RESTEasy

Jakarta RESTFul Web Services

RESTEasy: Jakarta RESTFul Web Services

5.0.10.Final

Preface

Commercial development support, production support and training for RESTEasy is available from Red Hat Support. [https://access.redhat.com/support]

In some of the example listings, what is meant to be displayed on one line does not fit inside the available page width. These lines have been broken up. A '\' at the end of a line means that a break has been introduced to fit in the page, with the following lines indented. So:

```
Let's pretend to have an extremely \
long line that \
does not fit
This one is short
```

Is really:

```
Let's pretend to have an extremely long line that does not fit
This one is short
```

Chapter 1. Overview

Originally released in 2009, RESTEasy has tracked and implemented a series of official specifications that provide a Java API for RESTful Web Services over the HTTP protocol:

Table 1.1.

RESTEasy version	Specification
2+	JAX-RS 1.0 [https://download.oracle.com/otndocs/jcp/jaxrs-1.0-fr-eval-oth-JSpec/]
3+	JAX-RS 2.0 [https://jcp.org/en/jsr/detail?id=339]
3.5+	Jakarta RESTful Web Services 2.1 [https://jakarta.ee/specifications/restful-ws/2.1/restful-ws-spec-2.1.html]

JAX-RS 1.0 (JSR-311 [https://download.oracle.com/otndocs/jcp/jaxrs-1.0-fr-eval-oth-JSpec/]), JAX-RS 2.0 (JSR-339 [https://jcp.org/en/jsr/detail?id=339]), and JAX-RS 2.1 (JSR-370 [https://jcp.org/en/jsr/detail?id=370]) are Java Community Process (JCP) [https://jcp.org/en/procedures/overview] specifications. In 2017, the Java Enterprise Edition (Java EE) specifications, including JAX-RS, were transferred to the Eclipse Foundation (Java EE Moves to the Eclipse Foundation [https://blogs.eclipse.org/post/mike-milinkovich/java-ee-moves-eclipse-foundation]), Java EE became Jakarta Enterprise Edition, and new governing committees under the umbrella of the Eclipse Jakarta EE Platform [https://projects.eclipse.org/proposals/eclipse-jakarta-ee-platform] were constituted. For JAX-RS, the specifications are provided by the Jakarta RESTful Web Services [https://jakarta.ee/specifications/restful-ws/] committee. The first specification released under the authority of Jakarta RESTful Web Services was Jakarta RESTful Web Services 2.1 [https://jakarta.ee/specifications/restful-ws/2.1/], that being essentially identical to JAX-RS 2.1.

RESTEasy is a portable implementation of these specifications which can run in any Servlet container. Tighter integration with the WildFly application server is also available to make the user experience nicer in that environment. RESTEasy also comes with additional features on top of the plain old specifications.

Note

References in this document to JAX-RS refer to the Jakarta RESTful Web Services unless otherwise noted.

Note

References in this document to JAXB refer to the Jakarta XML Binding unless otherwise noted.

Chapter 2. License

RESTEasy is distributed under the Apache License 2.0. Some dependencies are covered by other open source licenses.

Chapter 3. Installation/ Configuration

RESTEasy is installed and configured in different ways depending on which environment you are running in. If you are running in WildFly, RESTEasy is already bundled and integrated completely so there is very little you have to do. If you are running in a different environment, there is some manual installation and configuration you will have to do.

3.1. RESTEasy modules in WildFly

In WildFly, RESTEasy and the Jakarta RESTful Web Services API are automatically loaded into your deployment's classpath if and only if you are deploying a Jakarta RESTful Web Services application (as determined by the presence of Jakarta RESTful Web Services annotations). However, only some RESTEasy features are automatically loaded. See Table 3.1. If you need any of those libraries which are not loaded automatically, you'll have to bring them in with a jboss-deployment-structure.xml file in the WEB-INF directory of your WAR file. Here's an example:

```
<jboss-deployment-structure>
    <deployment>
        <dependencies>
            <module name="org.jboss.resteasy.resteasy-jackson-provider" services="import"/>
        </dependencies>
    </deployment>
</jboss-deployment-structure>
```

The `services` attribute must be set to "import" for modules that have default providers in a META-INF/services/javax.ws.rs.ext.Providers file.

To get an idea of which RESTEasy modules are loaded by default when Jakarta RESTful Web Services services are deployed, please see the table below, which refers to a recent WildFly ditribution patched with the current RESTEasy distribution. Clearly, future and unpatched WildFly distributions might differ a bit in terms of modules enabled by default, as the container actually controls this too.

Table 3.1.

Module Name	Loaded by Default	Description
org.jboss.resteasy.resteasy-atom-provider	yes	RESTEasy's atom library
org.jboss.resteasy.resteasy-cdi	yes	RESTEasy CDI integration

Module Name	Loaded by Default	Description
org.jboss.resteasy.resteasy-crypto	yes	S/MIME, DKIM, and support for other security formats.
org.jboss.resteasy.resteasy-jackson-provider	no	Integration with the JSON parser and object mapper Jackson (deprecated)
org.jboss.resteasy.resteasy-jackson2-provider	yes	Integration with the JSON parser and object mapper Jackson 2
org.jboss.resteasy.resteasy-jaxb-provider	yes	Jakarta XML Binding integration.
org.jboss.resteasy.resteasy-core	yes	Core RESTEasy libraries for server.
org.jboss.resteasy.resteasy-client	yes	Core RESTEasy libraries for client.
org.jboss.resteasy.jose-jwt	no	JSON Web Token support.
org.jboss.resteasy.resteasy-jsapi	yes	RESTEasy's Javascript API
org.jboss.resteasy.resteasy-json-p-provider	yes	JSON parsing API
org.jboss.resteasy.resteasy-json-binding-provider	yes	JSON binding API
javax.json.bind-api	yes	JSON binding API
org.eclipse.yasson	yes	RI implementation of JSON binding API
org.jboss.resteasy.resteasy-multipart-provider	yes	Support for multipart formats
org.jboss.resteasy.resteasy-spring	no	Spring provider
org.jboss.resteasy.resteasy-validator-provider	yes	RESTEasy's interface to Hibernate Bean Validation

3.1.1. Other RESTEasy modules

Not all RESTEasy modules are bundled with WildFly. For example, `resteasy-fastinfoset-provider` and `resteasy-wadl` are not included among the modules listed in Section 3.1, “RESTEasy modules in WildFly”. If you want to use them in your application, you can include them in your WAR as you would if you were deploying outside of WildFly. See Section 3.3, “Deploying to other servlet containers” for more information.

3.1.2. Upgrading RESTEasy within WildFly

RESTEasy is bundled with WildFly. However you may wish to upgrade to the latest version. With Galleon [https://docs.wildfly.org/24/Galleon_Guide.html] this makes upgrading RESTEasy in WildFly quite easy.

The first requirement is the WildFly installation is provisioned with Galleon. The simplest way to do this for a local installation is with Galleon CLI [https://docs.wildfly.org/galleon/#_galleon_cli_tool].

```
$ galleon.sh install wildfly:current
```

To install the RESTEasy upgrade on top of that you simply need to use the tool again with the Maven GAV:

```
$ galleon.sh install org.jboss.resteasy:galleon-feature-pack:5.0.10.Final
```

If you are using Maven to provision WildFly you can simply add the feature pack to your plugin configuration.

```
<plugin>
    <groupId>org.jboss.galleon</groupId>
    <artifactId>galleon-maven-plugin</artifactId>
    <configuration>
        <install-dir>${jboss.home}</install-dir>
        <record-state>true</record-state>
        <log-time>true</log-time>
        <offline>false</offline>
        <feature-packs>
            <feature-pack>
                <groupId>org.jboss.resteasy</groupId>
                <artifactId>galleon-feature-pack</artifactId>
                <version>5.0.10.Final</version>
            </feature-pack>
        </feature-packs>
    </configuration>
    <executions>
        <execution>
            <id>server-provisioning</id>
            <phase>generate-test-resources</phase>
            <goals>
                <goal>provision</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

3.2. Deploying a RESTEasy application to WildFly

RESTEasy is bundled with WildFly and completely integrated as per the requirements of Jakarta EE. You can use it with Jakarta Enterprise Beans and CDI and you can rely completely on WildFly to scan for and deploy your Jakarta RESTful Web Services services and providers. All you have to provide is your Jakarta RESTful Web Services service and provider classes packaged within a WAR either as POJOs, CDI beans, or Jakarta Enterprise Beans. A simple way to configure an application is by simply providing an empty web.xml file. You can of course deploy any custom servlet, filter or security constraint you want to within your web.xml, but none of them are required:

```
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/
web-app_3_0.xsd">
</web-app>
```

Also, web.xml can supply to RESTEasy init-params and context-params (see Section 3.5, “Configuration switches”) if you want to tweak or turn on/off any specific RESTEasy feature.

Since we're not using a <servlet-mapping> element, we must define a `javax.ws.rs.core.Application` class (see Section 3.6, “javax.ws.rs.core.Application”) that is annotated with the `javax.ws.rs.ApplicationPath` annotation. If you return any empty set for classes and singletons, which is the behavior inherited from `Application`, your WAR will be scanned for resource and provider classes as indicated by the presence of Jakarta RESTful Web Services annotations.

```
import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;

@ApplicationPath("/root-path")
public class MyApplication extends Application
{
}
```

Note. Actually, if the application jar contains an `Application` class (or a subclass thereof) which is annotated with an `ApplicationPath` annotation, a web.xml file isn't even needed. Of course, even in this case it can be used to specify additional information such as context parameters. If there is an `Application` class but it doesn't have an `@ApplicationPath` annotation, then a web.xml file with at least a <servlet-mapping> element is required.

Note. As mentioned in Section 3.1.1, “Other RESTEasy modules”, not all RESTEasy modules are bundled with WildFly. For example, resteasy-fastinfoset-provider and resteasy-wadl are not included among the modules listed in Section 3.1, “RESTEasy modules in WildFly”. If you want to use them in your application, you can include them in your WAR as you would if you were

deploying outside of WildFly. See Section 3.3, “Deploying to other servlet containers” for more information.

3.3. Deploying to other servlet containers

If you are using RESTEasy outside of WildFly, in a standalone servlet container like Tomcat or Jetty, for example, you will need to include the appropriate RESTEasy jars in your WAR file. You will need the core classes in the resteasy-core and resteasy-client modules, and you may need additional facilities like the resteasy-jaxb-provider module. We strongly suggest that you use Maven to build your WAR files as RESTEasy is split into a bunch of different modules:

```
<dependency>
    <groupId>org.jboss.resteasy</groupId>
    <artifactId>resteasy-core</artifactId>
    <version>5.0.10.Final</version>
</dependency>
<dependency>
    <groupId>org.jboss.resteasy</groupId>
    <artifactId>resteasy-client</artifactId>
    <version>5.0.10.Final</version>
</dependency>
<dependency>
    <groupId>org.jboss.resteasy</groupId>
    <artifactId>resteasy-jaxb-provider</artifactId>
    <version>5.0.10.Final</version>
</dependency>
```

You can see sample Maven projects in https://github.com/resteasy/resteasy-examples.

If you are not using Maven, you can include the necessary jars by hand. If you download RESTEasy (from http://resteasy.jboss.org/downloads.html, for example) you will get a file like resteasy-jaxrs-<version>-all.zip. If you unzip it you will see a lib/ directory that contains the libraries needed by RESTEasy. Copy these, as needed, into your /WEB-INF/lib directory. Place your Jakarta RESTful Web Services annotated class resources and providers within one or more jars within /WEB-INF/lib or your raw class files within /WEB-INF/classes.

3.3.1. Servlet 3.0 containers

RESTEasy provides an implementation of the Servlet 3.0 `ServletContainerInitializer` integration interface for containers to use in initializing an application. The container calls this interface during the application's startup phase. The RESTEasy implementation performs automatic scanning for resources and providers, and programmatic registration of a servlet. RESTEasy's implementation is provided in maven artifact, `resteasy-servlet-initializer`. Add this artifact dependency to your project's pom.xml file so the JAR file will be included in your WAR file.

```
<dependency>
```

```
    <groupId>org.jboss.resteasy</groupId>
    <artifactId>resteasy-servlet-initializer</artifactId>
    <version>5.0.10.Final</version>
</dependency>
```

3.3.2. Older servlet containers

The `resteasy-servlet-initializer` artifact will not work in Servlet versions older than 3.0. You'll then have to manually declare the RESTEasy servlet in your WEB-INF/web.xml file of your WAR project, and you'll have to use an `Application` class (see Section 3.6, “javax.ws.rs.core.Application”) which explicitly lists resources and providers. For example:

```
<web-app>
    <display-name>Archetype Created Web Application</display-name>

    <servlet>
        <servlet-name>Resteasy</servlet-name>
        <servlet-class>
            org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher
        </servlet-class>
        <init-param>
            <param-name>javax.ws.rs.Application</param-name>
            <param-value>com.restfully.shop.services.ShoppingApplication</param-value>
        </init-param>
    </servlet>

    <servlet-mapping>
        <servlet-name>Resteasy</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>

</web-app>
```

The RESTEasy servlet is responsible for initializing some basic components of RESTEasy.

Note. It is likely that support for pre-3.0 Servlet specifications will be deprecated and eliminated eventually.

3.4. Configuration

RESTEasy has two mutually exclusive mechanisms for retrieving configuration parameters (see Section 3.5, “Configuration switches”). The classic mechanism depends on context-params and init-params in a web.xml file. Alternatively, the Eclipse MicroProfile Config project (https://github.com/eclipse/microprofile-config) provides a flexible parameter retrieval mechanism that RESTEasy will use if the necessary dependencies are available. See Section 3.4.4, “Configuring MicroProfile Config” for more about that. If they are not available, it will fall back to an extended form of the classic mechanism.

3.4.1. RESTEasy with MicroProfile Config

In the presence of the Eclipse MicroProfile Config API jar and an implementation of the API (see Section 3.4.4, “Configuring MicroProfile Config”), RESTEasy will use the facilities of MicroProfile Config for accessing configuration properties (see Section 3.5, “Configuration switches”). MicroProfile Config offers to both RESTEasy users and RESTEasy developers a great deal of flexibility in controlling runtime configuration.

In MicroProfile Config, a `ConfigSource` represents a `Map<String, String>` of property names to values, and a `Config` represents a sequence of `ConfigSource`s, ordered by priority. The priority of a `ConfigSource` is given by an ordinal (represented by an `int`), with a higher value indicating a higher priority. For a given property name, the `ConfigSource`s are searched in order until a value is found.

MicroProfile Config mandates the presence of the following `ConfigSource`s:

1. a `ConfigSource` based on System.getProperties() (ordinal = 400)
2. a `ConfigSource` based on System.getenv() (ordinal = 300)
3. a `ConfigSource` for each META-INF/microprofile-config.properties file on the ClassPath, separately configurable via a config_ordinal property inside each file (default ordinal = 100)

Note that a property which is found among the System properties and which is also in the System environment will be assigned the System property value because of the relative priorities of the `ConfigSource`s.

The set of `ConfigSources` is extensible. For example, smallrye-config (https://github.com/smallrye/smallrye-config), the implementation of the MicroProfile Config specification currently used by RESTEasy, adds the following kinds of `ConfigSource`s:

1. `PropertiesConfigSource` creates a `ConfigSource` from a Java `Properties` object or a Map<String, String> object or a properties file (referenced by its URL) (default ordinal = 100).
2. `DirConfigSource` creates a `ConfigSource` that will look into a directory where each file corresponds to a property (the file name is the property key and its textual content is the property value). This `ConfigSource` can be used to read configuration from Kubernetes ConfigMap (default ordinal = 100).
3. `ZkMicroProfileConfig` creates a `ConfigSource`ConfigSource that is backed by Apache Zookeeper (ordinal = 150).

These can be registered programmatically by using an instance of `ConfigProviderResolver`:

```
Config config = new PropertiesConfigSource("file:/// ...");
ConfigProviderResolver.instance().registerConfig(config, getClass().getClassLoader());
```

where `ConfigProviderResolver` is part of the Eclipse API.

If the application is running in Wildfly, then Wildfly provides another set of `ConfigSources`, as described in the "MicroProfile Config Subsystem Configuration" section of the WildFly Admin guide (https://docs.wildfly.org/21/Admin_Guide.html#MicroProfile_Config_SmallRye).

Finally, RESTEasy automatically provides three more `ConfigSources`:

- `org.jboss.resteasy.microprofile.config.ServletConfigSource` represents a servlet's <init-param>s from web.xml (ordinal = 60).
- `org.jboss.resteasy.microprofile.config.FilterConfigSource` represents a filter's <init-param>s from web.xml (ordinal = 50). (See Section 3.8, “RESTEasy as a Servlet Filter” for more information.)
- `org.jboss.resteasy.microprofile.config.ServletContextConfigSource` represents <context-param>s from web.xml (ordinal = 40).

Note. As stated by the MicroProfile Config specification, a special property `config_ordinal` can be set within any RESTEasy built-in `ConfigSources`. The default implementation of getOrdinal() will attempt to read this value. If found and a valid integer, the value will be used. Otherwise the respective default value will be used.

3.4.2. Using pure MicroProfile Config

The MicroProfile Config API is very simple. A `Config` may be obtained either programatically:

```
Config config = ConfigProvider.getConfig();
```

or, in the presence of CDI, by way of injection:

```
@Inject Config config;
```

Once a `Config` has been obtained, a property can be queried. For example,

```
String s = config.getValue("prop_name", String.class);
```

or

```
String s = config.getOptionalValue("prop_name", String.class).orElse("d'oh");
```

Now, consider a situation in which "prop_name" has been set by `System.setProperty("prop_name", "system")` and also by

```
   <context-param>
      <param-name>prop_name</param-name>
      <param-value>context</param-value>
   </context-param>
```

Then, since the system parameter `ConfigSource` has precedence over (has a higher ordinal than) `ServletContextConfigSource`, `config.getValue("prop_name", String.class)` will return "system" rather than "context".

3.4.3. Using RESTEasy's extension of MicroProfile Config

RESTEasy offers a general purpose parameter retrieval mechanism which incorporates MicroProfile Config if the necessary dependencies are available, and which falls back to an extended version of the classic RESTEasy mechanism (see Section 3.4.5, “RESTEasy's classic configuration mechanism”) otherwise.

Calling

```
ConfigurationFactory.getInstance().getConfiguration()
```

will return an instance of org.jboss.resteasy.spi.config.Configuration:

```
public interface Configuration {

    /**
     * Returns the resolved value for the specified type of the named property.
     *
     * @param name the name of the parameter
     * @param type the type to convert the value to
     * @param <T>  the property type
     *
     * @return the resolved optional value
     *
     * @throws IllegalArgumentException if the type is not supported
     */
```

```
    <T> Optional<T> getOptionalValue(String name, Class<T> type);

    /**
     * Returns the resolved value for the specified type of the named property.
     *
     * @param name the name of the parameter
     * @param type the type to convert the value to
     * @param <T>  the property type
     *
     * @return the resolved value
     *
     * @throws IllegalArgumentException         if the type is not supported
     * @throws java.util.NoSuchElementException if there is no property associated with the name
     */
    <T> T getValue(String name, Class<T> type);
}
```

For example,

```
String                                          value                                                   =

 String.class).orElse("d'oh");
```

If MicroProfile Config is available, that would be equivalent to

```
String         value         =         ConfigProvider.getConfig().getOptionalValue("prop_name",
 String.class).orElse("d'oh");
```

If MicroProfile Config is not available, then an attempt is made to retrieve the parameter from the following sources:

1. system variables, followed by
2. environment variables, followed by
3. web.xml parameters, as described in Section 3.4.5, “RESTEasy's classic configuration mechanism”

3.4.4. Configuring MicroProfile Config

If an application is running inside Wildfly, then all of the dependencies are automatically available. Outside of Wildfly, an application will need the Eclipse MicroProfile API at compile time. In maven, for example, use

As of RESTEasy 5.0 you will first need to add the RESTEasy MicroProfile Config dependency.

```
        <dependency>
          <groupId>org.jboss.resteasy.microprofile</groupId>
          <artifactId>microprofile-config</artifactId>
          <scope>compile</scope>
        </dependency>
```

You will also need the MicroProfile Config API and an Implementation, in our case SmallRye.

```
        <dependency>
            <groupId>org.eclipse.microprofile.config</groupId>
            <artifactId>microprofile-config-api</artifactId>
            <scope>compile</scope>
        </dependency>
```

```
        <dependency>
            <groupId>io.smallrye</groupId>
            <artifactId>smallrye-config</artifactId>
            <scope>runtime</scope>
        </dependency>
```

3.4.5. RESTEasy's classic configuration mechanism

Prior to the incorporation of MicroProfile Config, nearly all of RESTEasy's parameters were retrieved from servlet init-params and context-params. Which ones are available depends on how a web application invokes RESTEasy.

If RESTEasy is invoked as a servlet, as in

```
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/
web-app_3_0.xsd">

   <context-param>
      <param-name>system</param-name>
      <param-value>system-context</param-value>
   </context-param>

   <servlet>
      <servlet-name>Resteasy</servlet-name>
     <servlet-class>org.jboss.resteasy.plugins.server.servlet.HttpServlet30Dispatcher</servlet-
class>
```

```
      <init-param>
         <param-name>system</param-name>
         <param-value>system-init</param-value>
      </init-param>

   </servlet>

   <servlet-mapping>
      <servlet-name>Resteasy</servlet-name>
      <url-pattern>/*</url-pattern>
   </servlet-mapping>
</web-app>
```

then the servlet specific init-params and the general context-params are available, with the former taking precedence over the latter. For example, the property "system" would have the value "system-init".

If RESTEasy is invoked by way of a filter (see Section 3.8, “RESTEasy as a Servlet Filter”), as in

```
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/
web-app_3_0.xsd">

   <context-param>
      <param-name>system</param-name>
      <param-value>system-context</param-value>
   </context-param>

   <filter>
      <filter-name>Resteasy</filter-name>
      <filter-class>org.jboss.resteasy.plugins.server.servlet.FilterDispatcher</filter-class>

      <init-param>
         <param-name>system</param-name>
         <param-value>system-filter</param-value>
      </init-param>

    </filter>

    <filter-mapping>
        <filter-name>Resteasy</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

</web-app>
```

then the filter specific init-params and the general context-params are available, with the former taking precedence over the latter. For example, the property "system" would have the value "system-filter".

Finally, if RESTEasy is invoked by way of a ServletContextListener (see Section 3.7, “RESTEasy as a ServletContextListener”), as in

```
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/
web-app_3_0.xsd">

   <listener>
      <listener-class>
         org.jboss.resteasy.plugins.server.servlet.ResteasyBootstrap
      </listener-class>
   </listener>

   <context-param>
      <param-name>system</param-name>
      <param-value>system-context</param-value>
   </context-param>
</web-app>
```

where `ResteasyBootstrap` is a `ServletContextListener`, then the context-params are available.

3.4.6. Overriding RESTEasy's configuration mechanism

Before adopting the default behavior, with or without MicroProfile Config, as described in previous sections, RESTEasy will use service loading to look for one or more implementations of the interface `org.jboss.resteasy.spi.config.ConfigurationFactory`, selecting one with the highest priority as determined by the value returned by `ConfigurationFactory.priority()`. Smaller numbers indicate higher priority. The default `ConfigurationFactory` is `org.jboss.resteasy.core.config.DefaultConfigurationFactory` with a priority of 500.

3.5. Configuration switches

RESTEasy can receive the following configuration options from any `ConfigSource`s that are available at runtime:

Table 3.2.

Option Name	Default Value	Description
resteasy.servlet.mapping.prefix	no default	If the url-pattern for the RESTEasy servlet-mapping is not /*
resteasy.providers	no default	A comma delimited list of fully qualified @Provider class names you want to register

Option Name	Default Value	Description
resteasy.use.builtin.providers	true	Whether or not to register default, built-in @Provider classes
resteasy.resources	no default	A comma delimited list of fully qualified Jakarta RESTful Web Services resource class names you want to register
resteasy.jndi.resources	no default	A comma delimited list of JNDI names which reference objects you want to register as Jakarta RESTful Web Services resources
javax.ws.rs.Application	no default	Fully qualified name of Application class to bootstrap in a spec portable way
resteasy.media.type.mappings	no default	Replaces the need for an Accept header by mapping file name extensions (like .xml or .txt) to a media type. Used when the client is unable to use an Accept header to choose a representation (i.e. a browser). See Chapter 20, *Jakarta RESTful Web Services Content Negotiation* for more details.
resteasy.language.mappings	no default	Replaces the need for an Accept-Language header by mapping file name extensions (like .en or .fr) to a language. Used when the client is unable to use an Accept-Language header to choose a language (i.e. a browser). See Chapter 20, *Jakarta RESTful Web Services Content Negotiation* for more details.
resteasy.media.type.param.mapping	no default	Names a query parameter that can be set to an acceptable media type, enabling content negotiation without an Accept

Option Name	Default Value	Description
		header. See Chapter 20, *Jakarta RESTful Web Services Content Negotiation* for more details.
resteasy.role.based.security	false	Enables role based security. See Chapter 43, *Securing Jakarta RESTful Web Services and RESTEasy* for more details.
resteasy.document.expand.entity.references	false	Expand external entities in org.w3c.dom.Document documents and Jakarta XML Binding object representations
resteasy.document.secure.processing.feature	true	Impose security constraints in processing org.w3c.dom.Document documents and Jakarta XML Binding object representations
resteasy.document.secure.disallow.DTDs	true	Prohibit DTDs in org.w3c.dom.Document documents and Jakarta XML Binding object representations
resteasy.wider.request.matching	false	Turns off the Jakarta RESTful Web Services spec defined class-level expression filtering and instead tries to match version every method's full path.
resteasy.use.container.form.params	false	Obtain form parameters by using HttpServletRequest.getParameterMap(). Use this switch if you are calling this method within a servlet filter or eating the input stream within the filter.
resteasy.rfc7232preconditions	false	Enables RFC7232 compliant HTTP preconditions handling.
resteasy.gzip.max.input	10000000	Imposes maximum size on decompressed gzipped .

Option Name	Default Value	Description
resteasy.secure.random.max.use	100	The number of times a SecureRandom can be used before reseeding.
resteasy.buffer.exception.entity	true	Upon receiving an exception, the client side buffers any response entity before closing the connection.
resteasy.add.charset	true	If a resource method returns a text/* or application/xml* media type without an explicit charset, RESTEasy will add "charset=UTF-8" to the returned Content-Type header. Note that the charset defaults to UTF-8 in this case, independent of the setting of this parameter.
resteasy.disable.html.sanitizer	false	Normally, a response with media type "text/html" and a status of 400 will be processed so that the characters "/", "<", ">", "&", """ (double quote), and "'" (single quote) are escaped to prevent an XSS attack. If this parameter is set to "true", escaping will not occur.
resteasy.patchfilter.disabled	false	Turns off the default patch filter to handle JSON patch and JSON Merge Patch request. A customerized patch method filter can be provided to serve the JSON patch and JSON merge patch request instead.
resteasy.patchfilter.legacy	true	Set this value to false, the jsonp provider will be activated to provide PatchFilter for Json patch or Json Merge patch functionalities. By default(true value), the Jackson provider will be used.

Option Name	Default Value	Description
resteasy.original.webapplicationexception.behavior	false	When set to "true", this parameter will restore the original behavior in which a Client running in a resource method will throw a Jakarta RESTful Web Services WebApplicationException instead of a Resteasy version with a sanitized `Response`. For more information, see section Resteasy WebApplicationExceptions
dev.resteasy.throw.options.exception	false	Setting this value to true will throw a `org.jboss.resteasy.spi.DefaultOptionsMetho` if the HTTP method "OPTIONS" is sent and the matching method is not annotated with `@OPTIONS`. This is the original behavior of RESTEasy. However, this has been changed to return the response so that it's processed with an `ExceptionMapper`.

Note. The resteasy.servlet.mapping.prefix <context param> variable must be set if your servlet-mapping for the RESTEasy servlet has a url-pattern other than /*. For example, if the url-pattern is

```
<servlet-mapping>
<servlet-name>Resteasy</servlet-name>
<url-pattern>/restful-services/*</url-pattern>
</servlet-mapping>
```

Then the value of resteasy.servlet.mapping.prefix must be:

```
<context-param>
<param-name>resteasy.servlet.mapping.prefix</param-name>
<param-value>/restful-services</param-value>
</context-param>
```

Resteasy internally uses a cache to find the resource invoker for the request url. This cache size and enablement can be controlled with these system properties.

Table 3.3.

System Property Name	Default Value	Description
resteasy.match.cache.enabled	true	If the match cache is enabled or not
resteasy.match.cache.size	2048	The size of this match cache

3.6. javax.ws.rs.core.Application

The `javax.ws.rs.core.Application` class is a standard Jakarta RESTful Web Services class that you may implement to provide information on your deployment. It is simply a class the lists all Jakarta RESTful Web Services root resources and providers.

```
/**
* Defines the components of a Jakarta RESTful Web Services application and supplies additional
* metadata. A Jakarta RESTful Web Services application or implementation supplies a concrete
* subclass of this abstract class.
*/
public abstract class Application
{
    private static final Set<Object> emptySet = Collections.emptySet();

    /**
    * Get a set of root resource and provider classes. The default lifecycle
    * for resource class instances is per-request. The default lifecycle for
    * providers is singleton.
    * <p/>
    * <p>Implementations should warn about and ignore classes that do not
    * conform to the requirements of root resource or provider classes.
    * Implementations should warn about and ignore classes for which
    * {@link #getSingletons()} returns an instance. Implementations MUST
    * NOT modify the returned set.</p>
    *
    * @return a set of root resource and provider classes. Returning null
    * is equivalent to returning an empty set.
    */
    public abstract Set<Class<?>> getClasses();

    /**
    * Get a set of root resource and provider instances. Fields and properties
    * of returned instances are injected with their declared dependencies
    * (see {@link Context}) by the runtime prior to use.
    * <p/>
    * <p>Implementations should warn about and ignore classes that do not
    * conform to the requirements of root resource or provider classes.
    * Implementations should flag an error if the returned set includes
    * more than one instance of the same class. Implementations MUST
    * NOT modify the returned set.</p>
    * <p/>
    * <p>The default implementation returns an empty set.</p>
    *
    * @return a set of root resource and provider instances. Returning null
    * is equivalent to returning an empty set.
```

```
    */
    public Set<Object> getSingletons()
    {
        return emptySet;
    }

}
```

Note. If your web.xml file does not have a <servlet-mapping> element, you must use an `Application` class annotated with `@ApplicationPath`.

3.7. RESTEasy as a ServletContextListener

This section is pretty much deprecated if you are using a Servlet 3.0 container or higher. Skip it if you are and read the configuration section above on installing in Servlet 3.0. The initialization of RESTEasy can be performed within a ServletContextListener instead of within the Servlet. You may need this if you are writing custom Listeners that need to interact with RESTEasy at boot time. An example of this is the RESTEasy Spring integration that requires a Spring ServletContextListener. The org.jboss.resteasy.plugins.server.servlet.ResteasyBootstrap class is a ServletContextListener that configures an instance of an ResteasyProviderFactory and Registry. You can obtain instances of a ResteasyProviderFactory and Registry from the ServletContext attributes org.jboss.resteasy.spi.ResteasyProviderFactory and org.jboss.resteasy.spi.Registry. From these instances you can programmatically interact with RESTEasy registration interfaces.

```
<web-app>
   <listener>
      <listener-class>
         org.jboss.resteasy.plugins.server.servlet.ResteasyBootstrap
      </listener-class>
   </listener>

  <!-- ** INSERT YOUR LISTENERS HERE!!!! -->

   <servlet>
      <servlet-name>Resteasy</servlet-name>
      <servlet-class>
         org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher
      </servlet-class>
   </servlet>

   <servlet-mapping>
      <servlet-name>Resteasy</servlet-name>
      <url-pattern>/Resteasy/*</url-pattern>
   </servlet-mapping>

</web-app>
```

3.8. RESTEasy as a Servlet Filter

This section is pretty much deprecated if you are using a Servlet 3.0 container or higher. Skip it if you are and read the configuration section above on installing in Servlet 3.0. The downside of running RESTEasy as a Servlet is that you cannot have static resources like .html and .jpeg files in the same path as your Jakarta RESTful Web Services services. RESTEasy allows you to run as a Filter instead. If a Jakarta RESTful Web Services resource is not found under the URL requested, RESTEasy will delegate back to the base servlet container to resolve URLs.

```
<web-app>
    <filter>
        <filter-name>Resteasy</filter-name>
        <filter-class>
            org.jboss.resteasy.plugins.server.servlet.FilterDispatcher
        </filter-class>
        <init-param>
            <param-name>javax.ws.rs.Application</param-name>
            <param-value>com.restfully.shop.services.ShoppingApplication</param-value>
        </init-param>
    </filter>

    <filter-mapping>
        <filter-name>Resteasy</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

</web-app>
```

3.9. Client side

Jakarta RESTful Web Services conforming implementations, such as RESTEasy, support a client side framework which simplifies communicating with restful applications. In RESTEasy, the minimal set of modules needed for the client framework consists of resteasy-core and resteasy-client. You can access them by way of maven:

```
<dependency>
    <groupId>org.jboss.resteasy</groupId>
    <artifactId>resteasy-core</artifactId>
    <version>5.0.10.Final</version>
</dependency>
<dependency>
    <groupId>org.jboss.resteasy</groupId>
    <artifactId>resteasy-client</artifactId>
    <version>5.0.10.Final</version>
</dependency>
```

Other modules, such as resteasy-jaxb-provider, may be brought in as needed.

Chapter 4. Using @Path and @GET, @POST, etc.

```
@Path("/library")
public class Library {

   @GET
   @Path("/books")
   public String getBooks() {...}

   @GET
   @Path("/book/{isbn}")
   public String getBook(@PathParam("isbn") String id) {
      // search my database and get a string representation and return it
   }

   @PUT
   @Path("/book/{isbn}")
   public void addBook(@PathParam("isbn") String id, @QueryParam("name") String name) {...}

   @DELETE
   @Path("/book/{id}")
   public void removeBook(@PathParam("id") String id {...}

}
```

Let's say you have the RESTEasy servlet configured and reachable at a root path of http://myhost.com/services. The requests would be handled by the Library class:

- GET http://myhost.com/services/library/books
- GET http://myhost.com/services/library/book/333
- PUT http://myhost.com/services/library/book/333
- DELETE http://myhost.com/services/library/book/333

The @javax.ws.rs.Path annotation must exist on either the class and/or a resource method. If it exists on both the class and method, the relative path to the resource method is a concatenation of the class and method.

In the @javax.ws.rs package there are annotations for each HTTP method. @GET, @POST, @PUT, @DELETE, and @HEAD. You place these on public methods that you want to map to that certain kind of HTTP method. As long as there is a @Path annotation on the class, you do not have to have a @Path annotation on the method you are mapping. You can have more than one HTTP method as long as they can be distinguished from other methods.

When you have a @Path annotation on a method without an HTTP method, these are called JAXRSResourceLocators.

4.1. @Path and regular expression mappings

The @Path annotation is not limited to simple path expressions. You also have the ability to insert regular expressions into @Path's value. For example:

```
@Path("/resources)
public class MyResource {

   @GET
   @Path("{var:.*}/stuff")
   public String get() {...}
}
```

The following GETs will route to the getResource() method:

```
GET /resources/stuff
GET /resources/foo/stuff
GET /resources/on/and/on/stuff
```

The format of the expression is:

```
"{" variable-name [ ":" regular-expression ] "}"
```

The regular-expression part is optional. When the expression is not provided, it defaults to a wildcard matching of one particular segment. In regular-expression terms, the expression defaults to

```
"([]*)"
```

For example:

@Path("/resources/{var}/stuff")

will match these:

```
GET /resources/foo/stuff
GET /resources/bar/stuff
```

but will not match:

```
GET /resources/a/bunch/of/stuff
```

Chapter 5. @PathParam

Note

RESTEasy supports `@PathParam` annotations with no parameter name..

@PathParam is a parameter annotation which allows you to map variable URI path fragments into your method call.

```
@Path("/library")
public class Library {

   @GET
   @Path("/book/{isbn}")
   public String getBook(@PathParam("isbn") String id) {
      // search my database and get a string representation and return it
   }
}
```

What this allows you to do is embed variable identification within the URIs of your resources. In the above example, an isbn URI parameter is used to pass information about the book we want to access. The parameter type you inject into can be any primitive type, a String, or any Java object that has a constructor that takes a String parameter, or a static valueOf method that takes a String as a parameter. For example, lets say we wanted isbn to be a real object. We could do:

```
@GET
@Path("/book/{isbn}")
public String getBook(@PathParam("isbn") ISBN id) {...}

public class ISBN {
   public ISBN(String str) {...}
}
```

Or instead of a public String constructors, have a valueOf method:

```
public class ISBN {
    public static ISBN valueOf(String isbn) {...}
}
```

5.1. Advanced @PathParam and Regular Expressions

There are a few more complicated uses of @PathParams not discussed in the previous section.

You are allowed to specify one or more path params embedded in one URI segment. Here are some examples:

```
1. @Path("/aaa{param}bbb")
2. @Path("/{name}-{zip}")
3. @Path("/foo{name}-{zip}bar")
```

So, a URI of "/aaa111bbb" would match #1. "/bill-02115" would match #2. "foobill-02115bar" would match #3.

We discussed before how you can use regular expression patterns within @Path values.

```
@GET
@Path("/aaa{param:b+}/{many:.*}/stuff")
public String getIt(@PathParam("param") String bs, @PathParam("many") String many) {...}
```

For the following requests, lets see what the values of the "param" and "many" @PathParams would be:

Table 5.1.

Request	param	many
GET /aaabb/some/stuff	bb	some
GET /aaab/a/lot/of/stuff	b	a/lot/of

5.2. @PathParam and PathSegment

The specification has a very simple abstraction for examining a fragment of the URI path being invoked on javax.ws.rs.core.PathSegment:

```
public interface PathSegment {

    /**
     * Get the path segment.
     * <p>
     * @return the path segment
```

```
     */
    String getPath();

    /**
     * Get a map of the matrix parameters associated with the path segment
     * @return the map of matrix parameters
     */
    MultivaluedMap<String, String> getMatrixParameters();

}
```

You can have RESTEasy inject a PathSegment instead of a value with your @PathParam.

```
@GET
@Path("/book/{id}")
public String getBook(@PathParam("id") PathSegment id) {...}
```

This is very useful if you have a bunch of @PathParams that use matrix parameters. The idea of matrix parameters is that they are an arbitrary set of name-value pairs embedded in a uri path segment. The PathSegment object gives you access to these parameters. See also MatrixParam.

A matrix parameter example is:

GET http://host.com/library/book;name=EJB 3.0;author=Bill Burke

The basic idea of matrix parameters is that it represents resources that are addressable by their attributes as well as their raw id.

Chapter 6. @QueryParam

Note

RESTEasy supports `@QueryParam` annotations with no parameter name..

The @QueryParam annotation allows you to map a URI query string parameter or url form encoded parameter to your method invocation.

GET /books?num=5

```
@GET
public String getBooks(@QueryParam("num") int num) {
...
}
```

Currently since RESTEasy is built on top of a Servlet, it does not distinguish between URI query strings or url form encoded parameters. Like PathParam, your parameter type can be an String, primitive, or class that has a String constructor or static valueOf() method.

Chapter 7. @HeaderParam

Note

RESTEasy supports `@HeaderParam` annotations with no parameter name..

The @HeaderParam annotation allows you to map a request HTTP header to your method invocation.

GET /books?num=5

```
@GET
public String getBooks(@HeaderParam("From") String from) {
...
}
```

Like PathParam, your parameter type can be an String, primitive, or class that has a String constructor or static valueOf() method. For example, MediaType has a valueOf() method and you could do:

```
@PUT
public void put(@HeaderParam("Content-Type") MediaType contentType, ...)
```

7.1. `HeaderDelegate`S

In addition to the usual methods for translating parameters to and from strings, parameters annotated with `@HeaderParam` have another option: implementations of `RuntimeDelegate$HeaderDelegate`:

```
   /**
    * Defines the contract for a delegate that is responsible for
    * converting between the String form of a HTTP header and
    * the corresponding Jakarta RESTful Web Services type {@code T}.
    *
      * @param <T> a Jakarta RESTful Web Services type that corresponds to the value of a
HTTP header.
    */
   public static interface HeaderDelegate<T> {

       /**
```

```
         * Parse the supplied value and create an instance of {@code T}.
         *
         * @param value the string value.
         * @return the newly created instance of {@code T}.
         * @throws IllegalArgumentException if the supplied string cannot be
         *                                  parsed or is {@code null}.
         */
        public T fromString(String value);

        /**
         * Convert the supplied value to a String.
         *
         * @param value the value of type {@code T}.
         * @return a String representation of the value.
         * @throws IllegalArgumentException if the supplied object cannot be
         *                                  serialized or is {@code null}.
         */
        public String toString(T value);
    }
```

`HeaderDelegate` is similar to `ParamConverter`, but it is not very convenient to register a `HeaderDelegate` since, unlike, for example, `ParamConverterProvider`, it is not treated by the Jakarta RESTful Web Services specification as a provider. The class `javax.ws.rs.core.Configurable`, which is subclassed by, for example, `org.jboss.resteasy.spi.ResteasyProviderFactory` has methods like

```
    /**
     * Register a class of a custom Jakarta RESTful Web Services component (such as an extension
provider or
     * a {@link javax.ws.rs.core.Feature feature} meta-provider) to be instantiated
     * and used in the scope of this configurable context.
     *
     * ...
     *
      * @param componentClass Jakarta RESTful Web Services component class to be configured
in the scope of this
     *                       configurable context.
     * @return the updated configurable context.
     */
    public C register(Class<?> componentClass);
```

but it is not clear that they are applicable to `HeaderDelegate`s.

RESTEasy approaches this problem by allowing `HeaderDelegate`s to be annotated with `@Provider`. Not only will `ResteasyProviderFactory.register()` process `HeaderDelegate`s, but another useful consequence is that `HeaderDelegate`s can be discovered automatically at runtime.

Chapter 8. Linking resources

There are two mechanisms available in RESTEasy to link a resource to another, and to link resources to operations: the Link HTTP header, and Atom links inside the resource representations.

8.1. Link Headers

RESTEasy has both client and server side support for the Link header specification [http://tools.ietf.org/html/draft-nottingham-http-link-header-06]. See the javadocs for org.jboss.resteasy.spi.LinkHeader, org.jboss.resteasy.spi.Link, and org.jboss.resteasy.client.ClientResponse.

The main advantage of Link headers over Atom links in the resource is that those links are available without parsing the entity body.

8.2. Atom links in the resource representations

RESTEasy allows you to inject Atom links [http://tools.ietf.org/html/rfc4287#section-4.2.7] directly inside the entity objects you are sending to the client, via auto-discovery.

Warning

This is only available when using the Jackson2 or Jakarta XML Binding providers (for JSON and XML).

The main advantage over Link headers is that you can have any number of Atom links directly over the concerned resources, for any number of resources in the response. For example, you can have Atom links for the root response entity, and also for each of its children entities.

8.2.1. Configuration

There is no configuration required to be able to inject Atom links in your resource representation, you just have to have this maven artifact in your path:

Table 8.1. Maven artifact for Atom link injection

Group	Artifact	Version
org.jboss.resteasy	resteasy-links	5.0.10.Final

8.2.2. Your first links injected

You need three things in order to tell RESTEasy to inject Atom links in your entities:

- Annotate the Jakarta RESTful Web Services method with `@AddLinks` to indicate that you want Atom links injected in your response entity.
- Add `RESTServiceDiscovery` fields to the resource classes where you want Atom links injected.
- Annotate the Jakarta RESTful Web Services methods you want Atom links for with `@LinkResource`, so that RESTEasy knows which links to create for which resources.

The following example illustrates how you would declare everything in order to get the Atom links injected in your book store:

```
@Path("/")
@Consumes({"application/xml", "application/json"})
@Produces({"application/xml", "application/json"})
public interface BookStore {

    @AddLinks
    @LinkResource(value = Book.class)
    @GET
    @Path("books")
    public Collection<Book> getBooks();

    @LinkResource
    @POST
    @Path("books")
    public void addBook(Book book);

    @AddLinks
    @LinkResource
    @GET
    @Path("book/{id}")
    public Book getBook(@PathParam("id") String id);

    @LinkResource
    @PUT
    @Path("book/{id}")
    public void updateBook(@PathParam("id") String id, Book book);

    @LinkResource(value = Book.class)
    @DELETE
    @Path("book/{id}")
    public void deleteBook(@PathParam("id") String id);
}
```

And this is the definition of the Book resource:

```
@Mapped(namespaceMap = @XmlNsMap(jsonName = "atom", namespace = "http://www.w3.org/2005/Atom"))
@XmlRootElement
@XmlAccessorType(XmlAccessType.NONE)
public class Book {
    @XmlAttribute
    private String author;

    @XmlID
```

```
    @XmlAttribute
    private String title;

    @XmlElementRef
    private RESTServiceDiscovery rest;
}
```

If you do a GET /order/foo you will then get this XML representation:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<book xmlns:atom="http://www.w3.org/2005/Atom" title="foo" author="bar">
 <atom:link href="http://localhost:8081/books" rel="list"/>
 <atom:link href="http://localhost:8081/books" rel="add"/>
 <atom:link href="http://localhost:8081/book/foo" rel="self"/>
 <atom:link href="http://localhost:8081/book/foo" rel="update"/>
 <atom:link href="http://localhost:8081/book/foo" rel="remove"/>
</book>
```

And in JSON format:

```
{
 "book":
 {
  "@title":"foo",
  "@author":"bar",
  "atom.link":
   [
    {"@href":"http://localhost:8081/books","@rel":"list"},
    {"@href":"http://localhost:8081/books","@rel":"add"},
    {"@href":"http://localhost:8081/book/foo","@rel":"self"},
    {"@href":"http://localhost:8081/book/foo","@rel":"update"},
    {"@href":"http://localhost:8081/book/foo","@rel":"remove"}
   ]
 }
}
```

8.2.3. Customising how the Atom links are serialised

Because the `RESTServiceDiscovery` is in fact a Jakarta XML Binding type which inherits from `List` you are free to annotate it as you want to customise the Jakarta XML Binding serialisation, or just rely on the default with `@XmlElementRef`.

8.2.4. Specifying which Jakarta RESTful Web Services methods are tied to which resources

This is all done by annotating the methods with the `@LinkResource` annotation. It supports the following optional parameters:

Table 8.2.

`@LinkResource` parameters

Parameter	Type	Function	Default
value	`Class`	Declares an Atom link for the given type of resources.	Defaults to the entity body type (non-annotated parameter), or the method's return type. This default does not work with `Response` or `Collection` types, they need to be explicitly specified.
rel	`String`	The Atom link relation	list For `GET` methods returning a `Collection` self For `GET` methods returning a non-`Collection` remove For `DELETE` methods update For `PUT` methods add For `POST` methods

You can add several `@LinkResource` annotations on a single method by enclosing them in a `@LinkResources` annotation. This way you can add links to the same method on several resource types. For example the `/order/foo/comments` operation can belongs on the `Order` resource with the `comments` relation, and on the `Comment` resource with the `list` relation.

8.2.5. Specifying path parameter values for URI templates

When RESTEasy adds links to your resources it needs to insert the right values in the URI template. This is done either automatically by guessing the list of values from the entity, or by specifying the values in the `@LinkResource` `pathParameters` parameter.

8.2.5.1. Loading URI template values from the entity

URI template values are extracted from the entity from fields or Java Bean properties annotated with `@ResourceID`, Jakarta XML Binding's `@XmlID` or Jakarta Persistence's `@Id`. If there are more than one URI template value to find in a given entity, you can annotate your entity with `@ResourceIDs` to list the names of fields or properties that make up this entity's Id. If there are other URI template values required from a parent entity, we try to find that parent in a field or Java Bean property annotated with `@ParentResource`. The list of URI template values extracted up every `@ParentResource` is then reversed and used as the list of values for the URI template.

For example, let's consider the previous Book example, and a list of comments:

```
@XmlRootElement
@XmlAccessorType(XmlAccessType.NONE)
public class Comment {
    @ParentResource
    private Book book;

    @XmlElement
    private String author;

    @XmlID
    @XmlAttribute
    private String id;

    @XmlElementRef
    private RESTServiceDiscovery rest;
}
```

Given the previous book store service augmented with comments:

```
@Path("/")
@Consumes({"application/xml", "application/json"})
@Produces({"application/xml", "application/json"})
public interface BookStore {

    @AddLinks
    @LinkResources({
        @LinkResource(value = Book.class, rel = "comments"),
        @LinkResource(value = Comment.class)
    })
    @GET
    @Path("book/{id}/comments")
    public Collection<Comment> getComments(@PathParam("id") String bookId);

    @AddLinks
    @LinkResource
    @GET
    @Path("book/{id}/comment/{cid}")
   public Comment getComment(@PathParam("id") String bookId, @PathParam("cid") String commentId);

    @LinkResource
```

```
    @POST
    @Path("book/{id}/comments")
    public void addComment(@PathParam("id") String bookId, Comment comment);

    @LinkResource
    @PUT
    @Path("book/{id}/comment/{cid}")
    public void updateComment(@PathParam("id") String bookId, @PathParam("cid") String commentId, Comment comment

    @LinkResource(Comment.class)
    @DELETE
    @Path("book/{id}/comment/{cid}")
   public void deleteComment(@PathParam("id") String bookId, @PathParam("cid") String commentId);

}
```

Whenever we need to make links for a `Book` entity, we look up the ID in the `Book`'s `@XmlID` property. Whenever we make links for `Comment` entities, we have a list of values taken from the `Comment`'s `@XmlID` and its `@ParentResource`: the `Book` and its `@XmlID`.

For a `Comment` with `id` `"1"` on a `Book` with `title` `"foo"` we will therefore get a list of URI template values of `{"foo", "1"}`, to be replaced in the URI template, thus obtaining either `"/book/foo/comments"` or `"/book/foo/comment/1"`.

8.2.5.2. Specifying path parameters manually

If you do not want to annotate your entities with resource ID annotations (`@ResourceID`, `@ResourceIDs`, `@XmlID` or `@Id`) and `@ParentResource`, you can also specify the URI template values inside the `@LinkResource` annotation, using Unified Expression Language expressions:

Table 8.3.

`@LinkResource` URI template parameter

Parameter	Type	Function	Default
pathParameters	`String[]`	Declares a list of UEL expressions to obtain the URI template values.	Defaults to using `@ResourceID`, `@ResourceIDs`, `@XmlID` or `@Id` and `@ParentResource` annotations to extract the values from the model.

The UEL expressions are evaluated in the context of the entity, which means that any unqualified variable will be taken as a property for the entity itself, with the special variable `this` bound to the entity we're generating links for.

The previous example of `Comment` service could be declared as such:

```
@Path("/")
@Consumes({"application/xml", "application/json"})
@Produces({"application/xml", "application/json"})
public interface BookStore {

    @AddLinks
    @LinkResources({
        @LinkResource(value = Book.class, rel = "comments", pathParameters = "${title}"),
        @LinkResource(value = Comment.class, pathParameters = {"${book.title}", "${id}"})
    })
    @GET
    @Path("book/{id}/comments")
    public Collection<Comment> getComments(@PathParam("id") String bookId);

    @AddLinks
    @LinkResource(pathParameters = {"${book.title}", "${id}"})
    @GET
    @Path("book/{id}/comment/{cid}")
   public Comment getComment(@PathParam("id") String bookId, @PathParam("cid") String commentId);

    @LinkResource(pathParameters = {"${book.title}", "${id}"})
    @POST
    @Path("book/{id}/comments")
    public void addComment(@PathParam("id") String bookId, Comment comment);

    @LinkResource(pathParameters = {"${book.title}", "${id}"})
    @PUT
    @Path("book/{id}/comment/{cid}")
    public void updateComment(@PathParam("id") String bookId, @PathParam("cid") String commentId, Comment comment

    @LinkResource(Comment.class, pathParameters = {"${book.title}", "${id}"})
    @DELETE
    @Path("book/{id}/comment/{cid}")
   public void deleteComment(@PathParam("id") String bookId, @PathParam("cid") String commentId);

}
```

8.2.6. Securing entities

You can restrict which links are injected in the resource based on security restrictions for the client, so that if the current client doesn't have permission to delete a resource he will not be presented with the `"delete"` link relation.

Security restrictions can either be specified on the `@LinkResource` annotation, or using RESTEasy and Jakarta Enterprise Beans security annotation `@RolesAllowed` on the Jakarta RESTful Web Services method.

Table 8.4.

`@LinkResource` security restrictions

Parameter	Type	Function	Default
constraint	`String`	A UEL expression which must evaluate to true to inject this method's link in the response entity.	Defaults to using `@RolesAllowed` from the Jakarta RESTful Web Services method.

8.2.7. Extending the UEL context

We've seen that both the URI template values and the security constraints of `@LinkResource` use UEL to evaluate expressions, and we provide a basic UEL context with access only to the entity we're injecting links in, and nothing more.

If you want to add more variables or functions in this context, you can by adding a `@LinkELProvider` annotation on the Jakarta RESTful Web Services method, its class, or its package. This annotation's value should point to a class that implements the `ELProvider` interface, which wraps the default `ELContext` in order to add any missing functions.

For example, if you want to support the Seam annotation `s:hasPermission(target, permission)` in your security constraints, you can add a `package-info.java` file like this:

```
@LinkELProvider(SeamELProvider.class)
package org.jboss.resteasy.links.test;

import org.jboss.resteasy.links.*;
```

With the following provider implementation:

```
package org.jboss.resteasy.links.test;

import javax.el.ELContext;
import javax.el.ELResolver;
import javax.el.FunctionMapper;
import javax.el.VariableMapper;

import org.jboss.seam.el.SeamFunctionMapper;

import org.jboss.resteasy.links.ELProvider;

public class SeamELProvider implements ELProvider {

    public ELContext getContext(final ELContext ctx) {
        return new ELContext() {

            private SeamFunctionMapper functionMapper;

            @Override
            public ELResolver getELResolver() {
```

```
                return ctx.getELResolver();
            }

            @Override
            public FunctionMapper getFunctionMapper() {
                if (functionMapper == null)
                    functionMapper = new SeamFunctionMapper(ctx
                            .getFunctionMapper());
                return functionMapper;
            }

            @Override
            public VariableMapper getVariableMapper() {
                return ctx.getVariableMapper();
            }
        };
    }

}
```

And then use it as such:

```
@Path("/")
@Consumes({"application/xml", "application/json"})
@Produces({"application/xml", "application/json"})
public interface BookStore {

    @AddLinks
    @LinkResources({
       @LinkResource(value = Book.class, rel = "comments", constraint = "${s:hasPermission(this,
 'add-comment')}"),
        @LinkResource(value = Comment.class, constraint = "${s:hasPermission(this, 'insert')}")
    })
    @GET
    @Path("book/{id}/comments")
    public Collection<Comment> getComments(@PathParam("id") String bookId);

    @AddLinks
    @LinkResource(constraint = "${s:hasPermission(this, 'read')}")
    @GET
    @Path("book/{id}/comment/{cid}")
  public Comment getComment(@PathParam("id") String bookId, @PathParam("cid") String commentId);

    @LinkResource(constraint = "${s:hasPermission(this, 'insert')}")
    @POST
    @Path("book/{id}/comments")
    public void addComment(@PathParam("id") String bookId, Comment comment);

    @LinkResource(constraint = "${s:hasPermission(this, 'update')}")
    @PUT
    @Path("book/{id}/comment/{cid}")
    public void updateComment(@PathParam("id") String bookId, @PathParam("cid") String commentId, Comment comment

    @LinkResource(Comment.class, constraint = "${s:hasPermission(this, 'delete')}")
    @DELETE
    @Path("book/{id}/comment/{cid}")
```

```
    public void deleteComment(@PathParam("id") String bookId, @PathParam("cid") String commentId);

}
```

8.2.8. Resource facades

Sometimes it is useful to add resources which are just containers or layers on other resources. For example if you want to represent a collection of `Comment` with a start index and a certain number of entries, in order to implement paging. Such a collection is not really an entity in your model, but it should obtain the `"add"` and `"list"` link relations for the `Comment` entity.

This is possible using resource facades. A resource facade is a resource which implements the `ResourceFacade<T>` interface for the type `T`, and as such, should receive all links for that type.

Since in most cases the instance of the `T` type is not directly available in the resource facade, we need another way to extract its URI template values, and this is done by calling the resource facade's `pathParameters()` method to obtain a map of URI template values by name. This map will be used to fill in the URI template values for any link generated for `T`, if there are enough values in the map.

Here is an example of such a resource facade for a collection of `Comments`:

```
@XmlRootElement
@XmlAccessorType(XmlAccessType.NONE)
public class ScrollableCollection implements ResourceFacade<Comment> {

    private String bookId;
    @XmlAttribute
    private int start;
    @XmlAttribute
    private int totalRecords;
    @XmlElement
    private List<Comment> comments = new ArrayList<Comment>();
    @XmlElementRef
    private RESTServiceDiscovery rest;

    public Class<Comment> facadeFor() {
        return Comment.class;
    }

    public Map<String, ? extends Object> pathParameters() {
        HashMap<String, String> map = new HashMap<String, String>();
        map.put("id", bookId);
        return map;
    }
}
```

This will produce such an XML collection:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
```

```
<collection xmlns:atom="http://www.w3.org/2005/Atom" totalRecords="2" start="0">
 <atom.link href="http://localhost:8081/book/foo/comments" rel="add"/>
 <atom.link href="http://localhost:8081/book/foo/comments" rel="list"/>
 <comment xmlid="0">
  <text>great book</text>
  <atom.link href="http://localhost:8081/book/foo/comment/0" rel="self"/>
  <atom.link href="http://localhost:8081/book/foo/comment/0" rel="update"/>
  <atom.link href="http://localhost:8081/book/foo/comment/0" rel="remove"/>
  <atom.link href="http://localhost:8081/book/foo/comments" rel="add"/>
  <atom.link href="http://localhost:8081/book/foo/comments" rel="list"/>
 </comment>
 <comment xmlid="1">
  <text>terrible book</text>
  <atom.link href="http://localhost:8081/book/foo/comment/1" rel="self"/>
  <atom.link href="http://localhost:8081/book/foo/comment/1" rel="update"/>
  <atom.link href="http://localhost:8081/book/foo/comment/1" rel="remove"/>
  <atom.link href="http://localhost:8081/book/foo/comments" rel="add"/>
  <atom.link href="http://localhost:8081/book/foo/comments" rel="list"/>
 </comment>
</collection>
```

Chapter 9. @MatrixParam

Note

RESTEasy supports `@MatrixParam` annotations with no parameter name..

The idea of matrix parameters is that they are an arbitrary set of name-value pairs embedded in a uri path segment. A matrix parameter example is:

GET http://host.com/library/book;name=EJB 3.0;author=Bill Burke

The basic idea of matrix parameters is that it represents resources that are addressable by their attributes as well as their raw id. The @MatrixParam annotation allows you to inject URI matrix parameters into your method invocation

```
@GET
public String getBook(@MatrixParam("name") String name, @MatrixParam("author") String author)
 {...}
```

There is one big problem with @MatrixParam that the current version of the specification does not resolve. What if the same MatrixParam exists twice in different path segments? In this case, right now, its probably better to use PathParam combined with PathSegment.

Chapter 10. @CookieParam

Note

RESTEasy supports `@CookieParam` annotations with no parameter name..

The @CookieParam annotation allows you to inject the value of a cookie or an object representation of an HTTP request cookie into your method invocation

GET /books?num=5

```
@GET
public String getBooks(@CookieParam("sessionid") int id) {
...
}

@GET
public String getBooks(@CookieParam("sessionid") javax.ws.rs.core.Cookie id) {...}
```

Like PathParam, your parameter type can be an String, primitive, or class that has a String constructor or static valueOf() method. You can also get an object representation of the cookie via the javax.ws.rs.core.Cookie class.

Chapter 11. @FormParam

Note

RESTEasy supports `@FormParam` annotations with no parameter name..

When the input request body is of the type "application/x-www-form-urlencoded", a.k.a. an HTML Form, you can inject individual form parameters from the request body into method parameter values.

```
<form method="POST" action="/resources/service">
First name:
<input type="text" name="firstname">
<br>
Last name:
<input type="text" name="lastname">
</form>
```

If you post through that form, this is what the service might look like:

```
@Path("/")
public class NameRegistry {

   @Path("/resources/service")
   @POST
  public void addName(@FormParam("firstname") String first, @FormParam("lastname") String last)
 {...}
```

You cannot combine @FormParam with the default "application/x-www-form-urlencoded" that unmarshalls to a MultivaluedMap<String, String>. i.e. This is illegal:

```
@Path("/")
public class NameRegistry {

   @Path("/resources/service")
   @POST
   @Consumes("application/x-www-form-urlencoded")
    public void addName(@FormParam("firstname") String first, MultivaluedMap<String, String>
 form) {...}
```

Chapter 12. @Form

This is a RESTEasy specific annotation that allows you to re-use any @*Param annotation within an injected class. RESTEasy will instantiate the class and inject values into any annotated @*Param or @Context property. This is useful if you have a lot of parameters on your method and you want to condense them into a value object.

```
public class MyForm {

    @FormParam("stuff")
    private int stuff;

    @HeaderParam("myHeader")
    private String header;

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


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

When somebody posts to /myservice, RESTEasy will instantiate an instance of MyForm and inject the form parameter "stuff" into the "stuff" field, the header "myheader" into the header field, and call the setFoo method with the path param variable of "foo".

Also, @Form has some expanded @FormParam features. If you specify a prefix within the Form param, this will prepend a prefix to any form parameter lookup. For example, let's say you have one Address class, but want to reference invoice and shipping addresses from the same set of form parameters:

```
public static class Person
{
    @FormParam("name")
    private String name;

    @Form(prefix = "invoice")
    private Address invoice;

    @Form(prefix = "shipping")
    private Address shipping;
}

public static class Address
{
```

```
    @FormParam("street")
    private String street;
}

@Path("person")
public static class MyResource
{
    @POST
    @Produces(MediaType.TEXT_PLAIN)
    @Consumes(MediaType.APPLICATION_FORM_URLENCODED)
    public String post(@Form Person p)
    {
        return p.toString();
    }
}
```

In this example, the client could send the following form parameters:

```
name=bill
invoice.street=xxx
shipping.street=yyy
```

The Person.invoice and Person.shipping fields would be populated appropriately. Also, prefix mappings also support lists and maps:

```
public static class Person {
    @Form(prefix="telephoneNumbers") List<TelephoneNumber> telephoneNumbers;
    @Form(prefix="address") Map<String, Address> addresses;
}

public static class TelephoneNumber {
    @FormParam("countryCode") private String countryCode;
    @FormParam("number") private String number;
}

public static class Address {
    @FormParam("street") private String street;
    @FormParam("houseNumber") private String houseNumber;
}

@Path("person")
public static class MyResource {

    @POST
    @Consumes(MediaType.APPLICATION_FORM_URLENCODED)
    public void post (@Form Person p) {}
```

The following form params could be submitted and the Person.telephoneNumbers and Person.addresses fields would be populated appropriately

```
request.addFormHeader("telephoneNumbers[0].countryCode", "31");
request.addFormHeader("telephoneNumbers[0].number", "0612345678");
request.addFormHeader("telephoneNumbers[1].countryCode", "91");
request.addFormHeader("telephoneNumbers[1].number", "9717738723");
request.addFormHeader("address[INVOICE].street", "Main Street");
request.addFormHeader("address[INVOICE].houseNumber", "2");
request.addFormHeader("address[SHIPPING].street", "Square One");
request.addFormHeader("address[SHIPPING].houseNumber", "13");
```

Chapter 13. Improved `@…Param` annotations

With the addition of parameter names in the bytecode since Java 8, it is no longer necessary to require users to specify parameter names in the following annotations: `@PathParam`, `@QueryParam`, `@FormParam`, `@CookieParam`, `@HeaderParam` and `@MatrixParam`. In order to benefit from this feature, you have to switch to new annotations with the same name, in a different package, which have an optional value parameter. To use this, follow these steps:

- Import the `org.jboss.resteasy.annotations.jaxrs` package to replace annotations from the Jakarta RESTful Web Services spec.
- Tell your build system to record method parameter names in the bytecode.
- Remove the annotation value if the name matches the name of the annotated variable.

Note that you can omit the annotation name for annotated method parameters as well as annotated fields or JavaBean properties.

For Maven users, recording method parameter names in the bytecode can be enabled by setting the `maven.compiler.parameters` to `true`:

```
    <properties>
        <maven.compiler.parameters>true</maven.compiler.parameters>
    </properties>
```

Usage:

```
import org.jboss.resteasy.annotations.jaxrs.*;

@Path("/library")
public class Library {

   @GET
   @Path("/book/{isbn}")
   public String getBook(@PathParam String isbn) {
      // search my database and get a string representation and return it
   }
}
```

If your annotated variable does not have the same name as the path parameter, you can still specify the name:

```
import org.jboss.resteasy.annotations.jaxrs.*;

@Path("/library")
public class Library {

   @GET
   @Path("/book/{isbn}")
   public String getBook(@PathParam("isbn") String id) {
      // search my database and get a string representation and return it
   }
}
```

Chapter 14. Optional parameter types

RESTEasy offers a mechanism to support a series of `java.util.Optional` types as a wrapper object types. This will give users the ability to use optional typed parameters, and eliminate all null checks by using methods like `Optional.orElse()`.

Here is the sample:

```
@Path("/double")@GETpublic String optDouble(@QueryParam("value") OptionalDouble value) {
 return Double.toString(value.orElse(4242.0));}
ble")
@GETpublic String optDouble(@QueryParam("value") OptionalDouble value)
{    return
Double.toString(value.orElse(4242.0));
```

From the above sample code we can see that the `OptionalDouble` can be used as parameter type, and when users don't provide a value in `@QueryParam`, then the default value will be returned.

Here is the list of supported optional parameter types:

- @QueryParam
- @FormParam
- @MatrixParam
- @HeaderParam
- @CookieParam

As the list shown above, those parameter types support the Java-provided `Optional` types. Please note that the `@PathParam` is an exception for which `Optional` is not available. The reason is that `Optional` for the `@PathParam` use case would just be a NO-OP, since an element of the path cannot be omitted.

The `Optional` types can also be used as type of the fields of a `@BeanParam`'s class.

Here is an example of endpoint with a `@BeanParam`:

```
@Path("/double")@GETpublic   String   optDouble(@BeanParam   Bean   bean)   {           return
 Double.toString(bean.value.orElse(4242.0));}
ble")
@GETpublic String optDouble(@BeanParam Bean bean)
{    return
```

```
Double.toString(bean.value.orElse(4242.0));
```

The corresponding class `Bean`:

```
public class Bean {    @QueryParam("value")    OptionalDouble value;}
{
@QueryParam("value")    OptionalDouble
value;
```

Finally, the `Optional` types can be used directly as type of the fields of a Jakarta RESTful Web Services resource class.

Here is an example of a Jakarta RESTful Web Services resource class with an `Optional` type:

```
@RequestScopedpublic class OptionalResource {    @QueryParam("value")    Optional<String> value;
    ...}
questScopedpublic class OptionalResource
 {
 @QueryParam("value")    Optional<String>
 value;
```

Chapter 15. @DefaultValue

@DefaultValue is a parameter annotation that can be combined with any of the other @*Param annotations to define a default value when the HTTP request item does not exist.

```
@GET
public String getBooks(@QueryParam("num") @DefaultValue("10") int num) {...}
```

Chapter 16. @Encoded and encoding

Jakarta RESTful Web Services allows you to get encoded or decoded @*Params and specify path definitions and parameter names using encoded or decoded strings.

The @javax.ws.rs.Encoded annotation can be used on a class, method, or param. By default, inject @PathParam and @QueryParams are decoded. By additionally adding the @Encoded annotation, the value of these params will be provided in encoded form.

```
@Path("/")
public class MyResource {

    @Path("/{param}")
    @GET
    public String get(@PathParam("param") @Encoded String param) {...}
}
```

In the above example, the value of the @PathParam injected into the param of the get() method will be URL encoded. Adding the @Encoded annotation as a paramater annotation triggers this affect.

You may also use the @Encoded annotation on the entire method and any combination of @QueryParam or @PathParam's values will be encoded.

```
@Path("/")
public class MyResource {

    @Path("/{param}")
    @GET
    @Encoded
    public String get(@QueryParam("foo") String foo, @PathParam("param") String param) {}
}
```

In the above example, the values of the "foo" query param and "param" path param will be injected as encoded values.

You can also set the default to be encoded for the entire class.

```
@Path("/")
@Encoded
```

```
public class ClassEncoded {

    @GET
    public String get(@QueryParam("foo") String foo) {}
}
```

The @Path annotation has an attribute called encode. Controls whether the literal part of the supplied value (those characters that are not part of a template variable) are URL encoded. If true, any characters in the URI template that are not valid URI character will be automatically encoded. If false then all characters must be valid URI characters. By default this is set to true. If you want to encoded the characters yourself, you may.

```
@Path(value="hello%20world", encode=false)
```

Much like @Path.encode(), this controls whether the specified query param name should be encoded by the container before it tries to find the query param in the request.

```
@QueryParam(value="hello%20world", encode=false)
```

Chapter 17. @Context

The @Context annotation allows you to inject instances of

- javax.ws.rs.core.HttpHeaders
- javax.ws.rs.core.UriInfo
- javax.ws.rs.core.Request
- javax.servlet.http.HttpServletRequest
- javax.servlet.http.HttpServletResponse
- javax.servlet.ServletConfig
- javax.servlet.ServletContext
- javax.ws.rs.core.SecurityContext

objects.

Chapter 18. Jakarta RESTful Web Services Resource Locators and Sub Resources

Resource classes are able to partially process a request and provide another "sub" resource object that can process the remainder of the request. For example:

```
@Path("/")
public class ShoppingStore {

    @Path("/customers/{id}")
    public Customer getCustomer(@PathParam("id") int id) {
        Customer cust = ...; // Find a customer object
        return cust;
    }
}

public class Customer {

    @GET
    public String get() {...}

    @Path("/address")
    public String getAddress() {...}
}
```

Resource methods that have a @Path annotation, but no HTTP method are considered sub-resource locators. Their job is to provide an object that can process the request. In the above example ShoppingStore is a root resource because its class is annotated with @Path. The getCustomer() method is a sub-resource locator method.

If the client invoked:

```
GET /customer/123
```

The ShoppingStore.getCustomer() method would be invoked first. This method provides a Customer object that can service the request. The http request will be dispatched to the Customer.get() method. Another example is:

```
GET /customer/123/address
```

In this request, again, first the ShoppingStore.getCustomer() method is invoked. A customer object is returned, and the rest of the request is dispatched to the Customer.getAddress() method.

Another interesting feature of Sub-resource locators is that the locator method result is dynamically processed at runtime to figure out how to dispatch the request. So, the ShoppingStore.getCustomer() method does not have to declare any specific type.

```
@Path("/")
public class ShoppingStore {

   @Path("/customers/{id}")
   public java.lang.Object getCustomer(@PathParam("id") int id) {
      Customer cust = ...; // Find a customer object
      return cust;
   }
}

public class Customer {

    @GET
    public String get() {...}

    @Path("/address")
    public String getAddress() {...}
}
```

In the above example, getCustomer() returns a java.lang.Object. Per request, at runtime, the Jakarta RESTful Web Services server will figure out how to dispatch the request based on the object returned by getCustomer(). What are the uses of this? Well, maybe you have a class hierarchy for your customers. Customer is the abstract base, CorporateCustomer and IndividualCustomer are subclasses. Your getCustomer() method might be doing a Hibernate polymorphic query and doesn't know, or care, what concrete class is it querying for, or what it returns.

```
@Path("/")
public class ShoppingStore {

   @Path("/customers/{id}")
   public java.lang.Object getCustomer(@PathParam("id") int id) {
      Customer cust = entityManager.find(Customer.class, id);
      return cust;
   }
}

public class Customer {

    @GET
    public String get() {...}

    @Path("/address")
    public String getAddress() {...}
```

```
}

public class CorporateCustomer extends Customer {

    @Path("/businessAddress")
    public String getAddress() {...}
}
```

Chapter 19. Resources metadata configuration

When processing Jakarta RESTful Web Services deployments, RESTEasy relies on *ResourceBuilder* to create metadata for each Jakarta RESTful Web Services resource. Such metadata is defined using the metadata SPI in package *org.jboss.resteasy.spi.metadata*, in particular the *ResourceClass* interface:

```
package org.jboss.resteasy.spi.metadata;

public interface ResourceClass
{
  String getPath();

  Class<?> getClazz();

  ResourceConstructor getConstructor();

  FieldParameter[] getFields();

  SetterParameter[] getSetters();

  ResourceMethod[] getResourceMethods();

  ResourceLocator[] getResourceLocators();
}
```

Among the other classes and interfaces defining metadata SPI, the following interfaces are worth a mention here:

```
public interface ResourceConstructor
{
  ResourceClass getResourceClass();

  Constructor getConstructor();

  ConstructorParameter[] getParams();
}

public interface ResourceMethod extends ResourceLocator
{
  Set<String> getHttpMethods();

  MediaType[] getProduces();

  MediaType[] getConsumes();
```

```
  boolean isAsynchronous();

  void markAsynchronous();
}

public interface ResourceLocator
{
  ResourceClass getResourceClass();

  Class<?> getReturnType();

  Type getGenericReturnType();

  Method getMethod();

  Method getAnnotatedMethod();

  MethodParameter[] getParams();

  String getFullpath();

  String getPath();

}
```

Now, the interesting point is that RESTEasy allows tuning the metadata generation by providing implementations of the *ResourceClassProcessor* interface:

```
package org.jboss.resteasy.spi.metadata;

public interface ResourceClassProcessor
{

  /**
   * Allows the implementation of this method to modify the resource metadata represented by
   * the supplied {@link ResourceClass} instance. Implementation will typically create
   * wrappers which modify only certain aspects of the metadata.
   *
   * @param clazz The original metadata
   * @return the (potentially modified) metadata (never null)
   */
  ResourceClass process(ResourceClass clazz);

}
```

The processors are meant to be, and are resolved as, regular Jakarta RESTful Web Services annotated providers. They allow for wrapping resource metadata classes with custom versions that can be used for various advanced scenarios like

- adding additional resource method/locators to the resource
- altering the http methods

- altering the @Produces / @Consumes media types
- ...

Chapter 20. Jakarta RESTful Web Services Content Negotiation

The HTTP protocol has built in content negotiation headers that allow the client and server to specify what content they are transferring and what content they would prefer to get. The server declares content preferences via the @Produces and @Consumes headers.

@Consumes is an array of media types that a particular resource or resource method consumes. For example:

```
@Consumes("text/*")
@Path("/library")
public class Library {

    @POST
    public String stringBook(String book) {...}

    @Consumes("text/xml")
    @POST
    public String jaxbBook(Book book) {...}
}
```

When a client makes a request, Jakarta RESTful Web Services first finds all methods that match the path, then, it sorts things based on the content-type header sent by the client. So, if a client sent:

```
POST /library
Content-Type: text/plain

This is a nice book
```

The stringBook() method would be invoked because it matches to the default "text/*" media type. Now, if the client instead sends XML:

```
POST /library
Content-Type: text/xml

<book name="EJB 3.0" author="Bill Burke"/>
```

The jaxbBook() method would be invoked.

The @Produces is used to map a client request and match it up to the client's Accept header. The Accept HTTP header is sent by the client and defines the media types the client prefers to receive from the server.

```
@Produces("text/*")
@Path("/library")
public class Library {

@GET
@Produces("application/json")
public String getJSON() {...}

@GET
public String get() {...}
```

So, if the client sends:

```
GET /library
Accept: application/json
```

The getJSON() method would be invoked.

@Consumes and @Produces can list multiple media types that they support. The client's Accept header can also send multiple types it might like to receive. More specific media types are chosen first. The client Accept header or @Produces @Consumes can also specify weighted preferences that are used to match up requests with resource methods. This is best explained by RFC 2616 section 14.1 . RESTEasy supports this complex way of doing content negotiation.

A variant in Jakarta RESTful Web Services is a combination of media type, content-language, and content encoding as well as etags, last modified headers, and other preconditions. This is a more complex form of content negotiation that is done programmatically by the application developer using the javax.ws.rs.Variant, VarianListBuilder, and Request objects. Request is injected via @Context. Read the javadoc for more info on these.

20.1. URL-based negotiation

Some clients, like browsers, cannot use the Accept and Accept-Language headers to negotiation the representation's media type or language. RESTEasy allows you to map file name suffixes like (.xml, .txt, .en, .fr) to media types and languages. These file name suffixes take the place and override any Accept header sent by the client. You configure this using the resteasy.media.type.mappings and resteasy.language.mappings parameters. If configured within your web.xml, it would look like:

```
<web-app>
    <display-name>Archetype Created Web Application</display-name>
    <context-param>
        <param-name>resteasy.media.type.mappings</param-name>
      <param-value>html : text/html, json : application/json, xml : application/xml</param-value>
    </context-param>

   <context-param>
        <param-name>resteasy.language.mappings</param-name>
        <param-value>en : en-US, es : es, fr : fr</param-value>
   </context-param>

   ...
</web-app>
```

See Section 3.4, “Configuration” for more information about application configuration.

Mappings are a comma delimited list of suffix/mediatype or suffix/language mappings. Each mapping is delimited by a ':'. So, if you invoked GET /foo/bar.xml.en, this would be equivalent to invoking the following request:

```
GET /foo/bar
Accept: application/xml
Accept-Language: en-US
```

The mapped file suffixes are stripped from the target URL path before the request is dispatched to a corresponding Jakarta RESTful Web Services resource.

20.2. Query String Parameter-based negotiation

RESTEasy can do content negotiation based in a parameter in query string. To enable this, the parameter resteasy.media.type.param.mapping can be configured. In web.xml, it would look like the following:

```
<web-app>
    <display-name>Archetype Created Web Application</display-name>
    <context-param>
        <param-name>resteasy.media.type.param.mapping</param-name>
        <param-value>someName</param-value>
    </context-param>

    ...
</web-app>
```

See Section 3.4, “Configuration” for more information about application configuration.

The param-value is the name of the query string parameter that RESTEasy will use in the place of the Accept header.

Invoking http://service.foo.com/resouce?someName=application/xml, will give the application/xml media type the highest priority in the content negotiation.

In cases where the request contains both the parameter and the Accept header, the parameter will be more relevant.

It is possible to left the param-value empty, what will cause the processor to look for a parameter named 'accept'.

Chapter 21. Content Marshalling/ Providers

21.1. Default Providers and default Jakarta RESTful Web Services Content Marshalling

RESTEasy can automatically marshal and unmarshal a few different message bodies.

Table 21.1.

Media Types	Java Type
application/*+xml, text/*+xml, application/*+json, application/*+fastinfoset, application/atom+*	Jakarta XML Binding annotated classes
application/*+xml, text/*+xml	org.w3c.dom.Document
/	java.lang.String
/	java.io.InputStream
text/plain	primitives, java.lang.String, or any type that has a String constructor, or static valueOf(String) method for input, toString() for output
/	javax.activation.DataSource
/	java.io.File
/	byte[]
application/x-www-form-urlencoded	javax.ws.rs.core.MultivaluedMap

Note. When a `java.io.File` is created, as in

```
@Path("/test")
public class TempFileDeletionResource
{
   @POST
   @Path("post")
   public Response post(File file) throws Exception
   {
      return Response.ok(file.getPath()).build();
   }
}
```

a temporary file is created in the file system. On the server side, that temporary file will be deleted at the end of the invocation. On the client side, however, it is the responsibility of the user to delete the temporary file.

21.2. Content Marshalling with @Provider classes

The Jakarta RESTful Web Services specification allows you to plug in your own request/response body reader and writers. To do this, you annotate a class with @Provider and specify the @Produces types for a writer and @Consumes types for a reader. You must also implement a MessageBodyReader/Writer interface respectively. Here is an example:

```
        @Provider
        @Produces("text/plain")
        @Consumes("text/plain")
        public class DefaultTextPlain implements MessageBodyReader, MessageBodyWriter {

            public boolean isReadable(Class type, Type genericType, Annotation[] annotations,
MediaType mediaType) {
               // StringTextStar should pick up strings
               return !String.class.equals(type) && TypeConverter.isConvertable(type);
            }

               public Object readFrom(Class type, Type genericType, Annotation[] annotations,
MediaType mediaType, MultivaluedMap httpHeaders, InputStream entityStream) throws IOException,
WebApplicationException {
               InputStream delegate = NoContent.noContentCheck(httpHeaders, entityStream);
               String value = ProviderHelper.readString(delegate, mediaType);
               return TypeConverter.getType(type, value);
            }

            public boolean isWriteable(Class type, Type genericType, Annotation[] annotations,
MediaType mediaType) {
               // StringTextStar should pick up strings
               return !String.class.equals(type) && !type.isArray();
            }

          public long getSize(Object o, Class type, Type genericType, Annotation[] annotations,
MediaType mediaType) {
               String charset = mediaType.getParameters().get("charset");
               if (charset != null)
                  try {
                     return o.toString().getBytes(charset).length;
                  } catch (UnsupportedEncodingException e) {
                     // Use default encoding.
                  }
               return o.toString().getBytes(StandardCharsets.UTF_8).length;
            }

          public void writeTo(Object o, Class type, Type genericType, Annotation[] annotations,
MediaType mediaType, MultivaluedMap httpHeaders, OutputStream entityStream) throws IOException,
WebApplicationException {
               String charset = mediaType.getParameters().get("charset");
```

```
                                                                    if    (charset    ==    null)
entityStream.write(o.toString().getBytes(StandardCharsets.UTF_8));
               else entityStream.write(o.toString().getBytes(charset));
            }
         }
```

Note that in order to support Async IO, you need to implement the `AsyncMessageBodyWriter` interface, which requires you to implement this extra method:

```
         @Provider
         @Produces("text/plain")
         @Consumes("text/plain")
         public class DefaultTextPlain implements MessageBodyReader, AsyncMessageBodyWriter {
            // ...
             public CompletionStage<Void> asyncWriteTo(Object o, Class type, Type genericType,
Annotation[] annotations, MediaType mediaType, MultivaluedMap httpHeaders, AsyncOutputStream
entityStream) {
               String charset = mediaType.getParameters().get("charset");
               if (charset == null)
                 return entityStream.asyncWrite(o.toString().getBytes(StandardCharsets.UTF_8));
               else
                  return entityStream.asyncWrite(o.toString().getBytes(charset));
            }
         }
```

The RESTEasy ServletContextLoader will automatically scan your WEB-INF/lib and classes directories for classes annotated with @Provider or you can manually configure them in web.xml. See Installation/Configuration.

21.3. Providers Utility Class

javax.ws.rs.ext.Providers is a simple injectable interface that allows you to look up MessageBodyReaders, Writers, ContextResolvers, and ExceptionMappers. It is very useful, for instance, for implementing multipart providers. Content types that embed other random content types.

```
public interface Providers
{

   /**
    * Get a message body reader that matches a set of criteria. The set of
    * readers is first filtered by comparing the supplied value of
    * {@code mediaType} with the value of each reader's
    * {@link javax.ws.rs.Consumes}, ensuring the supplied value of
    * {@code type} is assignable to the generic type of the reader, and
    * eliminating those that do not match.
    * The list of matching readers is then ordered with those with the best
```

```
    * matching values of {@link javax.ws.rs.Consumes} (x/y > x&#47;* > *&#47;*)
    * sorted first. Finally, the
    * {@link MessageBodyReader#isReadable}
    * method is called on each reader in order using the supplied criteria and
    * the first reader that returns {@code true} is selected and returned.
    *
    * @param type        the class of object that is to be written.
    * @param mediaType   the media type of the data that will be read.
    * @param genericType the type of object to be produced. E.g. if the
    *                    message body is to be converted into a method parameter, this will be
    *                    the formal type of the method parameter as returned by
    *                    <code>Class.getGenericParameterTypes</code>.
    * @param annotations an array of the annotations on the declaration of the
    *                   artifact that will be initialized with the produced instance. E.g. if the
    *                    message body is to be converted into a method parameter, this will be
    *                    the annotations on that parameter returned by
    *                    <code>Class.getParameterAnnotations</code>.
    * @return a MessageBodyReader that matches the supplied criteria or null
    *         if none is found.
    */
   <T> MessageBodyReader<T> getMessageBodyReader(Class<T> type,
                                                                 Type genericType, Annotation
annotations[], MediaType mediaType);

   /**
    * Get a message body writer that matches a set of criteria. The set of
    * writers is first filtered by comparing the supplied value of
    * {@code mediaType} with the value of each writer's
    * {@link javax.ws.rs.Produces}, ensuring the supplied value of
    * {@code type} is assignable to the generic type of the reader, and
    * eliminating those that do not match.
    * The list of matching writers is then ordered with those with the best
    * matching values of {@link javax.ws.rs.Produces} (x/y > x&#47;* > *&#47;*)
    * sorted first. Finally, the
    * {@link MessageBodyWriter#isWriteable}
    * method is called on each writer in order using the supplied criteria and
    * the first writer that returns {@code true} is selected and returned.
    *
    * @param mediaType   the media type of the data that will be written.
    * @param type        the class of object that is to be written.
    * @param genericType the type of object to be written. E.g. if the
    *                    message body is to be produced from a field, this will be
    *                    the declared type of the field as returned by
    *                    <code>Field.getGenericType</code>.
    * @param annotations an array of the annotations on the declaration of the
    *                    artifact that will be written. E.g. if the
    *                    message body is to be produced from a field, this will be
    *                    the annotations on that field returned by
    *                    <code>Field.getDeclaredAnnotations</code>.
    * @return a MessageBodyReader that matches the supplied criteria or null
    *         if none is found.
    */
   <T> MessageBodyWriter<T> getMessageBodyWriter(Class<T> type,
                                                                 Type genericType, Annotation
annotations[], MediaType mediaType);

   /**
    * Get an exception mapping provider for a particular class of exception.
    * Returns the provider whose generic type is the nearest superclass of
```

```
     * {@code type}.
     *
     * @param type the class of exception
     * @return an {@link ExceptionMapper} for the supplied type or null if none
     *         is found.
     */
    <T extends Throwable> ExceptionMapper<T> getExceptionMapper(Class<T> type);

    /**
     * Get a context resolver for a particular type of context and media type.
     * The set of resolvers is first filtered by comparing the supplied value of
     * {@code mediaType} with the value of each resolver's
     * {@link javax.ws.rs.Produces}, ensuring the generic type of the context
     * resolver is assignable to the supplied value of {@code contextType}, and
     * eliminating those that do not match. If only one resolver matches the
     * criteria then it is returned. If more than one resolver matches then the
     * list of matching resolvers is ordered with those with the best
     * matching values of {@link javax.ws.rs.Produces} (x/y > x&#47;* > *&#47;*)
     * sorted first. A proxy is returned that delegates calls to
     * {@link ContextResolver#getContext(java.lang.Class)} to each matching context
     * resolver in order and returns the first non-null value it obtains or null
     * if all matching context resolvers return null.
     *
     * @param contextType the class of context desired
     * @param mediaType   the media type of data for which a context is required.
     * @return a matching context resolver instance or null if no matching
     *         context providers are found.
     */
    <T> ContextResolver<T> getContextResolver(Class<T> contextType,
                                              MediaType mediaType);
}
```

A Providers instance is injectable into MessageBodyReader or Writers:

```
@Provider
@Consumes("multipart/fixed")
public class MultipartProvider implements MessageBodyReader {

    private @Context Providers providers;

    ...

}
```

21.4. Configuring Document Marshalling

XML document parsers are subject to a form of attack known as the XXE (Xml eXternal Entity) Attack (https://owasp.org/www-community/vulnerabilities/XML_External_Entity_(XXE)_Processing), in which expanding an external entity causes an unsafe file to be loaded. For example, the document

```
<?xml version="1.0"?>
<!DOCTYPE foo
[<!ENTITY xxe SYSTEM "file:///etc/passwd">]>
<search>
    <user>bill</user>
    <file>&xxe;<file>
</search>
```

could cause the passwd file to be loaded.

By default, RESTEasy's built-in unmarshaller for org.w3c.dom.Document documents will not expand external entities, replacing them by the empty string instead. It can be configured to replace external entities by values defined in the DTD by setting the parameter

to "true". If configured in the web.xml file, it would be:

```
<context-param>
    <param-name>resteasy.document.expand.entity.references</param-name>
    <param-value>true</param-value>
</context-param>
```

See Section 3.4, “Configuration” for more information about application configuration.

Another way of dealing with the problem is by prohibiting DTDs, which RESTEasy does by default. This behavior can be changed by setting the parameter

to "false".

Documents are also subject to Denial of Service Attacks when buffers are overrun by large entities or too many attributes. For example, if a DTD defined the following entities

```
<!ENTITY foo 'foo'>
<!ENTITY foo1 '&foo;&foo;&foo;&foo;&foo;&foo;&foo;&foo;&foo;&foo;'>
<!ENTITY foo2 '&foo1;&foo1;&foo1;&foo1;&foo1;&foo1;&foo1;&foo1;&foo1;&foo1;'>
<!ENTITY foo3 '&foo2;&foo2;&foo2;&foo2;&foo2;&foo2;&foo2;&foo2;&foo2;&foo2;'>
<!ENTITY foo4 '&foo3;&foo3;&foo3;&foo3;&foo3;&foo3;&foo3;&foo3;&foo3;&foo3;'>
<!ENTITY foo5 '&foo4;&foo4;&foo4;&foo4;&foo4;&foo4;&foo4;&foo4;&foo4;&foo4;'>
<!ENTITY foo6 '&foo5;&foo5;&foo5;&foo5;&foo5;&foo5;&foo5;&foo5;&foo5;&foo5;'>
```

then the expansion of &foo6; would result in 1,000,000 foos. By default, RESTEasy will limit the number of expansions and the number of attributes per entity. The exact behavior depends on the underlying parser. The limits can be turned off by setting the parameter

to "false".

21.5. Text media types and character sets

The Jakarta RESTful Web Services specification says

```
When writing responses, implementations SHOULD respect application-supplied character
set metadata and SHOULD use UTF-8 if a character set is not specified by the application
or if the application specifies a character set that is unsupported.
```

On the other hand, the HTTP specification says

```
When no explicit charset parameter is provided by the sender, media subtypes of the
"text" type are defined to have a default charset value of "ISO-8859-1" when received
via HTTP. Data in character sets other than "ISO-8859-1" or its subsets MUST be labeled
with an appropriate charset value.
```

It follows that, in the absence of a character set specified by a resource or resource method, RESTEasy SHOULD use UTF-8 as the character set for text media types, and, if it does, it MUST add an explicit charset parameter to the Content-Type response header. RESTEasy started adding the explicit charset parameter in releases 3.1.2.Final and 3.0.22.Final, and that new behavior could cause some compatibility problems. To specify the previous behavior, in which UTF-8 was used for text media types, but the explicit charset was not appended, the parameter "resteasy.add.charset" may be set to "false". It defaults to "true".

Note. By "text" media types, we mean

- a media type with type "text" and any subtype;
- a media type with type ""application" and subtype beginning with "xml".

The latter set includes "application/xml-external-parsed-entity" and "application/xml-dtd".

Chapter 22. Jakarta XML Binding providers

As required by the specification, RESTEasy includes support for (un)marshalling Jakarta XML Binding annotated classes. RESTEasy provides multiple providers to address some subtle differences between classes generated by XJC and classes which are simply annotated with @XmlRootElement, or working with JAXBElement classes directly.

For the most part, developers using the Jakarta RESTful Web Services API, the selection of which provider is invoked will be completely transparent. For developers wishing to access the providers directly (which most folks won't need to do), this document describes which provider is best suited for different configurations.

A Jakarta XML Binding Provider is selected by RESTEasy when a parameter or return type is an object that is annotated with Jakarta XML Binding annotations (such as @XmlRootEntity or @XmlType) or if the type is a JAXBElement. Additionally, the resource class or resource method will be annotated with either a @Consumes or @Produces annotation and contain one or more of the following values:

- text/*+xml
- application/*+xml
- application/*+fastinfoset
- application/*+json

RESTEasy will select a different provider based on the return type or parameter type used in the resource. This section describes how the selection process works.

@XmlRootEntity When a class is annotated with a @XmlRootElement annotation, RESTEasy will select the JAXBXmlRootElementProvider. This provider handles basic marshaling and unmarshalling of custom Jakarta XML Binding entities.

@XmlType Classes which have been generated by XJC will most likely not contain an @XmlRootEntity annotation. In order for these classes to marshalled, they must be wrapped within a JAXBElement instance. This is typically accomplished by invoking a method on the class which serves as the XmlRegistry and is named ObjectFactory.

The JAXBXmlTypeProvider provider is selected when the class is annotated with an XmlType annotation and not an XmlRootElement annotation.

This provider simplifies this task by attempting to locate the XmlRegistry for the target class. By default, a Jakarta XML Binding implementation will create a class called ObjectFactory and is located in the same package as the target class. When this class is located, it will contain a "create"

method that takes the object instance as a parameter. For example, if the target type is called "Contact", then the ObjectFactory class will have a method:

public JAXBElement createContact(Contact value) {..

JAXBElement<?> If your resource works with the JAXBElement class directly, the RESTEasy runtime will select the JAXBElementProvider. This provider examines the ParameterizedType value of the JAXBElement in order to select the appropriate JAXBContext.

22.1. Jakarta XML Binding Decorators

Resteasy's Jakarta XML Binding providers have a pluggable way to decorate Marshaller and Unmarshaller instances. The way it works is that you can write an annotation that can trigger the decoration of a Marshaller or Unmarshaller. Your decorators can do things like set Marshaller or Unmarshaller properties, set up validation, stuff like that. Here's an example. Let's say we want to have an annotation that will trigger pretty-printing, nice formatting, of an XML document. If we were doing raw Jakarta XML Binding, we would set a property on the Marshaller of Marshaller.JAXB_FORMATTED_OUTPUT. Let's write a Marshaller decorator.

First we define a annotation:

```
import org.jboss.resteasy.annotations.Decorator;

@Target({ElementType.TYPE, ElementType.METHOD, ElementType.PARAMETER, ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
@Decorator(processor = PrettyProcessor.class, target = Marshaller.class)
public @interface Pretty {}
```

To get this to work, we must annotate our @Pretty annotation with a meta-annotation called @Decorator. The target() attribute must be the Jakarta XML Binding Marshaller class. The processor() attribute is a class we will write next.

```
import org.jboss.resteasy.core.interception.DecoratorProcessor;
import org.jboss.resteasy.annotations.DecorateTypes;

import javax.xml.bind.Marshaller;
import javax.xml.bind.PropertyException;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.Produces;
import java.lang.annotation.Annotation;

/**
 * @author <a href="mailto:bill@burkecentral.com">Bill Burke</a>
 * @version $Revision: 1 $
 */
@DecorateTypes({"text/*+xml", "application/*+xml"})
public class PrettyProcessor implements DecoratorProcessor<Marshaller, Pretty>
```

```
{
    public Marshaller decorate(Marshaller target, Pretty annotation,
                  Class type, Annotation[] annotations, MediaType mediaType)
    {
       target.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
    }
}
```

The processor implementation must implement the DecoratorProcessor interface and should also be annotated with @DecorateTypes. This annotation specifies what media types the processor can be used with. Now that we've defined our annotation and our Processor, we can use it on our Jakarta RESTful Web Services resource methods or Jakarta XML Binding types as follows:

```
@GET
@Pretty
@Produces("application/xml")
public SomeJAXBObject get() {...}
```

If you are confused, check the RESTEasy source code for the implementation of @XmlHeader

22.2. Pluggable JAXBContext's with ContextResolvers

You should not use this feature unless you know what you're doing.

Based on the class you are marshalling/unmarshalling, RESTEasy will, by default create and cache JAXBContext instances per class type. If you do not want RESTEasy to create JAXBContexts, you can plug-in your own by implementing an instance of javax.ws.rs.ext.ContextResolver

```
public interface ContextResolver<T>
{
    T getContext(Class<?> type);
}

@Provider
@Produces("application/xml")
public class MyJAXBContextResolver implements ContextResolver<JAXBContext>
{
    JAXBContext getContext(Class<?> type)
    {
       if (type.equals(WhateverClassIsOverridedFor.class)) return JAXBContext.newInstance()...;
    }
}
```

You must provide a @Produces annotation to specify the media type the context is meant for. You must also make sure to implement ContextResolver<JAXBContext>. This helps the runtime match to the correct context resolver. You must also annotate the ContextResolver class with @Provider.

There are multiple ways to make this ContextResolver available.

1. Return it as a class or instance from a javax.ws.rs.core.Application implementation
2. List it as a provider with resteasy.providers
3. Let RESTEasy automatically scan for it within your WAR file. See Configuration Guide
4. Manually add it via ResteasyProviderFactory.getInstance().registerProvider(Class) or registerProviderInstance(Object)

22.3. Jakarta XML Binding + XML provider

RESTEasy is required to provide Jakarta XML Binding provider support for XML. It has a few extra annotations that can help code your app.

22.3.1. @XmlHeader and @Stylesheet

Sometimes when outputting XML documents you may want to set an XML header. RESTEasy provides the @org.jboss.resteasy.annotations.providers.jaxb.XmlHeader annotation for this. For example:

```
@XmlRootElement
public static class Thing
{
    private String name;

    public String getName()
    {
        return name;
    }

    public void setName(String name)
    {
        this.name = name;
    }
}

@Path("/test")
public static class TestService
{
    @GET
    @Path("/header")
    @Produces("application/xml")
    @XmlHeader("<?xml-stylesheet type='text/xsl' href='${baseuri}foo.xsl' ?>")
    public Thing get()
    {
        Thing thing = new Thing();
        thing.setName("bill");
        return thing;
    }
}
```

The @XmlHeader here forces the XML output to have an xml-stylesheet header. This header could also have been put on the Thing class to get the same result. See the javadocs for more details on how you can use substitution values provided by resteasy.

RESTEasy also has a convenience annotation for stylesheet headers. For example:

```
@XmlRootElement
public static class Thing
{
    private String name;

    public String getName()
    {
        return name;
    }

    public void setName(String name)
    {
        this.name = name;
    }
}

@Path("/test")
public static class TestService
{
    @GET
    @Path("/stylesheet")
    @Produces("application/xml")
    @Stylesheet(type="text/css", href="${basepath}foo.xsl")
    @Junk
    public Thing getStyle()
    {
        Thing thing = new Thing();
        thing.setName("bill");
        return thing;
    }
}
```

22.4. Jakarta XML Binding + JSON provider

RESTEasy allows you to marshall Jakarta XML Binding annotated POJOs to and from JSON. This provider wraps the Jackson2 library to accomplish this.

To use this integration with Jackson you need to import the resteasy-jackson2-provider Maven module.

For example, consider this Jakarta XML Binding class:

```
@XmlRootElement(name = "book")
public class Book
{
```

```
    private String author;
    private String ISBN;
    private String title;

    public Book()
    {
    }

    public Book(String author, String ISBN, String title)
    {
        this.author = author;
        this.ISBN = ISBN;
        this.title = title;
    }

    @XmlElement
    public String getAuthor()
    {
        return author;
    }

    public void setAuthor(String author)
    {
    this.author = author;
    }

    @XmlElement
    public String getISBN()
    {
        return ISBN;
    }

    public void setISBN(String ISBN)
    {
        this.ISBN = ISBN;
    }

    @XmlAttribute
    public String getTitle()
    {
        return title;
    }

    public void setTitle(String title)
    {
        this.title = title;
    }
}
```

And we can write a method to use the above entity:

```
@Path("/test_json")
@GET
@Produces(MediaType.APPLICATION_JSON)
public Book test_json() {
```

```
    Book book = new Book();
    book.setTitle("EJB 3.0");
    book.setAuthor("Bill Burke");
    book.setISBN("596529260");
    return book;
}
```

Requesting from the above method, and we can see the default Jackson2 marshaller would return JSON that looked like this:

```
$ http localhost:8080/dummy/test_json
HTTP/1.1 200
...
Content-Type: application/json

{
"ISBN": "596529260",
"author": "Bill Burke",
"title": "EJB 3.0"
}
```

22.5. Jakarta XML Binding + FastinfoSet provider

RESTEasy supports the FastinfoSet mime type with Jakarta XML Binding annotated classes. Fast infoset documents are faster to serialize and parse, and smaller in size, than logically equivalent XML documents. Thus, fast infoset documents may be used whenever the size and processing time of XML documents is an issue. It is configured the same way the provider is so really no other documentation is needed here.

To use this integration with Fastinfoset you need to import the resteasy-fastinfoset-provider Maven module. Older versions of RESTEasy used to include this within the resteasy-jaxb-provider but we decided to modularize it more.

22.6. Arrays and Collections of Jakarta XML Binding Objects

RESTEasy will automatically marshal arrays, java.util.Set's, and java.util.List's of Jakarta XML Binding objects to and from XML, JSON, Fastinfoset (or any other new Jakarta XML Binding mapper Restasy comes up with).

```
@XmlRootElement(name = "customer")
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer
{
    @XmlElement
```

```
    private String name;

    public Customer()
    {
    }

    public Customer(String name)
    {
        this.name = name;
    }

    public String getName()
    {
        return name;
    }
}

@Path("/")
public class MyResource
{
    @PUT
    @Path("array")
    @Consumes("application/xml")
    public void putCustomers(Customer[] customers)
    {
        Assert.assertEquals("bill", customers[0].getName());
        Assert.assertEquals("monica", customers[1].getName());
    }

    @GET
    @Path("set")
    @Produces("application/xml")
    public Set<Customer> getCustomerSet()
    {
        HashSet<Customer> set = new HashSet<Customer>();
        set.add(new Customer("bill"));
        set.add(new Customer("monica"));

        return set;
    }

    @PUT
    @Path("list")
    @Consumes("application/xml")
    public void putCustomers(List<Customer> customers)
    {
        Assert.assertEquals("bill", customers.get(0).getName());
        Assert.assertEquals("monica", customers.get(1).getName());
    }
}
```

The above resource can publish and receive Jakarta XML Binding objects. It is assumed that are wrapped in a collection element

```
<collection>
```

```
    <customer><name>bill</name></customer>
    <customer><name>monica</name></customer>
<collection>
```

You can change the namespace URI, namespace tag, and collection element name by using the @org.jboss.resteasy.annotations.providers.jaxb.Wrapped annotation on a parameter or method

```
@Target({ElementType.PARAMETER, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface Wrapped
{
    String element() default "collection";

    String namespace() default "http://jboss.org/resteasy";

    String prefix() default "resteasy";
}
```

So, if we wanted to output this XML

```
<foo:list xmlns:foo="http://foo.org">
    <customer><name>bill</name></customer>
    <customer><name>monica</name></customer>
</foo:list>
```

We would use the @Wrapped annotation as follows:

```
@GET
@Path("list")
@Produces("application/xml")
@Wrapped(element="list", namespace="http://foo.org", prefix="foo")
public List<Customer> getCustomerSet()
{
    List<Customer> list = new ArrayList<Customer>();
    list.add(new Customer("bill"));
    list.add(new Customer("monica"));

    return list;
}
```

22.6.1. Retrieving Collections on the client side

If you try to retrieve a `List` or `Set` of Jakarta XML Binding objects in the obvious way on the client side:

```
Response response = request.get();
List<Customer> list = response.readEntity(List.class);
```

the call to `readEntity()` will fail because it has no way of knowing the element type `Customer`. The trick is to use an instance of `javax.ws.rs.core.GenericType`:

```
Response response = request.get();
GenericType<List<Customer>> genericType = new GenericType<List<Customer>>() {};
List<Customer> list = response.readEntity(genericType);
```

For more information about `GenericType`, please see its javadoc.

The same trick applies to retrieving a `Set`:

```
Response response = request.get();
GenericType<Set<Customer>> genericType = new GenericType<Set<Customer>>() {};
Set<Customer> set = response.readEntity(genericType);
```

On the other hand, `GenericType` is not necessary to retrieve an array of Jakarta XML Binding objects:

```
Response response = request.get();
Customer[] array = response.readEntity(Customer[].class);
```

22.6.2. JSON and Jakarta XML Binding Collections/arrays

RESTEasy supports using collections with JSON. It encloses lists, sets, or arrays of returned XML objects within a simple JSON array. For example:

```
@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public static class Foo
{
    @XmlAttribute
    private String test;

    public Foo()
    {
    }
```

```
    public Foo(String test)
    {
        this.test = test;
    }

    public String getTest()
    {
        return test;
    }

    public void setTest(String test)
    {
        this.test = test;
    }
}
```

This a List or array of this Foo class would be represented in JSON like this:

```
[{"foo":{"@test":"bill"}},{"foo":{"@test":"monica}"}}]
```

It also expects this format for input

22.7. Maps of XML Objects

RESTEasy will automatically marshal maps of &XML-BIND-API; objects to and from XML, JSON, Fastinfoset (or any other new &XML-BIND-API; mapper RESTEasy comes up with). Your parameter or method return type must be a generic with a String as the key and the &XML-BIND-API; object's type.

```
@XmlRootElement(namespace = "http://foo.com")
public static class Foo
{
    @XmlAttribute
    private String name;

    public Foo()
    {
    }

    public Foo(String name)
    {
        this.name = name;
    }

    public String getName()
    {
        return name;
    }
}
```

```
@Path("/map")
public static class MyResource
{
    @POST
    @Produces("application/xml")
    @Consumes("application/xml")
    public Map<String, Foo> post(Map<String, Foo> map)
    {
        Assert.assertEquals(2, map.size());
        Assert.assertNotNull(map.get("bill"));
        Assert.assertNotNull(map.get("monica"));
        Assert.assertEquals(map.get("bill").getName(), "bill");
        Assert.assertEquals(map.get("monica").getName(), "monica");
        return map;
    }
}
```

The above resource can publish and receive XML objects within a map. By default, they are wrapped in a "map" element in the default namespace. Also, each "map" element has zero or more "entry" elements with a "key" attribute.

```
<map>
    <entry key="bill" xmlns="http://foo.com">
        <foo name="bill"/>
    </entry>
    <entry key="monica" xmlns="http://foo.com">
        <foo name="monica"/>
    </entry>
</map>
```

You can change the namespace URI, namespace prefix and map, entry, and key element and attribute names by using the @org.jboss.resteasy.annotations.providers.jaxb.WrappedMap annotation on a parameter or method

```
@Target({ElementType.PARAMETER, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface WrappedMap
{
    /**
     * map element name
     */
    String map() default "map";

    /**
     * entry element name *
     */
    String entry() default "entry";

    /**
     * entry's key attribute name
```

```
     */
    String key() default "key";

    String namespace() default "";

    String prefix() default "";
}
```

So, if we wanted to output this XML

```
<hashmap>
    <hashentry hashkey="bill" xmlns:foo="http://foo.com">
        <foo:foo name="bill"/>
    </hashentry>
</map>
```

We would use the @WrappedMap annotation as follows:

```
@Path("/map")
public static class MyResource
{
    @GET
    @Produces("application/xml")
    @WrappedMap(map="hashmap", entry="hashentry", key="hashkey")
    public Map<String, Foo> get()
    {
        ...
        return map;
    }
}
```

22.7.1. Retrieving Maps on the client side

If you try to retrieve a `Map` of XML objects in the obvious way on the client side:

```
      Response response = request.get();
      Map<String, Customer> map = response.readEntity(Map.class);
```

the call to `readEntity()` will fail because it has no way of knowing the element type `Customer`. The trick is to use an instance of `javax.ws.rs.core.GenericType`:

```
      Response response = request.get();
    GenericType<Map<String, Customer> genericType = new GenericType<Map<String, Customer>>() {};
```

```
      Map<String, Customer> map = response.readEntity(genericType);
```

For more information about `GenericType`, please see its javadoc.

22.7.2. JSON and XML maps

RESTEasy supports using maps with JSON. It encloses maps returned XML objects within a simple JSON map. For example:

```
@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public static class Foo
{
    @XmlAttribute
    private String test;

    public Foo()
    {
    }

    public Foo(String test)
    {
        this.test = test;
    }

    public String getTest()
    {
        return test;
    }

    public void setTest(String test)
    {
        this.test = test;
    }
}
```

This a List or array of this Foo class would be represented in JSON like this:

```
{ "entry1" : {"foo":{"@test":"bill"}}, "entry2" : {"foo":{"@test":"monica}"}}}
```

It also expects this format for input

22.8. Interfaces, Abstract Classes, and Jakarta XML Binding

Some objects models use abstract classes and interfaces heavily. Unfortunately, Jakarta XML Binding doesn't work with interfaces that are root elements and RESTEasy can't unmarshal pa-

rameters that are interfaces or raw abstract classes because it doesn't have enough information to create a JAXBContext. For example:

```
public interface IFoo {}

@XmlRootElement
public class RealFoo implements IFoo {}

@Path("/xml")
public class MyResource {

    @PUT
    @Consumes("application/xml")
    public void put(IFoo foo) {...}
}
```

In this example, you would get an error from RESTEasy of something like "Cannot find a MessageBodyReader for...". This is because RESTEasy does not know that implementations of IFoo are Jakarta XML Binding classes and doesn't know how to create a JAXBContext for it. As a workaround, RESTEasy allows you to use the Jakarta XML Binding annotation @XmlSeeAlso on the interface to correct the problem. (NOTE, this will not work with manual, hand-coded Jakarta XML Binding).

```
@XmlSeeAlso(RealFoo.class)
public interface IFoo {}
```

The extra @XmlSeeAlso on IFoo allows RESTEasy to create a JAXBContext that knows how to unmarshal RealFoo instances.

22.9. Configuring Jakarta XML Binding Marshalling

As a consumer of XML datasets, Jakarta XML Binding is subject to a form of attack known as the XXE (Xml eXternal Entity) Attack (https://owasp.org/www-community/vulnerabilities/XML_External_Entity_(XXE)_Processing), in which expanding an external entity causes an unsafe file to be loaded. Preventing the expansion of external entities is discussed in Section 21.4, “Configuring Document Marshalling”. The same parameter,

applies to Jakarta XML Binding unmarshallers as well.

Section 21.4, “Configuring Document Marshalling” also discusses the prohibition of DTDs and the imposition of limits on entity expansion and the number of attributes per element. The parameters

and

discussed there, and their default values, also apply to the representation of Jakarta XML Binding objects.

Chapter 23. RESTEasy Atom Support

From W3.org (http://tools.ietf.org/html/rfc4287):

"Atom is an XML-based document format that describes lists of related information known as "feeds". Feeds are composed of a number of items, known as "entries", each with an extensible set of attached metadata. For example, each entry has a title. The primary use case that Atom addresses is the syndication of Web content such as weblogs and news headlines to Web sites as well as directly to user agents."

Atom is the next-gen RSS feed. Although it is used primarily for the syndication of blogs and news, many are starting to use this format as the envelope for Web Services, for example, distributed notifications, job queues, or simply a nice format for sending or receiving data in bulk from a service.

23.1. RESTEasy Atom API and Provider

RESTEasy has defined a simple object model in Java to represent Atom and uses Jakarta XML Binding to marshal and unmarshal it. The main classes are in the org.jboss.resteasy.plugins.providers.atom package and are Feed, Entry, Content, and Link. If you look at the source, you'd see that these are annotated with Jakarta XML Binding annotations. The distribution contains the javadocs for this project and are a must to learn the model. Here is a simple example of sending an atom feed using the RESTEasy API.

```
import org.jboss.resteasy.plugins.providers.atom.Content;
import org.jboss.resteasy.plugins.providers.atom.Entry;
import org.jboss.resteasy.plugins.providers.atom.Feed;
import org.jboss.resteasy.plugins.providers.atom.Link;
import org.jboss.resteasy.plugins.providers.atom.Person;

@Path("atom")
public class MyAtomService
{
    @GET
    @Path("feed")
    @Produces("application/atom+xml")
    public Feed getFeed() throws URISyntaxException
    {
        Feed feed = new Feed();
        feed.setId(new URI("http://example.com/42"));
        feed.setTitle("My Feed");
        feed.setUpdated(new Date());
        Link link = new Link();
        link.setHref(new URI("http://localhost"));
        link.setRel("edit");
        feed.getLinks().add(link);
```

```
        feed.getAuthors().add(new Person("Bill Burke"));
        Entry entry = new Entry();
        entry.setTitle("Hello World");
        Content content = new Content();
        content.setType(MediaType.TEXT_HTML_TYPE);
        content.setText("Nothing much");
        entry.setContent(content);
        feed.getEntries().add(entry);
        return feed;
    }
}
```

Because RESTEasy's atom provider is Jakarta XML Binding based, you are not limited to sending atom objects using XML. You can automatically re-use all the other Jakarta XML Binding providers that RESTEasy has like JSON and fastinfoset. All you have to do is have "atom+" in front of the main subtype. i.e. @Produces("application/atom+json") or @Consumes("application/atom+fastinfoset")

23.2. Using Jakarta XML Binding with the Atom Provider

The org.jboss.resteasy.plugins.providers.atom.Content class allows you to unmarshal and marshal Jakarta XML Binding annotated objects that are the body of the content. Here's an example of sending an Entry with a Customer object attached as the body of the entry's content.

```
@XmlRootElement(namespace = "http://jboss.org/Customer")
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer
{
    @XmlElement
    private String name;

    public Customer()
    {
    }

    public Customer(String name)
    {
        this.name = name;
    }

    public String getName()
    {
        return name;
    }
}

@Path("atom")
public static class AtomServer
{
    @GET
```

```
    @Path("entry")
    @Produces("application/atom+xml")
    public Entry getEntry()
    {
        Entry entry = new Entry();
        entry.setTitle("Hello World");
        Content content = new Content();
        content.setJAXBObject(new Customer("bill"));
        entry.setContent(content);
        return entry;
    }
}
```

The Content.setJAXBObject() method is used to tell the content object you are sending back an object and want it marshalled appropriately. If you are using a different base format other than XML, i.e. "application/atom+json", this attached object will be marshalled into that same format.

If you have an atom document as your input, you can also extract Jakarta XML Binding objects from Content using the Content.getJAXBObject(Class clazz) method. Here is an example of an input atom document and extracting a Customer object from the content.

```
@Path("atom")
public static class AtomServer
{
    @PUT
    @Path("entry")
    @Produces("application/atom+xml")
    public void putCustomer(Entry entry)
    {
        Content content = entry.getContent();
        Customer cust = content.getJAXBObject(Customer.class);
    }
}
```

Chapter 24. JSON Support via Jackson

RESTEasy supports integration with the Jackson project. For more on Jackson 2, see https://github.com/FasterXML/jackson-databind/wiki. Besides Jakarta XML Binding like APIs, it has a JavaBean based model, described at https://github.com/FasterXML/jackson-databind/wiki/Data-bind-annotations, which allows you to easily marshal Java objects to and from JSON. RESTEasy integrates with the JavaBean model. While Jackson does come with its own Jakarta RESTful Web Services integration, RESTEasy expanded it a little, as decribed below.

NOTE. The resteasy-jackson-provider module, which is based on the outdated Jackson 1.9.x, is currently deprecated, and will be removed in a release subsequent to 3.1.0.Final. The resteasy-jackson2-provider module is based on Jackson 2.

24.1. Using Jackson 1.9.x Outside of WildFly

If you're deploying RESTEasy outside of WildFly, add the RESTEasy Jackson provder to your WAR pom.xml build:

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

24.2. Using Jackson 1.9.x Inside WildFly 8

If you're deploying RESTEasy with WildFly 8, there's nothing you need to do except to make sure you've updated your installation with the latest and greatest RESTEasy. See the Installation/Configuration section of this documentation for more details.

24.3. Using Jackson 2 Outside of WildFly

If you're deploying RESTEasy outside of WildFly, add the RESTEasy Jackson provder to your WAR pom.xml build:

```
<dependency>
   <groupId>org.jboss.resteasy</groupId>
   <artifactId>resteasy-jackson2-provider</artifactId>
   <version>${version.resteasy}</version>
</dependency>
```

24.4. Using Jackson 2 Inside WildFly 9 and above

If you're deploying RESTEasy with WildFly 9 or above, there's nothing you need to do except to make sure you've updated your installation with the latest and greatest RESTEasy. See the Installation/Configuration section of this documentation for more details.

24.5. Additional RESTEasy Specifics

The first extra piece that RESTEasy added to the integration was to support "application/*+json". Jackson would only accept "application/json" and "text/json" as valid media types. This allows you to create json-based media types and still let Jackson marshal things for you. For example:

```
@Path("/customers")
public class MyService {

    @GET
    @Produces("application/vnd.customer+json")
    public Customer[] getCustomers() {}
}
```

24.6. JSONP Support

If you're using Jackson, RESTEasy has JSONP [http://en.wikipedia.org/wiki/JSONP] that you can turn on by adding the provider `org.jboss.resteasy.plugins.providers.jackson.JacksonJsonpInterceptor` (Jackson2JsonpInterceptor if you're using the Jackson2 provider) to your deployments. If the media type of the response is json and a callback query parameter is given, the response will be a javascript snippet with a method call of the method defined by the callback parameter. For example:

```
GET /resources/stuff?callback=processStuffResponse
```

will produce this response:

```
processStuffResponse(<nomal JSON body>)
```

This supports the default behavior of jQuery [http://api.jquery.com/jQuery.ajax/]. To enable JacksonJsonpInterceptor in WildFly, you need to import annotations from `org.jboss.resteasy.resteasy-jackson-provider` module using jboss-deployment-structure.xml:

```
<jboss-deployment-structure>
  <deployment>
    <dependencies>
      <module name="org.jboss.resteasy.resteasy-jackson-provider" annotations="true"/>
    </dependencies>
  </deployment>
</jboss-deployment-structure>
```

You can change the name of the callback parameter by setting the callbackQueryParameter property.

JacksonJsonpInterceptor can wrap the response into a try-catch block:

```
try{processStuffResponse(<normal JSON body>)}catch(e){}
```

You can enable this feature by setting the resteasy.jsonp.silent property to true

Note. Because JSONP can be used in **Cross Site Scripting Inclusion (XSSI) attacks**, `Jackson2JsonpInterceptor` is disabled by default. Two steps are necessary to enable it:

1. As noted above, `Jackson2JsonpInterceptor` must be included in the deployment. For example, a service file META-INF/services/javax.ws.rs.ext.Providers with the line

   ```
   org.jboss.resteasy.plugins.providers.jackson.Jackson2JsonpInterceptor
   ```

   may be included on the classpath

2. Also, the parameter parameter "resteasy.jsonp.enable" must be set to "true". [See Section 3.4, “Configuration” for more information about application configuration.]

24.7. Jackson JSON Decorator

If you are using the Jackson 2 provider, RESTEasy has provided a pretty-printing annotation similar with the one in Jakarta XML Binding provider:

```
org.jboss.resteasy.annotations.providers.jackson.Formatted
```

Here is an example:

```
```

```
@GET
@Produces("application/json")
@Path("/formatted/{id}")
@Formatted
public Product getFormattedProduct()
{
    return new Product(333, "robot");
}
```

As the example shown above, the @Formatted annotation will enable the underlying Jackson option "SerializationFeature.INDENT_OUTPUT".

24.8. JSON Filter Support

In Jackson2 , there is new feature JsonFilter [http://fasterxml.github.io/jackson-annotations/javadoc/2.13/com/fasterxml/jackson/annotation/JsonFilter.html] to allow annotate class with @JsonFilter and doing dynamic filtering. Here is an example which defines mapping from "nameFilter" to filter instances and filter bean properties when serilize to json format:

```
@JsonFilter(value="nameFilter")
public class Jackson2Product {
    protected String name;
    protected int id;
    public Jackson2Product() {
    }
    public Jackson2Product(final int id, final String name) {
        this.id = id;
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
}
```

@JsonFilter annotates resource class to filter out some property not to serialize in the json response. To map the filter id and instance we need to create another jackson class to add the id and filter instance map:

```
public class ObjectFilterModifier extends ObjectWriterModifier {
```

```
 public ObjectFilterModifier() {
 }
 @Override
 public ObjectWriter modify(EndpointConfigBase<?> endpoint,
   MultivaluedMap<String, Object> httpHeaders, Object valueToWrite,
   ObjectWriter w, JsonGenerator jg) throws IOException {

  FilterProvider filterProvider = new SimpleFilterProvider().addFilter(
    "nameFilter",
    SimpleBeanPropertyFilter.filterOutAllExcept("name"));
  return w.with(filterProvider);

 }
}
```

Here the method `modify()` will take care of filtering all properties except "name" property before write. To make this work, we need let RESTEasy know this mapping info. This can be easily set in a WriterInterceptor using Jackson's `ObjectWriterInjector`:

```
@Provider
public class JsonFilterWriteInterceptor implements WriterInterceptor{

 private ObjectFilterModifier modifier = new ObjectFilterModifier();
 @Override
 public void aroundWriteTo(WriterInterceptorContext context)
   throws IOException, WebApplicationException {
  //set a threadlocal modifier
     ObjectWriterInjector.set(modifier);
  context.proceed();
 }

}
```

Alternatively, Jackson's documentation suggest doing the same in a servlet filter; that however potentially leads to issues on RESTEasy, as the ObjectFilterModifier ends up being stored using a ThreadLocal object and there's no guarantee the same thread serving the servlet filter will be running the resource endpoint execution too. So, for the servlet filter scenario, RESTEasy offers its own injector that relies on the current thread context classloader for carrying over the specified modifier:

```
public class ObjectWriterModifierFilter implements Filter {
 private static ObjectFilterModifier modifier = new ObjectFilterModifier();

 @Override
 public void init(FilterConfig filterConfig) throws ServletException {
```

```
 }

 @Override
 public void doFilter(ServletRequest request, ServletResponse response,
   FilterChain chain) throws IOException, ServletException {
  ResteasyObjectWriterInjector.set(Thread.currentThread().getContextClassLoader(), modifier);
  chain.doFilter(request, response);
 }

 @Override
 public void destroy() {
 }

}
```

24.9. Polymorphic Typing deserialization

Due to numerous CVEs for a specific kind of Polymorphic Deserialization (see details in FasterXML Jackson documentation), starting from Jackson 2.10 users have a mean to allow only specified classes to be deserialized. RESTEasy enables this feature by default and allows controlling the contents of whitelist of allowed classes/packages.

Table 24.1.

Property	Description	
resteasy.jackson.deserialization.whitelist.allowIfBaseType.prefix	Whether to apply whitelisting to Base Type matcher that will allow all subtypes in cases where nominal base type's class name starts with specific prefix. "*" can be used for allowing any class.	
resteasy.jackson.deserialization.whitelist.allowIfSubType.prefix	Whether to apply whitelisting to Sub Type matcher that will allow specific subtype (regardless of declared base type) in cases where subclass name starts with specified prefix. "*" can be used for allowing any class.	

Chapter 25. JSON Support via Jakarta EE JSON-P API

No, this is not the JSONP you are thinking of! JSON-P is a Jakarta EE parsing API. Horrible name for a new JSON parsing API! What were they thinking? Anyways, RESTEasy has a provider for it. If you are using WildFly, it is required by Jakarta EE so you will have it automatically bundled. Otherwise, use this maven dependency.

```
<dependency>
   <groupId>org.jboss.resteasy</groupId>
   <artifactId>resteasy-json-p-provider</artifactId>
   <version>5.0.10.Final</version>
</dependency>
```

It has built in support for JsonObject, JsonArray, and JsonStructure as request or response entities. It should not conflict with Jackson if you have that in your path too.

Chapter 26. Multipart Providers

RESTEasy has rich support for the "multipart/*" and "multipart/form-data" mime types. The multipart mime format is used to pass lists of content bodies. Multiple content bodies are embedded in one message. "multipart/form-data" is often found in web application HTML Form documents and is generally used to upload files. The form-data format is the same as other multipart formats, except that each inlined piece of content has a name associated with it.

RESTEasy provides a custom API for reading and writing multipart types as well as marshalling arbitrary List (for any multipart type) and Map (multipart/form-data only) objects

Classes `MultipartInput` and `MultipartOutput` provides read and write support for mime type "multipart/mixed" messages respectively. They provide for multiple part messages, in which one or more different sets of data are combined in a single body.

`MultipartRelatedInput` and `MultipartRelatedOutput` classes provide read and write support for mime type "multipart/related" messages. These are messages that contain multiple body parts that are inter-related.

`MultipartFormDataInput` and `MultipartFormDataOutput` classes provide read and write support for mine type "multipart/form-data". This type is used when returning a set of values as the the result of a user filling out a form or for uploading files.

26.1. Multipart/mixed

26.1.1. Writing multipart/mixed messages

`MultipartOutput` provides a set of addPart methods for registering message content and specifying special marshalling requirements. In all cases the addPart methods require an input parameter, Object and a MediaType that declares the mime type of the object. Sometimes you may have an object in which marshalling is sensitive to generic type metadata. In such cases, use an addPart method in which you declare the GenericType of the entity Object. Perhaps a file will be passed as content and it will require UTF-8 encoding. Setting input parameter, utf8Encode to `true` will indicate to RESTEasy to process the filename according to the character set and language encoding rules of `rfc5987`. This flag is only processed when mime type "multipart/form-data" is specified.

`MultipartOutput` automatically generates a unique message boundary identifier when it is created. Method setBoundary is provided in case you wish to declare a different identifier.

```
public class MultipartOutput
{
   public OutputPart addPart(Object entity, MediaType mediaType);
   public OutputPart addPart(Object entity, MediaType mediaType,
        String filename);
```

```
   public OutputPart addPart(Object entity, MediaType mediaType,
        String filename, boolean utf8Encode);
   public OutputPart addPart(Object entity, GenericType<?> type,
        MediaType mediaType);
   public OutputPart addPart(Object entity, GenericType<?> type,
        MediaType mediaType, String filename);
   public OutputPart addPart(Object entity, GenericType<?> type,
        MediaType mediaType, String filename, boolean utf8Encode);
   public OutputPart addPart(Object entity, Class<?> type, Type genericType,
        MediaType mediaType);
   public OutputPart addPart(Object entity, Class<?> type, Type genericType,
        MediaType mediaType, String filename);
   public OutputPart addPart(Object entity, Class<?> type, Type genericType,
        MediaType mediaType, String filename, boolean utf8Encode);
   public List<OutputPart> getParts();
   public String getBoundary();
   public void setBoundary(String boundary);
}
```

Each message part registered with `MultipartOutput` is represented by an `OutputPart` object. Class `MultipartOutput` generates an `OutputPart` object for each addPart method call.

```
public class OutputPart {
   public OutputPart(final Object entity, final Class<?> type,
        final Type genericType, final MediaType mediaType);
   public OutputPart(final Object entity, final Class<?> type,
        final Type genericType, final MediaType mediaType,
        final String filename);
   public OutputPart(final Object entity, final Class<?> type,
        final Type genericType, final MediaType mediaType,
        final String filename, final boolean utf8Encode);
   public MultivaluedMap<String, Object> getHeaders();
   public Object getEntity();
   public Class<?> getType();
   public Type getGenericType();
   public MediaType getMediaType();
   public String getFilename();
   public boolean isUtf8Encode();
}
```

26.1.2. Reading multipart/mixed messages

`MultipartInput` and `InputPart` are interface classes that provide access to multipart/mixed message data. RESTEasy provides an implementation of these classes. They perform the work to retrieve message data.

```
package org.jboss.resteasy.plugins.providers.multipart;

import java.util.List;
```

```
public interface MultipartInput {
   List<InputPart> getParts();
   String getPreamble();
   /**
    * Call this method to delete any temporary files created from unmarshalling
    * this multipart message
    * Otherwise they will be deleted on Garbage Collection or JVM exit.
    */
   void close();
}
```

```
package org.jboss.resteasy.plugins.providers.multipart;

import javax.ws.rs.core.GenericType;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.MultivaluedMap;
import java.io.IOException;
import java.lang.reflect.Type;

/**
 * Represents one part of a multipart message.
 */
public interface InputPart {
   /**
    * If no content-type header is sent in a multipart message part
    * "text/plain; charset=ISO-8859-1" is assumed.
    *
    * This can be overwritten by setting a different String value in
    * {@link org.jboss.resteasy.spi.HttpRequest#setAttribute(String, Object)}
    * with this ("resteasy.provider.multipart.inputpart.defaultContentType")
    * String as key. It should be done in a
    * {@link javax.ws.rs.container.ContainerRequestFilter}.
    */
   String DEFAULT_CONTENT_TYPE_PROPERTY =
    "resteasy.provider.multipart.inputpart.defaultContentType";

   /**
    * If there is a content-type header without a charset parameter,
    * charset=US-ASCII is assumed.
    *
    * This can be overwritten by setting a different String value in
    * {@link org.jboss.resteasy.spi.HttpRequest#setAttribute(String, Object)}
    * with this ("resteasy.provider.multipart.inputpart.defaultCharset")
    * String as key. It should be done in a
    * {@link javax.ws.rs.container.ContainerRequestFilter}.
    */
   String DEFAULT_CHARSET_PROPERTY =
    "resteasy.provider.multipart.inputpart.defaultCharset";

   /**
    * @return headers of this part
    */
   MultivaluedMap<String, String> getHeaders();
   String getBodyAsString() throws IOException;
```

```
   <T> T getBody(Class<T> type, Type genericType) throws IOException;
   <T> T getBody(GenericType<T> type) throws IOException;

   /**
    * @return "Content-Type" of this part
    */
   MediaType getMediaType();

   /**
    * @return true if the Content-Type was resolved from the message, false if
    *         it was resolved from the server default
    */
   boolean isContentTypeFromMessage();

   /**
    * Change the media type of the body part before you extract it.
    * Useful for specifying a charset.
    * @param mediaType media type
    */
   void setMediaType(MediaType mediaType);
}}
```

26.1.3. Simple multipart/mixed message example

The following example shows how to read and write a simple multipart/mixed message.

The data to be transfered is a very simple class, Soup.

```
package org.jboss.resteasy.test.providers.multipart.resource;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlElement;

@XmlRootElement(name = "soup")
@XmlAccessorType(XmlAccessType.FIELD)
public class Soup {
    @XmlElement
    private String id;

    public Soup(){}
    public Soup(final String id){this.id = id;}
    public String getId(){return id;}
}
```

This code fragment creates a multipart/mixed message passing Soup information using class, `MultipartOutput`.

```
```

```
MultipartOutput multipartOutput = new MultipartOutput();
multipartOutput.addPart(new Soup("Chicken Noodle"),
      MediaType.APPLICATION_XML_TYPE);
multipartOutput.addPart(new Soup("Vegetable"),
      MediaType.APPLICATION_XML_TYPE);
multipartOutput.addPart("Granny's Soups", MediaType.TEXT_PLAIN_TYPE);
```

This code fragment uses class `MultipartInput` to extract the Soup information provided by multipartOutput above.

```
// MultipartInput multipartInput, the entity returned in the client in a
// Response object or the input value of an endpoint method parameter.
for (InputPart inputPart : multipartInput.getParts()) {
    if (MediaType.APPLICATION_XML_TYPE.equals(inputPart.getMediaType())) {
        Soup c = inputPart.getBody(Soup.class, null);
        String name = c.getId();
    } else {
        String s = inputPart.getBody(String.class, null);
    }
}
```

Returning a multipart/mixed message from an endpoint can be done in two ways. `MultipartOutput` can be returned as the method's return object or as an entity in a `Response` object.

```
@GET
@Path("soups/obj")
@Produces("multipart/mixed")
public MultipartOutput soupsObj() {
    return multipartOutput;
}

@GET
@Path("soups/resp")
@Produces("multipart/mixed")
public Response soupsResp() {
  return Response.ok(multipartOutput, MediaType.valueOf("multipart/mixed"))
                .build();
}
```

There is no difference in the way a client retrieves the message from the endpoint. It is done as follows.

```
```

```
      ResteasyClient client = (ResteasyClient)ClientBuilder.newClient();
      ResteasyWebTarget target = client.target(THE_URL);
      Response response = target.request().get();
      MultipartInput multipartInput = response.readEntity(MultipartInput.class);

      for (InputPart inputPart : multipartInput.getParts()) {
          if (MediaType.APPLICATION_XML_TYPE.equals(inputPart.getMediaType())) {
              Soup c = inputPart.getBody(Soup.class, null);
              String name = c.getId();
          } else {
              String s = inputPart.getBody(String.class, null);
          }
      }

      client.close();
```

A client sends the message, multipartOutput, to an endpoint as an entity object in an HTTP method call in this code fragment.

```
        ResteasyClient client = (ResteasyClient)ClientBuilder.newClient();
        ResteasyWebTarget target = client.target(SOME_URL + "/register/soups");
        Entity<MultipartOutput> entity = Entity.entity(multipartOutput,
                new MediaType("multipart", "mixed"));
        Response response = target.request().post(entity);
```

Here is the endpoint receiving the message and extracting the contents.

```
@POST
@Consumes("multipart/mixed")
@Path("register/soups")
 public void registerSoups(MultipartInput multipartInput) throws IOException {

      for (InputPart inputPart : multipartInput.getParts()) {
         if (MediaType.APPLICATION_XML_TYPE.equals(inputPart.getMediaType())) {
                Soup c = inputPart.getBody(Soup.class, null);
                String name = c.getId();
         } else {
                String s = inputPart.getBody(String.class, null);
         }
      }
 }
```

26.1.4. Multipart/mixed message with GenericType example

This example shows how to read and write a multipart/mixed message whose content consists of a generic type, in this case a List<Soup>. The `MultipartOutput` and `MultipartIntput` methods that use `GenericType` parameters are used.

The multipart/mixed message is created using `MultipartOutput` as follows.

```
MultipartOutput multipartOutput = new MultipartOutput();
List<Soup> soupList = new ArrayList<Soup>();
soupList.add(new Soup("Chicken Noodle"));
soupList.add(new Soup("Vegetable"));
multipartOutput.addPart(soupList, new GenericType<List<Soup>>(){},
       MediaType.APPLICATION_XML_TYPE );
multipartOutput.addPart("Granny's Soups", MediaType.TEXT_PLAIN_TYPE);
```

The message data is extracted with `MultipartInput`. Note there are two `MultipartInput` getBody methods that can be used to retrieve data specifying `GenericType`. This code fragment uses the second one but shows the first one in comments.

```
<T> T getBody(Class<T> type, Type genericType) throws IOException;
<T> T getBody(GenericType<T> type) throws IOException;
```

```
// MultipartInput multipartInput, the entity returned in the client in a
// Response object or the input value of an endpoint method parameter.
GenericType<List<Soup>> gType = new GenericType<List<Soup>>(){};

for (InputPart inputPart : multipartInput.getParts()) {
   if (MediaType.APPLICATION_XML_TYPE.equals(inputPart.getMediaType())) {
      List<Soup> c = inputPart.getBody(gType);
   // List<Soup> c = inputPart.getBody(gType.getRawType(), gType.getType());
   } else {
      String s = inputPart.getBody(String.class, null);;
   }
}
```

26.1.5. java.util.List with multipart/mixed data example

When a set of message parts are uniform they do not need to be written using `MultipartOutput` or read with `MultipartInput`. They can be sent and received as a `List`. RESTEasy performs the necessary work to read and write the message data.

For this example the data to be transmitted is class, `ContextProvidersCustomer`

```
package org.jboss.resteasy.test.providers.multipart.resource;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement(name = "customer")
@XmlAccessorType(XmlAccessType.FIELD)
public class ContextProvidersCustomer {
   @XmlElement
   private String name;

   public ContextProvidersCustomer() { }
   public ContextProvidersCustomer(final String name) {
      this.name = name;
  }
   public String getName() { return name;}
}
```

In this code fragment the client creates and sends of list `ContextProvidersCustomers`.

```
  List<ContextProvidersCustomer> customers =
        new ArrayList<ContextProvidersCustomer>();
  customers.add(new ContextProvidersCustomer("Bill"));
  customers.add(new ContextProvidersCustomer("Bob"));

  Entity<ContextProvidersCustomer> entity = Entity.entity(customers,
    new MediaType("multipart", "mixed"));

  Client client = ClientBuilder.newClient();
  WebTarget target = client.target(SOME_URL);
  Response response = target.request().post(entity);
```

The endpoint receives the list, alters the contents and returns a new list.

```
   @POST
   @Consumes("multipart/mixed")
   @Produces(MediaType.APPLICATION_XML)
   @Path("post/list")
   public List<ContextProvidersName> postList(
         List<ContextProvidersCustomer> customers) throws IOException {

      List<ContextProvidersName> names = new ArrayList<ContextProvidersName>();

      for (ContextProvidersCustomer customer : customers) {
         names.add(new ContextProvidersName("Hello " + customer.getName()));
      }
      return names;
   }
```

The client receives the altered message data and processes it.

```
    Response response = target.request().post(entity);
    List<ContextProvidersCustomer> rtnList =
      response.readEntity(new GenericType<List<ContextProvidersCustomer>>(){});
        :
        :
```

26.2. Multipart/related

The Multipart/Related mime type is intended for compound objects consisting of several inter-related body parts, (RFC2387). There is a root or start part. All other parts are referenced from the root part. Each part has a unique id. The type and the id of the start part is presented in parameters in the message content-type header.

26.2.1. Writing multipart/related messages

RESTEasy provides class `MultipartRelatedOutput` to assist the user in specifying the required information and generating a properly formatted message. `MultipartRelatedOutput` is a subclass of `MultipartOutput`.

```
package org.jboss.resteasy.plugins.providers.multipart;

import javax.ws.rs.core.MediaType;

public class MultipartRelatedOutput extends MultipartOutput {
   private String startInfo;

   /**
```

```
    * The part used as the root.
    */
   public OutputPart getRootPart();

   /**
    * entity  object representing the part's body
    * mediaType Content-Type of the part
    * contentId  Content-ID to be used as identification for the current
    *            part, optional, if null one will be generated
    * contentTransferEncoding
    *            value used for the Content-Transfer-Encoding header
    *            field of the part. It's optional, if you don't want to set
    *            this pass null. Example values are: "7bit",
    *            "quoted-printable", "base64", "8bit", "binary"
    */
   public OutputPart addPart(Object entity, MediaType mediaType,
         String contentId, String contentTransferEncoding);

   /**
    * start-info parameter of the Content-Type.  An optional parameter.
    * As described in RFC2387, section 3.3. The Start-Info Parameter
    */
   public String getStartInfo();
}
```

26.2.2. Reading multipart/related messages

`MultipartRelatedInput` is an interface class that provides access to multipart/related message data. It is a subclass of `MultipartInput`. RESTEasy provides an implementation of this class. It performs the work to retrieve message data.

```
package org.jboss.resteasy.plugins.providers.multipart;

import javax.ws.rs.core.MediaType;

public class MultipartRelatedOutput extends MultipartOutput {
   private String startInfo;

   /**
    * The part used as the root.
    */
   public OutputPart getRootPart();

   /**
    * entity  object representing the part's body
    * mediaType Content-Type of the part
    * contentId  Content-ID to be used as identification for the current
    *            part, optional, if null one will be generated
    * contentTransferEncoding
    *            value used for the Content-Transfer-Encoding header
    *            field of the part. It's optional, if you don't want to set
    *            this pass null. Example values are: "7bit",
```

```
     *            "quoted-printable", "base64", "8bit", "binary"
     */
    public OutputPart addPart(Object entity, MediaType mediaType,
          String contentId, String contentTransferEncoding);

    /**
     * start-info parameter of the Content-Type.  An optional parameter.
     * As described in RFC2387, section 3.3. The Start-Info Parameter
     */
    public String getStartInfo();
}
```

26.2.3. Multipart/related message example

The client in this example creates a multipart/related message, POSTs it to the endpoint and processes the multipart/related message returned by the endpoint.

```
MultipartRelatedOutput mRelatedOutput = new MultipartRelatedOutput();
mRelatedOutput.setStartInfo("text/html");
mRelatedOutput.addPart("Bill", new MediaType("image", "png"), "bill", "binary");
mRelatedOutput.addPart("Bob", new MediaType("image", "png"), "bob", "binary");

Entity<MultipartRelatedOutput> entity = Entity.entity(mRelatedOutput,
    new MediaType("multipart", "related"));

Client client = ClientBuilder.newClient();
WebTarget target = client.target(SOME_URL);
Response response = target.request().post(entity);

MultipartRelatedInput result = response.readEntity(
      MultipartRelatedInput.class);
Map<String, InputPart> map = result.getRelatedMap();
Set<String> keys = map.keySet();
boolean a = keys.contains("Bill");
boolean b = keys.contains("Bob");
for (InputPart inputPart : map.values()) {
    String alterName = inputPart.getBody(String.class, null);
}
```

Here is the endpoint the client above is calling.

```
@POST
@Consumes("multipart/related")
@Produces("multipart/related")
@Path("post/related")
public MultipartRelatedOutput postRelated(MultipartRelatedInput input)
        throws IOException {
```

```
  MultipartRelatedOutput rtnMRelatedOutput = new MultipartRelatedOutput();
        rtnMRelatedOutput.setStartInfo("text/html");

  for (Iterator<InputPart> it = input.getParts().iterator(); it.hasNext(); ) {
      InputPart part = it.next();
      String name = part.getBody(String.class, null);
      rtnMRelatedOutput.addPart("Hello " + name,
                    new MediaType("image", "png"), name, null);
  }
  return rtnMRelatedOutput;
}
```

26.2.4. XML-binary Optimized Packaging (XOP)

RESTEasy supports XOP messages packaged as multipart/related messages (http://www.w3.org/TR/xop10/). A Jakarta XML Binding annotated POJO that also holds binary content can be transmitted using XOP. XOP allows the binary data to skip going through the XML serializer because binary data can be serialized differently from text and this can result in faster transport time.

RESTEasy requires annotation @XopWithMultipartRelated to be placed on any endpoint method that returns an object that is to be to be processed with XOP and on any endpoint input parameter that is to be processed by XOP.

RESTEasy highly recommends, if you know the exact mime type of the POJO's binary data, tag the field with annotation @XmlMimeType. This annotation tells Jakarta XML Binding the mime type of the binary content, however this is not required in order to do XOP packaging.

26.2.5. @XopWithMultipartRelated return object example

The data to be transmitted is class, `ContextProvidersXop`. Note that field `bytes` is identified as an application/octet-stream mime type using annotation @XmlMimeType

```
package org.jboss.resteasy.test.providers.multipart.resource;

import javax.ws.rs.core.MediaType;
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlMimeType;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class ContextProvidersXop {

   @XmlMimeType(MediaType.APPLICATION_OCTET_STREAM)
   private byte[] bytes;
```

```
   public ContextProvidersXop(final byte[] bytes) {
      this.bytes = bytes;
   }

   public ContextProvidersXop() {}
   public byte[] getBytes() {return bytes;}
   public void setBytes(byte[] bytes) {this.bytes = bytes;}
}
```

The endpoint returns an instance of `ContextProvidersXop`. Note annotation @XopWithMultipartRelated declared on the method because we want the return object to use XOP packaging.

```
   @GET
   @Path("get/xop")
   @Produces("multipart/related")
   @XopWithMultipartRelated
   public ContextProvidersXop getXop() {
      return new ContextProvidersXop("goodbye world".getBytes());
   }
```

The client retreives the data as follows

```
Client client = ClientBuilder.newClient();
WebTarget target = client.target(SOME_URL);
Response response = target.request().get();
ContextProvidersXo entity = response.readEntity(ContextProvidersXop.class);
client.close();
```

26.2.6. @XopWithMultipartRelated input parameter example

Here is an endpoint that has an input parameter that is transmitted as an XOP package. Note the @XopWithMultipartRelated annotation on input parameter xop.

```
   @POST
   @Path("post/xop")
   @Consumes("multipart/related")
   public String postXop(@XopWithMultipartRelated ContextProvidersXop xop) {
      return new String(xop.getBytes());
   }
```

This client is sending the data to the endpoint above.

```
ContextProvidersXop xop = new ContextProvidersXop("hello world".getBytes());
Entity<ContextProvidersXop> entity = Entity.entity(xop,
    new MediaType("multipart", "related"));

Client client = ClientBuilder.newClient();
WebTarget target = client.target(SOME_URL);
Response response = target.request().post(entity);
```

26.3. Multipart/form-data

The MultiPart/Form-Data mime type is used in sending form data (rfc2388). It can include data generated by user input, information that is typed, or included from files that the user has selected. "multipart/form-data" is often found in web application HTML Form documents and is generally used to upload files. The form-data format is the same as other multi-part formats, except that each inlined piece of content has a name associated with it.

26.3.1. Writing multipart/form-data messages

Form data consists of key/value pairs. RESTEasy provides class `MultipartFormDataOutput` to assist the user in specifying the required information and generating a properly formatted message. It is a subclass of `MultipartOutput`. And as with multipart/mixed data sometimes there may be marshalling which is sensitive to generic type metadata, in those cases use the methods containing input parameter GenericType.

```
package org.jboss.resteasy.plugins.providers.multipart;

public class MultipartFormDataOutput extends MultipartOutput
{
    public OutputPart addFormData(String key, Object entity,
            MediaType mediaType)
    public OutputPart addFormData(String key, Object entity, GenericType type,
            MediaType mediaType)
    public OutputPart addFormData(String key, Object entity, Class type,
            Type genericType, MediaType mediaType)
    public Map<String, OutputPart> getFormData()
    public Map<String, List<OutputPart>> getFormDataMap()
}
```

26.3.2. Reading multipart/form-data messages

`MultipartFormDataInput` is an interface class that provides access to multipart/form-data message data. It is a subclass of `MultipartInput`. RESTEasy provides an implementation of this class. It performs the work to retrieve message data.

```
package org.jboss.resteasy.plugins.providers.multipart;

import java.io.IOException;
import java.lang.reflect.Type;
import java.util.List;
import java.util.Map;

import javax.ws.rs.core.GenericType;

public interface MultipartFormDataInput extends MultipartInput {
   /**
    * @return A parameter map containing a list of values per name.
    */
   Map<String, List<InputPart>> getFormDataMap();
   <T> T getFormDataPart(String key, Class<T> rawType, Type genericType)
         throws IOException;
   <T> T getFormDataPart(String key, GenericType<T> type) throws IOException;
}
```

26.3.3. Simple multipart/form-data message example

The following example show how to read and write a simple multipart/form-data message.

The multipart/mixed message is created on the clientside using the `MultipartFormDataOutput` object. One piece of form data to be transfered is a very simple class, `ContextProvidersName`.

```
package org.jboss.resteasy.test.providers.multipart.resource;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement(name = "name")
@XmlAccessorType(XmlAccessType.FIELD)
public class ContextProvidersName {
   @XmlElement
   private String name;

   public ContextProvidersName() {}
   public ContextProvidersName(final String name) {this.name = name;}
   public String getName() {return name;}
}
```

The client creates and sends the message as follows:

```
MultipartFormDataOutput output = new MultipartFormDataOutput();
output.addFormData("bill", new ContextProvidersCustomer("Bill"),
   MediaType.APPLICATION_XML_TYPE);
output.addFormData("bob", "Bob", MediaType.TEXT_PLAIN_TYPE);

Entity<MultipartFormDataOutput> entity = Entity.entity(output,
    new MediaType("multipart", "related"));

Client client = ClientBuilder.newClient();
WebTarget target = client.target(SOME_URL);
Response response = target.request().post(entity);
```

The endpoint receives the message and processes it.

```
@POST
@Consumes("multipart/form-data")
@Produces(MediaType.APPLICATION_XML)
@Path("post/form")
public Response postForm(MultipartFormDataInput input)
      throws IOException {

   Map<String, List<InputPart>> map = input.getFormDataMap();
   List<ContextProvidersName> names = new ArrayList<ContextProvidersName>();

   for (Iterator<String> it = map.keySet().iterator(); it.hasNext(); ) {
      String key = it.next();
      InputPart inputPart = map.get(key).iterator().next();
      if (MediaType.APPLICATION_XML_TYPE.equals(inputPart.getMediaType())) {
         names.add(new ContextProvidersName(inputPart.getBody(
               ContextProvidersCustomer.class, null).getName()));
      } else {
         names.add(new ContextProvidersName(inputPart.getBody(
               String.class, null)));
      }
   }
   return Response.ok().build();
}
```

26.3.4. java.util.Map with multipart/form-data

When the data of a multipart/form-data message is uniform it does not need to be written in a `MultipartFormDataOutput` object. It can be sent and received as a `java.util.Map` object. RESTEasy performs the necessary work to read and write the message data, however the Map object must declare the type it is unmarshalling via the generic parameters in the Map type declaration.

Here is an example of a client creating and sending a multipart/form-data message.

```
    Map<String, ContextProvidersCustomer> customers =
            new HashMap<String, ContextProvidersCustomer>();
    customers.put("bill", new ContextProvidersCustomer("Bill"));
    customers.put("bob", new ContextProvidersCustomer("Bob"));

    Entity<Map<String, ContextProvidersCustomer>> entity =
      Entity.entity(customers, new MediaType("multipart", "form-data"));

    Client client = ClientBuilder.newClient();
    WebTarget target = client.target(SOME_URL);
    Response response = target.request().post(entity)
```

This is the endpoint the client above is calling. It receives the message and processes it.

```
@POST
@Consumes("multipart/form-data")
@Produces(MediaType.APPLICATION_XML)
@Path("post/map")
public Response postMap(Map<String, ContextProvidersCustomer> customers)
        throws IOException {

  List<ContextProvidersName> names = new ArrayList<ContextProvidersName>();
  for (Iterator<String> it = customers.keySet().iterator(); it.hasNext(); ) {
      String key = it.next();
      ContextProvidersCustomer customer = customers.get(key);
      names.add(new ContextProvidersName(key + ":" + customer.getName()));
  }
  return Response.ok().build();
}
```

26.3.5. Multipart/form-data java.util.Map as method return type

A `java.util.Map` object representing a multipart/form-data message can be returned from an endpoint as long as the message data is uniform, however the endpoint method MUST be an-

notated with @PartType which declares the media type of the Map entries and the Map object must declare the type it is unmarshalling via the generic parameters in the Map type declaration. RESTEasy requires this information so it can generate the message properly.

Here is an example of an endpoint returning a Map of `ContextProvidersCustomer` to the client.

```
@GET
@Produces("multipart/form-data")
@PartType("application/xml")
@Path("get/map")
public Map<String, ContextProvidersCustomer> getMap() {

   Map<String, ContextProvidersCustomer> map =
       new HashMap<String, ContextProvidersCustomer>();
   map.put("bill", new ContextProvidersCustomer("Bill"));
   map.put("bob", new ContextProvidersCustomer("Bob"));
   return map;
}
```

The client would retrieve the data as follows.

```
   Client client = ClientBuilder.newClient();
   WebTarget target = client.target(SOME_URL);
   Response response = target.request().get();
   MultipartFormDataInput entity = response.readEntity(
         MultipartFormDataInput.class);
   client.close();

   ContextProvidersCustomer bill = entity.getFormDataPart("bill",
         ContextProvidersCustomer.class, null);
   ContextProvidersCustomer bob = entity.getFormDataPart("bob",
         ContextProvidersCustomer.class, null);
```

26.3.6. @MultipartForm and POJOs

If you have an exact knowledge of your multipart/form-data packets, you can map them to and from a POJO class using the annotation `@org.jboss.resteasy.annotations.providers.multipart.MultipartForm` and the Jakarta RESTful Web Services `@FormParam` annotation. Simply define a POJO with at least a default constructor and annotate its fields and/or properties with `@FormParams`. These `@FormParams` must also be annotated with `@org.jboss.resteasy.annotations.providers.multipart.PartType` if you are doing output. For example:

```
            public class CustomerProblemForm {
    @FormParam("customer")
    @PartType("application/xml")
    private Customer customer;

    @FormParam("problem")
    @PartType("text/plain")
    private String problem;

    public Customer getCustomer() { return customer; }
    public void setCustomer(Customer cust) { this.customer = cust; }
    public String getProblem() { return problem; }
    public void setProblem(String problem) { this.problem = problem; }
}
```

After defining the POJO class you can use it to represent multipart/form-data. Here's an example of sending a `CustomerProblemForm` using the RESTEasy client framework:

```
@Path("portal")
public interface CustomerPortal {

   @Path("issues/{id}")
   @Consumes("multipart/form-data")
   @PUT
   public void putProblem(@MultipartForm CustomerProblemForm,
                          @PathParam("id") int id) {
      CustomerPortal portal = ProxyFactory.create(
            CustomerPortal.class, "http://example.com");
      CustomerProblemForm form = new CustomerProblemForm();
      form.setCustomer(...);
      form.setProblem(...);

      portal.putProblem(form, 333);
   }
}
```

Note that the `@MultipartForm` annotation was used to tell RESTEasy that the object has a `@FormParam` and that it should be marshalled from that. You can also use the same object to receive multipart data. Here is an example of the server side counterpart of our customer portal.

```
            @Path("portal")
public class CustomerPortalServer {

    @Path("issues/{id})
    @Consumes("multipart/form-data")
    @PUT
    public void putIssue(@MultipartForm CustoemrProblemForm,
                         @PathParam("id") int id) {
```

```
        ... write to database...
    }
}
```

In addition to the XML data format, JSON formatted data can be used to represent POJO classes. To achieve this goal, plug in a JSON provider into your project. For example, add the RESTEasy Jackson2 Provider into your project's dependency scope:

```
<dependency>
    <groupId>org.jboss.resteasy</groupId>
    <artifactId>resteasy-jackson2-provider</artifactId>
    <version>${resteasy.ver}</version>
</dependency>
```

Now you can write an ordinary POJO class, which Jackson2 will automatically serialize/deserialize into JSON format:

```
public class JsonUser {
   private String name;

   public JsonUser() {}
   public JsonUser(final String name) { this.name = name; }
   public String getName() { return name; }
   public void setName(String name) { this.name = name; }
}
```

The resource class can be written like this:

```
import org.jboss.resteasy.annotations.providers.multipart.MultipartForm;
import org.jboss.resteasy.annotations.providers.multipart.PartType;

import javax.ws.rs.Consumes;
import javax.ws.rs.FormParam;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;

@Path("/")
public class JsonFormResource {

    public JsonFormResource() {
    }

    public static class Form {
```

```
    @FormParam("user")
    @PartType("application/json")
    private JsonUser user;

    public Form() {
    }

    public Form(final JsonUser user) {
    this.user = user;
    }

    public JsonUser getUser() {
      return user;
    }
}

    @PUT
    @Path("form/class")
    @Consumes("multipart/form-data")
    public String putMultipartForm(@MultipartForm Form form) {
         return form.getUser().getName();
    }
}
```

As the code shown above, you can see the PartType of JsonUser is marked as "application/json", and it's included in the "@MultipartForm Form" class instance.

To send the request to the resource method, you need to send JSON formatted data that is corresponding with the JsonUser class. The easiest way to do this is to use a proxy class that has the same definition of the resource class. Here is the sample code of the proxy class that is corresponding with the JsonFormResource class:

```
import org.jboss.resteasy.annotations.providers.multipart.MultipartForm;

import javax.ws.rs.Consumes;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;

@Path("/")
public interface JsonForm {

@PUT
@Path("form/class")
@Consumes("multipart/form-data")
  String putMultipartForm(@MultipartForm JsonFormResource.Form form);
}
```

And then use the proxy class above to send the request to the resource method correctly. Here is the sample code:

```
ResteasyClient client = (ResteasyClient)ClientBuilder.newClient();
...
JsonForm proxy = client.target("your_request_url_address")
                       .proxy(JsonForm.class);
String name = proxy.putMultipartForm(new JsonFormResource
                  .Form(new JsonUser("bill")));
...
```

If your client side has the Jackson2 provider included, the request will be marshaled correctly. The JsonUser data will be converted into JSON format and sent to the server side. You can also use hand-crafted JSON data as your request and send it to server side, but you have to make sure the request data is in the correct form.

26.4. Note about multipart parsing and working with other frameworks

There are a lot of frameworks doing multipart parsing automatically with the help of filters and interceptors, like `org.jboss.seam.web.MultipartFilter` in Seam and `org.springframework.web.multipart.MultipartResolver` in Spring, however these incoming multipart request stream can be parsed only once. RESTEasy users working with multipart should make sure that nothing parses the stream before RESTEasy gets it.

26.5. Overwriting the default fallback content type for multipart messages

By default if no Content-Type header is present in a part, `"text/plain; charset=us-ascii"` is used as the fallback. This is the value defined by the MIME RFC. However some web clients, like most, if not all, web browsers, do not send Content-Type headers for all fields in a multipart/form-data request. They send them only for the file parts. This can cause character encoding and unmarshalling errors on the server side. To correct this there is an option to define an other, non-rfc compliant fallback value. This can be done dynamically per request with the filter facility of Jakarta RESTful Web Services 3.0. In the following example we will set `"*/*; charset=UTF-8"` as the new default fallback:

```
import org.jboss.resteasy.plugins.providers.multipart.InputPart;

@Provider
public class InputPartDefaultCharsetOverwriteContentTypeCharsetUTF8
   implements ContainerRequestFilter {

   @Override
   public void filter(ContainerRequestContext requestContext) throws IOException
```

```
   {
     requestContext.setProperty(InputPart.DEFAULT_CONTENT_TYPE_PROPERTY, "*/*; charset=UTF-8");
   }
}
```

26.6. Overwriting the content type for multipart messages

Using attribute, InputPart.DEFAULT_CONTENT_TYPE_PROPERTY and a filter enables the setting of a default Content-Type, It is also possible to override the Content-Type by setting a different media type with method `InputPart.setMediaType()`. Here is an example:

```
@POST
@Path("query")
@Consumes(MediaType.MULTIPART_FORM_DATA)
@Produces(MediaType.TEXT_PLAIN)
public Response setMediaType(MultipartInput input) throws IOException
{
    List<InputPart> parts = input.getParts();
    InputPart part = parts.get(0);
    part.setMediaType(MediaType.valueOf("application/foo+xml"));
    String s = part.getBody(String.class, null);
    ...
}
```

26.7. Overwriting the default fallback charset for multipart messages

Sometimes, a part may have a Content-Type header with no charset parameter. If the `InputPart.DEFAULT_CONTENT_TYPE_PROPERTY` property is set and the value has a charset parameter, that value will be appended to an existing Content-Type header that has no charset parameter. It is also possible to specify a default charset using the constant `InputPart.DEFAULT_CHARSET_PROPERTY` (actual value "resteasy.provider.multipart.inputpart.defaultCharset"):

```
import org.jboss.resteasy.plugins.providers.multipart.InputPart;

@Provider
public class InputPartDefaultCharsetOverwriteContentTypeCharsetUTF8
   implements ContainerRequestFilter {

   @Override
   public void filter(ContainerRequestContext requestContext) throws IOException
   {
```

```
        requestContext.setProperty(InputPart.DEFAULT_CHARSET_PROPERTY, "UTF-8");
    }
}
```

If both `InputPart.DEFAULT_CONTENT_TYPE_PROPERTY` and `InputPart.DEFAULT_CHARSET_PROPERTY` are set, then the value of `InputPart.DEFAULT_CHARSET_PROPERTY` will override any charset in the value of `InputPart.DEFAULT_CONTENT_TYPE_PROPERTY`.

Chapter 27. Jakarta RESTful Web Services 2.1 Additions

Jakarta RESTful Web Services 2.1 adds more asynchronous processing support in both the Client and the Server API. The specification adds a Reactive programming style to the Client side and Server-Sent Events (SSE) protocol support to both client and server.

27.1. `CompletionStage` support

The specification adds support for declaring asynchronous resource methods by returning a `CompletionStage` instead of using the `@Suspended` annotation.

Note

RESTEasy supports more reactive types than the specification.

27.2. Reactive Clients API

The specification defines a new type of invoker named RxInvoker, and a default implementation of this type named CompletionStageRxInvoker. CompletionStageRxInvoker implements Java 8's interface CompletionStage. This interface declares a large number of methods dedicated to managing asynchronous computations.

There is also a new rx method which is used in a similar manner to async.

27.3. Server-Sent Events (SSE)

SSE is part of HTML standard, currently supported by many browsers. It is a server push technology, which provides a way to establish a one-way channel to continuously send data to clients. SSE events are pushed to the client via a long-running HTTP connection. In case of lost connection, clients can retrieve missed events by setting a "Last-Event-ID" HTTP header in a new request.

SSE stream has text/event-stream media type and contains multiple SSE events. SSE event is a data structure encoded with UTF-8 and contains fields and comment. The field can be event, data, id, retry and other kinds of field will be ignored.

From Jakarta RESTful Web Services 2.1, Server-sent Events APIs are introduced to support sending, receiving and broadcasting SSE events.

27.3.1. SSE Server

As shown in the following example, a SSE resource method has the text/event-stream produce media type and an injected context parameter SseEventSink. The injected SseEventSink is the

connected SSE stream where events can be sent. Another injected context Sse is an entry point for creating and broadcasting SSE events. Here is an example to demonstrate how to send SSE events every 200ms and close the stream after a "done" event.

Example 27.1.

```
  @GET
  @Path("domains/{id}")
  @Produces(MediaType.SERVER_SENT_EVENTS)
  public void startDomain(@PathParam("id") final String id, @Context SseEventSink sink @Context
Sse sse)
  {
     ExecutorService service = (ExecutorService) servletContext
           .getAttribute(ExecutorServletContextListener.TEST_EXECUTOR);
     service.execute(new Thread()
     {
        public void run()
        {
           try
           {
              sink.send(sse.newEventBuilder().name("domain-progress")
                    .data(String.class, "starting domain " + id + " ...").build());
              Thread.sleep(200);
              sink.send(sse.newEvent("domain-progress", "50%"));
              Thread.sleep(200);
              sink.send(sse.newEvent("domain-progress", "60%"));
              Thread.sleep(200);
              sink.send(sse.newEvent("domain-progress", "70%"));
              Thread.sleep(200);
              sink.send(sse.newEvent("domain-progress", "99%"));
              Thread.sleep(200);
              sink.send(sse.newEvent("domain-progress", "Done.")).thenAccept((Object obj) -> {
                 sink.close();
              });
           }
           catch (final InterruptedException e)
           {
              logger.error(e.getMessage(), e);
           }
        }
     });
  }
```

Note

RESTEasy supports sending SSE events via reactive types.

27.3.2. SSE Broadcasting

With SseBroadcaster, SSE events can be broadcasted to multiple clients simultaneously. It will iterate over all registered SseEventSinks and send events to all requested SSE Stream. An application can create a SseBroadcaster from an injected context Sse. The broadcast method on a SseBroadcaster is used to send SSE events to all registered clients. The following code snippet is an example on how to create SseBroadcaster, subscribe and broadcast events to all subscribed consumers.

Example 27.2.

```
@GET
@Path("/subscribe")
@Produces(MediaType.SERVER_SENT_EVENTS)
public void subscribe(@Context SseEventSink sink) throws IOException
{
   if (sink == null)
   {
      throw new IllegalStateException("No client connected.");
   }
   if (sseBroadcaster == null)
   {
      sseBroadcaster = sse.newBroadcaster();
   }
   sseBroadcaster.register(sink);
}

@POST
@Path("/broadcast")
public void broadcast(String message) throws IOException
{
   if (sseBroadcaster == null)
   {
      sseBroadcaster = sse.newBroadcaster();
   }
   sseBroadcaster.broadcast(sse.newEvent(message));

}
```

27.3.3. SSE Client

SseEventSource is the entry point to read and process incoming SSE events. A SseEventSource instance can be initialized with a WebTarget. Once SseEventSource is created and connected to a server, registered event consumer will be invoked when an inbound event arrives. In case of errors, an exception will be passed to a registered consumer so that it can be processed. SseEventSource can automatically reconnect the server and continuously receive pushed events after the connection has been lost. SseEventSource can send lastEventId to the server by default when it is reconnected, and server may use this id to replay all missed events. But reply event

is really upon on SSE resource method implementation. If the server responds HTTP 503 with a RETRY_AFTER header, SseEventSource will automatically schedule a reconnect task with this RETRY_AFTER value. The following code snippet is to create a SseEventSource and print the inbound event data value and error if it happens.

Example 27.3.

```
public void printEvent() throws Exception
{
  WebTarget target = client.target("http://localhost:8080/service/server-sent-events"));
  SseEventSource msgEventSource = SseEventSource.target(target).build();
  try (SseEventSource eventSource = msgEventSource)
  {
     eventSource.register(event -> {
        System.out.println(event.readData(String.class));
     }, ex -> {
        ex.printStackTrace();
     });
     eventSource.open();
  }
}
```

27.4. Java API for JSON Binding

RESTEasy supports both JSON-B and JSON-P. In accordance with the specification, entity providers for JSON-B take precedence over those for JSON-P for all types except JsonValue and its sub-types.

The support for JSON-B is provided by the `JsonBindingProvider` from `resteasy-json-binding-provider` module. To satisfy Jakarta RESTful Web Services 2.1 requirements, JsonBindingProvider takes precedence over the other providers for dealing with JSON payloads, in particular the Jackson one. The JSON outputs (for the same input) from Jackson and JSON-B reference implementation can be slightly different. As a consequence, in order to allow retaining backward compatibility, RESTEasy offers a `resteasy.preferJacksonOverJsonB` context property that can be set to `true` to disable JsonBindingProvider for the current deloyment.

WildFly 14 supports specifying the default value for the `resteasy.preferJacksonOverJsonB` context property by setting a system property with the same name. Moreover, if no value is set for the context and system properties, it scans Jakarta RESTful Web Services deployments for Jackson annotations and sets the property to `true` if any of those annotations is found.

27.5. JSON Patch and JSON Merge Patch

RESTEasy supports apply partial modification to target resource with JSON Patch/JSON Merge Patch. Instead of sending json request which represents the whole modified resource with HTTP

PUT method, the json request only contains the modified part with HTTP PATCH method can do the same job.

JSON Patch request has an array of json object and each JSON object gives the operation to execute against the target resource. Here is an example to modify the target Student resource which has these fields and values: {"firstName":"Alice","id":1,"school":"MiddleWood School"}:

```
PATCH /StudentPatchTest/students/1 HTTP/1.1
Content-Type: application/json-patch+json
Content-Length: 184
Host: localhost:8090
Connection: Keep-Alive

[{"op":"copy","from":"/firstName","path":"/lastName"},
 {"op":"replace","path":"/firstName","value":"John"},
 {"op":"remove","path":"/school"},
 {"op":"add","path":"/gender","value":"male"}]
```

This JSON Patch request will copy the firstName to lastName field , then change the firstName value to "John". The next operation is remove the school value and add male gender to this "id=1" student resource. After this JSON Path is applied, the target resource will be modified to: {"firstName":"John","gender":"male","id":1,"lastName":"Taylor"}. The operation keyword here can be "add", "remove", "replace", "move", "copy", or "test". The "path" value must be a JSON Pointer value to point the part to apply this JSON Patch.

Unlike use the operation keyword to patch the target resource, JSON Merge Patch request directly send the expect json change and RestEasy merge this change to target resource which identified by the request URI. Like the below JSON Merge Patch request, it remove the "school" value and change the "firstName" to "Green". This is much straightforward:

```
PATCH /StudentPatchTest/students/1 HTTP/1.1
Content-Type: application/merge-patch+json
Content-Length: 34
Host: localhost:8090
Connection: Keep-Alive
{"firstName":"Green","school":null}
```

Enable JSON Patch or JSON Merge Patch only needs correctly annotate the resource method with mediaType: @Consumes(MediaType.APPLICATION_JSON_PATCH_JSON) is to enable JSON Patch and @Consumes("application/merge-patch+json") to enable JSON Merge Patch:

```
@GET
@Path("/{id}")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Student getStudent(@PathParam("id") long id)
{
Student student = studentsMap.get(id);
if (student == null)
{
throw new NotFoundException();
}
return student;
}
@PATCH
@Path("/{id}")
@Consumes(MediaType.APPLICATION_JSON_PATCH_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Student patchStudent(@PathParam("id") long id, Student student)
{
if (studentsMap.get(id) == null)
{
throw new NotFoundException();
}
studentsMap.put(id, student);
return student;
}
@PATCH
@Path("/{id}")
@Consumes("application/merge-patch+json")
@Produces(MediaType.APPLICATION_JSON)
public Student mergePatchStudent(@PathParam("id") long id, Student student)
{
if (studentsMap.get(id) == null)
{
throw new NotFoundException();
}
studentsMap.put(id, student);
return student;
}
```

Note

Before create JSON Patch or JSON Merge Patch resource method, there must be a GET method to get this target resource. As above code example, the first resource method is responsible for getting the target resource to apply patch.

It requires the patch filter to enable JSON Patch or JSON Merge Patch. The RestEasy PatchMethodFilter is enabled by default. This filter can be disabled by setting "resteasy.patchfilter.disabled" to true as described in Section 3.5, “Configuration switches”.

Client side needs create these json objects and send it with http PATCH method.

```
            //send JSON Patch request
                WebTarget patchTarget = client.target("http://localhost:8090/StudentPatchTest/
students/1"));
            javax.json.JsonArray patchRequest = Json.createArrayBuilder()
        .add(Json.createObjectBuilder().add("op", "copy").add("from", "/firstName").add("path",
 "/lastName").build())
            .build();
                    patchTarget.request().build(HttpMethod.PATCH,  Entity.entity(patchRequest,
 MediaType.APPLICATION_JSON_PATCH_JSON)).invoke();
            //send JSON Merge Patch request
                WebTarget patchTarget = client.target("http://localhost:8090/StudentPatchTest/
students/1");
                            JsonObject  object  =  Json.createObjectBuilder().add("lastName",
 "Green").addNull("school").build();
          Response result = patchTarget.request().build(HttpMethod.PATCH, Entity.entity(object,
 "application/merge-patch+json")).invoke();
```

Chapter 28. String marshalling for String based @*Param

28.1. Simple conversion

Parameters and properties annotated with `@CookieParam`, `@HeaderParam`, `@MatrixParam`, `@PathParam`, or `@QueryParam` are represented as strings in a raw HTTP request. The specification says that any of these injected parameters can be converted to an object if the object's class has a `valueOf(String)` static method or a constructor that takes one `String`parameter. In the following, for example,

```
public static class Customer {
   private String name;

   public Customer(String name) {
      this.name = name;
   }

   public String getName() {
      return name;
   }
}

@Path("test")
public static class TestResource {

   @GET
   @Path("")
   public Response test(@QueryParam("cust") Customer cust) {
      return Response.ok(cust.getName()).build();
   }
}

@Test
public void testQuery() throws Exception {
    Invocation.Builder request = ClientBuilder.newClient().target("http://localhost:8081/test?
cust=Bill").request();
   Response response = request.get();
   ...
}
```

the query "?cust=Bill" will be transformed automatically to an instance of `Customer` with name == "Bill".

28.2. ParamConverter

What if you have a class where `valueOf()`or this string constructor don't exist or are inappropriate for an HTTP request? Jakarta RESTful Web Services has the `javax.ws.rs.ext.ParamConverterProvider` to help in this situation.

A `ParamConverterProvider` is a provider defined as follows:

```
public interface ParamConverterProvider {

    public <T> ParamConverter<T> getConverter(Class<T> rawType, Type genericType, Annotation
 annotations[]);
}
```

where a `ParamConverter` is defined:

```
public interface ParamConverter<T> {
   ...
   public T fromString(String value);
   public String toString(T value);
}
```

For example, consider `DateParamConverterProvider` and `DateParamConverter`:

```
@Provider
public class DateParamConverterProvider implements ParamConverterProvider {

   @SuppressWarnings("unchecked")
   @Override
    public <T> ParamConverter<T> getConverter(Class<T> rawType, Type genericType, Annotation[]
 annotations) {
      if (rawType.isAssignableFrom(Date.class)) {
         return (ParamConverter<T>) new DateParamConverter();
      }
      return null;
   }
}

public class DateParamConverter implements ParamConverter<Date> {

   public static final String DATE_PATTERN = "yyyyMMdd";

   @Override
   public Date fromString(String param) {
      try {
         return new SimpleDateFormat(DATE_PATTERN).parse(param.trim());
```

```
      } catch (ParseException e) {
         throw new BadRequestException(e);
      }
   }

   @Override
   public String toString(Date date) {
      return new SimpleDateFormat(DATE_PATTERN).format(date);
   }
}
```

Sending a `Date` in the form of a query, e.g., "?date=20161217" will cause the string "20161217" to be converted to a `Date` on the server.

28.3. StringParameterUnmarshaller

In addition to the Jakarta RESTful Web Services `javax.ws.rs.ext.ParamConverterProvider`, RESTEasy also has its own `org.jboss.resteasy.StringParameterUnmarshaller`, defined

```
public interface StringParameterUnmarshaller<T>
{
   void setAnnotations(Annotation[] annotations);

   T fromString(String str);
}
```

It is similar to `javax.ws.rs.ext.ParamConverter` except that

- it converts only from `Strings`;
- it is configured with the annotations on the injected parameter, which allows for fine-grained control over the injection; and
- it is bound to a given parameter by an annotation that is annotated with the meta-annotation `org.jboss.resteasy.annotations.StringParameterUnmarshallerBinder`:

```
@Target({ElementType.ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface StringParameterUnmarshallerBinder
{
   Class<? extends StringParameterUnmarshaller> value();
}
```

For example,

```
@Retention(RetentionPolicy.RUNTIME)
@StringParameterUnmarshallerBinder(TestDateFormatter.class)
public @interface TestDateFormat {
   String value();
}

public static class TestDateFormatter implements StringParameterUnmarshaller<Date> {
   private SimpleDateFormat formatter;

   public void setAnnotations(Annotation[] annotations) {
    TestDateFormat format = FindAnnotation.findAnnotation(annotations, TestDateFormat.class);
      formatter = new SimpleDateFormat(format.value());
   }

   public Date fromString(String str) {
      try {
         return formatter.parse(str);
      } catch (ParseException e) {
         throw new RuntimeException(e);
      }
   }
}

@Path("/")
public static class TestResource {

   @GET
   @Produces("text/plain")
   @Path("/datetest/{date}")
   public String get(@PathParam("date") @TestDateFormat("MM-dd-yyyy") Date date) {
      Calendar c = Calendar.getInstance();
      c.setTime(date);
      return date.toString();
   }
}
```

Note that the annotation `@StringParameterUnmarshallerBinder` on the annotation `@TestDateFormat` binds the formatter `TestDateFormatter` to a parameter annotated with `@TestDateFormat`. In this example, `TestDateFormatter` is used to format the `Date` parameter. Note also that the parameter "MM-dd-yyyy" to `@TestDateFormat` is accessible from `TestDateFormatter.setAnnotations()`.

28.4. Collections

For parameters and properties annotated with `@CookieParam`, `@HeaderParam`, `@MatrixParam`, `@PathParam`, or `@QueryParam`, the Jakarta RESTful Web Services specification [https://jcp.org/aboutJava/communityprocess/final/jsr339/index.html] allows conversion as defined in the Javadoc of the corresponding annotation. In general, the following types are supported:

1. Types for which a `ParamConverter` is available via a registered `ParamConverterProvider`. See Javadoc for these classes for more information.

2. Primitive types.

3. Types that have a constructor that accepts a single `String` argument.

4. Types that have a static method named `valueOf` or `fromString` with a single `String` argument that return an instance of the type. If both methods are present then `valueOf` MUST be used unless the type is an enum in which case `fromString` MUST be used.

5. List<T>, Set<T>, or SortedSet<T>, where T satisfies 3 or 4 above.

Items 1, 3, and 4 have been discussed above, and item 2 is obvious. Note that item 5 allows for collections of parameters. How these collections are expressed in HTTP messages depends, by default, on the particular kind of parameter. In most cases, the notation for collections is based on convention rather than a specification.

28.4.1. @QueryParam

For example, a multivalued query parameter is conventionally expressed like this:

```
http://bluemonkeydiamond.com?q=1&q=2&q=3
```

In this case, there is a query with name "q" and value {1, 2, 3}. This notation is further supported in Jakarta RESTful Web Services by the method

```
public MultivaluedMap<String, String> getQueryParameters();
```

in `javax.ws.rs.core.UriInfo`.

28.4.2. @MatrixParam

There is no specified syntax for collections derived from matrix parameters, but

1. matrix parameters in a URL segment are conventionally separated by ";", and

2. the method

```
MultivaluedMap<String, String> getMatrixParameters();
```

in `javax.ws.rs.core.PathSegment` supports extraction of collections from matrix parameters.

RESTEasy adopts the convention that multiple instances of a matrix parameter with the same name are treated as a collection. For example,

```
http://bluemonkeydiamond.com/sippycup;m=1;m=2;m=3
```

is interpreted as a matrix parameter on path segment "sippycup" with name "m" and value {1, 2, 3}.

28.4.3. @HeaderParam

The HTTP 1.1 specification doesn't exactly specify that multiple components of a header value should be separated by commas, but commas are used in those headers that naturally use lists, e.g. Accept and Allow. Also, note that the method

```
public MultivaluedMap<String, String> getRequestHeaders();
```

in `javax.ws.rs.core.HttpHeaders` returns a `MultivaluedMap`. It is natural, then, for RESTEasy to treat

```
x-header: a, b, c
```

as mapping name "x-header" to set {a, b, c}.

28.4.4. @CookieParam

The syntax for cookies is specified, but, unfortunately, it is specified in multiple competing specifications. Typically, multiple name=value cookie pairs are separated by ";". However, unlike the case with query and matrix parameters, there is no specified Jakarta RESTful Web Services method that returns a collection of cookie values. Consequently, if two cookies with the same name are received on the server and directed to a collection typed parameter, RESTEasy will inject only the second one. Note, in fact, that the method

```
public Map<String, Cookie> getCookies();
```

in `javax.ws.rs.core.HttpHeaders` returns a `Map` rather than a `MultivaluedMap`.

28.4.5. @PathParam

Deriving a collection from path segments is somewhat less natural than it is for other parameters, but Jakarta RESTful Web Services supports the injection of multiple `javax.ws.rs.core.PathSegments`. There are a couple of ways of obtaining multiple `PathSegments`. One is through the use of multiple path variables with the same name. For example, the result of calling `testTwoSegmentsArray()` and `testTwoSegmentsList()` in

```
@Path("")
public static class TestResource {

   @GET
   @Path("{segment}/{other}/{segment}/array")
   public Response getTwoSegmentsArray(@PathParam("segment") PathSegment[] segments) {
      System.out.println("array segments: " + segments.length);
      return Response.ok().build();
   }

   @GET
   @Path("{segment}/{other}/{segment}/list")
   public Response getTwoSegmentsList(@PathParam("segment") List<PathSegment> segments) {
      System.out.println("list segments: " + segments.size());
      return Response.ok().build();
   }
}

...

   @Test
   public void testTwoSegmentsArray() throws Exception {
     Invocation.Builder request = client.target("http://localhost:8081/a/b/c/array").request();
      Response response = request.get();
      Assert.assertEquals(200, response.getStatus());
      response.close();
   }

   @Test
   public void testTwoSegmentsList() throws Exception {
      Invocation.Builder request = client.target("http://localhost:8081/a/b/c/list").request();
      Response response = request.get();
      Assert.assertEquals(200, response.getStatus());
      response.close();
   }
```

is

```
array segments: 2
list segments: 2
```

An alternative is to use a wildcard template parameter. For example, the output of calling `testWildcardArray()` and `testWildcardList()` in

```
@Path("")
public static class TestResource {

   @GET
   @Path("{segments:.*}/array")
   public Response getWildcardArray(@PathParam("segments") PathSegment[] segments) {
      System.out.println("array segments: " + segments.length);
      return Response.ok().build();
   }

   @GET
   @Path("{segments:.*}/list")
   public Response getWildcardList(@PathParam("segments") List<PathSegment> segments) {
      System.out.println("list segments: " + segments.size());
      return Response.ok().build();
   }

...

   @Test
   public void testWildcardArray() throws Exception {
     Invocation.Builder request = client.target("http://localhost:8081/a/b/c/array").request();
      Response response = request.get();
      response.close();
   }

   @Test
   public void testWildcardList() throws Exception {
      Invocation.Builder request = client.target("http://localhost:8081/a/b/c/list").request();
      Response response = request.get();
      response.close();
   }
```

is

```
array segments: 3
list segments: 3
```

28.5. Extension to `ParamConverter` semantics

In the Jakarta RESTful Web Services semantics, a `ParamConverter` is supposed to convert a single `String` that represents an individual object. RESTEasy extends the semantics to allow a

`ParamConverter` to parse the `String` representation of multiple objects and generate a `List<T>`, `Set<T>`, `SortedSet<T>`, array, or, indeed, any multivalued data structure whatever. First, consider the resource

```
@Path("queryParam")
public static class TestResource {

   @GET
   @Path("")
   public Response conversion(@QueryParam("q") List<String> list) {
      return Response.ok(stringify(list)).build();
   }
}

private static <T> String stringify(List<T> list) {
   StringBuffer sb = new StringBuffer();
   for (T s : list) {
      sb.append(s).append(',');
   }
   return sb.toString();
}
```

Calling `TestResource` as follows, using the standard notation,

```
@Test
public void testQueryParamStandard() throws Exception {
   Client client = ClientBuilder.newClient();
          Invocation.Builder    request    =    client.target("http://localhost:8081/queryParam?
q=20161217&q=20161218&q=20161219").request();
   Response response = request.get();
   System.out.println("response: " + response.readEntity(String.class));
}
```

results in

```
response: 20161217,20161218,20161219,
```

Suppose, instead, that we want to use a comma separated notation. We can add

```
public static class MultiValuedParamConverterProvider implements ParamConverterProvider

   @SuppressWarnings("unchecked")
```

```
   @Override
    public <T> ParamConverter<T> getConverter(Class<T> rawType, Type genericType, Annotation[]
 annotations) {
      if (List.class.isAssignableFrom(rawType)) {
         return (ParamConverter<T>) new MultiValuedParamConverter();
      }
      return null;
   }
}

public static class MultiValuedParamConverter implements ParamConverter<List<?>> {

   @Override
   public List<?> fromString(String param) {
      if (param == null || param.trim().isEmpty()) {
         return null;
      }
      return parse(param.split(","));
   }

   @Override
   public String toString(List<?> list) {
      if (list == null || list.isEmpty()) {
         return null;
      }
      return stringify(list);
   }

   private static List<String> parse(String[] params) {
      List<String> list = new ArrayList<String>();
      for (String param : params) {
         list.add(param);
      }
      return list;
   }
}
```

Now we can call

```
@Test
public void testQueryParamCustom() throws Exception {
   Client client = ClientBuilder.newClient();
           Invocation.Builder    request    =    client.target("http://localhost:8081/queryParam?
q=20161217,20161218,20161219").request();
   Response response = request.get();
   System.out.println("response: " + response.readEntity(String.class));
}
```

and get

```
response: 20161217,20161218,20161219,
```

Note that in this case, `MultiValuedParamConverter.fromString()` creates and returns an `ArrayList`, so `TestResource.conversion()` could be rewritten

```
@Path("queryParam")
public static class TestResource {

   @GET
   @Path("")
   public Response conversion(@QueryParam("q") ArrayList<String> list) {
      return Response.ok(stringify(list)).build();
   }
}
```

On the other hand, `MultiValuedParamConverter` could be rewritten to return a `LinkList` and the parameter list in `TestResource.conversion()` could be either a `List` or a `LinkedList`.

Finally, note that this extension works for arrays as well. For example,

```
  public static class Foo {
      private String foo;
      public Foo(String foo) {this.foo = foo;}
      public String getFoo() {return foo;}
   }

   public static class FooArrayParamConverter implements ParamConverter<Foo[]> {

      @Override
      public Foo[] fromString(String value)
      {
         String[] ss = value.split(",");
         Foo[] fs = new Foo[ss.length];
         int i = 0;
         for (String s : ss) {
            fs[i++] = new Foo(s);
         }
         return fs;
      }

      @Override
      public String toString(Foo[] values)
      {
         StringBuffer sb = new StringBuffer();
         for (int i = 0; i < values.length; i++) {
            sb.append(values[i].getFoo()).append(",");
         }
         if (sb.length() > 0) {
            sb.deleteCharAt(sb.length() - 1);
```

```
         }
         return sb.toString();
      }
   }

   @Provider
   public static class FooArrayParamConverterProvider implements ParamConverterProvider {

      @SuppressWarnings("unchecked")
      @Override
     public <T> ParamConverter<T> getConverter(Class<T> rawType, Type genericType, Annotation[]
 annotations) {
         if (rawType.equals(Foo[].class));
         return (ParamConverter<T>) new FooArrayParamConverter();
      }
   }

   @Path("")
   public static class ParamConverterResource {

      @GET
      @Path("test")
      public Response test(@QueryParam("foos") Foo[] foos) {
         return Response.ok(new FooArrayParamConverter().toString(foos)).build();
      }
   }
```

28.6. Default multiple valued `ParamConverter`

RESTEasy includes two built-in `ParamConverters` in the resteasy-core module, one for `Collections`:

```
org.jboss.resteasy.plugins.providers.MultiValuedCollectionParamConverter,
```

and one for arrays:

```
org.jboss.resteasy.plugins.providers.MultiValuedArrayParamConverter,
```

which implement the concepts in the previous section.

In particular, `MultiValued*ParamConverter.fromString()` can transform a string representation coming over the network into a `Collection` or array, and `MultiValued*ParamConverter.toString()` can be used by a client side proxy to transform `Collections` or arrays into a string representation.

String representations are determined by `org.jboss.resteasy.annotations.Separator`, a parameter annotation in the resteasy-core module:

```
@Target({ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
public @interface Separator
{
   public String value() default "";
}
```

The value of `Separator.value()` is used to separate individual elements of a `Collection` or array. For example, a proxy implementing

```
@Path("path/separator/multi/{p}")
@GET
public String pathMultiSeparator(@PathParam("p") @Separator("-") List<String> ss);
```

will turn

```
List<String> list = new ArrayList<String>();
list.add("abc");
list.add("xyz");
proxy.pathMultiSeparator(list);
```

and "path/separator/multi/{p}" into ".../path/separator/multi/abc-xyz". On the server side, the RESTEasy runtime will turn "abc-xyz" back into a list consisting of elements "abc" and "xyz" for

```
@Path("path/separator/multi/{p}")
@GET
public String pathMultiSeparator(@PathParam("p") @Separator("-") List<String> ss) {
   StringBuffer sb = new StringBuffer();
   for (String s : ss) {
      sb.append(s);
      sb.append("|");
   }
   return sb.toString();
}
```

which will return "abc|xyz|".

In fact, the value of the `Separator` annotations may be a more general regular expression, which is passed to `String.split()`. For example, "[-,;]" tells the server side to break up a string using either "-", ",", or ";". On the client side, a string will be created using the first element, "-" in this case.

If a parameter is annotated with `@Separator` with no value, then the default value is

- "," for a `@HeaderParam`, `@MatrixParam`, `@PathParam`, or `@QueryParam`, and
- "-" for a `@CookieParam`.

The `MultiValued*ParamConverters` depend on existing facilities for handling the individual elements. On the server side, once it has parsed the incoming string into substrings, `MultiValued*ParamConverter` turns each substring into an Java object according to Section 3.2 "Fields and Bean Properties" of the Jakarta RESTful Web Services specification. On the client side, `MultiValued*ParamConverter` turns a Java object into a string as follows:

1. look for a `ParamConverter`;
2. if there is no suitable `ParamConverter` and the parameter is labeled `@HeaderParam`, look for a `HeaderDelegate`; or
3. call `toString()`.

These `ParamConverters` are meant to be fairly general, but there are a number of restrictions:

1. They don't handle nested `Collections` or arrays. That is, `List<String>` and `String[]` are OK, but `List<List<String>>` and `String[][]` are not.
2. The regular expression used in `Separator` must match the regular expression

```
"\\p{Punct}|\\[\\p{Punct}+\\]"
```

   That is, it must be either a single instance of a punctuation symbol, i.e., a symbol in the set

```
!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
```

   or a class of punctuation symbols like "[-,;]".
3. For either of these `ParamConverters` to be available for use with a given parameter, that parameter must be annotated with `@Separator`.

There are also some logical restrictions:

1. Cookie syntax, as specified in https://tools.ietf.org/html/rfc6265#section-4.1.1, assigns a meaning to ";", so it cannot be used as a separator.

2. If a separator character appears in the content of an element, then there will be problems. For example, if "," is used as a separator, then, if a proxy sends the array `["a","b,c","d"]`, it will turn into the string "a,b,c,d" on the wire and be reconstituted on the server as four elements.

These built-in `ParamConverters` have the lowest priority, so any user supplied `ParamConverters` will be tried first.

Chapter 29. Responses using javax.ws.rs.core.Response

You can build custom responses using the javax.ws.rs.core.Response and ResponseBuilder classes. If you want to do your own streaming, your entity response must be an implementation of javax.ws.rs.core.StreamingOutput. See the java doc for more information.

Chapter 30. Exception Handling

30.1. Exception Mappers

ExceptionMappers are custom, application provided, components that can catch thrown application exceptions and write specific HTTP responses. They are classes annotated with @Provider and that implement this interface

```
package javax.ws.rs.ext;

import javax.ws.rs.core.Response;

/**
* Contract for a provider that maps Java exceptions to
* {@link javax.ws.rs.core.Response}. An implementation of this interface must
* be annotated with {@link Provider}.
*
* @see Provider
* @see javax.ws.rs.core.Response
*/
public interface ExceptionMapper<E>
{
   /**
   * Map an exception to a {@link javax.ws.rs.core.Response}.
   *
   * @param exception the exception to map to a response
   * @return a response mapped from the supplied exception
   */
   Response toResponse(E exception);
}
```

When an application exception is thrown it will be caught by the Jakarta RESTful Web Services runtime. Jakarta RESTful Web Services will then scan registered ExceptionMappers to see which one support marshalling the exception type thrown. Here is an example of ExceptionMapper

```
@Provider
public class EJBExceptionMapper implements ExceptionMapper<javax.ejb.EJBException>
{
   public Response toResponse(EJBException exception) {
      return Response.status(500).build();
   }
}
```

You register ExceptionMappers the same way you do MessageBodyReader/Writers. By scanning for @Provider annotated classes, or programmatically through the ResteasyProviderFactory class.

30.2. RESTEasy Built-in Internally-Thrown Exceptions

RESTEasy has a set of built-in exceptions that are thrown by it when it encounters errors during dispatching or marshalling. They all revolve around specific HTTP error codes. You can find them in RESTEasy's javadoc under the package org.jboss.resteasy.spi. Here's a list of them:

Table 30.1.

Exception	HTTP Code	Description
ReaderException	400	All exceptions thrown from MessageBodyReaders are wrapped within this exception. If there is no ExceptionMapper for the wrapped exception or if the exception isn't a WebApplicationException, then resteasy will return a 400 code by default.
WriterException	500	All exceptions thrown from MessageBodyWriters are wrapped within this exception. If there is no ExceptionMapper for the wrapped exception or if the exception isn't a WebApplicationException, then resteasy will return a 400 code by default.
o.j.r.plugins.providers.jaxb.JAXBUnmarshalException	400	The Jakarta XML Binding providers throw this exception on reads. They may be wrapping JAXBExceptions. This class extends ReaderException
o.j.r.plugins.providers.jaxb.JAXBMarshalException	500	The Jakarta XML Binding providers throw this exception on writes. They may be wrapping JAXBExceptions. This class extends WriterException

Exception	HTTP Code	Description
ApplicationException	N/A	This exception wraps all exceptions thrown from application code. It functions much in the same way as InvocationTargetException. If there is an ExceptionMapper for wrapped exception, then that is used to handle the request.
Failure	N/A	Internal RESTEasy. Not logged
LoggableFailure	N/A	Internal RESTEasy error. Logged
DefaultOptionsMethodException	N/A	If the user invokes HTTP OPTIONS and no Jakarta RESTful Web Services method for it, RESTEasy provides a default behavior by throwing this exception. This is only done if the property `dev.resteasy.throw.options.exception` is set to true.
UnrecognizedPropertyExceptionHandler	400	A Jackson provider throws this exception when JSON data is determine to be invalid.

30.3. Resteasy WebApplicationExceptions

Suppose a client at local.com calls the following resource method:

```
@GET
@Path("remote")
public String remote() throws Exception {
   Client client = ClientBuilder.newClient();
   return client.target("http://third.party.com/exception").request().get(String.class);
}
```

If the call to http://third.party.com returns a status code 3xx, 4xx, or 5xx, then the `Client` is obliged by the Jakarta RESTful Web Services specification to throw a `WebApplicationException`. Moreover, if the `WebApplicationException` contains a `Response`, which it normally would in RESTEasy, the server runtime is obliged by the Jakarta RESTful Web Services specification to return that `Response`. As a result, information from the server at third.party.com, e.g., headers

and body, will get sent back to local.com. The problem is that that information could be, at best, meaningless to the client and, at worst, a security breach.

RESTEasy has a solution that works around the problem and still conforms to the Jakarta RESTful Web Services specification. In particular, for each `WebApplicationException` it defines a new subclass:

```
WebApplicationException
+-ResteasyWebApplicationException
+-ClientErrorException
| +-ResteasyClientErrorException
| +-BadRequestException
| | +-ResteasyBadRequestException
| +-ForbiddenException
| | +-ResteasyForbiddenException
| +-NotAcceptableException
| | +-ResteasyNotAcceptableException
| +-NotAllowedException
| | +-ResteasyNotAllowedException
| +-NotAuthorizedException
| | +-ResteasyNotAuthorizedException
| +-NotFoundException
| | +-ResteasyNotFoundException
| +-NotSupportedException
| | +-ResteasyNotSupportedException
+-RedirectionException
| +-ResteasyRedirectionException
+-ServerErrorException
| +-ResteasyServerErrorException
| +-InternalServerErrorException
| | +-ResteasyInternalServerErrorException
| +-ServiceUnavailableException
| | +-ResteasyServiceUnavailableException
```

The new `Exception`s play the same role as the original ones, but RESTEasy treats them slightly differently. When a `Client` detects that it is running in the context of a resource method, it will throw one of the new `Exception`s. However, instead of storing the original `Response`, it stores a "sanitized" version of the `Response`, in which only the status and the Allow and Content-Type headers are preserved. The original `WebApplicationException`, and therefore the original `Response`, can be accessed in one of two ways:

```
// Create a NotAcceptableException.
NotAcceptableException                      nae                      =                      new
 NotAcceptableException(Response.status(406).entity("ooops").build());

// Wrap the NotAcceptableException in a ResteasyNotAcceptableException.
ResteasyNotAcceptableException         rnae         =         (ResteasyNotAcceptableException)
 WebApplicationExceptionWrapper.wrap(nae);

// Extract the original NotAcceptableException using instance method.
```

```
NotAcceptableException nae2 = rnae.unwrap();
Assert.assertEquals(nae, nae2);

// Extract the original NotAcceptableException using class method.
NotAcceptableException              nae3              =              (NotAcceptableException)
 WebApplicationExceptionWrapper.unwrap(nae); // second way
Assert.assertEquals(nae, nae3);
```

Note that this change is intended to introduce a safe default behavior in the case that the `Exception` generated by the remote call is allowed to make its way up to the server runtime. It is considered a good practice, though, to catch the `Exception` and treat it in some appropriate manner:

```
   @GET
   @Path("remote/{i}")
   public String remote(@PathParam("i") String i) throws Exception {
      Client client = ClientBuilder.newClient();
      try {
         return client.target("http://remote.com/exception/" + i).request().get(String.class);
      } catch (WebApplicationException wae) {
         ...
      }
   }
```

Note. While RESTEasy will default to the new, safer behavior, the original behavior can be restored by setting the configuration parameter "resteasy.original.webapplicationexception.behavior" to "true".

30.4. Overriding RESTEasy Builtin Exceptions

You may override RESTEasy built-in exceptions by writing an ExceptionMapper for the exception. For that matter, you can write an ExceptionMapper for any thrown exception including WebApplicationException

Chapter 31. Configuring Individual Jakarta RESTful Web Services Resource Beans

If you are scanning your path for Jakarta RESTful Web Services annotated resource beans, your beans will be registered in per-request mode. This means an instance will be created per HTTP request served. Generally, you will need information from your environment. If you are running within a servlet container using the WAR-file distribution, in 1.0.0.Beta-2 and lower, you can only use the JNDI lookups to obtain references to Jakarta EE resources and configuration information. In this case, define your Jakarta EE configuration (i.e. ejb-ref, env-entry, persistence-context-ref, etc...) within web.xml of the resteasy WAR file. Then within your code do jndi lookups in the java:comp namespace. For example:

web.xml

```
<ejb-ref>
  <ejb-ref-name>ejb/foo</ejb-ref-name>
  ...
</ejb-ref>
```

resource code:

```
@Path("/")
public class MyBean {

   public Object getSomethingFromJndi() {
      new InitialContext.lookup("java:comp/ejb/foo");
   }
...
}
```

You can also manually configure and register your beans through the Registry. To do this in a WAR-based deployment, you need to write a specific ServletContextListener to do this. Within the listener, you can obtain a reference to the registry as follows:

```
```

```
public class MyManualConfig implements ServletContextListener
{
   public void contextInitialized(ServletContextEvent event)
   {

                                          Registry       registry       =       (Registry)
 event.getServletContext().getAttribute(Registry.class.getName());

   }
...
}
```

Please also take a look at our Spring Integration as well as the Embedded Container's Spring Integration

Chapter 32. Content encoding

32.1. GZIP Compression/Decompression

RESTEasy supports (though not by default - see below) GZIP decompression. If properly configured, the client framework or a Jakarta RESTful Web Services service, upon receiving a message body with a Content-Encoding of "gzip", will automatically decompress it. The client framework can (though not by default - see below) automatically set the Accept-Encoding header to be "gzip, deflate" so you do not have to set this header yourself.

RESTEasy also supports (though not by default - see below) automatic compression. If the client framework is sending a request or the server is sending a response with the Content-Encoding header set to "gzip", RESTEasy will (if properly configured) do the compression. So that you do not have to set the Content-Encoding header directly, you can use the @org.jboss.resteasy.annotation.GZIP annotation.

```
@Path("/")
public interface MyProxy {

   @Consumes("application/xml")
   @PUT
   public void put(@GZIP Order order);
}
```

In the above example, we tag the outgoing message body, order, to be gzip compressed. You can use the same annotation to tag server responses

```
@Path("/")
public class MyService {

   @GET
   @Produces("application/xml")
   @GZIP
   public String getData() {...}
}
```

32.1.1. Configuring GZIP compression / decompression

Note. Decompression carries a risk of attack from a bad actor that can package an entity that will expand greatly. Consequently, RESTEasy disables GZIP compression / decompression by default.

There are three interceptors that are relevant to GZIP compression / decompression:

1. org.jboss.resteasy.plugins.interceptors.GZIPDecodingInterceptor: If the Content-Encoding header is present and has the value "gzip", `GZIPDecodingInterceptor` will install an `InputStream` that decompresses the message body.

2. org.jboss.resteasy.plugins.interceptors.GZIPEncodingInterceptor: If the Content-Encoding header is present and has the value "gzip", `GZIPEncodingInterceptor` will install an `OutputStream` that compresses the message body.

3. org.jboss.resteasy.plugins.interceptors.AcceptEncodingGZIPFilter: If the Accept-Encoding header does not exist, `AcceptEncodingGZIPFilter` will add Accept-Encoding with the value "gzip, deflate". If the Accept-Encoding header exists but does not contain "gzip", `AcceptEncodingGZIPFilter` will append ", gzip". Note that enabling GZIP compression / decompression does not depend on the presence of this interceptor.

If GZIP decompression is enabled, an upper limit is imposed on the number of bytes `GZIPDecodingInterceptor` will extract from a compressed message body. The default limit is 10,000,000, but a different value can be configured. See below.

32.1.1.1. Server side configuration

The interceptors may be enabled by including their classnames in a META-INF/services/javax.ws.rs.ext.Providers file on the classpath. The upper limit on deflated files may be configured by setting the parameter "resteasy.gzip.max.input". [See Section 3.4, “Configuration” for more information about application configuration.] If the limit is exceeded on the server side, GZIPDecodingInterceptor will return a `Response` with status 413 ("Request Entity Too Large") and a message specifying the upper limit.

Note. As of release 3.1.0.Final, the GZIP interceptors have moved from package `org.jboss.resteasy.plugins.interceptors.encoding` to `org.jboss.resteasy.plugins.interceptors`. and they should be named accordingly in javax.ws.rs.ext.Providers.

32.1.1.2. Client side configuration

The interceptors may be enabled by registering them with, for example, a `Client` or `WebTarget`. For example,

```
Client client = ClientBuilder.newBuilder() // Activate gzip compression on client:
                    .register(AcceptEncodingGZIPFilter.class)
                    .register(GZIPDecodingInterceptor.class)
                    .register(GZIPEncodingInterceptor.class)
                    .build();
```

The upper limit on deflated files may configured by creating an instance of `GZIPDecodingInterceptor` with a specific value:

```
Client client = ClientBuilder.newBuilder() // Activate gzip compression on client:
                    .register(AcceptEncodingGZIPFilter.class)
                    .register(new GZIPDecodingInterceptor(256))
                    .register(GZIPEncodingInterceptor.class)
                    .build();
```

If the limit is exceeded on the client side, GZIPDecodingInterceptor will throw a `ProcessingException` with a message specifying the upper limit.

32.2. General content encoding

The designation of a compressible entity by the use of the `@GZIP` annotation is a built in, specific instance of a more general facility supported by RESTEasy. There are three components to this facility.

1. The annotation `org.jboss.resteasy.annotations.ContentEncoding` is a "meta-annotation" used on other annotations to indicate that they represent a Content-Encoding. For example, `@GZIP` is defined

```
@Target({ElementType.TYPE, ElementType.METHOD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@ContentEncoding("gzip")
public @interface GZIP
{
}
```

   The value of `@ContentEncoding` indicates the represented Content-Encoding. For `@GZIP` it is "gzip".

2. `ClientContentEncodingAnnotationFeature` and `ServerContentEncodingAnnotationFeature`, two `DynamicFeatures` in package `org.jboss.resteasy.plugins.interceptors`, examine resource methods for annotations decorated with `@ContentEncoding`.

3. For each value found in a `@ContentEncoding` decorated annotation on a resource method, an instance of `ClientContentEncodingAnnotationFilter` or `ServerContentEncodingAnnotationFilter`, `javax.ws.rs.ext.WriterInterceptors` in package `org.jboss.resteasy.plugins.interceptors`, is registered. They are responsible for adding an appropriate Content-Encoding header. For example, `ClientContentEncodingAnnotationFilter` is defined

```
@ConstrainedTo(RuntimeType.CLIENT)
@Priority(Priorities.HEADER_DECORATOR)
```

```
public class ClientContentEncodingAnnotationFilter implements WriterInterceptor
{
   protected String encoding;

   public ClientContentEncodingAnnotationFilter(String encoding)
   {
      this.encoding = encoding;
   }

   @Override
       public  void  aroundWriteTo(WriterInterceptorContext  context)  throws  IOException,
 WebApplicationException
   {
      context.getHeaders().putSingle(HttpHeaders.CONTENT_ENCODING, encoding);
      context.proceed();
   }
}
```

When it is created, `ClientContentEncodingAnnotationFeature` passes in the value to be used for Content-Encoding headers.

The annotation `@GZIP` is built into RESTEasy, but `ClientContentEncodingAnnotationFeature` and `ServerContentEncodingAnnotationFeature` will also recognize application defined annotations. For example,

```
@Target({ElementType.TYPE, ElementType.METHOD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@ContentEncoding("compress")
public @interface Compress
{
}

@Path("")
public static class TestResource {

   @GET
   @Path("a")
   @Compress
   public String a() {
      return "a";
   }
}
```

If `TestResource.a()` is invoked as follows

```
@Test
public void testCompress() throws Exception
{
   Client client = ClientBuilder.newClient();
```

```
        Invocation.Builder request = client.target("http://localhost:8081/a").request();
        request.acceptEncoding("gzip,compress");
        Response response = request.get();
        System.out.println("content-encoding: "+ response.getHeaderString("Content-Encoding"));
        client.close();
    }
```

the output will be

```
content-encoding: compress
```

Chapter 33. CORS

RESTEasy has a `ContainerRequestFilter` that can be used to handle CORS preflight and actual requests. `org.jboss.resteasy.plugins.interceptors.CorsFilter`. You must allocate this and register it as a singleton provider from your Application class. See the javadoc or its various settings.

```
CorsFilter filter = new CorsFilter();
filter.getAllowedOrigins().add("http://localhost");
```

Chapter 34. Content-Range Support

RESTEasy supports `Range` requests for `java.io.File` response entities.

```
@Path("/")
public class Resource {
   @GET
   @Path("file")
   @Produces("text/plain")
   public File getFile()
   {
      return file;
   }
}

   Response response = client.target(generateURL("/file")).request()
           .header("Range", "1-4").get();
   Assert.assertEquals(response.getStatus(), 206);
   Assert.assertEquals(4, response.getLength());
   System.out.println("Content-Range: " + response.getHeaderString("Content-Range"));
```

Chapter 35. RESTEasy Caching Features

RESTEasy provides numerous annotations and facilities to support HTTP caching semantics. Annotations to make setting Cache-Control headers easier and both server-side and client-side in-memory caches are available.

35.1. @Cache and @NoCache Annotations

RESTEasy provides an extension to Jakarta RESTful Web Services that allows you to automatically set Cache-Control headers on a successful GET request. It can only be used on @GET annotated methods. A successful @GET request is any request that returns 200 OK response.

```
package org.jboss.resteasy.annotations.cache;

public @interface Cache
{
   int maxAge() default -1;
   int sMaxAge() default -1;
   boolean noStore() default false;
   boolean noTransform() default false;
   boolean mustRevalidate() default false;
   boolean proxyRevalidate() default false;
   boolean isPrivate() default false;
}

public @interface NoCache
{
   String[] fields() default {};
}
```

While @Cache builds a complex Cache-Control header, @NoCache is a simplified notation to say that you don't want anything cached; i.e. Cache-Control: nocache.

These annotations can be put on the resource class or interface and specifies a default cache value for each @GET resource method. Or they can be put individually on each @GET resource method.

35.2. Client "Browser" Cache

RESTEasy has the ability to set up a client-side, browser-like, cache. You can use it with the Client Proxy Framework, or with ordinary requests. This cache looks for Cache-Control headers sent back with a server response. If the Cache-Control headers specify that the client is allowed

to cache the response, Resteasy caches it within local memory. The cache obeys max-age requirements and will also automatically do HTTP 1.1 cache revalidation if either or both the Last-Modified and/or ETag headers are sent back with the original response. See the HTTP 1.1 specification for details on how Cache-Control or cache revalidation works.

It is very simple to enable caching. Here's an example of using the client cache with the Client Proxy Framework

```
@Path("/orders")
public interface OrderServiceClient {

   @Path("{id}")
   @GET
   @Produces("application/xml")
   public Order getOrder(@PathParam("id") String id);
}
```

To create a proxy for this interface and enable caching for that proxy requires only a few simple steps in which the `BrowserCacheFeature` is registered:

```
ResteasyWebTarget target = (ResteasyWebTarget) ClientBuilder.newClient().target("http://
localhost:8081");
BrowserCacheFeature cacheFeature = new BrowserCacheFeature();
OrderServiceClient                                orderService                                =
 target.register(cacheFeature).proxy(OrderServiceClient.class);
```

`BrowserCacheFeature` will create a Resteasy `LightweightBrowserCache` by default. It is also possible to configure the cache, or install a completely different cache implementation:

```
ResteasyWebTarget target = (ResteasyWebTarget) ClientBuilder.newClient().target("http://
localhost:8081");
LightweightBrowserCache cache = new LightweightBrowserCache();
cache.setMaxBytes(20);
BrowserCacheFeature cacheFeature = new BrowserCacheFeature();
cacheFeature.setCache(cache);
OrderServiceClient                                orderService                                =
 target.register(cacheFeature).proxy(OrderServiceClient.class);
```

If you are using the standard Jakarta RESTful Web Services client framework to make invocations rather than the proxy framework, it is just as easy:

```
ResteasyWebTarget target = (ResteasyWebTarget) ClientBuilder.newClient().target("http://
localhost:8081/orders/{id}");
```

```
BrowserCacheFeature cacheFeature = new BrowserCacheFeature();
target.register(cacheFeature);
String rtn = target.resolveTemplate("id", "1").request().get(String.class);
```

The LightweightBrowserCache, by default, has a maximum 2 megabytes of caching space. You can change this programmatically by callings its setMaxBytes() method. If the cache gets full, the cache completely wipes itself of all cached data. This may seem a bit draconian, but the cache was written to avoid unnecessary synchronizations in a concurrent environment where the cache is shared between multiple threads. If you desire a more complex caching solution or if you want to plug in a thirdparty cache please contact our resteasy-developers list and discuss it with the community.

35.3. Local Server-Side Response Cache

RESTEasy has a server-side cache that can sit in front of your Jakarta RESTful Web Services services. It automatically caches marshalled responses from HTTP GET Jakarta RESTful Web Services invocations if, and only if your Jakarta RESTful Web Services resource method sets a Cache-Control header. When a GET comes in, the RESTEasy Server Cache checks to see if the URI is stored in the cache. If it does, it returns the already marshalled response without invoking your Jakarta RESTful Web Services method. Each cache entry has a max age to whatever is specified in the Cache-Control header of the initial request. The cache also will automatically generate an ETag using an MD5 hash on the response body. This allows the client to do HTTP 1.1 cache revalidation with the IF-NONE-MATCH header. The cache is also smart enough to perform revalidation if there is no initial cache hit, but the Jakarta RESTful Web Services method still returns a body that has the same ETag.

The cache is also automatically invalidated for a particular URI that has PUT, POST, or DELETE invoked on it. You can also obtain a reference to the cache by injecting a org.jboss.resteasy.plugins.cache.ServerCache via the @Context annotation

```
@Context
ServerCache cache;

@GET
public String get(@Context ServerCache cache) {...}
```

To set up the server-side cache you must register an instance of org.jboss.resteasy.plugins.cache.server.ServerCacheFeature via your Application getSingletons() or getClasses() methods. The underlying cache is Infinispan. By default, RESTEasy will create an Infinispan cache for you. Alternatively, you can create and pass in an instance of your cache to the ServerCacheFeature constructor. You can also configure Infinispan by specifying various parameters. First, if you are using Maven you must depend on the cache-core artifact:

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

The next thing you should probably do is set up the Infinispan configuration. In your web.xml, it would look like

```
<web-app>
    <context-param>
        <param-name>server.request.cache.infinispan.config.file</param-name>
        <param-value>infinispan.xml</param-value>
    </context-param>

    <context-param>
        <param-name>server.request.cache.infinispan.cache.name</param-name>
        <param-value>MyCache</param-value>
    </context-param>

</web-app>
```

server.request.cache.infinispan.config.file can either be a classpath or a file path. server.request.cache.infinispan.cache.name is the name of the cache you want to reference that is declared in the config file.

See Section 3.4, “Configuration” for more information about application configuration.

35.4. HTTP preconditions

Jakarta RESTful Web Services provides an API for evaluating HTTP preconditions based on `"If-Match"`, `"If-None-Match"`, `"If-Modified-Since"` and `"If-Unmodified-Since"` headers.

```
        Response.ResponseBuilder rb = request.evaluatePreconditions(lastModified, etag);
```

By default RESTEasy will return status code 304 (Not modified) or 412 (Precondition failed) if any of conditions fails. However it is not compliant with RFC 7232 which states that headers `"If-Match"`, `"If-None-Match"` MUST have higher precedence. You can enable RFC 7232 compati-

ble mode by setting the parameter `resteasy.rfc7232preconditions` to `true`. See Section 3.4, “Configuration” for more information about application configuration.

Chapter 36. Filters and Interceptors

Jakarta RESTful Web Services has two different concepts for interceptions: Filters and Interceptors. Filters are mainly used to modify or process incoming and outgoing request headers or response headers. They execute before and after request and response processing.

36.1. Server Side Filters

On the server-side you have two different types of filters. ContainerRequestFilters run before your Jakarta RESTful Web Services resource method is invoked. ContainerResponseFilters run after your Jakarta RESTful Web Services resource method is invoked. As an added caveat, ContainerRequestFilters come in two flavors: pre-match and post-matching. Pre-matching ContainerRequestFilters are designated with the @PreMatching annotation and will execute before the Jakarta RESTful Web Services resource method is matched with the incoming HTTP request. Pre-matching filters often are used to modify request attributes to change how it matches to a specific resource method (i.e. strip .xml and add an Accept header). ContainerRequestFilters can abort the request by calling ContainerRequestContext.abortWith(Response). A filter might want to abort if it implements a custom authentication protocol.

After the resource class method is executed, Jakarta RESTful Web Services will run all ContainerResponseFilters. These filters allow you to modify the outgoing response before it is marshalling and sent to the client. So given all that, here's some pseudo code to give some understanding of how things work.

```
// execute pre match filters
for (ContainerRequestFilter filter : preMatchFilters) {
    filter.filter(requestContext);
    if (isAborted(requestContext)) {
       sendAbortionToClient(requestContext);
       return;
    }
}
// match the HTTP request to a resource class and method
JaxrsMethod method = matchMethod(requestContext);

// Execute post match filters
for (ContainerRequestFilter filter : postMatchFilters) {
   filter.filter(requestContext);
   if (isAborted(requestContext)) {
      sendAbortionToClient(requestContext);
      return;
   }
}

// execute resource class method
method.execute(request);

// execute response filters
for (ContainerResponseFilter filter : responseFilters) {
```

```
            filter.filter(requestContext, responseContext);
        }
```

36.1.1. Asynchronous filters

It is possible to turn filters into asynchronous filters, if you need to suspend execution of your filter until a certain resource has become available. This turns the request asynchronous, but requires no change to your resource method declaration. In particular, synchronous and asynchronous resource methods continue to work as specified, regardless of whether or not a filter turned the request asynchronous. Similarly, one filter turning the request asynchronous requires no change in the declaration of further filters.

In order to turn a filter's execution asynchronous, you need to cast the `ContainerRequestContext` into a `SuspendableContainerRequestContext` (for pre/post request filters), or cast the `ContainerResponseContext` into a `SuspendableContainerResponseContext` (for response filters).

These context objects can turn the current filter's execution to asynchronous by calling the `suspend()` method. Once asynchronous, the filter chain is suspended, and will only resume after one of the following method is called on the context object:

`abortWith(Response)`
: Terminate the filter chain, return the given `Response` to the client (only for `ContainerRequestFilter`).

`resume()`
: Resume execution of the filter chain by calling the next filter.

`resume(Throwable)`
: Abort execution of the filter chain by throwing the given exception. This behaves as if the filter were synchronous and threw the given exception.

You can also do async processing inside your `AsyncWriterInterceptor` (if you are using Async IO), which is the asynchronous-supporting equivalent to `WriterInterceptor`. In this case, you don't need to manually suspend or resume the request.

36.2. Client Side Filters

On the client side you also have two types of filters: ClientRequestFilter and ClientResponseFilter. ClientRequestFilters run before your HTTP request is sent over the wire to the server. ClientResponseFilters run after a response is received from the server, but before the response body is unmarshalled. ClientRequestFilters are also allowed to abort the execute of the request and provide a canned response without going over the wire to the server. ClientResponseFilters can modfiy the Response object before it is handed back to application code. Here's some pseudo code to illustrate things.

```
```

```
            // execute request filters
            for (ClientRequestFilter filter : requestFilters) {
               filter.filter(requestContext);
               if (isAborted(requestContext)) {
                  return requestContext.getAbortedResponseObject();
               }
            }

            // send request over the wire
            response = sendRequest(request);

            // execute response filters
            for (ClientResponseFilter filter : responseFilters) {
               filter.filter(requestContext, responseContext);
            }
```

36.3. Reader and Writer Interceptors

While filters modify request or response headers, interceptors deal with message bodies. Interceptors are executed in the same call stack as their corresponding reader or writer. ReaderInterceptors wrap around the execution of MessageBodyReaders. WriterInterceptors wrap around the execution of MessageBodyWriters. They can be used to implement a specific content-encoding. They can be used to generate digital signatures or to post or pre-process a Java object model before or after it is marshalled.

Note that in order to support Async IO, you can implement `AsyncWriterInterceptor`, which is a subtype of `WriterInterceptor`.

36.4. Per Resource Method Filters and Interceptors

Sometimes you want a filter or interceptor to only run for a specific resource method. You can do this in two different ways: register an implementation of DynamicFeature or use the @NameBinding annotation. The DynamicFeature interface is executed at deployment time for each resource method. You just use the Configurable interface to register the filters and interceptors you want for the specific resource method. @NameBinding works a lot like CDI interceptors. You annotate a custom annotation with @NameBinding and then apply that custom annotation to your filter and resource method. The custom annotation must use @Retention(RetentionPolicy.RUNTIME) in order for the attribute to be picked up by the RESTEasy runtime code when it is deployed.

```
            @NameBinding
            @Retention(RetentionPolicy.RUNTIME)
            public @interface DoIt {}

            @DoIt
            public class MyFilter implements ContainerRequestFilter {...}

            @Path("/root")
            public class MyResource {
```

```
    @GET
    @DoIt
    public String get() {...}
}
```

36.5. Ordering

Ordering is accomplished by using the @BindingPriority annotation on your filter or interceptor class.

Chapter 37. Asynchronous HTTP Request Processing

Asynchronous HTTP Request Processing is a relatively new technique that allows you to process a single HTTP request using non-blocking I/O and, if desired in separate threads. Some refer to it as COMET capabilities. The primary use case for Asynchronous HTTP is in the case where the client is polling the server for a delayed response. The usual example is an AJAX chat client where you want to push/pull from both the client and the server. These scenarios have the client blocking a long time on the server's socket waiting for a new message. What happens in synchronous HTTP where the server is blocking on incoming and outgoing I/O is that you end up having a thread consumed per client connection. This eats up memory and valuable thread resources. Not such a big deal in 90% of applications (in fact using asynchronous processing may actually hurt your performance in most common scenarios), but when you start getting a lot of concurrent clients that are blocking like this, there's a lot of wasted resources and your server does not scale that well.

37.1. Using the `@Suspended` annotation

The Jakarta RESTful Web Services specification includes asynchronous HTTP support via two classes. The `@Suspended` annotation, and AsyncResponse interface.

Injecting an AsynchronousResponse as a parameter to your Jakarta RESTful Web Services methods tells RESTEasy that the HTTP request/response should be detached from the currently executing thread and that the current thread should not try to automatically process the response.

The AsyncResponse is the callback object. The act of calling one of the resume() methods will cause a response to be sent back to the client and will also terminate the HTTP request. Here is an example of asynchronous processing:

```
import javax.ws.rs.Suspend;
import javax.ws.rs.core.AsynchronousResponse;

@Path("/")
public class SimpleResource
{

   @GET
   @Path("basic")
   @Produces("text/plain")
   public void getBasic(@Suspended final AsyncResponse response) throws Exception
   {
      Thread t = new Thread()
      {
         @Override
         public void run()
         {
            try
```

```
            {
               Response jaxrs = Response.ok("basic").type(MediaType.TEXT_PLAIN).build();
               response.resume(jaxrs);
            }
            catch (Exception e)
            {
               response.resume(e);
            }
         }
      };
      t.start();
   }
}
```

AsyncResponse also has other methods to cancel the execution. See javadoc for more details.

NOTE: The old RESTEasy proprietary API for async http has been deprecated and may be removed as soon as RESTEasy 3.1. In particular, the RESTEasy @Suspend annotation is replaced by `javax.ws.rs.container.Suspended`, and `org.jboss.resteasy.spi.AsynchronousResponse` is replaced by `javax.ws.rs.container.AsyncResponse`. Note that @Suspended does not have a value field, which represented a timeout limit. Instead, `AsyncResponse.setTimeout()` may be called.

37.2. Using Reactive return types

The Jakarta RESTful Web Services 2.1 specification adds support for declaring asynchronous resource methods by returning a `CompletionStage` instead of using the `@Suspended` annotation.

Whenever a resource method returns a `CompletionStage`, it will be subscribed to, the request will be suspended, and only resumed when the `CompletionStage` is resolved either to a value (which is then treated as the return value for the method), or as an error case, in which case the exception will be processed as if it were thrown by the resource method.

Here is an example of asynchronous processing using `CompletionStage`:

```
import javax.ws.rs.Suspend;
import javax.ws.rs.core.AsynchronousResponse;

@Path("/")
public class SimpleResource
{

   @GET
   @Path("basic")
   @Produces("text/plain")
   public CompletionStage<Response> getBasic() throws Exception
   {
      final CompletableFuture<Response> response = new CompletableFuture<>();
      Thread t = new Thread()
```

```
      {
         @Override
         public void run()
         {
            try
            {
               Response jaxrs = Response.ok("basic").type(MediaType.TEXT_PLAIN).build();
               response.complete(jaxrs);
            }
            catch (Exception e)
            {
               response.completeExceptionally(e);
            }
         }
      };
      t.start();
      return response;
   }
}
```

Note

RESTEasy supports more reactive types for asynchronous programming.

37.3. Asynchronous filters

It is possible to write filters that also turn the request asynchronous. Whether or not filters turned the request asynchronous before execution of your method makes absolutely no difference to your method: it does not need to be declared asynchronous in order to function as specified. Synchronous methods and asynchronous methods will work as specified by the spec.

37.4. Asynchronous IO

Some backends support asynchronous IO operations (Servlet, Undertow, Vert.x, Quarkus, Netty), which are exposed using the `AsyncOutputStream` subtype of `OutputStream`. It includes async variants for writing and flushing the stream.

Some backends have what is called an "Event Loop Thread", which is a thread responsible for doing all IO operations. Those backends require the Event Loop Thread to never be blocked, because it does IO for every other thread. Those backends typically require Jakarta RESTful Web Services endpoints to be invoked on worker threads, to make sure they never block the Event Loop Thread.

Sometimes, with Async programming, it is possible for asynchronous Jakarta RESTful Web Services requests to be resumed from the Event Loop Thread. As a result, Jakarta RESTful Web Services will attempt to serialise the response and send it to the client. But Jakarta RESTful Web Services is written using "Blocking IO" mechanics, such as `OutputStream` (used by `MessageBody-`

`Writer` and `WriterInterceptor`), which means that sending the response will block the current thread until the response is received. This would work on a worker thread, but if it happens on the Event Loop Thread it will block it and prevent it from sending the response, resulting in a deadlock.

As a result, we've decided to support and expose Async IO interfaces in the form of `AsyncOutputStream`, `AsyncMessageBodyWriter` and `AsyncWriterInterceptor`, to allow users to write Async IO applications in RESTEasy.

Most built-in `MessageBodyWriter` and `WriterInterceptor` support Async IO, with the notable exceptions of:

- `HtmlRenderableWriter`, which is tied to servlet APIs
- `ReaderProvider`
- `StreamingOutputProvider`: use `AsyncStreamingOutput` instead
- `GZIPEncodingInterceptor`

Async IO will be preferred if the following conditions are met:

- The backend supports it
- The writer supports it
- All writer interceptors support it

If those conditions are not met, and you attempt to use Blocking IO on an Event Loop Thread (as determined by the backend), then an exception will be thrown.

Chapter 38. Asynchronous Job Service

The RESTEasy Asynchronous Job Service is an implementation of the Asynchronous Job pattern defined in O'Reilly's "Restful Web Services" book. The idea of it is to bring asynchronicity to a synchronous protocol.

38.1. Using Async Jobs

While HTTP is a synchronous protocol it does have a faint idea of asynchronous invocations. The HTTP 1.1 response code 202, "Accepted" means that the server has received and accepted the response for processing, but the processing has not yet been completed. The RESTEasy Asynchronous Job Service builds around this idea.

```
POST http://example.com/myservice?asynch=true
```

For example, if you make the above post with the asynch query parameter set to true, RESTEasy will return a 202, "Accepted" response code and run the invocation in the background. It also sends back a Location header with a URL pointing to where the response of the background method is located.

```
HTTP/1.1 202 Accepted
Location: http://example.com/asynch/jobs/3332334
```

The URI will have the form of:

```
/asynch/jobs/{job-id}?wait={millisconds}|nowait=true
```

You can perform the GET, POST, and DELETE operations on this job URL. GET returns whatever the Jakarta RESTful Web Services resource method you invoked returned as a response if the job was completed. If the job has not completed, this GET will return a response code of 202, Accepted. Invoking GET does not remove the job, so you can call it multiple times. When RESTEasy's job queue gets full, it will evict the least recently used job from memory. You can manually clean up after yourself by calling DELETE on the URI. POST does a read of the JOB response and will remove the JOB it has been completed.

Both GET and POST allow you to specify a maximum wait time in milliseconds, a "wait" query parameter. Here's an example:

```
POST http://example.com/asynch/jobs/122?wait=3000
```

If you do not specify a "wait" parameter, the GET or POST will not wait at all if the job is not complete.

NOTE!! While you can invoke GET, DELETE, and PUT methods asynchronously, this breaks the HTTP 1.1 contract of these methods. While these invocations may not change the state of the resource if invoked more than once, they do change the state of the server as new Job entries with each invocation. If you want to be a purist, stick with only invoking POST methods asynchronously.

Security NOTE! RESTEasy role-based security (annotations) does not work with the Asynchronous Job Service. You must use XML declarative security within your web.xml file. Why? It is impossible to implement role-based security portably. In the future, we may have specific JBoss integration, but will not support other environments.

NOTE. A `SecureRandom` object is used to generate unique job ids. For security purposes, the `SecureRandom` is periodically reseeded. By default, it is reseeded after 100 uses. This value may be configured with the servlet init parameter "resteasy.secure.random.max.use".

38.2. Oneway: Fire and Forget

RESTEasy also supports the notion of fire and forget. This will also return a 202, Accepted response, but no Job will be created. This is as simple as using the oneway query parameter instead of asynch. For example:

```
POST http://example.com/myservice?oneway=true
```

Security NOTE! RESTEasy role-based security (annotations) does not work with the Asynchronous Job Service. You must use XML declaritive security within your web.xml file. Why? It is impossible to implement role-based security portably. In the future, we may have specific JBoss integration, but will not support other environments.

38.3. Setup and Configuration

You must enable the Asynchronous Job Service, as it is not turned on by default. If the relevant configuration properties are configured in web.xml, it would look like the following:

```
```

```
<web-app>
    <!-- enable the Asynchronous Job Service -->
    <context-param>
        <param-name>resteasy.async.job.service.enabled</param-name>
        <param-value>true</param-value>
    </context-param>

    <!-- The next context parameters are all optional.
         Their default values are shown as example param-values -->

    <!-- How many jobs results can be held in memory at once? -->
    <context-param>
        <param-name>resteasy.async.job.service.max.job.results</param-name>
        <param-value>100</param-value>
    </context-param>

    <!-- Maximum wait time on a job when a client is querying for it -->
    <context-param>
        <param-name>resteasy.async.job.service.max.wait</param-name>
        <param-value>300000</param-value>
    </context-param>

    <!-- Thread pool size of background threads that run the job -->
    <context-param>
        <param-name>resteasy.async.job.service.thread.pool.size</param-name>
        <param-value>100</param-value>
    </context-param>

    <!-- Set the base path for the Job uris -->
    <context-param>
        <param-name>resteasy.async.job.service.base.path</param-name>
        <param-value>/asynch/jobs</param-value>
    </context-param>

    ...
</web-app>
```

See Section 3.4, “Configuration” for more information about application configuration.

Chapter 39. Asynchronous Injection

Pluggable Asynchronous Injection, also referred to as Asynch Injection, is a feature that allows users to create custom injectable asynchronous types. For example it is now possible to declare an injector for `Single<Foo>` and inject it into an endpoint as a class variable or as a method parameter using `@Context Foo`. The response will be made asynchronous automatically and the resource method will only be invoked once the `Single<Foo>` object is resolved to `Foo`. Resolution is done in a non-blocking manner.

Note. Asynch injection is only attempted at points where asynchronous injection is permitted, such as on resource creation and resource method invocation. It is not enabled at points where the API does not allow for suspending the request, for example on `ResourceContext.getResource(Foo.class)`.

39.1. org.jboss.resteasy.spi.ContextInjector Interface

The `org.jboss.resteasy.spi.ContextInjector` interface must be implemented on any custom asynch injector object. The implementation class must be tagged with the `@Provider` annotation.

```
/**
 * @param <WrappedType> A class that wraps a data type or data object
 *                            (e.g. Single<Foo>)
 * @param <UnwrappedType> The data type or data object declared in the
 *                              WrappedType (e.g. Foo)
*/
public interface ContextInjector<WrappedType, UnwrappedType> {
/**
 * This interface allows users to create custom injectable asynchronous types.
 *
 * Asynch injection is only attempted at points where asynchronous injection is
 * permitted, such as on resource creation and resource method invocation. It
 * is not enabled at points where the API does not allow for suspending the
 * request
 *
 * @param rawType
 * @param genericType
 * @param annotations The annotation list is useful to parametrize the injection.
 * @return
 */
 public WrappedType resolve(
            Class<? extends WrappedType> rawType,
            Type genericType,
            Annotation[] annotations);
  }
```

39.2. Single<Foo> Example

```
package my.test;

public class Foo {
   private String value = "PRE-SET-VALUE";

   public void setValue(String s) {
      this.value = s;
   }

   public String getValue() {
      return this.value;
   }
}
```

```
package my.test.asyc.resources;

import io.reactivex.Single;
import javax.ws.rs.ext.Provider;
import org.jboss.resteasy.spi.ContextInjector;
import my.test.Foo;

@Provider
public class FooAsychInjectorProvider implements
            ContextInjector<Single<Foo>,Foo> {

   public Single<Foo> resolve(Class<? extends Single<Foo>> rawType,
            Type genericType,
            Annotation[] annotations)
   {
      Foo value = new Foo();
      return Single.just(value.setValue("made it"));
   }
}
```

39.3. Async Injector With Annotations Example

A convenience interface to provide annotation parameter designators

```
@Retention(RUNTIME)
@Target({ FIELD, METHOD, PARAMETER })
public @interface AsyncInjectionPrimitiveInjectorSpecifier
{
   public enum Type {
      VALUE, NULL, NO_RESULT;
```

```
   }

   Type value() default Type.VALUE;
}
```

```
@Provider
public class AsyncInjectionFloatInjector implements
            ContextInjector<CompletionStage<Float>, Float>
{

   @Override
   public CompletionStage<Float> resolve(
      Class<? extends CompletionStage<Float>> rawType,
            Type genericType,
            Annotation[] annotations)
    {
       for (Annotation annotation : annotations)
       {
           if(annotation.annotationType() ==
              AsyncInjectionPrimitiveInjectorSpecifier.class) {
             AsyncInjectionPrimitiveInjectorSpecifier.Type value =
               ((AsyncInjectionPrimitiveInjectorSpecifier)annotation).value();
             switch(value) {
               case NO_RESULT:
                  return null;
               case NULL:
                  return CompletableFuture.completedFuture(null);
               case VALUE:
                  return CompletableFuture.completedFuture(4.2f);
             }
             break;
          }
       }
       return CompletableFuture.completedFuture(4.2f);
    }
}
```

Chapter 40. Reactive programming support

With version 2.1, the Jakarta RESTful Web Services specification (https://jcp.org/en/jsr/detail?id=370) takes its first steps into the world of **Reactive Programming**. There are many discussions of reactive programming on the internet, and a general introduction is beyond the scope of this document, but there are a few things worth discussing. Some primary aspects of reactive programming are the following:

- Reactive programming supports the declarative creation of rich computational structures. The representations of these structures can be passed around as first class objects such as method parameters and return values.
- Reactive programming supports both synchronous and asynchronous computation, but it is particularly helpful in facilitating, at a relatively high level of expression, asynchronous computation. Conceptually, asynchronous computation in reactive program typically involves pushing data from one entity to another, rather than polling for data.

40.1. CompletionStage

In Java 1.8 and Jakarta RESTful Web Services, the support for reactive programming is fairly limited. Java 1.8 introduces the interface `java.util.concurrent.CompletionStage`, and Jakarta RESTful Web Services mandates support for the `javax.ws.rs.client.CompletionStageRxInvoker`, which allows a client to obtain a response in the form of a `CompletionStage`.

One implementation of `CompletionStage` is the `java.util.concurrent.CompleteableFuture`. For example:

```
@Test
public void testCompletionStage() throws Exception {
   CompletionStage<String> stage = getCompletionStage();
   log.info("result: " + stage.toCompletableFuture().get());
}

private CompletionStage<String> getCompletionStage() {
   CompletableFuture<String> future = new CompletableFuture<String>();
   future.complete("foo");
   return future;
}
```

Here, a `CompleteableFuture` is created with the value "foo", and its value is extracted by the method `CompletableFuture.get()`. That's fine, but consider the altered version:

```
@Test
public void testCompletionStageAsync() throws Exception {
   log.info("start");
   CompletionStage<String> stage = getCompletionStageAsync();
   String result = stage.toCompletableFuture().get();
   log.info("do some work");
   log.info("result: " + result);
}

private CompletionStage<String> getCompletionStageAsync() {
   CompletableFuture<String> future = new CompletableFuture<String>();
   Executors.newCachedThreadPool().submit(() -> {sleep(2000); future.complete("foo");});
   return future;
}

private void sleep(long l) {
   try {
      Thread.sleep(l);
   } catch (InterruptedException e) {
      e.printStackTrace();
   }
}
```

with output something like:

```
3:10:51 PM INFO: start
3:10:53 PM INFO: do some work
3:10:53 PM INFO: result: foo
```

It also works, but it illustrates the fact that `CompletableFuture.get()` is a blocking call. The `CompletionStage` is constructed and returned immediately, but the value isn't returned for two seconds. A version that is more in the spirit of the reactive style is:

```
@Test
public void testCompletionStageAsyncAccept() throws Exception {
   log.info("start");
   CompletionStage<String> stage = getCompletionStageAsync();
   stage.thenAccept((String s) -> log.info("s: " + s));
   log.info("do some work");
   ...
}
```

In this case, the lambda (String s) -> log.info("s: " + s) is registered with the `CompletionStage` as a "subscriber", and, when the `CompletionStage` eventually has a value, that value is passed to the lambda. Note that the output is something like

```
3:23:05 INFO: start
3:23:05 INFO: do some work
3:23:07 INFO: s: foo
```

Executing `CompletionStage`s asynchronously is so common that there are several supporting convenience methods. For example:

```
@Test
public void testCompletionStageSupplyAsync() throws Exception {
   CompletionStage<String> stage = getCompletionStageSupplyAsync();;
   stage.thenAccept((String s) -> log.info("s: " + s));
}

private CompletionStage<String> getCompletionStageSupplyAsync() {
   return CompletableFuture.supplyAsync(() -> "foo");
}
```

The static method `ComputableFuture.supplyAsync()` creates a `ComputableFuture`, the value of which is supplied asynchronously by the lambda () -> "foo", running, by default, in the default pool of `java.util.concurrent.ForkJoinPool`.

One final example illustrates a more complex computational structure:

```
@Test
public void testCompletionStageComplex() throws Exception {
   ExecutorService executor = Executors.newCachedThreadPool();
   CompletionStage<String> stage1 = getCompletionStageSupplyAsync1("foo", executor);
   CompletionStage<String> stage2 = getCompletionStageSupplyAsync1("bar", executor);
    CompletionStage<String> stage3 = stage1.thenCombineAsync(stage2, (String s, String t) -> s
 + t, executor);
   stage3.thenAccept((String s) -> log.info("s: " + s));
}

private   CompletionStage<String>   getCompletionStageSupplyAsync1(String   s,   ExecutorService
 executor) {
   return CompletableFuture.supplyAsync(() -> s, executor);
}
```

`stage1` returns "foo", `stage2` returns "bar", and `stage3`, which runs when both `stage1` and `stage2` have completed, returns the concatenation of "foo" and "bar". Note that, in this example, an explict `ExecutorService` is provided for asynchronous processing.

40.2. CompletionStage in Jakarta RESTful Web Services

On the client side, the Jakarta RESTful Web Services specification mandates an implementation of the interface `javax.ws.rs.client.CompletionStageRxInvoker`:

```
public interface CompletionStageRxInvoker extends RxInvoker<CompletionStage> {

    @Override
    public CompletionStage<Response> get();

    @Override
    public <T> CompletionStage<T> get(Class<T> responseType);

    @Override
    public <T> CompletionStage<T> get(GenericType<T> responseType);
    ...
```

That is, there are invocation methods for the standard HTTP verbs, just as in the standard `javax.ws.rs.client.SyncInvoker`. A `CompletionStageRxInvoker` is obtained by calling `rx()` on a `javax.ws.rs.client.Invocation.Builder`, which extends `SyncInvoker`. For example,

```
Invocation.Builder builder = client.target(generateURL("/get/string")).request();
CompletionStageRxInvoker invoker = builder.rx(CompletionStageRxInvoker.class);
CompletionStage<Response> stage = invoker.get();
Response response = stage.toCompletableFuture().get();
log.info("result: " + response.readEntity(String.class));
```

or

```
CompletionStageRxInvoker          invoker          =          client.target(generateURL("/get/
string")).request().rx(CompletionStageRxInvoker.class);
CompletionStage<String> stage = invoker.get(String.class);
String s = stage.toCompletableFuture().get();
log.info("result: " + s);
```

On the server side, the Jakarta RESTful Web Services specification requires support for resource methods with return type `CompletionStage<T>`. For example,

```
@GET
@Path("get/async")
public CompletionStage<String> longRunningOpAsync() {
```

```
   CompletableFuture<String> cs = new CompletableFuture<>();
   executor.submit(
      new Runnable() {
         public void run() {
            executeLongRunningOp();
            cs.complete("Hello async world!");
         }
      });
   return cs;
}
```

The way to think about `longRunningOpAsync()` is that it is asynchronously creating and returning a `String`. After `cs.complete()` is called, the server will return the `String` "Hello async world!" to the client.

An important thing to understand is that the decision to produce a result asynchronously on the server and the decision to retrieve the result asynchronously on the client are independent. Suppose that there is also a resource method

```
@GET
@Path("get/sync")
public String longRunningOpSync() {
   return "Hello async world!";
}
```

Then all three of the following invocations are valid:

```
public void testGetStringAsyncAsync() throws Exception {
   CompletionStageRxInvoker invoker = client.target(generateURL("/get/async")).request().rx();
   CompletionStage<String> stage = invoker.get(String.class);
   log.info("s: " + stage.toCompletableFuture().get());
}
```

```
public void testGetStringSyncAsync() throws Exception {
   Builder request = client.target(generateURL("/get/async")).request();
   String s = request.get(String.class);
   log.info("s: " + s);
}
```

and

```
public void testGetStringAsyncSync() throws Exception {
   CompletionStageRxInvoker invoker = client.target(generateURL("/get/sync")).request().rx();
```

```
   CompletionStage<String> stage = invoker.get(String.class);
   log.info("s: " + stage.toCompletableFuture().get());
}
```

Note

`CompletionStage` in Jakarta RESTful Web Services is also discussed in the chapter Asynchronous HTTP Request Processing.

Note

Since running code asynchronously is so common in this context, it is worth pointing out that objects obtained by way of the annotation `@Context` or by way of calling `ResteasyContext.getContextData()` are sensitive to the executing thread. For example, given resource method

```
@GET
@Path("test")
@Produces("text/plain")
public CompletionStage<String> text(@Context HttpRequest request) {
   System.out.println("request (inline): " + request);
                  System.out.println("application      (inline):      "      +
 ResteasyContext.getContextData(Application.class));
   CompletableFuture<String> cs = new CompletableFuture<>();
   ExecutorService executor = Executors.newSingleThreadExecutor();
   executor.submit(
         new Runnable() {
            public void run() {
               try {
                  System.out.println("request (async): " + request);
                                    System.out.println("application  (async):  "  +
 ResteasyContext.getContextData(Application.class));
                  cs.complete("hello");
               } catch (Exception e) {
                  e.printStackTrace();
               }
            }
         });
   return cs;
}
```

the output will look something like

```
application    (inline):    org.jboss.resteasy.experiment.Test1798CompletionStage
$TestApp@23c57474
```

```
request                                                                    (inline):
 org.jboss.resteasy.plugins.server.servlet.Servlet3AsyncHttpRequest@2ce23138
application (async): null
org.jboss.resteasy.spi.LoggableFailure: RESTEASY003880: Unable to find contextual
 data of type: org.jboss.resteasy.spi.HttpRequest
```

The point is that it is the developer's responsibility to extract information from these context objects in advance. For example:

```
@GET
@Path("test")
@Produces("text/plain")
public CompletionStage<String> text(@Context HttpRequest req) {
   System.out.println("request (inline): " + request);
                  System.out.println("application      (inline):      "      +
 ResteasyContext.getContextData(Application.class));
   CompletableFuture<String> cs = new CompletableFuture<>();
   ExecutorService executor = Executors.newSingleThreadExecutor();
   final String httpMethodFinal = request.getHttpMethod();
                    final       Map<String,       Object>       mapFinal       =
 ResteasyContext.getContextData(Application.class).getProperties();
   executor.submit(
         new Runnable() {
            public void run() {
               System.out.println("httpMethod (async): " + httpMethodFinal);
               System.out.println("map (async): " + mapFinal);
               cs.complete("hello");
            }
         });
   return cs;
}
```

Alternatively, you can use RESTEasy's support of MicroProfile Context Propagation [https://github.com/eclipse/microprofile-context-propagation] by using `ThreadContext.contextualRunnable` around your `Runnable`, which will take care of capturing and restoring all registered contexts (you will need to import the `org.jboss.resteasy.microprofile:microprofile-context-propagation` module):

```
@GET
@Path("test")
@Produces("text/plain")
public CompletionStage<String> text(@Context HttpRequest req) {
   System.out.println("request (inline): " + request);
                  System.out.println("application      (inline):      "      +
 ResteasyContext.getContextData(Application.class));
   CompletableFuture<String> cs = new CompletableFuture<>();
   ThreadContext threadContext = ThreadContext.builder()
                                        .propagated(ThreadContext.ALL_REMAINING)
                                                  .unchanged()
```

```
                                                    .cleared()
                                                    .build();
   ExecutorService executor = Executors.newSingleThreadExecutor();
   executor.submit(
         threadContext.contextualRunnable(new Runnable() {
            public void run() {
               try {
                  System.out.println("request (async): " + request);
                                 System.out.println("application  (async):  "  +
 ResteasyContext.getContextData(Application.class));
                  cs.complete("hello");
               } catch (Exception e) {
                  e.printStackTrace();
               }
            }
         }));
   return cs;
}
```

As another alternative you can use the RESTEasy SPI's `ContextualExecutor` if the MicroProfile Context Propagation is not available. This requires a dependency on `org.jboss.resteasy:resteasy-core`.

```
@GET
@Path("test")
@Produces(MediaType.TEXT_PLAIN)
public CompletionStage<String> text(@Context UriInfo uriInfo) {
  CompletableFuture<String> cs = new CompletableFuture<>();
  ExecutorService executor = ContextualExecutors.threadPool();
  executor.submit(() -> {
     try {
        cs.complete("hello from: " + uriInfo.getAbsolutePath());
     } catch (Exception e) {
        e.printStackTrace();
     }
  });
  return cs;
}
```

40.3. Beyond CompletionStage

The picture becomes more complex and interesting when sequences are added. A `CompletionStage` holds no more than one potential value, but other reactive objects can hold multiple, even unlimited, values. Currently, most Java implementations of reactive programming are based on the project Reactive Streams (http://www.reactive-streams.org/), which defines a set of four interfaces and a specification, in the form of a set of rules, describing how they interact:

```
public interface Publisher<T> {
```

```
    public void subscribe(Subscriber<? super T> s);
}

public interface Subscriber<T> {
    public void onSubscribe(Subscription s);
    public void onNext(T t);
    public void onError(Throwable t);
    public void onComplete();
}

public interface Subscription {
    public void request(long n);
    public void cancel();
}

public interface Processor<T, R> extends Subscriber<T>, Publisher<R> {
}
```

A `Producer` pushes objects to a `Subscriber`, a `Subscription` mediates the relationship between the two, and a `Processor` which is derived from both, helps to construct pipelines through which objects pass.

One important aspect of the specification is flow control, the ability of a `Suscriber` to control the load it receives from a `Producer` by calling `Suscription.request()`. The general term in this context for flow control is **backpressure**.

There are a number of implementations of Reactive Streams, including

1. **RxJava**: https://github.com/ReactiveX/RxJava (end of life, superceded by RxJava 2)
2. **RxJava 2**: https://github.com/ReactiveX/RxJava
3. **Reactor**: http://projectreactor.io/
4. **Flow**: https://community.oracle.com/docs/DOC-1006738/ [https://community.oracle.com/docs/DOC-1006738]: (Java JDK 9+)

RESTEasy currently supports RxJava (deprecated) and RxJava2.

40.4. Pluggable reactive types: RxJava 2 in RESTEasy

Jakarta RESTful Web Services doesn't currently require support for any Reactive Streams implementations, but it does allow for extensibility to support various reactive libraries. RESTEasy's optional module `resteasy-rxjava2` adds support for RxJava 2 [https://github.com/ReactiveX/RxJava].

More in details, `resteasy-rxjava2` contributes support for reactive types `io.reactivex.Single`, `io.reactivex.Flowable`, and `io.reactivex.Observable`. Of these, `Single` is similar to `CompletionStage` in that it holds at most one potential value. `Flowable` implements `io.reactivex.Publisher`, and `Observable` is very similar to `Flowable` except that it doesn't support backpressure. So, if you import `resteasy-rxjava2`, you can just start returning these reactive types from your resource methods on the server side and receiving them on the client side.

Note

When you use RESTEasy's modules for RxJava, the reactive contexts are automatically propagated to all supported RxJava types, which means you don't need to worry about `@Context` injection not working within RxJava lambdas, contrary to `CompletionStage` (as previously noted).

1. Server side

Given the class `Thing`, which can be represented in JSON:

```
public class Thing {

   private String name;

   public Thing() {
   }

   public Thing(String name) {
      this.name = name;
   }
   ...
}
```

the method `postThingList()` in the following is a valid resource method:

...

```
@POST
@Path("post/thing/list")
@Produces(MediaType.APPLICATION_JSON)
@Stream
public Flowable<List<Thing>> postThingList(String s) {
   return buildFlowableThingList(s, 2, 3);
}

static Flowable<List<Thing>> buildFlowableThingList(String s, int listSize, int elementSize) {
   return Flowable.create(
      new FlowableOnSubscribe<List<Thing>>() {

         @Override
         public void subscribe(FlowableEmitter<List<Thing>> emitter) throws Exception {
            for (int i = 0; i < listSize; i++) {
               List<Thing> list = new ArrayList<Thing>();
               for (int j = 0; j < elementSize; j++) {
                  list.add(new Thing(s));
               }
               emitter.onNext(list);
            }
```

```
            emitter.onComplete();
          }
      },
      BackpressureStrategy.BUFFER);
}
```

The somewhat imposing method `buildFlowableThingList()` probably deserves some explanation. First,

```
Flowable<List<Thing>>         Flowable.create(FlowableOnSubscribe<List<Thing>>         source,
 BackpressureStrategy mode);
```

creates a `Flowable<List<Thing>>` by describing what should happen when the `Flowable<List<Thing>>` is subscribed to. `FlowableEmitter<List<Thing>>` extends `io.reactivex.Emitter<List<Thing>>`:

```
/**
 * Base interface for emitting signals in a push-fashion in various generator-like source
 * operators (create, generate).
 *
 * @param <T> the value type emitted
 */
public interface Emitter<T> {

    /**
     * Signal a normal value.
     * @param value the value to signal, not null
     */
    void onNext(@NonNull T value);

    /**
     * Signal a Throwable exception.
     * @param error the Throwable to signal, not null
     */
    void onError(@NonNull Throwable error);

    /**
     * Signal a completion.
     */
    void onComplete();
}
```

and `FlowableOnSubscribe` uses a `FlowableEmitter` to send out values from the `Flowable<List<Thing>>`:

```
/**
 * A functional interface that has a {@code subscribe()} method that receives
```

```
 * an instance of a {@link FlowableEmitter} instance that allows pushing
 * events in a backpressure-safe and cancellation-safe manner.
 *
 * @param <T> the value type pushed
 */
public interface FlowableOnSubscribe<T> {

    /**
     * Called for each Subscriber that subscribes.
     * @param e the safe emitter instance, never null
     * @throws Exception on error
     */
    void subscribe(@NonNull FlowableEmitter<T> e) throws Exception;
}
```

So, what will happen when a subscription to the `Flowable<List<Thing>>` is created is, the `FlowableEmitter.onNext()` will be called, once for each `<List<Thing>>` created, followed by a call to `FlowableEmitter.onComplete()` to indicate that the sequence has ended. Under the covers, RESTEasy subscribes to the `Flowable<List<Thing>>` and handles each element passed in by way of `onNext()`.

2. Client side

On the client side, Jakarta RESTful Web Services supports extensions for reactive classes by adding the method

```
/**
 * Access a reactive invoker based on a {@link RxInvoker} subclass provider. Note
 * that corresponding {@link RxInvokerProvider} must be registered in the client runtime.
 *
 * This method is an extension point for Jakarta RESTful Web Services implementations to support
 other types
 * representing asynchronous computations.
 *
 * @param clazz {@link RxInvoker} subclass.
 * @return reactive invoker instance.
 * @throws IllegalStateException when provider for given class is not registered.
 * @see javax.ws.rs.client.Client#register(Class)
 * @since 2.1
 */
public <T extends RxInvoker> T rx(Class<T> clazz);
```

to interface `javax.ws.rs.client.Invocation.Builder`. Resteasy module `resteasy-rxjava2` adds support for classes:

1. `org.jboss.resteasy.rxjava2.SingleRxInvoker`,
2. `org.jboss.resteasy.rxjava2.FlowableRxInvoker`
3. `org.jbosss.resteasy.rxjava2.ObservableRxInvoker`

which allow accessing `Singles`, `Observables`, and `Flowables` on the client side.

For example, given the resource method `postThingList()` above, a `Flowable<List<Thing>>` can be retrieved from the server by calling

```
@SuppressWarnings("unchecked")
@Test
public void testPostThingList() throws Exception {
   CountDownLatch latch = new CountdownLatch(1);
               FlowableRxInvoker     invoker     =     client.target(generateURL("/post/thing/
list")).request().rx(FlowableRxInvoker.class);
     Flowable<List<Thing>> flowable = (Flowable<List<Thing>>) invoker.post(Entity.entity("a",
 MediaType.TEXT_PLAIN_TYPE), new GenericType<List<Thing>>() {});
   flowable.subscribe(
         (List<?> l) -> thingListList.add(l),
         (Throwable t) -> latch.countDown(),
         () -> latch.countDown());
   latch.await();
   Assert.assertEquals(aThingListList, thingListList);
}
```

where `aThingListList` is

```
[[Thing[a], Thing[a], Thing[a]], [Thing[a], Thing[a], Thing[a]]]
```

Note the call to `Flowable.suscribe()`. On the server side, RESTEasy subscribes to a returning `Flowable` in order to receive its elements and send them over the wire. On the client side, the user subscribes to the `Flowable` in order to receive its elements and do whatever it wants to with them. In this case, three lambdas determine what should happen 1) for each element, 2) if a `Throwable` is thrown, and 3) when the `Flowable` is done passing elements.

3. Representation on the wire

Neither Reactive Streams nor Jakarta RESTful Web Services have anything to say about representing reactive types on the network. RESTEasy offers a number of representations, each suitable for different circumstances. The wire protocol is determined by 1) the presence or absence of the `@Stream` annotation on the resource method, and 2) the value of the `value` field in the `@Stream` annotation:

```
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface Stream
{
   public enum MODE {RAW, GENERAL};
   public String INCLUDE_STREAMING_PARAMETER = "streaming";
```

```
   public MODE value() default MODE.GENERAL;
   public boolean includeStreaming() default false;
}
```

Note that `MODE.GENERAL` is the default value, so `@Stream` is equivalent to `@Stream(Stream.MODE.GENERAL)`.

No `@Stream` annotation on the resource method

Resteasy will collect every value until the stream is complete, then wrap them into a `java.util.List` entity and send to the client.

`@Stream(Stream.MODE.GENERAL)`

This case uses a variant of the SSE format, modified to eliminate some restrictions inherent in SSE. (See the specification at https://html.spec.whatwg.org/multipage/server-sent-events.html [https://html.spec.whatwg.org/multipage/server-sent-events.html] for details.) In particular, 1) SSE events are meant to hold text data, represented in character set UTF-8. In the general streaming mode, certain delimiting characters in the data ('\r', '\n', and '\') are escaped so that arbitrary binary data can be transmitted. Also, 2) the SSE specification requires the client to reconnect if it gets disconnected. If the stream is finite, reconnecting will induce a repeat of the stream, so SSE is really meant for unlimited streams. In general streaming mode, the client will close, rather than automatically reconnect, at the end of the stream. It follows that this mode is suitable for finite streams.

Note. The Content-Type header in general streaming mode is set to

```
applicaton/x-stream-general;"element-type=<element-type>"
```

where <element-type> is the media type of the data elements in the stream. The element media type is derived from the @Produces annotation. For example,

```
@GET
@Path("flowable/thing")
@Stream
@Produces("application/json")
public Flowable<Thing> getFlowable() { ... }
```

induces the media type

```
application/x-stream-general;"element-type=application/json"
```

```
```

which describes a stream of JSON elements.

`@Stream(Stream.MODE.RAW)`

In this case each value is written directly to the wire, without any formatting, as it becomes available. This is most useful for values that can be cut in pieces, such as strings, bytes, buffers, etc., and then re-concatenated on the client side. Note that without delimiters as in general mode, it isn't possible to reconstruct something like `List<List<String>>`.

Note. The Content-Type header in raw streaming mode is derived from the `@Produces` annotation. The `@Stream` annotation offers the possibility of an optional `MediaType` parameter called "streaming". The point is to be able to suggest that the stream of data emanating from the server is unbounded, i.e., that the client shouldn't try to read it all as a single byte array, for example. The parameter is set by explicitly setting the `@Stream` parameter `includeStreaming()` to `true`. For example,

```
@GET
@Path("byte/default")
@Produces("application/octet-stream;x=y")
@Stream(Stream.MODE.RAW)
public Flowable<Byte> aByteDefault() {
   return Flowable.fromArray((byte) 0, (byte) 1, (byte) 2);
}
```

induces the `MediaType` "application/octet-stream;x=y", and

```
@GET
@Path("byte/true")
@Produces("application/octet-stream;x=y")
@Stream(value=Stream.MODE.RAW, includeStreaming=true)
public Flowable<Byte> aByteTrue() {
   return Flowable.fromArray((byte) 0, (byte) 1, (byte) 2);
}
```

induces the `MediaType` "application/octet-stream;x=y;streaming=true".

Note that browsers such as Firefox and Chrome seem to be comfortable with reading unlimited streams without any additional hints.

4. Examples.

Example 1.

```
```

```
@POST
@Path("post/thing/list")
@Produces(MediaType.APPLICATION_JSON)
@Stream(Stream.MODE.GENERAL)
public Flowable<List<Thing>> postThingList(String s) {
   return buildFlowableThingList(s, 2, 3);
}
...
@SuppressWarnings("unchecked")
@Test
public void testPostThingList() throws Exception {
   CountDownLatch latch = new CountdownLatch(1);
               FlowableRxInvoker     invoker     =     client.target(generateURL("/post/thing/
list")).request().rx(FlowableRxInvoker.class);
    Flowable<List<Thing>> flowable = (Flowable<List<Thing>>) invoker.post(Entity.entity("a",
 MediaType.TEXT_PLAIN_TYPE), new GenericType<List<Thing>>() {});
   flowable.subscribe(
         (List<?> l) -> thingListList.add(l),
         (Throwable t) -> latch.countDown(),
         () -> latch.countDown());
   latch.await();
   Assert.assertEquals(aThingListList, thingListList);
}
```

This is the example given previously, except that the mode in the `@Stream` annotation (which defaults to MODE.GENERAL) is given explicitly. In this scenario, the `Flowable` emits `<List<Thing>>` elements on the server, they are transmitted over the wire as SSE events:

```
data: [{"name":"a"},{"name":"a"},{"name":"a"}]
data: [{"name":"a"},{"name":"a"},{"name":"a"}]
```

and the `FlowableRxInvoker` reconstitutes a `Flowable` on the client side.

Example 2.

```
@POST
@Path("post/thing/list")
@Produces(MediaType.APPLICATION_JSON)
public Flowable<List<Thing>> postThingList(String s) {
   return buildFlowableThingList(s, 2, 3);
}
...
@Test
public void testPostThingList() throws Exception {
   Builder request = client.target(generateURL("/post/thing/list")).request();
    List<List<Thing>> list = request.post(Entity.entity("a", MediaType.TEXT_PLAIN_TYPE), new
 GenericType<List<List<Thing>>>() {});
   Assert.assertEquals(aThingListList, list);
}
```

In this scenario, in which the resource method has no `@Stream` annotation, the `Flowable` emits stream elements which are accumulated by the server until the `Flowable` is done, at which point the entire JSON list is transmitted over the wire:

```
[[{"name":"a"},{"name":"a"},{"name":"a"}],[{"name":"a"},{"name":"a"},{"name":"a"}]]
```

and the list is reconstituted on the client side by an ordinary invoker.

Example 3.

```
@GET
@Path("get/bytes")
@Produces(MediaType.APPLICATION_OCTET_STREAM)
@Stream(Stream.MODE.RAW)
public Flowable<byte[]> getBytes() {
   return Flowable.create(
      new FlowableOnSubscribe<byte[]>() {

         @Override
         public void subscribe(FlowableEmitter<byte[]> emitter) throws Exception {
            for (int i = 0; i < 3; i++) {
               byte[] b = new byte[10];
               for (int j = 0; j < 10; j++) {
                  b[j] = (byte) (i + j);
               }
               emitter.onNext(b);
            }
            emitter.onComplete();
         }
      },
      BackpressureStrategy.BUFFER);
}
...
@Test
public void testGetBytes() throws Exception {
   Builder request = client.target(generateURL("/get/bytes")).request();
   InputStream is = request.get(InputStream.class);
   int n = is.read();
   while (n > -1) {
      System.out.print(n);
      n = is.read();
   }
}
```

Here, the byte arrays are written to the network as they are created by the `Flowable`. On the network, they are concatenated, so the client sees one stream of bytes.

Note

Given that asynchronous code is common in this context, it is worth looking at the earlier Note.

5. Rx and SSE

Since general streaming mode and SSE share minor variants of the same wire protocol, they are, modulo the SSE restriction to character data, interchangeable. That is, an SSE client can connect to a resource method that returns a `Flowable` or an `Observable`, and a `FlowableRxInvoker`, for example, can connect to an SSE resource method.

Note. SSE requires a `@Produces("text/event-stream")` annotation, so, unlike the cases of raw and general streaming, the element media type cannot be derived from the `@Produces` annotation. To solve this problem, Resteasy introduces the

```
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface SseElementType
{
   public String value();
}
```

annotation, from which the element media type is derived.

Example 1.

```
@GET
@Path("eventStream/thing")
@Produces("text/event-stream")
@SseElementType("application/json")
public void eventStreamThing(@Context SseEventSink eventSink, @Context Sse sse) {
   new ScheduledThreadPoolExecutor(5).execute(() -> {
      try (SseEventSink sink = eventSink) {
         OutboundSseEvent.Builder  builder = sse.newEventBuilder();
         eventSink.send(builder.data(new Thing("e1")).build());
         eventSink.send(builder.data(new Thing("e2")).build());
         eventSink.send(builder.data(new Thing("e3")).build());
      }
   });
}
...
@SuppressWarnings("unchecked")
@Test
public void testFlowableToSse() throws Exception {
   CountDownLatch latch = new CountDownLatch(1);
   final AtomicInteger errors = new AtomicInteger(0);
```

```
               FlowableRxInvoker     invoker     =     client.target(generateURL("/eventStream/
thing")).request().rx(FlowableRxInvoker.class);
   Flowable<Thing> flowable = (Flowable<Thing>) invoker.get(Thing.class);
   flowable.subscribe(
      (Thing t) -> thingList.add(t),
      (Throwable t) -> errors.incrementAndGet(),
      () -> latch.countDown());
   boolean waitResult = latch.await(30, TimeUnit.SECONDS);
   Assert.assertTrue("Waiting for event to be delivered has timed out.", waitResult);
   Assert.assertEquals(0, errors.get());
   Assert.assertEquals(eThingList, thingList);
}
```

Here, a `FlowableRxInvoker` is connecting to an SSE resource method. On the network, the data looks like

```
data: {"name":"e1"}
data: {"name":"e2"}
data: {"name":"e3"}
```

Note that the character data is suitable for an SSE resource method.

Also, note that the `eventStreamThing()` method in this example induces the media type

```
    text/event-stream;element-type="application/json"
```

Example 2.

```
@GET
@Path("flowable/thing")
@Produces("text/event-stream")
@SseElementType("application/json")
public Flowable<Thing> flowableSSE() {
   return Flowable.create(
      new FlowableOnSubscribe<Thing>() {

         @Override
         public void subscribe(FlowableEmitter<Thing> emitter) throws Exception {
            emitter.onNext(new Thing("e1"));
            emitter.onNext(new Thing("e2"));
            emitter.onNext(new Thing("e3"));
            emitter.onComplete();
         }
      },
      BackpressureStrategy.BUFFER);
}
...
@Test
```

```
public void testSseToFlowable() throws Exception {
   final CountDownLatch latch = new CountDownLatch(3);
   final AtomicInteger errors = new AtomicInteger(0);
   WebTarget target = client.target(generateURL("/flowable/thing"));
   SseEventSource msgEventSource = SseEventSource.target(target).build();
   try (SseEventSource eventSource = msgEventSource)
   {
      eventSource.register(
         event -> {thingList.add(event.readData(Thing.class, MediaType.APPLICATION_JSON_TYPE));
 latch.countDown();},
         ex -> errors.incrementAndGet());
      eventSource.open();

      boolean waitResult = latch.await(30, TimeUnit.SECONDS);
      Assert.assertTrue("Waiting for event to be delivered has timed out.", waitResult);
      Assert.assertEquals(0, errors.get());
      Assert.assertEquals(eThingList, thingList);
   }
}
```

Here, an SSE client is connecting to a resource method that returns a `Flowable`. Again, the server is sending character data, which is suitable for the SSE client, and the data looks the same on the network.

6. To stream or not to stream

Whether or not it is appropriate to stream a list of values is a judgment call. Certainly, if the list is unbounded, then it isn't practical, or even possible, perhaps, to collect the entire list and send it at once. In other cases, the decision is less obvious.

Case 1. Suppose that all of the elements are producible quickly. Then the overhead of sending them independently is probably not worth it.

Case 2. Suppose that the list is bounded but the elements will be produced over an extended period of time. Then returning the initial elements when they become available might lead to a better user experience.

Case 3. Suppose that the list is bounded and the elements can be produced in a relatively short span of time but only after some delay. Here is a situation that illustrates the fact that asynchronous reactive processing and streaming over the network are independent concepts. In this case it's worth considering having the resource method return something like `CompletionStage<List<Thing>>` rather than `Flowable<List<Thing>>`. This has the benefit of creating the list asynchronously but, once it is available, sending it to the client in one piece.

40.5. Proxies

Proxies, discussed in RESTEasy Proxy Framework, are a RESTEasy extension that supports a natural programming style in which generic Jakarta RESTful Web Services invoker calls are replaced by application specific interface calls. The proxy framework is extended to include both `CompletionStage` and the RxJava2 types `Single`, `Observable`, and `Flowable`.

Example 1.

```
@Path("")
public interface RxCompletionStageResource {

   @GET
   @Path("get/string")
   @Produces(MediaType.TEXT_PLAIN)
   public CompletionStage<String> getString();
}

@Path("")
public class RxCompletionStageResourceImpl {

   @GET
   @Path("get/string")
   @Produces(MediaType.TEXT_PLAIN)
   public CompletionStage<String> getString() { .... }
}

public class RxCompletionStageProxyTest {

   private static ResteasyClient client;
   private static RxCompletionStageResource proxy;

   static {
      client = (ResteasyClient)ClientBuilder.newClient();
      proxy = client.target(generateURL("/")).proxy(RxCompletionStageResource.class);
   }

   @Test
   public void testGet() throws Exception {
      CompletionStage<String> completionStage = proxy.getString();
      Assert.assertEquals("x", completionStage.toCompletableFuture().get());
   }
}
```

Example 2.

```
public interface Rx2FlowableResource {

   @GET
   @Path("get/string")
   @Produces(MediaType.TEXT_PLAIN)
   @Stream
   public Flowable<String> getFlowable();
}

@Path("")
public class Rx2FlowableResourceImpl {

   @GET
   @Path("get/string")
```

```
   @Produces(MediaType.TEXT_PLAIN)
   @Stream
   public Flowable<String> getFlowable() { ... }
}

public class Rx2FlowableProxyTest {

   private static ResteasyClient client;
   private static Rx2FlowableResource proxy;

   static {
      client = (ResteasyClient)ClientBuilder.newClient();
      proxy = client.target(generateURL("/")).proxy(Rx2FlowableResource.class);
   }

   @Test
   public void testGet() throws Exception {
      Flowable<String> flowable = proxy.getFlowable();
      flowable.subscribe(
         (String o) -> stringList.add(o),
         (Throwable t) -> errors.incrementAndGet(),
         () -> latch.countDown());
      boolean waitResult = latch.await(30, TimeUnit.SECONDS);
      Assert.assertTrue("Waiting for event to be delivered has timed out.", waitResult);
      Assert.assertEquals(0, errors.get());
      Assert.assertEquals(xStringList, stringList);
   }
}
```

40.6. Adding extensions

RESTEasy implements a framework that supports extensions for additional reactive classes. To understand the framework, it is necessary to understand the existing support for `CompletionStage` and other reactive classes.

Server side. When a resource method returns a `CompletionStage`, RESTEasy subscribes to it using the class `org.jboss.resteasy.core.AsyncResponseConsumer.CompletionStageResponseConsumer`. When the `CompletionStage` completes, it calls `CompletionStageResponseConsumer.accept()`, which sends the result back to the client.

Support for `CompletionStage` is built in to RESTEasy, but it's not hard to extend that support to a class like `Single` by providing a mechanism for transforming a `Single` into a `CompletionStage`. In module resteasy-rxjava2, that mechanism is supplied by `org.jboss.resteasy.rxjava2.SingleProvider`, which implements interface `org.jboss.resteasy.spi.AsyncResponseProvider<Single<?>>`:

```
public interface AsyncResponseProvider<T> {
   public CompletionStage toCompletionStage(T asyncResponse);
}
```

Given `SingleProvider`, RESTEasy can take a `Single`, transform it into a `CompletionStage`, and then use `CompletionStageResponseConsumer` to handle the eventual value of the `Single`.

Similarly, when a resource method returns a streaming reactive class like `Flowable`, RESTEasy subscribes to it, receives a stream of data elements, and sends them to the client. `AsyncResponseConsumer` has several supporting classes, each of which implements a different mode of streaming. For example, `AsyncResponseConsumer.AsyncGeneralStreamingSseResponseConsumer` handles general streaming and SSE streaming. Subscribing is done by calling `org.reactivestreams.Publisher.subscribe()`, so a mechanism is needed for turning, say, a `Flowable` into a `Publisher`. That is, an implementation of `org.jboss.resteasy.spi.AsyncStreamProvider<Flowable>` is called for, where `AsyncStreamProvider` is defined:

```
public interface AsyncStreamProvider<T> {
   public Publisher toAsyncStream(T asyncResponse);
}
```

In module resteasy-rxjava2, `org.jboss.resteasy.FlowableProvider` provides that mechanism for `Flowable`. [Actually, that's not too hard since, in rxjava2, a `Flowable` *is* a `Provider`.]

So, on the server side, adding support for other reactive types can be done by declaring a `@Provider` for the interface `AsyncStreamProvider` (for streams) or `AsyncResponseProvider` (for single values), which both have a single method to convert the new reactive type into (respectively) a `Publisher` (for streams) or a `CompletionStage` (for single values).

Client side. The Jakarta RESTful Web Services specification imposes two requirements for support of reactive classes on the client side:

1. support for `CompletionStage` in the form of an implementation of the interface `javax.ws.rs.client.CompletionStageRxInvoker`, and
2. extensibility in the form of support for registering providers that implement

```
public interface RxInvokerProvider<T extends RxInvoker> {
    public boolean isProviderFor(Class<T> clazz);
    public T getRxInvoker(SyncInvoker syncInvoker, ExecutorService executorService);
}
```

   Once an `RxInvokerProvider` is registered, an `RxInvoker` can be requested by calling the `javax.ws.rs.client.Invocation.Builder` method

```
public <T extends RxInvoker> T rx(Class<T> clazz);
```

That `RxInvoker` can then be used for making an invocation that returns the appropriate reactive class. For example,

```
FlowableRxInvoker           invoker           =           client.target(generateURL("/get/
string")).request().rx(FlowableRxInvoker.class);
Flowable<String> flowable = (Flowable<String>) invoker.get();
```

RESTEasy provides partial support for implementing `RxInvokers`. For example, `SingleProvider`, mentioned above, also implements `org.jboss.resteasy.spi.AsyncClientResponseProvider<Single<?>>`, where `AsyncClientResponseProvider` is defined

```
public interface AsyncClientResponseProvider<T> {
   public T fromCompletionStage(CompletionStage<?> completionStage);
}
```

`SingleProvider`'s ability to turn a `CompletionStage` into a `Single` is used in the implementation of `org.jboss.resteasy.rxjava2.SingleRxInvokerImpl`.

The same concept might be useful in implementing other `RxInvokers`. Note, though, that `ObservableRxInvokerImpl` and `FlowableRxInvokerImpl` in module resteasy-rxjava2 are each derived directly from the SSE implementation.

Chapter 41. Embedded Containers

RESTEasy has a few different plugins for different embedabble HTTP and/or Servlet containers if use RESTEasy in a test environment, or within an environment where you do not want a Servlet engine dependency.

41.1. Undertow

Undertow is a new Servlet Container that is used by WildFly (JBoss Community Server). You can embed Undertow as you wish. Here's a a test that shows it in action.

```
import io.undertow.servlet.api.DeploymentInfo;
import org.jboss.resteasy.plugins.server.undertow.UndertowJaxrsServer;
import org.jboss.resteasy.test.TestPortProvider;
import org.junit.AfterClass;
import org.junit.Assert;
import org.junit.BeforeClass;
import org.junit.Test;

import javax.ws.rs.ApplicationPath;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.core.Application;
import java.util.HashSet;
import java.util.Set;

/**
 * @author <a href="mailto:bill@burkecentral.com">Bill Burke</a>
 * @version $Revision: 1 $
 */
public class UndertowTest
{
   private static UndertowJaxrsServer server;

   @Path("/test")
   public static class Resource
   {
      @GET
      @Produces("text/plain")
      public String get()
      {
         return "hello world";
      }
   }

   @ApplicationPath("/base")
   public static class MyApp extends Application
   {
      @Override
      public Set<Class<?>> getClasses()
```

```
      {
         HashSet<Class<?>> classes = new HashSet<Class<?>>();
         classes.add(Resource.class);
         return classes;
      }
   }

   @BeforeClass
   public static void init() throws Exception
   {
      server = new UndertowJaxrsServer().start();
   }

   @AfterClass
   public static void stop() throws Exception
   {
      server.stop();
   }

   @Test
   public void testApplicationPath() throws Exception
   {
      server.deployOldStyle(MyApp.class);
      Client client = ClientBuilder.newClient();
      String val = client.target(TestPortProvider.generateURL("/base/test"))
                         .request().get(String.class);
      Assert.assertEquals("hello world", val);
      client.close();
   }

   @Test
   public void testApplicationContext() throws Exception
   {
      server.deployOldStyle(MyApp.class, "/root");
      Client client = ClientBuilder.newClient();
      String val = client.target(TestPortProvider.generateURL("/root/test"))
                         .request().get(String.class);
      Assert.assertEquals("hello world", val);
      client.close();
   }

   @Test
   public void testDeploymentInfo() throws Exception
   {
      DeploymentInfo di = server.undertowDeployment(MyApp.class);
      di.setContextPath("/di");
      di.setDeploymentName("DI");
      server.deploy(di);
      Client client = ClientBuilder.newClient();
      String val = client.target(TestPortProvider.generateURL("/di/base/test"))
                         .request().get(String.class);
      Assert.assertEquals("hello world", val);
      client.close();
   }
}
```

41.2. Sun JDK HTTP Server

The Sun JDK comes with a simple HTTP server implementation (com.sun.net.httpserver.HttpServer) which you can run RESTEasy on top of.

```
HttpServer httpServer = HttpServer.create(new InetSocketAddress(port), 10);
contextBuilder = new HttpContextBuilder();
contextBuilder.getDeployment().getActualResourceClasses().add(SimpleResource.class);
HttpContext context = contextBuilder.bind(httpServer);
context.getAttributes().put("some.config.info", "42");
httpServer.start();

contextBuilder.cleanup();
httpServer.stop(0);
```

Create your HttpServer the way you want then use the org.jboss.resteasy.plugins.server.sun.http.HttpContextBuilder to initialize Resteasy and bind it to an HttpContext. The HttpContext attributes are available by injecting in a org.jboss.resteasy.spi.ResteasyConfiguration interface using @Context within your provider and resource classes.

Maven project you must include is:

```
<dependency>
    <groupId>org.jboss.resteasy</groupId>
    <artifactId>resteasy-jdk-http</artifactId>
    <version>5.0.10.Final</version>
</dependency>
```

41.3. Netty

RESTEasy has integration with the popular Netty project as well..

```
public static void start(ResteasyDeployment deployment) throws Exception
{
   netty = new NettyJaxrsServer();
   netty.setDeployment(deployment);
   netty.setPort(TestPortProvider.getPort());
   netty.setRootResourcePath("");
   netty.setSecurityDomain(null);
   netty.start();
}
```

Maven project you must include is:

```
<dependency>
    <groupId>org.jboss.resteasy</groupId>
    <artifactId>resteasy-netty4</artifactId>
    <version>5.0.10.Final</version>
</dependency>
```

41.4. Reactor-Netty

RESTEasy integrates with the reactor-netty project. This server adapter was created to pair with our reactor-netty based Jakarta RESTful Web Services client integration. Ultimately, if using reactor-netty for both the server and server-contained clients you will be able to do things like share the same event loop for both server and client calls.

```
 public static void start(ResteasyDeployment deployment) throws Exception
 {
    ReactorNettyJaxrsServer server =  new ReactorNettyJaxrsServer();
    server.setDeployment(new ResteasyDeploymentImpl());
    server.setDeployment(deployment);
    server.setPort(TestPortProvider.getPort());
    server.setRootResourcePath("");
    server.setSecurityDomain(null);
    server.start();
 }
```

Maven project you must include is:

```
<dependency>
    <groupId>org.jboss.resteasy</groupId>
    <artifactId>resteasy-reactor-netty</artifactId>
    <version>5.0.10.Final</version>
</dependency>
```

41.5. Vert.x

RESTEasy has integration with the popular Vert.x project as well..

```
 public static void start(VertxResteasyDeployment deployment) throws Exception
 {
    VertxJaxrsServer server = new VertxJaxrsServer();
    server.setDeployment(deployment);
```

```
        server.setPort(TestPortProvider.getPort());
        server.setRootResourcePath("");
        server.setSecurityDomain(null);
        server.start();
    }
```

Maven project you must include is:

```
<dependency>
    <groupId>org.jboss.resteasy</groupId>
    <artifactId>resteasy-vertx</artifactId>
    <version>5.0.10.Final</version>
</dependency>
```

The server will bootstrap its own Vert.x instance and Http server.

When a resource is called, it is done with the Vert.x Event Loop thread, keep in mind to not block this thread and respect the Vert.x programming model, see the related Vert.x manual page [http://vertx.io/docs/vertx-core/java/#_don_t_block_me].

Vert.x extends the RESTEasy registry to provide a new binding scope that creates resources per Event Loop:

```
VertxResteasyDeployment deployment = new VertxResteasyDeployment();
// Create an instance of resource per Event Loop
deployment.getRegistry().addPerInstanceResource(Resource.class);
```

The per instance binding scope caches the same resource instance for each event loop providing the same concurrency model than a verticle deployed multiple times.

Vert.x can also embed a RESTEasy deployment, making easy to use Jakarta RESTful Web Services annotated controller in Vert.x applications:

```
Vertx vertx = Vertx.vertx();
HttpServer server = vertx.createHttpServer();

// Set an handler calling Resteasy
server.requestHandler(new VertxRequestHandler(vertx, deployment));

// Start the server
server.listen(8080, "localhost");
```

Vert.x objects can be injected in annotated resources:

```
@GET
@Path("/somepath")
@Produces("text/plain")
public String context(
    @Context io.vertx.core.Context context,
    @Context io.vertx.core.Vertx vertx,
    @Context io.vertx.core.http.HttpServerRequest req,
    @Context io.vertx.core.http.HttpServerResponse resp) {
  return "the-response";
}
```

41.6. EmbeddedJaxrsServer

`EmbeddedJaxrsServer` is an interface provided to enable each embedded container wrapper class to configurate, start and stop its container in a standard fashion. Each of `UndertowJaxrsServer`, `SunHttpJaxrsServer`, `NettyJaxrsServer`, and `VertxJaxrsServer` implements `EmbeddedJaxrsServer`.

```
    public interface EmbeddedJaxrsServer<T> {
       T deploy();
       T start();
       void stop();
       ResteasyDeployment getDeployment();
       T setDeployment(ResteasyDeployment deployment);
       T setPort(int port);
       T setHostname(String hostname);
       T setRootResourcePath(String rootResourcePath);
       T setSecurityDomain(SecurityDomain sc);
    }
```

Chapter 42. Server-side Mock Framework

Although RESTEasy has an Embeddable Container, you may not be comfortable with the idea of starting and stopping a web server within unit tests (in reality, the embedded container starts in milli seconds), or you might not like the idea of using Apache HTTP Client or java.net.URL to test your code. RESTEasy provides a mock framework so that you can invoke on your resource directly.

```
import org.jboss.resteasy.mock.*;
...

      Dispatcher dispatcher = MockDispatcherFactory.createDispatcher();

      POJOResourceFactory noDefaults = new POJOResourceFactory(LocatingResource.class);
      dispatcher.getRegistry().addResourceFactory(noDefaults);

      {
         MockHttpRequest request = MockHttpRequest.get("/locating/basic");
         MockHttpResponse response = new MockHttpResponse();

         dispatcher.invoke(request, response);


         Assert.assertEquals(HttpServletResponse.SC_OK, response.getStatus());
         Assert.assertEquals("basic", response.getContentAsString());
      }
```

See the RESTEasy Javadoc for all the ease-of-use methods associated with MockHttpRequest, and MockHttpResponse.

Chapter 43. Securing Jakarta RESTful Web Services and RESTEasy

Because RESTEasy is deployed as a servlet, you must use standard web.xml constraints to enable authentication and authorization.

Unfortunately, web.xml constraints do not mesh very well with Jakarta RESTful Web Services in some situations. The problem is that web.xml URL pattern matching is very very limited. URL patterns in web.xml only support simple wildcards, so Jakarta RESTful Web Services resources like:

```
/{pathparam1}/foo/bar/{pathparam2}
```

Cannot be mapped as a web.xml URL pattern like:

```
/*/foo/bar/*
```

To get around this problem you will need to use the security annotations defined below on your Jakarta RESTful Web Services methods. You will still need to set up some general security constraint elements in web.xml to turn on authentication.

RESTEasy supports the @RolesAllowed, @PermitAll and @DenyAll annotations on Jakarta RESTful Web Services methods. By default though, RESTEasy does not recognize these annotations. You have to configure RESTEasy to turn on role-based security by setting the appropriate parameter. NOTE!!! Do not turn on this switch if you are using Jakarta Enterprise Beans. The Jakarta Enterprise Beans container will provide this functionality instead of RESTEasy. To configure this switch as a context-param, do this:

```
<web-app>
...
   <context-param>
      <param-name>resteasy.role.based.security</param-name>
      <param-value>true</param-value>
   </context-param>
</web-app>
```

See Section 3.4, “Configuration” for more information about application configuration.

There is a bit of quirkiness with this approach. You will have to declare all roles used within the RESTEasy war file that you are using in your Jakarta RESTful Web Services classes and set up a security constraint that permits all of these roles access to every URL handled by the Jakarta RESTful Web Services runtime. You'll just have to trust that RESTEasy Jakarta RESTful Web Services authorizes properly.

How does RESTEasy do authorization? Well, its really simple. It just sees if a method is annotated with @RolesAllowed and then just does HttpServletRequest.isUserInRole. If one of the @RolesAllowed passes, then allow the request, otherwise, a response is sent back with a 401 (Unauthorized) response code.

So, here's an example of a modified RESTEasy WAR file. You'll notice that every role declared is allowed access to every URL controlled by the RESTEasy servlet.

```
<web-app>

   <context-param>
      <param-name>resteasy.role.based.security</param-name>
      <param-value>true</param-value>
   </context-param>

   <security-constraint>
      <web-resource-collection>
         <web-resource-name>Resteasy</web-resource-name>
         <url-pattern>/security</url-pattern>
      </web-resource-collection>
       <auth-constraint>
         <role-name>admin</role-name>
         <role-name>user</role-name>
      </auth-constraint>
  </security-constraint>

   <login-config>
      <auth-method>BASIC</auth-method>
      <realm-name>Test</realm-name>
   </login-config>

   <security-role>
      <role-name>admin</role-name>
   </security-role>
   <security-role>
      <role-name>user</role-name>
   </security-role>

   ...
</web-app>
```

Chapter 44. JSON Web Signature and Encryption (JOSE-JWT)

JSON Web Signature and Encryption (JOSE JWT) is a new specification that can be used to encode content as a string and either digitally sign or encrypt it. I won't go over the spec here Do a Google search on it if you're interested

44.1. JSON Web Signature (JWS)

To digitally sign content using JWS, use the `org.jboss.resteasy.jose.jws.JWSBuilder` class. To unpack and verify a JWS, use the `org.jboss.resteasy.jose.jws.JWSInput` class. (TODO, write more doco here!) Here's an example:

```
@Test
public void testRSAWithContentType() throws Exception
{
   KeyPair keyPair = KeyPairGenerator.getInstance("RSA").generateKeyPair();

   String encoded = new JWSBuilder()
           .contentType(MediaType.TEXT_PLAIN_TYPE)
           .content("Hello World", MediaType.TEXT_PLAIN_TYPE)
           .rsa256(keyPair.getPrivate());

   System.out.println(encoded);

   JWSInput input = new JWSInput(encoded, ResteasyProviderFactory.getInstance());
   System.out.println(input.getHeader());
   String msg = (String)input.readContent(String.class);
   Assert.assertEquals("Hello World", msg);
   Assert.assertTrue(RSAProvider.verify(input, keyPair.getPublic()));

}
```

44.2. JSON Web Encryption (JWE)

To encrypt content using JWE, use the `org.jboss.resteasy.jose.jwe.JWEBuilder` class. To decrypt content using JWE, use the `org.jboss.resteasy.jose.jwe.JWEInput` class. (TODO, write more doco here!) Here's an example:

```
@Test
public void testRSA() throws Exception
```

```
  {
     KeyPair keyPair = KeyPairGenerator.getInstance("RSA").generateKeyPair();

     String content = "Live long and prosper.";

     {
                                                  String        encoded        =        new
JWEBuilder().contentBytes(content.getBytes()).RSA1_5((RSAPublicKey)keyPair.getPublic());
     System.out.println("encoded: " + encoded);
                                                    byte[]         raw         =         new
JWEInput(encoded).decrypt((RSAPrivateKey)keyPair.getPrivate()).getRawContent();
     String from = new String(raw);
     Assert.assertEquals(content, from);
     }
     {
                                                          String      encoded      =      new
JWEBuilder().contentBytes(content.getBytes()).RSA_OAEP((RSAPublicKey)keyPair.getPublic());
        System.out.println("encoded: " + encoded);
                                                             byte[]       raw       =       new
JWEInput(encoded).decrypt((RSAPrivateKey)keyPair.getPrivate()).getRawContent();
        String from = new String(raw);
        Assert.assertEquals(content, from);
     }
     {
                                                          String      encoded      =      new
JWEBuilder().contentBytes(content.getBytes()).A128CBC_HS256().RSA1_5((RSAPublicKey)keyPair.getPublic());
        System.out.println("encoded: " + encoded);
                                                             byte[]       raw       =       new
JWEInput(encoded).decrypt((RSAPrivateKey)keyPair.getPrivate()).getRawContent();
        String from = new String(raw);
        Assert.assertEquals(content, from);
     }
     {
                                                          String      encoded      =      new
JWEBuilder().contentBytes(content.getBytes()).A128CBC_HS256().RSA_OAEP((RSAPublicKey)keyPair.getPublic());
        System.out.println("encoded: " + encoded);
                                                             byte[]       raw       =       new
JWEInput(encoded).decrypt((RSAPrivateKey)keyPair.getPrivate()).getRawContent();
        String from = new String(raw);
        Assert.assertEquals(content, from);
     }
  }

  @Test
  public void testDirect() throws Exception
  {
     String content = "Live long and prosper.";
     String encoded = new JWEBuilder().contentBytes(content.getBytes()).dir("geheim");
     System.out.println("encoded: " + encoded);
     byte[] raw = new JWEInput(encoded).decrypt("geheim").getRawContent();
     String from = new String(raw);
     Assert.assertEquals(content, from);

  }
```

Chapter 45. Doseta Digital Signature Framework

Digital signatures allow you to protect the integrity of a message. They are used to verify that a message sent was sent by the actual user that sent the message and was modified in transit. Most web apps handle message integrity by using TLS, like HTTPS, to secure the connection between the client and server. Sometimes though, we have representations that are going to be forwarded to more than one recipient. Some representations may hop around from server to server. In this case, TLS is not enough. There needs to be a mechanism to verify who sent the original representation and that they actually sent that message. This is where digital signatures come in.

While the mime type multiple/signed exists, it does have drawbacks. Most importantly it requires the receiver of the message body to understand how to unpack. A receiver may not understand this mime type. A better approach would be to put signatures in an HTTP header so that receivers that don't need to worry about the digital signature, don't have to.

The email world has a nice protocol called Domain Keys Identified Mail [http://dkim.org] (DKIM). Work is also being done to apply this header to protocols other than email (i.e. HTTP) through the DOSETA specifications [https://tools.ietf.org/html/draft-crocker-doseta-base-02]. It allows you to sign a message body and attach the signature via a DKIM-Signature header. Signatures are calculated by first hashing the message body then combining this hash with an arbitrary set of metadata included within the DKIM-Signature header. You can also add other request or response headers to the calculation of the signature. Adding metadata to the signature calculation gives you a lot of flexibility to piggyback various features like expiration and authorization. Here's what an example DKIM-Signature header might look like.

```
DKIM-Signature: v=1;                a=rsa-sha256;               d=example.com;
    s=burke;                c=simple/simple;                h=Content-Type;
 x=0023423111111;               bh=2342322111;              b=M232234=
 v=1;
a=rsa-sha256;
 d=example.com;
 s=burke;
c=simple/simple;
h=Content-Type;
 x=0023423111111;
 bh=2342322111;
```

As you can see it is a set of name value pairs delimited by a ';'. While its not THAT important to know the structure of the header, here's an explanation of each parameter:

v

Protocol version. Always 1.

a

Algorithm used to hash and sign the message. RSA signing and SHA256 hashing is the only supported algorithm at the moment by RESTEasy.

d

Domain of the signer. This is used to identify the signer as well as discover the public key to use to verify the signature.

s

Selector of the domain. Also used to identify the signer and discover the public key.

c

Canonical algorithm. Only simple/simple is supported at the moment. Basically this allows you to transform the message body before calculating the hash

h

Semi-colon delimited list of headers that are included in the signature calculation.

x

When the signature expires. This is a numeric long value of the time in seconds since epoch. Allows signer to control when a signed message's signature expires

t

Timestamp of signature. Numeric long value of the time in seconds since epoch. Allows the verifier to control when a signature expires.

bh

Base 64 encoded hash of the message body.

b

Base 64 encoded signature.

To verify a signature you need a public key. DKIM uses DNS text records to discover a public key. To find a public key, the verifier concatenates the Selector (s parameter) with the domain (d parameter)

<selector>._domainKey.<domain>

It then takes that string and does a DNS request to retrieve a TXT record under that entry. In our above example burke._domainKey.example.com would be used as a string. This is a every interesting way to publish public keys. For one, it becomes very easy for verifiers to find public keys. There's no real central store that is needed. DNS is a infrastructure IT knows how to deploy.

Verifiers can choose which domains they allow requests from. RESTEasy supports discovering public keys via DNS. It also instead allows you to discover public keys within a local Java KeyStore if you do not want to use DNS. It also allows you to plug in your own mechanism to discover keys.

If you're interested in learning the possible use cases for digital signatures, here's a blog [http://bill.burkecentral.com/2011/02/21/multiple-uses-for-content-signature/] you might find interesting.

45.1. Maven settings

You must include the resteasy-crypto project to use the digital signature framework.

```
      <dependency>         <groupId>org.jboss.resteasy</groupId>         <artifactId>resteasy-
crypto</artifactId>            <version>5.0.10.Final</version>       </dependency>
pendency>
 <groupId>org.jboss.resteasy</groupId>
<artifactId>resteasy-crypto</artifactId>
 <version>5.0.10.Final</version>
```

45.2. Signing API

To sign a request or response using the RESTEasy client or server framework you need to create an instance of org.jboss.resteasy.security.doseta.DKIMSignature. This class represents the DKIM-Signature header. You instantiate the DKIMSignature object and then set the "DKIM-Signature" header of the request or response. Here's an example of using it on the server-side:

```
import                                org.jboss.resteasy.security.doseta.DKIMSignature;import
  java.security.PrivateKey;@Path("/signed")public  static  class  SignedResource{      @GET
 @Path("manual")   @Produces("text/plain")   public Response getManual()   {       PrivateKey
 privateKey = ....; // get the private key to sign message          DKIMSignature signature = new
 DKIMSignature();     signature.setSelector("test");     signature.setDomain("samplezone.org");
             signature.setPrivateKey(privateKey);             Response.ResponseBuilder  builder
   =   Response.ok("hello   world");                builder.header(DKIMSignature.DKIM_SIGNATURE,
   signature);               return   builder.build();       }}//   client   exampleDKIMSignature
    signature   =   new   DKIMSignature();PrivateKey   privateKey   =   ...;   //   go   find

        request       =       new       ClientRequest("http://...");request.header("DKIM-Signature",
   signature);request.body("text/plain",   "some   body   to   sign");ClientResponse   response   =
 request.put();
org.jboss.resteasy.security.doseta.DKIMSignature;import


java.security.PrivateKey;@Path("/
signed")public static class
SignedResource
{
@GET
@Path("manual")   @Produces("text/
plain")   public Response
getManual()
{      PrivateKey privateKey = ....; // get the private key to sign
```

```
message
       DKIMSignature signature = new
DKIMSignature();
signature.setSelector("test");
signature.setDomain("samplezone.org");

signature.setPrivateKey(privateKey);      Response.ResponseBuilder builder = Response.ok("hello
world");      builder.header(DKIMSignature.DKIM_SIGNATURE,
signature);      return
builder.build();
}

}// client

exampleDKIMSignature signature = new
DKIMSignature();PrivateKey privateKey = ...; // go find
it
signature.setSelector("test");
signature.setDomain("samplezone.org");

signature.setPrivateKey(privateKey);ClientRequest request = new
ClientRequest("http://...");request.header("DKIM-Signature",
signature);request.body("text/plain", "some body to
sign");ClientResponse response =
```

To sign a message you need a PrivateKey. This can be generated by KeyTool or manually using regular, standard JDK Signature APIs. RESTEasy currently only supports RSA key pairs. The DKIMSignature class also allows you to add and control how various pieces of metadata are added to the DKIM-Signature header and the signature calculation. See the javadoc for more details.

If you are including more than one signature, then just add additional DKIMSignature instances to the headers of the request or response.

45.2.1. @Signed annotation

Instead of using the API, RESTEasy also provides you an annotation alternative to the manual way of signing using a DKIMSignature instances is to use the @org.jboss.resteasy.annotations.security.doseta.Signed annotation. It is required that you configure a KeyRepository as described later in this chapter. Here's an example:

```
@GET
@Produces("text/plain")
@Path("signedresource")
@Signed(selector="burke", domain="sample.com", timestamped=true, expires=@After(hours=24))
public String getSigned()
{
   return "hello world";
}
```

The above example using a bunch of the optional annotation attributes of @Signed to create the following Content-Signature header:

```
DKIM-Signature: v=1;                a=rsa-sha256;                c=simple/simple;
     domain=sample.com;                s=burke;                t=02342342341;
 x=02342342322;               bh=m0234fsefasf==;               b=mababaddbb==
 v=1;
a=rsa-sha256;
c=simple/simple;
 domain=sample.com;
 s=burke;
 t=02342342341;
 x=02342342322;
 bh=m0234fsefasf==;
 b=mababaddbb==
```

This annotation also works with the client proxy framework.

45.3. Signature Verification API

If you want fine grain control over verification, this is an API to verify signatures manually. Its a little tricky because you'll need the raw bytes of the HTTP message body in order to verify the signature. You can get at an unmarshalled message body as well as the underlying raw bytes by using a org.jboss.resteasy.spi.MarshalledEntity injection. Here's an example of doing this on the server side:

```
import      org.jboss.resteasy.spi.MarshalledEntity;@POST@Consumes("text/plain")@Path("verify-
manual")public  void  verifyManual(@HeaderParam("Content-Signature")  DKIMSignature  signature,
                                    @Context  KeyRepository  repository,
                      @Context  HttpHeaders  headers,
 MarshalledEntity<String> input) throws Exception{      Verifier verifier = new Verifier();
  Verification verification = verifier.addNew();      verification.setRepository(repository);
    verification.setStaleCheck(true);      verification.setStaleSeconds(100);      try {
   verifier.verifySignature(headers.getRequestHeaders(), input.getMarshalledBytes, signature);
        } catch (SignatureException ex) {        }        System.out.println("The  text  message
 posted is: " + input.getEntity());}


org.jboss.resteasy.spi.MarshalledEntity;
@POST@Consumes("text/
plain")@Path("verify-
manual")public void verifyManual(@HeaderParam("Content-Signature") DKIMSignature
signature,                          @Context KeyRepository repository,
                         @Context HttpHeaders headers,
                         MarshalledEntity<String> input) throws
Exception
{      Verifier verifier = new
Verifier();      Verification verification =
verifier.addNew();
verification.setRepository(repository);
verification.setStaleCheck(true);
verification.setStaleSeconds(100);      try
{               verifier.verifySignature(headers.getRequestHeaders(),
input.getMarshalledBytes,
signature);      } catch (SignatureException ex)
{
```

```
}      System.out.println("The text message posted is: " +
input.getEntity());
```

MarshalledEntity is a generic interface. The template parameter should be the Java type you want the message body to be converted into. You will also have to configure a KeyRepository. This is describe later in this chapter.

The client side is a little bit different:

```
ClientRequest          request          =          new          ClientRequest("http://local
host:9095/signed"));ClientResponse<String>   response   =   request.get(String.class);Verifier
         verifier         =         new         Verifier();Verification         verification        =

  verifier);//  signature  verification  happens  when  you  get  the  entityString  entity  =
 response.getEntity();


ClientRequest("http://localhost:9095/signed"));ClientResponse<String>
response = request.get(String.class);Verifier verifier
= new Verifier();Verification
verification
 =

verifier.addNew();verification.setRepository(repository);response.getProperties().put(Verifier.class.getName(), ve
get the entityString
```

On the client side, you create a verifier and add it as a property to the ClientResponse. This will trigger the verification interceptors.

45.3.1. Annotation-based verification

The easiest way to verify a signature sent in a HTTP request on the server side is to use the @@org.jboss.resteasy.annotations.security.doseta.Verify (or @Verifications which is used to verify multiple signatures). Here's an example:

```
     @POST
     @Consumes("text/plain")
     @Verify
     public void post(String input)
     {
     }
```

In the above example, any DKIM-Signature headers attached to the posted message body will be verified. The public key to verify is discovered using the configured KeyRepository (discussed later in this chapter). You can also specify which specific signatures you want to verify as well as define multiple verifications you want to happen via the @Verifications annotation. Here's a complex example:

```
@POST
@Consumes("text/plain")
@Verifications(
   @Verify(identifierName="d", identiferValue="inventory.com", stale=@After(days=2)),
   @Verify(identifierName="d", identiferValue="bill.com")
}
public void post(String input) {...}
```

The above is expecting 2 different signature to be included within the DKIM-Signature header.

Failed verifications will throw an org.jboss.resteasy.security.doseta.UnauthorizedSignatureException. This causes a 401 error code to be sent back to the client. If you catch this exception using an ExceptionHandler you can browse the failure results.

45.4. Managing Keys via a KeyRepository

RESTEasy manages keys for you through a org.jboss.resteasy.security.doseta.KeyRepository. By default, the KeyRepository is backed by a Java KeyStore. Private keys are always discovered by looking into this KeyStore. Public keys may also be discovered via a DNS text (TXT) record lookup if configured to do so. You can also implement and plug in your own implementation of KeyRepository.

45.4.1. Create a KeyStore

Use the Java keytool to generate RSA key pairs. Key aliases MUST HAVE the form of:

<selector>._domainKey.<domain>

For example:

```
$ keytool -genkeypair -alias burke._domainKey.example.com -keyalg RSA -keysize 1024 -keystore
 my-apps.jks
```

You can always import your own official certificates too. See the JDK documentation for more details.

45.4.2. Configure Restreasy to use the KeyRepository

Next you need to configure the KeyRepository in your web.xml file so that it is created and made available to RESTEasy to discover private and public keys.You can reference a Java key store you want the Resteasy signature framework to use within web.xml using either `resteasy.keystore.classpath` or `resteasy.keystore.filename` context parameters. You must also specify the password (sorry its clear text) using the `resteasy.keystore.password` context parameter. The resteasy.context.objects is used to create the instance of the repository. For example:

```
     <context-param>          <param-name>resteasy.doseta.keystore.classpath</param-name>
       <param-value>test.jks</param-value>      </context-param>      <context-param>
 <param-name>resteasy.doseta.keystore.password</param-name>        <param-value>geheim</param-
value>     </context-param>     <context-param>          <param-name>resteasy.context.objects</
param-name>                   <param-value>org.jboss.resteasy.security.doseta.KeyRepository  :
 org.jboss.resteasy.security.doseta.ConfiguredDosetaKeyRepository</param-value>     </context-
param>
param>        <param-name>resteasy.doseta.keystore.classpath</param-
name>        <param-value>test.jks</param-
value>    </context-
param>    <context-
param>        <param-name>resteasy.doseta.keystore.password</param-
name>        <param-value>geheim</param-
value>    </context-
param>    <context-
param>        <param-name>resteasy.context.objects</param-
name>                            <param-value>org.jboss.resteasy.security.doseta.KeyRepository
: org.jboss.resteasy.security.doseta.ConfiguredDosetaKeyRepository</param-
value>    </context-
```

You can also manually register your own instance of a KeyRepository within an Application class. For example:

```
import                                              org.jboss.resteasy.core.Dispatcher;import


 javax.ws.rs.core.Application;import javax.ws.rs.core.Context;public class SignatureApplication
  extends  Application{      private  HashSet<Class<?>>  classes  =  new  HashSet<Class<?>>();
      private  KeyRepository  repository;      public  SignatureApplication(@Context  Dispatcher
  dispatcher)       {              classes.add(SignedResource.class);              repository  =
   new   DosetaKeyRepository();               repository.setKeyStorePath("test.jks");
  repository.setKeyStorePassword("password");            repository.setUseDns(false);
  repository.start();            dispatcher.getDefaultContextObjects().put(KeyRepository.class,
 repository);   }   @Override   public Set<Class<?>> getClasses()   {      return classes;   }}
org.jboss.resteasy.core.Dispatcher;import
org.jboss.resteasy.security.doseta.KeyRepository;import

org.jboss.resteasy.security.doseta.DosetaKeyRepository;import
javax.ws.rs.core.Application;import

javax.ws.rs.core.Context;public class SignatureApplication extends
Application
{   private HashSet<Class<?>> classes = new HashSet<Class<?
>>();   private KeyRepository

repository;   public SignatureApplication(@Context Dispatcher
dispatcher)
{

classes.add(SignedResource.class);      repository = new
DosetaKeyRepository();
repository.setKeyStorePath("test.jks");
repository.setKeyStorePassword("password");
repository.setUseDns(false);
```

```
repository.start();      dispatcher.getDefaultContextObjects().put(KeyRepository.class,
repository);

}
@Override   public Set<Class<?>>
getClasses()
{      return
classes;
}
```

On the client side, you can load a KeyStore manually, by instantiating an instance of org.jboss.resteasy.security.doseta.DosetaKeyRepository. You then set a request attribute, "org.jboss.resteasy.security.doseta.KeyRepository", with the value of the created instance. Use the ClientRequest.getAttributes() method to do this. For example:

```
DosetaKeyRepository          keyRepository          =          new          DoestaKeyReposito
ry();repository.setKeyStorePath("test.jks");repository.setKeyStorePassword("password");repository.setUseDns(false)
  signature  =  new  DKIMSignature();signature.setDomain("example.com");ClientRequest  request
 =  new  ClientRequest("http://...");request.getAttributes().put(KeyRepository.class.getName(),
repository);request.header("DKIM-Signature", signatures);
ry

=


new DoestaKeyRepository();repository.setKeyStorePath("test.jks");repository.setKeyStorePassword("password");
repository.setUseDns(false);

repository.start();DKIMSignature signature = new
DKIMSignature();signature.setDomain("example.com");ClientRequest
 request =
```

45.4.3. Using DNS to Discover Public Keys

Public keys can also be discover by a DNS text record lookup. You must configure web.xml to turn this feature:

```
      <context-param>             <param-name>resteasy.doseta.use.dns</param-name>
 <param-value>true</param-value>      </context-param>      <context-param>            <param-
name>resteasy.doseta.dns.uri</param-name>      <param-value>dns://localhost:9095</param-value>
    </context-param>
param>        <param-name>resteasy.doseta.use.dns</param-
name>        <param-value>true</param-
value>    </context-
param>    <context-
param>        <param-name>resteasy.doseta.dns.uri</param-
name>        <param-value>dns://localhost:9095</param-
value>    </context-
```

The resteasy.doseta.dns.uri context-param is optional and allows you to point to a specific DNS server to locate text records.

45.4.3.1. Configuring DNS TXT Records

DNS TXT Records are stored via a format described by the DOSETA specification. The public key is defined via a base 64 encoding. You can obtain this text encoding by exporting your public keys from your keystore, then using a tool like openssl to get the text-based format. For example:

```
$ keytool -export -alias bill._domainKey.client.com -keystore client.jks -file bill.der $ openssl
 x509 -noout -pubkey -in bill.der -inform der > bill.pem
 bill.der $ openssl x509 -noout -pubkey -in bill.der -inform der
```

The output will look something like:

```
-----BEGIN PUBLIC KEY-----MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCKxct5GHz8dFw0mzAMfvNju2b3oeAv/
EOPfVb9mD73Wn+CJYXvnryhqo99Y/q47urWYWAF/bqH9AMyMfibPr6IlP8mO9pNYf/Zsqup/7oJxrvzJU7T0IGdLN1hHcC
+qRnwkKddNmD8UPEQ4BXiX4xFxbTjNvKWLZVKGQMyy6EFVQIDAQAB-----END PUBLIC KEY-----
KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCKxct5GHz8dFw0mzAMfvNju2b3oeAv/EOPfVb9mD73Wn+CJYXvnryhqo99Y/q47urWYWAF/
bqH9AMyMfibPr6IlP8mO9pNYf/Zsqup/7oJxrvzJU7T0IGdLN1hHcC
+qRnwkKddNmD8UPEQ4BXiX4xFxbTj
NvKWLZVKGQMyy6EFVQIDAQAB-----END PUBLIC
```

The DNS text record entry would look like this:

```
test2._domainKey                                        IN                                TXT
                "v=DKIM1;     p=MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCIKFLFWuQfDfBug688BJ0dazQ/x
+GEnH443KpnBK8agpJXSgFAPhlRvf0yhqHeuI
+J5onsSOo9Rn4fKaFQaQNBfCQpHSMnZpBC3X0G5Bc1HWq1AtBl6Z1rbyFen4CmGYOyRzDBUOIW6n8QK47bf3hvoSxqpY1pHdgYoVK0YdIP
+wIDAQAB; t=s"
```

Notice that the newlines are take out. Also, notice that the text record is a name value ';' delimited list of parameters. The p field contains the public key.

Chapter 46. Body Encryption and Signing via SMIME

S/MIME (Secure/Multipurpose Internet Mail Extensions) is a standard for public key encryption and signing of MIME data. MIME data being a set of headers and a message body. Its most often seen in the email world when somebody wants to encrypt and/or sign an email message they are sending across the internet. It can also be used for HTTP requests as well which is what the RESTEasy integration with S/MIME is all about. RESTEasy allows you to easily encrypt and/or sign an email message using the S/MIME standard. While the API is described here, you may also want to check out the example projects that come with the RESTEasy distribution. It shows both Java and Python clients exchanging S/MIME formatted messages with a Jakarta RESTful Web Services service.

46.1. Maven settings

You must include the resteasy-crypto project to use the smime framework.

```
      <dependency>         <groupId>org.jboss.resteasy</groupId>         <artifactId>resteasy-
crypto</artifactId>            <version>5.0.10.Final</version>       </dependency>
pendency>
 <groupId>org.jboss.resteasy</groupId>
<artifactId>resteasy-crypto</artifactId>
 <version>5.0.10.Final</version>
```

46.2. Message Body Encryption

While HTTPS is used to encrypt the entire HTTP message, S/MIME encryption is used solely for the message body of the HTTP request or response. This is very useful if you have a representation that may be forwarded by multiple parties (for example, HornetQ's REST Messaging integration!) and you want to protect the message from prying eyes as it travels across the network. RESTEasy has two different interfaces for encrypting message bodies. One for output, one for input. If your client or server wants to send an HTTP request or response with an encrypted body, it uses the `org.jboss.resteasy.security.smime.EnvelopedOutput` type. Encrypting a body also requires an X509 certificate which can be generated by the Java keytool command-line interface, or the openssl tool that comes installed on many OS's. Here's an example of using the `EnvelopedOutput` interface:

```
//  server  side     @Path("encrypted")@GETpublic  EnvelopedOutput  getEncrypted(){     Customer
  cust  =  new  Customer();      cust.setName("Bill");            X509Certificate  certificate
 = ...;   EnvelopedOutput output = new EnvelopedOutput(cust, MediaType.APPLICATION_XML_TYPE);
      output.setCertificate(certificate);      return  output;}//  client  sideX509Certificate
  cert  =  ...;  Customer  cust  =  new  Customer();cust.setName("Bill");EnvelopedOutput  output
```

```
  =  new  EnvelopedOutput(cust,  "application/xml");output.setCertificate(cert);Response  res  =
 target.request().post(Entity.entity(output, "application/pkcs7-mime").post();


@Path("encrypted")
@GETpublic EnvelopedOutput
getEncrypted()
{   Customer cust = new
Customer();
cust.setName("Bill");
    X509Certificate certificate =
...;   EnvelopedOutput output = new EnvelopedOutput(cust,
MediaType.APPLICATION_XML_TYPE);
output.setCertificate(certificate);   return
output;


}// client
sideX509Certificate cert = ...;
 Customer cust = new
Customer();
cust.setName("Bill");EnvelopedOutput output = new EnvelopedOutput(cust, "application/
xml");
output.setCertificate(cert);Response res = target.request().post(Entity.entity(output, "application/pkcs7-
```

An EnvelopedOutput instance is created passing in the entity you want to marshal and the media type you want to marshal it into. So in this example, we're taking a Customer class and marshalling it into XML before we encrypt it. RESTEasy will then encrypt the EnvelopedOutput using the BouncyCastle framework's SMIME integration. The output is a Base64 encoding and would look something like this:

```
Content-Type:   application/pkcs7-mime;   smime-type=enveloped-data;   name="smime.p7m"Content-
Transfer-Encoding:                   base64Content-Disposition:                   attachment;

O34DFl2p2zm+xZQ6R+94BqZHdtEWQN2evrcgtAng+f2ltILxr/PiK+8bE8wDO5GuCg
+k92uYp2rLKlZ5BxCGb8tRM4kYC9sHbH2dPaqzUBhMxjgWdMCX6Q7E130u9MdGcP74Ogwj8fNl3lD4sx/0k02/
QwgaukeY7uNHzCABgkqhkiG9w0BBwEwFAYIKoZIhvcNAwcECDRozFLsPnSgoIAEQHmqjSKAWlQbuGQL9w4nKw4l
+44WgTjKf7mGWZvYY8tOCcdmhDxRSM1Ly682Imt+LTZf0LXzuFGTsCGOUo742N8AAAAAAAAAAAAA
type=enveloped-data; name="smime.p7m"
Content-Transfer-Encoding: base64Content-

Disposition:

attachment; filename="smime.p7m"MIAGCSqGSIb3DQEHA6CAMIACAQAxgewwgekCAQAwUjBFMQswCQYDVQQGEwJBVTETMBEGA1UECBMKU29tZ
DQYJKoZIhvcNAQEBBQAEgYCfnqPK/O34DFl2p2zm+xZQ6R+94BqZHdtEWQN2evrcgtAng
+f2ltILxr/
PiK+8bE8wDO5GuCg
+k92uYp2rLKlZ5BxCGb8tRM4kYC9sHbH2dPaqzUBhMxjgWdMCX6Q7E130u9MdGcP74Ogwj8fNl3lD4sx/0k02/
```

Decrypting an S/MIME encrypted message requires using the org.jboss.resteasy.security.smime.EnvelopedInput interface. You also need both the private key and X509Certificate used to encrypt the message. Here's an example:

```
//   server   side@Path("encrypted")@POSTpublic   void   postEncrypted(EnvelopedInput<Customer>
    input){          PrivateKey   privateKey   =   ...;          X509Certificate   certificate
    =    ...;           Customer    cust    =    input.getEntity(privateKey,    certificate);}//
   client   sideClientRequest   request   =   new   ClientRequest("http://localhost:9095/smime/
encrypted");EnvelopedInput  input  =  request.getTarget(EnvelopedInput.class);Customer  cust  =
 (Customer)input.getEntity(Customer.class, privateKey, cert);

side
@Path("encrypted")
@POSTpublic void postEncrypted(EnvelopedInput<Customer>
input)
{   PrivateKey privateKey =
...;   X509Certificate certificate =
...;   Customer cust = input.getEntity(privateKey,
certificate);

}// client

sideClientRequest request = new ClientRequest("http://localhost:9095/smime/
encrypted");EnvelopedInput input =
request.getTarget(EnvelopedInput.class);Customer cust = (Customer)input.getEntity(Customer.class, privateKey,
```

Both examples simply call the getEntity() method passing in the PrivateKey and X509Certificate instances requires to decrypt the message. On the server side, a generic is used with EnvelopedInput to specify the type to marshal to. On the server side this information is passed as a parameter to getEntity(). The message is in MIME format: a Content-Type header and body, so the EnvelopedInput class now has everything it needs to know to both decrypt and unmarshall the entity.

46.3. Message Body Signing

S/MIME also allows you to digitally sign a message. It is a bit different than the Doseta Digital Signing Framework. Doseta is an HTTP header that contains the signature. S/MIME uses the multipart/signed data format which is a multipart message that contains the entity and the digital signature. So Doseta is a header, S/MIME is its own media type. Generally I would prefer Doseta as S/MIME signatures require the client to know how to parse a multipart message and Doseta doesn't. Its up to you what you want to use.

RESTEasy has two different interfaces for creating a multipart/signed message. One for input, one for output. If your client or server wants to send an HTTP request or response with an multipart/signed body, it uses the `org.jboss.resteasy.security.smime.SignedOutput` type. This type requires both the PrivateKey and X509Certificate to create the signature. Here's an example of signing an entity and sending a multipart/signed entity.

```
// server-side   @Path("signed")   @GET   @Produces("multipart/signed")   public SignedOutput
  getSigned()     {          Customer  cust  =  new  Customer();          cust.setName("Bill");
        SignedOutput  output  =  new  SignedOutput(cust,  MediaType.APPLICATION_XML_TYPE);
 output.setPrivateKey(privateKey);      output.setCertificate(certificate);      return output;
    }// client side       Client client = ClientBuilder.newClient();       WebTarget target =
 client.target("http://localhost:9095/smime/signed");       Customer cust = new Customer();
```

```
   cust.setName("Bill");      SignedOutput output = new SignedOutput(cust, "application/xml");
      output.setPrivateKey(privateKey);      output.setCertificate(cert);      Response res =
 target.request().post(Entity.entity(output, "multipart/signed");

side
@Path("signed")
@GET   @Produces("multipart/
signed")   public SignedOutput
getSigned()
{      Customer cust = new
Customer();

cust.setName("Bill");      SignedOutput output = new SignedOutput(cust,
MediaType.APPLICATION_XML_TYPE);
output.setPrivateKey(privateKey);
output.setCertificate(certificate);      return
output;


}// client
side      Client client =
ClientBuilder.newClient();      WebTarget target = client.target("http://localhost:9095/smime/
signed");      Customer cust = new
Customer();
cust.setName("Bill");      SignedOutput output = new SignedOutput(cust, "application/
xml");
output.setPrivateKey(privateKey);
output.setCertificate(cert);      Response res = target.request().post(Entity.entity(output, "multipart/
```

An SignedOutput instance is created passing in the entity you want to marshal and the media type you want to marshal it into. So in this example, we're taking a Customer class and marshalling it into XML before we sign it. RESTEasy will then sign the SignedOutput using the BouncyCastle framework's SMIME integration. The output iwould look something like this:

```
Content-Type: multipart/signed; protocol="application/pkcs7-signature"; micalg=sha1;   bound
ary="----=_Part_0_1083228271.1313024422098"------=_Part_0_1083228271.1313024422098Content-
Type:     application/xmlContent-Transfer-Encoding:     7bit<customer     name="bill"/>------
=_Part_0_1083228271.1313024422098Content-Type:   application/pkcs7-signature;   name=smime.p7s;
    smime-type=signed-dataContent-Transfer-Encoding:    base64Content-Disposition:    attachment;
                filename="smime.p7s"Content-Description:                S/MIME                Cryptographic
FH32BfR1l1vzDshtQvJrgvpGvjADMA0GCSqGSIb3DQEBAQUABIGAL3KVi3ul9cPRUMYcGgQmWtsZ0bLbAldO
+okrt8mQ87SrUv2LGkIJbEhGHsOlsgSU80/YumP+Q4lYsVanVfoI8GgQH3Iztp
+Rce2cy42f86ZypE7ueynI4HTPNHfr78EpyKGzWuZHW4yMo70LpXhk5RqfM9a/n4TEa9QuTU76atAAAAAAAAA=------
=_Part_0_1083228271.1313024422098--

micalg=sha1;
 boundary="----=_Part_0_1083228271.1313024422098"------
=_Part_0_1083228271.1313024422098Content-Type: application/

xmlContent-Transfer-Encoding:
 7bit
<customer name="bill"/>------=_Part_0_1083228271.1313024422098Content-Type: application/pkcs7-signature;
 name=smime.p7s; smime-type=signed-
dataContent-Transfer-Encoding: base64
```

```
Content-Disposition: attachment; filename="smime.p7s"Content-Description:

S/
MIME

Cryptographic

SignatureMIAGCSqGSIb3DQEHAqCAMIACAQExCzAJBgUrDgMCGgUAMIAGCSqGSIb3DQEHAQAAMYIBVzCCAVMCAQEwUjBFMQswCQYDVQQGEwJBVTETM
FH32BfR1l1vzDshtQvJrgvpGvjADMA0GCSqGSIb3DQEBAQUABIGAL3KVi3ul9cPRUMYcGgQmWtsZ0bLbAldO
+okrt8mQ87SrUv2LGkIJbEhGHsOlsgSU80/
YumP+Q4lYsVanVfoI8GgQH3Iztp
```

To unmarshal and verify a signed message requires using the `org.jboss.resteasy.security.smime.SignedInput` interface. You only need the X509Certificate to verify the message. Here's an example of unmarshalling and verifying a multipart/signed entity.

```
// server side    @Path("signed")    @POST    @Consumes("multipart/signed")    public void
  postSigned(SignedInput<Customer> input) throws Exception     {           Customer cust =
  input.getEntity();          if (!input.verify(certificate))          {               throw
  new WebApplicationException(500);          }     }// client side          Client client =
 ClientBuilder.newClient();     WebTarget target = client.target("http://localhost:9095/smime/
signed");      SignedInput input = target.request().get(SignedInput.class);      Customer cust
 = (Customer)input.getEntity(Customer.class)      input.verify(cert);

side
@Path("signed")
@POST   @Consumes("multipart/
signed")   public void postSigned(SignedInput<Customer> input) throws
Exception
{      Customer cust =
input.getEntity();      if (!
input.verify(certificate))
{         throw new
WebApplicationException(500);
}

}// client
side      Client client =
ClientBuilder.newClient();      WebTarget target = client.target("http://localhost:9095/smime/
signed");      SignedInput input =
target.request().get(SignedInput.class);      Customer cust =
(Customer)input.getEntity(Customer.class)
```

46.4. application/pkcs7-signature

application/pkcs7-signature is a data format that includes both the data and the signature in one ASN.1 binary encoding.

SignedOutput and SignedInput can be used to return application/pkcs7-signature format in binary form. Just change the @Produces or @Consumes to that media type to send back that format.

Also, if your @Produces or @Consumes is text/plain instead, SignedOutput will be base64 encoded and sent as a string.

Chapter 47. Jakarta Enterprise Beans Integration

To integrate with Jakarta Enterprise Beans you must first modify your beans published interfaces. RESTEasy currently only has simple portable integration with Jakarta Enterprise Beans so you must also manually configure your RESTEasy WAR.

RESTEasy currently only has simple integration with Jakarta Enterprise Beans. To make a bean a Jakarta RESTful Web Services resource, you must annotate an SLSB's @Remote or @Local interface with Jakarta RESTful Web Services annotations:

```
@Local
@Path("/Library")
public interface Library {

   @GET
   @Path("/books/{isbn}")
   public String getBook(@PathParam("isbn") String isbn);
}

@Stateless
public class LibraryBean implements Library {

...

}
```

Next, in RESTEasy's web.xml file you must manually register the bean with RESTEasy using the resteasy.jndi.resources <context-param>

```
<web-app>
   <display-name>Archetype Created Web Application</display-name>
   <context-param>
      <param-name>resteasy.jndi.resources</param-name>
      <param-value>LibraryBean/local</param-value>
   </context-param>

   ...
</web-app>
```

This is the only portable way we can offer Jakarta Enterprise Beans integration. Future versions of RESTEasy will have tighter integration with WildFly so you do not have to do any manual registrations or modifications to web.xml. For right now though, we're focusing on portability.

If you're using RESTEasy with an EAR and Jakarta Enterprise Beans, a good structure to have is:

```
my-ear.ear
|------myejb.jar
|------resteasy-jaxrs.war
       |
       ----WEB-INF/web.xml
       ----WEB-INF/lib (nothing)
|------lib/
       |
       ----All RESTEasy jar files
```

From the distribution, remove all libraries from WEB-INF/lib and place them in a common EAR lib. OR. Just place the RESTEasy jar dependencies in your application server's system classpath. (i.e. In JBoss put them in server/default/lib)

An example EAR project is available from our testsuite here.

Chapter 48. Spring Integration

RESTEasy integrates with Springframework in various forms. In this chapter we introduce different methods to integrate Springframework with RESTEasy.

IMPORTANT: As of RESTEasy 5.0.0 the Spring integration has moved to a new project, group id and version. The new group id is `org.jboss.resteasy.spring`. Currently the artifact id's have not changed.

RESTEasy currently supports Spring version 5.3

48.1. Basic Integration

For Maven users, you must use the `org.jboss.resteasy.spring:resteasy-spring` artifact. And here is the dependency you should use:

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

RESTEasy comes with its own `ContextLoaderListener` that registers a RESTEasy specific `BeanPostProcessor` that processes Jakarta RESTful Web Services annotations when a bean is created by a `BeanFactory`. And it will automatically scan for `@Provider` and Jakarta RESTful Web Services resource annotations on your bean class and register them as Jakarta RESTful Web Services resources.

Here is the content that you should add into your `web.xml` file:

```
<listener>       <listener-class>org.jboss.resteasy.plugins.server.servlet.ResteasyBootstrap</
listener-class></listener><listener>                                                    <listener-
class>org.jboss.resteasy.plugins.spring.SpringContextLoaderListener</listener-class></
listener>
tener>        <listener-
class>org.jboss.resteasy.plugins.server.servlet.ResteasyBootstrap</
listener-class>

</
listener><listener>      <listener-
class>org.jboss.resteasy.plugins.spring.SpringContextLoaderListener</
listener-class>
```

Please note that the `SpringContextLoaderListener` must be declared after `ResteasyBootstrap` as it uses `ServletContext` attributes initialized by it.

And you can configure the Springframework to scan for the Jakarta RESTful Web Services resources and beans in a Spring configuration file. The content of the file is shown as follow:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:context="http://www.springframework.org/schema/context"
                   xsi:schemaLocation="http://www.springframework.org/schema/context   http://
www.springframework.org/schema/context/spring-context.xsd

     http://www.springframework.org/schema/beans   http://www.springframework.org/schema/beans/
spring-beans.xsd">

    <context:component-scan base-package="org.jboss.resteasy.examples.springbasic">
        <context:include-filter type="annotation" expression="javax.ws.rs.Path"/>
    </context:component-scan>
    <context:annotation-config/>

</beans>
```

Let's say the above file is named `resteasy-spring-basic.xml`, then in your `web.xml` the can be configured like this:

```
<context-param>             <param-name>contextConfigLocation</param-name>             <param-
value>classpath:resteasy-spring-basic.xml</param-value></context-param>
param>    <param-name>contextConfigLocation</param-
name>    <param-value>classpath:resteasy-spring-basic.xml</param-
value></context-
```

In addition, you need to configure your RESTEasy servlet in `web.xml`. Here is the example:

```
<servlet>
    <servlet-name>Resteasy</servlet-name>
      <servlet-class>org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher</servlet-
class>
</servlet>

<servlet-mapping>
    <servlet-name>Resteasy</servlet-name>
    <url-pattern>/rest/*</url-pattern>
</servlet-mapping>
```

Instead of using `HttpServletDispatcher` for deployment, you can also use the `FilterDispatcher` in `web.xml`:

```
<filter>        <filter-name>resteasy-filter</filter-name>        <filter-class>
 org.jboss.resteasy.plugins.server.servlet.FilterDispatcher    </filter-class></filter>
ter>    <filter-name>resteasy-filter</filter-
```

```
name>    <filter-
class>
org.jboss.resteasy.plugins.server.servlet.FilterDispatcher    </filter-
class></
```

To see a complete example of the above basic usage, please check the Basic Example [https://github.com/resteasy/resteasy-examples/tree/master/resteasy-spring-basic] we provided.

48.2. Customized Configuration

If you do not want to use the `SpringContextLoaderListener` provided by RESTEasy, and want to create your bean factories, then you can manually register the RESTEasy `BeanFactoryPostProcessor` by creating an instance of the RESTEasy `SpringBeanProcessor`.

And you must configure the `RestasyBootstrap` into the scope to create the `ResteasyDeployment` so the relative classes can be fetched from `ServletContext`.

There is also a `SpringBeanProcessorServletAware` class that implements the `ServletContextAware` interface provided by Springframework. The `SpringBeanProcessorServletAware` can be used to fetch the `Registry` and `ResteasyProviderFactory` from the `ServletContext`.

To demonstrate the above process, we have also provide an example. Please check the Spring and Resteasy Customized Example [https://github.com/resteasy/resteasy-examples/tree/master/resteasy-spring-customized] we provided.

Our Spring integration supports both singletons and the "prototype" scope. RESTEasy handles injecting `@Context` references. Constructor injection is not supported though. Also, with the "prototype" scope, RESTEasy will inject any `@*Param` annotated fields or setters before the request is dispatched.

NOTE: You can only use auto-proxied beans with our base Spring integration. You will have undesirable affects if you are doing handcoded proxying with Spring, i.e., with `ProxyFactoryBean`. If you are using auto-proxied beans, you will be ok.

48.3. Spring MVC Integration

RESTEasy can also integrate with the Spring MVC framework. Generally speaking, Jakarta RESTful Web Services can be combined with a Spring `DispatcherServlet` and used in the same web application.

An application combined in this way allows you to dispatch to either the Spring controller or the Jakarta RESTful Web Services resource using the same base URL. In addition you can use the Spring `ModelAndView` objects as return arguments from `@GET` resource methods.

The setup requires you to use the Spring `DispatcherServlet` in your `web.xml` file, as well as importing the `springmvc-resteasy.xml` file into your base Spring beans xml file. Here's an example `web.xml` file:

```
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"           xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"            xsi:schemaLocation="http://java.sun.com/xml/ns/
javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">      <display-name>resteasy-spring-
mvc</display-name>    <servlet>         <servlet-name>resteasy-spring-mvc</servlet-name>
    <servlet-class>org.jboss.resteasy.springmvc.ResteasySpringDispatcherServlet</servlet-class>
         <init-param>              <param-name>contextConfigLocation</param-name>
 <param-value>classpath:resteasy-spring-mvc-servlet.xml</param-value>         </init-param>
 </servlet>    <servlet-mapping>         <servlet-name>resteasy-spring-mvc</servlet-name>
   <url-pattern>/rest/*</url-pattern>    </servlet-mapping></web-app>
javaee"         xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/
javaee/web-
app_3_0.xsd">    <display-name>resteasy-spring-mvc</display-

name>
<servlet>        <servlet-name>resteasy-spring-mvc</servlet-
name>          <servlet-class>org.jboss.resteasy.springmvc.ResteasySpringDispatcherServlet</
servlet-
class>        <init-
param>            <param-name>contextConfigLocation</param-
name>            <param-value>classpath:resteasy-spring-mvc-servlet.xml</param-
value>        </init-
param>    </

servlet>    <servlet-
mapping>        <servlet-name>resteasy-spring-mvc</servlet-
name>        <url-pattern>/rest/*</url-
pattern>    </servlet-

mapping></web-
```

Then within the `resteasy-spring-mvc-servlet.xml`, it should import the `spring-mvc-resteasy.xml` file:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
   http://www.springframework.org/schema/context http://www.springframework.org/schema/context/
spring-context-2.5.xsd
       http://www.springframework.org/schema/util  http://www.springframework.org/schema/util/
spring-util-2.5.xsd
http://www.springframework.org/schema/beans       http://www.springframework.org/schema/beans/
spring-beans.xsd
">

    <!-- Import basic SpringMVC RESTEasy integration -->
    <import resource="classpath:springmvc-resteasy.xml"/>
....
```

And then you need to tell Spring the package to scan for your Jakarta RESTful Web Services resource classes:

```
<context:component-scan base-package="org.jboss.resteasy.examples.springmvc"/>
<context:annotation-config/>
```

Above is the basic configuration for Spring MVC framework. To see a complete example, please check the Spring MVC Integration Example [https://github.com/resteasy/resteasy-examples/tree/master/resteasy-spring-mvc] we provided.

In addition, A `javax.ws.rs.core.Application` subclass can be combined with a Spring `DispatcherServlet` and used in the same web application.

A servlet definition is required for both the Spring `DispatcherServlet` and the `javax.ws.rs.core.Application` subclass in the `web.xml`, as well as RESTEasy Configuration Switch, `resteasy.scan.resources`. Here is an example of the minimum configuration information needed in the `web.xml`.

```
<web-app>
    <servlet>
        <servlet-name>mySpring</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>mySpring</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>

    <servlet>
        <servlet-name>myAppSubclass</servlet-name>
        <servlet-class>org.my.app.EntryApplicationSubclass</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>myAppSubclass</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>

    <!-- required RESTEasy Configuration Switch directs auto scanning
         of the archive for Jakarta RESTful Web Services resource files
    -->
    <context-param>
        <param-name>resteasy.scan.resources</param-name>
        <param-value>true</param-value>
    </context-param>
</web-app>
```

Note that RESTEasy parameters like resteasy.scan.resources may be set in a variety of ways. See Section 3.4, “Configuration” for more information about application configuration.

If your web application contains Jakarta RESTful Web Services provider classes the RESTEasy Configuration Switch, `resteasy.scan.providers`, will also be needed. And if the url-pattern for the Jakarta RESTful Web Services Application subclass is other than `/*` you will need to declare the RESTEasy Configuration Switch, `resteasy.servlet.mapping.prefix`. This switch can be

declare either as a context-param or as a servlet init-param. It's value must be the text that preceeds the `/*`. Here is an example of such a `web.xml`:

```
<web-app>
    <servlet>
        <servlet-name>spring</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>spring</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>

    <servlet>
        <servlet-name>myAppSubclass</servlet-name>
        <servlet-class>org.my.app.EntryApplicationSubclass</servlet-class>

        <init-param>
            <param-name>resteasy.servlet.mapping.prefix</param-name>
            <param-value>/resources</param-value>
        </init-param>
    </servlet>
    <servlet-mapping>
        <servlet-name>myAppSubclass</servlet-name>
        <url-pattern>/resources/*</url-pattern>
    </servlet-mapping>

    <context-param>
        <param-name>resteasy.scan.resources</param-name>
        <param-value>true</param-value>
    </context-param>
    <context-param>
        <param-name>resteasy.scan.providers</param-name>
        <param-value>true</param-value>
    </context-param>
</web-app>
```

Above are the usages of RESTEasy Spring MVC integration usages.

48.4. Undertow Embedded Spring Container

We provide a undertow-based embedded spring container module, called "resteasy-undertow-spring". To use it, you need to add the following additional dependencies into your project:

```
<dependency>  <groupId>org.jboss.resteasy</groupId>  <artifactId>resteasy-undertow</artifactId>
      <scope>test</scope></dependency><dependency>      <groupId>org.jboss.resteasy</groupId>
 <artifactId>resteasy-undertow-spring</artifactId>  <scope>test</scope></dependency>
pendency>
 <groupId>org.jboss.resteasy</groupId>
<artifactId>resteasy-undertow</artifactId>
 <scope>test</
scope>
</dependency><dependency>
```

```
 <groupId>org.jboss.resteasy</groupId>  <artifactId>resteasy-
undertow-spring</artifactId>
 <scope>test</
```

In the "resteasy-undertow-spring" module, we have a embedded server class called "UndertowJaxrsSpringServer". In its "undertowDeployment(...)" method, it will accept the spring context configuration file:

```
public DeploymentInfo undertowDeployment(String contextConfigLocation, String mapping)
```

We can provide a minimal spring config like the following:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:util="http://www.springframework.org/schema/util"
       xsi:schemaLocation="
          http://www.springframework.org/schema/context http://www.springframework.org/schema/
context/spring-context-2.5.xsd
        http://www.springframework.org/schema/util http://www.springframework.org/schema/util/
spring-util-2.5.xsd
      http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/
spring-beans.xsd
        ">
    <context:component-scan base-package="org.jboss.resteasy.springmvc.test"/>
    <context:annotation-config/>
    <import resource="classpath:springmvc-resteasy.xml"/>
</beans>
```

In above configuration, the "springmvc-resteasy.xml" in the classpath is provided by the "resteasy-spring" module by default. Let's name the above configuration file with "spring-servlet.xml", and the following code will include it and setup the UndertowJaxrsSpringServer and start it:

```
UndertowJaxrsSpringServer         server         =         new         UndertowJaxrsSpringServ
er();server.start();DeploymentInfo  deployment  =  server.undertowDeployment("classpath:spring-
servlet.xml",
 null);deployment.setDeploymentName(BasicSpringTest.class.getName());deployment.setContextPath("/");deployment.set
=

 new UndertowJaxrsSpringServer();server.start();DeploymentInfo

deployment


=
```

Above is the code example to setup and start UndertowJaxrsSpringServer. To see a complete example, please check the Demo Of Undertow Embedded Spring Container [https://github.com/resteasy/resteasy-examples/tree/master/resteasy-spring-undertow] as usage example.

48.5. Processing Spring Web REST annotations in RESTEasy

RESTEasy also provides the ability to process Spring Web REST annotations (i.e. Spring classes annotated with `@RestController`) and handle related REST requests without delegating to Spring MVC. This functionality is currently experimental.

In order for RESTEasy to be able to process Spring `@RestController`, you first need to include the following dependency.

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

Currently RESTEasy does not auto-scan for `@RestController` annotated classes, so you need to add all `@RestController` annotated classes to your `web.xml` file as shown in the following example.

```
<web-app>
   <display-name>RESTEasy application using Spring REST annotations</display-name>

    <context-param>
        <param-name>resteasy.scanned.resource.classes.with.builder</param-name>
                                                                                <param-
value>org.jboss.resteasy.spi.metadata.SpringResourceBuilder:org.example.Controller1,org.example.Controller2</
param-value>
    </context-param>

   ...
</web-app>
```

In the example above, `Controller1` and `Controller2` are registered and are expected to be annotated with `@RestController`.

The list of the currently supported annotations can be found below:

Table 48.1.

Annotation	Comment
@RestController	

Annotation	Comment
@RequestMapping	
@GetMapping	
@PostMapping	
@PutMapping	
@DeleteMapping	
@PatchMapping	
@RequestParam	
@RequestHeader	
@MatrixVariable	
@PathVariable	
@CookieValue	
@RequestBody	
@ResponseStatus	Only supported as a method annotation
@RequestParam	

Furthermore, the use of `org.springframework.http.ResponseEntity` as a return value is supported as is the use of `javax.servlet.http.HttpServletRequest` and `javax.servlet.http.HttpServletResponse` as method parameters.

To see an example of the usage, please check the RESTEasy support of Spring REST annotations [https://github.com/resteasy/resteasy-examples/tree/master/resteasy-spring-rest] sample project we provided.

48.6. Spring Boot starter

The RESTEasy project has its support for Spring Boot integration. It was originally developed by PayPal team and has been donated to RESTEasy community. The project is currently maintained here: RESTEasy Spring Boot Starter Project [https://github.com/resteasy/resteasy-spring-boot].

Here is the usage in brief:

Firstly, add dependency to your Spring Boot application:

```
<dependency>        <groupId>org.jboss.resteasy</groupId>        <artifactId>resteasy-spring-
boot-starter</artifactId>      <version>${latest_version_of_restesy_spring_boot}</version>
 <scope>runtime</scope></dependency>
pendency>
 <groupId>org.jboss.resteasy</groupId>   <artifactId>resteasy-spring-
boot-starter</artifactId>
<version>${latest_version_of_restesy_spring_boot}</version>
 <scope>runtime</
```

And then you can use Spring annotation `@Component` to register your Jakarta RESTful Web Services `Application` class:

```
package          com.sample.app;import          org.springframework.stereotype.Component;import
 javax.ws.rs.ApplicationPath;import  javax.ws.rs.core.Application;@Component@ApplicationPath("/
sample-app/")public class JaxrsApplication extends Application {}

com.sample.app;import
org.springframework.stereotype.Component;import
javax.ws.rs.ApplicationPath;import

javax.ws.rs.core.Application;
@Component@ApplicationPath("/sample-
app/")public class JaxrsApplication extends Application
{
```

Finally, to register Jakarta RESTful Web Services resources and providers, just define them as Spring beans, and they will be automatically registered. Notice that Jakarta RESTful Web Services resources can be singleton or request scoped, while Jakarta RESTful Web Services providers must be singletons.

To see complete examples, please check the sample-app [https://github.com/resteasy/resteasy-spring-boot/tree/master/sample-app] in the project codebase.

48.7. Upgrading in WildFly

Note. As noted inSection 3.1.2, “Upgrading RESTEasy within WildFly”, the RESTEasy distribution comes with a zip file called `resteasy-jboss-modules-<version>.zip`, which can be unzipped into the modules/system/layers/base/ directory of WildFly to upgrade to a new version of RESTEasy. Because of the way resteasy-spring is used in WildFly, after unzipping the zip file, it is also necessary to remove the old resteasy-spring jar from modules/system/layers/base/org/jboss/resteasy/resteasy-spring/main/bundled/resteasy-spring-jar.

Chapter 49. CDI Integration

This module provides integration with JSR-299 (Contexts and Dependency Injection for the Jakarta EE platform)

49.1. Using CDI beans as Jakarta RESTful Web Services components

Both the Jakarta RESTful Web Services and CDI specifications introduce their own component model. On the one hand, every class placed in a CDI archive that fulfills a set of basic constraints is implicitly a CDI bean. On the other hand, explicit decoration of your Java class with `@Path` or `@Provider` is required for it to become a Jakarta RESTful Web Services component. Without the integration code, annotating a class suitable for being a CDI bean with Jakarta RESTful Web Services annotations leads into a faulty result (Jakarta RESTful Web Services component not managed by CDI) The resteasy-cdi module is a bridge that allows RESTEasy to work with class instances obtained from the CDI container.

During a web service invocation, resteasy-cdi asks the CDI container for the managed instance of a Jakarta RESTful Web Services component. Then, this instance is passed to RESTEasy. If a managed instance is not available for some reason (the class is placed in a jar which is not a bean deployment archive), RESTEasy falls back to instantiating the class itself.

As a result, CDI services like injection, lifecycle management, events, decoration and interceptor bindings can be used in Jakarta RESTful Web Services components.

49.2. Default scopes

A CDI bean that does not explicitly define a scope is `@Dependent` scoped by default. This pseudo scope means that the bean adapts to the lifecycle of the bean it is injected into. Normal scopes (request, session, application) are more suitable for Jakarta RESTful Web Services components as they designate component's lifecycle boundaries explicitly. Therefore, the resteasy-cdi module alters the default scoping in the following way:

- If a Jakarta RESTful Web Services root resource does not define a scope explicitly, it is bound to the Request scope.
- If a Jakarta RESTful Web Services Provider or `javax.ws.rs.Application` subclass does not define a scope explicitly, it is bound to the Application scope.

Warning

Since the scope of all beans that do not declare a scope is modified by resteasy-cdi, this affects session beans as well. As a result, a conflict occurs if the scope of a

stateless session bean or singleton is changed automatically as the spec prohibits these components to be @RequestScoped. Therefore, you need to explicitly define a scope when using stateless session beans or singletons. This requirement is likely to be removed in future releases.

49.3. Configuration within WildFly

CDI integration is provided with no additional configuration with WildFly.

49.4. Configuration with different distributions

Provided you have an existing RESTEasy application, all that needs to be done is to add the resteasy-cdi jar into your project's `WEB-INF/lib` directory. When using maven, this can be achieve by defining the following dependency.

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

Furthermore, when running a pre-Servlet 3 container, the following context parameter needs to be specified in web.xml. (This is done automatically via web-fragment in a Servlet 3 environment)

```
<context-param>
    <param-name>resteasy.injector.factory</param-name>
    <param-value>org.jboss.resteasy.cdi.CdiInjectorFactory</param-value>
</context-param>
```

When deploying an application to a Servlet container that does not support CDI out of the box (Tomcat, Jetty, Google App Engine), a CDI implementation needs to be added first. Weld-servlet module [http://docs.jboss.org/weld/reference/latest/en-US/html/environments.html] can be used for this purpose.

Chapter 50. RESTEasy Client API

50.1. Jakarta RESTful Web Services Client API

The Jakarta RESTful Web Services includes a client API so that you can make http requests to your remote RESTful web services. It is a 'fluent' request building API with really 3 main classes: Client, WebTarget, and Response. The Client interface is a builder of WebTarget instances. WebTarget represents a distinct URL or URL template from which you can build more sub-resource WebTargets or invoke requests on.

There are really two ways to create a Client. Standard way, or you can use the ResteasyClientBuilder class. The advantage of the latter is that it gives you a few more helper methods to configure your client.

```
Client client = ClientBuilder.newClient();
... or...
Client client = ClientBuilder.newBuilder().build();
WebTarget target = client.target("http://foo.com/resource");
Response response = target.request().get();
String value = response.readEntity(String.class);
response.close();  // You should close connections!

Client client = ClientBuilder.newClient();
WebTarget target = client.target("http://foo.com/resource");
```

RESTEasy will automatically load a set of default providers. (Basically all classes listed in all META-INF/services/javax.ws.rs.ext.Providers files). Additionally, you can manually register other providers, filters, and interceptors through the Configuration object provided by the method call Client.configuration(). Configuration also lets you set various configuration properties that may be needed.

Each WebTarget has its own Configuration instance which inherits the components and properties registered with its parent. This allows you to set specific configuration options per target resource. For example, username and password.

One RESTEasy extension to the client API is the ability to specify that requests should be sent in "chunked" transfer mode. There are two ways of doing that. One is to configure an `org.jboss.resteasy.client.jaxrs.ResteasyWebTarget` so that all requests to that target are sent in chunked mode:

```
ResteasyClient client = (ResteasyClient)ClientBuilder.newClient();
ResteasyWebTarget target = client.target("http://localhost:8081/test");
target.setChunked(b.booleanValue());
Invocation.Builder request = target.request();
```

Alternatively, it is possible to configure a particular request to be sent in chunked mode:

```
ResteasyClient client = (ResteasyClient)ClientBuilder.newClient();
ResteasyWebTarget target = client.target("http://localhost:8081/test");
ClientInvocationBuilder request = (ClientInvocationBuilder) target.request();
request.setChunked(b);
```

Note that `org.jboss.resteasy.client.jaxrs.internal.ClientInvocationBuilder`, unlike `javax.ws.rs.client.Invocation.Builder`, is a RESTEasy class.

Note. The ability to send in chunked mode depends on the underlying transport layer; in particular, it depends on which implementation of `org.jboss.resteasy.client.jaxrs.ClientHttpEngine` is being used. Currently, only the default implementation, `ApacheHttpClient43Engine`, supports chunked mode. See Section Apache HTTP Client 4.x and other backends for more information.

Note

To follow REST principles and avoid introducing state management in applications, `javax.ws.rs.client.Client` instances do not provide support for cookie management by default. However, you can enable it if necessary using `ResteasyClientBuilder`:

```
Client client = ((ResteasyClientBuilder)
ClientBuilder.newBuilder()).enableCookieManagement().build();
```

50.2. RESTEasy Proxy Framework

The RESTEasy Proxy Framework is the mirror opposite of the Jakarta RESTful Web Services server-side specification. Instead of using Jakarta RESTful Web Services annotations to map an incoming request to your RESTFul Web Service method, the client framework builds an HTTP request that it uses to invoke on a remote RESTful Web Service. This remote service does not have to be a Jakarta RESTful Web Services service and can be any web resource that accepts HTTP requests.

RESTEasy has a client proxy framework that allows you to use Jakarta RESTful Web Services annotations to invoke on a remote HTTP resource. The way it works is that you write a Java

interface and use Jakarta RESTful Web Services annotations on methods and the interface. For example:

```
public interface SimpleClient
{
   @GET
   @Path("basic")
   @Produces("text/plain")
   String getBasic();

   @PUT
   @Path("basic")
   @Consumes("text/plain")
   void putBasic(String body);

   @GET
   @Path("queryParam")
   @Produces("text/plain")
   String getQueryParam(@QueryParam("param")String param);

   @GET
   @Path("matrixParam")
   @Produces("text/plain")
   String getMatrixParam(@MatrixParam("param")String param);

   @GET
   @Path("uriParam/{param}")
   @Produces("text/plain")
   int getUriParam(@PathParam("param")int param);
}
```

RESTEasy has a simple API based on Apache HttpClient. You generate a proxy then you can invoke methods on the proxy. The invoked method gets translated to an HTTP request based on how you annotated the method and posted to the server. Here's how you would set this up:

```
            Client client = ClientBuilder.newClient();
            WebTarget target = client.target("http://example.com/base/uri");
            ResteasyWebTarget rtarget = (ResteasyWebTarget)target;

            SimpleClient simple = rtarget.proxy(SimpleClient.class);
            simple.putBasic("hello world");
```

Alternatively you can use the RESTEasy client extension interfaces directly:

```
            ResteasyClient client = (ResteasyClient)ClientBuilder.newClient();
            ResteasyWebTarget target = client.target("http://example.com/base/uri");
```

```
            SimpleClient simple = target.proxy(SimpleClient.class);
            simple.putBasic("hello world");
```

@CookieParam works the mirror opposite of its server-side counterpart and creates a cookie header to send to the server. You do not need to use @CookieParam if you allocate your own javax.ws.rs.core.Cookie object and pass it as a parameter to a client proxy method. The client framework understands that you are passing a cookie to the server so no extra metadata is needed.

The framework also supports the Jakarta RESTful Web Services locator pattern, but on the client side. So, if you have a method annotated only with @Path, that proxy method will return a new proxy of the interface returned by that method.

50.2.1. Abstract Responses

Sometimes you are interested not only in the response body of a client request, but also either the response code and/or response headers. The Client-Proxy framework has two ways to get at this information

You may return a javax.ws.rs.core.Response.Status enumeration from your method calls:

```
@Path("/")
public interface MyProxy {
   @POST
   Response.Status updateSite(MyPojo pojo);
}
```

Internally, after invoking on the server, the client proxy internals will convert the HTTP response code into a Response.Status enum.

If you are interested in everything, you can get it with the javax.ws.rs.core.Response class:

```
@Path("/")
public interface LibraryService {

   @GET
   @Produces("application/xml")
   Response getAllBooks();
}
```

50.2.2. Response proxies

A further extension implemented by the RESTEasy client proxy framework is the "response proxy facility", where a client proxy method returns an interface that represents the information contained in a `javax.ws.rs.core.Response`. Such an interface must be annotated with `@ResponseObject` from package `org.jboss.resteasy.annotations`, and its methods may be further annotated with `@Body`, `@LinkHeaderParam`, and `@Status` from the same package, as well as `javax.ws.rs.HeaderParam`. Consider the following example.

```
@ResponseObject
public interface TestResponseObject {

   @Status
   int status();

   @Body
   String body();

   @HeaderParam("Content-Type")
   String contentType();

   ClientResponse response();
}

@Path("test")
public interface TestClient {

   @GET
   TestResponseObject get();
}

@Path("test")
public static class TestResource {

   @GET
   @Produces("text/plain")
   public String get() {
      return "ABC";
   }
}
```

Here, `TestClient` will define the client side proxy for `TestResource`. Note that `TestResource.get()` returns a `String` but the proxy based on `TestClient` will return a `TestResponseObject` on a call to `get()`:

```
     Client client = ClientBuilder.newClient();
    TestClient ClientInterface = ProxyBuilder.builder(TestClient.class, client.target("http://
localhost:8081")).build();
     TestResponseObject tro = ClientInterface.get();
```

The methods of `TestResponseObject` provide access to various pieces of information about the response received from `TestResponse.get()`. This is where the annotations on those methods come into play. `status()` is annotated with `@Status`, and a call to `status()` returns the HTTP status. Similarly, `body()` returns the returned entity, and `contentType()` returns the value of the response header Content-Type:

```
System.out.println("status: " + tro.status());
System.out.println("entity: " + tro.body());
System.out.println("Content-Type: " + tro.contentType());
```

will yield

```
status: 200
entity: ABC
Content-Type: text/plain;charset=UTF-8
```

Note that there is one other method in `TestResponseObject`, `response()`, that has no annotation. When RESTEasy sees a method in an interface annotated with `@ResponseObject` that returns a `javax.ws.rs.core.Response` (or a subclass thereof), it will return a `org.jboss.resteasy.client.jaxrs.internal.ClientResponse`. For example,

```
ClientResponse clientResponse =  tro.response();
System.out.println("Content-Length: " + clientResponse.getLength());
```

Perhaps the most interesting piece of the response proxy facility is the treatment of methods annotated with `@LinkHeaderParam`. Its simplest use is to assist in accessing a `javax.ws.rs.core.Link` returned by a resource method. For example, let's add

```
@GET
@Path("/link-header")
public Response getWithHeader(@Context UriInfo uri) {
   URI subUri = uri.getAbsolutePathBuilder().path("next-link").build();
   Link link = new LinkBuilderImpl().uri(subUri).rel("nextLink").build();
   return Response.noContent().header("Link", link.toString()).build();
}
```

```

```

to `TestResource`, add

```
@GET
@Path("link-header")
ResponseObjectInterface performGetBasedOnHeader();
```

to `ClientInterface`, and add

```
@LinkHeaderParam(rel = "nextLink")
URI nextLink();
```

to `ResponseObjectInterface`. Then calling

```
ResponseObjectInterface obj = ClientInterface.performGetBasedOnHeader();
System.out.println("nextLink(): " + obj.nextLink());
```

will access the `LinkHeader` returned by `TestResource.getWithHeader()`:

```
nextlink: http://localhost:8081/test/link-header/next-link
```

Last but not least, let's add

```
@GET
@Produces("text/plain")
@Path("/link-header/next-link")
public String getHeaderForward() {
   return "forwarded";
}
```

to `TestResource` and

```
@GET
@LinkHeaderParam(rel = "nextLink")
String followNextLink();
```

to `ResponseObjectInterface`. Note that, unlike `ResponseObjectInterface.nextLink()`, `followNextLink()` is annotated with `@GET`; that is, it qualifies as (the client proxy to) a resource method. When executing `followNextLink()`, RESTEasy will retrieve the value of the `Link` returned by `TestResource.getWithHeader()` and then will make a GET invocation on the `URL` in that `Link`. Calling

```
System.out.println("followNextLink(): " + obj.followNextLink());
```

causes RESTEasy to retrieve the `URL` http://localhost:8081/test/link-header/next-link from the call to `TestResource.getWithHeader()` and then perform a GET on it, invoking `TestResource.getHeaderForward()`:

```
followNextLink(): forwarded
```

Note. This facility for extracting a `URL` and following it is a step toward supporting the Representation State Transfer principle of HATEOAS. For more information, see RESTful Java with JAX-RS 2.0, 2nd Edition [http://shop.oreilly.com/product/0636920028925.do] by Bill Burke.

50.2.3. Giving client proxy an ad hoc URI

Client proxies figure out appropriate URIs for targeting resource methods by looking at `@Path` annotations in the client side interface, but it is also possible to pass URIs explicitly to the proxy through the use of the `org.jboss.resteasy.annotations.ClientURI` annotation. For example, let `TestResource` be a client side interface and `TestResourceImpl` a server resource:

```
@Path("")
public interface TestResource {

   @GET
   @Path("dispatch")
   public String dispatch(@ClientURI String uri);
}

@Path("")
public static class TestResourceImpl {
```

```
   @GET
   @Path("a")
   public String a() {
      return "a";
   }

   @GET
   @Path("b")
   public String b() {
      return "b";
   }
}
```

Calling `TestResource.dispatch()` allows specifying a specific URI for accessing a resource method. In the following, let BASE_URL be the address of the `TestResourceImpl` resource.

```
private static String BASE_URL = "http://localhost:8081/";
...
public void test() throws Exception
{
   ResteasyClient client = (ResteasyClient)ClientBuilder.newClient();
   TestResource proxy = client.target(BASE_URL).proxy(TestResource.class);
   String name = proxy.dispatch(BASE_URL + "a");
   System.out.println("name: " + name);
   name = proxy.dispatch(BASE_URL + "b");
   System.out.println("name: " + name);
   client.close();
}
```

Then passing "http://localhost:8081/a" and "http://localhost/b" to `dispatch()` invokes `TestResourceImp.a()` and `TestResourceImpl.b()` respectively, yielding the output

```
name: a
name: b
```

50.2.4. Sharing an interface between client and server

It is generally possible to share an interface between the client and server. In this scenario, you just have your Jakarta RESTful Web Services services implement an annotated interface and then reuse that same interface to create client proxies to invoke on the client-side.

50.3. Apache HTTP Client 4.x and other backends

Network communication between the client and server is handled by default in RESTEasy. The interface between the RESTEasy Client Framework and the network is defined by RESTEasy's `ClientHttpEngine` interface. RESTEasy ships with multiple implementations of this interface.

The default implementation is `ApacheHttpClient43Engine`, which uses version 4.3 of the `HttpClient` from the Apache `HttpComponents` project.

`ApacheHttpAsyncClient4Engine`, instead, is built on top of *HttpAsyncClient* (still from the Apache *HttpComponents* project) with internally dispatches requests using a non-blocking IO model.

`JettyClientEngine` is built on top of *Eclipse Jetty* HTTP engine, which is possibly an interesting option for those already running on the Jetty server.

`VertxClientHttpEngine` is built on top of *Eclipse Vert.x*, which provides a non-blocking HTTP client based on Vert.x framework.

`ReactorNettyClientHttpEngine` is built on top of *Reactor Netty*, which provides a non-blocking HTTP client based on Netty framework.

Finally, `InMemoryClientEngine` is an implementation that dispatches requests to a server in the same JVM and `URLConnectionEngine` is an implementation that uses `java.net.HttpURLConnection`.

Table 50.1.

RESTEasy ClientHttpEngine implementations	
ApacheHttpClient43Engine	Uses HttpComponents HttpClient 4.3+
ApacheHttpAsyncClient4Engine	Uses HttpComponents HttpAsyncClient
JettyClientEngine	Uses Eclipse Jetty
ReactorNettyClientHttpEngine	Uses Reactor Netty
VertxClientHttpEngine	Uses Eclipse Vert.x
InMemoryClientEngine	Dispatches requests to a server in the same JVM
URLConnectionEngine	Uses java.net.HttpURLConnection

The RESTEasy Client Framework can also be customized. The user can provide their own implementations of `ClientHttpEngine` to the `ResteasyClient`.

```
ClientHttpEngine myEngine = new ClientHttpEngine() {
    protected SSLContext sslContext;
    protected HostnameVerifier hostnameVerifier;
```

```
    @Override
    public ClientResponse invoke(ClientInvocation request) {
        // implement your processing code and return a
        // org.jboss.resteasy.client.jaxrs.internal.ClientResponse
        // object.
    }

    @Override
    public SSLContext getSslContext() {
       return sslContext;
    }

    @Override
    public HostnameVerifier getHostnameVerifier() {
       return hostnameVerifier;
    }

    @Override
    public void close() {
       // do nothing
    }
};

ResteasyClient                                        client                                        =
 ((ResteasyClientBuilder)ClientBuilder.newBuilder()).httpEngine(myEngine).build();
```

RESTEasy and `HttpClient` make reasonable default decisions so that it is possible to use the client framework without ever referencing `HttpClient`. For some applications it may be necessary to drill down into the `HttpClient` details. `ApacheHttpClient43Engine` can be supplied with an instance of `org.apache.http.client.HttpClient` and an instance of `org.apache.http.protocol.HttpContext`, which can carry additional configuration details into the `HttpClient` layer.
`HttpContextProvider` is a RESTEasy provided interface through which a custom `HttpContext` is supplied to `ApacheHttpClient43Engine`.

```
package org.jboss.resteasy.client.jaxrs.engines;

import org.apache.http.protocol.HttpContext;

public interface HttpContextProvider {
   HttpContext getContext();
}
```

Here is an example of providing a custom HttpContext

```
DefaultHttpClient httpClient = new DefaultHttpClient();
ApacheHttpClient43Engine engine = new ApacheHttpClient43Engine(httpClient,
```

```
   new HttpContextProvider() {
           @Override
           public HttpContext getContext() {
              // Configure HttpClient to authenticate preemptively
              // by prepopulating the authentication data cache.
              // 1. Create AuthCache instance
              AuthCache authCache = new BasicAuthCache();
              // 2. Generate BASIC scheme object and add it to the local auth cache
              BasicScheme basicAuth = new BasicScheme();
              authCache.put(getHttpHost(url), basicAuth);
              // 3. Add AuthCache to the execution context
              BasicHttpContext localContext = new BasicHttpContext();
              localContext.setAttribute(ClientContext.AUTH_CACHE, authCache);
              return localContext;
           }
});
```

50.3.1. HTTP redirect

The `ClientHttpEngine` implementations based on Apache `HttpClient` support HTTP redirection. The feaure is disabled by default and has to be enabled by users explicitly:

```
ApacheHttpClient43Engine engine = new ApacheHttpClient43Engine();
engine.setFollowRedirects(true);
Client client = ((ResteasyClientBuilder)ClientBuilder.newBuilder()).httpEngine(engine).build();
```

50.3.2. Configuring SSL

To enable SSL on client, a `ClientHttpEngine` containing a SSLContext can be created to build client as in the following example:

```
ClientHttpEngine myEngine = new ClientHttpEngine() {
   ...
   public void setSslContext(SSLContext sslContext) {
      this.sslContext = sslContext;
   }

   @Override
   public HostnameVerifier getHostnameVerifier() {
      return hostnameVerifier;
   }
};
myEngine.setSslContext(mySslContext)
ResteasyClient                                    client                                    =
 ((ResteasyClientBuilder)ClientBuilder.newBuilder()).httpEngine(myEngine).build();
```

An alternative is to set up a keystore and truststore and pass a custom SslContext to ClientBuilder:

```
Client sslClient = ClientBuilder.newBuilder().sslContext(mySslContext).build();
```

If you don't want to create a SSLContext, you can build client with a keystore and truststore. Note if both SSLContext and keystore/truststore are configured, the later will be ignored by Resteasy ClientBuilder.

```
Client sslClient = ClientBuilder.newBuilder().keystore(keystore,mypassword).
                     trustKeystore(trustStore).build();
```

During handshaking, a custom HostNameVerifier can be called to allow the connection if URL's hostname and the server's identification hostname match.

```
Client                          sslClient                          =
 ((ResteasyClientBuilder)ClientBuilder.newBuilder()).sslContext(mysslContext)
                      .hostnameVerifier(myhostnameVerifier).build();
```

Resteasy provides another simple way to set up a HostnameVerifier. It allows configuring ResteasyClientBuilder with a `HostnameVerificationPolicy` without creating a custom HostNameVerifier:

```
Client                          sslClient                          =
 ((ResteasyClientBuilder)ClientBuilder.newBuilder()).sslContext(mysslContext)

 .hostnameVerification(ResteasyClientBuilder.HostnameVerificationPolicy.ANY).build();
```

- Setting HostnameVerificationPolicy.ANY will allow all connections without a check.
- HostnameVerificationPolicy.WILDCARD only allows wildcards in subdomain names i.e. *.foo.com.
- HostnameVerificationPolicy.STRICT checks if DNS names match the content of the Public Suffix List (https://publicsuffix.org/list/public_suffix_list.dat). Please note if this public suffix list isn't the check you want, you should create your own HostNameVerifier instead of this policy setting.

50.3.3. HTTP proxy

The `ClientHttpEngine` implementations based on Apache `HttpClient` support HTTP proxy. This feature can be enabled by setting specific properties on the builder:

- `org.jboss.resteasy.jaxrs.client.proxy.host`
- `org.jboss.resteasy.jaxrs.client.proxy.port`
- `org.jboss.resteasy.jaxrs.client.proxy.scheme`

```
Client                                          client                                          =

 "someproxy.com").property("org.jboss.resteasy.jaxrs.client.proxy.port", 8080).build();
```

50.3.4. Apache HTTP Client 4.3 APIs

The RESTEasy Client framework automatically creates and properly configures the underlying Apache HTTP Client engine. When the `ApacheHttpClient43Engine` is manually created, though, the user can either let it build and use a default `HttpClient` instance or provide a custom one:

```
public ApacheHttpClient43Engine() {
   ...
}

public ApacheHttpClient43Engine(HttpClient httpClient) {
   ...
}

public ApacheHttpClient43Engine(HttpClient httpClient, boolean closeHttpClient) {
   ...
}
```

The *closeHttpClient* parameter on the last constructor above allows controlling whether the Apache `HttpClient` is to be closed upon engine finalization. The default value is *true*. When a custom `HttpClient` instance is not provided, the default instance will always be closed together with the engine.

For more information about HttpClient (4.x), see the documentation at https://hc.apache.org/index.html/ [https://hc.apache.org/index.html].

Note. It is important to understand the difference between "releasing" a connection and "closing" a connection. **Releasing** a connection makes it available for reuse. **Closing** a connection frees its resources and makes it unusable.

If an execution of a request or a call on a proxy returns a class other than `Response`, then RESTEasy will take care of releasing the connection. For example, in the fragments

```
WebTarget target = client.target("http://localhost:8081/customer/123");
String answer = target.request().get(String.class);
```

or

```
ResteasyWebTarget target = client.target("http://localhost:8081/customer/123");
RegistryStats stats = target.proxy(RegistryStats.class);
RegistryData data = stats.get();
```

RESTEasy will release the connection under the covers. The only counterexample is the case in which the response is an instance of `InputStream`, which must be closed explicitly.

On the other hand, if the result of an invocation is an instance of `Response`, then Response.close() method must be used to released the connection.

```
WebTarget target = client.target("http://localhost:8081/customer/123");
Response response = target.request().get();
System.out.println(response.getStatus());
response.close();
```

You should probably execute this in a try/finally block. Again, releasing a connection only makes it available for another use. **It does not normally close the socket.**

On the other hand, `ApacheHttpClient43Engine.finalize()` will close any open sockets, unless the user set *closeHttpClient* as *false* when building the engine, in which case he is responsible for closing the connections.

Note that if `ApacheHttpClient43Engine` has created its own instance of `HttpClient`, it is not necessary to wait for `finalize()` to close open sockets. The `ClientHttpEngine` interface has a `close()` method for this purpose.

If your javax.ws.rs.client.Client class has created the engine automatically for you, you should call Client.close() and this will clean up any socket connections.

Finally, given having explicit `finalize()` methods can badly affect performances, the `org.jboss.resteasy.client.jaxrs.engines.ManualClosingApacheHttpClient43Engine` flavour of `org.jboss.resteasy.client.jaxrs.engines.ApacheHttpClient43Engine` can be

used. With that the user is always responsible for calling `close()` as no `finalize()` is there to do that before object garbage collection.

50.3.5. Asynchronous HTTP Request Processing

RESTEasy's default async engine implementation class is *ApacheHttpAsyncClient4Engine*. It can be set as the active engine by calling method *useAsyncHttpEngine* in *ResteasyClientBuilder*.

```
                                                Client            asyncClient            =
((ResteasyClientBuilder)ClientBuilder.newBuilder()).useAsyncHttpEngine()
                            .build();
   Future<Response> future = asyncClient
                            .target("http://locahost:8080/test").request()
                            .async().get();
   Response res = future.get();
   Assert.assertEquals(HttpResponseCodes.SC_OK, res.getStatus());
   String entity = res.readEntity(String.class);
```

50.3.5.1. InvocationCallbacks

InvocationCallbacks are called from within the io-threads and thus must not block or else the application may slow down to a halt. Reading the response is safe because the response is buffered in memory, as are other async and in-memory client-invocations that submit-calls returning a future not containing Response, InputStream or Reader.

```
  final CountDownLatch latch = new CountDownLatch(1);
  Future<String> future = nioClient.target(generateURL("/test")).request()
        .async().get(new InvocationCallback<String>()
               {
                  @Override
                  public void completed(String s)
                  {
                     Assert.assertEquals("get", s);
                     latch.countDown();
                     throw new RuntimeException("for the test of it");
                  }

                  @Override
                  public void failed(Throwable error)
                  {
                  }
               });
  String entity = future.get();
  Assert.assertEquals("get", entity);
```

InvocationCallbacks may be called seemingly "after" the future-object returns. Thus, responses should be handled solely in the InvocationCallback.

InvocationCallbacks will see the same result as the future-object and vice versa. Thus, if the invocationcallback throws an exception, the future-object will not see it. This is the reason to handle responses only in the InvocationCallback.

50.3.5.2. Async Engine Usage Considerations

Asynchronous IO means non-blocking IO utilizing few threads, typically at most as many threads as number of cores. As such, performance may profit from fewer thread switches and less memory usage due to fewer thread-stacks. But doing synchronous, blocking IO (the invoke-methods not returning a future) may suffer, because the data has to be transferred piecewise to/from the io-threads.

Request-Entities are fully buffered in memory, thus *HttpAsyncClient* is unsuitable for very large uploads. Response-Entities are buffered in memory, except if requesting a Response, InputStream or Reader as Result. Thus for large downloads or COMET, one of these three return types must be requested, but there may be a performance penalty because the response-body is transferred piecewise from the io-threads. When using InvocationCallbacks, the response is always fully buffered in memory.

50.3.6. Jetty Client Engine

As a drop in replacement, RESTEasy allows selecting a Jetty 9.4+ based HTTP engine. The Jetty implementation is newer and less tested, but it may end up being a good choice when relying on Jetty as server side already. The Jetty Server can even share execution resources with Client libraries if you configure them to use e.g. the same QueuedThreadPool.

The Jetty engine is enabled by adding a dependency to the *org.jboss.resteasy:resteasy-client-jetty* artifact to the Maven project; then the client can be built as follows:

```
ResteasyClient client = ((ResteasyClientBuilder)ClientBuilder.newBuilder()).clientEngine(
   new JettyClientEngine(new HttpClient())).build();
```

50.3.7. Vertx Client Engine

Still as a drop in replacement, RESTEasy allows selecting a Vert.x-based HTTP engine. The Vert.x implementation can perform asynchronous client invocations. It provides the following features:

- HTTP/1.1
- HTTP/2
- SSL/TLS (including native SSL engine)

- Efficient client connection pooling
- Optional native IO on Linux and BSD for greater performance
- Domain sockets
- HTTP Metrics with Dropwizard or Micrometer

The Vert.x engine is enabled by adding a dependency to the *org.jboss.resteasy:resteasy-client-vertx* artifact to the Maven project; then the client can be built as follows:

```
VertxClientHttpEngine engine = new VertxClientHttpEngine();
ResteasyClient client = ((ResteasyClientBuilder)ClientBuilder.newBuilder())
  .clientEngine(engine).build();
```

A Vert.x instance can also be provided when creating the client engine, as well as options configuration:

```
HttpClientOptions options = new HttpClientOptions()
 .setSsl(true);
 .setTrustStoreOptions(new JksOptions()
   .setPath("/path/to/your/truststore.jks")
   .setPassword("password-of-your-truststore")
);
VertxClientHttpEngine engine = new VertxClientHttpEngine(vertx, options);
```

You can read more about HttpClient configuration here [https://vertx.io/docs/vertx-core/java/#_making_requests].

50.3.8. Reactor Netty Client Engine

Still as a drop in replacement, RESTEasy allows selecting a Reactor Netty based HTTP engine. The Reactor Netty implementation is newer and less tested, but can be a good choice if the user application is already dependening on Netty and performs asynchronous client invocations.

The Reactor Netty engine is enabled by adding a dependency to the *org.jboss.resteasy:resteasy-client-reactor-netty* artifact to the Maven project; then the client can be built as follows:

```
ReactorNettyClientHttpEngine engine = new ReactorNettyClientHttpEngine(
   HttpClient.create(),
   new DefaultChannelGroup(new DefaultEventExecutor()),
   HttpResources.get());
ResteasyClient client = ((ResteasyClientBuilder)ClientBuilder.newBuilder())
   .clientEngine(engine).build();
```

When coupled with the MonoRxInvoker, this has several benefits. It supports things like this:

```
webTarget.path("/foo").get().rx(MonoRxInvoker.class).map(...).subscribe()
```

in order to achieve non-blocking HTTP client calls. This allows leveraging some reactor features:

- the ability for a `Mono#timeout` set on the response to aggressively terminate the HTTP request;
- the ability to pass a (reactor) context from client calls into `ReactorNettyClientHttpEngine`.

For some sample code, see `org.jboss.resteasy.reactor.ReactorTest` in the RESTEasy module resteasy-reactor.

Chapter 51. MicroProfile Rest Client

As the microservices style of system architecture (see, for example, Microservices [https://martinfowler.com/articles/microservices.html] by Martin Fowler) gains increasing traction, new API standards are coming along to support it. One set of such standards comes from the Microprofile Project [https://microprofile.io/] supported by the Eclipse Foundation, and among those is one, MicroProfile Rest Client [https://microprofile.io/project/eclipse/microprofile-rest-client], of particular interest to RESTEasy and Jakarta RESTful Web Services. In fact, it is intended to be based on, and consistent with, Jakarta RESTful Web Services, and it includes ideas already implemented in RESTEasy. For a more detailed description of MicroProfile Rest Client, see https://github.com/eclipse/microprofile-rest-client. In particular, the API code is in https://github.com/eclipse/microprofile-rest-client/tree/master/api. and the specification is in https://github.com/eclipse/microprofile-rest-client/tree/master/spec.

IMPORTANT: As of RESTEasy 5.0.0 the MicroProfile integration has moved to a new project, group id, artifact id and version. The new group id is `org.jboss.resteasy.microprofile`.The artifact id for the client is now `microprofile-client`. To use the MicroProfile Config sources the artifact id is `microprofile-config`. Finally for context propagation the new artifact is is `microprofile-context-propagation`

You could also use the RESTEasy MicroProfile BOM:

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

51.1. Client proxies

One of the central ideas in MicroProfile Rest Client is a version of *distributed object communication*, a concept implemented in, among other places, CORBA [http://www.corba.org/orb_basics.htm], Java RMI, the JBoss Remoting project, and RESTEasy. Consider the resource

```
@Path("resource")
public class TestResource {

   @Path("test")
   @GET
   String test() {
      return "test";
   }
```

```
}
```

The Jakarta RESTful Web Services native way of accessing `TestResource` looks like

```
Client client = ClientBuilder.newClient();
String response = client.target("http://localhost:8081/test").request().get(String.class);
```

The call to `TestResource.test()` is not particularly onerous, but calling `test()` directly allows a more natural syntax. That is exactly what Microprofile Rest Client supports:

```
@Path("resource")
public interface TestResourceIntf {

   @Path("test")
   @GET
   public String test();
}

TestResourceIntf service = RestClientBuilder.newBuilder()
                           .baseUrl("http://localhost:8081/")
                           .build(TestResourceIntf.class);
String s = service.test();
```

The first four lines of executable code are spent creating a proxy, `service`, that implements `TestResourceIntf`, but once that is done, calls on `TestResource` can be made very naturally in terms of `TestResourceIntf`, as illustrated by the call `service.test()`.

Beyond the natural syntax, another advantage of proxies is the way the proxy construction process quietly gathers useful information from the implemented interface and makes it available for remote invocations. Consider a more elaborate version of `TestResourceIntf`:

```
@Path("resource")
public interface TestResourceIntf2 {

   @Path("test/{path}")
   @Consumes("text/plain")
   @Produces("text/html")
   @POST
   public String test(@PathParam("path") String path, @QueryParam("query") String query, String
 entity);
}
```

Calling `service.test("p", "q", "e")` results in an HTTP message that looks like

```
POST /resource/test/p/?query=q HTTP/1.1
Accept: text/html
Content-Type: text/plain
Content-Length: 1

e
```

The HTTP verb is derived from the `@POST` annotation, the request URI is derived from the two instances of the `@Path` annotation (one on the class, one on the method) plus the first and second parameters of `test()`, the Accept header is derived from the `@Produces` annotation, and the Content-Type header is derived from the `@Consumes` annotation,

Using the Jakarta RESTful Web Services API, `service.test("p", "q", "e")` would look like the more verbose

```
Client client = ClientBuilder.newClient();
String response = client.target("http://localhost:8081/resource/test/p")
                     .queryParam("query", "q")
                     .request()
                     .accept("text/html")
                     .post(Entity.entity("e", "text/plain"), String.class);
```

One other basic facility offered by MicroProfile Rest Client is the ability to configure the client environment by registering providers:

```
TestResourceIntf service = RestClientBuilder.newBuilder()
                              .baseUrl("http://localhost:8081/")
                              .register(MyClientResponseFilter.class)
                              .register(MyMessageBodyReader.class)
                              .build(TestResourceIntf.class);
```

Naturally, the registered providers should be relevant to the client environment, rather than, say, a `ContainerResponseFilter`.

Note

So far, the MicroProfile Rest Client should look familiar to anyone who has used the RESTEasy client proxy facility (Section ""RESTEasy Proxy Framework"). The construction in the previous listing would look like

```
ResteasyClient client = (ResteasyClient) ResteasyClientBuilder.newClient();
```

```
TestResourceIntf service = client.target("http://localhost:8081/")
                                 .register(MyClientResponseFilter.class)
                                 .register(MyMessageBodyReader.class)
                                 .proxy(TestResourceIntf.class);
```

in RESTEasy.

51.2. Concepts imported from Jakarta RESTful Web Services

Beyond the central concept of the client proxy, some basic concepts in MicroProfile Client originate in Jakarta RESTful Web Services. Some of these have already been introduced in the previous section, since the interface implemented by a client proxy represents the facilities provided by a Jakarta RESTful Web Services server. For example, the HTTP verb annotations and the `@Consumes` and `@Produces` annotations originate on the Jakarta RESTful Web Services server side. Injectable parameters annotated with `@PathParameter`, `@QueryParameter`, etc., also come from Jakarta RESTful Web Services.

Nearly all of the provider concepts supported by MicroProfile Client also originate in Jakarta RESTful Web Services. These are:

- javax.ws.rs.client.ClientRequestFilter
- javax.ws.rs.client.ClientResponseFilter
- javax.ws.rs.ext.MessageBodyReader
- javax.ws.rs.ext.MessageBodyWriter
- javax.ws.rs.ext.ParamConverter
- javax.ws.rs.ext.ReaderInterceptor
- javax.ws.rs.ext.WriterInterceptor

Like Jakarta RESTful Web Services, MicroProfile Client also has the concept of mandated providers. These are

- JSON-P `MessageBodyReader` and `MessageBodyWriter` must be provided.
- JSON-B `MessageBodyReader` and `MessageBodyWriter` must be provided if the implementation supports JSON-B.
- `MessageBodyReaders` and `MessageBodyWriters` must be provided for the following types:
  - byte[]
  - String

- InputStream
- Reader
- File

51.3. Beyond Jakarta RESTful Web Services and RESTEasy

Some concepts in MicroProfile Rest Client do not appear in either Jakarta RESTful Web Services or RESTEasy.

1. Default media type

Whenever no media type is specified by, for example, `@Consumes` or `@Produces` annotations, the media type of a request entity or response entity is "application/json". This is different than Jakarta RESTful Web Services, where the media type defaults to "application/octet-stream".

2. Declarative registration of providers

In addition to programmatic registration of providers as illustrated above, it is also possible to register providers declaratively with annotations introduced in MicroProfile Rest Client. In particular, providers can be registered by adding the `org.eclipse.microprofile.rest.client.annotation.RegisterProvider` annotation to the target interface:

```
@Path("resource")
@RegisterProvider(MyClientResponseFilter.class)
@RegisterProvider(MyMessageBodyReader.class)
public interface TestResourceIntf2 {

   @Path("test/{path}")
   @Consumes("text/plain")
   @Produces("text/html")
   @POST
   public String test(@PathParam("path") String path, @QueryParam("query") String query, String
 entity);
}
```

Declaring `MyClientResponseFilter` and `MyMessageBodyReader` with annotations eliminates the need to call `RestClientBuilder.register()`.

3. Global registration of providers

One more way to register providers is by implementing one or both of the listeners in package `org.eclipse.microprofile.rest.client.spi`:

```
public interface RestClientBuilderListener {

    void onNewBuilder(RestClientBuilder builder);
}

public interface RestClientListener {

    void onNewClient(Class<?> serviceInterface, RestClientBuilder builder);
}
```

which can access a `RestClientBuilder` upon creation of a new `RestClientBuilder` or upon the execution of `RestClientBuilder.build()`, respectively. Implementations must be declared in

```
META-INF/services/org.eclipse.microprofile.rest.client.spi.RestClientBuilderListener
```

or

```
META-INF/services/org.eclipse.microprofile.rest.client.spi.RestClientListener
```

4. Declarative specification of headers

One way of declaring a header to be included in a request is by annotating one of the resource method parameters with `@HeaderValue`:

```
@POST
@Produces(MediaType.TEXT_PLAIN)
@Consumes(MediaType.TEXT_PLAIN)
String contentLang(@HeaderParam(HttpHeaders.CONTENT_LANGUAGE) String contentLanguage, String
 subject);
```

That option is available with RESTEasy client proxies as well, but in case it is inconvenient or otherwise inappropriate to include the necessary parameter, MicroProfile Client makes a declarative alternative available through the use of the `org.eclipse.microprofile.rest.client.annotation.ClientHeaderParam` annotation:

```
@POST
@Produces(MediaType.TEXT_PLAIN)
@Consumes(MediaType.TEXT_PLAIN)
@ClientHeaderParam(name=HttpHeaders.CONTENT_LANGUAGE, value="en")
```

```
String contentLang(String subject);
```

In this example, the header value is hardcoded, but it is also possible to compute a value:

```
@POST
@Produces(MediaType.TEXT_PLAIN)
@Consumes(MediaType.TEXT_PLAIN)
@ClientHeaderParam(name=HttpHeaders.CONTENT_LANGUAGE, value="{getLanguage}")
String contentLang(String subject);

default String getLanguage() {
   return ...;
}
```

5. Propagating headers on the server

An instance of `org.eclipse.microprofile.rest.client.ext.ClientHeadersFactory`,

```
public interface ClientHeadersFactory {

/**
 * Updates the HTTP headers to send to the remote service. Note that providers
 * on the outbound processing chain could further update the headers.
 *
 * @param incomingHeaders - the map of headers from the inbound Jakarta RESTful Web Services
 request. This will
 * be an empty map if the associated client interface is not part of a Jakarta RESTful Web
 Services request.
 * @param clientOutgoingHeaders - the read-only map of header parameters specified on the
 * client interface.
 * @return a map of HTTP headers to merge with the clientOutgoingHeaders to be sent to
 * the remote service.
 */
MultivaluedMap<String, String> update(MultivaluedMap<String, String> incomingHeaders,
                                      MultivaluedMap<String, String> clientOutgoingHeaders);
}
```

if activated, can do a bulk transfer of incoming headers to an outgoing request. The default instance `org.eclipse.microprofile.rest.client.ext.DefaultClientHeadersFactoryImpl` will return a map consisting of those incoming headers listed in the comma separated configuration property

```
org.eclipse.microprofile.rest.client.propagateHeaders
```

In order for an instance of `ClientHeadersFactory` to be activated, the interface must be annotated with `org.eclipse.microprofile.rest.client.annotation.RegisterClientHeaders`. Optionally, the annotation may include a value field set to an implementation class; without an explicit value, the default instance will be used.

Although a `ClientHeadersFactory` is not officially designated as a provider, it is now (as of MicroProfile REST Client specification 1.4) subject to injection. In particular, when an instance of `ClientHeadersFactory` is managed by CDI, then CDI injection is mandatory. When a REST Client is executing in the context of a Jakarta RESTful Web Services implementation, then @Context injection into a `ClientHeadersFactory` is currently optional. RESTEasy supports CDI injection and does not currently support @Context injection.

6. ResponseExceptionMapper

The `org.eclipse.microprofile.rest.client.ext.ResponseExceptionMapper` is the client side inverse of the `javax.ws.rs.ext.ExceptionMapper` defined in Jakarta RESTful Web Services. That is, where `ExceptionMapper.toResponse()` turns an `Exception` thrown during server side processing into a `Response`, `ResponseExceptionMapper.toThrowable()` turns a `Response` received on the client side with an HTTP error status into an `Exception`. `ResponseExceptionMappers` can be registered in the same manner as other providers, that is, either programmatically or declaratively. In the absence of a registered `ResponseExceptionMapper`, a default `ResponseExceptionMapper` will map any response with status >= 400 to a `WebApplicationException`.

7. Proxy injection by CDI

MicroProfile Rest Client mandates that implementations must support CDI injection of proxies. At first, the concept might seem odd in that CDI is more commonly available on the server side. However, the idea is very consistent with the microservices philosophy. If an application is composed of a number of small services, then it is to be expected that services will often act as clients to other services.

CDI (Contexts and Dependency Injection) is a fairly rich subject and beyond the scope of this Guide. For more information, see Jakarta Contexts and Dependency Injection [https://jakarta.ee/specifications/cdi/3.0/jakarta-cdi-spec-3.0.html] (the specification), Jakarta EE Tutorial [https://eclipse-ee4j.github.io/jakartaee-tutorial/], or WELD - CDI Reference Implementation [https://docs.jboss.org/weld/reference/latest-3.1/en-US/html_single/].

The fundamental thing to know about CDI injection is that annotating a variable with `javax.inject.Inject` will lead the CDI runtime (if it is present and enabled) to create an object of the appropriate type and assign it to the variable. For example, in

```
public interface Book {
   public String getTitle();
   public void setTitle(String title);
}
```

```
public class BookImpl implements Book {

   private String title;

   @Override
   public String getTitle() {
      return title;
   }

   @Override
   public void setTitle(String title) {
      this.title = title;
   }
}

public class Author {

   @Inject private Book book;

   public Book getBook() {
      return book;
   }
}
```

The CDI runtime will create an instance of `BookImpl` and assign it to the private field `book` when an instance of `Author` is created;

In this example, the injection is done because `BookImpl` is assignable to `book`, but greater discrimination can be imposed by annotating the interface and the field with **qualifier** annotations. For the injection to be legal, every qualifier on the field must be present on the injected interface. For example:

```
@Qualifier
@Target({ElementType.TYPE, ElementType.METHOD, ElementType.PARAMETER, ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
public @interface Text {}

@Qualifier
@Target({ElementType.TYPE, ElementType.METHOD, ElementType.PARAMETER, ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
public @interface Graphic {}

@Text
public class TextBookImpl extends BookImpl { }

@Graphic
public class GraphicNovelImpl extends BookImpl { }

public class Genius {

   @Inject @Graphic Book book;
}
```

Here, the class `TextBookImpl` is annotated with the `@Text` qualifier and `GraphicNovelImpl` is annotated with `@Graphic`. It follows that an instance of `GraphicNovelImpl` is eligible for assignment to the field `book` in the `Genius` class, but an instance of `TextBookImpl` is not.

Now, in MicroProfile Rest Client, any interface that is to be managed as a CDI bean must be annotated with `@RegisterRestClient`:

```
  @Path("resource")
  @RegisterProvider(MyClientResponseFilter.class)
  public static class TestResourceImpl {

     @Inject TestDataBase db;

     @Path("test/{path}")
     @Consumes("text/plain")
     @Produces("text/html")
     @POST
       public String test(@PathParam("path") String path, @QueryParam("query") String query,
String entity) {
         return db.getByName(query);
     }
  }

  @Path("database")
  @RegisterRestClient
  public interface TestDataBase {

     @Path("")
     @POST
     public String getByName(String name);
  }
```

Here, the MicroProfile Rest Client implementation creates a proxy for a `TestDataBase` service, allowing easy access by `TestResourceImpl`. Notice, though, that there's no indication of where the `TestDataBase` implementation lives. That information can be supplied by the optional `@RegisterProvider` parameter `baseUri`:

```
  @Path("database")
  @RegisterRestClient(baseUri="https://localhost:8080/webapp")
  public interface TestDataBase {

     @Path("")
     @POST
     public String getByName(String name);
  }
```

which indicates that an implementation of `TestDatabase` can be accessed at https://localhost:8080/webapp. The same information can be supplied externally with the system variable

```
<fqn of TestDataBase>/mp-rest/uri=<URL>
```

or

```
<fqn of TestDataBase>/mp-rest/url=<URL>
```

which will override the value hardcoded in `@RegisterRestClient`. For example,

```
com.bluemonkeydiamond.TestDatabase/mp-rest/url=https://localhost:8080/webapp
```

A number of other properties will be examined in the course of creating the proxy, including, for example

```
com.bluemonkeydiamond.TestDatabase/mp-rest/providers
```

a comma separated list of provider classes to be registered with the proxy. See the MicroProfile Client documentation for more such properties.

These properties can be simplified through the use of the `configKey` field in `@RegisterRestClient`. For example, setting the `configKey` as in

```
@Path("database")
@RegisterRestClient(configKey="bmd")
public interface TestDataBase { ... }
```

allows the use of properties like

```
bmd/mp-rest/url=https://localhost:8080/webapp
```

Note that, since the configKey is not tied to a particular interface name, multiple proxies can be configured with the same properties.

8. Proxy lifecycle

Proxies should be closed so that any resources they hold can be released. Every proxy created by `RestClientBuilder` implements the `java.io.Closeable` interface, so it is always possible to cast a proxy to `Closeable` and call `close()`. A nice trick to have the proxy interface explicitly extend `Closeable`, which not only avoids the need for a cast but also makes the proxy eligible to use in a try-with-resources block:

```
@Path("resource")
public interface TestResourceIntf extends Closeable {

   @Path("test")
   @GET
   public String test();
}

TestResourceIntf service = RestClientBuilder.newBuilder()
                              .baseUrl("http://localhost:8081/")
                              .build(TestResourceIntf.class);
try (TestResourceIntf tr = service) {
   String s = service.test();
}
```

9. Asynchronous support

An interface method can be designated as asynchronous by having it return a `java.util.concurrent.CompletionStage`. For example, in

```
public interface TestResourceIntf extends Closeable {

   @Path("test")
   @GET
   public String test();

   @Path("testasync")
   @GET
   public CompletionStage<String> testAsync();
}
```

the `test()` method can be turned into the asynchronous method `testAsync()` by having it return a `CompletionStage<String>` instead of a `String`.

Asynchronous methods are made to be asynchronous by scheduling their execution on a thread distinct from the calling thread. The MicroProfile Client implementation will have a default means of doing that, but `RestClientBuilder.executorService(ExecutorService)` provides a way of substituting an application specific `ExecutorService`.

The classes `AsyncInvocationInterceptorFactory` and `AsyncInvocationInterceptor` in package `org.eclipse.microprofile.rest.client.ext` provides a means of communication between the calling thread and the asynchronous thread:

```
public interface AsyncInvocationInterceptorFactory {

    /**
     * Implementations of this method should return an implementation of the
     * AsyncInvocationInterceptor interface.  The MP Rest Client
     * implementation runtime will invoke this method, and then invoke the
     * prepareContext and applyContext methods of the
     * returned interceptor when performing an asynchronous method invocation.
     * Null return values will be ignored.
     *
     * @return Non-null instance of AsyncInvocationInterceptor
     */
    AsyncInvocationInterceptor newInterceptor();
}

public interface AsyncInvocationInterceptor {

    /**
     * This method will be invoked by the MP Rest Client runtime on the "main"
     * thread (i.e. the thread calling the async Rest Client interface method)
     * prior to returning control to the calling method.
     */
    void prepareContext();

    /**
     * This method will be invoked by the MP Rest Client runtime on the "async"
     * thread (i.e. the thread used to actually invoke the remote service and
     * wait for the response) prior to sending the request.
     */
    void applyContext();

    /**
     * This method will be invoked by the MP Rest Client runtime on the "async"
     * thread (i.e. the thread used to actually invoke the remote service and
     * wait for the response) after all providers on the inbound response flow
     * have been invoked.
     *
     * @since 1.2
     */
     void removeContext();
}
```

The following sequence of events occurs:

1. `AsyncInvocationInterceptorFactory.newInterceptor()` is called on the calling thread to get an instance of the `AsyncInvocationInterceptor`.
2. `AsyncInvocationInterceptor.prepareContext()` is executed on the calling thread to store information to be used by the request execution.

3. `AsyncInvocationInterceptor.applyContext()` is executed on the asynchronous thread.
4. All relevant outbound providers such as interceptors and filters are executed on the asynchronous thread, followed by the request invocation.
5. All relevant inbound providers are executed on the asynchronous thread, followed by executing `AsyncInvocationInterceptor.removeContext()`
6. The asynchronous thread returns.

An `AsyncInvocationInterceptorFactory` class is enabled by registering it on the client interface with `@RegisterProvider`.

10. SSL

The MicroProfile Client `RestClientBuilder` interface includes a number of methods that support the use of SSL:

```
RestClientBuilder hostnameVerifier(HostnameVerifier hostnameVerifier);
RestClientBuilder keyStore(KeyStore keyStore, String keystorePassword);
RestClientBuilder sslContext(SSLContext sslContext);
RestClientBuilder trustStore(KeyStore trustStore);
```

For example:

```
KeyStore trustStore = ... ;
HostnameVerifier verifier ... ;
TestResourceIntf service = RestClientBuilder.newBuilder()
                               .baseUrl("http://localhost:8081/")
                               .trustStore(trustStore)
                               .hostnameVerifier(verifier)
                               .build(TestResourceIntf.class);
```

It is also possible to configure `HostnameVerifier`s, `KeyStore`s, and `TrustStore`s using configuration properties:

- com.bluemonkeydiamond.TestResourceIntf/mp-rest/hostnameVerifier
- com.bluemonkeydiamond.TestResourceIntf/mp-rest/keyStore
- com.bluemonkeydiamond.TestResourceIntf/mp-rest/keyStorePassword
- com.bluemonkeydiamond.TestResourceIntf/mp-rest/keyStoreType
- com.bluemonkeydiamond.TestResourceIntf/mp-rest/trustStore
- com.bluemonkeydiamond.TestResourceIntf/mp-rest/trustStorePassword

- com.bluemonkeydiamond.TestResourceIntf/mp-rest/trustStoreType

The values of the ".../mp-rest/keyStore" and "../mp-rest/trustStore" parameters can be either classpath resources (e.g., "classpath:/client-keystore.jks") or files (e.g., "file:/home/user/client-keystore.jks").

Chapter 52. AJAX Client

RESTEasy resources can be accessed in JavaScript using AJAX using a proxy API generated by RESTEasy.

52.1. Generated JavaScript API

RESTEasy can generate a JavaScript API that uses AJAX calls to invoke Jakarta RESTful Web Services operations.

Example 52.1. First Jakarta RESTful Web Services JavaScript API example

Let's take a simple Jakarta RESTful Web Services API:

```
@Path("orders")
public interface Orders {
 @Path("{id}")
 @GET
 public String getOrder(@PathParam("id") String id){
  return "Hello "+id;
 }
}
```

The preceding API would be accessible using the following JavaScript code:

```
var order = Orders.getOrder({id: 23});
```

52.1.1. JavaScript API servlet

In order to enable the JavaScript API servlet you must configure it in your web.xml file as such:

```
<servlet>
 <servlet-name>RESTEasy JSAPI</servlet-name>
 <servlet-class>org.jboss.resteasy.jsapi.JSAPIServlet</servlet-class>
</servlet>

<servlet-mapping>
 <servlet-name>RESTEasy JSAPI</servlet-name>
 <url-pattern>/rest-js</url-pattern>
</servlet-mapping>
```

52.1.2. JavaScript API usage

Each Jakarta RESTful Web Services resource class will generate a JavaScript object of the same name as the declaring class (or interface), which will contain every Jakarta RESTful Web Services method as properties.

Example 52.2. Structure of Jakarta RESTful Web Services generated JavaScript

For example, if the Jakarta RESTful Web Services resource X defines methods Y and Z:

```
@Path("/")
public interface X{
 @GET
 public String Y();
 @PUT
 public void Z(String entity);
}
```

Then the JavaScript API will define the following functions:

```
var X = {
 Y : function(params){…},
 Z : function(params){…}
};
```

Each JavaScript API method takes an optional object as single parameter where each property is a cookie, header, path, query or form parameter as identified by their name, or the following special parameters:

Warning

The following special parameter names are subject to change.

Table 52.1. API parameter properties

Property name	Default	Description
$entity		The entity to send as a PUT, POST request.
$contentType	As determined by @Consumes.	The MIME type of the body entity sent as the Content-Type header.
$accepts	Determined by @Provides, defaults to */*.	The accepted MIME types sent as the Accept header.

Property name	Default	Description
$callback		Set to a function(httpCode, xmlHttpRequest, value) for an asynchronous call. If not present, the call will be synchronous and return the value.
$apiURL	Determined by container	Set to the base URI of your Jakarta RESTful Web Services endpoint, not including the last slash.
$username		If username and password are set, they will be used for credentials for the request.
$password		If username and password are set, they will be used for credentials for the request.

Example 52.3. Using the API

Here is an example of Jakarta RESTful Web Services API:

```
@Path("foo")
public class Foo{
 @Path("{id}")
 @GET
 public String get(@QueryParam("order") String order, @HeaderParam("X-Foo") String header,
                @MatrixParam("colour") String colour, @CookieParam("Foo-Cookie") String cookie){
  &
 }
 @POST
 public void post(String text){
 }
}
```

We can use the previous Jakarta RESTful Web Services API in JavaScript using the following code:

```
var text = Foo.get({order: 'desc', 'X-Foo': 'hello',
                    colour: 'blue', 'Foo-Cookie': 123987235444});
Foo.put({$entity: text});
```

52.1.3. Work with @Form

@Form is a RESTEasy specific annotation that allows you to re-use any @*Param annotation within an injected class. The generated JavaScript API will expand the parameters for use automatically. Support we have the following form:

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

    @FormParam("number")
    private int number;

    @HeaderParam("myHeader")
    private String header;
}
```

And the resource is like:

```
@Path("/")
public class MyResource {

    @POST
    public String postForm(@Form MyForm myForm) {...}

}
```

Then we could call the method from JavaScript API like following:

```
MyResource.postForm({stuff:"A", myHeader:"B", number:1});
```

Also, @Form supports prefix mappings for lists and maps:

```
public static class Person {
    @Form(prefix="telephoneNumbers") List<TelephoneNumber> telephoneNumbers;
    @Form(prefix="address") Map<String, Address> addresses;
}

public static class TelephoneNumber {
    @FormParam("countryCode") private String countryCode;
    @FormParam("number") private String number;
}

public static class Address {
    @FormParam("street") private String street;
    @FormParam("houseNumber") private String houseNumber;
}
```

```
@Path("person")
public static class MyResource {
    @POST
    public void postForm(@Form Person p) {...}
}
```

From JavaScript we could call the API like this:

```
MyResource.postForm({
 telephoneNumbers:[
  {"telephoneNumbers[0].countryCode":31},
  {"telephoneNumbers[0].number":12345678},
  {"telephoneNumbers[1].countryCode":91},
  {"telephoneNumbers[1].number":9717738723}
 ],
 address:[
  {"address[INVOICE].street":"Main Street"},
  {"address[INVOICE].houseNumber":2},
  {"address[SHIPPING].street":"Square One"},
  {"address[SHIPPING].houseNumber":13}
 ]
});
```

52.1.4. MIME types and unmarshalling.

The Accept header sent by any client JavaScript function is controlled by the $accepts parameter, which overrides the @Produces annotation on the Jakarta RESTful Web Services endpoint. The returned value however is controlled by the Content-Type header sent in the response as follows:

Table 52.2. Return values by MIME type

MIME	Description
text/xml,application/xml,application/*+xml	The response entity is parsed as XML before being returned. The return value is thus a DOM Document.
application/json	The response entity is parsed as JSON before being returned. The return value is thus a JavaScript Object.
Anything else	The response entity is returned raw.

Example 52.4. Unmarshalling example

The RESTEasy JavaScript client API can automatically unmarshall JSON and XML:

```
@Path("orders")
public interface Orders {
```

```
 @XmlRootElement
 public static class Order {
  @XmlElement
  private String id;

  public Order(){}

  public Order(String id){
   this.id = id;
  }
 }

 @Path("{id}/xml")
 @GET
 @Produces("application/xml")
 public Order getOrderXML(@PathParam("id") String id){
  return new Order(id);
 }

 @Path("{id}/json")
 @GET
 @Produces("application/json")
 public Order getOrderJSON(@PathParam("id") String id){
  return new Order(id);
 }
}
```

Let us look at what the preceding Jakarta RESTful Web Services API would give us on the client side:

```
// this returns a JSON objectvar orderJSON = Orders.getOrderJSON({id: "23"});orderJSON.id
  == "23";// this one returns a DOM Document whose root element is the order, with one
  child (id)// whose child is the text node valuevar orderXML = Orders.getOrderXML({id:
 "23"});orderXML.documentElement.childNodes[0].childNodes[0].nodeValue == "23";
objectvar orderJSON = Orders.getOrderJSON({id:
 "23"});orderJSON.id ==

 "23";// this one returns a DOM Document whose root element is the order, with one child
 (id)// whose child is the text node
 valuevar orderXML = Orders.getOrderXML({id:
 "23"});orderXML.documentElement.childNodes[0].childNodes[0].nodeValue ==
```

52.1.5. MIME types and marshalling.

The Content-Type header sent in the request is controlled by the $contentType parameter which overrides the @Consumes annotation on the Jakarta RESTful Web Services endpoint. The value passed as entity body using the $entity parameter is marshalled according to both its type and content type:

Table 52.3. Controlling sent entities

Type	MIME	Description
DOM Element	Empty or text/xml,application/xml,application/*+xml	The DOM Element is marshalled to XML before being sent.
JavaScript Object (JSON)	Empty or application/json	The JSON object is marshalled to a JSON string before being sent.
Anything else	Anything else	The entity is sent as is.

Example 52.5. Marshalling example

The RESTEasy JavaScript client API can automatically marshall JSON and XML:

```
@Path("orders")
public interface Orders {

 @XmlRootElement
 public static class Order {
  @XmlElement
  private String id;

  public Order(){}

  public Order(String id){
   this.id = id;
  }
 }

 @Path("{id}/xml")
 @PUT
 @Consumes("application/xml")
 public void putOrderXML(Order order){
  // store order
 }

 @Path("{id}/json")
 @PUT
 @Consumes("application/json")
 public void putOrderJSON(Order order){
  // store order
 }
}
```

Let us look at what the preceding Jakarta RESTful Web Services API would give us on the client side:

```
//   this   saves   a   JSON   objectOrders.putOrderJSON({$entity:   {id:   "23"}});//   It   is
   a   bit   more   work   with   XMLvar   order   =   document.createElement("order");var   id   =
```

```
 order});
jectOrders.putOrderJSON({$entity: {id:

"23"}});// It is a bit more work with
XMLvar order =
document.createElement("order");var id =
document.createElement("id");
order.appendChild(id);
id.appendChild(document.createTextNode("23"));Orders.putOrderXML({$entity:
```

52.2. Using the JavaScript API to build AJAX queries

The RESTEasy JavaScript API can also be used to manually construct your requests.

52.2.1. The REST object

The REST object contains the following read-write properties:

Table 52.4. The REST object

Property	Description
apiURL	Set by default to the Jakarta RESTful Web Services root URL, used by every JavaScript client API functions when constructing the requests.
log	Set to a function(string) in order to receive RESTEasy client API logs. This is useful if you want to debug your client API and place the logs where you can see them.

Example 52.6. Using the REST object

The REST object can be used to override RESTEasy JavaScript API client behaviour:

```
// Change the base URL used by the API:
REST.apiURL = "http://api.service.com";

// log everything in a div element
REST.log = function(text){
 jQuery("#log-div").append(text);
};
```

52.2.2. The REST.Request class

The REST.Request class is used to build custom requests. It has the following members:

Table 52.5. The REST.Request class

Member	Description
execute(callback)	Executes the request with all the information set in the current object. The value is never returned but passed to the optional argument callback.
setAccepts(acceptHeader)	Sets the Accept request header. Defaults to */*.
setCredentials(username, password)	Sets the request credentials.
setEntity(entity)	Sets the request entity.
setContentType(contentTypeHeader)	Sets the Content-Type request header.
setURI(uri)	Sets the request URI. This should be an absolute URI.
setMethod(method)	Sets the request method. Defaults to GET.
setAsync(async)	Controls whether the request should be asynchronous. Defaults to true.
addCookie(name, value)	Sets the given cookie in the current document when executing the request. Beware that this will be persistent in your browser.
addQueryParameter(name, value)	Adds a query parameter to the URI query part.
addMatrixParameter(name, value)	Adds a matrix parameter (path parameter) to the last path segment of the request URI.
addHeader(name, value)	Adds a request header.

Example 52.7. Using the REST.Request class

The REST.Request class can be used to build custom requests:

```
var r = new REST.Request();
r.setURI("http://api.service.com/orders/23/json");
r.setMethod("PUT");
r.setContentType("application/json");
r.setEntity({id: "23"});
r.addMatrixParameter("JSESSIONID", "12309812378123");
r.execute(function(status, request, entity){
 log("Response is "+status);
});
```

52.3. Caching Features

RESTEasy AJAX Client works well with server side caching features. But the buggy browsers cache will always prevent the function to work properly. If you'd like to use RESTEasy's caching feature with its AJAX client, you can enable 'antiBrowserCache' option:

```
REST.antiBrowserCache = true;
```

The above setting should be set once before you call any APIs.

Chapter 53. RESTEasy WADL Support

RESTEasy has its own support to generate WADL for its resources, and it supports several different containers. The following text will show you how to use this feature in different containers.

53.1. RESTEasy WADL Support for Servlet Container(Deprecated)

Note

The content introduced in this section is outdated, and the `ResteasyWadlServlet` class is deprecated because it doesn't support the GRAMMAR generation. Please check the `ResteasyWadlDefaultResource` introduced in the later section.

RESTEasy WADL uses ResteasyWadlServlet to support servlet container. It can be registered into web.xml to enable WADL feature. Here is an example to show the usages of ResteasyWadlServlet in web.xml:

```
<servlet>
 <servlet-name>RESTEasy WADL</servlet-name>
 <servlet-class>org.jboss.resteasy.wadl.ResteasyWadlServlet</servlet-class>
</servlet>

<servlet-mapping>
 <servlet-name>RESTEasy WADL</servlet-name>
 <url-pattern>/application.xml</url-pattern>
</servlet-mapping>
```

The preceding configuration in web.xml shows how to enable ResteasyWadlServlet and mapped it to /application.xml. And then the WADL can be accessed from the configured URL:

```
/application.xml
```

53.2. RESTEasy WADL Support for Servlet Container(Updated)

This section introduces the recommended way to enable WADL support under Servlet Container situation. Firstly, you need to add a class then extends the `ResteasyWadlDefaultResource` to serve a resource path. Here is an example:

```
import                              org.jboss.resteasy.wadl.ResteasyWadlDefaultResource;import
 javax.ws.rs.Path;@Path("/")public class MyWadlResource extends ResteasyWadlDefaultResource {}
org.jboss.resteasy.wadl.ResteasyWadlDefaultResource;import

javax.ws.rs.Path;
@Path("/")public class MyWadlResource extends ResteasyWadlDefaultResource
{
```

As the sample shown above, it will enable the `ResteasyWadlDefaultResource` and serves this URL by default:

```
/application.xml
```

To enable the GRAMMAR generation, you can extend the `ResteasyWadlDefaultResource` list this:

```
import                              org.jboss.resteasy.wadl.ResteasyWadlDefaultResource;import
   org.jboss.resteasy.wadl.ResteasyWadlWriter;import   javax.ws.rs.Path;@Path("/")public   class
   MyWadlResource   extends   ResteasyWadlDefaultResource   {           public   MyWadlResource()
    {                            ResteasyWadlWriter.ResteasyWadlGrammar    wadlGrammar    =    new
 ResteasyWadlWriter.ResteasyWadlGrammar();         wadlGrammar.enableSchemaGeneration();
  getWadlWriter().setWadlGrammar(wadlGrammar);    }}
org.jboss.resteasy.wadl.ResteasyWadlDefaultResource;import

org.jboss.resteasy.wadl.ResteasyWadlWriter;import

javax.ws.rs.Path;
@Path("/")public class MyWadlResource extends ResteasyWadlDefaultResource

{    public MyWadlResource()
{                             ResteasyWadlWriter.ResteasyWadlGrammar    wadlGrammar    =
new
ResteasyWadlWriter.ResteasyWadlGrammar();
wadlGrammar.enableSchemaGeneration();

getWadlWriter().setWadlGrammar(wadlGrammar);
}
```

With the above setup, the WADL module will generate GRAMMAR automatically and register the service under this url:

```
/wadl-extended/xsd0.xsd
```

Above is the basic usage of WADL module under servlet container deployment.

53.3. RESTEasy WADL support for Sun JDK HTTP Server

RESTEasy has provided a ResteasyWadlDefaultResource to generate WADL info for its embedded containers. Here is and example to show how to use it with RESTEasy's Sun JDK HTTP Server container:

```
com.sun.net.httpserver.HttpServer  httpServer  =  com.sun.net.httpserver.HttpServer.create(new
    InetSocketAddress(port),    10);org.jboss.resteasy.plugins.server.sun.http.HttpContextBuilder
                        contextBuilder                          =


               .put("/",                                ResteasyWadlGenerator
 .generateServiceRegistry(contextBuilder.getDeployment()));httpServer.start();
= com.sun.net.httpserver.HttpServer.create(new InetSocketAddress(port),

10);org.jboss.resteasy.plugins.server.sun.http.HttpContextBuilder contextBuilder =
  new

org.jboss.resteasy.plugins.server.sun.http.HttpContextBuilder();
contextBuilder.getDeployment().getActualResourceClasses()
.add(ResteasyWadlDefaultResource.class);

contextBuilder.bind(httpServer);
ResteasyWadlDefaultResource.getServices()
.put("/",
ResteasyWadlGenerator

.generateServiceRegistry(contextBuilder.getDeployment()));
```

From the above code example, we can see how ResteasyWadlDefaultResource is registered into deployment:

```
contextBuilder.getDeployment().getActualResourceClasses()
 .add(ResteasyWadlDefaultResource.class);
```

Another important thing is to use ResteasyWadlGenerator to generate the WADL info for the resources in deployment at last:

```
ResteasyWadlDefaultResource.getServices()
 .put("/",
  ResteasyWadlGenerator
   .generateServiceRegistry(contextBuilder.getDeployment()));
```

After the above configuration is set, then users can access "/application.xml" to fetch the WADL info, because ResteasyWadlDefaultResource has @PATH set to "/application.xml" as default:

```
@Path("/application.xml")
public class ResteasyWadlDefaultResource
```

53.4. RESTEasy WADL support for Netty Container

RESTEasy WADL support for Netty Container is simliar to the support for JDK HTTP Server. It also uses ResteasyWadlDefaultResource to serve '/application.xml' and ResteasyWadlGenerator to generate WADL info for resources. Here is the sample code:

```
ResteasyDeployment     deployment     =     new     ResteasyDeploymentImpl();netty     =     new

         .addPerRequestResource(ResteasyWadlDefaultResource.class);
                                          ResteasyWadlDefaultResource.getServices()           .put("/",
 ResteasyWadlGenerator.generateServiceRegistry(deployment));

= new ResteasyDeploy
men
tIm
pl();
netty


=
 new NettyJaxrsServer();netty.setDeployment(deployment);netty.setPort(port);netty.setRootResourcePath("");netty.se

.addPerRequestResource(ResteasyWadlDefaultResource.class);
```

Please note for all the embedded containers like JDK HTTP Server and Netty Container, if the resources in the deployment changes at runtime, the ResteasyWadlGenerator.generateServiceRegistry() need to be re-run to refresh the WADL info.

53.5. RESTEasy WADL Support for Undertow Container

The RESTEasy Undertow Container is a embedded Servlet Container, and RESTEasy WADL provides a connector to it. To use RESTEasy Undertow Container together with WADL support, you need to add these three components into your maven dependencies:

```
<dependency>
 <groupId>org.jboss.resteasy</groupId>
 <artifactId>resteasy-wadl</artifactId>
 <version>${project.version}</version>
</dependency>
<dependency>
 <groupId>org.jboss.resteasy</groupId>
 <artifactId>resteasy-wadl-undertow-connector</artifactId>
 <version>${project.version}</version>
</dependency>
<dependency>
```

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

The resteasy-wadl-undertow-connector provides a WadlUndertowConnector to help you to use WADL in RESTEasy Undertow Container. Here is the code example:

```
UndertowJaxrsServer server = new UndertowJaxrsServer().start();
WadlUndertowConnector connector = new WadlUndertowConnector();
connector.deployToServer(server, MyApp.class);
```

The MyApp class shown in above code is a standard Jakarta RESTful Web Services Application class in your project:

```
@ApplicationPath("/base")
public static class MyApp extends Application {
    @Override
    public Set<Class<?>> getClasses() {
        HashSet<Class<?>> classes = new HashSet<Class<?>>();
        classes.add(YourResource.class);
        return classes;
    }
}
```

After the Application is deployed to the UndertowJaxrsServer via WadlUndertowConnector, you can access the WADL info at "/application.xml" prefixed by the @ApplicationPath in your Application class. If you want to override the @ApplicationPath, you can use the other method in WadlUndertowConnector:

```
public   UndertowJaxrsServer   deployToServer(UndertowJaxrsServer   server,   Class<?   extends
 Application> application, String contextPath)
```

The "deployToServer" method shown above accepts a "contextPath" parameter, which you can use to override the @ApplicationPath value in the Application class.

Chapter 54. RESTEasy Tracing Feature

54.1. Overview

Tracing feature is a way for the users of the RESTEasy to understand what's going on internally in the container when a request is processed. It's different from the pure logging system or profiling feature, which provides more general information about the request and response.

The tracing feature provides more internal states of the Jakarta RESTful Web Services container. For example, it could be able to show what filters a request is going through, or how long time a request is processed and other kinds of information.

Currently it doesn't have a standard or spec to define the tracing feature, so the tracing feature is tightly coupled with the concrete Jakarta RESTful Web Services implementation itself. In this chapter, let's check the design and usage of the tracing feature.

54.2. Tracing Info Mode

The RESTEasy tracing feature supports three logging mode:

- OFF
- ON_DEMAND
- ALL

"ALL" will enable the tracing feature. "ON_DEMAND" mode will give the control to client side: A client can send a tracing request via HTTP header and get the tracing info back from response headers. "OFF" mode will disable the tracing feature, and this is the default mode.

54.3. Tracing Info Level

The tracing info has three levels:

- SUMMARY
- TRACE
- VERBOSE

The "SUMMARY" level will emit some brief tracing information. The "TRACE" level will produce more detailed tracing information, and the "VERBOSE" level will generate extremely detailed tracing information.

The tracing feature relies on the JBoss Logging framework to produce the tracing info, so the JBoss Logging configuration actually controls the final output of the tracing info. So it is JBoss Logging framework configuration that controls the logging threshold of the tracing info.

54.4. Basic Usages

By default, the tracing feature is turned off. If you want to enable the tracing feature, you need to add the following dependency in your project:

```
<dependency>      <groupId>org.jboss.resteasy</groupId>      <artifactId>resteasy-tracing-api</
artifactId></dependency>
pendency>
 <groupId>org.jboss.resteasy</groupId>    <artifactId>resteasy-
tracing-api</
```

Because the tracing feature is an optional feature, the above dependency is provided by the resteasy-extensions [https://github.com/resteasy/resteasy-extensions] project.

After including the dependency in your project, you can set the tracing mode and tracing level via the context-param parameters in your web project's web.xml file. Here is the example:

```
<context-param>    <param-name>resteasy.server.tracing.type</param-name>    <param-value>ALL</
param-value>         <param-name>resteasy.server.tracing.threshold</param-name>         <param-
value>SUMMARY</param-value></context-param>
param>    <param-name>resteasy.server.tracing.type</param-
name>    <param-value>ALL</param-
value>    <param-name>resteasy.server.tracing.threshold</param-
name>    <param-value>SUMMARY</param-
value></context-
```

Besides the above configuration, we also need to make sure that the underlying JBoss Logger is configured properly so it can output the tracing info as required. Here is an example of the "logging.properties":

```
#    Additional    logger    names    to    configure    (root    logger    is    always    config
ured)#loggers=org.foo.bar,    org.foo.baz#    Root    logger    levellogger.level=ALL#    Declare
    handlers    for    the    root    loggerlogger.handlers=CONSOLE,    FILE#    Declare    handlers
     for     additional     loggers#logger.org.foo.bar.handlers=XXX,     YYY#     Console     handler

                                                                                    File

tmp/jboss.loghandler.FILE.formatter=PATTERN#    The    log    format    pattern    for    both

%d{HH:mm:ss,SSS} %-5p [%c{1}] %m%n
ger is
 always configured)
#loggers=org.foo.bar,
 org.foo.baz# Root logger levellogger.level=ALL# Declare
 handlers
```

```
for the root loggerlogger.handlers=CONSOLE, FILE#
 Declare
handlers for additional loggers
#logger.org.foo.bar.handlers=XXX,

YYY
#

Console handler configurationhandler.CONSOLE=org.jboss.logmanager.handlers.ConsoleHandlerhandler.CONSOLE.propertie
handler.CONSOLE.level=ALL
handler.CONSOLE.autoFlush=true
handler.CONSOLE.formatter=PATTERN
#
 File
handler
 configurationhandler.FILE=org.jboss.logmanager.handlers.FileHandlerhandler.FILE.level=ALLhandler.FILE.properties=

format
 pattern for both logsformatter.PATTERN=org.jboss.logmanager.formatters.PatternFormatter
```

In above setting, we have set the logger level to "ALL", and output log file to "/tmp/jboss.log". In this case, we can make sure that we get all the tracing info.

After enabling the tracing feature as shown above, we should get the tracing info output like following:

```
16:21:40,110                        INFO                                                [general]
             org.jboss.resteasy.plugins.server.servlet.Servlet3AsyncHttpRequest@721299ff             START
     baseUri=[http://localhost:8081/]     requestUri=[http://localhost:8081/type]     method=[GET]
   authScheme=[n/a]   accept=n/a   accept-encoding=n/a   accept-charset=n/a   accept-language=n/
a   content-type=n/a   content-length=n/a      [   ----   ms]16:21:40,110   TRACE   [general]
     org.jboss.resteasy.plugins.server.servlet.Servlet3AsyncHttpRequest@721299ff     START_HEADERS
  Other  request  headers:  Connection=[Keep-Alive]  Host=[localhost:8081]  User-Agent=[Apache-
HttpClient/4.5.4   (Java/1.8.0_201)]       [    ----    ms]16:21:40,114    INFO       [general]
 org.jboss.resteasy.plugins.server.servlet.Servlet3AsyncHttpRequest@721299ff  PRE_MATCH_SUMMARY
   PreMatchRequest    summary:    0    filters    [    0.04    ms]16:21:40,118    DEBUG    [general]
    org.jboss.resteasy.plugins.server.servlet.Servlet3AsyncHttpRequest@721299ff    REQUEST_FILTER
   Filter   by   [io.weli.tracing.HttpMethodOverride   @60353244]   [   0.02   ms]...16:21:40,164
  INFO   [general]  org.jboss.resteasy.plugins.server.servlet.Servlet3AsyncHttpRequest@721299ff
 RESPONSE_FILTER_SUMMARY Response summary: 1 filters [ 8.11 ms]16:21:40,164 INFO  [general]
 org.jboss.resteasy.plugins.server.servlet.Servlet3AsyncHttpRequest@721299ff FINISHED Response
 status: 200 [ ---- ms]
                                    content-length=n/a
          [
----  ms]16:21:40,110  TRACE  [general]  org.jboss.resteasy.plugins.server.servlet.Servlet3AsyncHttpRequest@72129
 request headers: Connection=[Keep-Alive] Host=[localhost:8081]
User-Agent=[Apache-HttpClient/4.5.4                       (Java/1.8.0_201)]
  [  ----
 ms]16:21:40,114 INFO  [general] org.jboss.resteasy.plugins.server.servlet.Servlet3AsyncHttpRequest@721299ff PRE_M
                                    summary:                                    0
filters    [
0.04 ms]16:21:40,118 DEBUG [general] org.jboss.resteasy.plugins.server.servlet.Servlet3AsyncHttpRequest@721299ff F

by                       [io.weli.tracing.HttpMethodOverride                        @60353244]
[
0.02 ms]...16:21:40,164 INFO  [general] org.jboss.resteasy.plugins.server.servlet.Servlet3AsyncHttpRequest@721299f
```

```
                        summary:                        1                       filters
 [ 8.11 ms]
16:21:40,164 INFO  [general] org.jboss.resteasy.plugins.server.servlet.Servlet3AsyncHttpRequest@721299ff FINISHED
```

From the above tracing log output shown above, we can see that the entry of tracing log contains several parts:

- Level Of The Log Entry

  We can see the log entries have different log levels, such as "TRACE", "INFO", "DEBUG". The tracing feature maps its own tracing info levels to the JBoss Logger output levels like this.

- The Request Scope Id

  We can see the request id like:

```
org.jboss.resteasy.plugins.server.servlet.Servlet3AsyncHttpRequest@721299ff
```

  So we can know which request the log entry belongs to.

- The Type Of The Tracing Log

  tracing log entries are divided into multiple categories, such as "START_HEADERS", "REQUEST_FILTER", "FINISHED", etc.

- The Detail Of The Log Entry

  The last part of a log entry is the detail message of this entry.

In next section let's see how do we fetch the tracing info from client side.

54.5. Client Side Tracing Info

From client side, we can send request to the server side as usual, and if the server side is configured properly to produce tracing info, then the info will also be sent back to client side via response headers. For example, we can send request to the server like this:

```
$ curl -i http://localhost:8081/foo
```

And then we can get the tracing info from the response header like the following:

```
HTTP/1.1 200 OK
X-RESTEasy-Tracing-026:
 org.jboss.resteasy.plugins.server.servlet.Servlet3AsyncHttpRequest@7a49a8aa MBW       [ ---- /
  61.57 ms |   ---- %] [org.jboss.resteasy.plugins.providers.InputStreamProvider @1cbf0b08] is
 skipped
```

```
...
Date: Wed, 27 Mar 2019 09:39:50 GMT
Connection: keep-alive
X-RESTEasy-Tracing-000:
 org.jboss.resteasy.plugins.server.servlet.Servlet3AsyncHttpRequest@7a49a8aa START     [ ---- /
    ----  ms  |    ----  %]  baseUri=[http://localhost:8081/]  requestUri=[http://localhost:8081/
type] method=[GET] authScheme=[n/a] accept=*/* accept-encoding=n/a accept-charset=n/a accept-
language=n/a content-type=n/a content-length=n/a
...
X-RESTEasy-Tracing-025:
 org.jboss.resteasy.plugins.server.servlet.Servlet3AsyncHttpRequest@7a49a8aa MBW       [ ---- /
 61.42 ms |  ---- %] [org.jboss.resteasy.plugins.providers.FileRangeWriter @35b791fa] is skipped
```

From above output, we can see the tracing info is in response headers, and it's marked in sequence as in the form of "X-RESTEasy-Tracing-nnn".

54.6. Json Formatted Response

The tracing log can be returned to client side in JSON format. To use this feature, we need to choose a JSON provider for tracing module to generate JSON formatted info. There are two JSON providers you can choose from and they both support the JSON data marshalling. The first choice is to use the jackson2 provider:

```
<dependency>   <groupId>org.jboss.resteasy</groupId>   <artifactId>resteasy-jackson2-provider</
artifactId></dependency>
pendency>
 <groupId>org.jboss.resteasy</groupId>    <artifactId>resteasy-
jackson2-provider</
```

The second choice is to use the json-binding provider:

```
<dependency>      <groupId>org.jboss.resteasy</groupId>      <artifactId>resteasy-json-binding-
provider</artifactId></dependency>
pendency>
 <groupId>org.jboss.resteasy</groupId>    <artifactId>resteasy-json-
binding-provider</
```

After including either of the above module, we can send request to server to get the JSON formatted tracing info. Here is a request example(the example is provided at last section of this chapter):

```
$ curl -H "X-RESTEasy-Tracing-Accept-Format: JSON" -i http://localhost:8081/type
```

In the above curl command, we have added "X-RESTEasy-Tracing-Accept-Format: JSON" into request header, in this way we are requesting the json formatted tracing info from server, and the tracing info in response header is like the following:

```
X-RESTEasy-Tracing-000:

localhost:8081/]    requestUri=[http://localhost:8081/type]    method=[GET]    authScheme=[n/
a]         accept=*/*         accept-encoding=n/a         accept-charset=n/a         accept-
language=n/a                        content-type=n/a                        content-length=n/a

{"event":"START_HEADERS","duration":0,"timestamp":195286695053606,"text":"Other        request
  headers:  Accept=[*/*]  Host=[localhost:8081]  User-Agent=[curl/7.54.0]  X-RESTEasy-Tracing-
Accept-Format=[JSON]

{"event":"FINISHED","duration":0,"timestamp":195286729758836,"text":"Response          status:
 200","requestId":"org.jboss.resteasy.plugins.server.servlet.Servlet3AsyncHttpRequest@7f8a33b9"}]
```

The above text is the raw output from response, and we can format it to make it readable:

```
[{
    "X-RESTEasy-Tracing-000": [
        {
            "event": "START",
            "duration": 0,
            "timestamp": 195286694509932,
                 "text": "baseUri=[http://localhost:8081/] requestUri=[http://localhost:8081/
type] method=[GET] authScheme=[n/a] accept=*/* accept-encoding=n/a accept-charset=n/a accept-
language=n/a content-type=n/a content-length=n/a ",
                                                                                "requestId":
 "org.jboss.resteasy.plugins.server.servlet.Servlet3AsyncHttpRequest@7f8a33b9"
        },
        {
            "event": "START_HEADERS",
            "duration": 0,
            "timestamp": 195286695053606,
                  "text": "Other request headers: Accept=[*/*] Host=[localhost:8081] User-
Agent=[curl/7.54.0] X-RESTEasy-Tracing-Accept-Format=[JSON] ",
                                                                                "requestId":
 "org.jboss.resteasy.plugins.server.servlet.Servlet3AsyncHttpRequest@7f8a33b9"
        },
        {
            "event": "PRE_MATCH_SUMMARY",
            "duration": 14563,
            "timestamp": 195286697637157,
            "text": "PreMatchRequest summary: 0 filters",
                                                                                "requestId":
 "org.jboss.resteasy.plugins.server.servlet.Servlet3AsyncHttpRequest@7f8a33b9"
        },
 ...
        {
            "event": "FINISHED",
            "duration": 0,
            "timestamp": 195286729758836,
            "text": "Response status: 200",
                                                                                "requestId":
 "org.jboss.resteasy.plugins.server.servlet.Servlet3AsyncHttpRequest@7f8a33b9"
        }
    ]
```

```
}]
```

From above we can see the tracing info is returned as JSON text.

54.7. List Of Tracing Events

The tracing events are defined in RESTEasyServerTracingEvent [https://github.com/resteasy/resteasy-extensions/blob/master/tracing/src/main/java/org/jboss/resteasy/tracing/api/RESTEasyServerTracingEvent.java]. Here is a complete list of the tracing events and its descriptions:

- DISPATCH_RESPONSE

  Resource method invocation results to Jakarta RESTful Web Services Response.

- EXCEPTION_MAPPING

  ExceptionMapper invoked.

- FINISHED

  Request processing finished.

- MATCH_LOCATOR

  Matched sub-resource locator method.

- MATCH_PATH_FIND

  Matching path pattern.

- MATCH_PATH_NOT_MATCHED

  Path pattern not matched.

- MATCH_PATH_SELECTED

  Path pattern matched/selected.

- MATCH_PATH_SKIPPED

  Path pattern skipped as higher-priority pattern has been selected already.

- MATCH_RESOURCE

  Matched resource instance.

- MATCH_RESOURCE_METHOD

  Matched resource method.

- MATCH_RUNTIME_RESOURCE

  Matched runtime resource.

- MATCH_SUMMARY

  Matching summary.

- METHOD_INVOKE

  Resource method invoked.

- PRE_MATCH

  RESTEasy HttpRequestPreprocessor invoked.

- PRE_MATCH_SUMMARY

  RESTEasy HttpRequestPreprocessor invoked.

- REQUEST_FILTER

  ContainerRequestFilter invoked.

- REQUEST_FILTER_SUMMARY

  ContainerRequestFilter invocation summary.

- RESPONSE_FILTER

  ContainerResponseFilter invoked.

- RESPONSE_FILTER_SUMMARY

  ContainerResponseFilter invocation summary.

- START

  Request processing started.

- START_HEADERS

  All HTTP request headers.

54.8. Tracing Example

In the "resteasy-example" project, it contains a RESTEasy Tracing Example [https://github.com/resteasy/resteasy-examples/tree/master/tracing-example] to show the usages of tracing features. Please check the example to see the usages in action.

Chapter 55. Validation

RESTEasy provides the support for validation mandated by the Jakarta RESTful Web Services [https://jakarta.ee/specifications/restful-ws/], given the presence of an implementation of the Bean Validation specification [https://beanvalidation.org/2.0/spec/] such as Hibernate Validator [http://hibernate.org/validator/].

Validation provides a declarative way of imposing constraints on fields and properties of beans, bean classes, and the parameters and return values of bean methods. For example, in

```
@Path("all")
@TestClassConstraint(5)
public class TestResource
{
   @Size(min=2, max=4)
   @PathParam("s")
   String s;

   private String t;

   @Size(min=3)
   public String getT()
   {
      return t;
   }

   @PathParam("t")
   public void setT(String t)
   {
      this.t = t;
   }

   @POST
   @Path("{s}/{t}/{u}")
   @Pattern(regexp="[a-c]+")
   public String post(@PathParam("u") String u)
   {
      return u;
   }
}
```

the field `s` is constrained by the Bean Validation built-in annotation `@Size` to have between 2 and 4 characters, the property `t` is constrained to have at least 3 characters, and the `TestResource` object is constrained by the application defined annotation `@TestClassConstraint` to have the combined lengths of `s` and `t` less than 5:

```
@Constraint(validatedBy = TestClassValidator.class)
@Target({TYPE})
@Retention(RUNTIME)
```

```
public @interface TestClassConstraint
{
   String message() default "Concatenation of s and t must have length > {value}";
   Class<?>[] groups() default {};
   Class<? extends Payload>[] payload() default {};
   int value();
}

public    class    TestClassValidator    implements    ConstraintValidator<TestClassConstraint,
 TestResource>
{
   int length;

   public void initialize(TestClassConstraint constraintAnnotation)
   {
      length = constraintAnnotation.value();
   }

   public boolean isValid(TestResource value, ConstraintValidatorContext context)
   {
      boolean b = value.retrieveS().length() + value.getT().length() < length;
   }
}
```

See the links above for more about how to create validation annotations.

Also, the method parameter `u` is constrained to have no more than 5 characters, and the return value of method `post` is constrained by the built-in annotation `@Pattern` to match the regular expression "[a-c]+".

The sequence of validation constraint testing is as follows:

1. Create the resource and validate property, and class constraints.
2. Validate the resource method parameters.
3. If no violations have been detected, call the resource method and validate the return value

Note. Though fields and properties are technically different, they are subject to the same kinds of constraints, so they are treated the same in the context of validation. Together, they will both be referred to as "properties" herein.

55.1. Violation reporting

If a validation problem occurs, either a problem with the validation definitions or a constraint violation, RESTEasy will set the return header `org.jboss.resteasy.api.validation.Validation.VALIDATION_HEADER` ("validation-exception") to "true".

If RESTEasy detects a structural validation problem, such as a validation annotation with a missing validator class, it will return a String representation of a `javax.validation.ValidationException`. For example

```
javax.validation.ValidationException: HV000028: Unexpected exception during isValid call.
[org.jboss.resteasy.test.validation.TestValidationExceptions$OtherValidationException]
```

If any constraint violations are detected, RESTEasy will return a report in one of a variety of formats. If one of "application/xml" or "application/json" occur in the "Accept" request header, RESTEasy will return an appropriately marshalled instance of `org.jboss.resteasy.api.validation.ViolationReport`:

```
@XmlRootElement(name="violationReport")
@XmlAccessorType(XmlAccessType.FIELD)
public class ViolationReport
{
   ...

   public ArrayList<ResteasyConstraintViolation> getPropertyViolations()
   {
      return propertyViolations;
   }

   public ArrayList<ResteasyConstraintViolation> getClassViolations()
   {
      return classViolations;
   }

   public ArrayList<ResteasyConstraintViolation> getParameterViolations()
   {
      return parameterViolations;
   }

   public ArrayList<ResteasyConstraintViolation> getReturnValueViolations()
   {
      return returnValueViolations;
   }

   ...
}
```

where `org.jboss.resteasy.api.validation.ResteasyConstraintViolation` is defined:

```
@XmlRootElement(name="resteasyConstraintViolation")
@XmlAccessorType(XmlAccessType.FIELD)
public class ResteasyConstraintViolation implements Serializable
{
   ...

   /**
    * @return type of constraint
    */
   public ConstraintType.Type getConstraintType()
```

```
   {
      return constraintType;
   }

   /**
    * @return description of element violating constraint
    */
   public String getPath()
   {
      return path;
   }

   /**
    * @return description of constraint violation
    */
   public String getMessage()
   {
      return message;
   }

   /**
    * @return object in violation of constraint
    */
   public String getValue()
   {
      return value;
   }

   /**
    * @return String representation of violation
    */
   public String toString()
   {
      return "[" + type() + "]\r[" + path + "]\r[" + message + "]\r[" + value + "]\r";
   }

   /**
    * @return String form of violation type
    */
   public String type()
   {
      return constraintType.toString();
   }
}
```

and `org.jboss.resteasy.api.validation.ConstraintType` is the enumeration

```
public class ConstraintType
{
   public enum Type {CLASS, PROPERTY, PARAMETER, RETURN_VALUE};
}
```

If both "application/xml" or "application/json" occur in the "Accept" request header, the media type is chosen according to the ranking given by implicit or explicit "q" parameter values. In the case of a tie, the returned media type is indeterminate.

If neither "application/xml" or "application/json" occur in the "Accept" request header, RESTEasy returns a report with a String representation of each `ResteasyConstraintViolation`, where each field is delimited by '[' and ']', followed by a '\r', with a final '\r' at the end. For example,

```
[PROPERTY]
[s]
[size must be between 2 and 4]
[a]

[PROPERTY]
[t]
[size must be between 3 and 5]
[z]

[CLASS]
[]
[Concatenation of s and t must have length > 5]
[org.jboss.resteasy.validation.TestResource@68467a6f]

[PARAMETER]
[test.<cross-parameter>]
[Parameters must total <= 7]
[[5, 7]]

[RETURN_VALUE]
[g.<return value>]
[size must be between 2 and 4]
[abcde]
```

where the four fields are

1. type of constraint
2. path to violating element (e.g., property name, class name, method name and parameter name)
3. message
4. violating element

The `ViolationReport` can be reconsititued from the `String` as follows:

```
Client client = ClientBuilder.newClient();
Invocation.Builder request = client.target(...).request();
Response response = request.get();
if (Boolean.valueOf(response.getHeaders().getFirst(Validation.VALIDATION_HEADER)))
```

```
{
   String s = response.readEntity(String.class);
   ViolationReport report = new ViolationReport(s);
}
```

If the path field is considered to be too much server side information, it can be surpressed by setting the parameter "resteasy.validation.suppress.path" to "true". In that case, "*" will be returned in the path fields. [See Section 3.4, “Configuration” for more information about application configuration.]

55.2. Validation Service Providers

The form of validation mandated by the Jakarta RESTful Web Services specification, based on Bean Validation 1.1 or greater, is supported by the RESTEasy module resteasy-validator-provider, which produces the artifact resteasy-validator-provider-<version>.jar. Validation is turned on by default (assuming resteasy-validator-provider-<version>.jar is available), though parameter and return value validation can be turned off or modified in the validation.xml configuration file. See the Hibernate Validator [https://docs.jboss.org/hibernate/stable/validator/reference/en-US/html_single/?v=6.0] documentation for the details.

RESTEasy obtains a bean validation implementation by looking in the available META-INF/services/javax.ws.rs.Providers files for an implementation of `ContextResolver<GeneralValidator>`, where `org.jboss.resteasy.spi.GeneralValidator` is

```
public interface GeneralValidator
{
   /**
    * Validates all constraints on {@code object}.
    *
    * @param object object to validate
    * @param groups the group or list of groups targeted for validation (defaults to
    *        {@link Default})
    * @return constraint violations or an empty set if none
    * @throws IllegalArgumentException if object is {@code null}
    *         or if {@code null} is passed to the varargs groups
    * @throws ValidationException if a non recoverable error happens
    *         during the validation process
    */
   public abstract void validate(HttpRequest request, Object object, Class<?>... groups);
   /**
    * Validates all constraints placed on the parameters of the given method.
    *
    * @param <T> the type hosting the method to validate
    * @param object the object on which the method to validate is invoked
    * @param method the method for which the parameter constraints is validated
    * @param parameterValues the values provided by the caller for the given method's
    *        parameters
    * @param groups the group or list of groups targeted for validation (defaults to
    *        {@link Default})
    * @return a set with the constraint violations caused by this validation;
    *         will be empty if no error occurs, but never {@code null}
```

```
    * @throws IllegalArgumentException if {@code null} is passed for any of the parameters
    *         or if parameters don't match with each other
    * @throws ValidationException if a non recoverable error happens during the
    *         validation process
    */
  public abstract void validateAllParameters(HttpRequest request, Object object, Method method,
Object[] parameterValues, Class<?>... groups);

   /**
    * Validates all return value constraints of the given method.
    *
    * @param <T> the type hosting the method to validate
    * @param object the object on which the method to validate is invoked
    * @param method the method for which the return value constraints is validated
    * @param returnValue the value returned by the given method
    * @param groups the group or list of groups targeted for validation (defaults to
    *        {@link Default})
    * @return a set with the constraint violations caused by this validation;
    *         will be empty if no error occurs, but never {@code null}
    * @throws IllegalArgumentException if {@code null} is passed for any of the object,
    *         method or groups parameters or if parameters don't match with each other
    * @throws ValidationException if a non recoverable error happens during the
    *         validation process
    */
   public abstract void validateReturnValue(
          HttpRequest request, Object object, Method method, Object returnValue, Class<?>...
groups);

   /**
    * Indicates if validation is turned on for a class.
    *
    * @param clazz Class to be examined
    * @return true if and only if validation is turned on for clazz
    */
   public abstract boolean isValidatable(Class<?> clazz);

   /**
    * Indicates if validation is turned on for a method.
    *
    * @param method method to be examined
    * @return true if and only if validation is turned on for method
    */
   public abstract boolean isMethodValidatable(Method method);

   void checkViolations(HttpRequest request);
}
```

The methods and the javadoc are adapted from the Bean Validation 1.1 classes `javax.validation.Validator` and `javax.validation.executable.ExecutableValidator`.

RESTEasy module resteasy-validator-provider supplies an implementation of `GeneralValidator`. An alternative implementation may be supplied by implementing `ContextResolver<GeneralValidator>` and `org.jboss.resteasy.spi.validation.GeneralValidator`.

A validator intended to function in the presence of CDI must also implement the subinterface

```
public interface GeneralValidatorCDI extends GeneralValidator
{
   /**
    * Indicates if validation is turned on for a class.
    *
    * This method should be called from the resteasy-core module. It should
    * test if injectorFactor is an instance of CdiInjectorFactory, which indicates
    * that CDI is active.  If so, it should return false. Otherwise, it should
    * return the same value returned by GeneralValidator.isValidatable().
    *
    * @param clazz Class to be examined
    * @param injectorFactory the InjectorFactory used for clazz
    * @return true if and only if validation is turned on for clazz
    */
   public boolean isValidatable(Class<?> clazz, InjectorFactory injectorFactory);

   /**
    * Indicates if validation is turned on for a class.
    * This method should be called only from the resteasy-cdi module.
    *
    * @param clazz Class to be examined
    * @return true if and only if validation is turned on for clazz
    */
   public abstract boolean isValidatableFromCDI(Class<?> clazz);

   /**
    * Throws a ResteasyViolationException if any validation violations have been detected.
    * The method should be called only from the resteasy-cdi module.
    * @param request
    */
   public void checkViolationsfromCDI(HttpRequest request);

   /**
    * Throws a ResteasyViolationException if either a ConstraintViolationException or a
    * ResteasyConstraintViolationException is embedded in the cause hierarchy of e.
    *
    * @param request
    * @param e
    */
   public void checkForConstraintViolations(HttpRequest request, Exception e);
}
```

The validator in resteasy-validator-provider implements GeneralValidatorCDI.

55.3. Validation Implementations

As mentioned above, RESTEasy validation requires an implementation of the Bean Validation specification [https://beanvalidation.org/2.0/spec/] such as Hibernate Validator [http://hibernate.org/validator/]. Hibernate Validator is supplied automatically when RESTEasy is running in the context of WildFly. Otherwise, it should be made available. For example, in maven

```
        <dependency>
            <groupId>org.hibernate.validator</groupId>
            <artifactId>hibernate-validator/artifactId>
        </dependency>
```

Chapter 56. Internationalization and Localization

With the help of the JBoss Logging project, all log and exception messages in RESTEasy are internationalized. That is, they have a default value in English which can be overridden in any given locale by a file which gives translated values. For more information about internationalization and localization in Java, see, for example, http://docs.oracle.com/javase/tutorial/i18n. For more about JBoss Logging Tooling, see https://jboss-logging.github.io/jboss-logging-tools/ [https://jboss-logging.github.io/jboss-logging-tools/], Chapters 4 and 5.

56.1. Internationalization

Each module in RESTEasy that produces any text in the form of logging messages or exception messages has an interface named org.jboss.resteasy...i18n.Messages which contains the default messages. Those modules which do any logging also have an interface named org.jboss.resteasy...i18n.LogMessages which gives access to an underlying logger. With the exception of the resteasy-core-spi module, all messages are in the Messages class. resteasy-core-spi has exception messages in the Messages class and log messages in the LogMessages class.

Each message is prefixed by the project code "RESTEASY" followed by an ID which is unique to RESTEasy. These IDs belong to the following ranges:

Table 56.1.

Range	Module
2000-2999	resteasy-core-spi log messages
3000-4499	resteasy-core-spi exception messages
4500-4999	resteasy-client
5000-5499	providers/resteasy-atom
5500-5999	providers/fastinfoset
6000-6499	providers/resteasy-html
6500-6999	providers/jaxb
7500-7999	providers/multipart
8000-8499	providers/resteasy-hibernatevalidator-provider
8500-8999	providers/resteasy-validator-provider
9500-9999	async-http-servlet-3.0
10000-10499	cache-core
10500-10999	resteasy-cdi
11500-11999	resteasy-jsapi

Range	Module
12000-12499	resteasy-links
12500-12999	resteasy-servlet-initializer
13000-13499	resteasy-spring
13500-13999	security/resteasy-crypto
14000-14499	security/jose-jwt
14500-14999	security/keystone/keystone-as7
15000-15499	security/keystone/keystone-core
15500-15999	security/resteasy-oauth
16000-16499	security/skeleton-key-idm/skeleton-key-as7
16500-16999	security/skeleton-key-idm/skeleton-key-core
17000-17499	security/skeleton-key-idm/skeleton-key-idp
17500-17999	server-adapters/resteasy-jdk-http
18500-18999	server-adapters/resteasy-netty4

For example, the Jakarta XML Binding provider contains the interface

`org.jboss.resteasy.plugins.providers.jaxb.i18.Messages`

which looks like

```
@MessageBundle(projectCode = "RESTEASY")
public interface Messages
{
   Messages MESSAGES = org.jboss.logging.Messages.getBundle(Messages.class);
   int BASE = 6500;

   @Message(id = BASE + 00, value = "Collection wrapping failed, expected root element name of
 {0} got {1}", format=Format.MESSAGE_FORMAT)
   String collectionWrappingFailedLocalPart(String element, String localPart);

    @Message(id = BASE + 05, value = "Collection wrapping failed, expect namespace of {0} got
 {1}", format=Format.MESSAGE_FORMAT)
   String collectionWrappingFailedNamespace(String namespace, String uri);
   ...
```

The value of a message is retrieved by referencing a method and passing the appropriate parameters. For example,

```
throw                                                                          new

 ele.getName().getLocalPart()));
```

56.2. Localization

When RESTEasy is built with the "i18n" profile, a template properties file containing the default messages is created in a subdirectory of target/generated-translation-files. In the Jakarta XML Binding provider, for example, the

goes in the

directory, and the first few lines are

```
# Id: 6500
# Message: Collection wrapping failed, expected root element name of {0} got {1}
# @param 1: element -
# @param 2: localPart -
collectionWrappingFailedLocalPart=Collection wrapping failed, expected root element name of {0}
 got {1}
# Id: 6505
# Message: Collection wrapping failed, expect namespace of {0} got {1}
# @param 1: namespace -
# @param 2: uri -
collectionWrappingFailedNamespace=Collection wrapping failed, expect namespace of {0} got {1}
```

To provide the translation of the messages for a particular locale, the file should be renamed, replacing "locale", "COUNTRY", and "VARIANT" as appropriate (possibly omitting the latter two), and copied to the src/main/resources directory. In the Jakarta XML Binding provider, it would go in

For testing purposes, each module containing a Messages interface has two sample properties files, for the locale "en" and the imaginary locale "xx", in the src/test/resources directory. They are copied to src/main/resources when the module is built and deleted when it is cleaned.

The Messages.i18n_xx.properties file in the Jakarta XML Binding provider, for example, looks like

```
# Id: 6500
# Message: Collection wrapping failed, expected root element name of {0} got {1}
# @param 1: element -
# @param 2: localPart -
collectionWrappingFailedLocalPart=Collection wrapping failed, expected root element name of {0}
 got {1}
# Id: 6505
# Message: Collection wrapping failed, expect namespace of {0} got {1}
# @param 1: namespace -
# @param 2: uri -
collectionWrappingFailedNamespace=aaa {0} bbb {1} ccc
...
```

Note that the value of collectionWrappingFailedNamespace is modified.

Chapter 57. Maven and RESTEasy

JBoss's Maven Repository is at: https://repository.jboss.org/nexus/content/groups/public/

RESTEasy is modularized into 20 plus components. Each component is accessible as a Maven artifact. As a convenience RESTEasy provides a BOM containing the complete set of components with the appropriate versions for the "stack".

It is recommended to declare the BOM in your POM file, that way you will always be sure to get the correct version of the artifacts. In addition, you will not need to declare the version of each RESTEasy artifact called out in the dependencies section.

Declare the BOM file in the dependencyManagement section of the POM file like this. Note that Maven version 2.0.9 or higher is required to process BOM files.

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.jboss.resteasy</groupId>
      <artifactId>resteasy-bom</artifactId>
      <version>${version.resteasy}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

Declare the specific RESTEasy artifacts you require in the dependencies section of the POM file like this.

```
 <dependencies>
   <dependency>
      <groupId>org.jboss.resteasy</groupId>
      <artifactId>resteasy-client</artifactId>
   </dependency>
    ....
</dependencies>
```

It is possible to reference a RESTEasy artifact version not in the current BOM by specifying a version in the dependency itself.

```
  <dependencies>
    <dependency>
       <groupId>org.jboss.resteasy</groupId>
       <artifactId>resteasy-client</artifactId>
       <version>${version.some.other}<version>
    </dependency>
     ....
</dependencies>
```

Chapter 58. Migration from older versions

58.1. Migration to RESTEasy 3.0 series

Many facilities from RESTEasy 2 appear in a different form in RESTEasy 3. For example, much of the client framework in RESTEasy 2 is formalized, in modified form, in JAX-RS 2.0. RESTEasy versions 3.0.x implement both the older deprecated form and the newer conformant form. The deprecated form is moved to legacy module in RESTEasy 3.1 and finally removed in RESTEasy 4. For more information on upgrading from various deprecated facilities in RESTEasy 2, see http://docs.jboss.org/resteasy/docs/resteasy-upgrade-guide-en-US.pdf

58.2. Migration to RESTEasy 3.1 series

RESTEasy 3.1.0.Final release comes with many changes compared to previous 3.0 point releases. User discernible changes in RESTEasy 3.1.0.Final include

- module reorganization
- package reorganization
- new features
- minor behavioral changes
- miscellaneous changes

In this chapter we focus on changes that might cause existing code to fail or behave in new ways. The audience for this discussion may be partitioned into three subsets, depending on the version of RESTEasy currently in use, the API currently in use, and the API to be used after an upgrade to RESTEasy 3.1. The following APIs are available:

1. **RESTEasy 2**: RESTEasy 2 implements the JAX-RS 1 specification, and adds a variety of additional facilities, such as a client API, a caching system, an interceptor framework, etc. All of these user facing classes and interfaces comprise the RESTEasy 2 API.
2. **RESTEasy 3**: RESTEasy 3 implements the JAX-RS 2 specification, and adds some additional facilities. Many of the non-spec facilities from the RESTEasy 2 API are formalized, in altered form, in JAX-RS 2, in which case the older facilites are deprecated. The non-deprecated user facing classes and interfaces in RESTEasy 3 comprise the RESTEasy 3 API.

These definitions are rather informal and imprecise, since the user facing classes / interfaces in Resteasy 3.0.19.Final, for example, are a proper superset of the user facing classes / interfaces in RESTEasy 3.0.1.Final. For this discussion, we identify the API with the version currently in use in a given project.

Now, there are three potential target audiences of users planning to upgrade to RESTEasy 3.1.0.Final:

1. Those currently using RESTEasy API 3 with some RESTEasy 3.0.x release
2. Those currently using RESTEasy API 2 with some RESTEasy 2.x or 3.0.x release and planning to upgrade to RESTEasy API 3
3. Those currently using RESTEasy API 2 with some RESTEasy 2.x or 3.0.x release and planning to continue to use RESTEasy API 2

Of these, users in Group 2 have the most work to do in upgrading from RESTEasy API 2 to RESTEasy API 3. They should consult the separate guide Upgrading from RESTEasy 2 to RESTEasy 3 [http://docs.jboss.org/resteasy/docs/resteasy-upgrade-guide-en-US.pdf].

Ideally, users in Groups 1 and 3 might make some changes to take advantage of new features but would have no changes forced on them by reorganization or altered behavior. Indeed, that is almost the case, but there are a few changes that they should be aware of.

1. Upgrading with RESTEasy 3 API

All RESTEasy changes are documented in JIRA issues. Issues that describe detectable changes in release 3.1.0.Final that might impact existing applications include

- RESTEASY-1341: Build method of org.jboss.resteasy.client.jaxrs.internal.ClientInvocationBuilder always return the same instance. [https://issues.jboss.org/browse/RESTEASY-1341]

  When a `build()` method from

  - `org.jboss.resteasy.client.jaxrs.internal.ClientInvocationBuilder` in resteasy-client,
  - `org.jboss.resteasy.specimpl.LinkBuilderImpl` in resteasy-core,
  - `org.jboss.resteasy.specimpl.ResteasyUriBuilder` in resteasy-jaxrs

  is called, it will return a new object. This behavior might be seen indirectly. For example,

```
Builder builder = client.target(generateURL(path)).request();
...
Link link = new LinkBuilderImpl().uri(href).build();
...
URI uri = uriInfo.getBaseUriBuilder().path("test").build();
```

- RESTEASY-1433: Compile with JDK 1.8 source/target version [https://issues.jboss.org/browse/RESTEASY-1433]

As it says. Depending on the application, it might be necessary to recompile with a target of JDK 1.8 so that calls to RESTEasy code can work.

- RESTEASY-1484: CVE-2016-6346: Abuse of GZIPInterceptor in can lead to denial of service attack [https://issues.jboss.org/browse/RESTEASY-1484]

  Prior to release 3.1.0.Final, the default behavior of RESTEasy was to use GZIP to compress and decompress messages whenever "gzip" appeared in the Content-Encoding header. However, decompressing messages can lead to security issues, so, as of release 3.1.0.Final, GZIP compression has to be enabled explicitly. For details, see Chapter GZIP Compression/Decompression.

  Note. Because of some package reorganization due to RESTEASY-1531 (see below), the GZIP interceptors, which used to be in package `org.jboss.resteasy.plugins.interceptors.encoding` are now in `org.jboss.resteasy.plugins.interceptors`.

- RESTEASY-1531: Restore removed RESTEasy internal classes into a deprecated/disabled module [https://issues.jboss.org/browse/RESTEASY-1531]

  This issue is related to refactoring deprecated elements of the RESTEasy 2 API into a separate module, and, ideally, would have no bearing at all on RESTEasy 3. However, a reorganization of packages has led to moving some non-deprecated API elements in the resteasy-core module:

  - `org.jboss.resteasy.client.ClientURI` is now `org.jboss.resteasy.annotations.ClientURI`
  - `org.jboss.resteasy.core.interception.JaxrsInterceptorRegistryListener` is now `org.jboss.resteasy.core.interception.jaxrs.JaxrsInterceptorRegistryListener`
  - `org.jboss.resteasy.spi.interception.DecoratorProcessor` is now `org.jboss.resteasy.spi.DecoratorProcessor`
  - All of the dynamic features and interceptors in the package `org.jboss.resteasy.plugins.interceptors.encoding` are now in `org.jboss.resteasy.plugins.interceptors`

2. Upgrading with RESTEasy 2 API

Most of the deprecated classes and interfaces from RESTEasy 2 have been segregated in a separate module, resteasy-legacy, as of release 3.1.0.Final. A few remain in module resteasy-jaxrs for technical reasons. Eventually, all such classes and interfaces will be removed from RESTEasy. Most of the relocated elements are internal, so ensuring that resteasy-legacy is on

the classpath will make most changes undetectable. One way to do that, of course, is to include it in an application's WAR. In the context of WildFly, it is also possible to use a jboss-deployment-structure.xml file in the WEB-INF directory of your WAR file. For example:

```
<jboss-deployment-structure>
    <deployment>
        <dependencies>
            <module name="org.jboss.resteasy.resteasy-legacy"/>
        </dependencies>
    </deployment>
</jboss-deployment-structure>
```

There are a few API classes and interfaces from resteasy-jaxrs that have moved to a new package in resteasy-legacy. These are

- `org.jboss.resteasy.annotations.ClientResponseType` is now `org.jboss.resteasy.annotations.legacy.ClientResponseType`
- `org.jboss.resteasy.spi.Link` is now `org.jboss.resteasy.client.Link`
- `org.jboss.resteasy.spi.LinkHeader` is now `org.jboss.resteasy.client.LinkHeader`

58.3. Migration to RESTEasy 3.5+ series

RESTEasy 3.5 series is a spin-off of the old RESTEasy 3.0 series, featuring Jakarta RESTful Web Services implementation.

The reason why 3.5 comes from 3.0 instead of the 3.1 / 4.0 development streams is basically providing users with a selection of RESTEasy 4 critical / strategic new features, while ensuring full backward compatibility. As a consequence, no major issues are expected when upgrading RESTEasy from 3.0.x to 3.5.x. The 3.6 and all other 3.x minors after that are backward compatible evolutions of 3.5 series.

The natural upgrade path for users already on RESTEasy 3.1 series is straight to RESTEasy 4 instead.

58.4. Migration to RESTEasy 4 series

User migrating from RESTEasy 3.0 and 3.5+ series should be aware of the changes mentioned in the Section 58.2, “Migration to RESTEasy 3.1 series”. In addition to that, the aspects from the following sections are to be considered.

1. Public / private API

The `resteasy-jaxrs` and `resteasy-client` modules in RESTEasy 3 contain most of the framework classes and there's no real demarcation between what is internal implementation detail and what is for public consumption. In WildFly, the artifact archives from those modules are also included in a public module. Given the common expectation of full backward compatibility of whatever comes from public modules, to allow for easier project evolution and maintenance, in RESTEasy 4.0.0.Final those big components have been split as follows:

1.1. resteasy-core-spi

The public classes of the former `resteasy-jaxrs` module; the following packages are included:

- `org.jboss.resteasy.annotations`
- `org.jboss.resteasy.api.validation`
- `org.jboss.resteasy.spi`
- `org.jboss.resteasy.plugins.providers.validation`

1.2. resteasy-core

The internal details of the former `resteasy-jaxrs` module, including classes from the following packages:

- `org.jboss.resteasy.core`
- `org.jboss.resteasy.mock`
- `org.jboss.resteasy.plugins`
- `org.jboss.resteasy.specimpl`
- `org.jboss.resteasy.tracing`
- `org.jboss.resteasy.util`

1.3. resteasy-client-api

The public classes from the former `resteasy-client` module, basically whatever is used for configuring the RESTEasy client additions:

- `ClientHttpEngine` and `ClientHttpEngineBuilder`
- `ProxyBuilder` and `ProxyConfig`
- `ResteasyClient`

- `ResteasyClientBuilder`
- `ResteasyWebTarget`

1.4. resteasy-client

The remainings of the former `resteasy-client` module, internal details.

As a consequence of the split, all modules except `resteasy-core-spi` and `resteasy-client-api` are effectively private / internal. User applications and integration code should not directly rely on classes from those modules, which can be changed without going through any formal deprecation process.

Unfortunately, the refactoring that led to this implied some unavoidable class moves and changes breaking backward compatibility. A detailed list of the potentially problematic changes is available on the refactoring PR [https://github.com/resteasy/Resteasy/pull/1697].

2. Deprecated classes and modules removal

All classes and modules that were deprecated in RESTEasy 3 have been dropped in 4. In particular, this includes the legacy modules (`resteasy-legacy`, `security-legacy`) that were introduced in 3.1.

In addition to the legacy modules, few other modules have been dropped for multiple different reasons, including dependency on unsupported / abandoned libraries, better options available, etc:

- `resteasy-jackson-provider`, users should rely on `resteasy-jackson2-provider` instead;
- `resteasy-jettison-provider`, users should rely on `resteasy-jackson2-provider` instead;
- `abdera-atom-provider`;
- `resteasy-yaml-provider`;
- `resteasy-rx-java`, users should rely on `resteasy-rx-java2` instead;
- `tjws`.

The `resteasy-validator-provider-11` is also gone, with the `resteasy-validator-provider` one now supporting Bean Validation 2.0.

3. Behavior changes

With the `ClientHttpEngine` based on Apache HTTP Client 4.0 having gone (it was previously deprecated) and the engine based on version 4.3 of the same library being the default, the user might want to double check the notes about connection close in Section 50.3.4, “Apache HTTP Client 4.3 APIs”.

The conversion of `String` objects to `MediaType` objects is quite common in RESTEasy; for performances reasons a cache has been added to store the results of that conversion; by default the cache keeps the result of 200 conversions, but the number can be configured by setting the `org.jboss.resteasy.max_mediatype_cache_size` system property.

4. Other changes

- In releases 3.x, when bean validation (Chapter 55, *Validation*) threw instances of exceptions
  - `javax.validation.ConstraintDefinitionException`,
  - `javax.validation.ConstraintDeclarationException`, or
  - `javax.validation.GroupDefinitionException`,

  they were wrapped in a `org.jboss.resteasy.api.validation.Resteasy.ResteasyViolationException`, which `org.jboss.resteasy.api.validation.ResteasyViolationExceptionMapper`, the built-in implementation of `javax.ws.rs.ext.ExceptionMapper<javax.validation.ValidationException>`, then turned into descriptive text. As of release 4.0.0, instances of `ConstraintDefinitionException`, etc., are thrown as is. They are still caught by `ResteasyViolationExceptionMapper`, so, in general, there is no detectable change. It should be noted, however, that an implementation of `ExceptionMapper<ResteasyViolationException>`, which, prior to release 4.0.0, would have caught wrapped instances of `ConstraintDefinitionException`, will not catch unwrapped instances.
- The `ResteasyProviderFactory` is now an abstract class and is meant to be created using its `getInstance()` and `newInstance()` methods. Moreover, on client side, the resolution of the current instance is cached for each thread local context classloader.
- The `ResteasyClient` and `ResteasyClientBuilder` are now abstract classes (from `resteasy-client-api`) and are not meant for user direct instantiation; plain Jakarta RESTful Web Services API usage is expected instead:

```
//ResteasyClient client = new ResteasyClientBuilder().build(); NO!
//if plain Jakarta RESTful Web Services is enough ...
Client client = ClientBuilder.newClient();
...
//if RESTEasy API is needed ...
ResteasyClient client = (ResteasyClient)ClientBuilder.newClient();


//ResteasyClientBuilder builder = new ResteasyClientBuilder(); NO!
//if plain Jakarta RESTful Web Services is enough ...
ClientBuilder builder = ClientBuilder.newBuilder();
...
//if RESTEasy API is needed ...
ResteayClientBuilder builder = (ResteasyClientBuilder)ClientBuilder.newBuilder();
```

- The package `org.jboss.resteasy.plugins.stats` (which contains a resource and some related classes) has been moved out of the `resteasy-jaxb-provider` module into a `new resteasy-stats` module.

Chapter 59. Books You Can Read

There are a number of great books that you can learn REST and Jakarta RESTful Web Services from

- RESTful Web Services [http://oreilly.com/catalog/9780596529260/] by Leonard Richardson and Sam Ruby. A great introduction to REST.
- RESTful Java with JAX-RS [http://oreilly.com/catalog/9780596158040/] by Bill Burke. Overview of REST and detailed explanation of JAX-RS. Book examples are distributed with RESTEasy.
- RESTful Web Services Cookbook [http://oreilly.com/catalog/9780596808679/] by Subbu Allamaraju and Mike Amundsen. Detailed cookbook on how to design RESTful services.