Accelerate the design and development of Java Enterprise Applications

Use model-driven engineering principles to automatically design and implement EJBs and JAX-RS services

Marija Selakovic

March 06, 2015

This article shows how to apply Model Driven Architecture principles to accelerate the design and development of Java Enterprise Applications that use mainstream technology, such as Java Persistence API, Enterprise Java Beans and Java API for RESTful Web Services. It investigates each step of the model-driven development process from the initial domain design to the generation of EJB 3.0 and JAX-RS design and implementations.

Introduction

The scope of this article is accelerating the design and development of Java Enterprise Applications that use mainstream technologies, such as Java Persistence API (JPA), Enterprise Java Beans (EJB) and Java API for RESTful Architecture. Following the principles of RESTful Architecture, I chose to model the Resources based on the entities that constitute the business domain. Enterprise Java Beans are used as an intermediate layer to take advantage of the transaction management support that they offer. IBM® Rational® Software Architect offers set of predefined model to code transformations that support development of Java Enterprise Applications with the mainstream technologies.

How to automatically produce the design and implementation of EJBs and JAX-RS services by applying custom model transformations when starting from the JPA domain model and by doing this, can you increase the speed and productivity of design and development of Java Enterprise Applications are the two goals of this article.

Starting from the UML model that represents business domain in the context of JPA technology it is possible to generate Java artifacts that contain Java Persistence API entities. Furthermore, the UML to EJB 3.0 transformation generates Enterprise Java Beans and Java code from UML model elements, extended to capture the EJB technology domain. The JAX-RS code can be obtained by using UML to Java transformation and its extension called UML to JAX-RS transformation extension, in order to generate the code for JAX-RS Web Service elements.
To reach the goal, a new enhancement of Model Driven Architecture support is provided as shown in Figure 1.

Figure 1. New model to model and model to code transformation extensions

To design the EJB and JAX-RS models, you will generate two new model to model transformations.

- JPA to EJB 3.0
- EJB 3.0 to JAX-RS

Implemented as plugins, these model to model transformations extend the Rational Software Architect tool.

The implementation of generated JPA operations occurs when two new transformation extensions are invoked for method implementation. These transformation extensions are:

- UML to EJB 3.0
- UML to JAX-RS

Model to model transformation design and rules definition

In this section, you will design JPA and EJB 3.0 models by using model to model transformations that are built with the Model Transformation Authoring Framework (MTAF). MTAF is a tool used to implement various transformations in IBM Rational Software Architect and IBM® InfoSphere Data Architect. The MTAF Framework enables the definition of transformations between arbitrary domains. It also provides a well-established API that supports both imperative and declarative mappings. Figure 2 shows the design of JPA to EJB 3.0 model to model transformation.

Figure 2. The design of JPA to EJB 3.0 model to model transformation
The main transformation, which extends the `RootTransform` class, is organized as a set of `UMLKindTransform` classes. In these particular examples there is only one instance of the `UMLKindTransform` class. It contains the set of transformation rule classes inherited from the `ModelRule` class.

However, both the JPA and EJB 3.0 designs are based on the same transformation features, shown in Table 1.

### Table 1. Design features of model to model transformations

<table>
<thead>
<tr>
<th>Directionality</th>
<th>Unidirectional</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source – target relationship</td>
<td>Different source and target models. Source model can be a nested package. Target cannot be a nested package.</td>
</tr>
<tr>
<td>Incrementality</td>
<td>Utilization of the Fuse Utility Class that calls the compare and merge framework.</td>
</tr>
<tr>
<td>Management of transformation lifecycle</td>
<td>Controls the start and the end of the transformation by calling IrunTransformationListener interface.</td>
</tr>
<tr>
<td>Mapping type</td>
<td>Allows explicit control of execution of transformation rules. Mapping type is imperative.</td>
</tr>
<tr>
<td>Rules definition</td>
<td>Domain: Meta Object Facility (MOF)</td>
</tr>
<tr>
<td></td>
<td>Application conditions: Implemented in canAccept() method</td>
</tr>
<tr>
<td></td>
<td>Parameterization: generics as a parameter</td>
</tr>
<tr>
<td></td>
<td>Rules scheduling: Rules are executed in a sequential order</td>
</tr>
</tbody>
</table>

### Model to code transformation design

IBM Software Architect includes a predefined UML to Java transformation that you can extend by using the Eclipse extension mechanism. The transformation author can contribute by extending only the part of the transformation which executes one or more UML model elements. The UML to EJB 3.0 and UML to JAX-RS transformation extensions are developed based on this feature. Both extensions are designed to extend the part of UML to Java transformation that transforms the UML Operation class into method declaration.

### Working example

The sample domain design UML model, shown in Figure 3, is used throughout the article as an input to model transformations. Starting from this model the EJB 3.0 and JAX-RS design and implementation are automatically generated.

The sample JPA model shown in Figure 3 consists of two entities: **User** and **Pages** with the composition relationship between them. The idea behind this business domain is to create the application called *VirtualDiary* which allows a user to have an online diary consisting of an arbitrary number of pages. Each page has a title, content, and a creation date. Each user is associated with an unlimited number of pages. The user profile contains the following information.

- Name
- Surname
The user name and password fields are important because they allow the user to login, and for this specific operation the custom named query is created.

**Figure 3. Sample JPA domain model**

---

**Prerequisites**

Before you continue through the steps in the article, pre- and post-conditions must be met. Those conditions are:

**Supported execution environments**

Deploy the JPA, EJB and Web project that contain generated code on the IBM® WebSphere® Application Server or IBM® WebSphere® Liberty Profile.

**The application of built-in and custom UML profiles**

Apply the following UML profiles to the EJB target model.

- **EJB 3.0 Transformation Profile**: Built-in profile available in Rational Software Architect
- **Java Transformation Profile**: Built-in profile available in Rational Software Architect
- **JPA Profile for JPA to EJB transformation**: Custom profile, available in the `com.ibm.rational.transform.demo.jpa2ejb` plugin
- **EJB Profile for JPA to EJB transformation**: custom profile, available in the `com.ibm.rational.transform.demo.jpa2ejb` plugin

Apply the following UML profiles to the JAX-RS target model.

- **REST**: Built-in profile available in Rational Software Architect
- **JAX - RS**: Built-in profile available in Rational Software Architect
- **CDI Profile for EJB to JAX-RS transformation**: Custom profile, available in the plugin: `com.ibm.rational.transform.demo.ejb2jaxrs`

**Imported model libraries**

Import the J2EE helper library, available in `com.ibm.rational.transform.demo.jpa2ejb` and `com.ibm.rational.transform.demo.ejb2jaxrs` plugins, to the EJB and JAX-RS target model. Import the Java Primitive Types library, available in rational Software Architect to the JAX-RS model.
Design and implement EJB 3.0 methods

This section of the article shows you how to generate the EJB 3.0 model and code starting with the domain JPA model.

JPA to EJB 3.0 model to model transformation

EJBs are generally used as a façade to JPA Entities, because they offer transaction management capabilities. The UML model of the JPA Entities represents the source to the JPA to EJB 3.0 model to model transformation. The transformation output is a UML model of EJB façades. The mapping rules are provided throughout this article and their output examples are shown using general names of UML elements.

Mapping conversion for the Package rule

Description

The Package rule transforms every UML package in the source model to a UML package in the target model. The package structure is followed and every generated package is appended with one nested package named ejbs.

Elements

- JPA Element
- Transformation output

Diagrams

Figure 4. Example of input JPA model to Package rule

Figure 5. Generated UML packages in EJB model

Application condition

The rule accepts the source package as an input parameter.

Mapping conversion for the Entity to Bean rule

Description

The entity in a JPA model is mapped to either a Stateless or a Stateful bean. If there is an implementation relationship between the session bean and interface, the JPA model is also mapped to one local and one remote interface (at most). You can choose to generate a Singleton bean, but only in combination with Stateful or Stateless bean types. You specify the type of bean.
and the interface as parameters of the transformation. You can also choose the Container or Bean Transaction Management type.

- **Container**: When you create the session bean, the persistence context is injected and the new Entity Manager is instantiated.
- **Stateful**: The persistence context is extended. Every session bean and interface implements operations to:
  - Create the entity
  - Update the entity
  - Delete the entity
  - Find the entity by ID. If the ID value of the entity is composite, all fields of the key are passed to the `findById()` operation.
- **Singleton session bean**: Instead of CRUD operations, singleton session beans contain three public methods:
  - `createCache()`
  - `deleteCache()`
  - `refreshCache()`

**Elements**

- JPA Element
- Transformation output

**Diagrams**

**Figure 6. Example of input JPA entity with one id attribute**

![Diagram of input JPA entity with one id attribute]

**Bean and interface type**

- Stateless session beans with Local and Remote Interface

**Transaction management type**

- Container

**Figure 7. Generated session bean with two interfaces as an output of Entity to Bean rule**

![Diagram of generated session bean with two interfaces]
Application condition

The rule accepts the class with the application of «Entity» stereotype, as an input parameter. This is available in the Java Persistence API Transformation profile.

Mapping conversion for the property rule

Description

You can choose to generate a finder operation for each field of the Entity for every type of bean based on the value of the Generate Query methods for attributes. For Singleton beans these methods are private.

Elements

- JPA Element
- Transformation output

Diagrams

Figure 8. Example of input JPA Entity with one id and non-id attributes

Figure 9. Session bean class with generated findByAttributeName operation

Application condition

The rule is called if the input parameter is the class with «Entity» stereotype and if a transformation property Generate Query method for attributes is set to be true.

Mapping conversion for the Named query rule

Description

Session beans can implement additional operations for every custom named query defined in the source entity. The parameters of the named query become parameters of the operation. The query is parsed to determine the type and number of the parameters. The new dependency relationships are also created. The first dependency relationship connects the source operation associated with the @NamedQuery stereotype and the second connects the source operation with generated operation. This step is made in order to retrieve the query name in a later step of the development process (when the UML-to-EJB transformation extension is invoked). The second one is generated between the entity class and the EJB container class from the resulting operation.
Note:
This relationship is used in the implementation of JAX-RS methods.

Elements

- JPA Element
- Transformation output

Diagrams

Figure 10. Specification of named query element in JPA entity

[Diagram of JPA entity with named query element]

Named query string value:

Figure 11. Specification of custom named query string

[Diagram showing custom named query string]

Figure 12. Generated EJB 3.0 operation from Named query rule

[Diagram showing generated EJB 3.0 operation]

Application condition

The rule accepts the UML operation as an input parameter if:

- The stereotype «NamedQuery» is applied and it is contained in Java Persistence API Transformation profile.
- The transformation property Generate operation for custom named queries is set to true.

Table 2 is an overview of transformation parameters, including the parameters defined in the custom defined configuration tab.

Table 2. Transformation parameters in JPA to EJB 3.0 model to model transformation

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Column headers]</td>
<td>[Column headers]</td>
<td>[Column values]</td>
</tr>
</tbody>
</table>
After you define the business model, the next step is to generate Java code for the JPA entities. To do this, call the predefined UML to JPA Transformation. In order for the transformation to parse any custom defined named queries, it is necessary to implement the entities before you invoke the JPA to EJB 3.0 model to model transformation. Perform the following steps to obtain the EJB 3.0 model.

1. File > New > Transformation Configuration
2. Select JPAtoEJBTransformation from the JPA to EJB model to model transformation folder
3. Enter the name and configuration file destination
4. Click Next
5. Select the source and target model
6. Click Next
7. The extensions page shows the list of available transformation extensions. Check the ID box for: com.ibm.rational.transform.demo.jpa2ejb.extension.GUI as shown in Figure 13.
8. Click Next

Figure 13. Extension tab in JPA to EJB 3.0 transformation configuration file

9. Figure 14 shows the first set of transformation properties.
   - To Generate query methods for attributes, set the value of this property to true
   - To Generate operations from custom NamedQueries, set the value of this property to true
   - Enter the JPA project name where the JPA entities are implemented.
10. Click Next
11. Choose the bean type, transaction, and local interface for each entity, as shown in Figure 15.

Figure 15. Custom tab with additional properties in JPA to EJB 3.0 transformation configuration file

12. To start the transformation click **Finish**.
13. Open the first page of the transformation configuration file.
14. Click **Run**.

The resulting EJB 3.0 model is shown in Figure 16.

Figure 16. EJB 3.0 model as a result of JPA to EJB 3.0 model to model transformation
A limitation with the JPA to EJB 3.0 model to model transformation is the transformation of inherited classes. JPA inheritance is complex because it can be mapped to different database structures (Single Table, Joined or Table Per Class). Because of this complexity, it is out of the scope of this article.

**UML to EJB model to code transformation extension**

The EJB façades need to implement all CRUD methods to access the JPA entities. The implementation of these methods is based on programming patterns that are similar to the JPA Manager Beans supported by Rational Software Architect and IBM(R) Rational(R) Application Developer. However, programming patterns that reflect the behavior of Stateful and Singleton Beans are added to the UML to EJB model to code transformation extension. These patterns are distinguished by the transaction management type. As mentioned in the JPA to EJB 3.0 model to model transformation section, you can choose between the following items:

- A container transaction where the EJB container sets the boundaries of the transactions
- A bean managed transaction: the EJB façades are explicitly marked.

Use the Entity Manager API to:

- Create an entity
- Update an entity
- Remove persistent entity instances
- Find an entity by its primary key
- Query the entities

For container managed transactions, the Entity Manager is created by the container which uses the information in the persistence.xml. The Entity Manager is injected into the EJB class through the [@PersistenceContext](#) annotation.

For a bean managed transaction, the persistence context is not propagated to session beans, and, the lifecycle of the Entity Manager instance is managed by the application. In this case the application creates Entity Manager objects by using the `createEntityManager()` method of `[javax.Persistence.EntityManagerFactory](#)`. 

You need to create a new transformation configuration tab to generate EJB 3.0 code with the implementation of CRUD methods. Create the tab.

1. File > New > Transformation Configuration
2. Select UML-to-EJB 3.0 from the Java Transformations folder
3. Enter the name and configuration file destination
4. Click Next
5. Select the generated EJB 3.0 model as a source and the EJB or Web project as a target
6. Click Finish and open the transformation configuration file
7. Under the tab Extensions, Check the box for the ID associated with `com.ibm.xtexts.transform.uml2.ejb3.java.internal.UML2EJBTransformExtension`
Click the **Run** button on the first page of the transformation configuration file to start the transformation.

The EJB 3.0 code is generated with the implementation of methods specified in the EJB 3.0 design.

**Generating the JAX-RS design and implementation of JAX-RS methods**

This section shows how to generate the JAX-RS design and implementation of its methods by using the EJB 3.0 to JAX-RS model to model transformation. The starting model in this section is the generated EJB 3.0 model from the previous section.

**EJB to JAX-RS model to model transformation**

The EJB façades can be exposed as RESTful resources, in order to be used by Web 2.0 applications. The UML model of the EJB façades is transformed into a UML model of the JAX-RS resource by applying a custom EJB 3.0 to JAX-RS model to model transformation. The mapping rules are defined with usage of general names for the UML elements.

**Mapping conversion for the Packagerule**

**Description**

If the target is a model, a new package is created in the target model. If the target is a package, then the target package must be the root. The package structure of the source model is recreated in the target model, but each nested package in the source model called **ejbs** is transformed into package called **jaxrs**.

**Elements**

- JPA element
- Transformation output

**Diagrams**
Figure 18. Package structure of input EJB 3.0 model element

Figure 19. Generated package structure in JAX-RS model

Application condition

The rule accepts the source package as an input parameter.

Mapping conversion for the Application rule

Description

The transformation creates a new UML class. The name is constructed as the name of target JAX-RS model and appended with the suffix `Application`. The class is generated in the root package of the JAX-RS model. From the REST profile, the stereotype `Application` is applied to the generated class.

Elements

- JPA element
- Transformation output

Diagrams

Figure 20. Input EJB 3.0 model element to Application rule

Figure 21. Generated Application class element as a result of Application rule

Application condition

The rule accepts the source package as an input parameter.
Mapping conversion for the Bean to Resource rule

Description

Every session bean (with an application of either a **Stateless** or **Singleton** stereotype) is mapped to exactly one resource. The structure of the JPA model determines the structure of the JAX-RS model. Because of this, it is important to keep a dependency relationship between the source EJB class and its related entity. In this context, UML classes that represent JPA entities are classified in two groups, depending on whether they are the part end of a composite aggregation with another Entity Class:

- **The JPA entity is not "contained" in any other JPA Entity** – The corresponding EJB Class element is transformed to a JAX-RS Root Resource. A new dependency relationship is created between the Root Resource and the Application class with a "{name}/{entity name}". The «Path» stereotype is applied to the dependency. A new attribute of the corresponding session bean type is added to the resource class with the application of «Inject» stereotype. This stereotype is available in the custom CDI Profile.

- **The JPA entity is "contained" in another JPA Entity** – The corresponding EJB Class element is transformed to a JAX-RS Sub-resource. A Dependency relationship is created between the Sub-resource element and its container resource. Because Sub-resources can only be reached from their parent, a new Sub-resource locator operation is added to the resource container. The URI name of the relationship between the root and the Sub-resource is "/{entity name}/{id attribute}". The new attribute of the corresponding session bean type is also added. Locate the bean using Java Naming and Directory Interface (JNDI).

Another important aspect of this rule is the visibility scope of the generated resources. The Singleton bean is mapped to resource with «ApplicationScoped» stereotype, while Stateless bean to resource with «RequestScoped» stereotype.

Elements

- JPA Element
- Transformation output

Diagrams

Figure 22. Input session bean generated from an entity that is not “contained” in any other JPA entity
Figure 23. Generated Root Resource class element with dependency relationship to Application class

Application condition

The rule accepts the UML class element with an application of one of either the «Stateless» or «Singleton» stereotype.

Mapping conversion for the Operation rule

Description

From each operation in a session bean a new operation in JAX-RS Resources is created with the same name and parameters as in the originating method. The stereotype that denotes the request method designator is applied to each of the generated methods. Table 4 shows the application of these stereotypes.

Elements

- JPA element
- Transformation output

Diagrams

Figure 24. Session bean operations as an input to the Operation Rule

Figure 25. Generated POST, PUT and DELETE operations in the Resource class and two GET operations with their path values
Application condition

The rule accepts as source elements UML Operations, contained in a UML class of the following stereotypes: «Stateless» or «Singleton». Additionally, the container must have a dependency relationship to a JPA entity and the operation must be public.

The stereotype that denotes the request method designator is applied to each of generated methods is shown in Table 4.

Table 4. Application of stereotypes that denote HTTP method designator on JAX-RS methods

<table>
<thead>
<tr>
<th>Operation type</th>
<th>Stereotype that denotes method designator</th>
<th>Path value</th>
</tr>
</thead>
<tbody>
<tr>
<td>create</td>
<td>«POST»</td>
<td>is not specified</td>
</tr>
<tr>
<td>update</td>
<td>«PUT»</td>
<td>is not specified</td>
</tr>
<tr>
<td>delete</td>
<td>«DELETE»</td>
<td>is not specified</td>
</tr>
<tr>
<td>find by id</td>
<td>«GET»</td>
<td><em>/[id]/</em></td>
</tr>
<tr>
<td>other finders</td>
<td>«GET»</td>
<td><em>/ /</em> plus name of the operation</td>
</tr>
</tbody>
</table>

The transformation parameters, including those defined in the custom defined configuration tab are shown in Table 5.

Table 5. Transformation parameters of EJB 3.0 to JAX-RS model to model transformation

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consume media types for create, update and delete type of methods</td>
<td>String</td>
<td>JSON, XML, JSON (XML)</td>
</tr>
<tr>
<td>Allowed combination of consume media type and annotation applied on input parameters for custom finder methods</td>
<td>String</td>
<td>@FormParam/FORM_URLENCODED /JSON /XML /JSON(XML) @QueryParam/</td>
</tr>
<tr>
<td>Produces media types</td>
<td>String</td>
<td>JSON, XML, JSON (XML)</td>
</tr>
</tbody>
</table>

The generated methods can use and produce messages either in JSON or XML format. The assumption is that the resulting JAX-RS API is mainly used with the WebSphere Application Server or WebSphere Liberty Profile. However, these environments provide the support for serialization and deserialization of JSON and XML formats.

Generate the JAX-RS design from a previously generated EJB 3.0 mode.

Important

Dependencies with JPA model elements must be preserved

1. File > New > Transformation Configuration
2. Select **EJB-to-JAX-RS Transformation**
3. Specify the name and configuration file destination then click **Next**.
4. Select the previously generated EJB 3.0 model as the source model.
5. Select the target model.
6. Click **Next**.
7. The page that contains the list of available transformation extensions opens. Check the extension with the ID **com.ibm.rational.transform.demo.ejb2jaxrs.extension.GUI**, as shown in Figure 26.
8. Click **Next**

**Figure 26. Extension tab of EJB 3.0 to JAX-RS transformation configuration file**

9. On the last page of the Transformation properties wizard, choose the method type and what that type **Consumes** and **Producers**.
10. Choose the annotation type to apply on parameters for some methods
11. Click **Finish**. Figure 27 shows this final page.

**Figure 27. Parameters of EJB 3.0 to JAX-RS transformation configuration file**

12. Open the first page of the transformation configuration file. To invoke the transformation click **Run**.

Figure 28 shows the generated JAX-RS model.
The programming patterns chosen to generate the implementation of JAX-RS methods are based on resource type. The JAX-RS root resource class obtains a reference to an instance of an enterprise bean through the context dependency injection (CDI). This means that the reference of an enterprise bean in the root resource is annotated with @Inject annotation. However, in a sub resource, the reference of the enterprise bean is obtained via JNDI lookup, using the Java Naming and Directory Interface syntax. According to the JAX-RS specification, the sub resource cannot be exposed as a CDI bean. In this transformation, the java:global JNDI namespace is used as the portable way of finding enterprise beans via JNDI lookups. The JNDI address is constructed using the following form.

```
java:global[/application name]/module name/enterprise bean name
```

The application name is optional and required only if the application is packaged within an EAR. You can specify this as a transformation parameter in the configuration tab extension.

If an application name is defined, it is included in the lookup string. The module name is inspected in either:

- The `ejb-jar.xml` file. Used when EJB project is the name and session beans are implemented
- The `web.xml` descriptor. If a custom module name is not specified, the transformation takes the default module name in the JNDI address string.

Each JAX-RS operation calls the corresponding EJB method.

Perform these steps to generate the JAX-RS code:

1. File > New > Transformation Configuration
2. Select UML to Java from the Java Transformations folder.
3. Enter the name and configuration file destination.
4. Click **Next**.
5. Select the generated EJB 3.0 model as a source and Web project as a target.
6. Click **Finish**.
7. On the extensions tab select the check boxes for those with the ID `com.ibm.xtools.transform.uml2.jaxrs` and `com.ibm.xtools.uml2jaxrsExtension`

**Figure 29. Extension tab of UML to Java Transformation**

8. Click **Run** on the main page of the transformation configuration file to start the transformation.

**Conclusion**

The initial solution, presented in this article is manual. The time and detail involved with the manual solution significantly increases depending on the increasing complexity of the domain JPA model.

The proposed solution to generate the EJB and JAX-RS models and code is fully automated. The process takes a couple of hours, regardless of the complexity of the initial JPA model.

Binding choices was also discussed in this article. They are important in order to achieve the predictability of the resulting JAX-RS API, which is another important aspect of development productivity.

Following the well established standards and formats, best practices and options available in this article should lead to a consistent, easy to understand, and predictable API design.

© Copyright IBM Corporation 2015  
**Trademarks**  
(www.ibm.com/developerworks/ibm/trademarks/)