JBoss BPM Suite 6

Business Process Management with Red Hat JBoss BPM Suite 6

Babak Mozaffari
Member of Technical Staff
Systems Engineering

Version 1.0

March 2014

100 East Davie Street
Raleigh, NC 27601 USA
Phone: +1 919 754 3700
Phone: 888 733 4281
Fax: +1 919 754 3701
PO Box 13588
Research Triangle Park NC 27709 USA

JBoss is a registered trademarks of Red Hat, Inc. in the United States and other countries.

Linux is a registered trademark of Linus Torvalds. JBoss, Red Hat, Red Hat Enterprise Linux and the Red Hat "Shadowman" logo are registered trademarks of Red Hat, Inc. in the United States and other countries.

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

All other trademarks referenced herein are the property of their respective owners.

© 2014 by Red Hat, Inc. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, V1.0 or later (the latest version is presently available at http://www.opencontent.org/openpub/).

The information contained herein is subject to change without notice. Red Hat, Inc. shall not be liable for technical or editorial errors or omissions contained herein.

Distribution of modified versions of this document is prohibited without the explicit permission of Red Hat Inc.

Distribution of this work or derivative of this work in any standard (paper) book form for commercial purposes is prohibited unless prior permission is obtained from Red Hat Inc.

The GPG fingerprint of the security@redhat.com key is:
CA 20 86 86 2B D6 9D FC 65 F6 EC C4 21 91 80 CD DB 42 A6 0E

Comments and Feedback

In the spirit of open source, we invite anyone to provide feedback and comments on any reference architectures. Although we review our papers internally, sometimes issues or typographical errors are encountered. Feedback allows us to not only improve the quality of the papers we produce, but allows the reader to provide their thoughts on potential improvements and topic expansion to the papers.

Feedback on the papers can be provided by emailing refarch-feedback@redhat.com. Please refer to the title within the email.

Staying In Touch

Join us on some of the popular social media sites where we keep our audience informed on new reference architectures as well as offer related information on things we find interesting.

Like us on Facebook:

https://www.facebook.com/rhrefarch

Follow us on Twitter:

https://twitter.com/RedHatRefArch

Plus us on Google+:

https://plus.google.com/114152126783830728030/

Table of Contents

1 Executive Summary.. 1
2 Red Hat JBoss BPM Suite 6.. 2
2.1 Overview.. 2
2.2 Installation Options.. 3
2.3 Administration and Configuration.. 5
2.4 Design and Development.. 7
2.5 Process Simulation.. 8
2.6 Business Activity Monitoring.. 9
2.7 REST API.. 10
3 Reference Architecture Environment.. 11
3.1 Overview.. 11
3.2 BPMS 6.0.1.. 11
3.3 JBoss EAP 6 Cluster.. 12
3.4 ZooKeeper Cluster.. 13
3.5 PostgreSQL database.. 13
3.6 BPM Example Application.. 13
3.7 Runtime Cluster.. 14
4 Creating the Environment.. 15
4.1 Prerequisites.. 15
4.2 Downloads.. 15
4.3 Installation.. 16
4.4 Configuration.. 17
4.5 Review.. 20
5 Design and Development.. 31
5.1 BPM Suite Example Application.. 31
5.2 Project Setup.. 32
5.3 Data Model.. 39
5.4 Business Process.. 45
5.5 Forms.. 75
5.6 Business Rules.. 86
5.7 Credit Report Web Service.. 106

6 Life Cycle.. 107

6.1 Asset Repository Interaction.. 107

6.2 JBoss Developer Studio.. 107

6.3 Process Simulation.. 109

6.4 Business Activity Monitoring.. 111

6.5 Governance... 112

6.6 Process Execution... 112

6.7 Maven Integration.. 117

6.8 Session Strategy... 118

6.9 Timer Implementation.. 118

6.10 REST Deployment... 120

6.11 Continuous Integration... 120

7 Conclusion.. 121

1 Executive Summary

With the increased prevalence of automation, service integration and electronic data collection, it is prudent for any business to take a step back and review the design and efficiency of its business processes.

A business process is a defined set of business activities that represents the steps required to achieve a business objective. It includes the flow and use of information and resources. *Business process management (BPM)* is a systematic approach to defining, executing, managing and refining business processes. Processes typically involve both machine and human interactions, integrate with various internal and external systems, and include both static and dynamic flows that are subject to both business rules and technical constraints.

This reference architecture reviews **`Red Hat JBoss BPM Suite (BPMS) 6`** and walks through the design, implementation and deployment of a sample BPM application. Various features are showcased and a thorough description is provided at each step, while citing the rationale and explaining the alternatives at each decision juncture, when applicable. Within time and scope constraints, potential challenges are discussed, along with common solutions to each problem. A BPMS repository is provided that can be cloned directly to reproduce the application assets. Other artifacts, including supporting code, are also included in the attachment.

2 Red Hat JBoss BPM Suite 6

2.1 Overview

Red Hat JBoss BPM Suite (BPMS) 6 is an open source BPM suite that combines business process management and business rules management, enabling business and IT users to create, manage, validate, and deploy business processes and rules. **BPMS 6** provides advanced business process management capabilities compliant with the widely adopted **BPMN 2.0** standard. The primary goal of BPMN is to provide a notation that is readily understandable by all business users, from the business analysts that create the initial drafts of the processes, to the technical developers responsible for implementing the technology that will perform those processes, and finally, to the business people who will manage and monitor those processes. Thus, BPMN creates a standardized bridge for the gap between the business process design and process implementation.[1]

BPMS 6 comes with a choice of modeling tools; it includes a business-user-friendly, web-based authoring environment as well as an **Eclipse** plugin for developers, to enable all project stakeholders to collaborate effectively and build sophisticated process and decision-automation solutions. The inclusion of **Red Hat JBoss Business Rules Management System (BRMS)** adds seamless integration with business rules and complex event processing functions to ease the development and facilitate the maintenance of processes in the face of rapidly changing requirements. *Business activity monitoring (BAM)* and process *dashboards* provide invaluable information to help manage processes while *process simulation* helps refine business processes by enabling their analysis and assessment of the dynamic behavior of processes over time.

With its 6.0 release, **Red Hat JBoss BPM Suite** also includes **Business Resource Planner** as tech preview. **Business Resource Planner** is a lightweight, embeddable planning engine that optimizes planning problems.

Red Hat JBoss BRMS and **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.[2]

1 http://www.omg.org/spec/BPMN/2.0/PDF
2 https://access.redhat.com/site/documentation/en-US/Red_Hat_JBoss_BPM_Suite/6.0/html/User_Guide/chap-Introduction.html

2.2 Installation Options

2.2.1 Server Platform

Red Hat JBoss BPM Suite comes in two versions[3]:

- *jboss-bpms-6.MINOR_VERSION-redhat-x-deployable-eap6.x.zip*: version adapted for deployment on **Red Hat JBoss Enterprise Application Platform (EAP 6)**.
- *jboss-bpms-6.MINOR_VERSION-redhat-x-deployable-generic.zip*: the deployable version with additional libraries adapted for deployment on **Red Hat JBoss Enterprise Web Server (EWS)** and other supported containers.

The minimum supported configuration of **Red Hat JBoss EAP** for **Red Hat JBoss BPM Suite** installation is **EAP 6.1.1**.

BPMS 6.0.1 is provided as a module layer installed on top of **EAP 6.1.1** with two web applications that can be deployed on the server instances. Other required configuration, however minimal, is best performed by substituting the **EAP** server configuration file (*domain.xml* or *standalone.xml*) and other files and scripts (under *bin/*) with the provided files in the **BPM Suite** download. A layers configuration file is also provided to apply the new layer.

A separate download is provided for other supported containers. Third-party application servers do not make use of **JBoss Modules** and as such, the deployment model for these containers bundles all required libraries within the two web applications. Security policy files are also provided and need to be incorporated based on the container instructions.

For installation and configuration of **JBoss EAP 6**, refer to the previously published JBoss EAP 6 Clustering Reference Architecture.

2.2.2 Clustering

For **Red Hat JBoss BPM Suite**, clustering may refer to various components and aspects of the environment. The following may be clustered:

- Artifact repository: virtual-file-system (VFS) repository that holds the business assets so that all cluster nodes use the same repository
- Execution server and web applications: the runtime server that resides in the container (in this case, **Red Hat JBoss EAP**) along with **BRMS** and **BPM Suite** web applications so that nodes share the same runtime data. For instructions on clustering the application, refer to the previously mentioned JBoss EAP 6 Clustering Reference Architecture.
- Back-end database: database with the state data, such as process instances, KIE sessions, history log, etc., for fail-over purposes

3 https://access.redhat.com/site/documentation/en-US/Red_Hat_JBoss_BPM_Suite/6.0/html-single/Installation_Guide/index.html#chap-Installation_options

For further instructions on clustering the **BPMS** environment, refer to the official Red Hat documentation.[4]

2.2.3 Red Hat JBoss Developer Studio

Red Hat JBoss Developer Studio (JBDS) is the **JBoss** *integrated development environment (IDE)* based on **Eclipse** and available from the Red Hat customer support portal.[5] **Red Hat JBoss Developer Studio** provides plugins with tools and interfaces for **Red Hat JBoss BRMS** and **Red Hat JBoss BPM Suite**. These plugins are based on the community version of these products, so the **BRMS** plugin is called the **Drools** plugin and the **BPM Suite** plugin is called the **jBPM** plugin.

Refer to the **Red Hat JBoss Developer Studio** documentation for installation and setup instructions.[6] For instructions on installing the plugins, setting the runtime library, configuring the **BPMS Server** and importing projects from a Git repository, refer to the official documentation.[7]

4 https://access.redhat.com/site/documentation/en-US/Red_Hat_JBoss_BPM_Suite/6.0/html/Installation_Guide/chap-Clustering.html
5 https://access.redhat.com
6 https://access.redhat.com/site/documentation/en-US/Red_Hat_JBoss_Developer_Studio/
7 https://access.redhat.com/site/documentation/en-US/Red_Hat_JBoss_BPM_Suite/6.0/html/Installation_Guide/chap-Red_Hat_JBoss_Developer_Studio.html

2.3 Administration and Configuration

2.3.1 Business Central

Business Central is a web-based application for asset creation, management, and monitoring of business assets, providing an integrated environment with the respective tools, such as rule and process authoring tools, business asset management tools for work with artifact repository, runtime data management tools, resource editors, BAM (business activity monitoring) tools, task management tools, and **BRMS** tools. It is the main user interface for interacting with **Red Hat JBoss BPM Suite**.

Like most other web applications, **Business Central** configures standard declarative security in *business-central.war/WEB-INF/web.xml*. A number of security roles are defined to grant various levels of access to users:

- *admin*: administrates **BPMS** system and has full access rights to make any changes necessary including the ability to add and remove users from the system.
- *developer*: implements code required for processes to work and has access to everything except administration tasks.
- *analyst*: creates and designs processes and forms and instantiates the processes. This role is the similar to a developer, without access to asset repository and deployments.
- *user*: claims, performs, and invokes other actions (such as, escalation, rejection, etc.) on assigned tasks, but has no access to authoring functions.
- *manager*: monitors the system and its statistics; only has access to the dashboard.

Use the standard **EAP** *add-user.sh* script to create application users in the *ApplicationRealm* and give them one or more of the above security roles. For further details, refer to the official Red Hat documentation.[8]

2.3.2 Asset Repository

Business rules, process definition files and other assets and resources created in **Business Central** are stored in the asset repository called the **Knowledge Store**. The **Knowledge Store** is a centralized repository for business knowledge and uses a Git repository to store its data. **Business Central** provides a web front-end that allows users to view and update the stored content.

To create a new repository or clone an existing one in **Business Central**, visit the administration section under the authoring menu and select an option from the Repositories menu. Refer to the official Red Hat documentation for further details.[9]

8 https://access.redhat.com/site/documentation/en-US/Red_Hat_JBoss_BPM_Suite/6.0/html/Administration_And_Configuration_Guide/chap-Business_Central_configuration.html

9 https://access.redhat.com/site/documentation/en-US/Red_Hat_JBoss_BPM_Suite/6.0/html/Administration_And_Configuration_Guide/chap-Asset_repository.html

2.3.3 Data Persistence

The **BPMS** platform stores the runtime data of the processes in data stores. This includes various items:

- Session state: session ID, date of last modification, the session data that business rules would need for evaluation, state of timer jobs.
- Process instance state: process instance ID, process ID, date of last modification, date of last read access, process instance start date, runtime data (execution status: the node being executed, variable values), event types.
- Work item runtime state: work item ID, creation date, name, process instance ID, state

Based on the persisted data, it is possible to restore the state of execution of all running process instances in case of failure, or to temporarily remove running instances from memory and restore them later.

With **BPM Suite** deployed on **EAP**, persistence is through **Java Persistence API (JPA)**. To set the data source, database type and other properties, configure the standard **JPA** persistence file at *business-central.war/WEB-INF/classes/META-INF/persistence.xml*. Refer to the official Red Hat documentation for further details.[10]

2.3.4 Audit Logging

The audit logging mechanism allows the system to store information about the execution of a process instance. A special event listener listens on the process engine, capture any relevant events, and logs them to the designated destination. Depending on the execution model used in a project, the log can potentially be stored separately from the runtime data and in a separately configured data source. This configuration is outside the scope of this reference architecture and is not discussed in any detail. For further information about the audit logger, refer to the official Red Hat documentation.[11]

2.3.5 Task Execution Configuration

The execution environment may be configured to run a number of business rules when a new task is created or an existing task is completed. To take advantage of this behavior, place two files called *default-add-task.drl* and *default-complete-task.drl* in the root classpath of the server environment. Any business rules within these two files will be evaluated and potentially executed when a task is respectively created and completed.

The task execution engine accesses a mail session as required for escalation, notification or other similar functions. To enable the email functionality, configured a mail session with its "*jndi-name*" set to *"java:/mail/bpmsMailSession".* Configure a corresponding socket binding for the outgoing port.

Refer to the official Red Hat documentation for further details.[12]

10 https://access.redhat.com/site/documentation/en-US/Red_Hat_JBoss_BPM_Suite/6.0/html-single/Administration_And_Configuration_Guide/index.html#chap-Persistence

11 https://access.redhat.com/site/documentation/en-US/Red_Hat_JBoss_BPM_Suite/6.0/html/Administration_And_Configuration_Guide/chap-Logging.html

12 https://access.redhat.com/site/documentation/en-US/Red_Hat_JBoss_BPM_Suite/6.0/html/Administration_And_Configuration_Guide/chap-

2.4 Design and Development

2.4.1 Data Model

Both rules and business processes require a data model to represent the data. Plain Old Java Objects (POJO) are used in BPMS to represent custom data types.[13]

Business Central provides the Data modeler, a custom graphical editor, for defining data objects. The created data types are JavaBeans with annotations added to adapt to the graphical editor.

2.4.2 Process Designer

The Process Designer is the BPMS process modeler. The output of the modeler is a BPMN 2.0 process definition file, which is normally saved in the Knowledge Repository under 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:

- **`JDBS Process Designer`**: Thick-client version of the Process Designer integrated in the Red Hat JBDS plugin
- **`Web Process Designer`**: Thin-client version of the Process Designer integrated in BPM Central

The Process Designer implementation is different for the JDBS Process Designer and the Web Process Designer, but both adhere to the notation specified in BPMN 2.0 and generate similar compliant process files. For further details, consult the official Red Hat documentation.[14]

2.4.3 Forms

A form is a layout definition for a page (defined as HTML) that is displayed as a dialog window to the user, either on process instantiation or task completion; the form is then respectively referred to, as a process form or a task form. It serves for acquiring data for a process or a task, from a human user: a process can accept its process variables as input and a task takes DataInputSet variables with assignment defined and returns DataOutputSet variables that are typically mapped back to process variables.[15]

JBoss BPM Suite provides a web-based custom editor for defining forms.

Execution_server.html

13 https://access.redhat.com/site/documentation/en-US/Red_Hat_JBoss_BPM_Suite/6.0/html/User_Guide/sect-Data_models.html

14 https://access.redhat.com/site/documentation/en-US/Red_Hat_JBoss_BPM_Suite/6.0/html/User_Guide/chap-Process_Designer.html

15 https://access.redhat.com/site/documentation/en-US/Red_Hat_JBoss_BPM_Suite/6.0/html/User_Guide/sect-Forms.html

2.5 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.

The **`Path Finder`** helps identify the various possible paths that a process execution can take. In the web process designer, this tool is available from the toolbar.

Running a simulation requires that simulation properties be correctly set up for each individual element in the process model. This includes setting a probability for each sequence flow leaving a diverging gateway. For an XOR gateway, the sum of all the probability values should be 100%.

Run validation on the process and correct any issues before attempting process simulation. Viewing all issues in the web process designer helps find various simulation-related issues as well.

To run process simulation, specify the number of process instances that are to be started. The interval between process instances can be specified in units as small as millisecond and as large as days. This, coupled with realistic properties set up on each process element, such as the availability of user task actors and minimum and maximum processing time for various automatic and manual tasks can help provide a useful analysis of future process performance.

Once process simulation successfully executes, the results are presented in various charts and tables. Use the legend provided in the graphs to filter out items such as minimum or maximum values.

For further details, refer to the official Red Hat documentation.[16]

16 https://access.redhat.com/site/documentation/en-US/Red_Hat_JBoss_BPM_Suite/6.0/html/User_Guide/chap-Process_simulation.html

2.6 Business Activity Monitoring

`Red Hat JBoss Dashboard Builder` is a web-based dashboard application that provides Business Activity Monitoring (BAM) support in the form of 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.

The included dashboard requests information from the BPMS execution engine and provides real-time information on its runtime data; however, custom dashboards may also be built over other data resources.

The **`Dashboard Builder`** is accessed directly from the Dashboards menu of the Business Central application:

- *Process & Task Dashboards*: displays a pre-defined dashboard based on runtime data from the execution server. An entity may be selected in the menu on the left and the widgets on the right will display the data for that entity.
- *Business Dashboards*: display the environment where custom dashboards are created.

The Dashboard Builder can establish connections to external data sources including 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 BPMS engine by default and queries it for the required data for its jBPM Dashboard indicators (widgets with visualizations of the data available on the pages of the jBPM Dashboard workspace).

If operating over a database, the data provider uses a SQL query to obtain the data and if operating over a CVS 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.[17]

17 https://access.redhat.com/site/documentation/en-US/Red_Hat_JBoss_BPM_Suite/6.0/html/User_Guide/part-BAM.html

2.7 REST API

Representational State Transfer (REST) is a style of software architecture of distributed systems (applications). It allows for a highly abstract client-server communication; clients initiate requests to servers to a particular URL with potentially required parameters and servers process the requests and return appropriate responses based on the requested URL. The requests and responses are built around the transfer of representations of resources. A resource can be any coherent and meaningful concept that may be addressed (such as a repository, a process, a rule, etc.).

Refer to the official Red Hat documentation for further details on the REST API and its usage.[18]

2.7.1 Knowledge Store REST API

REST API calls to the Knowledge Store enable management of the content and manipulation of the static data in the repositories.

The calls are asynchronous and continue their execution after a response is returned to the caller. A job ID is returned by every call to allow the subsequent of request the job status and verify whether the job completed successfully.

These calls provide required parameters as JSON entities.

2.7.2 Deployment REST API

`JBoss BPM Suite` modules can be deployed or undeployed using either the UI or REST API calls. Similar to calls to the Knowledge Store, deployment calls are also asynchronous and quickly return with a job ID that can later be used to query the status of the job.

2.7.3 Runtime REST API

Runtime REST API are calls to the execution servers for process execution, task execution and business rule engine. These calls are synchronous and return the requested data as **`Java Architecture for XML Binding (JAXB)`** objects.

18 https://access.redhat.com/site/documentation/en-US/Red_Hat_JBoss_BPM_Suite/6.0/html/Development_Guide/chap-REST_API.html

3 Reference Architecture Environment

3.1 Overview

This reference architecture builds upon the JBoss EAP 6 Clustering Reference Architecture to create a cluster of BPMS 6.0.1 servers. This environment is a full cluster of all BPMS capabilities, including both design-time and runtime artifacts. An alternative setup is described in the last section under the title of Runtime Cluster where a lighter-weight and more efficient cluster setup is designed for production use; this setup trades off design-time replication for higher efficiency.

The JBoss EAP 6 Clustering Reference Architecture sets up an active/passive pair of clusters with two separate Apache web servers front-ending them. While the same setup can be followed for the purpose of this reference architecture, the focus will remain on the active cluster. The steps would be identical for the passive cluster.

The BPMS binaries are installed on top of each EAP installation before running the configuration scripts that create the EAP cluster domain. BPMS is configured to use **`Apache ZooKeeper`** and **`Apache Helix`** to cluster its artifact repository and manage the cluster. The **`uberfire`** framework provides the backbone of the web applications, while also providing the **`Virtual File System (VFS)`** implementation of the application environment.

The HTTP Server always redirects to a logical EAP 6 Application Server for all functionality, which in turn uses the **`PostgreSQL database`** instance for its persistence.

3.2 BPMS 6.0.1

BPMS 6 includes the **`Business Central`** and **`Dashbuilder`** web applications, which are hosted on the EAP 6 servers. Supporting libraries are added to the EAP 6 servers as a module layer. Security policy files and a few configurations are also applied to the EAP domains.

BPMS uses Git repositories for both its asset repositories and workbench configurations. In this reference environment, Apache ZooKeeper is leveraged as a clustered and replicated VFS for the Git repositories.

Application artifacts are stored as **`Maven`** projects. Each repository may contain one or more projects, each of which is described by a project object model (*.pom*) file and may contain various asset types.

BPMS 6 uses the central PostgreSQL database instance to store runtime process information, **`Quartz`** timer data and audit log information that is used for business activity monitoring. For clients with a large number of process executions, it may be advisable to use a separate database for the BAM data. Persistence for Business Central is configured in the *persistence.xml* file but the changes required to separate the datasources is outside the scope of this guide and depends on the execution model.

3.3 JBoss EAP 6 Cluster

BPMS 6 includes a number of modules and web applications that are configured and deployed on top of JBoss EAP 6. Accordingly, the foundation of the BPMS environment is the setup of an EAP 6 cluster.

3.3.1 JBoss EAP 6 Clustering Reference Architecture

The previously published *JBoss EAP 6 Clustering* reference architecture thoroughly describes the installation and configuration of EAP 6, while providing scripts and assets to facilitate the automation of such a setup. The clustering of a number of components is described and implemented. While certain configurations and components may not be directly used by BPMS or this reference environment, they can still be used to host other applications that do or do not interact with BPMS.

Clients with access to the Red Hat Customer Portal may download the reference architecture and attachments from https://access.redhat.com/site/articles/524633

A public version of this document, without the file attachment, is available from http://www.redhat.com/resourcelibrary/reference-architectures/jboss-eap-6-clustering

The EAP 6 Clustering guide provides a great foundation for this reference architecture environment. Two clusters are set up as part of this guide to provide an active/passive configuration and eliminate any downtime due to maintenance and upgrades. The focus of additional setup and configuration will be the active cluster, but the same instructions may be followed for the passive cluster.

The active EAP 6 cluster includes three instances of JBoss EAP 6.1.1 servers installed on separate virtual (or physical) machines.

3.3.2 JBoss EAP Apache HTTP Server

As part of the EAP 6 Cluster, two instances of **`JBoss EAP 6 Apache HTTP Server`** (based on **`Apache's httpd`**) are installed on a separate machine to front-end the active and passive EAP clusters, providing sticky-session load balancing and failover. The HTTP servers use mod_cluster and the AJP protocol, to forward HTTP requests to an appropriate EAP node.

In this setup, the front-ending Apache web server is used to access these two applications and reroutes the requests to an EAP node as appropriate. This may include both browser requests that require sticky-session load balancing as well as REST calls that are stateless and may be routed to any cluster member.

3.4 ZooKeeper Cluster

In this reference architecture, ZooKeeper is clustered alongside EAP, so that three nodes of ZooKeeper and Helix run alongside the three EAP 6.1.1 nodes of each cluster.

An Apache ZooKeeper cluster is known as an ensemble and requires a majority of the servers to be functional for the service to be available. Choosing an odd number for the cluster size is always preferable. Both a three-member and a four-member cluster can only withstand the loss of a single member without losing a functioning majority, so as shown by this example, groups with an odd number of members provide higher efficiency.

ZooKeeper allows BPMS to replicate its Git repositories. BPMS also uses ZooKeeper to replicate other locally maintained data, including deployments. Only a single instance of ZooKeeper is required to allow BPMS to replicate its data; the ZooKeeper ensemble serves to provide redundancy and protect against the failure of ZooKeeper itself.

Helix works in conjunction with ZooKeeper as the cluster management component that registers all the cluster details (the cluster itself, nodes, resources).

3.5 PostgreSQL database

The JBoss EAP 6 Clustering Reference Architecture installs and configured a single PostgreSQL database instance for both the active and passive clusters. In this reference architecture, BPMS uses the same database to persist its data. This includes active business process data, audit log and **`Quartz`** timer data.

The one instance of PostgreSQL Database can be considered a single point of failure. The Database itself can also be clustered, but the focus of this effort is BPMS 6 and clustering of the Database is beyond the scope of this reference architecture.

3.6 BPM Example Application

This reference architecture provides a step-by-step guide to the design and development of an example application. The completed application is provided in the attachments and is also directly available for download from the Red Hat customer support portal. Refer to the section on BPM Suite Example Application for further details.

3.7 Runtime Cluster

This reference environment provides protection against failover as well as opportunity for load balancing at various levels for multiple tiers and components. This includes:

- Artifact Repository: The use of ZooKeeper and Helix effectively allows for the replication of the virtual file system used by BPMS. Most notably, this includes the Git repositories that hold the workbench data and the application projects. However the content of these repositories only change during design and development. Another type of artifact that is replicated through this mechanism is deployments. Without ZooKeeper, any new deployment would need to be pushed to every single server in the cluster.
- Execution Server and Web Applications: JBoss EAP Clustering provides failover and load balancing for HTTP sessions, Enterprise Java Beans (EJB) and Java Message Service (JMS). While BPMS applications may be designed to interface with any of these components, neither the provided framework nor the example application in this reference architecture includes any of these items. Business Central and Dashbuilder are not configured to be distributable and even if they are modified to use session replication, neither holds any significant state that can benefit from this feature.
- Back-end Database: This reference environment includes an instance of PostgreSQL database that is external to the EAP servers. As previously noted, the database itself is best clustered to protect against a single point of failure. However regardless of the redundancy strategy employed for the database, it provides failover capability and load balancing opportunity for the EAP nodes. Data stored in the database from one node is available to another on a subsequent request and the external storage of data protects against the failure of an EAP Server.

BPMS 6 provides the Business Central application as a design and development environment as well as an analysis and testing tool and a production runtime platform. Developers and analyst may use the web process designer to create or update business processes and the various rule editors to implement business rules. In later stages, test scenarios can help verify expected functionality and the QA facilities are useful for reviews. Finally, in production, Business Central may be used either directly through its forms to start processes and work on forms, or through its REST interface to delegate the same functions from a custom application. The latter set of activities can broadly be categorized as runtime and distinguished from the type of design-time work that was described earlier.

There are numerous benefits to separating design-time and runtime BPMS instances. For a production instance where design-time activity is not supported, clustering the asset repository and the execution server is often unnecessary. This cost can be avoided by simply making sure deployments are made available on every server. Failover and load balancing is still achieved through the use of a central external database and a web service front-end to route browser and REST requests. Instead of an EAP cluster, the domain can simply include a number of nodes that exclude HA capability. There is no valuable in-memory state and the persisted local state only includes design-time artifacts that can be ignored and deployments that need to be externally managed and replicated.

4 Creating the Environment

4.1 Prerequisites

This reference architecture uses the JBoss EAP 6 Clustering Reference Architecture as its foundation and shares the same prerequisites. This includes a supported Operating System and JDK. Refer to Red Hat documentation for supported environments.[19]

With minor changes, almost any RDBMS may be used in lieu of PostgreSQL Database for both this reference environment and the EAP 6 cluster, but if PostgreSQL is used, the details of the download and installation are also considered a prerequisite for this reference architecture. On a RHEL system, installing PostgreSQL can be as simple as running:

```
# yum install postgresql-server.i686.
```

4.2 Downloads

The JBoss EAP 6 Clustering Reference Architecture requires a number of installation files as well as its own attachment to be downloaded. Refer to the downloads section of the document for details.

Also download the attachments to this document. These scripts and files will be used in configuring the reference architecture environment:

https://access.redhat.com/site/node/785313/40/1

If you do not have access to the Red Hat customer portal, See the Comments and Feedback section to contact us for alternative methods of access to these files.

Download JBoss BPM Suite 6.0.1 and its supplementary tools from Red Hat's Customer Support Portal[20]:

- Red Hat JBoss BPM Suite 6.0.1 Deployable for EAP 6.1.1
- Red Hat JBoss BPM Suite 6.0.1 Supplementary Tools

19 https://access.redhat.com/site/articles/704703
20 https://access.redhat.com/jbossnetwork/restricted/listSoftware.html?downloadType=distributions&product=bpm.suite&version=6.0.1

4.3 Installation

4.3.1 JBoss EAP Apache HTTP Server

Follow the installation section of the JBoss EAP 6 Clustering Reference Architecture document to set up JBoss EAP Apache HTTP Server.

4.3.2 JBoss Enterprise Application Platform

Follow the installation section of the JBoss EAP 6 Clustering Reference Architecture document to set up JBoss EAP 6.1.1.

4.3.3 ZooKeeper

ZooKeeper and Helix are distributed as part of the supplementary tools provided with BPM Suite. Simply unzip the archive on all nodes:

```
# unzip jboss-bpms-brms-6.0.1.GA-redhat-2-supplementary-tools.zip
```

For convenience, create symbolic links to ZooKeeper and Helix directories on each node:

```
# ln -s /root/jboss-bpms-brms-6.0.1.GA-redhat-2-supplementary-
tools/zookeeper-3.3.4 /zookeeper
# ln -s /root/jboss-bpms-brms-6.0.1.GA-redhat-2-supplementary-tools/helix-
core-0.6.2-incubating /helix
```

4.3.4 JBoss BPM Suite

JBoss BPM Suite is installed as a module layer on top of JBoss EAP 6.1.1, with two web applications that are later deployed on the server instances. There are a few other configuration items that are applied by copying over the provided files in the BPM Suite download, including the EAP server configuration file (*domain.xml* or *standalone.xml*) and other files and scripts (under *bin/*). A layers configuration file is also provided to apply the new layer. On each node where EAP 6.1.1 is installed, follow these steps:

```
# unzip jboss-bpms-6.0.1.GA-redhat-1-deployable-eap6.x.zip -d ~
# mv -f ~/jboss-eap-6.1/domain/configuration/domain.xml
                      /opt/jboss-eap-6.1_active/domain/configuration/
# mv -f ~/jboss-eap-6.1/bin/* /opt/jboss-eap-6.1_active/bin/
# mv -f ~/jboss-eap-6.1/modules/layers.conf
                      /opt/jboss-eap-6.1_active/modules/
# mv ~/jboss-eap-6.1/modules/system/layers/bpms
                      /opt/jboss-eap-6.1_active/modules/system/layers/
```

A web application has to be in archive form to deploy to EAP 6 in domain mode. The Business Central and Dashbuilder web applications in the BPM Suite download are in exploded form and need to be archived. This can be accomplished with a simple jar command, for example:

```
# cd ~/jboss-eap-6.1/standalone/deployments/business-central.war/
# jar cvf ~/business-central.war **
```

4.4 Configuration

Follow the general configuration section of the JBoss EAP 6 Clustering Reference Architecture to open the required firewall ports, set up **SELinux** and otherwise configure the operating system as appropriate.

In this reference environment, ZooKeeper is set up to use port 2181 for client communication. The ZooKeeper ensemble uses port 2188 for followers to connect to the leader as well as port 3188 for leader election. To open these three ports within the set of IP addresses used in this reference environment in a Linux environment, use the following IPTables instructions:

```
# iptables -I INPUT 24 -p tcp -s 10.16.139.0/24
                                             --dport 2181 -m tcp -j ACCEPT
# iptables -I INPUT 24 -p tcp -s 10.16.139.0/24
                                             --dport 2888 -m tcp -j ACCEPT
# iptables -I INPUT 24 -p tcp -s 10.16.139.0/24
                                             --dport 3888 -m tcp -j ACCEPT
```

The firewall rules may then be persisted to survive reboots:

```
# /sbin/service iptables save
```

To transfer assets between JBoss Developer Studio and Business Central through Git, also open the Git SSH port for the server. This reference environment uses port 8003 for Git over SSH.

4.4.1 JBoss EAP Apache HTTP Server

Configure the web server as detailed in the JBoss EAP 6 Clustering Reference Architecture.

4.4.2 PostgreSQL Database

Configure the PostgreSQL Database as detailed in the JBoss EAP 6 Clustering Reference Architecture and create the *eap6* database for the *jboss* user as described.

Additionally, create a new database called *bpms* with the same *jboss* user as the owner:

```
CREATE DATABASE bpms WITH OWNER jboss;
```

Making the *jboss* user the owner of the *bpms* database ensures that the necessary privileges to create tables and modify data are assigned.

4.4.3 JBoss Enterprise Application Server

Configure the EAP instances as detailed in the JBoss EAP 6 Clustering Reference Architecture.

4.4.4 ZooKeeper

Create a data directory for ZooKeeper on each node:

```
# mkdir /zookeeper/data
```

To create a ZooKeeper cluster, assign a node ID to each member that runs ZooKeeper. For this reference environment, use “1”, “2” and “3” respectively for node 1, node 2 and node 3. The node ID is specified in a file called *myid* under the data directory of ZooKeeper on each node. For example, on node 1:

```
# echo "1" > /zookeeper/data/myid
```

Place the ZooKeeper configuration file in the *conf* directory:

```
# cp zoo.cfg /zookeeper/conf/
```

Start ZooKeeper on each node by running the start script:

```
# /zookeeper/bin/zkServer.sh start
```

View the *zookeeper.out* log and ensure that the ensemble is formed successfully. One of the nodes should be elected as leader with the other two nodes following it.

Once the ZooKeeper ensemble is started, the next step is to configure and start Helix. Helix only needs to be configured once and from a single node. The configuration is then stored by the ZooKeeper ensemble and shared as appropriate.

Remember that the client port for ZooKeeper in the reference environment has been set to 2181. The ZooKeeper connection string is a comma delimited list of host and port values, so using the IP addresses of the three nodes, the connection string is:

```
10.16.139.101:2181,10.16.139.102:2181,10.16.139.103:2181
```

Use the helix admin script to define a cluster with an arbitrary name:

```
# /helix/bin/helix-admin.sh
          –zkSvr 10.16.139.101:2181,10.16.139.102:2181,10.16.139.103:2181
          --addCluster repoCluster
```

Identify each ZooKeeper node with a *host:port* combination. The values of host and port for the purpose of this identifier are arbitrary and are not used to access the servers. It is only important to use the same consistent values for each node's host and port in the following commands as the system properties used to configure the BPMS instances.

This reference environment uses the name of each node as its host and its standard HTTP port. Based on this convention, use the helix admin script to add the three nodes to the cluster:

```
# /helix/bin/helix-admin.sh
          --zkSvr 10.16.139.101:2181,10.16.139.102:2181,10.16.139.103:2181
          --addNode repoCluster node1:8080
# /helix/bin/helix-admin.sh
          --zkSvr 10.16.139.101:2181,10.16.139.102:2181,10.16.139.103:2181
          --addNode repoCluster node2:8080
# /helix/bin/helix-admin.sh
          --zkSvr 10.16.139.101:2181,10.16.139.102:2181,10.16.139.103:2181
          --addNode repoCluster node3:8080
```

Finally, add the virtual file system and rebalance the cluster as follows:

```
# /helix/bin/helix-admin.sh
          --zkSvr 10.16.139.101:2181,10.16.139.102:2181,10.16.139.103:2181
          --addResource repoCluster vfs-repo 1 LeaderStandby AUTO_REBALANCE
# /helix/bin/helix-admin.sh
          --zkSvr 10.16.139.101:2181,10.16.139.102:2181,10.16.139.103:2181
          --rebalance repoCluster vfs-repo 2
```

Once the Helix configuration is completed, run the helix controller on each node of the ZooKeeper ensemble:

```
# /helix/bin/run-helix-controller.sh
          --zkSvr 10.16.139.101:2181,10.16.139.102:2181,10.16.139.103:2181
          --cluster repoCluster
```

4.4.5 JBoss BPM Suite

At this point, the configuration script has executed and set up the EAP 6 domain for the cluster. Configuring BPM Suite is very similar and uses a CLI script that is based on the same framework. To use this script, replace the EAP 6 configuration properties with the *configuration.properties* file provided with this guide. Place *bpms-config.jar* in the same directory and run the BPMS configuration:

```
# java -cp bpms-config.jar:configuration.jar:
                     /opt/jboss-eap-6.1/bin/client/jboss-cli-client.jar
                     org.jboss.refarch.bpms6.BPMS
```

The BPMS configuration script takes a few minutes to execute, restarts the servers and concludes by deploying Business Central and Dashbuilder.

To verify successful setup, shut down all three servers and start them again deliberately, one after the other, ensuring that the domain controller has started before starting up the two other nodes.

4.5 Review

4.5.1 JBoss EAP Apache HTTP Server

The configuration of the web server is reviewed in detail in the JBoss EAP 6 Clustering Reference Architecture.

4.5.2 PostgreSQL Database

Follow the steps described in the review section of the database detailed in the JBoss EAP 6 Clustering Reference Architecture. The database configuration is largely the same, with the addition of a new database for BPMS.

4.5.3 JBoss Enterprise Application Server

The configuration of EAP 6.1.1 is reviewed in the JBoss EAP 6 Clustering Reference Architecture and remains identical in this reference architecture.

4.5.4 ZooKeeper

ZooKeeper is configured through the */zookeeper/conf/zoo.cfg* configuration file. This reference architecture uses default values for the first few parameters:

```
# The number of milliseconds of each tick
tickTime=2000
# The number of ticks that the initial
# synchronization phase can take
initLimit=10
# The number of ticks that can pass between
# sending a request and getting an acknowledgement
syncLimit=5
```

The data directory for ZooKeeper on each node is created in the previous section and configured in this file:

```
# the directory where the snapshot is stored.
dataDir=/zookeeper/data
```

The client port is also left at it default value of 2181. This is the port used by BPMS instances to talk to ZooKeeper.

```
# the port at which the clients will connect
clientPort=2181
```

The ZooKeeper ensemble is also defined in this configuration file. Each member of the ensemble is described on a separate line with its node ID identifying it. The host address, leader port and leader election port of the node are provided as the values for each node, with a colon used as the delimiter.

```
server.1=10.16.139.101:2888:3888
server.2=10.16.139.102:2888:3888
server.3=10.16.139.103:2888:3888
```

For a node identified by *server.1*, the file *myid* in the data directory of that node's ZooKeeper is expected to contain the value of “1”.

4.5.5 JBoss BPM Suite

The BPM Suite servers are configured by running a Java class that executes a series of CLI instructions. This class is built on the same foundation as the CLI class created for the JBoss EAP 6 Clustering Reference Architecture and reads an expanded version of the same configuration property file:

```
public class BPMS
{

	private Client client;
	private String postgresDriverName;
	private String postgresUsername;
	private String postgresPassword;
	private String bpmsConnectionUrl;
	private String bpmsDS;
	private String bpmsNonJTA_DS;
	private String gitDir;
	private String indexDir;
	private String quartzProperties;
	private String helixClusterId;
	private String zookeeper;
	private String businessCentral;
	private String dashbuilder;

	public static void main(String[] args) throws CommandLineException, 
IOException
	{
		String propertyFile = "./configuration.properties";
		if( args.length == 1 )
		{
			propertyFile = args[0];
		}
		Properties props = new Properties();
		props.load( new FileReader( propertyFile ) );
		System.out.println( "properties loaded as: " + props );
		BPMS configuration = new BPMS( props );
		configuration.configure();
		System.out.println( "Done!" );
	}
```

The additional properties are added to *configuration.properties*:

```
bpmsConnectionUrl=jdbc:postgresql://10.16.139.100:5432/bpms
bpmsDS=bpmsDS
bpmsNonJTA_DS=bpmsNonJTA_DS
gitDir=/opt/bpms/repo
indexDir=/opt/bpms/index
quartzProperties=/root/quartz-definition.properties
helixClusterId=repoCluster
zookeeper=10.16.139.101:2181,10.16.139.102:2181,10.16.139.103:2181
dashbuilder=/root/files/dashbuilder.war
businessCentral=/root/files/business-central.war
```

Once instantiated, the BPMS class reads the required properties and saves them as fields:

```
public BPMS(Properties properties) throws CommandLineException
{
    String username = properties.getProperty( "username" );
    String password = properties.getProperty( "password" );
    String domainController = properties.getProperty( "domainController" );
    client = new Client( username, password.toCharArray(), domainController,
9999 );
    postgresDriverName = properties.getProperty( "postgresDriverName" );
    postgresUsername = properties.getProperty( "postgresUsername" );
    postgresPassword = properties.getProperty( "postgresPassword" );

    bpmsConnectionUrl = properties.getProperty( "bpmsConnectionUrl" );
    bpmsDS = properties.getProperty( "bpmsDS" );
    bpmsNonJTA_DS = properties.getProperty( "bpmsNonJTA_DS" );
    gitDir = properties.getProperty( "gitDir" );
    indexDir = properties.getProperty( "indexDir" );
    quartzProperties = properties.getProperty( "quartzProperties" );
    helixClusterId = properties.getProperty( "helixClusterId" );
    zookeeper = properties.getProperty( "zookeeper" );
    businessCentral = properties.getProperty( "businessCentral" );
    dashbuilder = properties.getProperty( "dashbuilder" );
}
```

The configuration tasks begin with the configure method and the first step is to discover all the registered hosts:

```
private void configure() throws CommandLineException, IOException
{
    List<Resource> hosts = client.getResourcesByType( null, "host" );
```

This Java code uses the existing framework to find all the hosts and map them to Resource objects. To list hosts in straight CLI, use the following instruction:

```
:read-children-names(child-type=host)
```

The code then continues to set the required system properties:

```
private void setSystemProperties(List<Resource> hosts) throws IOException
{
    String[][] newProperties = new String[2][];
    newProperties[0] = new String[] {"org.uberfire.nio.git.dir",
"org.uberfire.metadata.index.dir", "org.quartz.properties",
"jboss.node.name", "org.uberfire.cluster.id", "org.uberfire.cluster.zk",
"org.uberfire.cluster.local.id", "org.uberfire.cluster.vfs.lock",
"org.uberfire.nio.git.deamon.port", "org.uberfire.cluster.autostart",
"org.uberfire.nio.git.ssh.port"};
    newProperties[1] = new String[] {gitDir, indexDir, quartzProperties,
null, helixClusterId, zookeeper, null, "vfs-repo", "9418", "false", "8003"};
```

Eleven properties are defined in a two dimensional array where the first dimension holds the property name and the second contains the new values.

As shown in the code, four of these eleven properties are set to constant values, while five values are retrieved from the properties.

The fourth property is the node name and its value is queried from domain.

```
     for( Resource host : hosts )
     {
           newProperties[1][3] = host.getName();
```

In other words, in each iteration, the value of the property is the corresponding value retrieved as part of the previous CLI query.

The script then proceeds to discover the bind address for each host.

```
String bindAddress;
Resource server = client.getResourcesByType( host, "server" ).get( 0 );
Resource platformMBean = new Resource( "core-service", "platform-mbean" );
platformMBean.setParent( server );
Resource runtime = new Resource( "type", "runtime" );
runtime.setParent( platformMBean );
client.load( runtime );
ModelNode bindAddressNode = runtime.getAttribute( "system-properties" )
                                 .getValue().get( "jboss.bind.address" );
if( bindAddressNode.isDefined() )
{
     bindAddress = bindAddressNode.asString();
}
else
{
     bindAddress = null;
}
```

To query the bind address in straight CLI, retrieve a set of properties by issuing the following command and look for the value of *jboss.bind.address* in the result.

```
/host=node1/core-service=platform-mbean/type=runtime
                                 :read-attribute(name=system-properties)
```

While the reference environment only configures a single server group for each host of a domain, the script remains generic and avoids making any such assumptions.

```
List<Resource> serverConfigs = client.getResourcesByType
                                                  ( host, "server-config" );
for( Resource serverConfig : serverConfigs )
{
```

For each server, the script queries the binding port offset and uses it to calculate the HTTP port off the base of 8080. The previously retrieved host name is combined with the listen port to derive the ZooKeeper cluster member identifier:

```
Attribute portOffsetAttr = client.readAttribute( serverConfig,
                                                  "socket-binding-port-offset" );
int port = 8080 + portOffsetAttr.getValue().asInt();
newProperties[1][6] = newProperties[1][3] + "_" + port;
```

The port offset may also be queried in straight CLI:

```
/host=node1/server-config=node1-active-server:read-attribute
                                          (name=socket-binding-port-offset)
```

To set the eleven properties on each server, the script creates a template Resource object that can be set up with the common characteristics and subsequently modified for each property.

```
Resource systemProperty = new Resource( "system-property", null );
systemProperty.setParent( serverConfig );
systemProperty.addAttribute( new Attribute( "boot-time", false ) );
```

The template is then used to create the eleven properties in a loop:

```
for( int index = 0; index < newProperties[0].length; index++ )
{
    systemProperty.setName( newProperties[0][index] );
    systemProperty.setAttribute( "value", newProperties[1][index] );
    client.create( systemProperty );
}
```

The system properties can easily be created in straight CLI as well, for example:

```
/host=node1/server-config=node1-active-server/
                           system-property=org.uberfire.nio.git.ssh.port:
                           add(boot-time=false,value=8003)
```

Two further properties are set up next, based on the bind address.

```
if( bindAddress != null )
{
    Resource bindAddressProperty = new Resource( "system-property",
                                    "org.uberfire.nio.git.daemon.host" );
    bindAddressProperty.setParent( serverConfig );
    bindAddressProperty.setAttribute( "value", bindAddress );
    client.create( bindAddressProperty );
    bindAddressProperty.setName( "org.uberfire.nio.git.ssh.host" );
    client.create( bindAddressProperty );
}
```

The final result of the *setSystemProperties* method is to set thirteen system properties on each server of each host. This can be viewed in the host XML files:

```
<servers>
    <server name="node1-active-server" group="cluster-server-group-1" auto-
start="true">
        <system-properties>
            <property name="org.uberfire.nio.git.dir"
                      value="/opt/bpms/repo" boot-time="false"/>
            <property name="org.uberfire.metadata.index.dir"
                      value="/opt/bpms/index" boot-time="false"/>
            <property name="org.quartz.properties"
                      value="/root/quartz-definition.properties"
                      boot-time="false"/>
            <property name="jboss.node.name"
                      value="node1" boot-time="false"/>
            <property name="org.uberfire.cluster.id"
                      value="repoCluster" boot-time="false"/>
            <property name="org.uberfire.cluster.zk"
                      value="10.16.139.101:2181" boot-time="false"/>
            <property name="org.uberfire.cluster.local.id"
                      value="node1_8080" boot-time="false"/>
            <property name="org.uberfire.cluster.vfs.lock"
                      value="vfs-repo" boot-time="false"/>
            <property name="org.uberfire.nio.git.deamon.port"
                      value="9418" boot-time="false"/>
            <property name="org.uberfire.cluster.autostart"
                      value="false" boot-time="false"/>
            <property name="org.uberfire.nio.git.ssh.port"
                      value="8003" boot-time="false"/>
            <property name="org.uberfire.nio.git.daemon.host"
                      value="10.16.139.101" />
            <property name="org.uberfire.nio.git.ssh.host"
                      value="10.16.139.101" />
        </system-properties>
```

The next method call of the CLI script sets up the JVM for each host. The maximum memory allowed for the permanent generation of the Java virtual machine is increased to 512 megabytes to better accommodate the BPM Suite:

```
for( Resource host : hosts )
{
     List<Resource> serverConfigs = client.getResourcesByType( host, "server-
config" );
     for( Resource serverConfig : serverConfigs )
     {
           Resource jvm = new Resource( "jvm", "serverJVM" );
           jvm.setParent( serverConfig );
           jvm.setAttribute( "max-permgen-size", "512m" );
           client.create( jvm );
     }
}
```

To set up the JVM for a known host through straight CLI, simply add a jvm to the server configuration:

```
/host=node1/server-config=node1-active-server/jvm=serverJVM
                                             :add(max-permgen-size=512m)
```

The result is a jvm XML element in the server configuration in the host XML:

```
        <jvm name="serverJVM">
            <permgen max-size="512m"/>
        </jvm>
    </server>
```

The next set of changes apply to the profiles. The first step is to query all the configured profiles:

```
List<Resource> profiles = client.getResourcesByType( null, "profile" );
```

To list profiles in straight CLI, use the following instruction:

```
:read-children-names(child-type=host)
```

The next step is to set up clustered single sign-on (SSO) for each profile. That's achieved by removing the existing SSO entry and recreating one with the desired settings.

```
for( Resource profile : profiles )
{
     Resource web = new Resource( "subsystem", "web" );
     web.setParent( profile );
     List<Resource> virtualServers =
                client.getResourcesByType( web, "virtual-server" );
     for( Resource virtualServer : virtualServers )
     {
           List<Resource> ssoConfigs =
                      client.getResourcesByType( virtualServer, "sso" );
           for( Resource sso : ssoConfigs )
           {
                client.remove( sso );
           }
           Resource sso = new Resource( "sso", "configuration" );
           sso.setParent( virtualServer );
           sso.setAttribute( "cache-container", "web" );
           sso.setAttribute( "cache-name", "sso" );
           sso.setAttribute( "domain-name", "bpms" );
           sso.setAttribute( "reauthenticate", false );
           client.create( sso );
     }
}
```

To remove the existing SSO setup and configure clustered SSO with a direct CLI command, issue the following for each profile:

```
/profile=full-ha-1/subsystem=web/virtual-server=default-host
                                             /sso=configuration:remove()

/profile=full-ha-1/subsystem=web/virtual-server=default-host
                      /sso=configuration:add(cache-container=web,
                            cache-name=sso,domain=bpms,reauthenticate=false)
```

The SSO is set up as a configuration in the virtual server definition within the web subsystem of each profile:

```
<virtual-server name="default-host" enable-welcome-root="true">
    <alias name="localhost"/>
    <alias name="example.com"/>
    <sso cache-container="web" cache-name="sso" domain="bpms"
                                             reauthenticate="false"/>
</virtual-server>
```

After setting up single sign-on, the script creates one transactional and one non-transactional datasource for each profile.

```
setupDataSources( profiles );
```

The transactional datasource is configured as follows:

```
private void setupDataSources(List<Resource> profiles) throws IOException
{
     Resource dataSources = new Resource( "subsystem", "datasources" );

     Resource bpmsJTA = new Resource( "data-source", bpmsDS );
     bpmsJTA.setParent( dataSources );
     bpmsJTA.setAttribute( "enabled", true );
     bpmsJTA.setAttribute( "jta", true );
     bpmsJTA.setAttribute( "jndi-name", "java:jboss/datasources/" + bpmsDS );
     bpmsJTA.setAttribute( "connection-url", bpmsConnectionUrl );
     bpmsJTA.setAttribute( "driver-class", "org.postgresql.xa.PGXADataSource"
);
     bpmsJTA.setAttribute( "driver-name", postgresDriverName );
     bpmsJTA.setAttribute( "user-name", postgresUsername );
     bpmsJTA.setAttribute( "password", postgresPassword );
     bpmsJTA.setAttribute( "use-java-context", true );
     bpmsJTA.setAttribute( "use-ccm", true );
```

The non-transactional datasource is similar:

```
     Resource bpmsNonJTA = new Resource( "data-source", bpmsNonJTA_DS );
     bpmsNonJTA.setParent( dataSources );
     bpmsNonJTA.setAttribute( "enabled", true );
     bpmsNonJTA.setAttribute( "jta", false );
     bpmsNonJTA.setAttribute( "jndi-name", "java:jboss/datasources/" +
bpmsNonJTA_DS );
     bpmsNonJTA.setAttribute( "connection-url", bpmsConnectionUrl );
     bpmsNonJTA.setAttribute( "driver-class", "org.postgresql.Driver" );
     bpmsNonJTA.setAttribute( "driver-name", postgresDriverName );
     bpmsNonJTA.setAttribute( "user-name", postgresUsername );
     bpmsNonJTA.setAttribute( "password", postgresPassword );
     bpmsNonJTA.setAttribute( "use-java-context", true );
     bpmsNonJTA.setAttribute( "use-ccm", true );
```

Once both Resource objects have been set up, they can be created under the existing datasources configuration for each profile:

```
     for( Resource profile : profiles )
     {
           dataSources.setParent( profile );
           client.create( bpmsJTA );
           client.create( bpmsNonJTA );
     }
```

The syntax for a direct CLI command to create a datasource looks as follows:

```
/profile=full-ha-1/subsystem=datasources/data-source=bpmsDS
     :add(enabled=true,jta=true,jndi-name="java:jboss/datasources/bpmsDS",
           connection-url="jdbc:postgresql://10.16.139.100:5432/bpms",
           driver-class="org.postgresql.xa.PGXADataSource",
           driver-name="postgresql-9.2-1003.jdbc4.jar",
           user-name=jboss,password=password,use-java-context=true,
           use-ccm=true)
```

The created datasources can be viewed in the domain.xml file under each profile. The transactional datasource is created to be used by Business Central.

```
<datasource jta="true" jndi-name="java:jboss/datasources/bpmsDS" pool-
name="bpmsDS" enabled="true" use-java-context="true" use-ccm="true">
    <connection-url>jdbc:postgresql://10.16.139.100:5432/bpms</connection-
url>
    <driver-class>org.postgresql.xa.PGXADataSource</driver-class>
    <driver>postgresql-9.2-1003.jdbc4.jar</driver>
    <security>
        <user-name>jboss</user-name>
        <password>password</password>
    </security>
</datasource>
```

The non-transactional datasource is configured similarly. It is required by *Quartz*.

```
<datasource jta="false" jndi-name="java:jboss/datasources/bpmsNonJTA_DS"
          pool-name="bpmsNonJTA_DS" enabled="true" use-java-context="true"
                                                              use-ccm="true">
  <connection-url>jdbc:postgresql://10.16.139.100:5432/bpms</connection-url>
  <driver-class>org.postgresql.Driver</driver-class>
  <driver>postgresql-9.2-1003.jdbc4.jar</driver>
  <security>
      <user-name>jboss</user-name>
      <password>password</password>
  </security>
</datasource>
```

Once datasources have been configured, the script restarts all the servers in the domain.

```
    stopAllServers();
    for( Resource host : hosts )
    {
          waitForServerShutdown( host );
    }
    startAllServers();
```

At this point, the BPM Suite has been fully configured to function in a cluster. The last step is to deploy the BPMS web applications.

```
    client.deploy( businessCentral );
    client.deploy( dashbuilder );
```

The CLI script simply disconnects and exits after the web applications have been deployed.

```
    client.disconnect();
    System.out.println( "Disconnected" );
```

5 Design and Development

5.1 BPM Suite Example Application

This reference architecture includes an example application that is designed, developed, deployed, tested and described in this document. The application consists of a business process that manages the various automatic and manual steps involved in processing a mortgage application, up until the approval or denial of the mortgage.

This application is alternatively referred to as “BPM Suite Example Application”, “jboss-bpm-example” and “mortgage demo” in various contexts and is available for download both as an attachment to this reference architecture and as a download from the Red Hat Customer Portal, alongside the BPMS product itself. Due to divergent update schedules and restricted release cycles, the copy used in this reference architecture may at different times be either older or newer than the product download.

While a complete copy of the example application is provided with this reference architecture, this section walks the reader through every step of design and development. By following the steps outlined in this section, the reader is able to replicate the original effort and recreate every component of the application. This document explains the design decisions at each step and outlines some best practices.

5.2 Project Setup

5.2.1 Business Central

This reference architecture assumes that the previous installation and configuration steps have been followed and the environment set up. The document further assumes that a user has been set up with the security role of *ADMIN*. Creating the project and developing the application as outlined in this section is mutually exclusive with cloning the provided repository and importing the artifacts. If the attached repository has been cloned into the Business Central environment, remove this repository before following these steps.

To use Business Central once BPMS has started, point your browser to http://localhost:8080/business-central and log in as a user with *ADMIN* privileges:

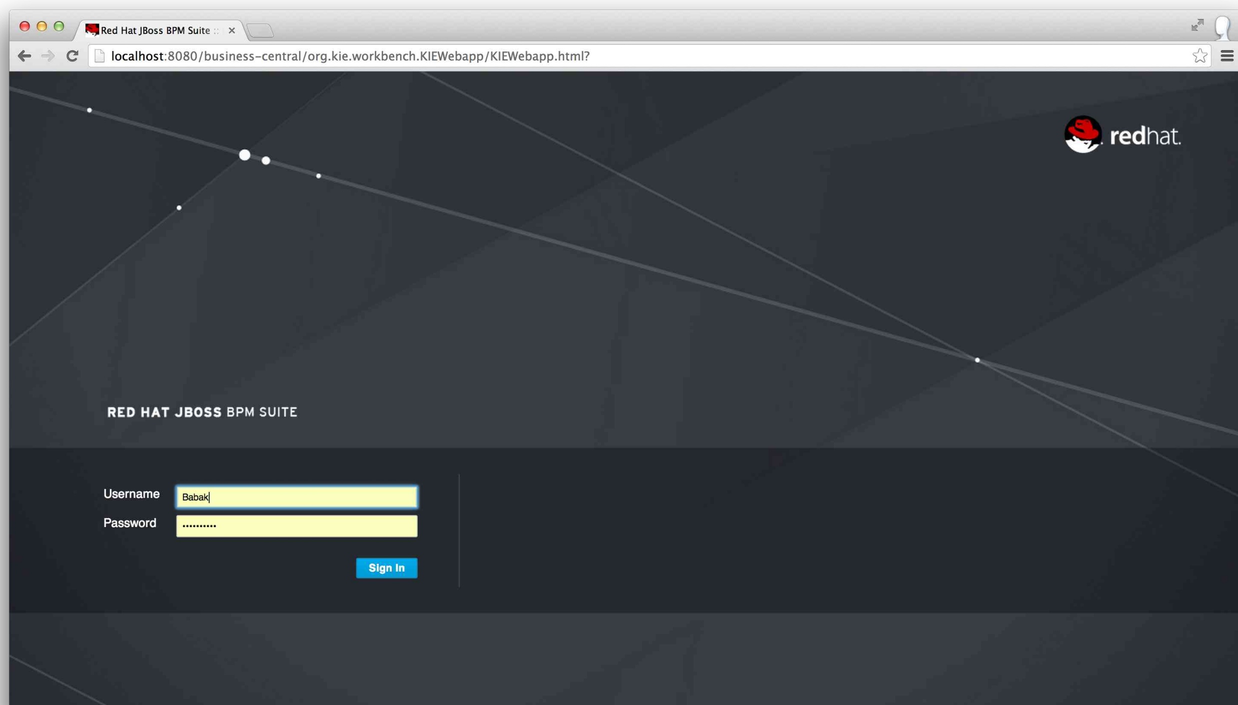

Figure 5.2.1-1:

Informational dialogs are provided on various screens of Business Central:

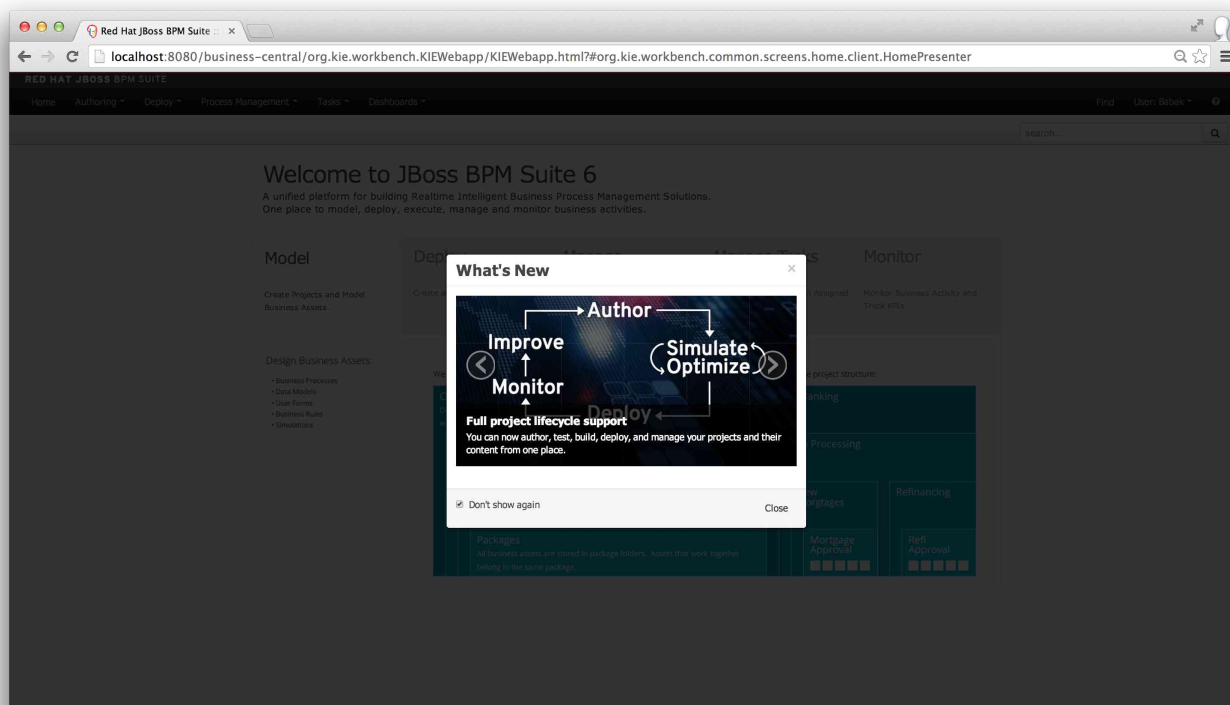

Figure 5.2.1-2:

Click the *Don't show again* checkbox before pressing close, to prevent a dialog from being displayed in the future.

The welcome page of Business Central provides a high level overview of its various capabilities in several tabs. The top-level menu provides persistent navigation across various pages and sections.

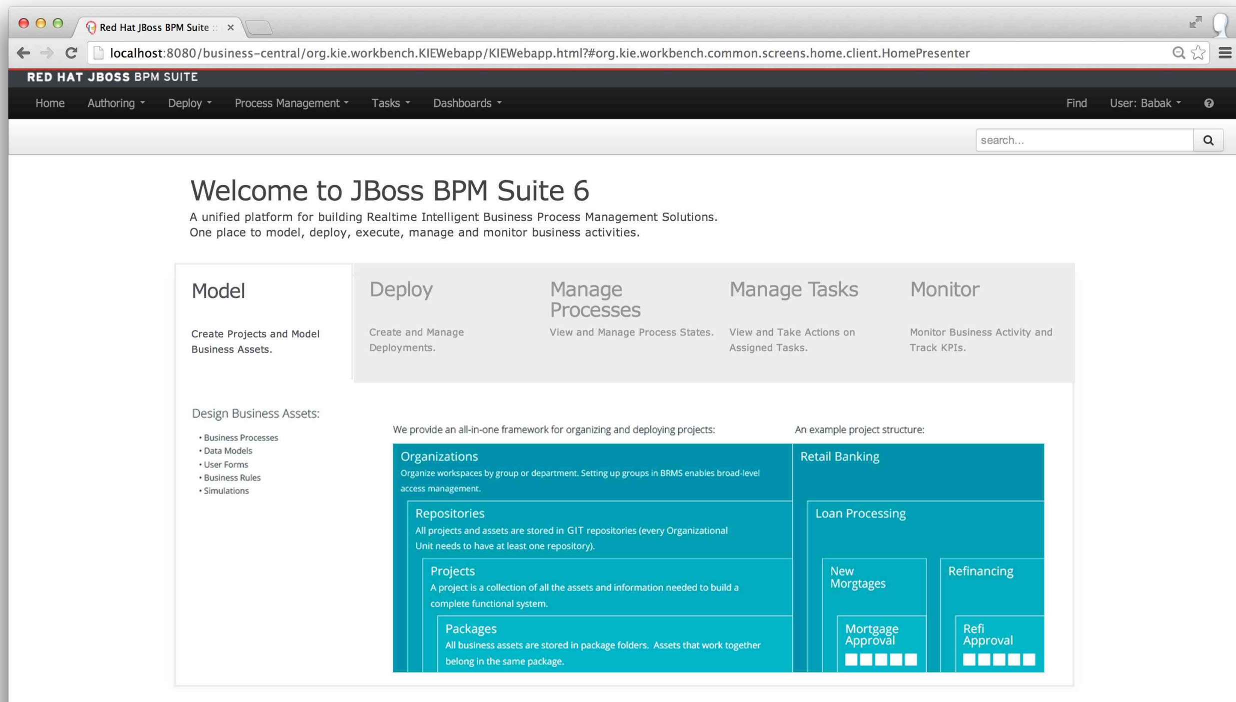

Figure 5.2.1-3:

5.2.2 Repositories

Business Central stores all business rules, process definition files and other assets and resources in an asset repository (knowledge store), which is backed by a **Git** repository. This makes it possible to import an entire knowledge store by cloning a **Git** repository or interact with the knowledge store through its **Git** URL.

To set up a new repository, navigate to *Administration* from the *Authoring* menu:

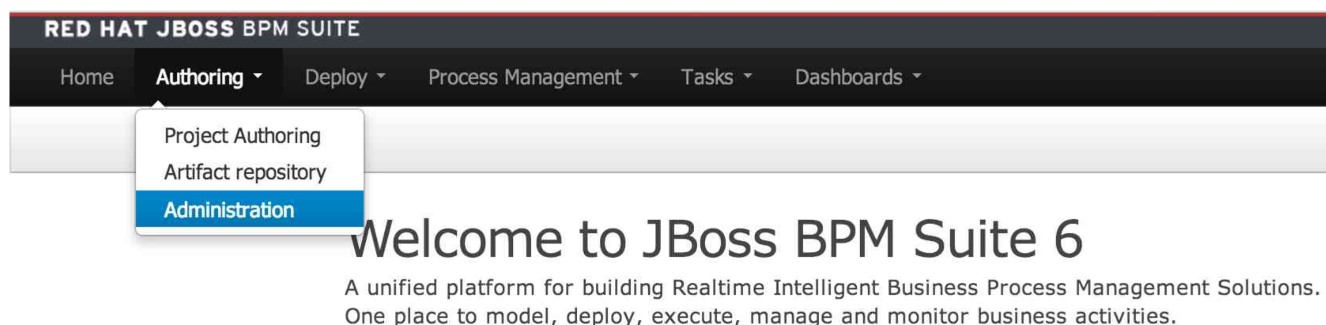

Figure 5.2.2-1:

From the *Administration* screen, select *New repository* from the *Repositories* menu.

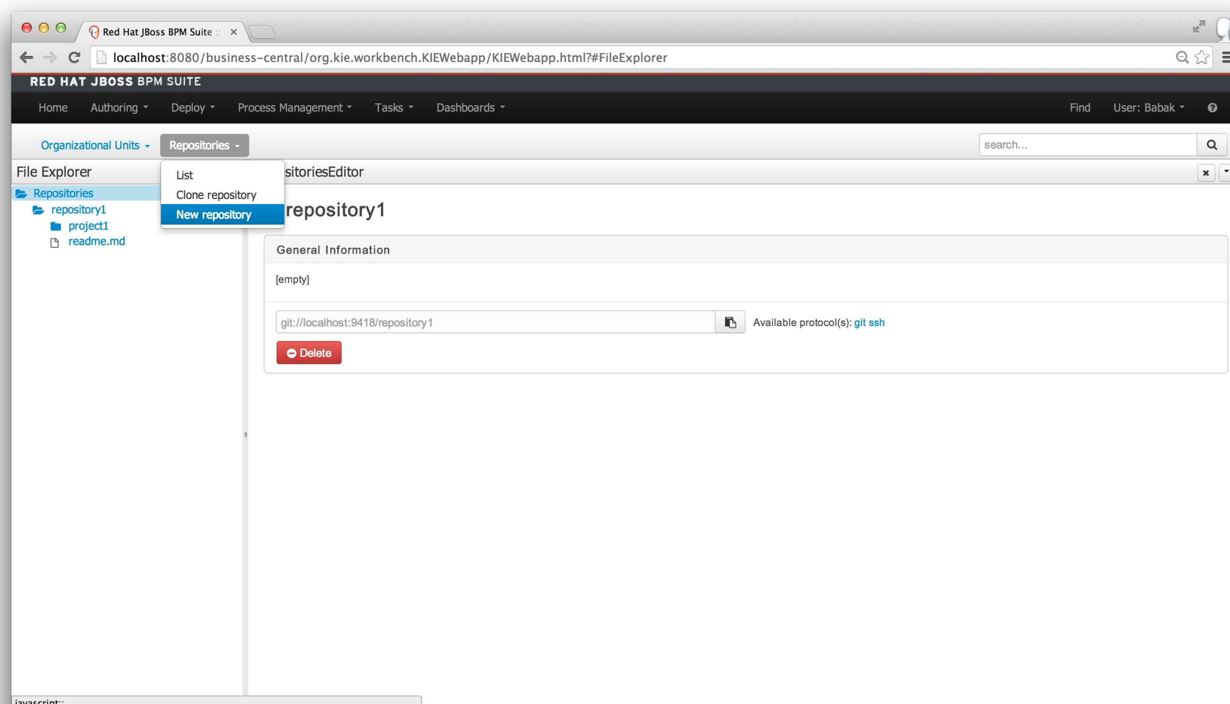

Figure 5.2.2-2:

Name the new repository "Mortgage" and assign it to the "example" organizational unit.

Create Repository
Repository Infomation * is required
* Repository Name
Mortgage
* Organizational Unit
example
Cancel
Create

Figure 5.2.2-3:

After creating the *Mortgage* repository, the *Administration* view will display both repositories and the URL to access each of them through **`Git`**.

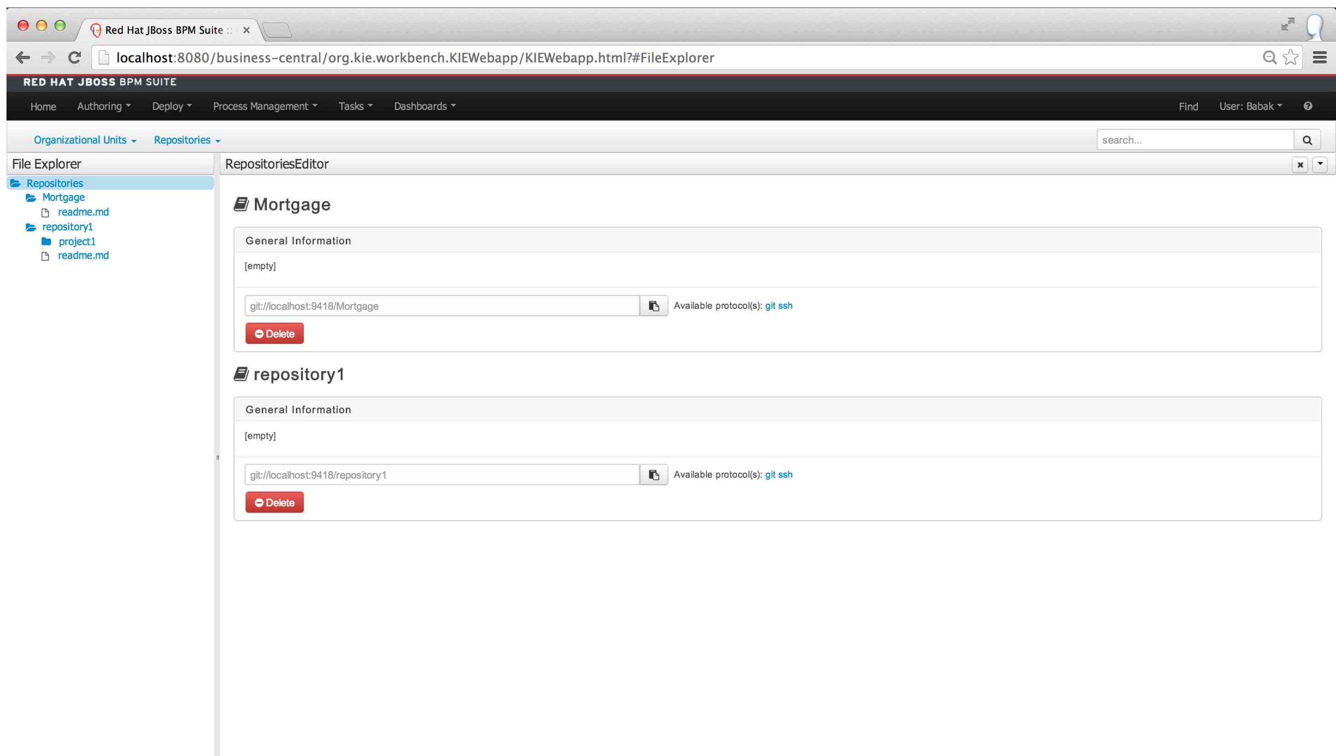

Figure 5.2.2-4:

5.2.3 Projects

Once a repository is created, the next step is to create a project inside that repository.

Select *Project Authoring* from the *Authoring* menu and switch the current repository to *Mortgage:*

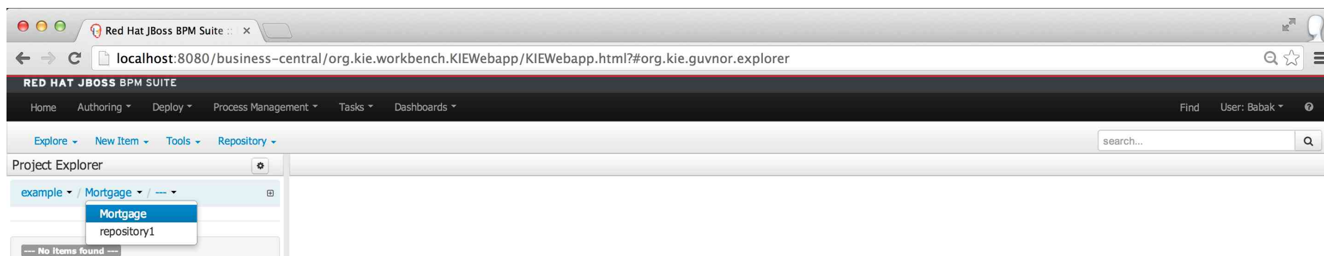

Figure 5.2.3-1:

Open the *New Item* menu. In the absence of a project, the only active option is to create a new project:

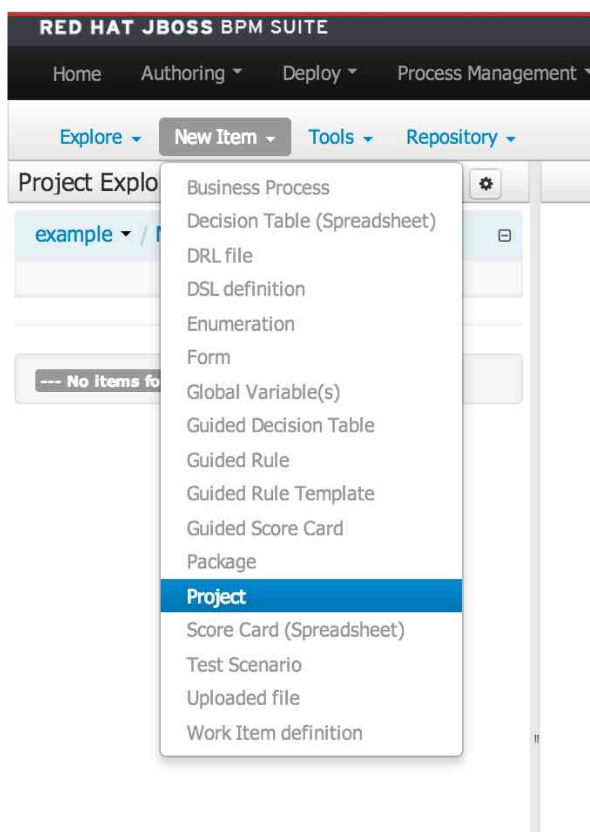

Figure 5.2.3-2:

Create a project called *MortgageApplication*:

Figure 5.2.3-3:

Enter a brief description of the project and fill out the maven artifact information. The example application uses a group ID of *com.redhat.bpms.examples*, artifact ID of *mortgage* and version ID of *1.0*.

Figure 5.2.3-4:

5.3 Data Model

A BPMS project typically contains many different types of assets. While the dependencies of these assets can often be complicated, the data model is almost always the most basic building block.

The BPMS web designer includes a web-based custom graphical data modeler that is accessed from the Tools menu:

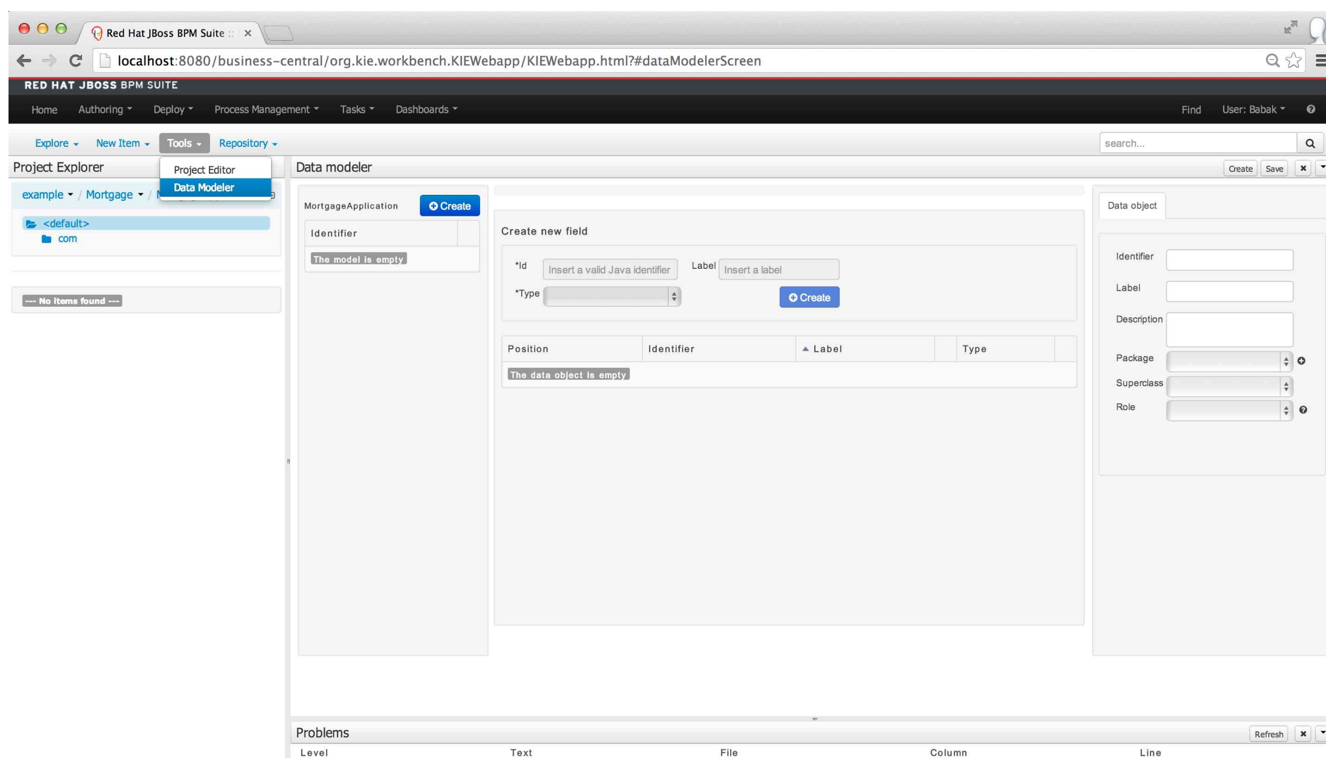

Figure 5.3-1:

Use the data modeler to create the required POJO definitions. For example, this project requires a class called Applicant to represent the mortgage applicant information. The identifier is the Java class name while the label is a more user-friendly name for the type. Create the required data model under the existing *com.redhat.bpms.examples.mortgage* package:

Figure 5.3-2:

Once a type has been created, proceed to define its fields. Similar to the type itself, each fields also has an identifier and a user-friendly label. The type of each field can either be a primitive or basic data type, or a custom type that has been previously creating using the data modeler or imported into the project. For example, an applicant would have a name which is in basic string format:

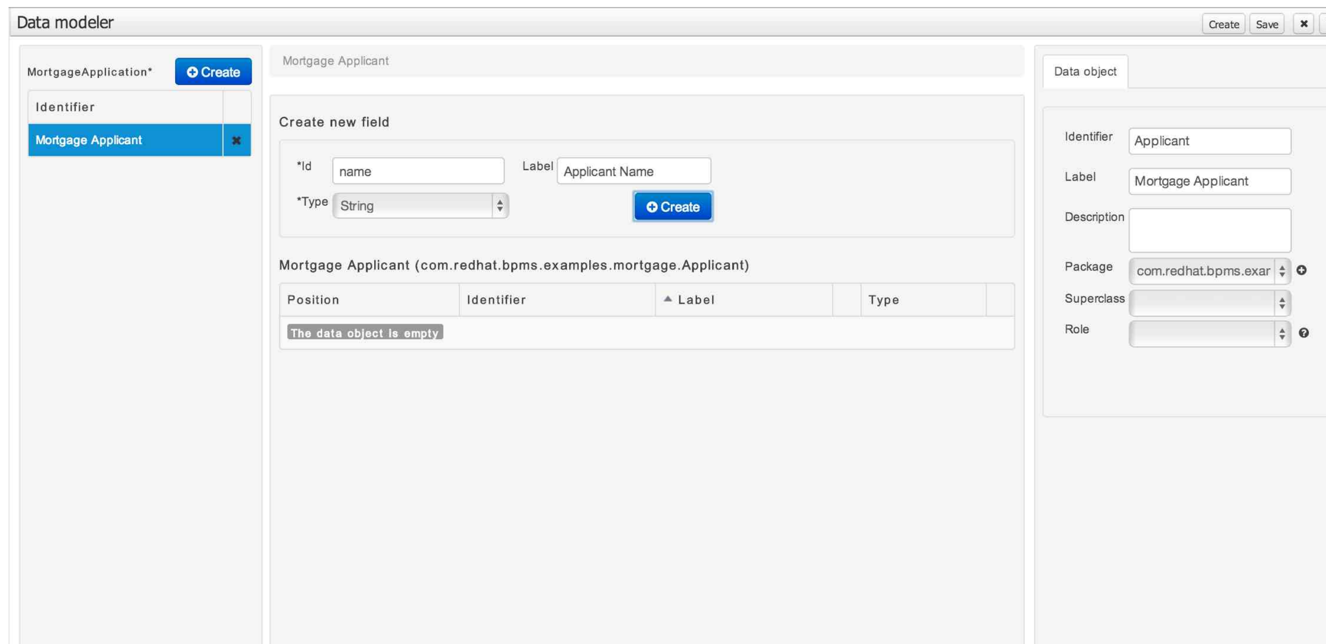

Figure 5.3-3:

Similarly, proceed to create the entire data model using the graphical tool. The following custom data types are required for the mortgage example:

Id	Label	Type
name	Applicant Name	String
ssn	Social Security Number	Integer
income	Annual Income	Integer
creditScore	Credit Score	Integer

Table 5.3-1: Applicant

Id	Label	Type
address	Property Address	String
price	Sale Price	Integer

Table 5.3-2: Property

Id	Label	Type
cause	Cause of Error	String

Table 5.3-3: ValidationError

Id	Label	Type
property	Appraised Property	Property
date	Appraisal Date	Date
value	Appraised Value	Integer

Table 5.3-4: Appraisal

As described in the Table 5.3-4: Appraisal, the property field of this data type is a type previously defined in the same data model, in Table 5.3-2: Property. As new data types are defined, they are added to the custom graphical editor for use in defining fields:

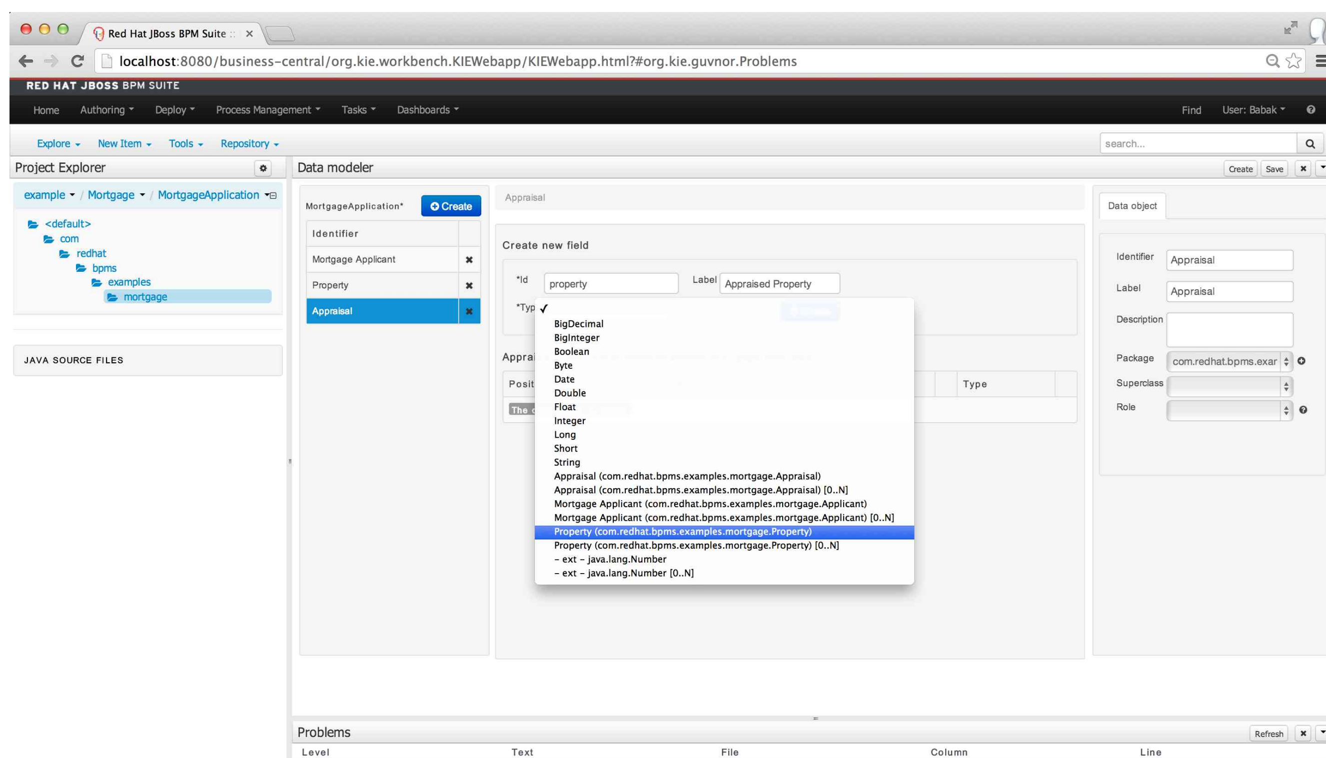

Figure 5.3-4:

The high level *Application* POJO holds and uses all of the above-defined types:

Id	Label	Type
applicant	Applicant	Applicant
property	Property	Property
appraisal	Appraisal	Appraisal
downPayment	Down Payment	Integer
amortization	Mortgage Amortization	Integer
mortgageAmount	Mortgage Amount	Integer
apr	Mortgage Interest APR	Double
validationErrors	Validation Errors	List<ValidationError>

Table 5.3-5: Application

Over all, the data model consists of five custom data types, each with a number of basic fields and two using custom fields of types previously defined in this same data model:

Figure 5.3-5:

5.4 Business Process

5.4.1 Create New Process

Navigate to the *com.redhat.bpms.examples.mortgage* package, where the data model was created. From the New Item menu, select Business Process:

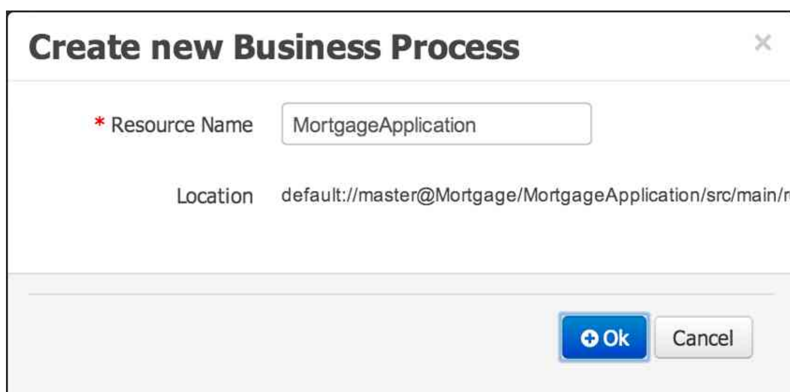

Figure 5.4.1-1:

Creating the process automatically opens the web process designer and provides a blank canvas to start the process design:

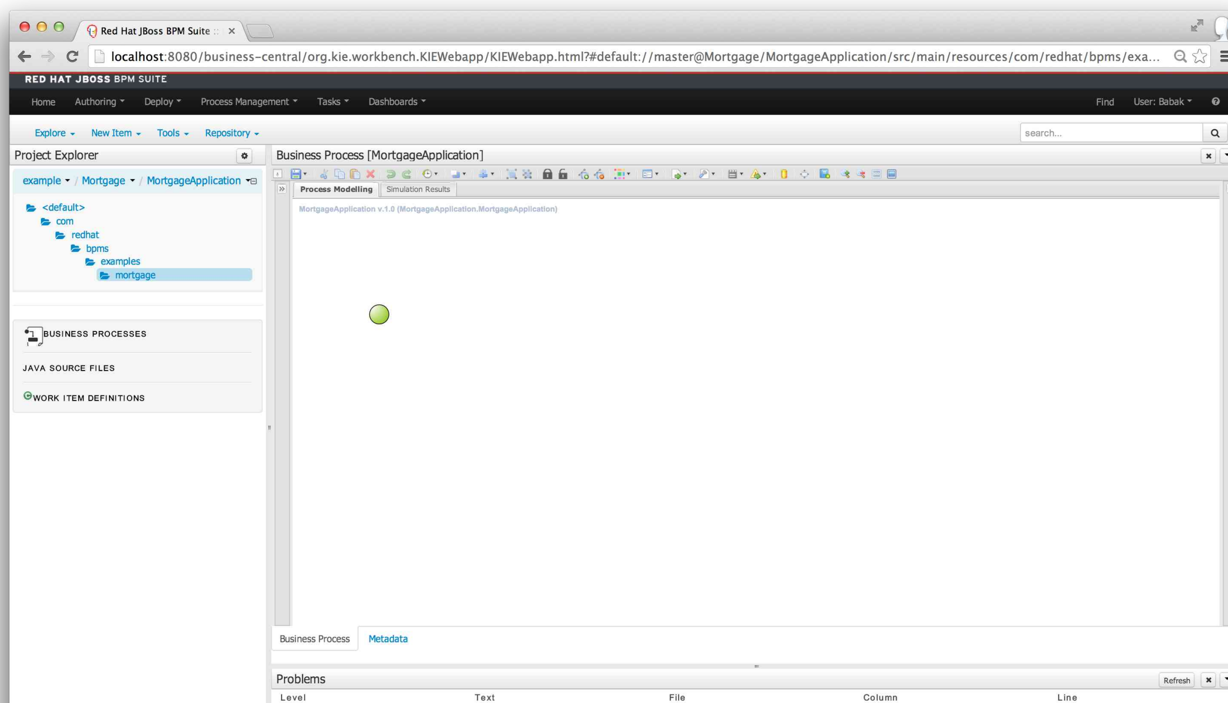

Figure 5.4.1-2:

Designing a business process is an iterative approach. The starting point is often an existing manual process that is documented and refined in multiple stages. For the purpose of this reference architecture, it is safe to assume that a detailed conceptual design is available. Even when the design is known with a great level of depth in advance, proceeding to model it in one or two quick steps is rarely a good idea.

Start by creating a very simple process that is structurally complete and can be executed and tested. For this example, place a script node after the start node and complete the process by connecting this script node to an end node.

Create a process variable representing a mortgage application. The *Application* class, created in the *com.redhat.bpms.examples.mortgage* package in the data modeler serves this purpose. Call this process variable, *application*.

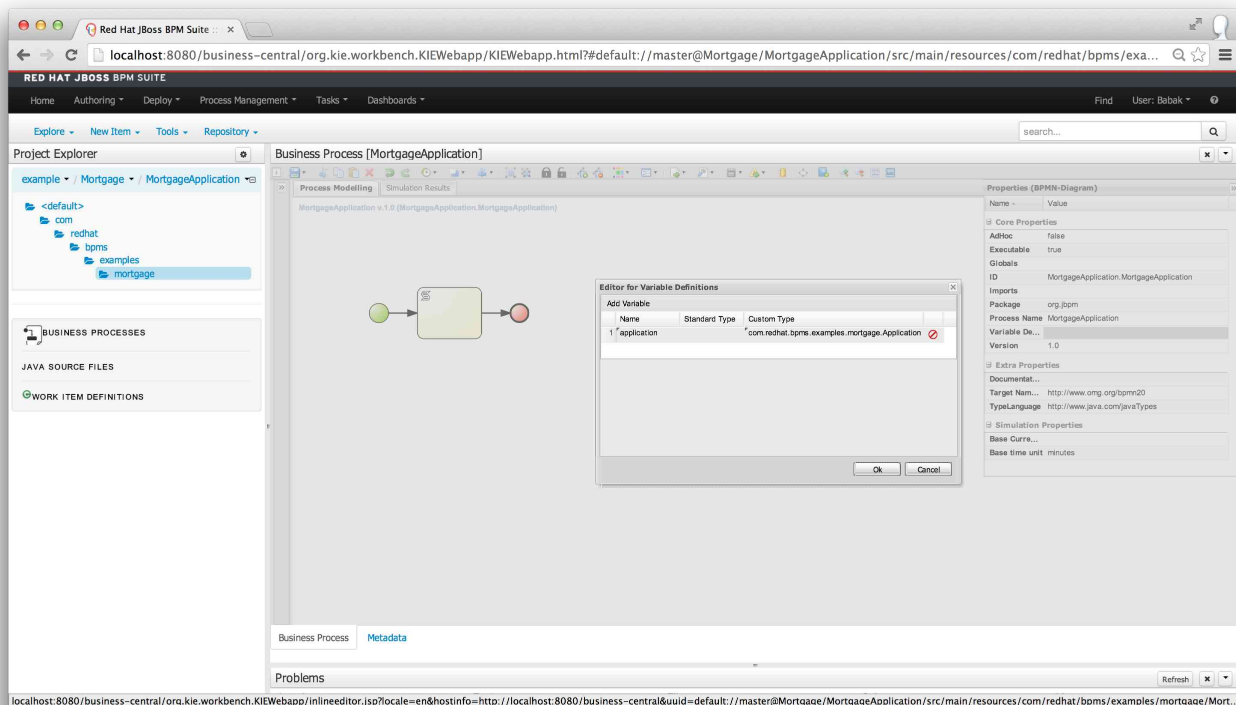

Figure 5.4.1-3:

Before the process can be saved, make sure to review the process properties. Assign the same *com.redhat.bpms.examples.mortgage* package to the process itself; ideally also as add the full package name as a prefix to the process ID:

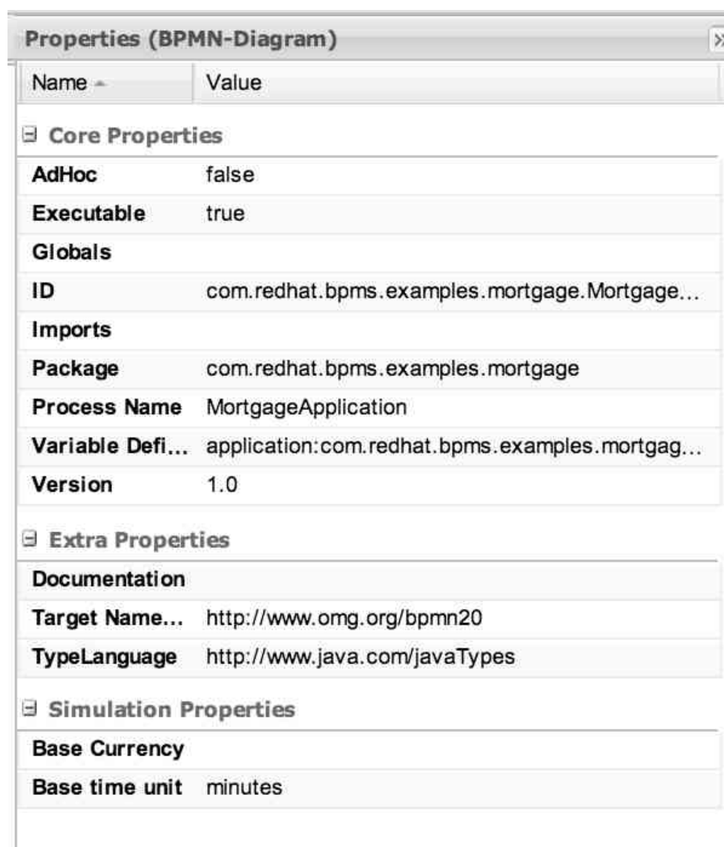

Figure 5.4.1-4:

Save the process and provide a meaningful explanation of the changes thus far:

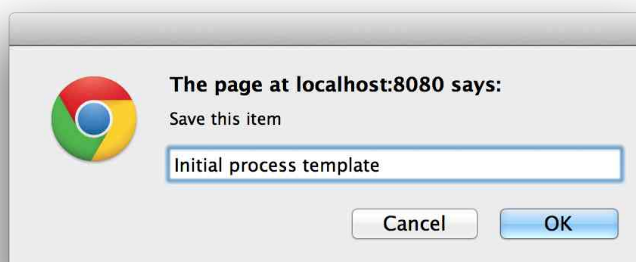

Figure 5.4.1-5:

5.4.2 Start Process Form

Now that the process skeleton has been created and saved, a process form is required to provide values for the mortgage application and test the process. Refer to the steps in the section on Process Form to create the process form.

Use a print statement in the script node to print out the application object.

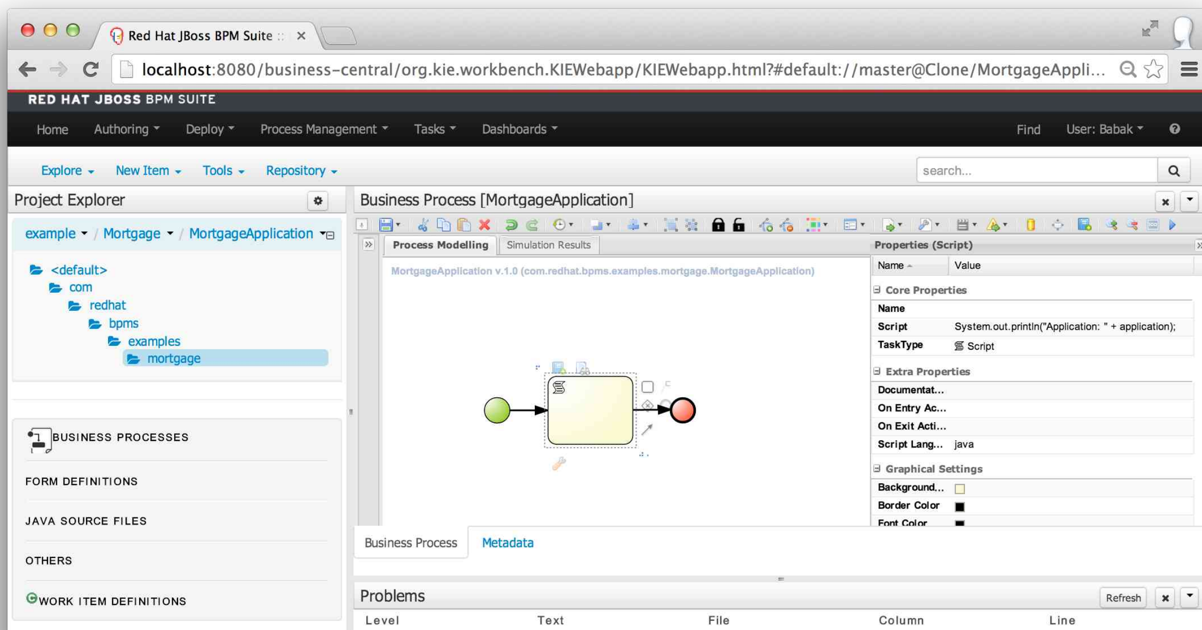

Figure 5.4.2-1:

Save the process, go to Tools, Project Editor, and build & deploy the project. Once built successfully, go to Process Management, Process Definitions and find the process listed there. Click the play button to start an instance of this process:

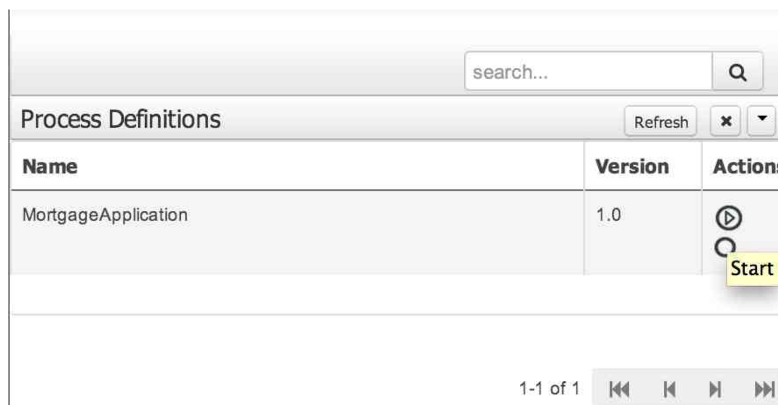

Figure 5.4.2-2:

If a process form is created, it will be opened by Business Central and allow the user to provide values for the mortgage application:

Form

MortgageApplication

Mortgage Applicant

Name

Babak

Social Security Number

333224444

Annual Income

200000

Property

Address

12300 Wilshire Blvd

Sales Price

1000000

Down Payment

200000

Figure 5.4.2-3:

A play button at the bottom of this form submits the provided data and creates a new process instance. An instance of the Application Java class is automatically created and provided to the process. The script node of the process prints out the application object so in the server log (or standard output of the server process), a line similar to the following gets printed:

```
21:49:24,976 INFO  [stdout] (http-localhost/127.0.0.1:8080-14) Application:
com.redhat.bpms.examples.mortgage.Application@6b08f5bc
```

To see the values stored in the application object, we could edit the Application class (which is merely an annotated **JavaBeans** class) and add a *toString()* method to it. It should be noted however that a subsequent edit by the data modeler would remove any such enhancements. The easier and more permanent alternative is to use the get methods to print the contents of application.

5.4.3 Validation

It is best practice to validate any input data. In a business process, the importance of this step is increased by the ability to correct data through human interaction.

Rule engines are often a good fit for data validation, as they allow validation rules to be stated individually and be enforced in concert, while making it easy to update each rule. BPMS includes a world-class rule engine, which makes the use of a rules for validation an obvious choice.

It is also best practice to maintain the process in a valid state as often as possible through the development. New features can be added step by step, then saved, built and tested before considering the next item on the agenda.

Move the script node and the end node to the far right of the canvas, break the flow from the start node and an *XOR gateway* instead.

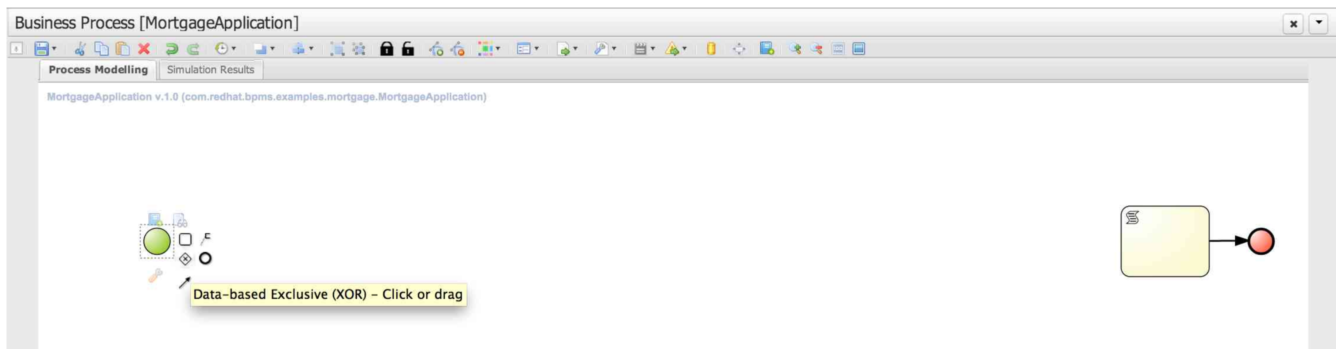

Figure 5.4.3-1:

Follow that up with a task node, that you then configure into a *Business Rule* task:

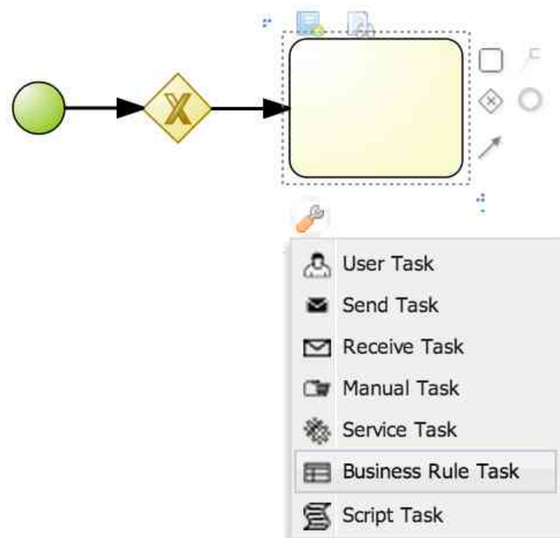

Figure 5.4.3-2:

Place another XOR gateway after the business rule task. This new gateway will be used to direct the process flow into two separate directions:

- The process will continue executing and moves forward if the data is valid.
- The process will go through correction if data is invalid; process flow will loop back into the first gateway, where it will undergo validation again.

For the valid case, use a sequence flow to connect the gateway all the way to the script task that was previously created to print the application data.

Figure 5.4.3-3:

The first gateway, which is the second node of our process, is a converging gateway. It expects two or more incoming sequence flows but regardless of how the process ends up at this step, provides a single outgoing flow to the next node. The second gateway (fourth node) in this diagram is a diverging XOR gateway, which accepts a single incoming sequence flow but has two or more outgoing flows. Java conditions or business rules determine which sequence flow is taken but in an XOR gateway, one and only one outgoing flow will always be chosen.

To place the data correction flow above the main flow, select all the existing nodes by drawing a large rectangle around them in the canvas. Once all the nodes are selected, drag one of the nodes and move it down, leaving sufficient room for at least two other rows above the main sequence where the nodes are currently located. When several nodes are selected, moving one of them moves all the selected nodes at the same time.

From the diverging XOR gateway, create a second outgoing sequence flow that connects it to a new task node. The easiest way to do this is to click on the gateway and wait for the web designer shortcuts to appear and then choose the rectangular node from the shortcut pallet. Then move this new task node directly above the diverging XOR gateway. Use the tools icon to turn it into another business rule task.

From this new business rule task, create another task node and place it to its left, directly above the converging (first) gateway. Change the type of this new node to a *User Task*. Use the sequence flow from this new user task to connect it back to the converging node directly underneath it.

At this point, the process is almost complete but a few refinements are required before the project can be built. Additionally, for the purpose of modeling and readability, it is important to name some of these elements.

To name an element or sequence flow in the web designer, simply double-click on the item in question and type the name. At minimum, label the first business rule task as *Validation*, the second business rule task as *Reset Validation* and the user task as *Data Correction*. It is also helpful to label the two outgoing sequence flows from the diverging XOR gateway as *Valid* and *Invalid*.

Figure 5.4.3-4:

The validation node links to business rules provided in the same package. Rules are designated as part of a rule flow group to be associated with a business rule task. Click on the validation node and edit its properties. Set the *Ruleflow Group* property to *validation:*

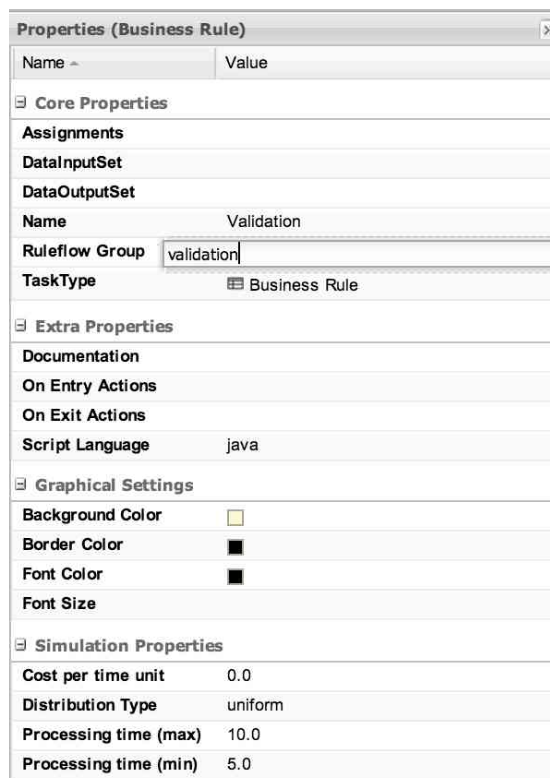

Figure 5.4.3-5:

Similarly, set the rule flow group for the second business rule task to *resetValidation*.

A business rule task can be thought of as an external service with a loose contract. The skill set of the process modeler may in fact be considered distinct from the skill set of a rule analyst. For validation, the data model serves as the common ground to define the interface. Rules require a number of facts to have been inserted into the rule engine's working memory. In this instance, instances of the following custom types must be inserted:

- *Application*
- *Applicant*
- *Property*

Validation rules apply constraints to these objects and their fields. The assumed contract is that in the instance that a validation rule is found to be violated, a *ValidationError* object would be generated and inserted into the working memory.

Refer to the Business Rules section to explore the rules used for Validation and Reset Validation.

Based on this validation contract, click on the *Invalid* sequence flow and set its conditions. Change the condition expression language to *drools* and the expression itself to *ValidationError()* so that this sequence flow is taken when the rules have instantiated an instance of *ValidationError*.

Conversely, set the expression for the *Valid* sequence flow to *not ValidationError()* while also choosing *drools* as the language.

Properties (Sequence Flow)

Name	Value
Core Properties	
Condition Expression	not ValidationError()
Condition Expression Language	drools
Name	Valid
Priority	
isImmediate	false

Figure 5.4.3-6:

Select the *Validation* task node and open the dialog for *On Entry Actions*. These actions are lines of Java code that execute before the node itself. Insert *Application*, *Applicant* and *Property* into the rule engine working memory:

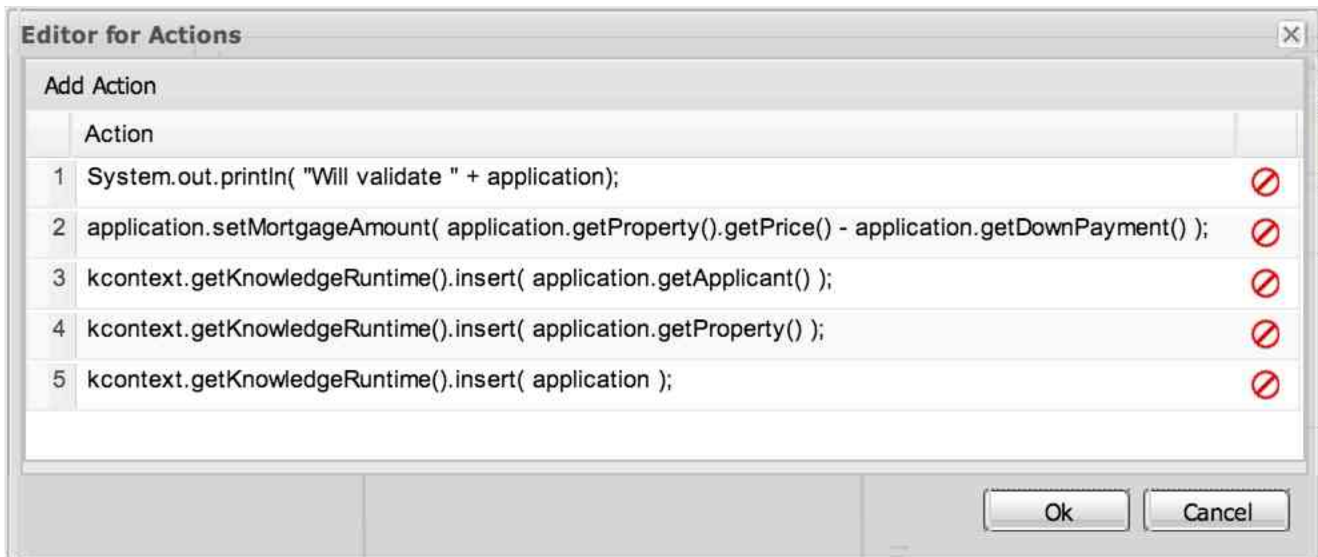

Figure 5.4.3-7:

Notice that the application also contains a field to represent the mortgage amount. This is a derived value, based on the property sale price and the down payment. Setting this value on entry of the validation task is not ideal but there are a number of constraints and each other option has its own disadvantages:

1. *getMortgageAmount()* can be written as a utility method that does the subtraction upon request. The big disadvantage of this approach is that it is not compatible with the data modeler and even if the class is manually modified to add such a method, a future update to the data type through the data modeler may overwrite it.
2. A rule can be written to calculate the mortgage amount and update the application object with it. Such a simple rule does not merit its own business rule task in the process and including it in validation rules is a poor choice. That implies that this value is only needed for validation, which is not the case.
3. Creating a separate script task (or for that matter, an earlier business rule task) to calculate the mortgage amount exposes this step as part of the model. The business process model should remain high level and exclude trivial and technical steps.
4. Data correction may indirectly result in a change to the mortgage amount so the subtraction must occur within the validation loop and the amount cannot be calculated earlier, at the time of initial data collection or its processing.

5.4.4 Data Correction

Data correction in the process is performed by a mortgage broker through the human interaction features of BPMS. For this purpose, a user task is created and assigned to the *broker* group. By assigning a task to a group, as opposed to a user, a certain degree of loose coupling between the work and the worker is achieved. Any broker who is available can claim or be assigned the created task and through the use of *swimlanes*, it can be mandated that the same specific user work on future tasks for this process instance, so that a desired degree of continuity is provided to the customer.

Edit the properties of the *Data Correction* task node. Set the *Task Name* to *DataCorrection*. This attribute is the technical name of the task, as opposed to its display name, which has already been entered into the model.

Set the *Groups* attribute to *broker* so that the task may be assigned to any broker.

Open the *DataInputSet* and add an input variable for the task. This data item will be provided to the human actor reviewing and completing the human task. Provide the entire mortgage application data to the broker by declaring an input variable with a distinct name of *taskInputApplication* and type of *com.redhat.bpms.examples.mortgage.Application*.

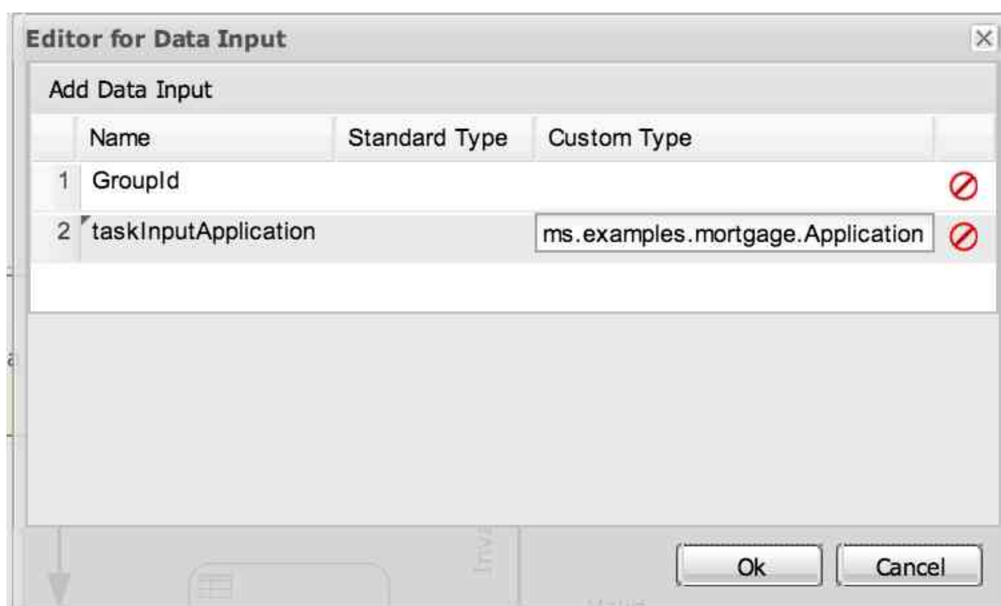

Figure 5.4.4-1:

The broker will review the data provided as part of the mortgage application and make any necessary corrections so that it would pass validation next time. To correct the data, the broker may need to contact the applicant or other intermediaries. Human tasks allow automated processes incorporate such manual steps.

Similar to the input variable, create an output variable for the data correction task to receive the corrected mortgage application. Declare the output variable with a distinct name of *taskOutputApplication* and type of *com.redhat.bpms.examples.mortgage.Application*.

Finally, open the assignments property of the task and map the process variable to the task variables:

1. Map the application variable of the process to taskInputApplication so that the broker can view the data and proceed to correct it.
2. Map taskOutputApplication back to the application variable of the process so that any corrections by the broker are applied to the process.

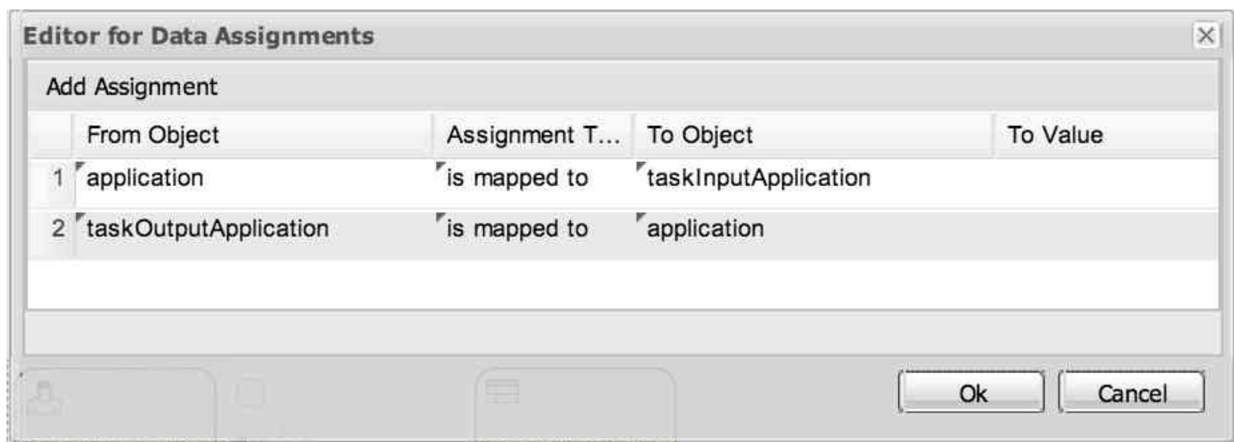

Figure 5.4.4-2:

Make sure to save the process and then navigate to *Tools* / *Project Editor* and make sure that *Build & Deploy* is successful.

When Business Central is used to work on human tasks, a task form is typically required for every user task node created in a process. Click on a task node to create or edit a corresponding task form:

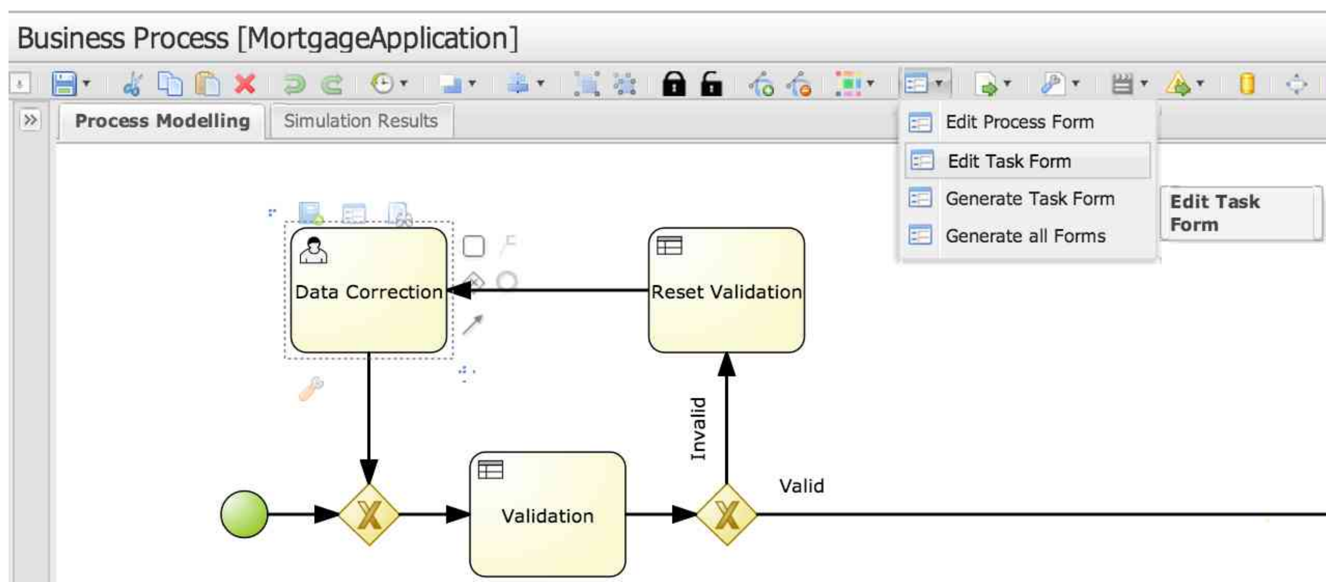

Figure 5.4.4-3:

Refer to the section on the Data Correction task form for details on the design and implementation of the form itself.

5.4.5 Web Service Task

The next step in processing the mortgage application is to determine the applicant's credit score. This application assumes an external Web Service that takes the applicant's social security number and returns their credit score. The simple Credit Report Web Service has been created for this purpose.

From the left side of the palette, open *Service Tasks* and drag the *WS* service task on the canvas:

Figure 5.4.5-1:

Rename the ws node to Credit Report and place it after the diverging XOR gateway so that the applicant's credit score is requested after the mortgage application data is validated. Drag the end point of the existing valid sequence flow to this new node and draw a new sequence flow from this service task node to the script task.

So far, the only required process variable has been the application variable, which holds all the required data within it. At this point, proceed to create process variables representing the input and output of the Web Service. As a reminder, process variables are declared in the web process designer by clicking on the canvas background to get access to the process properties.

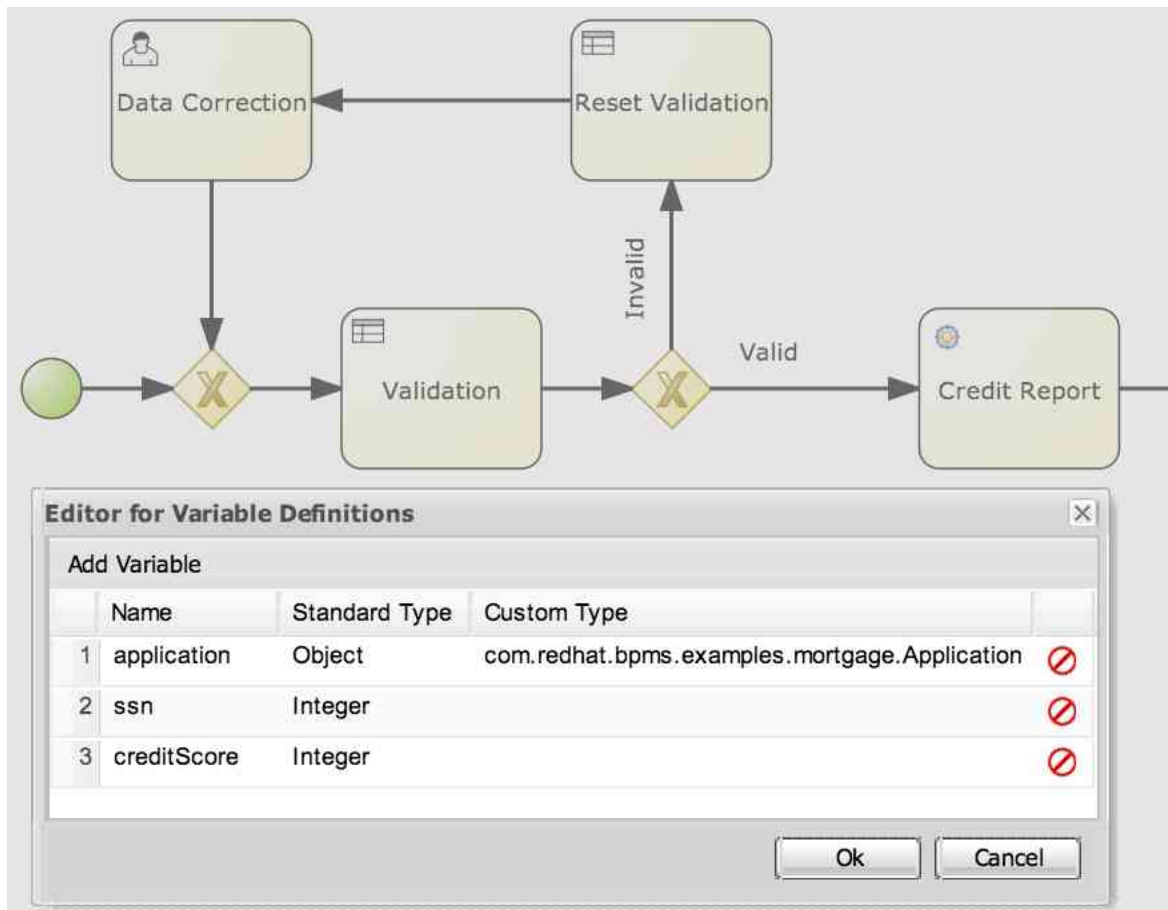

Figure 5.4.5-2:

Next, click on the Credit Report task node and add the follow lines of code as actions to be executed before and after the node.

On entry:

```
kcontext.setVariable( "ssn", application.getApplicant().getSsn() );
```

On exit:

```
application.getApplicant().setCreditScore( creditScore );
```

As the code clearly states, the ssn process variable is set up from the application object for the purpose of the web service call and once the credit score is retrieved, the application object is updated with its value.

Edit the data output set of the web service task. The result is configured as a generic object by default; modify its standard type to Integer and remove the custom type, to better represent the returned credit score value. Also edit the data input set of the web service and configure the web service parameter as a standard Integer, which is the correct type for ssn.

Use the variable assignments of the task to configure the URL, namespace, service name, service mode and operation name. Also map from the process variables to designate the applicant's ssn as the service parameter and the returned value as the credit score:

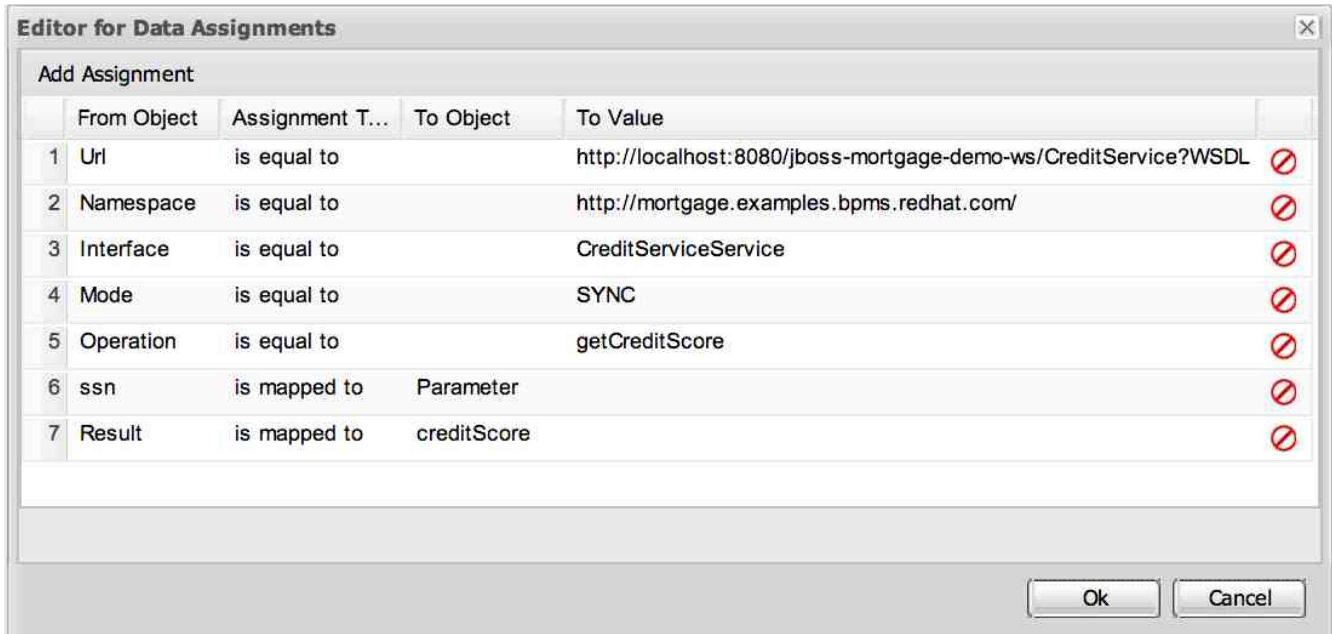

Figure 5.4.5-3:

Save the process and provide a meaningful description for its repository revision. At this point, the project editor can be used to build and deploy the project and starting a process instance should result in a credit score being (mock) calculated by the web service, assuming this service is created and deployed as described in the section Credit Report Web Service.

5.4.6 Error Handling

So far, conceivable error conditions could have arisen from invalid input data and the validation rules resulting a human data correction have been an adequate response to such errors. Calling a web service introduces new risks. The external service may be down and nonfunctional for unexpected reasons. Various communication, network and server errors may result in an invalid response.

To catch potential errors from the Credit Report service task, open the *Catching Intermediate Events* set of tools from the web designer palette and drag the Error event onto the canvas. Drop this node on the lower boundary of the service task; the boundary of the service task turns green to indicate that the event node is being dropped on the correct spot:

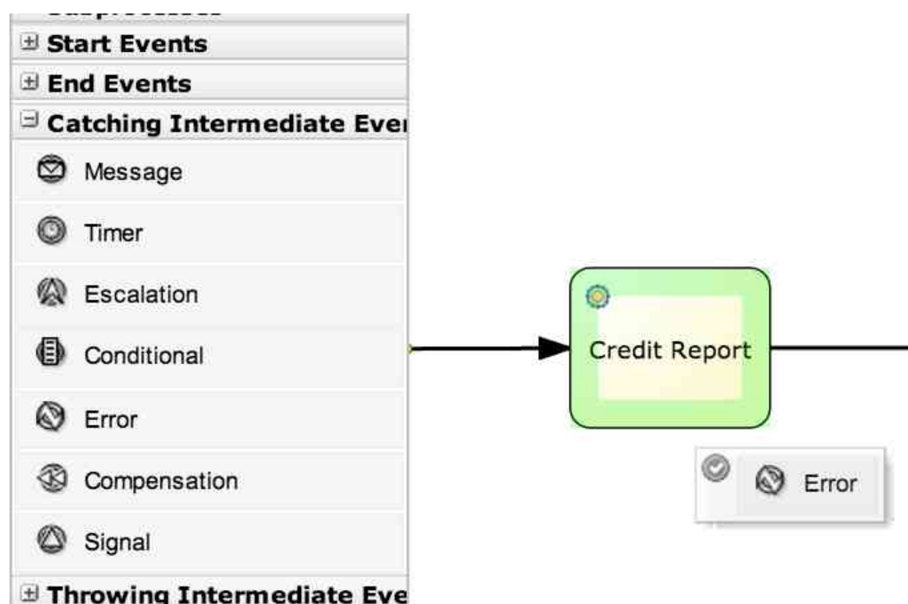

Figure 5.4.6-1:

The error event node catches any errors resulting from the execution of the service task. In a way, this node serves a purpose similar to the validation rules, in that once the error has been detected, the process provides a chance to inspect the error, remedy the situation and try again.

Add a data output variable to this boundary event node called *nodeError*. System errors results in a *WorkItemHandlerRuntimeException*, which will now be assigned to this new variable. Create a process variable called *wsError* and assign it the custom type of *org.jbpm.bpmn2.handler.WorkItemHandlerRuntimeException*. Open *DataOutputAssociation* on the error boundary event node and map from *nodeError* to *wsError*. This way, the thrown exception is made available to the process in the form of a process variable called *wsError*.

Create a new user task called *Troubleshoot* and assign it to the *admin* group. Draw a sequence flow from the error catching event to this user task.

Also place a new XOR gateway node before the Credit Report task and modify the existing Valid flow to connect to it. This new node then connects to the service task.

Now, connect the *Troubleshoot* user task to the new converging gateway. Similar to the data correction loop, this creates a troubleshooting loop where any errors from the Web Service call can be examined and corrected before looping and trying the call again. This is predicated on the Credit Report service being idempotent[21], as is the case here.

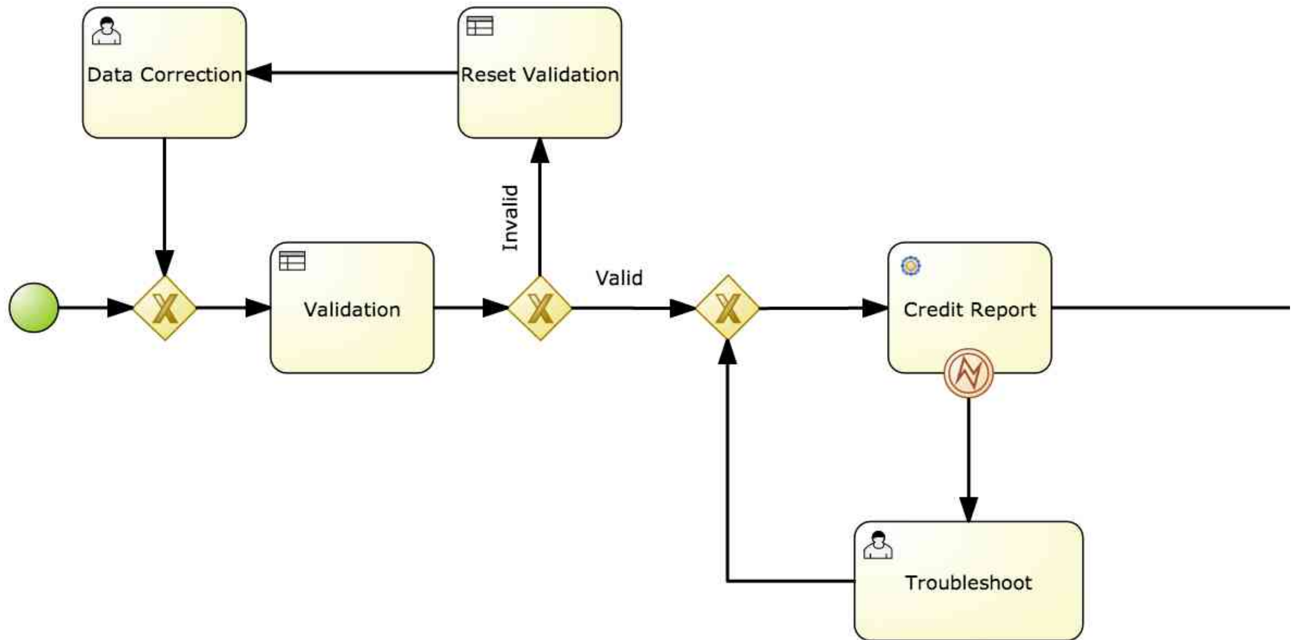

Figure 5.4.6-2:

Create a process variable of the standard type of String and call it *wsErrorStack*. The entire exception stack from the original exception will be converted to String form and stored in this variable. Do this by creating an On Entry Action for the *Troubleshoot* task and entering the following Java code as one action:

```
java.io.StringWriter errorStackWriter = new java.io.StringWriter();
wsError.getCause().printStackTrace(
                           new java.io.PrintWriter( errorStackWriter ) );
kcontext.setVariable( "wsErrorStack", errorStackWriter.toString() );
```

The cause of the system error is the actual exception that occurred during the web service invocation and in this case, its stack trace is retrieved and stored as a variable.

21 http://en.wikipedia.org/wiki/Idempotence#Computer_science_meaning

Create a Data Input variable for the *Troubleshoot* task and give it the name *errorStack* and standard type of *String*. In the task assignments, map from the process variable *wsErrorStack* to *errorStack*. This makes the stack trace of the thrown root exception available to the actor of this user task. To display the stack trace, create a task form for this user task.

This form only requires one field of type *Long text*; give the field the label of *Web Service Error* and call it something like *errorStack*. Set the *size* and *height* of the field to appropriate values for an exception stack (e.g., 200 and 10 respectively). Set the field as *read-only* and enter its *input binding expression* as *errorStack*, which is the name of the task variable.

Figure 5.4.6-3:

This task form helps provide a technical administrator more information about the cause of the error that occurred while calling the external web service. The administrator can review this information and use it to troubleshoot and correct the problem, before completing the task and having the process retry the web service call.

5.4.7 Mortgage Calculations

Once the applicant's credit score is determined, the next step is to assess the risk of the requested mortgage and determine the interest rate that can be offered to this applicant.

Such calculations are a natural fit for a rule engine. Using business rules to calculate the interest rate accelerates development and greatly reduces the cost of maintenance.

Once again, the only requirement is to insert the relevant objects into the rule engine's working memory. The mortgage amount is also recalculated to make sure it is updated with the correct value, as the down payment is subject to change (in sections that follow):

```
application.setMortgageAmount(
    application.getProperty().getPrice() - application.getDownPayment() );

kcontext.getKnowledgeRuntime().insert( application.getApplicant() );
kcontext.getKnowledgeRuntime().insert( application.getProperty() );
kcontext.getKnowledgeRuntime().insert( application );

if( application.getAppraisal() != null )
    kcontext.getKnowledgeRuntime().insert( application.getAppraisal() );
```

This time the rules operate directly on the application object by setting its apr field to the calculated interest rate value. The last set of rules clean up the working memory by retracting all the existing objects.

The process simply adds a business rule task, using the above lines of code as its on entry actions and settings its *ruleflow-group* attribute to *apr-calculation* so that Mortgage Calculations rules execute.

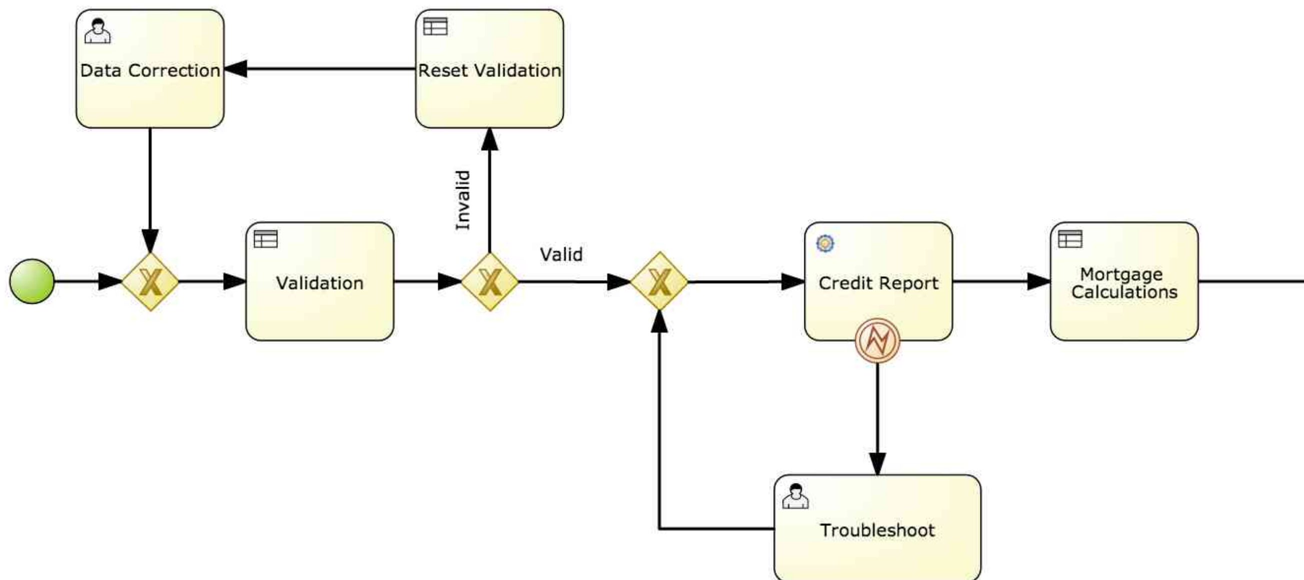

Figure 5.4.7-1:

5.4.8 Qualify Borrower

While the lending risk is already reflected in the calculated interest rate, there are also cases where the lender refuses to provide the applicant with a mortgage. One such case is the *front-end ratio* for the mortgage application exceeds the 28% threshold.

Calculating the *front-end ratio* is a simple matter of determining the monthly mortgage payment (this process ignores other housing costs) and dividing it by the applicant's income. This requires relatively simple arithmetics that is not a natural fit for a rule engine. For simplicity, this calculation can be performed in a script task.

Create a process variable called *borrowerQualified* of the standard type of *Boolean*. The script task will set this boolean variable to true or false, indicating whether the applicant is qualified for the mortgage based on the APR and front-end ratio.

Create a *Script Task* called *Qualify Borrower* and use the following code snippet as its *Script*:

```
System.out.println("Qualify Borrower");
double monthlyRate = application.getApr() / 1200;
double tempDouble = Math.pow(
                      1+monthlyRate, application.getAmortization() * 12);
tempDouble = tempDouble / (tempDouble - 1);
tempDouble = tempDouble * monthlyRate * application.getMortgageAmount();
System.out.println("Monthly Payment: " + tempDouble);
boolean qualified =
     (application.getApplicant().getIncome() / 12 * 0.28 > tempDouble);

kcontext.setVariable("borrowerQualified", Boolean.valueOf(qualified));
```

Place the new script task after the mortgage calculation business rules:

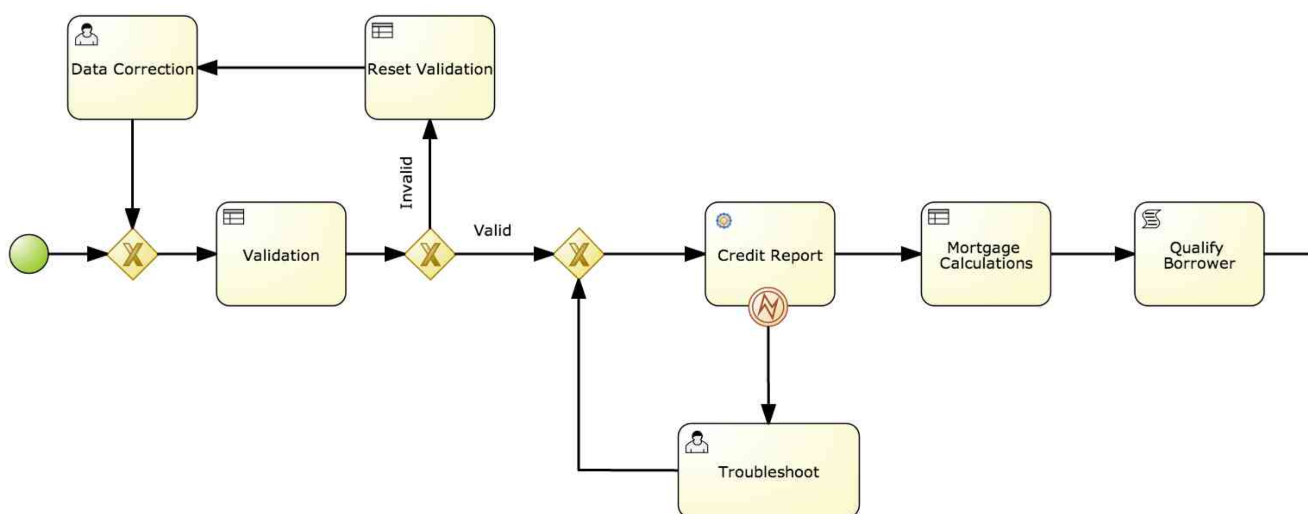

Figure 5.4.8-1:

5.4.9 Increase Down Payment

In an effort to avoid losing a business opportunity, the process explores alternative ways to qualify the mortgage applicant. One simple solution is to request a larger down payment.

Create a diverging *XOR gateway* after (preferably above) the *Qualify Borrower* script node. From this gateway, create two distinct sequence flows to handle the cases where the borrower may have been qualified or not qualified based on the monthly payment calculation.

Draw a sequence flow to the right and attach it to following nodes, at this point a node that simply prints out the application object and continues to terminate the process. Name this sequence flow *Qualified* and set a Java condition expression to verify that the *borrowerQualified* process variable is true:

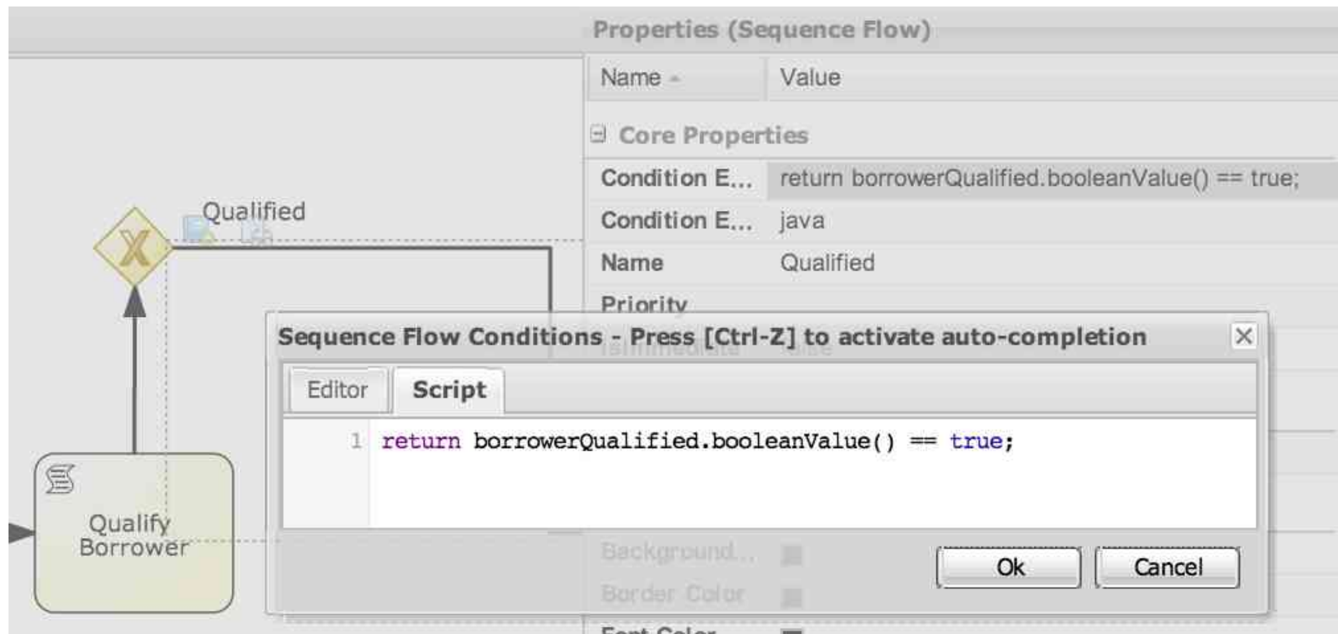

Figure 5.4.9-1:

Create a user task node on the left side of this gateway and draw a second sequence flow to it. Call this new sequence flow Not Qualified and set the condition expression language to Java again. The condition expression should look for *borrowQualified* to be false this time:

```
return borrowerQualified.booleanValue() == false;
```

The new user task will allow the business to contact the applicant and request a larger down payment to avoid declining the mortgage application. Once again, the task is assigned to the *broker* group and the input and output variables are *taskIntputApplication* and *taskOutputApplication*, which are both mapped to the application process variable. The task form for the Increase Down Payment task uses the application object to render the required data on the screen and updates the *downPayment* field of its output application variable

Once the down payment has been revised, mortgage calculations need to be renewed to determine if the applicant is now qualified. Create a new converging gateway between the credit report and mortgage calculations nodes to allow the process to join that flow and create a new loop to potentially continually revise the down payment amount upwards until the mortgage is qualified.

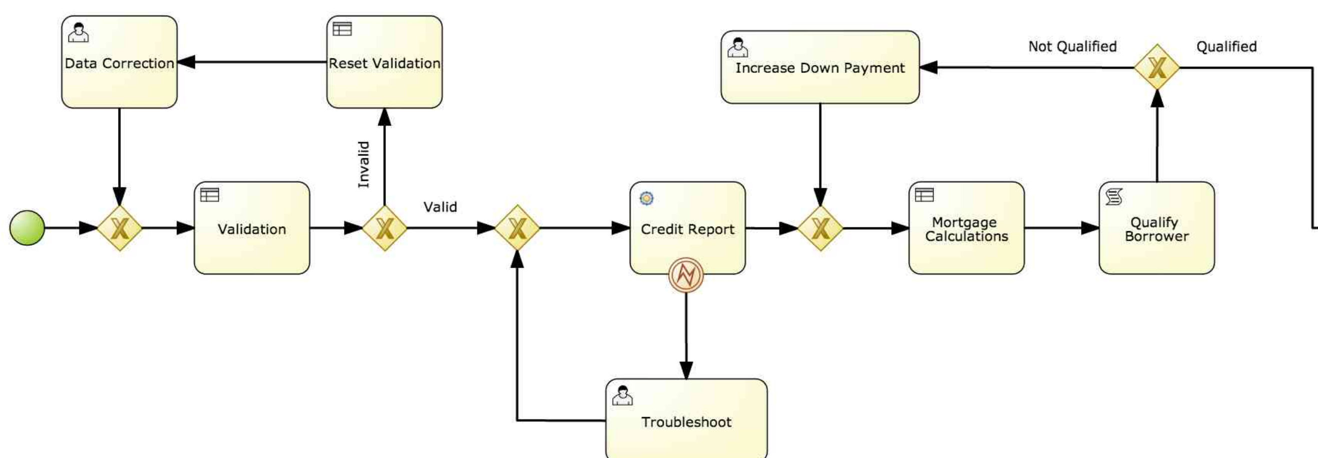

Figure 5.4.9-2:

While the loop allows the down payment to be increased continually without setting an arbitrary restriction on the number of loops, it also effectively removes the possibility of not qualifying a mortgage application. In other words, any application that is not qualified results in an infinite number of attempts to increase the down payment, even if the applicant is neither willing or able to do so.

A simple solution is to inspect the down payment and detect whether it has in fact been increased. Once the task fails to increase the down payment, a separate path may be taken to avoid an infinite loop scenario. To detect an increase in the down payment create a new process variable of standard type Integer and call it *downPayment*. On entry to the user task to increase down payment, add a new action that sets this process variable to the down payment value before its potential update by the user:

```
kcontext.setVariable( "downPayment", application.getDownPayment() );
```

Now that a different path can be taken for a mortgage that cannot be qualified, it would also be better process design to provide two distinct paths of mortgage approval and denial which would include two separate termination flows for the process.

Instead of immediately merging back into the main process flow, place a diverging *XOR gateway* after the user task to increase down payment. Create two outgoing sequence flows from this new gateway, where one merges back into the main process flow and gets back into the loop, but the other goes to a new script task node called *Deny Mortgage* and a new *End Event* after that. For the sake of consistency, also rename the previously created printing script task to *Approve Mortgage*.

The choice of sequence flow is based on whether the down payment was in fact increased or not. Accordingly, name the two sequence flows *Yes* and *No*. Use a Java condition expression that compares the previously recorded value of the down payment with its potentially updated value:

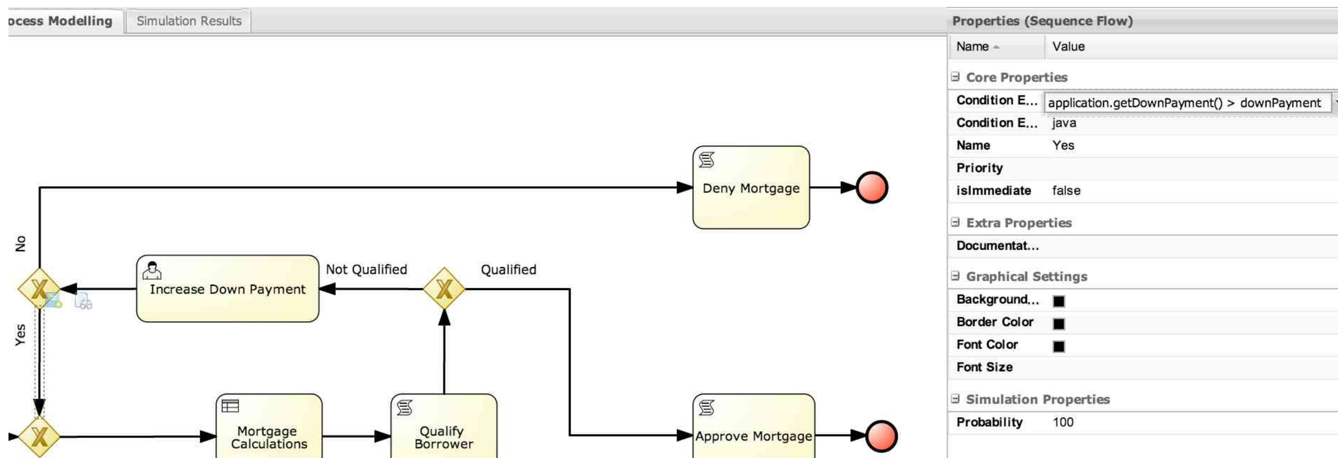

Figure 5.4.9-3:

This way, the loop continues while the applicant increases the down payment and repeated as long as it's not enough to qualify them, or until such point that the applicant declines to raise the down payment amount any further.

5.4.10 Financial Review

While the business process frequently solicits input from users, all the decisions so far have been automated. It is common for most business processes to have some sort of manual override. In this example, a manager may have the authority to approve a mortgage application that does not meet the standard criteria. As a last resort before declining the application, provide a user task form assigned to the manager group that allows a manager to approve the mortgage.

Create a variable called *brokerOverride* of the standard type of boolean. Once again map the application process variable to *taskInputApplication*, used by the task form, designed as instructed in the section on the Financial Review form, but this time allow the output to only be a boolean variable called *brokerOverrideTaskOutput* that maps back to *brokerOverride*.

Use this final decision in a diverging XOR gateway to decide whether to still proceed to decline the mortgage, or to approve it. Approving it means another converging gateway to merge the approval resulting from regular qualification with that of the manual override:

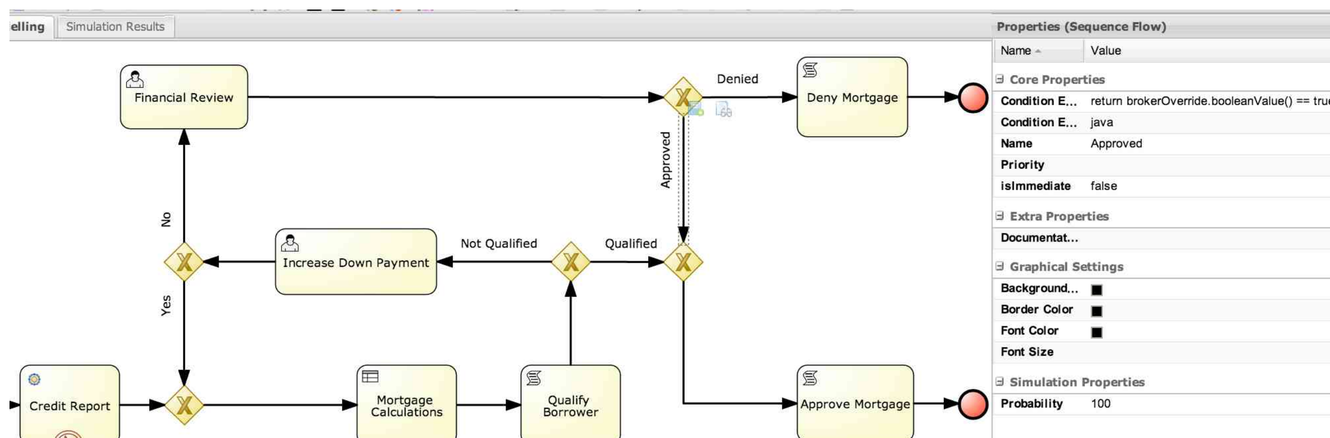

Figure 5.4.10-1:

5.4.11 Appraisal

Notice that up to this point, mortgage calculations have been based on the down payment as a ratio of the transaction price. One missing important step in this process is the appraisal of the property.

Property appraisal can be costly, so a well-designed business process delays incurring such a cost until other factors have all been taken into account. For this reason, the process adds the appraisal task as a last step before approving the mortgage.

However, appraisal does not act in a silo and much like most other steps of the business process, it can also result in a loop that affects other decisions. For example an applicant may easily qualify but the property maybe appraised at a lower value than the sale price. This necessitates a new round of calculation and as a result, the applicant may no longer qualify and need to increase the provided down payment. It may also be that the application had only been approved based on a higher down payment and/or a manager's override, but the appraisal throws an additional wrinkle into the mix, requiring yet further increases in the down payment amount or a renewed managerial override.

Fortunately, tying this additional requirement into the process is not complicated. The modularity of BPM and rules allow us to add this additional step with relative ease.

Creating a new diverging XOR gateway before *Approve Mortgage*. For better modeling readability, give this gateway a name of “*Appraised?*” and then create a *Yes* sequence flow that connects to a converging gateway and approves the mortgage afterwards. To check and see if the property has already been appraised, simply verify that the appraisal field of the application variable is not null:

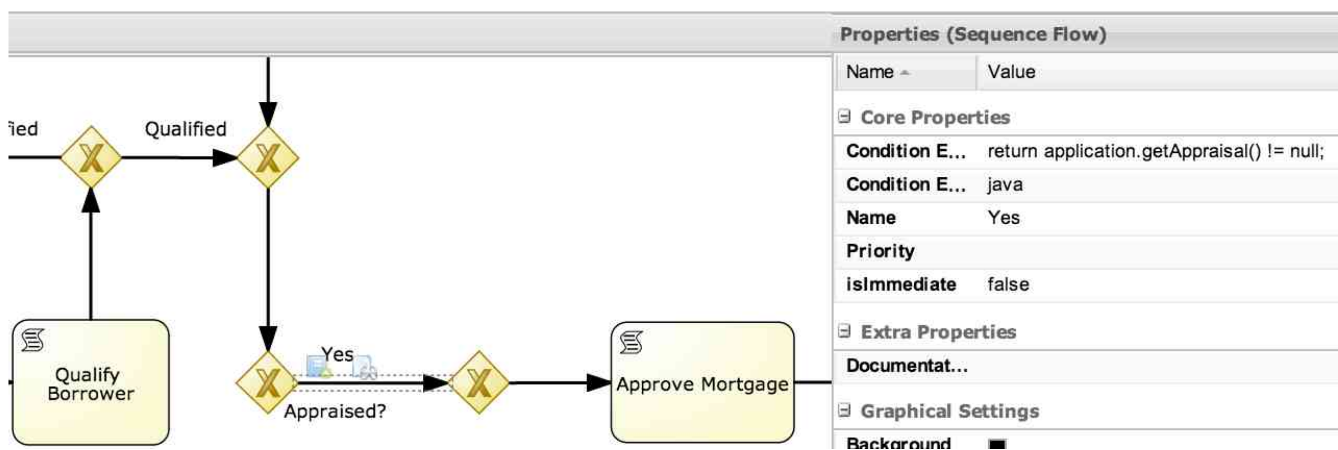

Figure 5.4.11-1:

The *No* sequence flow goes to a new user task called *Appraisal*, with a task form designed as described in the corresponding section, the Appraisal form. The appraisal task is assigned to the appraiser group and simply takes the application as its input and updates it as its output. The only part of the application that may be updated is the value field of the appraisal object within application.

Once appraisal is performed, compare the appraised value with the sale price of the property:

```
return (application.getAppraisal().getValue() >=
                              application.getProperty().getPrice());
```

Place another diverging gateway after the appraisal task and creating two sequence flows leaving it, *Sufficient Appraisal* if appraisal value is at least the sale price of the property, and *Low Appraisal* if it is appraised at a lower price:

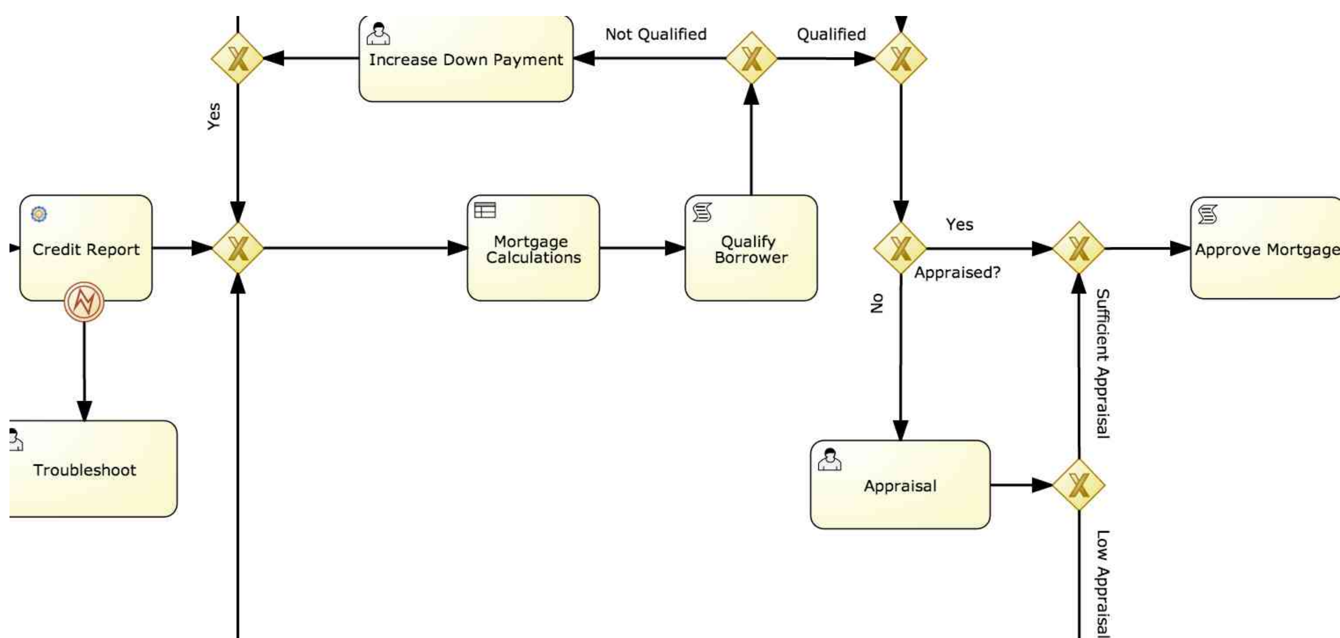

Figure 5.4.11-2:

In the case of a low appraisal, mortgage calculations would need to be repeated and there is a possibility that an otherwise qualified mortgage application would be declined unless there is further down payment increases or a manager override.

Business rules are designed to take appraisals into account, as demonstrated in Figure 5.6.3-7: *Low Down Payment based on Appraisal Result*.

5.4.12 Swimlanes and Business Continuity

The mortgage application business process includes a total of five user tasks, with each being assigned to a different group while *Data Correction* and *Increase Down Payment* are both assigned to the same group. Assigning tasks to groups has the advantage of avoiding tight coupling between individuals and business processes that may need attention outside that individual's working hours. In this model, any member of a group of users is able to view all assigned tasks and claim a task to work on. Further configuration makes it possible to notify group members or make more advanced assignment decisions.

In a process like this where the same task may be executed multiple times, or even in a case where a mortgage broker might need to contact the same customer once to correct data and another time to request a higher down payment, there is great business value in having the same group member handle both tasks.

Swimlanes allow a task to be assigned to a group, but to undergo assignment a single time for each business process instance. In other words, once a task in a swimlane has been claimed by an actor, all other instances of the same task or other tasks in the same swimlane will automatically be assigned to that actor again for the lifetime of the process instance. This avoids the situation where a user will have to deal with a different mortgage broker at each turning point, for the same mortgage application.

The user tasks in this process are as follows. A single swimlane will be used for the two tasks assigned to the broker group with a second swimlane used for the Financial Review task for the manager.

User Task Name	Group Assignment	Swimlane
Data Correction	broker	yes
Increase Down Payment	broker	yes
Financial Review	manager	yes
Troubleshoot	admin	no
Appraisal	appraiser	no

Table 5.4.12-1: User Tasks, Swimlanes

To create a swimlane, open the *Swimlanes* group from the palette and drag and drop the *Lane* onto the canvas. Resize the lane as appropriate to cover the two broker tasks and their adjacent nodes. Double-click the lane to give it name; call the lane *broker* to make clear that associated user activities will be performed by a member of the broker group.

By default, nodes are placed one on top of each other in the order in which they are placed in the canvas. Based on this behavior, the swimlane node covers all the process activity as it is placed and resized in the process. Select the lane and use the toolbar menu to send it to the back. Resize the lane in a way that its borders are around the nodes, so that they remain visible:

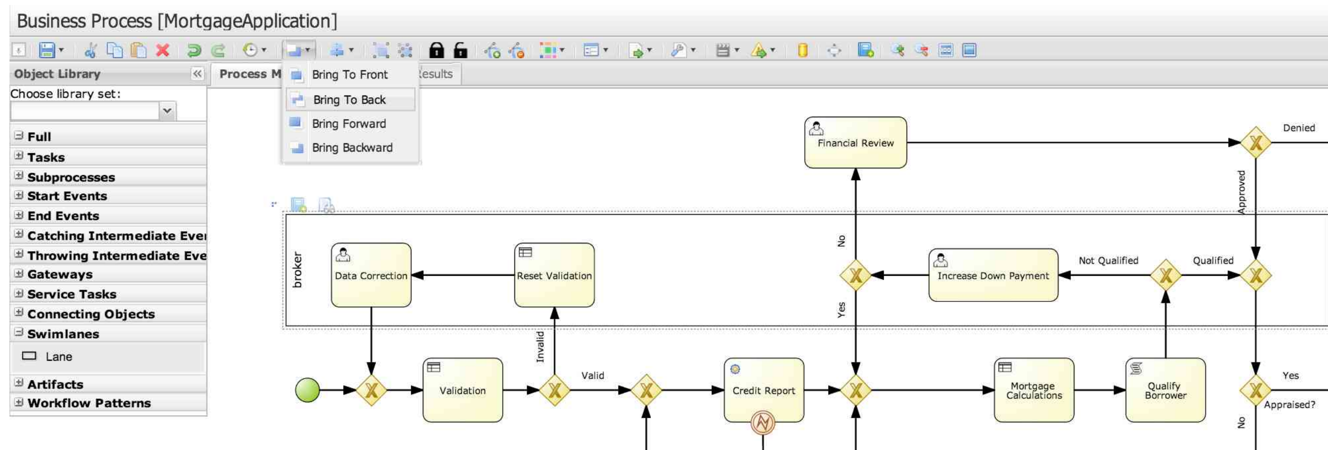

Figure 5.4.12-1:

5.4.13 Final Process Model

The final business process model is as follows:

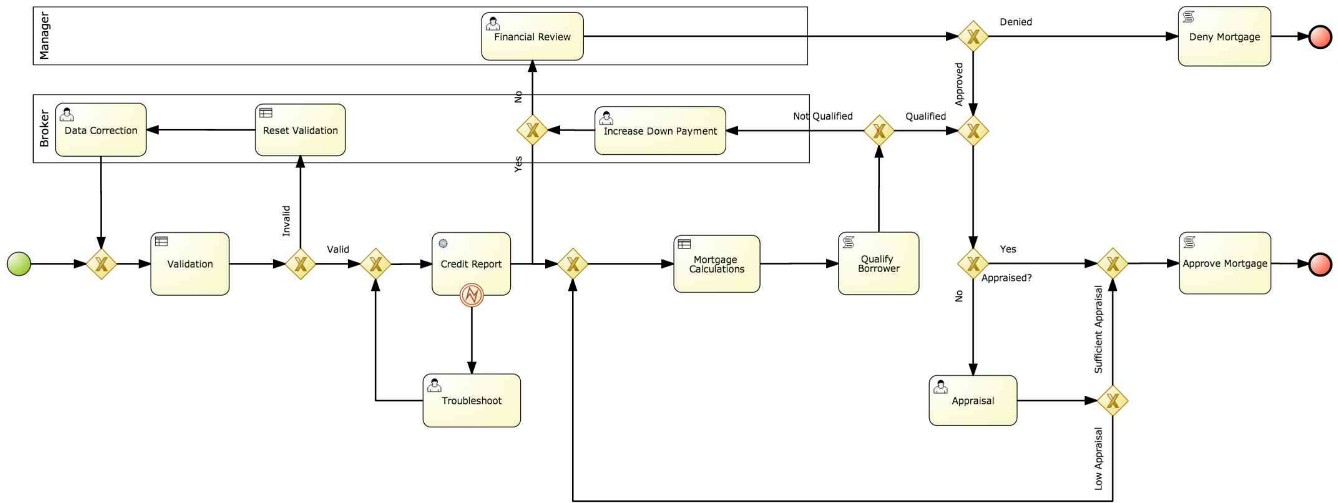

Figure 5.4.13-1:

5.5 Forms

5.5.1 Process Form

While remote callers can use the REST API to start a new instance of the MortgageApplication business process, users are also able to provide the same information through a Business Central form.

The process requires an *Application* object, a custom type, which itself embeds other custom types created in the data modeler. To be more precise, an *Application* object provided to the process contains an *Applicant* and a *Property* object. The process form therefore also embeds two subforms called *MortgageApplicant* and *MortgageProperty*. The naming pattern serves to remind that these subforms are not generic forms used for the corresponding data types, but rather created to represent the applicant and property fields of the application object. For example, *ApplicationApplicant* does not include a field for *creditScore*, since creditScore is not part of the input; it is calculated based on credit data that can be obtained with the applicant's social security number.

5.5.1.1 Applicant subform

Create a new form called *MortgageApplicant*. Add a data origin item called applicant, to be derived from a variable called applicant and also be mapped to the applicant variable upon submission. The type of this data origin is *com.redhat.bpms.examples.mortgage.Applicant*.

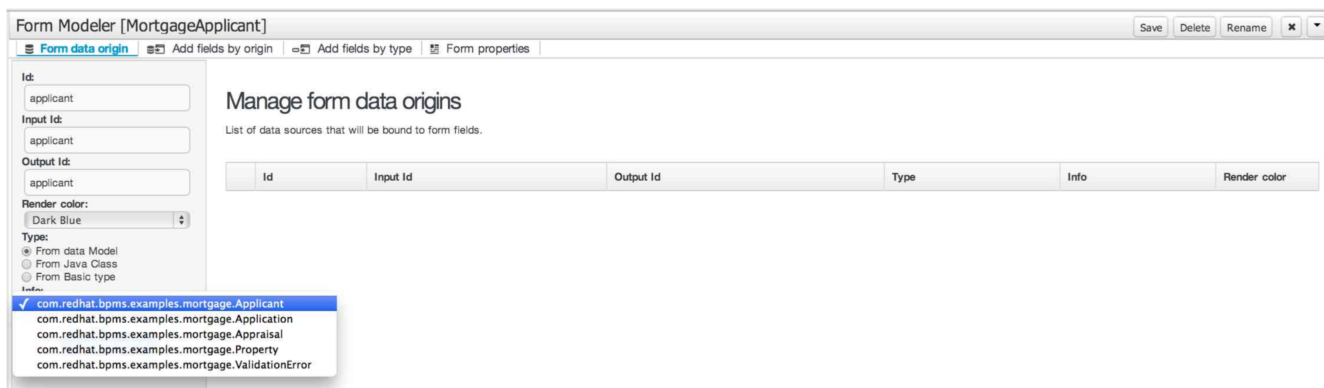

Figure 5.5.1-1:

Once the data origin is defined, go to *add fields by origin* and add the individual fields:

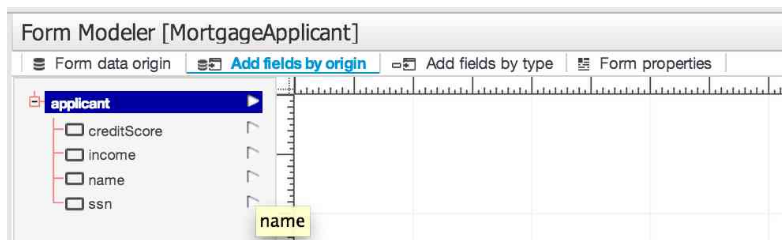

Figure 5.5.1-2:

Edit each added field by hovering over them and clicking the pencil icon. The generated values for each field are mostly sufficient but change the label to something more user friendly:

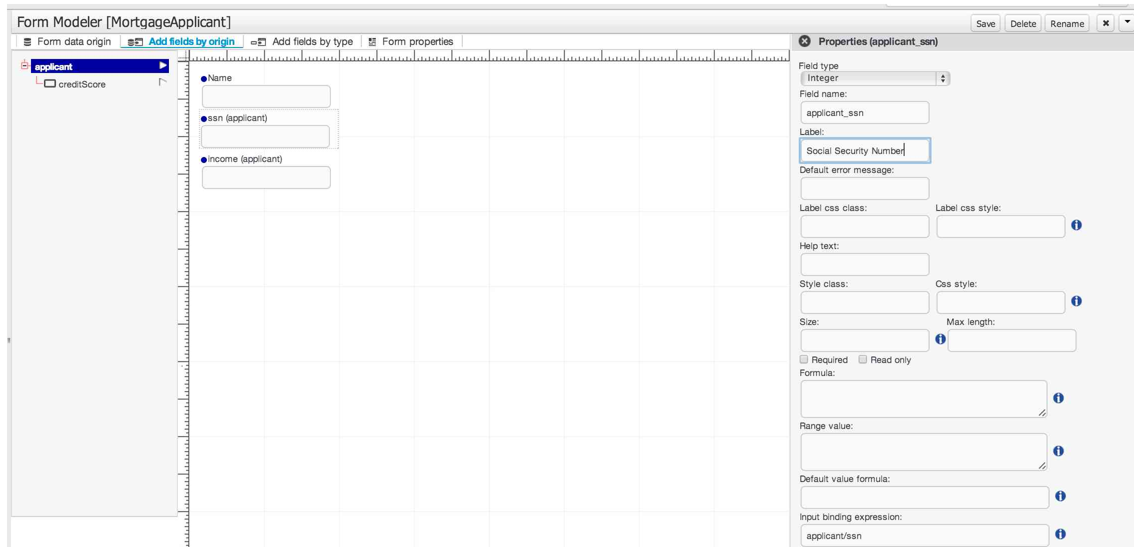

Figure 5.5.1-3:

Make sure to scroll all the way done and click the save button after editing a field.

Once all three fields have been added with a proper label, save the form and close it.

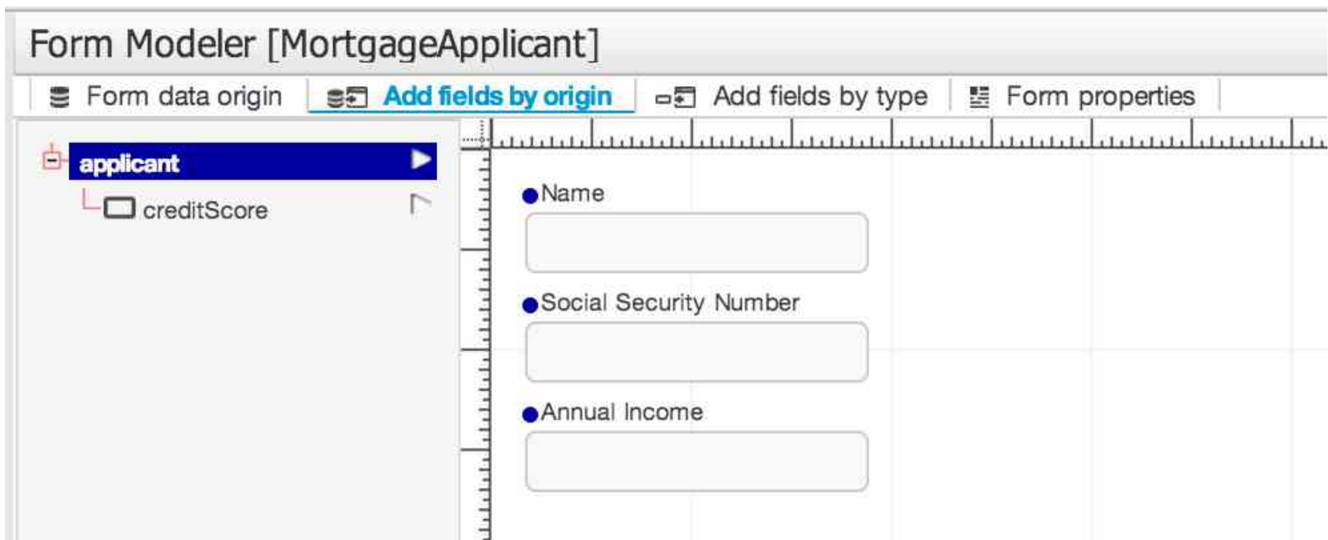

Figure 5.5.1-4:

5.5.1.2 Property subform

Follow similar steps to create a subform called MortgageProperty with both the fields from *com.redhat.bpms.examples.mortgage.Property.*

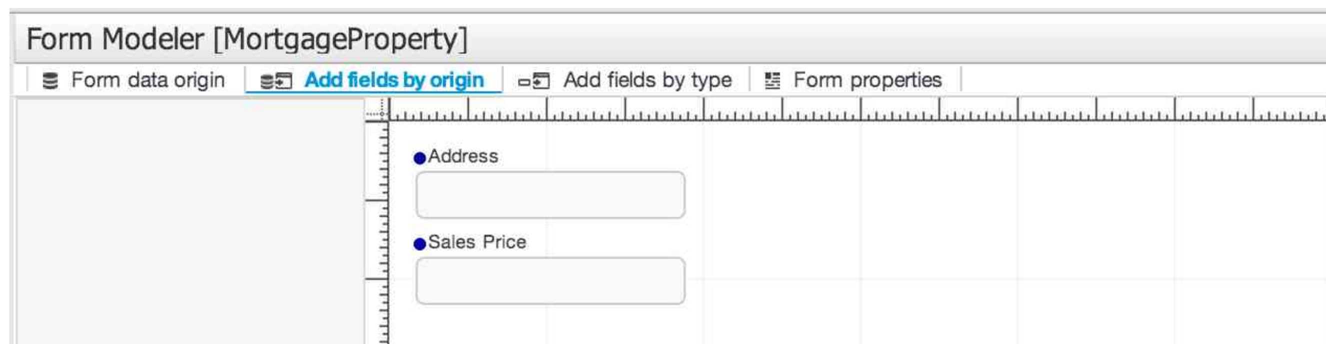

Figure 5.5.1-5:

5.5.1.3 Process Form

When a process is started in Business Central, the expected process form name is derived from the process ID. The easiest and least error-prone approach to creating a process form with the correct name is to open the process and select to edit process form from the top menu:

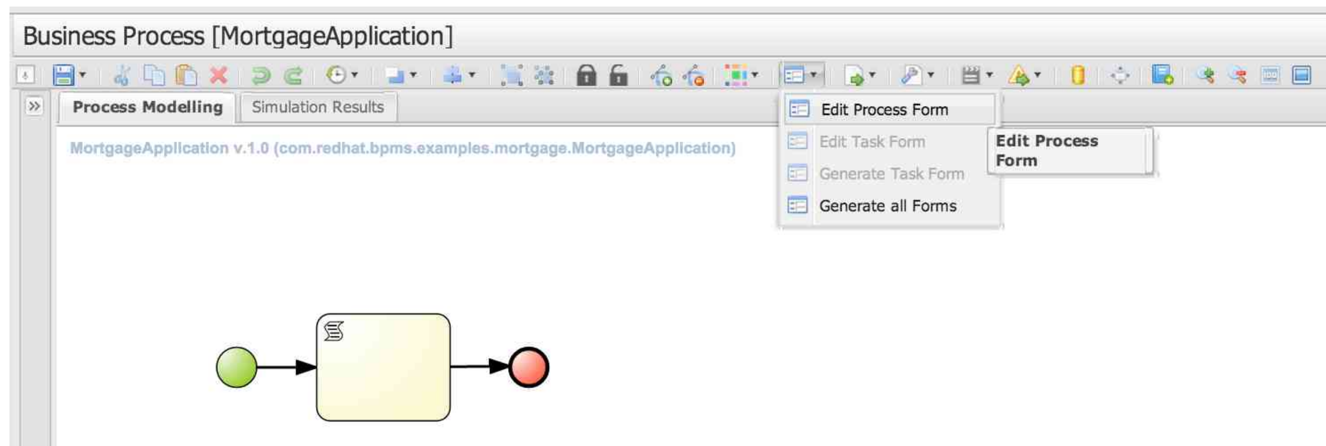

Figure 5.5.1-6:

Add application as the only data origin for the process form:

Manage form data origins

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

	Id	Input Id	Output Id	Type	Info	Render color
	application	application	application	dataModelerEntry	com.redhat.bpms.examples.mortgage.Application	

Figure 5.5.1-7:

Once again, go to *add fields by origin* to add the individual fields. Add applicant, property, down payment and amortization respectively:

Form Modeler [com.redhat.bpms.examples.mortgage.MortgageApplication-taskform]

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

application
appraisal
apr
mortgageAmount
validationErrors

applicant (application)
There is no default form.
property (application)
There is no default form.
downPayment (application)
amortization (application)

Figure 5.5.1-8:

Those fields of application that are not of basic and primitive type cannot be directly mapped to a field on the form. Edit each such fields and associate them with a previously created form that corresponds to the data type.

Edit the applicant field. Set its label to the more user friendly value of *Mortgage Applicant*. The field type should be *simple subform*. Selected the previously created Applicant subform as the default form for this field. Scroll down to the bottom and click save to store the properties and view the effect on the master form.

Save Delete Rename
Properties (application_applicant)
Field type
Simple subform
Field name:
application_applicant
Label:
Mortgage Applicant
Default error message:
Label css class:
Label css style:
Help text:
Style class:
Css style:
Read only
Default form:
MortgageApplicant.form
Input binding expression:
application/applicant
Output binding expression:
application/applicant
Save Cancel

Figure 5.5.1-9:

Follow similar steps to use the Property subform for the property field and set proper labels for down payment and amortization.

Form Modeler [com.redhat.bpms.examples.mortgage.MortgageApplication-taskform]

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

application
appraisal
apr
mortgageAmount
validationErrors

Mortgage Applicant
Name
Social Security Number
Annual Income
Property
Address
Sales Price
Down Payment
Amortization

Figure 5.5.1-10:

5.5.2 Data Correction

The data correction task form is very similar to the process form. The biggest difference is that the application variable is named differently for the task and there are in fact separate variable names for application on its input to and output from the task:

Manage form data origins

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

	Id	Input Id	Output Id	Type	Info	Render color
	application	taskInputApplication	taskOutputApplication	dataModelerEntry	com.redhat.bpms.examples.mortgage.Application	

Figure 5.5.2-1:

Similar to the process form, go to *add fields by origin* to add the individual fields. Add applicant, property, down payment and amortization respectively.

Edit each such field and set a proper label for them.

In the case of applicant and property, in additional to setting a user friendly label, also set the corresponding previously created form as the default form of the field. Scroll down to the bottom and click save to store the properties for each field, and view the effect on the master form.

The final data correction form looks as follows:

Figure 5.5.2-2:

5.5.3 Increase Down Payment

The task form to increase the down payment is very simple and consists of only four fields.

The sale price of the property is included, mostly as a reminder. The request to increase down payment is always due to a rejection of the original mortgage application. In the regular turn of event, the mortgage interest rate is calculated based on various factors and is in turn used to derive the monthly payment. The mortgage may not qualify and a higher down payment requested, simply because this calculated interest rate is too high. In another scenario, the appraisal of the property may result in a value that is lower than the sale price.

Base on these two common causes, the mortgage apr and the appraisal value (if appraisal is even performed yet) are presented as part of the task form.

These three fields (property price, appraisal value and mortgage APR) should be marked as read-only so that they cannot be modified by the task.

The fourth and final field of this task form is the down payment itself, which may be updated by the task.

Instead of using subforms, this task form directly navigates and references the values corresponding to the form fields. The input binding values for property price, appraisal value and mortgage APR respectively are *taskInputApplication/property/price*, *taskInputApplication/appraisal/value* and *taskInputApplication/apr*. All three fields are marked read-only and their output binding field is left blank.

Down payment has *taskInputApplication/downPayment* as its input binding and *taskOutputApplication/downPayment* as its output binding.

Property Sale Price
Appraised Value
Mortgage APR
Down Payment

Figure 5.5.3-1:

5.5.4 Financial Review

The financial review task form allows a manager to review a declined mortgage application and potentially override the decision. The information displayed along with the task to enable such a decision includes all the financial data pertaining to the mortgage application, including both the data provided by the applicant and the rates and values computed by the process. This includes Property Sale Price, Appraised Value, Down Payment, Amortization, Mortgage APR, Credit Score and Annual Income. All of these fields are marked as read-only and much like the Increase Down Payment task form, they are linked directly to the field nested within the application object instead of using a subform.

The decision to override the process and approve the mortgage application is made through this form through a simple checkbox that is mapped to a task variable in its output binding expression. This boolean variable is mapped by the task to an equivalent process variable which is used in a gateway to determine if the mortgage should be approved.

Figure 5.5.4-1:

5.5.5 Appraisal

The appraisal of the property happens only once in the process and is never updated. Accordingly, the appraisal value has no input binding and starts as blank before being entered into the task and updated within the appraisal field of the application object.

The appraisal field of the task may both be modeled as a simple field, or as a subform for the custom appraisal type. Modeling a simple field is easier and faster to do, while the subform is reusable and the associated extra effort pays off in terms of consistency and future ease of use.

Figure 5.5.5-1:

5.6 Business Rules

5.6.1 Validation

To validate the correctness of supplied data as part of the mortgage application, write a number of business rules using the web designer's guided rule editor.

For example, assume that this business does not offer mortgages for any properties with a sale price that is lower than $50,000. To enforce this rule, create a new item of type guided rule and call it *Validate Property Price*. Click the plus sign across from *when* to create the condition for this rule. From the dialog that opens, select Property to declare the constraint on the price field of the property:

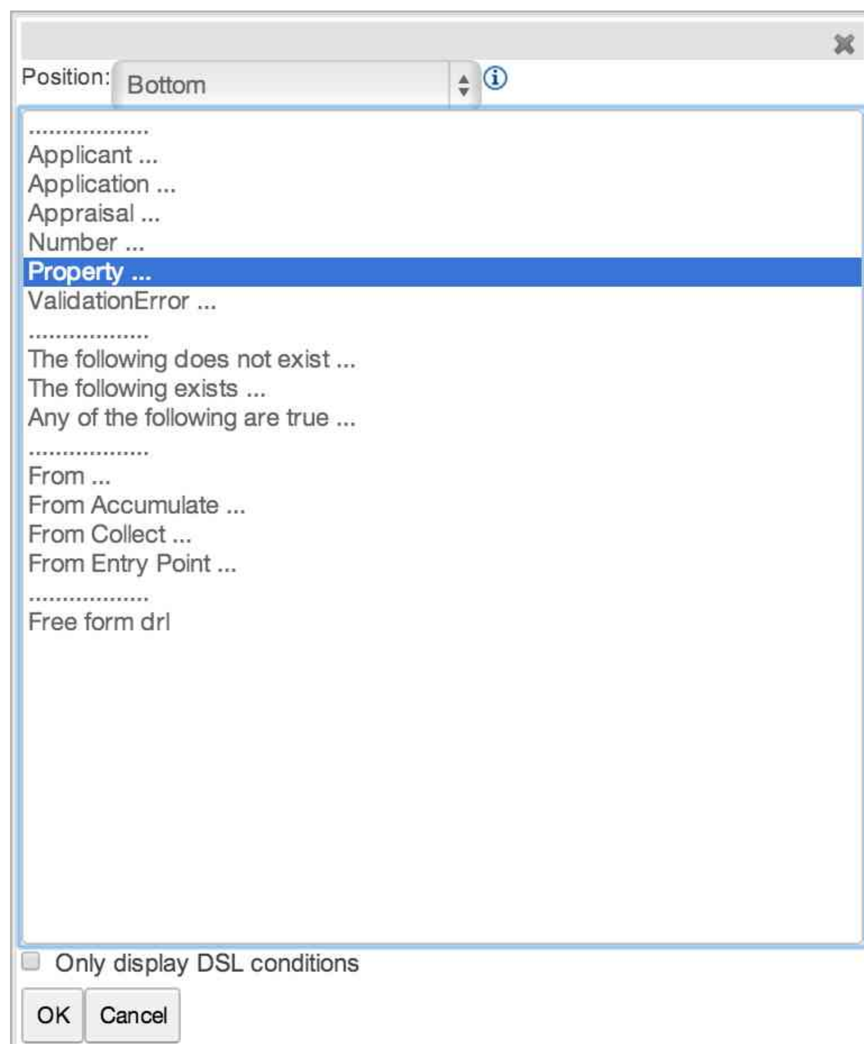

Figure 5.6.1-1:

The guided rule will then include a numbered item for its condition, simply stating:

```
There is a property
```

To further refine the condition, click on this sentence to open a dialog and add a restriction on the price field:

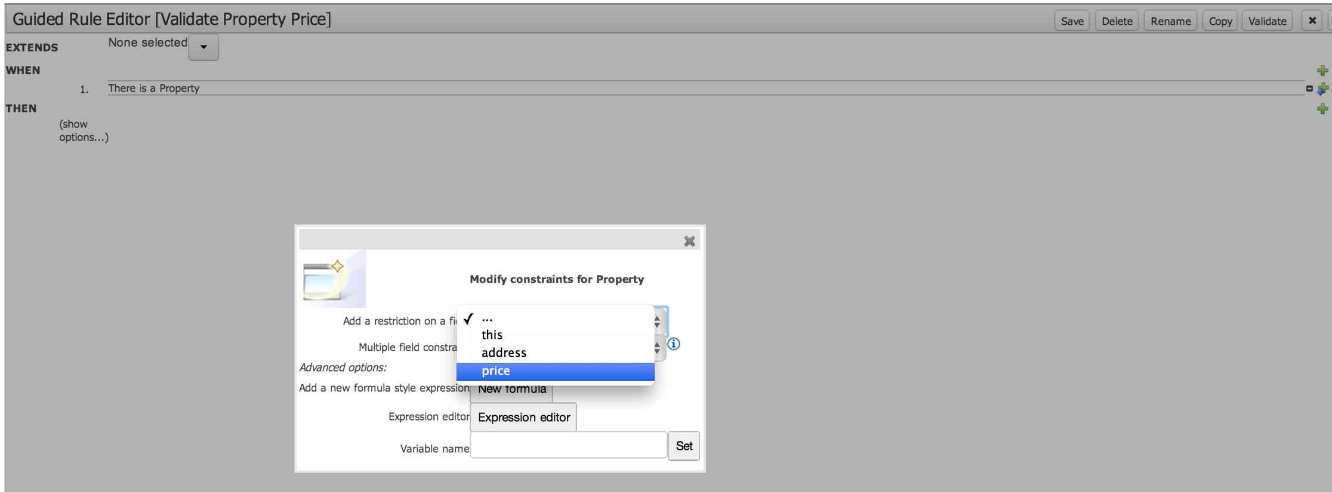

Figure 5.6.1-2:

Use the drop-down to constrain price when less than a value; click the pencil icon and declare the value to be a literal value and enter it as ***50000***.

Now click the plus icon next to *then*, to create a consequence for this rule. From the dialog, select to insert a *ValidationError* when the rule's conditions are met, which in this case means, when the property price is less than 50,000:

Guided Rule Editor [Validate Property Price]

EXTENDS None selected

WHEN

There is a Property with:

1. price less than 50000

THEN (show options...)

Position: Bottom

....
Insert fact Applicant...
Insert fact Application...
Insert fact Appraisal...
Insert fact Number...
Insert fact Property...
Insert fact ValidationError...
....
Logically Insert fact Applicant...
Logically Insert fact Application...
Logically Insert fact Appraisal...
Logically Insert fact Number...
Logically Insert fact Property...
Logically Insert fact ValidationError...
....
Add free form drl

Figure 5.6.1-3:

Once the action has been updated to reflect the selection, click on the phrase *Insert ValidationError* to further configure the action. Select to add a *cause* field and set this field to have a literal value that explains the validation error, for example: *Property price too low*

Next, click on *(show options)* and then click the corresponding plus icon to create a new option. Choose the *ruleflow-group* attribute from the drop-down and set its value to *validation:*

Guided Rule Editor [Validate Property Price]
EXTENDS None selected
WHEN
There is a Property with:
1. price less than 50000
THEN
Insert **ValidationError**:
1. cause Property price too low
(options)
Attributes:
dialect mvel
ruleflow-group validation

Figure 5.6.1-4:

Save this rule and enter a meaningful description for the purpose of the repository revision. The ruleflow-group of this rule ties it to a business rule task node in the process and causes the process engine to evaluate this rule when that node is reached. If the condition of the rule it true, the rule is said to have fired, which means its consequence, also known as its action, will be executed. In this case the action is to create a new instance of the *ValidationError* class, set the value of its *cause* field to a descriptive message and insert the object in the working memory. The XOR gateway in the process looks at the working memory to decide which path to take.

An example of a slightly more complicated rule is one that validates the amount of the down payment but ensuring that it is not a negative number and also that it is not larger than the sale price of the property itself.

For this purpose, create a rule called *Validate Down Payment*. Add a condition and select *Property* as the constraint. Click on *There is a Property* and enter a variable name for this constraint; for example: *property*.

Click the plus logo across from this new constraint which also has a down arrow superimposed on it. This indicates that a new constraint will be added directly under the constraint in question. This time declare the constraint to apply to *Application*. Further configure the generated *There is an Application* constraint by clicking on it and choosing to add a restriction on its *downPayment* field. Constrain any down payment that is *less than* the literal value of ***0*** and then click the right arrow next to it to add more options to this constraint. Select the option *or greater than* and use the *Expression editor* to select *property.price*.

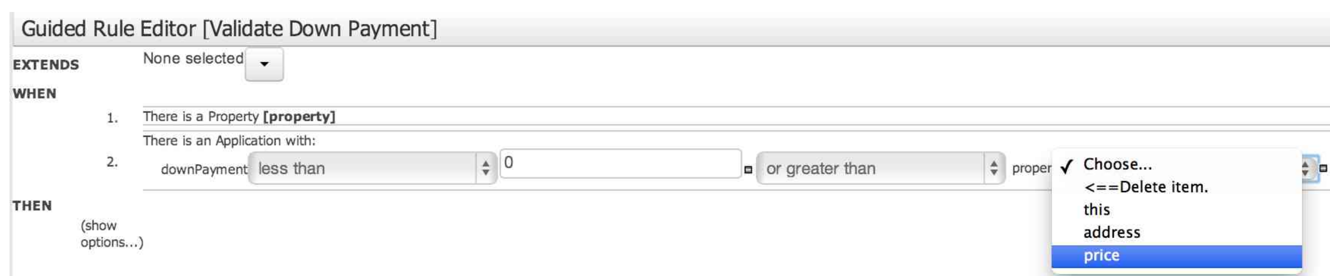

Figure 5.6.1-5:

When these conditions apply, then choose to *Insert fact ValidationError* with its *cause* field set to: *Down payment can't be negative or larger than property value*

Once again, show options and add an option, setting the *ruleflow-group* attribute to *validation* before saving and committing the rule to the repository.

Follow a similar pattern to create three more validation rules.

Create another guided rule and call it *Validate Income*.

Add the constraint on the *Applicant*. Add a restriction on the *income* field of the applicant and look for an income that is less than ***`10000`***.

Once again, create a corresponding validation error with an appropriate cause description:

Income too low

Remember to add a *ruleflow-group* attribute to the rule and set it to *validation*.

Guided Rule Editor [Validate Income] Save Delete Rename Copy Validate

EXTENDS None selected

WHEN

There is an Applicant with:

1. income less than 10000

THEN

Insert **ValidationError [fact0]**:

1. cause Income too low

(options)

Attributes:

dialect mvel

ruleflow-group validation

Figure 5.6.1-6:

Next, create a guided rule and call it *Validate SSN*. This will be a simple validation to make sure that the provided social security number is nine digits long and does not start with a zero. A more comprehensive validation is possible by following the guidelines of the Social Security Administration.

Add the constraint on the *Applicant*. Add a restriction on the *ssn* field of the applicant and look for any number that is either less than ***100000000*** or greater than ***999999999***.

Create a corresponding validation error with an appropriate cause description:

Invalid Social Security Number

Add a *ruleflow-group* attribute to the rule and set it to *validation*.

Guided Rule Editor [Validate SSN]
Save Delete Rename Copy Validate
EXTENDS None selected
WHEN
There is an Applicant with:
any of the following:
1. ssn less than 100000000
ssn greater than 999999999
THEN
Insert **ValidationError [fact0]:**
1. cause Invalid Social Security Number
(options)
Attributes:
dialect mvel
ruleflow-group validation

Figure 5.6.1-7:

Finally, create the last validation rule and call it *Validate Amortization*. Assume that only fixed-rate mortgages of 10, 15 and 30 years are provided by this business. Any amortization value other than these three would therefore be rejected.

Add the constraint on the *Application*. Add a restriction on the *amortization* field of the application and make the rule applicable if the amortization:

is not contained in the (comma separated) list

Provide of list of the acceptable amortization values: ***`10, 15, 30`***.

Create a corresponding validation error with an appropriate cause description:

Amortization can only be 10, 15 or 30 years

Add a *ruleflow-group* attribute to the rule and set it to *validation*.

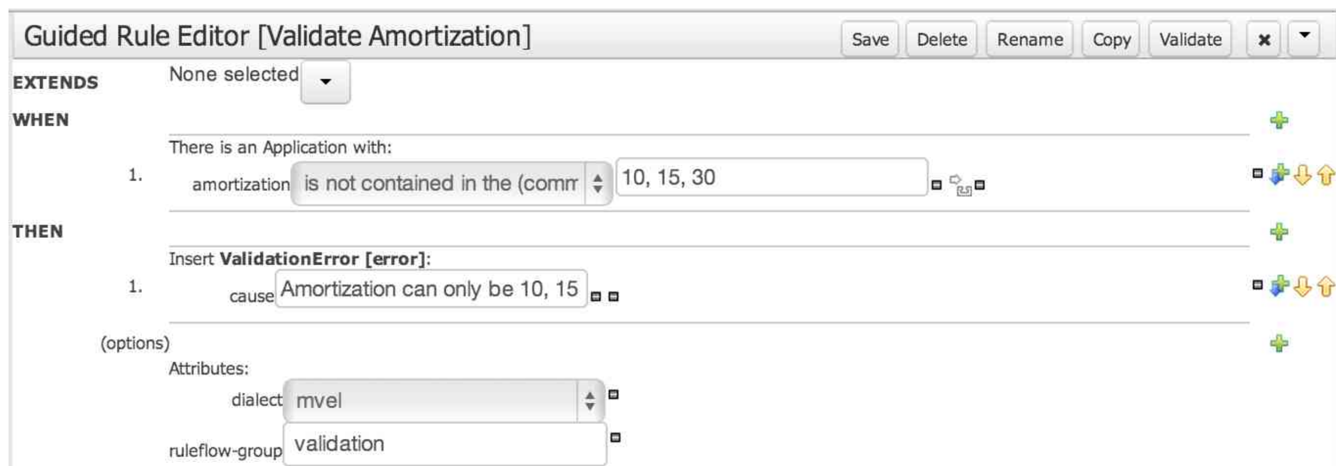

Figure 5.6.1-8:

Given the possibility that processes may run in a single knowledge session in a single-threaded model, it is important for the rules to clean up after themselves. In this case, the last set of rules would proceed to remove the *Application*, *Applicant* and *Property* objects that have been inserted for the express reason of validation. A negative *salience* attribute is employed on the rules to ensure that they don't execute before the actual validation rules.

Create a new *DRL file* as a new item in the package and call it: *Retract Facts After Validation*

Write rules that simply seek and remove any facts of these known types:

```
package com.redhat.bpms.examples.mortgage;

rule "Retract Applicant after Validation"
     dialect "mvel"
     ruleflow-group "validation"
     salience -10
     when
          fact : Applicant( )
     then
          retract(fact);
          System.out.println("Executed Rule: " +
drools.getRule().getName() );
end

rule "Retract Application after Validation"
     dialect "mvel"
     ruleflow-group "validation"
     salience -10
     when
          fact : Application( )
     then
          retract(fact);
          System.out.println("Executed Rule: " +
drools.getRule().getName() );
end

rule "Retract Appraisal after Validation"
     dialect "mvel"
     ruleflow-group "validation"
     salience -10
     when
          fact : Appraisal( )
     then
          retract(fact);
          System.out.println("Executed Rule: " +
drools.getRule().getName() );
end

rule "Retract Property after Validation"
     dialect "mvel"
     ruleflow-group "validation"
     salience -10
     when
          fact : Property( )
     then
          retract(fact);
          System.out.println("Executed Rule: " +
drools.getRule().getName() );
end
```

5.6.2 Reset Validation

Once a validation error has been raised, the process enters a loop of data correction and validation, until such time that the data is deemed completely valid. Errors are signaled by inserting a *ValidationError* object in the rule engine's working memory. This object is used by the *XOR gateway* to determine if data correction is necessary, but immediately after such a determination, the error object must be removed so that the next validation can take place with a clean slate.

To reset validation, write a simple guided rule that looks for the *ValidationError* object and removes it. Associate this rule with the business rule task in the process by specifying the correct ruleflow group.

Guided Rule Editor [Reset Validation]
Save Delete Rename Copy Validate

EXTENDS None selected

WHEN

1. There is a ValidationError **[error]**

THEN

1. delete **ValidationError [error]**

(options)

Attributes:

dialect mvel

ruleflow-group resetValidation

Figure 5.6.2-1:

5.6.3 Mortgage Calculations

This simplified business model prices a mortgage by first calculating a minimum interest rate, based on only the length of the fixed-term mortgage (i.e., APR) and the applicant's credit score. This is followed by a look at the down payment ratio and the APR is adjusted upward if less than 20% is provided. Finally, jumbo mortgages are identified and result in yet another potential increase in the mortgage APR.

Calculating the interest rate based on credit score and amortization is a natural tabular format and a great fit for a decision table. Create a guided decision table as a new item and call it *Mortgage Calculation*. Select to use the wizard and proceed to the next step. There is no need to import any Java types so once again, click next.

Choose *Applicant* and *Application* as the two fact patterns to use, as they hold the applicant's credit score and selected amortization respectively.

Figure 5.6.3-1:

In the next step, select each pattern, a field for that pattern and then create a constraint for that give field of the selected pattern.

Credit scores are considered in brackets so to designate a bracket, two separate constraints are required for the credit score where one defines the acceptable lower bound and the other, the upper bound.

Select *Applicant* and then *creditScore*, as its field. Click on the created condition template and complete it by declaring a column header of *Credit Score* >= to indicate that the provided values in the table are the lower bound. Set the operator to *greater than or equal to*. Set the value list based on credit score brackets: *,660,680,700,720,740*

The creating table allows an analyst to easily create or update rules. By providing a value list, the application limits the credit score brackets to know values and generate a drop-down instead of a free-form text field. The preceding comma allows a blank value in the drop-down, which, when used, is equivalent to not specifying a lower bound for the credit score:

Guided Decision Table Wizard
Summary
Imports
Add Fact Patterns
Add Constraints
Add Actions to update Facts
Add Actions to insert Facts
Columns to expand
Define constraints on the Facts\Patterns fields.
Available patterns
Applicant
Application
Available fields
this : this
creditScore : Whole number (i
income : Whole number (integ
name : Text
ssn : Whole number (integer)
[New Predicate]
>>
<<
Conditions
[Credit Score >=] creditScore
Calculation type: Literal value Formula
Column header (description): Credit Score >= *
Operator: greater than or equal to *
(optional) value list: ,660,680,700,720,740
Default value:
<- Previous Next -> Cancel Finish

Figure 5.6.3-2:

Select the *creditScore* field once again and click the double right arrow to add another condition based on this same field. This time the condition sets the upper bound value for the applicant's credit score. Name the column header *Credit Score* < this time and choose the operator of *less than*. Use the same value list again to allow empty values.

Next, select the *amortization* field of *Application* and add a condition based on this field. Call the column header *Fixed Mortgage Term* and use the *equal to* operator. Set the value list to only allow acceptable amortizations: *10, 15, 30*.

Click next to set the action corresponding with the condition. Selection *Application* as the pattern and then *apr* as its field. Click on the created action template under *Chosen fields* and enter the column header as *Mortgage APR*.

Figure 5.6.3-3:

Skip the remaining steps, as there is no need to insert any facts and the columns may be left expanded.

Once the table is created, an important step is to set its *ruleflow-group* attribute to *apr-calculation* so that it is associated with the corresponding business rule task in the process. To do this, expand the decision table configuration by clicking the plus sign and select to add a new column. Choose the following option:

Add a new Metadata\Attribute column

This action creates the attribute group under options. Expand options and enter the ruleflow group as the default attribute for all table rows:

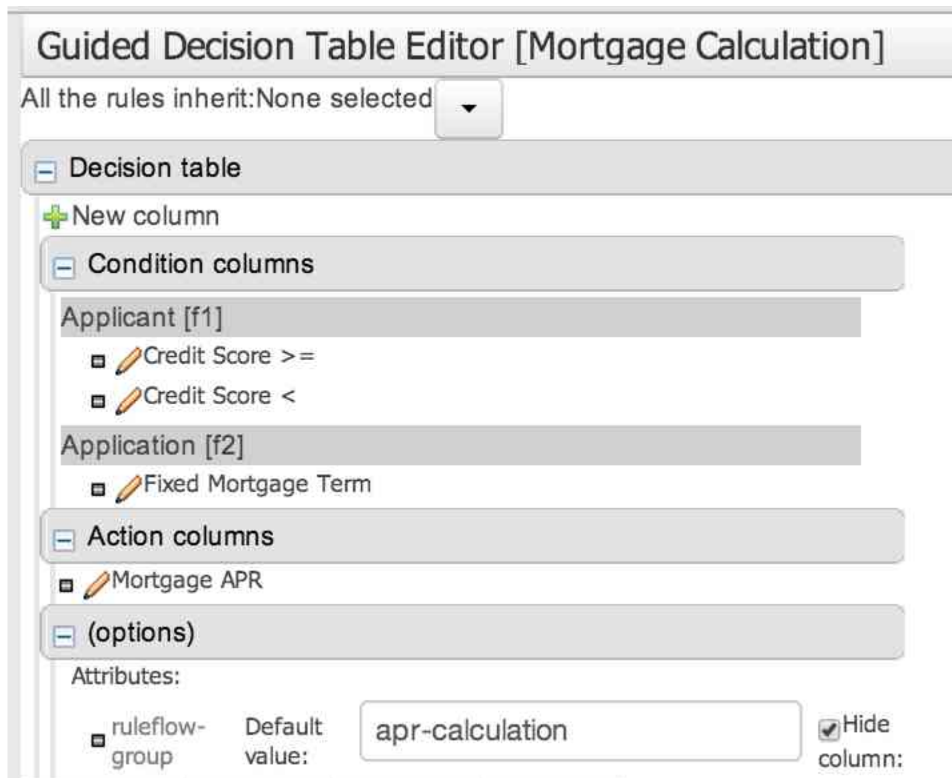

Figure 5.6.3-4:

Selecting the checkbox to hide the column helps shield rule analysts from technical details that are not directly relevant to the rules or the business requirements.

Fill out the decision table so that a mortgage APR is provided for each bracket of credit scores for any given amortization (only 10, 15 and 30).

Guided Decision Table Editor [Mortgage Calculation]

All the rules inherit:None selected

Decision table

Add row... | Otherwise | Analyze... | Audit log

#	Description	Credit Score >=	Credit Score <	Fixed Mortgage	Mortgage APR
1			660	30	10
2		660	680	30	5
3		680	700	30	5
4		700	720	30	4.75
5		720	740	30	4.625
6		740		30	4.5
7			660	15	9.25
8		660	680	15	4.25
9		680	700	15	4.25
10		700	720	15	4
11		720	740	15	3.875
12		740		15	3.75
13			660	10	10.5
14		660	680	10	5.5
15		680	700	10	5.5
16		700	720	10	5.3
17		720	740	10	5.15
18		740		10	5.05

Figure 5.6.3-5:

Save the guided decision table and provide a meaningful description for its repository revision.

After calculating the base interest rate for a given credit score and amortization, the immediate next step in the calculation is to consider the down payment. The base interest rate assumes an industry-standard down payment of twenty percent. Anything below that results in a higher interest rate.

Mortgage calculation takes place before appraisal to avoid the unnecessary cost of property appraisal for an applicant that is not otherwise qualified. However even if the application appears to qualify, property appraisal may result in an assessment of a property value that is significantly lower than the transaction price. Such an assessment may impact the down payment ratio and require a renewed calculation, this time based on the appraised value of the property.

Create two separate rules to cover the evaluation of the down payment in the two separate cases of before and after appraisal.

Call the first guided rule as follows: *Low Down Payment before Appraisal*

Guided Rule Editor [Low Down Payment before Appraisal]

EXTENDS None selected

WHEN

1. The following does not exist:
There is an Appraisal

2. There is a Property **[property]**

3. There is an Application **[application]** with:
mortgageAmount greater than property.price * 8 / 10

THEN

(options)

Attributes:

dialect mvel

ruleflow-group apr-calculation

salience -3

no-loop

Figure 5.6.3-6:

To create this rule as shown above, set the first constraint to: *The following does not exist*

Proceed to click on the generated phrase and select *Appraisal* the fact type. This composite constraint states that this rule is only applicable if an appraisal has not yet been performed.

Click the plus icon again to create a new pattern and select *Property*, then configuring it to have a variable name of *property* so that it can be referenced in the consequence of the rule.

Create a third pattern for *Application* and set a restriction on its *mortgageAmount* field to be greater than the following formula: *property.price * 8 / 10*

If the mortgage amount is greater than 80% of the property price, it follows that the down payment has been less than 20% of the total price.

Show options and add the *ruleflow-group* attribute, setting it to *apr-calculation*.

Also add the *salience* attribute and give it a value of *-3*. This ensures that the base APR calculation rules in the decision table, with a default salience of 0, have already executed before this rule.

Finally, add the no-loop attribute and check the corresponding box. This attribute avoids an infinite rule where the update of the application by this rule, through an APR surcharge, may trick the rule engine into thinking that something has changed and this rule must be reevaluated. In the case of this particular rule, it is nothing but a safety precaution.

The action of this rule is more complicated than any previous one. The guided rule facilities make it easy to author and update rules but are often not appropriate for more difficult technical syntax. Luckily, adding free-form drl is directly supported in the guided rule editor.

Enter the following DRL as the first part of this rule's consequence:

```
double ratio = application.getMortgageAmount().doubleValue() /
                              property.getPrice().doubleValue();
int brackets = (int)((ratio - 0.8) / 0.05);
brackets++;
double aprSurcharge = 0.75 * brackets;
```

At this point, with the rule executing, it is known that the ratio of the mortgage amount to the total property sale price is higher than 80% but this first line calculates this ratio.

While any ratio greater than 0.8 (as is certainly the case here) triggers an APR surcharge, the amount of the surcharge is constant for every little bracket of five percent. A ratio between 80% and 85% triggers an APR surcharge of 0.75 while the next bracket, between 85% and 90%, doubles the surcharge. The bracket number is calculated above and reindexed to 1 before being multiplied by 0.75 to determine the exact applicable surcharge.

Finally, add a second action to: *Change field values of application...*

Select the *apr* field and set it to the following formula: *application.getApr() + aprSurcharge*

Evaluating the sufficiency of the down payment after an appraisal is very similar. This is only necessary if the appraisal has resulted in an assessment of a value for the property that is lower than the sale price. In this case, the mortgage amount remain the same (down payment subtracted from the sale price) but it needs to be lower than 80% of the appraised value. In other words, it is being divided by a smaller denominator.

The rule is otherwise similar:

Guided Rule Editor [Low Down Payment based on Appraisal]

EXTENDS None selected

WHEN

1. There is an Appraisal **[appraised]**

2. There is an Application **[application]** with:
   mortgageAmount greater than appraised.value * 8 / 10

THEN

1.
```
double ratio = application.getMort
int brackets = (int)((ratio - 0.8) / 0.(
brackets++;
double aprSurcharge = 0.75 * bra
System.out.println( "aprSurcharge
```

2. Set value of Application **[application]** apr application.getApr() + aprSurcha

(options)

Attributes:

dialect mvel

ruleflow-group apr-calculation

salience -3

no-loop ✓

Figure 5.6.3-7:

The next potential adjustment to the calculated mortgage interest rate concerns jumbo loans. For simplicity, this business entity assumes a uniform conforming loan threshold of $417,000. Any mortgage amount above this threshold is considered a jumbo loan and subject to an APR surcharge of 0.5.

This rule is give a salience of -5 and applied after potential down payment surcharges.

Guided Rule Editor [Jumbo Mortgage] Save Delete Rename Copy Validate

EXTENDS None selected

WHEN

There is an Application [application] with:

1. mortgageAmount greater than 417000

THEN

1. Set value of Application [application] apr application.getApr() + 0.5

(options)

Attributes:

dialect mvel

ruleflow-group apr-calculation

salience -5

no-loop

Figure 5.6.3-8:

Once again, the no-loop attribute has been added and selected as a precaution.

As was the case with validation, it is also important here for the rules to clean up after themselves. Create a new *DRL* and call it: *Retract Facts After Calculation*

The rules to find and retract the fact types are almost identical:

```
package com.redhat.bpms.examples.mortgage;

rule "Retract Applicant after Calculation"
	dialect "mvel"
	ruleflow-group "apr-calculation"
	salience -10
	when
		fact : Applicant( )
	then
		retract(fact);
		System.out.println("Executed Rule: " +
drools.getRule().getName() );
end

rule "Retract Application after Calculation"
	dialect "mvel"
	ruleflow-group "apr-calculation"
	salience -10
	when
		fact : Application( )
	then
		retract(fact);
		System.out.println("Executed Rule: " +
drools.getRule().getName() );
end

rule "Retract Appraisal after Calculation"
	dialect "mvel"
	ruleflow-group "apr-calculation"
	salience -10
	when
		fact : Appraisal( )
	then
		retract(fact);
		System.out.println("Executed Rule: " +
drools.getRule().getName() );
end

rule "Retract Property after Calculation"
	dialect "mvel"
	ruleflow-group "apr-calculation"
	salience -10
	when
		fact : Property( )
	then
		retract(fact);
		System.out.println("Executed Rule: " +
drools.getRule().getName() );
end
```

5.7 Credit Report Web Service

For the purpose of this reference architecture where the focus is on the BPM Suite, assume that an external web service provides the required information on the credit worthiness of mortgage application.

For the sake of simplicity, create a basic Web Service that takes an applicant's social security number as its only input and returns their Credit Score as the result.

Creating a simple Web Service using **JSR-181**[22] and **JSR-224**[23] requires a simple Web Application with an empty *web.xml* file and an annotated Java class:

```
package com.redhat.bpms.examples.mortgage;

import javax.jws.WebMethod;
import javax.jws.WebService;

@WebService
public class CreditService
{

     @WebMethod
     public Integer getCreditScore(Integer ssn)
     {
          int lastDigit = ssn - 10 * ( ssn / 10 );
          int score = 600 + ( lastDigit * 20 );
          System.out.println( "For ssn " + ssn + ", will return credit score
of " + score );
          return score;
     }
}
```

This class simply uses the last digit of the social security number to mock up a credit score. The web deployment descriptor remains empty:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" 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://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">

</web-app>
```

Assuming that these two files are deployed in a standard web application structure and deployed as *jboss-mortgage-demo-ws.war* on a local service, the following address would be used to access this service:

http://localhost:8080/jboss-mortgage-demo-ws/CreditService?WSDL

22 https://jcp.org/en/jsr/detail?id=181
23 https://jcp.org/en/jsr/detail?id=224

6 Life Cycle

6.1 Asset Repository Interaction

Business Central uses Git as the implementation of its asset repository. In BPMS 6, this is implemented as a simply file-based Git repository in a hidden folder. Users are advised not to directly interact with this file-based repository.

When running, Business Central provides access to its repository through both the *git://* and *ssh://* protocols. While either protocol may be used to retrieve (i.e., pull) data from the repository, adding or updating data (i.e., push) is only possible through authenticated SSH to ensure security. The credentials used to log in to Business Central may also be used to authenticate with the asset repository through SSH.

When a design-time cluster is set up, the asset repositories of the cluster nodes are synchronized by ZooKeeper. This means that data may be pull from or even pushed to any single node while relying on the cluster to replicate the changes across the available nodes.

6.2 JBoss Developer Studio

While this reference architecture uses the web tooling and process designer to create the sample mortgage application, software developers will often benefit from a Java IDE with better development support and integration with other Java and non-Java technology.

JBoss Developer Studio is based on Eclipse and available from the Red Hat customer support portal. JBDS provides plugins with tools and interfaces for JBoss BRMS and BPMS, called the Drools plugin and the jBPM plugin respectively.

One of the plugins that is included in JBDS is the eclipse Git plugin. This plugin can be used to synchronize the content of the JBDS workspace with the asset repository of BPMS.

To set up a BPMS project in JBDS, after it has been created in the web designer environment, use *Import* and select *Projects from Git*. Select the option to *Clone URI*. Copy and paste the ssh URL for the repository in question into the URI field of the JDBS cloning dialog. This URL can be found in the *Authoring / Administration* page of Business Central, after changing it from the default of *git* to *ssh*. Depending on the IP address of the BPMS server in question, the user name used to access Business Central and the configured ssh port, this URL will be similar to the following:

```
ssh://10.16.139.101:8003/Mortgage
```

Pasting into the URI field automatically fills out the various fields including *Host*, *Repository Path*, *Protocol* and *Port*. Also fill in the user name and password configured to access Business Central; credentials are required when using the ssh protocol and allow additions and modifications to be pushed back to the server.

The completed dialog for the reference environment looks as follows:

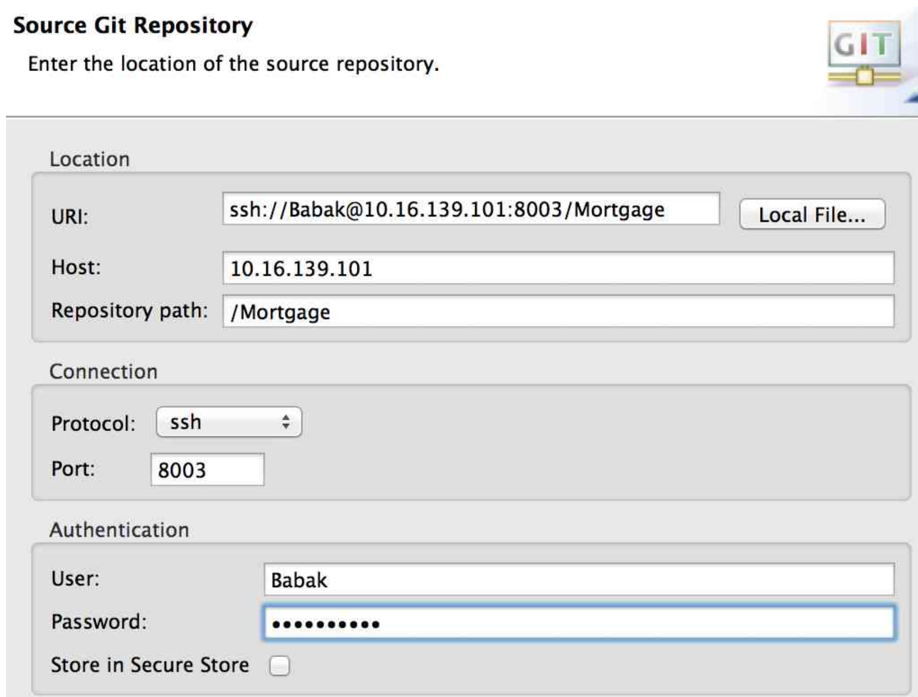

Figure 6.2-1:

Select *Next* to move forward to the next dialog. JBoss BPM Suite 6.0 uses the default *master* branch in its asset repository so this is typically the one and only branch that will be presented and selected at the next step. The third step is to select a local branch, also typically assigned to *master*, and a remote name which is set to *origin* by convention. In this step, select a local directory that will host the local copy of the repository. This is where the JBDS project will be stored.

Clicking *Next* after selecting the local destination opens the import project wizard selection. Select the last option to import the repository as a *general project*. The structure of the BPMS asset repository does not fit the expected structure of a jBPM project. This is remedied later by opening a jBPM file and when prompted, converting the project into a jBPM project. At this point, continue to select a descriptive local project name and finish the import process.[24]

Once a JBDS project has been set up as a clone of a BPMS asset repository, it acts like any other project that uses Git as source control. Changes can be made and committed locally and when ready, those changes can be pushed upstream to the asset repository. Similarly, changes made through the web tools can be retrieved by pulling from the remote repository.

JBoss BPM Suite 6.0 only uses the master branch of Git but developers may still take advantage of other branching and tagging features through third-party tooling. This can take place when working with a clone in JBDS or when otherwise interacting with the asset repository, as long as Business Central only deals with the master branch.

24 https://access.redhat.com/site/documentation/en-US/Red_Hat_JBoss_BPM_Suite/6.0/html/Getting_Started_Guide/Connecting_JBoss_Developer_Studio_to_the_Asset_Repository1.html

6.3 Process Simulation

Business process simulation is a valuable analytic instrument for the purpose of assessing the behavior of business processes over time. This can help estimate the number of external calls, plan for human resources to handle the user tasks and otherwise assist in sizing and estimating technical and business aspects of the process.

In the web process designer, make sure to validate the process, view all issues and correct them before proceeding to process simulation.

From the toolbar of the web process designer, open the process simulation menu item and selected *Process Paths*:

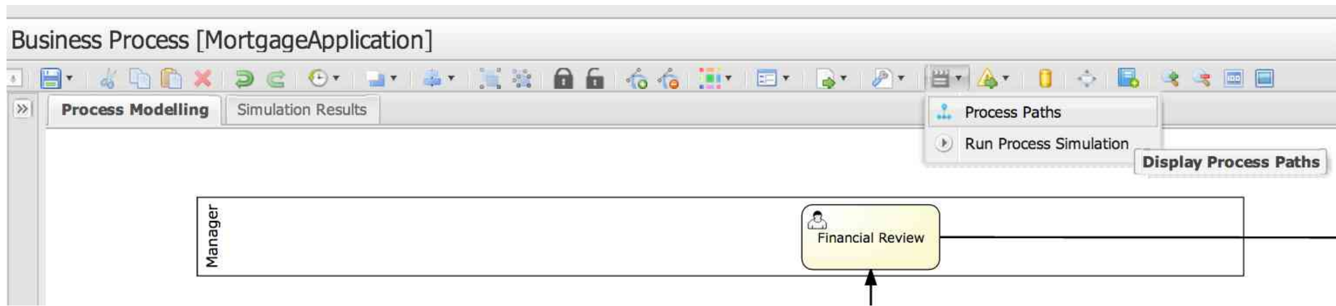

Figure 6.3-1:

This brings up a dialog that shows all the possible process paths, as calculated for this process. The number of paths depends on the number of decision points and the potential complexity of the process. For this process, you may see around 42 different paths calculated. Select a path and click the *Show Path* button to see it in the process designer:

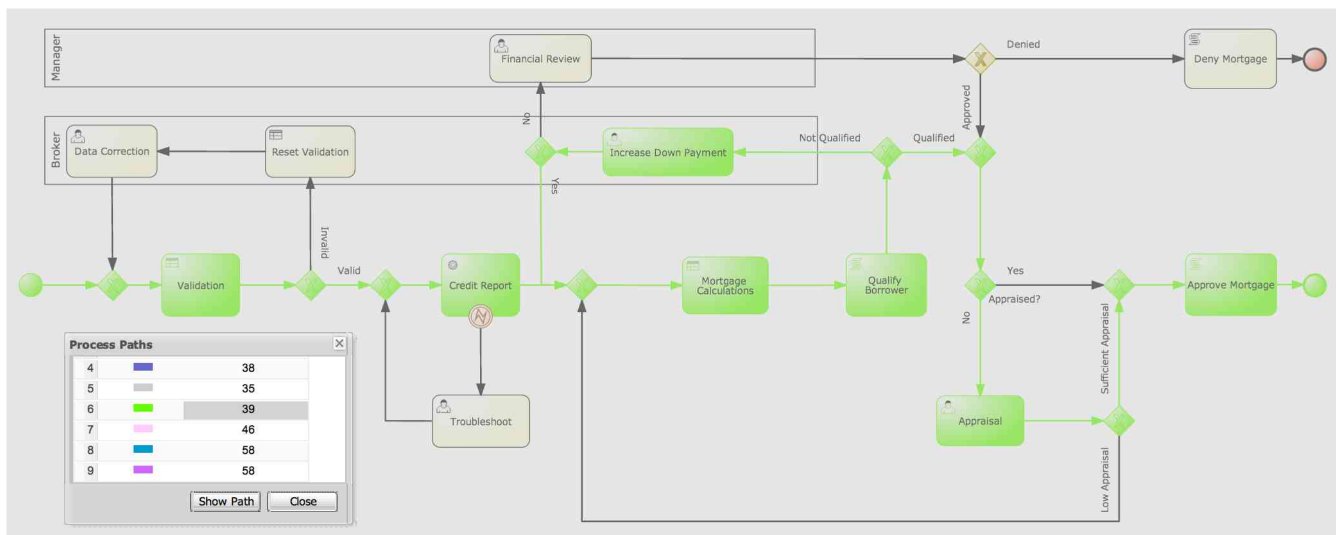

Figure 6.3-2:

Before running process simulation, various simulation properties need to be set up for the business process. This includes a *probability* value for each sequence flow that leaves a diverging gateway. As logically expected, the sum of probability values for all sequence flows leaving any given diverging gateway should always be 100. If this sum is not 100, *validation* will show a validation issue of type *simulation* to warn the user of the problem and prompt its correction.

Other simulation properties include the cost of executing a node per time unit, the minimum and maximum processing time that's envisioned for a node and how it may be distributed. For user tasks, staff availability and the number of working hours also effect process simulation.

For the boundary event of this mortgage process, the probability relates to the chances that an error would occur when calling the web service. In this case there are no multiple outgoing paths that must add up to 100.

From the same drop-down menu, select *Run Process Simulation*, enter the number of process instances to create and the interval at which they will be created in the desired time unit. Running simulation on the mortgage process for 200 instances with one instance started every 5 minutes generated a report such as the following:

Activity Instances

Intances (#)

238.00
212.00
Appraisal
148.0
88.00
134.00
201.00
223.00
88.00
148.00
114.00
145.00
134.00
238.00
181.00

Simulation Info

Process id: com.redhat.bpms.examples.mortg
Process name: MortgageApplication
Process version: 1.0
Simulation start: Fri, 14 Mar 2014 18:59:30
Simulation end: Fri, 14 Mar 2014 20:00:06
Num. of Executions: 200
Interval 5 minutess
Process
MortgageApplication (com.redhat.bpms.exar
Process Elements
Troubleshoot (_FC260A58-AFA2-4F45-9D78-
Appraisal (_E8451731-3BAE-414A-82E1-F91
Increase Down Payment (_3FC2B32E-BF4E-
Financial Review (_876EB580-9313-42B2-88
Data Correction (_D279073F-0496-4E5E-829
_B6CF928B-1B1F-4735-ACB8-673946CB42A
Deny Mortgage (_E63830FC-3DC8-4297-A32
Reset Validation (_530A84FA-BBA3-4ECE-A
Validation (_9E6B5706-BB05-417B-864C-13I
Credit Report (_6D36721D-07B7-4AE8-A84C
Mortgage Calculations (_71142E93-4DD3-44
Qualify Borrower (_7E0FE170-245A-4AF6-A
Approve Mortgage (_30904A7B-D46C-497B-
Paths
Path 1 (Path-182700646-19)
Path 2 (Path-182700646-18)
Path 3 (Path-70371891-22)
Path 4 (Path-33155454-4)
Path 5 (Path-109222054-1)
Path 6 (Path-1020484871-0)

Figure 6.3-3:

6.4 Business Activity Monitoring

The *Dashbuilder* application included in BPM Suite 6 allows the creation of reports, panels, pages and entire workspaces for the purpose of Business Activity Monitoring. The BPMS database is set up as a source of data by default but other external data sources may also be defined and added.

Business Central includes links to both *Business Dashboards* and the *Process & Task Dashboard*. These links redirect the browser to the *jBPM* dashboard of the *Dashbuilder* application. The jBPM dashboard is preconfigured with sample reports on BPMS processes and tasks by running queries against the BPMS database.

Start the mortgage process multiple times and use various users to claim the tasks that get created. Undeploy the credit web service for a number of process instantiations to create a number of Troubleshoot tasks. Then proceed to log in to Business Central as different users and claim the tasks to generate a more meaningful report:

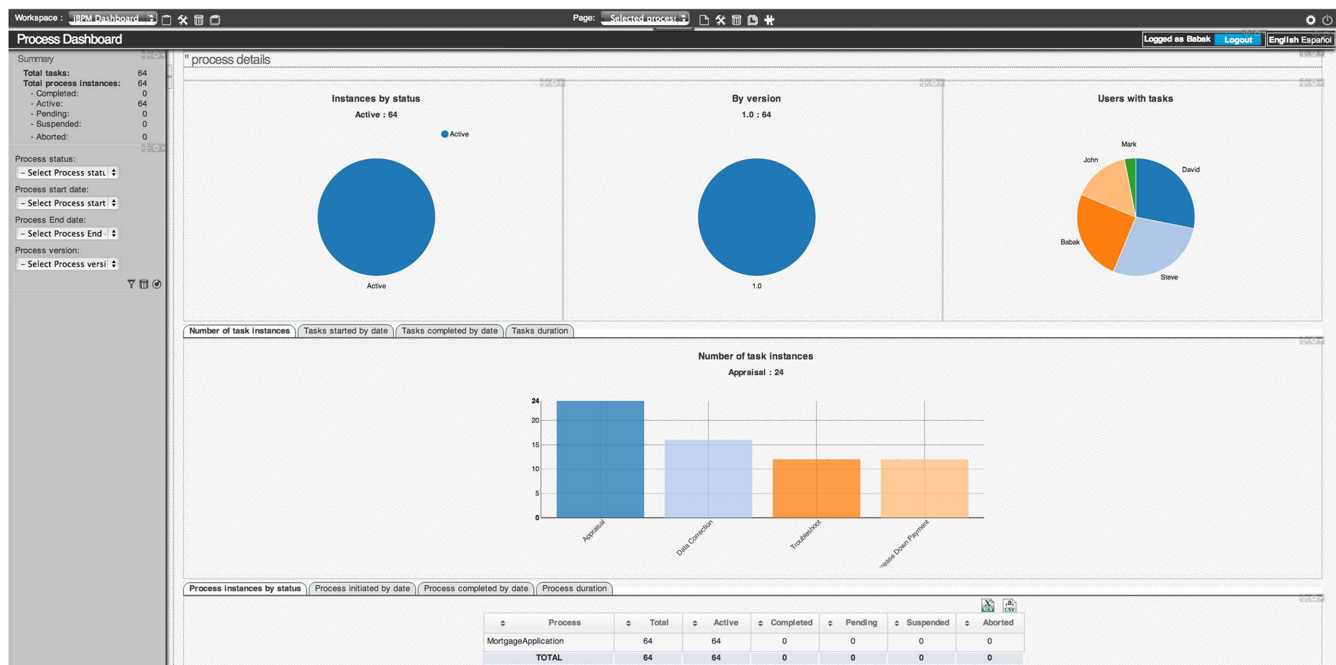

Figure 6.4-1:

6.5 Governance

Red Hat provides an implementation of the **SOA Repository Artifact Model and Protocol (S-RAMP)** specification by *OASIS*.[25] The specification and the implemented product can provide design-time governance for BPMS.

To use **Red Hat JBoss S-RAMP Repository**, download the separately provided *Red Hat JBoss S-RAMP Repository 6.0.0 Installer* from the Red Hat Customer Support Portal.[26]

While S-RAMP Repository is a core component of **JBoss Fuse Service Works**, it is also an entitlement provided for BPM Suite 6 and used for design-time governance with BPMS. For complete documentation of this component, refer to the Design Time Governance section of the JBoss FSW documentation.[27]

While the JBoss S-RAMP Repository may be configured to share the same Maven Repository with BPMS and therefore be aware of built artifacts, no integration is provided between the Git-based asset repository used in BPMS and S-RAMP in BPM Suite 6.0. To govern individual assets used in BPMS, manually upload the assets to S-RAMP. The S-RAMP repository provides a user-friendly interface, a REST API and a command-line interface that can all be used to create a manual or semi-automatic process for managing the BPMS asset repository content. Better integration with design-time governance is left to future versions of JBoss BPM Suite.

Governance workflows can be defined and implemented as BPMN 2.0 processes. This is achieved by configuring a query that detects interesting changes in the governance repository and triggers the deployed business process.[28]

6.6 Process Execution

While the simplest way to run a process is through Business Central, several alternative methods have been provided to accommodate various client requirements.

6.6.1 Business Central

Business Central provides a unified environment for design and development as well as execution and monitoring of business processes. The form designer helps create process forms that can instantiate a process while collecting the required initial information from the user. Forms that have been created with this tool are automatically associated with the process through a naming convention and any future attempt to run the process from Business Central renders the form and uses it to populate the process variables.

---

25 https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=s-ramp
26 https://access.redhat.com/jbossnetwork/restricted/softwareDetail.html?softwareId=27873&product=bpm.suite&version=&downloadType=distributions
27 https://access.redhat.com/site/documentation/en-US/Red_Hat_JBoss_Fuse_Service_Works/6/html/Development_Guide_Volume_3_Governance/chap-Design-Time_Governance.html
28 https://access.redhat.com/site/documentation/en-US/Red_Hat_JBoss_Fuse_Service_Works/6/html/Development_Guide_Volume_3_Governance/chap-Design-Time_Governance.html#sect-Governance_Workflows

6.6.2 Remote Client

Business Central also provides a REST server that can be used to start and interact with processes from a remote client. The client is deemed logically remote as it only interacts with the process through XML or JSON data sent through HTTP or JMS. This does not preclude the client application from physically residing in the same JVM or even being bundled in the same Web Application as Business Central.

The URL for sending REST / HTTP calls to Business Central always includes a static part that can be called the application context, which locates the server and the web application deployment. For a local default installation, the application context may be http://localhost:8080/business-central. This is always followed by */rest*, which is the sub-context for REST calls. All requests need to be authenticated calls. What follows that varies and depends on the type of call and the deployment and process that is being targeted.

6.6.2.1 Simple Process Calls

Simple process calls allow callers to start a new process instance, signal a waiting process, abort a running process, retrieve information about process variables or otherwise interact with a running process instance.

These calls either use simple HTTP GET methods or require HTTP POST methods that accept both JSON and XML as input.

For example, a process may be started through an HTTP POST call to the following URL:

/runtime/{deploymentId}/process/{procDefID}/start

For the BPMS example application running on a local default installation:

http://localhost:8080/business-central/rest/runtime/com.redhat.bpms.examples:mortgage:1/process/com.redhat.bpms.examples.mortgage.MortgageApplication/start

If any process variables are expected when the process is started, they should be provided as query parameters. To represent a map, submit each key-value pair of the map as a separate query parameter which suffixing the key with “map_”. For example, to pass a map of process variables with two entries where one sets the process variable *name* to *John* and the other sets the process variable *age* to *30*, provide the following query parameters:

.../start?map_name=John&map_age=30

A simpler example is a REST call to retrieve basic information about a running process instance. This is achieved through an HTTP GET call to the following URL:

/runtime/{deploymentId}/process/instance/{procInstanceID}

For the BPMS example application running on a local default installation, to query the first process instance that was created, this URL would look as follows:

http://localhost:8080/business-central/rest/runtime/com.redhat.bpms.examples:mortgage:1/process/instance/1

Note that this only queries the runtime, which excludes any process that has reached a wait state (user task, async callback, etc). Use History Calls for more complete information.

6.6.2.2 Simple Task Calls

Simple task calls allow callers to manage the lifecycle of a task by claiming it, releasing it, forwarding it, skipping it or otherwise managing or altering its assignment and lifecycle.

Simple task calls also query the task content and data, as well as complete a task while assigning required variables to the task.

These calls either use simple HTTP GET methods or require HTTP POST methods that accept both JSON and XML as input.

For example, a task content may be retrieved through an HTTP GET call to the following URL:

/task/content/{contentID}

For the BPMS example application running on a local default installation, to retrieve the details of task 1, this URL would look as follows:

http://localhost:8080/business-central/rest/task/1/

To claim this task for a given user, make an HTTP POST call to the following URL:

http://localhost:8080/business-central/rest/task/1/claim

The next step, according to the task lifecycle, would be to start work on the task on behalf of the user. This, again, is simply a matter of making an HTTP POST call to the following URL:

http://localhost:8080/business-central/rest/task/1/start

Finally, the task may be completed by making a call to:

http://localhost:8080/business-central/rest/task/1/complete

If any variables are expected when the task is completed, they should be provided as query parameters. To represent a map in HTTP calls, submit each key-value pair of the map as a separate query parameter which suffixing the key with "map_". For example, to pass a map of process variables with two entries where one sets the process variable *name* to *John* and the other sets the process variable *age* to *30*, provide the following query parameters:

.../complete?map_name=John&map_age=30

6.6.2.3 History Calls

To access the audit log and retrieve historical process and task information, issue simple GET commands to the history API.

For example, for an overview of all the process instances of the *MortgageApplication* process, send an HTTP GET query without any parameters as follows:

http://localhost:8080/business-central/rest/history/process/com.redhat.bpms.examples.mortgage.MortgageApplication

To get information on a specific process instance, use the process instance ID, for example:

http://localhost:8080/business-central/rest/history/instance/1

To retrieve the variable values of this process:

http://localhost:8080/business-central/rest/history/instance/1/variable

This can be further narrowed down to investigate how the value of a particular process variable changed over time. For example, to inquire about the *application* variable of this process, use the following query:

http://localhost:8080/business-central/rest/history/instance/1/variable/application

To view the value of the application in all process instances of all process definitions in this deployment, issue the following query:

http://localhost:8080/business-central/rest/history/variable/application

There is also a query that searches the variables directly for a value. As a practical matter, this results in a search of all process instances of any process definition type for a process variable with a given value.

For example, to look for mortgage applications by an applicant with a given Social Security Number, the following query may be issued:

http://localhost:8080/business-central/rest/history/variable/ssn/value/333224442

6.6.2.4 Command Execution

Advanced users looking to send a batch of commands via the REST API can use the execute operation. This is the only way to have the REST API process multiple commands in one operation.

The execute calls are available through both REST and JMS API. The only accepted input format is JAXB, which means that REST calls over HTTP may only send XML and cannot use the JSON format.

Issue Execute commands to /runtime/{deploymentId}/execute. For the sample deployment, the URL would be:

http://localhost:8080/business-central/rest/runtime/com.redhat.bpms.examples:mortgage:1/execute

The posted content must be the JAXB representation of an accepted command. Refer to the official Red Hat documentation for a full list of commands.[29]

6.6.2.5 Client API for REST or JMS calls

A client API is available to help Java clients interact with the BPMS REST API.

For example, to start a new instance of the mortgage process:

```
String deploymentId = "com.redhat.bpms.examples:mortgage:1";
String applicationContext = "http://localhost:8080/business-central";
String processId = "com.redhat.bpms.examples.mortgage.MortgageApplication";
URL jbpmURL = new URL( applicationContext );

RemoteRestRuntimeFactory remoteRestSessionFactory =
    new RemoteRestRuntimeFactory( deploymentId, jbpmURL, userId, password );
```

29 https://access.redhat.com/site/documentation/en-US/Red_Hat_JBoss_BPM_Suite/6.0/html/Development_Guide/Execute_calls.html

```
RuntimeEngine runtimeEngine = remoteRestSessionFactory.newRuntimeEngine();
KieSession kieSession = runtimeEngine.getKieSession();

Map<String, Object> processVariables = new HashMap<String, Object>();
Application application = new Application();
application.setAmortization( 30 );
...
processVariables.put( "application", application );

kieSession.startProcess( processId, processVariables );
```

This code results in the start process command being sent to the REST API.

Notice the choice of runtime factory in the above example:

```
RemoteRestRuntimeFactory remoteRestSessionFactory =
    new RemoteRestRuntimeFactory( deploymentId, jbpmURL, userId, password );
```

An alternative choice would have been the JMS API:

```
RemoteJmsRuntimeEngineFactory jmsFactory =
   new RemoteJmsRuntimeEngineFactory( deploymentId, new InitialContext() );

RuntimeEngine runtimeEngine = jmsFactory.newRuntimeEngine();
```

The *RuntimeEngine* interface provides abstraction from the underlying method and protocol so the rest of the client code remains unchanged when using JMS instead of HTTP REST calls.

The client API may also be used to create Command objects and execute them through REST/HTTP or JMS, or in fact execute a batch of commands at once.

6.6.3 Local Application

While the Business Central forms may provide a suitable method of starting processes and completing tasks for some client requirements, many will require greater control and flexibility on their user interface. In some cases the starting of processes and the completion of tasks may not directly be user-driven and in other cases when it is, the command may come from a JSF or other UI technology.

These applications can be deployed on the same server or cluster as BPMS. By declaring a dependency on the required BPMS modules and configuring the appropriate datasource, custom code may be used to interact with processes and tasks.

Custom applications can declare a maven dependency on the required packages, including the BPMS project.

Java EE dependency injection can simplify access to the *KieSession*, for example:

```
@Inject
@Singleton
private RuntimeManager runtimeManager;

RuntimeEngine runtime = runtimeManager.getRuntimeEngine(EmptyContext.get());
```

```
KieSession ksession = runtime.getKieSession();

ProcessInstance processInstance = ksession.startProcess(...
```

The runtime strategy may be specified through annotations as *@Singleton*, *@PerProcessInstance* or *@PerRequest*.

To interact with tasks, use the RuntimeManager to get the RuntimeEngine and then retrieve the TaskService and KieSession from the engine. At the end, if outside a container managed transaction, dispose of the retrieved engine explicitly.

Even when calls to start a process or complete a task originate from a custom application deployed alongside BPMS, the decision to treat it as a local client or a REST client requires some thought and depends on whether a loosely-coupled or a tightly-coupled design between the application and BPMS is more desirable.

6.7 Maven Integration

JBoss BPM Suite 6.0 uses Maven for build management. While the Maven repository is locally maintained and accessible through the file system, it is often the case that a BPMS cluster should be treated as a logical entity and intrusive integration including access to the local file system of an instance may have adverse side effects.

Once a project has been built by Business Central, BPMS 6.0 serves its Maven artifact by HTTP through the */maven2* sub-context. For example, to access the Maven project file for the mortgage application, the following query may be used:

http://localhost:8080/business-central/maven2/com/redhat/bpms/examples/mortgage/1/mortgage-1.pom

Similarly, the build JAR artifact for the project may be retrieved from the following URL:

http://localhost:8080/business-central/maven2/com/redhat/bpms/examples/mortgage/1/mortgage-1.jar

6.8 Session Strategy

JBoss BPM Suite support three different strategies for session management and reuse. Each strategy is described in more detail here. The choice of session strategy largely depends on the anticipated level of concurrency as well as the use of the rule engine in the processes. Using a single global session means that any object inserted into rule engine working memory for a process instance may impact firing of rules for other instances of the same process or even an entirely different process definition.

The concurrency consideration is also an important one in the choice of session strategy. Generally speaking, any work done within the scope of a KIE session is single-threaded. When a session is shared between various requests (or process instances), that means that the processing is serialized. This may or may not be acceptable for a given use case.

6.8.1 Singleton

In this model, a single KIE session is created within the given scope and used for all new and existing process instances of all process definitions.

When creating a deployment through Business Central, use *Singleton* to select this strategy.

The equivalent annotation, used in the client API, is *@Singleton*.

6.8.2 Per Process Instance

In this model, a new KIE session is created within the given scope for every new process instance of a given process definition and reused again when signaling, continuing or otherwise executing the same process instance in the future.

When creating a deployment through Business Central, use *Process instance* to select this strategy.

The equivalent annotation, used in the client API, is *@PerProcessInstance*.

6.8.3 Per Request Session

In this model, a new KIE session is created for every request, regardless of whether a new process instance is being created or a process instance previously created in another session is being continued.

When creating a deployment through Business Central, use *Request* to select this strategy.

The equivalent annotation, used in the client API, is *@PerRequest*.

6.9 Timer Implementation

BPMS 6 requires and uses timers for various reasons. The default timer used in BPMS 6.0 is an internal implementation using thread pools. Upon initialization, the BPMS environment looks for a Java system property called *org.quartz.properties*. If found, the value of this property is presumed to be the fully qualified location of the quartz property file.

In a cluster environment, Quartz should be configured to use a central database for persistence. The recommended interval for cluster discovery is 20 seconds and is set in the

org.quartz.jobStore.clusterCheckinInterval of the *quartz-definition.properties* file. Depending on your set up consider the performance impact and modify the setting as necessary.

Refer to the official Red Hat documentation for details on how to configure Quartz in BPMS 6.0.[30]

6.10 REST Deployment

One required step in build automation is deployment of business processes. To accommodate this requirement, the REST API provides support for deployment.

Assuming the deployment ID of the mortgage application, issue an HTTP POST to the following URL to trigger deployment:

http://localhost:8080/business-central/rest/deployment/com.redhat.bpms.examples:mortgage:1/deploy

The deploy operation is asynchronous and deployment will continue and complete after the response of the REST operation has been returned.

The session startegy can also be specified at the time of deployment through the REST interface. For example, to use “per process instance” as the session strategy:

http://localhost:8080/business-central/rest/deployment/com.redhat.bpms.examples:mortgage:1/deploy?strategy=PER_PROCESS_INSTANCE

6.11 Continuous Integration

Standard and widely used software tooling such as Git and Maven make it easier to include BPMS applications in an organization's various automated processes. Assets can be easily retrieved from and placed in the asset repository as outlined in Asset Repository Interaction. Maven Integration makes it easier to include BPMS In the build process and deployment. To deploy BPMS projects, deployment can be triggered through a simple REST call.

Additional functionality exposed through the REST API allows a caller to create an organizational unit, create a repository and create a project within that repository. At that point, assets can be pushed to the project repository before triggering build and deployment. Refer to the official Red Hat documentation for further details on the REST API for the Knowledge Store.[31]

This collection of features and resources helps implement continuous integration in BPMS projects.

30 https://access.redhat.com/site/documentation/en-US/Red_Hat_JBoss_BPM_Suite/6.0/html/Installation_Guide/Setting_up_Quartz.html

31 https://access.redhat.com/site/documentation/en-US/Red_Hat_JBoss_BPM_Suite/6.0/html/Development_Guide/chap-REST_API.html#sect-Knowledge_Store_REST_API

7 Conclusion

This reference environment sets up a cluster of BPMS 6.0.1 on top of a JBoss EAP 6 Cluster, as outlined and automated in the JBoss EAP 6 Clustering Reference Architecture. This automation is further extended in this reference architecture to include the configuration of the BPMS servers on top of the EAP cluster.

Combined with a ZooKeeper ensemble, a central database and other related configuration, this environment is set up as both a design-time and runtime cluster. The design-time cluster helps replicate assets during development, a step which may not necessary for most production servers, as long as deployments are properly pushed to every node.

By walking through every step in the design and development of the example application, various design techniques and best practices are outlined and presented in near-realistic circumstances.

Various other technical considerations are discussed as the software development life cycle for BPMS is reviewed, touching on disparate topics including the design environment, build and deployment, governance and monitoring, execution and runtime configuration.

Appendix A: Revision History

Revision 1.0	04/09/14	Babak Mozaffari
Initial Release		

Appendix B: Contributors

We would like to thank the following individuals for their time and patience as we collaborated on this process. This document would not have been possible without their many contributions.

Contributor	Title	Contribution
Maciej Swiderski	Sr. Software Engineer	Review, Cluster, Exception Handling, Troubleshooting
Ivo Bek	Quality Engineer	Review
Prakash Aradhya	Sr. Principal Product Manager, Technical	Review
Eric Schabell	Sr. Principal Product Marketing Manager	Review
Jeffrey DeLong	Sr. Manager, Solution Architecture	Review
Jeffrey Bride	Manager, Solution Architecture	Review