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.

1.1. Basics

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-api</artifactId>
        <version>${seam.rest.version}</version>
    </dependency>

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

1.2. Transitive dependencies

Besides, Seam REST has several transitive dependencies (which are added automatically when using maven). See *Table 6.1, “Transitive dependencies”* for more details.

1.3. Registering JAX-RS components explicitly

The Seam REST module hooks into the exception processing mechanism of JAX-RS by registering `SeamExceptionMapper`. Besides, `TemplatingMessageBodyWriter` is registered to provide templating support.

These components registered by default if classpath scanning of JAX-RS resources and providers is enabled (an empty `javax.ws.rs.coreApplication` is provided).

```
@ApplicationPath("/api/*")
public class MyApplication extends Application {}
```

Otherwise, if the Application's `getClasses()` method is overriden to select resources and providers explicitly, `SeamExceptionMapper` must be selected as well.

```
@ApplicationPath("/api/*")
public class MyApplication extends Application
{
  @Override
  public Set<Class<?>> getClasses()
  {
    Set<Class<?>> classes = new HashSet<Class<?>>();
    ...
    ...
    ...
    classes.add(SeamExceptionMapper.class);
    classes.add(TemplatingMessageBodyWriter.class);
    return classes;
  }
}
```

Exception Handling

The JAX-RS specification defines Exception Mapping Providers as a standard mechanism for treating Java exceptions. The Seam REST module comes with and alternative approach which is more consistent with the CDI programming model, easier to use and still portable.

Firstly, Seam Catch is plugged in which allows exceptions that occur in different parts of an application to be treated uniformly. Besides, a builtin exception handler is provided which enables simple exception-mapping rules to be defined declaratively.

2.1. Seam Catch Integration

Seam Catch can be used within the Seam REST module for dealing with exceptions. As a result, an exception that occurs during an invocation of a JAX-RS service is routed through the Catch exception handling mechanism which is similar to the CDI event bus and lets exception handling logic to be implemented in a loosely-coupled fashion.

The following code sample demonstrates a simple exception handler that converts the `NoResultException` exception to the 404 HTTP response.

Example 2.1. Seam Catch Integration - NoResultException handler

```
@HandlesExceptions                                                        (1)
public class ExceptionHandler
{
   @Inject @RestResource
   ResponseBuilder builder                                                (2)

   public void handleException(@Handles @RestRequest CaughtException<NoResultEx(3)ception> event)
   {
      builder.status(404).entity("The requested resource does not exist.");
   }
}
```

(1) The `@HandlesExceptions` annotation marks the `ExceptionHandler` bean as capable of handling exceptions.

(2) The `ResponseBuilder` for creating the HTTP response is injected.

(3) A method for hanling `NoResultException` instances. Note that the `ExceptionHandler` can define multiple exception handling methods for various exception types.

Similarly to the CDI event bus, exceptions to be handled by a handler method can be filtered by qualifiers. In the example above, we are only interested in exceptions that occur in a JAX-RS service invocation. (As opposed to all exceptions of the given type that occur in the application - in the view layer for example.) Thus, the `@RestRequest` qualifier is used.

Catch integration is optional and only enabled when Catch libraries are available on classpath. For more information on Seam Catch, see *Seam Catch reference documentation* [http://docs.jboss.org/seam/3/catch/latest/reference/en-US/html/] .

2.2. Declarative Exception Mapping

Often, exception-mapping rules are simple. Thus, they do not really need to be implemented in Java. Instead, declarative approach is more appropriate in these situations. 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.

2.2.1. 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 2.2. 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."));
   }
}
```

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

2.2.2. 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 2.3. 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 2.4. Exception mapping configuration in seam-beans.xml

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

 <exceptions:ExceptionMapping exceptionType="javax.persistence.NoResultException" statusCode="404">
            <exceptions:message>Requested resource does not exist.</exceptions:message>
         </exceptions:ExceptionMapping>
      </s:value>
      <s:value>
 <exceptions:ExceptionMapping exceptionType="java.lang.IllegalArgumentException" statusCode="400">
              <exceptions:message>Illegal parameter value.</exceptions:message>
          </exceptions:ExceptionMapping>
      </s:value>
  </exceptions:exceptionMappings>
</exceptions:ExceptionMappingConfiguration>
```

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

Example 2.5. Exception mapping configuration in seam-beans.xml

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

2.2.3. 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 2.6. 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 incomming 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)
{
...
}
```

3.1.1. Validating entity body

By default, the entity parameter (the parameter with no annotations that represent the body of the HTTP request) 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` or Seam Catch handler. .

Seam REST comes with a built-in `ValidationException` handler which is registered by default. The exception handler 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>
```

3.1.2. Validating resource fields

Besides the message body, the JAX-RS specification allows various parts of the HTTP request to be injected into the JAX-RS resource or passed as method parameters. These parameters are usually HTTP form parameters, query parameters, path parameters, headers, etc.

Example 3.2. JAX-RS resource

```
public class PersonResource
{
    @QueryParam("search")
    @Size(min = 1, max = 30)
    private String query;
    @QueryParam("start")
    @DefaultValue("0")
    @Min(0)
    private int start;
    @QueryParam("limit")
    @DefaultValue("20")
    @Min(0) @Max(50)
    private int limit;
...
```

If a method of a resource is annotated with `@ValidateRequest` annotation, the fields of a resource are validated by default.

Important

Since the JAX-RS injection occurs at resource creation time only, do not use the JAX-RS field injection for other than `@RequestScoped` resources.

Warning

As a consequence of *CDI-6* [https://jira.jboss.org/browse/CDI-6], some containers are not able to perform JAX-RS field injection on an intercepted CDI bean. In this situation, JAX-RS setter injection can be used as a workaround.

3.1.3. Validating other method parameters

The JAX-RS specification allows path parameters, query parameters, matrix parameters, cookie parameters and headers to be passed as parameters of a resource method.

Example 3.3. JAX-RS method parameters

```
@GET
public List<Person>search(@QueryParam("search") String query,
   @QueryParam("start") @DefaultValue("0") int start,
   @QueryParam("limit") @DefaultValue("20") int limit)
```

Note

Currently, Seam REST does not validate primitive method parameters (including String ones). Therefore, either use resource field validation described in *Section 3.1.2, "Validating resource fields"* or read further and use parameter objects.

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] . These objects aggregate multiple parameters into a single 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] . These parameters can be validated by Seam REST.

Example 3.4. RESTEasy parameter object

```
public class MyForm {
  @FormParam("stuff")
  @Size(min = 1, max = 30)
  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) {...}
```

3.2. Validation configuration

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
validateResourceFields	Enables/Disables validation of fields fields of a JAX-RS resource.	true
groups	Validation groups to be used for validation.	javax.validation.groups.Default

3.3. 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 3.5. Employee.java

```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 3.6. PartialUpdateGroup.java

```java
public interface PartialUpdateGroup {}
```

Example 3.7. Employee.java

```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 3.8. 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.

Templating support

Seam REST allows HTTP responses to be created using templates. Instead of being bound to a particlar templating engine, Seam REST comes with support for multiple templating engines and support for others can be plugged in.

4.1. Creating JAX-RS responses using templates

REST-based web services are often expected to return multiple representations of a resource. The templating support is useful for producing media formats such as XHTML and it can be also used instead of JAXB to produce domain-specific XML representations of a resource. Besides, almost any other representation of a resource can be described in a template.

To enable templating for a particular method, decorate the method with the `@ResponseTemplate` annotation. Path to a template file to be used for rendering is required.

Example 4.1. @ResponseTemplate in action

```
@ResponseTemplate("/freemarker/task.ftl")
public Task getTask(@PathParam("taskId") long taskId) {
...
}
```

The `@ResponseTemplate` annotation offers several other options. For example, it's possible for a method to offer multiple representations of a resource, each rendered using a different template. In this situation, the `produces` member of the `@ResponseTemplate` annotation is used to distinguish between produced media types.

Example 4.2. Multiple @ResponseTemplates

```
@GET
@Produces( { "application/json", "application/categories+xml", "application/categories-short
+xml" })
@ResponseTemplate.List({
    @ResponseTemplate(value = "/freemarker/categories.ftl", produces = "application/categories
+xml"),
        @ResponseTemplate(value = "/freemarker/categories-short.ftl", produces = "application/
categories-short+xml")
})
public List<Category> getCategories()
```

Table 4.1. @ResponseTemplate options

Name	Required	Default value	Description
value	true	-	Path to the template (e.g. /freemarker/categories.ftl)
produces	false	*/*	Restricts which media type is produced by the template. Useful in situations when a method produces multiple media types, each using a different template.
responseName	false	response	Name under which the object returned by the JAX-RS method is available in the template (e.g. Hello ${response.name})

4.1.1. Accessing the model

There are several ways of accessing the domain data within a template.

Firstly, the object returned by the JAX-RS method is available under the "response" name by default. The object can be made available under a different name using the `responseName` member of the `@ResponseTemplate` annotation.

Example 4.3. hello.ftl

```
Hello ${response.name}
```

Secondly, every bean reachable via an EL expression is available within a template.

Example 4.4. Using EL names in a template

```
#foreach(${student} in ${university.students})
   <student>${student.name}</student>
#end
```

Note

Note that the syntax of the expression depends on the particular templating engine and most of the time differs from the syntax of EL expressions. For example, `${university.students}` must be used instead of `#{university.students}` in a FreeMarker template.

Last but not least, the model can be populated programatically. In order to do that, inject the `TemplatingModel` bean and put the desired objects into the underlying `data` map. In the following example, the list of professors is available under the "professors" name.

Example 4.5. Defining model programatically

```
@Inject
private TemplatingModel model;

@GET
@ResponseTemplate("/freemarker/university.ftl")
public University getUniversity()
{
   // load university and professors
   University university = ...
   List<Professor> professors = ...

   model.getData().put("professors", professors);
   return university;
}
```

4.2. Builtin support for templating engines

Seam REST currently comes with builtin templating providers for FreeMarker and Apache Velocity.

4.2.1. FreeMarker

FreeMarker is one of the most popular templating engines. To enable Seam REST FreeMarker support, bundle the FreeMarker jar with the web application.

For more information on writing FreeMarker templates, refer to the *FreeMarker Manual* [http://freemarker.sourceforge.net/docs/index.html]

4.2.2. Apache Velocity

Apache Velocity is another popular Java-based templating engine. Similarly to FreeMarker support, Velocity support is enabled automatically if Velocity libraries are detected on the classpath.

For more information on writing Velocity templates, refer to the *Apache Velocity User Guide* [http://velocity.apache.org/engine/releases/velocity-1.5/user-guide.html]

4.2.3. Pluggable support for templating engines

All that needs to be done to extend the set of supported templating engines is to implement the `TemplatingProvider` interface. See the *Javadoc* [http://docs.jboss.org/seam/3/rest/latest/api/org/jboss/seam/rest/templating/TemplatingProvider.html] for hints.

4.2.4. Selecting prefered templating engine

In certain deployment scenarios it is not possible to control the classpath completely and multiple template engines may be available at the same time. If that happens, Seam REST fails to operate with the following message:

```
Multiple TemplatingProviders found on classpath. Select the prefered one.
```

Thus, the TemplatingProvider ambiguity needs to be resolved. To do so, choose the prefered templating engine using XML configuration, as demonstrated below.

Example 4.6. Prefered provider

```
<beans xmlns:templating="urn:java:org.jboss.seam.rest.templating">

  <templating:TemplatingMessageBodyWriter preferedTemplatingProvider="org.jboss.seam.rest.templating.freem
        <s:modifies />
    </templating:TemplatingMessageBodyWriter>
</beans>
```

Table 4.2. Builtin Templating Providers

Name	FQCN
FreeMarker	org.jboss.seam.rest.templating.freemarker.FreeMarkerProvider
Apache Velocity	org.jboss.seam.rest.templating.velocity.VelocityProvider

RESTEasy Client Framework Integration

The RESTEasy Client Framework is a framework for writing clients for REST-based web services. It reuses JAX-RS metadata for creating HTTP requests. For more information about the framework, refer to the *project documentation* [http://docs.jboss.org/resteasy/docs/2.0.0.GA/userguide/html/RESTEasy_Client_Framework.html] .

Integration with the RESTEasy Client Framework is optional in Seam REST and only available when RESTEasy is available on classpath.

5.1. Using RESTEasy Client Framework with Seam REST

Let's have the `TaskService` sample interface. A remote server implements this interface exposing a web service for getting task details.

Example 5.1. Sample JAX-RS annotated interface

```
@Path("/task")
@Produces(#application/xml#)
public interface TaskService
{
   @GET
   @Path(#/{id}#)
   Task getTask(@PathParam#(#id#)long id);
}
```

To access the remote web service, let Seam REST build and inject a client object of the web service.

Example 5.2. Injecting REST Client

```
@Inject @RestClient("http://example.com")
private TaskService taskService;

...

Task task = taskService.getTask(1);
```

What really happens is that the Seam REST module injects a proxied `TaskService` interface. Under the hood, the RESTEasy Client Framework converts every method invocation on the `TaskService` to an HTTP request and sends it over the wire to `http://example.com`. The HTTP response is unmarshalled automatically and the response object is returned by the method call.

In addition, it is possible to use EL expressions within the URI.

```
@Inject @RestClient("#{example.service.uri}")
```

5.2. Manual ClientRequest API

Besides proxying JAX-RS interfaces, the RESTEasy Client Framework provides the ClientRequest API for building the HTTP requests manually. For more information on the ClientRequest API, refer to the *project documentation* [http://docs.jboss.org/resteasy/docs/2.0.0.GA/userguide/html/RESTEasy_Client_Framework.html#ClientRequest]

Example 5.3. Injecting ClientRequest

```
@Inject @RestClient("http://localhost:8080/test/ping")
private ClientRequest request;

...

request.accept(MediaType.TEXT_PLAIN_TYPE);
ClientResponse<String> response = request.get(String.class);
```

5.3. ClientExecutor configuration

If not specified otherwise, every request is executed by a default Apache HTTP Client 4 configuration. This can be altered by providing a ClientExecutor bean.

Example 5.4. Custom Apache HTTP Client 4 configuration

```
@Produces
public ClientExecutor createExecutor()
{
   HttpParams params = new BasicHttpParams();
   ConnManagerParams.setMaxTotalConnections(params, 3);
   ConnManagerParams.setTimeout(params, 1000);

   SchemeRegistry schemeRegistry = new SchemeRegistry();
```

```
    schemeRegistry.register(new Scheme("http", PlainSocketFactory.getSocketFactory(), 80));

    ClientConnectionManager cm = new ThreadSafeClientConnManager(params, schemeRegistry);
    HttpClient httpClient = new DefaultHttpClient(cm, params);

    return new ApacheHttpClient4Executor(httpClient);
}
```

Seam REST Dependencies

The Seam REST module depends on the following transitive dependencies at runtime.

Table 6.1. Transitive dependencies

Name	Version
Seam Solder	3.0.0.Beta1
Seam Servlet	3.0.0.Alpha3
Seam Catch	3.0.0.Alpha2
JBoss Logging	3.0.0.Beta4