Creating a RESTful Root Resource Class

Root resource classes are POJOs that are either annotated with `@Path` or have at least one method annotated with `@Path` or a **request method designator**, such as `@GET`, `@PUT`, `@POST`, or `@DELETE`. **Resource methods** are methods of a resource class annotated with a request method designator. This section explains how to use JAX-RS to annotate Java classes to create RESTful web services.

Developing RESTful Web Services with JAX-RS

JAX-RS is a Java programming language API designed to make it easy to develop applications that use the REST architecture.

The JAX-RS API uses Java programming language annotations to simplify the development of RESTful web services. Developers decorate Java programming language class files with JAX-RS annotations to define resources and the actions that can be performed on those resources. JAX-RS annotations are runtime annotations; therefore, runtime reflection will generate the helper classes and artifacts for the resource. A Java EE application archive containing JAX-RS resource classes will have the resources configured, the helper classes and artifacts generated, and the resource exposed to clients by deploying the archive to a Java EE server.

Table 20-1 lists some of the Java programming annotations that are defined by JAX-RS, with a brief description of how each is used. Further information on the JAX-RS APIs can be viewed at http://docs.oracle.com/javaee/6/api/.

Table 20-1 Summary of JAX-RS Annotations

Annotation	Description
`@Path`	The `@Path` annotation's value is a relative URI path indicating where the Java class will be

	hosted: for example, `/helloworld`. You can also embed variables in the URIs to make a URI path template. For example, you could ask for the name of a user and pass it to the application as a variable in the URI: `/helloworld/{username}`.
`@GET`	The `@GET` annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP GET requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
`@POST`	The `@POST` annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP POST requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
`@PUT`	The `@PUT` annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP PUT requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
`@DELETE`	The `@DELETE` annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP DELETE requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
`@HEAD`	The `@HEAD` annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP HEAD requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
`@PathParam`	The `@PathParam` annotation is a type of parameter that you can extract for use in your resource class. URI path parameters are extracted from the request URI, and the parameter names correspond to the URI path template variable names specified in the `@Path` class-level annotation.
`@QueryParam`	The `@QueryParam` annotation is a type of parameter that you can extract for use in your resource class. Query parameters are extracted from the request URI query parameters.
`@Consumes`	The `@Consumes` annotation is used to specify the MIME media types of representations a resource can consume that were sent by the client.
`@Produces`	The `@Produces` annotation is used to specify the MIME media types of representations a resource can produce and send back to the client: for example, `"text/plain"`.

`@Provider`	The `@Provider` annotation is used for anything that is of interest to the JAX-RS runtime, such as `MessageBodyReader` and `MessageBodyWriter`. For HTTP requests, the `MessageBodyReader` is used to map an HTTP request entity body to method parameters. On the response side, a return value is mapped to an HTTP response entity body by using a `MessageBodyWriter`. If the application needs to supply additional metadata, such as HTTP headers or a different status code, a method can return a `Response` that wraps the entity and that can be built using `Response.ResponseBuilder`.

Overview of a JAX-RS Application

The following code sample is a very simple example of a root resource class that uses JAX-RS annotations:

```
package com.sun.jersey.samples.helloworld.resources;

import javax.ws.rs.GET;
import javax.ws.rs.Produces;
import javax.ws.rs.Path;

// The Java class will be hosted at the URI path "/helloworld"
@Path("/helloworld")
public class HelloWorldResource {

    // The Java method will process HTTP GET requests
    @GET
    // The Java method will produce content identified by the MIME Media
    // type "text/plain"
    @Produces("text/plain")
    public String getClichedMessage() {
        // Return some cliched textual content
        return "Hello World";
    }
}
```

The following sections describe the annotations used in this example.

- The `@Path` annotation's value is a relative URI path. In the preceding example, the Java class will be hosted at the URI path `/helloworld`. This is an extremely simple use of the `@Path` annotation, with a static URI path. Variables can be embedded in the URIs. **URI path templates** are URIs with variables embedded within the URI syntax.

- The `@GET` annotation is a request method designator, along with `@POST`, `@PUT`, `@DELETE`, and `@HEAD`, defined by JAX-RS and corresponding to the similarly named HTTP methods. In the example, the annotated Java method will process HTTP `GET` requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
- The `@Produces` annotation is used to specify the MIME media types 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"`.
- The `@Consumes` annotation is used to specify the MIME media types a resource can consume that were sent by the client. The example could be modified to set the message returned by the `getClichedMessage` method, as shown in this code example:

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

The `@Path` Annotation and URI Path Templates

The `@Path` annotation identifies the URI path template to which the resource responds and is specified at the class or method level of a resource. The `@Path` annotation's value is a partial URI path template relative to the base URI of the server on which the resource is deployed, the context root of the application, and the URL pattern to which the JAX-RS runtime responds.

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 braces (`{` and `}`). For example, look at the following `@Path` annotation:

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

In this kind of example, a user is prompted to type his or her name, and then a JAX-RS web service configured to respond to requests to this URI path template responds. For example, if the user types the user name "Galileo," the web service responds to the following URL:

```
http://example.com/users/Galileo
```

To obtain the value of the user name, the `@PathParam` annotation may be used on the method parameter of a request method, as shown in the following code example:

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

```
public class UserResource {

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

By default, the URI variable must match the regular expression `"[^/]+?"`. This variable may be customized by specifying a different regular expression after the variable name. For example, if a user name must consist only of lowercase and uppercase alphanumeric characters, override the default regular expression in the variable definition:

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

In this example the `username` variable will match only user names that begin with one uppercase or lowercase letter and zero or more alphanumeric characters and the underscore character. If a user name does not match that template, a 404 (Not Found) response will be sent to the client.

A `@Path` value isn't required to have leading or trailing slashes (/). The JAX-RS runtime parses URI path templates the same whether or not they have leading or trailing spaces.

A URI path template has one or more variables, with each variable name surrounded by braces: `{` to begin the variable name and `}` to end it. In the preceding example, `username` is the variable name. At runtime, a resource configured to respond to the preceding URI path template will attempt to process the URI data that corresponds to the location of `{username}` in the URI as the variable data for `username`.

For example, if you want to deploy a resource that responds to the URI path template `http://example.com/myContextRoot/resources/{name1}/{name2}/`, you must deploy the application to a Java EE server that responds to requests to the `http://example.com/myContextRoot` URI and then decorate your resource with the following `@Path` annotation:

```
@Path("/{name1}/{name2}/")
public class SomeResource {
    ...
}
```

In this example, the URL pattern for the JAX-RS helper servlet, specified in `web.xml`, is the default:

```
<servlet-mapping>
      <servlet-name>My JAX-RS Resource</servlet-name>
      <url-pattern>/resources/*</url-pattern>
```

```
</servlet-mapping>
```

A variable name can be used more than once in the URI path template.

If a character in the value of a variable would conflict with the reserved characters of a URI, the conflicting character should be substituted with percent encoding. For example, spaces in the value of a variable should be substituted with `%20`.

When defining URI path templates, be careful that the resulting URI after substitution is valid.

Table 20-2 lists some examples of URI path template variables and how the URIs are resolved after substitution. The following variable names and values are used in the examples:

- `name1`: `james`
- `name2`: `gatz`
- `name3`:
- `location`: `Main%20Street`
- `question`: `why`

Note - The value of the `name3` variable is an empty string.

Table 20-2 Examples of URI Path Templates

URI Path Template	URI After Substitution
`http://example.com/{name1}/{name2}/`	`http://example.com/james/gatz/`
`http://example.com/{question}/{question}/{question}/`	`http://example.com/why/why/why/`
`http://example.com/maps/{location}`	`http://example.com/maps/Main%20Street`
`http://example.com/{name3}/home/`	`http://example.com//home/`

Responding to HTTP Methods and Requests

The behavior of a resource is determined by the HTTP methods (typically, `GET`, `POST`, `PUT`, `DELETE`) to which the

resource is responding.

The Request Method Designator Annotations

Request method designator annotations are runtime annotations, defined by JAX-RS, that correspond to the similarly named HTTP methods. Within a resource class file, HTTP methods are mapped to Java programming language methods by using the request method designator annotations. The behavior of a resource is determined by which HTTP method the resource is responding to. JAX-RS defines a set of request method designators for the common HTTP methods `@GET`, `@POST`, `@PUT`, `@DELETE`, and `@HEAD`; you can also create your own custom request method designators. Creating custom request method designators is outside the scope of this document.

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

```
@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. For `OPTIONS`, the `Allow` response header will be set to the set of HTTP methods supported by the resource. In addition, the JAX-RS runtime will return a Web Application Definition Language (WADL) document describing the resource; see http://www.w3.org/Submission/wadl/ for more information.

Methods decorated with request method designators must return `void`, a Java programming language type, or a `javax.ws.rs.core.Response` object. Multiple parameters may be extracted from the URI by using the `@PathParam` or `@QueryParam` annotations as described in Extracting Request Parameters. Conversion between Java types and an

entity body is the responsibility of an entity provider, such as `MessageBodyReader` or `MessageBodyWriter`. Methods that need to provide additional metadata with a response should return an instance of the `Response` class. The `ResponseBuilder` class provides a convenient way to create a `Response` instance using a builder pattern. The HTTP `PUT` and `POST` methods expect an HTTP request body, so you should use a `MessageBodyReader` for methods that respond to `PUT` and `POST` requests.

Both `@PUT` and `@POST` can be used to create or update a resource. `POST` can mean anything, so when using `POST`, it is up to the application to define the semantics. `PUT` has well-defined semantics. When using `PUT` for creation, the client declares the URI for the newly created resource.

`PUT` has very clear semantics for creating and updating a resource. The representation the client sends must be the same representation that is received using a `GET`, given the same media type. `PUT` does not allow a resource to be partially updated, a common mistake when attempting to use the `PUT` method. A common application pattern is to use `POST` to create a resource and return a `201` response with a location header whose value is the URI to the newly created resource. In this pattern, the web service declares the URI for the newly created resource.

Using Entity Providers to Map HTTP Response and Request Entity Bodies

Entity providers supply mapping services between representations and their associated Java types. The two types of entity providers are `MessageBodyReader` and `MessageBodyWriter`. For HTTP requests, the `MessageBodyReader` is used to map an HTTP request entity body to method parameters. On the response side, a return value is mapped to an HTTP response entity body by using a `MessageBodyWriter`. If the application needs to supply additional metadata, such as HTTP headers or a different status code, a method can return a `Response` that wraps the entity and that can be built by using `Response.ResponseBuilder`.

Table 20-3 shows the standard types that are supported automatically for HTTP request and response entity bodies. You need to write an entity provider only if you are not choosing one of these standard types.

Table 20-3 Types Supported for HTTP Request and Response Entity Bodies

Java Type	Supported Media Types
`byte[]`	All media types (`*/*`)
`java.lang.String`	All text media types (`text/*`)
`java.io.InputStream`	All media types (`*/*`)
`java.io.Reader`	All media types (`*/*`)
`java.io.File`	All media types (`*/*`)

`javax.activation.DataSource`	All media types (`*/*`)
`javax.xml.transform.Source`	XML media types (`text/xml`, `application/xml`, and `application/*+xml`)
`javax.xml.bind.JAXBElement` and application-supplied JAXB classes	XML media types (`text/xml`, `application/xml`, and `application/*+xml`)
`MultivaluedMap<String, String>`	Form content (`application/x-www-form-urlencoded`)
`StreamingOutput`	All media types (`*/*`), `MessageBodyWriter` only

The following example shows how to use `MessageBodyReader` with the `@Consumes` and `@Provider` annotations:

```
@Consumes("application/x-www-form-urlencoded")
@Provider
public class FormReader implements MessageBodyReader<NameValuePair> {
```

The following example shows how to use `MessageBodyWriter` with the `@Produces` and `@Provider` annotations:

```
@Produces("text/html")
@Provider
public class FormWriter implements
        MessageBodyWriter<Hashtable<String, String>> {
```

The following example shows how to use `ResponseBuilder`:

```
@GET
public Response getItem() {
    System.out.println("GET ITEM " + container + " " + item);

    Item i = MemoryStore.MS.getItem(container, item);
    if (i == null)
        throw new NotFoundException("Item not found");
    Date lastModified = i.getLastModified().getTime();
    EntityTag et = new EntityTag(i.getDigest());
    ResponseBuilder rb = request.evaluatePreconditions(lastModified, et);
    if (rb != null)
        return rb.build();

    byte[] b = MemoryStore.MS.getItemData(container, item);
```

```
    return Response.ok(b, i.getMimeType()).
            lastModified(lastModified).tag(et).build();
}
```

Using `@Consumes` and `@Produces` to Customize Requests and Responses

The information sent to a resource and then passed back to the client is specified as a MIME media type in the headers of an HTTP request or response. You can specify which MIME media types of representations a resource can respond to or produce by using the following annotations:

- `javax.ws.rs.Consumes`
- `javax.ws.rs.Produces`

By default, a resource class can respond to and produce all MIME media types of representations specified in the HTTP request and response headers.

The `@Produces` Annotation

The `@Produces` annotation is used to specify the MIME media types or representations a resource can produce and send back to the client. If `@Produces` is applied at the class level, all the methods in a resource can produce the specified MIME types by default. If applied at the method level, the annotation overrides any `@Produces` annotations applied at the class level.

If no methods in a resource are able to produce the MIME type in a client request, the JAX-RS runtime sends back an HTTP “406 Not Acceptable” error.

The value of `@Produces` is an array of `String` of MIME types. For example:

```
@Produces({"image/jpeg,image/png"})
```

The following example shows how to apply `@Produces` at both the class and method levels:

```
@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 media 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 than one MIME media type, 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`, 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, the `doGetAsHtml` method will be invoked.

More than one media type may be declared in the same `@Produces` declaration. The following code example shows how this is done:

```
@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` is acceptable. If both are equally acceptable, the former will be chosen because it occurs first. The preceding examples refer explicitly to MIME media types for clarity. It is possible to refer to constant values, which may reduce typographical errors. For more information, see the constant field values of `MediaType` at http://jsr311.java.net/nonav/releases/1.0/javax/ws/rs/core/MediaType.html.

The `@Consumes` Annotation

The `@Consumes` annotation is used to specify which MIME media types of representations a resource can accept, or consume, from the client. If `@Consumes` is applied at the class level, all the response methods accept the specified MIME types by default. If applied at the method level, `@Consumes` overrides any `@Consumes` annotations applied at the class level.

If a resource is unable to consume the MIME type of a client request, the JAX-RS runtime sends back an HTTP 415 ("Unsupported Media Type") error.

The value of `@Consumes` is an array of `String` of acceptable MIME types. For example:

```
@Consumes({"text/plain,text/html"})
```

The following example shows how to apply `@Consumes` at both the class and method levels:

```
@Path("/myResource")
@Consumes("multipart/related")
public class SomeResource {
    @POST
    public String doPost(MimeMultipart mimeMultipartData) {
        ...
    }

    @POST
    @Consumes("application/x-www-form-urlencoded")
    public String doPost2(FormURLEncodedProperties formData) {
        ...
    }
}
```

The `doPost` method defaults to the MIME media type of the `@Consumes` annotation at the class level. The `doPost2` method overrides the class level `@Consumes` annotation to specify that it can accept URL-encoded form data.

If no resource methods can respond to the requested MIME type, an HTTP 415 (“Unsupported Media Type”) error is returned to the client.

The `HelloWorld` example discussed previously in this section can be modified to set the message by using `@Consumes`, as shown in the following code example:

```
@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`. Note that the resource method returns `void`. This means that no representation is returned and that a response with a status code of HTTP 204 (“No Content”) will be returned.

Extracting Request Parameters

Parameters of a resource method may be annotated with parameter-based annotations to extract information from a request. A previous example presented the use of the `@PathParam` parameter to extract a path parameter from the

path component of the request URL that matched the path declared in `@Path`.

You can extract the following types of parameters for use in your resource class:

- Query
- URI path
- Form
- Cookie
- Header
- Matrix

Query parameters are extracted from the request URI query parameters and are specified by using the `javax.ws.rs.QueryParam` annotation in the method parameter arguments. The following example, from the `sparklines` sample application, demonstrates using `@QueryParam` to extract query parameters from the `Query` component of the request URL:

```
@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 the query parameter `step` exists in the query component of the request URI, the value of `step` will be extracted and parsed as a 32-bit signed integer and assigned to the `step` method parameter. If `step` does not exist, 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, an HTTP 400 (“Client Error”) response is returned.

User-defined Java programming language types may be used as query parameters. The following code example shows the `ColorParam` class used in the preceding query parameter example:

```
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);
                }
            }
        }
    }
```

The constructor for `ColorParam` takes a single `String` parameter.

Both `@QueryParam` and `@PathParam` can be used only on the following Java types:

- All primitive types except `char`
- All wrapper classes of primitive types except `Character`
- Any class with a constructor that accepts a single `String` argument
- Any class with the static method named `valueOf(String)` that accepts a single `String` argument
- `List<T>`, `Set<T>`, or `SortedSet<T>`, where *T* matches the already listed criteria. Sometimes, parameters may contain more than one value for the same name. If this is the case, these types may be used to obtain all values

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

URI path parameters are extracted from the request URI, and the parameter names correspond to the URI path

template variable names specified in the `@Path` class-level annotation. URI parameters are specified using the `javax.ws.rs.PathParam` annotation in the method parameter arguments. The following example shows how to use `@Path` variables and the `@PathParam` annotation in a method:

```
@Path("/{username}")
public class MyResourceBean {
    ...
    @GET
    public String printUsername(@PathParam("username") String userId) {
        ...
    }
}
```

In the preceding snippet, the URI path template variable name `username` is specified as a parameter to the `printUsername` method. The `@PathParam` annotation is set to the variable name `username`. At runtime, before `printUsername` is called, the value of `username` is extracted from the URI and cast to a `String`. The resulting `String` is then available to the method as the `userId` variable.

If the URI path template variable cannot be cast to the specified type, the JAX-RS runtime returns an HTTP 400 (“Bad Request”) error to the client. If the `@PathParam` annotation cannot be cast to the specified type, the JAX-RS runtime returns an HTTP 404 (“Not Found”) error to the client.

The `@PathParam` parameter and the other parameter-based annotations (`@MatrixParam`, `@HeaderParam`, `@CookieParam`, and `@FormParam`) obey the same rules as `@QueryParam`.

Cookie parameters, indicated by decorating the parameter with `javax.ws.rs.CookieParam`, extract information from the cookies declared in cookie-related HTTP headers. **Header parameters**, indicated by decorating the parameter with `javax.ws.rs.HeaderParam`, extract information from the HTTP headers. **Matrix parameters**, indicated by decorating the parameter with `javax.ws.rs.MatrixParam`, extract information from URL path segments.

Form parameters, indicated by decorating the parameter with `javax.ws.rs.FormParam`, extract 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 in http://www.w3.org/TR/html401/interact/forms.html#h-17.13.4.1. This parameter is very useful for extracting information sent by `POST` in HTML forms.

The following example extracts the `name` form parameter from the `POST` form data:

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

```
    // Store the message
}
```

To obtain a general map of parameter names and values for query and path parameters, use the following code:

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

The following method extracts header and cookie parameter names and values into a map:

```
@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.

For form parameters, it is possible to do the following:

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