BEA AquaLogic Commerce Services

Promotion Rule Engine Developer Guide

Version 5.1
September 2007

Copyright

Restricted Rights Legend

Trademarks and Service Marks

Contents

Architecture Overview

At the core of the promotion rule system is the JBoss Rules (formerly Drools Rules) library. JBoss Rules is a third-party rules engine that uses a fast algorithm to evaluate rule conditions and execute their actions. The input to the JBoss Rules engine is a set of objects used in the condition evaluation and action execution as well as the set of rules, which we express as text in the proprietary JBoss Rules language.

The representation of a Rule in AquaLogic Commerce Services is an object model of the components of a rule such as conditions, actions, and parameters used by various rule elements. The object model is persisted in the database directly with one table corresponding to one class in the rule object model. This allows the graph of rule objects to be easily stored, retrieved and modified, and stored again. The objects in the rule object model are responsible for generating JBoss Rules language code that is passed to JBoss Rules. The generated code is not persisted and it is not possible to re-create the object model representation of a rule given the rule code.

The role of the Commerce Manager promotion editor is to allow the user to compose rules from rule elements and then store those rules in the database. The Storefront then retrieves the rules from the database as object graphs, requests the corresponding JBoss Rules code from the rule objects, and passes the rule code to the JBoss Rules engine. JBoss Rules will then determine which rules' actions should be executed on the Java objects that are passed to it.

JBoss rules support basic evaluation of objects' properties within the engine itself. However, more complex operations are not supported. In AquaLogic Commerce Services, nearly all conditions are evaluated in Java and the actions are also executed in Java. This allows the condition and action code to be easily debugged and unit tested. The PromotionRuleDelegate is the class that is responsible for computing conditions and executing actions as required by JBoss Rules.

Promotion Rule Technical Design

AquaLogic Commerce Services includes a promotion rule engine that provides a flexible tool for store operators to compose promotions that customers will see in the Storefront. Promotions are created through a graphical user interface in the Commerce Manager by composing available **rule elements** to create rules. A rule is composed of the following rule elements:

- **Eligibility**: Determines which customers are eligible to receive a promotion. When multiple eligibilities are specified, the user can choose whether the customer must meet **all** of the eligibilities or **any** of them.
- **Condition**: Describes the set of conditions that must be true for the action to be executed. Conditions are optional. If no condition is specified, then the action will execute whenever the customer is eligible. When multiple conditions are specified, the user can again choose whether **all** or **any** of them must be satisfied.
- **Action**: Describes the action that will be taken if the customer is eligible and all conditions are met. If multiple actions are present, all actions will be executed when the conditions are met.

Using the promotion rule editor in the Commerce Manager, users mix and match promotion rule elements to create rules. Therefore, all rule elements are independent of each other. For example, it is not possible for an action to use information determined by a condition.

Rules are classified into scenarios that determine when the rule will be evaluated and potentially executed while a customer is using the Storefront. Some rule elements are valid in all scenarios and others are specific to a particular scenario. In some cases, rule elements with the same meaning must be implemented differently for multiple scenarios. There are currently two scenarios:

- **Product Catalog**: The user is browsing products in the catalog or viewing product details. Simple rules that affect viewed products can be defined in this scenario.
- **Shopping Cart**: The user is viewing the items in their shopping cart. This scenario supports more complex rules that require awareness of the combinations of items in a customer's cart.

Key classes

The key classes that represent rules in the domain model are:

- **RuleSet** - Contains a set of rules valid in a given scenario
- **Rule** - Contains collections of rule elements (Eligibilities, Conditions, and Actions).
- **Eligibility** - Represents a condition under which a customer is eligible for a promotion
- **Condition** - Represents a general condition that must be true for a promotion to be available

- **Action** - Represents the action to be taken if the eligibilities and conditions are met
- **RuleParameter** - Actions and conditions typically require parameters held by a RuleParameter object

In addition to representing the rules, these domain objects are responsible for generating the JBoss Rules code corresponding to the rule. Each object contributes its own portion of the rule code. The rule domain objects are also responsible for self validation. The validation and code generation is invoked by clients at the top level (Rule Set) and propagated down to the child objects.

Other key classes include:

- **EpRuleEngine** - Retrieves rules from persistent storage, compiles them into drools language, and fires rules on domain objects when requested by clients.
- **PromotionRuleDelegateImpl** - Performs evaluations of rule conditions and executes rule actions.

JBoss Rules Rule Code

The rules objects generate JBoss Rules code such as the example below:

```
//Objects used in the evaluation or action must be imported
import com.elasticpath.domain.rules.PromotionRuleExceptions;
import com.elasticpath.domain.shoppingcart.CartItem;
import com.elasticpath.domain.catalog.ProductSku;
import com.elasticpath.domain.shoppingcart.ShoppingCart;
import com.elasticpath.domain.rules.PromotionRuleDelegate;
import com.elasticpath.domain.catalog.Product;

//Each rule must have a name, defined here
rule "First Time Buyer"

  //Salience determines the order of evaluation,
  // higher salience means higher priority
 salience -1

  //Agenda groups are evaluated and executed together
 agenda-group "SubtotalDependent"

    //Start of the "Conditions" Block
    when

         //Declare objects used in the rule
```

```
        delegate: PromotionRuleDelegate ( )
        cart: ShoppingCart ( )
        eval ( delegate.checkDateRange("0","0") )

        //Ask the delegate to evaluate whether conditions are true
        eval ( delegate.isFirstTimeBuyer(cart) )


    //The "then" block defines what will happen when the conditions in
    //the "when" block are all true
    then

        //Ask the delegate to execute an action
        delegate.applyOrderDiscountAmount(cart, 4489216, "75");
end
```

Firing Rules in the Storefront

The Store Front uses EpRuleEngineImpl to apply promotion rules to products and shopping carts. Most of the calls for firing the shopping cart rules are made by the shopping cart itself in response to changes in shopping cart state. Services typically fire rules on products as required. The set of rules to be passed to JBoss Rules is loaded at start time and cached by the EpRuleEngineImpl. EpRuleEngine.compileRuleBase() is invoked periodically by a quartz scheduler to re-load rules from the database.

Configuration

- The promotion rulebase can be configured to re-build periodically, for details see: quartz.xml. This means that rules will be re-loaded from the database and compiled into the input format for JBoss Rules. This rule compilation operation is expensive and should not be performed more frequently than every 5-10 seconds.
- For rules to take effect immediately after rulebase compilation, it is necessary to disable caching (Both the second-level cache and the product retrieval strategy defined in service.xml). This is useful for testing but disabling caching is not recommended for production environments.

Adding a new Promotion Rule element

Rules are composed of Rule Elements, which can be Eligibilities, Conditions, or Actions. The following steps demonstrate how to add new rule elements to AquaLogic Commerce Services.

1. Create a new class to represent your new rule element and have it extend from the appropriate base class. If you are creating an action, extend AbstractRuleActionImpl, otherwise extend AbstractRuleElementImpl. Your new class should implement the RuleCondition, RuleEligibility, or RuleAction depending on its type. See Appendix A for commented sample code that explains the functionality your new class must implement.
2. Declare your new rule element as a bean in domainModel.xml as shown in the example below.

```
<!-- Bean declaration for the NumItemsInCartConditionImpl class. Note
that the scope is set to "prototype" so that a new instance of this
class will be returned from the Spring context each time it is
requested. The parent bean definition "epDomain" makes this bean
definition extend from the bean with id "epDomain" which declares the
Spring configuration for a base class of the
NumItemsInCartConditionImpl. -->
<bean id="numItemsInCartCondition"
class="com.yourcompany.domain.rules.impl.NumItemsInCartConditionImpl"
scope="prototype" parent="epDomain">
</bean>
```

3. A class called a PromotionRuleDelegate is used by the JBoss Rules engine to invoke Java code. You will need to use an extended implementation of PromotionRuleDelegateImpl that will contain the Java method that will execute the action or check the condition of your new rule element. Create a new class with a name such as CustomPromotionRuleDelegateImpl that extends PromotionRuleDelegateImpl. You will need to add a new Spring bean definition for this class as follows:

```
<!-- Bean declaration for a custom promotion rule delegate class -->
<bean id="customPromotionRuleDelegate"
class="com.yourdomain.domain.rules.impl.CustomPromotionRuleDelegateImpl
" scope="singleton" parent="epDomain">

</bean>
```

   To use this custom class instead of the default one, wire it into the existing bean definition for the EpRuleEngineImpl in the service.xml file in the Storefront so that it is declared as shown below:

```
<bean id="epRuleEngine"
class="com.elasticpath.service.rules.impl.EpRuleEngineImpl"
scope="singleton">
```

```
<property name="ruleSetService">
      <ref bean="ruleSetService" />
</property>
<property name="elasticPath">
      <ref bean="elasticPath" />
</property>
<property name="utility">
          <ref bean="utility" />
</property>
<property name="propertiesDao">
          <ref bean="propertiesDao" />
</property>
<property name="timeService">
          <ref bean="timeService" />
</property>

<!-- Inject custom promotion rule delegate here -->
<property name="promotionRuleDelegate">
      <ref bean="customPromotionRuleDelegate" />
</property>

</bean>
```

4. To pass the custom promotion rule delegate to the JBoss Rules engine in place of the default delegate, it's necessary to extend the PromotionRuleImpl class. Please see ExtensiblePromotionRuleImpl in Appendix A for a sample extension of this class that changes the default delegate to a custom delegate with a different class name. In order to use this extended implementation, it's necessary to update the persistence layer by editing the PromotionRule.hbm.xml so that the ExtensiblePromotionRuleImpl implementation is used. You will also need to modify the RuleSet.hbm.xml file so that it references the new rule class. In addition to the persistence changes, it's necessary to inform the system that the ExtensiblePromotionRuleImpl class should be used instead of the standard PromotionRuleImpl class. Since the bean definition for this class isn't declared in Spring, it's necessary to update a bean definition entry in a class called PrototypeBeanFactory. One way to accomplish this is to extend the ElasticPathService (which is declared in Spring) with a new service that sets the correct implementation class. Modify the entry for the elasticPathService bean in the service.xml file so that it appears as shown below:

```
<bean id="elasticPathService"
class="com.yourcompany.service.impl.CustomElasticPathServiceImpl">
      <property name="elasticPathDao">
            <ref bean="elasticPathDao" />
      </property>
      <property name="elasticPath">
            <ref bean="elasticPath" />
      </property>

</bean>
```

You can then write an implementation as shown in Appendix A that will set the correct class name for the promotion rule bean.

5. To use the new CustomPromotionRuleDelegateImpl in JBoss Rules code, it's necessary to import it within the rules code. The imports to be added to the rules code are stored in a static list in the RuleSetImpl class. One easy way to add new imports is to add them to the static list in the CustomElasticPathService as shown in Appendix A.
6. In the service.xml file, you will need to add the rule element to the appropriate category in the rule service bean definition in service.xml. The bean id for rule service is "ruleService" and the bean id for the new rule element you are creating should be added to the appropriate list of allEligibilities, allConditions, or allActions.
7. Add the new rule element as a subclass definition in RuleElement.hbm.xml so that it can be persisted. A sample entry in this file is shown below.

<subclass name="
com.yourcompany.domain.rules.impl.CustomPromotionRuleDelegateImpl"
discriminator-value="numItemsInCartCondition" />

8. If your new rule element requires a new rule parameter type, you will need to modify the Javascript code for the Commerce Manager rule editor to support the new parameter. See the next section for more information on how to make the necessary customizations.

Commerce Manager Promotion Rules Editor

The Promotions tab of the Commerce Manager implements an editor that allows users to compose promotion rules from available rule elements. Developers who customize the promotions subsystem of AquaLogic Commerce Services should be familiar with the rules engine architecture and the Storefront promotion rule technical feature section.

Key files

The following files are relevant to the implementation of the Commerce Manager Promotions tab.

- **rule.vm** - The Velocity page that displays the Promotions tab in the Commerce Manager.
- **rule.js** - The Javascript controller for the Promotions tab. This file is primarily responsible for communicating with the server to save or retrieve rules, rule elements, and rule parameter values.
- **ruleLoader.js** - Declares several constants primarily used for rule parameter value input.
- **RuleIndexPane.js** - Left-side pane that lists user-defined rules.
- **RuleNamePane.js** - Implements the pane where users specify the rule name, enable dates, etc.
- **RuleEligibilityTab.js** - The top-right tab that allows users to select eligibility elements when composing or editing a rule.
- **RuleConditionTab.js** - The top-right tab that allows users to select condition elements when composing or editing a rule.
- **RuleActionTab.js** - The top-right tab that allows users to select action elements when composing or editing a rule.
- **RuleDefinitionPane.js** - The pane on the lower-right that displays the rule being composed or edited by the user in (nearly) natural language.
- **InlineEditBox.js** - A customized Dojo widget for inline editing of parameter values in the RuleDefinitionPane.

Product Pricing

Prices in AquaLogic Commerce Services are represented by implementations of the Price interface. The Price interface is designed to support multiple price tiers in which the price of an item depends on the quantity in which it is purchased. The interface also supports clients that require only one price for any quantity and do not provide quantity parameters. A price is specific to a particular currency and objects aggregating prices typically maintain a map from currency to price.

A single price object for a given purchase quantity aggregates a collection of price tiers, which specify the price in a given quantity. For each price tier, the following prices may be defined.

- **listPrice** - The full price of the product. This price is mandatory.
- **salePrice** - The discounted price of the product as specified in persistent storage.
- **computedPrice** - The price of a product as computed by the promotion rules engine.

In AquaLogic Commerce Services, prices are required at the product level and a collection of prices for each supported currency is maintained by Product objects. Prices may also be defined for ProductSkus. In this case, the SKU price will take precedence over the product price.

Key price classes

- **Price** - Defines the interface for a price.
- **AbstractPriceImpl** - Standard implementation of the Price interface.
- **ProductPriceImpl** - Extends AbstractPriceImpl to represent the price of a Product. This is required for Hibernate integration and has no members.
- **SkuPriceImpl** - Extends AbstractPriceImpl to represent the price of a ProductSku. This is required for Hibernate integration and has no members.
- **PriceTier** - Represents the price of an item when purchased in a given quantity.

Implementation Details

- Every Price must have at least one price tier.
- The minQty field of a price tier indicates the minimum purchase quantity that is necessary for the price defined in the price tier to apply.
- The first price tier may have a quantity greater than 1, this means that the minimum purchase quantity of the product is greater than 1 unit.
- The price object model stores and returns Money objects, not BigDecimals
- Prices are value objects rather than entities and the hibernate mappings for Prices are defined with the hibernate mapping files for Product and ProductSku
- Price.getLowestPrice() returns the lowest of the list, sale, and computed prices.

Frequently used Price API

```
// This method returns the lowest price of a product SKU in
// the specified currency when the customer purchases
```

```
// 'quantity' items of the SKU.
productSku.getPrice(currency).getLowestPrice(quantity);

// Returns the lowest price without specifying the purchase
// quantity. In this case the price for the tier with the
// lowest quantity is returned.
myPrice.getLowestPrice();

// Since cart items have quantities, the following method
// returns the price for the correct price tier without
// the need for clients to specify it.
cartItem.getAmountMoney(currency);
```

Appendix A – Code Samples

Commented Sample Rule Condition

```
/**
 * This sample condition class implements a rule condition that
 * checks that the number of items in the shopping cart matches
 * a given number provided while defining the rule in the
 * Commerce Manager.
 */
public class NumItemsInCartConditionImpl extends
AbstractRuleElementImpl implements RuleCondition  {


      /**
       * Serial version id.
       */
      public static final long serialVersionUID = 5000000001L;


      /**
       * Declare a unique constant type String for the condition.
       * This is returned by getElementType(). See that method for
       * more details.
       */
      private static final String CONDITION_TYPE =
"numItemsInCartCondition";


/**
 * This string is used to display the condition in the UI and
 * must follow a specific format. See the getDisplayText() method for
 * details.
 */
      private static final String CONDITION_TEXT = "The number of items
in the cart is [" + RuleParameter.NUM_ITEMS_KEY + "].";
```

```
	/**
	 * An array of the keys of parameters that must be specified to
	 * create rules with this RuleElement.
	 */
	private static final String[] PARAMETER_KEYS = new String[] {
RuleParameter.NUM_ITEMS_KEY };


	/**
	 * Returns a unique constant type String for the condition. This
	 * is used by the persistence layer to determine the Java type
	 * of this rule-element so that it can be instantiated when it is
	 * retrieved from a table containing
	 * all types of rule elements.
	 *
	 * This must match both the Spring context bean id and the
	 * discriminator value used by the persistence layer to identify
	 * the class to instantiate when re-creating objects of this
	 * class from the database.
	 * @return the type of the condition subclass.
	 */
	protected String getElementType() {
		return CONDITION_TYPE;
	}


	/**
	 * Returns the kind of this <code>RuleElement</code>
	 * Permissible values are: "eligibility", "condition", and
	 * "action".
	 * @return the kind
	 */
	protected String getElementKind() {
		return RuleCondition.CONDITION_KIND;
	}
```

```
	/**
	 * Returns the text representation of this condition.
	 * This String is displayed in the Commerce Manager promotion
	 * rule editor  and must follow the convention that any
	 * RuleParameters that are to be specified by the user must
	 * appear in this string in the format: [ RuleParameterKey ]
	 * somewhere in the string. Multiple parameters can be specified
	 * but must be contained in their own set of square brackets.
	 *
	 * @return the display text
	 */
	public String getDisplayText() {
		return CONDITION_TEXT;
	}

	/**
	 * Returns true if this rule element is valid in the specified
	 * scenario. In this example, we specify that the rule is valid
	 * in both the browse and the cart scenarios
	 *
	 * @param scenarioId the Id of the scenario to check (defined in
	 * RuleScenarios.java)
	 * @return true if the rule element is applicable in the given
	 * scenario
	 */
	public boolean appliesInScenario(final int scenarioId) {
return scenarioId == RuleScenarios.CATALOG_BROWSE_SCENARIO  ||
scenarioId == RuleScenarios.CART_SCENARIO;
	}

	/**
	 * Returns the JBoss Rules code corresponding to this rule
	 * condition. The code returned should generally call a method in
	 * a Java class that is referenced using the name "delegate" and
	 * pass that method any parameter values for this element.
	 *
	 * @return The rule code.
```

```
         *
         */
        public String getRuleCode() throws EpDomainException {
              validate();
              StringBuffer sbf = new StringBuffer();
              sbf.append("\t\teval ( delegate.numItemsInCartIs(");

        sbf.append(this.getParamValue(RuleParameter.NUM_ITEMS_KEY));
              sbf.append(") )\n");
              return sbf.toString();
        }

        /**
         * Return the array of the required parameter keys for the rule.
         *
         * @return an array of String of the required parameter keys for
         * the rule.
         */
        public String[] getParameterKeys() {
              return PARAMETER_KEYS.clone();
        }
}
```

ExtensiblePromotionRuleImpl

```
public class ExtensiblePromotionRuleImpl extends PromotionRuleImpl {

      private static final String STANDARD_DELEGATE_CLASS =
"PromotionRuleDelegate";

      private String delegateClass = "CustomPromotionRuleDelegateImpl";

      /**
       * Returns the JBoss rules code corresponding to this rule.
       *
       * @return the rule code.
       * @throws EpDomainException if the rule is not well formed
```

```
	 */
	public String getRuleCode() throws EpDomainException {
		if (delegateClass == null) {
			throw new IllegalStateException("delegateClass not
set");
		}

		String ruleCode = super.getRuleCode();
		ruleCode = ruleCode.replaceAll(STANDARD_DELEGATE_CLASS,
this.delegateClass);
		return ruleCode;
	}

	/**
	 * Set the name of the Java type to use as the
	 * PromotionRuleDelegate.
	 *
	 * @param delegateClass the class or interface name
	 * (not qualified)
	 */
	public void setDelegateClass(final String delegateClass) {
		this.delegateClass = delegateClass;
	}

}
```

CustomElasticPathServiceImpl

```
public class CustomElasticPathServiceImpl extends
ElasticPathServiceImpl {

	/**
	 * Override the init method to overwrite the bean definitions for
	 * the classes that needed modifications.
	 */
	public void init() {
		super.init();
```

```
            //(Over)write the map of bean names to fully-qualified
            // class names for beans that are not managed by Spring.
            ElasticPathImpl elasticPath = (ElasticPathImpl)
getElasticPath();
            PrototypeBeanFactory prototypeBeanFactory =
elasticPath.getDomainBeanFactory();
            prototypeBeanFactory.addBeanDefinition(
ContextIdNames.PROMOTION_RULE,

      "com.yourcompany.domain.rules.impl.ExtensiblePromotionRuleImpl");



            //This is a work-around to add new classes to be referenced
            //by the JBoss Rules code. RuleSet has a static list that
            //stores the import statements for each referenced class.
            RuleSet ruleSet = (RuleSet)
getElasticPath().getBean(ContextIdNames.RULE_SET);

      ruleSet.getImports().add("com.yourcompany.domain.rules.impl.Custo
mPromotionRuleDelegateImpl");


      }


}
```