developerWorks

Accessing external data in a rules application

Skill Level: Intermediate

Raj Rao (rrao2@us.ibm.com)
ILOG Solution Architect
IBM

15 Feb 2012

Although it is generally recommended that external application data not be accessed from a rule application deployed to a WebSphere® rule execution component, there are certain situations that warrant it. This article describes these situations, and compares options for handling them. An example scenario offers practical, hands-on techniques.

Introduction

A decision service is a reusable service operation that makes decisions, and in the context of this article, is implemented using business rule engine technology. Decision services are part of business service layer in the IBM SOA reference architecture and the design of such services should follow the best practices of loose coupling, coarse-grained operations, and minimum payload. Business rules are deployed as rulesets to the JRules Rule Execution Server (RES), which are then used by the decision service. Typically, business users manage business rules, but the decision service is entirely controlled by the IT group.

Rulesets are executed in the RES using a rule session API, which enables clients to acquire rule sessions and execute rules within that session. A decision service is a client for the rule session, but in non-SOA architecture, a web application can be the client that directly uses the rule session API from a servlet. This article explores data access options for a rule application that covers the markedly different options for data access from a web application rule client, decision service or directly from business rules.

An ideal rule session is stateless with no external dependencies. Therefore it is generally recommended that a JRules ruleset not access any reference data or

transactional data from a database or service during rule processing. Not performing data access from a ruleset during rule execution yields some key benefits:

- Better Rule Execution Server (RES) performance.
  Typically, accessing external data takes much longer than the rest of the rule execution and therefore has a big impact on the total processing time. Since each rule engine runs on a single thread, this thread is blocked during data access. Therefore, it is very likely that the rule server does not make full use of the CPU for most of the time. Additionally, the maximum number of external connections becomes another potential bottleneck for the size of the rule engine pool. Rule engines that do not perform data access require fewer RES servers for the same transaction load.

- Simpler exception handling:
  It is always possible that calls to external systems will fail, either because the external service is down or due to some network issue. When this happens, an exception is thrown. Not performing data access from the rule engine frees it from the responsibility of handling these exceptions.

- Easier unit testing:
  When the request payload is self-contained and defines all the data that the rule engine needs to make a decision, there is no dependency on any external service or database. This decoupled architecture means that changes to the external data source or data service have no impact on the rulesets. Because there are no dependencies, the ruleset is easier to unit test and regression test as an independent unit.

There are several situations that warrant runtime data access from a rule application. For example, if some rule processing configuration parameters are maintained in a database, a simple lookup could obtain this configuration information from the database. A more complex lookup could involve a transactional database; for example, a claim processing application may choose to simply send a claim id to the claim adjudication decision service and expect it to look up the claim history from a transactional database. There may also be cases when reference data needs to be looked up from the database. For instance, in a ruleset that manages vehicle inventory, the time period between oil changes depends on reference data stored in the database relating to the vehicle category, make, model and year. Another common situation is when external data is looked up during validation. For instance, claim validation rules may include a check to ensure that the claim procedure code is valid for the supplied claim diagnosis code. There may be tens of thousands of these records in a reference database that associate procedure codes with validation codes, which would then need to be accessed by this rule.

The source of data may be varied. It could be a database, a web service, another decision service or even a long-running process. Some of these data accesses may be expensive in terms of both time and money.

Another important dimension of differentiation is how dynamic the requests are. There are some situations where the parameters used for data access are computed by rules within a ruleset. These data requests are termed *dynamic* for the purposes of this article. An example of a dynamic data access request is a discount calculation ruleset that computes a loyalty rating and then uses that and the customer zip code to look up the discount rate from a database.

This article describes the various alternatives to handle these data lookup requirements, evaluates each one and provides an overall comparison.

Prerequisites

This article is written for the intermediate to advanced JRules technical developer and focuses on a specific area of implementation. It is assumed that the reader has a good understanding of the ILOG JRules product from a developer's perspective. Please refer to the Resources section for more information.

The product versions used in this article are:

- WebSphere ILOG Rule Studio V7.1.x
- WebSphere ILOG Rule Execution Server V7.1.x
- WebSphere Application Server V7.0
- Rational Application Developer V7.5 or Rational Software Architect V7.5

Note: The techniques described in this article are also applicable to the new version of JRules, WebSphere Operational Decision Management V7.5.

Scenario overview

For this article, we start with the tutorial that is provided with JRules V7.x called Tutorial: Creating a web application to call JRules on IBM Rational Application Developer and extend it to call an external web service. To fully understand the scenario, it is recommended that you go through this tutorial, or at least browse through it. Although the tutorial is specific to IBM Rational Application Developer, the data access options presented in this article are equally applicable to other IDEs and even other application server deployments. The tutorial uses a Java® EE rule session to execute a simple loan processing ruleset. As shown in Figure 1, this ruleset takes three parameters: a borrower (input), a loan (input and output) and a

report (output).

Figure 1. Ruleset parameters

Ruleset Parameters

Define ruleset parameters.

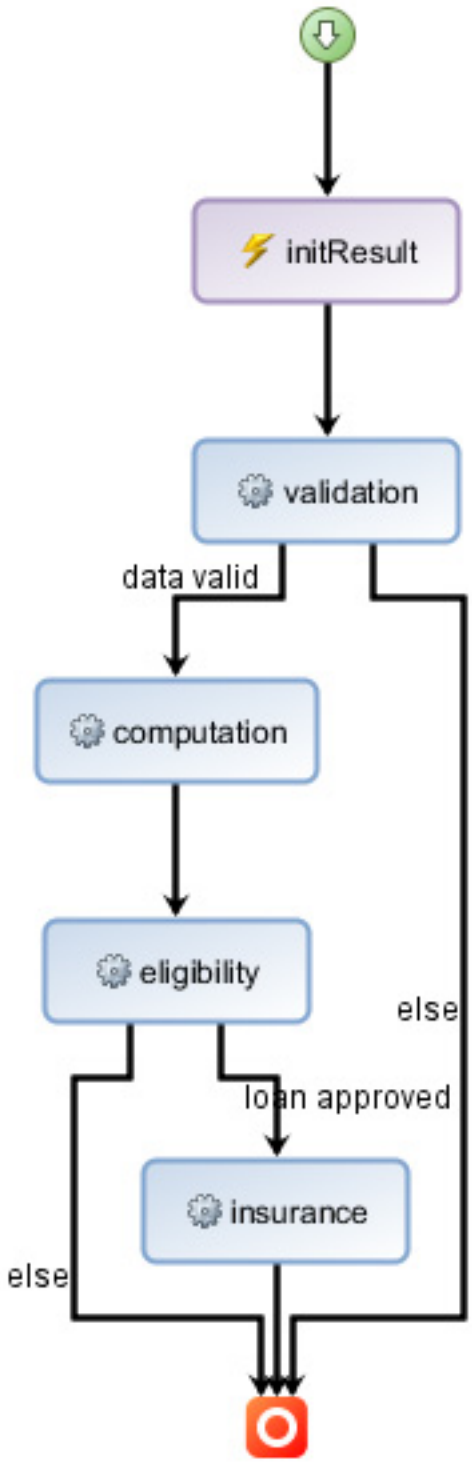
Name	Type	Direction	Default Value	Verbalization
borrower	loan.Borrower	IN		the borrower
loan	loan.Loan	IN_OUT		the loan
report	loan.Report	OUT		the loan report

As noted in the tutorial, the loan validation application:

1. Validates input data from a web application.
2. Calculates customer eligibility based on their personal profile, their score, and the requested loan amount.
3. Evaluates specific criteria or score to accept or reject the loan.
4. Computes the insurance rate if the loan is accepted, from a function of the computed score.

This is implemented using the ruleflow shown in Figure 2.

Figure 2. Loan processing ruleflow



New business policies

Now, imagine that the loan processing company has decided to tighten eligibility restrictions given the uncertain macroeconomic conditions prevailing today. They now have additional business policies that determine eligibility by checking an *employment stability index* – a newfangled index compiled by an external employment tracking agency. Specifically, the new business policies are:

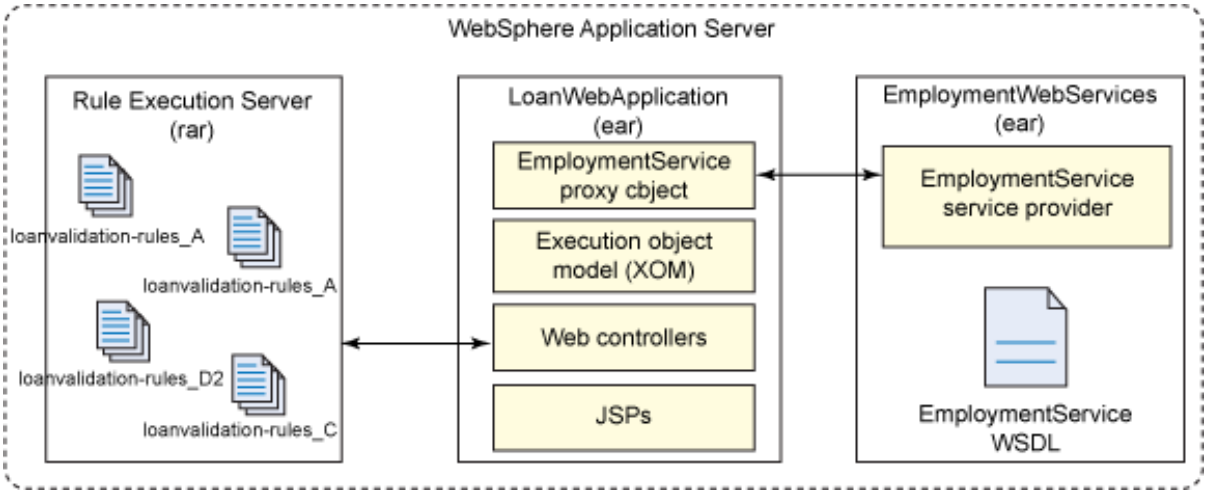
- Employment stability index must be at least 72 if the corporate score is less than 600 and the loan amount is more than $50,000.
- Employment stability index must be at least 60 if the credit score is less than 780 and the loan amount is less than $80,000.
- Employment stability index must be at least 64 if the credit score is less than 830 and the loan amount is more than $80,000.

The employment stability index for an applicant is obtained by accessing an external Employment Service. This service accepts the social security number (SSN) as input and returns an `EmploymentRecord` object. There is a cost associated with making this web service call, so these calls should be kept to a minimum.

Architecture

For this article, the Employment Service is implemented as a simple web service that returns a random value between 25 and 100 for the employment index. A web service client proxy is used in the `LoanValidationWeb` application to access this web service, as illustrated in Figure 3. Standard wizards in Rational Application Developer are used to develop the web service and the client proxy, the details of which are not important in the context of this article and are therefore omitted here. The main architectural execution modules, as shown in Figure 3, are the web application that gathers loan data, the Rule Execution Server (RES) and the employment web service.

Figure 3. WebSphere execution components



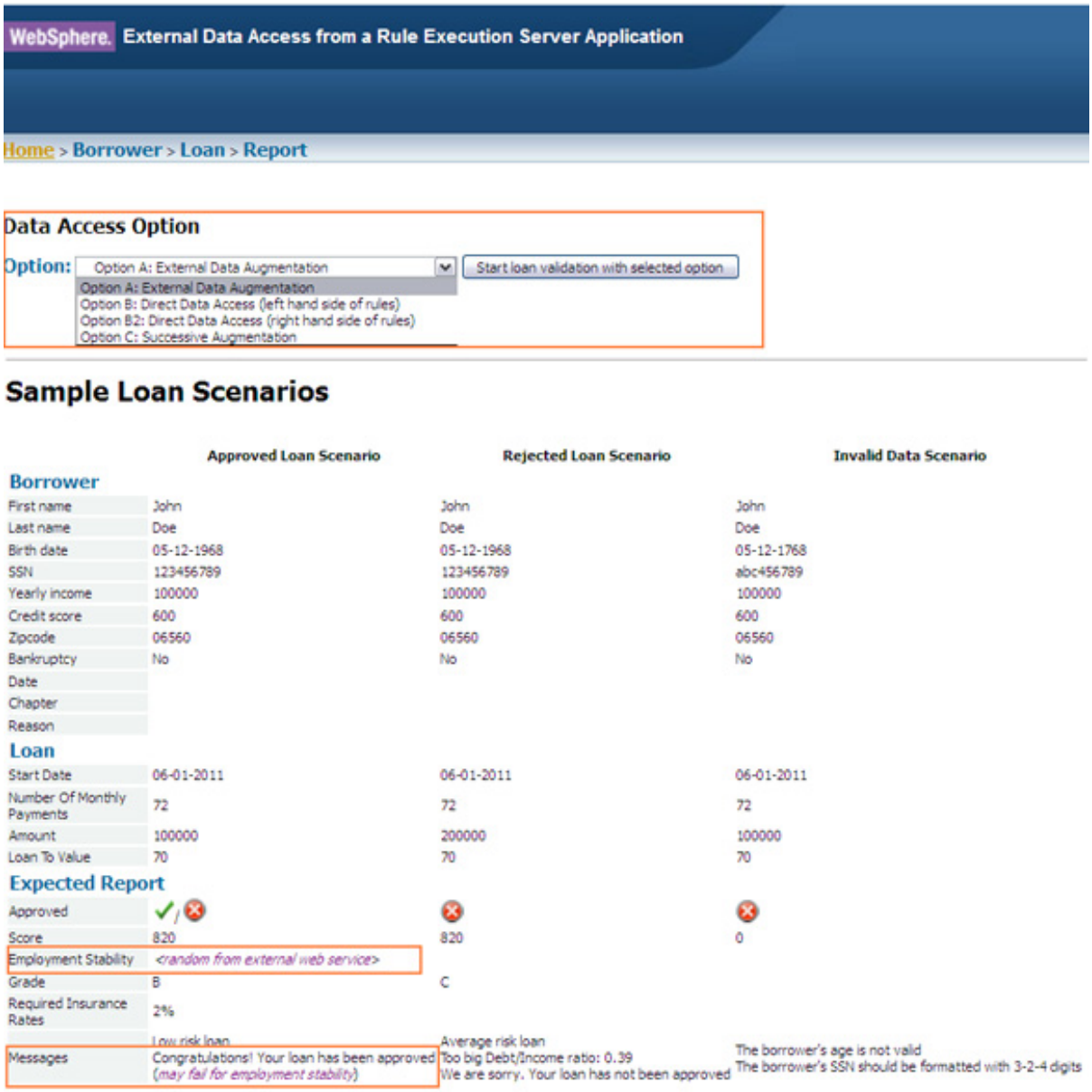
The four rulesets in the Rule Execution Server in Figure 3 are the rule implementations corresponding to the four data access options discussed in this article. These options are:

- Option A: External request augmentation.
- Option B: Direct data access initiated from the lefthand side of rules.
- Option B2: Direct data access initiated from the righthand side of rules.
- Option C: Successive request augmentation.

This article will describe these options in detail. Each of these rulesets is invoked by a corresponding controller class defined in the loan validation application. Unlike in the original tutorial, controllers are not embedded in JSPs but are defined as Java classes.

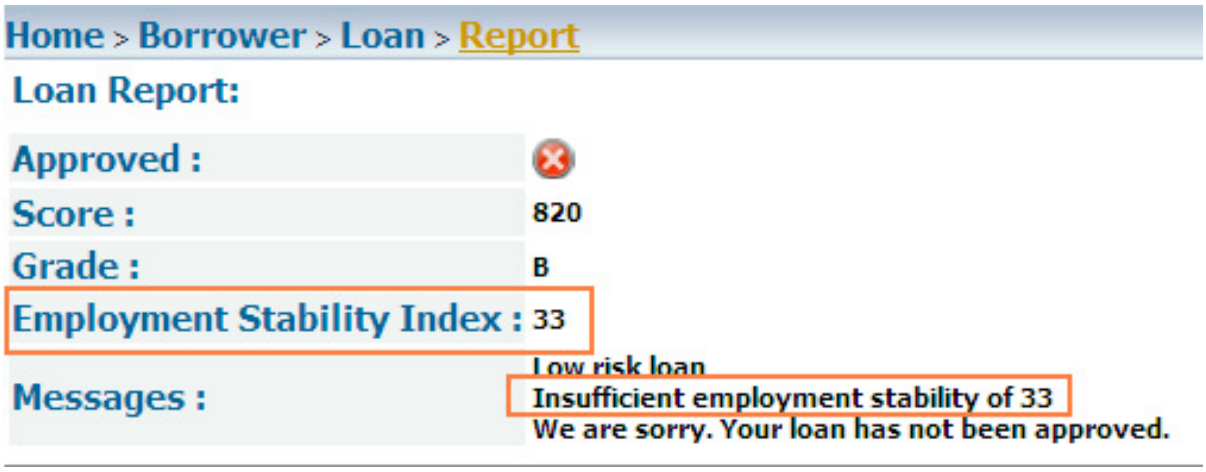
The web application user interface is modified from the original tutorial to include controller options and the employment stability index, as shown in Figure 4. Check the tutorial documentation for details on how to build, deploy and access the web application. The URL is slightly modified from the original tutorial, which by default is deployed as http://localhost:9080/LoanValidationWeb/.

Figure 4. Loan validation web application



Rules determine whether to approve or deny the loan based on loan data, borrower data and employment stability. After the processing is completed, a report is displayed that includes a set of messages explaining the reasons for approval or denial of the loan application. Figure 5 shows an example where the loan was denied due to insufficient stability index. These messages are generated in the rule engine.

Figure 5. Loan processing result

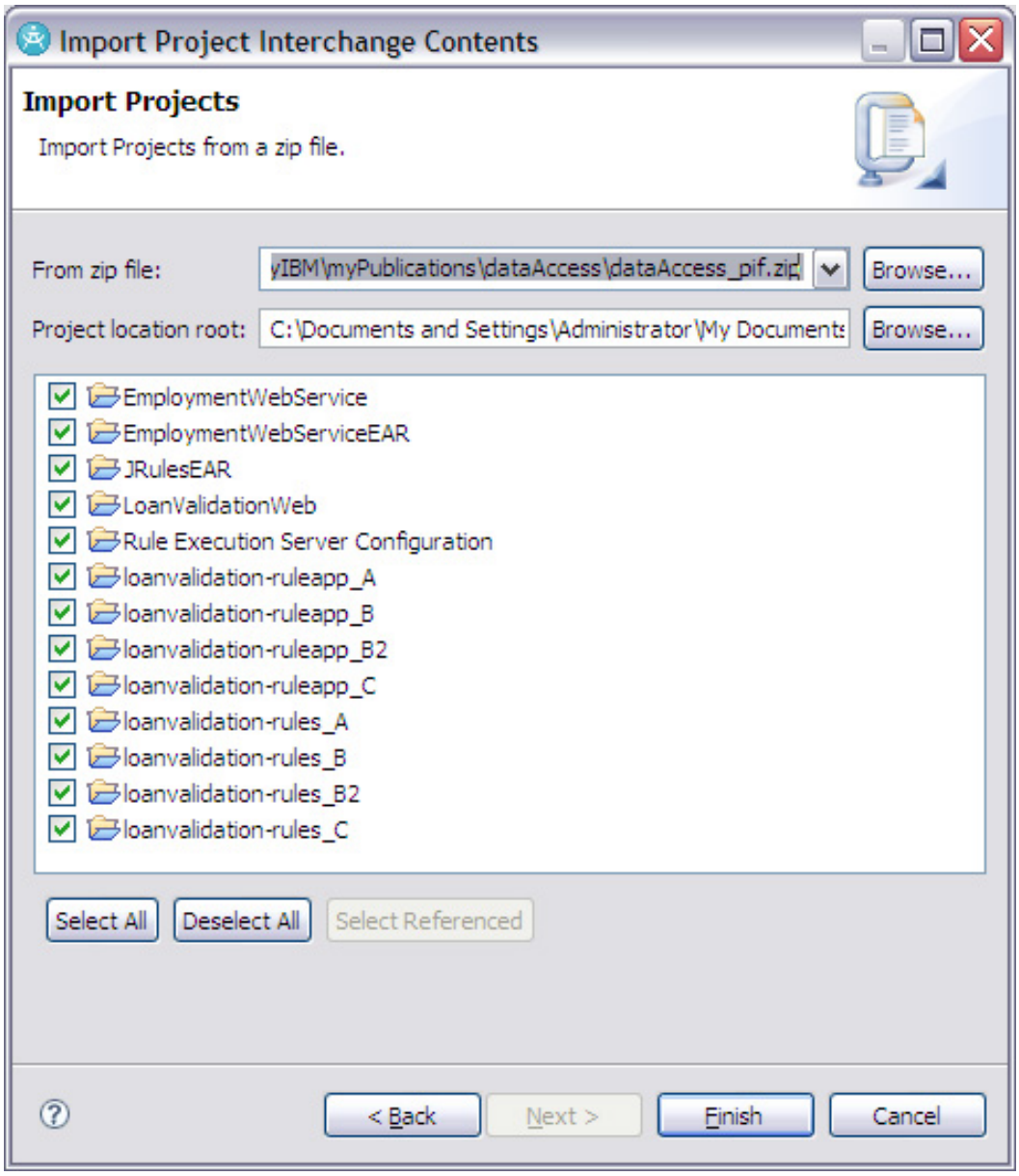


Download material

In this article, only the relevant details of the workspace are discussed. However, if you have Rule Studio V7.1.x installed along with Rational Application Developer V7.5, you can download the entire workspace and browse through all the details as a supplement to this article. See Resources for instructions on installing Rule Studio on Rational Application Developer. The workspace containing the different implementation options is provided for download with this article.

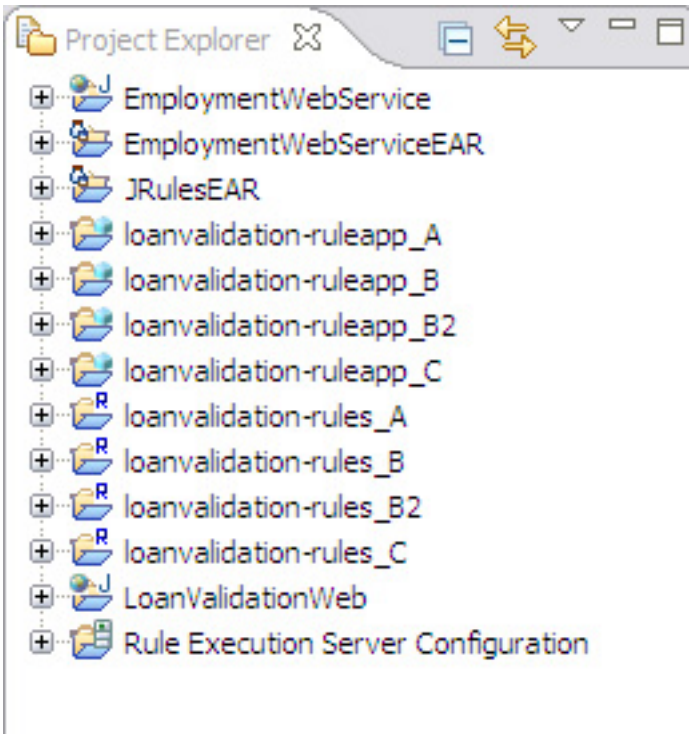
To import the projects, follow these steps:

1. Download the dataAccess_pif.zip file to your local disk.
2. Open Rational Application Developer V7.5 or Rational Software Architect V7.5 and create a new workspace.
3. Select **File => Import**.
4. In the Import dialog, select **Project Interchange** and click **Next**.
5. In the Import Project Interchange Contents dialog, select the .zip file that you just downloaded, click **Select All** and then click **Finish**, as shown in Figure 6.
   Figure 6. Import project



All the projects will now be imported into the workspace as shown in Figure 7.

Figure 7. Imported projects



6. You may get JSP errors, for example in footer.jsp because it is an imported JSP file that is not complete. To eliminate the error, select **Window => Preferences**. On the Preferences page, select **Validation** in the left pane and deselect **JSP Content Validator**. Click **Apply** then **OK**. When you rebuild the project, the JSP errors will not appear.

Our example uses separate projects to illustrate each of the implementation options. Table 1 shows the rule and ruleapp projects associated with each implementation. Each rule project contains the specific rule implementation and includes its corresponding business object model (BOM). The ruleapp project is simply the standard method for deploying the rules to RES.

Table 1. Rule and ruleapp projects for each option

Implementation Option	Rule Project	Ruleapp Project
Option A	loanvalidation-rules_A	loanvalidation-ruleapp_A
Option B	loanvalidation-rules_B	loanvalidation-ruleapp_B
Option B2	loanvalidation-rules_B2	loanvalidation-ruleapp_B2
Option C	loanvalidation-rules_C	loanvalidation-ruleapp_C

The other projects are common to all the options. Table 2 lists these projects.

Table 2. Project descriptions

Project	Description

EmploymentWebService
EmploymentWebServiceEAR
JRulesEAR
LoanValidationWeb
Rule Execution Server Configuration

Option A: Request augmentation

A simple option is one in which the rule request is augmented with external data through an external augmentation process. All data access is performed by the invoking application using either JDBC requests to external databases or SOAP calls to other web services and the external data is passed in with the request data, as depicted in Figure 8. From the perspective of the ruleset, the request is self-contained – in other words, the Execution Object Model (XOM) for the ruleset is the augmented request class, which includes the external data definitions.

Figure 8. Rule request augmentation

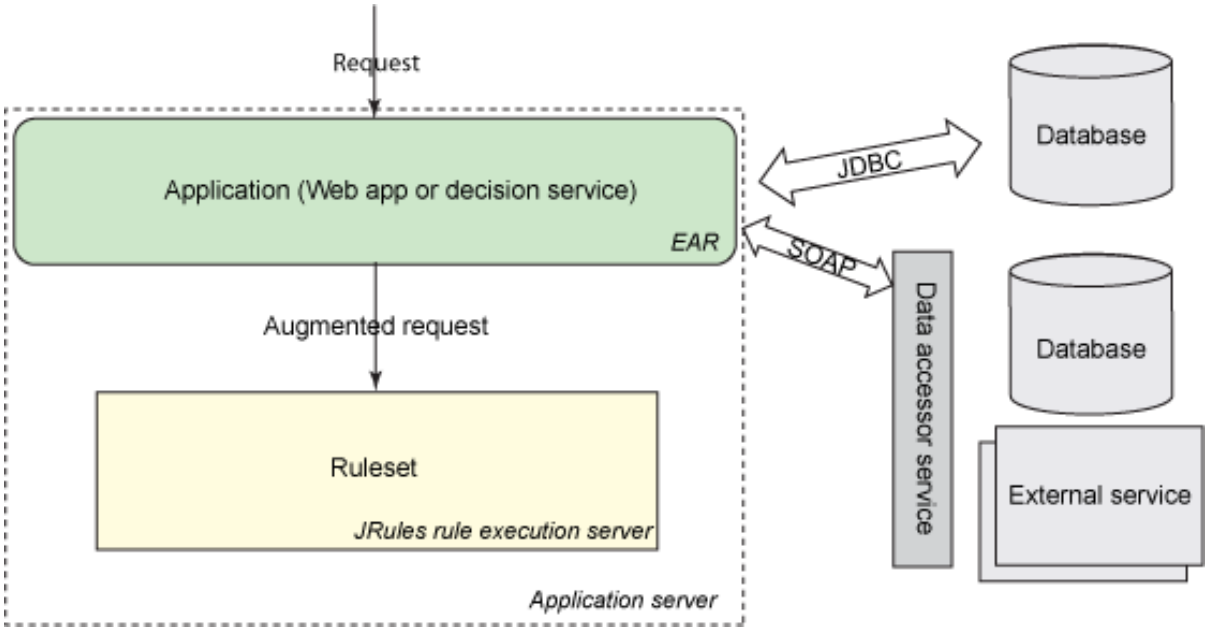
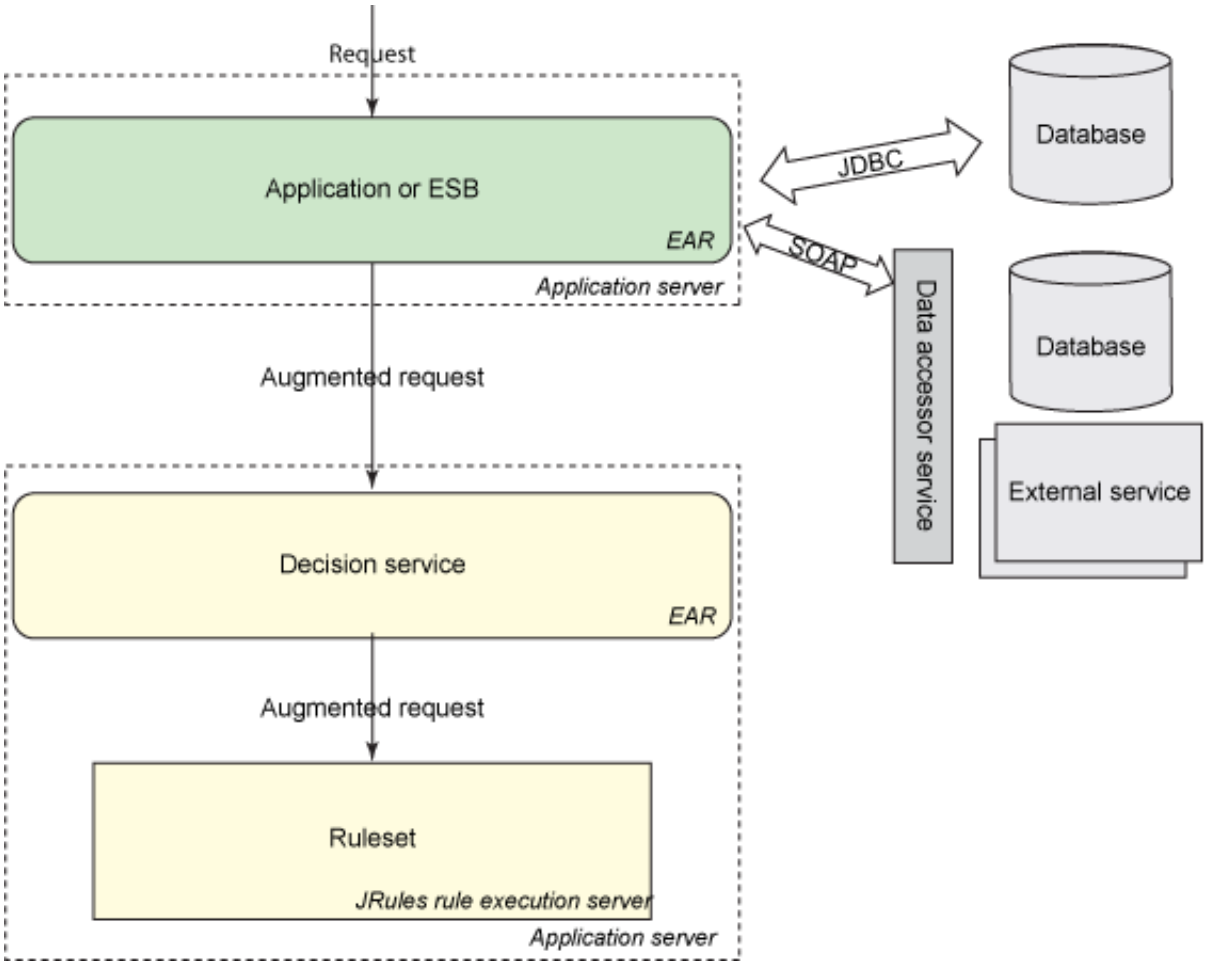


Figure 8 depicts the RES and the rule client, either a decision service or a web application, in the same application server JVM. It is also possible to reduce the load on the rule server by doing this augmentation on a remote server, or even on the enterprise service bus (ESB), as illustrated in Figure 9.

Figure 9. Remote augmentation



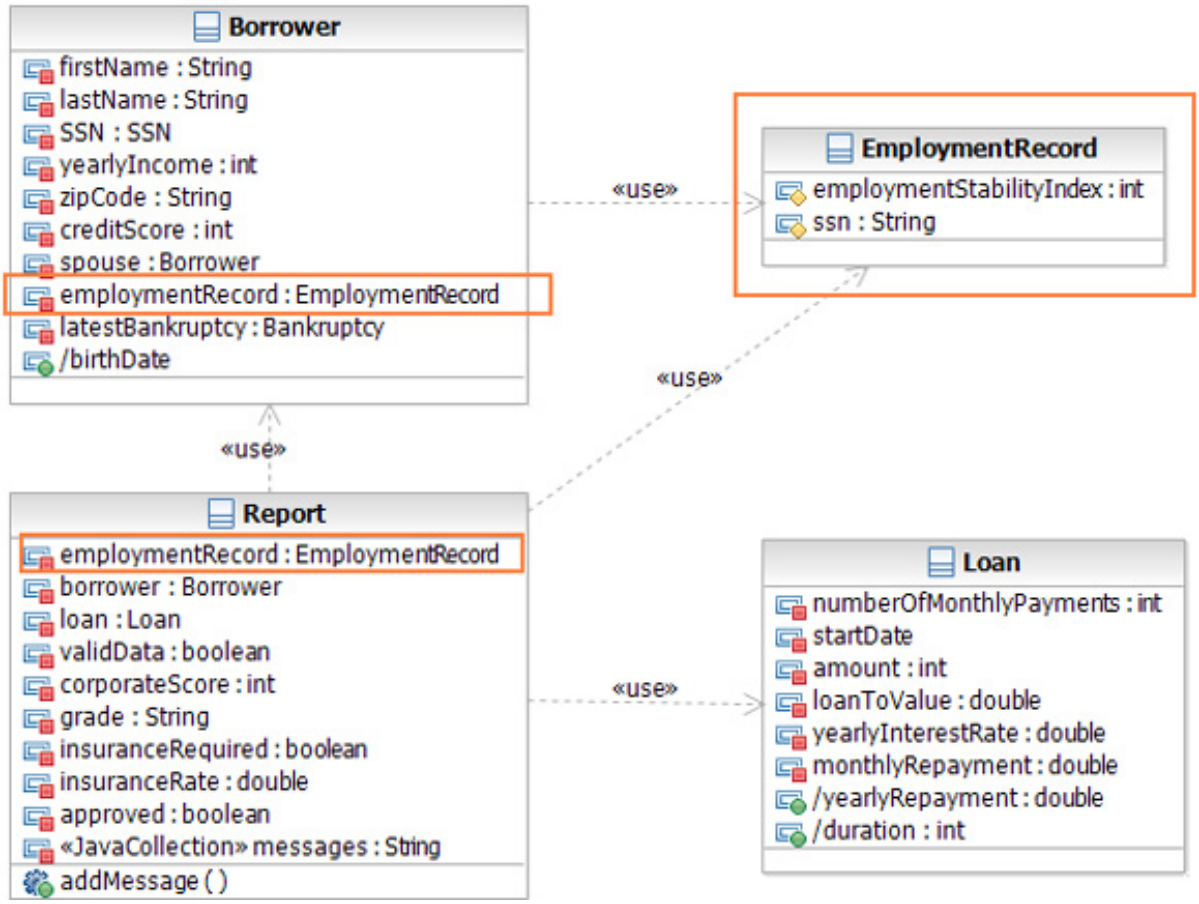
There may be an opportunity to improve data access performance by caching reference data at the application level or at the data accessor service, particularly if the data is accessed from a database. Depending on the size and usage characteristics of the data, this may be a partial cache that employs a strategy (such as First In First Out) to keep the most relevant data in cache. If the data is not too large, the entire reference data may be cached upon initialization. Usual tradeoff considerations between memory and performance apply. Of course, caching assumes that the data does not change frequently -- otherwise, a cache refresh mechanism becomes necessary. Therefore, it is more applicable to constant or slow-changing reference data.

Implementation details

As noted earlier, the XOM for the ruleset is the augmented request class, which includes the external data definitions. The XOM is therefore enhanced with a Borrower that has an associated `EmploymentRecord`, as shown in Figure 10. This `EmploymentRecord` class in turn contains the employment stability index that is used in the new business rules. The `Report` class is also augmented with the `EmploymentRecord` so that the stability index can be displayed as part of the final

report.

Figure 10. XOM augmentation



The responsibility of accessing the external web service to retrieve the employment record falls on the `LoanValidation` web application. In particular, the web controller (web.BusinessBeanController_A.java) has a method called `executeRulesOnPojoRuleSession` that accesses the Employment Web Service (line 12 below) and associates the `EmployeeRecord` with the Borrower (line 17), as shown in Listing 1.

Listing 1. web.BusinessBeanController_A.java

```
1. public Report executeRulesOnPojoRuleSession(String rulesetPath) throws
    IlrFormatException, IlrSessionException  {
2. IlrSessionRequest sessionRequest = pojoFactory.createRequest();
3. ExternalDataAccessorHelper dataAccessor =
        dataAccessorFactory.createExternalDataAccessorHelper();
4. Report report = null;
5.
6. sessionRequest.setRulesetPath(IlrPath.parsePath(rulesetPath));
7. sessionRequest.setForceUptodate(true);
8. // Enable trace to retrieve infos on executed rules
9. sessionRequest.setTraceEnabled(true);
10. sessionRequest.getTraceFilter().setInfoAllFilters(true);
```

```
11. // Set the input parameters for the execution of the rules
12. EmploymentRecord empRecord =
        dataAccessor.getEmploymentRecord(getBorrower().getSSN().toString());
13. if (empRecord == null) {
14. report = new Report(getBorrower(), getLoan());
15. report.addErrorMessage("Could not access employment profile. Please try again
                later");
16.  } else {
17. getBorrower().setEmploymentRecord(empRecord);
18. Map<String, Object> inputParameters = sessionRequest.getInputParameters();
19. inputParameters.put("borrower", getBorrower());
20. inputParameters.put("loan", getLoan());
21. // Create a stateless rule session
22. IlrStatelessSession ruleSession = pojoFactory.createStatelessSession();
23. // Execute the rules
24. IlrSessionResponse sessionResponse = ruleSession.execute(sessionRequest);
25. printResults(sessionResponse);
26. report = (Report) sessionResponse.getOutputParameters().get("report");
27.  }
28.  return report;
29. }
30.
31.
32. @Override
33. public Report executeRules() throws IlrFormatException, IlrSessionException {
34. return executeRulesOnPojoRuleSession("/loanvalidation_A/loanvalidationrules_A");
35.}
```

In this approach, exception handling is pretty straightforward. If there is an exception when accessing the employment web service, the employment record is null. When that happens, an error message is added to the report and the rule engine is bypassed (lines 13 – 15).

The rule execution uses the ruleset `/loanvalidation_A/loanvalidationrules_A` (line 34). For those following along with the downloadable code, these are the rules defined in the loanvalidation-rules_A rule project.

At this point, you may want to go back to section New business policies section to refresh your memory about the new business rules we set out to implement. Two business rules are added to the eligibility.employment rule package to implement the new policies relating to employment stability. One is implemented as a business action language (BAL) rule to check against corporate score and the other as a decision table to check against credit score. Listing 2 shows the BAL rule and Figure 11 shows the decision table. Essentially, these rules access the employment stability in the borrower in a straightforward manner as a simple attribute.

Listing 2. The BAL rule code

```
definitions
    set 'employment stability' to the employment stability index of the
employment record of 'the borrower';
if
    the amount of 'the loan' is more than  50000
    and the corporate score in 'the loan report' is less than  600
```

```
    and 'employment stability' is less than 72
then
    in 'the loan report', refuse the loan with the message "Insufficient
employment stability of " + 'employment stability' ;
```

Figure 11. Employment stability decision table

Loan Amount		Credit Score (min)	Employment Stability (min)	Message
min	max			
0	80,000	780	60	"Insufficient employment stability of " + the employm...
> 80,000		830	64	"Insufficient employment stability of " + the employm...

There is no change to the ruleflow in the ruleset, so the changes needed in the ruleset to incorporate these new business rules are very simple - a BOM update to get the new XOM elements (`EmployeeRecord`) and two new business rule artifacts.

On the negative side, the employment web service is invoked even in those situations when it is not necessary (for example, when loan amount is 30,000 and credit score is 800), which is wasteful. Moreover, if the employment web service is down, no loan is approved, even the ones where there is no need to check the employment stability index. Most importantly, while not the case in this scenario, this approach cannot handle dynamic data access. For example, if the web service invocation parameter was a computed value such as a corporate score derived during rule processing, then clearly it would not be feasible to employ this option.

Evaluation

Table 3 summarizes the pros and cons of this approach.

Table 3. Data augmentation evaluation

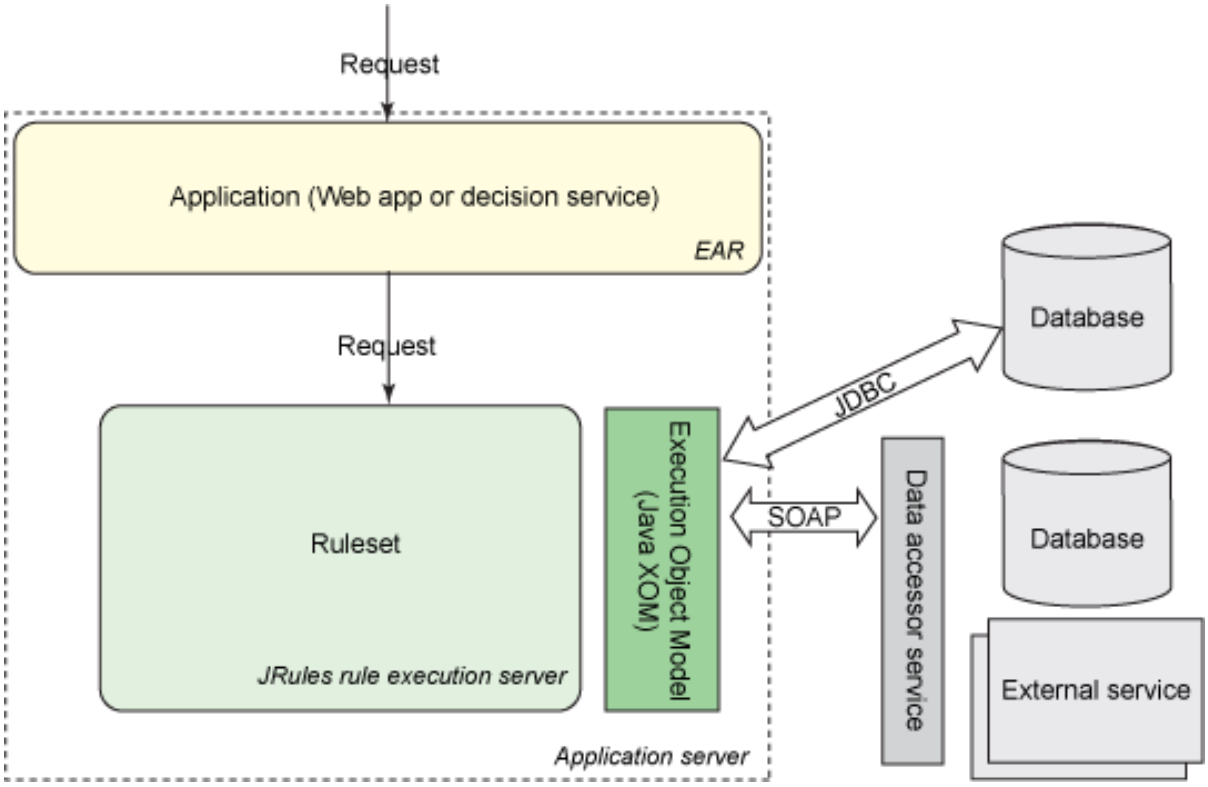
Pros	Cons
• Decision service is decoupled from other systems. • Easy to component test the rules. • Minimal impact on rule authoring. • Enables data access to be performed on a remote server, thereby reducing the CPU load on the RES server. • Allows for an XSD XOM, and therefore using Hosted Transparent Decision Service (HTDS) is a deployment option. • Augmentation can be delegated to an ESB mediator that is more	• Not feasible when there is truly dynamic data access requirements. • Since data access is not contextual, this can be wasteful if not all of the retrieved data is needed for processing that particular request. • If additional data elements become necessary as the application evolves over time, then the decision service signature needs to be altered if the data was remotely augmented. (as in Figure 9)

suitable for the task. This can even take advantage of data caching to reduce external access.

Option B: Direct data access during rule execution

A more direct option is to perform the data access as needed during rule execution. Here too, the data access can be a JDBC call to a database or a call to an external web service. The data access is performed using Java code. This Java code is part of the XOM and is invoked by rules during rule execution as shown in Figure 12.

Figure 12. Direct data access

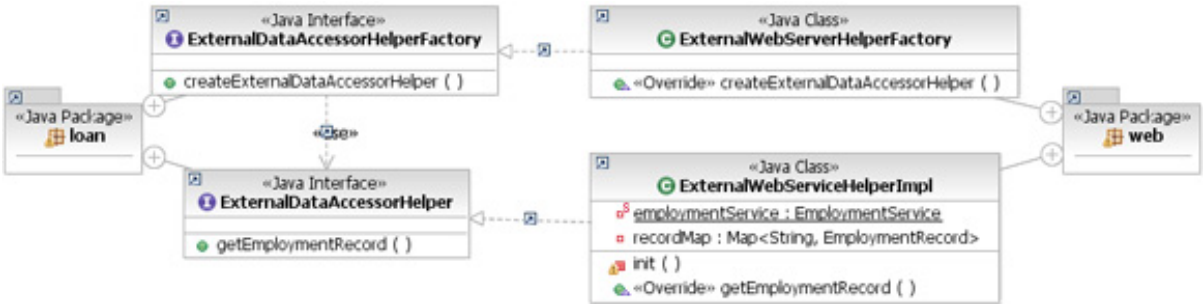


Implementation details

External data access is performed using Java classes in the XOM. Two key interfaces are added to the XOM to achieve this: `ExternalDataAccessorHelperFactory` and `ExternalDataAccessorHelper`. The factory is used to instantiate an `ExternalDataAccessorHelper`, which has a method called `getEmploymentRecord` to retrieve the employment record from the employment web service using a web service proxy.

These are defined as interfaces, and not as concrete classes, for two reasons: 1) to not clutter the BOM with unnecessary web service related classes, and 2) to enable a different implementation of this factory interface to be used for unit testing. In Figure 13, we see the `ExternalWebServerHelperFactory` and `ExternalWebServerHelperImpl` concrete classes implement these interfaces, using the classic factory design pattern.

Figure 13. External data accessor Java classes



(See a larger version of Figure 13.)

`ExternalWebServerHelperImpl` invokes the web service and stores the result in a local cache, as shown in Listing 3. In this manner, if the data access request is issued again by rules during the processing of a request, it can retrieve the value from the local cache (lines 6 -7) and does not have to make the external web service call.

Listing 3. Data accessor implementation

```
1. @Override
2. public EmploymentRecord getEmploymentRecord(String ssn) {
3. EmploymentRecord record = null;
4.
5. // check if it is already fetched during this session
6. record = recordMap.get(ssn);
7. if (record == null) {
8.   // access web service
9.   try {
10.         EmploymentServiceDelegate port =
                        employmentService.getEmploymentServicePort();
11.         record = port.getRecord(ssn);
12.         recordMap.put(ssn, record);
13. } catch (Exception e) {
14.         e.printStackTrace();
15.                 }
16. }
17. return record;
18. }
```

The two interfaces, `ExternalDataAccessorHelperFactory` and `ExternalDataAccessorHelper`, are imported into the BOM. Now rules may invoke the `getEmploymentRecord` as necessary. But how does the ruleset get the `ExternalDataAccessorHelperFactory`? It is passed in as an input parameter, as shown in Figure 14.

Figure 14. Ruleset parameters including the data accessor factory

Ruleset Parameters

Define ruleset parameters.

Name	Type	Direction	De...	Verbalization
borrower	loan.Borrower	IN		the borrower
loan	loan.Loan	IN_OUT		the loan
report	loan.Report	OUT		the loan report
dataAccessorFactory	loan.ExternalDataAccessorHelperFactory	IN		

The data accessory factory is not verbalized as it is not directly used in rules. What is used in rules is the data accessor, which is defined as a ruleset variable and verbalized, as shown in Figure 15.

Figure 15. Data accessor ruleset variable

Variable Set: loanValidationVars

Name	Type	Verbalization
dataAccessor	loan.ExternalDataAccessorHelper	the data accessor

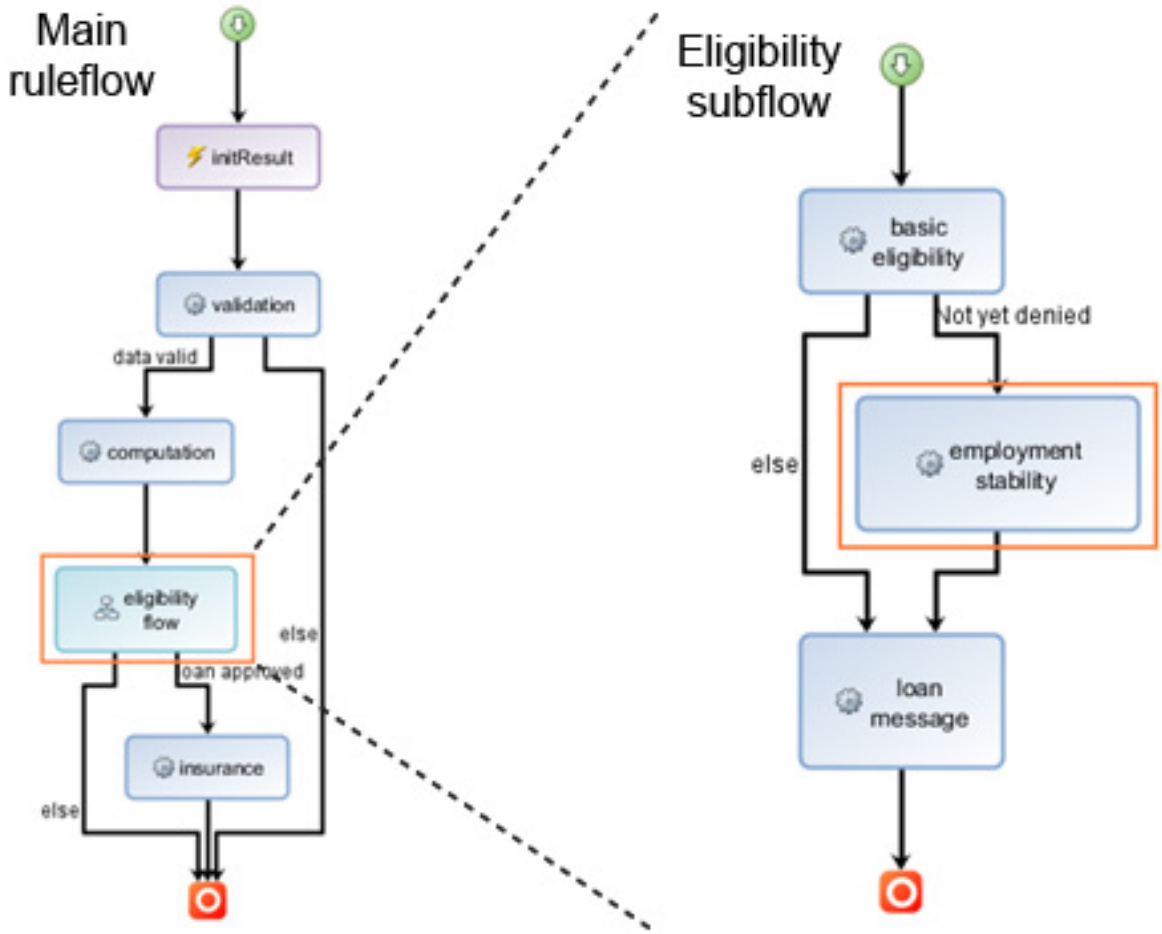
This variable is set in the initial action of the ruleflow, as shown in Listing 4.

Listing 4. Initial action of the ruleflow

```
loan.ExternalDataAccessorHelperFactory factory =
        (loan.ExternalDataAccessorHelperFactory)context.getParameterValue
        ("dataAccessorFactory");
context.setParameterValue("dataAccessor",
    factory.createExternalDataAccessorHelper());
```

The ruleflow is adjusted to invoke the eligibility rules only if necessary. Note that if the initial validation fails, there is no need to make the expensive call to the external web service. This is handled by creating an eligibility subflow that invokes employment stability guidelines only if the loan is not already denied. This subflow is depicted in Figure 16.

Figure 16. Eligibility subflow



Now we are ready to inspect the rules that invoke the web service. A quick note on terminology: a rule can be represented by the form: *if 'P' then 'Q'*, where 'P' is the set of conditions for the rule and is also called the lefthand side (LHS) of the rule, and 'Q' represents the actions for the rule and is also called the righthand side (RHS) of the rule.

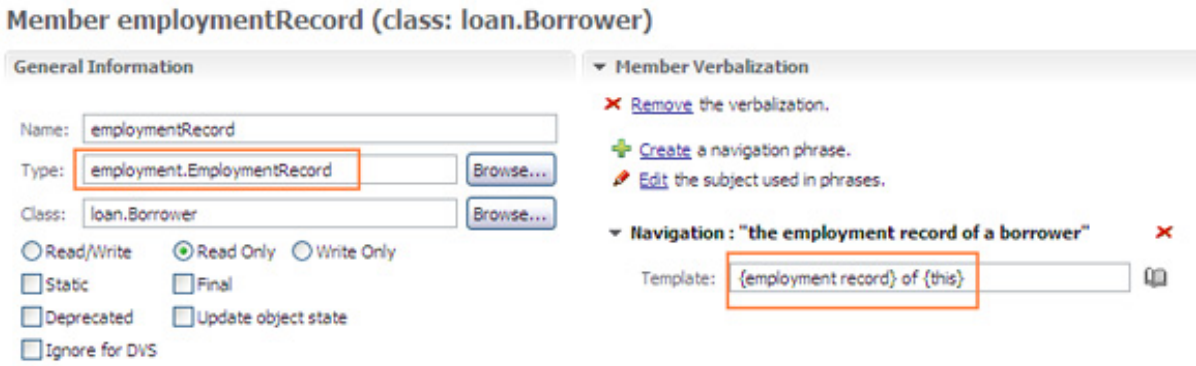
At this point, we have two options:

1. Option B: Invoke the data accessor from the rule conditions, that is, from the LHS of a rule.
2. Option B2: Invoke the data accessor explicitly from the rule actions, that is, from the RHS of a rule.

Option B: Accessing external data from rule conditions

The data accessor can be invoked transparently from the LHS of a rule by adding a virtual, read-only attribute to the Borrower called `employmentRecord`. This can done in the BOM editor, as shown in Figure 17.

Figure 17. A borrower's employmentRecord virtual attribute in the BOM



(See a larger version of Figure 17.)

This attribute is of type `EmploymentRecord`. Of course, since it is merely a virtual attribute, it needs to be backed up by BOM to XOM (B2X) mapping. It is in this B2X that we use the data accessor to call out to the external web service and return the employment record. Exception handling also occurs in this B2X. If the employment record returned from this service is null, it indicates that an exception occurred during data accesss and therefore an error message is created, as shown in Listing 5. The data accessor is invoked (line 2) using the SSN of the borrower. If there is an exception during this processing, then an error record is instantiated (lines 6 – 9).

Listing 5. Borrower.employmentRecord BOM to XOM mapping

```
1. ExternalDataAccessorHelper accessor = (ExternalDataAccessorHelper)
        context.getParameterValue("dataAccessor");
2. EmploymentRecord record = accessor.getEmploymentRecord(this.getSSN().toString());
3. Report rpt = (Report) context.getParameterValue("report");
4. if (record == null) {
5.         //create a "error" record with negative index
6. record = new EmploymentRecord();
7. record.ssn = "ERROR";
8. record.employmentStabilityIndex = -1;
9. rpt.addErrorMessage("Error retrieving employment profile!
                Please try again later.");
10.}
11.rpt.employmentRecord = record;
12.return record;
```

Now that the data access happens behind the scenes when accessing the employment record of a borrower, there is minimal intrusion of the data accessor into the rule itself. For instance, the rule to check the employment stability based on corporate score, shown in Listing 6, does not make any direct references to a data accessor.

Listing 6. Data access from LHS

```
if
```

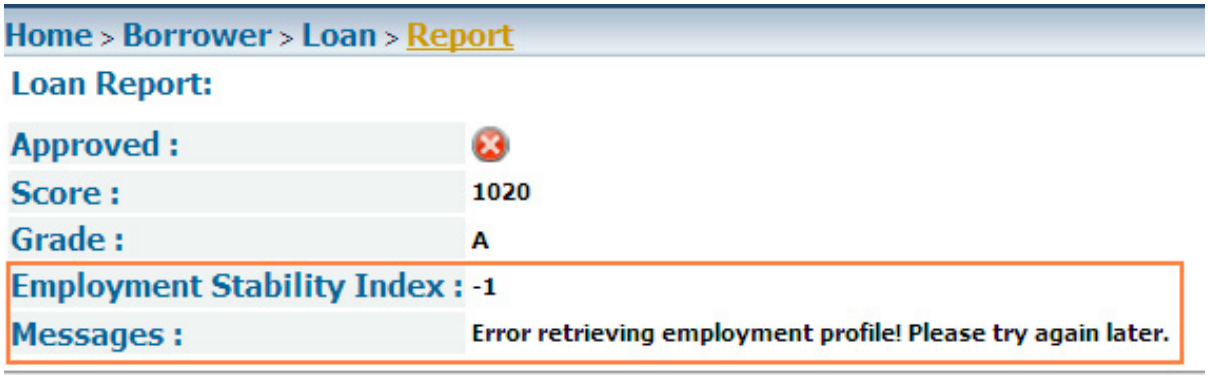
```
    the amount of 'the loan' is more than  50000
    and the corporate score in 'the loan report' is less than  600
    and the employment stability index of the employment record of
        'the borrower' is less than 72
then
    in 'the loan report', refuse the loan with the message "Insufficient
        employment stability of "
      + the employment stability index of the employment record of 'the
        borrower' ;
```

The key advantage to this option, as in Option A, is that the data access is transparent to the rule developer or business user. Additionally, the data access occurs only when it is needed and there is no wasteful data access. For instance, Figure 18 illustrates a scenario where the loan is approved and the credit score and corporate score were high enough that employment stability check was not necessary.

Figure 18. Scenario where employment record is not required

Another advantage is that exception handling happens behind the scenes without business user intervention. Figure 19 depicts the graceful handling of an exception scenario (when a web service is down). As seen earlier in Listing 5, this is handled in the B2X.

Figure 19. Data accessor exception result



On the flip side, since the data accessor factory is provided as an input ruleset parameter, the request data is not entirely self-contained and is therefore slightly harder to unit test. Also, if the employment web service interface changes, then the XOM (and the BOM) for the ruleset have to be correspondingly modified.

One other danger has to do with Rete network activation. If different, even if equivalent, objects are returned from the data accessor invocation for the same arguments, then there is a danger that the same rule is activated multiple times. This negatively impacts performance and could even lead to infinite looping. However, this is easily avoided by locally caching the responses in the data accessor and checking the cache before invoking the web service, as we did earlier in Listing 3. A note on caching: if there are a large number of data accesses for a rule request, you should take care to ensure that the memory used for caching remains at a manageable level.

Another drawback to this approach is that the user needs to be aware of the fact that data is retrieved on an as-needed basis. This needs to be carefully considered during the ruleflow design. If there are multiple data accesses within the same rule task, this can lead to execution of all these data access requests, even if some of them were not necessary. Therefore, the ruleflow should be designed to avoid wastefully fetching data by not having multiple dependent data accesses within the same rule task.

Evaluation

Table 4 shows the pros and cons of this approach.

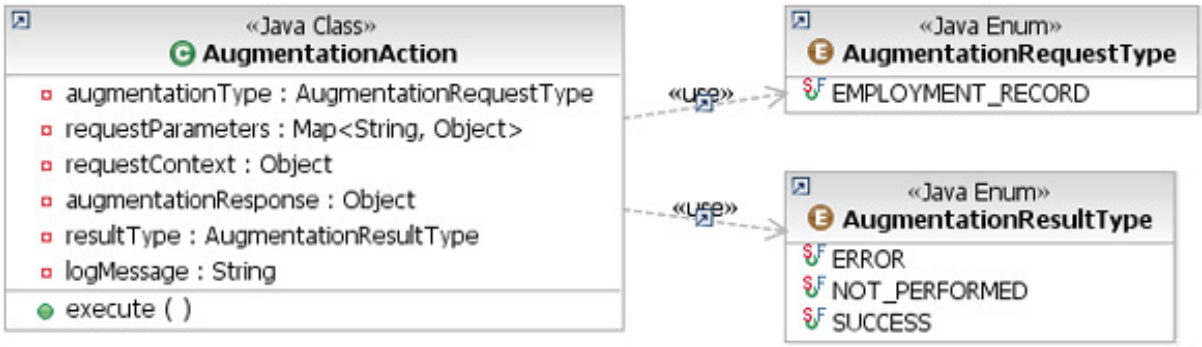
Table 4. Direct data access from LHS evaluation

Pros	Cons
• Data is retrieved only as needed. • Can handle truly dynamic data access requirements. • Over time, if additional data elements are required, the decision service signature need not to be altered. • Transparent data access and handling of exceptions.	• Negative performance impact of data access on decision service throughput. • Harder to component test - need a mock data access layer for controlled component testing. • Needs Java XOM and precludes use of HTDS. • Impacts BOM. • Ruleflow must be carefully designed to minimize wasteful data access.

Option B2: Rules accessing external data from rule actions

With this option, rules explicitly request additional data in their rule action (RHS) when necessary. This additional data request is implemented by an `AugmentationAction` class in the XOM, as shown in Figure 20.

Figure 20. AugmentationAction XOM



The attributes of this class are best illustrated with an example instance. An object instance may look like the following:

Data Element	Comments
augmentationType = EMPLOYMENT_RECORD	*This request is to fetch the employment record (only available option for this scenario)*
requestParameters = <SSN, “123-45-6789”>	*Fetch employment record for the specified SSN*
requestContext = 72	*The rule that created this request mandates a minimum stability index requirement is 72*

The data accessor is used to execute this action, and it results in other attributes of the action object being populated. For instance, after execution, the instance may have the following:

Data Element	Comments
augmentationResponse = EmploymentRecord<"123-45-6789", 56>	*The retrieved employment record has a stability index of 56*
resultType = SUCCESS	*Successfully retrieved*
logMessage = ""	*No messages from the data accessor.*

For instance, when the credit score is low enough to warrant an employment stability check, a rule could create an `AugmentationAction` object and insert it into the rule engine working memory. Listing 7 shows the rule

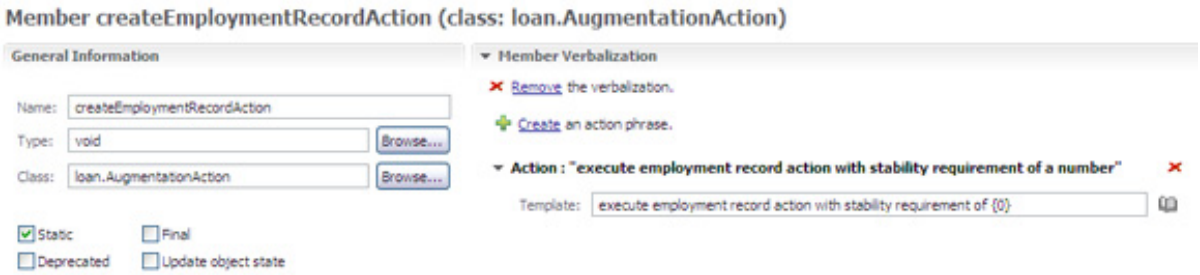
Listing 7. Rule to request additional data

```
1. if
2. the amount of 'the loan' is more than  50000
3. and the corporate score in 'the loan report' is less than  600
```

```
4. then
5.    execute employment record action with stability requirement of 72 ;
```

Here, the execute employment record action (line 5) is a static virtual method that is defined in the BOM as shown in Figure 21.

Figure 21. Virtual createEmploymentRecordAction method



As shown in Listing 8, the BOM to XOM mapping for this method creates an `AugmentationAction` instance (lines 1 – 2), sets the necessary attributes of it (lines 3 – 7), executes it using the data accessor to retrieve the employment record (lines 8), and inserts this object into working memory (line 9) so that other rules may act on it.

Listing 8. BOM to XOM for data access from RHS

```
1. definitions
2.  set 'augmentation action' to an augmentation action
3. where the augmentation type of this augmentation action is EMPLOYMENT_RECORD
4.   and the result type of this augmentation action is SUCCESS;
5.  set 'employment record' to an employment record from the response of 'augmentation
        action' ;
6.  set 'employment stability' to the employment stability index of 'employment record' ;
7.  set 'stability requirement' to the stability requirement of 'augmentation action' ;
8. if
9.  'employment stability' is less than 'stability requirement'
10. then
11. in 'the loan report', refuse the loan with the message
12.         "Insufficient employment stability of " + 'employment stability' ;
```

Two other generic rules are added to handle this `AugmentationAction` result that is posted to working memory. The first rule handles the case where the data was successfully retrieved, and is shown in Listing 9. This is a generic rule that handles successful data access (lines 2 – 4) and that denies the loan if the actual employment stability does not meet the guidelines set out in the rule that created the augmentation action object (line 9).

Listing 9. Rule to handle successful data access execution

```
1. definitions
2.   set 'augmentation action' to an augmentation action
```

```
3. where the augmentation type of this augmentation action is EMPLOYMENT_RECORD
4.    and the result type of this augmentation action is SUCCESS;
5.   set 'employment record' to an employment record from the response of
                'augmentation action' ;
6.   set 'employment stability' to the employment stability index of
        'employment record' ;
7.   set 'stability requirement' to the stability requirement of 'augmentation action' ;
8. if
9.   'employment stability' is less than 'stability requirement'
10. then
11. in 'the loan report', refuse the loan with the message
12.         "Insufficient employment stability of " + 'employment stability' ;
```

Listing 10. Rule to handle data access exception

```
definitions
    set 'augmentation action' to an augmentation action
        where the augmentation type of this augmentation action is
                EMPLOYMENT_RECORD
           and the result type of this augmentation action is ERROR;
then
    in 'the loan report', add the error message
            "System error while accessing employment profile. Please try
                again later.";
```

Evaluation

Table 5 summarizes the pros and cons of this approach.

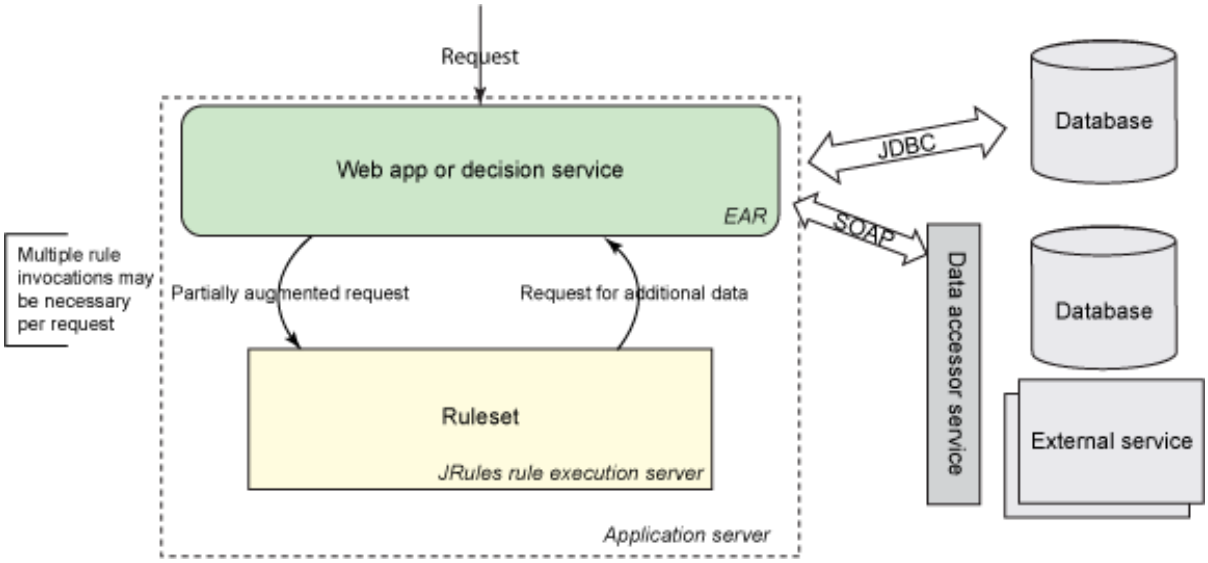
Table 5. Rules accessing external data from rule actions evaluation

Pros	Cons
• Data is retrieved only as needed. • Can handle truly dynamic data requirements. • Over time, if additional data elements are required, the decision service signature need not be altered. • More explicit control of data access.	• High level of complexity. • Heavy impact on rule authoring. • Performance impact of data access on decision service throughput. • Harder to component test - need a mock data access layer for controlled component testing. • Needs Java XOM and precludes HTDS. • Database related exception handling would need to be explicitly accommodated in rules. • Impacts ruleflow/rules and XOM. • May need clever ways of handling of return values (such as the usage of request context in this example).

Option C: Successive data enrichment

Using this option, a ruleset responds back to the rule client with requests to fetch additional data, which are then executed by the invoking application. In other words, data access is a joint responsibility between a ruleset and the invoking application with the ruleset specifying what data should be retrieved and the invoking application retrieving the data and resending the incrementally augmented request.

Figure 22. Successive enrichment of data



The sequence is as follows:

1. Invoking an application sends a rule request to the ruleset.
2. During rule execution, if external data is necessary, the ruleset response contains requests for this data.
3. If the ruleset response contains requests for data, invoking an application retrieves data and augments the request.
4. If ruleset response contains no additional requests for data, then rule processing is complete. Exit loop.
5. Go back to step 1.

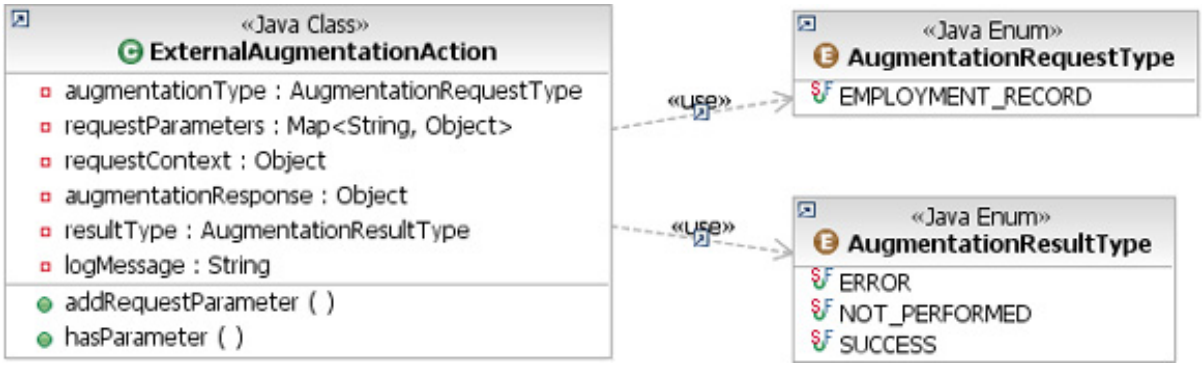
With this approach, multiple calls to the ruleset may be necessary to handle an external request. The requests are successively augmented with the data requested by the ruleset. The rule client may invoke the ruleset locally or remotely. In either case, owing to the orchestration requirements, the ruleset is only usable in conjunction with this rule client.

A small variation on this approach is where the rule client progressively invokes different rulesets as the data is enriched.

Implementation details

If you sense some degree of similarity between this and option B2, you are quite right. Here too, requests for additional data need to be captured as augmentation actions. In fact, we define an `ExternalAugmentationAction` class that is similar to the one we defined earlier, namely the `AugmentationAction` class, except that it does not have the `execute()` method attached to it, as shown in Figure 23.

Figure 23. XOM changes for option C



This `ExternalAugmentationAction` class holds the request parameters specified by the ruleset, as well as the augmentation response containing the data access results. This class is very similar to the `AugmentationAction` class detailed in section: Option B2: Rules accessing external data from rule actions.

There are two new ruleset parameters: `dataRequests` and `dataAugmentations`. The ruleset makes requests for additional data by creating instances of the `ExternalAugmentationAction` class and putting them in the output parameter called `dataRequests`. When an instance of `ExternalAugmentationAction` is created by rules, its status is initialized to `NOT_PERFORMED`. The application retrieves data from the external web service to satisfy these data requests and the augmented data is passed into the ruleset in the next invocation in the `dataAugmentations` input parameter. These instances will have a result type of either `SUCCESS` or `ERROR`, depending on whether or not the data was successfully retrieved from the external web service. Figure 24 shows the ruleset parameters.

Figure 24. Ruleset parameters with data requests and data augmentations

Ruleset Parameters

Define ruleset parameters.

Name	Type	Direction	Defa...	Verbalization
borrower	loan.Borrower	IN		the borrower
loan	loan.Loan	IN_OUT		the loan
report	loan.Report	OUT		the loan report
dataAugmentations	loan.ExternalAugmentationAction[]	IN		the data augmentations
dataRequests	loan.ExternalAugmentationAction[]	OUT		the data requests

The web controller has the responsibility of checking whether there are any data requests, and if so, satisfying them and sending the data augmentations as requested. The key pieces of the code are in bold in Listing 11. The code in lines 8 – 30 performs the successive data access. Rules are executed on line 18 and on line 21, the new data requests are identified. On line 24, the actual external data access is executed and on line 26, the data augmentations are set, and are then sent in the next ruleset invocation (line 14).

Listing 11. Excerpt of BusinessBeanController_C.java

```
1. public Report executeRulesOnPojoRuleSession(String rulesetPath) throws
      IlrFormatException, IlrSessionException  {
2. IlrSessionRequest sessionRequest = pojoFactory.createRequest();
3. ExternalDataAccessorHelper dataAccessor =
    dataAccessorFactory.createExternalDataAccessorHelper();
4. Report report = null;
5. ExternalAugmentationAction[] dataAugmentations = new
                ExternalAugmentationAction[0];
6. int iterationNum = 0;
7. boolean continueRuleInvocation = true;
8. while (continueRuleInvocation && iterationNum++ < MAX_INVOCATIONS) {
9. sessionRequest.setRulesetPath(IlrPath.parsePath(rulesetPath));
10.         // Set the input parameters for the execution of the rules
11.         Map<String, Object> inputParameters = sessionRequest.getInputParameters();
12.         inputParameters.put("borrower", getBorrower());
13.         inputParameters.put("loan", getLoan());
14.         inputParameters.put("dataAugmentations", dataAugmentations);
15.         // Create a stateless rule session
16.         IlrStatelessSession ruleSession = pojoFactory.createStatelessSession();
17.         // Execute the rules
18.         IlrSessionResponse sessionResponse = ruleSession.execute(sessionRequest);
19.         report = (Report) sessionResponse.getOutputParameters().get("report");
20.         ExternalAugmentationAction[] additionalDataRequests
21.                                 = (ExternalAugmentationAction[])
     sessionResponse.getOutputParameters().get("dataRequests");
22.         if (additionalDataRequests != null &&
                    additionalDataRequests.length > 0) {
23.                 //execute
24.                 execute (additionalDataRequests, dataAccessor);
25.                 //add it to response list!
26.                 dataAugmentations = addAll(dataAugmentations,
                ExternalAugmentationAction[] additionalDataRequests);
27.         } else {
28.                 continueRuleInvocation = false;
29.         }
30. }
31. return report;
32.   }
```

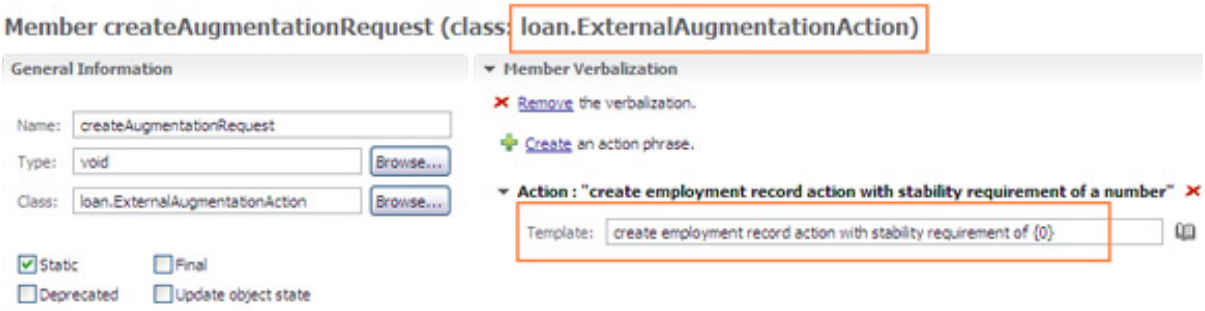
Listing 12 shows a rule that creates this data augmentation request. This rule has a condition (lines 4 – 7) that checks that the data augmentation has not already been attempted (either successfully or with an error).

Listing 12. Excerpt of Rule that requests additional data

```
1. if
2. the amount of 'the loan' is more than  50000
3. and the corporate score in 'the loan report' is less than  600
4. and the number of external augmentation actions
5. where the augmentation type of each external augmentation action is
                EMPLOYMENT_RECORD
6.      and the result type of each external augmentation action is one of
                { ERROR , SUCCESS }  ,
7. is 0
8. then
9.  create employment record action with stability requirement of 72;
```

The action for the rule (line 9) is a virtual static method that creates the data request and adds it to the output array as shown in Figure 25.

Figure 25. Virtual method to create augmentation request



This virtual method is backed up by B2X mapping code shown in Listing 13. In this method, an `ExternalAugmentationAction` is created and initialized with the appropriate data (lines 1 – 5) and inserted into the working memory (line 6). At the end, this newly created action instance is added to the output array (line 17).

Listing 13. Excerpt of BOM to XOM to create data augmentation request

```
1. ExternalAugmentationAction augmentationAction
2. = new ExternalAugmentationAction
                (AugmentationRequestType.EMPLOYMENT_RECORD);
3. Borrower borrower = (Borrower) context.getParameterValue("borrower");
4. augmentationAction.addRequestParameter("SSN", borrower.getSSN().toString());
5. augmentationAction.requestContext = new java.lang.Integer((int) stabilityIndex);
6. context.insert(augmentationAction);
7.
8. //add to the data augmentation requests to be sent back
9.   ExternalAugmentationAction[] request
10. = (ExternalAugmentationAction[]) context.getParameterValue
                ("dataRequests");
```

```
11.   ExternalAugmentationAction[] newRequestArray = new
                ExternalAugmentationAction[request.length + 1];
12.   int i = 0;
13.   for (i=0; i<request.length; i++) {
14.         newRequestArray[i] = request[i];
15.   }
16.   newRequestArray[request.length] = augmentationAction;
17.   context.setParameterValue("dataRequests", newRequestArray);
```

When augmented data is available, it is used to make decisions in rules. The rule shown in Listing 14 uses a successful augmentation action to make a decision to refuse a loan.

Listing 14. Rule to make decisions based on augmented data

```
definitions
   set 'augmentation action' to an external augmentation action
        where the augmentation type of this external augmentation action is
          EMPLOYMENT_RECORD
          and the result type of this external augmentation action is SUCCESS;
   set 'employment record' to an employment record from the response of
        'augmentation action' ;
   set 'employment stability' to the employment stability index of 'employment
        record' ;
   set 'stability requirement' to the stability requirement of 'augmentation
        action' ;
if
   'employment stability' is less than 'stability requirement'
then
   in 'the loan report', refuse the loan with the message "Insufficient employment
        stability of " + 'employment stability' ;
```

Exceptions are handled explicitly with rules. Listing 15 depicts a rule that adds an error message to the report (line 8) if any of the data requests has returned with an ERROR status (lines 2 – 4).

Listing 15. Rule to handle exceptions

```
1. definitions
2.   set 'augmentation action' to an external augmentation action
3.     where the augmentation type of this external augmentation action
        is EMPLOYMENT_RECORD
4.       and the result type of this external augmentation action is ERROR;
5. if
6.   true
7. then
8.       in 'the loan report', add the error message "System error while
                accessing employment profile. Please try again later." ;
```

As apparent by now, there is a lot of complexity to this approach. Rules explicitly request data and rules explicitly handle successful and failed data access responses. Some of these rules are very technical and fairly constant and therefore do not need to be maintained by the business user. These can be moved into a separate rule package for maintenance by IT. However, the complexity seeps

through even to the other business rules. The business user is not shielded very well from the intricacies of the data access. This may compel the rule maintenance be done outside of the realm of the business users and more in the hands of developers, thus canceling one of the key benefits of a business rules management system.

On the positive side, if rule development is in the purview of IT developers, then they have more control over how to handle data access exceptions. Additionally, this approach can handle truly dynamic data access requirements, such as a potential scenario where a web service is invoked with the credit rating as a parameter; the credit rating itself being derived during the rule processing. In such a scenario, the credit rating would be passed in with the `ExternalAugmentationAction` as a request parameter.

Evaluation

Table 6 summarizes the pros and cons of this approach.

Table 6. Successive data enrichment evaluation

Pros	Cons
• Data is retrieved only as needed. • Rule requests are self-contained and the ruleset has no external dependency. • No direct performance impact of data access on ruleset throughput. • Does not require a Java XOM and therefore HTDS is not precluded. However, it is a lot more flexible and convenient to have the XOM in Java.	• Multiple decision service invocations are necessary per request. This has a negative performance impact if a large number of roundtrips are necessary. • Harder to component test. • Adds a lot of complexity. • Explicit data access rules are required. Business users are not shielded from the data access details.

Comparison of the options

The table below summarizes the comparative evaluation of the data access options against a series of criteria. A plus sign indicates that the option positively satisfies the criterion, while a minus sign indicates otherwise. The number of symbols (between 1 and 3) indicates the degree to which the criteria is supported or violated by that option. For instance, 3 plus signs indicate that the criterion is strongly supported by that data access option.

	Request augmentation	Direct access (LHS)	Direct access (RHS)	Successive request

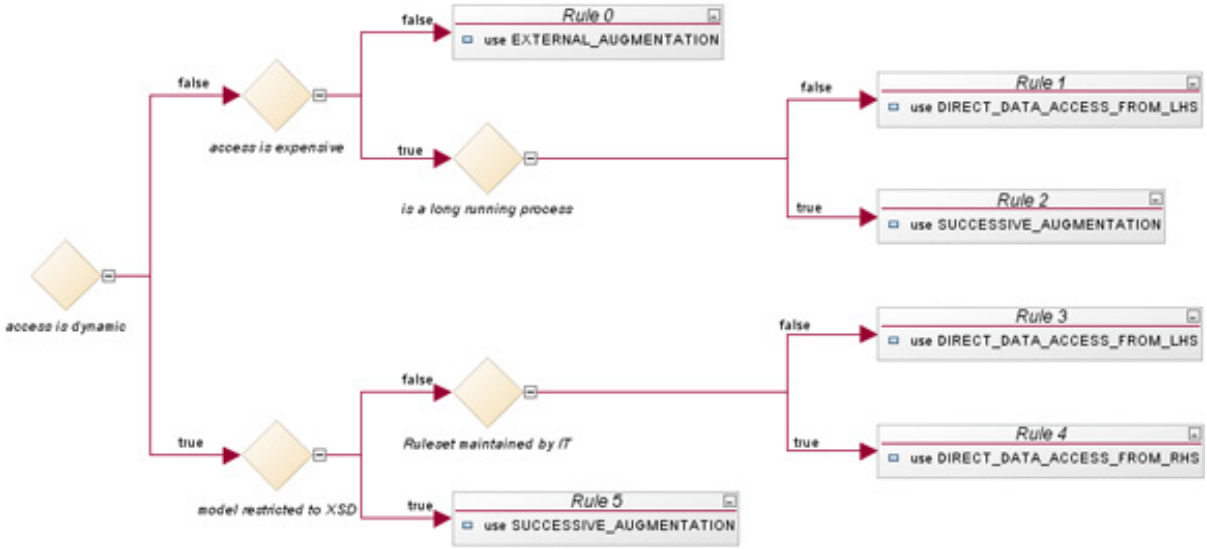
				enrichment
Loose coupling (ruleset is unaware of data access)	⊕⊕⊕	⊖	⊖	⊕
High ruleset throughput	⊕	⊖⊖	⊖⊖	⊖
Handle truly dynamic data access	⊖⊖⊖	⊕⊕⊕	⊕⊕⊕	⊕⊕⊕
Ease of exception handling	⊕⊕⊕	⊖	⊖⊖	⊖⊖
Reduce wasteful data access	⊖⊖⊖	⊕⊕⊕	⊕⊕⊕	⊕⊕⊕
Minimize integration layer complexity	⊕⊕	⊕	⊕	⊖⊖
Minimize object model complexity	⊕	⊕	⊖⊖	⊖⊖
Minimize rule authoring impact	⊕⊕⊕	⊕	⊖⊖	⊖⊖⊖
Ease of ruleset component testing	⊕⊕⊕	⊖	⊖	⊕
Allows HTDS (no Java XOM requirement)	⊕⊕⊕	⊖⊖⊖	⊖⊖⊖	⊕

From this table, it is clear that upfront request augmentation is the best approach to performing data access as long as two criteria are met:

1. It is feasible; that is, data gathering requirements are not truly dynamic
2. It is not too wasteful; that is, the data access operation is not too expensive.

If either of these criteria is not met, then in most cases it is preferable to use direct data access from the lefthand side of rules. Direct data access from the righthand side may be preferred if the ruleset is maintained by IT developers, as opposed to business users, who want to have greater control over exception handling. Based on these factors, Figure 26 shows a decision tree to determine the appropriate data access option.

Figure 26. Decision tree to determine data access option



(See a larger version of Figure 26.)

In the scenario under consideration, since the employment web service access is an expensive operation, the preferred approach is to use direct data access from the lefthand side of rules (Option B).

Conclusion

In this article, you saw how some business requirements mandate the use of external data. This data access may be simple, contextual or dynamic. There are multiple options to handle data access requirements. The simplest option is to fetch this external data up front and augment the rule request with this data. However, this may be wasteful or even infeasible for dynamic data access requirements. Other options include fetching the data during rule execution, either from the lefthand side or righthand side of a rule, or to successively augment the request with multiple round trips to the rule engine. This article described these options in detailed using a sample scenario and provided a comparative evaluation and a decision tree to help you select the data access option that is right for your situation.

Downloads

Description	Name	Size	Download method
Project interchange file	dataAccess_pif.zip	333KB	HTTP

Information about download methods

Resources

Learn

- Service-Oriented Architecture expands the vision of web services (developerWorks 2004)
- Develop decision services, Part 2: Rules development process (developerWorks 2011)
- Using the Web service wizards to create Web services and clients (Rational Application Developer Information Center)
- Installing Rule Studio in IBM Eclipse-based products (WebSphere ILOG BRMS Information Center)
- Factory Pattern (oodesign.com)
- WebSphere ILOG JRules Information Center
- WebSphere Operational Decision Management Information Center
- WebSphere ILOG JRules product information:
- WebSphere Operational Decision Management product information
- developerWorks BPM zone: Get the latest technical resources on IBM BPM solutions, including downloads, demos, articles, tutorials, events, webcasts, and more.
- IBM BPM Journal: Get the latest articles and columns on BPM solutions in this quarterly journal, also available in both Kindle and PDF versions.

Get products and technologies

- WebSphere ILOG JRules V7.1 trial download

About the author

Raj Rao

Rajesh (Raj) Rao has been working in the area of expert systems and business rule management systems for over 20 years, during which time he has applied business rules technology to build diagnostic, scheduling, qualification and configuration applications across various domains such as manufacturing, transportation and finance. He has been with IBM for close to 2 years. With a background in Artificial Intelligence, his interests include Natural Language Processing and Semantic Web.