Chapter 3. JAX-RS Application, Resources and Sub-Resources

Table of Contents

This chapter presents an overview of the core JAX-RS concepts - resources and sub-resources.

The JAX-RS 2.0.1 JavaDoc can be found online here.

The JAX-RS 2.0.1 specification draft can be found online here.

3.1. Root Resource Classes

Root resource classes are POJOs (Plain Old Java Objects) that are annotated with @Path have at least one method annotated with @Path or a resource method designator annotation such as @GET, @PUT, @POST, @DELETE. Resource methods are methods of a resource class annotated with a resource method designator. This section shows how to use Jersey to annotate Java objects to create RESTful web services.

The following code example is a very simple example of a root resource class using JAX-RS annotations. The example code shown here is from one of the samples that ships with Jersey, the zip file of which can be found in the maven repository here.

Example 3.1. Simple hello world root resource class

```
1  package org.glassfish.jersey.examples.helloworld;
2
3  import javax.ws.rs.GET;
4  import javax.ws.rs.Path;
5  import javax.ws.rs.Produces;
6
7  @Path("helloworld")
8  public class HelloWorldResource {
9      public static final String CLICHED_MESSAGE = "Hello World!";
10
11 @GET
12 @Produces("text/plain")
13     public String getHello() {
14         return CLICHED_MESSAGE;
15     }
16 }
```

Let's look at some of the JAX-RS annotations used in this example.

3.1.1. @Path

The @Path annotation's value is a relative URI path. In the example above, the Java class will be hosted at the URI path `/helloworld`. This is an extremely simple use of the @Path annotation. What makes JAX-RS so useful is that you can embed variables in the URIs.

URI path templates are URIs with variables embedded within the URI syntax. These variables are substituted at runtime in order for a resource to respond to a request based on the substituted URI. Variables are denoted by curly braces. For example, look at the following @Path annotation:

```
@Path("/users/{username}")
```

In this type of example, a user will be prompted to enter their name, and then a Jersey web service configured to respond to requests to this URI path template will respond. For example, if the user entered their username as "Galileo", the web service will respond to the following URL:
`http://example.com/users/Galileo`

To obtain the value of the username variable the @PathParam may be used on method parameter of a request method, for example:

Example 3.2. Specifying URI path parameter

```
@Path("/users/{username}")
public class UserResource {

    @GET
    @Produces("text/xml")
    public String getUser(@PathParam("username") String userName) {
        ...
    }
}
```

If it is required that a user name must only consist of lower and upper case numeric characters then it is possible to declare a particular regular expression, which overrides the default regular expression, "[^/]+", for example:

```
@Path("users/{username: [a-zA-Z][a-zA-Z_0-9]*}")
```

In this type of example the username variable will only match user names that begin with one upper or lower case letter and zero or more alpha numeric characters and the underscore character. If a user name does not match that a 404 (Not Found) response will occur.

A @Path value may or may not begin with a '/', it makes no difference. Likewise, by default, a @Path value may or may not end in a '/', it makes no difference, and thus request URLs that end or do not end in a '/' will both be matched.

3.1.2. @GET, @PUT, @POST, @DELETE, ... (HTTP Methods)

@GET, @PUT, @POST, @DELETE and @HEAD are *resource method designator* annotations defined by JAX-RS and which correspond to the similarly named HTTP methods. In the example above, the annotated Java method will process HTTP GET requests. The behavior of a resource is determined by which of the HTTP methods the resource is responding to.

The following example is an extract from the storage service sample that shows the use of the PUT method to create or update a storage container:

Example 3.3. PUT method

```
@PUT
public Response putContainer() {
    System.out.println("PUT CONTAINER " + container);

    URI uri = uriInfo.getAbsolutePath();
    Container c = new Container(container, uri.toString());

    Response r;
    if (!MemoryStore.MS.hasContainer(c)) {
        r = Response.created(uri).build();
    } else {
        r = Response.noContent().build();
    }

    MemoryStore.MS.createContainer(c);
    return r;
}
```

By default the JAX-RS runtime will automatically support the methods HEAD and OPTIONS, if not explicitly implemented. For HEAD the runtime will invoke the implemented GET method (if present) and ignore the response entity (if set). A response returned for the OPTIONS method depends on the requested media type defined in the 'Accept' header. The OPTIONS method can return a response with a set of supported resource methods in the 'Allow' header or return a WADL document. See wadl section for more information.

3.1.3. @Produces

The @Produces annotation is used to specify the MIME media types of representations a resource can produce and send back to the client. In this example, the Java method will produce representations identified by the MIME media type "text/plain". @Produces can be applied at both the class and method levels. Here's an example:

Example 3.4. Specifying output MIME type

```
@Path("/myResource")
@Produces("text/plain")
public class SomeResource {
    @GET
    public String doGetAsPlainText() {
        ...
    }

    @GET
    @Produces("text/html")
    public String doGetAsHtml() {
        ...
    }
}
```

The `doGetAsPlainText` method defaults to the MIME type of the @Produces annotation at the class level. The `doGetAsHtml` method's @Produces annotation overrides the class-level @Produces setting, and specifies that the method can produce HTML rather than plain text.

If a resource class is capable of producing more that one MIME media type then the resource method chosen will correspond to the most acceptable media type as declared by the client. More specifically the Accept header of the HTTP request declares what is most acceptable. For example if the Accept header is "`Accept: text/plain`" then the `doGetAsPlainText` method will be invoked. Alternatively if the Accept header is "`Accept: text/plain;q=0.9, text/html`", which declares that the client can accept media types of "text/plain" and "text/html" but prefers the latter, then the `doGetAsHtml` method will be invoked.

More than one media type may be declared in the same @Produces declaration, for example:

Example 3.5. Using multiple output MIME types

```
@GET
@Produces({"application/xml", "application/json"})
public String doGetAsXmlOrJson() {
    ...
}
```

The `doGetAsXmlOrJson` method will get invoked if either of the media types "application/xml" and "application/json" are acceptable. If both are equally acceptable then the former will be chosen because it occurs first.

Optionally, server can also specify the quality factor for individual media types. These are considered if several are equally acceptable by the client. For example:

Example 3.6. Server-side content negotiation

```
@GET
@Produces({"application/xml; qs=0.9", "application/json"})
public String doGetAsXmlOrJson() {
    ...
}
```

In the above sample, if client accepts both "application/xml" and "application/json" (equally), then a server always sends "application/json", since "application/xml" has a lower quality factor.

The examples above refers explicitly to MIME media types for clarity. It is possible to refer to constant values, which may reduce typographical errors, see the constant field values of MediaType.

3.1.4. @Consumes

The @Consumes annotation is used to specify the MIME media types of representations that can be consumed by a resource. The above example can be modified to set the cliched message as follows:

Example 3.7. Specifying input MIME type

```
@POST
@Consumes("text/plain")
public void postClichedMessage(String message) {
    // Store the message
}
```

In this example, the Java method will consume representations identified by the MIME media type "text/plain". Notice that the resource method returns void. This means no representation is returned and response with a status code of 204 (No Content) will be returned to the client.

@Consumes can be applied at both the class and the method levels and more than one media type may be declared in the same @Consumes declaration.

3.2. Parameter Annotations (@*Param)

Parameters of a resource method may be annotated with parameter-based annotations to extract information from a request. One of the previous examples presented the use of @PathParam to extract a path parameter from the path component of the request URL that matched the path declared in @Path.

@QueryParam is used to extract query parameters from the Query component of the request URL. The following example is an extract from the sparklines sample:

Example 3.8. Query parameters

```
@Path("smooth")
@GET
public Response smooth(
    @DefaultValue("2") @QueryParam("step") int step,
    @DefaultValue("true") @QueryParam("min-m") boolean hasMin,
    @DefaultValue("true") @QueryParam("max-m") boolean hasMax,
    @DefaultValue("true") @QueryParam("last-m") boolean hasLast,
    @DefaultValue("blue") @QueryParam("min-color") ColorParam minColor,
    @DefaultValue("green") @QueryParam("max-color") ColorParam maxColor,
    @DefaultValue("red") @QueryParam("last-color") ColorParam lastColor) {
    ...
}
```

If a query parameter "step" exists in the query component of the request URI then the "step" value will be extracted and parsed as a 32 bit signed integer and assigned to the step method parameter. If "step" does not exist then a default value of 2, as declared in the @DefaultValue annotation, will be assigned to the step method parameter. If the "step" value cannot be parsed as a 32 bit signed integer then a HTTP 404 (Not Found) response is returned. User defined Java types such as `ColorParam` may be used, which as implemented as follows:

Example 3.9. Custom Java type for consuming request parameters

```
public class ColorParam extends Color {

    public ColorParam(String s) {
        super(getRGB(s));
    }

    private static int getRGB(String s) {
        if (s.charAt(0) == '#') {
            try {
                Color c = Color.decode("0x" + s.substring(1));
                return c.getRGB();
            } catch (NumberFormatException e) {
                throw new WebApplicationException(400);
            }
        } else {
            try {
                Field f = Color.class.getField(s);
                return ((Color)f.get(null)).getRGB();
            } catch (Exception e) {
                throw new WebApplicationException(400);
            }
        }
    }
}
```

In general the Java type of the method parameter may:

1. Be a primitive type;
2. Have a constructor that accepts a single `String` argument;
3. Have a static method named `valueOf` or `fromString` that accepts a single `String` argument (see, for example, `Integer.valueOf(String)` and `java.util.UUID.fromString(String)`);
4. Have a registered implementation of `javax.ws.rs.ext.ParamConverterProvider` JAX-RS extension SPI that returns a `javax.ws.rs.ext.ParamConverter` instance capable of a "from string" conversion for the type. or
5. Be `List<T>`, `Set<T>` or `SortedSet<T>`, where `T` satisfies 2 or 3 above. The resulting collection is read-only.

Sometimes parameters may contain more than one value for the same name. If this is the case then types in 5) may be used to obtain all values.

If the @DefaultValue is not used in conjunction with @QueryParam and the query parameter is not present in the request then value will be an empty collection for`List`, `Set` or `SortedSet`, `null` for other object types, and the Java-defined default for primitive types.

The @PathParam and the other parameter-based annotations, @MatrixParam, @HeaderParam, @CookieParam, @FormParam obey the same rules as @QueryParam. @MatrixParam extracts information from URL path segments. @HeaderParam extracts information from the HTTP headers. @CookieParam extracts information from the cookies declared in cookie related HTTP headers.

@FormParam is slightly special because it extracts information from a request representation that is of the MIME media type `"application/x-www-form-urlencoded"` and conforms to the encoding specified by HTML forms, as described here. This parameter is very useful for extracting information that is POSTed by HTML forms, for example the following extracts the form parameter named "name" from the POSTed form data:

Example 3.10. Processing POSTed HTML form

```
@POST
@Consumes("application/x-www-form-urlencoded")
public void post(@FormParam("name") String name) {
    // Store the message
}
```

If it is necessary to obtain a general map of parameter name to values then, for query and path parameters it is possible to do the following:

Example 3.11. Obtaining general map of URI path and/or query parameters

```
@GET
public String get(@Context UriInfo ui) {
    MultivaluedMap<String, String> queryParams = ui.getQueryParameters();
    MultivaluedMap<String, String> pathParams = ui.getPathParameters();
}
```

For header and cookie parameters the following:

Example 3.12. Obtaining general map of header parameters

```
@GET
public String get(@Context HttpHeaders hh) {
    MultivaluedMap<String, String> headerParams = hh.getRequestHeaders();
    Map<String, Cookie> pathParams = hh.getCookies();
}
```

In general @Context can be used to obtain contextual Java types related to the request or response.

Because form parameters (unlike others) are part of the message entity, it is possible to do the following:

Example 3.13. Obtaining general map of form parameters

```
@POST
@Consumes("application/x-www-form-urlencoded")
public void post(MultivaluedMap<String, String> formParams) {
    // Store the message
}
```

I.e. you don't need to use the @Context annotation.

Another kind of injection is the @BeanParam which allows to inject the parameters described above into a single bean. A bean annotated with @BeanParam containing any fields and appropriate `*param` annotation(like @PathParam) will be initialized with corresponding request values in expected way as if these fields were in the resource class. Then instead of injecting request values like path param into a constructor parameters or class fields the @BeanParam can be used to inject such a bean into a resource or resource method. The @BeanParam is used this way to aggregate more request parameters into a single bean.

Example 3.14. Example of the bean which will be used as @BeanParam

```
public class MyBeanParam {
    @PathParam("p")
    private String pathParam;

    @MatrixParam("m")
    @Encoded
    @DefaultValue("default")
    private String matrixParam;

    @HeaderParam("header")
    private String headerParam;

    private String queryParam;

    public MyBeanParam(@QueryParam("q") String queryParam) {
        this.queryParam = queryParam;
    }

    public String getPathParam() {
        return pathParam;
    }
    ...
}
```

Example 3.15. Injection of MyBeanParam as a method parameter:

```
@POST
public void post(@BeanParam MyBeanParam beanParam, String entity) {
    final String pathParam = beanParam.getPathParam(); // contains injected path parameter "p"
    ...
}
```

The example shows aggregation of injections @PathParam, @QueryParam @MatrixParam and @HeaderParam into one single bean. The rules for injections inside the bean are the same as described above for these injections. The @DefaultValue is used to define the default value for matrix parameter matrixParam. Also the @Encoded annotation has the same behaviour as if it were used for injection in the resource method directly. Injecting the bean parameter into @Singleton resource class fields is not allowed (injections into method parameter must be used instead).

@BeanParam can contain all parameters injections injections (@PathParam, @QueryParam, @MatrixParam, @HeaderParam, @CookieParam, @FormParam). More beans can be injected into one resource or method parameters even if they inject the same request values. For example the following is possible:

Example 3.16. Injection of more beans into one resource methods:

```
@POST
public void post(@BeanParam MyBeanParam beanParam, @BeanParam AnotherBean anotherBean, @PathParam("p") pathParam,
String entity) {
    // beanParam.getPathParam() == pathParam
    ...
}
```

3.3. Sub-resources

@Path may be used on classes and such classes are referred to as root resource classes. @Path may also be used on methods of root resource classes. This enables common functionality for a number of resources to be grouped together and potentially reused.

The first way @Path may be used is on resource methods and such methods are referred to as *sub-resource methods*. The following example shows the method signatures for a root resource class from the jmaki-backend sample:

Example 3.17. Sub-resource methods

```
@Singleton
@Path("/printers")
public class PrintersResource {

    @GET
    @Produces({"application/json", "application/xml"})
    public WebResourceList getMyResources() { ... }

    @GET @Path("/list")
    @Produces({"application/json", "application/xml"})
    public WebResourceList getListOfPrinters() { ... }

    @GET @Path("/jMakiTable")
    @Produces("application/json")
    public PrinterTableModel getTable() { ... }

    @GET @Path("/jMakiTree")
    @Produces("application/json")
    public TreeModel getTree() { ... }

    @GET @Path("/ids/{printerid}")
    @Produces({"application/json", "application/xml"})
    public Printer getPrinter(@PathParam("printerid") String printerId) { ... }

    @PUT @Path("/ids/{printerid}")
    @Consumes({"application/json", "application/xml"})
    public void putPrinter(@PathParam("printerid") String printerId, Printer printer) { ... }

    @DELETE @Path("/ids/{printerid}")
    public void deletePrinter(@PathParam("printerid") String printerId) { ... }
}
```

If the path of the request URL is "printers" then the resource methods not annotated with @Path will be selected. If the request path of the request URL is "printers/list" then first the root resource class will be matched and then the sub-resource methods that match "list" will be selected, which in this case is the sub-resource method`getListOfPrinters`. So, in this example hierarchical matching on the path of the request URL is performed.

The second way @Path may be used is on methods **not** annotated with resource method designators such as @GET or @POST. Such methods are referred to as *sub-resource locators*. The following example shows the method signatures for a root resource class and a resource class from the optimistic-concurrency sample:

Example 3.18. Sub-resource locators

```
@Path("/item")
public class ItemResource {
    @Context UriInfo uriInfo;

    @Path("content")
    public ItemContentResource getItemContentResource() {
        return new ItemContentResource();
    }

    @GET
    @Produces("application/xml")
        public Item get() { ... }
    }
}

public class ItemContentResource {

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

    @PUT
    @Path("{version}")
    public void put(@PathParam("version") int version,
                    @Context HttpHeaders headers,
                    byte[] in) {
        ...
    }
}
```

The root resource class `ItemResource` contains the sub-resource locator method `getItemContentResource` that returns a new resource class. If the path of the request URL is "item/content" then first of all the root resource will be matched, then the sub-resource locator will be matched and invoked, which returns an instance of the `ItemContentResource` resource class. Sub-resource locators enable reuse of resource classes. A method can be annotated with the @Path annotation with empty path (`@Path("/")` or `@Path("")`) which means that the sub resource locator is matched for the path of the enclosing resource (without sub-resource path).

Example 3.19. Sub-resource locators with empty path

```
@Path("/item")
public class ItemResource {

    @Path("/")
    public ItemContentResource getItemContentResource() {
        return new ItemContentResource();
    }
}
```

In the example above the sub-resource locator method `getItemContentResource` is matched for example for request path "/item/locator" or even for only "/item".

In addition the processing of resource classes returned by sub-resource locators is performed at runtime thus it is possible to support polymorphism. A sub-resource locator may return different sub-types depending on the request (for example a sub-resource locator could return different sub-types dependent on the role of the principle that is authenticated). So for example the following sub resource locator is valid:

Example 3.20. Sub-resource locators returning sub-type

```
@Path("/item")
public class ItemResource {

    @Path("/")
    public Object getItemContentResource() {
        return new AnyResource();
    }
}
```

Note that the runtime will not manage the life-cycle or perform any field injection onto instances returned from sub-resource locator methods. This is because the runtime does not know what the life-cycle of the instance is. If it is required that the runtime manages the sub-resources as standard resources the `Class` should be returned as shown in the following example:

Example 3.21. Sub-resource locators created from classes

```
import javax.inject.Singleton;

@Path("/item")
public class ItemResource {
    @Path("content")
    public Class<ItemContentSingletonResource> getItemContentResource() {
        return ItemContentSingletonResource.class;
    }
}

@Singleton
public class ItemContentSingletonResource {
    // this class is managed in the singleton life cycle
}
```

JAX-RS resources are managed in per-request scope by default which means that new resource is created for each request. In this example the `javax.inject.Singleton` annotation says that the resource will be managed as singleton and not in request scope. The sub-resource locator method returns a class which means that the runtime will managed the resource instance and its life-cycle. If the method would return instance instead, the `Singleton` annotation would have no effect and the returned instance would be used.

The sub resource locator can also return a *programmatic resource model*. See resource builder section for information of how the programmatic resource model is constructed. The following example shows very simple resource returned from the sub-resource locator method.

Example 3.22. Sub-resource locators returning resource model

```
import org.glassfish.jersey.server.model.Resource;

@Path("/item")
public class ItemResource {

    @Path("content")
    public Resource getItemContentResource() {
        return Resource.from(ItemContentSingletonResource.class);
    }
}
```

The code above has exactly the same effect as previous example. `Resource` is a resource simple resource constructed from `ItemContentSingletonResource`. More complex programmatic resource can be returned as long they are valid resources.

3.4. Life-cycle of Root Resource Classes

By default the life-cycle of root resource classes is per-request which, namely that a new instance of a root resource class is created every time the request URI path matches the root resource. This makes for a very natural programming model where constructors and fields can be utilized (as in the previous section showing the constructor of the `SparklinesResource` class) without concern for multiple concurrent requests to the same resource.

In general this is unlikely to be a cause of performance issues. Class construction and garbage collection of JVMs has vastly improved over the years and many objects will be created and discarded to serve and process the HTTP request and return the HTTP response.

Instances of singleton root resource classes can be declared by an instance of Application.

Jersey supports two further life-cycles using Jersey specific annotations.

Table 3.1. Resource scopes

Scope	Annotation	Annotation full class name	Description
Request scope	@RequestScoped (or none)	org.glassfish.jersey.process.internal.RequestScoped	*Default lifecycle* (applied when no annotation is present). In this scope the resource instance is created for each new request and used for processing of this request. If the resource is used more than one time in the request processing, always the same instance will be used. This can happen when a resource is a sub resource is returned more times during the matching. In this

Scope	Annotation	Annotation full class name	Description
Per-lookup scope	@PerLookup	org.glassfish.hk2.api.PerLookup	In this scope the resource instance is created every time it is needed for the processing even it handles the same request.
Singleton	@Singleton	javax.inject.Singleton	In this scope there is only one instance per jax-rs application. Singleton resource can be either annotated with @Singleton and its class can be registered using the instance of Application. You can also create singletons by registering singleton instances into Application.

3.5. Rules of Injection

Previous sections have presented examples of annotated types, mostly annotated method parameters but also annotated fields of a class, for the injection of values onto those types.

This section presents the rules of injection of values on annotated types. Injection can be performed on fields, constructor parameters, resource/sub-resource/sub-resource locator method parameters and bean setter methods. The following presents an example of all such injection cases:

Example 3.23. Injection

```
@Path("{id:\\d+}")
public class InjectedResource {
    // Injection onto field
    @DefaultValue("q") @QueryParam("p")
    private String p;

    // Injection onto constructor parameter
    public InjectedResource(@PathParam("id") int id) { ... }

    // Injection onto resource method parameter
    @GET
    public String get(@Context UriInfo ui) { ... }

    // Injection onto sub-resource resource method parameter
    @Path("sub-id")
    @GET
    public String get(@PathParam("sub-id") String id) { ... }

    // Injection onto sub-resource locator method parameter
    @Path("sub-id")
    public SubResource getSubResource(@PathParam("sub-id") String id) { ... }

    // Injection using bean setter method
    @HeaderParam("X-header")
    public void setHeader(String header) { ... }
}
```

There are some restrictions when injecting on to resource classes with a life-cycle of singleton scope. In such cases the class fields or constructor parameters cannot be injected with request specific parameters. So, for example the following is not allowed.

Example 3.24. Wrong injection into a singleton scope

```
@Path("resource")
@Singleton
public static class MySingletonResource {

    @QueryParam("query")
    String param; // WRONG: initialization of application will fail as you cannot
                  // inject request specific parameters into a singleton resource.

    @GET
    public String get() {
        return "query param: " + param;
    }
}
```

The example above will cause validation failure during application initialization as singleton resources cannot inject request specific parameters. The same example would fail if the query parameter would be injected into constructor parameter of such a singleton. In other words, if you wish one resource instance to server more requests (in the same time) it cannot be bound to a specific request parameter.

The exception exists for specific request objects which can injected even into constructor or class fields. For these objects the runtime will inject proxies which are able to simultaneously server more request. These request objects are `HttpHeaders`, `Request`, `UriInfo`, `SecurityContext`. These proxies can be injected using the @Context annotation. The following example shows injection of proxies into the singleton resource class.

Example 3.25. Injection of proxies into singleton

```java
@Path("resource")
@Singleton
public static class MySingletonResource {
    @Context
    Request request; // this is ok: the proxy of Request will be injected into this singleton

    public MySingletonResource(@Context SecurityContext securityContext) {
        // this is ok too: the proxy of SecurityContext will be injected
    }

    @GET
    public String get() {
        return "query param: " + param;
    }
}
```

To summarize the injection can be done into the following constructs:

Table 3.2. Overview of injection types

Java construct	Description
Class fields	Inject value directly into the field of the class. The field can be private and must not be final. Cannot be used in Singleton scope except proxiable types mentioned above.
Constructor parameters	The constructor will be invoked with injected values. If more constructors exists the one with the most injectable parameters will be invoked. Cannot be used in Singleton scope except proxiable types mentioned above.
Resource methods	The resource methods (these annotated with @GET, @POST, ...) can contain parameters that can be injected when the resource method is executed. Can be used in any scope.
Sub resource locators	The sub resource locators (methods annotated with @Path but not @GET, @POST, ...) can contain parameters that can be injected when the resource method is executed. Can be used in any scope.
Setter methods	Instead of injecting values directly into field the value can be injected into the setter method which will initialize the field. This injection can be used only with @Context annotation. This means it cannot be used for example for injecting of query params but it can be used for injections of request. The setters will be called after the object creation and only once. The name of the method does not necessary have a setter pattern. Cannot be used in Singleton scope except proxiable types mentioned above.

The following example shows all possible java constructs into which the values can be injected.

Example 3.26. Example of possible injections

```java
@Path("resource")
public static class SummaryOfInjectionsResource {
    @QueryParam("query")
    String param; // injection into a class field


    @GET
    public String get(@QueryParam("query") String methodQueryParam) {
        // injection into a resource method parameter
        return "query param: " + param;
    }

    @Path("sub-resource-locator")
    public Class<SubResource> subResourceLocator(@QueryParam("query") String subResourceQueryParam) {
        // injection into a sub resource locator parameter
        return SubResource.class;
    }

    public SummaryOfInjectionsResource(@QueryParam("query") String constructorQueryParam) {
        // injection into a constructor parameter
    }


    @Context
    public void setRequest(Request request) {
        // injection into a setter method
        System.out.println(request != null);
    }
}

public static class SubResource {
    @GET
    public String get() {
        return "sub resource";
    }
}
```

The @FormParam annotation is special and may only be utilized on resource and sub-resource methods. This is because it extracts information from a request entity.

3.6. Use of @Context

Previous sections have introduced the use of @Context. Chapter 5 of the JAX-RS specification presents all the standard JAX-RS Java types that may be used with @Context.

When deploying a JAX-RS application using servlet then ServletConfig, ServletContext, HttpServletRequest and HttpServletResponse are available

using @Context.

3.7. Programmatic resource model

Resources can be constructed from classes or instances but also can be constructed from a programmatic resource model. Every resource created from from resource classes can also be constructed using the programmatic resource builder api. See resource builder section for more information.