



Red Hat Reference Architecture Series

JBoss BPM Suite 6.3

Business Process Management with Red Hat JBoss BPM Suite 6.3

Jeremy Ary, Babak Mozaffari

Version 1.0, July 2016

Table of Contents

- Comments and Feedback 2
 - Staying In Touch..... 2
 - Like us on Facebook: 2
 - Follow us on Twitter: 2
 - Plus us on Google+ 2
- 1. Executive Summary 3
- 2. Red Hat JBoss BPM Suite 6.3 4
 - 2.1. Overview 4
 - 2.2. Installation Options 5
 - 2.2.1. Server Platform 5
 - 2.2.2. Clustering 5
 - 2.2.3. Red Hat JBoss Developer Studio 6
 - 2.3. Administration and Configuration 7
 - 2.3.1. Business Central 7
 - 2.3.2. Asset Repository 8
 - 2.3.3. Data Persistence 8
 - 2.3.4. Audit Logging 8
 - 2.3.5. Task Execution Configuration 8
 - 2.4. Design and Development 9
 - 2.4.1. Data Model 9
 - 2.4.2. Process Designer 9
 - 2.4.3. Forms 9
 - 2.5. Process Simulation 10
 - 2.6. Business Activity Monitoring 11
 - 2.7. Rest API 11
 - 2.7.1. Knowledge Store REST API 12
 - 2.7.2. Deployment REST API 12
 - 2.7.3. Runtime REST API 12
- 3. Reference Architecture Environment 13
 - 3.1. Overview 13
 - 3.2. BPMS 6.3.0 13
 - 3.3. JBoss EAP 6.4.7 Cluster 13
 - 3.4. ZooKeeper Cluster 14
 - 3.5. Execution Servers 14
 - 3.6. PostgreSQL database 14
 - 3.7. BPM Example Application 15

- 3.8. Runtime Cluster 16
- 4. Creating the Environment 17
 - 4.1. Prerequisites 17
 - 4.2. Downloads 17
 - 4.3. Installation 18
 - 4.3.1. JBoss Enterprise Application Platform 18
 - 4.3.2. PostgreSQL Database 20
 - 4.3.3. JBoss BPM Suite 20
 - 4.4. Configuration 22
 - 4.4.1. JBoss BPM Suite 22
- 5. Design and Development 23
 - 5.1. BPM Suite Example Application 23
 - 5.2. Project Setup 23
 - 5.2.1. Business Central 23
 - 5.2.2. Repositories 25
 - 5.2.3. Projects 28
 - 5.3. Data Model 30
 - 5.4. Business Process 35
 - 5.4.1. Create New Process 35
 - 5.4.2. Initial Process Form 40
 - 5.4.3. Validation 51
 - 5.4.4. Data Correction 64
 - 5.4.5. Web Service Task 68
 - 5.4.6. Mortgage Calculation 76
 - 5.4.7. Qualify Borrower 91
 - 5.4.8. Increase Down Payment 92
 - 5.4.9. Financial Review 95
 - 5.4.10. Appraisal 97
 - 5.4.11. Swimlanes and Business Continuity 100
 - 5.4.12. Final Process Model 102
- 6. Life Cycle 103
 - 6.1. Asset Repository Interaction 103
 - 6.2. JBoss Developer Studio 103
 - 6.3. Process Simulation 105
 - 6.4. Business Activity Monitoring 106
 - 6.5. Governance 107
 - 6.6. Process Execution 108
 - 6.6.1. Business Central 108

- 6.6.2. Remote Client 108
- 6.6.3. Local Application 113
- 6.7. Maven Integration 113
- 6.8. Session Strategy 114
 - 6.8.1. Singleton 114
 - 6.8.2. Per Process Instance 114
 - 6.8.3. Per Request Session 115
- 6.9. Timer Implementation 115
- 6.10. REST Deployment 115
- 6.11. Continuous Integration 116
- 7. Conclusion 117
- Appendix A: Revision History 118
- Appendix B: Contributors 119

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

Linux is a registered trademark of Linus Torvalds. 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. Microsoft and Windows are U.S. registered trademarks of Microsoft Corporation. UNIX is a registered trademark of The Open Group. Intel, the Intel logo and Xeon are registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries. OpenStack is the trademark of the OpenStack Foundation. All other trademarks referenced herein are the property of their respective owners.

© 2016 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 of the 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/>

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) 6.3** 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 6.3

2.1. Overview

Red Hat JBoss BPM Suite (BPMS) 6.3 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 6.3** 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.

BPMS 6.3 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.

Red Hat JBoss BPM Suite also includes **Business Resource Planner**. 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 (reference [official Red Hat BPM Suite documentation](#) for more information).

BPMS 6.3 also includes **Realtime Decision Server** and **Intelligent Process Server**, both which serve as standalone, out-of-the-box components that can be used to instantiate and execute rules in provisioned containers through interfaces available for REST, JMS or a Java client side application. These self-contained environments encapsulate a compiled rule package and deployed rule instance, thus allowing for a looser coupling of rules and the execution code of the application. Created as a web deployable WAR file, these servers can be deployed on any web container.

Both decision servers ship with default extensions for JBoss BRMS & **Business Resource Planner**, with Intelligent Process Server also adding extensions for the BPM Suite. These servers, based on the KIE Execution Server, have low footprints, with minimal memory consumption, and therefore, can be deployed easily on a cloud instance. Each instance can open and instantiate multiple KIE Containers which allows execution of multiple rule services in parallel. Containers generated via execution

servers can be managed from within Business Central via *Rule Deployment*. In future BPM Suite releases, the currently-utilized Business Central execution server functionality is slated to be replaced with the newer KIE execution server variations. Currently, when KIE execution server derivatives are employed, Business Central only allows for management of the runtime servers; no runtime aspects, such as Process Definition, Process Instances, Tasks, etc., can be used. Such functionality is currently slated for inclusion in version 7. Even so, should you wish to take advantage of the KIE Servers' advantages of decoupling rules execution from authorship and maintenance, more information on the deployment, management, and interacting with the Realtime Decision & Intelligent Process Servers can be found in the [Red Hat JBoss BRMS 6.3 User Guide](#).

2.2. Installation Options

2.2.1. Server Platform

Red Hat JBoss BPM Suite offers means for installation on various containers (reference official [Red Hat BPM Suite Installation Guide](#) for further information):

- *jboss-bpmsuite-6.MINOR_VERSION-installer.jar*: jar utility providing guided installations of deployments on [Red Hat JBoss Enterprise Application Platform \(EAP 6\)](#).
- *jboss-bpmsuite-6.MINOR_VERSION-deployable-eap6.x.zip*: adapted for deployment on [Red Hat JBoss Enterprise Application Platform \(EAP 6\)](#).
- *jboss-bpmsuite-6.MINOR_VERSION-deployable-was8.zip*: adapted for deployment on [WebSphere 8.5](#).
- *jboss-bpmsuite-6.MINOR_VERSION-deployable-wls12c.zip*: adapted for deployment on [Oracle Weblogic 12c](#).
- *jboss-bpmsuite-6.MINOR_VERSION-deployable-generic.zip*: 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 6.3 installation is a base installation of EAP 6.4.0 patched to version 6.4.7.

A separate download is provided for other supported containers. Dependent on the target container, security policy and possibly other small changes to configuration files are likely required and need to be incorporated based on the container instructions.

2.2.2. Clustering

For Red Hat JBoss BPM Suite, clustering may refer to various components and aspect 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.

- *Back-end database*: database with the state data, such as process instances, KIE sessions, history log, etc., for fail-over purposes

For further instructions on clustering the BPMS environment, including details on using the installer script to generate a clustered setup as detailed in Chapter 4, refer to the [official Red Hat BPM Suite Installation Guide](#).

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](#). 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 **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. For instructions on installing the plugins, setting the runtime library, configuring the BPMS Server and importing projects from a Git repository, refer to the relevant section of the [Red Hat BPM Suite Installation Guide](#).

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*: administers BPMS system and has full access rights to make any necessary changes, 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.
- *rest-all*: May use all REST URLs
- *rest-project*: May use REST URLs relating to project management, including repository and organizational unit management
- *rest-deployment*: May use REST URLs relating to deployment management
- *rest-process*: May use REST URLs relating to process management
- *rest-process-read-only*: May use REST URLs that return info about processes
- *rest-task*: May use REST URLs relating to task management
- *rest-task-read-only*: May use REST URLs that return info about tasks
- *rest-query*: May use the query REST URLs
- *rest-client*: May use the REST URL relating to the java remote client

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](#).

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.

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.

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](#).

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.

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.

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 a tool for modeling business processes within BPM Suite. 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:

- *JBDS 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 JBDS 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](#).

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. For further details, consult the [official Red Hat documentation](#).

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

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](#).

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. For more information, refer to the [official Red Hat documentation](#).

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.

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.

3. Reference Architecture Environment

3.1. Overview

This reference architecture takes advantage of the provided BPM Suite 6.3 installer script to establish a basic clustered environment configuration with an architectural structure transferable to use cases on a larger scale.

The automatic configuration creates three **ZooKeeper** instances, a **Helix** cluster that uses these instances, and two **Quartz** datastores (one managed and one unmanaged). This Red Hat JBoss BPM Suite setup consists of two **EAP** nodes that share a **Maven** repository, use Quartz for coordinating timed tasks, and have **business-central.war**, **dashbuilder.war**, and **kie-server.war** deployed. To customize the setup to fit your scenario, or to use clustering with the deployable ZIP, see the BPM Suite Installation Guide Section regarding [Custom Configuration \(Deployable ZIP\)](#). You can also get more information in the *JBoss EAP documentation*.

3.2. BPMS 6.3.0

BPMS 6.3 includes the Business Central and Dashbuilder web applications, which are hosted on the EAP 6.4.7 servers. Supporting libraries are added to the EAP 6.4.7 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 6.3 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 6.4.7 Cluster

BPMS 6.3 includes a number of modules and web applications that are configured and deployed on top of JBoss EAP 6.4.7. Accordingly, the foundation of the BPMS environment is the setup of an EAP 6.4.7 cluster via the BPM Suite installer script.

3.4. ZooKeeper Cluster

In this reference architecture, ZooKeeper is clustered alongside EAP, so that three nodes of ZooKeeper and Helix run alongside each of the EAP 6.4.7 nodes.

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. For example, both a three-member and a four-member cluster can only withstand the loss of a single member without losing a functioning majority, so groups with an odd number of members provide higher efficiency.

ZooKeeper allows BPMS to replicate its Git repositories. 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. Execution Servers

Whereas ZooKeeper was previously responsible for replicating other locally maintained data, including deployments, this is no longer the case. When utilizing Business Central, as exemplified by the example application developed herein, the persistence database is utilized for deployment synchronization and other BPM data. When choosing to utilize the Realtime Decision Server or Intelligent Process Server, both being derivations of the KIE Server, the controller of the execution server assumes responsibility for deployment replication.

With BPM Suite 6.3, when choosing to use a KIE Server derivative, Business Central can be used to manage the KIE Servers and their containers, however, you can't utilize the runtime views of Business Central, such as Process Definition, Process Instances, Process Tasks, etc. This functionality is currently slated for inclusion in BPM Suite 7, where the Business Central execution server will be replaced with KIE Server integration.

3.6. PostgreSQL database

The BPM Suite installer script requires a data store for both EAP and BPMS instances, the latter of which utilizes persistence of active business process data, audit log and Quartz timer data. While various storage options are possible, given the interface with the persistence layer via JDBC, a single instance of PostgreSQL is assumed to have been established beforehand within this document with user *jboss* and password *password1!* allowing general access to all required tables.

While this single instance could be considered a single point of failure, the database itself can also be clustered, but the focus of this effort is BPMS 6.3 and clustering of the data source is beyond the scope of this reference architecture.

3.7. 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.

3.8. 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. However the content of these repositories only change during design and development.
- *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). Within BPM Suite, JMS is utilized for asynchronous execution and, when used alongside EAP Clustering, the JMS broker uses load balancing to execute jobs in all cluster members in a distributed fashion. 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 analysts 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. With the incorporation of KIE Server functionality, partitioning of business knowledge and clustering of environment components allows for independent scalability of such runtime environments, while simplified usage and deployment of different rules knowledge bases within different containers also provides a natural decoupling between the authoring and execution phases of a rules-based application.

4. Creating the Environment

4.1. Prerequisites

This reference architecture assumes a supported platform (Operating System and JDK). For further information, refer to the BPM Suite Installation Guide section regarding [supported environments](#).

As mentioned before, with minor changes, almost any RDBMS may be used in lieu of PostgreSQL Database for both this reference environment and the EAP 6.4.7 cluster, but if PostgreSQL is used, the details of the download and installation are also considered a prerequisite for this reference architecture. At the time of writing, version 9.2 was installed via **dnf** for the purposes of this document. On a RHEL system, installing PostgreSQL can be as simple as running:

```
# dnf install postgresql-server
```

4.2. Downloads

This reference architecture makes mention and use of several files included as attachments to this document. These files will be used in configuring the reference architecture environment & providing a copy of the application developed herein for reference:

<https://access.redhat.com/node/2474561/40/0>

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 6.3.0 and its supplementary tools from Red Hat's Customer Support Portal:

- Red Hat JBoss BPM Suite 6.3.0 Installer
- Red Hat JBoss BPM Suite 6.3.0 Supplementary Tools

4.3. Installation

4.3.1. JBoss Enterprise Application Platform

Begin installation of the EAP server by executing the EAP installer script (different from the BPM Suite installer script) `jboss-eap-6.4.0-installer.jar` in console mode:

```
# java -jar jboss-eap-6.4.0-installer.jar -console
```

After selecting a language to continue (or simply pressing enter to accept the default choice indicated in brackets, a common recurrence throughout the installer script), you will be prompted with Terms and Conditions, to which you can reply 1 to continue. Note that you will receive this prompt through the script following each step of the process:

- *Port & directory*: for the purposes of this document, port 9999 and `/opt/EAP-6.4_[active|passive]` will be utilized, along with the default selection of packs to install.
- *Pack installations*: default selection of packs is acceptable for the purposes of this document.
- *Admin user*: accept the default username `admin` and provide, then confirm the password `password1!` for the `admin` user.
- *Quickstarts*: while the quickstart examples aren't required for the purposes of this document, they should not interfere with the needed stack.
- *Port configuration*: accept the default of default port bindings for standalone and domain modes.
- *IPv6*: this document will not assume usage of IPv6.
- *Server launch*: in order to gain familiarity with starting the server outside of the script, accept the default of *Don't start the server*.
- *Logging levels*: the default selection of *No* is acceptable.
- *Runtime environment*: Select the default selection for default configuration, then accept.

At this point, you should be able to test and verify connectivity to your database. If you have trouble, please review your data source installation, run status, port availability, and reattempt. Once successful, you can proceed and the installer script will then perform various steps to complete the installation of EAP 6.4.0 at the location provided. When completed, the script will offer a chance to generate an automatic installation script & properties file, which isn't necessary, but could be leveraged to expedite installation on other nodes. For the purposes of this installation, the default of *no* is selected and the installation is completed.

Following base installation, we can now start the server up in standalone mode in order to apply the 6.4.7 patch file available from the [Red Hat Access Portal within the Patches tab](#) of the EAP version downloads:

```
# cd /opt/EAP-6.4_active/bin
# ./standalone.sh -b 0.0.0.0 -bmanagement 0.0.0.0
```

Note that in the lines above, an IP binding of `0.0.0.0` is provided for both the server & management ports of the address. If you've chosen to run installations across various virtual machines, you may also find these parameters necessary to avoid configuring the server away from `localhost`.

Once server startup is completed, you can access the administrative UI server screen by accessing the URL `172.16.71.100:9990/console`, substituting the IP of the running server. Once logged in, the UI provides an option for Patch Management from the homescreen.

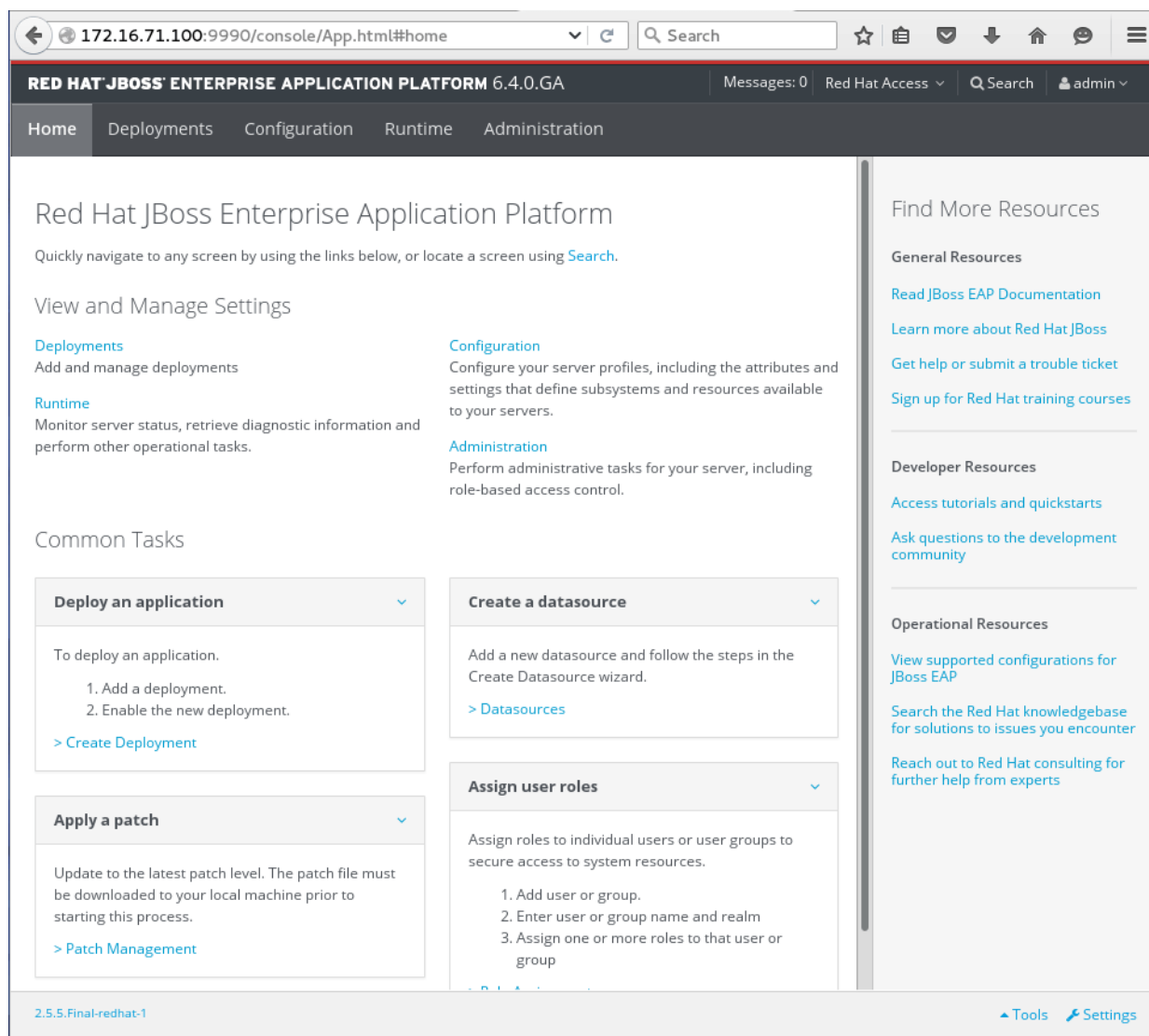


Figure 1. EAP Administration UI Console

By accessing Patch Management, you can then upload and apply the EAP 6.4.7 patch file to your server and restart when prompted. Once successful, you can now shut down the standalone EAP instance and proceed below.

4.3.2. PostgreSQL Database

If not already completed, install a single instance of PostgreSQL at this point. Once established, create an *eap6* database for the *jboss* user with password *password1!*.

Additionally, create new databases for BPMS, Business Central, and Dash Builder with the same *jboss* user as the owner:

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

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

4.3.3. JBoss BPM Suite

To install BPM Suite 6.3.0 on the freshly installed & patched instance of JBoss EAP 6.4.7, utilize the BPM Suite installer script, which should have already been downloaded from the [Red Hat BPM Suite Software Downloads Access Portal](#). Run the script in console mode:

```
# java -jar jboss-bpmsuite-6.3.0.GA-installer.jar -console
```

Yet again, you will be prompted with a series of steps provided by the installer script to complete the installation:

- *Terms and Conditions*: reply 1 to continue, yet again the "confirmation" prompt will be a common recurrence throughout the provided installer script
- *Port & directory*: for the purposes of this document, port 9999 and */opt/EAP-6.4_[active | passive]* (the HOME directory of the previous EAP installation performed) will be utilized.
- *Administrative user*: since an administrative user already exists, we can choose the default and skip adding a new admin user.
- *BPM Suite username*: the provided default username of *bpmsAdmin* is acceptable for the administrative user we will utilize further on to access the BPM Suite UI & create our application. Provide the password **password1!** for the user.
- *Additional user roles*: no further roles are necessary for the *bpmsAdmin* user at this time, so accept the default and move on.
- *Java security manager*: for the purposes of this document, the application container's

authentication and authorization mechanisms are sufficient for our security needs, so accept the default to bypass enabling of the JSM.

- *IPv6*: this document will not assume usage of IPv6.
- *Runtime environment*: Select the advanced selection of *[1]* to perform default configuration at this time, then accept.
- *Advanced Configuration*: accept defaults (blank, meaning skip) for all questions except for *Install clustered configuration*, *Install Business-Central Datasource* & *Install Dashbuilder Datasource*, which should all be marked 1 to complete.
- *Advanced - JDBC Driver*: choose 3 for PostgreSQL & provide the path to an appropriate driver .jar file, then hit enter to continue rather than providing second jar path.
- *Advanced - Datasource*: select the default of *[0]* for *postgresql* , then accept the default JNDI name provided, as well as the defaults for min/max pool size, choose not to use a securityDomain for the datasource. Provide the username and password of *jboss* and *password* which should have been previously established while following the PostgreSQL configuration portion of the EAP 6 Clustering Reference Architecture document.
 - Quartz database connection URL: *jdbc:postgresql://172.16.71.100:5432/bpms*
 - Quartz database user: *jboss*
 - Quartz database password: *password*
 - datasource type: accept default of *postgresql*
 - Business Central datasource name: *businesscentral*
 - Business Central JNDI name: accept default of *java:jboss/PostgresBusinessCentralDS*
 - min/max pool: accept defaults
 - securityDomain: choose **NOT** to use a securityDomain
 - username: *jboss*
 - password: *password*
 - connection URL: *jdbc:postgresql://172.16.71.100:5432/businesscentral*
 - Dashbuilder datasource name: *dashbuilder*
 - Dashbuilder JNDI name: accept default of *java:jboss/PostgresDashBuilderDS*
 - min/max pool: accept defaults
 - securityDomain: choose **NOT** to use a securityDomain
 - username: *jboss*
 - password: *password*
 - connection URL: *jdbc:postgresql://172.16.71.100:5432/dashbuilder*

After the installation finishes, *DO NOT* select to run the server immediately. When completed, the script will confirm with [**Console installation done**] and return to the command prompt. You then

need to start the cluster by moving to the directory `$EAP_HOME/jboss-brms-bpmsuite-6.3-supplementary-tools/helix-core` and executing the included launch script:

```
./startCluster.sh
```

This script launches the Helix cluster and ZooKeeper instances. You may now start the newly installed EAP server cluster in domain mode by moving to the directory `$EAP_HOME/bin` and running:

```
./domain.sh -b 0.0.0.0 -bmanagement 0.0.0.0
```

4.4. Configuration

Should you find that your ZooKeeper instances are unable to communicate, it's possible that some ports need to be opened in order to proceed. 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 172.16.17.0/24 --dport 2181 -m tcp -j ACCEPT
# iptables -I INPUT 24 -p tcp -s 172.16.17.0/24 --dport 2888 -m tcp -j ACCEPT
# iptables -I INPUT 24 -p tcp -s 172.16.17.0/24 --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 BPM Suite

At this point, the installer script has completed installation of BPM Suite in a clustered manner with deployments of Business Central and Dashbuilder.

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 herein. 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-bpmsuite-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 as part of the Quick Starts. 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 Quick Starts.

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 throughout.

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:

Business Central Login

image::images/business-central-login.png[Business Central Login, scaledwidth="90%", align="center

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.

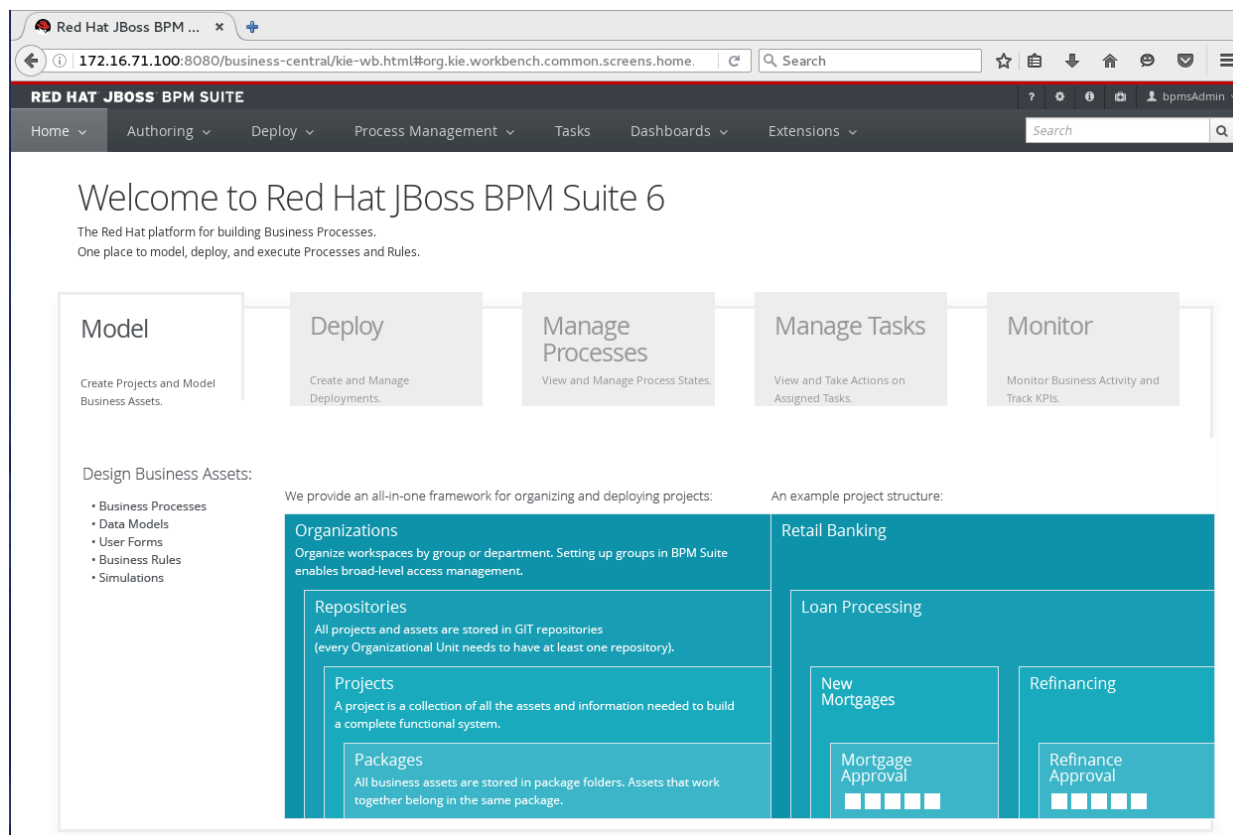


Figure 2. Business Central Welcome Page

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:

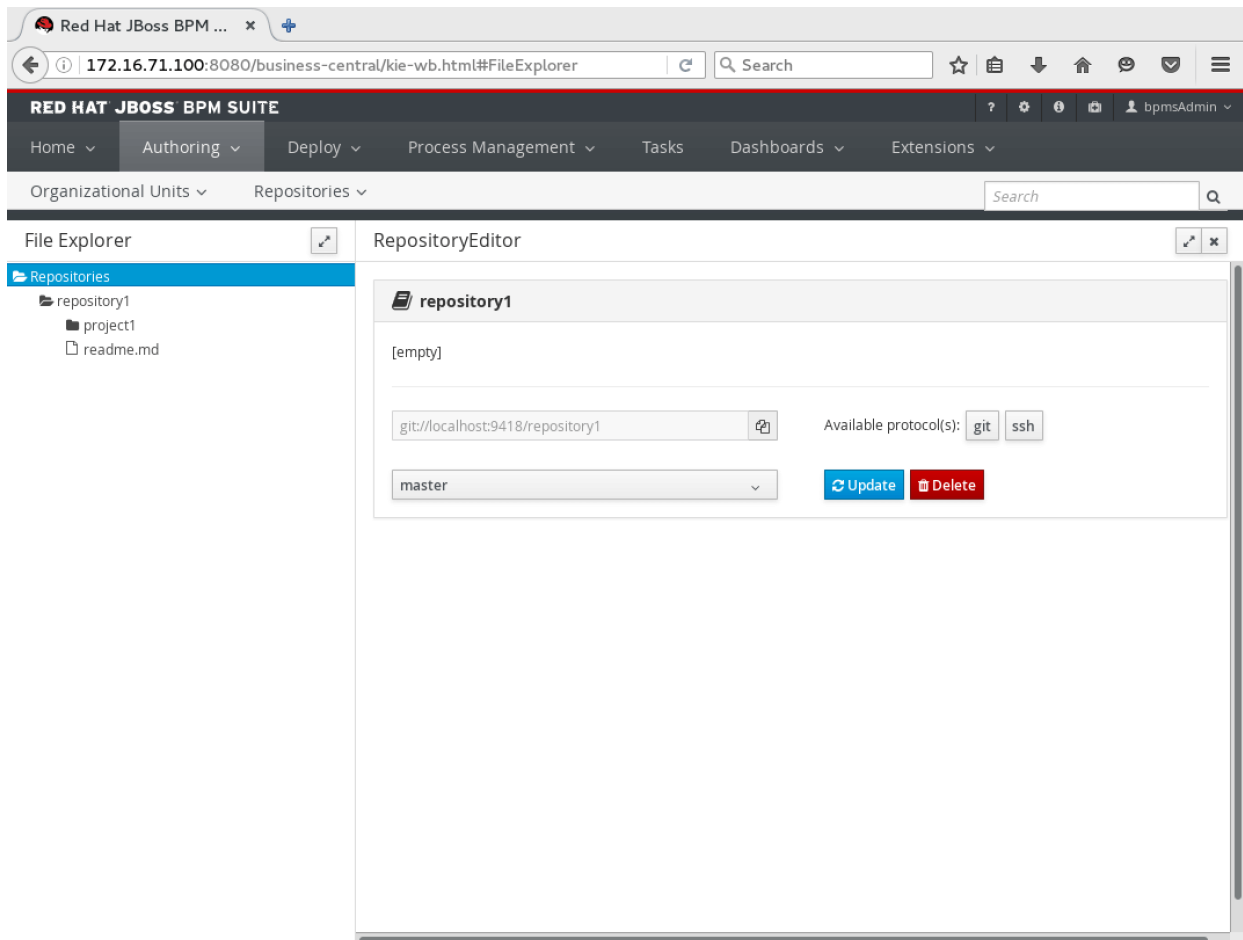


Figure 3. New Repository Creation

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

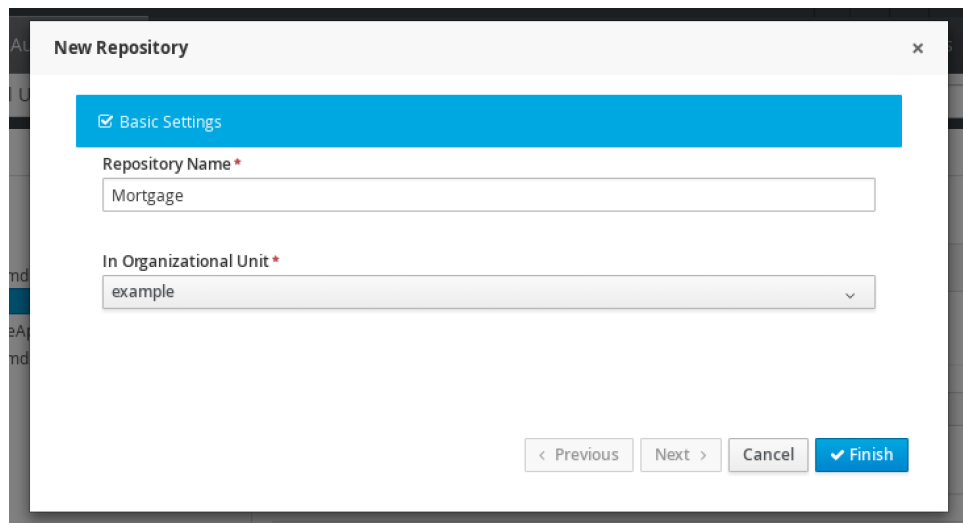


Figure 4. New Repository Information

After creating the *Mortgage* repository, the *Administration* view will display both the pre-existing and newly created repositories and the URL to access each through Git.

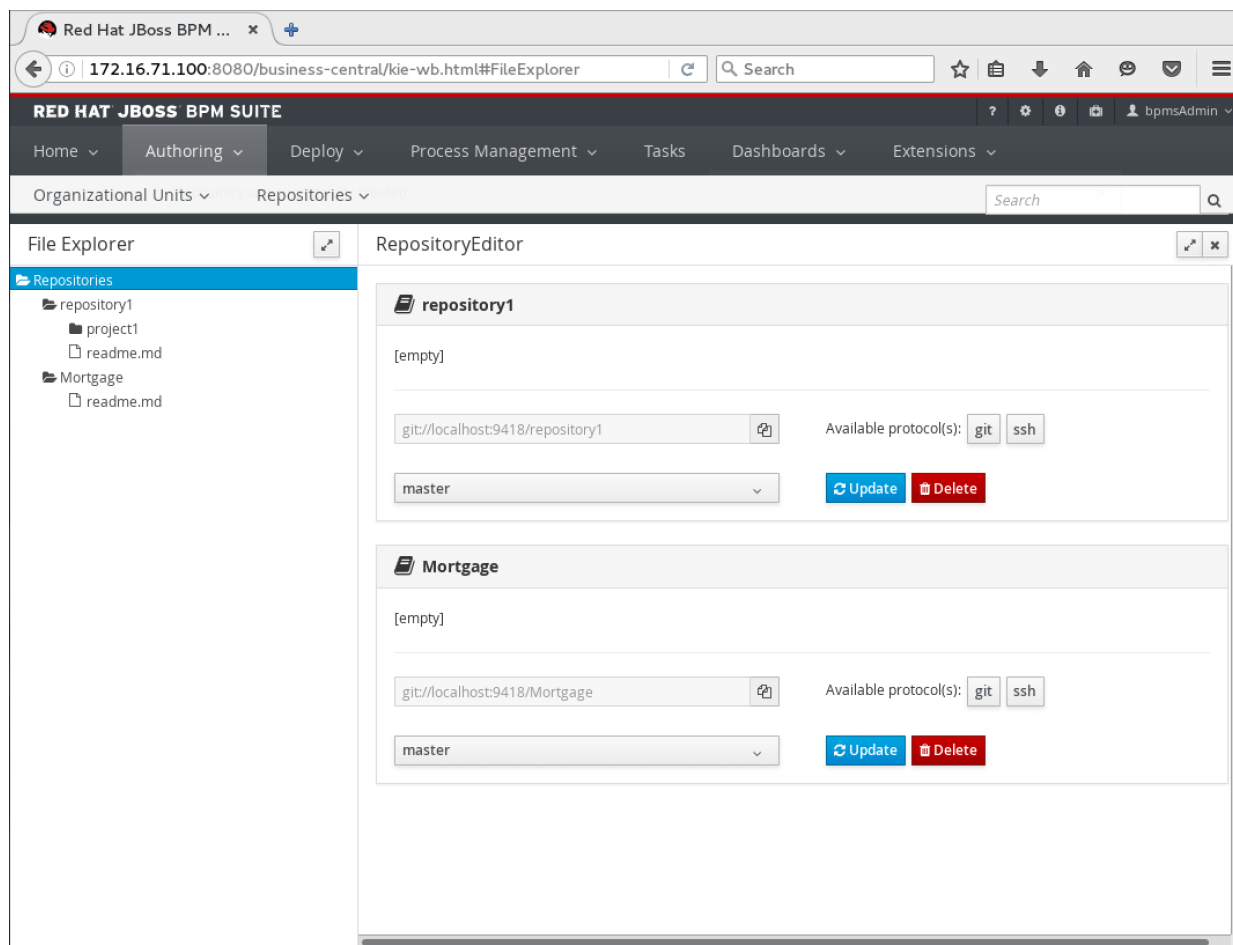


Figure 5. Administration Repository List

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, then switch the current repository to *Mortgage* in the **Project Explorer** breadcrumb trail as seen below:

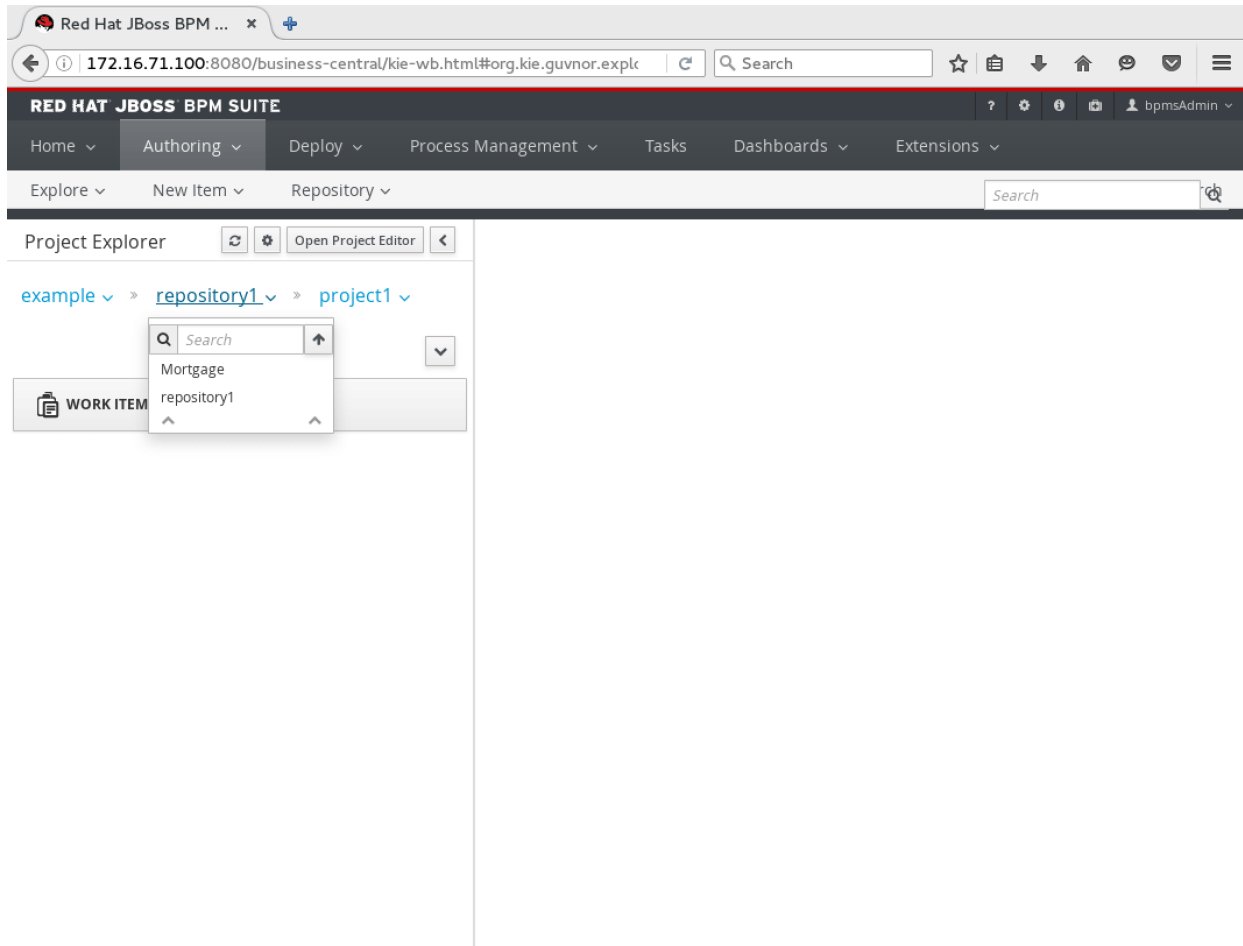


Figure 6. Switching Projects inside Project Authoring

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

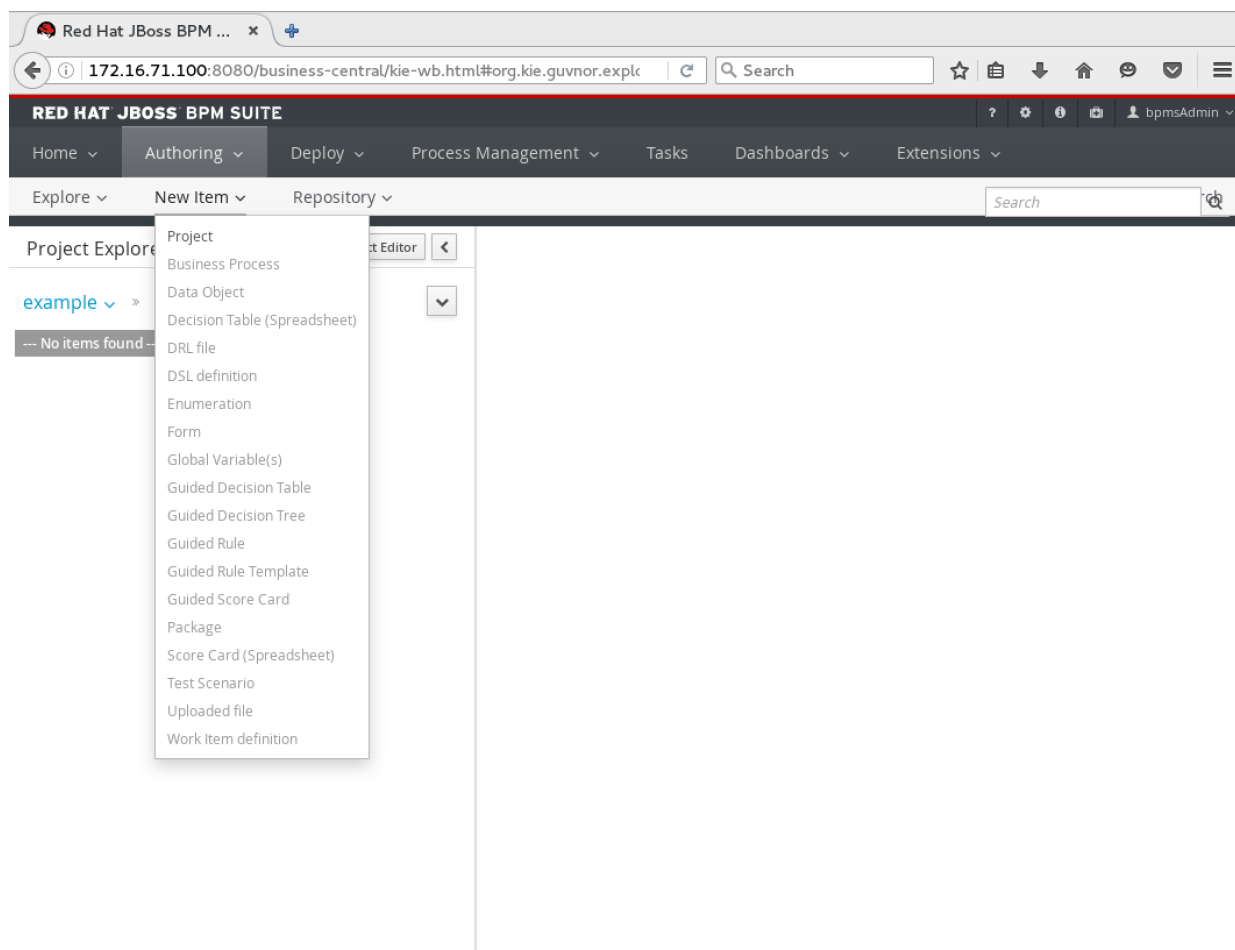


Figure 7. Creating a New Project

Create a project called *MortgageApplication* with the information provided below:

New Project
✕

New Project Wizard

Project General Settings

Project Name

Project Description

Group artifact version

Group ID
Example: com.myorganization.myprojects

Artifact ID
Example: MyProject

Version
Example: 1.0.0

Figure 8. Project Settings

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. Under *New Item*, select *Data Object* to create necessary POJO definitions. For example, this project requires a persistable 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:

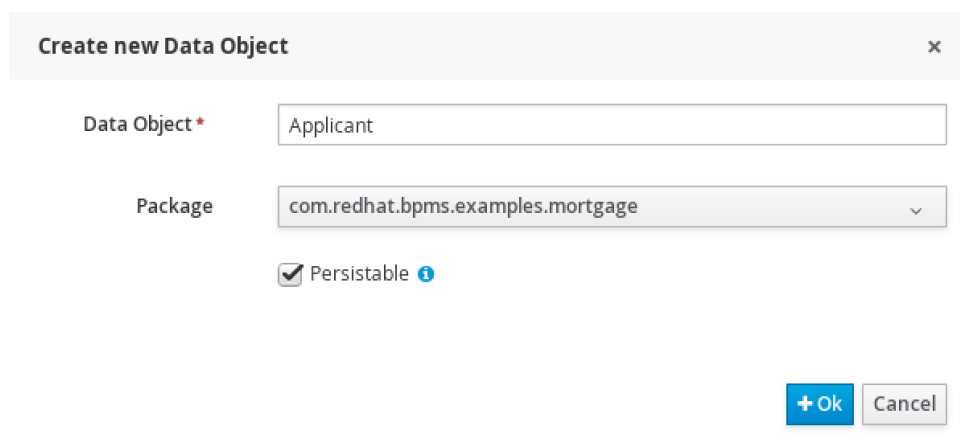


Figure 9. Applicant Object Creation

Once a type has been created, proceed to define its fields alongside the automatically generated *id* field (for persistence) by means of the *+add field* button:

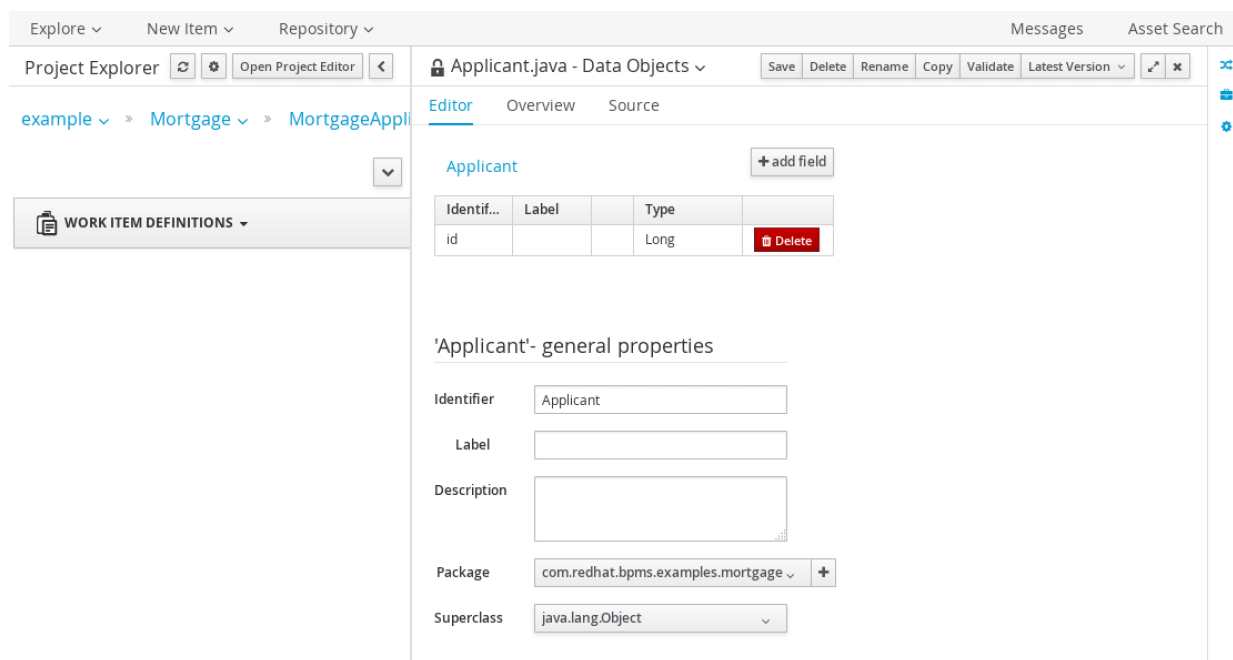
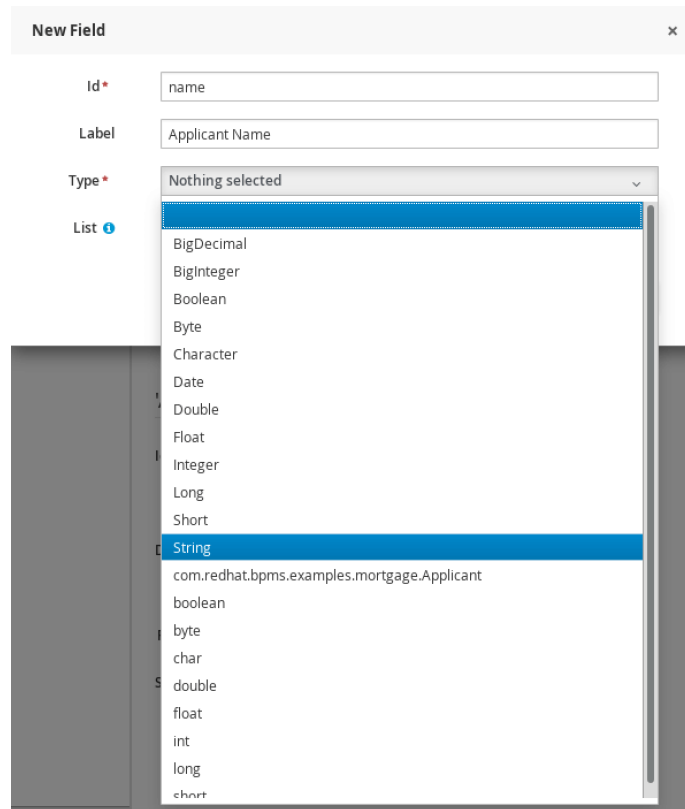


Figure 10. Applicant Object Properties

Each field consists of 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:



The image shows a 'New Field' dialog box with the following fields and values:

- Id***: name
- Label**: Applicant Name
- Type***: Nothing selected
- List**: A scrollable list of data types with 'String' selected. The list includes: BigDecimal, BigInteger, Boolean, Byte, Character, Date, Double, Float, Integer, Long, Short, String, com.redhat.bpms.examples.mortgage.Applicant, boolean, byte, char, double, float, int, long, short.

Figure 11. Applicant Name Field Creation

Using the steps above, create additional fields to complete the Applicant data model as well as further required object types, as listed below. Upon completion of each object, be sure to Save, providing a relevant commit message for each:

Table 1. Applicant

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

Table 2. Property

Id	Label	Type
address	Property Address	String
price	Sale Price	Integer

Table 3. ValidationError

Id	Label	Type
cause	Cause of Error	String

Table 4. Appraisal

Id	Label	Type
property	Appraised Property	com.redhat.bpms.examples.mortgage.Property
date	Appraisal Date	Date
value	Appraised Value	Integer

The final data object, *Application* consists of several of the above-defined types, as well as a List of validation errors, which is created using the *List* checkbox on the field creation modal:

Table 5. Application

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

Overall, the complete 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 the same data model:

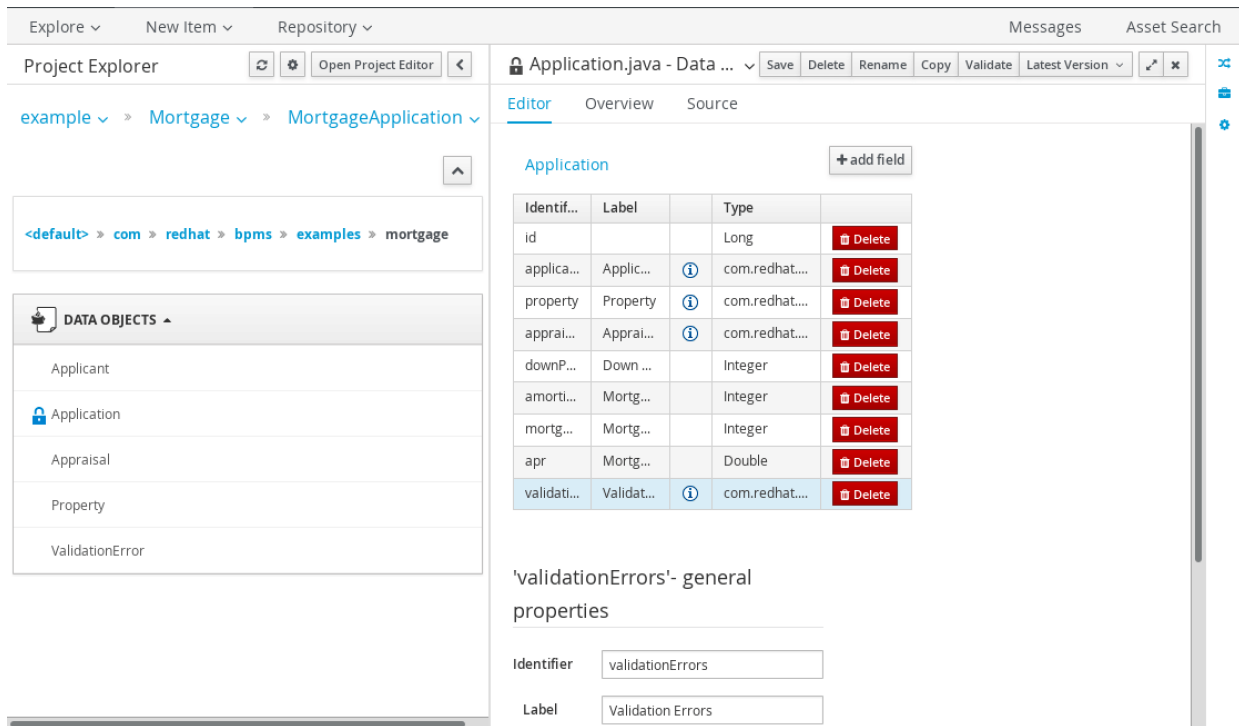


Figure 12. Completed Data Model

5.4. Business Process

5.4.1. Create New Process

Within the *New Item* menu previously used to start creation of *Data Object*, select *Business Process* to initialize a new process within the *com.redhat.bpms.examples.mortgage* package:

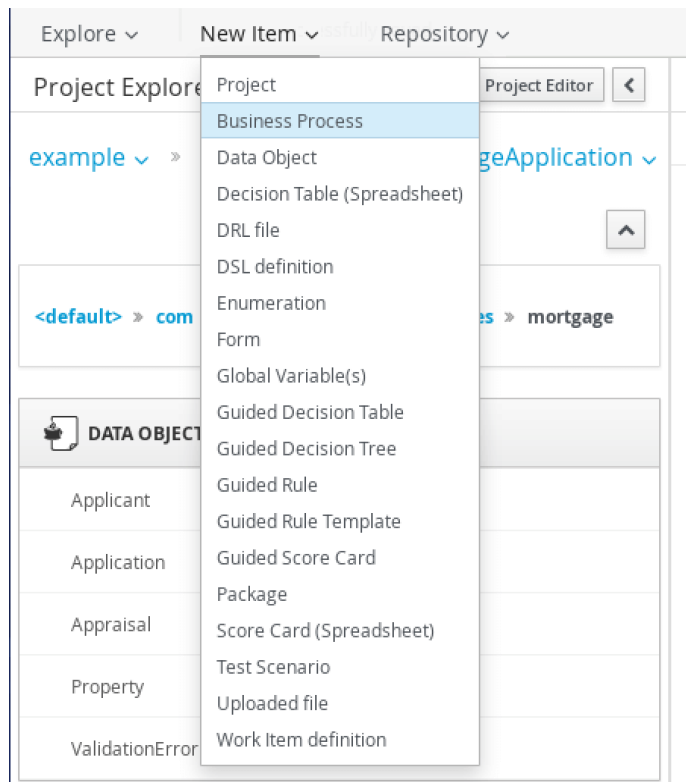


Figure 13. Create New Business Process

Create new Business Process
✕

Business Process *

Package

Figure 14. New Process Details

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

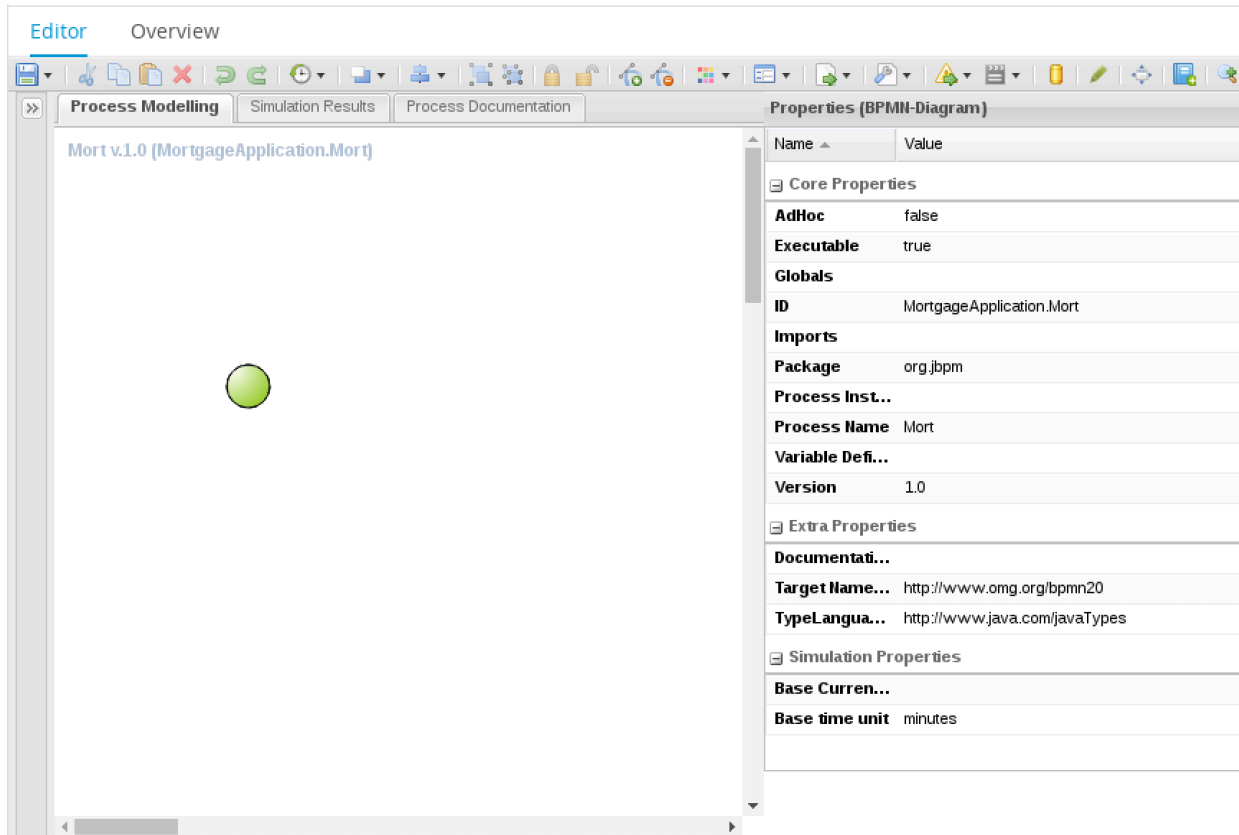


Figure 15. New Process Canvas

Going forward, while focusing on the canvas design, you may find it helpful to rearrange some of the tooling to allow more canvas space. You can click the '<' button next to the *Open Project Editor* button to collapse the Project Explorer window, then note the '>>' button on the left-hand side of the canvas, which will be used to access various predefined object types:

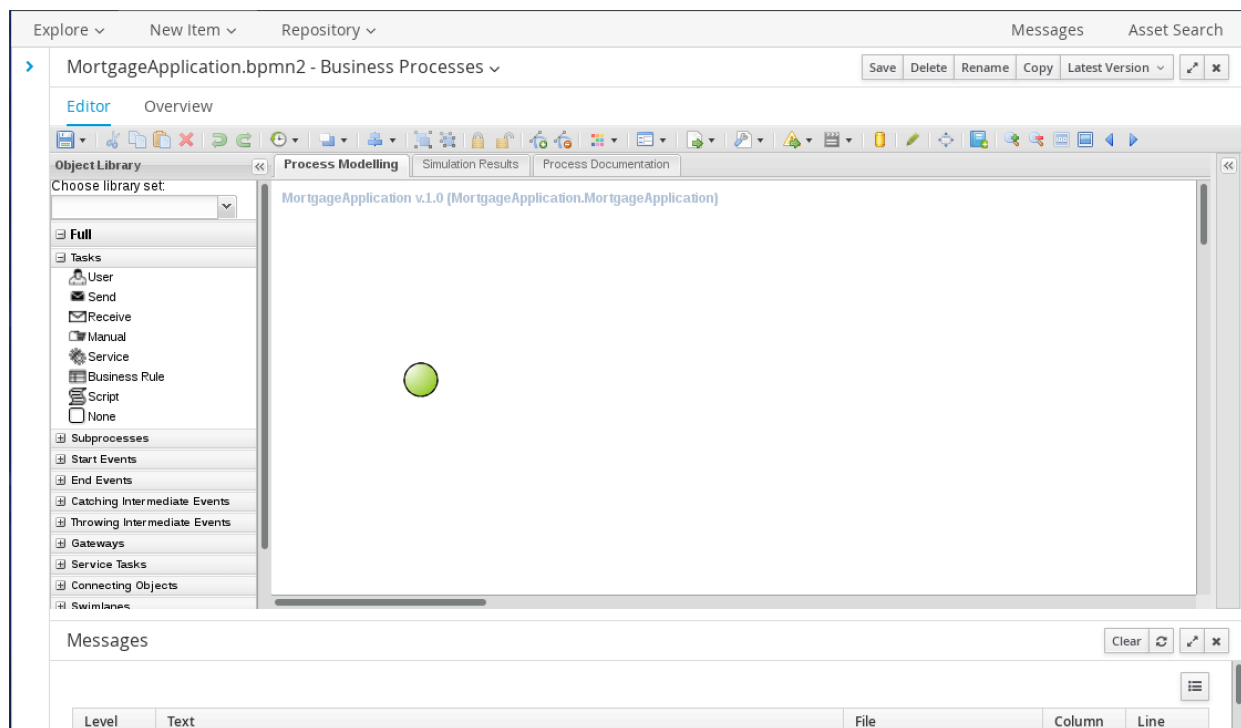


Figure 16. Expanded Canvas View

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 (available in the toolbar under *Tasks*) after the start node and complete the process by connecting this script node to a *None* end node (available under *End Events*).

Create a process variable representing a mortgage application by clicking on the white canvas background and accessing the expandable properties pane on the right-hand side, seen as *Variable Definitions*. The *Application* class, created in the *com.redhat.bpms.examples.mortgage* package in the data modeler serves this purpose. Call this process variable *application*.

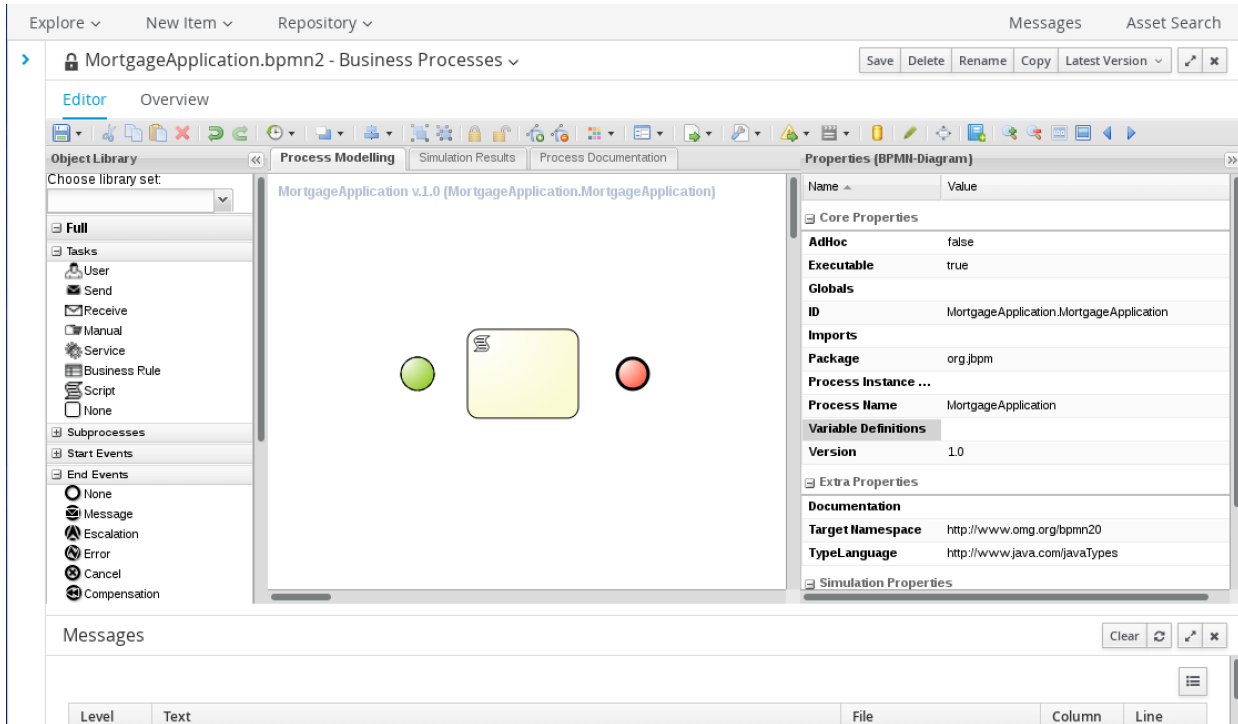


Figure 17. Accessing Process Properties

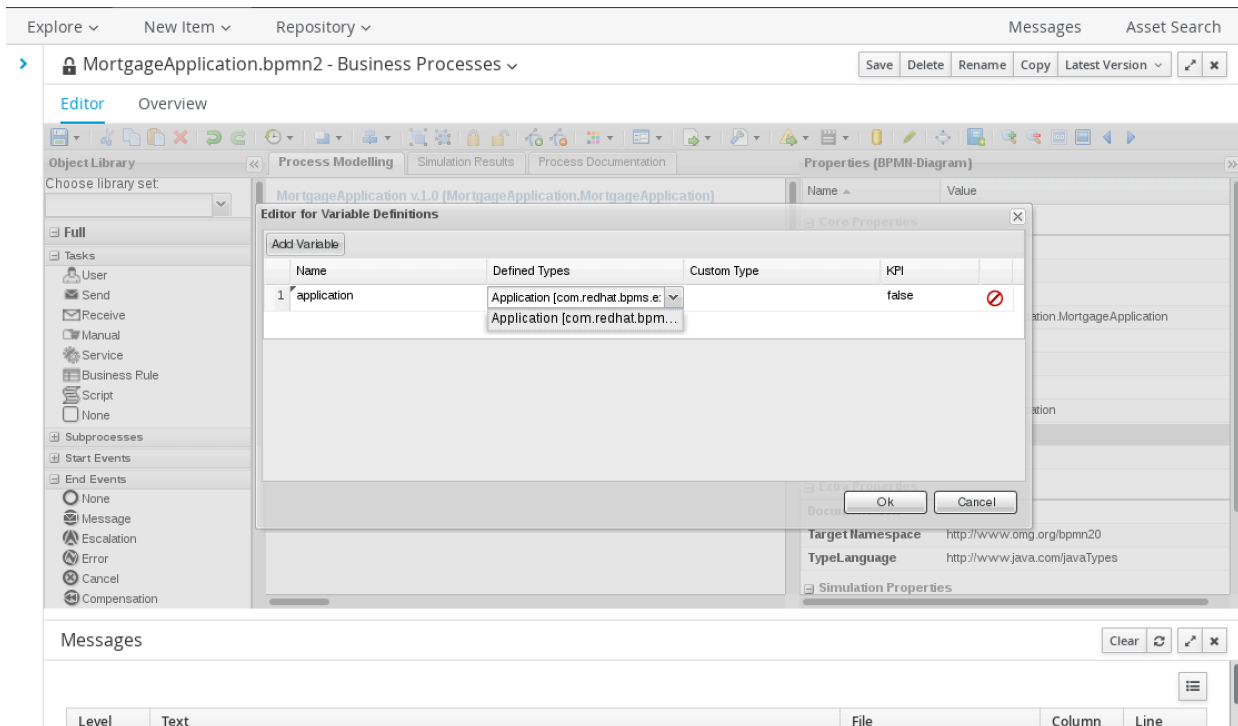


Figure 18. Process Variables

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 add the full package name as a prefix to the process ID:

Properties (BPMN-Diagram)	
Name ▲	Value
Core Properties	
AdHoc	false
Executable	true
Globals	
ID	com.redhat.bpms.examples.mortgage.MortgageApplica
Imports	
Package	com.redhat.bpms.examples.mortgage
Process Instance...	
Process Name	MortgageApplication
Variable Definitions	application:com.redhat.bpms.examples.mortgage.Appli
Version	1.0
Extra Properties	
Documentation	
Target Namespace	http://www.omg.org/bpmn20
TypeLanguage	http://www.java.com/javaTypes
Simulation Properties	

Figure 19. Process Properties

Utilize the *Save* button in the upper toolbar to save the process and provide a meaningful explanation of the changes thus far:

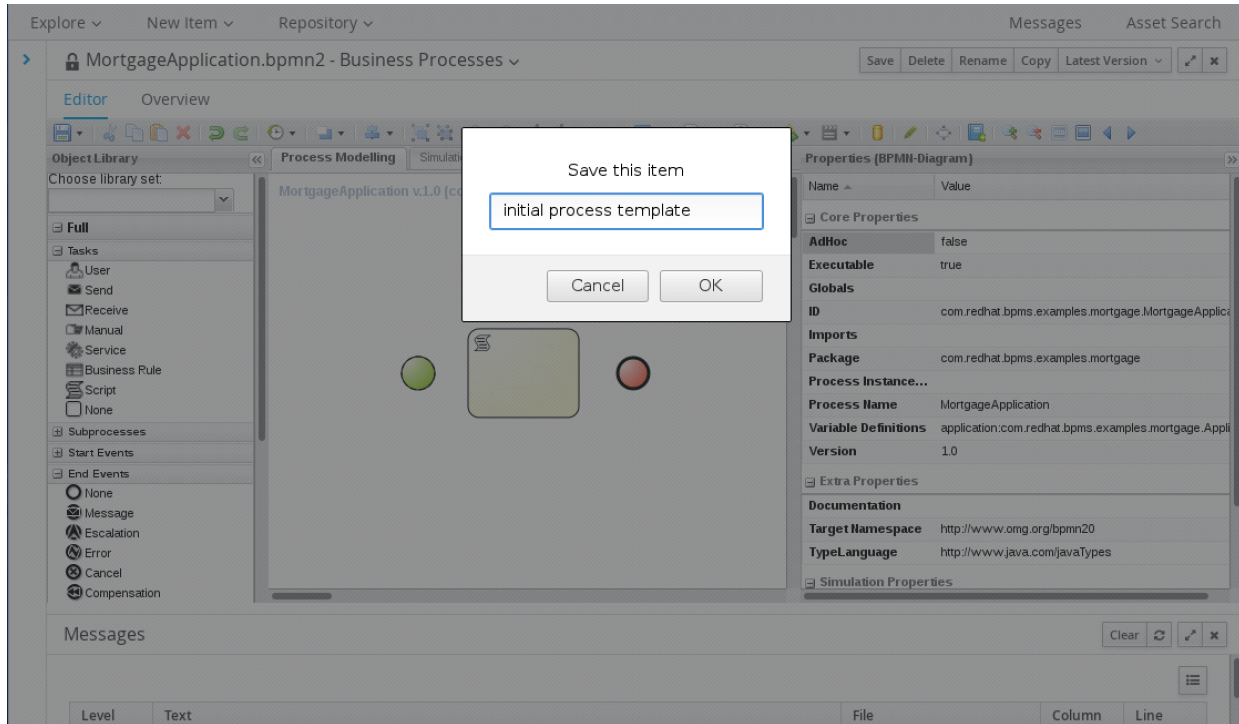


Figure 20. Process Save/Commit

5.4.2. Initial 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. While remote callers can utilize a REST API to start a new instance of the *MortgageApplication* business process, users can also provide the same information through a Business Central form, which we'll create now.

The process uses the custom *Application* object type, which itself embeds the other custom types created in the data modeler. The process form, therefore, should also embed two subforms called *MortgageApplicant* and *MortgageProperty* to collect information for the *Application*'s subtypes. 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.

Applicant Subform

We'll start the process by first creating the two subforms for Applicant and Property that will go into the Application form. From the *New Item* menu, select *Form* and provide the name *MortgageApplicant*:

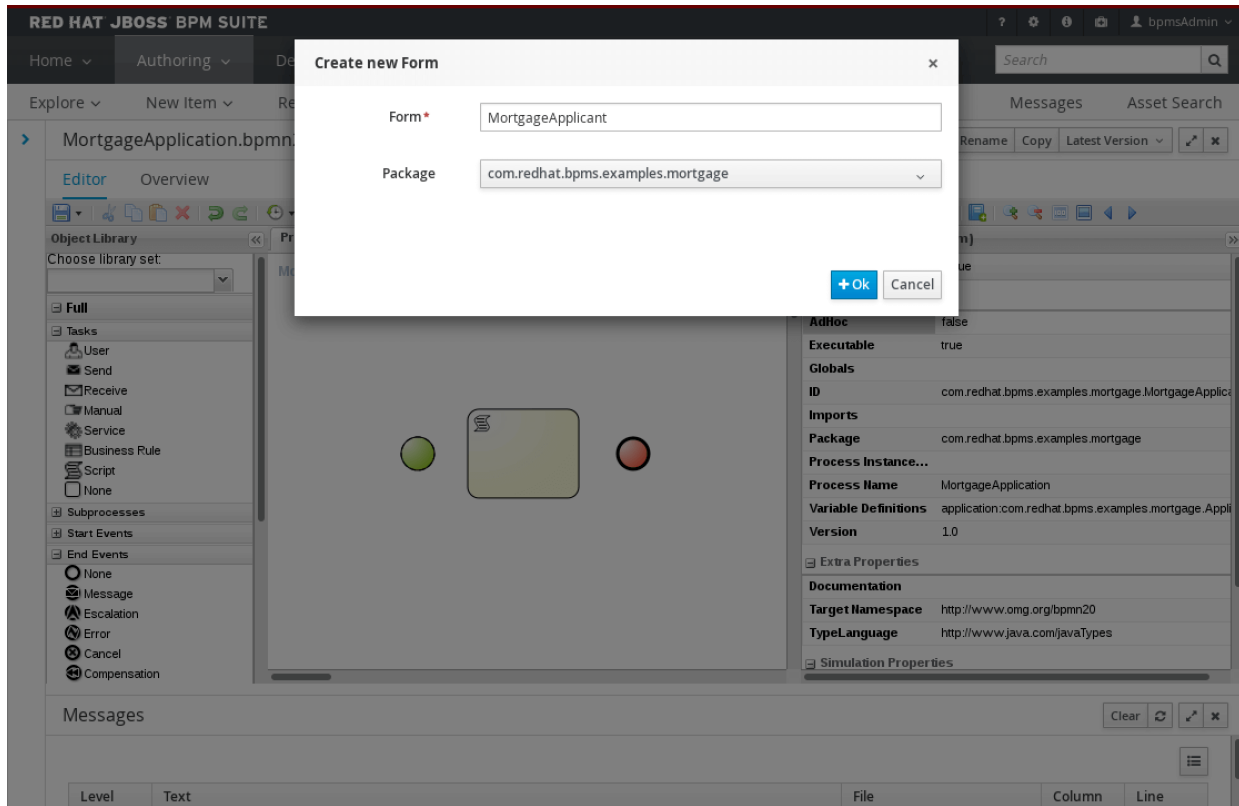


Figure 21. Create Form

Create a new form called *MortgageApplicant*. Add a data origin item called *applicant*, with Input Id and Output Id of *applicant* as well. These fields indicate what variable should be used to derive existing values when the form is opened and where to store the user-provided input upon submitting the form.

Select the render *Type* by choosing *From Data Model*, *com.redhat.bpms.examples.mortgage.Applicant* and pick a render color.

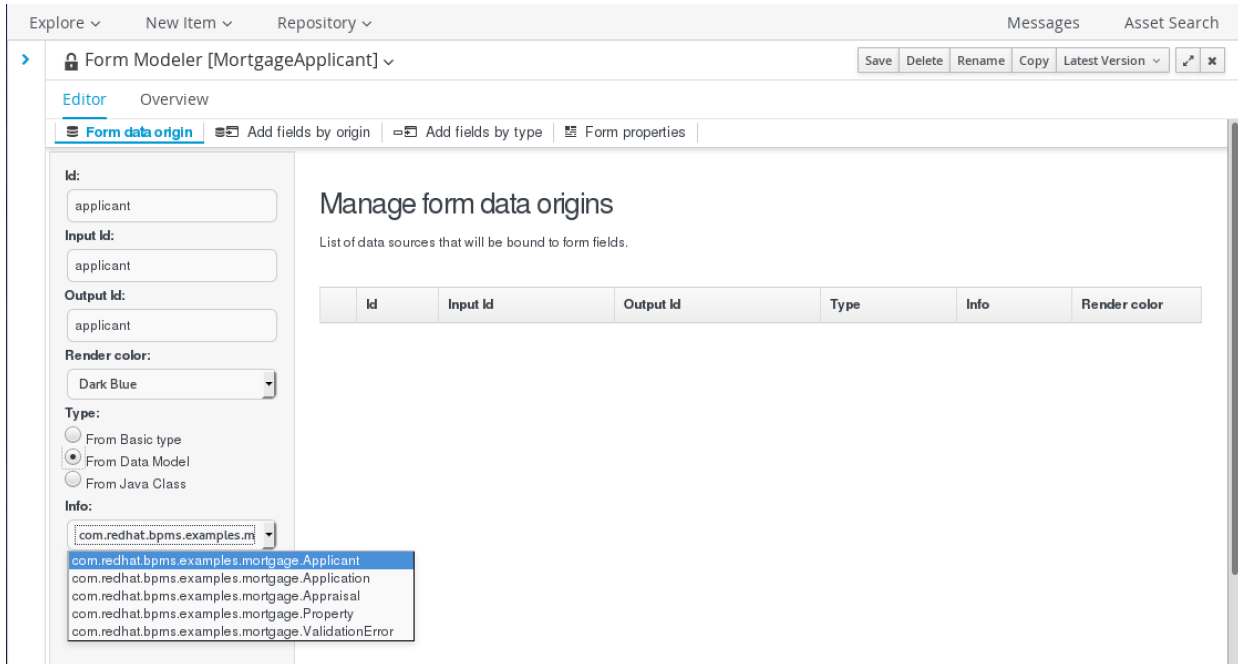


Figure 22. Creating the Applicant Subform

After adding the data holder, go to *Add fields by origin* in the form editor toolbar and expand the applicant node in the left-hand tree view. Click the white arrows next to name, ssn, and income fields to add them to the form. While hovering over the fields now present on the form, access the pencil icon to edit the properties for each input field and provide meaningful labels for each:

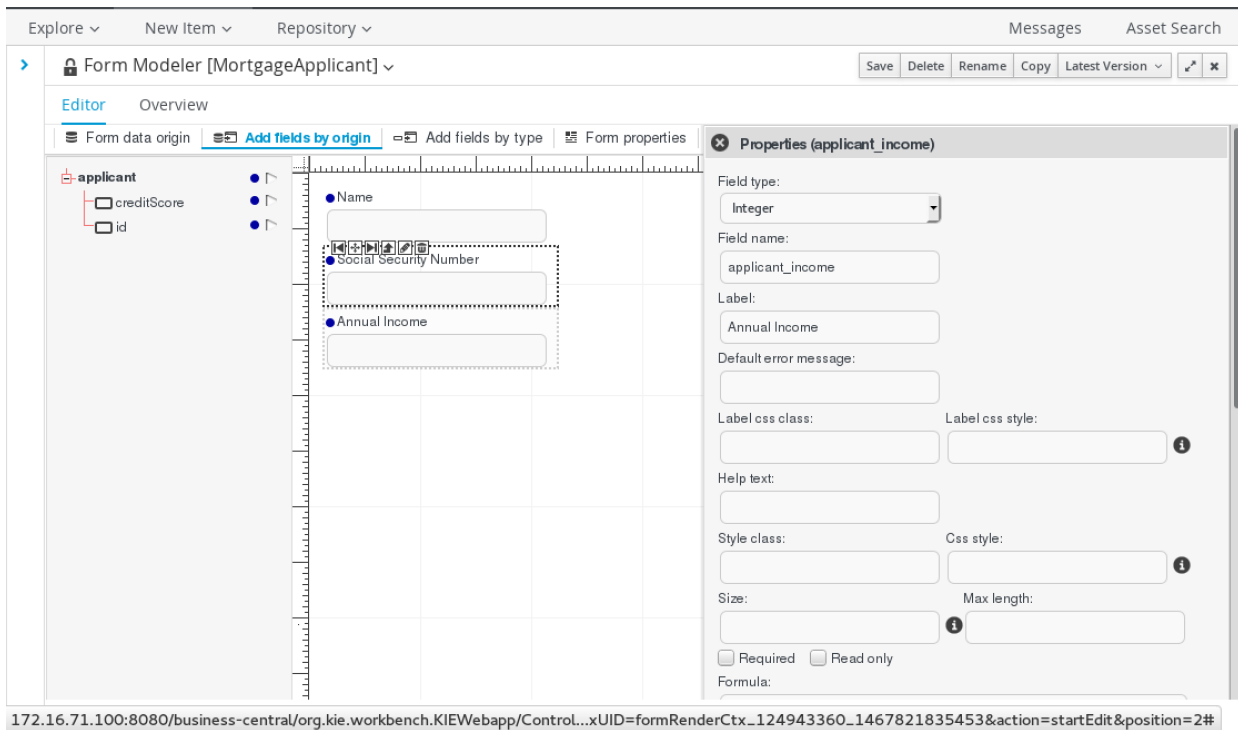


Figure 23. Applicant Subform Field Labels

Once complete, be sure to Save and provide a commit message for the new applicant subform.

Property Subform

Follow the same steps to create a new *MortgageProperty* subform for the *Property* object, using both the address and price fields.

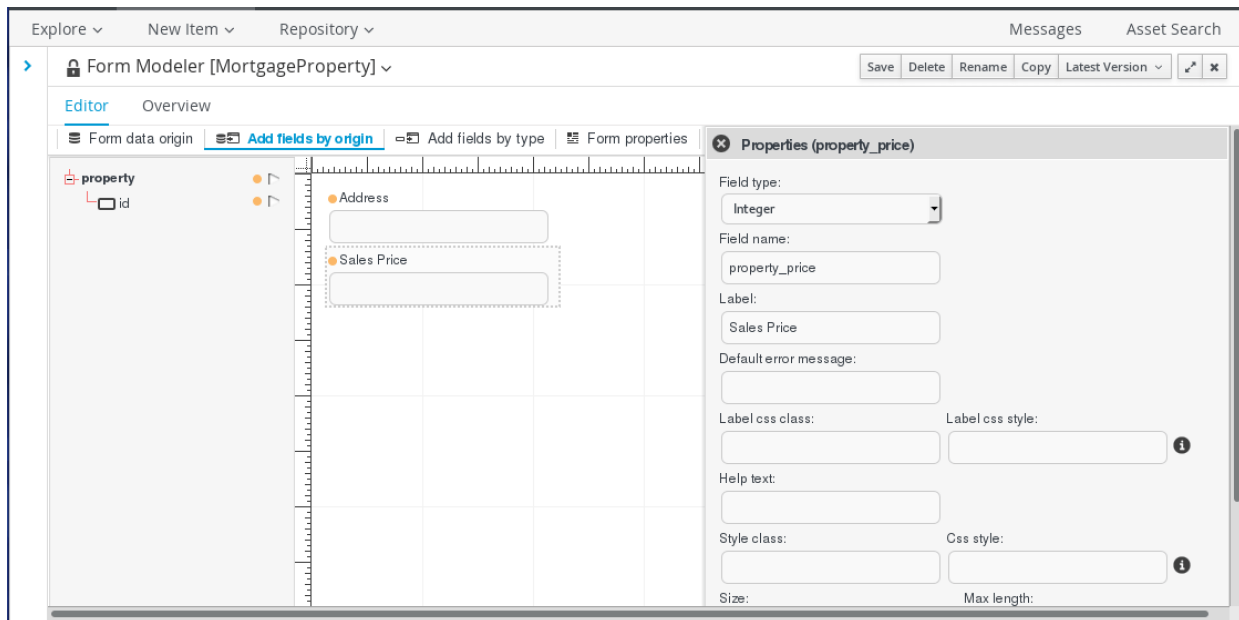


Figure 24. Property Subform

Following completion, be sure to save the form, then expand/use the project explorer pane on the far left-hand side to return to the MortgageApplication business process.

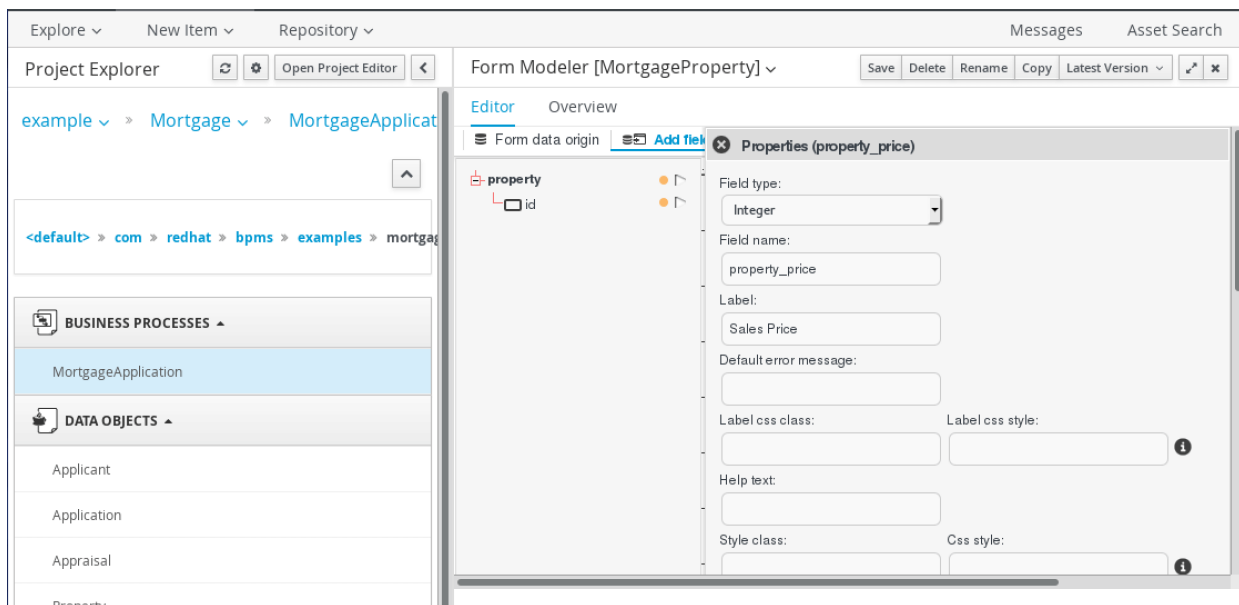


Figure 25. Using Project Explorer

Finishing the Initial Process Form

With the needed subforms now complete, the process form itself can be finished. When a process instance is kicked off in Business Central, the expected process form name, which will be shown to the user for collecting initial input, is derived from the process ID. The easiest and least error-prone approach to creating a process form when the correct name is to open the process and select *Edit Process Form* from the top menu, then select *Graphical Modeler*:

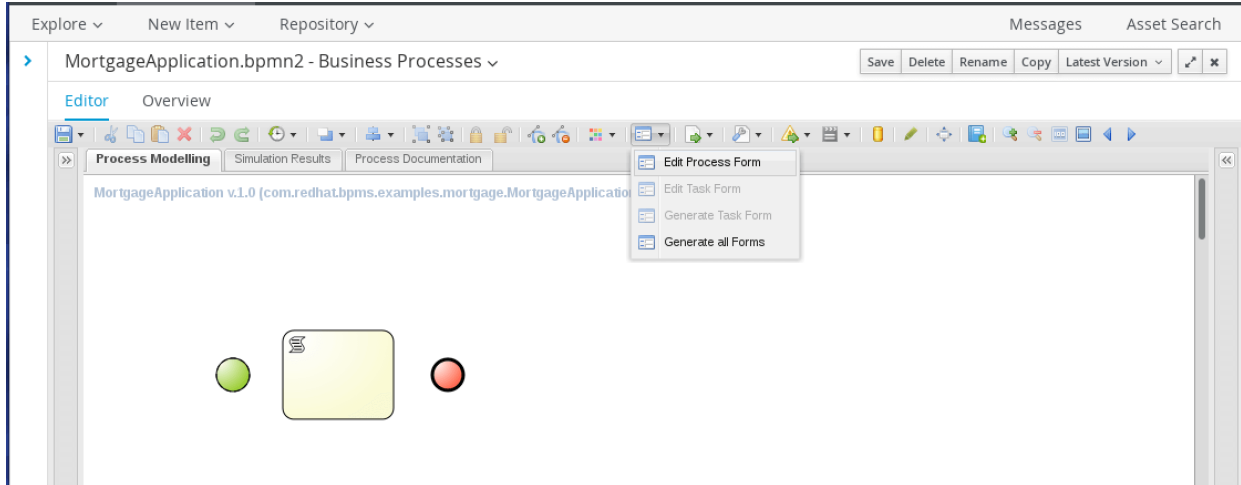


Figure 26. Edit Process Form

As before, add application as a data holder, which will be the only object needed for this form:

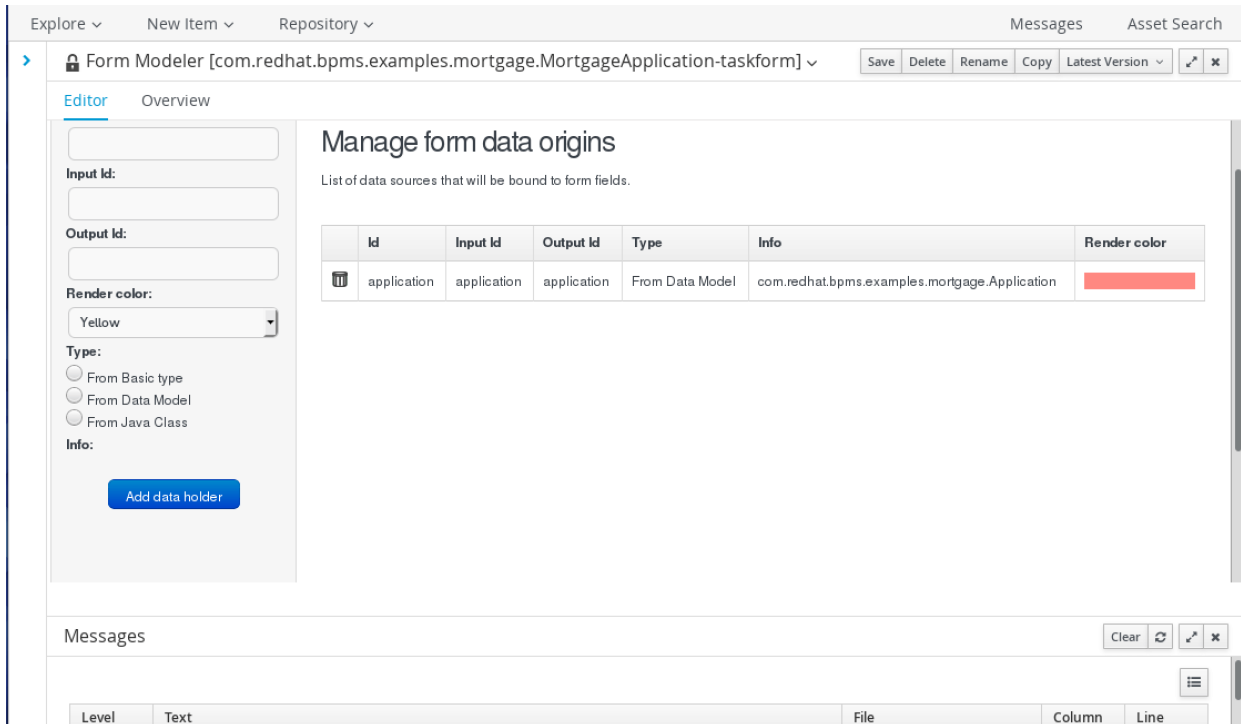


Figure 27. Application Form Data Holder

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

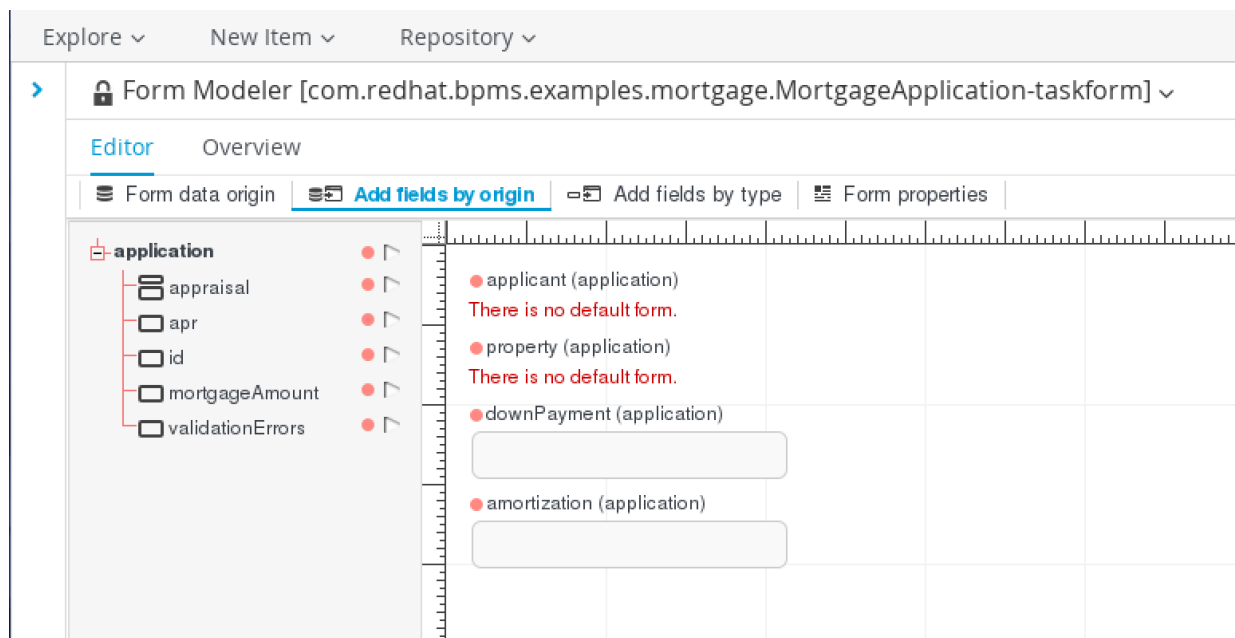


Figure 28. Application Fields by Origin

Those fields of application that are not of basic and primitive type (applicant and property) 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 properties via the pencil icon on hover. Set its label to the more user friendly value of Mortgage Applicant. The field type should be simple subform. Select the previously created Applicant subform as the default form for this field, which should then be reflected on the graphical editor:

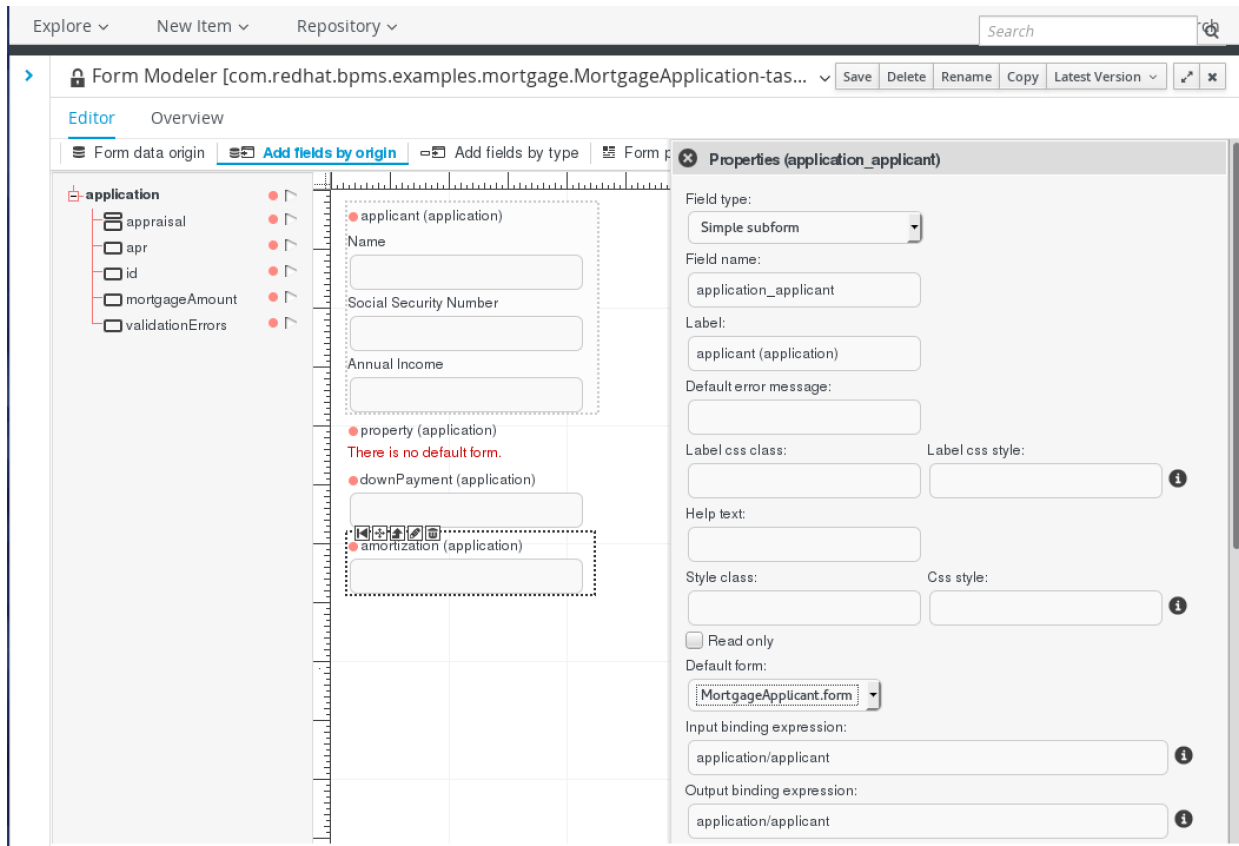


Figure 29. Applicant Form on Application

Follow similar steps to use the Property subform for the property field and set proper labels for down payment and amortization, making sure to save the form when complete:

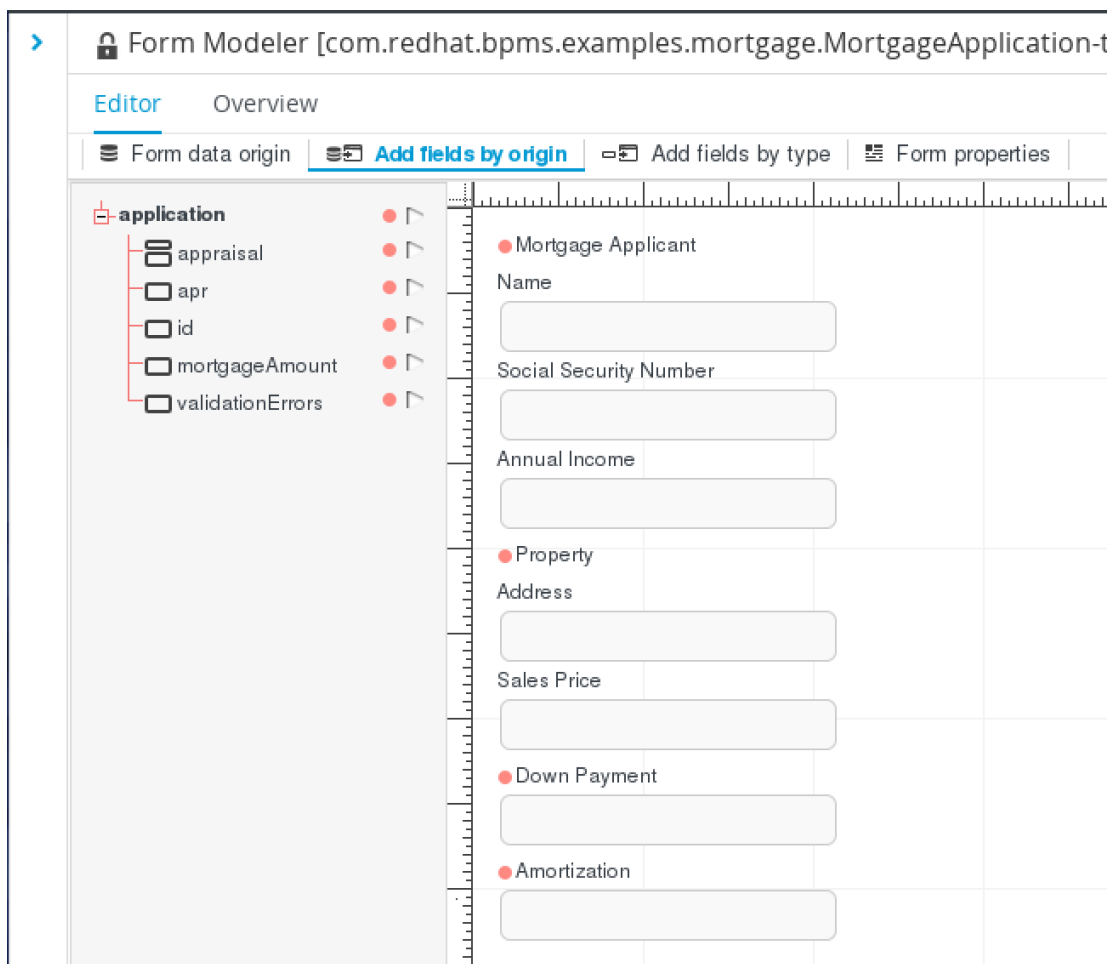


Figure 30. Application Form

Testing the Initial Process Form

Back in the MortgageApplication Business Process editor, use the properties panel of the script node to select the *Script* property, and inside the code editor, add a *println* statement for the application object:

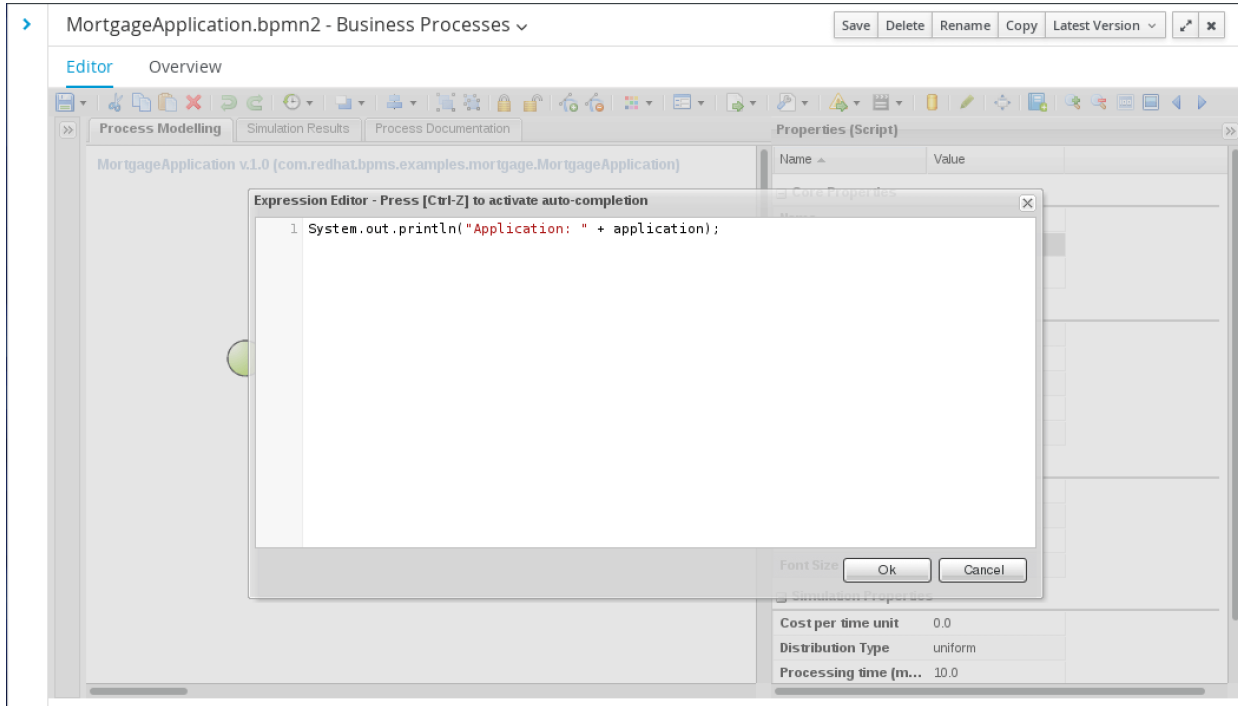


Figure 31. Editing the Script Node

Back on the editor canvas, click to select the green starting node, then click/drag the black arrow over to the script node and release, which will establish the process flow between the start and the script node. Likewise, create a connection between the script node and red end node to complete a simple, testable process.

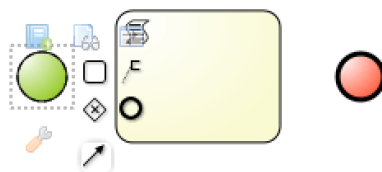


Figure 32. Canvas Element Properties



Figure 33. Process Flow Connected End-to-End

After saving the process, click the *Open Project Editor* button in the left-hand Project Explorer pane. In the toolbar atop the now open Project Editor, click the *Build* dropdown and select *Build & Deploy*. Once built successfully, go to Process Management in the top-most gray toolbar, then Process Definitions and find the process listed there. Click the *Start* button to kick off an instance of the process:

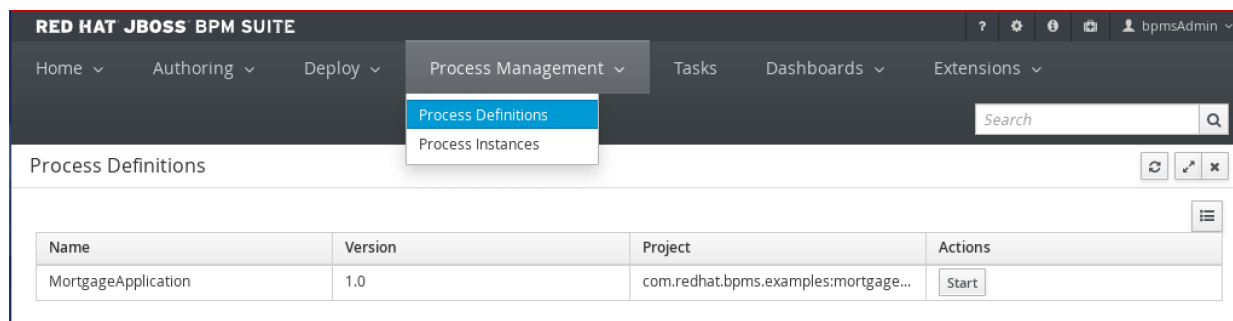


Figure 34. Start Process Instance

Once the process is initialized, the process Application form that was created earlier will be opened by Business Central and presented to the user to allow for initial values for the mortgage application:

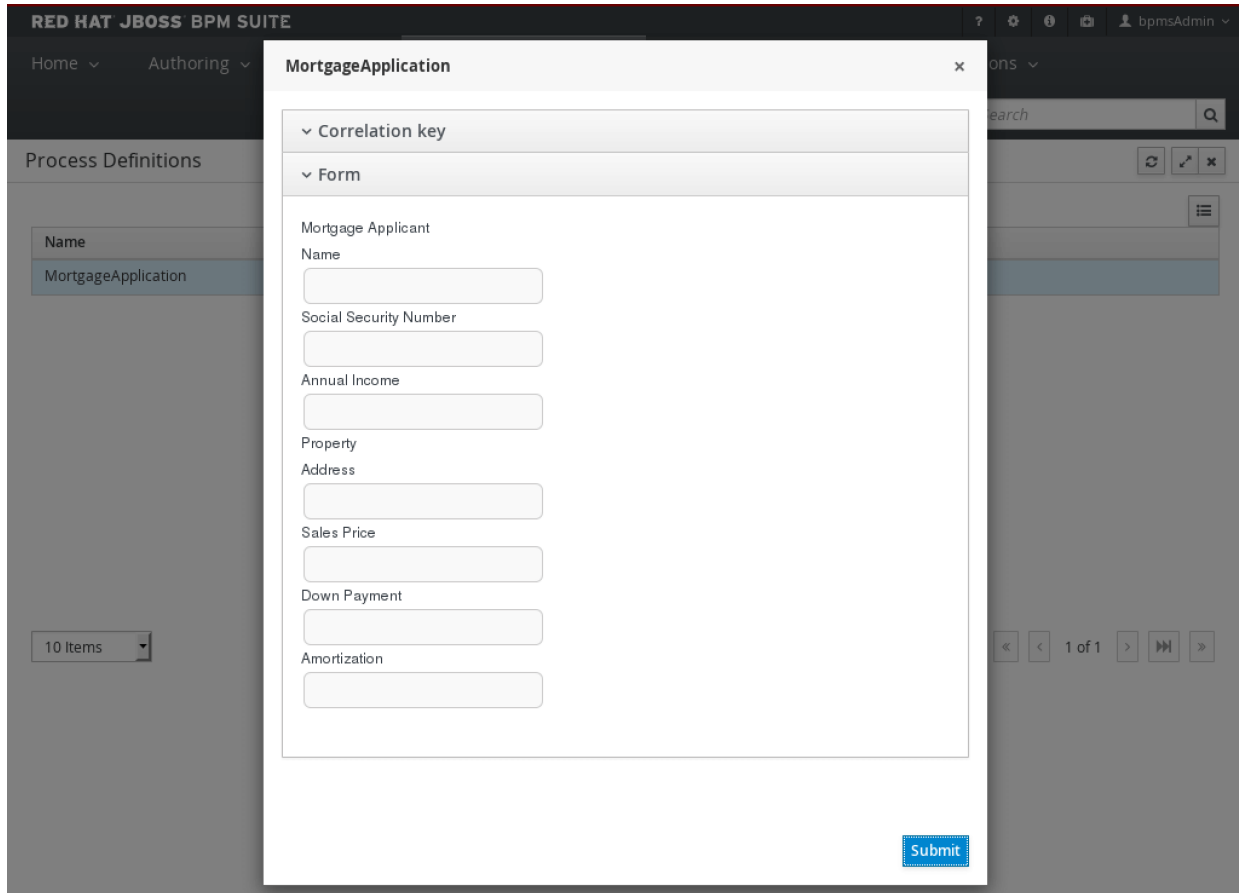


Figure 35. Mortgage Application Form

The 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:

```
13:43:06,394 INFO [stdout] (http-/0.0.0.0:8080-4) Application:
com.redhat.bpms.examples.mortgage.Application@665b07a0
```

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. However, it should be noted that any subsequent edits 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 the submitted application within the script node.

5.4.3. Validation

Validation Process Model

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 rules for validation an obvious choice.

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

To begin adding validation, return to Project Authoring & move the script node and end node to the far right of the canvas, break the flow from the start node and connect it with a new *Data-based Exclusive* (XOR for short) gateway instead:

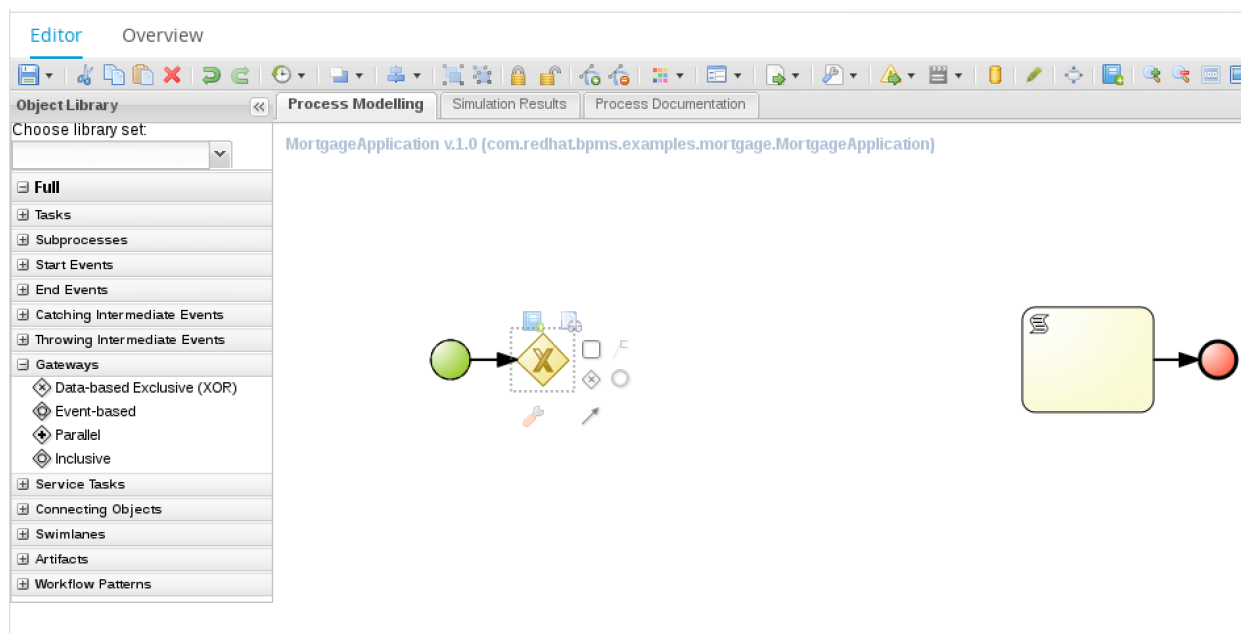


Figure 36. Adding XOR Gateway Node

Follow that up with a *Business Rule* task node and connect it with the XOR Gateway:

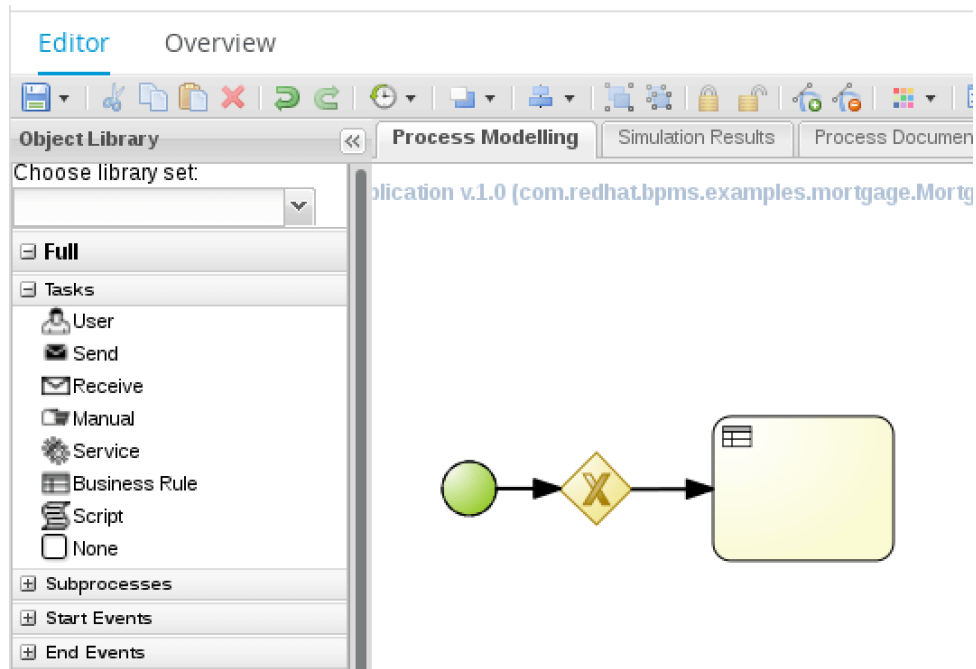


Figure 37. Business Rule Task Node

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.

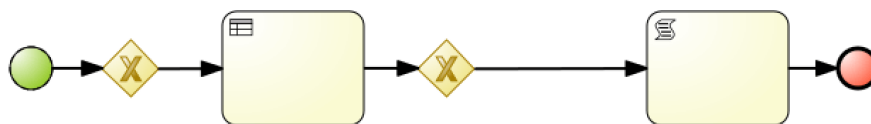


Figure 38. First Steps of Validation Flow

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 (we'll add the second shortly). 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*.

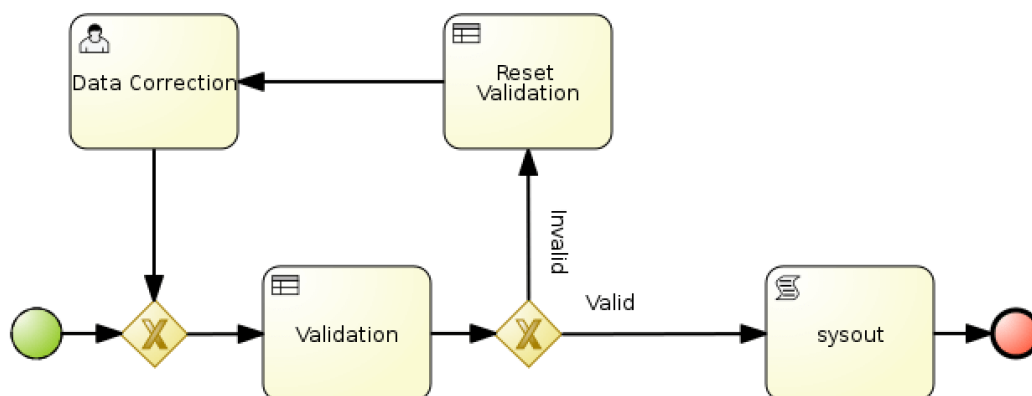


Figure 39. Validation Process Flow

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*:

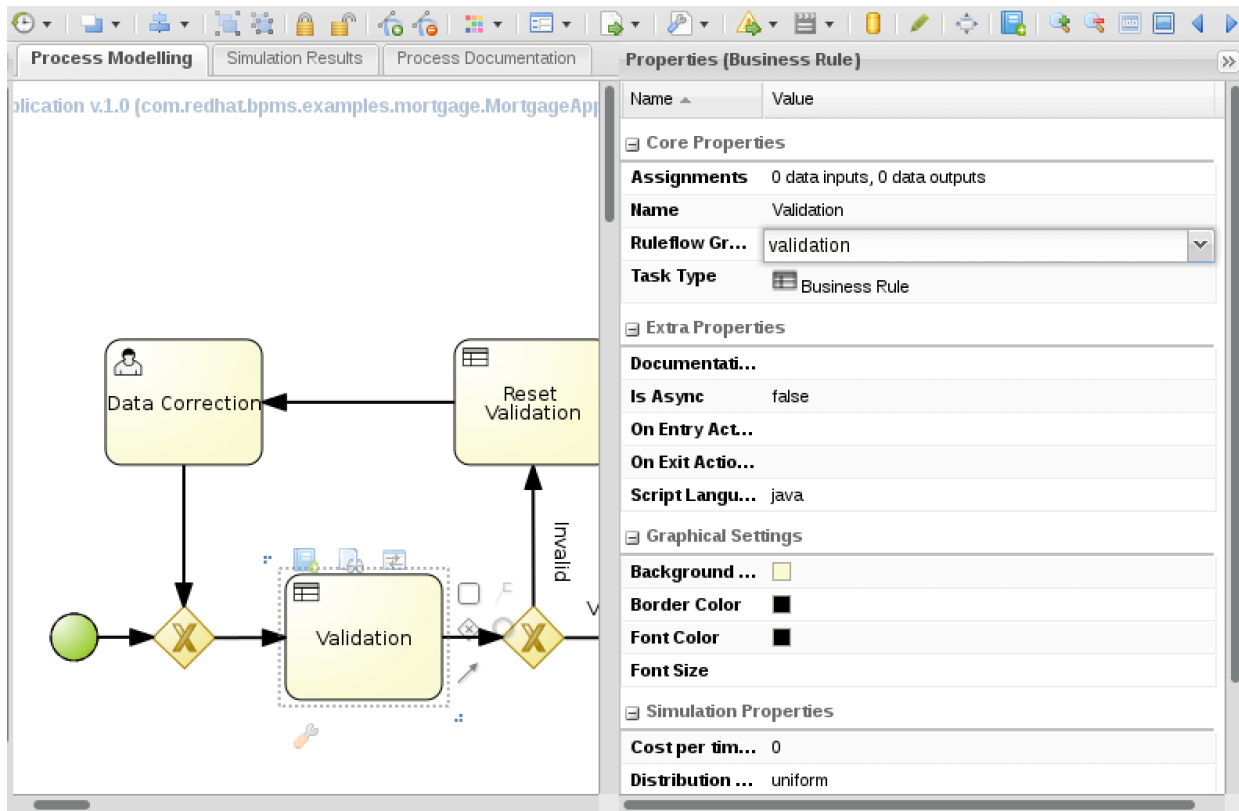


Figure 40. Validation Node Properties

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.

Validation Business Rules

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:

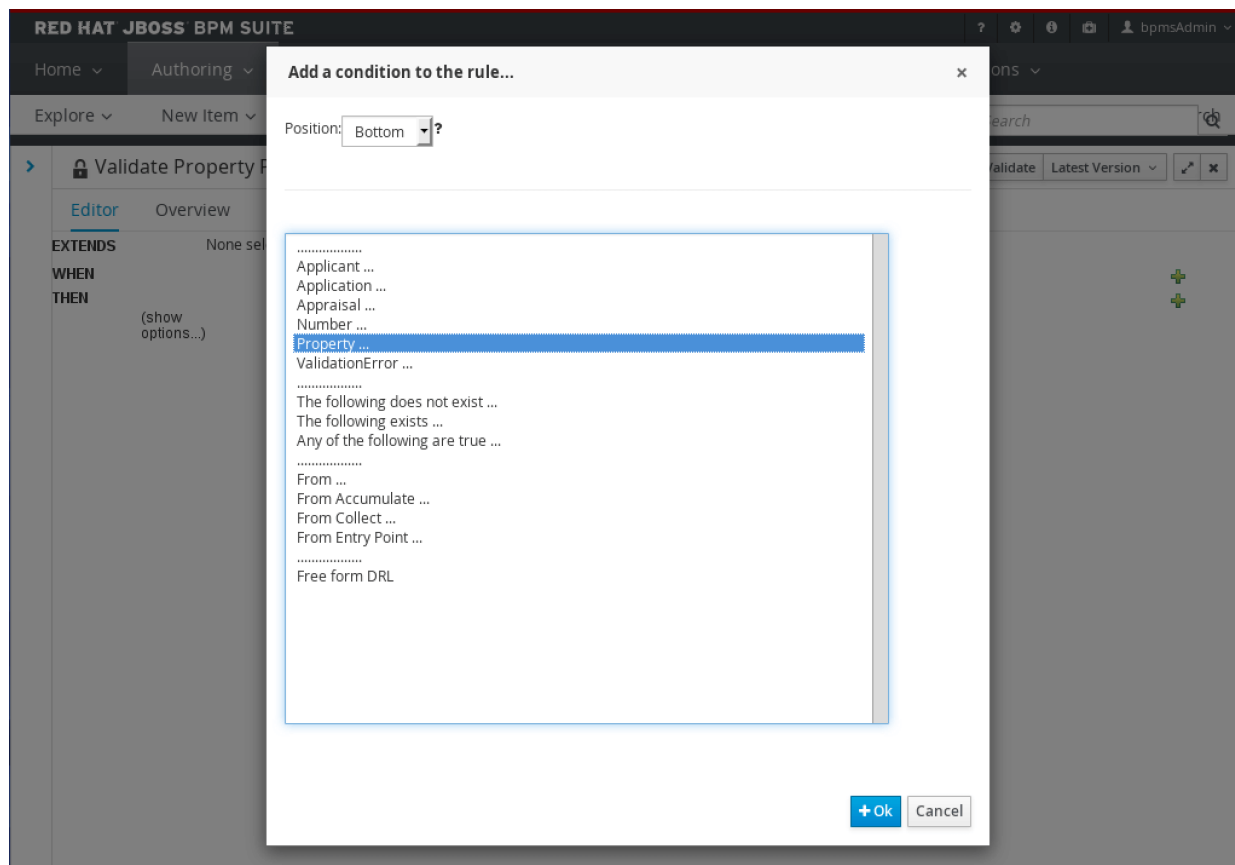


Figure 41. Adding Property to Guided Rule

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

There is a Property

To further refine the condition, hover and click on *There is a Property* to open a dialog and add a restriction on the price field:

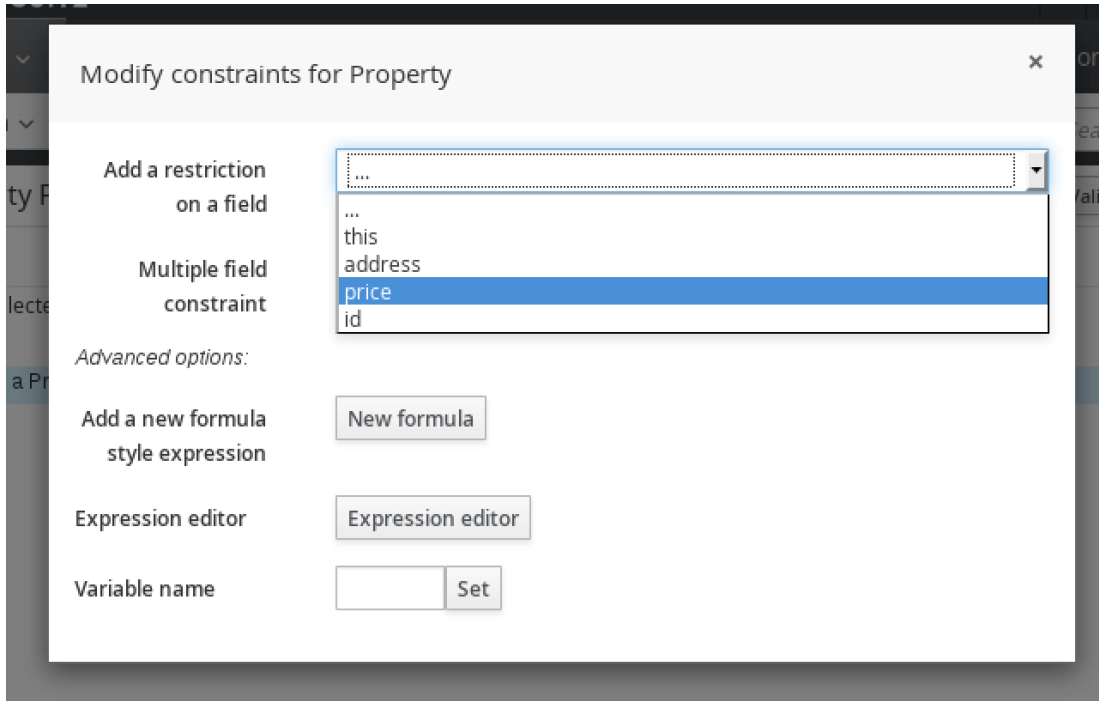


Figure 42. Adding Property Price

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.

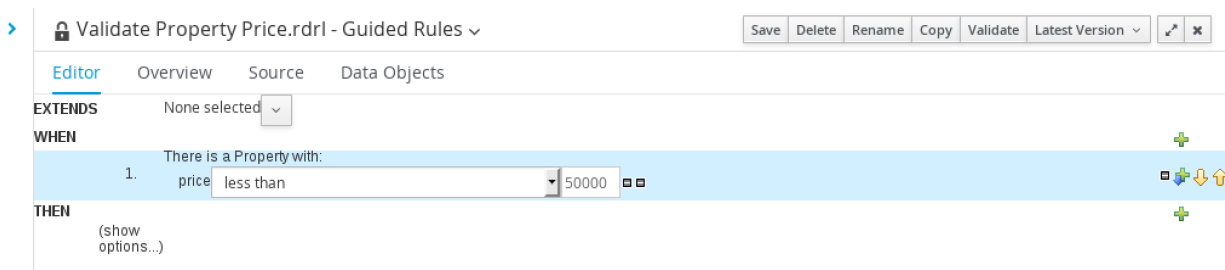


Figure 43. Property Price Low Constraint

Now click the plus icon to the far-right side of *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. Click the text *Insert ValidationError* and provide a literal value for the cause of "Property price too low":

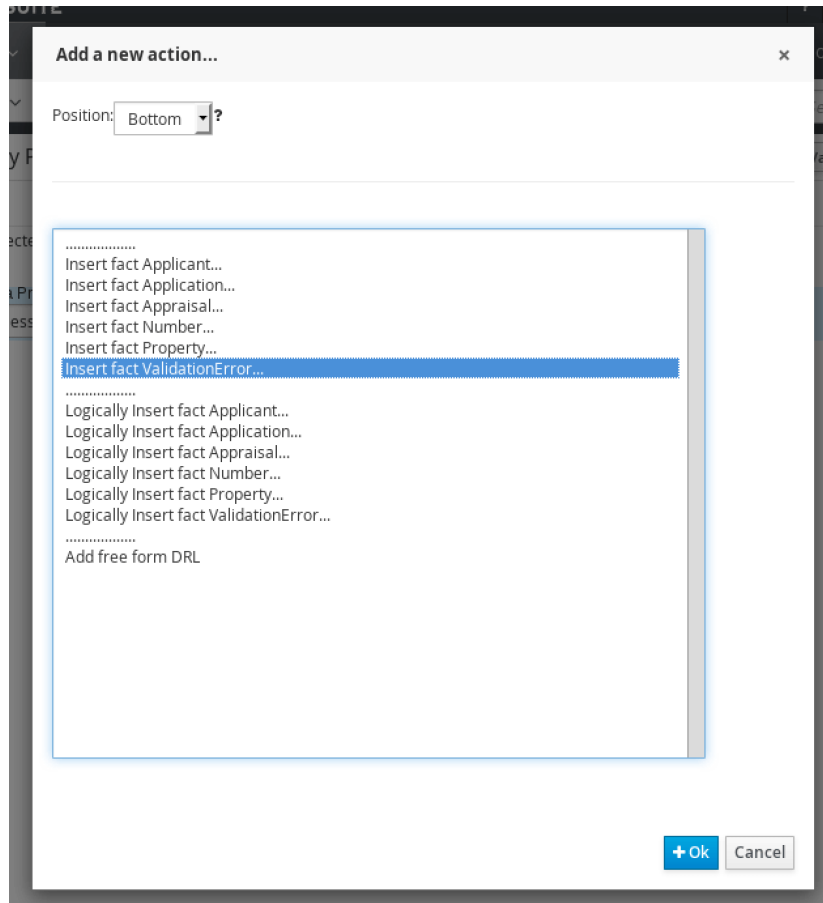


Figure 44. Adding ValidationError

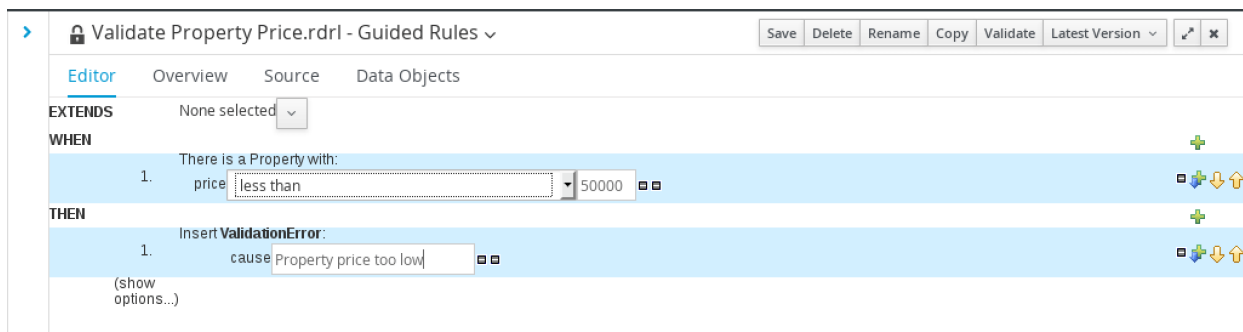


Figure 45. Literal Value for Error Cause

Next, click on *show options* and then click the corresponding plus icon to the right of *Attributes* to create a new option. Choose the ruleflow-group attribute from the drop-down and set its value to *validation*, thereby linking the new rule to the rule group triggered by the validation task node:

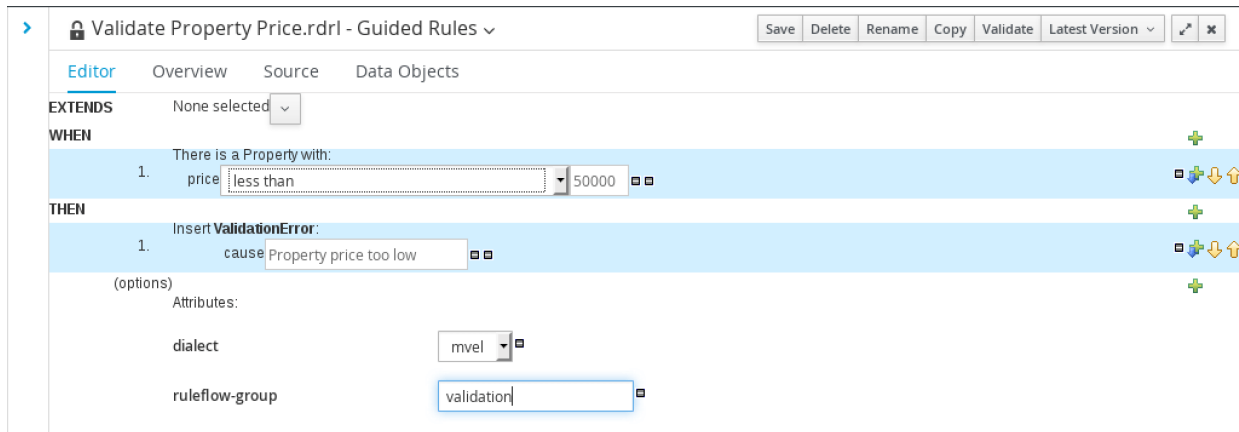


Figure 46. Linking Ruleflow Group

Save this rule and enter a meaningful description for the purpose of the repository revision. The ruleflow-group of this rule ties it to the business rule task node we added to the process earlier and causes the process engine to evaluate this rule when that node is reached. If the condition of the rule is 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 then 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 *Multiple field constraint* of type *Any of (Or)*. Now click on the *any of the following* sentence and add a restriction on its *downPayment* field. Constrain any down payment that is less than the literal value of 0, then click the *any of the following* text again to add another constraint. Select the option *or greater than* and use the Expression editor to select *property.price*. 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, click *show options* and add the *ruleflow-group* attribute of *validation* before saving and committing the rule to the repository.

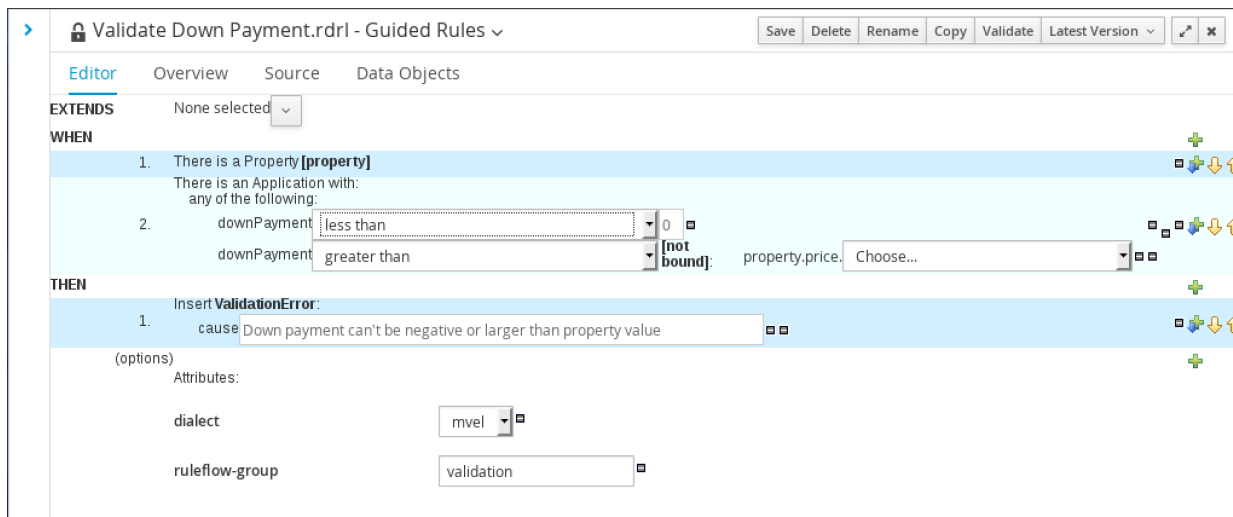


Figure 47. Down Payment Validation Rule

Next, 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 such as *Income too low*. Remember to add a *ruleflow-group* attribute to the rule and set it to *validation*.

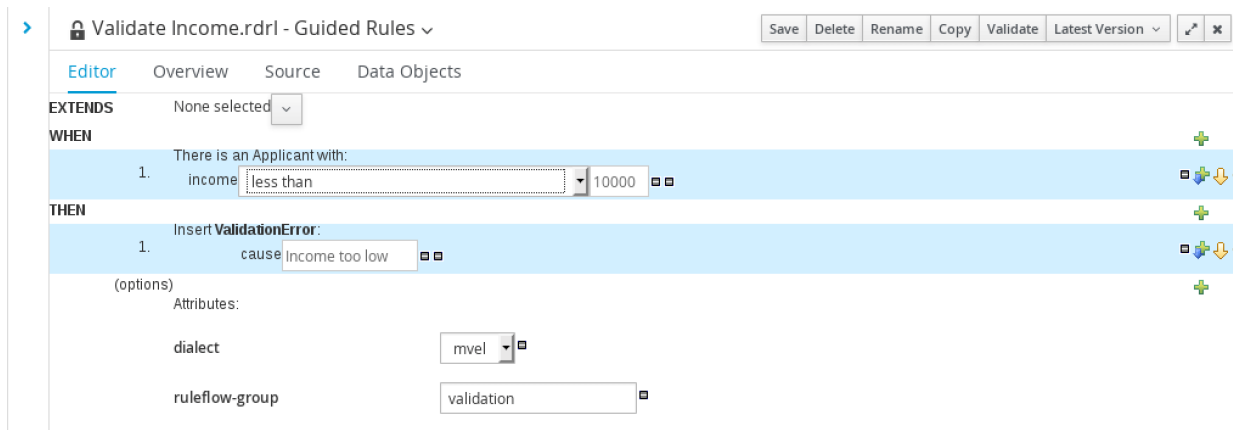


Figure 48. Income Validation Rule

Save and now create another 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 of *Invalid Social Security Number*. Add a ruleflow-group attribute to the rule and set it to validation:

The screenshot shows the 'Validate SSN.rdr1 - Guided Rules' editor. The 'WHEN' section has two rules: '1. ssn less than 100000000' and '2. ssn greater than 999999999'. The 'THEN' section has one rule: '1. Insert Validation Error: cause Invalid Social Security Number'. Below the rule, there are options for 'dialect' (set to 'mvel') and 'ruleflow-group' (set to 'validation').

Figure 49. SSN Validation Rule

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 a 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 a literal value for the list of the acceptable amortization values: 10, 15, 30. Create a corresponding validation error with an appropriate cause description of *Amortization can only be 10, 15 or 30 years*. Add a ruleflow-group attribute to the rule and set it to validation.

The screenshot shows the 'Validate Amortization.rdr1 - Guided Rules' editor. The 'WHEN' section has one rule: '1. amortization is not contained in the (comma separated) list: 10,15,30'. The 'THEN' section has one rule: '1. Insert Validation Error: cause Amortization can only be 10, 15, or 30 years'. Below the rule, there are options for 'dialect' (set to 'mvel') and 'ruleflow-group' (set to 'validation').

Figure 50. Amortization Validation Rule

Retracting Facts After Validation

Given the possibility that multiple 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 Item of type *DRL file* 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
```

Resetting 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 the rule with the corresponding business rule task in the process by specifying the correct ruleflow group.

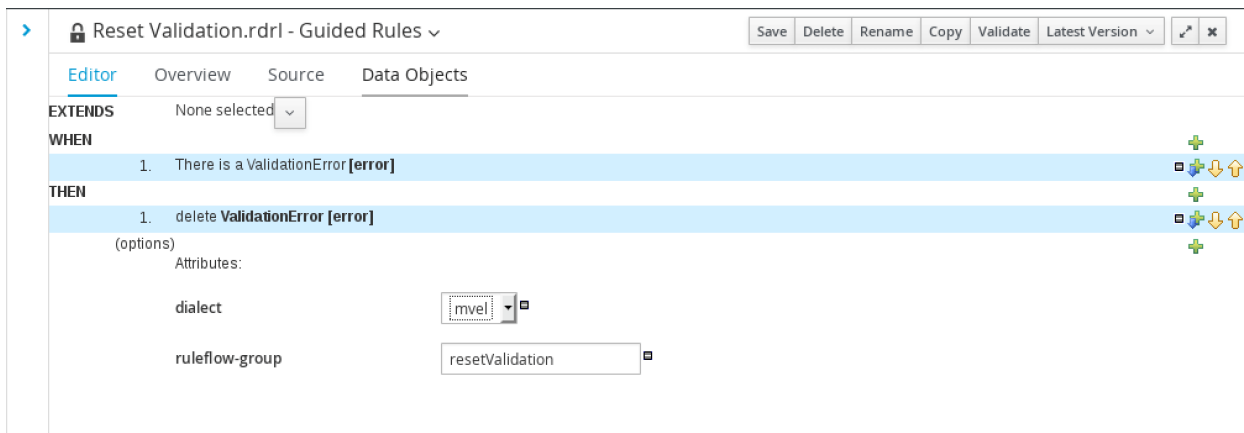


Figure 51. Reset Validation Rule

Linking Validation Rules & Process

Returning to the validation process flow in the process editor, 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 supplied rules of group *validation* 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.

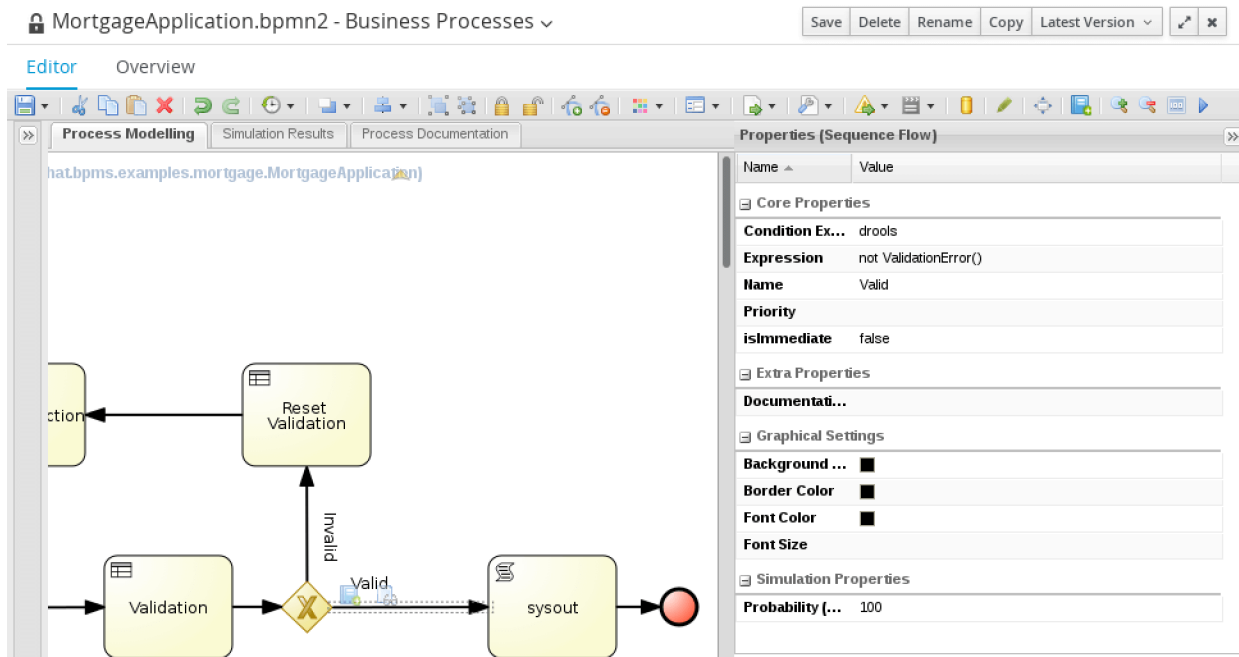


Figure 52. Setting Valid/Invalid Conditions

Next, 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:

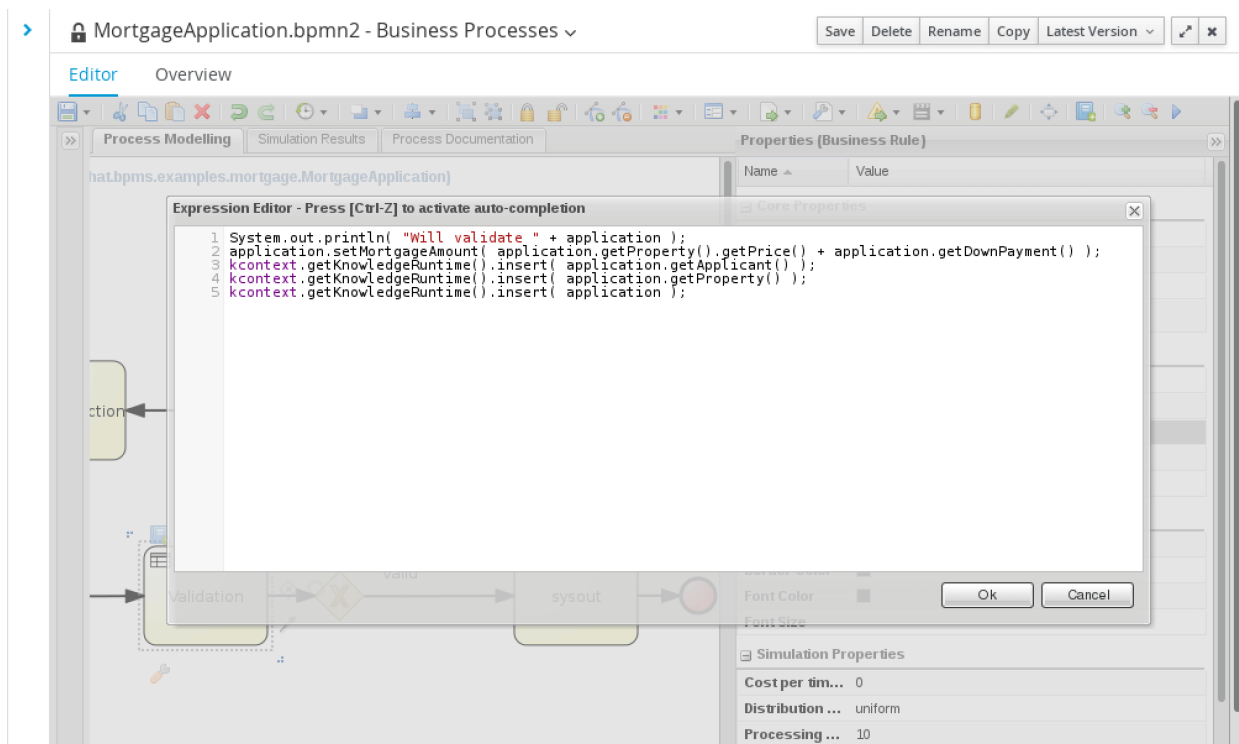


Figure 53. Validation On-Entry Actions

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 Process Model

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.

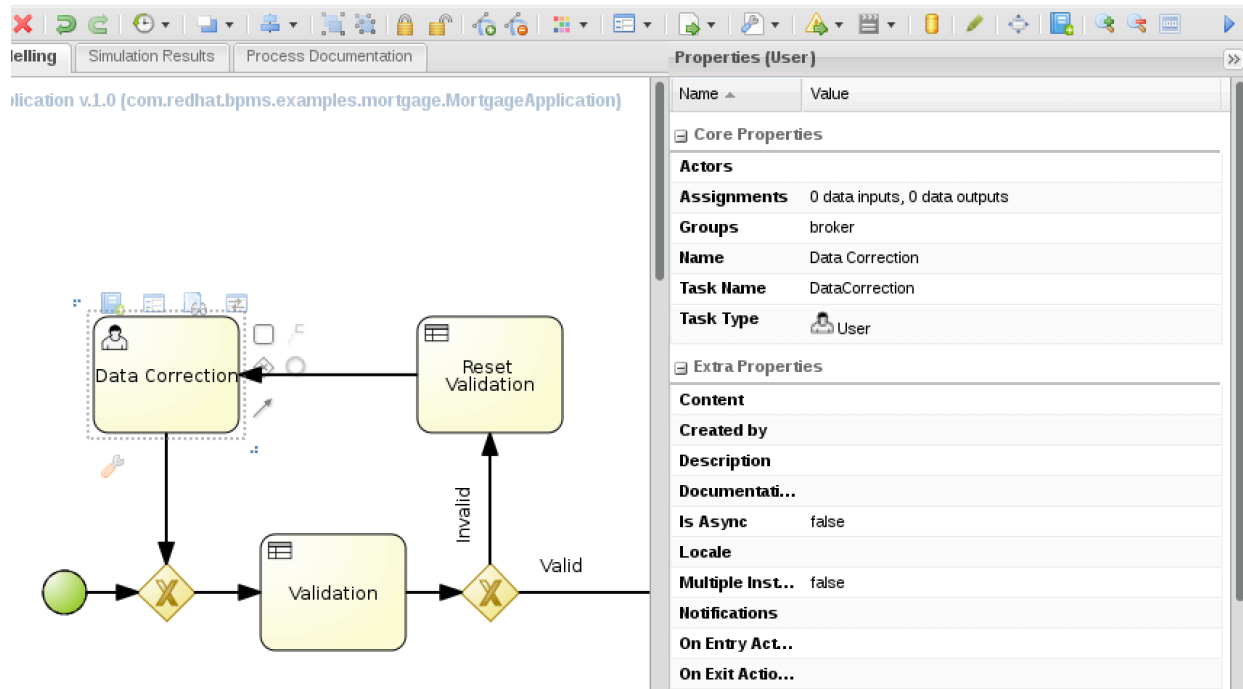
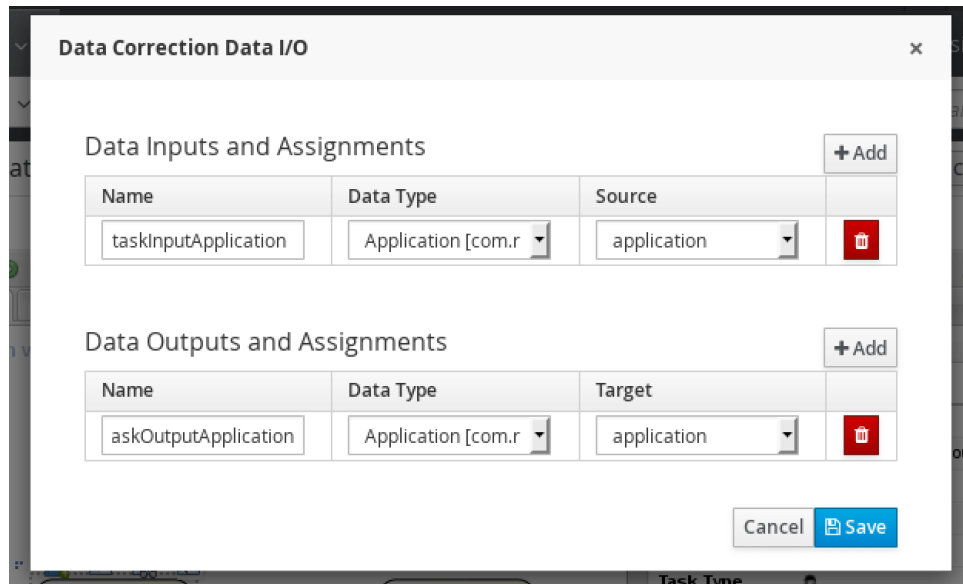



Figure 54. Data Correction Properties

Select the top-right icon for *Edit I/O* over the Data Correction node. Click *Add* next to *Data Inputs and Assignments*, which will serve to provide the application data to the human actor reviewing and completing the human task. Provide the entire mortgage application data to the broker by declaring the input variable with a distinct name of *taskInputApplication* and type of *com.redhat.bpms.examples.mortgage.Application*.

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*.



Data Inputs and Assignments			
Name	Data Type	Source	
taskInputApplication	Application [com.r]	application	


Data Outputs and Assignments			
Name	Data Type	Target	
askOutputApplication	Application [com.r]	application	

Figure 55. Data Correction Input/Output

By declaring the above input/output, we have mapped the process variable *application* to the task variables:

1. *application* variable is mapped to *taskInputApplication* so that the broker can view the data and proceed to correct it.
2. *taskOutputApplication* back to the *application* variable of the process so that any corrections made by the broker are applied to the entire process going forward.

Data Correction Form

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, then the *Edit Task Form* icon just above the node to create or edit a corresponding task form. As before, select the Graphical Modeler to build the form.

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:

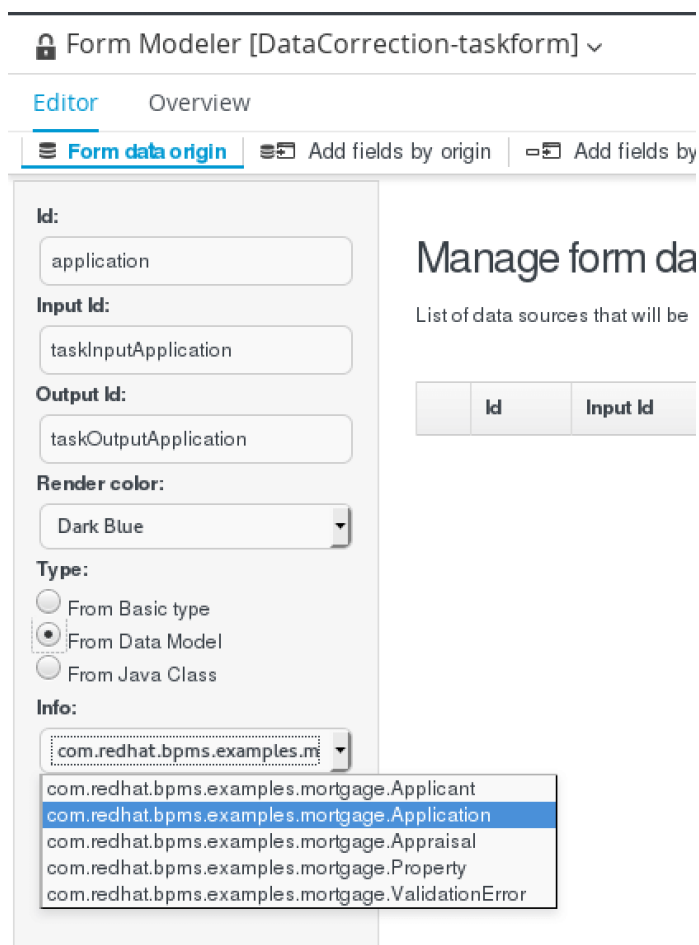


Figure 56. Data Correction Data Holders

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 addition to setting a user friendly label, also set the corresponding previously created form as the default form of the field. The final data correction form looks as follows:

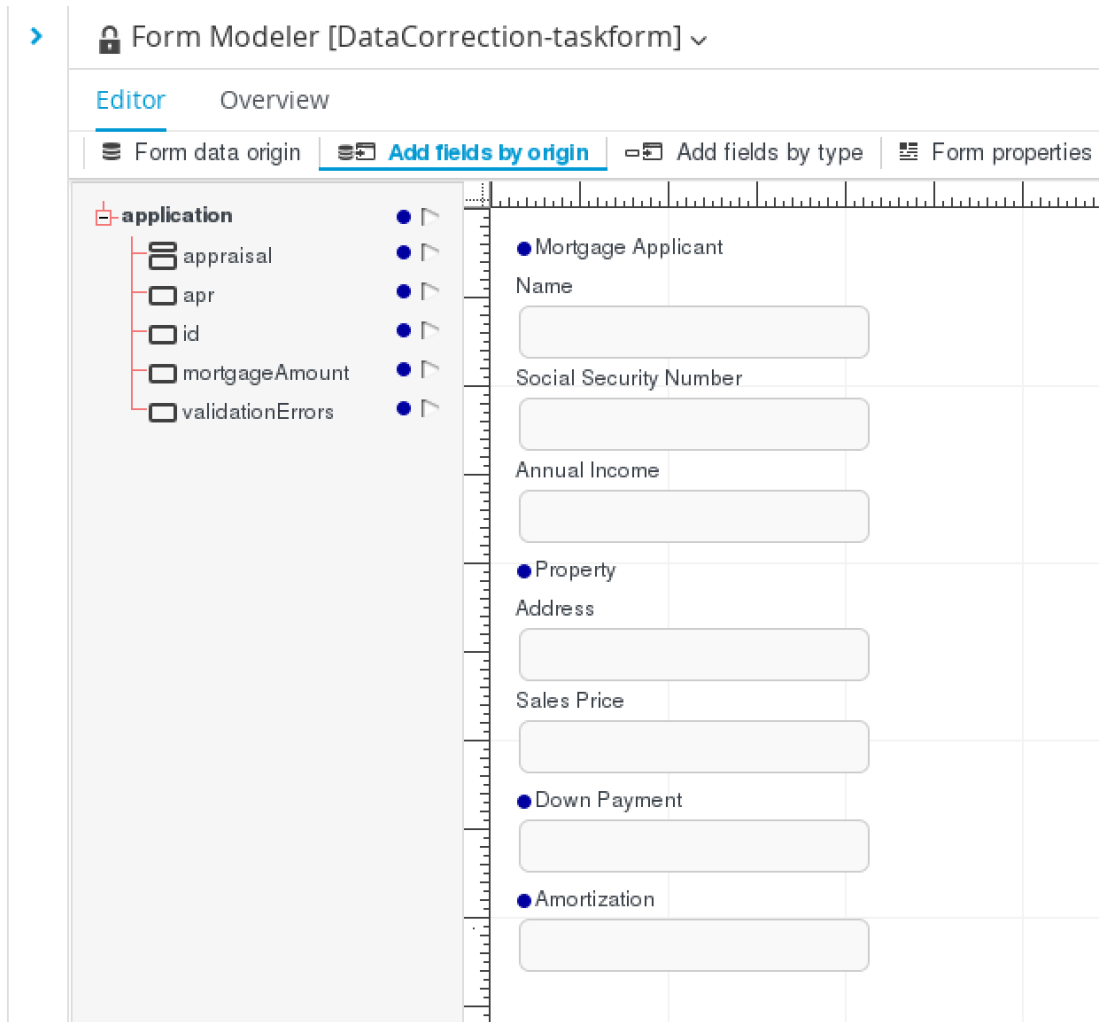


Figure 57. Data Correction Form

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 onto the canvas. 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.

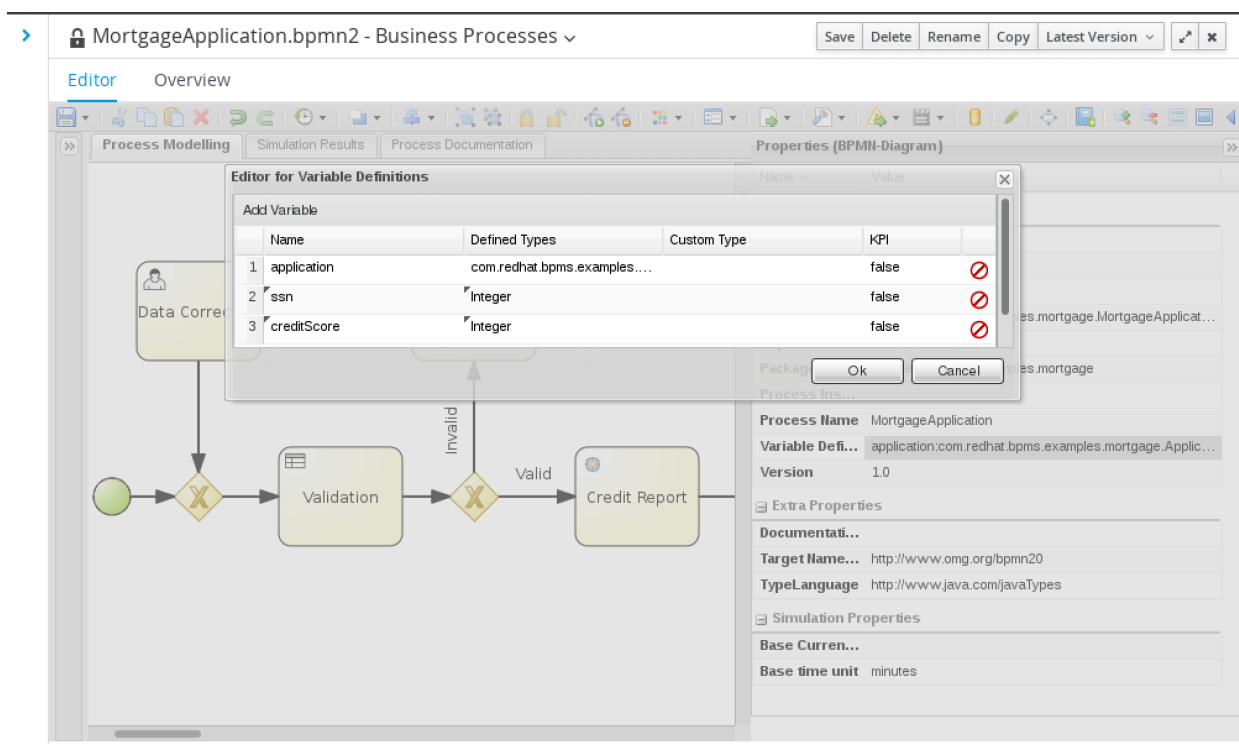


Figure 58. Additional Process Variables

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 inputs and outputs. Values are given as Constants unless noted otherwise:

- *Interface*: CreditServiceService
- *Parameter*: ssn (from dropdown)
- *Mode*: SYNC
- *Operation*: getCreditScore
- *Namespace*: <http://mortgage.examples.bpms.redhat.com/>
- *Url*: <http://localhost:8080/jboss-mortgage-demo-ws/CreditService?WSDL>
- *Endpoint*: remove this property by clicking the red trash icon
- *Result*: creditScore (from dropdown)

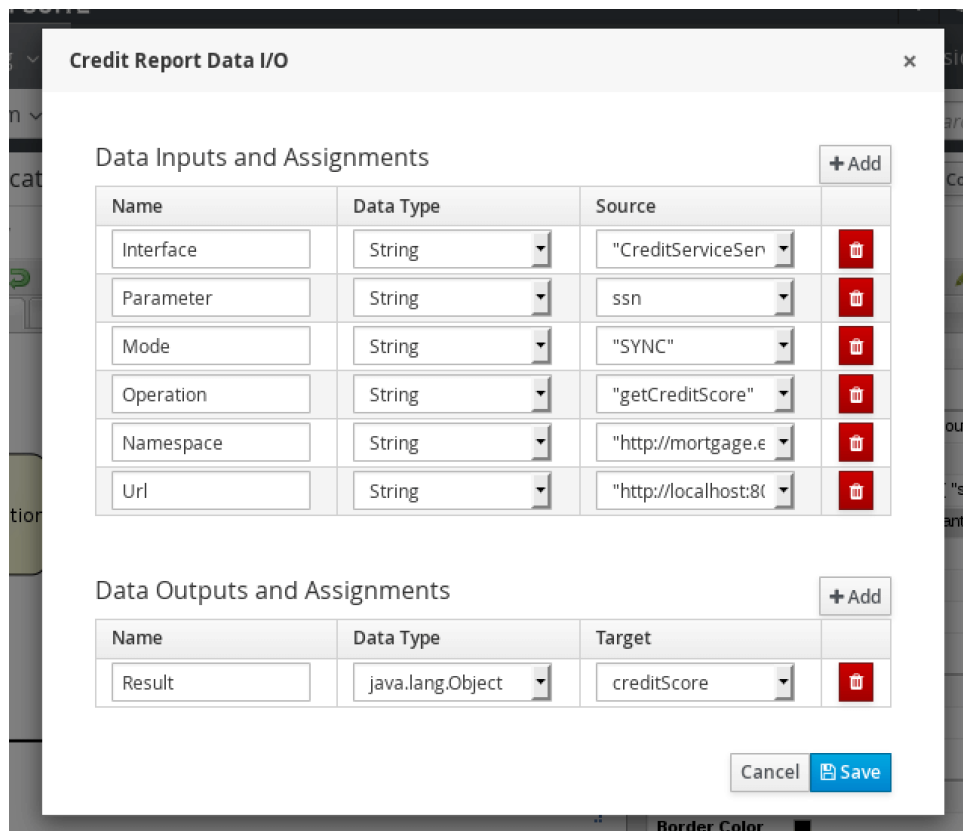


Figure 59. Web Service Task Data Assignments

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 below.

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. Rather than recreating the service, if you would prefer, you can find a deployable .war file in the accompanying download for this document.

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

On entry:

```
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>

Web Service 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:

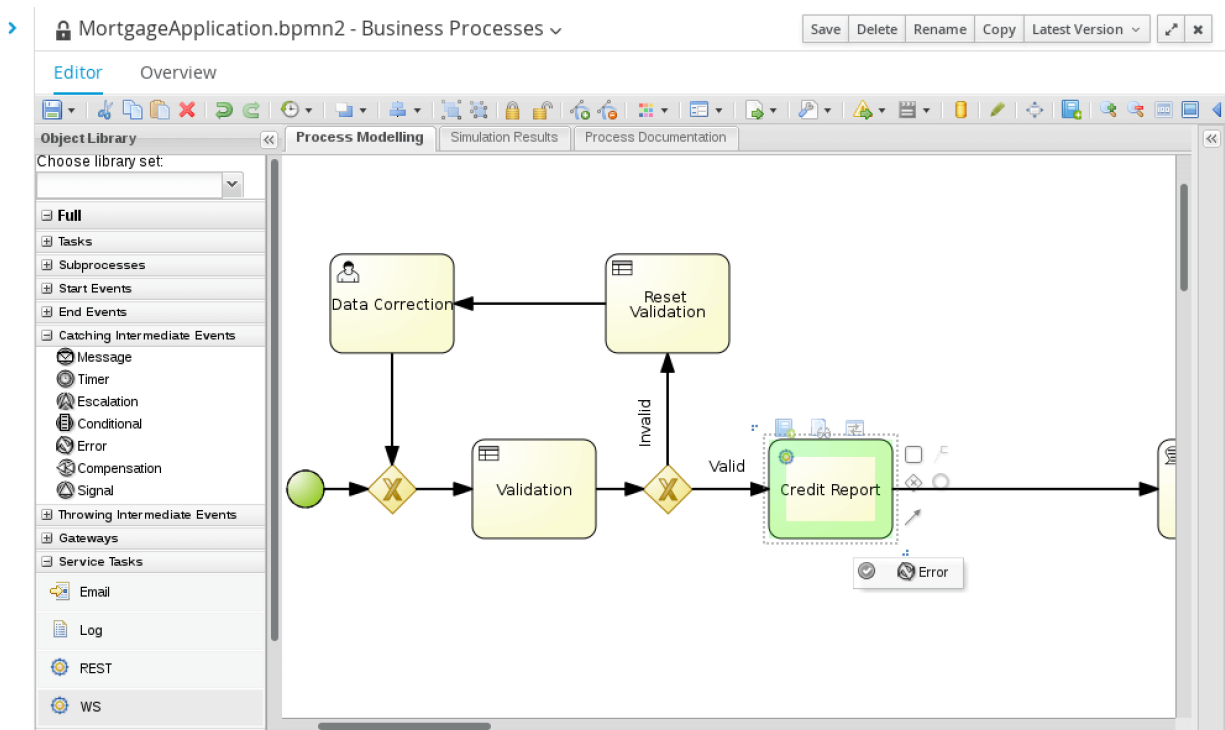


Figure 60. Web Service Error Placement

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*.

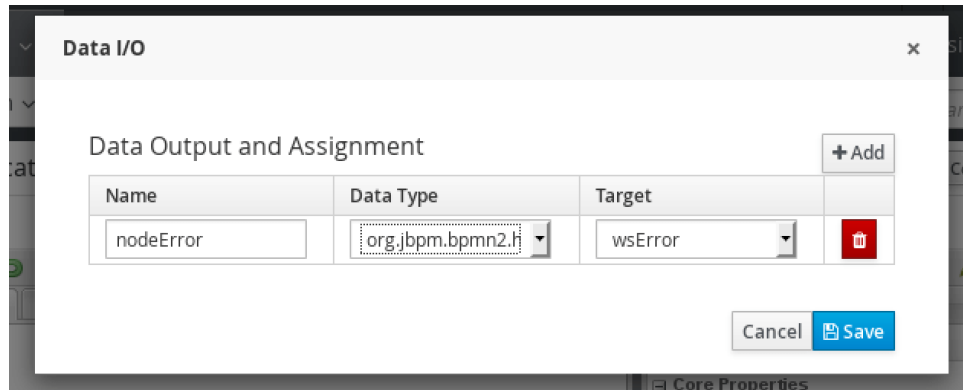


Figure 61. Web Service Data Output

Create a new user task called *Troubleshoot*, provide the same as the name property, 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**, as is the case here.

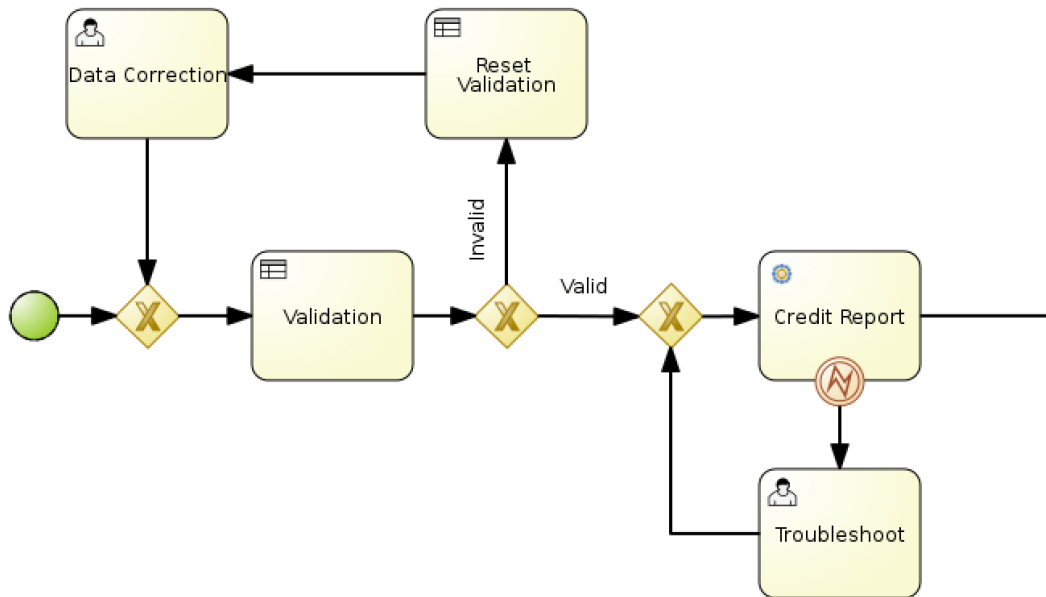


Figure 62. Web Service Error Flow

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.

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.

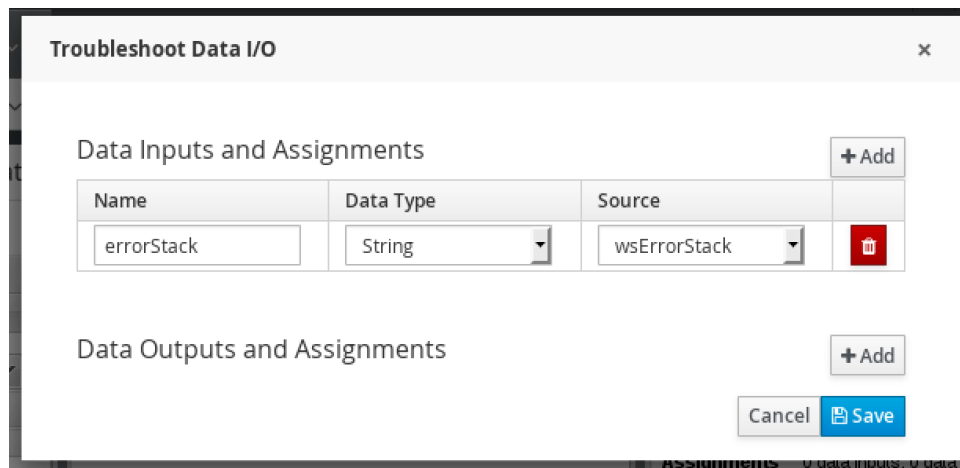


Figure 63. Troubleshoot Data Input

Using the Graphic Modeler, choose to *Add fields by type* this time. 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.

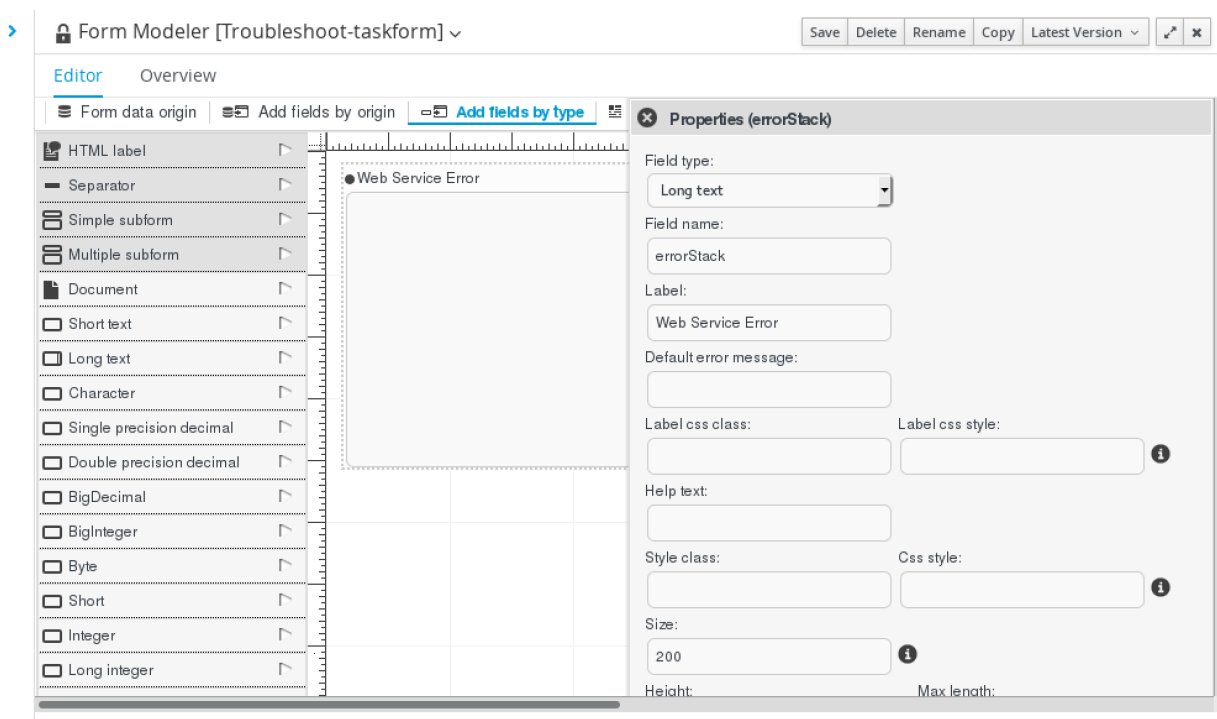


Figure 64. Troubleshoot Task Form

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.

5.4.6. Mortgage Calculation

Mortgage Calculation Process Model

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.

Mortgage Calculation Rules

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.

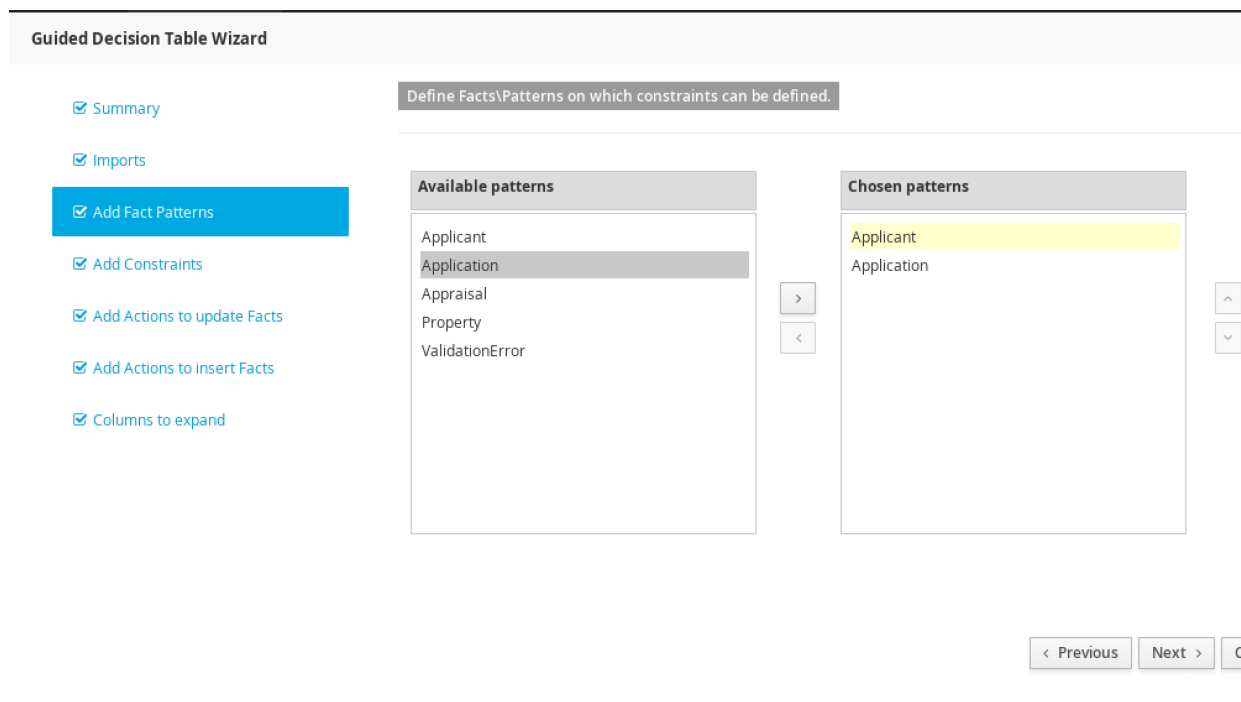


Figure 65. Guided Decision Table Wizard

In the next step, select each pattern, a field for that pattern, then create a constraint for that given 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* \geq 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 table allows an analyst to easily create or update rules. By providing a value list, the application limits the credit score brackets to known values and generates 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:

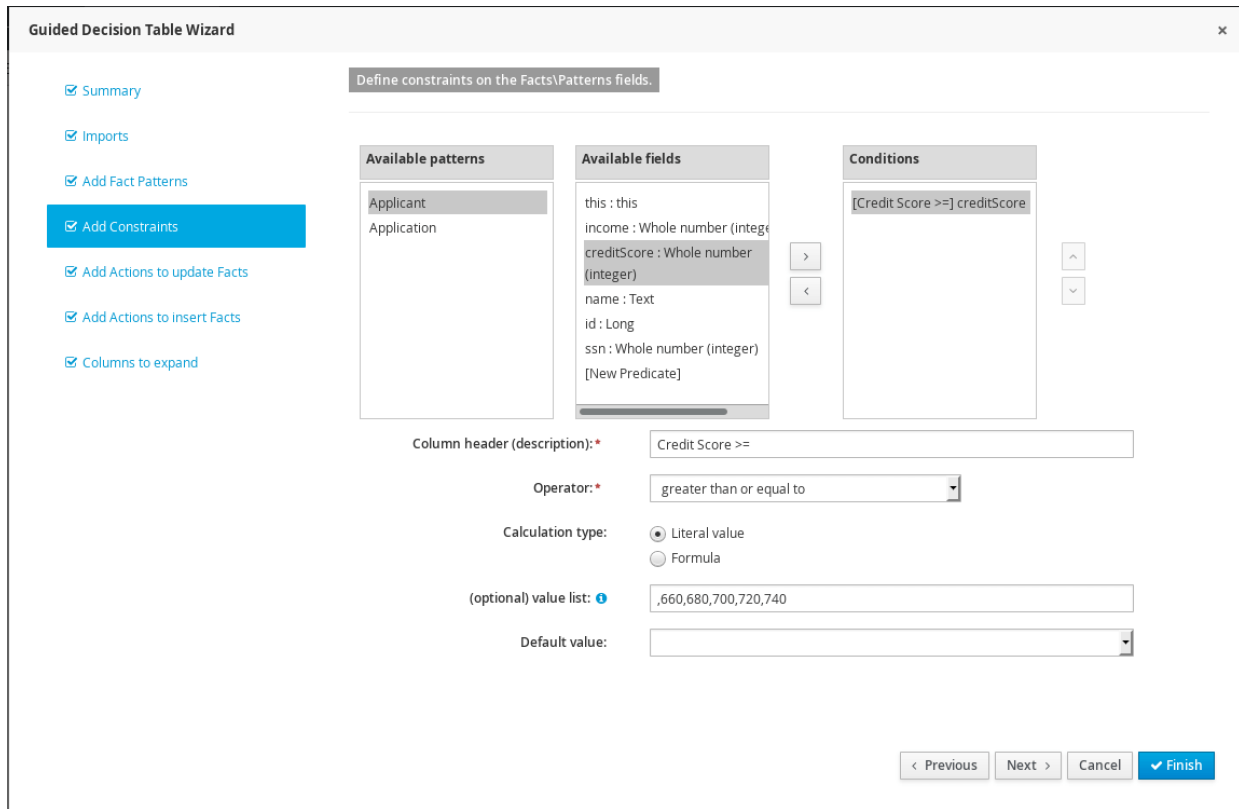


Figure 66. Credit Score Condition

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*.

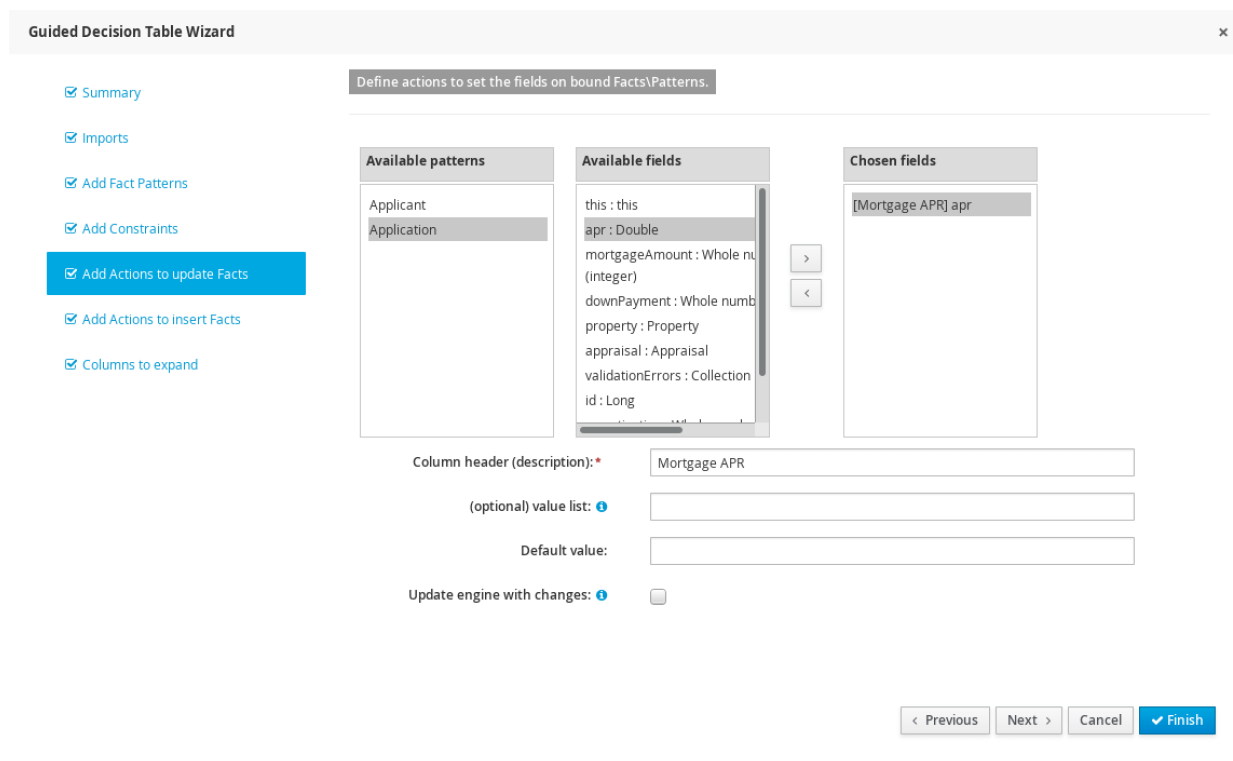


Figure 67. Decision Table Actions

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:

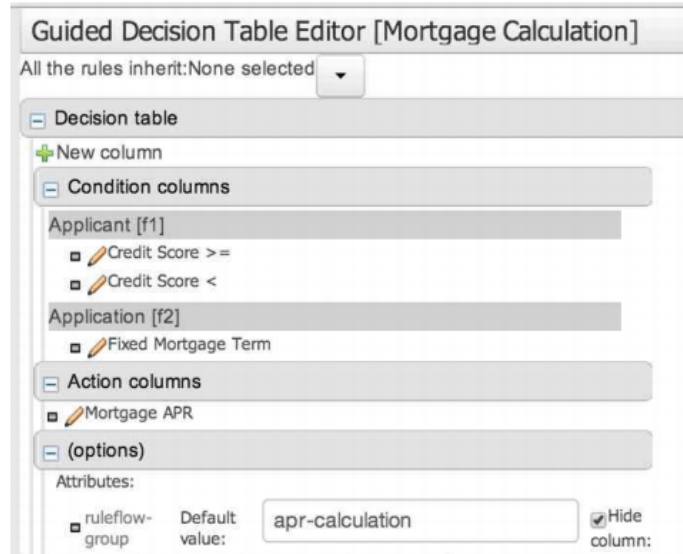


Figure 68. Adding Table Metadata

The generated table likely will require some manipulation to remove rows which can't logically occur, as well as providing APR values for each entry.

At the time of writing, [a known issue](#) sometimes prevents the addition of metadata on a guided decision table, showing no entries under (*options*). Should you encounter this issue, the above section serves as an excellent means of getting familiar with decision tables, however, you can delete the guided table you've created and manually enter a *DRL file* to accomplish the same end-goal:

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

rule "Row 1 Mortgage Calculation"
  ruleflow-group "apr-calculation"
  dialect "mvel"
  when
    f1 : Applicant( creditScore >= 660 , creditScore < 680 )
    f2 : Application( amortization == 10 )
  then
    f2.setApr( 10.0 );
  end

rule "Row 2 Mortgage Calculation"
  ruleflow-group "apr-calculation"
  dialect "mvel"
  when
    f1 : Applicant( creditScore >= 660 , creditScore < 680 )
    f2 : Application( amortization == 15 )
  then
    f2.setApr( 9.5 );
```

```
end

rule "Row 3 Mortgage Calculation"
  ruleflow-group "apr-calculation"
  dialect "mvel"
  when
    f1 : Applicant( creditScore >= 660 , creditScore < 680 )
    f2 : Application( amortization == 30 )
  then
    f2.setApr( 9.0 );
  end

rule "Row 4 Mortgage Calculation"
  ruleflow-group "apr-calculation"
  dialect "mvel"
  when
    f1 : Applicant( creditScore >= 660 , creditScore < 700 )
    f2 : Application( amortization == 10 )
  then
    f2.setApr( 9.5 );
  end

rule "Row 5 Mortgage Calculation"
  ruleflow-group "apr-calculation"
  dialect "mvel"
  when
    f1 : Applicant( creditScore >= 660 , creditScore < 700 )
    f2 : Application( amortization == 15 )
  then
    f2.setApr( 9.0 );
  end

rule "Row 6 Mortgage Calculation"
  ruleflow-group "apr-calculation"
  dialect "mvel"
  when
    f1 : Applicant( creditScore >= 660 , creditScore < 700 )
    f2 : Application( amortization == 30 )
  then
    f2.setApr( 8.5 );
  end

rule "Row 7 Mortgage Calculation"
  ruleflow-group "apr-calculation"
  dialect "mvel"
  when
    f1 : Applicant( creditScore >= 660 , creditScore < 720 )
    f2 : Application( amortization == 10 )
```

```
    then
      f2.setApr( 9.0 );
    end

rule "Row 8 Mortgage Calculation"
  ruleflow-group "apr-calculation"
  dialect "mvel"
  when
    f1 : Applicant( creditScore >= 660 , creditScore < 720 )
    f2 : Application( amortization == 15 )
  then
    f2.setApr( 8.5 );
  end

rule "Row 9 Mortgage Calculation"
  ruleflow-group "apr-calculation"
  dialect "mvel"
  when
    f1 : Applicant( creditScore >= 660 , creditScore < 720 )
    f2 : Application( amortization == 30 )
  then
    f2.setApr( 8.0 );
  end

rule "Row 10 Mortgage Calculation"
  ruleflow-group "apr-calculation"
  dialect "mvel"
  when
    f1 : Applicant( creditScore >= 660 , creditScore < 740 )
    f2 : Application( amortization == 10 )
  then
    f2.setApr( 8.5 );
  end

rule "Row 11 Mortgage Calculation"
  ruleflow-group "apr-calculation"
  dialect "mvel"
  when
    f1 : Applicant( creditScore >= 660 , creditScore < 740 )
    f2 : Application( amortization == 15 )
  then
    f2.setApr( 8.0 );
  end

rule "Row 12 Mortgage Calculation"
  ruleflow-group "apr-calculation"
  dialect "mvel"
  when
```

```
f1 : Applicant( creditScore >= 660 , creditScore < 740 )
f2 : Application( amortization == 30 )
then
  f2.setApr( 7.5 );
end

rule "Row 13 Mortgage Calculation"
  ruleflow-group "apr-calculation"
  dialect "mvel"
  when
    f1 : Applicant( creditScore >= 680 , creditScore < 700 )
    f2 : Application( amortization == 10 )
  then
    f2.setApr( 8.0 );
  end

rule "Row 14 Mortgage Calculation"
  ruleflow-group "apr-calculation"
  dialect "mvel"
  when
    f1 : Applicant( creditScore >= 680 , creditScore < 700 )
    f2 : Application( amortization == 15 )
  then
    f2.setApr( 7.5 );
  end

rule "Row 15 Mortgage Calculation"
  ruleflow-group "apr-calculation"
  dialect "mvel"
  when
    f1 : Applicant( creditScore >= 680 , creditScore < 700 )
    f2 : Application( amortization == 30 )
  then
    f2.setApr( 7.0 );
  end

rule "Row 16 Mortgage Calculation"
  ruleflow-group "apr-calculation"
  dialect "mvel"
  when
    f1 : Applicant( creditScore >= 680 , creditScore < 720 )
    f2 : Application( amortization == 10 )
  then
    f2.setApr( 7.5 );
  end

rule "Row 17 Mortgage Calculation"
  ruleflow-group "apr-calculation"
```

```
dialect "mvel"
when
  f1 : Applicant( creditScore >= 680 , creditScore < 720 )
  f2 : Application( amortization == 15 )
then
  f2.setApr( 7.0 );
end

rule "Row 18 Mortgage Calculation"
  ruleflow-group "apr-calculation"
  dialect "mvel"
  when
    f1 : Applicant( creditScore >= 680 , creditScore < 720 )
    f2 : Application( amortization == 30 )
  then
    f2.setApr( 6.5 );
  end

rule "Row 19 Mortgage Calculation"
  ruleflow-group "apr-calculation"
  dialect "mvel"
  when
    f1 : Applicant( creditScore >= 680 , creditScore < 740 )
    f2 : Application( amortization == 10 )
  then
    f2.setApr( 7.0 );
  end

rule "Row 20 Mortgage Calculation"
  ruleflow-group "apr-calculation"
  dialect "mvel"
  when
    f1 : Applicant( creditScore >= 680 , creditScore < 740 )
    f2 : Application( amortization == 15 )
  then
    f2.setApr( 6.5 );
  end

rule "Row 21 Mortgage Calculation"
  ruleflow-group "apr-calculation"
  dialect "mvel"
  when
    f1 : Applicant( creditScore >= 680 , creditScore < 740 )
    f2 : Application( amortization == 30 )
  then
    f2.setApr( 6.0 );
  end
```



```
rule "Row 22 Mortgage Calculation"
  ruleflow-group "apr-calculation"
  dialect "mvel"
  when
    f1 : Applicant( creditScore >= 700 , creditScore < 720 )
    f2 : Application( amortization == 10 )
  then
    f2.setApr( 6.5 );
  end
```

```
rule "Row 23 Mortgage Calculation"
  ruleflow-group "apr-calculation"
  dialect "mvel"
  when
    f1 : Applicant( creditScore >= 700 , creditScore < 720 )
    f2 : Application( amortization == 15 )
  then
    f2.setApr( 6.0 );
  end
```

```
rule "Row 24 Mortgage Calculation"
  ruleflow-group "apr-calculation"
  dialect "mvel"
  when
    f1 : Applicant( creditScore >= 700 , creditScore < 720 )
    f2 : Application( amortization == 30 )
  then
    f2.setApr( 5.5 );
  end
```

```
rule "Row 25 Mortgage Calculation"
  ruleflow-group "apr-calculation"
  dialect "mvel"
  when
    f1 : Applicant( creditScore >= 700 , creditScore < 740 )
    f2 : Application( amortization == 10 )
  then
    f2.setApr( 6.0 );
  end
```

```
rule "Row 26 Mortgage Calculation"
  ruleflow-group "apr-calculation"
  dialect "mvel"
  when
    f1 : Applicant( creditScore >= 700 , creditScore < 740 )
    f2 : Application( amortization == 15 )
  then
    f2.setApr( 5.5 );
```

```
end

rule "Row 27 Mortgage Calculation"
  ruleflow-group "apr-calculation"
  dialect "mvel"
  when
    f1 : Applicant( creditScore >= 700 , creditScore < 740 )
    f2 : Application( amortization == 30 )
  then
    f2.setApr( 5.0 );
  end

rule "Row 28 Mortgage Calculation"
  ruleflow-group "apr-calculation"
  dialect "mvel"
  when
    f1 : Applicant( creditScore >= 720 , creditScore < 740 )
    f2 : Application( amortization == 10 )
  then
    f2.setApr( 5.5 );
  end

rule "Row 29 Mortgage Calculation"
  ruleflow-group "apr-calculation"
  dialect "mvel"
  when
    f1 : Applicant( creditScore >= 720 , creditScore < 740 )
    f2 : Application( amortization == 15 )
  then
    f2.setApr( 5.0 );
  end

rule "Row 30 Mortgage Calculation"
  ruleflow-group "apr-calculation"
  dialect "mvel"
  when
    f1 : Applicant( creditScore >= 720 , creditScore < 740 )
    f2 : Application( amortization == 30 )
  then
    f2.setApr( 4.5 );
  end
```

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*

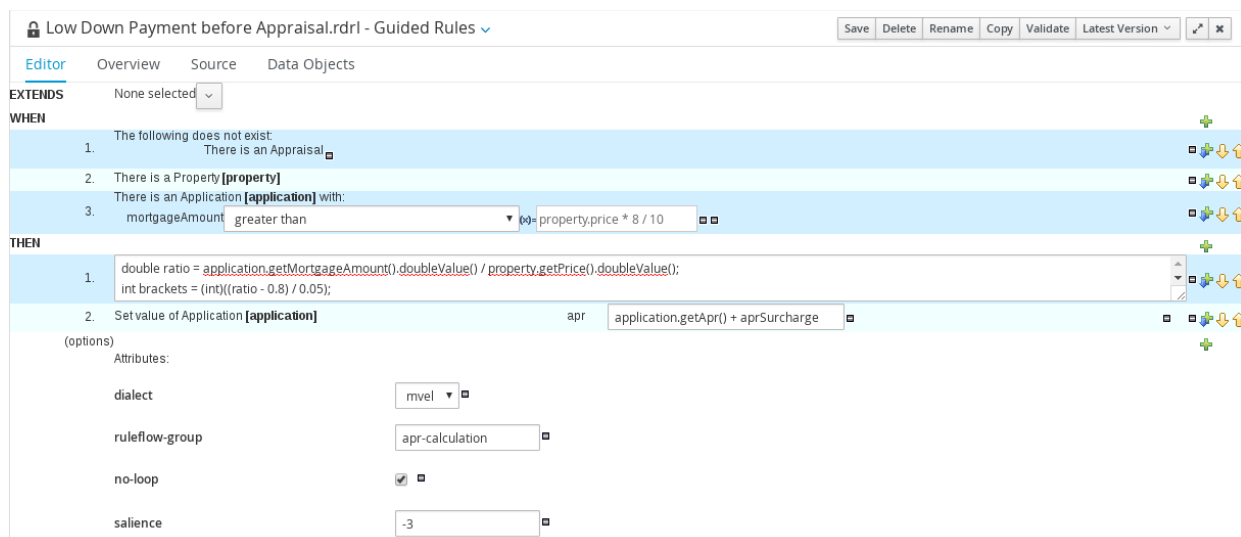


Figure 69. Low Down Payment Rule

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 *saliency* attribute and give it a value of -3. This ensures that the base APR calculation rules in the decision table, with a default saliency 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:

Figure 70. Low Down Payment after Appraisal

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.

Figure 71. Jumbo Mortgage Rule

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
```

5.4.7. 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:

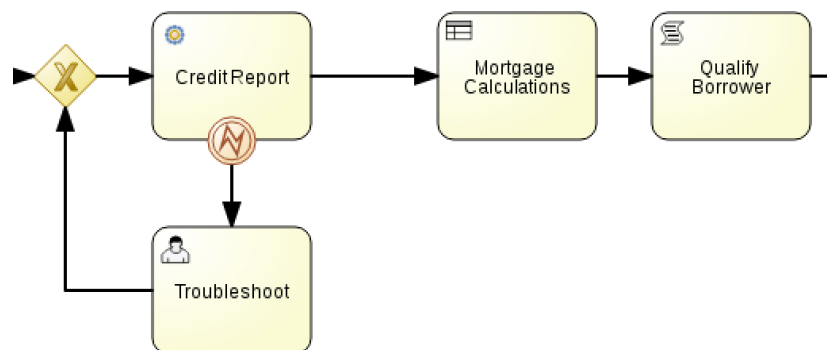


Figure 72. Qualify Borrower Script Task

5.4.8. Increase Down Payment

Increase Down Payment Process Model

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:

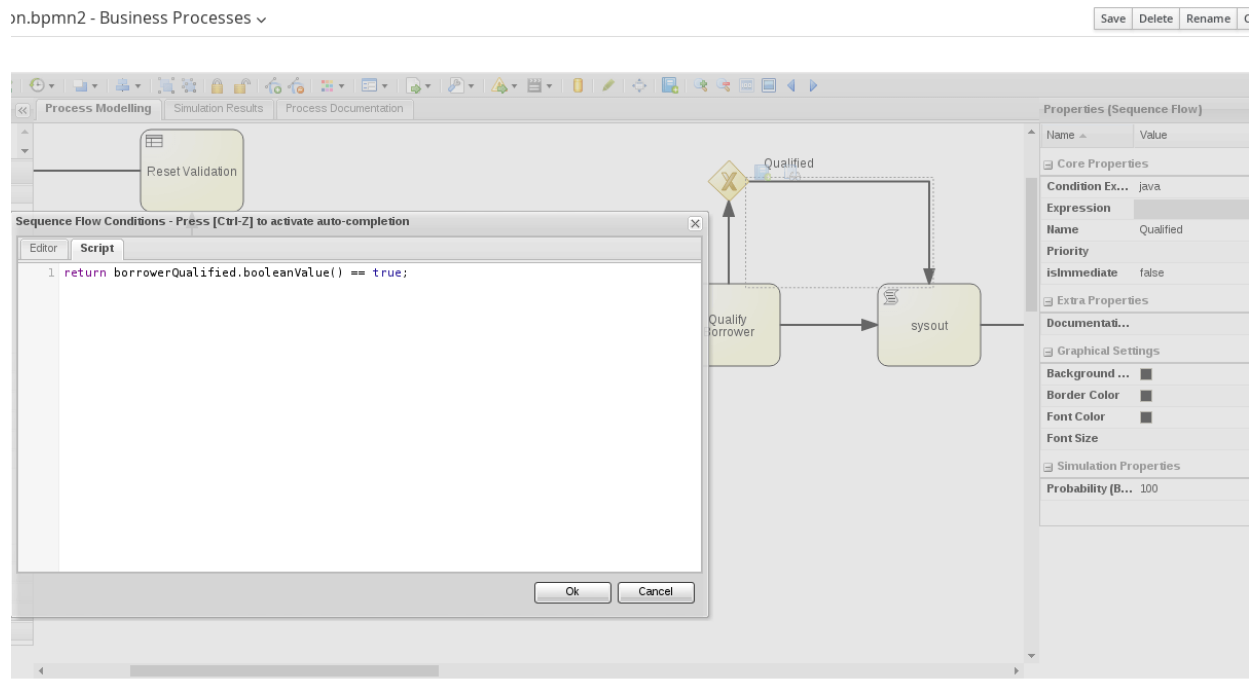


Figure 73. Qualified Borrower

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 `borrowerQualified` 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 `taskInputApplication` 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.

Increase Down Payment Task Form

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.

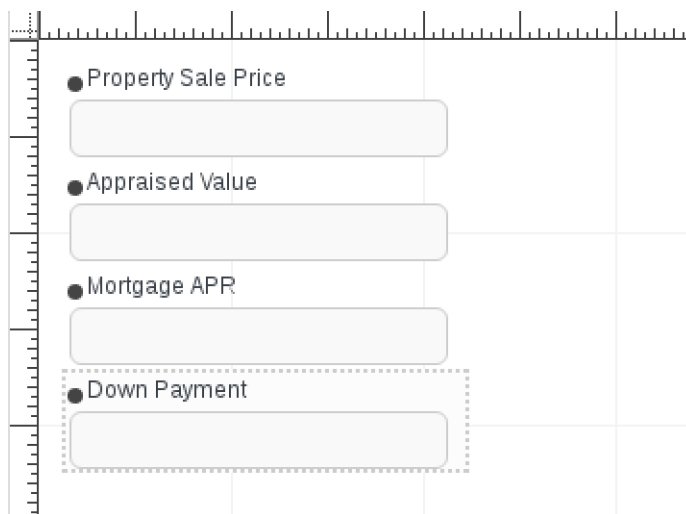
Based on these two common causes, the mortgage apr and the appraisal value (if appraisal is even performed yet) are presented as part of the task form.

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 utilizes *Add fields by type* and *Short text* inputs, thereby directly navigating to and referencing 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.



The image shows a screenshot of a task form titled "Down Payment Task Form". It contains four input fields arranged vertically. Each field has a small circular icon to its left. The fields are labeled as follows: "Property Sale Price", "Appraised Value", "Mortgage APR", and "Down Payment". The first three fields are read-only, indicated by a light gray background and a dashed border. The "Down Payment" field is the only one that is active, indicated by a solid border and a light blue background. The form is set against a grid background.

Figure 74. Down Payment Task Form

Limiting Increase Down Payment Loops

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.

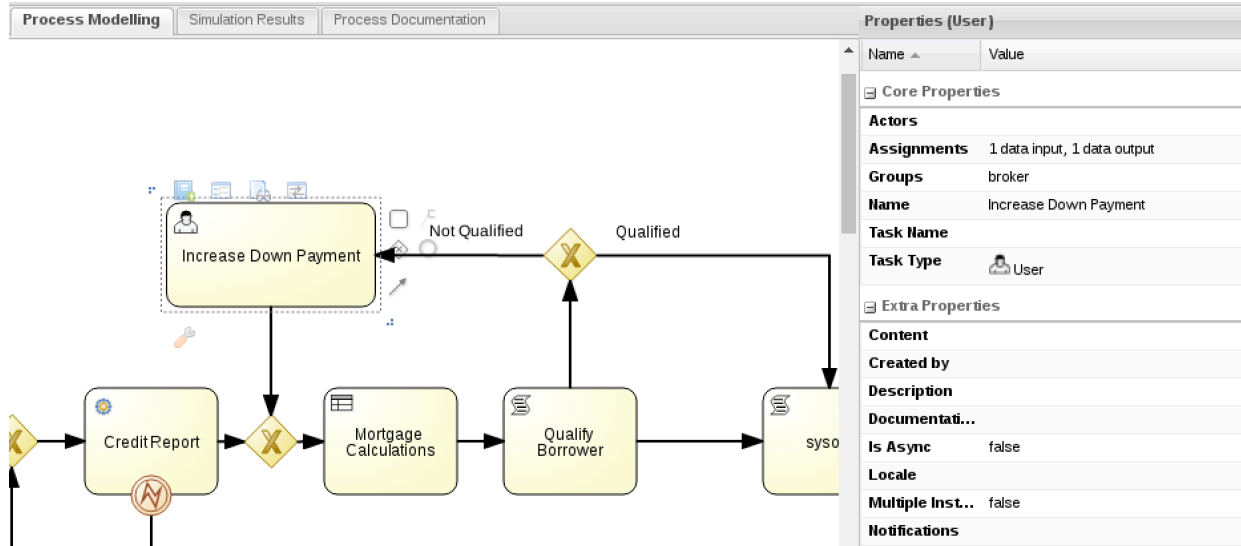


Figure 75. Down Payment Process Flow

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:

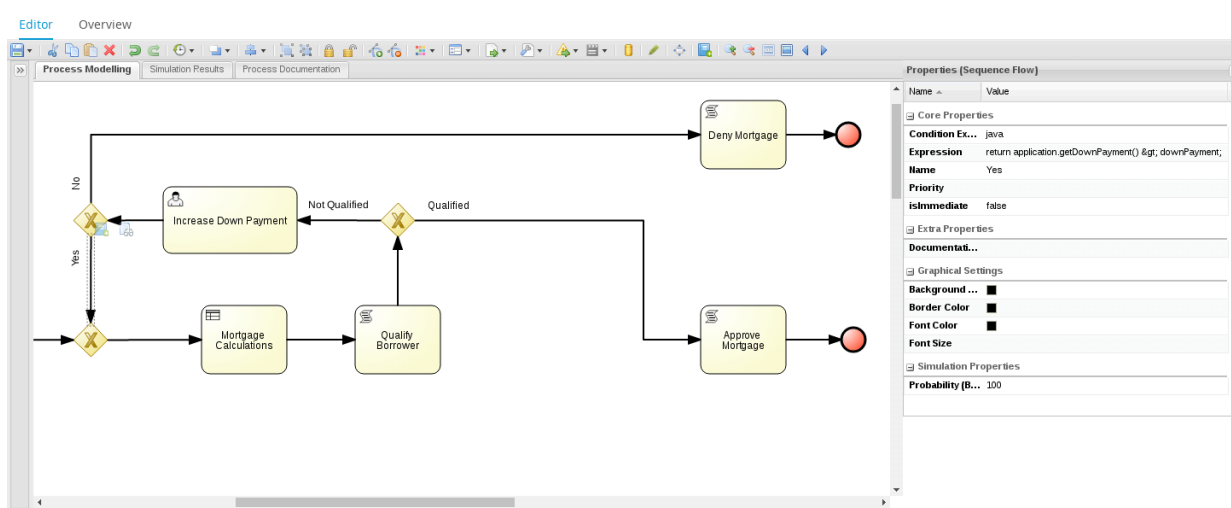


Figure 76. Deny Mortgage Process Flow

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.

5.4.9. Financial Review

Financial Review Process FlowModel

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 process 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 below, 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:

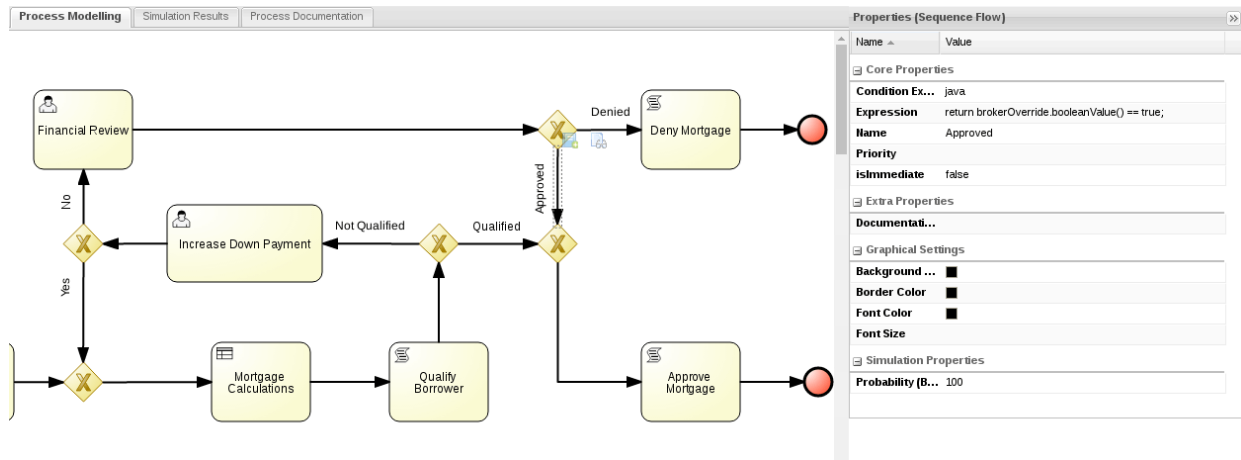


Figure 77. Broker Override Process Flow

Financial Review Task Form

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.

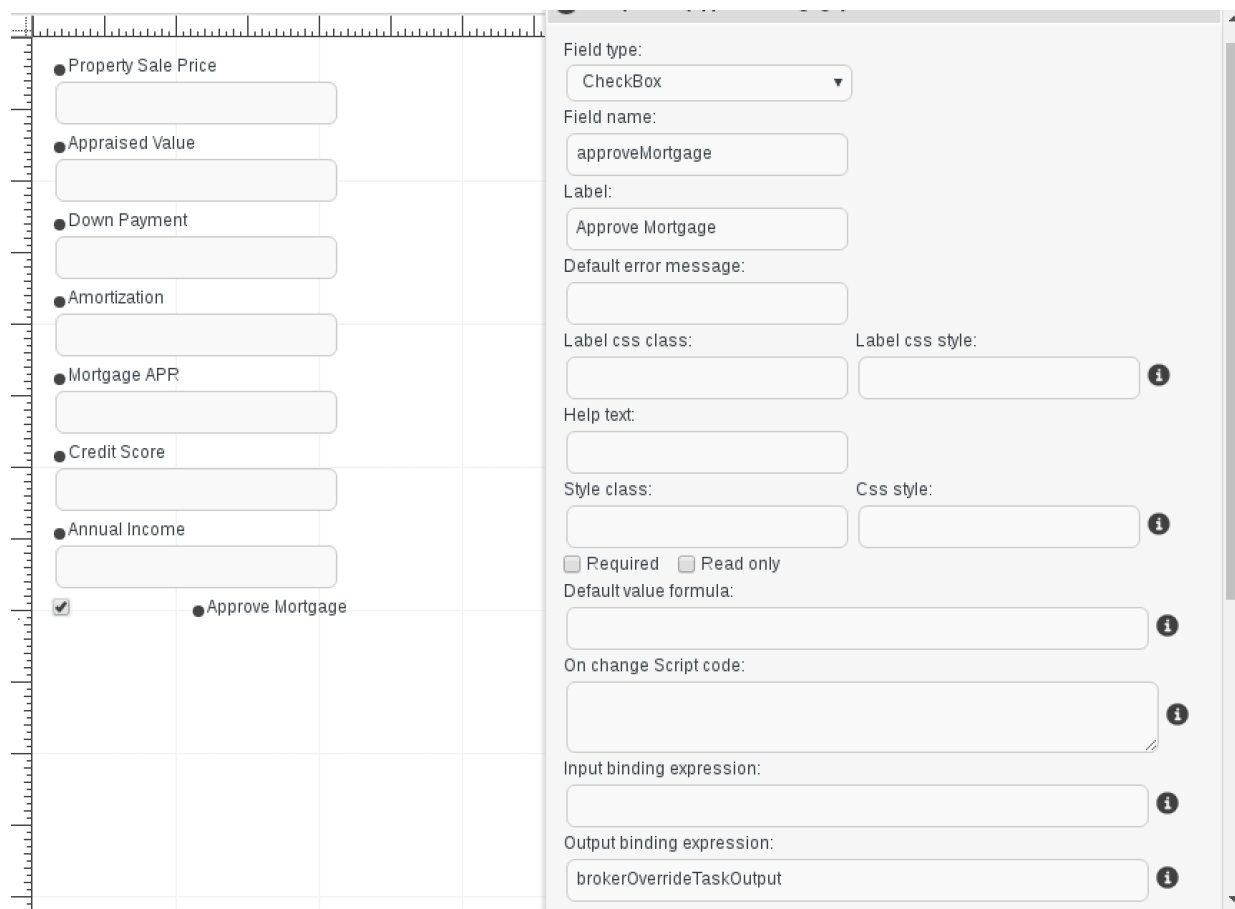


Figure 78. Broker Override Task Form

5.4.10. Appraisal

Appraisal Process Model

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.

Create 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:

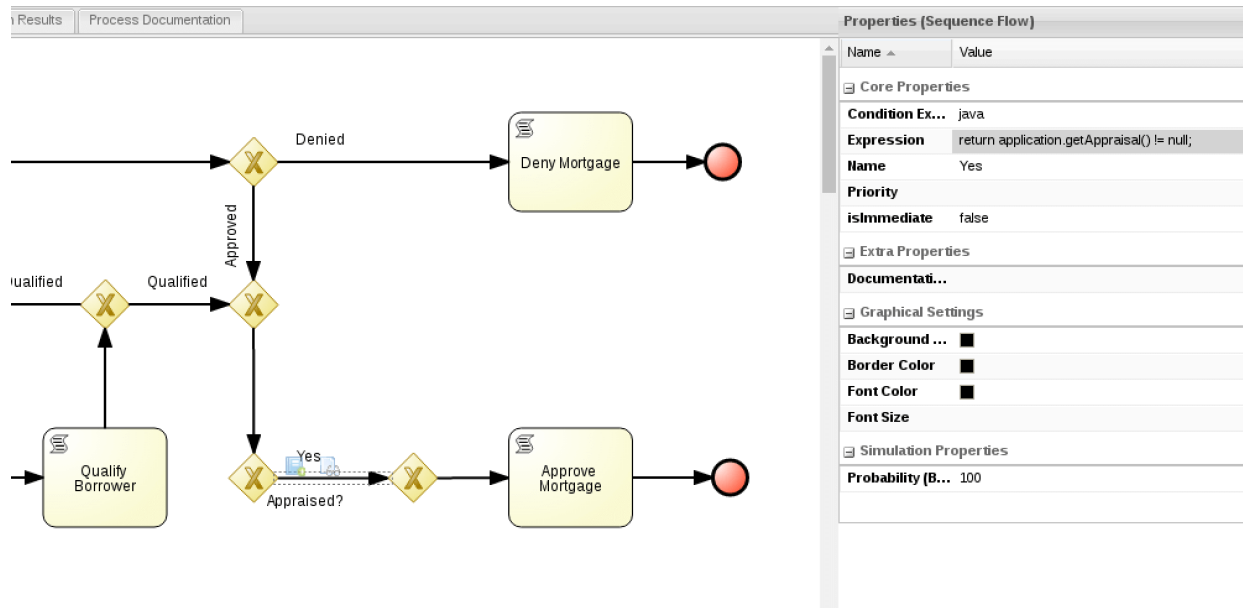


Figure 79. Appraised Process Flow

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:

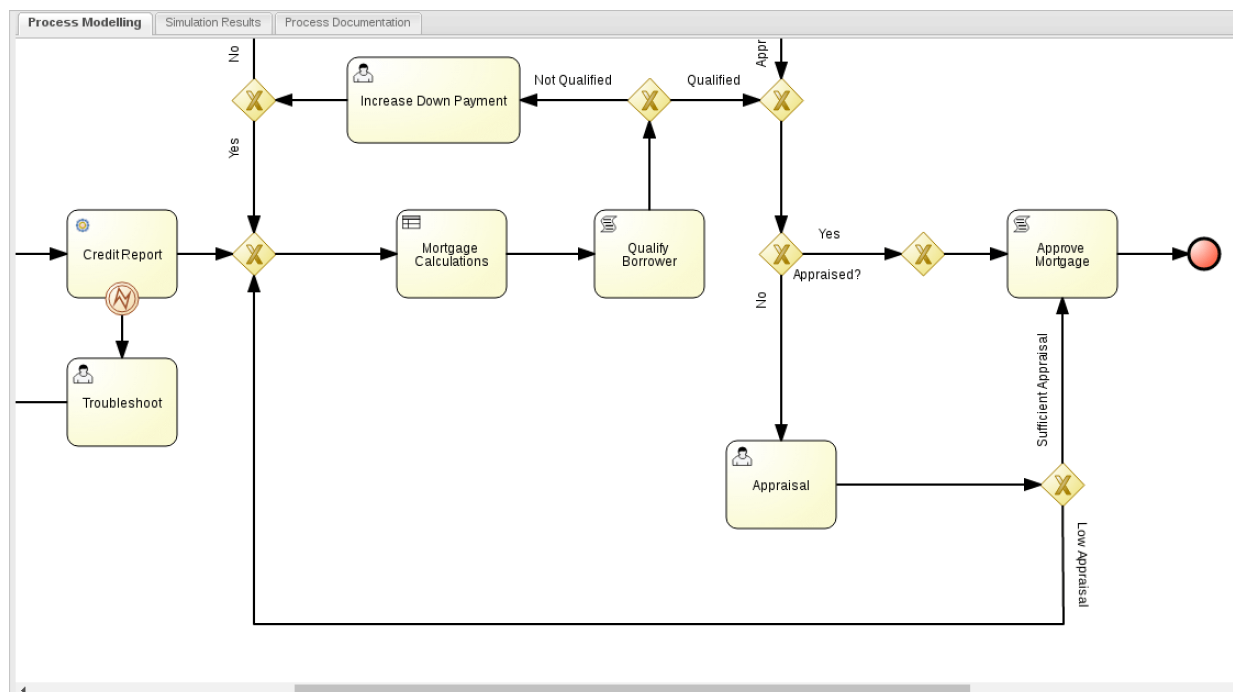


Figure 80. Insufficient Appraisal Process Flow

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 the Figure shown earlier in the chapter titled *Low Down Payment after Appraisal*.

Appraisal Task Form

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.

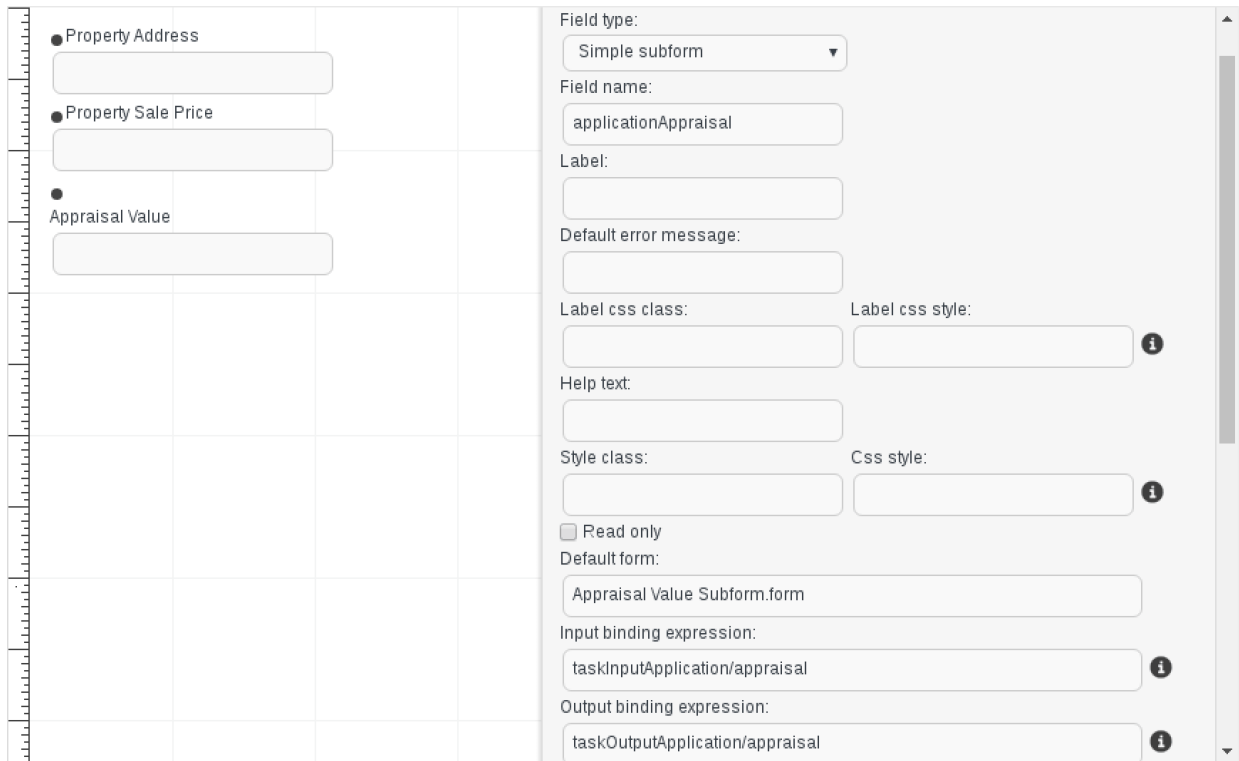


Figure 81. Appraisal Task Form

5.4.11. 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.

Table 6. User Tasks & Swimlanes

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

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:

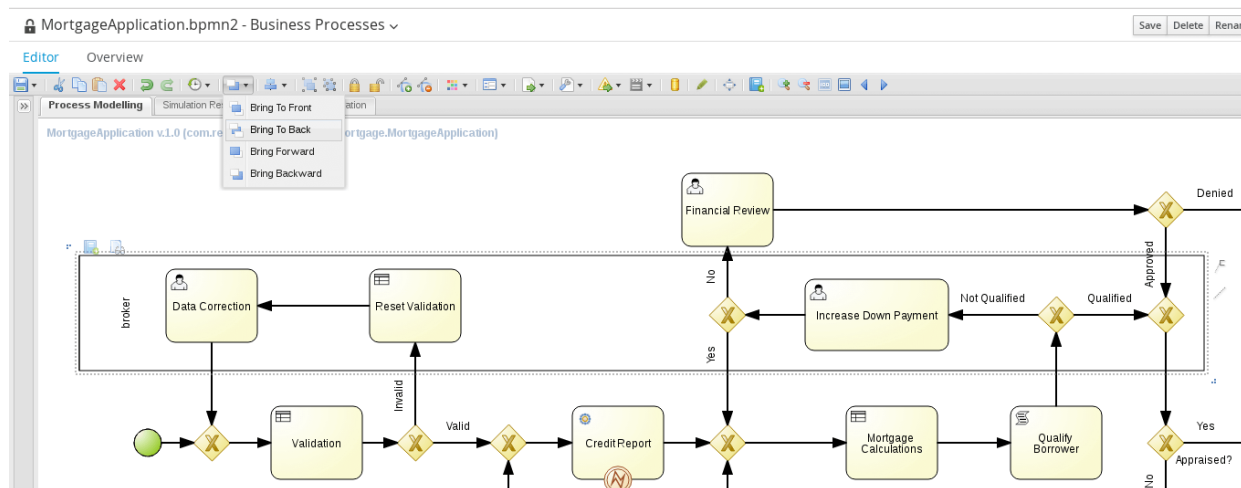


Figure 82. Broker Swimlane

Finally, add a second swimlane above the first encompassing Financial Review for managers.

5.4.12. Final Process Model

The final business process model is as follows:

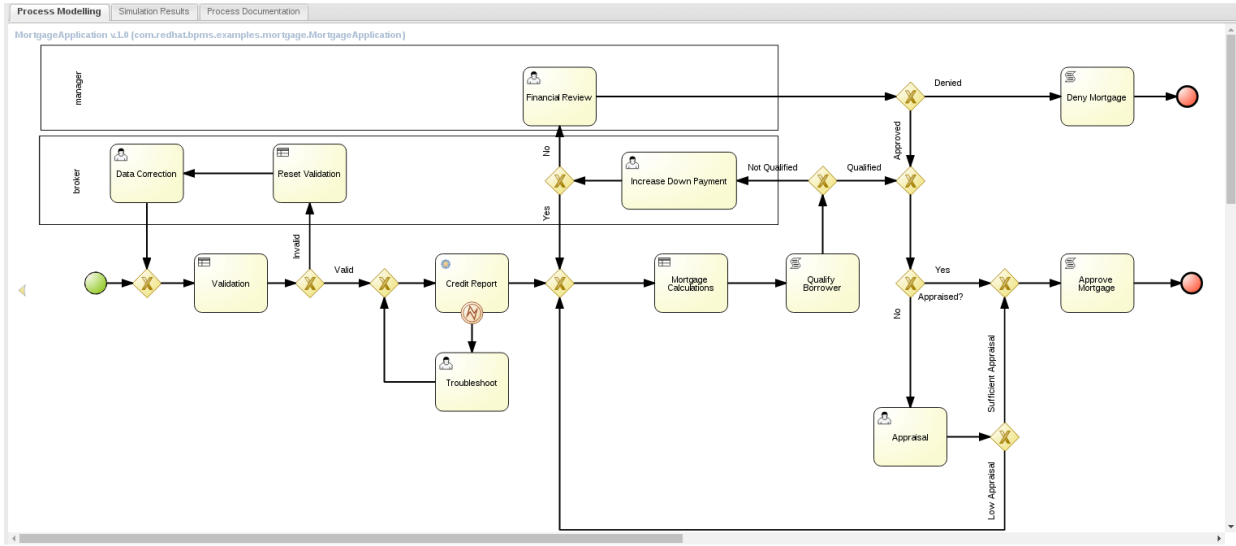


Figure 83. Final Process Model

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 JBDS 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://172.16.71.100:8003/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:

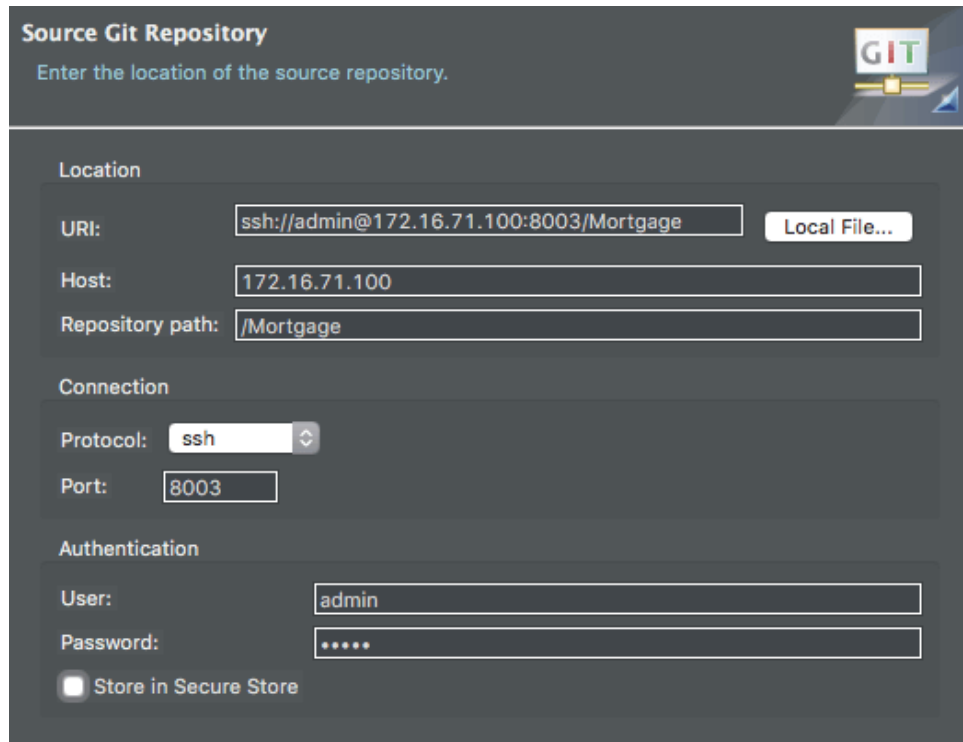


Figure 84. Source Git Import

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](#).

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 6.3 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.

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*:

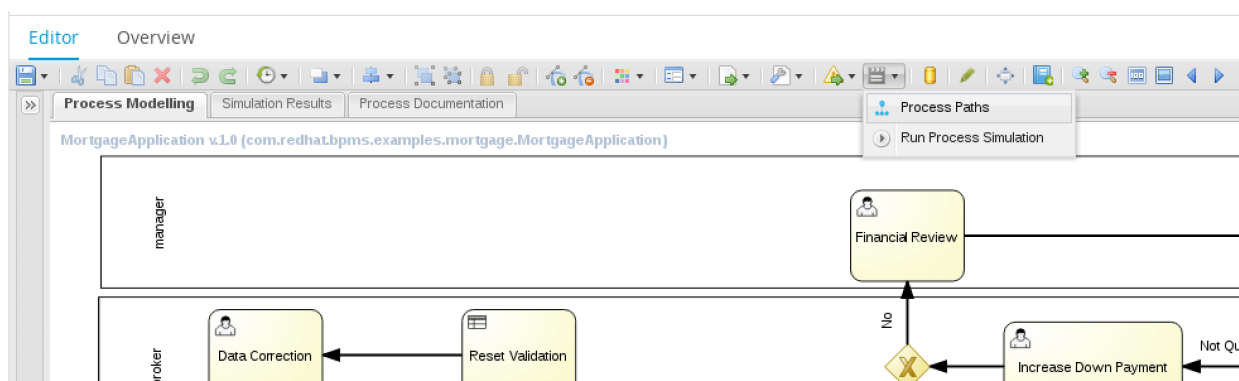


Figure 85. Process Path Button

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:

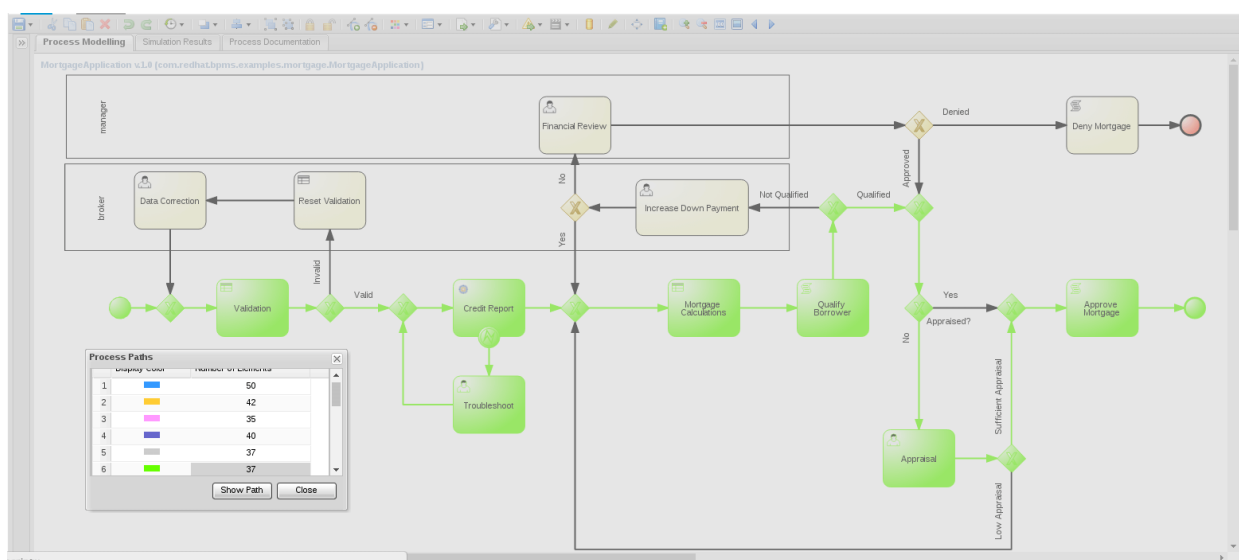


Figure 86. Process Paths

Before running process simulation, various simulation properties need to be set up for the business

process. This includes, as you’ve likely seen in process validation, a *probability* value for each sequence flow that leaves or enters a diverging/converging gateway. As logically expected, the sum of probability values for all sequence flows leaving or entering 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 affect 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:

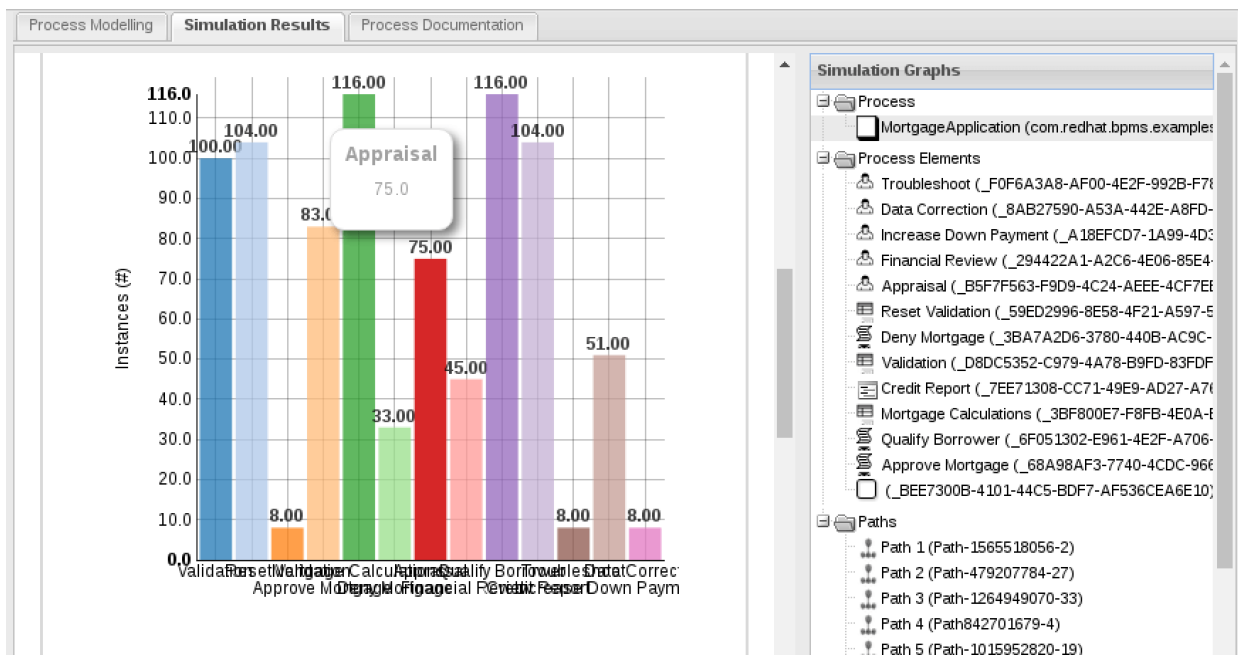


Figure 87. Process Simulation Results

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:

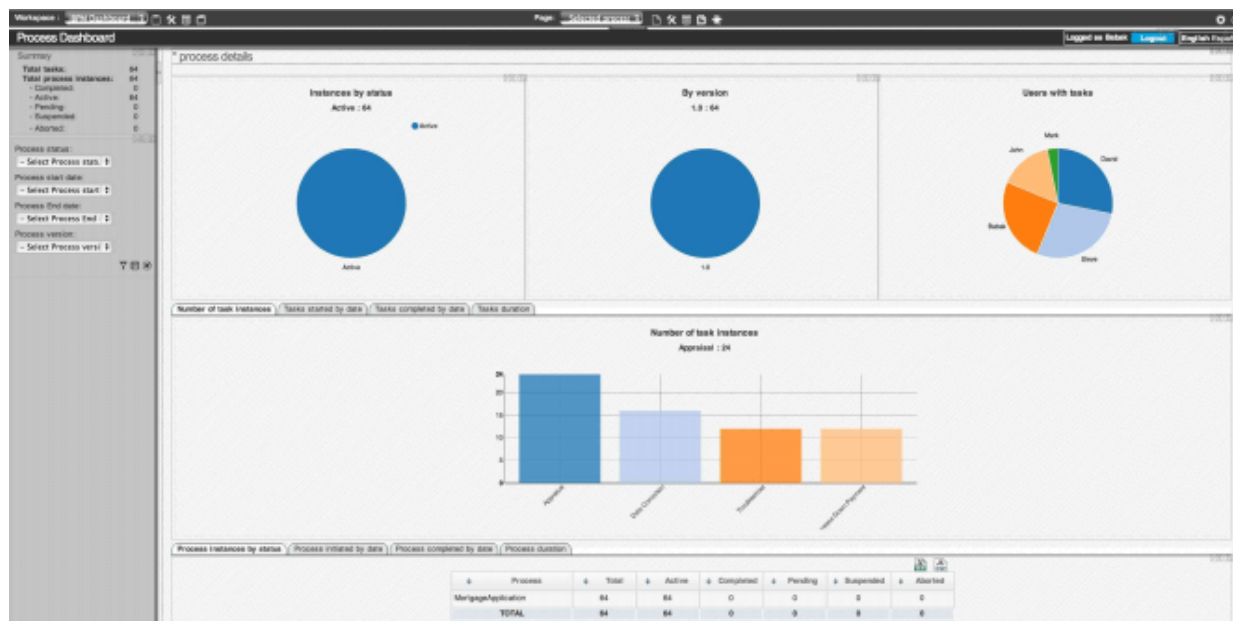


Figure 88. Dashbuilder Dashboard

6.5. Governance

Red Hat provides an implementation of the **SOA Repository Artifact Model and Protocol (S-RAMP)** specification by **OASIS**. 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.

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**.

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.

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.

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 subcontext 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.

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/businesscentral/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_”. By utilizing query parameters, the sending of objects via query is prevented, however, even the POST request doesn’t process the content provided, so it’s recommended to use KIE Server remote API or Command Execution (described below) instead. As an example of utilizing query parameters, 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/businesscentral/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.

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
```

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/businesscentral/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
```

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/businesscentral/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 RedHat documentation](#) for a full list of commands.

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 );

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/businesscentral/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/businesscentral/maven2/com/redhat/bpms/examples/mortgage/1/mortgage-1.jar
```

6.8. Session Strategy

JBoss BPM Suite supports 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 6.3.

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/businesscentral/rest/deployment/com.redhat.bpms.examples:mortgage:1/depoy
```

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

The session strategy 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/businesscentral/rest/deployment/com.redhat.bpms.examples:mortgage1/depoy?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.

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

7. Conclusion

This reference environment sets up a cluster of BPMS 6.3.0 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	Release Date	Author(s)
1.0	July 2016	Jeremy Ary & Babak Mozaffari

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
Ivo Bek	Quality Engineer	Technical Content Review
Maciej Swiderski	Principal Software Engineer	Technical Content Review