

JBoss Rules Administration For Web Applications

Problem

JBoss Rules is a great rules engine that warrants enterprise usage. Unfortunately, JBoss Rules doesn't provide an effective rules management mechanism. Many development groups struggle with the following:

- How should rules be version controlled?
- How should rules be deployed? Ideally, rule deployment will not necessarily mandate application re-deployment. It would be nice to GUI enable rules management/deployment.
- How should rules be stored for application usage? Caching a Rule Base makes sense.
- How does one test a rule set? This is important because rules will reference/manipulate POJO attributes. If a rule in the rule set has a typo or references a POJO attribute that isn't currently deployed, the rule will not compile. In this scenario, testing a list of DRLs prior to deployment makes sense.
- Provide a solution that works outside of JBoss Application Server. So, certain JBoss specific tricks are off limits, like packaging this solution as a SAR(Service archive).

Of course, resolving these issues and many others are on the JBoss Rules development road map, but the show stopper elements listed above can be addressed pragmatically today.

Assumptions

By following the KISS(Keep It Simple Stupid) model, we can leverage robust rules management right here, right now. Having said that, there are a few key assumptions that are required to deliver the goods with this approach quickly, efficiently and effectively.

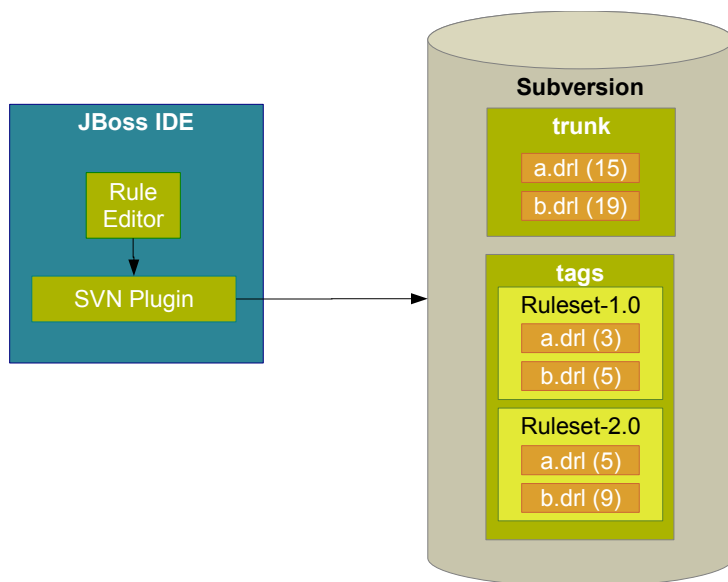
- The application is assumed to be a web application built using the J2EE standard JSF(Java Server Faces) technology. But, our solution could easily be converted to a simple servlet filter that works with Struts, Spring Web MVC, Tapestry, etc...
- Subversion is chosen as the source control system for this solution. That's because it's better than CVS, it's easy to setup, open source, and there are good Java APIs to interface with the source control system.
- A simple rule administration console will meet our needs, and it's acceptable for this console to live in our application WAR file. This last assumption sounds more like a hack than an enterprise grade solution, but this approach provides a great deal of flexibility, adds minimal code to the WAR itself and is completely self contained.

This approach has its limitations, but since this is a stop-gap solution there's no need to over engineer. Here's what we're avoiding.

- We don't want to re-write SVN, CVS or any other version control system. Let's just leverage what's freely available and known to work.
- Spend a lot of time developing features that fall outside of the 80/20 Rule. Remember, we're attempting to solve the "Problem" line items and nothing else. So, while rule re-use and rule-level version support might be "nice to have", they are by no means inhibiting our adoption of JBoss Rules.
- A one-size-fits all solution. It's likely that we'll mandate a web development framework for application development, so tying our solution to a particular web technology isn't a big deal.

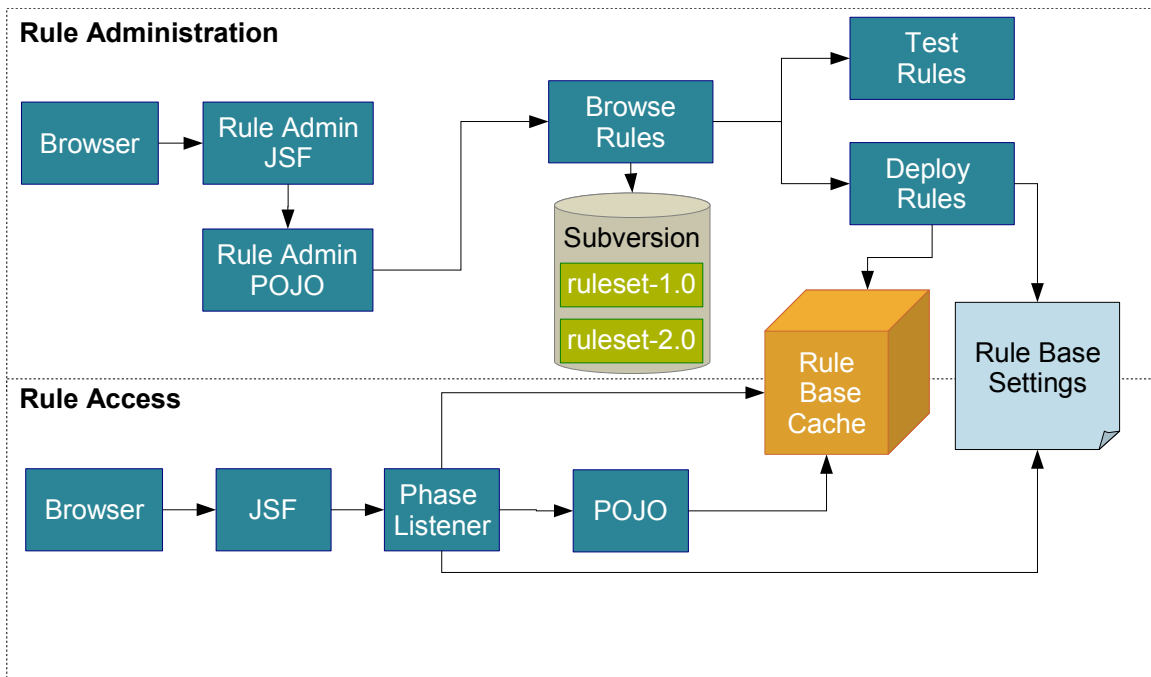
Rule Authoring and Version Control

The diagram below denotes the rule authoring and version control process for this solution. The solution leverages existing tools for rule authoring and an open source source control system.



JBoss IDE 2.0 contains the JBoss Rules plugins needed to author/debug rules. The rules are stored in .DRL files and these files should be version controlled. Subclipse is available from <http://subclipse.tigris.org/> and should be used as the source control client. Two different streams of source control are leveraged in this solution. The “trunk” repository path is for the active development stream. The “tags” repository path is for any grouping of one or more rules. Since, DRL files will likely be deployed in groups, it makes sense to group the DRL files into a version controlled rule set. That way, the Rule Administrator can easily switch out sets of DRL files.

Rule Engine Administration and Rule Engine Access



Rule Administration

For rule administration, a single page console is leveraged. This console provides rule browsing from the Subversion “tags” repository location. Remember, a Subversion tag in this solution is just a group of DRL files. One or more rules can be selected and viewed via the console. If desired, rules can be tested prior to deployment. Deploying rules will create a cached RuleBase instance.

Rule Access

For rule access, the application simply calls a POJO method that gets a pre-loaded WorkingMemory instance from the RuleBase, asserts objects and fireAllRules(). There is a PhaseListener that checks for the existence of a cached RuleBase. If no RuleBase exists, the PhaseListener will load a new RuleBase from a properties file that contains the last loaded rule list. This way, JBoss restarts will not require manual rule re-deployment.

Implementation Instructions

Okay, this is where the rubber meets the road. There's a little code, not a lot of code but then again this is a KISS solution. This implementation will involve adding some components to a WAR that needs rules engine administration and access.

Configure Subversion with HTTP Access

An HTTP accessible Subversion repository is needed. The recommended approach is to:

1. Install Subversion 1.4
2. Install Apache 2.0.x
3. Add the `mod_dav.so` and `mod_dav_svn.so` libraries to `$APACHE_HOME/modules`
4. Add the appropriate entries to `$APACHE_HOME/conf/http.conf`, so clients can access Subversion via HTTP.
5. Create a “trunk” and “tags” folder in Subversion. “trunk” is the development stream and “tags” represent the labeled rule sets that the Rule Admin GUI will use.

More info on how to install Subversion with HTTP support can be found at: <http://svnbook.red-bean.com/nightly/en/svn-book.html>.

JARS – Add all JARS to (`WEB-INF/lib`)

1. Add the drools jars and drools dependencies
2. Add `drools-rule-admin.jar`
3. Add `javasvn.jar`, `javasvn-cli.jar` and `javasvn-javahl.jar`

TIP: The reference `rules-admin-console` application contains all required JARS, so copying the `rule-admin-console` JARs in `WEB-INF/lib` will suffice.

Add `web.xml` Entries

The following entries must be added to the `web.xml` file. The entries provide the “`RulesAdmin`” managed bean with the location of the properties file that stores the Working Memory Pool settings.

```
<context-param>
  <param-name>rules.properties.url</param-name>
  <param-value>http://localhost/rules-config/rules.properties</param-value>
</context-param>
```

The “`rules.properties.url`” entry is for accessing the rule engine settings via URL and provides remote read-only access to the rules-settings and is handy in several server deployments.

```
<context-param>
  <param-name>rules.properties.file</param-name>
  <param-value>${APACHE_HOME}/htdocs/rules-config/rules.config.properties</param-value>
</context-param>
```

The “`rules.properties.file`” entry is for accessing the rule engine settings via a file system path reference and provides local write access to the rules-settings. In several server deployments, a single server should be responsible for changing rules engine settings.

Add faces-config.xml Entries

The following entries must be added to your faces-config.xml file. They provide access to the RulesAdmin POJO, PhaseListener and rules-admin console JSPs.

```
<managed-bean>
  <managed-bean-name>grind</managed-bean-name>
  <managed-bean-class>com.ruleadmin.ui.WorkingMemoryGrinder</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
</managed-bean>
```

The “*grind*” bean is optional and provides a way for you to stress/load test the cached RuleBase, versus loading a new RuleBase instance from the file system on every request. This bean is also useful in identifying how large the connection pool should be, given a pre-specified concurrent usage need.

```
<managed-bean>
  <managed-bean-name>rulesAdmin</managed-bean-name>
  <managed-bean-class>com.ruleadmin.ui.RulesAdmin</managed-bean-class>
  <managed-bean-scope>application</managed-bean-scope>
</managed-bean>
```

The “*rulesAdmin*” bean is required. But, if strict change management rules are in place, just add this bean and omit the JSPs. This bean needs to have APPLICATION scope, because the RuleBase is stored in this bean and it's an application wide, shared resource.

```
<navigation-rule>
  <navigation-case>
    <from-outcome>rule-admin-form</from-outcome>
    <to-view-id>/rule-admin-form</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>rule-admin-error</from-outcome>
    <to-view-id>/rule-admin-form-error</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>grinder-page</from-outcome>
    <to-view-id>/grinder-page</to-view-id>
  </navigation-case>
</navigation-rule>
```

Several navigation rules are needed. The “*rule-admin-form*” references the rule administration console JSP, “*rule-admin-error*” references the rule administration console error page, and the “*grinder-page*” a the grinder friendly JSP for creating load-stress tests against the rules engine.

```
<lifecycle>
  <phase-listener>com.ruleadmin.ui.RulesAdminListener</phase-listener>
</lifecycle>
```

The last piece of the puzzle is the PhaseListener. This entry must be added if you want the application to check for a RuleBase before requesting a WorkingMemory instance. The PhaseListener ensures that an application server restart will not require manual rules re-deployment through the console.

Add Rule Admin Console Pages

Several JSPs need to be added to the WAR. These pages represent the “*rule-admin-console*” GUI that the application will use to change rule engine settings. There's also a grinder friendly page that can be used to conduct performance tests on the RuleBase cache.

1. Add *WebContent/include/bottom-admin.jsp* and *WebContent/include/top-admin.jsp*. You can customize the look/feel of the rule admin console page with these JSPs.
2. Add *WebContent/theme/style.css* or edit *top-admin.jsp* to point to a different stylesheet.
3. Add *WebContent/rule-admin-form.jsp*, *WebContent/rule-admin-form-error.jsp* and *WebContent/grinder-page.jsp*.

Interacting with the Rules Engine

This solution makes it very easy for a POJO to communicate with the rules engine.

```
public void callRulesEngine() throws Exception {  
  
    //get working memory instance from a cached RuleBase  
    RulesUtil util = new RulesUtil();  
    WorkingMemory wm = util.getWorkingMemory();  
  
    //assert objects to WorkingMemory. The rules engine may alter these objects  
    wm.assertObject(obj1);  
    wm.assertObject(obj2);  
    wm.assertObject(obj3);  
  
    //call the rules engine and execute all rules  
    wm.fireAllRules();  
}
```

For all intensive purposes, the application should only require this level of interaction with the rules engine.

Reference Project – rule-admin-console

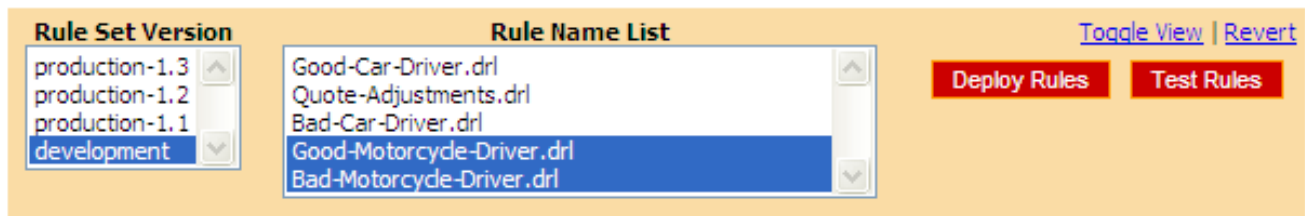
The rule-admin-console.war project contains all code needed to deploy/test rules and can be used as a shell WAR for your rules engine enabled application. There is also an extras.zip package that contains some noteworthy additions including:

- *bin* – contains the *drools-rule-admin.jar* and *test-model-pojos.jar* with some test POJOs. If this JAR isn't included in your *WEB-INF/lib* directory, the test-drl-files will not compile.
- *docs* – contains all documentation for this solution.
- *load-test* – contains sample grinder load tests/configuration files and a quick how-to guide on Grinder3.
- *test-drl-files* – can be checked into your subversion repository and used to test the rule-admin-console. These are simple files that require *test-model-pojos.jar* to compile.

To use this sample project in eclipse click *File > import > Web > War File* and point the importer to *rule-admin-console.war* file. You will need to add this project to a JBossAS 4.0.4 server instance. Lastly, you will need to add the *\$JBOSS_HOME/server/default/lib/log4j.jar* to your project's classpath because the rule-admin-console does use log4j.

Rule Administration Console Screenshot

Rule Base Successfully Reloaded



The screenshot shows the Rule Administration Console interface. On the left, the 'Rule Set Version' dropdown is set to 'development'. The 'Rule Name List' contains the following files: Good-Car-Driver.drl, Quote-Adjustments.drl, Bad-Car-Driver.drl, Good-Motorcycle-Driver.drl (highlighted), and Bad-Motorcycle-Driver.drl. On the right, there are two red buttons: 'Deploy Rules' and 'Test Rules'. Above these buttons are links for 'Toggle View' and 'Revert'.

Good-Motorcycle-Driver.drl

```
package com.regressive.quote;

import com.regressive.model.*;

rule "Good Motorcycle Driver"
    when
        quote : QuoteBean()
        driverDetail : DriverDetailBean( age > 25)
        motorcycleDetail : MotorcycleDetailBean( motorcycleCount >= 1)
    then
        quote.setPrice(100 * motorcycleDetail.getMotorcycleCount() );
        quote.addQuoteCriteria("Good motorcycle driver",
                               "For any good motorcycle driver");
    end
```

Bad-Motorcycle-Driver.drl

```
package com.regressive.quote;

import com.regressive.model.*;

rule "Bad Motorcycle Driver"
    when
        quote : QuoteBean()
        driverDetail : DriverDetailBean( age < 25)
        motorcycleDetail : MotorcycleDetailBean( motorcycleCount >= 1)
    then
        quote.setPrice(500 * motorcycleDetail.getMotorcycleCount() );
        quote.addQuoteCriteria("Bad motorcycle driver",
                               "For any bad motorcycle driver");
    end
```

This screen shot displays two DRL files that have been loaded into the cached RuleBase. The DRL files can also be tested to make sure that they compile before deployment. This console is actually interacting with the Subversion repository to display/load the DRL files into the RuleBase, so there's no way to deploy a DRL that's not checked into the version control system.

RuleBase Cache Justification

A cached RuleBase is leveraged to avoid the very costly RuleBase package process. The cache versus non-cache approaches are explained in more detail below:

Non-Cached Rules Engine Interaction

Basically, all rules are loaded from the file-system, compiled, then converted into a RuleBase that a WorkingMemory instance can use to fire rules. This process is somewhat inefficient due to the loading/compilation of DRL files from the file-system. Claiming an operation is inefficient needs proof, so here's a grinder results screen for "25" concurrent users, accessing the rules engine without the cached RuleBase.

Test	Description	Successful Tests	Errors	Mean Time	Mean Time Standard Deviation	TPS	Peak TPS	Mean Response Length	Response Bytes Per Second	Response Errors	Mean time to resolve host	Mean time to establish connection	Mean time to first byte
Test 100	Page 1	266	0	22200	8280	1.10	11.9	0.00	0.00	0	0.00	0.00	0.00
Test 101	POST gri...	266	0	22200	8290	1.10	11.9	2030	2230	0	16.3	19.1	22200
Total		532	0	22200	8280	2.20	23.8	1020	2230	0	8.13	9.53	11100

The average response time is 22.20 seconds and the standard deviation is 8.28 seconds. It is likely that a pooled solution can do much better.

Cached Rules Engine Interaction

This solution pre-loads a RuleBase instance and holds it in APPLICATION scope. Obtaining a WorkingMemory instance is wrapped by a convenient RulesUtil class. The drawback to this solution is the overhead of storing a RuleBase in memory. This solution reduces the rule engine response time considerably, but it is not without it's disadvantages. There are JVM garbage collection implications to storing a large RuleBase in memory, but the drawbacks are likely well worth the additional memory overhead. Here's a grinder results screen for "25" concurrent users.

Test	Description	Successful Tests	Errors	Mean Time	Mean Time Standard Deviation	TPS	Peak TPS	Mean Response Length	Response Bytes Per Second	Response Errors	Mean time to resolve host	Mean time to establish connection	Mean time to first byte
Test 100	Page 1	32396	0	111	301	215	285	0.00	0.00	0	0.00	0.00	0.00
Test 101	POST gri...	32396	0	107	301	215	284	2030	437000	0	0.220	0.245	101
Total		64792	0	109	301	431	569	1010	437000	0	0.110	0.123	50.6

The average response time is .11 seconds and the standard deviation is .30 seconds. This is more in-line with the response time requirements of a typical web application.

Additional Considerations

As mentioned earlier, this solution stores the RuleBase in APPLICATION scope.

This is a good use of APPLICATION scope, and the solution leverages APPLICATION scope instead of JBossCache and a .SAR. If a JBoss only solution is acceptable, a JBossCache enabled .SAR(Service Archive) should be used to store the RuleBase. The .SAR based solution will provide additional flexibility, most notably enable the rules engine to be treated as a shared service. Of course, this will require additional code and the sharing of rule bases across multiple applications does have architectural implications.