# Deploying Rules in Real Life

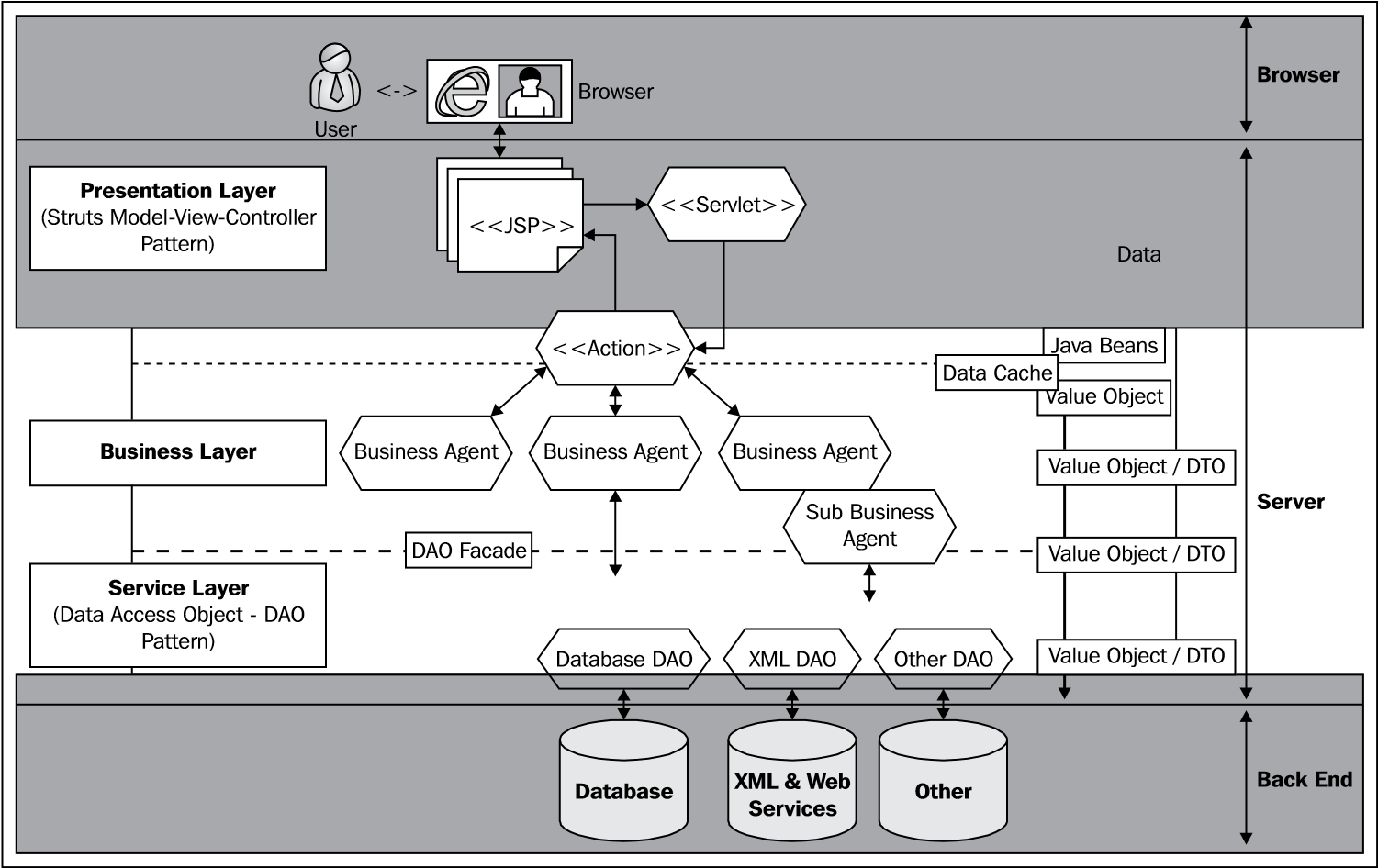
All of the previous chapters have shown you how to do many useful things with business rules—including how to write them in the Guvnor, the IDE, excel spreadsheet, and the near-English DSL formats. All of this is useless unless you can make the results of your hard labour available to end users who will interact with your business rules in some way. To make this possible, you need to get your business rules off the computer where you wrote them and onto a production server. It's like a teenager leaving his or her home for the first time and making his or her way in the wider world.

What we need to do is **deploy** our business rules.

## One size fits all architecture

If you were building a house, you'd employ an architect to design it for you before anybody started working on the site. That way, the bricklayers, plumbers, joiners, and electricians would know what the finished product will look like and can plan their work accordingly. Most computer systems cost far more than your average house (that is, when you include the cost the wages of the people involved in the IT project). So it makes sense to take a little time to prepare the architecture for these systems as well.

Fortunately, most computer systems these days are web based; that is, they use web pages to gather input from users and to display the result. It doesn't matter if these web pages can be seen only within your company or organisation, or by the world at large—both use the same underlying technology. There are entire bookshelves (or the online equivalent) devoted to the best web architecture, but pretty much all of the designs contain a similar core, as shown in the following diagram:



* Most web systems take information from the user (at the top of the diagram), do something with it (the white area in the middle of the diagram), and store it in a database (at the bottom of the diagram).
* The grey areas at the top (**Browser**) and bottom (**Back End Database** and other systems) are outside our immediate control. We have to 'talk' to them. But, typically, we're stuck with whatever web browser or database someone else has chosen.
* The part that we can control is in the middle, labelled **Server**. A server is simply a computer dedicated to running our software and making it available over the Web.
* Our system on the server is built in three layers: the **Presentation Layer**, the **Business Layer**, and the **Service Layer**.
* These three layers are like a sandwich. Standard technologies make up the **Presentation Layer** (which shows web pages to the user) and the **Service Layer** (which talks to databases and other 'backend' systems). The more interesting sandwich filling (unique to our system) is the **Business Layer** right in the middle of the diagram.
* As you'd expect from the name 'business rules', the **Business Layer** is where the business rules normally reside .

There you have the one-size-fits-all architecture. Now, what are you paying all those overpriced consultants for?

**What needs to be deployed?**

The architecture diagram contains a lot more than examples based on simple rules, which is what we've been looking at until now. There are a lot of pieces that make up a full Enterprise web system. Many of these should already have been installed as a 'one-off' by the friendly system administrators in your IT department. Others are up to you to provide.

Let's look at the parts of the solution that you'll need:

* Server: An application needs a machine to run on. This could be a box tucked away under somebody's desk (not ideal), a real server plugged into a rack somewhere (better), or a 'virtual' server hosted by an IT company or a secure third-party hosting company (better again).
* Operating system (OS): Our server is useless without the basic instructions that make it work. This would most likely be Windows (a version similar to what is on your desktop PC), Linux, or some other variant of UNIX. For the purpose of Java-based rules, our application can deploy on any of these.
* Java application server: We talked about deploying this on a desktop PC back in Chapter 2, when installing JBoss. This gives our application a 'home' to run on and provides common services such as security, database access, and a web server.
* Application: This consists of the non-rules Java code, web pages, and other configuration. This is pretty much everything except the business rules, as depicted on our 'one size fits all' architecture diagram. The application is often deployed as a **.war** (web archive) file, a type of ZIP file.
* In some cases the configuration files may be deployed separately, to allow easier changes later. (That is, you only have to update a single file, rather than the entire application.)
* Support libraries (such as web frameworks or drivers to allow us to talk to databases) are often deployed as part of the **WAR** (**Web Archive**) file, but can be deployed separately. The choice will often depend on the instructions provided with the library.
* Rules: The business rules that form the core of the system. To a large extent, the other components are deployed purely to support the rules.

In general, the server, OS, Java, and the application will be supplied by your IT department (with input from you). The web application may be provided by you, or by the development team that you are working with. The development team will also specify the support libraries required (and where on the Internet they can be obtained). Rules will be developed by you.

**Rules as code or data?**

Should business rules be embedded as a part of the application code that doesn't change very often? Or are rules more like your application data (for example, pricing lists), which you expect to change on an almost daily basis? Depending on your answers, you will deploy your rules very differently. The answer, somewhat confusingly, is that rules are 'both'.

* Rules are as powerful as the normal code and should be treated in the same way. (For example, before deployment, any changes should be thoroughly tested.)
* Rules are as easy to change as data because: first, they live outside the 'normal' application; and second, rule engines are expressly designed to allow easy changes to the business rules.

Organisations will typically have two sorts of deployment processes, which are as follows:

1. A heavier, more rigorous, process for the deployment of code as, traditionally, getting it wrong has been at best embarrassing, and at worst very, very costly.
2. A lighter process for deploying data changes—after all, the application is changing data itself all the time—such as saving user profiles in a database.

Remember that it is also possible to store rules in a database, even though most of the examples in this book store the rules in a standard text file. For example, hidden within the Guvnor is an Apache Jackrabbit repository. This repository can use an industry standard database (such as Oracle, Sybase, MySql, or Microsoft SQL Server) as its storage area.

Perhaps the best way of summing this up is that rules give you a lot of power, and many deployment choices; so be careful how you use them. Whatever view you take, any changes to your rules in the 'big bad world' of production, or a live system, should be part of a standard, well-documented process.

### Deployment process

We know how to write our business rules. We also know about the other pieces of the application puzzle (or at least we know who to ask them for). And we've decided whether we want to treat our rules as code or as data. Now, how do we get them from our computer (where we have written them) to the production server (where the entire company, or world, can interact with them)?

What we need is a deployment process.

If you don't have a deployment process, the worst case scenario is that you have to edit the rules on the production server. In doing so, you not only risk breaking an application that users are using, but you also risk not having a backup copy.

A slightly less bad scenario is that you edit the rules on your own PC, and then put them live. If you have no process to manage this transfer, you risk making a mistake. And your mistake may not be recognized until several days later. How do you return to the last known good version of your rules?

What we need is a deployment process. So what does a proper deployment process look like? Although there are many variations, at their core they contain the features shown in the following diagram:

Development

Test

Firewall

Production

The first surprising thing that you encounter is that three separate computers are used in the deployment process. One (maybe your local PC) is where you write your rules. Another is the production server, which is a server accessible to the end users. But the rules aren't deployed directly from development to production. They get there via a test box—a server that is as similar to the production server as possible. It allows verification of the code that we are about to deploy, without disrupting users.

We need a separate test server to verify our application (and rules). The main reason for this is that other people will need to test your rules before they go live. No matter how careful you are, you might miss something. Having a third-party test for your rules provides an additional level of confidence.

The second group of people involved in the deployment process are the system administrators. It is unlikely that you will move the code between the various servers—it will more likely be the sys admins who do the job for you. Their role is to ensure the stability of these servers, and not just to put your application live. Expect plenty of questions from the sys admins about differences between versions, the underlying software requirements, step-by-step instructions on how to install the software, and which repository to find the code to deploy in.

**What's a repository?**

One of the questions the system administrators will ask you is "Are you storing your source code in a repository?" Now, what's a repository? And why do they care—after all you're deploying the finished, compiled product, not the source code?

The problem with storing your files on a local file system is that if you delete a file accidentally, it's very difficult to recover. Yes, you have the recycle bin, but what happens if that is full? What happens if you delete a portion of a file, save it, then only realize your mistake five minutes later?

A repository solves this versioning problem. It allows you to save a complete history of all of the changes that you have made and roll back to whichever version you choose (should this prove necessary). By allowing the merging of files, an entire team can co-operate on developing an application. There are many repositories available from many suppliers, including Microsoft, and Subversion.

You'll remember that Guvnor has a built-in repository, and take advantage of this later in the chapter. This repository stores only rules and items directly related to rules. For version control of the other items in our application, we would still need a full repository.

### Deploying rules

Just to complicate things further, there are many options for how we can deploy the rules. They are as follows:

1. We can deploy the rules as plain-text source files—similar to the approach used in the examples in this book. Although this is perhaps the simplest approach, we will need the Drools compiler as a support library. This means that there can be problems, which we will find only after we deploy.
2. One alternative is to compile the rules and the dependent objects into a binary package. It is this package that is deployed on the server. This approach needs an extra step during our compile or build process, but it will find problems sooner.
3. A third approach is to pull the rules from the Guvnor repository. Although this is convenient, this approach doesn't always have the 'somebody else checks your rules before deployment' step. It is easy (perhaps far too easy) to push changes to your rules to the production.

There are further variations on the above three themes. They are listed as follows:

* We can mix and match any, or all, of these three solutions
* We can add or remove individual packages (or the entire rule base consisting of multiple packages) before, or during, our running of the rules
* When deploying source files, we can pre-check for syntax errors, but still allow rules to be compiled on the production server
* We can deploy our rules as code (as an integral part of the application) or as data (that is, build our application to check often for rule-changes)

Remember that Drools is pretty powerful. It can cope with rules swapping in and out, even when the rules are running as a part of long-term stateful rule session. We can also serialize (akin to freeze-drying) all of the Drools objects. This means we can de-serialize them later—a bit like adding water again to bring the rules and other Drools objects back to life. Serialisation means that 'live' rules can be sent over the network, and stored in a database for later use, and so on.

**Push or pull?**

In a normal deployment style, we 'push' the items to be deployed from the development PC or repository to the test server, and finally to the production server. The following diagram shows push deployment:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | |  | | --- | | Rule Repository | |  | |  | | --- | | Production Server | | |

The copy, or deployment, (as shown by the arrow) is often a manual copy of the rules files. This is great because the copy will often be managed by system administrators, and they will ask you tough questions (such as 'Are you really sure?') before they deploy. This push style fits very well with treating rules as traditional code.

On the other hand, aren't rules meant to be dynamic? What if we want to make a small change to a business rule, which is after all designed to change frequently? Do we have to wait for a system adminstrator to be available? Will all of the questions they ask just get in the way?

Rule

Repository

Production

Server

The pull style is meant to be more dynamic than this. In it, the production application using the rules will check every 'x' minutes, or hours, or days to see if there is a new set of rules in our rule repository. If so, it 'pulls' the new rules into the application. This process is similar to Windows Update, and can happen automatically, behind the scenes. A Drools component called 'RuleAgent' is provided to help build these kinds of systems.

The pull style works best if you are treating your rules as data that changes often. It also works best if the kind of changes you are making to your rules preclude costly mistakes—the sysadmin's 'Are you sure?' question comes from hard-earned experience. Remember that there is no 'best' deployment mechanism, but only the one that works best for your project.

### Loading our rules

Whether we push or pull our rules to the live system, or deploy the rules a plain text of a pre-compiled or processed binary, we've got the same problem: How does our application actually load the rules we give it? In our previous examples, we've glossed over this process while we concentrated on what the rule engine was doing. Now it's time to take a detailed look at how we call the rule engine from Java.

#### Looking inside RuleRunner.java

Almost all of our samples have used some variation on the **RuleRunner.java** file to do the hard work of calling our rules. The good news is that the process is the same, which means that you can reuse it in your own projects. The code in **RuleRunner** will cover the following scenarios in which we have to:

* Load rules in a stateless manner
* Load rules in a stateful manner
* Load decision tables
* Load rules with DSL
* Load rules with Ruleflow

#### Helper methods

Firstly, we'll look at the code that all of these scenarios will use. The following code identifies our package, imports the other files that we will need, defines some constants (the XLS file extension, which is unlikely to change any time during the life of the program), and also defines the **RuleRunner()** method—it makes **RuleRunner** a JavaBean. Although this is not vital in the book samples, it does make the code more reusable.

If you're happy being just a business rule author, you're unlikely to need the mechanics of rule loading to the detail we're going to discuss here. On the other hand, if you're a rule and Java developer—read on. We're going to talk about lots of useful stuff.

**package net.firstpartners.drools; import java.io.FileNotFoundException; import java.io.IOException; import java.io.InputStream; import java.io.InputStreamReader; import java.io.StringReader; import java.util.Collection; import java.util.HashMap; import org.apache.commons.logging.Log; import org.apache.commons.logging.LogFactory; import org.drools.RuleBase; import org.drools.RuleBaseFactory; import org.drools.StatefulSession; import org.drools.StatelessSession; import org.drools.rule.Package;**

**import org.drools.compiler.DroolsParserException; import org.drools.compiler.PackageBuilder; import org.drools.decisiontable.InputType; import org.drools.decisiontable.SpreadsheetCompiler; import net.firstpartners.drools.log.ILogger; public class RuleRunner { private static final String XLS\_FILE\_EXTENSION = ".xls"; private Log log = LogFactory.getLog(getClass());**

**public RuleRunner() {**

**}**

The imports fall into the following four broad categories:

* Standard Java objects, such as exceptions (errors) when a file is not found or cannot be opened (**IOException**), and utilities for reading files (**InputStream**, **InputStreamReader**, **StringReader**). It also includes Java standard collections and a Hashmap (a collection of name-value pairs).
* Objects from Apache, which provide many useful open source utilities. In this case, we import the logging utilities. These help us print messages about what is going on in our samples.
* Next, we import the Drools objects. We've seen them before—a **RuleBase** (or compiled set of rules) and a **RuleBaseFactory** to help us load the rules from a package. We have stateful and stateless sessions (working copies of the rules), as well as compiler for both standard and decision table based rules.
* Finally, we import other Java files from the project—in this case, a utility that we use to help write log entries for our rules.

The next section contains helper methods that are used by the stateful and stateless rule loaders.You’ll see that these are marked private, that is, only other methods within the same file—the public **runStatelessRules** and **getStatefulSession** methods that you will recognize from the samples in previous chapters. The main helper method used is **loadRules**, which finds the multiple rule files we pass into it, as well as any DSL or ruleflow files that we specify.

If we don't have a DSL or Ruleflow file, then we can pass in **null**, which is a special Java word meaning 'empty' or 'nothing'.

**private RuleBase loadRules(String[] rules,**

**String dslFileName,**

**String ruleFlowFileName) throws Exception{**

**RuleBase localRuleBase = RuleBaseFactory.newRuleBase();**

**PackageBuilder builder = new PackageBuilder();**

**for ( int i = 0; i < rules.length; i++ ) {**

**String ruleFile = rules[i]; log.info( "Loading file: " + ruleFile );**

**//Check the type of rule file, then load it if(ruleFile.endsWith(XLS\_FILE\_EXTENSION)){ loadExcelRules(ruleFile,builder);**

**} else {**

**loadRuleFile (ruleFile,dslFileName,ruleFlowFileName,**

**builder);**

**}**

**Package pkg = builder.getPackage(); localRuleBase.addPackage( pkg );**

**}**

**return localRuleBase; }**

Within the method, there are three main rules-related features that are highlighted in the code above. They are:

1. The line beginning with **for (...)** loops though all of the names of the rule files that we pass in.
2. The line beginning with **if** **( … XLS\_FILE\_EXTENSION)** checks to see if our rule file is an Excel decision table. Depending on the answer, it calls a helper method to load a normal, or an Excel, rules file.
3. All of the rules files loaded by the helper method are added to a package (**builder.** **getPackage**), and then combined into a single rulebase (**localRuleBase.** **addPackage**).

The **loadRuleFile** method (which loads DRL rules) is called by the **loadRules** method. However, it only takes the name of a single **drl** **rulefile** as a parameter (as well as the **dsl** and **ruleflow** files).

**private void loadRuleFile(String ruleFile, String dslFileName,**

**String ruleFlowFileName,**

**PackageBuilder addRulesToThisPackage ) throws DroolsParserException, IOException{**

**//This method is more flexible in finding resources on disk InputStream ruleSource = RuleRunner.class.getClassLoader() .getResourceAsStream(ruleFile);**

**//We must be able to find all rule files if(null==ruleSource){**

**throw new FileNotFoundException**

**("Cannot find rule file:"+ruleFile); } else {**

**log.info("found rule file:"+ruleFile);**

**}**

**//Check if the user has passed in a DSL if(dslFileName!=null){**

**//Load the rules, expanding them using the DSL Specified**

**InputStream dslSource = RuleRunner.class.getClassLoader()**

**.getResourceAsStream(dslFileName);**

**//We must be able to find all rule files if(null==dslSource){**

**throw new FileNotFoundException**

**("Cannot find dsl file:"+dslFileName);**

**} else {**

**log.info("found dsl file:"+ dslFileName);**

**}**

**//Load the rules, using DSL addRulesToThisPackage.addPackageFromDrl( new InputStreamReader(ruleSource), new InputStreamReader(dslSource));**

**} else {**

**//Load the rules, no DSL addRulesToThisPackage.addPackageFromDrl( new InputStreamReader(ruleSource)); }**

**//if we've specified a ruleflow, add this to the package if(ruleFlowFileName!=null){**

**//Load the rules , expanding them using the DSL Specified**

**InputStream ruleFlowSource =RuleRunner.class.getClassLoader()**

**.getResourceAsStream(ruleFlowFileName);**

**//We must be able to find all rule files if(null==ruleFlowSource){ throw new FileNotFoundException**

**("Cannot find dsl file:"+ruleFlowFileName);**

**} else {**

**log.info("found dsl file:"+ ruleFlowFileName); }**

**addRulesToThisPackage.addRuleFlow( new InputStreamReader(ruleFlowSource));**

**}**

**}**

Notable steps highlighted in this method are:

1. The Java method for locating and loading the **RuleRunner.class.**

**getClassLoader().getResourceAsStream(ruleFile)** file.Rather than being hardwired to a specific location, this file allows Java to search in the locality for files of the specific name. This is especially useful if this code is running as a part of a web application.

1. We try to load the rule file using this technique, and then throw an exception (error message) if the file cannot be found.
2. We check to see if a DSL has been passed in (**dslFileName!=null**). If yes, we call the Drools method **addPackageFromDrl(ruleSource,** **dslSource)**. If there no DSL file specifed, we call the more simple **addPackageFromDrl(rul eSource)** method.
3. We check to see if a ruleflow file has been specified (**ruleFlowFileName** **!=null**). If yes, we load and add it to the package by using the Drools method **addRuleFlow(** **ruleFlowSource)**.

The Excel equivalent of this method is **loadExcelRules**. This method takes the name of the Excel decision table that we wish to load, and the name of the package that we want to add it to.

This example doesn't accept a Ruleflow file to keep things simple. It is possible to use Ruleflow with Excel decision tables in a similar way as normal DRL rule files are used.

**private void loadExcelRules(String excelRuleFile,**

**PackageBuilder addRulesToThisPackage ) throws DroolsParserException, IOException{**

**//This method is more flexible in finding resources on disk**

**InputStream xlRuleSource = RuleRunner.class.getClassLoader(). getResourceAsStream(excelRuleFile); if(null==xlRuleSource){**

**throw new FileNotFoundException**

**("Cannot find file:"+excelRuleFile);**

**} else {**

**log.info("found file:"+excelRuleFile); }**

**//first we compile the decision table into a whole lot of rules. SpreadsheetCompiler compiler = new SpreadsheetCompiler();**

**String drl = compiler.compile(xlRuleSource, InputType.XLS);**

**//Show the DRL that is generated**

**log.debug(drl);**

**////same as previous - we add the drl to our package addRulesToThisPackage.addPackageFromDrl(**

**new StringReader(drl)); }**

**}**

This method also finds the named Excel file by using the **getResourceAsStream** **(excelRuleFile)** technique. However, there are two new lines (highlighted). They are explained as follows:

1. We create a **SpreadsheetCompiler** helper object (kindly provided by the Drools project).
2. We call the **compile** method on this helper object to transform the Excelbased decision table into the DRL format, which we are familiar with. Just to prove that our rules are now in the DRL format, we even print them to the screen by using the **log.debug** statement.

Once done, we add the rules to the package provided, which is similar to what we did for the DRL source rules.

### Public methods

All of these methods are going on 'under the covers', because the examples from previous chapters don't use them directly. In fact, unless you want to modify the behavior (for example, to add Ruleflow to rules based on decision tables), you might never need this amount of detail. What you will do in your project, like in our examples, is call one of the following methods:

* **runStatelessRules**
* **getStatefulSession**

As with the samples in the previous chapters, stateless rules are simpler as they fire once, and then give you a result immediately. Stateful rules, though more complex, are good for long-lived rules where we will update records and get results multiple times.

#### Stateless

The **runStatelessRules** method takes the following parameters:

* An array of text names—these are the rule files that the sample will search for, and then load.
* The **dslFileName** as text (string), or null if it is empty.
* The facts that we wish to pass to the rule engine—a collection of JavaBeans.
* A collection of global objects, such as name/value pairs, and so on.
* The **ruleFlowFileName** as text (string), or null if it is empty.
* A handle to the logger helper object. This allows the rules to print out what is happening.

Remember that the difference between facts and globals is that facts are designed to be updated as a result of business rules firing (and will cause other rules to fire in turn). Globals, on the other hand, are more for 'reference' data that may or may not change.

To see an example of how this method is called, look at the **ExcelDataRulesExample**

again. (It's part of the samples in Chapter 8.) The complete details of the **runStatelessRules** method:

**/\*\***

* **Run the rules**
* **@param rules - array of rule files that we need to load**
* **@param dslFileName - optional dsl file name (can be null)**
* **@param facts - Javabeans to pass to the rule engine**
* **@param globals - global variables to pass to the rule engine**
* **@param logger - handle to a logging object**
* **@throws Exception**

**\*/**

**public void runStatelessRules(String[] rules, String dslFileName,**

**Collection<Object> facts,**

**HashMap<String,Object> globals,**

**String ruleFlowFileName,**

**ILogger logger) throws Exception {**

**RuleBase masterRulebase=**

**loadRules(rules,dslFileName, ruleFlowFileName);**

**//Create a new stateless session**

**StatelessSession workingMemory =**

**masterRulebase.newStatelessSession(); for (String o: globals.keySet()){**

**log.info( "Inserting global name:"+o+" value:"+globals.get(o)); workingMemory.setGlobal(o, globals.get(o)); }**

**//Add the logger**

**log.info("Inserting handle to logger (via global)"); workingMemory.setGlobal("log", logger);**

**log.info("==== Calling Rule Engine ====");**

**//Fire using the facts workingMemory.execute(facts);**

**log.info("==== Rules Complete ====="); }**

The key features of this method (highlighted above) are:

1. Load the rule, or DSL, or ruleflow file, by using the **loadRules()** helper method.
2. From the loaded **RuleBase**, create **StatelessSession**. This is a one-time scratchpad based on the rules, which is unique to this session.
3. Into this **StatelessSession**, add the globals in a loop (starting at the **for** … line). The logger helper object is also added as a global.
4. Fire the rules, passing in the facts using the Drools method **workingMemory. execute** **(facts)**.

When the rules fire, the facts (JavaBeans) are updated. So when this method finishes, the rest of our Java code has access (via these JavaBeans) to the 'answers' that the rule engine comes up with.

#### Stateful

The stateful method takes almost the same set of parameters as does the stateless one. But there are two points of difference: we don't pass in the facts to it, and the stateful method returns a value (whereas the stateless method returns nothing). The returned value is **StatefulSession**. By returning this, we have a handle to the memory. This handle gives us the state of the rules, and allows us to come back to update that state with new facts later, if needed.

**/\*\***

* **Run the rules**
* **@param rules - array of rule files that we need to load**
* **@param dslFileName - optional DSL file name (can be null)**
* **@param ruleFlowFileName - optional (can be null)**
* **@param facts - JavaBeans to pass to the rule engine**
* **@param globals - global variables to pass to the rule engine**
* **@param logger - handle to a logging object**
* **@throws Exception**

**\*/**

**public StatefulSession getStatefulSession(String[] rules,String dslFileName, String**

**ruleFlowFileName,**

**HashMap globals,**

**ILoggerlogger) throws Exception {**

**RuleBase masterRulebase=loadRules**

**(rules,dslFileName,ruleFlowFileName);**

**//Create a new stateful session**

**StatefulSession workingMemory = masterRulebase.newStatefulSession(); for (String o : globals.keySet()){**

**log.info("Inserting global name: "+o+" value:"+globals.get(o) ); workingMemory.setGlobal(o, globals.get(o)); }**

**//Add the logger**

**log.info("Inserting handle to logger (via global)"); workingMemory.setGlobal("log", logger);**

**return workingMemory; }**

Two differences are highlighted in this method. Firstly, we use **newStatefulSession()** to get a working memory (scratchpad). This stateful session can be called more than once.

Secondly, the method is actually shorter than its stateless equivalent. The reason is that we set up the rules, but don't fire them. So if we're using this method (as in the **RuleflowExample.java** file from Chapter 9), you'll see that you have to do the hard work in the Java code that calls this method.

This 'hard work' is familiar, and explains why we don't have to pass in the facts. We have to insert ourselves in the calling method. An example of the calling code is as follows:

**Collection<Object> allRangeValues = ranges.getAllRangesAndCells(); for (Object fact: allRangeValues){**

**statefulSession.insert(fact);**

Finally, we also need to remember to call the rules:

**statefulSession.fireAllRules();**

### Alternative method—RuleAgent

Drools provides an alternative method for loading the rules, based on the **RuleAgent** (found in the **org.drools.agent** package). The **RuleAgent** helper can make your code simpler, as long as you follow the **RuleAgent** way of deployment. However, you don't get as much flexibility as you do with the code samples above. Typically, this means that pre-compiled (binary) packages of rules fit quite well with the way the Guvnor web editor works.

The way the **RuleAgent** works is through code plus a properties file. Your code calls the **RuleAgent** helper, similar to the following (remember that you will have **import org.drools.agent.RuleAgent** at the top of the file):

**RuleAgent ra = RuleAgent.newRuleAgent("/RuleAgent.properties");**

**RuleBase rb = ra.getRuleBase();**

**// now get a stateful or stateless sessions from RuleBase as before**

The **RuleAgent.properties** text file referred to in the above code snippet looks similar to the following:

**poll=60**

**NewInstance=false name=SomeNameUsedInLogging dir=/my/dir file = alternativeToDir**

**url=http://myfirstUrl http://anotherURLIfNeeded localCacheDir=file://c://temp**

Most of the options in the properties file are straightforward, as follows:

* **Poll** **=** **60**: This causes the RuleAgent to check for updates every 60 seconds.
* **NewInstance**: If this is true, and there is an update to the rules, then only new sessions that are created will pick up the new rules. The previous sessions will remain unchanged.
* **name**: This is the name that is used in logging.
* **dir**, **file**, or **url**: The location where the binary package of rules can be located. **url** can be used to point directly to Guvnor.
* **localCacheDir**: The **RuleAgent** will keep a local copy in case the remote URL is temporarily unavailable.

### Web deployment

Normally, you'd expect that we will take what we've just learned and use it to deploy an example on the Web. However, you could be using one of the hundreds of Java web frameworks. Each of those web frameworks would be worthy of a book in itself. Instead, we'll provide a set of notes about what to expect. These notes are not meant to be exhaustive, but to give you an idea of the steps involved in deploying Drools as a part of the web framework of your choice.

Each of these frameworks has three major points of integration with JBoss Rules. These are as follows:

* How do we package the project (with Rules and the Drools Libraries) so that it gets deployed to the web server?
* In which folder do we save the rules files in the web project?
* Where in the web framework do we call the rules code (seen previously in this chapter)?

For our walkthrough, we'll use the Appfuse framework (which uses Spring under the covers). The quick run-through also assumes that you have the MySql database installed from **http://dev.mysql.com/downloads/**—the community edition will do fine. The next few pages assume some knowledge of Java web development due to the complexity of the subject.

Why Appfuse? Underneath, it uses the popular Spring MVC framework. But it gives you a choice of web frameworks. It also generates most of the applications automatically, for immediate results—a bit like Ruby on Rails. More information on the Appfuse project is available at **www.appfuse.org**.

#### Maven for packaging

Previous chapters have used Maven as a build tool, mainly for setting up Eclipse projects. It can do a lot more, including pulling down Appfuse from the Web, creating the package structure for the project, and deploying the code to the web server. This can be seen with the help of the following steps:

1. Open a command window. Use Maven to download Appfuse, and create a standard project.

**mvn archetype:create -DarchetypeGroupId=org.appfuse.archetypes-**

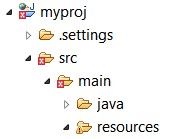
**DarchetypeArtifactId=appfuse-basic-spring -DremoteRepositories**

**=http://static.appfuse.org/releases -DarchetypeVersion=2.0.2 -**

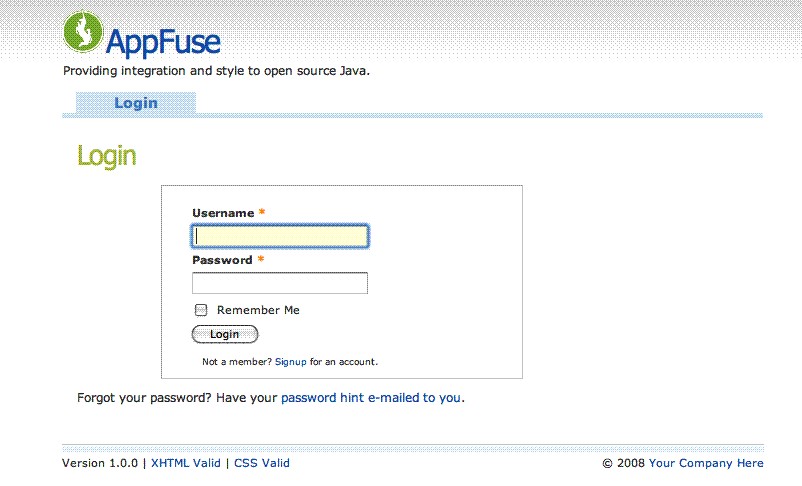
**DgroupId=net.firstpartners.chap10 -DartifactId=chap10-sample**

1. Change the director to the folder created in the previous step using the **cd chap10-sample** command.
2. Use Maven again, this time to generate a project skeleton, with the command **mvn** **war:inplace**.
3. Create an Eclipse project by using **mvn** **eclipse:eclipse**, and then open the project in Eclipse.

In the project structure, you'll see the place where we can save our rules files and other resources, under the **resources** folder, as shown in the screenshot below. By default, Maven expects this standard structure. The following figure shows the RuleFile location:



1. Maven can also run a test web server **- mvn** **jetty:runwar**. When the message **[INFO] Started Jetty Server [INFO] Starting scanner at interval of 3 seconds.** appears in the console, open a web browser at **http:// localhost:8080/login.jsp**. You will see the following web screen, or sample app, generated by Appfuse. If you want to log in and play around, enter **admin** as both the **Username** and the **Password**.



1. Stop Maven (by pressing *Ctrl+C* in the command window), when you're finished playing with it.
2. Now that we have a web application up and running, it's time to start adding Drools. In Eclipse, we'll modify the **pom.xml** file so that the web project knows where to find the Drools libraries. Rather than supplying the full details here, just copy and paste the **<dependencies>** section from the **pom.xml** file in the sample from Chapter 8.
3. Next, we need to add the code that calls JBoss Rules (that is, the code we saw previously in this chapter and in the samples). Typically, this is in the Java file (controller of the model-view-controller pattern) where the values submitted by the user on the Web have already been converted into JavaBeans. By happy coincidence, these JavaBeans are very suitable for passing into the rule engine.
4. More details on how to add a controller for Appfuse-Spring-MVC can be found at **http://www.appfuse.org/display/APF/Using+Spring+MVC**.

The file that we will modify according to these instructions is **PersonFormController**. In summary, this file (**PersonFormController**) extends another file (**BaseFormController**), and adds one method (that we will use to call the rules):

**public ModelAndView onSubmit(HttpServletRequest request, HttpServletResponse response, Object command,**

**BindException errors)**

**throws Exception {**

**log.debug("About to call Rules..."); //Place the code to call rules here, exactly**

**as previous examples**

**return new ModelAndView(success); }**

1. Within the updated **PersonFormController**, add the code that calls the rule engine—a copy-and-paste job from the examples in Chapter 9 should work. The trickiest part for this code is the rules (DRL) file. However, as long as the rules file is included in the Web Archive (WAR) file that contains the application, the code in our previous sample, **getClassLoader().getResourceAsStream(ruleFile)**, should find it.
2. Generate the web application ZIP file by using **Mvn** **War**, and deploy it to your favorite web application using the instructions that come with it.

Remember—this isn't meant to be a complete web application development tutorial. Rather, it's meant for people familiar with Java web development to be able to add rules to their applications. If you're not familiar with web development, it may give you some idea of the required steps.

So, let's answer the three big previous questions, here:

* + How to package the project? The answer is: Use Maven standard functionality (or an equivalent from another Java build framework) with the Drools libraries specified as a part of the build script (for example **pom.xml**).
  + In which folder do we save the rules files? The answer is: In the **resources** folder of the Java project.
  + Where do we call the rules code? The answer is: We call it from the Java code, typically the controller class. If you followed the instructions that come with Appfuse, whenever you click the **Submit** button on the web browser, the framework will eventually cause the above rule code to be executed.

Look again at the 'one size architecture fits all' diagram at the start of this chapter. As promised, the code we have just seen fits right in the middle of the picture, in the **Business Layer** of the system.

## Summary

This chapter has shown you how to deploy your business rules in the real world. We looked at the pieces that made up an entire web application, and where our rules fit into it. We saw the various options for deploying rules as a part of our application, and also the team that's involved in the process. Once deployed, we looked at the code that will load and run the rules—both home-grown, and those created using the standard RuleAgent. Finally, we saw how to combine this into a web project using the framework of your choice.