly, by introducing a runtime manager where you can simply ask for a reference to a ksession whenever you need it. The Runtime manager is responsible for initialization, configuration and disposal of the ksession (and task service), and three predefined strategies are available:
  - Singleton: the RuntimeManager reused the same ksession for all requests (and executes the requests in sequence, one at a time)
  - Session per request: the RuntimeManager instantiates a new ksession per request that will be used for executing that request and disposed at the end. Each request will receive its own ksession and they can all be executed in parallel.
  - Session per process instance: the RuntimeManager reuses the same ksession for all requests related to one specific process instance. This might be necessary if you are storing data inside your session (for example for rule evaluations) that you need to be available later in the process as well. Note that the session is disposed after each command but stored in the database so it can be restored whenever necessary.
- jBPM Services (CDI): To simplify integration of jBPM inside CDI-based applications, the jbpm-services module contains various CDI services that you can configure and use inside your application simply by injecting the necessary services (like a RuntimeManager or TaskService for example) inside your application, making integration easier than ever.

- Timer service: a Quartz-based timer service is now available, that allows you to dispose your session at any point in time, and the timer service will be responsible for rehydrating a ksession whenever a timer should be fired. This timer service also works in a clustered environment, where multiple nodes can work together on sharing the work load but timers will only be fired once by one of the nodes.
- Exception and compensation management: various improvements in this area allow you to use more BPMN2 constructs related to exception and compensation management in your processes, and various strategies have been extended and documented to better handle exceptions in different ways.
- Asynchronous handlers: asynchronous execution of interaction with external services can now be implemented by reusing the asynchronous job executor.
- Asynchronous auditing using JMS: audit logging can now also be done asynchronously by sending the events to a JMS queue rather than persisting them as part of the engine transaction.

The task service has been refactored significantly as well, and the TaskService APIs have been moved to the public kie-api. Although the TaskService interfaces themselves haven't changed a lot, the internal implementation has been simplified. Auditing for the task-related operations (similar to the runtime engine auditing) has been added.

By default, a local task service will always be used by a ksession to perform various task-related operations (creating a task, being notified when a task is completed). Setting up a remote singleton task service and connecting multiple ksessions to this (using Mina or HornetQ) as was possible in jBPM5 is no longer possible, as it introduces more challenges that it brings advantages. Since the jBPM execution service now also provides a remote API for all task-related operations, we believe this setup is no longer necessary, and has been replaced by the use of a local task service in all use cases.

25.2.3. jBPM Designer

jBPM designer has been reimplemented and is fully integrated into the workbench. It now easily integrates with many of the workbench services available. In addition, the following features were added/improved on:

- Improvement of jBPM Simulation engine and the UI. Added ability to specify simulation properties on more node type and added more results graphs such as the the Total Cost graph.
- Many updates to the Designer Toolbar for usability purposes.
- Visual Validation update - it now is a real-time visualization of issues done during process modeling.
- Ability to generate task forms for specific task node.
- Integration with the jBPM Form Modeler for both task and process forms.

- Update to process properties - added grouping of properties into sections making it more user friendly to find properties.
- Update to Object Library - added type specific tasks to palette (rather than having to morph to a certain type after adding a task to the canvas).
- Save/Remove/CopyDelete feature have been added directly into Designer and integrate with the workbench services for those operations.
- Autosave - option for users to enable auto-saving of their business process during modeling.
- Two new default Service Tasks (REST and Web Services)

25.2.4. jBPM Data Modeler

A new web-based data modeler is integrated in the workbench, which allows non-technical users to create data models (to be used in your processes and rules) in a user-friendly manner. These models are saved as Java classes (with the necessary annotations) in the project and added to the kjar upon build and deploy. Check the chapter on Data Modeler in the Workbench Part for all the details.

25.2.5. Form Modeler

A new web-based form modeler is integrated in the workbench, which allows non-technical users to create forms (for starting processes and/or completing human task). The form modeler is a WYSIWYG editor where you can drag and drop form elements (text boxes, labels, etc.), link it to data that is expected as input or output of the form, customize properties of each element and the layout, etc. These forms are then shown when starting the process or completing a task, integrated into the appropriate runtime views. Check the chapter on Form Modeler in the Workbench Part for all the details.

25.2.6. jBPM Console

The jBPM console has been reimplemented and is integrated into the workbench as well. It provides similar features as jBPM5 (starting process instances, inspecting current state and variables, looking at task lists) but is now much more powerful and exposes a lot more features. Check the chapter on Process and Task Management in the Workbench Part for all the details.

25.2.7. BAM / Reporting

A new web-based monitoring and reporting tool has been integrated in the workbench. This displays charts, tables, etc. about the current status of your application(s). It comes with some process and task dashboards out-of-the-box (showing for example the number of running process instances, the number of tasks completed per time frame, etc.). These dashboards however can be fully customized to show the data that is relevant to you, including for example your own data sources, making domain-specific charts (for example showing your key performance indicators (KPIs) instead of generic process-related charts). Check the chapter on Business Activity Monitoring in the Workbench Part for all the details.

25.2.8. Workbench

A workbench application, based on the UberFire framework, now unifies all web-based editors and tools into one large, configurable web application. It has many features, including:

- Configurable workspace where you layout your own views by dragging and dropping
- Unified login and role-based authentication, where what features you see depends on your role (admin, analyst, developer, user, manager, etc.).
- A new home screen that will guide you through the life cycle of your business processes (authoring, deployment, execution, tasks and reporting).
- Git-based repository that supports versioning and collaboration.
- New project structure where artifacts (processes, rules, etc.) are combined into kjars (we removed the custom binary packages and replaced them with a normal JAR, containing the source artifacts) when a project is built. These kjars now also include not only processes and rules, but also forms, configuration files, data models (Java classes), etc. Kjars are Maven artefacts themselves (they have a group, id and version) and exposed as a Maven repository. When creating a ksession, Maven can be used to download the necessary kjars for your project from this Maven repository.
- Sample `playground` repositories are (optionally) installed when starting up the workbench the first time, to get you started quickly with some predefined examples.

Check the Workbench Part for all the details.

25.2.9. Remote API

The remote API has been redesigned and allows users to remotely connect to a running execution server and pass commands. The remote runtime API exposes (almost) the entire KieSession and TaskService API using REST or JMS, so commands can be sent to the remote execution server for processing and the results are returned. See the chapter on Business Activity Monitoring for all the details.

Guvnor also provides a REST API to access the various repositories, projects and artifacts inside these projects and manage and build them.

25.3. New and Noteworthy in KIE Workbench 6.0.0

The workbench has had a big overhaul using a new base project called UberFire. UberFire is inspired by Eclipse and provides a clean, extensible and flexible framework for the workbench. The end result is not only a richer experience for our end users, but we can now develop more rapidly with a clean component based architecture. If you like he Workbench experience you can use UberFire today to build your own web based dashboard and console efforts.

As well as the move to a UberFire the other biggest change is the move from JCR to Git; there is an utility project to help with migration. Git is the most scalable and powerful source repository bar none. JGit provides a solid OSS implementation for Git. This addresses the continued performance problems with the various JCR implementations, which would slow down once the number of files and number of versions become too high. There has been a big "low tech" drive, to remove complexity. Everything is now stored as a file, including meta data. The database is only there to provide fast indexing and search. So importing and exporting is all standard Git and external sites, like GitHub, can be used to exchange repositories.

In 5.x developers would work with their own source repository and then push JCR, via the team provider. This team provider was not full featured and not available outside Eclipse. Git enables our repository to work any existing Git tool or team provider. While not yet supported in the UI, this will be added over time, it is possible to connect to the repo and tag and branch and restore things.

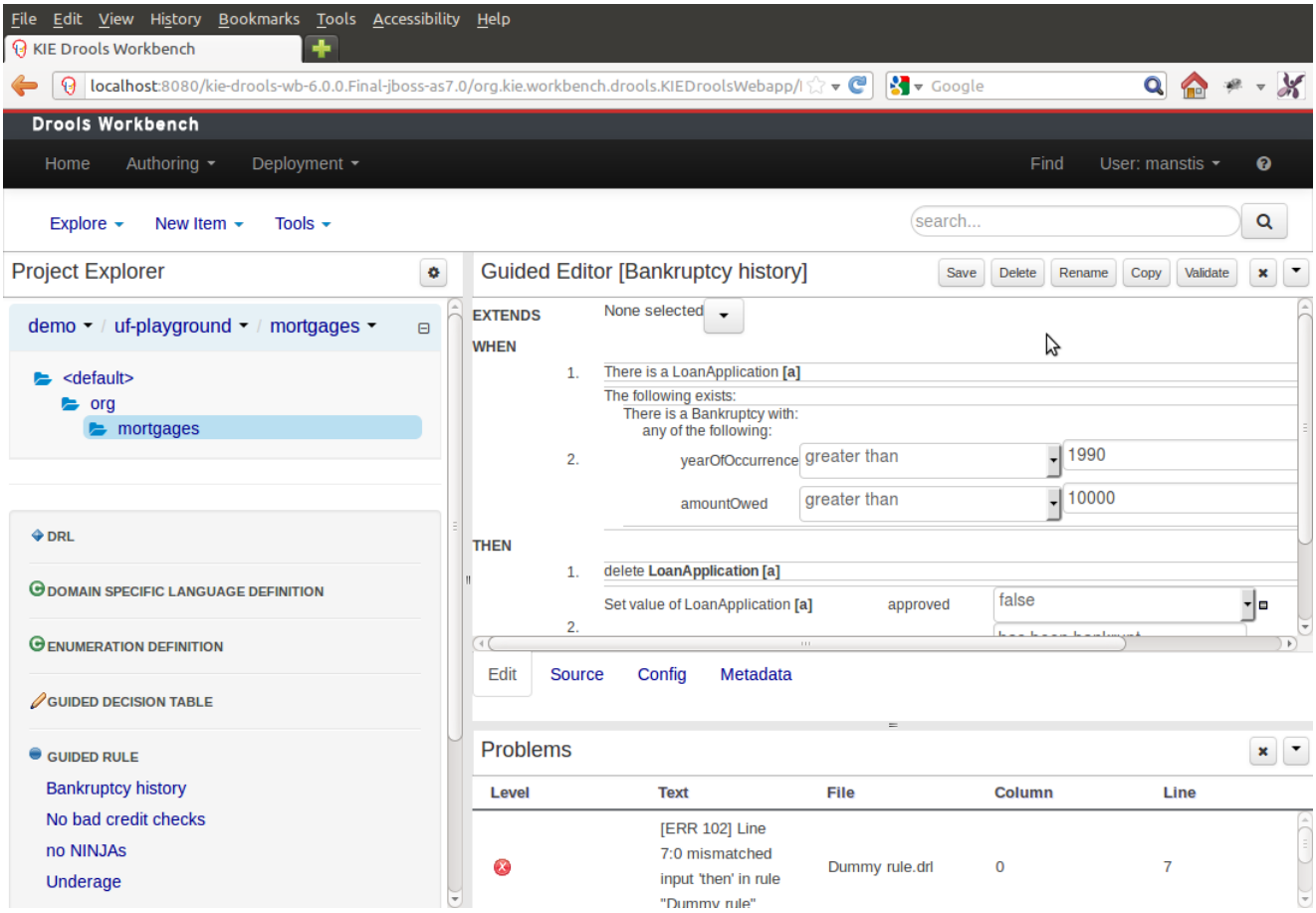


Figure 25.2. Workbench

The Guvnor brand leaked too much from its intended role; such as the authoring metaphors, like Decision Tables, being considered Guvnor components instead of Drools components. This wasn't helped by the monolithic projects structure used in 5.x for Guvnor. In 6.0 Guvnor 's focus has been narrowed to encapsulates the set of UberFire plugins that provide the basis for building a web based IDE. Such as Maven integration for building and deploying, management of Maven

repositories and activity notifications via inboxes. Drools and jBPM build workbench distributions using Uberfire as the base and including a set of plugins, such as Guvnor, along with their own plugins for things like decision tables, guided editors, BPMN2 designer, human tasks.

The "Model Structure" diagram outlines the new project anatomy. The Drools workbench is called KIE-Drools-WB. KIE-WB is the uber workbench that combines all the Guvnor, Drools and jBPM plugins. The jBPM-WB is ghosted out, as it doesn't actually exist, being made redundant by KIE-WB.

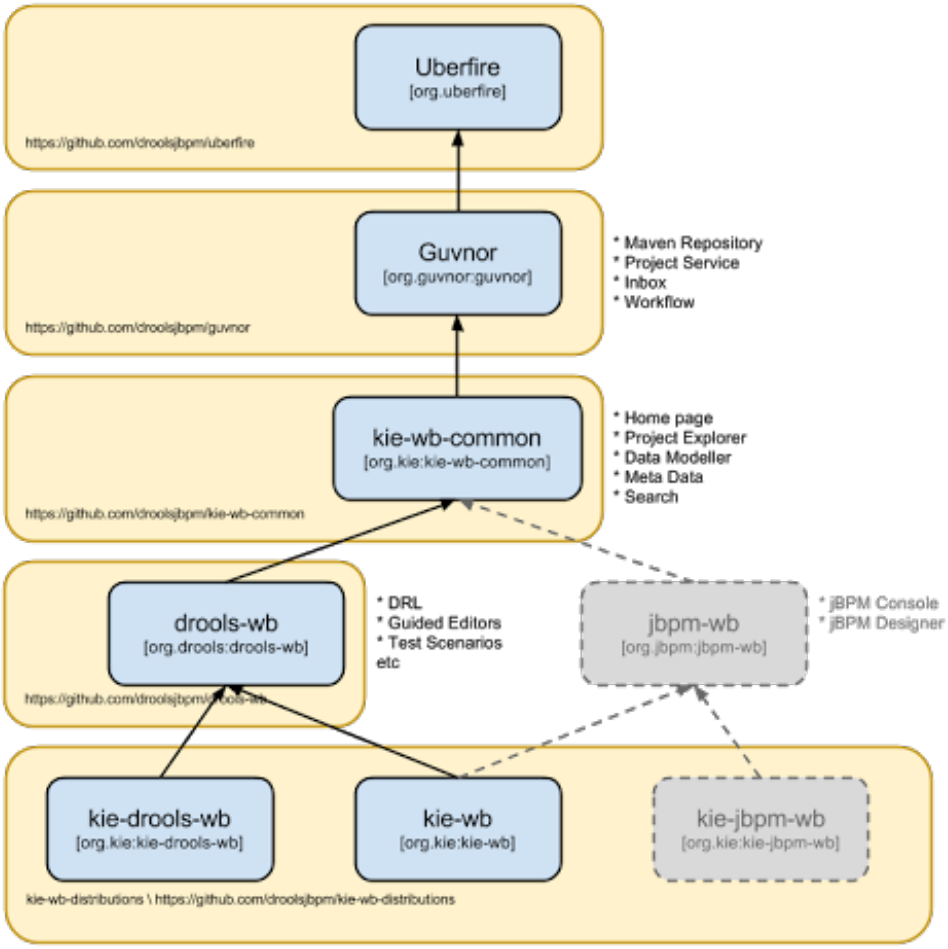


Figure 25.3. Module Structure

Important

KIE Drools Workbench and KIE Workbench share a common set of components for generic workbench functionality such as Project navigation, Project definitions, Maven based Projects, Maven Artifact Repository. These common features are described in more detail throughout this documentation.

The two primary distributions consist of:

- KIE Drools Workbench
  - Drools Editors, for rules and supporting assets.
  - jBPM Designer, for Rule Flow and supporting assets.
- KIE Workbench
  - Drools Editors, for rules and supporting assets.
  - jBPM Designer, for BPMN2 and supporting assets.
  - jBPM Console, runtime and Human Task support.
  - jBPM Form Builder.
  - BAM.

Workbench highlights:

- New flexible Workbench environment, with perspectives and panels.
- New packaging and build system following KIE API.
  - Maven based projects.
  - Maven Artifact Repository replaces Global Area, with full dependency support.
- New Data Modeller replaces the declarative Fact Model Editor; bringing authoring of Java classes to the authoring environment. Java classes are packaged into the project and can be used within rules, processes etc and externally in your own applications.
- Virtual File System replaces JCR with a default Git based implementation.
  - Default Git based implementation supports remote operations.
  - External modifications appear within the Workbench.
- Incremental Build system showing, near real-time validation results of your project and assets.

The editors themselves are largely unchanged; however of note imports have moved from the package definition to individual editors so you need only import types used for an asset and not the package as a whole.

25.4. New and Noteworthy in Integration 6.0.0

25.4.1. CDI

CDI is now tightly integrated into the KIE API. It can be used to inject versioned KieSession and KieBases.

```
@Inject
@KSession("kbase1")
@KReleaseId( groupId = "jar1", rtifactId = "art1", version = "1.0")
private KieBase kbase1v10;

@Inject
@KBase("kbase1")
@KReleaseId( groupId = "jar1", rtifactId = "art1", version = "1.1")
private KieBase kbase1v10;
```

Figure 25.4. Side by side version loading for 'jar1.KBase1' KieBase

```
@Inject
@KSession("ksession1")
@KReleaseId( groupId = "jar1", rtifactId = "art1", version = "1.0")
private KieSession ksessionv10;

@Inject
@KSession("ksession1")
@KReleaseId( groupId = "jar1", rtifactId = "art1", version = "1.1")
private KieSession ksessionv11;
```

Figure 25.5. Side by side version loading for 'jar1.KBase1' KieBase

25.4.2. Spring

Spring has been revamped and now integrated with KIE. Spring can replace the 'kmodule.xml' with a more powerful spring version. The aim is for consistency with kmodule.xml

25.4.3. Aries Blueprints

Aries blueprints is now also supported, and follows the work done for spring. The aim is for consistency with spring and kmodule.xml

25.4.4. OSGi Ready

All modules have been refactored to avoid package splitting, which was a problem in 5.x. Testing has been moved to PAX.