WHITEPAPER

JBOSS ENTERPRISE BRMS: BEST PRACTICES

INTRODUCTION

Business experts and application developers in organizations of any size need to be able to model, automate, measure, and improve their critical processes and policies. Red Hat® JBoss® Enterprise Business Rules Management System (BRMS) makes it possible with fully integrated business rules management, business process management (BPM), and complex event processing (CEP).[1]

This article gives insight into using JBoss Enterprise BRMS to create process- and rule-based applications that can be scaled to handle your current and future enterprise project needs. It also provides the basic information to ensure you can develop large-scale applications.

It is assumed you have a working knowledge of the JBoss BRMS product-this article does not cover the basics. A solid software architectural background is also useful, as we will discuss design decisions and ensuring project scalability in your enterprise architecture moving forward.

Finally, we will not be examining the CEP component in this article.

TABLE OF CONTENTS

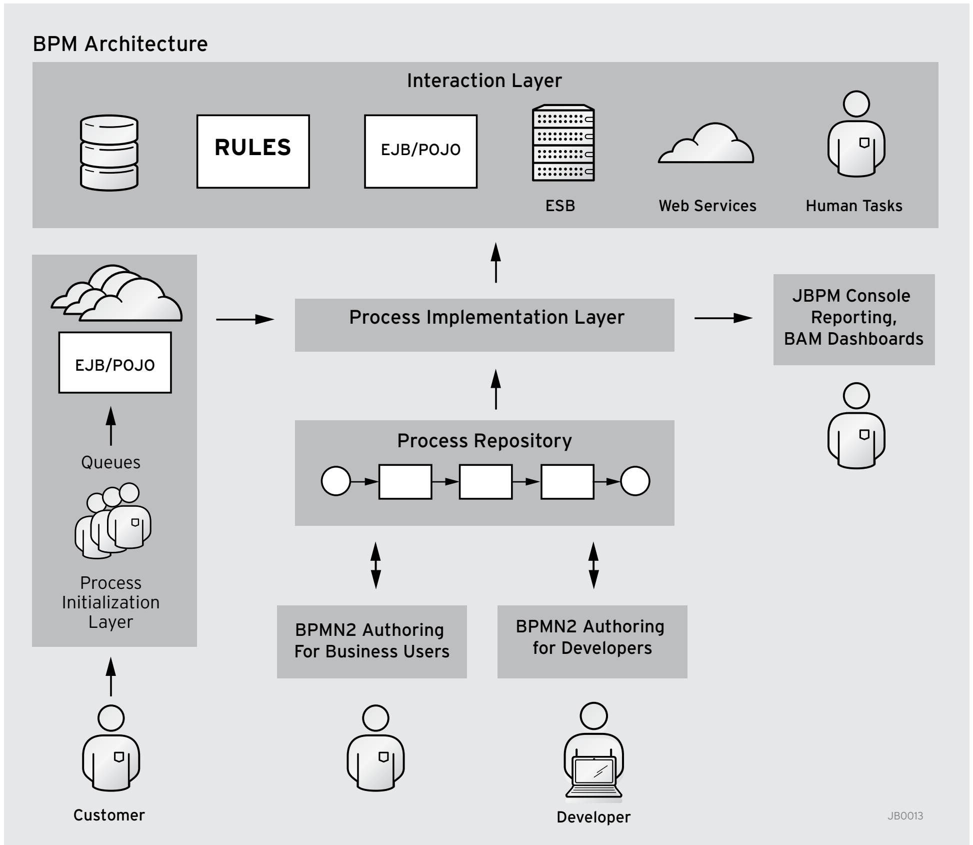
BPM Architecture
Interaction Layer
RULES
EJB/POJO
ESB
Web Services
Human Tasks
Process Implementation Layer
JBPM Console Reporting, BAM Dashboards
EJB/POJO
Queues
Process Initialization Layer
Process Repository
BPMN2 Authoring For Business Users
BPMN2 Authoring for Developers
Customer
Developer
JB0013

Illustration 1

PROCESSES

Take a closer look at a typical enterprise landscape. Peel back the layers like an onion—How we can provide BPM projects that scale well? Illustration 1 shows that there are several component layers where we will want to focus our attention:

- Initialization layer
- Implementation layer
- Interaction layer

The process initialization layer will be covered first. We present best practices for you and your customers that describe how processes can be started.

The process implementation layer is where processes are being maintained, with help from the process repository, tooling, and business users and developers that design them. Here you will also find the various implementation details, such as domain-specific extensions that cover specific node types within our projects. Finally, the process interaction layer is where your processes will connect to all manner of legacy systems, back-office systems, service layers, rules systems, and even third-party systems and services.

Initialization layer

We want to provide you the best practices for initializing processes we have seen used by large enterprises over the years. A main theme is gathering the customer, user, or system data that is needed to start your process, then simply injecting it by the startProcess call. This can be embedded in your application via the BRMS jBPM API call, make use of the RESTful service, or a standard Java web service call. No matter how you gather the data to initialize your process instances, you want to think about how you would scale out your initialization setup from the beginning. Often the initial projects are set up without much thought about the future, so certain issues have not been taken into consideration.

Customers

The customer defined here can be a person, a system, or some user that provides the initial process starting data. In Illustration 2, we provide a high-level look at how our customers provide process data that we then package up into a request to be dropped into one of the process queues. From the queues, we can then prioritize the process requests, let different mechanisms fetch them, and start a process instance with the provided request data. Here we've shown EJBs, MDBs, and clouds that represent any manner of scheduling that might be used to empty the process queues.

Queues

These queues can be as simple as database tables or as refined as message queues. They can be set up any way your project desires, such as last-in-first-out (LIFO) or first-in-first-out (FIFO). The benefit of using message queues is that you can prioritize them from your polling mechanism.

The reason for this setup is two-fold. First, you have ensured that by not directly starting the process instance from the customer interface that you have persisted the customer request. It will never be lost in route to the process engine. Second, you have the ability to prioritize future processes that might not be able to meet project requirements—for exmaple, a new process request that has to start 10 seconds after submission by the customer. If it gets put at the bottom of a queue that takes an hour to process the requests in front of it, you have a problem. By prioritizing your queues, you can adjust your polling mechanism to check the proper queues in the proper order each time.

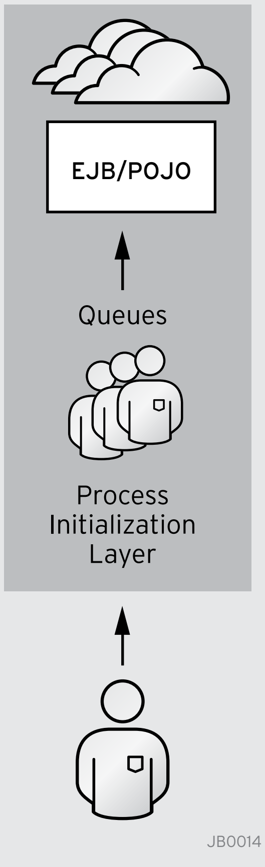

Illustration 2

Java and cloud

The Java icons shown in Illustration 2 represent any JEE mechanism you might want to use to deal with the process queues. It can be EJBs, MDBs, a scheduler you write yourself, or whatever you want to come up with to pick up process requests.

The cloud icons represent services that can be used by your software to call the final startProcess method to initialize the process instance being requested and pass it initial data. It is important to centralize this interaction with the jBPM API into a single service. This ensures minimal work if the API changes for version migrations in the future or you wish to expand in future projects to extend the service interaction with jBPM.

Implementation layer

This layer focuses on your business process designs, your implementations of custom actions in your processes, and extensions to your ways of working with your processes. The adoption of the standard BPMN2 for process design and execution has taken a lot of the trouble out of this layer of your BPM architecture. Process engines are forced to adhere and support the BPMN2 standard, which means you are limited in what you can do during the designing of processes.

There is an interesting concept within JBoss BRMS BPM that aids in building highly scalable process architectures. This is the concept of a knowledge session (KS), specifically a stateful knowledge session (SKS). This session holds your process information, including both data and an instance of your process specification. When running rules-based applications, it is normal procedure to run a single KS (note, not stateful) with all your rules and data using this single KS. With an SKS and processes, we want to leverage a single SKS per process instance. We can bundle this functionality into a single service to allow for concurrency and to facilitate our process instance life cycle management. Within this service, you can also embed eventual synchronous or asynchronous business activity monitoring (BAM) event producers as desired.[2, 3]

Interaction layer

There is much to be gained by having a good strategy for accessing business logic, back-end and back-office systems, user interfaces, other applications, third-party services, or whatever your business processes need to use to get their jobs done. Many enterprises are isolating these interactions in a service layer within a service-oriented architecture (SOA). This isolation provides for flexibility and scales nicely across all the various workloads that may be encountered. We will take a look at the BPM layer and mention just a few of these backend systems as an example of how to optimize your process projects in your enterprise.

The JBoss BRMS BPM architecture includes a separate Human Task (HT) server that runs as a service that implements the WS-HT specification. The architecture is pluggable, so you can host another server in your enterprise by exposing the WS-HT task life cycle in a service. Using a synchronous invocation model vastly simplifies the standard product implementation, which uses a HornetQ messaging system by default.

A second service, BAM, can provide great reporting scaleability. This service centralizes the BAM events, and pushes them to reliable, fast JMS queues. A separate machine can host these JMS BAM queues, processing the messages without putting load on the BPM engine and writing to a separate BAM database (optimizing with batch writing). Any clients that consume the BAM information will not be putting any load on the BPM engine either.

RULES

While BPM models sequences of actions with well-defined flows, business rules management (BRM) technologies are better suited to model actions that are loosely coupled and triggered by scenarios. An easier way of thinking about this concept is to remember the most common use case for rules implementation: decision management.

Generally speaking, decision management is the externalization and consolidation of all the rules and variables (data) involved in decision making for a given domain, as well as their management. These consolidated rules are made available to applications through decision services and can be centrally managed, audited, and maintained. They will have an independent life cycle from user applications. Decision management also provides other benefits.

Goals

Organizations adopt business rules management for many different reasons. Their application goals might include:

Decision automation: The first and most obvious benefit is the automation of decisions in the business application. Here we are talking about day-to-day, operational decisions that take the bulk of the time in any business environment. We are not referring to long-term or strategic decisions that require human intervention. The reality is that the vast majority of operational decisions in a business can be automated, thereby lowering response times and improving business performance. It also allows improved quality, consistency, and the ability to audit decision processes over time.

Expressiveness and visibility: Business rules engines (BRE) usually provide higher level metaphors and higher level languages for the authoring of rules. By using these higher level representations, rules become more concise, which makes it easier for business users to understand and verify. It also empowers business users to author the rules themselves.

Performance and scalability: BRE have specialized algorithms for the compilation and execution of rules that outperform hand-coded rules by automatically optimizing them. JBoss BRMS can efficiently execute and scale to hundreds of thousands of rules.

Centralization of knowledge: By externalizing and consolidating rules, businesses can ensure that the rules are correctly implemented and are consistent among all applications that use them. This architecture also promotes a clear separation between logic, data, and tasks—allowing the enterprise to improve agility and time-to-market. Finally, by centralizing business knowledge, it can be audited for compliance and optimization.

BRE-enabled applications

Developing BRE-enabled applications is not so different than developing traditional applications. However, there are some best practices that can be followed in order to maximize the benefits that BRMS tools provide. The next section is a list—in no particular order—of some of these best practices, grouped under architectural and authoring practices.

Architectural

Knowledge base partitioning

A knowledge base usually will contain assets such as rules, processes, and domain models that are related to one subject, business entity, or unit of work. Understanding how to partition these assets in knowledge bases can have a huge impact on the overall solution. BRMS tools are better at optimizing sets of rules than they are at optimizing individual rules. The larger the rule set, the better the results will be when compared to the same set of rules split among multiple rule sets.

On the other hand, increasing the rule set by including non-related rules has the opposite effect, as the engine will be unable to optimize unrelated rules. The application must still pay for the overhead of the additional logic. The first best practice is then to partition the knowledge bases by deploying only the related rules into a single knowledge base. Avoid monolithic knowledge bases, as well as too fine-grained knowledge bases.

Knowledge session partitioning

The creation of knowledge sessions is very inexpensive with regards to performance. BRMS systems typically scale better when increasing the number of rules and scale worse when increasing the volume of data (facts). We can therefore infer that the smaller the knowledge sessions are, the better the overall performance of the system will be.

Individual sessions can be run in parallel, so a system with many sessions will scale better on hardware with multiple processors. At the same time, we should minimize the fragmentation of data or facts, so we want to include only the related facts in the same session with the related rules. This typically comprises the facts relative to a transaction, service, or unit of work. When creating a session, it is more desirable to add all the facts to the session in a batch and then fire the rules than it is to add individual facts and fire the rules for each of them.

Domain model design

A BRE is in many ways similar to a database, from the underlying relational algorithms to the optimizations like data indexing. It is not a surprise that many of the best practices that are documented for the use of databases also apply to BRE. One of the most important is careful design of the domain model. The quality of the domain model is directly proportional to the performance and maintainability of the rules. A badly designed domain model not only affect the runtime of the engine, but also increase time and cost. Rules in a poor domain will be more complex to author and harder to maintain over time.

A good domain model represents the relationships between the multiple entities in the simplest possible way. Flatter models usually help by making constraints easier to write while small entities (entities with few attributes) prevent loops.

Rules authoring

Don't try to micro-manage

Rules should execute actions based on scenarios. Scenarios are the conditions of the rules. By following this simple principle, rules remain loosely coupled and allow rule authors to manage them individually. Rule engines further optimize the decoupled rules. Use conflict resolution strategies like salience, agenda-groups, or rule-flows only to orchestrate sets of rules, never for individual rules.

Don’t overload rules

Each rule should describe a mapping between one scenario and one list of actions. Don’t try to overload the rules with multiple scenarios, as it will make long-term maintenance harder. It also increases the complexity of testing and unnecessarily ties the scenarios to each other. Use the engine’s inference and chaining capabilities to model complex scenarios by decomposing it into multiple rules. The engine will share any common conditions between scenarios, so there is no performance penalty for doing so. For example:

```
rule “1 – Teenagers and Elders get Discount”
when
    Person age is between 16 and 18 or Person age is greater or equal to 65
then
    Assign 25% ticket discount
end

rule “2 – Elders can buy tickets in area A”
when
    Person age is greater or equal to 65
then
    Allow sales of area A tickets
end
```

The above rules are overloaded. They define in the same rules (a) the policies for what a teenager or elder is, and (b) the actual actions that should be taken for those classes of people. Pretend that the company had 1,000 rules that apply to elders. In each rule, it would repeat the condition “Person age is greater or equal to 65” to check for elders.

Now imagine that the company policy for elders, or the government law about it, changes and a person with age 60+ is now considered an elder. This simple policy change would require a change in all of the 1,000 existing rules, not to mention test scenarios, reports, etc. A much better way of authoring the same rules would be to have one rule defining what an elder is, another defining what a teenager is, and then the other 1,000 rules just using the inferred data. For example:

```
rule “0.a – Teenagers are 16-18” rule “0.b – Elders are older than 65”
when
    Person age is between 16 and 18
then
    Assert: the person is a Teenager
end

rule “0.b – Elders are older than 65”
when
    Person is older than 65
then
    Assert: the person is an Elder
end
rule “1 – Teenagers and Elders get discount”
when
    Teenager or Elder
then
    Assign 25% ticket discount
end
```

When authored like this, the user is leveraging the inference capabilities of the engine while making the rules simpler to understand and maintain. Also, the same change of policy for elders (from age 65 to age 60) would only affect one rule among the 1,000 rules in the example, reducing costs and complexity.

Control facts are inefficient

Control facts are facts introduced in the domain and used in the rules for the sole purpose of explicitly controlling the execution of rules. They are arbitrary, don't represent any entity in the domain, and usually are the first condition in a rule. Control facts are heavily used in engines that don't include the powerful, expressive conflict resolution strategies that JBoss BRMS has. Control facts also have many drawbacks: They lead to micro-control of rule executions, they cause massive bursts of work with unnecessary rule activations and cancellations. They degrade visibility and expressiveness of rules, making it harder for other users to understand as well as create dependencies between rules. Control facts are inefficient that should be avoided as a general best practice. Having said that, there is only one use case where control facts are acceptable, and that is to prevent an expensive join operation that should not happen until a given condition is met.

The right tool for the right job

JBoss BRMS has many advanced features that help users and rule authors better model their business. For instance, if one needs to query the session for data in order to make a decision or to return data to the application, then a user should use queries instead of rules. Queries are like rules, but they are always invoked by name, never execute actions, and always return data. Rules, on the other hand, are always executed by the engine (can't be invoked), should always execute actions when they match, and never return data.

Another feature that JBoss BRMS provides is the declarative models—i.e., fact types declared and defined as part of the knowledge base. For example:

```
declare Person
    name : String
    age : int
end
```

Declarative models are a great way to develop quick prototypes and to model auxiliary fact types that are used only by rules, not by an application. JBoss BRMS integrates natively with domain models developed in plain old Java objects (POJOs). The use of POJOs simplifies application integration and testing, and should be preferred whenever rules and application use the same domain entities.

REFERENCES

1 JBoss Business Rules Management System,
http://www.redhat.com/products/jbossenterprisemiddleware/business-rules.

2 JBoss BRMS – Adding on JasperReports for Reporting,
http://www.schabell.org/2012/08/jboss-brms-53-adding-on-jasperreports.html.

3 JBoss BRMS – Adding on Business Activity Monitoring (BAM) Reporting,
http://www.schabell.org/2012/07/jboss-brms-53-adding-on-business.html.