Chapter 13
Deploying with JRules

Target audience

- *Application architect, software architect, developer*

In this chapter you will learn

- *How rulesets are packaged as part of a RuleApp*
- *What are the ruleset versioning capabilities*
- *How to manage a RuleApp in Rule Team Server and in Rule Execution Server*
- *How to use the Rule Engine API, the JSR 94 or the Rule Execution Server rule session API to integrate rule engine processing into your application*
- *How to use a rule engine using JMS deployment*
- *The concept of Transparent Decision Service*
- *How to identify which rules executed using the Decision Warehouse capability*
- *How to develop queries to select the rules you want to have in your ruleset*

Key points

- *The main deployment unit when using the rule execution server is the RuleApp, which can be created and managed by a business user within rule team server.*
- *JRules offers a very flexible API to integrate the rule engine into the business application leveraging JEE or J2SE deployment model.*
- *Rule execution server is simple to use and delivers the rich set of features to manage a ruleset in production and scale vertically.*
- *Business users use Rule Team Server to author but also deploy rules to the different RES.*
- *Rulesets can be exposed as services, but for most business application deployed in SOA a decision service is part of reusable business services therefore better deigned with a meaningful interface and implemented using Java using the RES API.*

J. Boyer and H. Mili, *Agile Business Rule Development*,
DOI 10.1007/978-3-642-19041-4_13, © Springer-Verlag Berlin Heidelberg 2011

13.1 Introduction

In this chapter, we present the different deployment possibilities offered by IBM WebSphere ILOG JRules. We first go over a quick review of the concepts of operation (see also Sect. 8.2 for more details), with an emphasis on RuleApps, which are the deployable artifacts to the rule execution server. In Sect. 13.3, we talk about deploying rule application using the rule engine API. In particular, we review the classes for rule engine, rulesets, and various utilities that support rule execution tracing and debugging. In Sect. 13.4, we describe rule deployment using JRules' *Rule Execution Server* (RES), whereby rulesets are deployed as services that can be invoked by applications. Different RES configurations are discussed, depending on the needs of the business application. We review pure Java integration, JMS, and SCA. In Sect. 13.5, we address the deployment of Rule Team Server. We conclude in Sect. 13.6.

13.2 Reminder on the Concepts of Operation

JRules has two authoring environments from where we can deploy rules: Rule Studio and Rule Team Server (RTS). As seen in Chap. 5, developers use Rule Studio to build the project structure, to develop the Business Object Model, to organize the flow of execution, and to implement rules. From Rule Studio, they also define the different configurations they may want to use for the ruleset deployment. Basically, we can deploy rules to a rule engine embedded in a business application or to a managed rule execution environment, which offers a richer set of application management features. The choices between the two depend on the application requirements. In SOA, an architect may leverage the JEE container to support service binding, security, transaction support, pool for connections to the different data sources, and so forth. Therefore, the natural deployment leverages the Rule Execution Server (RES) component, which is a Java Connector Architecture implementation. The alternative is to use an embedded deployment in which we package the engine jars with the application and we deploy and execute the ruleset (as ruleset archive) on a single JVM. This last approach uses the lower level API to access the rule engine and the ruleset archive: a packaging for the rules (see Sect. 13.3.1 later). There is a third deployment using the RES in J2SE, which we present in Sect. 13.4.1.

Figure 13.1 illustrates the tasks a developer has to perform within Rule Studio to prepare for an embedded integration.

The right side of Fig. 13.1 presents a potential application packaging, including the domain data model, the application logic (business service interfaces and implementations), the rule engine, and the ruleset. The domain data model is most likely a Java model accessed through a data access object layer, so it can be packaged as a standalone reusable jar: dom.jar. The business logic code uses the rule engine and the ruleset API, which is in the jrules-engine.jar.

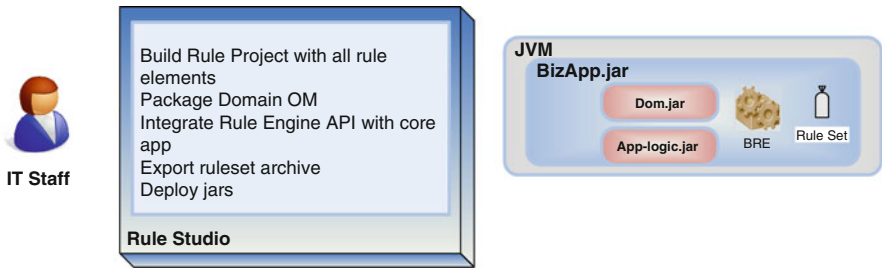

Fig. 13.1 Packaging a rule application

The embedded integration is not recommended when the requirements for management of rules and rulesets are becoming crucial. IT developers will most likely re-implement most of the features supported by the rule execution server, like database persistence for the ruleset, management stack to control the life cycle of the ruleset, rule processing statistics and logging mechanism, transaction support, security control, engines pooling, etc. When the application needs to support multiple rulesets, parallel execution, or is processing data within a transactional context, we must leverage the services of a JEE container. Rule Execution Server, deployed on an application server as a Java Connector Architecture[1] resource adapter (RA), supports transaction management, security controls, and rule engine pooling.

In RES, there are multiple patterns to invoke a ruleset: using java object (POJO), local or remote EJBs, web service protocols, or as JMS listener using a Message Driven Bean. We will detail the RES subcomponents and the activities to deploy a ruleset in Sect. 13.4.

At the lowest level, a Ruleset is packaged as a rule archive which is a jar including rule files,[2] meta data files such as the reference to BOM, the ruleset signature description, and the exported rule properties. If we use the rule engine API we have to parse the rule archive before calling the rule execution. This parsing has to be performed only at the application initialization. When using the Rule Execution Server, the rule archive is packaged within a RuleApp and transparently parsed at the first call for rule execution.

A RuleApp contains one or more rulesets. In Fig. 13.2, the ClaimProcessing-RuleApp has four rulesets with rules and one with the BOM entries.

In Rule Studio, RuleApps are managed inside projects. A RuleApp project includes XML descriptors to describe the rule project dependencies, the ruleset path, and a list of ruleset archive files. As a good design approach, a RuleApp should include rulesets that share the same domain object model and are in the

[1]Java Connector Architecture: http://java.sun.com/j2ee/connector/reference/industrysupport/index.html.

[2]One irl (ILOG Rule Language) file per rule. The format is a text file with IRL syntax.

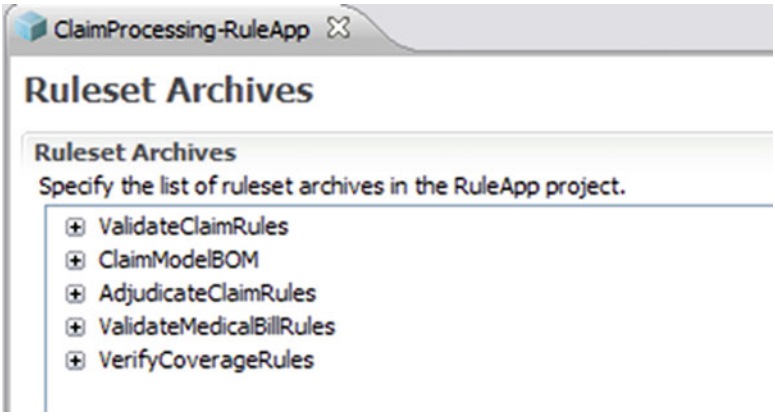

Fig. 13.2 RuleApp and Ruleset archives

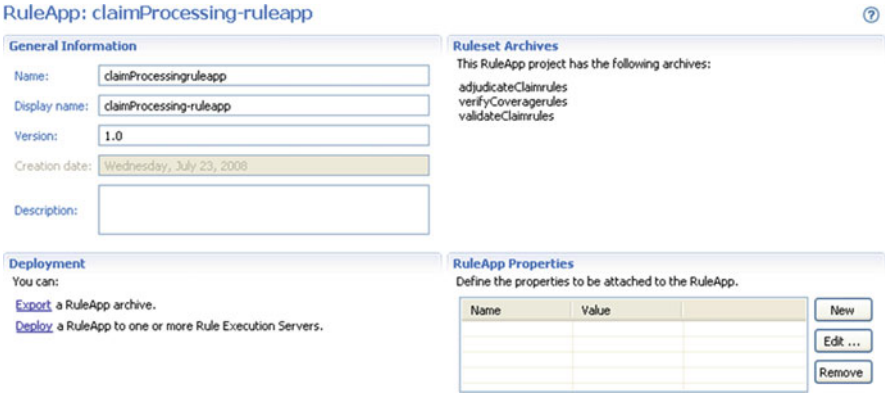

Fig. 13.3 RuleApp editor

same application context. From a RuleApp project, the developer can perform all the pure administration activities like versioning, deployment, adding management properties, export, and so forth. Figure 13.3 shows the Rule Studio RuleApp editor for the claim processing RuleApp, which includes three rulesets, "adjudicateClaimrules", "verifyCoveragerules", and "validateClaimrules."

Each ruleset in a RuleApp can be invoked using a ruleset path. A ruleset path includes the reference of the RuleApp name and the ruleset name inside the RuleApp. The following ruleset path/ClaimProcessing-RuleApp/validateClaimrules refers to the current version of both RuleApp and ruleset. A path such as/ClaimProcessing-RuleApp/**1.0**/validateClaimrules/**2.1** references specific versions

Fig. 13.4 Rule App management in RTS

for both elements. Using the ruleset path it is possible to use different versions of a ruleset within the calling client code. The API supports opening a session with the rule execution server and to specify the ruleset path to use.

The following simple practice can be applied to control the version number of the ruleset:

- Increase the X of X.Y version number for each major release of the ruleset
- Increase the Y of X.Y version number to manage subversion deployment

If we change the business logic in any way in one of our rule projects, we have to upgrade the RuleApp archive to take the modifications into account. It is possible for a business analyst using Rule Team Server to author rules and to deploy RuleApps directly to a Rule Execution Server (Using the Configure Tab). In RTS, a business user can create baseline, which can be seen as a tag applied to each rule to deploy, and then he can deploy the RuleApp to the target execution environment. Figure 13.4 presents RuleApps management screen with the set of buttons to drive the deployment of a RuleApp.

It is important to use the correct RuleApp and ruleset names when defining the RuleApp in RTS: they have to be the same as the ones specified in the ruleset path as seen in previous section. If not the rule execution will not find the rulesets.

Finally, the Rule Execution Server has also a web interface used by administrator to manage the deployed ruleset archives and RuleApps. It is also possible to perform basic monitoring, to view execution statistics, and to deploy, change, and manage business rules without stopping the server. The information provided is rich as we can see the rules deployed in ILOG Rule Language format and the rules that were executed for given input data.[3] The central panel displays the content and status of a ruleset (Fig. 13.5).

We will detail the RES Console capabilities when detailing the new decision warehouse function in Sect. 13.4.4.

[3] A new capability called Decision Warehouse, see product documentation at http://publib.boulder.ibm.com/infocenter/brjrules/v7r1/index.jsp.

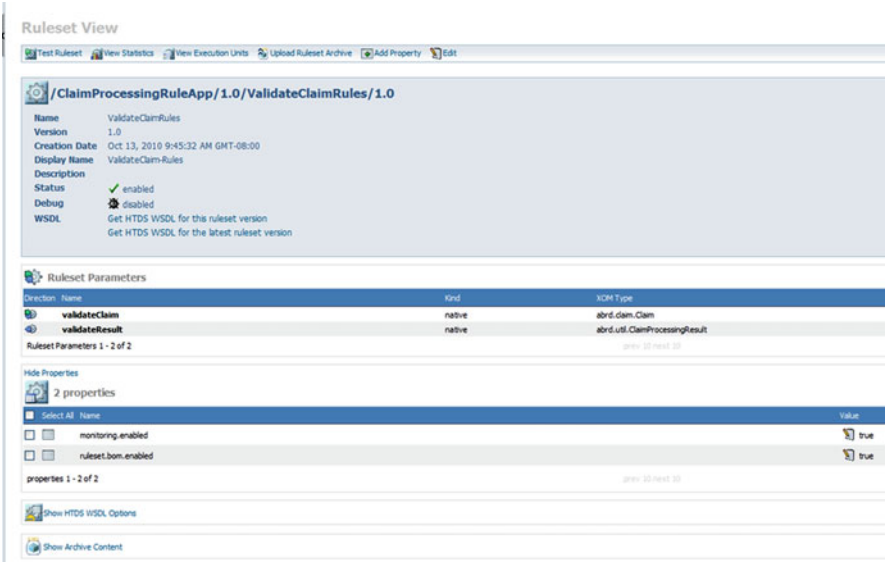

Fig. 13.5 Ruleset view as deployed in RES

13.3 Integration with JRules Engine

In this section, we cover the integration of JRules engine using the different API offered, engine and ruleset API in Sect. 13.3.1; JSR94 in Sect. 13.3.2. Using this kind of integration may be a realistic use case when none of the out of the box features supported by the rule execution server are needed by the application.

13.3.1 Deploying with the Rule Engine API

When we execute rules using the engine and ruleset API, we deploy and execute a ruleset archive in a single Java Virtual Machine. The rule engine API is packaged as jar and added to the classpath of the application. The client code using the rule engine needs to load the ruleset archive from its data source, to parse it, to instantiate a rule engine, to prepare the data as ruleset parameters or as facts inserted into the working memory, to execute the rules, and finally to process the result.

As it takes some time to build the RETE Network and other internal objects, the operation of loading and parsing the ruleset should be done only the first time the application is started. The basic API used includes the class `IlrContext` for the rule engine, `IlrRuleset` for the ruleset, plus some helper classes to load and parse ruleset archive. The following code sample presents a class called ProcessClaimImplWithJrules with a constructor preparing the rulesets by loading the archive, parsing it, and creating the `IlrRuleset` object.

```
public class ProcessClaimImplWithJrules implements ProcessClaim {
   // ruleset name…
   protected String validateClaimRsName = "validateClaim-rules.jar";
   protected static IlrRuleset validateClaimRuleset;

// constructor
public ProcessClaimImplWithJrules(){
   JarInputStream is;
   try {
     is = new JarInputStream(new FileInputStream( new File( validate-
ClaimRsName)));
      // prepare a Ruleset archive loader
      IlrRulesetArchiveLoader rulesetloader = new IlrJarArchiveLoader(is);
      // then a parser
      IlrRulesetArchiveParser rulesetparser = new IlrRulesetArchiveParser();
     validateClaimRuleset = new IlrRuleset();
     rulesetparser.setRuleset(validateClaimRuleset);
     // finally parse to create the ruleset
     rulesetparser.parseArchive(rulesetloader);
        …
```

Parsing the ruleset archive can generate errors; it is therefore recommended to stop if an error occurs. Building `IlrRuleset` may take time so we need to avoid creating it at each rule execution call, for example, by using a factory and static variable protected with the singleton pattern. The last part of the class supports the implementation of the business methods. It needs to get an engine instance, sets the input parameters, optionally initializes the working memory, executes the rules, and finally gets the output parameters. Below is an example of code for the validateClaim operation:

```
public Result validateClaim(Claim claim) {
// Create the engine with a reference to the ruleset
ilog.rules.engine.IlrContext context = new IlrContext(validateClaimRuleset);
// Initialize the input parameters
IlrParameterMap inputs = new IlrParameterMap();
inputs.setParameter("validateClaim", claim);
   inputs.setParameter("validateResult",new Result());
   context.setParameters(inputs);
   // Initialize the working memory
    context.insert(claim.getPolicy());
   // Execute the ruleset
   IlrParameterMap outputs = context.execute();
   // Get the result
   Result rOut=(Result)outputs.getObjectValue("validateResult");
   // Clean the context
      context.retractAll();
      context.end();
      return rOut;
    }
```

The engine API is simple and is enough to support a lot of basic applications. Other transaction heavy applications should leverage the RES API, which we will detail in Sect. 13.4. But first, let us look at how JRules is supporting JSR94.

13.3.2 JSR94: JRules Specifics

As introduced in the previous chapter, the JSR 94 offers the advantage of interacting with a rule engine without any knowledge of the underlying product API. As of this writing, the value of using JSR94 is questionable until there is an agreed format to exchange rules between different engine vendors. W3C is working on defining a standard called Rule Interchange Format or RIF.[4] For more detailed explanations of the JSR94 API, see Sect. 12.3.4.

JSR-94 delegates its processing using JRules rule sessions deployed in a Java archive file named jrules-res-jsr94.jar. The JSR-94 interface is implemented with the Rule Execution Server and not with the rule engine API, which is an additional layer added on top of the engine, in order to have a common interface for J2SE and J2EE executions.

For JRules, it is not mandatory to deploy a ruleset with the JSR-94 management API to execute this ruleset. The ruleset is already being deployed to RES and the client code uses the JSR94 run time API to load object and execute rules. The Uniform Resource Identifier (URI) used should be a valid ruleset path including ruleappname/rulesetname. This also means we need to deploy a RuleApp archive to get access to the rule execution set with JSR94.

For creating a rule execution set, the input stream has to point to a XML ruleset descriptor which looks like:

```
<?xml version="1.0" encoding="UTF-8"?>
<rule-execution-set>
   <!-- The value attribute could be a valid path or a valid URL on a RuleApp ar-
chive file -->
    <location value="res_data/validateClaim-rules.jar"/>
</rule-execution-set>
```

It defines the path to your RuleApp archive file, generated by Rule Studio. The client code needs to create a service provider, which in the case of JRules should use the following URL:

```
   // Get the rule service provider from the provider manager
Class.forName(IlrRuleServiceProvider.class.getName());
RuleServiceProvider serviceProvider = RuleServiceProviderManager.getRuleServicePro
vider("ilog.rules.bres.jsr94");
```

[4]See detail at W3C URL: http://www.w3.org/2005/rules/wiki/RIF_Working_Group.

13.3.3 Monitoring and Tracing Rule Execution

As part of the integration there is a need to be able to trace and monitor execution of the rules. RES supports monitoring of rule execution out of the box. We will detail this in Sect. 13.4. When using the low-level API, it is still possible to attach a monitoring tool to get events from the rule engine when it processes business data. We can attach a notification observer using the engine API connectTool(engine-Observer).

```
// create a rule engine - with a ruleset
IlrContext context = new IlrContext(validateClaimRuleset);
//add the observer
context.connectTool(new EngineObserver());
```

The observer is an extension of the ilog.rules.engine.IlrToolAdapter class or an implementation of the IlrTool interface; some callback methods can be overridden to trace the execution. For example, the method notifyBeginInstance is invoked in RETE mode when the engine executes a rule, so we can log the name of the rule.

```
public class EngineObserver extends IlrToolAdapter {
….
    public void notifyBeginInstance(IlrRuleInstance instance) {
                logger.info(instance.getRuleName());
    }
```

In production environment, logging has to be designed with care. We will most likely prepare the minimum information during the rule processing and use an asynchronous call to send the message to a messaging queue for future processing done by a listener. Such heavy processing include saving the information to a database. The goal is to avoid impacting the performance of the rule engine. Asynchronous calls do not block the caller and make the receiver take care of the logging and of the persisting of the events into a data source.

13.3.4 Resource Pooling

As discussed in the previous chapter, it is possible to pool rule engines for parallel processing. JRules offers this capability out of the box using JCA connection pooling inside the RES. Even in a J2SE deployment, the JRules implementation (in jar jrules-res-execution.jar) of the JCA API is using engine pooling. The pool size can be configured using a XML descriptor file named ra.xml.

13.4 Deploying with the Rule Execution Server

In this section, we review the most important capabilities of RES in the context of application integration and ruleset deployment. We start by presenting RES architecture as a JCA resource adapter, detailing the rule engine pooling and the ruleset deployment. Then we review the RES session API to use to call for rule execution in application server or a J2SE application. We detailed the JMS deployment in Sect. 13.4.2 and the SCA deployment in Sect. 13.4.3. We present in Sect. 13.4.5 the concept of transparent decision service (TDS) as the simplest way to demonstrate smooth integration. Finally in Sect. 13.4.4 we present the decision warehouse feature, used to monitor the rule execution with RES.

RES can be deployed as a centralized service, executing multiple rulesets on the requests of multiple clients. It can also be packaged within a unique business application (WAR or EAR) and only visible by the code of this application. This packaging does not mean we cannot reuse rulesets, in fact the business services can be reused and are callable using Web Service, SCA, JMS, local Java call, or RMI depending on the communication choices. RES is based around a modular architecture that can be deployed as a set of Java Plain Old Java Objects (POJOs) running in a J2SE JVM, hosted using Apache Tomcat, or run within a full Java EE compliant application server.

RES is a resource adapter of the Java Connector Architecture. JCA is designed to provide a unified way to access external resources from Enterprise Information System (EIS), instead of having proprietary adapters for each external system. JCA enables an EIS vendor to provide a standard resource adapter for its EIS (Fig. 13.6). By plugging into an application server, the resource adapter collaborates with the

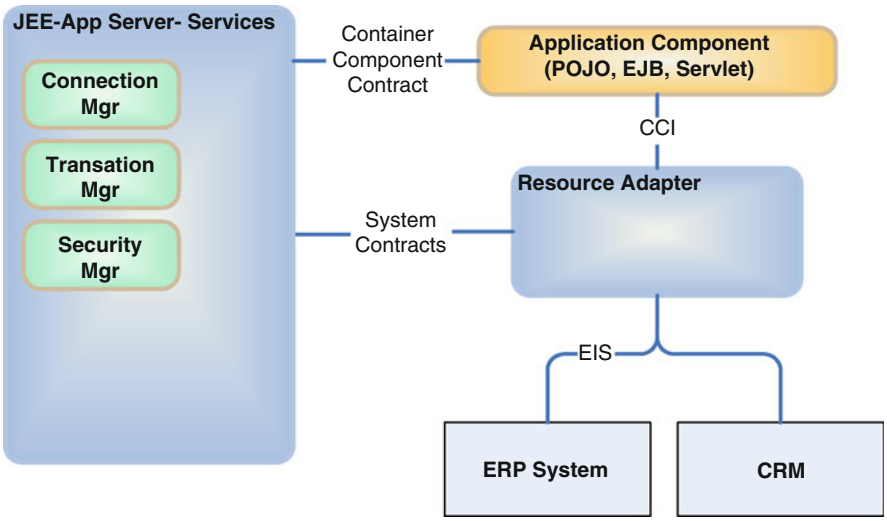

Fig. 13.6 JCA basic architecture
Source: java-sun JCA 1.5 specification

server which provides the underlying mechanisms for transactions, security, and connection pooling mechanisms.

Considering a rule engine as an EIS may look strange, at first, as a rule engine does not access EIS per say, but the goal of this implementation is to leverage the contracts provided by the JEE container such as transaction, security, and connection management without reinventing those services. Resource adapters implement two things: the Common Client Interface (CCI) used to expose the high level JCA API to the caller, and the implementation of the functionality expected using underlying EIS resource.

There are two main types of contracts that a resource adapter (RA) implements in order to get compliant with the JCA:

- The application level contract defines what the RA needs to support so components within the JEE container can communicate to the EIS.
- The system level contracts: which are connection management, transaction management, and security management.

Connection management provides a connection factory and connection interface based on the CCI. It pools the connections to the EIS to improve performance. A rule engine is attached to a connection. So rule engine pooling is linked to connection pooling. Transaction management allows EIS resources to be included in the transaction initiated by the container's transaction manager. The RA manages a set of shared EIS resources to participate in a XA or local transaction. Finally, security management secures access to the EIS through user identification, authentication, and authorization and uses communication security protocols.

The resource adapter in JRules is named eXecution Unit (XU) and aims to handle the low-level details of initializing and invoking the rule engine. It adds a management layer used to access resource adapter, resources such as connections to a ruleset data source and exposing configuration and run time data. An XU is packaged as an independently deployable unit called a resource adapter archive (RAR) .

There is only one XU deployed (.rar) per Application Server instance. Figure 13.7 illustrates a classical deployment within a JEE container. The clients are decision services, which are using the rule session factory and the rule session to access the rule engine. The implementation of a session is getting SPI connection from a pool managed by the JEE container (JCA pool).

The XU provides scalability by using context pooling and ruleset caching: Each `IlrContext` is linked to an SPI connection, which the application server caches within the JCA pool. In fact due to the transaction support requirement, asynchronous ruleset parsing, and hot ruleset deployment use cases, one JCA SPI connection is associated to a set of `IlrContext`.

The `IlrRuleset` is shared between engines and kept in memory until there is no more `IlrContext` using it (SPI connection reference). At the end of an execution, the server may decide to put the SPI connection back into the JCA pool. In this case, the associated `IlrContext` will be reset and ready for another execution. In the case of XML binding usage, the dynamic classes are attached to the ruleset and are therefore made available directly to the XU. For Java implementations, all the classes are passed to the XU by the rule session class

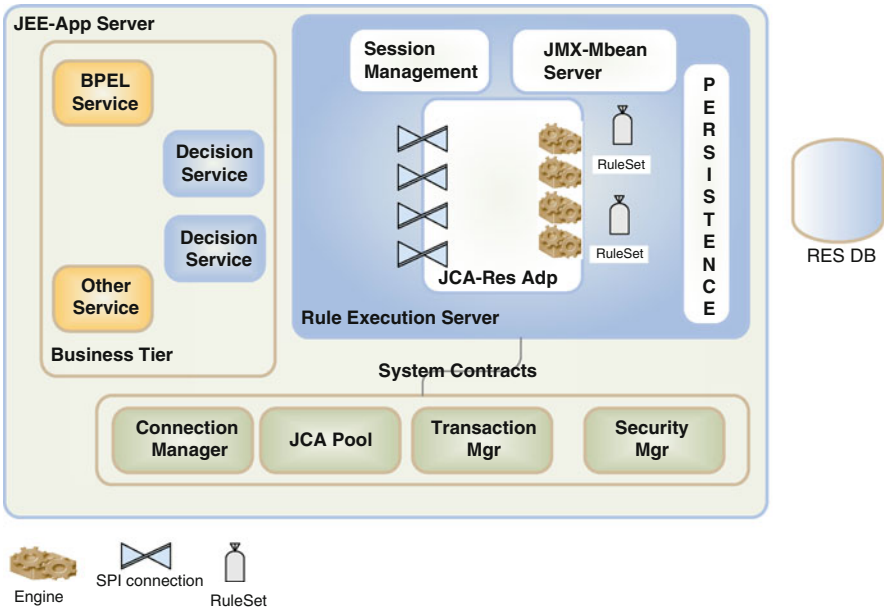

Fig. 13.7 Rule Execution Server as resource adapter

loader. In Fig. 13.7, we can imagine the decision service as packaged within a WAR or an EAR. If we deploy it in Tomcat 6 for example, we use the J2SE packaging which includes jrules-res-session.jar and the JCA API: j2ee_connector-1_5-fr.jar. The same data source must be used for the management and execution stacks. To do so we get a ra.xml file from <jrules-home>/executionserver/bin and add it to the classpath. This file will override the default_ra.xml descriptor provided in the jrules-res-execution.jar. When we use a data base to persist RuleApps we need to change some of the properties in this file, like the `persistenceType`, and the `persistenceProperties`.

The last important component of RES is the management model. When deploying a ruleset using the RES Console, this one saves rulesets to a data source, and signals changes to the management stack of RES using the JMX[5] protocol. The management model is based on the JMX Mbeans specification and is used to deploy, to manage, and to monitor the execution resources of Rule Execution Server. The various MBeans of the RES model are the runtime proxies of each entity within the model. There are three Mbeans deployed in each managed server:

- The `IlrJmxModelMBean` is the root of the Rule Execution Server management model. It controls every RuleApp deployed on Rule Execution Server. This MBean performs actions such as adding and removing references to the RuleApps contained within the model.

[5]See http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/ for details.

- The `IlrJmxRuleAppMBean` is a management entity that controls a deployed RuleApp. This MBean performs actions such as adding and removing references to the rulesets contained within a RuleApp.
- The `IlrJmxRulesetMBean` is the management entity that represents the execution resources for the Execution Unit (XU). This MBean exposes some runtime metrics of the execution. These metrics are computed from the various data collected on each XU in a cluster. It exposes an API to set the resources and properties that are used at execution time and provides a "hot" deployment entry point to update the rules at execution time.

The XU reads rulesets from the persistent layer whenever it needs to, that is, when the application server has removed a cache entry from the JCA connection pool, or when a new ruleset was deployed to the data source and the XU receives a JMX notification message.

13.4.1 Using RES Session API

When using a Rule Execution Server, the implementation of a business interface is leveraging the rule session object to communicate with the rule engine. Starting with JRules 6.0 rule sessions use a factory interface to allow clients to obtain management session, stateless, or stateful execution sessions. The factory represents the entry point to communicate with the RES. The code used to get a rule session factory has to specify if we use a POJO, J2EE, or J2SE sessions. Most of the time when deployed into a JEE container, it is simpler to use a Plain Old Java Object approach and create a POJO factory as a singleton within the business service implementation. We recommend defining the rule session factory as the singleton design pattern: during the first call to a ruleset, the RES creates the XU resource like the connection pool, loads the classes, and parses the ruleset. Using a singleton enforces that each subsequent call will only execute the rules and not the ruleset parsing. Using an factory instance enforces parsing the ruleset at each call.

> ***Concept*: Singleton**
> Singleton[6] is a design pattern to restrict the instantiation of a class to one unique object. In Java, this restriction applies within the classloader and uses the "static" keyword to declare the unique instance.

The factory has different implementation depending of the type of deployment. The following table summarizes each possible implementation:

[6]See Wikipedia detailed definition at http://en.wikipedia.org/wiki/Singleton_pattern.

Name	Description	Comment
J2SE Session Factory	Used in pure J2SE environment. It is thread safe.	The rule session implementation provided by this factory does not support transactions.
POJO Session Factory	Session used for JEE deployment. No EJB support.	Simplest interaction with the engine.
EJB3 Session Factory	Used by EJB code to get access to a rule session with JNDI lookup. The sessions are EJB session obtained from the JNDI namespace.	Pure EJB pattern, but with transparent life cycle.

Rule sessions help execute the rules in stateless or stateful mode. Most business applications are using stateless, stateful mode is rarely used. The stateful mode aims to maintain a long runtime communication model with the engine in particular to control the working memory and to keep object references at each execution call. The implementation of a stateful mode is a bit more complex as it forces developers to manage the full life cycle of the engine and its working memory. A rule session provides the class loader for the Java XOM and therefore will almost certainly be packaged in every client application. This class is dependent on the application server used. So copy the jar file from <jrules-home>/j2ee/<application-server>/jrules-res-session-<appserver>.jar and package it with your application ear.

A decision service, which uses the RES API, follows the same pattern already seen before: Get a session, set the parameters, call the rule execution, parse the results, and return the result to the caller. The session request is open using a canonical path to the ruleset under execution. The path includes the reference to the RuleApp and ruleset:

```
// Create a session request object
IlrSessionRequest sessionRequest = factory.createRequest();
sessionRequest.setRulesetPath(IlrPath.parsePath("/ClaimProcessingRuleApp/Validate
ClaimRules"));
// … Set the input parameters for the execution of the rules
Map inputParameters = new HashMap();
inputParameters.put("validateClaim", claim);
inputParameters.put("validateResult",new Result());
sessionRequest.setInputParameters(inputParameters);
  try {
   // Create the stateless rule session.
   session = factory.createStatelessSession();
   // Execute rules
   IlrSessionResponse sessionResponse = session.execute(sessionRequest);
   // get result
   result=(Result) sessionResponse.getOutputParameters().get("validateResult");
  …
```

This code is best written in Rule Studio using the java client for RuleApp wizard, and then integrated into the decision service implementation. If we need to use a stateful session, some care has to be taken to reuse the factory, the rule session, and other objects to avoid losing the stateful management of the working memory.

The choice of session type is linked to the deployment strategy of each decision service. When the service is deployed within the same server as the RES, a local rule session can be used. The POJO or EJB rule sessions are the possible choices to interact with the RES. The use of EJB session is relevant to support transaction propagation and security requirements. The session is coming from one of the possible session factory. Below is an example of EJB3 rule session factory to use within the decision service code:

```
IlrSessionFactory factory = new IlrEJB3SessionFactory();
// work on the session request the same way as code above …
IlrStatelessSession session = factory.createStatelessSession()
```

When the client code is remote to the RES, remote EJB can be used, or Message Driven Bean. For remote EJB, the IlrEJB3SessionFactory has a simple API to set a remote flag to get remote session. Message Driven Bean represents one of the most common deployments when we need to integrate with Enterprise Service Bus, legacy application connected with IBM WebSphere MQ or any asynchronous event architecture.

13.4.2 JMS Deployment

As detailed in the previous chapter, Message-Oriented Middleware is the technology of choice for asynchronous processing of messages. JRules Rule Execution Server delivers an out of the box Message Driven Beans (MDB) (`IlrRuleExecutionBean`) to invoke the XU within the onMessage() call using a simple session and then posts the execution results to a JMS destination. The MDB with the rule session are packaged as an EAR file and deployed in the JEE container. This implementation may be useful for event driven application or with mainframe application integration. The JMS message needs to include the ruleset path and a status property. The message body has to include the ruleset parameters using key-value pairs. The client code posting messages to a topic or a queue needs to specify the ruleset path it wants to execute. This strongly coupled integration between the client and the rule service is not a common usage of JMS. Most of the architectures which are leveraging message-oriented middleware or ESB use a loosely coupled approach where clients post messages without any knowledge of what the consumer is. So if we need to implement an Event Driven Architecture decision service which can be used with messaging communication we may need to leverage our own MDB implementation which will hide the fact we are using a rule set. The onMessage() method can do the unmarshalling of the JMS message payload into a Java data model and then can synchronously call the business service responsible to process the business objects. The outcome of the decision services can be processed by a publisher class back to the JMS layer. Queue or topic listener needs to get references to the business service

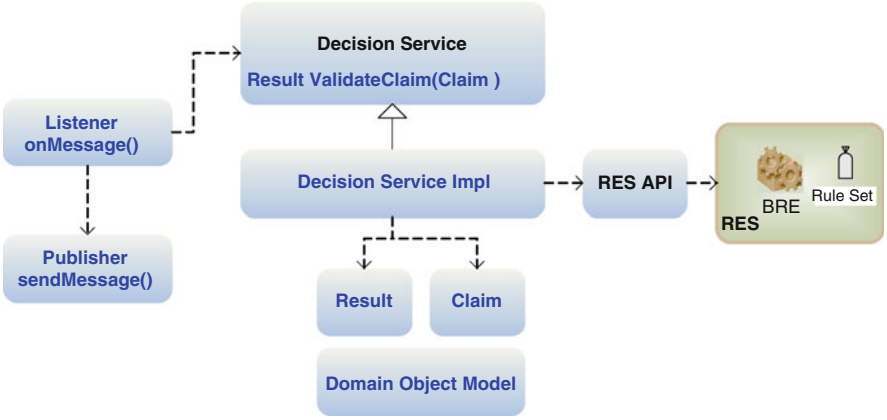

Fig. 13.8 JMS – Rule engine deployment

implementations and to the publisher so that it can send the result back to a topic for future processing. Figure 13.8 illustrates a design where the decision service implemented following the concepts presented in previous section can be re-used with the JMS communication without a lot of work.

13.4.3 SCA Component

Another interesting wizard within Rule Studio is used for generating RuleApp client code with all the needed artifacts to deploy the rule execution as a Service Component Architecture (SCA) component. One of the main goals of SCA is to clearly separate the communication details from the business logic: the protocols and quality of service are wired at execution time, the developer focuses on defining reusable services and components supporting the business functions to develop. By looking at the generated java code, there is no difference with standard RES client implementation (see Sect. 13.4.1); the only difference is coming from the composite descriptor used to define the SCA component. The component statement in the composite file may look like:

```
<component name="ClaimProcessingComponent">
 <implementation.java class="claimProcessing.server.ClaimProcessingImpl"/>
</component>
```

Before release 7.1.1 of JRules, the wizard leveraged the Apache Tuscany[7] Java runtime on client side to call the service. In later releases of JRules, the SCA

[7] http://tuscany.apache.org.

implementation used is the one coming from the IBM WebSphere SCA feature pack. Most likely a business application will not use the RuleApp exposed "as is" as a SCA component but will use a business service as façade for the rule execution. In that case, the generated client code can be used as a starting point for the implementation of such a business service. The caller of an SCA component needs to get a SCADomain instance by specifying the composite descriptor, then get the service reference, and finally call the business method (e.g., validateClaim).

```
   // Create a Tuscany runtime
   SCADomain scaDomain = SCADomain.newInstance ("ClaimProcessing.composite");
   ClaimProcessing service = scaDomain.getService(
    ClaimProcessing.class,"ClaimProcessingComponent");
   // prepare objects like the claim … then call the execution using the decision
service API
   service.validateClaim(theClaim);
```

For WebSphere SCA feature pack[8] the access to the service is done using the service manager like:

```
com.ibm.websphere.sca.ServiceManager.INSTANCE.locateService
("ClainProcessingComponent");
```

13.4.4 Monitoring and Decision Warehouse

When an administrator wants to monitor the rule execution, he can use the Rule Execution Server Console which is a web application deployed to a servlet container like Tomcat or WebSphere Application Server. The RES Console includes the JMX MBean server used to receive rule execution statistics. The application server needs to support JMX, JNDI, and JDBC data sources. In a J2SE deployment if we want to have the monitoring capabilities we need to have the RES Console and the RES-XU in the same JVM.

One of the first monitoring functions is to verify the server status. In the RES Console, the Diagnostic tab allows the execution of a set of predefined tests and to present color coded results. The tests address connection, resource adapter information, rule app and ruleset status, etc. (see Fig. 13.9).

In Fig. 13.9, the XU lookup and XU MBean are yellow because we did not execute a ruleset yet. The XU MBean is created when the XU connector is created. The XU connector is created when a connection is requested so a rule session opened.

[8]See details at http://www-01.ibm.com/software/webservers/appserv/was/featurepacks/sca/.

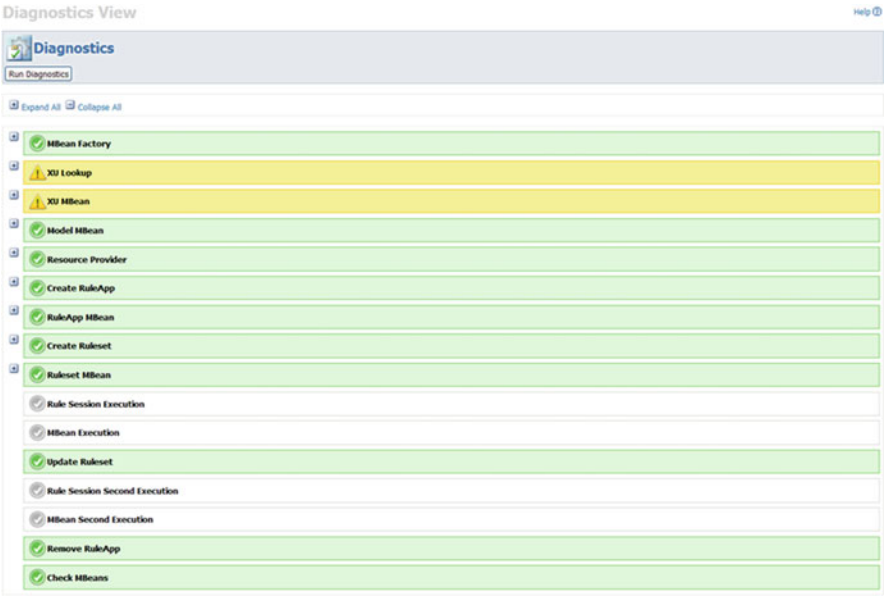

Fig. 13.9 Rule Execution Server diagnostic report

As part of the monitoring capabilities in JRules v7.x is a new feature called Decision Warehouse (Decision Validation Service add-on) which stores the rule execution results in a data source. The type of data persisted may vary depending on the application, but it is possible to get the list of rules fired, the rule tasks executed for a given transaction, and the content of the ruleset parameters. When the ruleset is deployed from rule team server, it is possible to get within the trace, hyperlinks back to the corresponding rule in RTS repository. This capability helps to quickly assess for a given business transaction what were the conditions which made the rule fire. Using the Decision Warehouse tab of the Rule Execution Server Console, we can search the rule execution trace by specifying search criteria. Figure 13.10 presents this capability.

It is interesting to note from Fig. 13.10 that the first rule execution took more processing time (328ms), as the RES was parsing the ruleset. The detail of the decisions made on the last claim processed gives us information about the claim sent, the path of execution within the ruleset with the rules fired. Figure 13.11 presents such results.

To configure the Decision Warehouse, we need to enable the ruleset execution monitoring by setting the `monitoring.enabled` ruleset property to `true`. It is possible to use Rule Studio, Rule Team Server, or Rule Execution server to set such property. RES offers a simple Add Property command inside the ruleset View. This command supports predefined property (Fig. 13.12).

Usually developers will want to set those properties at the rule project level in Studio by using the rule project property and the ruleset property menu. When the rule flow defines a task that uses the sequential or Fastpath algorithm, we need to

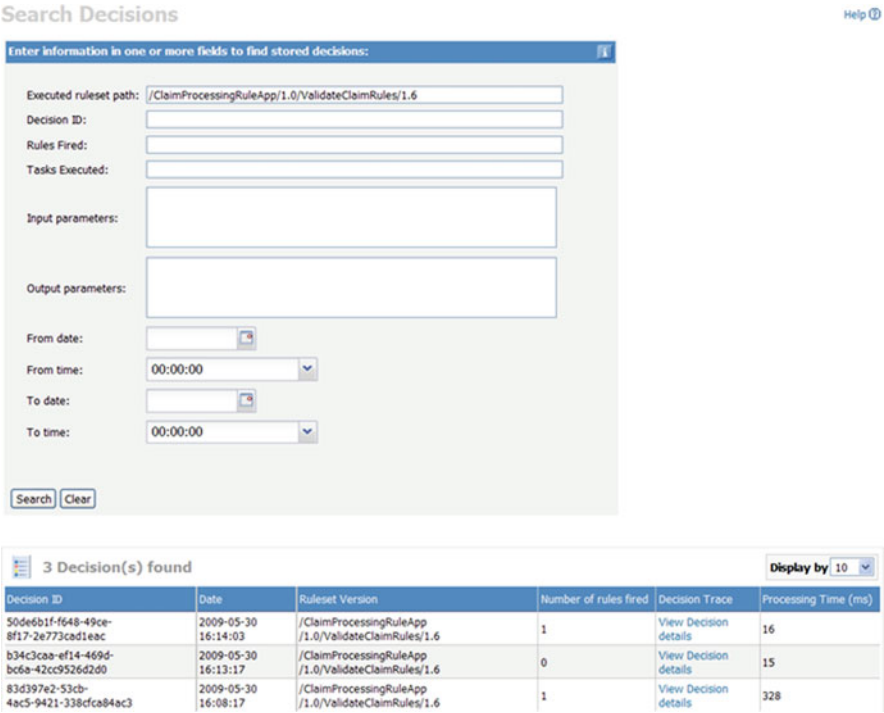

Fig. 13.10 Search Decision Warehouse

add another ruleset property called: `sequential.trace.enabled`. Lastly as we may want to trace what the inputs and outputs were and as such we may need to set `ruleset.bom.enabled` property to true. If the ruleset is based on an XSD XOM, the input/output parameters are stored as XML documents. If the ruleset is based on a Java XOM, the toString() method of the ruleset parameter (s) type stores the content. Using toString we can limit the information persisted to improve the performance.

The Decision Warehouse stores its results to different data sources. It is important to properly design how to organize the data sources and partition the traces according to the different decision services the application is supporting. It is also possible to add our own DAO to store the information in another database; therefore, a business user can run Business Intelligence report from it. The product documentation details all of these and provides some customization samples.

13.4.5 Transparent Decision Service

We have seen quite often the term Decision Service in previous chapters. In a SOA, it is a service which is making a business decision on business data. Such

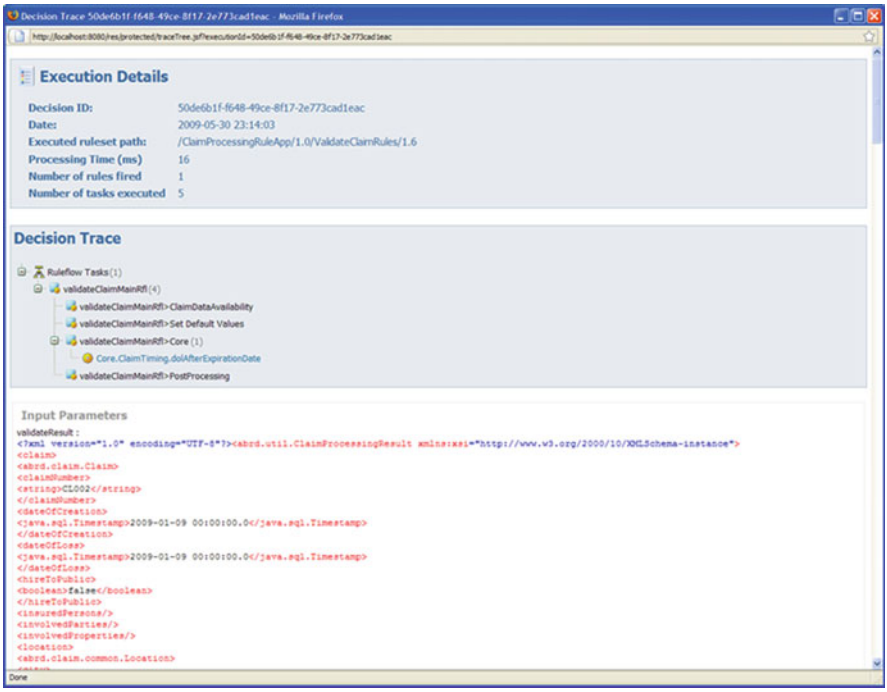

Fig. 13.11 A rule execution report from Decision Warehouse

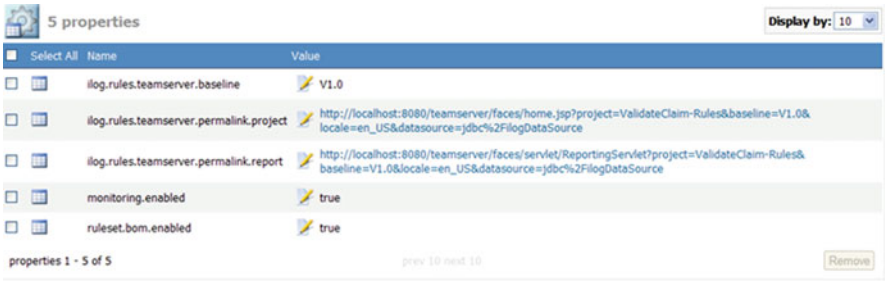

Fig. 13.12 Ruleset properties set in RES Console

services are most often supported by a business ruleset executed by a rule engine. In IBM WebSphere JRules, there is also the concept of transparent decision service (TDS). The term transparent means we can define the decision logic externally using a rule repository without having to dig into application code to understand the rules. The Rule Execution Server can expose a ruleset automatically as a web service as soon as it uses an XSD as executable model (or is limited to using basic java types). The interface definition is based on the ruleset parameters, which are defined according to the nature of the business decisions to

make, and should not be dependent on the needs of a single business process. TDS is accessed using SOAP and can offer an easy deployment within a SOA. As we explained in previous chapters, the design of reusable service consumers want to reuse enforces respecting a set of best practices like loosely coupled, coarse-grained interfaces, simple data model a client can create and send to the service provider, keep the business meaning, and the service provider responsible for his processing like loading the data. A ruleset supports one operation of a business service: service end point groups operations over a single place, an URL, over a single protocol; therefore, a ruleset has to be one operation within a packaged business service. For efficient rule processing, the data needed by the rules need to be present, a good design practice is to let the service have the responsibility of loading the data it needs and not ask the client to send the complete data set. This is truly relevant with reference data. This is not to say we need to load the data within the ruleset, but before calling the rules. Loading of the data in a ruleset may be an attractive solution but has a lot of drawbacks. In particular, when we need to support transaction initiation or propagation, the call to load the data can occur at the beginning of the rule flow; but if there are exceptions or a time out, the management of such events is more complex to support in rules than in traditional code. Finally, most of the ruleset processing needs some reference to other technical service to access them for getting reference data, or other business objects. Those services should be hidden to the caller.

There are two types of TDS in JRules: the hosted and the monitored:

- A *hosted transparent decision service* (HTDS) is a ruleset deployed as a Web service. It is installed and integrated on the same application server as the Rule Execution Server. It includes a JMX MBean and is packaged as an EAR (for example jrules-res-htds-WAS7.ear for Websphere) or as a WAR for Apache Tomcat 6 (named DecisionService.war). This web application defines some web service servlet and HTTP listeners which process SOAP requests and route the message payload (XML document) to the ruleset. Any ruleset is mapped to the following URL:

```
http://<hostname>:<httpport>/DecisionService/ws/<NameOfTheRuleApp>1.0
/<NameOfRuleSet/ 1.0?WSDL
```

  The objects defined in the BOM and the ruleset parameters are used to generate the WSDL file. When the BOM was created by using XSD, HTDS is very easy to set. The WSDL binding is using SOAP over HTTP with a Document/literal style. A developer can import this WSDL and generate the client code with tool such as Apache axis wsdl2 java.
- A *monitored transparent decision service* (MTDS) resides on the same application server, but is not integrated with RES. It is generated from Rule Studio as a web app using the wizard: New > Client Project for RuleApps > Web Service. There are two projects created by this wizard. One client project which includes the client code calling the web service. And the other one the server side project

which includes the web service definition using the reference implementation of JAX-WS so it is not supported in all application servers. MTDS manage rulesets that use an XML schema or a Java XOM with any object types (not limited to basic java types).

Starting with the server side generated code, the generated project includes java files, ant script, Execution Unit configuration file (ra.xml), and other xml descriptors to create the web service (web.xml, application server specific deployment descriptor). Using the ant targets, we can generate a war file for the application and deploy it to the target application server. Each rule project part of the Rule App is exposed as a WebMethod using a signature like `<RuleSetName>Result execute<RuleSetName> (<RuleSetName>Request request)`. The operation results and request are mapped to wrapper objects which include references to the ruleset parameters. For example, the ValidateClaimRuleRequest has a reference to the Claim (Fig. 13.13).

The implementation of the web method is using the RES API to set the parameters and to call the rule execution. The Java objects used for the data model must respect the JavaBean specification. The generated code can be used to implement the business services we need to code and exposed as reusable service. At the interface definition level we are not specific to rule execution. The service method can be renamed to better serve business operations. If we need to hook up some reference to other services needed by the rules, we can do so in the implementation class. The listener class can be reused to offer statistic reports in the RES Console. The client code is also an excellent starter code to implement some simulator or functional test framework.

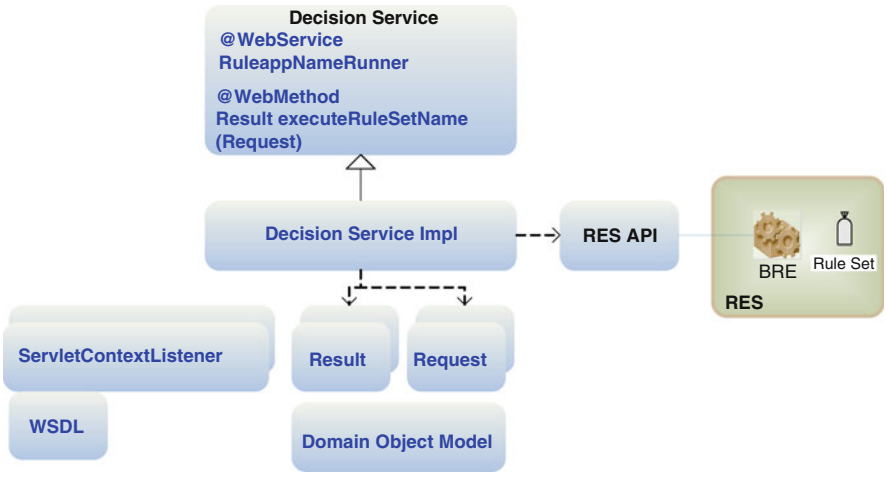

Fig. 13.13 Monitored TDS generated components

13.5 Rule Team Server

In this section, we present the deployment of rule team server web application within the IT architecture and how to leverage the data source mapping to support multiple rule repositories with one web application. Then, in Sect. 13.5.2, we briefly present the concept of queries, an element used to control the ruleset packaging.

13.5.1 Physical Deployment

As a management application, the war file does not need to be deployed on the same node and server as the rule execution server. It is better to deploy it on a different server, because it can use resources that may impact the performance of the rule processing. The deployment follows traditional Web App deployment using a database: we need to configure the JNDI data source reference and specify in the web descriptor which JNDI name to lookup. Rule team server is also delivered with an Installation Manager which helps to deploy the DB schema when the database does not exist. It is important to note that when you are using a different rule meta model the database schema is different. This is easily done by loading the XML files describing the extension model into RTS (Fig. 13.14).

All the rule projects within RTS share the same meta properties; therefore, if there is a need to have different extension model, architect may need to define different rule repository data sources. By default, the data source used is `jdbc/ilogDataSource`. If we want to specify a different data source, we have to pass it as a request parameter in the URL, for example, http://localhost:8080/teamserver?datasource=jdbc/otherteamserverds.

This capability is also used to support different development branches: one data source is used as trunk and other for other releases. We will detail that in Chap. 17. It is also possible to define one data source per group of users or line of business: finance and marketing teams may have two rule repositories clearly separated but one RTS deployed.

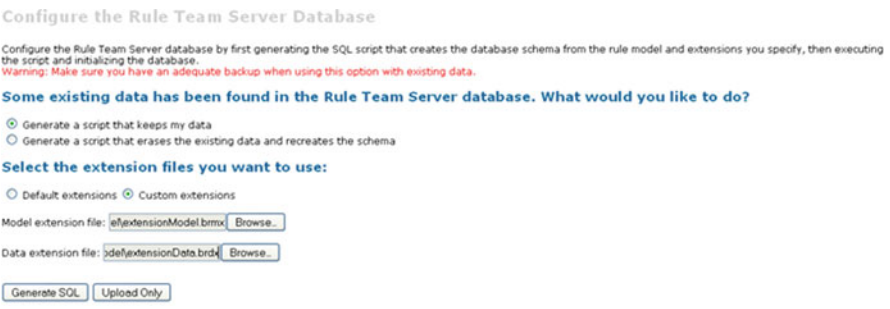

Fig. 13.14 Add custom rule properties in Rule Team Server

As part of the physical deployment is the support of "single sign on" integration for getting users information like userid, group, and password from a central directory service. RTS can be deployed in an application server and will leverage the container contract as RTS uses the JAAS API to retrieve user's data. The important configuration to complete before running RTS is the group definition and assignment of the user to one of the four groups of RTS: rtsAdministrator, rtsConfigManager, rtsUser, and rtsInstaller.

Finally as RTS is the main component to control the rule project, it is important to avoid duplicating rule repositories between multiple platforms. It is possible to manage the ruleset deployment to different RES platforms from one central deployed RTS. This is the simplest and most efficient deployment. The second pattern is to use one RTS per target platforms, as most IT environment includes at least development, test, UAT, production, we can have unnecessary deployed RTS. Rulesets are deployed to the different deployed RES. Finally, another common deployment is to use two RTS, one for all the rule authoring done by the business user, used to deploy ruleset to any execution platforms except production. And one in production managed by IT and mostly used to support rule 'hot fix', exclusively deploy to production RES. It is this last RTS instance that will be used for ruleset deployment to production server.

13.5.2 Queries

Queries are an important element of the Ruleset deployment. Using query and ruleset extractors, we can control the ruleset deployment for different purposes (e.g., test, simulation, and production) and platforms. Common dimensions used in business are the effective date and expiration date for some business entities like a product, a pricing campaign, a medicine availability, a loan eligibility, ... Business rules defining constraints on those entities have to follow the effective and expiration dates patterns. With a ruleset extractor, business users may extract only the rules valid at a given time, or can search for rules in a given status. Queries can be added to RTS repository by any type of user with the create query permission. We will detail in Chap. 17 the fine grained permission management RTS provides. Figure 13.15 presents a query developed to extract rules ready for production.

Once the queries are defined, a RTS administrator can define extractors using the feature Configure > Edit Ruleset Extractors. An extractor is defined using a name, a query, and a validator (Fig. 13.16).

Extractors are then used in the creation of the ruleset archives by specifying the extractor name. By default all the rules are extracted to the archive (Fig. 13.17).

With queries and extractors we can package rulesets as intended for the different purposes platform dependant like test, simulation, and production, or time-oriented, or any business needs.

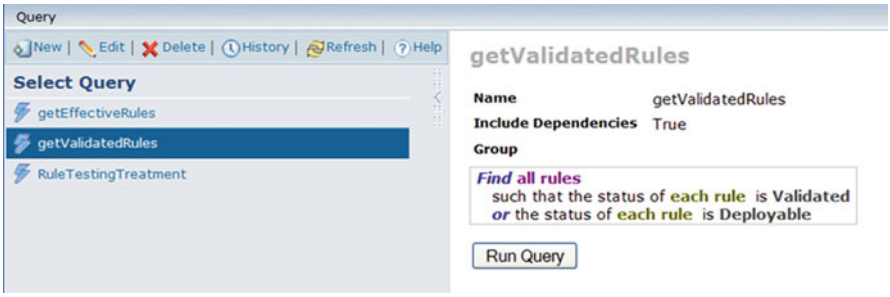

Fig. 13.15 Query to get rules ready for production

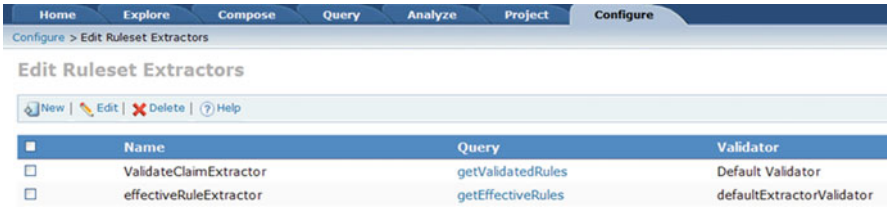

Fig. 13.16 Manage ruleset extractors in RTS

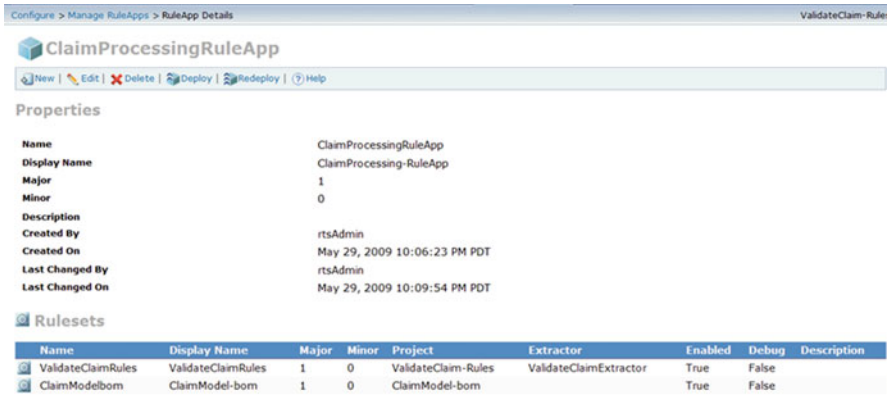

Fig. 13.17 Ruleset archives built using extractor

13.6 Summary

We reviewed the concepts of operation of JRules and detailed the RuleApp element, which includes one to many rulesets and which represents the deployable unit to the rule execution server. RES supports monitoring of rule execution, with

the option to persist the trace in a decision warehouse. To support vertical scalability, the RES leverages the JCA connection pool, so parallel executions are possible as soon as the server has multiple CPUs or Cores. Ruleset parsing takes time at the first call, but once parsed a ruleset stays in the RES cache for future processing. When using a Java XOM, the ruleset parsing needs the class information as part of the classloader of the class using the rule session API. Most decision service, even if exposed as web service, should leverage a java layer, which implements the business service and completes the work of preparing the data graph for the rule processing. It is important to recall that lazy loading of data may make the rule's conditions not evaluate as true: for example, a collection of objects may not be loaded, and so a test with the *in* operator will fail. The decision service uses a stateless processing, sending all the data in one call. The deployment mode can include different patterns from JMS, for message processing, to pure POJO or EJB. As a new programming model, SCA is also supported, and SCA component implemented in Java, uses the RES API to call the rule execution. Finally, we covered the rule team server deployment, where a set of features help the business analyst to deploy the RuleApp to RES.

13.7 Further Reading

For more technical information and tutorials, the product documentation is accessible at http://publib.boulder.ibm.com/infocenter/brjrules/v7r1/index.jsp.

Service component architecture presentation can be read at IBM developerworks web site at http://www.ibm.com/developerworks/library/specification/ws-sca/, and the specification is accessible at the Open service oriented architecture portal http://www.osoa.org/display/main/service+component+architecture+home.

The rule interchange format recommendation is part of the semantic web work done at W3C. and aims to provide interoperability between rule based systems, reader can access the description of this recommendation at http://www.w3.org/blog/SW/2010/06/22/w3c_rif_recommendation_published.

The SCA support pack for WebSphere Application Server can be studied at http://www-01.ibm.com/software/webservers/appserv/was/featurepacks/sca/.