



Red Hat JBoss BPM Suite 6.1

User Guide

The User Guide for Red Hat JBoss BPM Suite

Red Hat JBoss BPM Suite 6.1 User Guide

The User Guide for Red Hat JBoss BPM Suite

Doug Hoffman

Eva Kopalova

B Long

Red Hat Engineering Content Services

belong@redhat.com

Gemma Sheldon

Red Hat Engineering Content Services

gsheldon@redhat.com

Joshua Wulf

jwulf@redhat.com

Legal Notice

Copyright © 2015 Red Hat, Inc.

This document is licensed by Red Hat under the [Creative Commons Attribution-ShareAlike 3.0 Unported License](#). If you distribute this document, or a modified version of it, you must provide attribution to Red Hat, Inc. and provide a link to the original. If the document is modified, all Red Hat trademarks must be removed.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat Software Collections is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

A guide to defining and managing business processes with Red Hat JBoss BPM Suite.

Table of Contents

CHAPTER 1. INTRODUCTION	5
1.1. USE CASE: PROCESS-BASED SOLUTIONS IN THE LOAN INDUSTRY	5
1.2. COMPONENTS	6
1.3. RED HAT JBOSS BPM SUITE AND BRMS	6
1.4. BUSINESS CENTRAL	6
CHAPTER 2. BASIC CONCEPTS	11
PART I. MODELING	13
CHAPTER 3. PROJECT	14
3.1. CREATING A PROJECT	14
3.2. ADDING DEPENDENCIES	15
3.3. DEFINING KIE BASE	16
3.4. DEFINING SESSIONS	18
3.5. CREATING A RESOURCE	19
3.6. ASSET METADATA AND VERSIONING	20
3.7. PROCESS DEFINITION	21
3.8. SYSTEM PROPERTIES	24
CHAPTER 4. PROCESS DESIGNER	27
4.1. CONFIGURING AUTOMATIC SAVING	28
4.2. DEFINING PROCESS PROPERTIES	28
4.3. DESIGNING A PROCESS	29
4.4. EXPORTING A PROCESS	33
4.5. PROCESS ELEMENTS	34
4.6. FORMS	35
4.7. FORM MODELER	37
4.8. VARIABLES	53
4.9. ACTION SCRIPTS	56
4.10. INTERCEPTOR ACTIONS	57
4.11. ASSIGNMENT	57
4.12. CONSTRAINTS	58
4.13. DATA MODELS	60
4.14. DOMAIN-SPECIFIC TASKS	61
4.15. EXCEPTION MANAGEMENT	77
CHAPTER 5. ADVANCED PROCESS MODELING	79
5.1. PROCESS MODELING OPTIONS	79
5.2. WORKFLOW PATTERNS	96
CHAPTER 6. SOCIAL EVENTS	98
FOLLOW USER	98
ACTIVITY TIMELINE	98
PART II. SIMULATION AND TESTING	99
CHAPTER 7. PROCESS SIMULATION	100
7.1. PATH FINDER	100
7.2. SIMULATING A PROCESS	101
CHAPTER 8. TESTING	106
8.1. UNIT TESTING	106
8.2. SESSION CREATION	107

PART III. PLUG-IN	111
CHAPTER 9. PLUG-IN	112
9.1. CREATING BPM PROJECT	112
9.2. CREATING PROCESS	112
9.3. USING THE DEBUG PERSPECTIVE	113
9.4. CHECKING SESSION LOGS	114
PART IV. DEPLOYMENT AND RUNTIME MANAGEMENT	116
CHAPTER 10. DEPLOYING PROJECTS	117
10.1. PROCESS INSTANCES	117
10.2. USER TASKS	118
CHAPTER 11. LOGGING	120
CHAPTER 12. EXAMPLES	121
PART V. BAM	122
CHAPTER 13. RED HAT JBOSS DASHBOARD BUILDER	123
WHAT IS BUSINESS ACTIVITY MONITORING?	123
13.1. ACCESSING DASHBOARD BUILDER	123
13.2. BASIC CONCEPTS	123
13.3. ENVIRONMENT	124
13.4. DATA SOURCES	124
CHAPTER 14. MANAGEMENT CONSOLE	133
CHAPTER 15. GRAPHIC RESOURCES	134
GRAPHIC RESOURCES DEFINITIONS	134
15.1. WORKING WITH GRAPHIC RESOURCES	134
APPENDIX A. PROCESS ELEMENTS	135
A.1. PROCESS	135
A.2. EVENTS MECHANISM	137
A.3. COLLABORATION MECHANISMS	138
A.4. TRANSACTION MECHANISMS	140
A.5. TIMING	141
A.6. PROCESS ELEMENTS	142
A.7. START EVENT	143
A.8. INTERMEDIATE EVENTS	145
A.9. END EVENTS	150
A.10. GATEWAYS	152
A.11. ACTIVITIES, TASKS AND SUB-PROCESSES	154
A.12. CONNECTING OBJECTS	162
A.13. SWIMLANES	163
A.14. ARTIFACTS	164
APPENDIX B. SERVICE TASKS	166
B.1. LOG TASK	166
B.2. EMAIL TASK	166
B.3. REST TASK	166
B.4. WS TASK	167
APPENDIX C. SIMULATION DATA	169
C.1. PROCESS	169

C.2. START EVENT	169
C.3. CATCHING INTERMEDIATE EVENTS	169
C.4. SEQUENCE FLOW	169
C.5. THROWING INTERMEDIATE EVENTS	169
C.6. HUMAN TASKS	170
C.7. SERVICE TASKS	170
C.8. END EVENTS	171
C.9. DISTRIBUTION TYPES	171
APPENDIX D. REVISION HISTORY	173

CHAPTER 1. INTRODUCTION

Red Hat JBoss BPM Suite is an open source business process management suite that combines Business Process Management and Business Rules Management and enables business and IT users to create, manage, validate, and deploy Business Processes and Rules.

To accommodate Business Rules component, JBoss BPM Suite includes integrated Red Hat JBoss BRMS.

Red Hat JBoss BRMS and Red Hat JBoss BPM Suite use a centralized repository where all resources are stored. This ensures consistency, transparency, and the ability to audit across the business. Business users can modify business logic and business processes without requiring assistance from IT personnel.

Business Resource Planner is also included with this release.

1.1. USE CASE: PROCESS-BASED SOLUTIONS IN THE LOAN INDUSTRY

This section describes a use case of deploying JBoss BPM Suite to automate business processes (such as loan approval process) at a retail bank. This use case is a typical process-based specific deployment that might be the first step in a wider adoption of JBoss BPM Suite throughout an enterprise. It leverages features of both business rules and processes of JBoss BPM Suite.

A retail bank offers several types of loan products each with varying terms and eligibility requirements. Customers requiring a loan must file a loan application with the bank. The bank then processes the application in several steps, such as verifying eligibility, determining terms, checking for fraudulent activity, and determining the most appropriate loan product. Once approved, the bank creates and funds a loan account for the applicant, who can then access funds. The bank must be sure to comply with all relevant banking regulations at each step of the process, and has to manage its loan portfolio to maximize profitability. Policies are in place to aid in decision making at each step, and those policies are actively managed to optimize outcomes for the bank.

Business analysts at the bank model the loan application processes using the BPMN2 authoring tools (Process Designer) in JBoss BPM Suite. Here is the process flow:

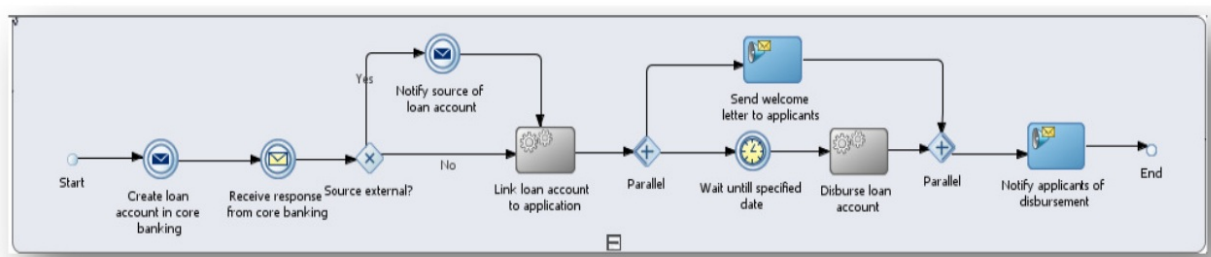


Figure 1.1. High-level loan application process flow

Business rules are developed with the rule authoring tools in JBoss BPM Suite to enforce policies and make decisions. Rules are linked with the process models to enforce the correct policies at each process step.

The bank's IT organization deploys the JBoss BPM Suite so that the entire loan application process can be automated.

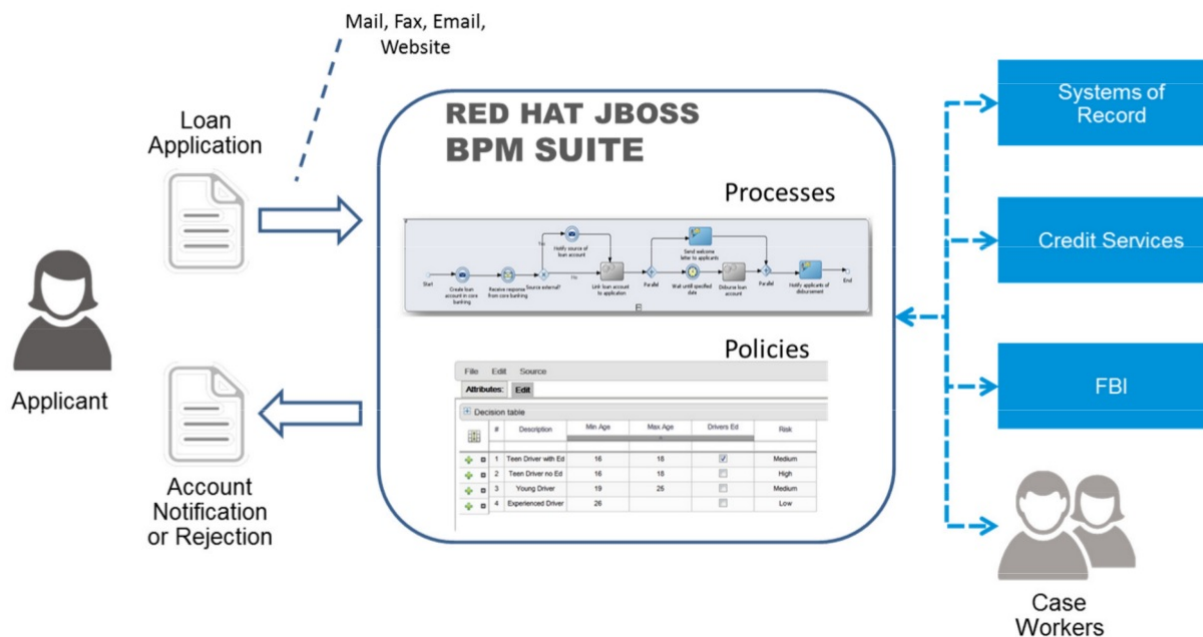


Figure 1.2. Loan Application Process Automation

The entire loan process and rules can be modified at any time by the bank's business analysts. The bank is able to maintain constant compliance with changing regulations, and is able to quickly introduce new loan products and improve loan policies in order to compete effectively and drive profitability.

1.2. COMPONENTS

Red Hat JBoss BPM Suite has the following components:

- **Business Central**, which is a web-based application (**business-central.war** and **dashbuilder.war**) and provides tools for creating, editing, building, managing, and monitoring of business assets as well as a Task client
- **Artifact repository** (Knowledge Store), which is the set of data the application operates over and is accessed by the Execution Server
- **Execution Server**, which provides the runtime environment for business assets

A more detailed description of components is available in the *Red Hat JBoss BPM Suite Administration and Configuration Guide*.

1.3. RED HAT JBOSS BPM SUITE AND BRMS

Red Hat JBoss BPM Suite comes with integrated Red Hat JBoss BRMS, a rule engine and rule tooling, so you can define rules governing Processes or Tasks. Based on a Business Rule Task call, the Process Engine calls the Rule Engine to evaluate the rule based on specific data from the Process instance. If the defined rule condition is met, the action defined by the rule is taken (refer to [Section A.11.3.7, "Business Rule Task"](#) and the Red Hat JBoss BRMS documentation for further information).

1.4. BUSINESS CENTRAL

Business Central is a web console that allows you to operate over individual components in a unified web-based environment: to create, manage, and edit your Processes, to run, manage, and monitor

Process instances, generate reports, and manage the Tasks produced, as well as create new Tasks and notifications.

- Process management capabilities allow you to start new process instances, acquire the list of running process instances, inspect the state of a specific process instances, etc.
- User Task management capabilities allow you to work with User Tasks; claim User Tasks, complete Tasks through Task forms, etc.

Business Central integrates multiple tools:

- **Process Designer and other editors** for modeling Processes and their resources (form item editor, work item editor, data model editor, etc.), as well as process model simulation tools (refer to [Chapter 4, *Process Designer*](#))
- **Rules Modeler** for designing Business Rules models and their resources (refer to Red Hat JBoss BRMS documentation)
- **Task client** for managing and creating User Tasks (refer to [Section 10.2, “User tasks”](#))
- **Process Manager** for managing process instances (refer to [Section 10.1, “Process instances”](#))
- **Dashboard Builder**, the BAM component, for monitoring and reporting (refer to [Chapter 13, *Red Hat JBoss Dashboard Builder*](#))
- **Business Asset Manager** for accessing the Knowledge Repository resources, building and deploying business assets (refer to [Chapter 3, *Project*](#))

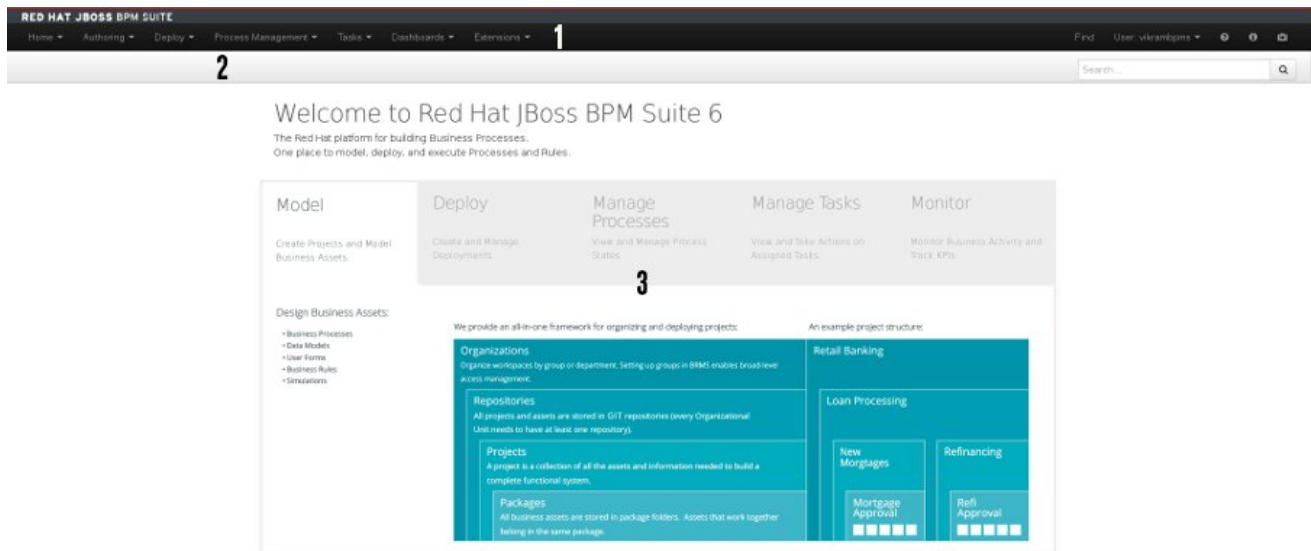
Artifact repository (Knowledge Store) is the set of data over which Business Central operates. It provides a centralized store for your business knowledge, which can consist of multiple repositories with business assets and resources (for further information, refer to the *Red Hat JBoss BPM Suite Administration and Configuration Guide*).

Business Central can be accessed from your web browser on <https://HOSTNAME/business-central> (for instances running on localhost <https://localhost:8080/business-central>).

The tools are accessible from the **Views** and **BPM** menus on the main menu:

- **Process Definitions** displays the **Process Definition List** with the Process definitions available in the connected repository.
- **Process Instances** displays the **Process Instance List** with the Process instances currently running on the Process Engine.
- **Tasks** displays a view of the Tasks list for the currently logged-in user. You can call a Task List in the grid view or in the calendar view from the **BPM** menu.

1.4.1. Business Central Environment



1	The main menu contains the links to the Home page and all available perspectives.
2	The perspective menu contains menus for the selected perspective (here it is empty; note that the content differs for individual perspectives) Section 1.4.2, “Perspectives” .
3	The perspective area contains the perspective tools (here the home page with links to individual perspectives and their views), such as views and editors.

Figure 1.3. Home page

1.4.2. Perspectives

Business Central provides the following groups of perspectives accessible from the main menu:

- **Authoring** group:
 - **Project Authoring** perspective contains the **Project Explorer** view (by default on the left) with the overview of available repository structure, and information on available resources, such as, business process definitions, form definitions, etc.; the editor area on the right, where the respective editor appears when a resource is opened; and the **Messages** view with validation messages.
 - **Artifact Repository** perspective contains a list of jars which can be added as dependencies. The available operations in this perspective are upload/download artifact and open (view) the **pom.xml** file.
 - **Administration** perspective (available only for users with the **ADMIN** role) contains the **File Explorer** view (by default on the left) with available asset repositories; the editor area on the right, where the respective editor appears when a resource is opened. The perspective allows an administrator to connect Knowledge Store to a repository with assets and to create a new repository (refer to *Administration and Configuration Guide*).
- **Deploy** group:
 - **Deployments** perspective contains a list of the deployed resources and allows you to build and deploy an undeploy new units.

- **Process Management** group:
 - **Process Definitions** perspective contains a list of the deployed Process definitions. It allows you to instantiate and manage the deployed Processes.
 - **Process Instances** perspective contains a list of the instantiated Processes. It allows you to view their execution workflow and its history.
- **Tasks** group:
 - **Task List** perspective contains a list of Tasks produced by Human Task of the Process instances or produced manually. Only Tasks assigned to the logged-in user are visible. It allows you to claim Tasks assigned to a group you are a member of.
- **Dashboards** group (the BAM component):
 - **Process & Task Dashboard** perspective contains a prepared dashboard with statistics on runtime data of the Execution Server
 - **Business Dashboards** perspective contains the full BAM component, the Dashbuilder, including administration features available for users with the **ADMIN** role.

1.4.3. Embedding Business Central

Business Central provides a set of editors to author assets in different formats. A specialized editor is used according to the asset format.

Business Central provides the ability to embed it in your own (Web) Applications using standalone mode. This allows you to edit rules, processes, decision tables, et cetera, in your own applications without switching to Business Central.

In order to embed Business Central in your application, you will need the Business Central 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 1.1. HTTP Query Parameters for Standalone Mode

Parameter Name	Explanation	Allow Multiple Values	Example
standalone	This parameter switches Business Central 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.GuvnorM2RepoPerspective

Parameter Name	Explanation	Allow Multiple Values	Example
header	Defines the name of the header that should be displayed (useful for context menu headers).	yes	ComplementNavArea

The following example demonstrates how to set up an embedded Author Perspective for Business Central.

```

===test.html===
<html>
  <head>
    <title>Test</title>
  </head>
  <body>
    <iframe id="ifrm" width="1920" height="1080"
src='http://localhost:8080/business-central?
standalone=&perspective=AuthoringPerspective&header=AppNavBar'></iframe>
  </body>
</html>

```

X-frame options can be set in **web.xml** of business-central. The default value for ***x-frame-options*** is as follows:

```

<param-name>x-frame-options</param-name>
  <param-value>SAMEORIGIN</param-value>

```

CHAPTER 2. BASIC CONCEPTS

Red Hat JBoss BPM Suite provides tools for creating, editing, running, and runtime management of BPMN process models. The models are defined using the BPMN2 language, either directly in its XML form or using visual BPMN Elements that represent the Process workflow (refer to [Chapter 4, *Process Designer*](#)). Alternatively, you can create Processes from your Java application using the JBoss BPM Suite API. Some of these capabilities can be used also via REST API (See *Red Hat JBoss BPM Suite Developer Guide*).

Process models serve as templates for Process instances. To separate the static Process models from their dynamic runtime versions (Process instances), they live in two different entities: Process models live in a Kie Base (or Knowledge Base) and their data cannot be changed by the Process Engine; Process instances live in a Kie Session(or Knowledge Session) which exists in the Process Engine and contains the runtime data, which are changed during runtime by the Process Engine.

You can define a Kie Base and its Kie Session in the Project Editor of the GUI application or using the provided API (refer to [Section 3.3, “Defining Kie Base”](#) and [Section 3.4, “Defining Sessions”](#)).

Note that a single Kie Base can be shared across multiple Kie Sessions. When instantiating a Kie Base using the respective API call it is usual to create one Kie Base at the start of your application as creating a Kie Base can be rather heavy-weight as it involves parsing and compiling the process definitions. From the Kie Base, you can then start multiple Kie Sessions. The underlying Kie Bases can be changed at runtime so you can add, remove, or migrate process definitions.

To have multiple independent processing units, it might be convenient to create multiple Kie Sessions on the particular Kie Base (for example, if you want all process instances from one customer to be independent from process instances for another customer; multiple Sessions might be useful for scalability reasons as well).

A Kie Session can be either stateful or stateless. Stateful sessions are long-living sessions with explicit call to dispose them; if the **dispose()** call is not issued, the session remains alive and causes memory leaks. Also note that the FireAllRules command is not automatically called at the end of a stateful session.

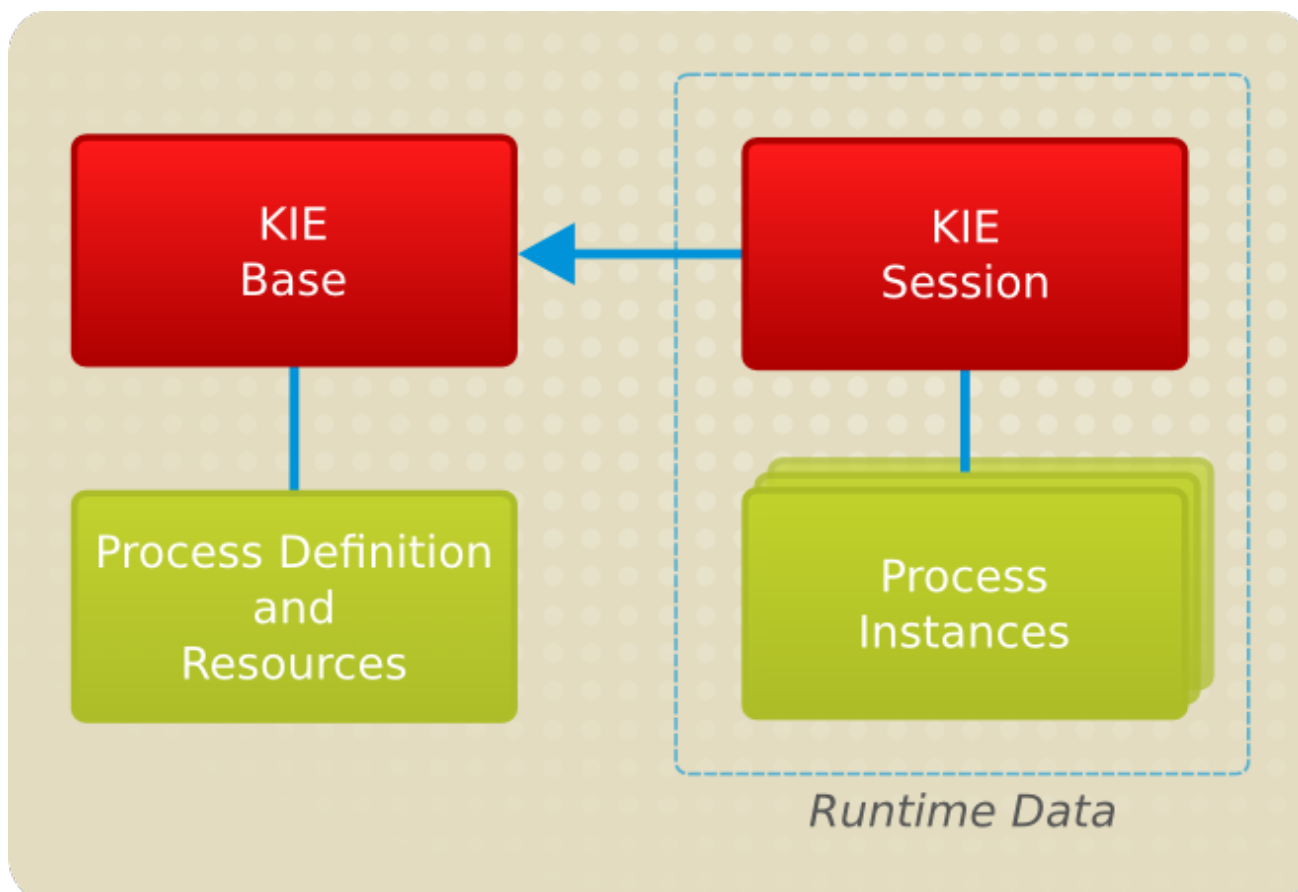


Figure 2.1. Kie Base and Kie Session relationship

PART I. MODELING

CHAPTER 3. PROJECT

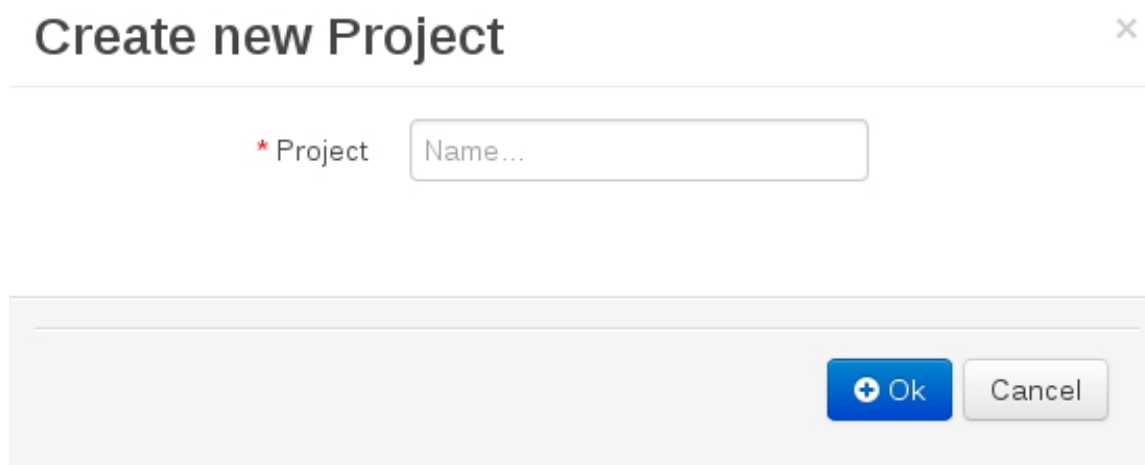
A project is a container for asset packages (business processes, rules, work definitions, decision tables, fact models, data models, and DSLs) that lives in the Knowledge Repository. It is this container that defines the properties of the KIE Base and KIE Session that are applied to its content. In the GUI, you can edit these entities in the Project Editor.

As a project is a Maven project, it contains the Project Object Model file (**pom.xml**) with information on how to build the output artifact. It also contains the Module Descriptor file, **kmodule.xml**, that contains the KIE Base and KIE Session configuration for the assets in the project.

3.1. CREATING A PROJECT

To create a project, do the following:

1. Open the **Project Authoring** perspective: on the main menu, click **Authoring** → **Project Authoring**.
2. In the **Project Explorer**, select the organizational unit and the repository where you want to create the project.
3. In the perspective menu, go to **New Item** → **Project**.
4. In the **Create new Project** dialog window, define the project details:
 - a. In the **Project** text box, enter the project name.



5. The explorer refreshes to show a **New Project Wizard** pop-up window.

New Project

✓ New Project Wizard

Project General Settings

Project Name

MortgageProject

Project Description

Insert a project description for documentation purposes ...

Group artifact version

Group ID

example

Example: com.myorganization.myprojects ?

Artifact ID

MortgageProject

Example: MyProject ?

Version

1.0

1.0.0 ?

< Previous

Next >

Cancel

✓ Finish

6. Define the **Project General Settings** and **Group artifact version** details for this new project. These parameters are stored inside the **pom.xml** Maven configuration file.

- **Project Name:** The name for the project; for example **MortgageProject**
- **Project Description:** The description of the project which may be useful for the project documentation purpose.
- **Group ID:** group ID of the project; for example **org.mycompany.commons**
- **Artifact ID:** artifact ID unique in the group; for example **myframework**. Avoid using a space or any special character that might lead to an invalid name.
- **Version ID:** version of the project; for example **2.1.1**

The **Project Screen** view is updated with the new project details as defined in the **pom.xml** file. Note, that you can switch between project descriptor files in the drop down-box with **Project Settings** and **Knowledge Base Setting**, and edit their contents.

3.2. ADDING DEPENDENCIES

To add dependencies to your project, do the following:

1. Open the Project Editor for the given project:
 - a. In the **Project Explorer** view of the **Project Authoring** perspective, open the project directory.

Open Project Editor

- b. Click on the button to open the project view.

15

2. In the **Project Screen** view, select in the **Project Settings** drop-down box the **Dependencies** item.
3. On the updated **Project Screen**, click the **Add** button to add a maven dependency or click the **Add from repository** button to add a dependency from the Knowledge Store (Artifact repository):
 - When adding a maven dependency, a user has to define the **Group ID**, **Artifact ID** and the **Version ID** in the new row which is created in the dependency table.
 - When adding a dependency from the Knowledge Store, select the dependency in the displayed dialog box: the dependency will be added to the dependency table.
4. To apply the various changes, the dependencies must be saved.



WARNING

If working with modified artifacts, do not re-upload modified non-snapshot artifacts as Maven will not know these artifacts have been updated, and it will not work if it is deployed in this manner.

3.3. DEFINING KIE BASE

You can create a Kie Base either using the API or in the **kmodule.xml** project descriptor file of your project via the Project Editor.

Defining Kie Base in the Project Editor

To define a Kie Base in the web environment, which is stored in the **kmodule.xml** file, do the following:

1. To open **Project Explorer**, click **Authoring > Project Authoring** and select or navigate to your project.
2. Open your project properties in Project Editor: in the perspective menu, click **Tools → Project Editor**.
3. In the drop-down menu on the **Project Screen** view, click **Knowledge bases and sessions**.

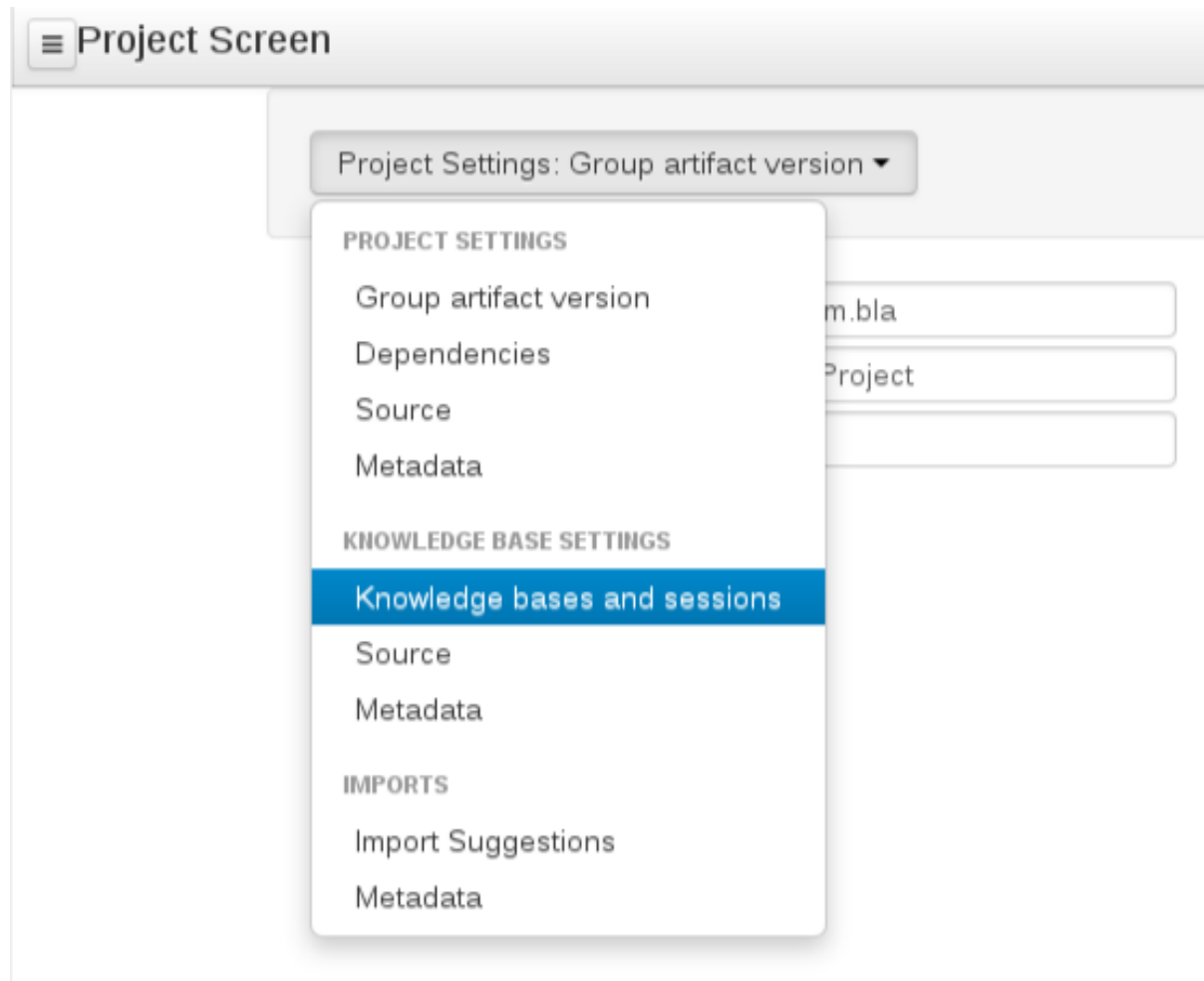


Figure 3.1. Project properties selection

4. In the Knowledge Bases area on the updated **Project Screen**, define and select the Knowledge (Kie) Base definition.

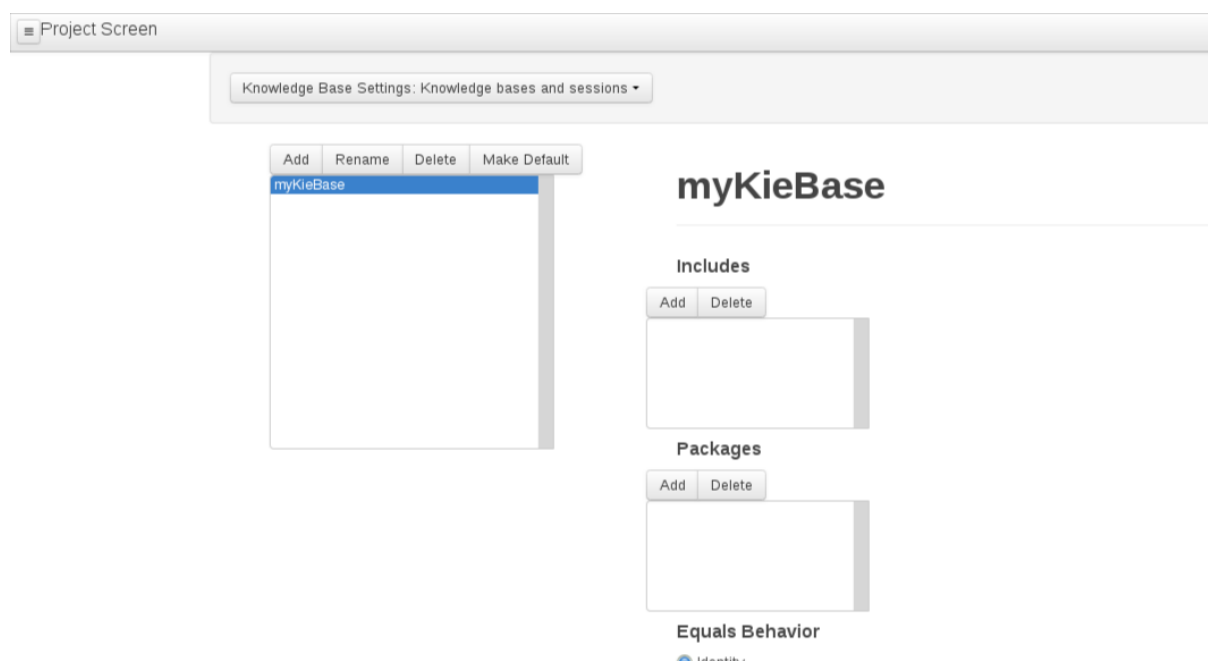


Figure 3.2. New Knowledge Base created

Defining Kie Base using API

To define Kie Base using API, use the following code:

```
import org.kie.api.KieServices;
import org.kie.api.builder.KieRepository;
import org.kie.api.builder.KieFileSystem;
import org.kie.api.builder.KieBuilder;
import org.kie.api.runtime.KieContainer;
import org.kie.api.KieBase;
...
KieServices kServices = KieServices.Factory.get();
KieRepository kRepository = kServices.getRepository();
KieFileSystem kFileSystem = kServices.newKieFileSystem();

kFileSystem.write(ResourceFactory.newClassPathResource("MyProcess.bpmn"));

KieBuilder kBuilder = kServices.newKieBuilder(kFileSystem);
kBuilder.buildAll();

KieContainer kContainer =
kServices.newKieContainer(kRepository.getDefaultReleaseId());
KieBase kBase = kContainer.getKieBase();
```

3.4. DEFINING SESSIONS

You can create a Kie Session either using the API or in the **kmodule.xml** project descriptor file of your project via the Project Editor.

Defining Kie Session using API

Once you have loaded your Kie Base, you need to create a Kie Session to interact with the Process Engine to run and manage its Process instances.

Example 3.1. Creating a Kie Session and starting a MyProcess instance

```
import org.kie.api.runtime.KieSession;
import org.kie.api.runtime.process.ProcessInstance;
...
KieSession ksession = kBase.newKieSession();

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

You can interact with Process instances through the **ProcessRuntime** interface that the Kie Sessions implement.

ProcessRuntime methods

ProcessInstance startProcess(String processId);

The method starts a new Process instance of the process with the specified ID and returns the ProcessInstance reference

ProcessInstance startProcess(String processId, Map<String, Object> parameters);

The method starts a new process instance of the process with the specified ID and returns the `ProcessInstance` reference. Additional parameters provided as a `Map` (as name-value pairs) are set as variables of the process instance.

`void signalEvent(String type, Object event);`

The method signals the *Process Engine* that an event of the defined type has occurred. The event parameter can contain additional information related to the event. All process instances that are listening to this type of external event are notified. For performance reasons, it is recommended to use this type of event signaling only if exactly one process instance is able to notify other process instances. For internal event within one process instance, use the `signalEvent` method that also includes the `processInstanceId` of the respective process instance.

`void signalEvent(String type, Object event, long processInstanceId);`

The method signals to a `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 are notified. Note that the event is only processed inside the given `Process` instance. No other `Process` instances waiting for this type of event are notified.

`Collection<ProcessInstance> getProcessInstances();`

The method returns a collection of the currently active `Process` instances. Only `Process` instances that are currently loaded and active in the `Process Engine` are returned. When using persistence, the persisted `Process` instances are not returned. It is recommended to use the history log to collect the information about the state of your `Process` instances instead.

`ProcessInstance getProcessInstance(long processInstanceId);`

The method returns the `Process` instance with the given id. Only active `Process` instances are returned: if a `Process` instance has been completed, the method returns `null`.

`void abortProcessInstance(long processInstanceId);`

The method aborts the `Process` instance with the given ID. If the `Process` instance has been completed or aborted, or it cannot be found, the method throws an `IllegalArgumentException`.

`WorkItemManager getWorkItemManager();`

The method returns the `WorkItemManager` related to the `Kie Session`. The returned object reference can be used to register new `WorkItemHandlers` or to complete or abort `WorkItems`.

Defining Kie Session in the Project Editor

To define a `Kie Session` in the web environment in the `kmodule.xml` file, do the following:

1. Open your project properties with the `Project Editor`: in the `Project Explorer`, locate your project root. In the perspective menu, go to **Tools** → **Project Editor**.
2. In the drop-down box of the **Project Screen** view, click **Knowledge bases and sessions**.
3. On the left side, select **kbase**.
4. You can now add a new `ksession` and set default, state, and clock.

3.5. CREATING A RESOURCE

A Project may contain an arbitrary number of packages, which contain files with resources, such as Process definition, Work Item definition, Form definition, Business Rule definition, etc.

To create a resource, select the Project and the package in the **Project Explorer** and click **New Item** on the perspective menu and select the resource you want to create.



NOTE

It is recommended to create your resources, such as Process definitions, Work Item definitions, Data Models, etc., inside a package of a Project to allow importing of resources and referencing their content.

To create a package, do the following:

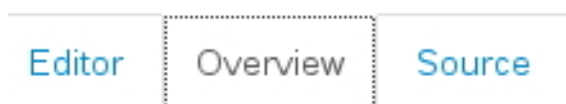
- In the **Repository** view of the Project Explorer, navigate to the **REPOSITORY/PROJECT/src/main/resources/** directory.
- Go to **New Item** → **Package**.
- In the **New resource** dialog, define the package name and check the location of the package in the repository.

3.6. ASSET METADATA AND VERSIONING

Most assets within Business Central have some metadata and versioning information associated with them. In this section, we will go through the metadata screens and version management for one such asset (a DRL asset). Similar steps can be used to view and edit metadata and versions for other assets.

Metadata Management

To open up the metadata screen for a DRL asset, click on the **Overview** tab. If an asset doesn't have an **Overview** tab, it means that there is no metadata associated with that asset.



The **Overview** section opens up in the **Version history** tab, and you can switch to the actual metadata by clicking on the **Metadata** tab.

The metadata section allows you to view or edit the **Categories**, **Subject**, **Type**, **External Link** and **Source metadata** for that asset. However, the most interesting metadata is the description of the asset that you can view/edit in the description field and the comments that you and other people with access to this asset can enter and view.

Comments can be entered in the text box provided in the comments section. Once you have finished entering a comment, press enter for it to appear in the comments section.



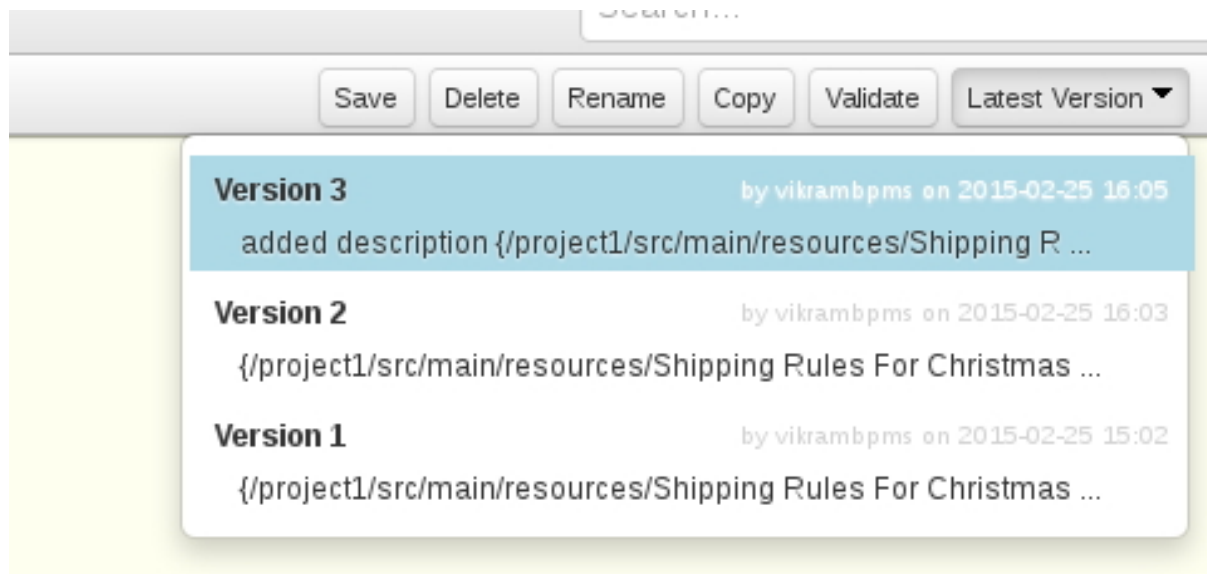
IMPORTANT

You must hit the **Save** button for all metadata changes to be persisted, including the comments.

Version Management

Every time you make a change in an asset and save it, a new version of the asset is created. You can switch between different versions of an asset in one of two ways:

- Click the **Latest Version** button in the asset toolbar and select the version that you are interested in. Business Central will load this version of the asset.



- Alternatively, open up the **Overview** section. The **Version history** section shows you all the available versions. **Select** the version that you want to restore.

In both cases, the **Save** button will change to **Restore**. Click this button to persist changes.

3.7. PROCESS DEFINITION

A Process definition is a BPMN 2.0-compliant file that serves as container for a Process and its BPMN Diagram. A Process definition itself defines the import entry, imported Processes, which can be used by the Process in the Process definition, and relationship entries. We refer to a Process definition as a business process. .

Example 3.2. BPMN2 source of a Process definition

```
<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>
```

```

    PROCESS
  </process>

  <bpmndi:BPMNDiagram>
    BPMN DIAGRAM DEFINITION
  </bpmndi:BPMNDiagram>

</definitions>

```

3.7.1. Creating a Process definition

Make sure you have logged in to JBoss BPM Suite or you are in JBoss Developer Studio with the repository connected.

To create a Process, do the following:

1. Open the Project Authoring perspective (**Authoring** → **Project Authoring**).
2. In **Project Explorer** (**Project Authoring** → **Project Explorer**), navigate to the project where you want to create the Process definition (in the **Project** view, select the respective repository and project in the drop-down lists; in the **Repository** view, navigate to **REPOSITORY/PROJECT/src/main/resources/** directory).



NOTE

It is recommended to create your resources, including your Process definitions, in a package of a Project to allow importing of resources and their referencing. To create a package, do the following:

- In the **Repository** view of the Project Explorer, navigate to the **REPOSITORY/PROJECT/src/main/resources/** directory.
- Go to **New Item** → **Package**.
- In the **New resource** dialog, define the package name and check the location of the package in the repository.

3. From the perspective menu, go to **New Item** → **Business Process**.
4. In the **New Processes** dialog box, enter the Process name and click **OK**. Wait until the Process Editor with the Process diagram appears.

3.7.2. Importing a Process definition

To import an existing BPMN2 or JSON definition, do the following:

1. In the **Project Explorer**, select a Project and the respective package to which you want to import the Process definition.
2. Create a new Business Process to work in by going to **New Item** → **Business Process**.


3. In the Process Designer toolbar, click the **Import**  icon in the editor toolbar and pick the format of the imported process definition. Note that you have to choose to overwrite the existing process definition in order to import.
4. From the **Import** window, locate the Process file and click **Import**.



Figure 3.3. Import Window

Whenever a process definition is imported, the existing imported definition is overwritten. Make sure you are not overwriting a process definition you have edited so as not to lose any changes.

A process can also be imported to the git repository in the filesystem by cloning the repository, adding the process files, and pushing the changes back to git. In addition to alternative import methods, you can copy and paste a process or just open a file in the import dialog.

When importing processes, the Process Designer provides visual support for Process elements and therefore requires information on element positions on the canvas. If the information is not provided in the imported Process, you need to add it manually.

3.7.3. Importing jPDL 3.2 to BPMN2

To migrate and import a jPDL definition to BPMN2, in the Process Designer, click on the import button then scroll down and select **Migrate jPDL 3.2 to BPMN2**.

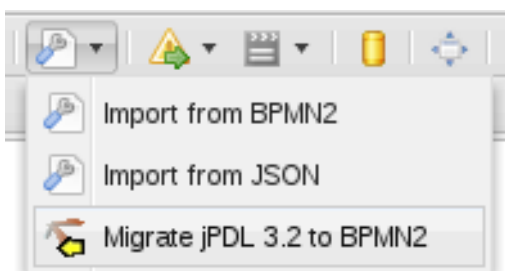


Figure 3.4. Migrate jPDL 3.2 to BPMN2

In the **Migrate to BPMN2** dialog box, select the process definition file and the name of the **gpd** file. Confirm by clicking the **Migrate** button.

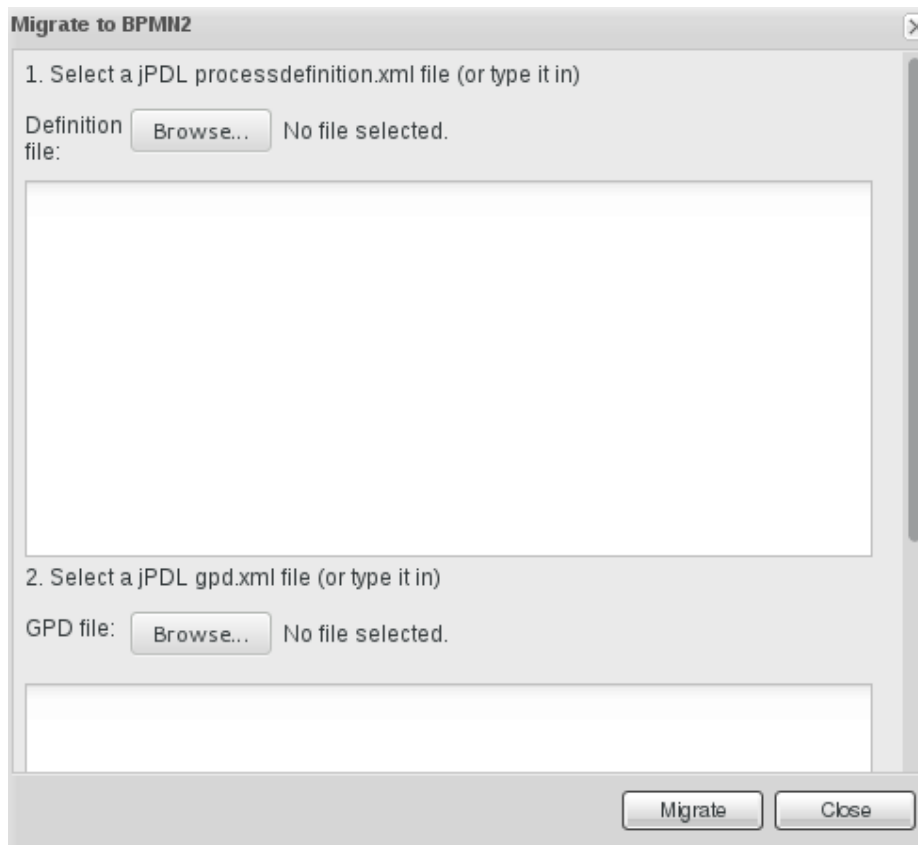




Figure 3.5. Migrate to BPMN2 dialog box



IMPORTANT

The migration tool for jPDL 3.2 to BPMN2 is a technical preview feature, and therefore not currently supported in Red Hat JBoss BPM Suite.

3.7.4. Deleting a Process definition

To delete a Process definition, open it in the Process Designer, click the **Save** menu () and then **Delete** ().

3.8. SYSTEM PROPERTIES

The following is a list of all system properties:

Table 3.1. System Properties

Property	Description
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

Property	Description
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 certificates 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.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.uberfire.domain	Security-domain name used by uberfire. Default: ApplicationRealm

Property	Description
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-6.2.0-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:

- Edit the file `$ JBOSS_HOME/domain/configuration/host.xml`.
- 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>
```

CHAPTER 4. PROCESS DESIGNER

The *Process Designer* is the Red Hat JBoss BPM Suite process modeler. The output of the modeler is a BPMN 2.0 process definition file, which is saved in the Knowledge Repository, under normal circumstances with a package of a project. The definition then serves as input for JBoss BPM Suite Process Engine, which creates a Process instance based on the definition.

The editor is delivered in two variants:

JBoss Developer Studio Process Designer

Thick-client version of the Process Designer integrated in the JBoss Developer Studio plug-in

Web Process Designer

Thin-client version of the Process Designer integrated in BPM Central

The graphical user interface of the Process Designer is the same for both the JBoss Developer Studio Process Designer and the Web Process Designer.

❶	The canvas represents the process diagram. Here you can place the elements from the palette which will constitute the Process. Note that one Process definition may contain exactly one process diagram; therefore a Process definition equals to a Process diagram (this may differ in other products).
❷	The Object Library (palette) contains groups of BPMN2 elements. Details on execution semantics and properties of individual BPMN2 shapes are available in Appendix A, <i>Process Elements</i> .
❸	The Properties panel displays the properties of the selected element. If no element is selected, the panel contains Process properties.
❹	The editor toolbar allows you to select an operation to be applied to the Elements on the canvas.

Figure 4.1. Process Designer environment

NOTE

To enlarge the Process Designer screen (or any screen while working in Business Central), click on the



button shown here: . This will make your current editor fill the entire Business Central screen. To go back, simply click the button again.

4.1. CONFIGURING AUTOMATIC SAVING

To configure automatic saving, click the **Save** button in Process Designer and then **Enable autosave**.

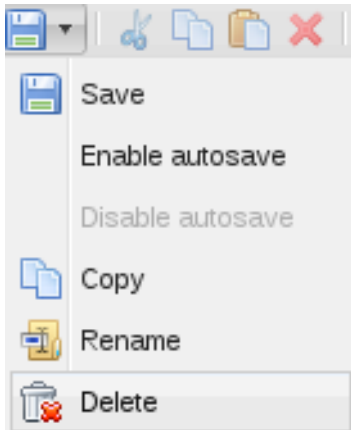


Figure 4.2. Enable autosave

4.2. DEFINING PROCESS PROPERTIES

To define Process properties, do the following:

1. Open the Process file in the Process Designer.
2. Click anywhere on the canvas: make sure, no Process element is selected.



IMPORTANT

Do not use Unicode characters when defining the Process name or the Process ID: usage of such characters is not supported and results in unexpected behavior of the Process designer when saving and retrieving Process assets.

3. Expand the Properties panel on the left if applicable and define the Process properties on the tab by clicking individual entries. For entries that require other input than just string input, the respective editors can be used by clicking the arrow icon. Note that the editors for complex fields mostly provide validation and auto-completion features.

Properties (BPMN-Diagram)	
Name ▲	Value
Core Properties	
AdHoc	false
Executable	true
Globals	
ID	myProject.myProc
Imports	
Package	org.jbpm
Process Name	myProc
Variable Defi...	<input type="text"/>
Version	1.0

Figure 4.3. Opening the variable editor

4. To save your changes, click **File** and **Save changes**

4.3. DESIGNING A PROCESS

To model a Process, do the following:

1. In the Project view of the Project Explorer, select your Project and click the respective Process under **Business Processes**. Alternatively, locate the process definition in the Repository view of the Project Explorer and double-click the file to open it in the Process Designer.
2. Add the required shapes to the process diagram on the canvas:
 - o Drag and drop the shapes from the **Object Library** palette to the required position on the canvas.

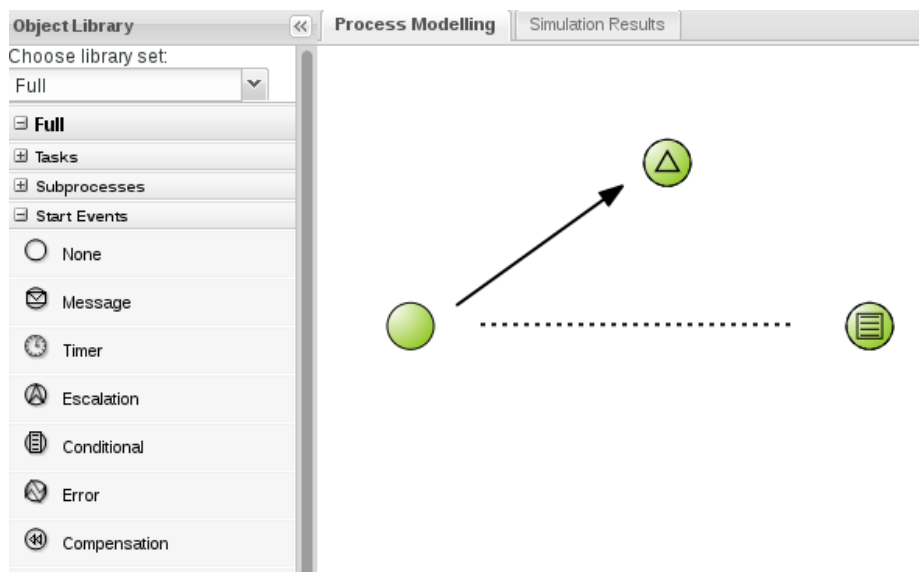



Figure 4.4. Object Library

- Select a shape already placed on the canvas: the quick linker menu appears. The quick linker feature displays only the elements that can be connected to the selected shape and connects them with a valid Association element.



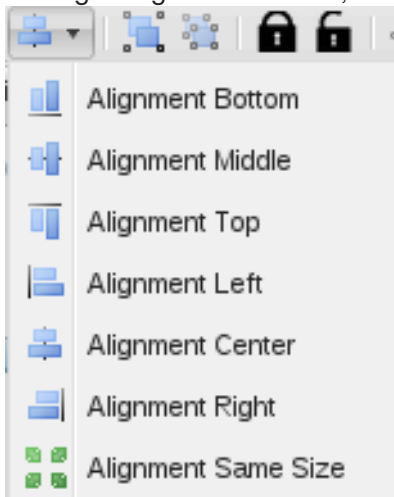
NOTE

To change the type of an already placed element to an element that extends this element, click it and select the **Morph shape** () icon.

3. Double-click an element to provide its Name. Consider defining the element properties in the Properties view.
4. Repeat the previous step until the Process Diagram defines the required workflow.

4.3.1. Aligning Elements

To align diagram Elements, select the elements and click the respective button in the alignment toolbar



- **Bottom**: the selected elements will be aligned with the element located at the lowest position
- **Middle**: the selected elements will be aligned to the middle relative to the highest and lowest element
- **Top**: the selected elements will be aligned with the element located at the highest position
- **Left**: the selected elements will be aligned with the leftmost element
- **Center**: the selected elements will be aligned to the center relative to the leftmost and rightmost element
- **Right**: the selected elements will be aligned with the rightmost element

Note that dockers of Connection elements are not influenced by aligning and you might need to remove them.

4.3.2. Changing Element layering

To change the Element layering, select the element or a group of element and click the respective button in the toolbar:

- **Bring To Front** : bring the selected element to foreground to the uppermost layer
- **Bring To Back** : send the selected element to background to the lowest layer
- **Bring Forward** : bring the selected element to foreground by one layer
- **Bring Backward** : send the selected element to background by one layer
- **Center** : the selected elements will be aligned to the center relative to the leftmost and rightmost element
- **Right** : the selected elements will be aligned with the rightmost element

Note that Connection Elements are not influenced by layering and remain always visible.

4.3.3. Bending Connection Elements

When moving an Element with incoming or outgoing Connection elements, dockers are automatically added to accommodate the appropriate Connection shape. To create a docker manually, click and pull



the respective point of the Connection object. To delete a docker, click the **Delete a Docker** button in the toolbar and then click the respective Docker. Once you delete dockers of a Connection object, no more dockers will be created automatically.

4.3.4. Resizing Elements

To resize Elements on the canvas, select the element, and click and pull the blue arrow displayed in the upper left or lower right corner of the element.



To make the size of multiple elements identical, select the Elements and then click the icon in the toolbar and then click on **Alignment Same Size**: all Elements will be resized to the size of the largest selected Element.

Note that only Activity Elements can be resized.

4.3.5. Grouping Elements

Group Elements behave on the canvas as one item.

To create an element group, select the respective items on the canvas and click the **Groups all**



selected shapes button in the toolbar. To ungroup such elements, select the group and click



the **Delete the group of all selected elements** button.

4.3.6. Locking Elements

Locking Elements of a Process model prevents their editing: locked Elements are visualized as locked and cannot be moved or edited unless unlocked.



To lock Elements, select the elements and click the **Lock Elements** button in the toolbar. To



unlock such Element, select them and click the **Unlock Elements** button in the toolbar.

4.3.7. Changing the color scheme

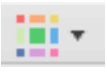
Color schemes define the color used for individual Process Elements in its diagram.

Color schemes are stored in the **themes.json** file, which is located in the **global** directory of each repository.

Procedure 4.1. Creating a new color schema


1. Locate your project in the Project Explorer and switch to the **Repository** view.
2. Open the **global** directory and locate and open the **themes.json** file.
3. In the displayed Default Editor, add your theme definition at the end of the file and click the **Save** button.

To apply a new color scheme or any other defined scheme, in the Process Designer, click the **Color**

Scheme  button in the toolbar and select the respective color scheme from the drop-down menu.

4.3.8. Recording local history

Local history keeps track of any changes, you apply to your Process model so as to allow you to restore any previous status of the Process model. By default, this feature is turned off.



To turn on local history recording, click the **Local History**  button and select **Enable Local History** entry. From this menu, you can also display the local history records and apply the respective status to the Process as well as disable the feature or clear the current local history log.

4.3.9. Enlarging and shrinking canvas


To change the size of the canvas, click the respective yellow arrow on the canvas edge.

4.3.10. Validating a Process

Process validation can be set up to be continuous or to be only immediate.

To validate your Process model continuously, click the **Validate** () button in the toolbar of the Process Designer with the Process and click **Start Validating**. If validation errors have been detected, the elements with errors are highlighted in orange. Click on the invalid element on the canvas to display a dialog with the summary of its validation errors. To disable continuous validation, click the **Validate** () button in the toolbar of the Process Designer with the Process and click **Stop Validating**.

Also note that errors on the element properties are visualized in further details in the Properties view of the respective element.

If you want to display the validation errors and not to keep the validation feature activated, click the **Validate** () button in the toolbar of the Process Designer with the Process and click **View all issues**.

Additionally after you save your Process, any validation errors are also displayed in the **Messages** view.

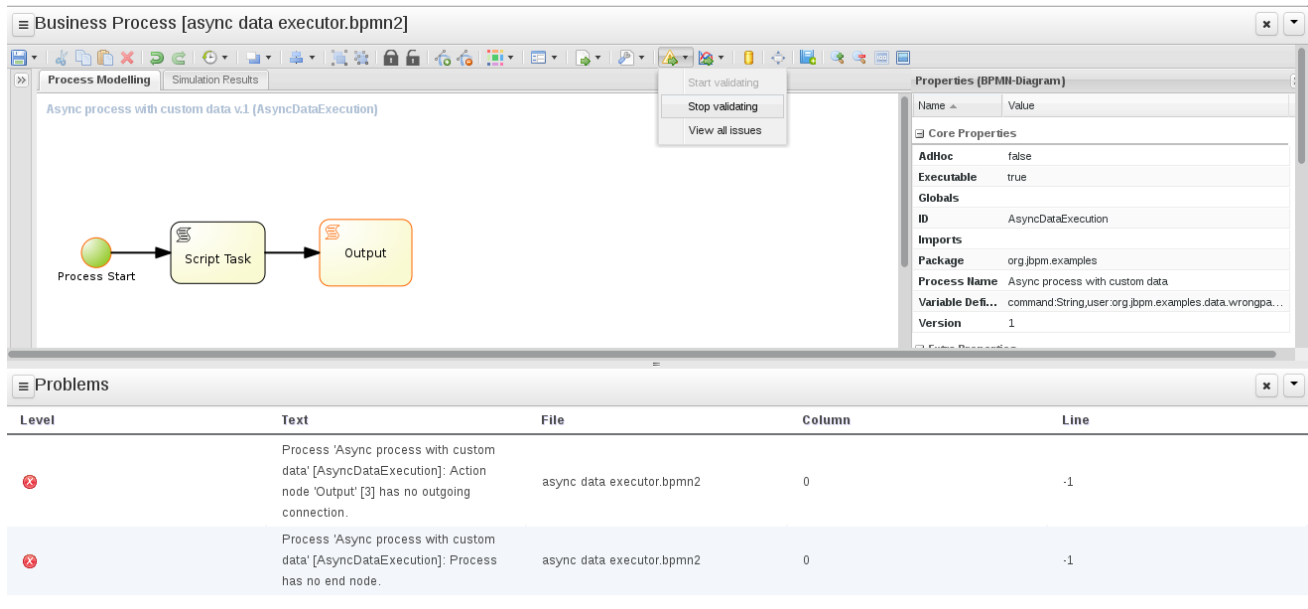


Figure 4.5. Stopping continuous validation

4.4. EXPORTING A PROCESS

To export your Process definition to one of the supported formats (PNG, PDF, BPMN2, JSON, SVG, or ERDF), open the Process and click the Export drop-down icon. The Process will be exported to a file with the name of the Process definition.

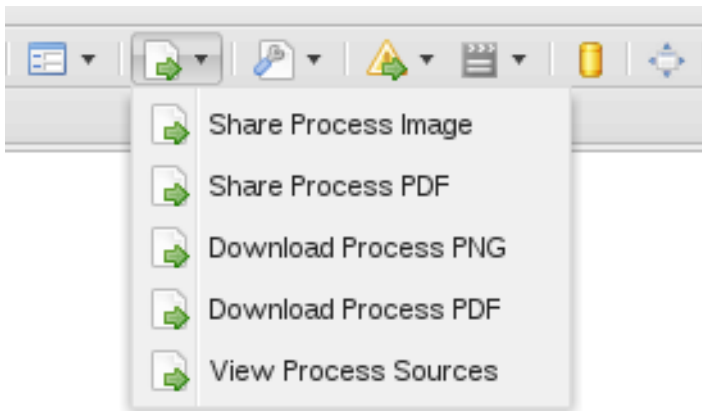


Figure 4.6. Export Icon

- **Share Process Image** - Generates PNG file into the repository.
- **Share Process PDF** - Generates PDF file into the repository.
- **Download Process PNG** - Generates PNG file into the repository and the browser starts downloading this file.
- **Download Process PDF** - Generates PDF file into the repository and the browser starts downloading this file.
- **View Process Sources** - Opens the "Process Sources" dialog box which contains the BPMN2, JSON, SVG, and ERDF source codes. The "Download BPMN2" button allows the user to download BPMN2 files. Pressing CTRL+A allows you to select the source code in a particular format. Pressing CTRL+F enables the find tool (Use /re/ syntax for regexp search).

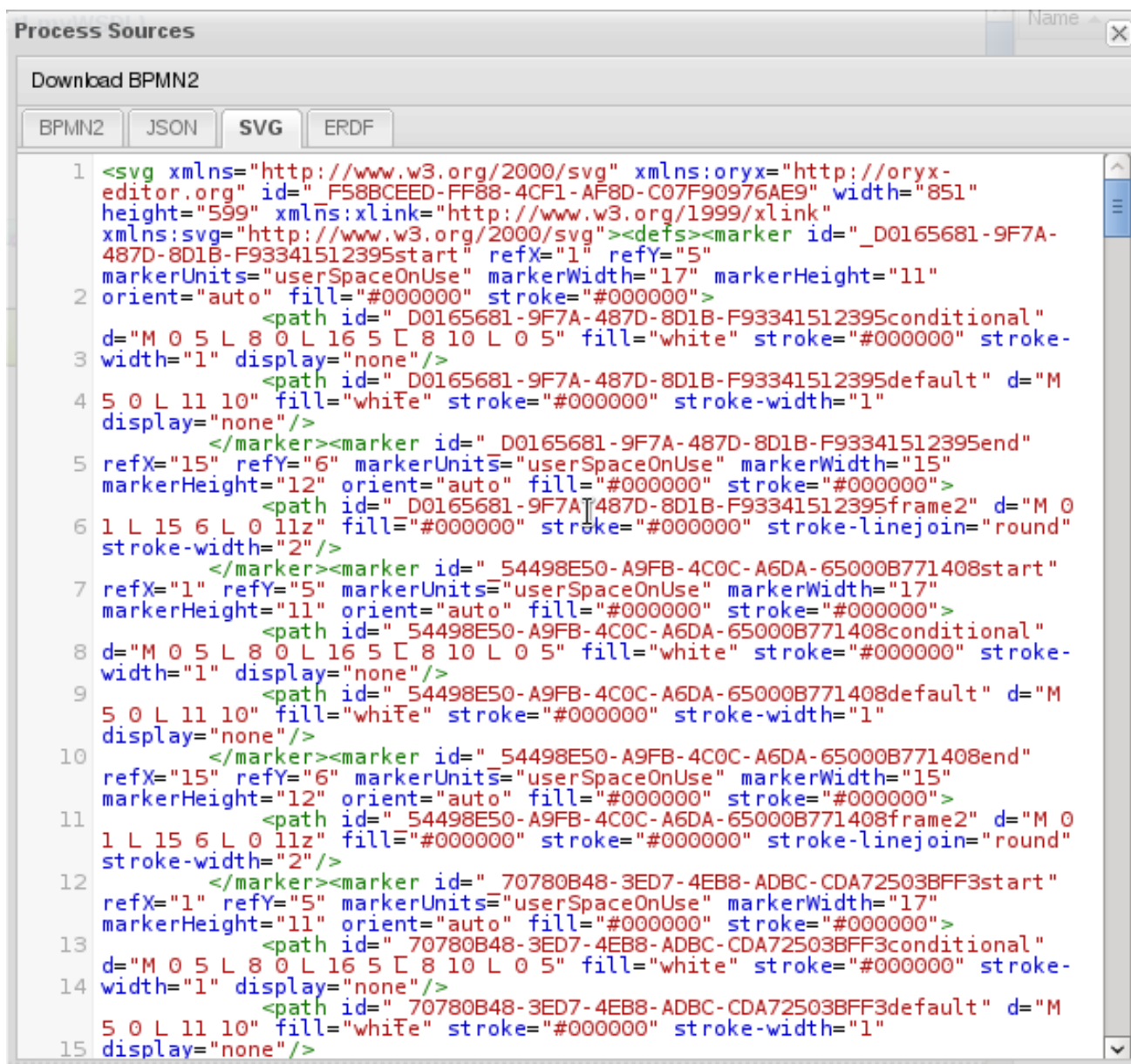


Figure 4.7. Process Sources

4.5. PROCESS ELEMENTS

4.5.1. Generic properties of visualized Process elements

All Process elements have the following visualization properties, which can be defined in their **Properties** tab:

Background

The background color of the element in the diagram

Border color

The border color of the element in the diagram

Font color

The color of the font in the element name

Font size

The size of the font in the element name

Name

The element name displayed on the BPMN diagram


4.5.2. Defining Process elements properties

All Process Elements, including the Process, define a set of properties that define the following:

- **Core** properties, which include the basic properties of an element (typically Name, Data Set, Scripts, etc.).
- **Extra** properties, which include the properties necessary for the element execution (refer to [Section A.6, “Process Elements”](#)), data mapping (variable mapping) and local variable definitions (see [Section 4.8.1, “Globals”](#)), properties that represent an extension of the jBPM engine, typically onExitAction, Documentation, etc.
- **Graphical** properties, which include graphical representation of elements (colors, text settings).
- **Simulation** properties are used by the Simulation engine.

In element properties of the String type can use **#{expression}** to embed a value. The value will be retrieved on element instantiation, and the substitution expression will be replaced with the result of calling the toString() method on the variable defined in the expression. The expression could 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}**.

To define Element properties, do the following:

1. Open the Process definition in the Process Designer:
2. On the canvas, select the Element.
3. Click the double arrow () in the upper left corner of the Process Designer to display the **Properties** view.
4. In the displayed Properties view, click the property value fields to edit them. Note that where applicable, you can click the drop-down arrow and the relevant value editor appears in a new dialog box.
5. To save your changes, click the **Save** icon and select option **Save**.

4.6. FORMS

A **form** is a layout definition for a page (defined as HTML) that is displayed as a dialog window to the user on a

- **process instantiation** or a
- **task instantiation**.

The form is then respectively referred to as a **Process form** or a **Task form**. It serves for acquiring data for the Element instance execution, be it a Process or Task, from a human user: a Process form can take as its input and output Process variables; a Task form can take as its input DataInputSet variables with assignment defined, and as its output DataOutputSet variables with assignment defined.


For example, you could ask the user to provide the input parameters needed for Process instantiation and display any variable data connected to the Process; or using a Human Task show information and request input for further Process execution.

This data can be mapped to the Task as DataInputSet and used as the Task's local variables and to DataOutputSet to provide the data to the parent Process instance (refer to [Section 4.11, “Assignment”](#)).

4.6.1. Defining Process form

A Process form is a form that is displayed at Process instantiation to the user who instantiated the Process.

To create a Process form, do the following:


1. Open your Process definition in the Process Designer.
2. In the editor toolbar, click the **Form** () icon and then **Edit Process Form**.
3. Select the editor to use to edit the form. Note that this document deals only with the **Graphical Modeler** option.

Note that the Form is created in the root of your current Project and is available from any other Process definitions in the Projects.


4.6.2. Defining Task form

A Task form is a form that is displayed at User Task instantiation, that is, when the execution flow reaches the Task, to the Actor of the User Task.

To create a Task form, do the following:

1. Open your Process definition with the User Task in the Process Designer.
2. Select the Task on the canvas and click the **Edit Process Form** () in the User Task menu.
3. In the displayed Form Editor, define the Task form.

4.6.3. Defining form fields

Once you have created a form definition, you need to define its content: that is its fields and the data they are bound to. You can add either the pre-defined field types to your form, or define your own data origin and use the custom field types in your form definition. 



NOTE

Automatic form generation is not recursive, which means that when custom data objects are used, only the top-level form is generated (no subforms). The user is responsible for creating forms that represent the custom data objects and link them to the parent form.

4.7. FORM MODELER

Red Hat JBoss BPM Suite provides a custom editor for defining forms called Form Modeler.

Form Modeler includes the following 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

Form Modeler comes with predefined field types, such as **Short Text**, **Long Text**, or **Integer**, which you place onto the canvas to create a form. In addition to that, Form Modeler also allows you to create custom types based on data modeler classes, Java classes (must be on the classpath), or primitive Java data types. For this purpose, the **Form data origin** tab contains three options: **From Data Model**, **From Java Class**, and **From basic type**.

Use the **Add fields by origin** tab visible in the following figure to select fields based on their source.

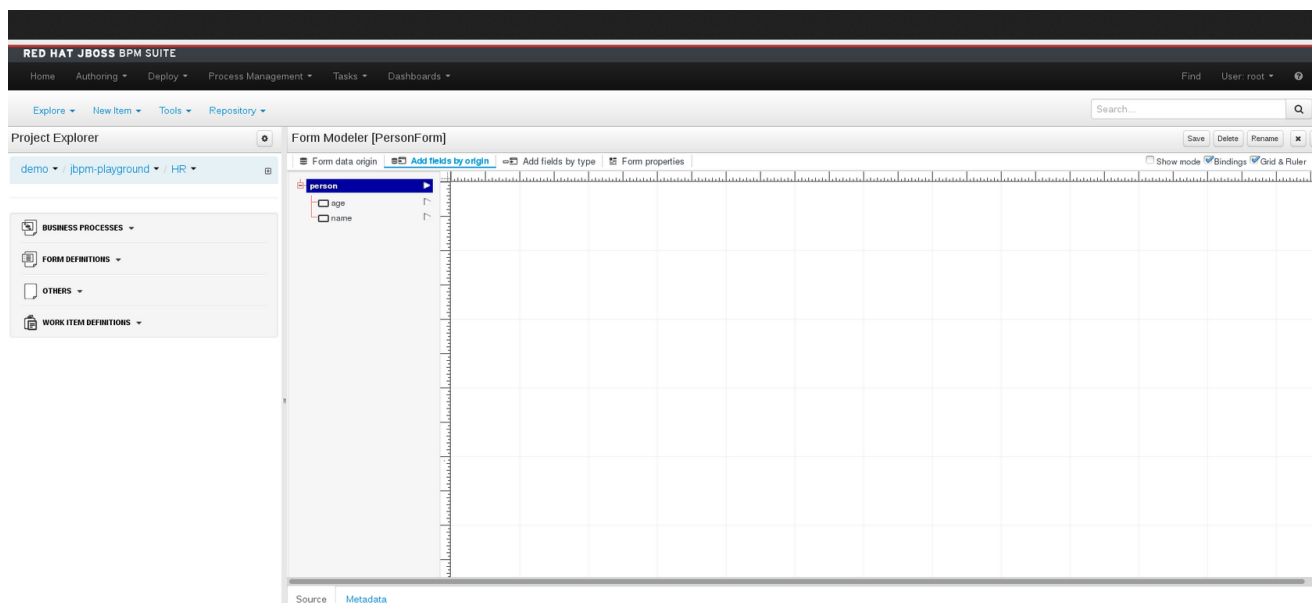


Figure 4.8. Adding fields by origin

To view and add Java classes created in Data Modeler in Form Modeler, go to section **Form data origin** and select the **From Data Model** option shown in the following figure.

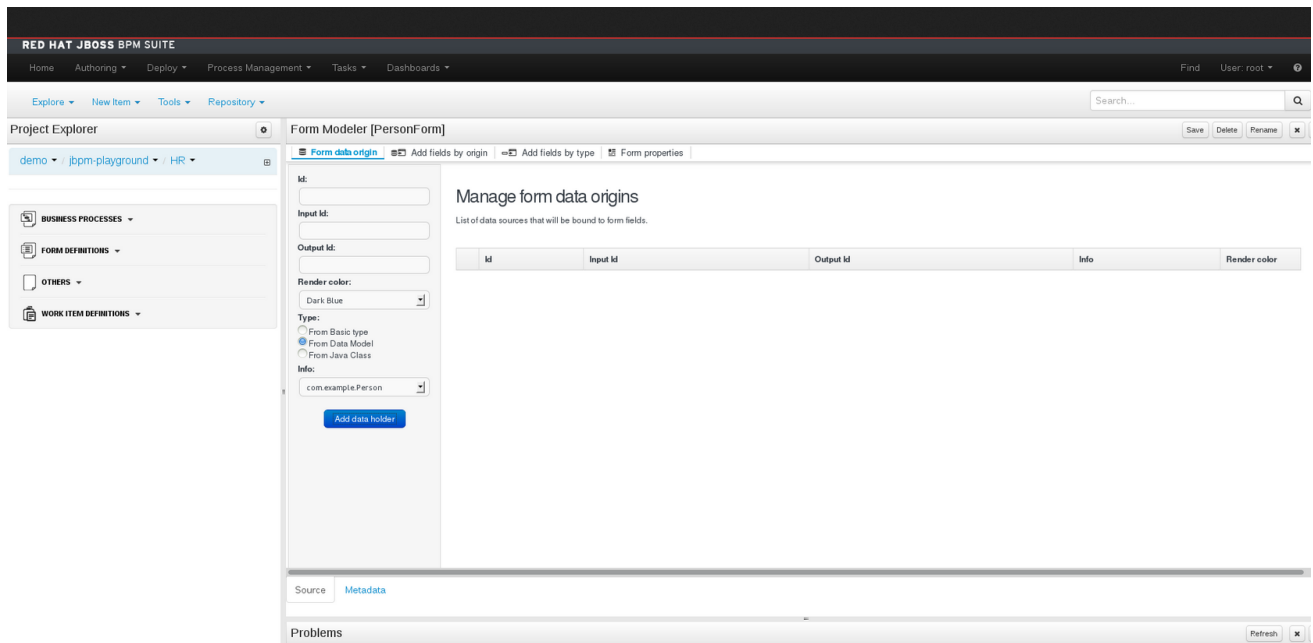


Figure 4.9. Adding classes from data model


You can adjust the form layout using the **Form Properties** tab that contains a **Predefined** layout selected by default, as well as a **Custom** option.

When a task or process calls a form, it sends the form a map of objects, which include local variables of the process or task. Also, when the form is completed, a map is sent back to the process or task with the data acquired in the form. The form assigns this output data to the local variables of the task or process, and the output data can therefore be further processed.

4.7.1. Creating a Form in Form Modeler

To create a new form in Form Modeler, do the following:

1. In Business Central, go to **Authoring** → **Project Authoring**.
2. On the perspective menu, select **New Item** → **Form**.
3. In the **Create New Form** dialog that opens, fill out the name of your form in **Resource Name** and click **OK**.

The newly created form will open up. You can add various fields to it when you select the **Add fields by type** option on the Form Modeler tab. Use the  button to place the field types onto the canvas, where you can modify them. To modify the field types, use the icons that display when you place the cursor over a field: **First**, **Move field**, **Last**, **Group with previous**, **Edit**, or **Clear**. The icons enable you to change the order of the fields in the form, group the fields, or clear and edit their content.

The following figure shows a new form created in Form Modeler.

Form Modeler [Order Form] Save Delete Rename x ▾

Form data origin | Add fields by origin | Add fields by type | Form properties

Id:

Input Id:

Output Id:

Render color:
Dark Blue ▾

Type:
☐ From Basic type
☐ From Data Model
☐ From Java Class

Info:

[Add data holder](#)

Manage form data origins

List of data sources that will be bound to form fields.

Id	Input Id	Output Id	Type	Info	Render color
----	----------	-----------	------	------	--------------

Figure 4.10. New form

4.7.2. Opening an Existing Form in Form Modeler

To open an existing form in a project that already has a form defined, go to **Form Definitions** in Project Explorer and select the form you want to work with from the displayed list.

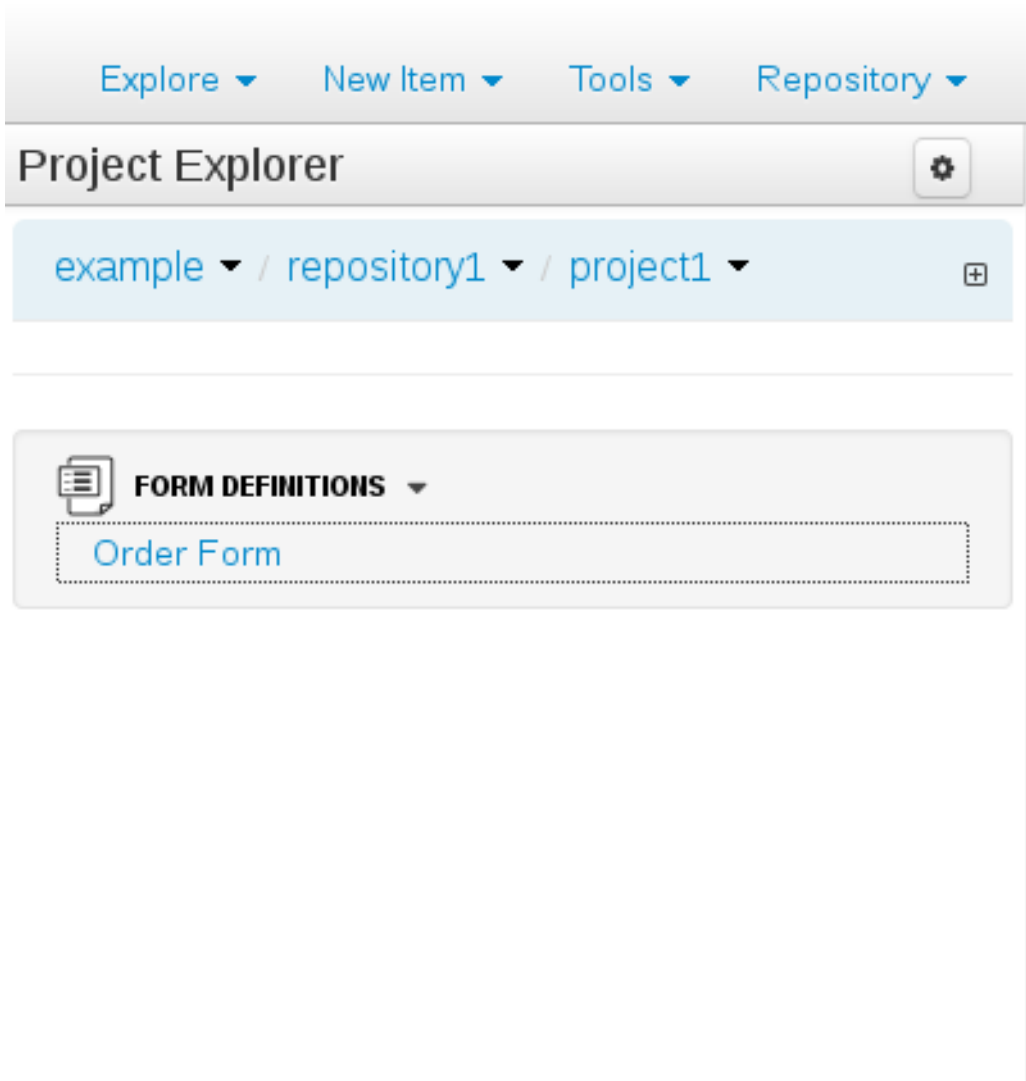




Figure 4.11. Opening an Existing Form

4.7.3. Setting Properties of a Form Field in Form Modeler

To set the properties of a form field, do the following:

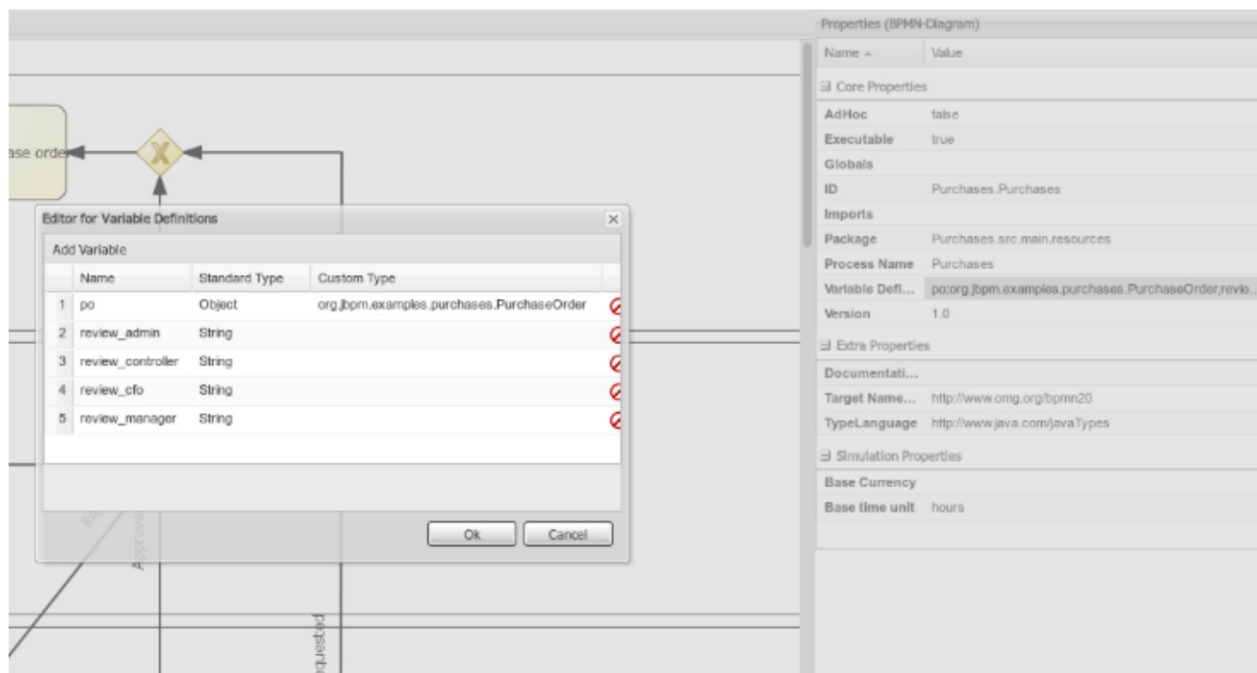
1. In Form Modeler, select the **Add fields by type** tab and click the arrow  button to the right of a field type. The field type is added to the canvas.
2. On the canvas, place the cursor on the field and click the edit  icon.
3. In the **Properties** dialog that opens on the right, set the form field properties and click **Apply** at the bottom of the dialog for HTML Labels. For other form field properties, the properties change once you have removed focus from the property that you are modifying.

4.7.4. Configuring a Process in Form Modeler

You can generate forms automatically from process variables and task definitions and later modify the forms using the form editor. In runtime, forms receive data from process variables, display it to the user, capture user input, and update the process variables with the new values. To configure a process in Form Modeler, do the following:

1. Create process variables to hold values entered into forms. Variables can be simple (e.g. 'string') or complex. You can define complex variables using Data Modeler, or create them in any Java integrated development environment (Java IDE) as regular plain Java objects.
2. Declare the process variables in the 'variables definition' property.
3. Determine which variables you want to set as input parameters for the task, which shall receive response from the form, and establish mappings by setting the 'DataInputSet', 'DataOutputSet', and 'Assignments' properties for any human task. To do so, use the Editor for Data Input, Editor for Data Output, and Editor for Data Assignment.

Example 4.1. Defining a Variable using Data Modeler



4.7.5. Generating Forms from Task Definitions

In the Process Designer module, you can generate forms automatically from task and variable definitions, and easily open concrete forms from Form Modeler by using the following menu option:

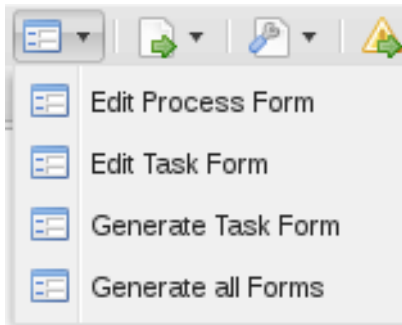


Figure 4.12. Generating Forms Automatically

To open and edit a form directly, click the Edit Task Form icon () located above a user task.

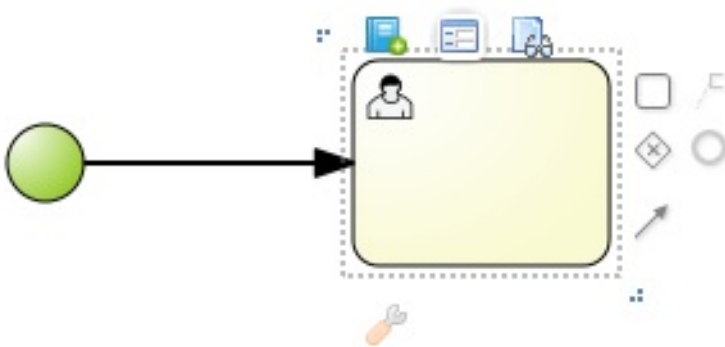


Figure 4.13.

Forms follow a naming convention that relates them to tasks. If you define a form named **formName-taskform** in the same package as the process, the human task engine will use the form to display and capture information entered by the user. If you create a form named **ProcessId-task**, the application will use it as the initial form when starting the process.

4.7.6. Editing Forms

After you generate a form, you can start editing it. If the form has been generated automatically, the **Form data origin** tab contains the process variables as the origin of the data, which allows you to bind form fields with them and create data bindings. Data bindings determine the way task input is mapped to form variables, and when the form is validated and submitted, the way values update output of the task. You can have as many data origins as required, and use different colors to differentiate them in the **Render color** drop down menu. If the form has been generated automatically, the application creates a data origin for each process variable. For each data origin bindable item, there is a field in the form, and these automatically generated fields also have defined bindings. When you display the fields in the editor, the color of the data origin is displayed over the field to give you quick information on correct binding and implied data origin. To customize a form, you can for example **move fields**, **add new fields**, **configure fields**, or **set values for object properties**.

4.7.7. Moving a Field in Form Modeler

You can place fields in different areas of the form. To move a field, access the field's contextual menu and select the **Move field** option shown on the following screenshot. This option displays the different regions of the form where you can place the field.

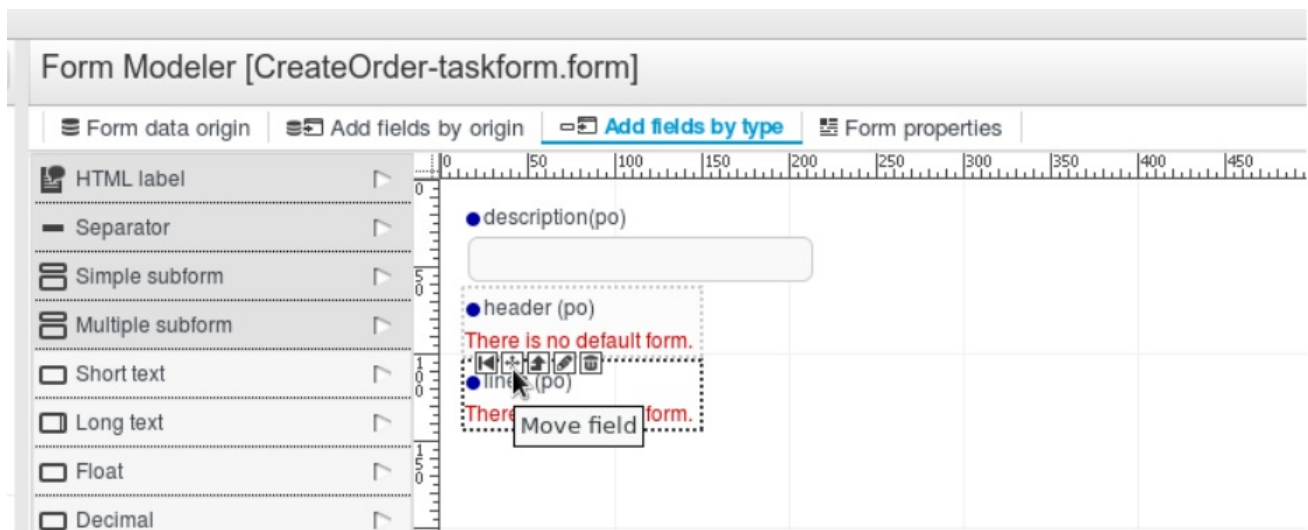


Figure 4.14. Moving a Form Field in Form Modeler

After you click the **Move field** option, a set of rectangular contextual icons appears. To move a field, select one of them according to the desired new position of the field.

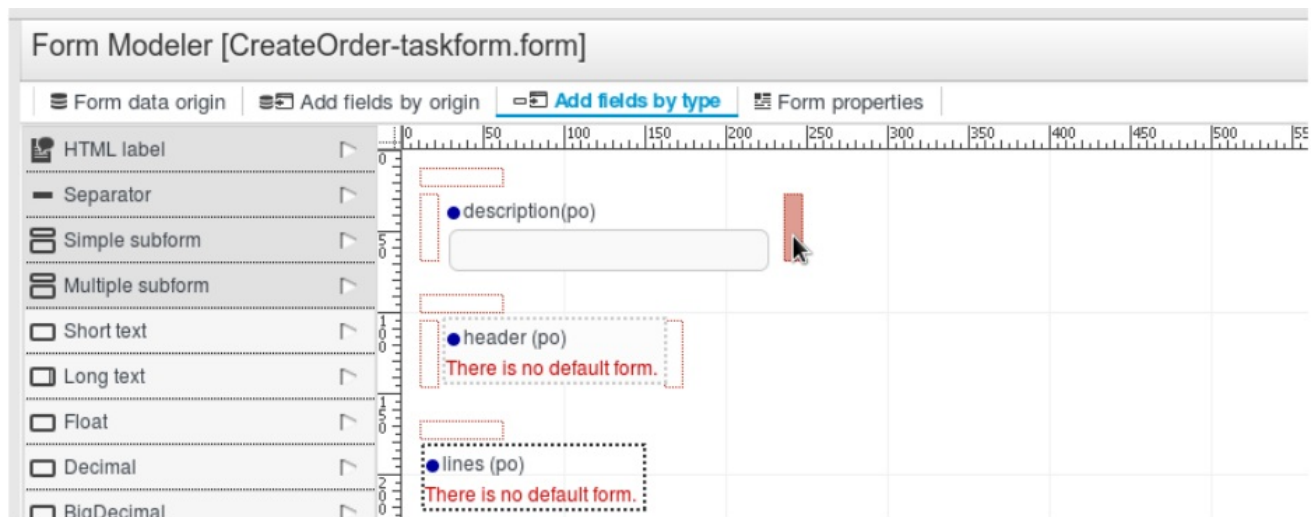


Figure 4.15. Destination Areas to Move a Field

4.7.8. Adding New Fields to a Form

You can add fields to a form by their origin or by selecting the type of the form field. The **Add fields by origin** tab allows you to add fields to the form based on defined data origins.

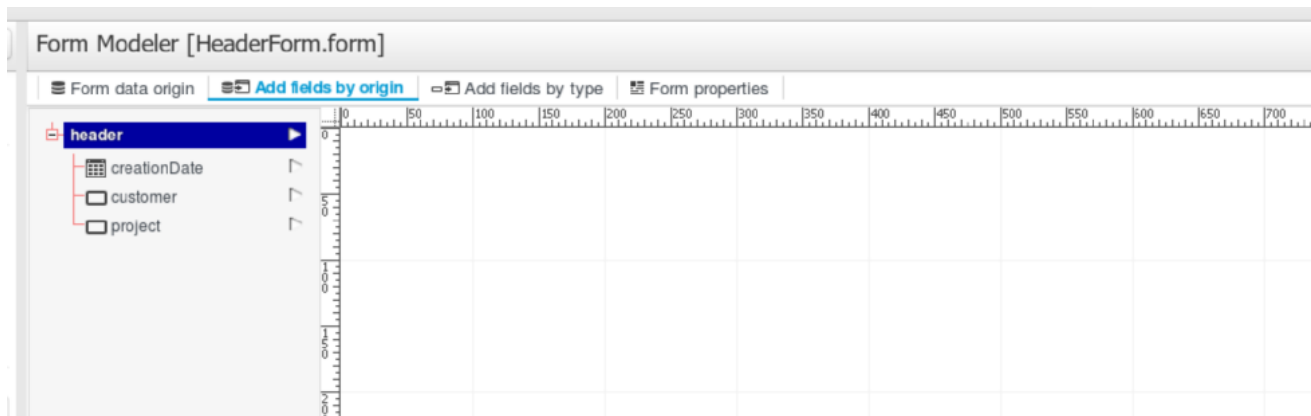


Figure 4.16. Adding Fields by Origin

The fields then have correct configuration of the **Input binding expression** and **Output binding expression** properties, so when the form is submitted, the values in the fields are stored in the corresponding data origin. The **Add fields by type** tab allows you to add fields to the form from the fields type palette of the Form Modeler. The fields do not store their value for any data origin until they have correct configuration of the **Input binding expression** and **Output binding expression** properties.

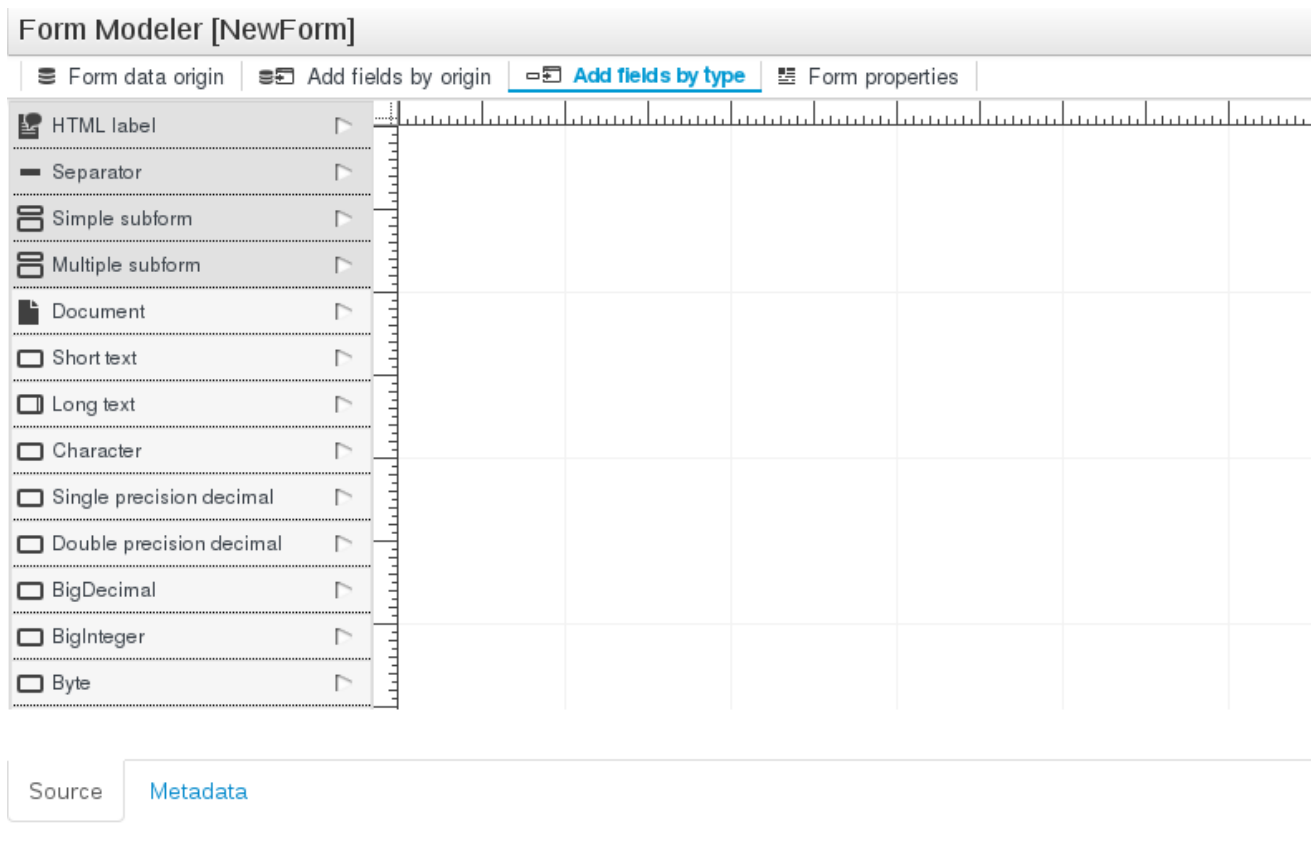


Figure 4.17. Adding Fields by Type

There are three kinds of field types you can use to model your form: simple types, complex types, and decorators. The **simple types** are used to represent simple properties like texts, numeric values, or dates. The following table presents a complete list of supported simple field types:

Table 4.1. Simple 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	HTML Editor 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 and time values.	java.util.Date	yes
Short Date	Input to enter date values.	java.util.Date	no
Document	Allows the user to upload documents to the form.	org.jbpm.document.Document	No

Complex field types are designed for work with properties that are not basic types but Java objects. To use these field types, it is necessary to create extra forms in order to display and write values to the specified Java objects.

Table 4.2. Complex Field Types

Name	Description	Java Type	Default on generated forms
Simple subform	Renders the form; it is used to deal with 1:1 relationships.	java.lang.Object	yes
Multiple subform	This field type is used for 1:N relationships. It allows the user to create, edit, and delete a set child Objects.Text area to enter long text.	java.util.List	yes

Decorators are a kind of field types that does not store data in the object displayed in the form. You can use them for decorative purposes.

Table 4.3. Decorators

Name	Description
HTML label	Allows the user to create HTML code that will be rendered in the form.
Separator	Renders an HTML separator.

4.7.9. Configuring Fields of a Form

Each field can be configured to enhance performance of the form. There is a group of common properties called generic field properties and a group of specific properties that differs by field type.

Generic field properties:

- **Field Type** - can change the field type to other compatible field types.
- **Field Name** - is used as an identifier in calculating of formulas.
- **Label** - the text that is displayed as a field label.
- **Error Message** - a message displayed when there is a problem with a field, for example in validation.
- **Label CCS Class** - allows you to enter a class css to apply in label visualization.
- **Label CCS Style** - allows you to directly enter the style to be applied to the label.

- **Help Text** - introduced text displayed as an alternative attribute to help the user in data introduction.
- **Style Class** - allows you to enter a class CSS to be applied in field visualization.
- **CSS Style** - allows you to directly enter the style to be applied to the label.
- **Read Only** - a field with this property allows reading only, no write access.
- **Input Binding Expression** - defines the link between the field and the process task input variable. In runtime, it is used to set the field value to the task input variable data.
- **Output Binding Expression** - defines the link between the field and the process task output variable. In runtime, it is used to set the task output variable.

4.7.10. Creating Subforms with Simple and Complex Field Types

Complex Field types is a category of fields in a form. You can use the complex field types to model form properties that are Java Objects. Simple subform and Multiple subform are the two types of complex field types. A simple subform represents a single object and a multiple subform represents an object array inside a parent form. Once you add one of these fields into a form, you must configure the form with information on how it must display these objects during execution. For example, if your form has fields representing an object array, you can define a tabular display of these fields in the form. You cannot represent them as simple inputs such as text box, checkbox, text area, and date selector.

Procedure 4.2. To create and insert a subform containing a single object inside a parent form:

1. In Business Central, go to **Authoring** → **Project Authoring**.
2. On the perspective menu, select **New Item** → **Form**.

A new form opens in the Form Modeler. You must now configure the new form with information of the object it must contain.

3. Enter the values for the required fields in the **Form data origin** tab and click **Add data holder**.

The screenshot shows the 'Form Modeler [subform]' window. The 'Form data origin' tab is active. On the left, there are input fields for 'Id' (name), 'Input Id' (name_in), and 'Output Id' (name_out). Below these is a 'Render color' dropdown set to 'Yellow' and a 'Type' section with radio buttons for 'From Basic type' (selected), 'From Data Model', and 'From Java Class'. An 'Info' dropdown is set to 'java.lang.Boolean'. A blue 'Add data holder' button is at the bottom of the sidebar. The main area is titled 'Manage form data origins' and contains a table with columns: Id, Input Id, Output Id, Type, Info, and Render color. The table is currently empty.

Figure 4.18. Create Subform

4. Click **Add fields by origin** tab and add the listed fields to the form.

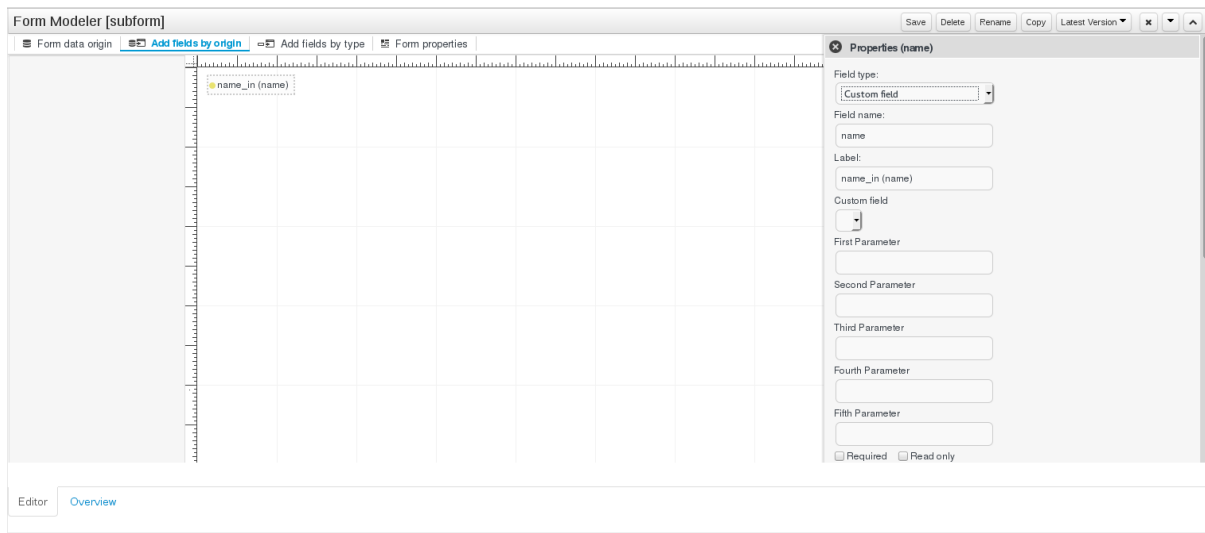


Figure 4.19. Add fields by origin

5. Click the Edit icon on the field in the form to open the **Properties** tab.
6. In the **Properties** tab, configure the form by providing required values to the fields and click **Save** to save the subform.
7. Open the parent form to configure the properties of the object.
8. In the parent form, click the **Add fields by type** tab. Select the object on the form and configure it in the **Properties** tab.
9. In the **Properties** tab, select **Simple subform** for the **Field type** property. Then select the newly created subform for the **Default form** field property.

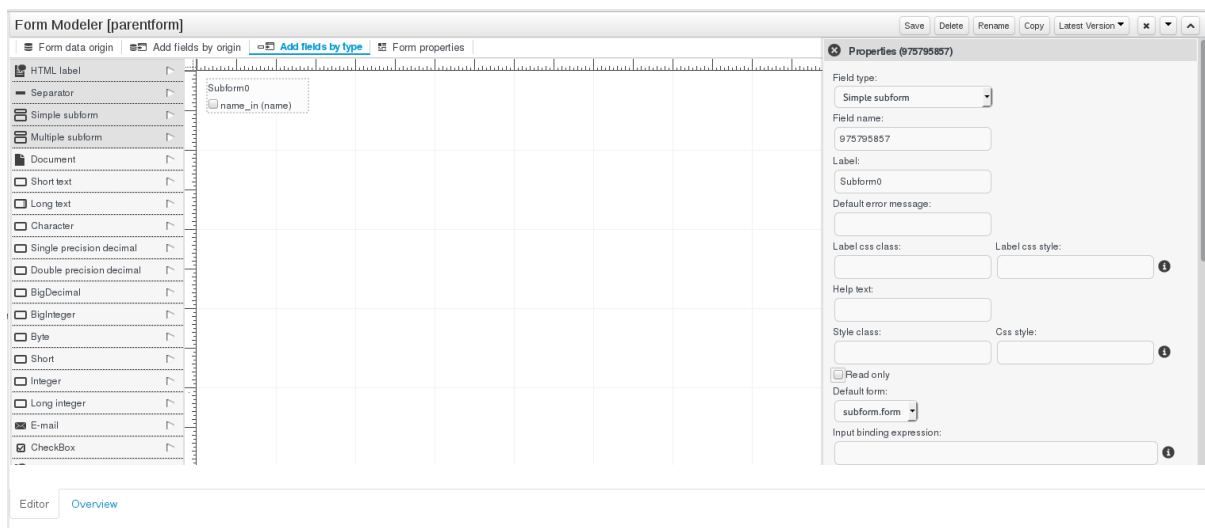


Figure 4.20. Configure the Parent Form

10. Click **Save** to save the parent form.

This inserts your subform containing a single Java object inside the parent form.

Procedure 4.3. To insert a subform with multiple objects inside a parent form:

1. In Business Central, go to **Authoring** → **Project Authoring**.

- On the perspective menu, select **New Item** → **Form**.

A new form opens in the Form Modeler. You must now configure the new form with information on the object array it must contain.

- Enter the values for the required fields in the **Form data origin** tab and click **Add data holder**.
- Click **Add fields by origin** tab and add the listed fields to the form.
- Click the Edit icon on the field in the form to open the **Properties** tab.
- In the **Properties** tab, configure the form by providing required values to the fields. You can use the Formula Engine to automatically calculate field values.
- Click **Save** to save the subform.
- Open the parent form to configure the properties of each of the objects.
- In the parent form, click the **Add fields by type** tab. Select each object on the form one by one and configure them in the **Properties** tab.
- In the **Properties** tab, select **Multiple subform** for the **Field type** property. Then select the newly created subform for the **Default form** field property.

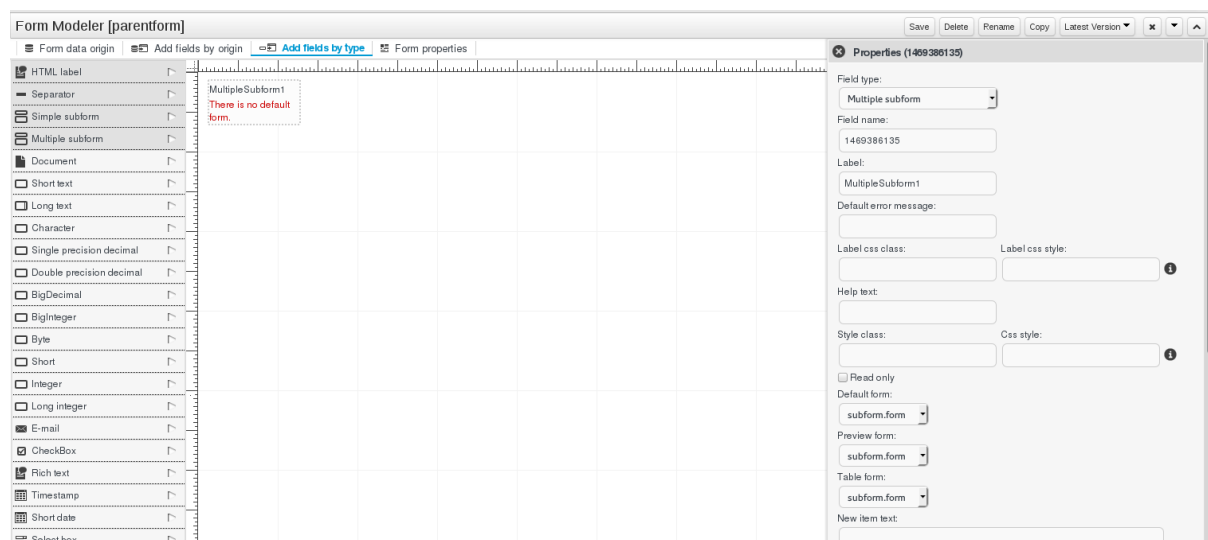


Figure 4.21. Configure the Parent Form

- Click **Save** to save the parent form.

This inserts your subform containing an array of Java objects inside the parent form.

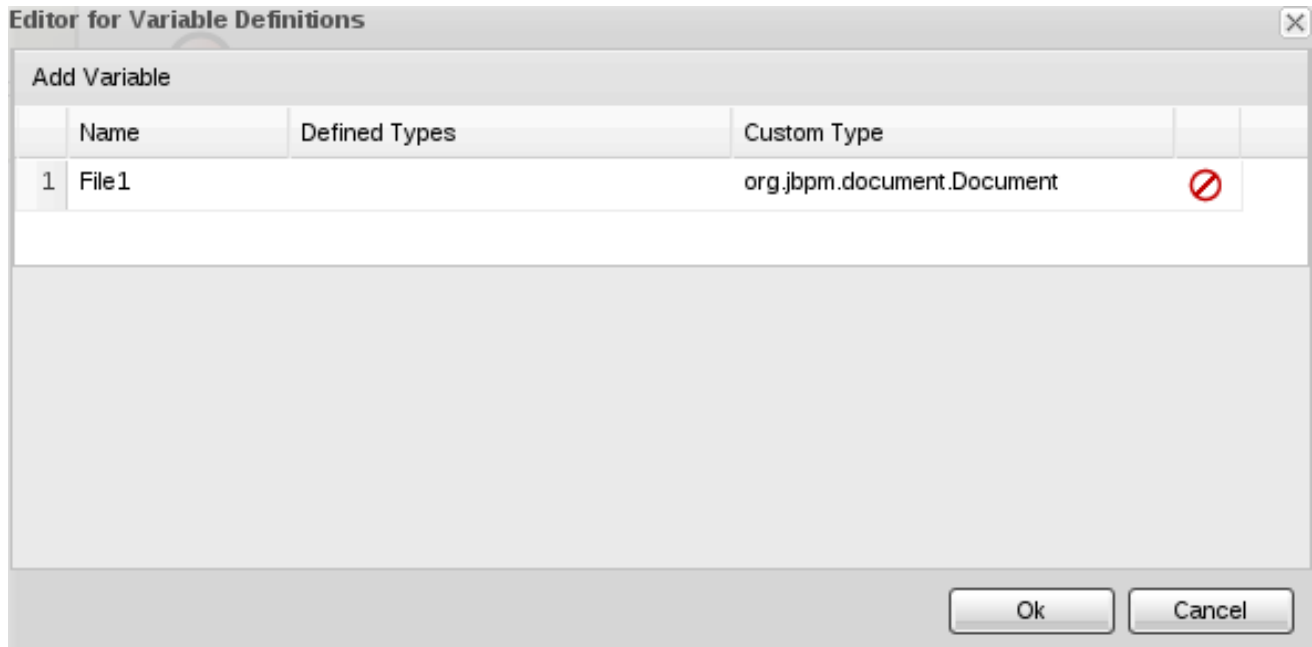
4.7.11. Attaching Documents to a Form

The 6.1 release of Red Hat JBoss BPM Suite has introduced the concept of attaching documents to a form by using a **Document** form field. This field can be attached to any form, process or task based.


To attach a **Document** field to a form, click on it while creating or editing an existing form (in the **Add fields by type** section).

<input checked="" type="radio"/> Radio group	⌵	:
<input type="checkbox"/> Document	⌵	:

Alternatively, you can let the system automatically generate the **Document** form fields based on the presence of a `org.jbpm.document.Document` variable type in your process.



The dialog box titled "Editor for Variable Definitions" contains a table with the following data:

	Name	Defined Types	Custom Type	
1	File1		org.jbpm.document.Document	

At the bottom right are "Ok" and "Cancel" buttons.

When the end user uploads a document using this form, the uploaded document is available in a list in the task list (**task details**) and process instance details (**Document** tab). The uploaded documents are shown as links and the users can click to view them.

Details

1 - TestMe

Options ▾  

Instance Details

Process Variables

Documents

Logs



Name

Size

Actions

The Process Engine generates an instance of `org.jbpm.document.Document` to be stored in a process variable, which you can pass through your persistence strategy to decide where and how to store that document.

Pluggable Variable Persistence

New with this release is also the ability for you to store your document in a location of your choice. This is defined as *Pluggable Variable Persistence* and this allows you to store these documents automatically in a centralized content management system (CMS) of choice, behind the scenes.

To implement your custom persistence strategy, start by defining your *Marshaling Strategy*. This strategy is declared to the Process Engine by the use of deployment descriptors (see *Red Hat JBoss BPM Suite Administration and Configuration Guide*) using the `<marshalling-strategy>` element. This element should name a type that provides an implementation of the `org.kie.api.marshalling` interface.

The following methods in this interface help you create your strategy.

- **public boolean accept(Object object)**: Determines if the given object can be marshalled by the strategy.
- **byte[] marshal(Context context, ObjectOutputStream os, Object object)**: Marshals the given object and returns the marshalled object as `byte[]`.
- **Object unmarshal(Context context, ObjectInputStream is, byte[] object, ClassLoader classloader)**: Reads the object received as `byte[]` and returns the unmarshalled object
- **void write(ObjectOutputStream os, Object object)**: same as `marshal` method, provided for backwards compatibility.
- **Object read(ObjectInputStream os)**: same as `unmarshal`, provided for backwards compatibility.

For example, if you create a custom strategy that stores your uploaded documents in Google Drive, your implementation class should not only implement the methods of the `org.kie.api.marshalling` package, but this implementation should also be made available to the Process Engine, by putting the classes in its classpath (and declaring the type in the deployment descriptors).

There is a default marshalling strategy that simply saves the uploaded documents in the file system under a folder called **docs**. This default implementation is defined by the **DocumentStorageService** class and is implemented through the **DocumentStorageServiceImpl** class.

4.7.12. Rendering Forms for External Use

Forms generated by the Form Builder can be reused in other client applications with the help of the REST API and a JavaScript library. The REST API defines the end points for the external client applications to call and the JavaScript library makes it easy to interact with these endpoints and to render these forms.

To use this API you will need to integrate the Forms REST JavaScript library in your client application. The details of the library and the methods that it provides are given in the following section, along with a simple example. Details of the REST API are present in the *Red Hat JBoss BPM Suite Developers Guide*, although you should probably only use the REST API via the JavaScript library described here.

4.7.12.1. JavaScript Library for Form Reuse

The JavaScript API for Form Reuse makes it easy to use forms created in one Business Central application to be used in remote applications and allows loading of these forms from different Business Central instances, submitting them, launching processes or task instances, and executing callback functions when the actions are completed.

Blueprint for using the JavaScript Library

A simple example of using this API would involve the following steps:

1. Integrate the JavaScript library in the codebase for the external client application so that its functions are available.

2. Create a new instance of the **jBPMFormsAPI** class in your own JavaScript code. This is the starting point for all interactions with this library.

```
var jbpRestAPI = new jBPMFormsAPI();
```

3. Call your desired methods on this instance. For example, if you want to show a form, you would use the following method:

```
jbpRestAPI.showStartProcessForm(hostUrl, deploymentId, processId,
divId, onSuccess, onError);
```

and provide the relevant details (hostUrl, deploymentId, processId and so on. A full list of the methods and parameters follows after this section).

4. Do post processing with the optional **onSuccess** and **onError** methods.
5. Work with the form, starting processes (**startProcess()**), claiming tasks (**claimTask()**) starting tasks (**startTask()**) or completing tasks (**completeTask**). Full list of available methods follows after this section.
6. Once you're finished with the form, clear the container that displayed it using **clearContainer()** method.

Full list of available methods in the JavaScript Library

The JavaScript library is pretty comprehensive and provides several methods to render and process forms.

1. **showStartProcessForm(hostUrl, deploymentId, processId, divId, onSuccessCallback, onErrorCallback)**: Makes a call to the REST endpoint to obtain the form URL. If it receives a valid response, it embeds the process start form in the stated div. You need these parameters:
 - **hostURL**: The URL of the Business Central instance that holds the deployments.
 - **deploymentId**: The deployment identifier that contains the process to run.
 - **processId**: The identifier of the process to run.
 - **divId**: The identifier of the div that has to contain the form.
 - **onSuccessCallback** (optional): A JavaScript function executed if the form is going to be rendered. This function will receive the server response as a parameter.
 - **onErrorCallback** (optional): A JavaScript function executed if any error occurs and it is impossible to render the form. This function will receive the server response as a parameter.
2. **startProcess(divId, onSuccessCallback, onErrorCallback)**: Submits the form loaded on the stated div and starts the process. You need these parameters:
 - **divId**: The identifier of the div that contains the form.
 - **onSuccessCallback**(optional): A JavaScript function executed after the process is started. This function receives the server response as a parameter.

- **onerrorCallback** (optional): A JavaScript function executed if any error occurs and it is impossible to start the process. This function receives the server response as a parameter.
3. **showTaskForm(hostUrl, taskId, divId, onSuccessCallback, onerrorCallback)**: Makes a call to the REST endpoint to obtain the form URL. If it receives a valid response, it embeds the task form in the stated div. You need these parameters:
- **hostURL**: The URL of the Business Central instance that holds the deployments.
 - **taskId**: The identifier of the task to show the form.
 - **divId**: The identifier of the div that has to contain the form.
 - **onSuccessCallback** (optional): A JavaScript function executed if the form is going to be rendered. This function receives the server response as a parameter.
 - **onerrorCallback** (optional): A JavaScript function executed if any error occurs and it is impossible to render the form. This function receives the server response as a parameter.
4. **claimTask(divId, onSuccessCallback, onerrorCallback)**: Claims the task whose form is being rendered. You need these parameters:
- **divId**: The identifier of the div that contains the form.
 - **onSuccessCallback** (optional): A JavaScript function executed after the task is claimed. This function receives the server response as a parameter.
 - **onerrorCallback** (optional): A JavaScript function executed if any error occurs and it is impossible to claim the task. This function receives the server response as a parameter.
5. **startTask(divId, onSuccessCallback, onerrorCallback)**: Starts the task whose form is being rendered. You need these parameters:
- **divId**: The identifier of the div that contains the form.
 - **onSuccessCallback** (optional): A JavaScript function executed after the task is claimed. This function receives the server response as a parameter.
 - **onerrorCallback** (optional): A JavaScript function executed if any error occurs and it is impossible to claim the task. This function receives the server response as a parameter.
6. **releaseTask(divId, onSuccessCallback, onerrorCallback)**: Releases the task whose form is being rendered. You need these parameters:
- **divId**: The identifier of the div that contains the form.
 - **onSuccessCallback** (optional): A JavaScript function executed after the task is claimed. This function receives the server response as a parameter.
 - **onerrorCallback** (optional): A JavaScript function executed if any error occurs and it is impossible to claim the task. This function receives the server response as a parameter.
7. **saveTask(divId, onSuccessCallback, onerrorCallback)**: Submits the form and saves the state of the task whose form is being rendered. You need these parameters:
- **divId**: The identifier of the div that contains the form.

- **onsuccessCallback** (optional): A JavaScript function executed after the task is claimed. This function receives the server response as a parameter.
 - **onerrorCallback** (optional): A JavaScript function executed if any error occurs and it is impossible to claim the task. This function receives the server response as a parameter.
8. **completeTask(divId, onsuccessCallback, onerrorCallback)**: Submits the form and completes task whose form is being rendered. You need these parameters:
- **divId**: The identifier of the div that contains the form.
 - **onsuccessCallback** (optional): A JavaScript function executed after the task is claimed. This function receives the server response as a parameter.
 - **onerrorCallback** (optional): A JavaScript function executed if any error occurs and it is impossible to claim the task. This function receives the server response as a parameter.
9. **clearContainer(divId)**: Cleans the div content and the related data stored on the component. You need these parameters:
- **divId**: The identifier of the div that contains the form.

4.8. VARIABLES

Variables are elements that serve for storing a particular type of data during runtime. The type of data a variable contains is defined by its data type.

Just like any context data, every variable has its scope that defines its "visibility". An element, such as a Process, Sub-Process, or Task can only access variables in its own and parent contexts: variables defined in the element's child elements cannot be accessed. Therefore, when an element requires access to a variable on runtime, its own context is searched first. If the variable cannot be found directly in the element's context, the immediate parent context is searched. The search continues to "level up" until the Process context is reached; in case of Globals, the search is performed directly on the Session container. If the variable cannot be found, a read access request returns **null** and a write access produces an error message, and the Process continues its execution. Variables are searched for based on their ID.

In Red Hat JBoss BPM Suite, variables can live in the following contexts:

- Session context: **Globals** are visible to all Process instances and assets in the given Session and are intended to be used primarily by business rules and by constraints. They are created dynamically by the rules or constraints.
- Process context: **Process variables** are defined as properties in the BPMN2 definition file and are visible within the Process instance. They are initialized at Process creation and destroyed on Process finish.
- Element context: **Local variables** are available within their Process element, such as an Activity. They are initialized when the element context is initialized, that is, when the execution workflow enters the node and execution of the OnEntry action finished if applicable. They are destroyed when the element context is destroyed, that is, when the execution workflow leaves the element.

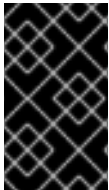
Values of local variables can be mapped to Global or Process variables using the Assignment mechanism (refer to [Section 4.11, "Assignment"](#)). This allows you to maintain relative independence of the parent Element that accommodates the local variable. Such isolation may

help prevent technical exceptions.

4.8.1. Globals

A global is a variable that exists in a Knowledge Session and can be accessed and is shared by all assets in that Session. Globals belong to the particular Session of the Knowledge Base: they are used to pass information to the engine.

Every global defines its ID and item subject reference: the ID serves as the variable name and must be unique within the Process definition. The item subject reference defines the data type the variable stores.



IMPORTANT

The rules are evaluated at the moment the fact is inserted. Therefore, if you are using a Global to constraint a fact pattern, and the global is not set, the system returns a **NullPointerException**.

4.8.1.1. Creating Globals

Globals are initialized when the Process with the variable definition is added to the Session or when the Session is initialized with Globals as its parameters. Their value can be changed by the Process Activities using the Assignment, when the global variable is associated with the local Activity context, local Activity variable, or by a direct call to the variable from a child context.

Procedure 4.4. Defining a Global in the Process Designer

To define a Process variable, do the following:

1. Open the Process in the Process Designer.
2. In the **Properties** panel of the BPMN Diagram expand the Extra item.
3. Click the empty value cell next to the Globals and click the arrow.

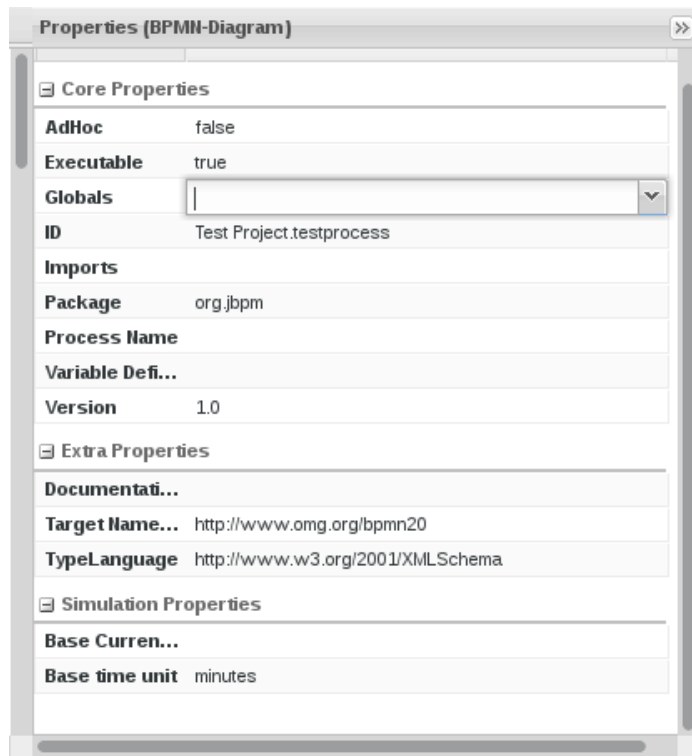


Figure 4.22. Global properties cell

4. In the **Editor for Variable Definitions** window, click the **Add Variable** button and define the variable details.

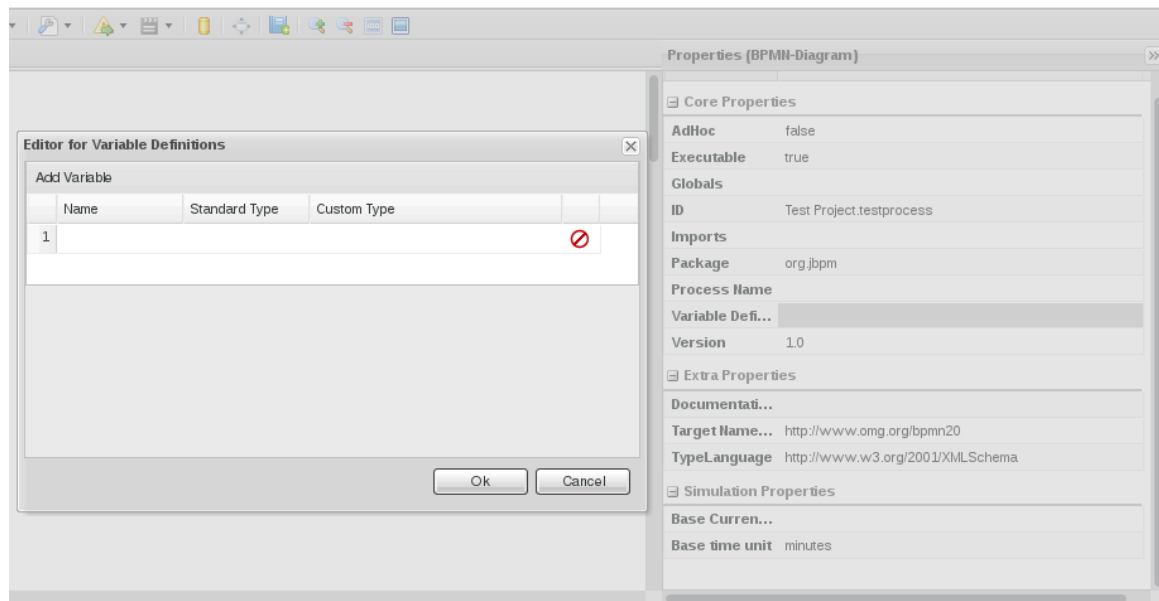


Figure 4.23. Editor for Variable Definitions

Procedure 4.5. Defining and Initializing a Global using the API

To define and initialize global variables at process instantiation using API, do the following:

1. Define the variables as a Map of the `<String, Object>` values.
2. Provide the map as a parameter to the `startProcess()` method.

Example 4.2. Code instantiating a Process with a Global

```
Map<String, Object> params = new HashMap<String, Object>();
params.put("var", "variable value");
ksession.startProcess("Process Definition Name", params);
```

4.8.1.2. Accessing Globals

```
processInstance.getContextInstance().getVariable("globalStatus")
```

4.8.1.3. Process variables

A Process variable is a variable that exists in a Process context and can be accessed by its Process or its child elements: Process variables belong to the particular Process instance and cannot be accessed by other Process instances. Every Process variable defines its ID and item subject reference: the ID serves as the variable name and must be unique within the Process definition. The item subject reference defines the data type the variable stores.

Process variables are initialized when the Process instance is CREATED. Their value can be changed by the Process Activities using the Assignment, when the global variable is associated with the local Activity context, local Activity variable, or by a direct call to the variable from a child context .

4.8.2. Local variables

A local variable is a variable that exists in a child element context of a Process and can be accessed only from within this context: local variables belong to the particular element of a Process.

For Tasks, with the exception of the Script Task, the user can define local variable in the DataInputSet and DataOutputSet parameters: DataInputSet define variables that enter the Task and therefore provide the entry data needed for the Task execution, while the DataOutputSet variables can refer to the context of the Task after execution to acquire output data.

User Tasks typically present data related to the User Task to the actor that is executing the User Task and usually also request the actor to provide result data related to the execution. To request and provide such data, you can use Task forms and map the acquired data into the DataInputSet parameter to serve as input data of the User Task and into the DataOutputSet parameter from the User Task namespace back to the parent namespace to serve as the User Task output data (refer to [Section 4.11](#), “Assignment”).



NOTE

Local variables are initialized when the Process element instance is CREATED. Their value can be changed by their parent Activity by a direct call to the variable.

4.8.2.1. Accessing local variables

To set a variable value, call the respective setter on the variable field from the Script Activity; for example, `person.setAge(10)` sets the **Age** field of the **person** global variable to **10**.

4.9. ACTION SCRIPTS

Action scripts are pieces of code that define the Script property or an Element's interceptor action. They have access to globals, the Process variables, and the predefined variable **kcontext**. Accordingly,

kcontext is an instance of `ProcessContext` class and the interface content can be found at the following location: [Interface ProcessContext](#).

Currently, dialects Java and MVEL are supported for action script definitions. Note that MVEL accepts any valid Java code and additionally provides support for nested access of parameters, for example, the MVEL equivalent of Java call `person.getName()` is `person.name`. It also provides other improvements over Java and MVEL expressions are generally more convenient for the business user.

Example 4.3. Action script that prints out the name of the person

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

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

4.10. INTERCEPTOR ACTIONS

For every Activity you can define actions that are executed before the Activity execution starts (right after the Activity has received the token), called On Entry Actions, and after the Activity execution (before the outgoing Flow is taken), called On Exit Actions.

The actions can be defined in Java in the **Properties** tab of the given Activity and you can define them either in Java or MVEL: the language is set in the **ScriptLang** property.

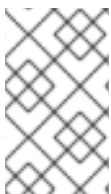
4.11. ASSIGNMENT

The assignment mechanism allows you to assign a value to an object, such as a variable, before or after the particular Element is executed.

When defining assignment on an Activity Element, the value assignment is performed either before or after Activity execution. If the assignment defines mapping to a local variable, the time when the assignment is performed depends on whether the local variable is defined as an `DataInput` or `DataOutput` item.

For example, if you need to assign a Task to a user whose ID is a Process variable, use the assignment to map the variable to the parameter `ActorId`.

Assignment is defined in the `Assignments` property in case of Activity Elements and in the `DataInputAssociations` or `DataOutputAssociations` property in case of non-Activity Elements.



NOTE

As parameters of the type `String` can make use of the assignment mechanism by applying the respective syntax directly in their value, `#{userVariable}`, assignment is rather intended for mapping of properties that are not of type `String`.

4.11.1. Defining Assignments

The assignment mechanism is available within the properties view for Activity elements as the `Assignment` property, and the assignment mechanism is available for non-Activity Elements in `DataInputAssociations` and `DataOutputAssociations` properties (note that not both are always available

depending on the semantics of the Elements).

Data Assignments are divided into two groups:

- **Data Input** - works only with Variables and Data Inputs.
- **Data Output** - works only with Variables and Data Outputs.

To define an assignment, select the activity and then open up the Properties panel. Click in the **Value** column for the **Assignments** property, and then click the drop-down arrow. This will open up the Assignment Editor.

Click one of the three buttons in this editor to add a new assignment. Available objects and assignment values, created separately, are available after clicking the drop-down arrow in the respective field.

	Assignment Type	From Object	Assignment Type	To Object	To Value	
1	DataInput	DI1	is equal to		65.0	✗
2	DataInput	var1	is mapped to	DI2		✗
3	DataOutput	DO1	is mapped to	var2		✗

1. Input Assignment: Lets you assign a Data Input to a literal value.
2. Input Mapping: Lets you map a Data Input variable to an object.
3. Output Mapping: Lets you map a Data Output variable to an object variable.

The resulting string (as shown in the figure) for the assignments will resemble: **[din]DI1=65.0, [din]var1->DI2, [dout]DO1->var2**

It is no longer possible to add invalid assignments as used to be case in previous versions of the Assignment Editor.

4.12. CONSTRAINTS

A Constraint is a boolean expression that is evaluated when the Element with the constraint is executed. The workflow continues depending on the result of the evaluation (**true** or **false**).

There are two types of constraints:

- Code constraints are defined either in Java or MVEL. They have access to data in the working memory, including the Globals and Process variables.

Example 4.4. Code constraint defined in Java

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

Example 4.5. Code constraint defined in MVEL

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

- Rule constraints are defined in the form of BRMS rule conditions. They have access to data in the Working Memory, including the Globals. However, they cannot access the variables in its Process directly, but through the Process instance: to acquire the reference of the parent Process instance, use the **processInstance** variable of the type **WorkflowProcessInstance**. Note that you need to insert the Process instance into the Session and update it if necessary, for example, using Java code or an on-entry or on-exit or explicit action in your Process.

Example 4.6. Rule constraint with process variable assignment

```
import org.kie.api.runtime.process.ProcessInstance;
import org.kie.api.runtime.process.WorkflowProcessInstance;
...
processInstance : WorkflowProcessInstance()
Person( name == ( processInstance.getVariable("name") ) )
```

The rule constraint acquires the Process variable **name**.

JBoss BPM Suite includes a script editor for Java expressions; the constrain condition allows code constraints for scripts in Java as demonstrated by the editor below.

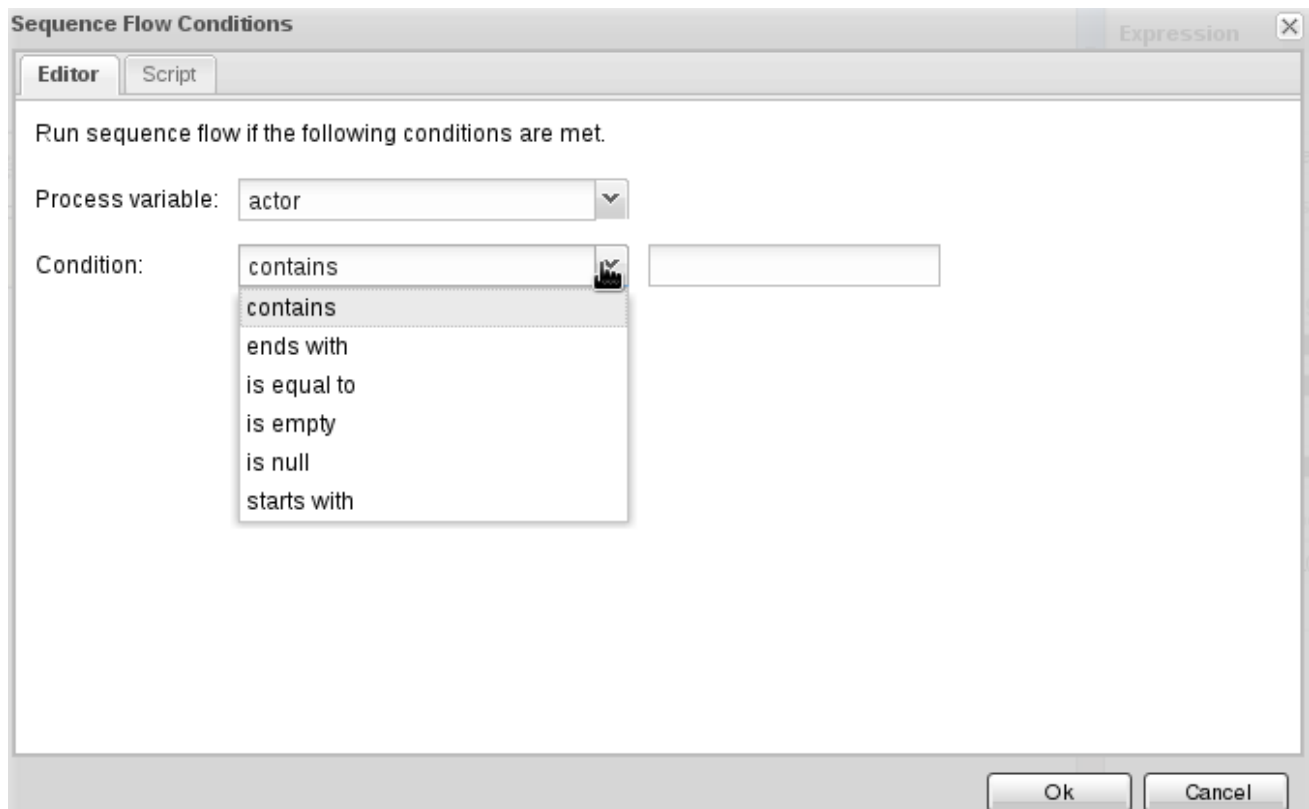


Figure 4.24. Script Editor

When a script for Java cannot be represented by this editor, it shows an alert like the following:

```
return true;
```

4.13. DATA MODELS

Data models are models of data objects. A data object is a custom complex data type (for example, a Person object with data fields Name, Address, and Date of Birth).

Data models are saved in data models definitions stored in your Project. Red Hat JBoss BPM Suite provides the Data modeler, a custom graphical editor, for defining data objects.



IMPORTANT

Every data object is implemented as a POJO and you need to import its class explicitly into your Process definition to allow the Process definition to see the data object.

4.13.1. Data Modeler

The Data Modeler is the built-in editor for creating data objects as part of a Project data model from Business Central. Data objects are custom data types implemented as POJOs. These custom data types can then be used in any resource (such as a Process) after importing them.

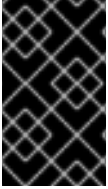
To open the editor, open the Project Authoring perspective, click **New Item** → **Data Object** on the perspective menu. If you want to edit an existing model, these files are located under **Data Objects** in **Project Explorer**.

You will be prompted to enter the name of this model object, when creating a new model, and asked to select a location for it (in terms of the package). On successful addition, it will bring up the editor where you can create fields for your model object.

The Data Modeler supports roundtrips between the **Editor** and **Source** tabs, along with source code preservation. This allows you to make changes to your model in external tools, like JBDS, and the Data Modeler updates the necessary code blocks automatically.

4.13.2. Creating a data object

1. Open the Data Modeler: in the Project Authoring perspective, click **New Item** → **Data Object** on the perspective menu. Enter the name (unique across the whole project and not just the package) and the location and press the **Ok** button.
2. Create fields of the data object:
 - a. In the **Create new field** part of the **Fields** panel, define the field properties:
 - **Id**: field ID unique within the data object
 - **Label**: label to be used in the **Fields** panel
 - **Type**: data type of the field
 - b. Click **Create**.



IMPORTANT

To use a data object, make sure you import the data model into your resource. This is necessary even if the data model lives in the same Project as your resource (Business Process).

4.13.3. Annotations in Data Modeler

The Data Modeler in Business Central supports the editing of pre-defined annotations of fact model classes and attributes. For the fact model, the annotations supported are: TypeSafe, Role, Timestamp, Duration and Expires. For the fields within the fact model, the position annotation is supported.

If you want to add/edit custom or pre-defined annotations, you can switch to the source tab and modify the source code directly (in JBDS or Business Central).

4.14. DOMAIN-SPECIFIC TASKS

A domain-specific Task represents a custom Task element with custom properties and handling specific tasks for the given field or company. It is used repeatedly in different business processes and typically accommodates interactions with other technical system.

In Red Hat JBoss BPM Suite, domain-specific task nodes are referred to as **custom work items** or **custom service nodes**.

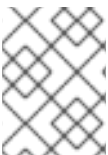
When creating custom work items, you define the following:

work item handler

The work item handler is a Java class that defines **how to execute** the custom task type. (Just like all Process elements, Tasks are executed in the Execution Engine (more precisely in the Task Engine), which contains a *work item handler* class, that defines how to handle the particular work item. Therefore, to allow the Execution Engine to execute *your* custom work item, you need to create a work item handler class for the custom work item and register it with the Execution Engine.)

work item definition

The work item definition defines how the custom task is presented (its name, icon, parameters).



NOTE

In BPMN2, custom work items are defined either as types of <task> nodes or <serviceTask> or <sendTask> nodes.

4.14.1. Work item definition

A work item definition is a resource in a project that defines how a work item is presented (its name, icon, parameters), that is, it defines the **what** part (the **how** part is implemented as a class that implements **WorkItemHandler**).

Depending on the Process Designer you are using, you define a work item definition in the following ways:

Web Process Designer

A work item definition is defined as an MVEL construct in a project resource (the custom work item node will appear on the palette; refer to [Section 4.14.1.2, “Creating a work item definition”](#)).

JBoss Developer Studio Process Designer

The BPMN2 <task> or <task>-type elements can be modified to work with **WorkItemHandler** implementations.

To do so, create a **WID_NAME.wid** file under the **META-INF** directory. (Example: **\$PROJECT_HOME/src/main/resources/META-INF/WID_NAME.wid**). The contents of this file will be the same as the ones that you will create as if under Business Central (Web Process Designer). If there are any icons, create these icons under a folder **\$PROJECT_HOME/src/main/resources/**, and store the icon images files in **icons** folder.

Once you save this file, you can use your custom service task with the JBDS Process Designer. You can find your task in the category which is defined in **WID_NAME.wid** file.

A work item has the following properties:

- **name** unique in the given work item set
- **description** with arbitrary text
- **version** number
- **parameters** with a set of work item parameters used as properties
- **displayName** used in the palette
- **icon** with the path to the icon file for the Task element
- **category** the node is added to in the palette (if the defined category does not exist, a new category is created)
- **defaultHandler** with the class that implements the **WorkItemHandler** class and is used to execute the work item
- **dependencies** the defaultHandler requires for its execution



IMPORTANT

Work item definition contains a collection of work item definitions. Therefore make sure to use square brackets correctly.

Also make sure you import any used classes and that you validate the definition once finished.

Example 4.7. Calendar work item definition

```
import org.drools.core.process.core.datatype.impl.type.StringDataType;

[
  [
    "name" : "Google Calendar",
    "description" : "Create a meeting in Google Calendar",
```

```

    "version" : "1.0",
    "parameters" : [
      "FilePath" : new StringDataType(),
      "User" : new StringDataType(),
      "Password" : new StringDataType(),
      "Body" : new StringDataType()
    ],
    "displayName" : "Google Calendar",
    "icon" : "calendar.gif",
    "eclipse:customEditor" :
"org.drools.eclipse.flow.common.editor.editpart.work.SampleCustomEditor"
  ,
    "category" : "Google",
    "defaultHandler" :
"org.jbpm.process.workitem.google.calendar.GoogleCalendarWorkItemHandler"
  ,
    "dependencies" : [
    ]
  ]
]

```

4.14.1.1. Work item handler

A work item handler is a Java class used to execute or abort work items (work items need to be aborted if their execution is to be asynchronous). The class defines the business logic of the work item, for example how to contact another system and request information which is then parsed into the custom Task parameters. Every work item handler must implement the **org.kie.api.runtime.process.WorkItemHandler** interface.



NOTE

You can customize how a custom work item is processed on a particular system by registering different work item handlers on different systems. You can also substitute a work item handler with a mock `WorkItemHandler` for testing.

Red Hat JBoss BPM Suite comes with multiple work item handlers in the following modules:

- The `jbpm-bpm2` module in the `org.jbpm.bpmn2.handler` package contains the following work item handlers:
 - `ReceiveTaskHandler` (for the BPMN `<receiveTask>` element)
 - `SendTaskHandler` (for the BPMN `<sendTask>` element)
 - `ServiceTaskHandler` (for the BPMN `<serviceTask>` element)
- The `jbpm-workitems` module in packages within `org.jbpm.process.workitem` contains work item handlers, some of which are listed below:
 - `ArchiveWorkItemHandler` creates a ZIP archive (it takes a list of files as its parameter, which are included in the archive)
 - `WebServiceWorkItemHandler`

- TransformWorkItemHandler
- RSSWorkItemHandler
- RESTWorkItemHandler
- JavaInvocationWorkItemHandler
- JabberWorkItemHandler
- JavaHandlerWorkItemHandler
- FTPUploadWorkItemHandler
- ExecWorkItemHandler
- EmailWorkItemHandler

The work item handlers must define the **executeWorkItem()** and **abortWorkItem()** methods as defined by the **WorkItemHandler** interface. These are called during runtime on work item execution.

When a work item is executed, the following is performed:

1. Information about the Task are extracted from the WorkItem instance.
2. The work item business logic is performed.
3. The Process instance is informed that the work item execution finished (completed or aborted) using the respective method of the WorkItemManager:
 - for completing execution:

```
import org.kie.api.runtime.process.WorkItemManager;
...
WorkItemManager.completeWorkItem(long workItemId, Map<String,
Object> results)
```

- for aborting execution:

```
import org.kie.api.runtime.process.WorkItemManager;
...
WorkItemManager.abortWorkItem(long workItemId, Map<String,
Object> results)
```

If a work item cannot be completed immediately and it is required that the Process execution continues while the work item completes the execution, the Process execution can continue *asynchronously* and the work item manager can be notified about the work item completion later.

To abort the work item, use the **WorkItemHandler.abortWorkItem()** before it is completed [Section 5.1.4.1, “Asynchronous execution”](#).

4.14.1.2. Creating a work item definition

To create and define a work item definition in Web Process Designer, do the following:

1. In the **Project Explorer** panel (the **Project Authoring** perspective), select your project.

2. In the perspective menu, click **New Item** → **Work Item Definition**.
3. In the **Create new** dialogue box, define the definition details:
 - o In the **Name** field provide the definition name.
 - o Click the **OK** button.
4. A new tab with the work item definition template opens up in the Work Item editor.



NOTE

Whenever a user creates a new business process in some project, the default WID will be created. Users will be able to reuse or directly alter the WID file whenever necessary. In addition, there will always be a default WID once the BPMN process is created.

5. In the editor, edit the source of the MVEL work item definition. The definition is stored in the current package.

If you are planning to add the work item using the service repository as opposed to adding the work item handler to the classpath, make sure to define its dependencies, category, etc.

If you are creating the definition out of Business Central, your project directory structure should be similar to **PROJECT_NAME/src/main/resources/PACKAGE_NAME/WID_NAME.wid** (visible in the Repository view).

Example 4.8. Example wid file

```
import
org.drools.core.process.core.datatype.impl.type.StringDataType;
import
org.drools.core.process.core.datatype.impl.type.ObjectDataType;

[
  [
    "name" : "MyTask",
    "parameters" : [
      "MyFirstParam" : new StringDataType(),
      "MySecondParam" : new StringDataType(),
      "MyThirdParam" : new ObjectDataType()
    ],
    "results" : [
      "Result" : new ObjectDataType("java.util.Map")
    ],
    "displayName" : "My Task",
    "icon" : ""
  ]
]
```

6. Upload and assign an icon to the Work Item:
 - a. Click **New Item** → **Uploaded file**.

- b. In the **Create new Uploaded file** dialogue box, define the resource name and make sure to include the file's extension in the name. Click the **Choose File** option to locate and upload the file (**png** or **gif**, 16x16 pixels). Click **Ok**.
- c. Make sure your mouse is positioned within the blank " " of the icon parameter:

```
"icon" : " "
```

Click the **Select icon to add** drop-down and click the icon file. The icon path will appear within the parameter:

```
"icon" : "ExampleIcon.png"
```



NOTE

The path to describe the location of an icon can be relative or absolute, as long as the icon can be found using one of these methods. For example, the following paths are supported:

```
"icon" : "ExampleIcon.png"           (icon in the
same folder as the .wid file)
"icon" : "com/test/ExampleIcon.png"  (icon relative
to .wid file location)
"icon" : "../test/ExampleIcon.png"   (icon relative
to .wid file location)
"icon" : "/HR/src/main/resources/ExampleIcon.png"
(absolute path to icon)
```

7. In the Process Designer, check if your work item is available in the palette.

4.14.1.3. Creating a work item handler

Once you have created the work item definition, do the following:

1. Create a maven project with your implementation of a *work item handler* with the required business logic. Make sure to call the **completeWorkItem()** function to finish the business logic execution and add the **kie-api** artifact with the **6.x.x.redhat-x** version value as the project dependency.

Example 4.9. Notification work item handler

```
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) {
        String from = (String) workItem.getParameter("From");
```

```

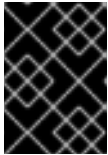
        String to = (String) workItem.getParameter("To");
        String message = (String) workItem.getParameter("Message");
        String priority = (String) workItem.getParameter("Priority");

    /*
     * Send email.
     * The ServiceRegistry class is an example class implementing the
     * task business logic.
     */
        EmailService service =
        ServiceRegistry.getInstance().getEmailService();
        service.sendEmail(from, to, "Notification", message);

    /*
     * Notify manager that work item has been completed.
     * The completeWorkItem() call completes the work item execution.
     */
        manager.completeWorkItem(workItem.getId(), null);
    }

    public void abortWorkItem(WorkItem workItem, WorkItemManager
manager) {
        // Do nothing, notifications cannot be aborted
    }
}

```



IMPORTANT

If the **WorkItemManager** is not notified about the work item completion, the process engine is never notified that your work item node has completed.

2. Register the work item handler in the **DEPLOY_DIR/business-central.war/WEB-INF/classes/META-INF/CustomWorkItemHandlers.conf** file.

The **CustomWorkItemHandlers.conf** file contains information like the following:

```

[
  "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()
]

```

Notice the "Rest" value in the previous file. This indicates the **WorkItemHandler** is capable of interacting with REST services. It supports both secured/authenticated and open/not authenticated services.

This REST value is defined in the project's WID file in the following manner:

```

[

```

```

    "name" : "Rest",
    "parameters" : [

        //Url - Mandatory resource location to be invoked.
        "Url" : new StringDataType(),

        //Method - Defaults to GET and is the HTTP method that will be
        executed.
        "Method" : new StringDataType(),

        //ConnectionTimeout - Defaults to 60 seconds for the connection
        timeout.
        "ConnectTimeout" : new StringDataType(),

        //ReadTimeout - Defaults to 60 seconds for the read timeout.
        "ReadTimeout" : new StringDataType(),

        //Username - The username for authentication that overrides the
        one given on handler initialization.
        "Username" : new StringDataType(),

        //Password - The password for authentication that overrides the
        one given on handler initialization.
        "Password" : new StringDataType()
    ],
    "results" : [
        "Result" : new ObjectDataType(),
    ],
    "displayName" : "REST",
    "icon" : "defaultserviceicon.png"
]

```

The configuration options displayed about must be given via the work item parameter. The authentication information can be given on handler initialization, but it can be overridden via the work item parameter.

3. Compile the project. The resulting JAR file should be placed in (***DEPLOY_DIR/business-central.war/WEB-INF/lib/***).
4. Restart the server.

Registering via **kmodule.xml**

An alternative to registering work item handlers in **CustomWorkItemHandlers.conf** is to configure them with **kmodule.xml**. This is beneficial in that it avoids a complete server restart.

1. Register the work item handler in the **Administration** menu path
PROJECT_NAME/src/main/resources/META-INF/kmodule.xml
2. Make sure the work item handler is given as a MVEL expression; for example, ***new org.jbpm.wih.CustomHandler()*** or FQCN expression: ***org.jbpm.wih.CustomHandler***.
3. Compile the project. Upload the work item handler JAR into Business Central via the Artifact Repository. Then add it as a dependency for the project where the user wants to use this handler.

4.14.1.4. Registering a Work Item handler

When executing processes in Business Central, **WorkItemHandlers** are registered in the **ksession** automatically. However, in order for them to be used in embedded mode, the **WorkItemManager** registers **WorkItemHandler** instances. Likewise, in the example below, the **NotificationWorkItemHandler** needs to be registered in order for it to be used with a process containing a **Notification** work item:

1. Register the work item handler like the following:

```
/* Create the drools name of the <task> and the custom work item
handler instance */
KieSession kieSession = kieBase.newKieSession();
ksession.getWorkItemManager().registerWorkItemHandler(
    "Notification",
    new NotificationWorkItemHandler()
);
```

2. Look at the BPMN2 syntax for the process. The previous registration example would appear as follows:

```
<?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" >

    ...
    <!-- The tns:taskName attribute in the <task> node is necessary for
    the WorkItemManager to be able to see which WorkItemHandler instance
    should be used with which task or work item. -->
    <task id="_5" name="Notification Task"
    tns:taskName="Notification" >

    ...
```



NOTE

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.

4.14.2. Service repository

The service repository feature allows you to import an already existing work item from a repository directly into your project. It allows multiple users to reuse generic work items, such as work items allowing integration with Twitter, performing file system operations, etc. Imported work items are

automatically added to your palette and ready to use.



IMPORTANT

A public service repository with various predefined work items is available at <http://docs.jboss.org/jbpm/v6.0/repository/>.



NOTE

Although you can import any of these work items, please note that in Red Hat JBoss BPM Suite, only the following work items are available by default (and supported): Log, Email, Rest, WS. You can still import the other work items, but they are not supported by Red Hat.

4.14.2.1. Importing from a service repository

To import a work item from a service repository, do the following:

1. Open your Process in the Web Process Designer.
2. In the editor menu, click the **Connect to a Service Repository**  button.
3. In the **Service Repository Connection** window, define the location of the repository on the location input line and click **Connect**.

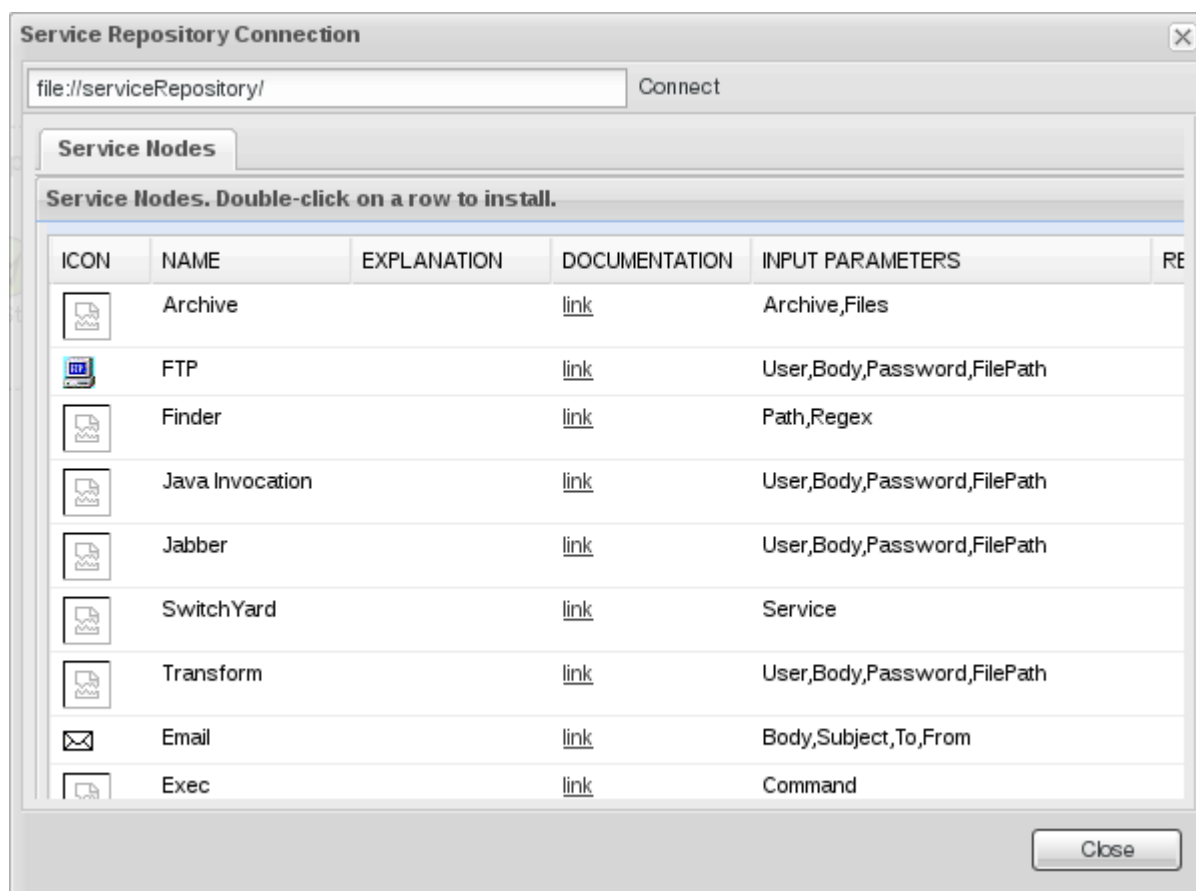
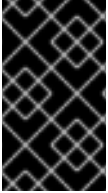


Figure 4.25. Establishing connection to a service repository

4. Double-click the asset to import it.



IMPORTANT

Every work item must be registered in the **`DEPLOY_DIRECTORY/business-central.war/WEB-INF/classes/META-INF/CustomWorkItemHandler.conf`** file. If a work item is not registered in the file, it will not be available for use.

4.14.2.2. Setting up a service repository

A service repository can be any repository, local or remote, with the **`index.conf`** file in its root directory.

Repository configuration file

The **`index.conf`** file must be located in the root directory of the service repository. It contains a list of any directory within the repository that are to be considered directories of the service repository.

Example 4.10. `index.conf`

```
Email
FileSystem
ESB
FTP
Google
Java
Jabber
Rest
RSS
Transform
Twitter
```

Each directory contains either another **`index.conf`** file so as to serve as a directory or work item resources. 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.

Work items and their resources

Directories with work items must contain:

- A **work item configuration file** is a file with the same name as the parent directory (for example, **`Twitter.conf`**) that contains details about the work item resources in the service repository. The file is an extension of the work item definition file (refer to [Section 4.14.1, “Work item definition”](#)). Note, that the configuration file must contain references to any dependencies the work item handler requires. Optionally, it can define the documentation property with a path to documentation and category which defines the category the custom work item is placed under in the repository.

Example 4.11. Work item configuration file

```
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.SampleCustomE
ditor",
    "icon" : "twitter.gif",
    "category" : "Communication",
    "defaultHandler" :
    "org.jbpm.process.workitem.twitter.TwitterHandler",
    "documentation" : "index.html",
    //Every work item definition should specify dependencies even if
it doesn't have one.
    "dependencies" : []
    [
        "file:./lib/jbpm-twitter.jar",
        "file:./lib/twitter4j-core-2.2.2.jar"
    ]
    ]
]

```

- All **resources referenced** in the work item configuration file: icon, documentation, and dependencies.

4.14.3. User Task calls

The User Task service exposes a Java API for managing the life cycle of its User Tasks via the **TaskClient** class. The API is intended for developers to allow direct managing of the lifecycle of User Tasks. End users are advised to use the Business Central web application for User Task management.

To manage user tasks via a public API use the methods of the `org.kie.api.task.TaskService` class. The methods of this interface take the following arguments:

- **taskId**: ID of the target Task instance usually extracted from the currently selected User Task in the user task list in the user interface
- **userId**: ID of the user that is executing the action called by the method; usually the ID of the user that is logged in

The following is a subset of methods provided by the **org.kie.api.task.TaskService** class:

```

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);

```

Example 4.12. Starting and completing a simple user task

```

import org.kie.api.runtime.KieSession;
import org.kie.api.runtime.manager.RuntimeEngine;
import org.kie.api.runtime.manager.RuntimeManager;

```

```

import org.kie.api.runtime.process.ProcessInstance;
import org.kie.api.task.TaskService;
import org.kie.api.task.model.TaskSummary;

....
KieSession ksession = runtimeEngine.getKieSession();
TaskService taskService = runtimeEngine.getTaskService();
ProcessInstance processInstance =
ksession.startProcess("com.sample.bpmn.hello");

// John is assigned a task and he completes it
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);

...

```

4.14.4. Actor assignment calls

User Tasks must define either the **ActorID** or the **GroupID** parameter, which define the users who can or should execute the User Tasks. It is in the Task List of these users the Task appears.

If the User Task element defines exactly one user, the User Task appears only in the Task List of that particular user. If a User Task is assigned to more than one user, that is, to multiple actors or to a group, it appears in the Task List of all the users and any of the users can claim and execute the User Task.

End users define these properties in the Process Designer. However, the provided actor and group IDs need to be registered with the User Task service before they can be used by User Tasks.

You can manage actors dynamically on the TaskService.

Example 4.13. Adding user Kris and group Developers on taskSession

```

EntityManagerFactory emf =
Persistence.createEntityManagerFactory("org.jbpm.task");
TaskService taskService = new TaskService(emf,
SystemEventListenerFactory.getSystemEventListener());
TaskServiceSession taskSession = taskService.createSession();

// registering new user and group:
taskSession.addUser(new User("Kris"));
taskSession.addGroup(new Group("Developers"));

```

Also, you can specify the groups a user is a member of, such as, the default **admin** roles as well as your custom roles.

Example 4.14. Requesting the list of tasks the user is a potential owner of

```
List<String> groups = new ArrayList<String>();

groups.add("sales");

taskClient.getTasksAssignedAsPotentialOwner("sales-rep", groups, "en-UK", taskSummaryHandler);
```



IMPORTANT

The Administrator can manipulate the life cycle of all Tasks, even if not being their potential owner. By default, a special user with `userId` **Administrator** is the administrator of each Task. It is therefore recommended to always define at least user Administrator when registering the list of valid users with the User Task service.

4.14.4.1. Connecting to custom directory information services

It is often necessary to establish connection and transfer data from existing systems and services, such as LDAP, to acquire data on actors and groups for User Tasks. This can be done by implementing the `UserGroupInfoProducer` interface which allows you to create your own implementation for user and group management, and then configuring it over CDI for Business Central. These are the steps required to implement and make this interface active:

1. Create an implementation of the `UserGroupInfoProducer` interface and provide your own custom callback (see [Section 4.14.5.1, "Connecting to LDAP"](#)) and user info implementations according to the needs from the producer.

This implementation must be annotated with the `@Selectable` qualifier for it to be found by Business Central. The listing below shows an example LDAP implementation:

```
import javax.enterprise.context.ApplicationScoped;
import javax.enterprise.inject.Alternative;
import javax.enterprise.inject.Produces;

import org.jbpm.services.task.identity.LDAPUserGroupCallbackImpl;
import org.jbpm.services.task.identity.LDAPUserInfoImpl;
import org.jbpm.shared.services.cdi.Selectable;
import org.kie.api.task.UserGroupCallback;
import org.kie.internal.task.api.UserInfo;

@ApplicationScoped
@Alternative
@Selectable
public class LDAPUserGroupInfoProducer implements
    UserGroupInfoProducer {

    private UserGroupCallback callback = new
    LDAPUserGroupCallbackImpl(true);
    private UserInfo userInfo = new LDAPUserInfoImpl(true);

    @Override
    @Produces
    public UserGroupCallback produceCallback() {
        return callback;
    }
}
```

```

    }

    @Override
    @Produces
    public UserInfo produceUserInfo() {
        return userInfo;
    }
}

```

2. Package your custom implementations (the **LDAPUserGroupInfoProducer**, the **LDAPUserGroupCallbackImpl** and the **LDAPUserInfoImpl** classes from the example above) into a bean archive (jar with META-INF/beans.xml so it can be found by CDI container). Add this jar file to **business-central.war/WEB-INF/lib**.
3. Modify **business-central.war/WEB-INF/beans.xml** and add the implementation (**LDAPUserGroupInfoProducer** from the example above) as an alternative to be used by Business Central.

```

<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>com.test.services.producer.LDAPUserGroupInfoProducer</class>
    </alternatives>
</beans>

```



WARNING

The use of a custom **UserGroupInfoProducer** requires internal APIs, which may be broken in future releases. Using a custom **UserGroupInfoProducer** is not recommended or supported by Red Hat.

4. Restart your server and your custom callback implementation should now be used by Business Central.

4.14.5. LDAP connection

A dedicated UserGroupCallback implementation for LDAP servers is provided with the product to allow the User Task service to retrieve information on users, and groups and roles directly from an LDAP service.

The LDAP UserGroupCallback implementation takes the following properties:

- **ldap.bind.user**: username used to connect to the LDAP server (optional if LDAP server accepts anonymous access)

- `ldap.bind.pwd`: password used to connect to the LDAP server (optional if LDAP server accepts anonymous access)
- `ldap.user.ctx`: context in LDAP with user information (mandatory)
- `ldap.role.ctx`: context in LDAP with group and role information (mandatory)
- `ldap.user.roles.ctx`: context in LDAP with user group and role membership information (optional; if not specified, `ldap.role.ctx` is used)
- `ldap.user.filter`: filter used to search for user information; usually contains substitution keys `{0}`, which are replaced with parameters (mandatory)
- `ldap.role.filter`: filter used to search for group and role information, usually contains substitution keys `{0}`, which are replaced with parameters (mandatory)
- `ldap.user.roles.filter`: filter used to search for user group and role membership information, usually contains substitution keys `{0}`, which are replaced with parameters (mandatory)
- `ldap.user.attr.id`: attribute name of the user ID in LDAP (optional; if not specified, `uid` is used)
- `ldap.roles.attr.id`: attribute name of the group and role ID in LDAP (optional; if not specified `cn` is used)
- `ldap.user.id.dn`: user ID in a DN, instructs the callback to query for user DN before searching for roles (optional, by default `false`)
- `java.naming.factory.initial`: initial context factory class name (by default `com.sun.jndi.ldap.LdapCtxFactory`)
- `java.naming.security.authentication`: authentication type (possible values are `none`, `simple`, `strong`; by default `simple`)
- `java.naming.security.protocol`: security protocol to be used; for instance `ssl`
- `java.naming.provider.url`: LDAP url (by default `ldap://localhost:389`; if the protocol is set to `ssl` then `ldap://localhost:636`)

4.14.5.1. Connecting to LDAP

To be able to use the LDAP `UserGroupCallback` implementation configure the respective LDAP properties (refer to [Section 4.14.5, “LDAP connection”](#)) in one of the following ways:

- **programmatically**: build a `Properties` object with the respective `LDAPUserGroupCallbackImpl` properties and create `LDAPUserGroupCallbackImpl` with the `Properties` object as its parameter.

Example 4.15.

```
import org.kie.api.PropertiesConfiguration;
import org.kie.api.task.UserGroupCallback;
...
Properties properties = new Properties();
properties.setProperty(LDAPUserGroupCallbackImpl.USER_CTX,
    "ou=People,dc=my-domain,dc=com");
properties.setProperty(LDAPUserGroupCallbackImpl.ROLE_CTX,
```



```

"ou=Roles,dc=my-domain,dc=com");
properties.setProperty(LDAPUserGroupCallbackImpl.USER_ROLES_CTX,
"ou=Roles,dc=my-domain,dc=com");
properties.setProperty(LDAPUserGroupCallbackImpl.USER_FILTER, "
(uid={0})");
properties.setProperty(LDAPUserGroupCallbackImpl.ROLE_FILTER, "
(cn={0})");
properties.setProperty(LDAPUserGroupCallbackImpl.USER_ROLES_FILTER
, "(member={0})");

UserGroupCallback ldapUserGroupCallback = new
LDAPUserGroupCallbackImpl(properties);

UserGroupCallbackManager.getInstance().setCallback(ldapUserGroupCa
llback);

```

- **declaratively**: create the **jbpms.usergroup.callback.properties** file in the root of your application or specify the file location as a system property: -
Djbpm.usergroup.callback.properties=FILE_LOCATION_ON_CLASSPATH

Make sure to register the LDAP callback when starting the User Task server.

```

#ldap.bind.user=
#ldap.bind.pwd=
ldap.user.ctx=ou\=People,dc\=my-domain,dc\=com
ldap.role.ctx=ou\=Roles,dc\=my-domain,dc\=com
ldap.user.roles.ctx=ou\=Roles,dc\=my-domain,dc\=com
ldap.user.filter=(uid\={0})
ldap.role.filter=(cn\={0})
ldap.user.roles.filter=(member\={0})
#ldap.user.attr.id=
#ldap.roles.attr.id=

```

4.15. EXCEPTION MANAGEMENT

When an unexpected event, that deviates from the normative behavior, occurs in a Process instance, it is referred to as an exception. There are two types of exceptions: business exceptions and technical exceptions.

Business exceptions

Business exceptions relate to the possible incorrect scenarios of the particular Process, for example, trying to debit an empty bank account. Handling of such exceptions is designed directly in the Process model using BPMN Process elements.

When modeling business exception management, the following mechanisms are to be used:

Errors

An Error is a signal that an unexpected situation occurred (refer to [Section A.4.1, “Errors”](#)). The mechanism can be used immediately when the problem arises and does not allow for any compensation.

Compensation

Compensation is equivalent to the Error mechanism; however, it can be used only on Sub-Processes when it is required that the execution flow continues after the compensation using the "regular" outgoing Flow (execution continues after the compensation as if no compensation occurred).

Canceling

Canceling is equivalent to the Error mechanism; however, it can be used only on Sub-Processes and it is required that the Sub-Process takes the flow leaving the respective Cancel Intermediate Event so that the "normal" execution flow is never taken as opposed to compensation.

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 Java Exception being thrown by the system. Technical components used in a Process fail in a way that can not be described using BPMN (for further information, refer to [Section 5.1.5, "Technical exceptions"](#)).

CHAPTER 5. ADVANCED PROCESS MODELING

5.1. PROCESS MODELING OPTIONS

You can create Processes in multiple ways:

Using one of the graphical editors

You can use two delivered graphical editors. Process Designer is available through Business Central and Eclipse Process Designer. See *Red Hat JBoss BPM Suite User Guide* for more information on how to use the editors.

Using an XML editor

You can use any XML or text editor to create a process specification using the BPMN2 XML schema.

Using the Process API

You can use the JBoss BPM Suite **core** API directly. The most important process model elements are defined in the packages `org.jbpm.workflow.core` and `org.jbpm.workflow.core.node`.

5.1.1. Process modeling using XML

The BPMN2 file must meet the BPMN2 schema. The file content comprises the following parts:

XML prolog

The XML prolog consists of the XML declaration and DTD declaration.

The process element

The **process** element defines process attributes and contains definitions of the process elements (nodes and connections).

BPMN diagram definition

The **BPMNDiagram** element contains definitions for visualization of the Process elements in the Process Diagram.

Example 5.1. BPMN2 example file

```
<?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>

```

5.1.2. Process modeling using API

To be able to execute processes from within your application, you need to perform the following in your code:

1. Create a Runtime Manager in your Execution Server.
 - Singleton: allows sequential execution of multiple instances in the one session
 - PerProcessInstance: allows you to create multiple process instances; every instance is created within its own session.

- **PerRequestSession**: every external interaction with a process instances causes that the process session finishes and the process instance is re-created in a new session.
2. Get a runtime context and create a session in it.
 3. Start a Process from the underlying Knowledge Base.
 4. Close the Runtime Manager.

Example 5.2. Process instantiation in a session of Per Process Instance Runtime Manager

```
import org.kie.api.runtime.manager.RuntimeManager;
import org.kie.api.runtime.manager.RuntimeManagerFactory.Factory;
import org.kie.api.runtime.manager.RuntimeEngine;
import org.kie.api.runtime.KieSession;
...
RuntimeManager manager =
    RuntimeManagerFactory.Factory.get()
        .newPerProcessInstanceRuntimeManager(environment);

RuntimeEngine runtime =
    manager.getRuntimeEngine(
        ProcessInstanceIdContext.get());

KieSession ksession = runtime.getKieSession();
// do something here, e.g.
ksession.startProcess("org.jbpm.hello");

manager.disposeRuntimeEngine(engine);
manager.close();
```

5.1.3. Process update

5.1.3.1. Process update

When updating a Process definition, the new Process definition must define an increased version number and an update policy: the update policy defines how to handle the running Process instances of the older Process definition. You can decide to apply on them one of the following strategies:

- **Abort**: any running Process instances are aborted. If necessary, you can have the Process instance restarted using the new Process definition.
- **Transfer**: any running Process instances are migrated to the new process definition: once the instance has been migrated successfully, it will continue its execution based on the updated process logic. For further information refer to [Section 5.1.3.3, “Migrating a Process instance”](#).

Note that the older version of the Process definition remains in the repository as well as in the respective sessions. Therefore, the new process should have a different ID, though the name can remain the same, and you can use the version parameter to show when a Process is updated (the version parameter is just a String and is not validated).

Example 5.3. Process abort update

```

import org.kie.api.KieBase;
import org.kie.api.KieServices;
import org.kie.api.runtime.KieSessionConfiguration;

// build kbase with the replace-version-1.bpmn process
KieBase kbase =
KieServices.Factory.get().newKieSessionConfiguration();
    kbase.addKnowledgePackages(getProcessPackages("replace-version-
1.bpmn"));

    KieSession ksession = kbase.newStatefulKnowledgeSession();
    try {
        // start a replace-version-1.bpmn process instance
        ksession.startProcess("com.sample.process", Collections.
<String, Object>singletonMap("name", "process1"));

        // add the replace-version-2.bpmn process and start its
instance
        kbase.addKnowledgePackages(getProcessPackages("replace-
version-2.bpmn"));
        ksession.startProcess("com.sample.process", Collections.
<String, Object>singletonMap("name", "process2"));

        // signal all processes in the session to continue (both
instances finish)
        ksession.signalEvent("continue", null);
    } finally {
        ksession.dispose();
    }

```

5.1.3.2. Process instance migration

Every Process instance contains complete runtime information that are relevant for the Process instance so that it can continue its execution after interruption. This information includes all data linked to the Process instance, such as variables, the current state in the Process diagram, information on the instance of the Element that is active.

A Process instance does not contain information that is not runtime relevant; The runtime data and state are linked to a particular Process using ID references, that represent the process logic that needs to be followed when executing the Process instance. This separation of Process definition and runtime state allows reuse of the Process definition across multiple Process instances and minimizes the size of the runtime state. Consequently, updating a running Process instance to an updated Process definition is a matter of changing the referenced Process ID to the new ID.

However, this does not take into account that the state of the Process instance (the variable instances and the node instances) might need to be migrated as well. In cases where the Process is only extended and all existing wait states are kept, this is pretty straightforward, the runtime state of the process instance does not need to change at all. However, it is also possible that a more sophisticated mapping is necessary. For example, when an existing wait state is removed, or split into multiple wait states, an existing process instance that is waiting in that state cannot simply be updated. Or when a new process variable is introduced, that variable might need to be initiated correctly so it can be used in the remainder of the (updated) process.

5.1.3.3. Migrating a Process instance

The `WorkflowProcessInstanceUpgrader` can be used to upgrade a workflow process instance to a newer process instance. Of course, you need to provide the process instance and the new process id. By default, jBPM will automatically map old node instances to new node instances with the same id. But you can provide a mapping of the old (unique) node id to the new node id. The unique node id is the node id, preceded by the node ids of its parents (with a colon in between), to uniquely identify a node when composite nodes are used (as a node id is only unique within its node container. The new node id is simply the new node id in the node container (so no unique node id here, simply the new node id). The following code snippet shows a simple example.

Example 5.4. Process transfer with custom active Element mapping

```
import org.kie.api.KieBase;
import org.kie.api.runtime.KieSession;
import org.kie.api.runtime.process.ProcessInstance;
import org.kie.api.runtime.process.WorkflowProcessInstance;

// build kbase with the replace-version-1.bpmn process
KieBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
kbase.addKnowledgePackages(getProcessPackages("replace-version-1.bpmn"));

KieSession ksession = kbase.newStatefulKnowledgeSession();
try {
    // start two instances of the replace-version-1.bpmn
    process
        ProcessInstance pi =
ksession.startProcess("com.sample.process", Collections.<String,
Object>singletonMap("name", "process1"));
        ProcessInstance pi2 =
ksession.startProcess("com.sample.process", Collections.<String,
Object>singletonMap("name", "process2"));

        // add the replace-version-3.bpmn process to the kbase and
        start its instance
        kbase.addKnowledgePackages(getProcessPackages("replace-
version-3.bpmn"));
        ksession.startProcess("com.sample.process2", Collections.
<String, Object>singletonMap("name", "process3"));

        // upgrade: active nodes from the replace-version-1.bpmn
        process are mapped to the same nodes in the process

WorkflowProcessInstanceUpgrader.upgradeProcessInstance(ksession,
pi.getId(), "com.sample.process2", null);

        // upgrade the process using custom mapping
        Map<String, Long> mapping = new HashMap<String, Long>();
        mapping.put("3", 8L);
        mapping.put("2", 7L);

WorkflowProcessInstanceUpgrader.upgradeProcessInstance(ksession,
pi2.getId(), "com.sample.process2", mapping);
        ksession.fireAllRules();

        // signal all processes they may continue (all of them
finish)
```

```

        ksession.signalEvent("continue", null);
    } finally {
        ksession.dispose();
    }
}

```

If this kind of mapping is still insufficient, you can still describe your own custom mappers for specific situations. Be sure to first disconnect the process instance, change the state accordingly and then reconnect the process instance, similar to how the `WorkflowProcessinstanceUpgrader` does it.

5.1.4. Multi-threading

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 splits in two processes that are executed in a parallel fashion.

The Process engine supports logical multi-threading; for example, in Processes that include a parallel Gateway. The logical multi-threading is implemented using one technical thread: A Process that includes logical multi-threading is executed in one technical thread. Avoiding technical multi-threading prevents further implementation complexity as multiple technical threads of one Process instance need to communicate their state information to each other. While it might seem that technical multi-threading would bring significant performance benefits, the extra logic needed to make sure the threads can work together well may cancel out these benefits.

In general, the execution engine also executes actions in serial. For example, when the engine encounters a Script Task, it synchronously executes the script and waits for it to complete before continuing execution. Similarly, when a Process encounters a Parallel Gateway, it sequentially triggers each of the outgoing Flows. This is possible since execution is almost always instantaneous. Similarly, action scripts in a Process are also executed synchronously, and the engine waits for them to finish before continuing execution. That means, that if the execution needs to wait, for example, a `Thread.sleep()` call is being executed, the engine does not continue any execution — it remains blocked during the wait period.

5.1.4.1. Asynchronous execution

If a work item execution of a Task does not execute instantaneously, but needs to wait, for example, to receive a response from an external system, the service handler must handle your service asynchronously. The asynchronous handler only invokes the service and notifies the engine once the results are available. In the mean time, the process engine continues the execution of the process.

A typical example of a service that requires asynchronous invocation is a Human Task: The engine is not to wait until a human actor responds to the request but continue and process the result of the Task when it becomes available. The human task handler creates a new task on the task list of the assigned actor and the engine is then allowed to continue the execution: The handler notifies the engine asynchronously when the user completes the task.

To implement an asynchronous service handler, implement the actual service in a new thread using the `executeWorkItem()` method in the work item handler that allows the Process instance to continue its execution.

Example 5.5. Example of asynchronous service handling in Java

```
import org.kie.api.runtime.process.WorkItem;
```



```

import org.kie.api.runtime.process.WorkItemHandler;
import org.kie.api.runtime.process.WorkItemManager;

public class MyServiceTaskHandler implements WorkItemHandler {

    public void executeWorkItem(WorkItem workItem, WorkItemManager
manager) {

        new Thread(new Runnable() {

            public void run() {

                // The main thread with the parent element execution
                }
            }).start();
        }

        public void abortWorkItem(WorkItem workItem, WorkItemManager manager)
        {

        }

    }
}

```

It is recommended to have your handler contact a service that executes the business operation, instead of performing the task, as failure of the business operation will not affect your process. This approach also provides greater flexibility when developing and reusing services.

For example, your human task handler can invoke the human task service to add a task to the service. To implement an asynchronous handler, you usually have to do an asynchronous invocation of the handler. This usually depends on the technology you use to do the communication and might be an asynchronous invocation of a web service, or sending a JMS message to the external service.

The Red Hat JBoss BPM Suite Job Executor

Red Hat JBoss BPM Suite, from 6.1 version, implements a special component, the Job Executor, to accommodate asynchronous execution. The Executor provides an environment independent from the process runtime environment for instantiation of asynchronous executions defined in a **Command** object.

In particular, the JBoss BPM Suite Job Executor provides advanced handling of common asynchronous execution operations, like error handling, retry, cancellation and history logging. You can continue to delegate work to a separate thread, as described above, in a custom implementation of **WorkItemHandler** and use the JBoss BPM Suite Job Executor to handle these operations for you.

The JBoss BPM Suite Job Executor works on instances of the Command interface. This interface implements a single method **execute()** that accepts the **CommandContext** data transfer object with serializable data. The Command instance should only contain the business logic to be executed and have no reference to the underlying process engine or runtime related data. At the minimum, the **CommandContext** should provide the **businessKey** - the unique identifier of the caller and **callback** - the fully qualified classname (FQCN) of the CommandCallback instance to be called on command completion. Of course, when executed as part of a process (**WorkItemHandler**), you will need to provide additional data like the **workItem**, **processInstanceId** and **deploymentId**.

The result for the execution is returned via an instance of the **ExecutionResults** class. As with the input data, the data provided by this class must also be serializable.

Business Central provides a way for you to view the current jobs within the system; to schedule them, cancel them, or view their history. Go to **Deploy** → **Jobs** to access this screen. You can even create a new job by clicking on the **New Job** button.

The Asynchronous WorkItemHandler

Red Hat JBoss BPM Suite 6.1 provides an out of the box asynchronous **WorkItemHandler** that is backed by the JBoss BPM Suite Job Executor. All the features that the JBoss BPM Suite Executor delivers are available for background execution within a process instance. There are two ways to configure this **AsyncWorkItemHandler** class:

1. As a generic handler where you provide the command name as part of the work item parameters. In Business Central while modeling a process, if you need to execute some work item asynchronously: specify **async** as the value for the TaskName property, create a data input called **CommandClass** and assign the FQCN of this **CommandClass** as the data input.
2. As a specific handler which is created to handle a given type of work item, thus allowing you to register different instances of **AsyncWorkItemHandler** for different work items. Commands are most likely to be dedicated to a particular work item, which allows you to specify the **CommandClass** at registration time instead of requiring it at design time, as with the first approach. But this means that an additional CDI bean that implements **WorkItemHandlerProducer** interface needs to be provided and placed on the application classpath so that the CDI container can find it. When you are ready to model your process, set the value of the TaskName property to the one provided at registration time.

Configuring the JBoss BPM Suite Executor

The JBoss BPM Suite executor can be configured to allow fine tuning of its environment via the following system properties:

1. `org.kie.executor.disabled`: true OR false - enable/disable the executor.
2. `org.kie.executor.pool.size`: Integer. Specify the thread pool size for the executor, which by default is 1.
3. `org.kie.executor.retry.count`: Integer. Specifies the default number of retries in case of an error executing a job. The default value is 3.
4. `org.kie.executor.interval`: Integer. Specifies the time to wait between checking for waiting jobs. The default value is 3 seconds.
5. `org.kie.executor.timeunit`: NANOSECONDS OR MICROSECONDS OR MILLISECONDS OR SECONDS OR MINUTES OR HOURS OR DAYS. Specifies the unit for the interval property. The default is SECONDS.

5.1.4.2. Multiple Sessions and persistence

The simplest way to run multiple Process instances is to run them in one knowledge session. However, it is possible to run multiple Process instances in different knowledge sessions or in different technical threads.

When using multiple knowledge session with multiple processes and adding persistence, use a database that allows row-level as well as table-level locks: There could be a situation when there are 2 or more threads running, each within its own knowledge session instance. On each thread, a Process is being started using the local knowledge session instance. In this use case, a race condition exists in which both thread A and thread B have coincidentally simultaneously finished a Process instance. At this point, both thread A and B are 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.
- Thread B has the opposite situation: It has a lock on the `SessionInfo` table, having just committed a change.
- 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 are not be able to solve, unless row-level locks are possible and enabled in the database and tables used.

5.1.5. Technical exceptions

Technical exceptions occur when a technical component of a Process acts in an unexpected way. When using Java-based systems, this often results in a Java Exception. As these exceptions cannot be handled using BPMN2, it is important to handle them in expected ways.

The following types of code might throw exceptions:

- Code present directly in the process definition
- Code that is not part of the product executed during a Process
- Code that interacts with a technical component outside of the Process Engine

This includes the following:

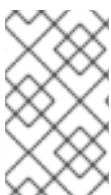
- Code in Element properties, such as the Script property of a **Script Task** element or in the definitions of the interception actions, that is, the **onEntry** and **onExit** properties
- Code in **WorkItemHandlers** associated with **task** and task-type nodes

Code in Element properties

Exceptions thrown by code defined in Element properties can cause the Process instance to fail in an unrecoverable way. Often, it is the code that starts the Process that will end up throwing the exception generated by a Process without returning a reference to the Process instance. Such code includes for example the **onEntry** and **onExit** properties, Script defined for the Script Task, etc.

Therefore, it is important to limit the scope of the code in these Elements so that it operates only over 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 instance.

To interact with other systems, use **task** Elements, **serviceTask** Elements and other **task-type** Elements. Do not use the **scriptTask** nodes for these purposes.



NOTE

If the script defined in a **scriptTask** causes the problem, the Process Engine usually throws the **WorkflowRuntimeException** with information on the Process (refer to [Section 5.1.5.1.5, “Extracting information from WorkflowRuntimeException”](#)).

Code in WorkItemHandlers

WorkItemHandlers are used when your Process interacts with other technical systems (for more information on WorkItemHandlers refer to [Section 4.14.1, “Work item definition”](#)).

You can either build exception handling into your own WorkItemHandler implementations or wrap your implementation into the **handler decorator** classes (for examples and detailed information refer to [Section 5.1.5.1.2, “Exception handling classes”](#)). These classes include the logic that is executed when an exception is thrown during the execution or abortion of a work item:

SignallingTaskHandlerDecorator

catches the exception and signals it to the Process instance using a configurable event type when the **executeWorkItem()** or **abortWorkItem** methods of the original **WorkItemHandler** instance throw an exception. The exception thrown is passed as part of the event. This functionality can be also used to signal to an Event SubProcess defined in the Process definition.

LoggingTaskHandlerDecorator

logs error about any exceptions thrown by the **executeWorkItem()** and **abortWorkItem()** methods. It also saves any exceptions thrown to an internal list so that they can be retrieved later for inspection or further logging. The content and format of the message logged are configurable.

While the classes described above covers most cases involving exception handling as it catches any throwable objects, you might still want to write a custom WorkItemHandler that includes exception handling logic. In such a case, consider the following:

- Does the implementation catch all exceptions the code could return?
- Does the implementation complete or abort the work item after an exception has been caught or uses a mechanisms to retry the process later (in some cases, incomplete process instances might be acceptable)?
- Does the implementation define any other actions that need to be taken when an exception is caught? Would it be beneficial to interact with other technical systems? Should a Sub-Process be triggered to handle the exception?



IMPORTANT

If WorkItemManager to signals that the work item has been completed or aborted, make sure the signal is sent after any signals to the Process instance were sent. Depending on how your Process definition, calling `WorkItemManager.completeWorkItem()` or `WorkItemManager.abortWorkItem()` triggers the completion of the Process instance as these methods trigger further execution of the Process execution flow.

5.1.5.1. Technical exception examples

5.1.5.1.1. Service Task handlers

The example involves a Throwing Error Intermediate Event caught by an Error Event Sub-Process.

When the Throwing Error Intermediate Event throws the Error, the Process instance is interrupted:

1. Execution of the Process instance stops: no other parts of the Process are executed.
2. The Process instance finishes as ABORTED.

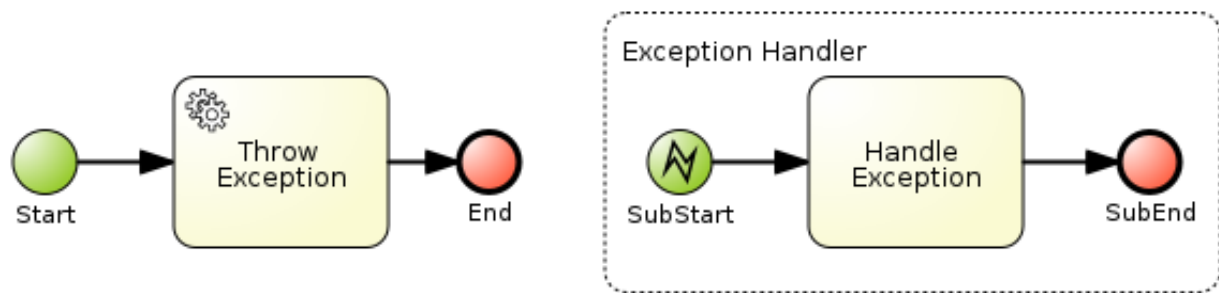


Figure 5.1. Process with an exception handling Event Sub-Process

Parts of the BPMN2 definition of the example Process relevant for exception handling

```

<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.process.WorkItem"/>
4

<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	The itemDefinition element defines a data structure used in the serviceInputItem property of the Process.
2	The message element (1st reference) defines a <i>message</i> that contains the String defined by the itemDefinition element on the line above. The interface element below then refers to the itemDefinition element (2nd reference) in order to define what type of content the service (defined by the interface) expects.
3	The error element (1st reference) defines an error that is used to trigger the <i>Event SubProcess</i> of the Process. The content of the error is defined by the itemDefinition element defined below the error element.

- 4 This **itemDefinition** element (1st 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 element itself, a **property** element (4th reference) that contains the initial **itemDefinition**. This allows the *Event SubProcess* to store the *error* it receives in that property (5th reference).

5.1.5.1.2. Exception handling classes

The **serviceTask** tasks use the `org.jbpm.bpmn2.handler.ServiceTaskHandler` class as its task handler class unless the **serviceTask** defines a custom **WorkItemHandler** implementation.

To catch and handle any technical exceptions a **WorkItemHandler** of a task might throw, wrap or decorate the handler class with a **SignallingTaskHandlerDecorator** instance.

IMPORTANT

When sending a signal of an event to the Process Engine, consider the rules for signaling process events:

- Error events are signaled by sending an **Error-errorCode attribute value** value to the session.
- Signal events are signaled by sending the name of the signal to the session.
- If you wanted to send an error event to a Boundary Catch Error Event, the error type should be of the format: **"Error-" + \$AttachedNodeID + "-" + \$ERROR_CODE**. For example, **Error-SubProcess_1-888** would be a valid error type.

However, this is *NOT* a recommended practice because sending the signal this way bypasses parts of the boundary error event functionality and it relies on internal implementation details that might be changed in the future. For a way to programmatically trigger a boundary error event when an Exception is thrown in **WorkItemHandler** see this KnowledgeBase [article](#).

Example 5.6. Using SignallingTaskHandlerDecorator

The **ServiceTaskHandler** calls the `ExceptionService.throwException()` method to throw an exception (refer to the `_handlingServiceInterface` interface element in the BPMN2).

The **SignallingTaskHandlerDecorator** that wraps the **ServiceTaskHandler** sends to the Process instance the **error** with the set **error code**.

```
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

        1
        = new SignallingTaskHandlerDecorator(ServiceTaskHandler.class,
            eventType);
        signallingTaskWrapper.setWorkItemExceptionParameterName(ExceptionService

        1
        .exceptionParameterName());
        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("ProcessWithExceptionHandlingError", params);
        return processInstance;
    }

    private static KieSession createKieSession() {
        KnowledgeBuilder kbuilder =
            KnowledgeBuilderFactory.newKnowledgeBuilder();
        kbuilder.add(ResourceFactory.newClassPathResource("exceptions/ExceptionH
            andlingWithError.bpmn2"), ResourceType.BPMN2);
        KieBase kbase = kbuilder.newKnowledgeBase();
        return kbase.newKieSession();
    }
}

```



Definition of the **Error-code** event to be sent to the process instance when the wrapped **WorkItemHandler** implementation throws an exception

- | | |
|---|---|
| ❶ | <p>Construction of the SignallingTaskHandlerDecorator class instance with the WorkItemHandler implementation and eventType as parameters</p> <p>Note that a SignallingTaskHandlerDecorator class constructor that takes an <i>instance</i> of a WorkItemHandler implementation as its parameter is also available. This constructor is useful if the WorkItemHandler implementation does not allow a no-argument constructor.</p> |
| ❷ | <p>Registering the WorltemHandler with the session</p> <p>When an exception is thrown by the wrapped WorkItemHandler, the SignallingTaskHandlerDecorator saves it as a parameter in the WorkItem instance with a parameter name configured in the SignallingTaskHandlerDecorator (see the code below for the ExceptionService).</p> |

5.1.5.1.3. Exception service

In [Section 5.1.5.1.1, “Service Task handlers”](#) the BPMN2 process definition defines the exception service using the **ExceptionService** class as follows:

```
<interface id="_handlingServiceInterface"
name="org.jbpm.examples.exceptions.service.ExceptionService">
<operation id="_handlingServiceOperation" name="handleException">
```

The exception service uses the **ExceptionService** class to provide the exception handling abilities. The class is implemented as follows:

```
import org.kie.api.runtime.process.WorkItem;
...
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;
}
}
```

You can specify any Java class with the default or another no-argument constructor as the class to provide the exception service so that it is executed as part of a **serviceTask**.

5.1.5.1.4. Handling errors with Signals

In the example in [Section 5.1.5.1.1, “Service Task handlers”](#), an **Error event** occurs during Process execution and the execution is interrupted immediately: no other Flows or Activities are executed.

However, you might want to complete the execution. In such case you can use a **Signal event** as the Process execution continues after the Signal is processed (that is, after the *Signal Event SubProcess* or another Activities that the Signal triggered, finish their execution). Also, the Process execution finished successfully, *not* in an aborted state, which is the case if an Error is used.

In the example process, we define the **error** element which is then used to throw the Error:

```
<error id="_exception" errorCode="code" structureRef="_exceptionItem"/>
```

To use a Signal instead, do the following:

1. Remove the line defining the **error** element and define a **<signal>** element:

```
<signal id="exception-signal" structureRef="_exceptionItem"/>
```

2. Make sure to change all references from the **"_exception" <error>** to the **"exception-signal" <signal>**.

Change the **<errorEventDefinition>** element in the **<startEvent>**,

```
<errorEventDefinition id="_X-1_ED_1" errorRef="_exception" />
```

to a **<signalEventDefinition>**:

```
<signalEventDefinition id="_X-1_ED_1" signalRef="exception-signal"/>
```

5.1.5.1.5. Extracting information from WorkflowRuntimeException

If a scripts in your Process definition may throw or threw an exception, you need to retrieve more information about the exception and related information.

If it is a **scriptTask** element that causes an exception, you can extract the information from the **WorkflowRuntimeException** as it is the wrapper of the scriptTask.

The **WorkflowRuntimeException** instance stores the information outlined in [Table 5.1, “Information in WorkflowRuntimeException instances”](#). Values of all fields listed can be obtained using the standard **get*** methods.

Table 5.1. Information in WorkflowRuntimeException instances

Field name	Type	Description
------------	------	-------------

Field name	Type	Description
processInstanceId	long	<p>The id of the ProcessInstance instance in which the exception occurred</p> <p>Note that the ProcessInstance may not exist anymore or be available in the database if using persistence.</p>
processId	String	<p>The id of the process definition that was used to start the process (that is, "ExceptionScriptTask" in</p> <pre>ksession.startProcesses("ExceptionScriptTask");</pre>
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 (<i>experimental</i>)
message	String	The short message with information on the exception
cause	Throwable	The original exception that was thrown

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", params);
        } 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/ScriptTaskEx
            ception.bpmn2"), ResourceType.BPMN2);
        KieBase kbase = kbuilder.newKnowledgeBase();
        return kbase.newKieSession();
    }
}

```

5.2. WORKFLOW PATTERNS

Workflow patterns are predefined blocks of Process elements that allow you to reuse once defined combination of Process elements: they include multiple nodes that are connected and form a common executable pattern that can be reused in a Process model.

Workflow patterns are available in the Shape Repository stencil set and can be drag-and-dropped on the canvas just like any other elements. To attach a pattern to an element on the canvas, select the element and then drag-and-drop the pattern from the palette onto the canvas. The pattern will be automatically connected to the element.

Multiple predefined workflow patterns are provided by default and you can define your own workflow patterns as necessary. The definitions are defined as JSON objects in the **`$JBoss_HOME/standalone/deployments/business-central.war/org.kie.workbench.KIEWebapp/defaults/patterns.json`** file.

5.2.1. Defining workflow patterns

To define custom workflow patterns, do the following:

1. In the stencil set of the Process Designer, locate the workflow pattern that resembles most to and that will use as base for your workflow pattern.
2. Open the **`$JBOSS_HOME/standalone/deployments/business-central.war/org.kie.workbench.KIEWebapp/defaults /patterns.json`** file in a text editor.
3. Locate the JSON object with the description property set to the base workflow pattern name (for example, **`"description" : "Sequence Pattern"`**).
4. Copy the JSON object and modify its elements as needed. Note that all the JSON objects are nested in a pair of square brackets and are comma separated.

CHAPTER 6. SOCIAL EVENTS

In Red Hat JBoss BPM Suite 6.1, users can follow other users and gain an insight into what activities are being performed by that user. They can also listen for and follow timelines of regular events. This capability comes via the implementation of a Social Activities framework. This framework ensures that event notifications are generated by different activities within the system and that these notifications are broadcast for registered actors to view.

Multiple activities trigger events. These include: new repository creation, adding and updating resources and adding and updating processes. With the right credentials, a user can view these notifications once they are logged into Business Central.

FOLLOW USER

To follow a user, search for the user by entering his name in the search box in the *People* perspective. You get to this perspective by navigating to it from **Home** → **People**.

You must know the login name of the other user that you want to follow. As you enter the name in the search box, the system will try and auto-complete the name for you and display matches based on your partial entry. Select the user that you want to follow from these matches and the perspective will update to display more details about this user.

You can choose to follow the user by clicking on the **Follow** button. The perspective refreshes to showcase the user details and their recent activities.

ACTIVITY TIMELINE

Click on **Home** → **Timeline** to see a list of recent assets that have been modified (in the left hand window) and a list of changes made in the selected repository in the right hand side. You can click on the assets to directly open the editor for the assets (if you have the right permissions).

PART II. SIMULATION AND TESTING

CHAPTER 7. PROCESS SIMULATION

Process simulation allows users to simulate a business process based on the simulation parameters and get a statistical analysis of the process models over time in form of graphs. This helps to optimize pre and post execution of a process, minimizing the risk of change in business processes, performance forecast, and promote improvements in performance, quality and resource utilization of a process.

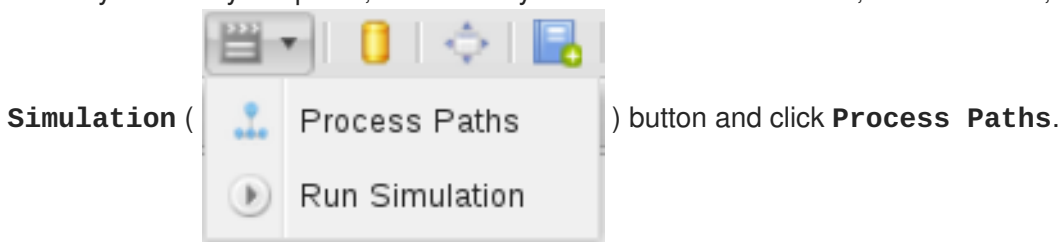
The simulation process runs in the Simulation engine extension, which relies on the possible execution paths rather than Process data. On simulation, the engine generates events for every simulated activity, which are stored in the simulation repository.

Simulation input data include general data about the Process simulation as well as simulation data for individual Process Elements. Process Elements executed by the engine automatically do not require any input data; however, the Process itself, Human Tasks, Intermediate Event, and Flows leaving a split Gateway, need such data: further information on Simulation data is available in [Section C.1, “Process”](#) and the subsequent sections.

7.1. PATH FINDER

Path Finder is a tool that allows you to identify all possible paths a Process execution can take.

Before you identify the paths, make sure your Process is valid. Then, on the toolbar, click the **Process**



NOTE

Note that when you click this button only core process paths are searched for. In order to view Embedded or Event subprocess paths, you have to click on the subprocess, making sure that it is selected and then click the **Process Path** button. This will focus on paths that are specific to this subprocess.

A dialog with data on individual path appears: to visualize any of the identified paths, select the path in the dialog and click **Show Path**.

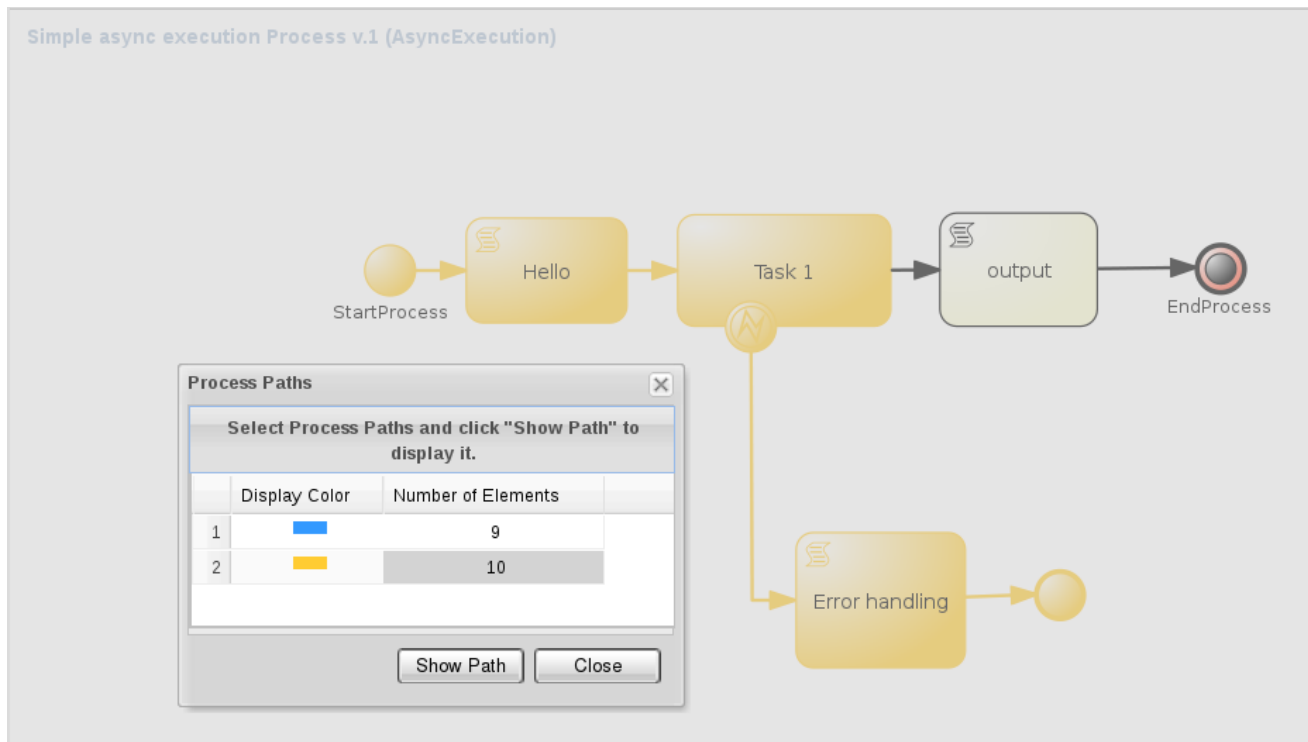


Figure 7.1. Process Paths

7.2. SIMULATING A PROCESS

7.2.1. Defining Simulation details on Elements

To define the input data for a Process simulation, you need to define the Simulation data on the Process and its Elements in the **Properties** tab.

Information on Simulation data for individual Process Elements and the Process itself are available in [Section C.1, "Process"](#) and subsequent sections.

7.2.2. Running a Simulation

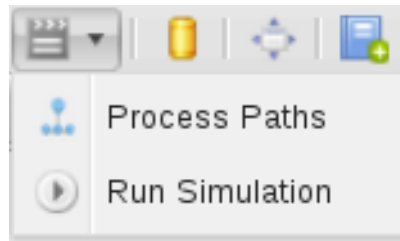
To run a Process Simulation, do the following:

1. Open the Process in the Process Designer and make sure you have defined the simulation parameters for individual Elements.

Simulation Properties	
Cost per time unit	20
Distribution Type	normal
Processing time (me...	10
Staff availability	20
Standard Deviation	1
Working Hours	8.0

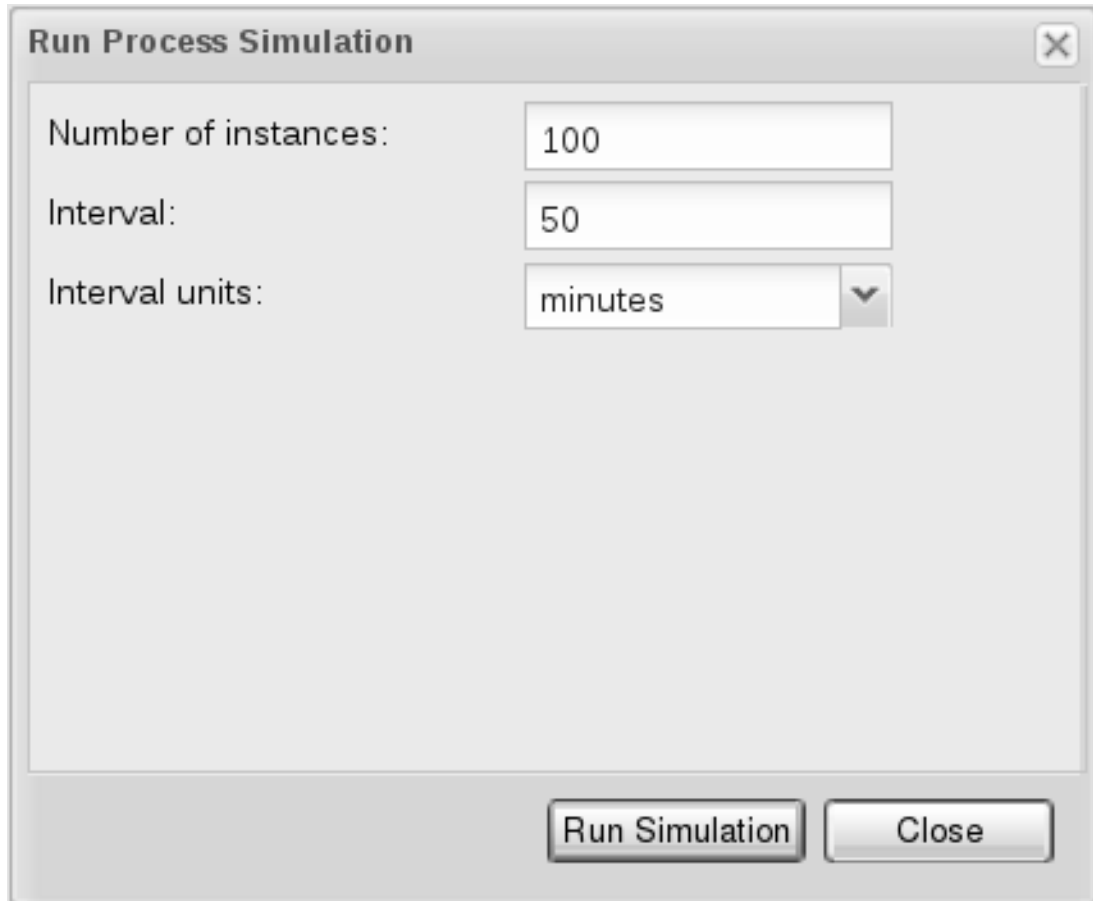
Figure 7.2. Simulation Properties

2. Click the **Process Simulation** () icon in the toolbar and



then click the Run Simulation entry.

3. In the **Run Process Simulation** dialog window, define the simulation session details:



The dialog window titled "Run Process Simulation" contains the following fields and controls:

- Number of instances:** A text input field containing the value "100".
- Interval:** A text input field containing the value "50".
- Interval units:** A dropdown menu with "minutes" selected.
- Buttons:** "Run Simulation" and "Close" buttons at the bottom right.

Figure 7.3. Run Process Simulation properties dialog window

- Number of instance: number of Process instances to be created and triggered
- Interval: interval between individual Process instantiations
- Interval units: time of unit the Interval is defined in

4. Click **Run Simulation**.

After you start the simulation, the Process Designer focuses the **Simulation Results** tab with the simulation Process results displayed in the **Simulation Graph** pane on the right.

7.2.3. Examining Simulation results

After you run a Process simulation, the Process Designer focuses the **Simulation Results** tab. The tab show the list of available simulation result in the **Simulation Graphs** on the right.

The results are divided into three sections with graphs:

- The Process section with general Process simulation graphs
- The Activities section with individual Activities' simulation graphs

Activities graphs for Human Tasks include **Execution Time** with the Max, Min, and Average execution time for the given Activity, **Resource Utilization** for the hours a resource has been used, and the **Cost Parameters** graph if applicable (if you defined the Cost parameter for the Activity). For Script Tasks only the **Execution Time** with the Max, Min, and Average execution time, is available.

- The Paths section with simulation graph of the Paths taken during the simulation.

The graphs contain the Process model with the respective Path highlighted and execution statistics on the Path.

Graph types

Click a graph entry to display the graph in the canvas area: the graph is visualized as a vertical bar chart; however, you can change the graph visualization type by clicking on the respective icon in the upper right corner of the canvas area.

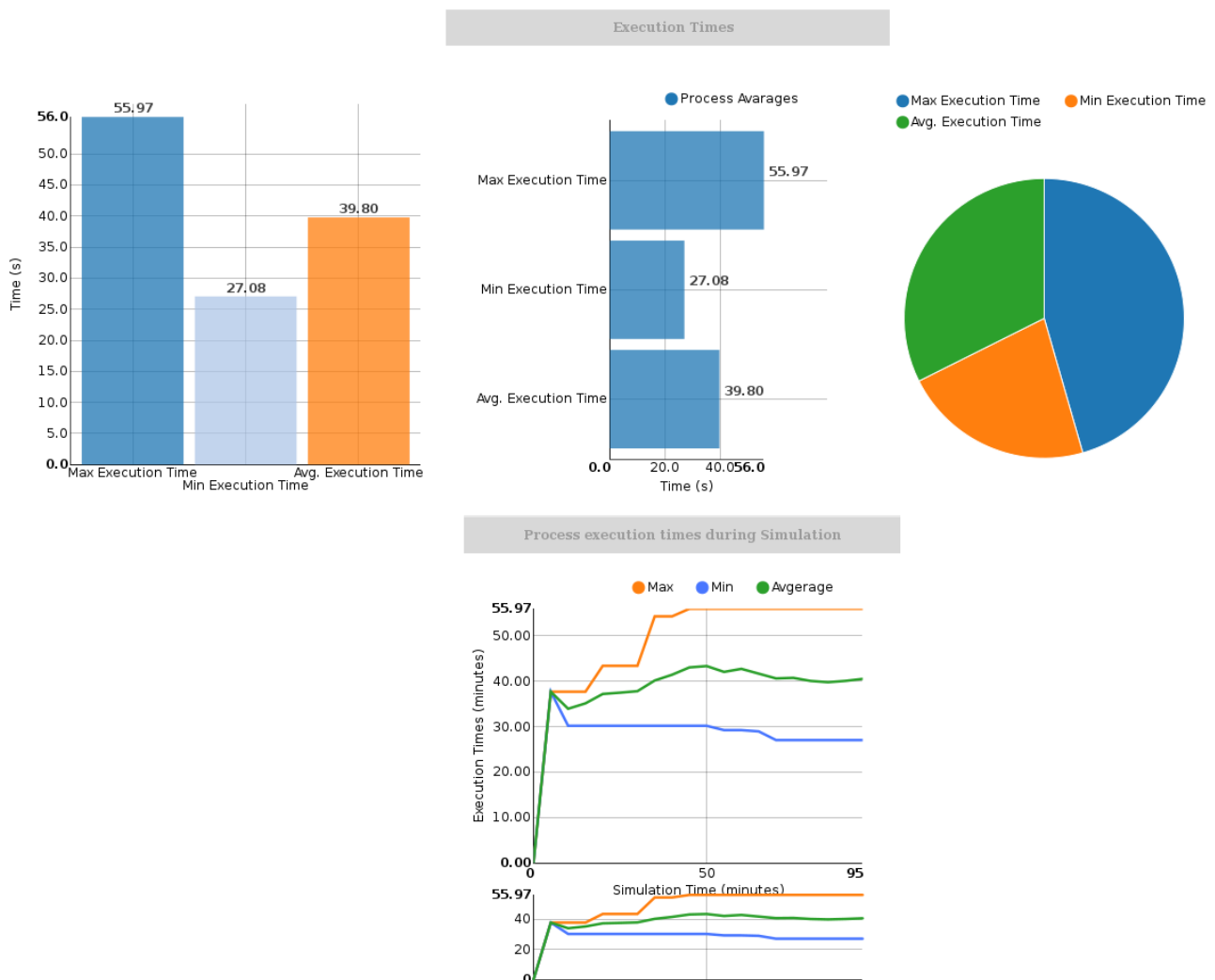


Figure 7.4. Simulation Graph types

Filters

To filter data in a chart, click the item radiobutton in the chart legend.

Execution Times

- Max Execution Time ● Min Execution Time
- Avg. Execution Time

Process execution times during Simulation

- Max
- Min
- Average

Figure 7.5. Graph item radiobutton

Timeline

The Timeline feature allows you to view the graph at the particular stage during simulation execution. Every event is included in the timeline as a new status.


To activate the feature, click the **Timeline**  in the upper right corner of the respective graph: The timeline depicting individual events is displayed in the lower part of the canvas. Click the arrows on the right and left from the chart to move through the timeline. The data current for the particular moment are applied to the chart depicted above instantly.



Figure 7.6. Process Simulation Timeline

Note that in line charts, you can point to a point on a line to see the value of the item at the given time.

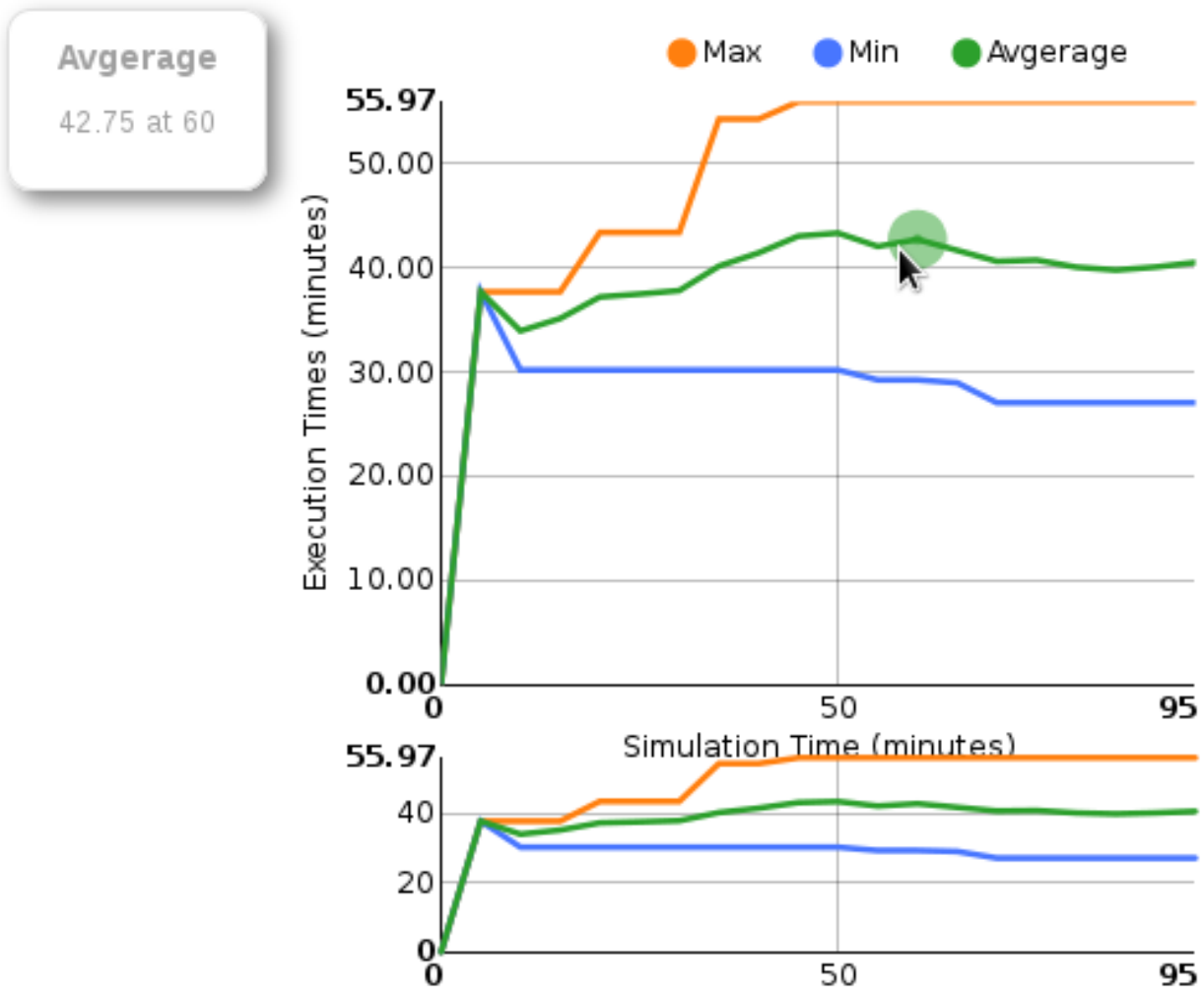


Figure 7.7. Line Chart

CHAPTER 8. TESTING

Even though business processes should not be viewed as code, should be as high-level as possible and should not contain implementation details, they also have a life cycle just like other development artefacts. Therefore testing your Process definitions is just as important as it is when programming.

8.1. UNIT TESTING

When unit testing your Process, you test whether the process behaves as expected in specific use cases; for example, you test the output based on the existing input. To simplify unit testing, the helper class `org.jbpm.test.JbpmJUnitBaseTestCase` is provided in the `jbpm-bpmn2` test module that offers the following:

- helper methods to create a new kie base and session for given processes (Also, you can select if persistence is to be used.)
- assert statements to check among other also the following:
 - 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
 - the value of variables

Example 8.1. JUnit test of the `com.sample.bpmn.hello` Process

```
import org.kie.api.runtime.KieSession;
import org.kie.api.runtime.manager.RuntimeEngine;
import org.kie.api.runtime.process.ProcessInstance;

public class MyProcessTest extends org.jbpm.test.JbpmJUnitBaseTestCase
{
    public void testProcess() {

        // create singleton runtime manager and load the given process(es)
        createRuntimeManager("sample.bpmn");

        // get the single kie session
        RuntimeEngine engine = getRuntimeEngine();
        KieSession ksession = engine.getKieSession();

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

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

        // check whether the given nodes were executed during the process
        execution
            assertNodeTriggered(processInstance.getId(), "StartProcess",
                "Hello", "EndProcess");
    }
}
```

```
}
}
```

The JUnit test will create a new session, start the `com.sample.bpmn.hello` process and verify whether the Process instance completed successfully and whether the nodes `StartProcess`, `Hello`, `EndProcess` have been executed.

8.2. SESSION CREATION

To create a session, it is required to first create a `RuntimeManager` and `RuntimeEngine` from which you will get the session. The following methods can be used to create `RuntimeManager`:

- **`createRuntimeManager(String... process)`** creates default configuration of the `RuntimeManager` with singleton strategy and all processes added to the kie base. There will only be one `RuntimeManager` created during single test and the processes shall be added to the kie base.
- **`createRuntimeManager(Strategy strategy, String identifier, String... process)`** creates default configuration of `RuntimeManager` with given strategy and all processes being added to the kie base. **`Strategy`** selects the strategies that are supported, and **`identifier`** identifies the `RuntimeManager`.
- **`createRuntimeManager(Map<String, ResourceType> resources)`** creates default configuration of `RuntimeManager` with singleton strategy and all resources being added to kie base. The **`resources`** code identifies the processes, rules, etc that shall be added to the kie base.
- **`createRuntimeManager(Map<String, ResourceType> resources, String identifier)`** creates default configuration of `RuntimeManager` with singleton strategy and all resources added to kie base. Like the method above but with an **`identifier`** that identifies the `RuntimeManager`.
- **`createRuntimeManager(Strategy strategy, Map<String, ResourceType> resources)`** creates default configuration of `RuntimeManager` with given strategy and all resources being added to the kie base. There will be only one `RuntimeManager` created during single test. The **`strategy`** code is the selected strategy of those that are supported. The **`resources`** code are all the resources that shall be added to the kie base.
- **`createRuntimeManager(Strategy strategy, Map<String, ResourceType> resources, String identifier)`** creates default configuration of `RuntimeManager` with given strategy and all resources being added to kie base. There will be only one `RuntimeManager` created during single test. The **`strategy`** code selects the supported strategies. The **`resources`** code identifies the resources that shall be added to the kie base. The **`identifier`** code identifies the `RuntimeManger`.
- **`createRuntimeManager(Strategy strategy, Map<String, ResourceType> resources, RuntimeEnvironment environment, String identifier)`** is the lowest level of creation of `RuntimeManager` that expects to get `RuntimeEnvironment` to be given as an argument. It does not assume any particular configuration; that is, it allows you to configure every single piece of `RuntimeManager` manually. The **`strategy`** code selects the strategies of

those that are supported. The **resources** code identifies the resources added to the kie base. The **environment** code is the runtime environment used for RuntimeManager creation. The **identifier** code identifies the RuntimeManager.

The following methods can be used to get RuntimeEngine:

- **getRuntimeEngine()** returns a new RuntimeEngine built from the manager of the test case. It uses EmptyContext that is suitable for the following strategies: singleton and request.
- **getRuntimeEngine(Context<?> context)** returns a new RuntimeEngine built from the manager of the test case. Common use case would be to maintain the same session for the process instance and thus a ProcessInstanceContext shall be used. The **context** code is the instance of the context that shall be used to create RuntimeManager.

8.2.1. Assertions

The following assertions are available for testing the current state of a process instance:

- **assertProcessInstanceActive(long processInstanceId, KieSession ksession)**: checks whether the Process instance with the given id is active.
- **assertProcessInstanceCompleted(long processInstanceId, KieSession ksession)**: checks whether the Process instance with the given id has completed. successfully
- **assertProcessInstanceAborted(long processInstanceId, KieSession ksession)**: checks whether the Process instance with the given id was aborted.
- **assertNodeActive(long processInstanceId, KieSession ksession, String... name)**: checks whether the process instance with the given id contains at least one active node with the given node names.
- **assertNodeTriggered(long processInstanceId, String... nodeNames)**: checks for each given node name whether a node instance was triggered during the execution of the Process instance.
- **getVariableValue(String name, long processInstanceId, KieSession ksession)**: retrieves the value of the variable with the given name from the given Process instance.

8.2.2. Integration with external services

To test possible scenarios connected to collaboration of Process Tasks with external services, you can use TestWorkItemHandler. TestWorkItemHandler is provided by default can be registered to collect all Work Items of a given type, for example sending an email or invoking a service, and contains all the data related to that task).

This test handler can be queried during unit testing to check whether specific work was requested during the execution of the Process and that the data associated with the work was correct.

Example 8.2. Testing an Email Task

Let's assume we want to test the Process depicted in [Figure 8.1, "Process with a custom Email Service Task"](#). The test should in particular check if an exception is raised when the email sending fails. The failure is simulated by notifying the engine that the sending the email could not be completed:

```
import org.kie.api.runtime.manager.RuntimeManager;
import org.kie.api.runtime.manager.RuntimeEngine;
```



```

import org.kie.api.runtime.KieSession;
import org.kie.api.runtime.process.WorkItem;
import org.kie.api.runtime.process.WorkItemHandler;
import org.kie.api.runtime.process.ProcessInstance;
import org.kie.api.runtime.process.WorkItemManger;

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");
}

```

The test case uses a test handler that registers when an email is requested and allows you to test the data related to the email. Once the engine has been notified the email could not be sent by the **abortWorkItem(..)** method call, the unit test verifies that the Process handles this case by logging the fact and terminating with an error.

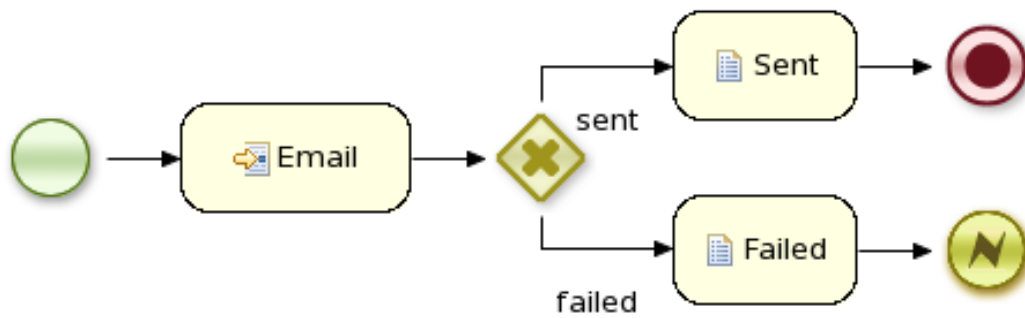


Figure 8.1. Process with a custom Email Service Task

8.2.3. Persistence

To simplify unit testing, jBPM includes a helper class called **JbpmJUnitBaseTestCase** in the `jbpm-test` module that greatly simplifies your junit testing. This helper class provides methods to create a new **RuntimeManager** and **RuntimeEngine** for a given process or set of processes. By default, persistence is not used, and it is controlled by the super constructor. The helper method allows you to select whether or not you wish to use persistence. The example below shows a helper class to allow persistence:

Example 8.3.

```

public class ProcessHumanTaskTest extends JbpmJUnitBaseTestCase {

    public ProcessPersistenceTest() {

        // setup data source, enable persistence

        super(true, true);

    }

    ....
  
```

PART III. PLUG-IN

CHAPTER 9. PLUG-IN

Red Hat JBoss BPM Suite comes with a plug-in for Red Hat JBoss Developer Studio to provide support for the development of business processes in the Eclipse-based environment, such as debugging and testing. It also provides a graphical Process Designer for business process editing.

Note that the repository structure follows the maven structure and is described in [Chapter 3, Project](#).

For instructions on how to install and set up the plug-in refer to the *Red Hat JBoss BPM Suite Installation Guide*.

9.1. CREATING BPM PROJECT

To create a BPM project, do the following:

1. On the main menu of JBoss Developer Studio, click **File** → **New** → **jBPM project** and then **File** → **New** → **Other**.
2. Then choose **jBPM** → **jBPM project**.
3. In the **New jBPM Project** dialog, define the project name and location and click **Next**.
4. Select the required content of the project and click **Next**.
5. Select the runtime to be used by the project or click **Configure Workspace Settings** and define a new runtime (for details on runtime resources, refer to the *Red Hat JBoss BPM Suite Installation Guide*).
6. Select the required compatibility mode and click **Finish**.

The project with predefined maven structure and imported libraries is created in the defined workspace location and appears in navigation views (**Package Explorer**, **Navigator**).



NOTE

It is possible to create a Mavenized jBPM Project by selecting the following:

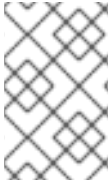
- Click **jBPM** → **jBPM Project (Maven)**

9.2. CREATING PROCESS

In JBoss Developer Studio with the Red Hat JBoss BPM Suite plug-in, a Process is created the same way as other resources:

1. Choose **File** → **New** → **Other...**
2. Select **jBPM** → **BPMN2 Process**.
3. In the displayed dialog box, define the location and the filename of the Process. Make sure you follow maven structure requirements.

Once created, the Process is opened for editing in the graphical Process Designer.



NOTE

Note that this Process may also be created with the following category:

- Select **BPMN2** → **jBPM Process Diagram**.

9.3. USING THE DEBUG PERSPECTIVE

In the Red Hat JBoss Developer Studio with Red Hat JBoss BPM Suite plug-in, you can make use of the extended debugging feature (debugging allows you to visualize and inspect the current state of running process instances).

Note that breakpoints on Process elements are currently not supported. However, you can define breakpoints inside any Java code in your Process; that is, your application code that is invoking the engine or invoked by the engine, listeners, etc. or inside rules that are evaluated in the context of a Process.

Procedure 9.1. The Debug Perspective

1. Open the Process Instance view **Window > Show View > Other ...**
2. Select **Process Instances and Process Instance** under the **Drools** category
3. Use a Java breakpoint to stop your application at a specific point (for example, after starting a new process instance).
4. In the Debug perspective, select the ksession you would like to inspect.
5. The *Process Instances* view will show the process instances that are currently active inside that ksession.
6. When double-clicking a process instance, the process instance viewer will graphically show the progress of that process instance.
7. Sometimes, when double-clicking a process instance, the process instance viewer complains that it cannot find the process. This means that the plug-in was not 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.

The screenshot below illustrates the running process instance with an id of "1". This example process instance relies on a human actor to perform "Task 1".

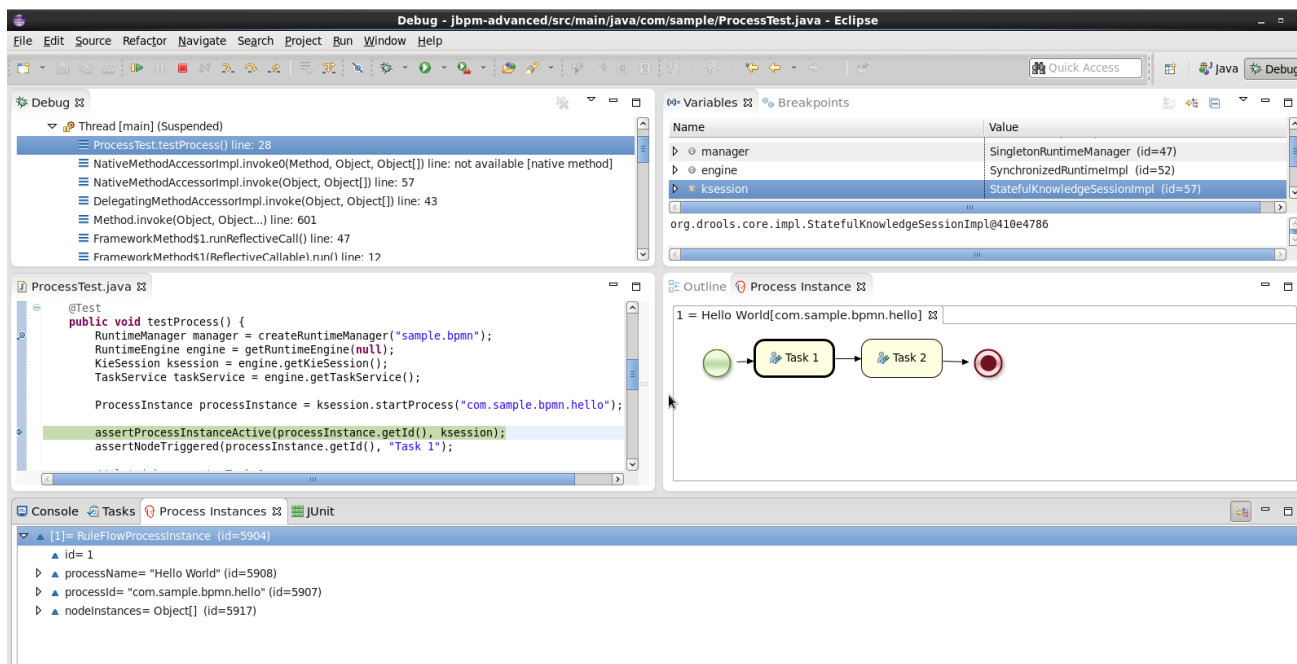


Figure 9.1. Process Instance in the Debugger

NOTE

The process instances view shows the process instances currently active inside the selected ksession. When using persistence, process instances are not kept in memory inside the ksession; that is, 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.

The environment provides also other views that are related to rule execution like the working memory view, the agenda view, etc. For further information, refer to the Red Hat JBoss BRMS documentation.

9.4. CHECKING SESSION LOGS

You can check the session logs in the audit log, which is a log of all events that were logged from the session. Audit log is an XML-based log file which contains a log of all the events that occurred while executing a specific ksession.

Procedure 9.2. Creating a logger

1. To create a logger, use KieServices as depicted below:

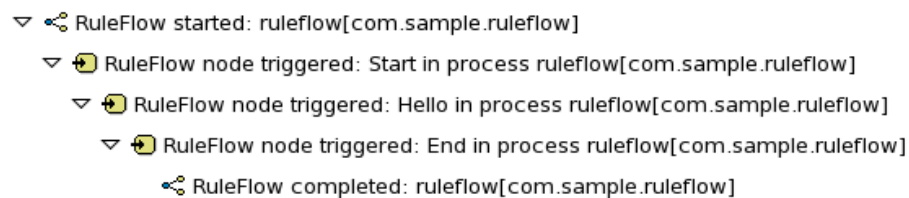
```
KieRuntimeLogger logger = KieServices.Factory.get().getLoggers()
    .newThreadedFileLogger(ksession, "mylogfile", 1000);
// do something with the ksession here
logger.close();
```

2. Attach the new logger to a ksession.
3. Be sure to close the logger after usage.

Procedure 9.3. Using Audit View

Procedure 9.3. USING Audit view

1. To use *Audit View*, open **Window > Show View > Other ...**
2. Under the **Drools** category, select **Audit**.
3. To open a log file in *Audit View*, select the log file 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*.
4. A tree-based view is generated based on the data inside the audit log. Depicted below is an example tree-based view:

**Figure 9.2. Tree-Based View**

5. An event is shown as a subnode of another event if the child event is caused by a direct consequence of the parent event.

**NOTE**

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 updated 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).

PART IV. DEPLOYMENT AND RUNTIME MANAGEMENT

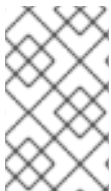
CHAPTER 10. DEPLOYING PROJECTS

Once you have created a project with your Process definition and relevant resources, you need to build it and deploy it to the Process engine. Once deployed, you can create Process instances based on the deployed resources.

To deploy your project from Business Central, do the following:

1. Open the **Project Editor** on your project (in **Project Explorer** navigate to your project and in the top menu, click **Tools** → **Project Editor**).
2. You can define the Kie Base and Kie Session properties. If not, the default kbase and ksession will be used.

3. On the top menu, click the **Build & Deploy**  button.



NOTE

From the 6.1 version of Red Hat JBoss BPM Suite, deployment units are stored inside the database instead of the GIT repository. To override this behavior, set the `org.kie.git.deployments.enabled` property to true.


10.1. PROCESS INSTANCES

Once you have modeled and deployed a Process definition along with all the resources it requires, you can create its runtime instance, which will run on the Process engine.

From the Business Central you can further manage the instance during runtime, monitor its execution, and work with the Tasks the instance produces if having the proper roles assigned.

10.1.1. Instantiating a Process

To instantiate a deployed Process definition, do the following:

1. Display the **Process Definitions** view: on the top menu, click **Process Management** → **Process Definitions**.
2. Look up the Process Definition and in the respective row, click the **Instantiate**  icon.
3. In the displayed dialog view, enter the properties and input parameters for the Process instance.

10.1.2. Monitoring a Process instance

You can monitor the progress of a running Process instance from the Business Console:

1. On the top menu of the Business Central, go to **Process Management** → **Process Instances**.
2. In the list on the **Process Instances** tab, locate the required running Process instance and

click the **Details**



button in the instance row.

10.1.3. Aborting a Process instance

You can abort a running Process instance either using the provided API or from the Management Console.

Aborting a Process instance using API

To abort a Process instance using the Kie Session API, use the `void abortProcessInstance(long processInstanceId)` call on the parent Kie Session.

Aborting a Process instance from the Management Console

To abort a Process instance from the Management Console, do the following:

1. On the top menu of the Management Console, go to **Process Management** → **Process Instances**.
2. In the list on the **Process Instances** tab, locate the required Process instance and click the

Abort  button in the instance row.


10.2. USER TASKS

A User Task represents a piece of work the given user can claim and perform. User Tasks can be handled within the Task client perspective of the Business Central: the view displays the Task List for the given user. You can think about it as a to-do item. The User Task appears in your list either because the User Task element generated the User Task as part of Process execution or because someone has created the User Task directly in the Business Central console..

A User Task can be assigned to a particular actor, multiple actors, or to a group of actors. If assigned to multiple actors or a group of actors, it is visible in the Task Lists of all the actors and any of the possible actors can claim the task and execute it. The moment the Task is claimed by one actor, it disappears from the Task List of other actors.

Task client

User Tasks are displayed in the Tasks perspective, that are an implementation of a Task client, in the Business Central console: to display the Tasks perspective, click **Tasks** → **Tasks List**. The perspective provides multiple Task views: to switch between the views, click the respective button in the menu bar of the **Tasks List** view (**Grid** or **Calendar**). The **Calendar** view provides the Task list in **Day**, **Week**, or **Month** layouts. You can filter out the Tasks based on their status using the buttons **Active**,

Personal, **Group**, and **All**  in the view toolbar.

Note that group Tasks are marked with the group icon and you can claim them by clicking the **Claim**



button. To undo the claim process, click the button again.

The Task details are available in the panel that appears after clicking the Details icon in the Actions



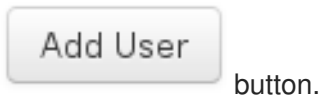
column : the panel contains the Work, Assignments, Details, and Comments button: On the Comments button, you can review comments provided by other users on the Task and add your own comments. To work on the Task, open the Work tab: the respective Form defined for the Task appears (if the current user hasn't been assigned the task, a form to claim the task appears first)). If no Form for the Task is defined a default Form is generated based on the `jbpm-playground.git/globals/forms/DefaultTask.ftl` file and on the input and output data of the Task.

10.2.1. Creating user task

A user task can be created either by a User Task element executed as part of a Process instance or you can create a user task directly in Business Central.

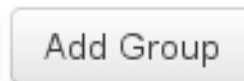
To create a user task in the web environment, do the following:

1. Open the **Tasks** drop down menu (**Tasks** → **Tasks List**).
2. On the **Tasks List** tab, click **New Task** and define the task parameters:
 - Task Name: the task display name
 - Advanced
 - Due Date
 - User: user name of the person to execute the task
 - Priority: priority level
3. A **Task** cannot be created without a **User** or a **Group**. To add more users, select the **Add User**



button.

4. To add more groups, select the **Add Group**
5. Click the **Create** button when you are satisfied with the information provided.



button.

CHAPTER 11. LOGGING

Logs with execution information are created based on events generated by the process engine during execution. It is the engine providing a generic mechanism listening to events. Information about the caught event can be extracted from these logs and then persisted in a data storage. To restrain the logged information, the log filtering mechanism is provided (for further information, refer to the *Red Hat JBoss BPM Suite Administration and Configuration Guide*).

CHAPTER 12. EXAMPLES

Red Hat JBoss BPM Suite comes with a project with assets examples to demonstrate the possible usage and capabilities of the product.

Also, the project contains Junit tests for each Element, which are simple working examples. These test processes can serve as simple examples. The entire list can be found in the `src/test/resources` folder for the `jbpm-bpmn2` module. Note that each of the processes is accompanied by a junit test that tests the implementation of the construct.

PART V. BAM

CHAPTER 13. RED HAT JBOSS DASHBOARD BUILDER

Red Hat JBoss Dashboard Builder is a web-based dashboard application that provides Business Activity Monitoring (BAM) support, that is, visualization tools for monitored metrics (Key Performance Indicators, or KPIs) in real time. It comes integrated in the Business Central environment under the **Dashboards** menu.

It comes with a dashboard that requests information from the Red Hat JBoss BPM Suite Execution Engine and provides real-time information on its runtime data; however, you can create also custom dashboards over other data resources, which leaves the application relatively standalone.

WHAT IS BUSINESS ACTIVITY MONITORING?

Business Activity Monitoring (BAM) software helps to monitor business activities that take place on a computer system in real time. The software monitors particular metrics, such as, the status of running processes, the number of invoices to be processed, processing times, etc. It provides tools for visualization of the collected data in graphs, tables, etc.

13.1. ACCESSING DASHBOARD BUILDER

Dashboard Builder is accessible in both *business-central* and as a standalone applications.

Within *business-central*, the *Dashboard Builder* is accessed directly from the **Dashboards** menu, and it integrates the jBPM Dashboard workspace that is accessible using **Dashboards > Process & Task Dashboards** menus.

- **Process & Task Dashboards** displays a pre-defined dashboard based on runtime data from the Execution Server. In the menu on the left, select the entity you are interested. The widgets on the right will display the data for the entity.

As a standalone application, *Dashboard Builder* can be accessed in one of the following ways:

- Using the URL <https://HOSTNAME/dashbuilder> (with the appropriate Hostname).
- Using *business-central* with **Dashboards > Business Dashboard** menus.
 - **Business Dashboards** displays the environment in which you can create your own dashboards. Procedures on how to create a custom dashboard are provided below.

13.2. BASIC CONCEPTS

The Dashboard Builder can establish connections to external data sources such as databases. These connections are then used for creating data providers that obtain data from the data sources. The Dashboard Builder is connected to the local JBoss BPM Suite engine by default and acquires from it the data for its JBoss BPM Suite Dashboard indicators (widgets with visualizations of the data available on the pages of the JBoss BPM Suite Dashboard workspace).

If operating over a database, the data provider uses an SQL query to obtain the data and if operating over a CSV file, the data provider automatically obtains all the data from the file. So it is the data providers that keep the data you work with.

Data from the data providers can then be visualized in indicators, special panels, on pages as graphs or tables. Pages are contained within a workspace and can define permission access rights. The number of pages within a workspace is arbitrary. A set of pages that present related information on similar KPIs is referred to as a dashboard.

13.3. ENVIRONMENT

There are two ways to access the *Dashbuilder* environment: through *business-central* and your local host.

Procedure 13.1. To access *Dashbuilder* through *business-central*

1. Log into *business-central* with your user account.
2. Select **Dashboard > Business Dashboard**.
3. *Dashbuilder* will open in a new browser / window tab.

Procedure 13.2. To access *Dashbuilder* through web browser.

1. Go to *Dashbuilder* directly through <https://HOSTNAME/dashbuilder>
2. An example instance running the local host would be <https://localhost:8080/dashbuilder>

After you log in, you are redirected to the Showcase workspace with the welcome page displayed.

On the top is the menu panel with the options on workspaces and pages, and logout and general configuration buttons. This part is common for all workspaces.

Below is the dashboard area with variable content. It consists of the sidebar or lateral menu on the left and the main dashboard area on the right.

13.4. DATA SOURCES

Red Hat JBoss Dashboard Builder can be connected to an external database, be it using JNDI of the container or connecting directly only using the JDBC driver to access the database. Connections to databases can be configured in workspace *Showcase* on page *External Connections*. After you have established the connection to the database, you need to create a data provider that will collect the data from the database and allow you to visualize it as an indicator in the dashboard area of a page.

When connecting to CSV files to acquire data, the connection is established directly through the data provider.

Note that Red Hat JBoss Dashboard Builder makes use of its own local internal database to store its local data. This database is read-only for Dashboard Builder, but is accessible from outside.

13.4.1. Connecting to data sources

You can connect either to a JNDI data source, that is, a data source set up and accessible from the application container, or directly to the data source as a custom data source, if the application container has the correct JDBC driver deployed.

To connect to an external data source, do the following:

1. Make sure the data source is up and running and that the application server has access to the data source. (Check the driver, the login credentials, etc. In Red Hat JBoss EAP 6, you can do so in the Management Console under **Subsystems** → **Connector** → **Datasources**)
2. In Dashboard Builder, on the Tree Menu (by default located on the of the Showcase perspective), go to **Administration** → **External connections**.

3. On the displayed External Connection panel, click the **New DataSource**

 **Create new DataSource** button.

4. Select the data source type (JNDI or Custom DataSource) and provide the respective data source parameters below.

If you wish the jBPM Dashboard to use the new data source, modify also the respective data providers (jBPM Count Processes, jBPM Process Summary, jBPM Task Summary). Note that the data source needs to have access to jBPM history.

13.4.2. Security considerations



IMPORTANT

When creating an external datasource using JBoss Dashboard Builder, it needs to use the local connection so that the user can be passed through. Otherwise, with a connection that uses `<host>:<port>`, every user would have the same virtual database (VDB) permissions.

13.4.3. Building a Dashboard for Large Volumes of Data

You can connect Red Hat JBoss Dashboard Builder to external databases and load data for generating reports and charts. Generally, if the volume of data is small (up to 2MB), Red Hat JBoss Dashboard Builder preloads the data into (local) memory and uses this data for report and chart generation. However, in case of large volumes of data, it is not possible to load the entire data set into the Dashboard Builder's local memory.

Based on the volume of data you are dealing with, you can choose to query the database to build a dashboard report in any one of the following strategies:

- The in-memory strategy

The in-memory strategy is to create a data provider that loads all the required data from the database by executing a single SQL query on the relevant tables, into the Dashboard Builder's memory. In this case, every indicator on the Dashboard Builder shares the same data set. When you use filters from the Dashboard Builder user interface to access specific data from this data set, the Dashboard Builder fetches the data from the internal memory and does not execute another SQL query again on the database. This strategy has a simple data retrieval logic as it deals with creating a single data provider. As all the data set properties are available to you at once, it allows you to configure KPIs faster. However, this approach is not suitable for large data sets as it would lead to poor performance.

- The native strategy

The native approach is to create a data provider for every indicator in the Dashboard Builder and does not require loading all the data into the internal memory at once. So each time you use a filter from the Dashboard Builder user interface, the corresponding SQL queries get executed and fetches the required data from the database. So there is no data in the Dashboard Builder's internal memory. This strategy works best in case of large volumes of data, however it needs proper indexing on the database tables. Also, setting up data providers for multiple KPIs is complicated as compared to creating a single data provider in case of in-memory strategy.

Example

Let us consider a case when you want to create a stock exchange dashboard comprising the following charts and reports:

- Bar chart for Average price per company
- Area chart for Sales price evolution
- Pie chart for Companies per country
- Table report for Stock prices at closing date

For these charts and reports, let us assume that the Dashboard Builder accesses data from the following tables:

- Company: Comprising columns ID, NAME, and COUNTRY.
- Stock: Comprising columns ID, ID_COMPANY, PRICE_PER_SHARE, and CLOSING_DATE.

For the in-memory strategy of building a dashboard, the following SQL query fetches all the required data from these two tables:

```
SELECT C.NAME, C.COUNTRY, S.PRICE_PER_SHARE, S.CLOSING_DATE
FROM COMPANY C JOIN STOCK S ON (C.ID=S.ID_COMPANY)
```

The output of this query is saved in the Dashboard Builder's local memory. The Dashboard accesses this data every time a filter is run.

On the other hand, if you are using the native strategy for huge volumes of data, an SQL query is executed on every filter request made by the Dashboard Builder and corresponding data is fetched from the database. In this case here is how each filter accesses the database:

- For the bar chart on *Average price per company*, the following SQL query is executed:

```
SELECT C.NAME, AVG(S.PRICE_PER_SHARE)
FROM COMPANY C JOIN STOCK S ON (C.ID=S.ID_COMPANY)
WHERE {sql_condition, optional, c.country, country}
AND {sql_condition, optional, c.name, name}
GROUP BY C.NAME
```

- For the area chart on *Sales price evolution*, the following SQL query is executed:

```
SELECT S.CLOSING_DATE, AVG(S.PRICE_PER_SHARE)
FROM COMPANY C JOIN STOCK S ON (C.ID=S.ID_COMPANY)
WHERE {sql_condition, optional, c.country, country}
AND {sql_condition, optional, c.name, name}
GROUP BY CLOSING_DATE
```

- For the pie chart on *Companies per country*, the following SQL query is executed:

```
SELECT COUNTRY, COUNT(ID)
FROM COMPANY
WHERE {sql_condition, optional, country, country}
AND {sql_condition, optional, name, name}
GROUP BY COUNTRY
```

- For the table report on *Stock prices at closing date*, the following SQL query is executed:

```
SELECT C.NAME, C.COUNTRY, S.PRICE_PER_SHARE, S.CLOSING_DATE
FROM COMPANY C JOIN STOCK S ON (C.ID=S.ID_COMPANY)
WHERE {sql_condition, optional, c.country, country}
AND {sql_condition, optional, c.name, name}
```

For each of these queries, you need to create a separate SQL data provider.

In the examples above, each KPI delegates the filter and group by operations to the database through the `{sql_condition}` clauses. The signature of the `{sql_condition}` clause is the following:

```
{sql_condition, [optional | required], [db column], [filter property]}
```

Here,

- `optional`: This indicates that if there is no filter for the given property, then the condition is ignored.
- `required`: This indicates that if there is no filter for the given property, then the SQL returns no data.
- `db column`: This indicates the database column where the current filter is applied.
- `filter property`: This indicates the selected UI filter property.

When a filter occurs in the UI, the Dashboard Builder parses and injects all the SQL data providers referenced by the KPIs into these SQL statements. Every time a filter occurs in the UI, the Dashboard Builder gets all the SQL data providers referenced by the KPIs and injects the current filter selections made by the user into these SQLs.


13.4.4. Data providers

Data providers are entities that are configured to connect to a data source (a CSV file or database), collect the required data, and assign them the data type. You can think about them as database queries.

The collected data can be then visualized in indicators on pages, exported as XLS or CSV, etc.

13.4.4.1. Creating data providers

To create a new data provider, do the following:


1. In the Tree Menu (the panel in the lateral menu of the Showcase workspace), click **Administration** → **Data providers**.
2. In the **Data Providers** panel, click the **Create new data provider**  **Create new data provider** button.
3. In the updated **Data Providers** panel, select in the **Type** dropdown menu the type of the data provider depending on the source you want the data provider to operate on.
4. Define the data provider parameters:

Data provider over a CSV file

- Name: user-friendly name and its locale
- CSV file URL: the url of the file (for example, **file:///home/me/example.csv**)
- Data separator: the symbol used as separator in the CSV file (the default value is semicolon; if using comma as the separator sign, make sure to adapt the number format if applicable)
- Quoting symbol: the symbol used for quotes (the default value is the double-quotes symbol; note that the symbol may vary depending on the locale)
- Escaping symbol: the symbol used for escaping the following symbol in order to keep its literal value
- Date format: date and time format
- Number format: the format of numbers as resolved to thousands and decimals

Data provider over a database (SQL query)

- Name: user-friendly name and its locale
- Data source: the data source to query (the default value is **local**, which allows you to query the Dashboard Builder database)
- Query: query that returns the required data


5. Click **Attempt data load**  to verify the parameters are correct.
6. Click **Save**.
7. In the table with the detected data, define the data type and if necessary provide a user-friendly name for the data. Click **Save**.

The data provider can now be visualized in an indicator on a page of your choice.

13.4.5. Workspace

A workspace is a container for pages with panels or indicators.

By default, the Showcase and jBPM Dashboard workspaces are available.

To switch between workspaces, select the required workspace in the Workspace dropdown box in the top panel on the left. To create a new workspace, click the **Create workspace** icon () in the top menu on the left. You can also edit the current workspace properties, delete the current workspace, and duplicate the current workspace using icons in the top panel.

Every workspace uses a particular skin and envelope, which define the workspace's graphical properties.

13.4.5.1. Creating a workspace

To create a new workspace, do the following:

1. Click the **Create workspace** button on the top menu.

The management console with the **Workspace** node expanded and workspace management area with workspace details on the right is displayed.

2. In the **Create workspace** table on the right, set the workspace parameters:

- Name: workspace name and its locale
- Title: workspace title and its locale
- Skin: skin to be applied on the workspace resources
- Envelope: envelope to be applied on the workspace resources

3. Click **Create workspace**.

4. Optionally, click the workspace name in the tree menu on the left and in the area with workspace properties on the right define additional workspace parameters:

- URL: the workspace URL
- User home search: the home page setting

If set to **Role assigned page**, the home page as as in the page permissions is applied; that is, every role can have a different page displayed as its home page. If set to **Current page**, all users will use the current home page as their home page.

13.4.5.2. Pages

Pages are units that live in a workspace and provide space (dashboard) for panels. By default, you can display a page by selecting it in the Page dropdown menu in the top panel.


Every page is divided in two main parts: the lateral menu and the central part of the page. The parts are divided further (the exact division is visible when placing a new panel on a page). Note that the lateral menu allows you to insert panels only below each other, while in the central part of the page you can insert panels below each other as well as tab them.

A page also has a customizable header part and logo area.

13.4.5.2.1. Creating Pages

To create a new page, do the following:

1. Make sure you are in the correct workspace.

2. Next to the **Page** dropdown box  in the top menu, click the **Create new page**  button .

3. The management console with the **Pages** node expanded and page management area with page details on the right is displayed.

4. In the **Create new page** table on the right, set the page parameters:


- Name: page name and its locale

- Parent page: parent page of the new page
 - Skin: skin to be applied on the page
 - Envelope: envelope to be applied on the page
 - Page layout: layout of the page
5. Click **Create new page**.
 6. Optionally, click the page name in the tree menu on the left and in the area with workspace properties on the right define additional page parameters:
 - URL: the page URL
 - Visible page: visibility of the page
 - Spacing between regions and panels

13.4.5.2.2. Defining Page permissions

Although users are usually authorized using the authorization method setup for the underlying application container (on Red Hat JBoss EAP, the **other** security domain by default), the Red Hat JBoss Dashboard Builder has its own role-based access control (RBAC) management tool to facilitate permission management on an individual page or multiple pages.

To define permissions on a page or all workspace pages for a role, do the following:

1. On the top menu, click the **General configuration**  button : the management console is displayed.
2. Under the **Workspace** node on the left, locate the page or the **Pages** node.
3. Under the page/pages node, click the **Page permissions** node.
4. In the **Page permissions** area on the right, delete previously defined permission definition if applicable and define the rights for the required role:
 - a. In the **Permission assignment** table, locate the **Select role** dropdown menu and pick the respective role.
 - b. In the **Actions** column of the table, enable or disable individual permissions.
5. Click **Save**.

13.4.5.3. Panels

A panel is a GUI widget, which can be placed on a page. There are three main types of panels:

Dashboard panels

are the primary BAM panels and include the following:

- Data provider manager: a panel with a list of available data providers and data provider management options

- Filter and Drill-down: a panel that displays all KPIs and their values to facilitate filtering in indicators on the given page defined over a data provider
- HTML Editor panel: a panel with static content
- Key Performance Indicator (indicator): a panel that visualizes the data of a data provider

Navigation panels

are panels that provide navigation functions and include the following:

- Breadcrumb: a panel with the full page hierarchy pointing to the current page
- Language menu: a panel with available locales (by default in the top center)
- Logout panel: a panel with the name of the currently logged-in user and the logout button
- Page menu custom: a panel with vertically arranged links to all pages in the workspace (the list of pages can be adjusted) and general controls for the HTML source of the page
- Page menu vertical: a panel with vertically arranged links to all pages in the workspace (the list of pages can be adjusted)
- Page menu horizontal: a panel with horizontally arranged links to all pages in the workspace (the list of pages can be adjusted)
- Tree menu: a panel with the links to essential features such as Administration, Home (on the Home page of the Showcase workspace displayed on the left, in the lateral menu)
- Workspace menu custom: a panel with links to available workspaces (the list of workspaces can be adjusted) and general controls for the HTML source of the workspace
- Workspace menu horizontal: a horizontal panel with links to available workspaces (the list of workspaces can be adjusted)
- Workspace menu vertical: a vertical panel with links to available workspaces (the list of workspaces can be adjusted)


System panels

are panels that provide access to system setting and administration facilities and include the following:

- Data source manager: a panel for management of external data sources
- Export dashboards: a panel export of dashboards
- Export/Import workspaces: a panel for exporting and importing of workspaces

13.4.5.3.1. Adding panels

To add an existing panel to a page or to create a new panel, do the following:

1. Make sure the respective page is open (in the **Page** dropdown menu of the top menu select the page).
2. In the top menu, click the **Create a new panel in current page**  button.

3. In the displayed dialog box, expand the panel type you want to add (**Dashboard**, **Navigation**, or **System**) and click the panel you wish to add.
4. From the **Components** menu on the left, drag and drop the name of an existing panel instance or the **Create panel** item into the required location on the page.

If inserting a new indicator, the Panel view with the graph settings will appear. Define the graph details and close the dialog.

If adding an instance of an already existing indicator, you might not be able to use it, if it is linked to the KPIs on the particular original page. In such a case, create a new panel.

5. If applicable, edit the content of the newly added panel.

CHAPTER 14. MANAGEMENT CONSOLE

The management console is accessible in standalone *Dashbuilder* using the **General configuration** button located in the top left of the menu. It is NOT accessible in the **Process & Task Dashboard** that is built into *business-central*.

The management console page contains a tree menu with the main administration resources on the left:

- Workspaces tree with individual workspaces and their pages (general item settings are displayed on the right)
- Graphic resources tree with options for upload of new graphic resources and management of the existing ones
- General permissions with access roles definitions and access permission management

To switch back to the last workspace page, click the **Workspace**

 button in the upper left corner.

CHAPTER 15. GRAPHIC RESOURCES

Red Hat JBoss Dashboard Builder uses the following components to define the environment appearance and thus divide the representation resources from content and data:

- Skins define a set of style sheets, images, and icons
- Region layouts define layouts of regions for pages
- Envelopes define an HTML template used as page frames

GRAPHIC RESOURCES DEFINITIONS

All graphics components are deployed as zip files as part of the Red Hat JBoss Dashboard Builder in the `$DEPLOYMENT_LOCATION/dashbuilder.war/WEB-INF/etc/` directory.

Every component definition contains the following:

- properties file that defines the name of the component for individual supported locales, the name of the css file to be applied on the component, and mapping of file to individual component elements
- JSP, HTML, CSS files, and image and icon resources referenced from the properties file

When creating custom graphic resources, it is recommended to download one of the existing components and modify it as necessary. This will prevent unnecessary mistakes in your definition.

15.1. WORKING WITH GRAPHIC RESOURCES

1. On the top menu, click the **General configuration** button .
2. Under the **Graphic resources** node on the left, click the component type you want to work with (**Skins**, **Layouts**, **Envelopers**). The window on the right will display the content relevant for the given component type.
3. On the right, you can now do the following:
 - Upload a new component: you need to provide a unique ID for the component and the resource zip file. Then click **Add**.
 - Download a component definition or preview the component: in the table below the Add view, click the respective icon in the **Actions** column.

APPENDIX A. PROCESS ELEMENTS

A Process Element is a node of the Process definition. The term covers the nodes with execution semantics as well as those without.

A.1. PROCESS

A Process is a named element defined in a Process Definition. It exists in a Knowledge Base and is identified by its ID.

A Process represents a namespace and serves as a container for a set of modeling elements: it contains elements that specify the execution workflow of a Business Process or its part using Flow objects and Flows. Every Process must contain at least one Start Event and one End Event.

A Process is accompanied by its BPMN Diagram, which is also part of the Process Definition and defines how the Process execution workflow is depicted when visualized, for example in the Process Designer.

Apart from the execution workflow and Process attributes, a Process can define Process variables, which store Process data during runtime. For more information on Process variables, refer to [Section 4.8, “Variables”](#).

Runtime

During runtime, a Process serves as a blueprint for a Process instance (the concept of class and its object in OOP). A Process instance lives in a Session that may contain multiple Process instances. This allows the instances to share data, for example, using Globals that live in the Session instance, not in the Process instance. Every Process instance has its own context and ID.

Knowledge Runtime, called **kcontext**, holds all the Process runtime data. You can call it in your code, for example, in Action scripts, to obtain or modify the runtime data:

- Getting the currently executed Element instance so as to query further Element data, such as its name and type, or cancel the Element instance.

Example A.1. Getting the currently executed Element

```
NodeInstance element = kcontext.getNodeInstance();
String name = element.getNodeName();
```

- Getting the currently executed Process instance so as to query further Process instance data, such as, its name, ID, or abort or send an event, such as a Signal.

Example A.2. Getting the currently executed Process and sending it a Signal event

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

- Getting and setting the values of variables
- Execute calls on the Knowledge runtime, for example, start Process instances, insert data, etc.

A Process instance goes through the following life cycle:

1. The **createProcessInstance** method is called on a Process: a new process instance based on a Process is created and Process variables are initialized . The process instance is in status **CREATED**.
2. The **start()** method is called on the ProcessInstance: the execution of the Process instance is triggered (the token on the Start Event is generated). If the Process was instantiated manually, the token is generated only on its None Start Event. If it is instantiated using another mechanism, such as Signal, Message, or Error, the token is generated on the Start Event of the respective type that is defined to handle the particular object. The process instance becomes **ACTIVE**.
3. Once there is no token in the flow (tokens are consumed by End Events and destroyed by Terminating Events), the Process instance is finished and becomes **CANCELLED**.

The runtime state of a Process instance can be made persistent, for example, in a database. This allows to restore the state of execution in case of environment failure, or to temporarily remove running instances from memory and restore them later. By default, process instances are not made persistent. For more information on persistence refer to the *Administration and Configuration Guide for JBoss BPM Suite*.

Properties

ID

Process ID defined as a String unique in the parent Knowledge Base

Example value: **org.jboss.exampleProcess**

It is recommended to use the ID form **<packageName>.<processName>.<version>**.

Name

Process display name

Version

Process version

Package

Parent package the Process belongs to (Process namespace)

Possible values: **true, false**

Target Namespace

BPMN2 xsd location

Executable

type of the process as concerns its executability

Possible values: **true, false**

Imports

Imported Process

Documentation

Documentation is a generic element attribute that can contain element description. It has no impact on runtime.

AdHoc

Boolean property defining whether a Process is an Ad-hoc Process:

If set to **true**, the flow of the Process execution is controlled exclusively by a human user.

Executable

Boolean property defining whether a Process is intended for execution or not (if set to **false**, the process cannot be instantiated)

Globals

Set of global variables visible for other Processes to allow sharing of data between them

A.2. EVENTS MECHANISM

During process execution, the Process Engine makes sure that all the relevant tasks are executed according to the Process definition, the underlying work items, and other resources. However, a Process instance often needs to react to a particular event it was not directly requesting. Such events can be created and caught by the Intermediate Event elements (refer to the *User Guide*). Explicitly representing these events in a Process allows the author to specify how the particular Event should be handled.

An Event must specify the type of event it should handle. It can also define the name of a variable, which will store the data associated with the event. This allows subsequent elements in the Process to access the event data and take appropriate action based on this data.

An event can be signaled to a running instance of a process in a number of ways:

- Internal event: Any action inside a process (for example, the action of an action node, or an on-entry or on-exit action of some node) can signal the occurrence of an internal event to the surrounding Process instance.

Example A.3. Schema of the call sending an event to the Process instance

```
kcontext.getProcessInstance().signalEvent(type, eventData);
```

- External event: A process instance can be notified of an event from outside

Example A.4. Schema of the call notifying a Process instance about an external event

```
processInstance.signalEvent(type, eventData);
```

- External event using event correlation: Instead of notifying a Process instance directly, you can notify the entire Session and let the engine determine which Process instances might be interested in the event using event correlation. Event correlation is determined based on the event type. A Process instance that contains an Event element listening to external events of some type is notified whenever such an event occurs. To signal such an event to the process engine, write code such as:

Example A.5. Schema of the call notifying a Session about an external event

```
-
```

```
ksession.signalEvent(type, eventData);
```

Events can also be used to start a process. Whenever a Message Start Event defines an event trigger of a specific type, a new process instance starts every time that type of event is signalled to the process engine.

This mechanism is used for implementation of the Intermediate Events and can be used to define custom Events if necessary.

A.3. COLLABORATION MECHANISMS

Elements with execution semantics make use of general collaboration mechanisms. Different Elements allow you to access and use the mechanism in a different way; for example, there is a mechanism called signalling: a Signal is sent by a Throw Signal Intermediate Event Element and received by a Catch Signal Intermediate Event (two Elements with execution semantics make use of the same Signal mechanism).

Collaboration mechanism includes the following:

- **Messages:** Messages are used to communicate within the process and between process instances. Messages are implemented as signals which makes them scoped only for a given **KSession** instance.

For external system interaction send and receive task should be used with proper handler implementation.

- **Escalations:** mainly inter-process (between processes) signalling of escalation to trigger escalation handling
- **Errors:** mainly inter-process signalling of escalation to trigger escalation handling
- **Signals:** general, mainly inter-process instance communication

All the events are managed by the signaling mechanism. To distinguish individual objects of individual mechanism the signal use different signal codes or names.

A.3.1. Messages

“A Message represents the content of a communication between two Participants. In BPMN 2.0, a Message is a graphical decorator (it was a supporting element in BPMN 1.2). An ItemDefinition is used to specify the Message structure.^[1]”

Messages are objects that can be sent between Processes or Process elements, that participate in the respective communication; They are sent by the Message Intermediate Throw Events and Send Tasks, and can be consumed by the Message Start Events, Message Intermediate Catch Events, Message End Events, and Receive Tasks. One Message can be consumed by an arbitrary number of Processes and Process elements.

Attributes

Mandatory Attributes

Message

string with the message

A.3.2. Escalation

"An Escalation identifies a business situation that a Process might need to react to." [2]

The Escalation mechanism is intended for the handling of events that need the attention of someone of higher rank, or require additional handling.

Escalation is represented by an Escalation object that is propagated across the Process instances. It is produced by the Escalation Intermediate Throw Event or Escalation End Event, and can be consumed by exactly one Escalation Start Event or Escalation Intermediate Catch Event. Once produced, it is propagated within the current context and then further up the contexts until caught by an Escalation Start Event or Escalation Intermediate Catch Event, which is waiting for an Escalation with the particular Escalation Code. If an Escalation remains uncaught, the Process instance is ABORTED.

Attributes

Mandatory Attributes

Escalation Code

string with the escalation code

A.3.3. Signals

A Signal is an object, which can be propagated on execution within its parent Process instance as well as to other Processes. Every Signal can be consumed by multiple elements in multiple Process instance within the same Session.

Every Signal defines its Signal Reference, which is unique in the respective Session. A Signal can be generated by a Throw Signal Event and an action of an Activity. Once generated, it is propagated as an object through all the context in the parent Session of the Signal element.

NOTE

To trigger a Signal using API, you can use in your code the following:

- To instantiate a Process instance directly with a Signal, you can use the following API function:

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

The `eventType` parameter defines the Signal's Event Type, the `data` parameter defines the data accompanying the Signal, and `processInstanceId` defines the ID of the Process to be instantiated.

- To trigger a Signal from a script, that is, from a Script Task or using on-entry or on-exit actions of a node, you can use the following API function:

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

A.4. TRANSACTION MECHANISMS

A.4.1. Errors

An Error represents a critical problem in a Process execution and is indicated by the Error End Event. When a Process finishes with an Error End Event, the event produces an Error object with a particular Error Code that identifies the particular error end that occurred. The Error End Event represents an unsuccessful execution of the given Process or Activity. Once generated, it is propagated as an object within the current context and then further up the contexts until caught by the respective catching Error Intermediate Event or Error Start Event, which is waiting for an Error with the particular Error Code. If the Error is not caught and is propagated to the upper-most Process context, the Process instance becomes ABORTED.

Every Error defines its Error Code, which is unique in the respective Process.

Attributes

Error Code

Error Code defined as a String unique within the Process.

A.4.2. Compensation

Compensation is a mechanism that allows you to handle business exceptions that might occur in a Process or Sub-Process (Business transactions), that is to compensate for a failed transaction, where the transaction is presented by the Process or Sub-Process, and then continues the execution using the regular Flow path. Note, that compensation is triggered only after the execution of the transaction has finished and that either with a Compensation End Event or with a Cancel End Event.



NOTE

Consider implementing handling of business exceptions in the following cases:

- When an interaction with an external party or 3rd party system may fail or be faulty
- When you can not fully check the input data received by your Process (for example, a client's address information)
- When there are parts of your Process that are particularly dependent on one of the following:
 - Company policy or policy governing certain in-house procedures
 - Laws governing the business process (such as, age requirements)

If a business transaction finishes with a Compensation End Event, the Event produces a "request" for compensation handling. The compensation request is identified by ID and can be consumed only by the respective Compensation Intermediate Event placed on the boundary of the transaction Elements and Compensation Start Event. The Compensation Intermediate Event is connected with an Association Flow to the Activity that defines the compensation, such as a Sub-Process or Task. The execution flow either waits for the compensation activity to finish or resumes depending on the Wait for completion property set on the Compensation End Event of the business transaction that is being compensated.

If a business transaction contains an Event Sub-Process that starts with a Compensation Start Event, the Event Sub-Process is run as well if compensation is triggered.

The Activity the Compensation Intermediate Event points to, might be a Sub-Process. Note that the Sub-Process must start with the Compensation Start Event.

If running over a multi-instance Sub-Process, compensation mechanism of individual instances do not influence each other.

A.5. TIMING

Timing is a mechanism for scheduling actions and is used by Timer Intermediate and Timer Start events. It allows you to delay further execution of a process or task.



NOTE

A timer event can be triggered only after the transaction commits, while the timer countdown starts right after entering the node (the attached node in case of a boundary event). In other words, a timer event is only designed for those use cases where there is a wait state, such as a "User Task". If you want to be notified of the timeout of a synchronous operation without a wait state, a *boundary timer event is not suitable*.

The timing strategy is defined by the following timer properties:

Time Duration

defines the period for which the execution of the event is put on hold. The execution continues after the defined period has elapsed. The timer is applied only once.

Time Cycle

This defines the time between subsequent timer activations. If the period is **0**, the timer is triggered only once.

The value for these properties can be provided as either Cron or as an expression by defining the, *Time Cycle Language* property.

Cron

[#d][#h][#m][#s][#ms]

Example A.6. Timer period with literal values

1d 2h 3m 4s 5ms

The element will be executed after 1 day, 2 hours, 3 minutes, 4 seconds, and 5 milliseconds.

Any valid **ISO8601** date format that supports both one shot timers and repeatable timers can be used. Timers can be defined as date and time representation, time duration or repeating intervals. For example:

Date

2013-12-24T20:00:00.000+02:00 - fires exactly at Christmas Eve at 8PM

Duration

PT2S - fires 1 after 2 seconds

Repetable Intervals

R/PT1S - fires every second, no limit, alternatively R5/PT1S will fire 5 times every second

None

`#{expression}`

Example A.7. Timer period with expression

```
myVariable.getValue()
```

The element will be executed after time period returned by the call `myVariable.getValue()`.

A.6. PROCESS ELEMENTS

IMPORTANT

This chapter contains introduction to BPMN elements and their semantics. By no means does it aspire to be an exhaustive language specification. For details about BPMN refer to Business Process Model and Notation, Version 2.0: The BPMN 2.0 specification is an OMG specification that defines standards on how to graphically represent a business process, defines execution semantics for the elements along with an XML format of process definitions source.

The specification also includes details on choreographies and collaboration. Note that Red Hat JBoss BPM Suite focuses exclusive on executable processes and supports a significant subset of the BPMN elements including the most common types that can be used inside executable processes.

A Process Element is a node of the Process definition. The term covers the nodes with execution semantics as well as those without. Elements with execution semantics define the execution workflow of the Process, while Elements without execution semantics (Artifacts) allow users to provide notes and further information on the Process or any of its Elements so as to accommodate collaboration of multiple users with different roles, such as, business analyst, business manager, process designer.

All Elements with execution semantics define their generic properties.

Generic Process Element Properties

ID

ID defined as a String unique in the parent Knowledge Base

Name

Element display name

A.7. START EVENT

A Start Event is a modeling element that indicates where a particular Process workflow starts. Every Process must have at least one Start Event. Every Start Event has no incoming and exactly one outgoing Flow. When the parent Process is instantiated and started, the Start Event is executed and the node's outgoing Flow is taken.

Multiple Start Event types are supported:

- None Start Event
- Signal Start Event
- Timer Start Event
- Conditional Start Event
- Message Start Event
- Compensation Start Event

All but the None Start Event, must define a certain trigger type: when a Process instance is started, the trigger needs to be fulfilled before the outgoing flow can be taken. If no Start Event can be triggered, the Process is never instantiated.

A.7.1. Start Event types

A.7.1.1. None Start Event

The None Start Event is a Start Event without a trigger condition.

A Process or Sub-Process can contain at most one None Start Event, which is triggered on Process or Sub-Process start by default and the outgoing flow is taken immediately.

When used in a Sub-Process, the execution is transferred from the parent Process into the Sub-Process and the None Start Event is triggered (the token is taken from the parent Sub-Process Activity and the None Start Event of the Sub-Process generates a token).

A.7.1.2. Message Start Event

A Process or an Event Sub-Process can contain multiple Message Start Events, which are triggered when triggered by a particular Message. The Process instance with a Message Start Event only starts its execution from this event after it has received the respective Message: The Process is instantiated and its Message Start Event is executed immediately (its outgoing Flow is taken).

As a Message can be consumed by an arbitrary number of Processes and Process elements, including no Elements, one Message can trigger multiple Message Start Events and therefore instantiate multiple Processes.

Attributes

Message

ID of the expected Message object

A.7.1.3. Timer Start Event

The Timer Start Event is a Start Event with a Timing definition (for details on Timing see [Section A.5, “Timing”](#)).

A Process can contain at multiple Timer Start Events, which is triggered on Process start by default and then the Timing is applied.

When used in a Sub-Process, the execution is transferred from the parent Process into the Sub-Process and the Timer Start Event is triggered: the token is taken from the parent Sub-Process Activity and the Timer Start Event of the Sub-Process is triggered and waits for the Timing to be fulfilled. Once the time defined by the Timing definition has been reached, the outgoing Flow is taken.

Attributes

Timer

Timing definition

A.7.1.4. Escalation Start Event

The Escalation Start Event is a Start Event that is triggered by an Escalation with a particular Escalation code (see [Section A.3.2, “Escalation”](#)).

Process can contain multiple Escalation Start Events. The Process instance with an Escalation Start Event only starts its execution from this event after it has received the respective Escalation object: The Process is instantiated and its Escalation Start Event is executed immediately (its outgoing Flow is taken).

Attributes

Escalation Code

Expected Escalation Code

A.7.1.5. Conditional Start Event

The Conditional Start Event is a Start Event with a Boolean condition definition. The Process execution with such a Start Event continues only if the condition is evaluated to **true** after the Start Event has been instantiated.

The execution is triggered always when the condition is evaluated to **false** and then to **true**.

A Process can contain at multiple Conditional Start Events.

Attributes

Condition

Boolean condition

A.7.1.6. Error Start Event

An Error Start Event can be used to start a Process or Sub-Process. These can contain multiple Error Start Events, which are triggered when an Error object with a particular ErrorRef is received. The Error object can be produced by an Error End Event and signals an incorrect Process ending. The Process

instance with the Error Start Event starts execution after it has received the respective Error object so as to handle such incorrect ending: The Error Start Event is executed immediately (its outgoing Flow is taken).

Attributes

ErrorCode

code of the expected Error object

A.7.1.7. Compensation Start Event

A Compensation Start Event is used to start an Compensation Event Sub-Process when using a Sub-Process as the target Activity of a Compensation Intermediate Event.

A.7.1.8. Signal Start Event

The Signal Start Event is a Start Event that is triggered by a Signal with a particular Signal Code (see [Section A.3.3, “Signals”](#)).

Process can contain multiple Signal Start Events. The Signal Start Event only starts its execution within the Process instance after the instance has received the respective Signal: on Signal receiving, the Signal Start Event is executed immediately (its outgoing Flow is taken).

Attributes

SignalCode

Expected Signal Code

A.8. INTERMEDIATE EVENTS

A.8.1. Intermediate Events

“... the Intermediate Event indicates where something happens (an Event) somewhere between the start and end of a Process. It will affect the flow of the Process, but will not start or (directly) terminate the Process.^[3]”

Intermediate Event handles a particular situation that occurs during Process execution. The situation is the trigger of the Intermediate Event.

In a Process, Intermediate Events can be placed as follows:

in a Process workflow with one optional incoming and one outgoing Flow:

The event is executed as part of the workflow. If the Event has no incoming Flow, its execution is triggered always when the respective trigger occurs during the entire Process instance execution. If the Event has an incoming Flow it is executed as part of the Process workflow. Once triggered, the Event's outgoing Flow is taken only after the respective Event has occurred.

on an Activity boundary with one outgoing Flow:

If the Event occurs while the Activity is being executed, the Event triggers its execution to the outgoing Flow. One Activity may have multiple boundary Intermediate Events. Note that depending on the behavior you require from the Activity with the boundary Intermediate Event, you can use either of the following Intermediate Event type:

- interrupting: the Activity execution is interrupted and the execution of the Intermediate Event is triggered.
- non-interrupting: the Intermediate Event is triggered and the Activity execution continues.

Based on the type of Event cause the execution of the Intermediate Event (triggers), the following Intermediate Events are distinguished:

Timer Intermediate Event

delays the execution of the outgoing Flow.

Conditional Intermediate Event

is triggered when its condition evaluates to **true**.

Error Intermediate Event

is triggered by an Error object with the given Error Code.

Escalation Intermediate Event

has two subtypes: Catching Escalation Intermediate Event that is triggered by a Escalation and Throwing Escalation Intermediate Event that produces an Escalation when executed.

Signal Intermediate Event

has two subtypes: Catching Signal Intermediate Event that is triggered by a Signal and Throwing Signal Intermediate Event that produces a Signal when executed.

Message Intermediate Event

has two subtypes: Catching Message Intermediate Event that is triggered by a Message and Throwing Message Intermediate Event that produces a Message when executed.

A.8.2. Intermediate Event types

A.8.2.1. None Intermediate Event

None Intermediate Event is an abstract Intermediate Event and displays all possible Intermediate Event properties.

A.8.2.2. Timer Intermediate Event

A Timer Intermediate Event allows you to delay further workflow execution or to trigger the workflow execution periodically. It represents a timer that can trigger one or multiple times after a given period of time: always when triggered the timer condition (the defined time) is checked and once the time event occurs, the outgoing Flow is taken.

The Event defines the Timer delay and Timer period properties, that use the Timing mechanism as described in [Section A.5, “Timing”](#). When placed in the Process workflow, a Timer Intermediate Event has one incoming Flow and one outgoing Flow and its execution starts when the incoming Flow transfers to the Event. When placed on an Activity boundary, the execution is trigger at the same time as the Activity execution.

The timer is canceled if the timer element is canceled, for example, by completing or aborting the enclosing process instance).

Attributes

Message

ID of the expected Message object

Timer delay

Time delay before the Event triggers its outgoing Flow for the first time

Timer period

Period between two subsequent triggers

If set to **0**, the Event execution is not repeated.

A.8.2.3. Conditional Intermediate Event

A Conditional Intermediate Event is an Intermediate Event with a boolean condition as its trigger. The Event triggers further workflow execution when the condition evaluates to **true** and its outgoing Flow is taken.

The Event must define its boolean Conditional. When placed in the Process workflow, a Conditional Intermediate Event has one incoming Flow and one outgoing Flow and its execution starts when the incoming Flow transfers to the Event. When placed on an Activity boundary, the execution is triggered at the same time as the Activity execution. Note, that if the Event is non-interrupting, the Event triggers continuously while the condition is **true**.

Attributes

Condition

Boolean condition that must be evaluated to true for the execution to continue

A.8.2.4. Message Intermediate Event

A Message Intermediate Event is an Intermediate Event that allows you to produce or consume a Message object. Depending on the action the event element is to perform, you need to use either of the following:

- **Throwing Message Intermediate Event** produces a Message object based on the defined properties
- **Catching Message Intermediate Event** listens for a Message object with the defined properties

A.8.2.5. Compensation Intermediate Event

A Compensation Intermediate Event is a boundary event that is attached to a Activity in a transaction Sub-Process that may finish with a Compensation End Event or a Cancel End Event. The Compensation Intermediate Event must have one outgoing Association Flow that connects to an Activity that defines the compensation action needed to compensate for the action performed by the Activity.

On runtime, if the transaction Sub-Process finishes with the Compensation End Event, the Activity associated with the boundary Compensation Intermediate Event is executed and the execution continues with the respective Flow leaving the transaction Sub-Process.

A.8.2.6. Message Intermediate Event types

A.8.2.6.1. Throwing Message Intermediate Event

When reached on execution, a Throwing Message Intermediate Event produces a Message and the execution continues to its outgoing Flow.

Attributes

CancelActivity

if the Event is placed on the boundary of an Activity and the Cancel Activity property is set to **true**, the Activity execution is cancelled immediately when the Event receives its Escalation object.

MessageRef

ID of the produced Message object

A.8.2.6.2. Catching Message Intermediate Event

When reached on execution, a Catching Message Intermediate Event awaits a Message defined in its properties. Once the Message is received, the Event triggers execution of its outgoing Flow.

Attributes

MessageRef

ID of the expected Message object

A.8.2.7. Escalation Intermediate Event

An Escalation Intermediate Event is an Intermediate Event that allows you to produce or consume an Escalation object. Depending on the action the event element is to perform, you need to use either of the following:

- **Throwing Escalation Intermediate Event** produces an Escalation object based on the defined properties
- **Catching Escalation Intermediate Event** listens for an Escalation object with the defined properties

A.8.2.8. Escalation Intermediate Event types

A.8.2.8.1. Throwing Escalation Intermediate Event

When reached on execution, a Throwing Escalation Intermediate Event produces an Escalation object and the execution continues to its outgoing Flow.

Attributes

EscalationCode

ID of the produced Escalation object

A.8.2.8.2. Catching Escalation Intermediate Event

When reached on execution, a Catching Escalation Intermediate Event awaits an Escalation object defined in its properties. Once the object is received, the Event triggers execution of its outgoing Flow.

Attributes

EscalationCode

code of the expected Escalation object

CancelActivity

if the Event is placed on the boundary of an Activity and the Cancel Activity property is set to **true**, the Activity execution is cancelled immediately when the Event receives its Escalation object.

A.8.2.9. Error Intermediate Event

An Error Intermediate Event is an Intermediate Event that can be used only on an Activity boundary. It allows the Process to react to an Error End Event in the respective Activity. The Activity must not be atomic. When the Activity finishes with an Error End Event that produces an Error with the respective ErrorCode, the Error Intermediate Event catches the Error object and execution continues to the outgoing Flow of the Error Intermediate Event.

Attributes

ErrorRef

reference number of the Error object the Event is listening for

A.8.2.10. Error Intermediate Event types

A.8.2.10.1. Throwing Error Intermediate Event

When reached on execution, a Throwing Error Intermediate Event produces an Error object and the execution continues to its outgoing Flow.

Attributes

ErrorRef

reference number of the produced Error object

A.8.2.10.2. Catching Error Intermediate Event

When reached on execution, a Catching Error Intermediate Event awaits an Error object defined in its properties. Once the object is received, the Event triggers execution of its outgoing Flow.

Attributes

ErrorRef

reference number of the expected Error object

CancelActivity

if the Event is place on the boundary of an Activity and the Cancel Activity property is set to **true**, the Activity execution is cancelled immediately when the Event receives its Escalation object.

A.8.2.11. Signal Intermediate Event

A Signal Intermediate Event is an Intermediate Event that allows you to produce or consume a Signal object. Depending on the action the event element is to perform, you need to use either of the following:

- **Throwing Signal Intermediate Event** produces a Signal object based on the defined properties
- **Catching Signal Intermediate Event** listens for a Signal object with the defined properties

A.8.2.12. Signal Intermediate Event types**A.8.2.12.1. Throwing Signal Intermediate Event**

When reached on execution, a Throwing Signal Intermediate Event produces a Signal object and the execution continues to its outgoing Flow.

Attributes**SignalRef**

The Signal code that is to be sent or consumed

CancelActivity

if the Event is place on the boundary of an Activity and the Cancel Activity property is set to **true**, the Activity execution is cancelled immediately when the Event receives its Escalation object.

A.8.2.12.2. Catching Signal Intermediate Event

When reached on execution, a Catching Signal Intermediate Event awaits a Signal object defined in its properties. Once the object is received, the Event triggers execution of its outgoing Flow.

Attributes**SignalRef**

reference code of the expected Signal object

A.9. END EVENTS

An End Event is a node that ends a particular workflow. It has one or more incoming Sequence Flows and no outgoing Flow.

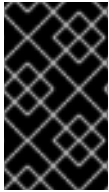
A Process must contain at least one End Event.

During runtime, an End Event finishes the Process workflow. It might finish only the workflow that reached the End Event or all workflows in the Process instance depending on its type.

A.9.1. End Event types

A.9.1.1. Simple End Event

The Simple End Event finishes the incoming workflow (consumes the incoming token). Any other running workflows in the Process or Sub-Process remain uninfluenced.



IMPORTANT

In JBoss BPM Suite, the Simple End Event has the **Terminate** property in its Property tab. This is a boolean property, which turns a Simple End Event into a Terminate End Event when set to **true**.

A.9.1.2. Message End Event

When a Flow enters a Message End Event, the Flow finishes and the Event produces a Message as defined in its properties.

A.9.1.3. Escalation End Event

The Escalation End Event finishes the incoming workflow (consumes the incoming token) and produces an Escalation signal as defined in its properties, triggering the escalation process.

A.9.1.4. Terminate End Event

The Terminate End Event finishes all execution flows in the given Process instance, that is, at the moment, one execution flow enters the end event, all execution flows in the Process instance are terminated and the Process instance becomes completed: if there are activities being executed, they are instantly cancelled. If a Terminate End Event is reached in a Sub-Process, the entire Process instance is terminated.



NOTE

If you are designing your Process in the Process Designer and you want to create a Terminate End Event, create a Simple End Event and define the **Terminate** property in the Property tab of the Simple End Event as **true**.

A.9.1.5. Throwing Error End Event

The Throwing Error End Event finishes the incoming workflow (consumes the incoming token). Any other running workflows in the Process or Sub-Process remain uninfluenced.

Attributes

ErrorRef

reference code of the produced Error object

A.9.1.6. Cancel End Event

If a Process or Sub-Process finishes with a Cancel End Event, any compensations defined for the namespace are executed, and the Process or Sub-Process finishes as CANCELLED.

A.9.1.7. Compensation End Event

A Compensation End Event is used to finish a transaction Sub-Process and trigger the compensation defined by the Compensation Intermediate Event attached to the boundary of the Sub-Process activities.

A.9.1.8. Signal End Event

A Throwing Signal End Event is used to finish a Process or Sub-Process flow. When the execution flow enters the element, the execution flow finishes and a Signal identified by its SignalRef property value.

A.10. GATEWAYS

A.10.1. Gateways

“Gateways are used to control how Sequence Flows interact as they converge and diverge within a Process.^[4]”

Gateways are used to create or synchronize branches in the workflow using a set of conditions which is called the gating mechanism. Gateways are either converging (multiple Flows into one Flow) or diverging (One Flow into multiple Flows).

One Gateway cannot have multiple incoming *and* multiple outgoing Flows.

Depending on the gating mechanism you want to apply, you can use the following types of gateways:

- Parallel (AND): in a converging gateway, waits for all incoming Flows. In a diverging gateway, takes all outgoing Flows simultaneously;
- Inclusive (OR): in a converging gateway, waits for all incoming Flows whose condition evaluates to true. In a diverging gateway takes all outgoing Flows whose condition evaluates to true;
- Exclusive (XOR): in a converging gateway, only the first incoming Flow whose condition evaluates to true is chosen. In a diverging gateway only one outgoing Flow is chosen.
- Event-based: used only in diverging gateways for reacting to events. See [Section A.10.2.1, “Event-based Gateway”](#)
- Data-based Exclusive: used in both diverging and converging gateways to make decisions based on available data. See [Section A.10.2.4, “Data-based Exclusive Gateway”](#)

A.10.2. Gateway types

A.10.2.1. Event-based Gateway

“The Event-Based Gateway has pass-through semantics for a set of incoming branches (merging behavior). Exactly one of the outgoing branches is activated afterwards (branching behavior), depending on which of Events of the Gateway configuration is first triggered. ^[5]”

The Gateway is only diverging and allows you to react to possible Events as opposed to the Data-based Exclusive Gateway, which reacts to the process data. It is the Event that actually occurs that decides which outgoing Flow is taken. As it provides the mechanism to react to exactly one of the possible Events, it is exclusive, that is, only one outgoing Flow is taken.

The Gateway might act as a Start Event, where the process is instantiated only if one the Intermediate Events connected to the Event-Based Gateway occurs.

A.10.2.2. Parallel Gateway

“A Parallel Gateway is used to synchronize (combine) parallel flows and to create parallel flows.^[6]”

Diverging

Once the incoming Flow is taken, all outgoing Flows are taken simultaneously.

Converging

The Gateway waits until all incoming Flows have entered and only then triggers the outgoing Flow.

A.10.2.3. Inclusive Gateway

Diverging

Once the incoming Flow is taken, all outgoing Flows whose condition evaluates to true are taken. Connections with lower priority numbers are triggered before triggering higher priority ones; priorities are evaluated but the BPMN2 specification doesn't guarantee this. So for portability reasons it is recommended that you do not depend on this.



IMPORTANT

Make sure that at least one of the outgoing Flow evaluates to true at runtime; otherwise, the process instance terminates with a runtime exception.

Converging

The Gateway merges all incoming Flows previously created by a diverging Inclusive Gateway; that is, it serves as a synchronizing entry point for the Inclusive Gateway branches.

Attributes

Default gate

The outgoing Flow taken by default if no other Flow can be taken

A.10.2.4. Data-based Exclusive Gateway

Diverging

The Gateway triggers exactly one outgoing Flow: the Flow with the constraint evaluated to true and the *lowest* Priority is taken. After evaluating the constraints that are linked to the outgoing Flows: the constraint with the lowest priority number that evaluates to true is selected.



IMPORTANT

Make sure that at least one of the outgoing Flows evaluates to true at runtime: if no Flow can be taken, the execution returns a runtime exception.

Converging

The Gateway allows a workflow branch to continue to its outgoing Flow as soon as it reaches the Gateway; that is, whenever one of the incoming Flows triggers the Gateway, the workflow is sent to the

outgoing Flow of the Gateway; if it is triggered from more than one incoming connection, it triggers the next node for each trigger.

Attributes

Default gate

The outgoing Flow taken by default if no other Flow can be taken

A.11. ACTIVITIES, TASKS AND SUB-PROCESSES

A.11.1. Activity

"An Activity is work that is performed within a Business Process." [7]

This is opposed to the execution semantics of other elements that defined the Process logic.

An Activity can be further specified as a Sub-Process or a Task: while Task is atomic, that is represents a single piece of work, a Sub-Process is compound, that is it can be broken down into multiple Process elements.

An Activity in jBPM expects one incoming and one outgoing flow. If you want to design an activity with multiple incoming and multiple outgoing flows, set the value of the system property `jbpm.enable.multi.con` to true.

Activities have the basic properties just like any other Process element (ID and Name). Note that Activities (all of their subtypes, that is, Tasks, Sub-Process) have additional properties specific for the given Activity or Task type.

A.11.2. Activity mechanisms

A.11.2.1. Multiple instances

Activities can be run in multiple instances on execution. Individual instances are run in a sequential manner. The instances are run based on a collection of elements: for every element in the collection, a new Activity instance is created..

Every Activity has therefore the Collection Expression attribute that defines the collection with elements to iterate through.

A.11.2.2. Activity types

A.11.2.2.1. Call Activity

"A Call Activity identifies a point in the Process where a global Process or a Global Task is used. The Call Activity acts as a 'wrapper' for the invocation of a global Process or Global Task within the execution. The activation of a call Activity results in the transfer of control to the called global Process or Global Task. [8]"

A Call Activity, previously Reusable Sub-Process, represents an invocation of a Process from within a Process. The Activity must have one incoming and one outgoing Flow.

When the execution flow reaches the Activity, an instance of the Process with the ID defined by the Activity is created.

Attributes

Called Element

ID of the Process to be called and instantiated by the Activity

A.11.3. Tasks

A.11.3.1. Task types

A Task is the smallest unit of work in a Process flow and to help identify the various types of Tasks that can be performed, Red Hat JBoss BPM Suite uses the BPMN guidelines to separate them based on the types of inherent behavior that the Tasks might represent.

A Task that doesn't serve a direct defined purpose is either called the **None** Task or the **Abstract** Task (deprecated).

The different types of tasks available in JBoss BPM Suite are listed here, except for the **User** Task.

We define the User Task in another section (See [Section A.11.5, "User Task"](#)).

A.11.3.2. Generic Task

"Abstract Task: Upon activation, the Abstract Task completes. This is a conceptual model only; an Abstract Task is never actually executed by an IT system." [9]

A.11.3.3. Send Task

"Send Task: Upon activation, the data in the associated Message is assigned from the data in the Data Input of the Send Task. The Message is sent and the Send Task completes." [10]

Attributes

MessageRef

the MessageRef of the generated Message

A.11.3.4. Receive Task

"Upon activation, the Receive Task begins waiting for the associated Message. When the Message arrives, the data in the Data Output of the Receive Task is assigned from the data in the Message, and Receive Task completes." [11]

Attributes

MessageRef

the associated Message

A.11.3.5. Manual Task

"Upon activation, the Manual Task is distributed to the assigned person or group of people. When the work has been done, the Manual Task completes. This is a conceptual model only; a Manual Task is never actually executed by an IT system." [12]

A.11.3.6. Service Task

A Service Task is used with the built-in **ServiceTaskHandler** to invoke Java methods or Web Services.

Attributes

Implementation

The underlying technology that will be used to implement this task. You can use **unspecified** or **WebService** where **WebService** is the default value.

OperationRef

This attribute specifies the operation that is invoked by the Service Task. (typically method of Java class or method of WebService).

A.11.3.7. Business Rule Task

"A Business Rule Task provides a mechanism for the Process to provide input to a Business Rules Engine and to get the output of calculations that the Business Rules Engine might provide. [13]"

The Task defines a set of rules that need to be evaluated and fired on Task execution. Any rule defined as part of the ruleflow group in a rule resource is fired.

When a Rule Task is reached in the Process, the engine starts executing the rules with the defined ruleflow group. When there are no more active rules with the ruleflow group, the execution continues to the next Element. During the ruleflow group execution, new activations belonging to the active ruleflow group can be added to the Agenda as these are changed by the other rules. Note that the Process continues immediately to the next Element if there are no active rules of the ruleflow group.

If the ruleflow group was already active, the ruleflow group remains active and the execution continues if all active rules of the ruleflow group have been completed.

Attributes

RuleFlow Group

the name of the rule flow group that includes the set of rules to be evaluated by the Task

A.11.3.8. Script Task

A Script Task represents a script that should be executed during the Process execution.

The associated Script can access any variables and globals.

When using a Script Task follow these rules:

- Avoid low-level implementation details in the Process: A Script Task could be used to manipulate variables but other concepts like a Service Task should be your first choice when modeling more complex behavior in a higher-level manner.
- The script should be executed immediately; if there is the possibility that the execution could take some time, use an asynchronous Service Task.
- Avoid contacting external services through a Script Task: it would be interacting with external services without the knowledge of the engine, which can be problematic. 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.

When a Script Task is reached during execution, the script is performer and the outgoing Flow is taken.

Attributes

Script

script to be executed

ScriptLanguage

language the script is defined in (currently supported languages are Java and MVEL)

A.11.4. Sub-Process

“A Sub-Process is an Activity whose internal details have been modeled using Activities, Gateways, Events, and Sequence Flows. A Sub-Process is a graphical object within a Process, but it also can be ‘opened up’ to show a lower-level Process. [14]”

Therefore, a Sub-Process can be understood as a compound Activity or a *Process in a Process*. When reached during execution, the Element context is instantiated and the encapsulated process triggered. Note that, if you use a Terminating End Event inside a Sub-Process, the entire Process instance that contains the Sub-Process is terminated, not just the Sub-Process. A Sub-Process, just like a Process, ends when there are no more active Elements in it.

The following Sub-Process types are supported:

- Ad-Hoc Sub-Process: Sub-Process with no strict Element execution order
- Embedded Sub-Process: a "real" Sub-Process that is a part of the Parent Process execution and shares its data
- Reusable Sub-Process: a Sub-Process that is independent from its parent Process
- Event Sub-Process: a Sub-Process that is only triggered on a start event or a timer.

Note that any Sub-Process type can be also a Multi-Instance Sub-Process.

A.11.4.1. Embedded Sub-Process

An Embedded Sub-Process is a Sub-Process that encapsulates a part of the process.

It must contain a Start Event and at least one End Event. Note that the Element allows you to define local Sub-Process variables, that are accessible to all Elements inside this container.

A.11.4.2. AdHoc Sub-Process

“An Ad-Hoc Sub-Process is a specialized type of Sub-Process that is a group of Activities that have no REQUIRED sequence relationships. A set of Activities can be defined for the Process, but the sequence and number of performances for the Activities is determined by the performers of the Activities. [15]”

“An Ad-Hoc Sub-Process or Process contains a number of embedded inner Activities and is intended to be executed with a more flexible ordering compared to the typical routing of Processes. Unlike regular Processes, it does not contain a complete, structured BPMN diagram descriptionâi.e., from Start Event to End Event. Instead the Ad-Hoc Sub-Process contains only Activities, Sequence Flows, Gateways, and Intermediate Events. An Ad-Hoc Sub-Process MAY also contain Data Objects and Data Associations. The Activities within the Ad-Hoc Sub- Process are not REQUIRED to have incoming and outgoing Sequence Flows. However, it is possible to specify Sequence Flows between some of the contained Activities. When used, Sequence Flows will provide the same ordering constraints as in a regular Process. To have any meaning, Intermediate Events will have outgoing Sequence Flows and they can be triggered multiple times while the Ad-Hoc Sub-Process is active.[16]”

The Elements of an AdHoc Sub-Process are executed in parallel.

Attributes

AdHocCompletionCondition

the condition that once met the execution is considered successful and finishes

AdHocCancelRemainingInstances

if set to **true**, once the AdHocCompletionCondition is met, execution of any Elements is immediately cancelled

A.11.4.3. Multi-instance Sub-Process

A Multiple Instance Sub-Process is a Sub-Process that is instantiated/run multiple times when its execution is triggered. The instances are created in a sequential manner: a new Sub-Process instance is created only after the previous instance has finished.

A Multiple instance Sub-Process has one incoming Connection and one outgoing Connection.

Attributes

Collection expression

Variable that represents the collection of elements that are to be iterated over (The variable must be an array or be of the java.util.Collection type.)

If the collection expression evaluates to null or an empty collection, the Multi-Instances Sub-Process is completed immediately and the outgoing flow is taken.

Variable Name

Variable that will store the collection element used in the currently running iteration

A.11.4.4. Event Sub-Process

An Event Sub-Process becomes active when its start event gets triggered. It can interrupt the parent process context or run in parallel to it.

With no outgoing or incoming connections, only an event or a timer can trigger these Sub-Processes. These Sub-Processes are not part of the regular control flow. Although self-contained, they are executed in the context of the bounding Sub-Process.

You would use these Sub-Processes within a process flow to handle events that happen external to the main process flow. For example, while booking a flight, two events may occur: (interrupting) cancel booking, or (non-interrupting) check booking status. Both these events can be modeled using the Event Sub-Process.

A.11.5. User Task

"A User Task is a typical 'workflow' Task where a human performer performs the Task with the assistance of a software application and is scheduled through a task list manager of some sort." [17]

The User Task cannot be performed automatically by the system and therefore requires an intervention of a human user, the Actor. Also, it is relatively atomic as opposed to such non-atomic Elements as Sub-Processes.

On execution, the User Task element is instantiated as a User Task that appears in the list of Tasks of one or multiple Actors.

If a User Task element defines a **GroupID**, it is displayed in Task lists of all users that are members of the group: any of the users can claim the Task. Once claimed, the Task disappears from the Task lists of the other users.

Note that User Task is implemented as a domain-specific Tasks and serve as base for your custom Task (refer to [Section 4.14.1, "Work item definition"](#)).

Attributes

Actors

comma-separated list of users who are entitled to perform the generated User Task

Comment

A comment associated with this User Task. The JBoss BPM Suite Engine does not use this field but business users can enter extra information about this task.

Content

The data associated with this task. This attribute does not affect TaskService's behavior.

CreatedBy

name of the user or ID of the Process that created the task

GroupID

comma-separated list of groups who are entitled to perform the generated User Task

Locale

locale the Element is defined for. This was intended to support internationalization (i18n), but this property is not used by the JBoss BPM Suite engine at the moment.

Notifications

Definition of notification applied on the Human Task (refer to [Section A.11.5.3, “Notification”](#))

Priority

Integer value defining the User Task priority (the value influences the User Task ordering in the user Task list and the simulation outcome)

Reassignment

Definition of escalation applied on the Human Task (refer to [Section A.11.5.2, “Reassignment”](#))

ScriptLanguage

One of Java or MVEL.

Skippable

Boolean value that defines if the User Task can be skipped (if **true**, the actor of the User Task can decide not to complete it and the User Task is never executed)

Task Name

Name of the User Task generated on runtime (displayed in the Task List of Business Central)

Note that any other displayed attributes are used by features not restricted to the User Task element and are described in the chapters dealing with the particular mechanism.

A.11.5.1. User Task lifecycle

When a User Task element is encountered during Process execution, a User Task instance is created. The User Task instance execution is preformed by the User Task service of the Task Execution Engine (refer to *Red Hat JBoss BPM Suite Administration and Configuration Guide*). The Process instance leaves the User Task element and continues the execution only when the associated User Task has been completed or aborted.

When the Process instance enters the User Task element, the User Task is the **Created** stage. This is usually a transient state and the User Task enters the **Ready** state immediately: the User Task appears in the Task Lists of all actors that are allowed to execute the task. As soon as one of the actors claims the User Task to indicate they are executing it, the User Task becomes **Reserved**. If a User Task has only one potential actor, it is automatically assigned to that actor upon creation. When the user who has claimed the User Task starts the execution, the User Task status changes to **InProgress**. On completion, the status changes to **Completed** or **Failed** depending on the execution outcome.

Note that the User Task lifecycle can include other statuses if the User Task is reassigned (delegated or escalated), revoked, suspended, stopped, or skipped. For further details, on the User Task lifecycle refer to the [Web Services Human Task](#) specification.

A.11.5.2. Reassignment

The reassignment mechanism is the mechanism implementing the escalation and delegation capabilities for User Tasks, that is, automatic reassignment of a User Task to another actor or group after a User Task has remain inactive for a certain amount of time.

Reassignment can be defined to take place either if the given User Task is for a given time in either of the following states:

- not started: **READY** or **RESERVED**
- not completed: **IN_PROGRESS**

When the conditions defined in the reassignment are met, the User Task is reassigned to the users or groups defined in the reassignment. If the actual owner is included in the new users or groups definition, the User Task is reset and reset to the READY state.

Reassignment is defined in the Reassignment property of User Task elements. The property can take an arbitrary number of reassignment definitions with the following parameters:

- **Users**: comma-separated list of user IDs that are reassigned to the task on escalation (Strings or expressions #{user-id})
- **Groups**: comma separated list of group IDs that are reassigned to the task on escalation (Strings or expressions #{group-id})
- **Expires At**: time definition when escalation is triggered (String values and expressions #{expiresAt}; for information on time format, refer to [Section A.5, “Timing”](#))
- **Type**: state the task needs to be in at the given Expires At time so that the escalation is triggered.

A.11.5.3. Notification

The Notification mechanism provides the capability to send an e-mail notification if a User Task is at the given time in one of the following states:

- not started: **READY** or **RESERVED**
- not completed: **IN_PROGRESS**

Notification is defined in the Notification property of User Task elements. The property can take an arbitrary number of notification definitions with the following parameters:

- **Type**: state the User Task needs to be in at the given Expires At time so that the notification is triggered
- **Expires At**: time definition when notification is triggered (String values and expressions #{expiresAt}; for information on time format, refer to [Section A.5, “Timing”](#))
- **From**: user or group ID of users used in the From field of the email notification message (Strings or expressions)
- **To Users**: comma-separated list of user IDs the notification is to be sent to (Strings or expressions #{user-id})
- **To Groups**: comma separated list of group IDs the notification is to be sent to (Strings or expressions #{group-id})
- **Reply To**: user or group ID that receives any replies to the notification (Strings or expressions #{group-id})
- **Subject**: subject of the email notification (Strings or expressions)
- **Body**: body of the email notification (Strings and expression)

Available variables

Notification can reference process variables (**`#{processVariable}`**) and Task variables (**`${taskVariable}`**). Note that process variables are resolved at Task creation and Task variables are resolved at notification time. In addition to custom Task variables, the notification mechanism can make use of the following local Task variables:

- **taskId**: internal ID of the User Task instance
- **processInstanceId**: internal ID of Task's parent Process instance
- **workItemId**: internal ID of a work item that created the User Task
- **processSessionId**: knowledge session ID of the parent Process instance
- **owners**: list of users and groups that are potential owners of the User Task
- **doc**: map that contains regular task variables

Example A.8. Body of notification with variables

```
<html>
  <body>
    <b>${owners[0].id} you have been assigned to a task (task-id
    ${taskId})</b><br>
    You can access it in your task
    <a href="http://localhost:8080/jbpm-
    console/app.html#errai_ToolSet_Tasks;Group_Tasks.3">inbox</a><br>
    Important technical information that can be of use when working on
    it<br>
    - process instance id - ${processInstanceId}<br>
    - work item id - ${workItemId}<br>

    <hr/>

    Here are some task variables available
    <ul>
      <li>ActorId = ${doc['ActorId']}</li>
      <li>GroupId = ${doc['GroupId']}</li>
      <li>Comment = ${doc['Comment']}</li>
    </ul>
    <hr/>
    Here are all potential owners for this task
    <ul>
      $foreach{orgEntity : owners}
        <li>Potential owner = ${orgEntity.id}</li>
      $end{}
    </ul>

    <i>Regards from jBPM team</i>
  </body>
</html>
```

A.12. CONNECTING OBJECTS

A.12.1. Connecting Objects

Connecting object connect two elements. There are two main types of Connecting object:

- Sequence Flow, which connect Flow elements of a Process and define the flow of the execution (transport the token from one element to another)
- Association Flow, which connect any Process elements but have no execution semantics

A.12.2. Connecting Objects types

A.12.2.1. Sequence Flow

A Sequence Flow represents the transition between two Flow elements: it establishes an oriented relationship between Activities, Events, and Gateways and defines their execution order.

Properties

Condition Expression

A condition that needs to be true to allow the workflow to take the Sequence Flow

If a Sequence Flow has a Gateway element as its source, you need to define a Conditional Expression, which is evaluated before the Sequence Flow is taken. If false, the workflow attempts to switch to another Sequence Flow. If true, the Sequence Flow is taken.

When defining the condition in Java, make sure to return a boolean value:

```
return <expression resolving to boolean>;
```

Condition Expression Language

You can use either Java or Drools to define the Condition Expression.



NOTE

When defining a Condition Expression, make sure to call process and global variables. You can also call the **kcontext** variable, which holds the Process instance information.

A.13. SWIMLANES

Swimlanes are a process element to visually group tasks related to one group or user. For example, you can create a Marketing task to group all User Tasks related to marketing activities into one Lane.

A.13.1. Lanes

"A Lane is a sub-partition within a Process (often within a Pool)... " [18]

A Lane allows you to group some of the Process elements and define their common parameters. Note that a Lane may contain another Lane.

To add a new Lane, open up the **Swimlanes** menu item in the Object Library to show the Lane artifact. Drag and drop the Lane artifact to your Process Model. This artifact is a box in which you can add your User Tasks.

Lanes should be given distinguishing names and background colors to fully separate them into functional groups. You can do so by selecting a lane and opening up the Properties panel.

At runtime, Lanes auto-claim/assign task to user who has done another task of that Lane within the same process instance. This user must be eligible for claiming a task, that is, this user must be a potential owner. If a User Task doesn't have an actor or group assigned it marks the task as having no potential owners (and therefore, at runtime, the process will just stop).

For example, let's say there are two User Tasks (UT1 and UT2) located in the same Lane. UT1 and UT2 have group field set to the **analyst** value. When the Process is started, and UT1 is claimed/started/completed by an **analyst** user, UT2 gets claimed and assigned to the user who completed UT1. On the other hand, if only UT1 had the **analyst** group assigned, and UT2 had no user or group assignments, the process would stop after UT1 had been completed.

A.14. ARTIFACTS

A.14.1. Artifacts

Artifacts are considered any object depicted in the BPMN diagram that are not part of the Process workflow: they have no incoming or outgoing Flow objects.

The purpose of Artifacts is to provide additional information needed to understand the diagram.

A.14.2. Data Objects

Data Objects are visualizations of Process or Sub-Process variables. Note that not every Process or Sub-Process variable must be depicted as a Data Object in the BPMN diagram.

Also note, that Data Objects have the visualization properties and the variable properties.

[1] Business Process Model and Notation (BPMN). Version 2.0, OMG Document Number: formal/2011-01-03
<http://www.omg.org/spec/BPMN/2.0>

[2] *Business Process Model and Notation (BPMN)*. Version 2.0, OMG Document Number: formal/2011-01-03
<http://www.omg.org/spec/BPMN/2.0>

[3] Business Process Model and Notation (BPMN). Version 2.0, OMG Document Number: formal/2011-01-03
<http://www.omg.org/spec/BPMN/2.0>

[4] Business Process Model and Notation (BPMN). Version 2.0, OMG Document Number: formal/2011-01-03
<http://www.omg.org/spec/BPMN/2.0>

[5] Business Process Model and Notation (BPMN). Version 2.0, OMG Document Number: formal/2011-01-03
<http://www.omg.org/spec/BPMN/2.0>

[6] Business Process Model and Notation (BPMN). Version 2.0, OMG Document Number: formal/2011-01-03
<http://www.omg.org/spec/BPMN/2.0>

[7] *Business Process Model and Notation (BPMN)*. Version 2.0, OMG Document Number: formal/2011-01-03
<http://www.omg.org/spec/BPMN/2.0>

- [8] Business Process Model and Notation (BPMN). Version 2.0, OMG Document Number: formal/2011-01-03
<http://www.omg.org/spec/BPMN/2.0>
- [9] *Business Process Model and Notation (BPMN)*. Version 2.0, OMG Document Number: formal/2011-01-03
<http://www.omg.org/spec/BPMN/2.0>
- [10] *Business Process Model and Notation (BPMN)*. Version 2.0, OMG Document Number: formal/2011-01-03
<http://www.omg.org/spec/BPMN/2.0>
- [11] *Business Process Model and Notation (BPMN)*. Version 2.0, OMG Document Number: formal/2011-01-03
<http://www.omg.org/spec/BPMN/2.0>
- [12] *Business Process Model and Notation (BPMN)*. Version 2.0, OMG Document Number: formal/2011-01-03
<http://www.omg.org/spec/BPMN/2.0>
- [13] Business Process Model and Notation (BPMN). Version 2.0, OMG Document Number: formal/2011-01-03
<http://www.omg.org/spec/BPMN/2.0>
- [14] Business Process Model and Notation (BPMN). Version 2.0, OMG Document Number: formal/2011-01-03
<http://www.omg.org/spec/BPMN/2.0>
- [15] Business Process Model and Notation (BPMN). Version 2.0, OMG Document Number: formal/2011-01-03
<http://www.omg.org/spec/BPMN/2.0>
- [16] Business Process Model and Notation (BPMN). Version 2.0, OMG Document Number: formal/2011-01-03
<http://www.omg.org/spec/BPMN/2.0>
- [17] *Business Process Model and Notation (BPMN)*. Version 2.0, OMG Document Number: formal/2011-01-03
<http://www.omg.org/spec/BPMN/2.0>
- [18] *Business Process Model and Notation (BPMN)*. Version 2.0, OMG Document Number: formal/2011-01-03
<http://www.omg.org/spec/BPMN/2.0>

APPENDIX B. SERVICE TASKS

For the convenience of the user, Red Hat JBoss BPM Suite comes with the following predefined types of Service Tasks: Log Task for logging a message to the server standard output (refer to [Section B.1, “Log Task”](#)), Email Task for sending emails via the setup mail server (refer to [Section B.2, “Email Task”](#)), REST Task for sending REST calls (refer to [Section B.3, “REST Task”](#)), and WS Task for invoking web services as a webservice client (refer to [Section B.4, “WS Task”](#)). Note that since the Tasks extend the Service Task, their attributes are implemented as Assignments, and DataInputSet and DataOutputSet, not as separate properties.

If other Task types are required, implement your Task as instructed in [Section 4.14, “Domain-specific Tasks”](#).



NOTE

Service Task is a Task that uses a service, such as a mail service, web service, or another system (for further information, refer to the BPMN 2.0 specification).

B.1. LOG TASK

The Log Task is a special type of Service Task that logs an entry in the server log.

Input attributes

Message

the log message text

B.2. EMAIL TASK

The Email Task is a special type of the Service Task that sends an email based on the Task properties.

Input attributes

To

the recipient of the email

From

the sender of the email

Subject

the subject of the email

Body

the body of the email

B.3. REST TASK

The REST Task is a special type of the Service Task that performs REST calls and outputs the response as an Object.

Input attributes

Method

the REST method of the call (such as, **GET**, **POST**, etc.)

ConnectTimeout

the call timeout

Username

the user name to be used to perform the call

Password

the user password

ReadTimeout

timeout on response receiving

Url

target URL including the request body

Output attributes

Result

string with the result of the call

B.4. WS TASK

The WS Task is a special type Service Task that serves as a web service client with the web service response stored in the Result String.

Input attributes

Parameter

The object or array to be sent for the operation

Mode

One of SYNC, ASYNC or ONEWAY.

Interface

The name of a service. For example: **Weather**

Namespace

namespace of the web service, such as **http://ws.cdyne.com/WeatherWS//**

URL

the web service URL, such as **http://ws.cdyne.com/**

Operation

the actual method name to call

Output attributes

Result

object with the result

APPENDIX C. SIMULATION DATA

C.1. PROCESS

Simulation attributes

Base currency

currency to be used for simulation

Base time unit

time unit to apply to all time definitions in the Process

C.2. START EVENT

Simulation attributes

Wait time

time to wait as to simulate the execution of the Start Event

Time unit

time unit to be used for the wait time property

C.3. CATCHING INTERMEDIATE EVENTS

Simulation attributes

Wait time

time to wait as to simulate the execution of the Catching Intermediate Event

Time unit

time unit to be used for the wait time property

C.4. SEQUENCE FLOW

Simulation attributes

Probability

probability the Flow is taken in percent

The probability value is applied only if the Flow's source element is a Gateway and there are multiple Flow elements leaving the Gateway. When defining Flow probabilities, make sure their sum is 100.

C.5. THROWING INTERMEDIATE EVENTS

Simulation attributes**Distribution type**

distribution type to be applied to the processing time values

C.6. HUMAN TASKS

Simulation attributes**Cost per time unit**

costs for every time unit lapsed during simulation

Currency

currency of the cost per unit property

Staff availability

number of actors available to work on the given Task

Example C.1. Staff availability impact

Let's assume a simulation of 3 instances of a Process. A new instance is created every 30 minutes. The Process contains a None Start Event, a Human Task, and a Simple End Event. The Human Task takes 3 hours to complete, the Working hours is set to **3**. We have only one person to work on the Human Tasks, that is Staff availability is set to **1**. That results in the following:

- The Human Task generated by the first Process instance will be executed after 3 hours;
- The Human Task generated by the second Process instance will be executed in approx. 6 hours (the second Process instance is created after 30 minutes; however, the actor is busy with the first Task; he becomes available only after another 2.5 hours, and takes 3 hours to execute the second Task).
- The Human Task generated by the third Process instance will be executed in approx. 9 hours (the second Human Task instance is finished after 3 hours; the actor needs another 3 hours to complete the third Human Task).

Working hours

time period for simulation of the Task execution

C.7. SERVICE TASKS

Simulation attributes**Cost per time unit**

costs for every time unit lapsed during simulation

Currency

currency of the cost per unit property

Distribution type

distribution type to be applied to the element execution time

C.8. END EVENTS**Simulation attributes****Distribution Type**

distribution type to be applied to the element execution time

C.9. DISTRIBUTION TYPES

The Distribution type property defines the distribution of possible time values (scores) of Process elements.

The elements might use one of the following score distribution types on simulation:

- **normal**: bell-shaped, symmetrical
- **uniform**: rectangular distribution; every score (time period) is applied the same number of times
- **Poisson**: negatively skewed normal distribution

C.9.1. Normal

The element values are picked based on the normal distribution type, which is bell-shaped and symmetrical.

Normal distribution type**Processing time (mean)**

mean processing time the element needs to be processed (in time unit defined in the time units property)

Standard deviation

standard deviation of the processing time (in time unit defined in the time units property)

Time unit

time unit to be used for the mean processing time and standard deviation values

C.9.2. Uniform

The Uniform distribution or rectangular distribution returns the possible values with the same levels of probability.

Uniform distribution type**Processing time (max)**

maximum processing time of the element

Processing time (min)

minimum processing time of the element

Time unit

time unit to be used for the processing time values

C.9.3. Poisson

The Poisson distribution returns the possible values similarly as normal distribution; however, the distribution is negatively skewed, not symmetrical. The mean and the variant are equal.

Poisson distribution type

Processing time (mean)

mean time for the element processing (in time unit defined in the time units property)

Time unit

time unit to be used for the mean processing time

APPENDIX D. REVISION HISTORY

Revision 1.0.0-43

Thu Dec 17 2015

Vidya Iyengar

Build includes various enhancements and fixes