

Seam REST Module

Reference Guide

Jozef Hartinger

| | |
|---|-----------|
| Introduction | v |
| 1. Installation | 1 |
| 1.1. Basics | 1 |
| 1.2. Transitive dependencies | 1 |
| 1.3. Registering JAX-RS components explicitly | 1 |
| 2. Exception Handling | 3 |
| 2.1. Seam Catch Integration | 3 |
| 2.2. Declarative Exception Mapping | 4 |
| 2.2.1. Annotation-based configuration | 4 |
| 2.2.2. XML configuration | 5 |
| 2.2.3. Declarative exception mapping processing | 6 |
| 3. Bean Validation Integration | 9 |
| 3.1. Validating HTTP requests | 9 |
| 3.1.1. Validating entity body | 9 |
| 3.1.2. Validating resource fields | 10 |
| 3.1.3. Validating other method parameters | 11 |
| 3.2. Validation configuration | 12 |
| 3.3. Using validation groups | 12 |
| 4. Templating support | 15 |
| 4.1. Creating JAX-RS responses using templates | 15 |
| 4.1.1. Accessing the model | 16 |
| 4.2. Built-in support for templating engines | 17 |
| 4.2.1. FreeMarker | 17 |
| 4.2.2. Apache Velocity | 18 |
| 4.2.3. Pluggable support for templating engines | 18 |
| 4.2.4. Selecting preferred templating engine | 18 |
| 5. RESTEasy Client Framework Integration | 19 |
| 5.1. Using RESTEasy Client Framework with Seam REST | 19 |
| 5.2. Manual ClientRequest API | 20 |
| 5.3. ClientExecutor Configuration | 20 |
| 6. Seam REST Dependencies | 23 |
| 6.1. Transitive Dependencies | 23 |
| 6.2. Optional dependencies | 23 |
| 6.2.1. Seam Catch | 23 |
| 6.2.2. Seam Config | 23 |
| 6.2.3. FreeMarker | 24 |
| 6.2.4. Apache Velocity | 24 |
| 6.2.5. RESTEasy | 24 |

Introduction

Seam REST is a lightweight module that provides additional integration of technologies within the Java EE platform as well as third party technologies.

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

Installation

The Seam REST module runs only on Java EE 6 compliant servers such as [JBoss Application Server](http://www.jboss.org/jbossas) [http://www.jboss.org/jbossas] or [GlassFish](https://glassfish.dev.java.net/) [https://glassfish.dev.java.net/].

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). Refer to [Table 6.1, "Transitive dependencies"](#) for more details.

1.3. Registering JAX-RS components explicitly

The Seam REST module registers `SeamExceptionHandler` to hook into the exception processing mechanism of JAX-RS and `TemplatingMessageBodyWriter` to provide templating support.

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

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

Otherwise, if the `Application`'s `getClasses()` method is overridden to select resources and providers explicitly add `SeamExceptionHandler` and `TemplatingMessageBodyWriter`.

```
@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 the mechanism for exception mapping providers as the standard mechanism for Java exception handling. The Seam REST module comes with an alternative approach, which is more consistent with the CDI programming model. It is also easier to use and still remains portable.

The Seam REST module allows you to:

- integrate with Seam Catch and thus handle exceptions that occur in different parts of an application uniformly;
- define exception handling rules declaratively with annotations or XML.

2.1. Seam Catch Integration

Seam Catch handles exceptions within the Seam REST module: as result, an exception that occurs during an invocation of a JAX-RS service is routed through the Catch exception handling mechanism similar to the CDI event bus. This allows you to implement the exception handling logic in a loosely-coupled fashion.

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

Example 2.1. Seam Catch Integration - NoResultException handler

```
@HandlesExceptions ❶
public class ExceptionHandler
{
    @Inject @RestResource
    ResponseBuilder builder ❷

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

- ❶ The `@HandlesExceptions` annotation marks the `ExceptionHandler` bean as capable of handling exceptions.
- ❷ The `ResponseBuilder` for creating the HTTP response is injected.

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

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

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

2.2. Declarative Exception Mapping

Exception-mapping rules are often fairly simple. Thus, instead of being implemented programmatically, they can be expressed declaratively through metadata such as Java annotations or XML. The Seam REST module supports both ways of declarative configurations.

For each exception type, you can specify a status code and an error message of the HTTP response.

2.2.1. Annotation-based configuration

You can configure Seam REST exception mapping directly in your Java code with Java Annotations. An exception mapping rule is defined as a `@ExceptionHandler` annotation. Use an `@ExceptionHandler.List` annotation to define multiple exception mappings.

Example 2.2. Annotation-based exception mapping configuration

```
@ExceptionHandler.List({
    @ExceptionHandler(exceptionType=NoResultException.class,status=404,message="Requested
        resource does not exist."),
    @ExceptionHandler(exceptionType=IllegalArgumentException.class,status=400,message="Illegal
        argument value.")
})
@ApplicationPath("/api")
public MyApplication extends Application {
```

The `@ExceptionHandler` annotation can be applied on any Java class in the deployment. However, it is recommended to keep all exception mapping declarations in the same place, for example, in the `javax.ws.rs.core.Application` subclass.

Table 2.1. @ExceptionHandler properties

| Name | Required | Default value | Description |
|------------------------|----------|---------------|---|
| exceptionType | true | - | Fully-qualified class name of the exception class |
| status | true | - | HTTP status code |
| message | false | - | Error message sent within the HTTP response |
| useExceptionMessage | false | false | Exception error message |
| interpolateMessageBody | false | true | Enabling/disabling the EL interpolation of the error message |
| useJaxb | false | true | Enabling/disabling wrapping of the error message within a JAXB object. This allows marshalling to various media formats such as application/xml, application/json, etc. |

2.2.2. XML configuration

As an alternative to the annotation-based configuration, you can use the Seam Config module to configure the `SeamRestConfiguration` class in XML.

First, add the Seam Config module to the application. If you are using maven, you can do this by specifying the following dependency:

Example 2.3. Seam XML dependency added to the pom.xml file.

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

For more information on the Seam Config module, refer to the [Seam Config reference documentation](http://docs.jboss.org/seam/3/config/latest/reference/en-US/html_single/) [http://docs.jboss.org/seam/3/config/latest/reference/en-US/html_single/]. Once you have added the Seam XML module, 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

```
<rest:SeamRestConfiguration>
  <rest:mappings>
    <s:value>

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

    <exceptions:Mapping exceptionType="java.lang.IllegalArgumentException" statusCode="400">
      <exceptions:message>Illegal value.</exceptions:message>
    </exceptions:Mapping>
    </s:value>
  </rest:mappings>
</rest:SeamRestConfiguration>
```

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

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

```
<exceptions:Mapping exceptionType="javax.persistence.NoResultException" statusCode="404">
  <exceptions:message>Requested resource ({uriInfo.path}) does not exist.</
exceptions:message>
</exceptions:Mapping>
```

2.2.3. Declarative exception mapping processing

When an exception occurs at runtime, the `SeamExceptionHandler` first looks for a matching exception mapping rule. 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 and is thus 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 follows the Bean Validation specification and the incoming HTTP requests can be validated with 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 incomingTask)
{
  ...
}
```

Now, the HTTP request's entity body (the `incomingTask` parameter) will be validated prior to invoking the method.

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, a `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 `ExceptionHandler` 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 the 400 (Bad request) status code. Furthermore, it sends messages relevant to the violated constraints 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 an `@ValidateRequest` annotation, the fields of a resource are validated by default.



Important

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

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 validates only JavaBean parameters (as opposed to primitive types, Strings and so on). Therefore, to validate these types of parameters, 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://source-making.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. To trigger the validation, annotate the parameter with a `javax.validation.Valid` annotation.

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(@Valid @Form MyForm form) {...}

```

3.2. Validation configuration

Table 3.1. @ValidateRequest annotation properties

| @ValidateRequest attribute | Description | Default value |
|----------------------------|--|---------------------------------|
| validateMessageBody | Enabling/disabling validation of message body parameters | true |
| validateResourceFields | Enabling/disabling validation of 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 for validation of web service parameters. These constraints are usually weaker than the default constraints of a domain model. Take partial updates as an example.

Consider the following example:

Example 3.5. 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 values of modified fields are required to be sent within the update request, while the non-null values of the received object are updated. Therefore, two groups of constraints are needed: group for partial updates (including @Size and @Email, excluding @NotNull) and the default group (@NotNull).

A validation group is a simple Java interface:

Example 3.6. PartialUpdateGroup.java

```
public interface PartialUpdateGroup {}
```

Example 3.7. 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 @GroupSequence annotation indicates that both validation groups are used by default (for example, 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) ❶
public void updateEmployee(Employee e, @PathParam("id") long id)
{
    Employee employee = em.find(Employee.class, id);
    if (e.getName() != null) ❷
    {
        employee.setName(e.getName());
    }
    if (e.getEmail() != null)
    {
        employee.setEmail(e.getEmail());
    }
}
```

- ❶ The partial update validation group is used for web service parameter validation.
- ❷ 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 to create HTTP responses based on the defined templates. Instead of being bound to a particular templating engine, Seam REST comes with a 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 is possible for a method to offer multiple representations of a resource, each rendered with a different template. In the example below, 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 (for example /freemarker/categories.ftl) |
| produces | false | */* | Restriction of media type produced by the template (useful in situations when a method produces multiple media types, with different templates) |
| responseName | false | response | Name under which the object returned by the JAX-RS method is available in the template (for example, 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 mostly 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 programmatically. 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 programmatically

```
@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. Built-in support for templating engines

Seam REST currently comes with built-in 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) [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) [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. Refer to [Javadoc](http://docs.jboss.org/seam/3/rest/latest/api/org/jboss/seam/rest/templating/TemplatingProvider.html) [http://docs.jboss.org/seam/3/rest/latest/api/org/jboss/seam/rest/templating/TemplatingProvider.html] for hints.

4.2.4. Selecting preferred 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 preferred one.

In such case, define the preferred templating engine in the XML configuration as demonstrated below to resolve the `TemplatingProvider` ambiguity.

Example 4.6. Preferred provider

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

  <templating:TemplatingMessageBodyWriter preferredTemplatingProvider="org.jboss.seam.rest.templating.freemarker.FreeMarkerProvider"
    <s:modifies />
  </templating:TemplatingMessageBodyWriter>
</beans>
```

Table 4.2. Built-in 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) [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 us assume as an example that a remote server exposes a web service for providing task details to the client through the `TaskService` interface below.

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, Seam REST builds and injects 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);
```

The Seam REST module injects a proxied `TaskService` interface and 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.

URI definition supports EL expressions.

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

5.2. Manual ClientRequest API

Besides proxying JAX-RS interfaces, the RESTEasy Client Framework provides the `ClientRequest` API for manual building of HTTP requests. 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) [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 the 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

6.1. Transitive Dependencies

The Seam REST module depends on the transitive dependencies at runtime listed in table [Table 6.1, “Transitive dependencies”](#).

Table 6.1. Transitive dependencies

| Name | Version |
|-------------|-------------|
| Seam Solder | 3.0.0.Beta2 |

6.2. Optional dependencies

6.2.1. Seam Catch

Seam Catch can be used for handling Java exceptions. For more information on using Seam Catch with Seam REST, refer to [Section 2.1, “Seam Catch Integration”](#)

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

6.2.2. Seam Config

Seam Config can be used to configure Seam REST using XML. For more information on using Seam Config with Seam REST, refer to [Section 2.2.2, “XML configuration”](#)

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

6.2.3. FreeMarker

FreeMarker can be used for rendering HTTP responses. For more information on using FreeMarker with Seam REST, refer to [Section 4.2.1, “FreeMarker”](#)

```
<dependency>
  <groupId>org.freemarker</groupId>
  <artifactId>freemarker</artifactId>
  <version>${freemarker.version}</version>
</dependency>
```

6.2.4. Apache Velocity

Apache Velocity can be used for rendering HTTP responses. For more information on using Velocity with Seam REST, refer to [Section 4.2.2, “Apache Velocity”](#)

```
<dependency>
  <groupId>org.apache.velocity</groupId>
  <artifactId>velocity</artifactId>
  <version>${velocity.version}</version>
</dependency>
<dependency>
  <groupId>org.apache.velocity</groupId>
  <artifactId>velocity-tools</artifactId>
  <version>${velocity.tools.version}</version>
</dependency>
```

6.2.5. RESTEasy

RESTEasy Client Framework can be used for building clients of RESTful web services. For more information on using RESTEasy Client Framework, refer to [Chapter 5, RESTEasy Client Framework Integration](#)

```
<dependency>
  <groupId>org.jboss.resteasy</groupId>
  <artifactId>resteasy-jaxrs</artifactId>
  <version>${resteasy.version}</version>
</dependency>
```

**Note**

Note that RESTEasy is provided on JBoss Application Server 6 and thus you do not need to bundle it with the web application.

