Seam REST Module

Reference Guide

Jozef Hartinger

Introduction

Seam REST is a lightweight module that aims to provide additional integration with technologies within the Java EE platform as well as third party technologies.

Seam REST is independent of CDI and JAX-RS implementations and thus fully portable between Java EE 6 environments.

Installation

The Seam REST module requires a Java EE 6 compliant server such as *JBoss Application Server* [http://www.jboss.org/jbossas] or *GlassFish* [https://glassfish.dev.java.net/] to run on.

To use the Seam REST module, add `seam-rest` and `seam-rest-api` jars into the web application. If using Maven, add the following dependency into the web application's `pom.xml` configuration file.

Example 1.1. Dependency added to pom.xml

```
<dependency>
   <groupId>org.jboss.seam.rest</groupId>
   <artifactId>seam-rest</artifactId>
   <version>${seam.rest.version}</version>
</dependency>
```

Declarative Exception Mapping

The JAX-RS specification comes with Exception Mapping Providers as a standard mechanism for treating Java exceptions. This approach works fine for complex cases, however, the exception handling logic is often trivial and not worth implementing Exception Mapper Providers for each exception type. In these situations, declarative approach is more appropriate. The Seam REST module allows exception types to be bound to HTTP responses declaratively.

For each exception type, it is possible to specify the status code and the error message of the HTTP response. There are two ways of exception mapping configuration in Seam REST.

2.1. Programmatic configuration

Seam REST exception mapping can be configured from Java code. Firstly, create an `ExceptionMappingConfiguration` subclass which `@Specializes` the provided one. Then, implement a `@PostConstruct`-annotated method in which the `ExceptionMapping` definitions are added as shown in the following example.

Example 2.1. Programmatic exception mapping configuration

```
@Specializes
public class CustomExceptionMappingConfiguration extends ExceptionMappingConfiguration {
{
   @PostConstruct
   public void setup()
   {
      addExceptionMapping(new ExceptionMapping(NoResultException.class, 404, "Requested resource does not exist."));
     addExceptionMapping(new ExceptionMapping(IllegalArgumentException.class, 400, "Illegal parameter value."));
   }
}
```

When the `NoResultException` is thrown at runtime, the client receives the following HTTP response.

Table 2.1. ExceptionMapping properties

Name	Required	Default value	Description
exceptionType	true	-	Fully-qualified class name of the exception class

Name	Required	Default value	Description
statusCode	true	-	HTTP status code
message	false	-	Error message sent within the HTTP response
interpolateMessageBody	false	true	Enables/Disables EL interpolation of the error message

2.2. XML configuration

An alternative and more practical way of configuration is to use the Seam XML module to configure the `ExceptionMappingConfiguration` and `ExceptionMapping` classes in XML.

Firstly, the Seam XML module needs to be added to the application. If using maven, this can be done by specifying the following dependency:

Example 2.2. Seam XML dependency added to the pom.xml file.

```
<dependency>
   <groupId>org.jboss.seam.xml</groupId>
   <artifactId>seam-xml-config</artifactId>
   <version>${seam.xml.version}</version>
</dependency>
```

For more information on the seam-xml module, refer to the *Seam XML reference documentation* [http://docs.jboss.org/seam/3/xml-config/latest/reference/en-US/html_single/] Once the Seam XML module is added, specify the configuration in the `seam-beans.xml` file, located in the `WEB-INF` or `META-INF` folder of the web archive.

Example 2.3. Exception mapping configuration in seam-beans.xml

```
<rest:ExceptionMappingConfiguration>
   <s:replaces/>
   <rest:exceptionMappings>
      <s:value>

 <rest:ExceptionMapping exceptionType="javax.persistence.NoResultException" statusCode="404">
            <rest:message>Requested resource does not exist.</rest:message>
         </rest:ExceptionMapping>
      </s:value>
      <s:value>
```

```
 <rest:ExceptionMapping exceptionType="java.lang.IllegalArgumentException" statusCode="400">
            <rest:message>Illegal parameter value.</rest:message>
         </rest:ExceptionMapping>
      </s:value>
   </rest:exceptionMappings>
</rest:ExceptionMappingConfiguration>
```

Furthermore, EL expressions can be used in message templates to provide dynamic and more descriptive error messages.

Example 2.4. Exception mapping configuration in seam-beans.xml

```
<rest:ExceptionMapping exceptionType="javax.persistence.NoResultException" statusCode="404">
      <rest:message>Requested resource with id #{pathParameters['id']} does not exist.</
rest:message>
</rest:ExceptionMapping>
```

2.3. Exception Mapping

When an exception occurs at runtime, the `SeamExceptionMapper` first looks for a matching `ExceptionMapping`. If it finds one, it creates an HTTP response with the specified status code and error message.

The error message is marshalled within a JAXB object. As a result, the error message is available in multiple media formats. The most commonly used formats are XML and JSON. Most JAX-RS implementations provide media providers for both of these formats. In addition, the error message is also available in plain text.

Example 2.5. Sample HTTP response

```
HTTP/1.1 404 Not Found
Content-Type: application/xml
Content-Length: 123

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<error>
   <message>Requested resource does not exist.</message>
</error>
```

Bean Validation Integration

Bean Validation (JSR-303) is a specification introduced as a part of Java EE 6. It aims to provide a standardized way of validating the domain model across all application layers.

The Seam REST module integrates with the Bean Validation specification. This allows message bodies of HTTP requests to be validated using this standardized mechanism.

3.1. Validating HTTP requests

Firstly, enable the `ValidationInterceptor` in the `beans.xml` configuration file.

```
<interceptors>
   <class>org.jboss.seam.rest.validation.ValidationInterceptor</class>
</interceptors>
```

Then, enable validation of a particular method by decorating it with the `@ValidateRequest` annotation.

```
@PUT
@ValidateRequest
public void updateTask(Task incommingTask)
{
...
}
```

By default, the message body (the parameter with no annotations) is validated. If the object is valid, the web service method is executed. Otherwise, the `ValidationException` exception is thrown.

The `ValidationException` exception is a simple carrier of constraint violations found by the Bean Validation provider. The exception can be handled by an `ExceptionMapper` .

Seam REST comes with a built-in `ValidationExceptionMapper` which is registered by default. The exception mapper converts the `ValidationException` to an HTTP response with 400 (Bad request) status code. Furthermore, messages relevant to the violated constraints are sent within the message body of the HTTP response.

Example 3.1. HTTP response

```
HTTP/1.1 400 Bad Request
Content-Type: application/xml
Content-Length: 129
```

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<error>
   <messages>
      <message>Name length must be between 1 and 100.</message>
   </messages>
</error>
```

Besides the message body, the JAX-RS specification allows various parts of the HTTP request to be passed as method parameters. These parameters are usually HTTP form parameters, query parameters, path parameters, headers, etc. In order to prevent an oversized method signature when the number of parameters is too large, JAX-RS implementations provide implementations of the *Parameter Object pattern* [http://sourcemaking.com/refactoring/introduce-parameter-object]. For example *RESTEasy Form Object* [http://docs.jboss.org/resteasy/docs/2.0.0.GA/userguide/html/_Form.html] or *Apache CXF Parameter Bean* [http://cxf.apache.org/docs/jax-rs.html#JAX-RS-Parameterbeans]. Seam REST validates these Parameter Objects by default.

Example 3.2. RESTEasy @Form object MyForm.java

```
public class MyForm {
   @FormParam("stuff")
   private int stuff;

   @HeaderParam("myHeader")
   private String header;

   @PathParam("foo")
   public void setFoo(String foo) {...}
}

@POST
@Path("/myservice")
@ValidateRequest
public void post(@Form MyForm form) {...}
```

Table 3.1. @ValidateRequest annotation properties

@ValidateRequest attribute	Description	Default value
validateMessageBody	Enables/Disables validation of message body parameters.	true
validateParameterObjects	Enables/Disables validation of parameter objects.	true

@ValidateRequest attribute	Description	Default value
groups	Validation groups to be used for validation.	javax.validation.groups.Default

3.2. Using validation groups

In some cases, it is desired to have a specific group of constraints used to validate the web service parameters. These constraints are usually weaker compared to the default constraints of a domain model. Take partial updates as an example.

Consider the following example:

Example 3.3. Employee.java

```
public class Employee {
    @NotNull
    @Size(min = 2, max = 30)
    private String name;
    @NotNull
    @Email
    private String email;
    @NotNull
    private Department department;

    // getters and setters
}
```

The Employee resource in the example above is not allowed to have the null value specified in any of its fields. Thus, the entire representation of a resource (including the department and related object graph) must be sent to update the resource.

When using partial updates, only the values of modified fields are required to be sent within the update request. Only the non-null values of the received object are updated. Therefore, two groups of constraints are needed: one for partial updates (including @Size and @Email, excluding @NotNull) and the default one (@NotNull).

A validation group is a simple Java interface:

Example 3.4. PartialUpdateGroup.java

```
public interface PartialUpdateGroup {}
```

Example 3.5. Employee.java

```
@GroupSequence({ Default.class, PartialUpdateGroup.class })                    (3)
public class Employee {
   @NotNull                                                                    (1)
   @Size(min = 2, max = 30, groups = PartialUpdateGroup.class)                 (2)
   private String name;
   @NotNull
   @Email(groups = PartialUpdateGroup.class)
   private String email;
   @NotNull
   private Department department;

   // getters and setters
}
```

(1) The `@NotNull` constraint belongs to the default validation group.

(2) The `@Size` constraint belongs to the partial update validation group.

(3) The `@GroupsSequence` annotation indicates that both validation groups will be used by default (e.g. when persisting the entity).

Finally, the `ValidationInterceptor` is configured to validate the `PartialUpdateGroup` group only.

Example 3.6. EmployeeResource.java

```
@Path("/{id}")
   @PUT
   @Consumes("application/xml")
   @ValidateRequest(groups = PartialUpdateGroup.class)                         (1)
   public void updateEmployee(Employee e, @PathParam("id") long id)
   {
      Employee employee = em.find(Employee.class, id);
      if (e.getName() != null)                                                 (2)
      {
         employee.setName(e.getName());
      }
      if (e.getEmail() != null)
      {
         employee.setEmail(e.getEmail());
      }
```

```
}
```

1. The partial update validation group is used for web service parameter validation.
2. Partial update — only the not-null fields of the transferred representation are used for update. The null fields are not updated.