Implementing REST Services using JAX-RS API

9

ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Objectives

This chapter teaches how to handle communications with REST Services

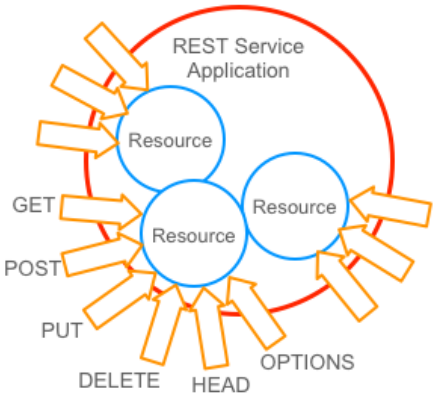
- Understand REST service conventions
- Create REST services using JAX-RS API
- Consume REST service within the client tier

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

REST Service Conventions and Resources

REST Service Application represents one or more Resources
Each Resource represents a business entity and is mapped to its own URL
Each Resource defines a number of operations mapped to HTTP Methods
Typical conventional use of HTTP Methods:

- GET receives (queries) a collection of elements or a specific element identified by client
- POST creates new element
- PUT updates existing element
- DELETE removes an element
- Other operations are sometimes used to retrieve metadata



❖ Unlike SOAP, there is no standard for REST Services - they are considered to be a coding style rather than an actual protocol.

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

REST is centered around an abstraction known as a "resource." Any named piece of information can be a resource.

A resource is identified by a uniform resource identifier (URI).

Clients request and submit representations of resources.

There can be many different representations available to represent the same resource.

A representation consists of data and metadata. – Metadata often takes the form of HTTP headers. Resources are interconnected by hyperlinks.

A RESTful web service is designed by identifying the resources. This is similar to Object Oriented Analysis and Design where you identify nouns in use-cases.

Unlike SOAP Services, there is no standard for REST Services. REST is considered to be a style of coding rather than an actual protocol.

REST Communication Model

REST Resources and REST Clients characteristics:

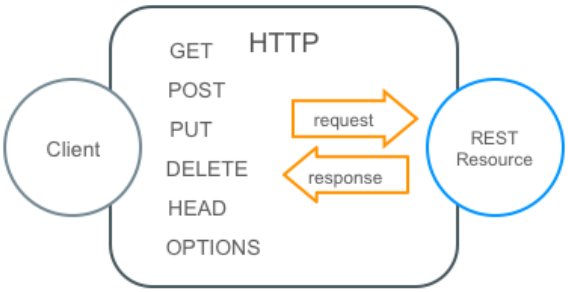
- REST Clients are often implemented by using JavaScript in Browser or Mobile Applications.
- Use HTTP as a transport layer and dispatch requests by using the GET, POST, PUT, DELETE, HEAD, and OPTIONS methods containing data that the client wants to pass to the REST Service.
- Clients handle responses that contain HTTP status codes and may also contain a body with server-generated content.
- Use of HTTP methods is conventional – REST services are not required to handle these in the same way.
- REST Resources may use different media types such as:
  - JSON
  - Plain Text
  - XML
  - And so on

XML

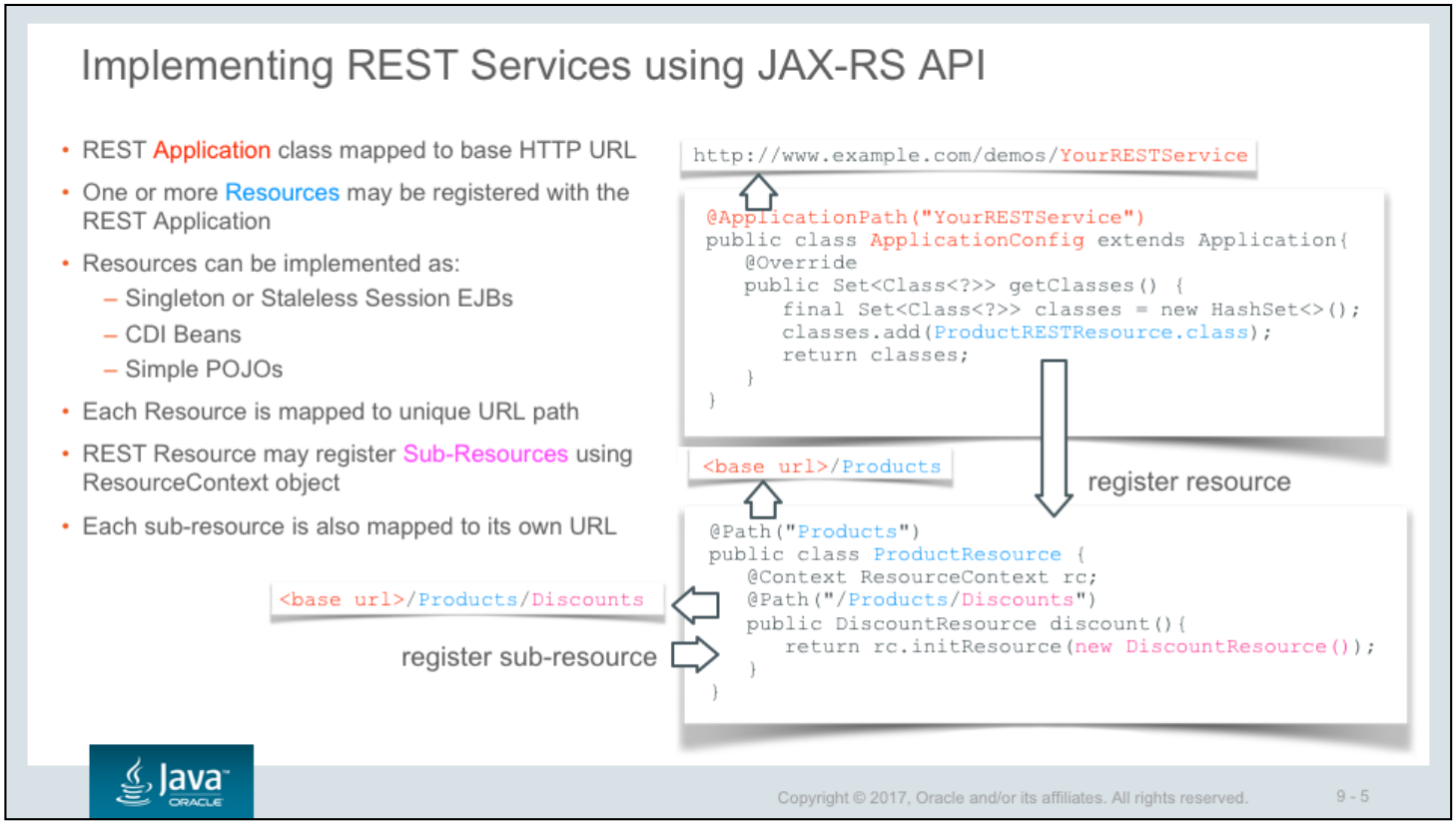
```
<Product>
   <id>101</id>
   <name>Tea</name>
   <price>1.99</price>
</Product>
```

JSON

```
{"Product": {
  "id": "101",
  "name": "Tea",
  "price": "1.99"
}}
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.



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 artefacts for the resource. A Java EE application archive containing JAX-RS resource classes will have the resources configured, the helper classes and artefacts generated, and the resource exposed to clients by deploying the archive to a Java EE server.

A JAX-RS application consists of at least one resource class packaged within a WAR file. The base URI from which an application's resources respond to requests can be set one of two ways:

Using the @ApplicationPath annotation in a subclass of javax.ws.rs.core.Application packaged within the WAR. By default, all the resources in an archive will be processed for resources. Override the getClasses method to manually register the resource classes in the application with the JAX-RS runtime.

```
package demos.ws;


import java.util.Set;
import javax.ws.rs.core.Application;
```

```
@ApplicationPath("/YourRESTService")
public class MyApplication extends Application {
  @Override
  public Set<Class<?>> getClasses() {
    final Set<Class<?>> classes = new HashSet<>();
    classes.add(Products.class);
    return classes;
  }
  @Override
  public Set<Object> getSingletons() {
     return super.getSingletons();
  }
}
```

Or instead of the @ApplicationPath annotating use the servlet-mapping tag within the WAR's web.xml deployment descriptor

```
<servlet-mapping>
  <servlet-name>demos.ws.ApplicationConfig</servlet-name>
  <url-pattern>/YourRESTService/*</url-pattern>
</servlet-mapping>
```

All root resource classes (classes with @Path("") at the class level) will appear under the path value in the @ApplicationPath("") annotation. If you have only one root resource class, you can use @Path("/") to make the root resource available at the path in the @ApplicationPath("resources") annotation.

Root resource classes can be POJOs, CDI-managed beans, and stateless and singleton session beans. The JAX-RS implementation will obtain your root resource class from an Application subclass.

If the class is returned from getClasses():

- JAX-RS will create an instance-per-request
- Perform a JAX-RS dependency inject
- Call any @PostConstruct methods
- Call the resource or resource-locator method

If the class is returned from getSingletons():

- It is expected to be a singleton
- It is typically used for providers instead of resources

Mapping Resources to URI Paths

Path annotation identifies the URI path template to which the resource responds

- Can be specified at the class or method level
- Can restrict values using regular expressions
- Can handle multiple parameters
- Operations may Produce or Consume various Media Types typically JSON or XML

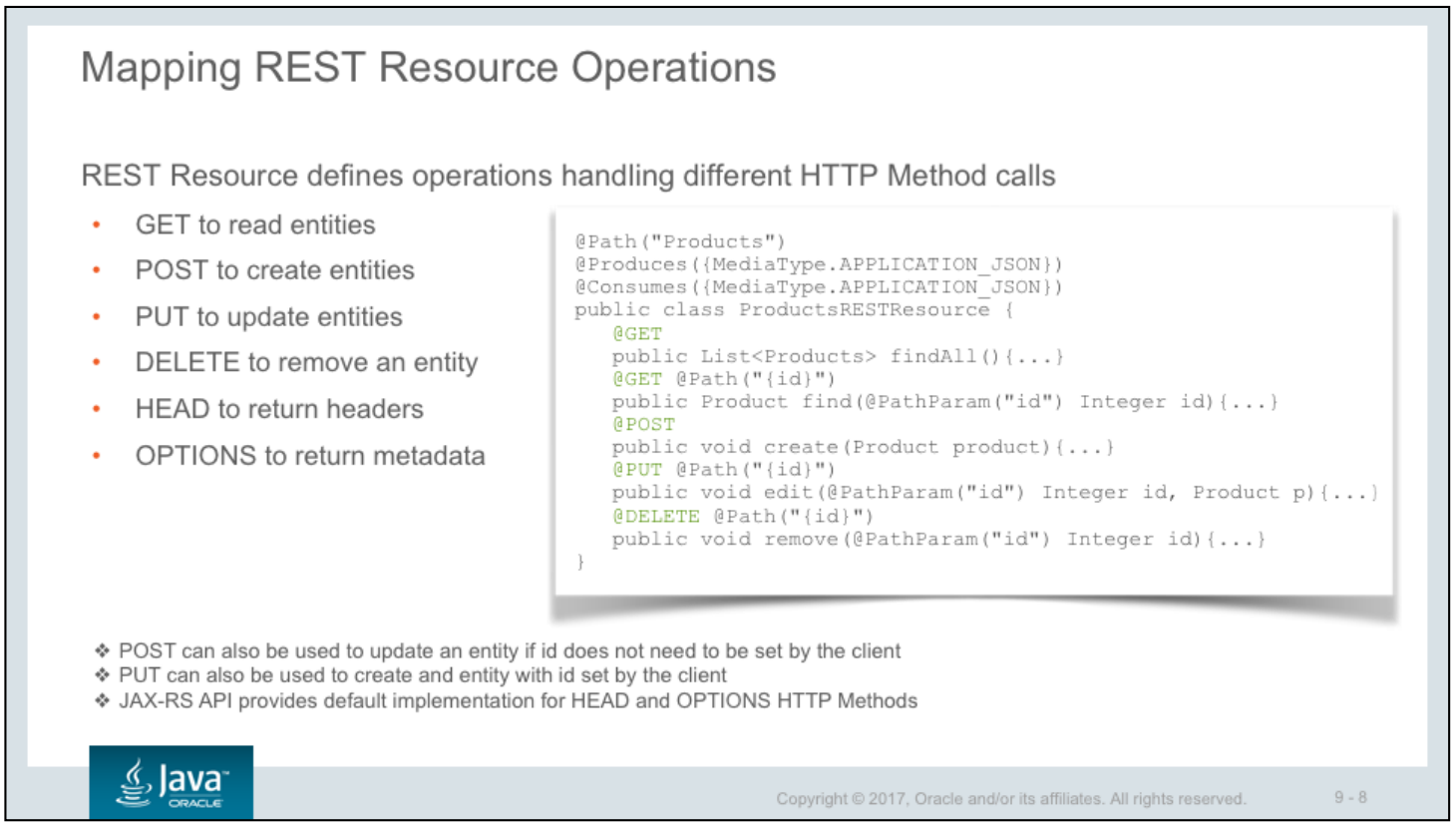
```
<base url>/Products/X42
```

```
<base url>/Products/10/20
```

```
@Path("Products")
@Produces({MediaType.APPLICATION_JSON})
@Consumes({MediaType.APPLICATION_JSON})
public class ProductsRESTResource {
    @GET
    @Path("{code: [A-Z][0-9]+}")
    public List<Product> findRange(@PathParam("code") String code){...}
    @GET
    @Path("{from}/{to}")
    public List<Product> findRange(@PathParam("from") Integer from,
                                   @PathParam("to") Integer to){...}
}
```

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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: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. If provided value does not match specified expression, a 404 (Not Found) response will be sent to the client.



It is possible that some REST services may use POST method not only to create new, but also to update existing elements. However, it implies that client did not provide an element id. Therefore, server should be able assign new id to an element if it is creating it, or to determine which element to update without id supplied by the client. In a similar way PUT method can be used to create as well as update elements. When using PUT method client is supposed to provide an element id.

JAX-RS API provides default implementation for HEAD and OPTIONS HTTP Methods:

- HEAD by default invokes the implemented GET method (if present) and ignores the response entity (if set). This method can be used for obtaining metadata about the entity implied by the request without transferring the entity-body itself. It is often used for testing URL links for validity, accessibility, and recent modification. The response to a HEAD request may be cacheable in the sense that the information contained in the response may be used to update a previously cached entity from that resource. If the new field values indicate that the cached entity differs from the current entity (as would be indicated by a change in Content-Length, Content-MD5, ETag or Last-Modified), then the cache MUST treat the cache entry as stale.
- OPTIONS by default produce a list HTTP methods supported by the resource and return Web Application Definition Language (WADL)

Handling Different Media Types

REST Services may produce and consume information using various Media Types

- Handle various Media Types typically JSON or XML
- JAX-RS API auto-converts HTTP request and response message bodies to and from java objects using MessageBodyReader and MessageBodyWriter classes
- @Produces and @Consumes annotations can be set on a class or method level to describe Media Types that this service supports

```
@Stateless
@Path("Products")
@Produces({MediaType.APPLICATION_JSON})
public class ProductsRESTResource {
   @GET @Path("{id}") @Produces({MediaType.APPLICATION_XML})
   public List<Product> find(@PathParam("id") Integer id){...}
   @GET @Path("count") @Produces(MediaType.TEXT_PLAIN)
   public String countProducts(){...}
   @PUT @Path("{id}") @Consumes({MediaType.APPLICATION_XML,MediaType.APPLICATION_JSON})
   public void edit(Product product){...}
}
```

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The @Produces and @Consumes annotations can be placed at the class level to define defaults for all class methods.

Adding the annotations at the method level overrides the class-level default. A method may both produce and consume content (depending on the HTTP method).

The @Produces and @Consumes annotations expect a string array for their value attribute: @Produces({"text/plain", "text/html"})

Use MediaType constants to avoid typos: @Produces({MediaType.TEXT_HTML, MediaType.TEXT_PLAIN})

Internally, JAX-RS uses a MessageBodyReader and MessageBodyWriter to convert the HTTP body to and from a Java object.

- MessageBodyReader converts the HTTP body from the entity stream to a Java object (param method).
- MessageBodyWriter converts the HTTP body from a Java object to an entity stream (return method).

A RESTful service will typically produce and consume XML or JSON. XML support is provided by JAXB.

The supported entity types are:

- byte[] – All media types (*/*)
- java.lang.String – All media types (*/*)
- 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 types (text/xml, application/xml and media types of the form application/*+xml)
- javax.xml.bind.JAXBElement and application-supplied JAXB classes XML types – (text/xml and application/xml and media types of the form application/*+xml)
- MultivaluedMap<String,String> – Form content (application/x-www-form- urlencoded)
- StreamingOutput – All media types (*/*), MessageBodyWriter only
- java.lang.Boolean, java.lang.Character, java.lang.Number – Only for text/plain. Corresponding primitive types supported via boxing/unboxing conversion.
- JsonStructure, JsonObject, and JsonArray – (application/json) On platforms that support JSON-P

Passing Parameters

REST Resource Handling classes may accepts parameters in different forms

- Path Parameters
- Query Parameters
- Matrix Parameters
- HTTP Header values as Parameters
- Cookie values as Parameters
- As application/x-www-form-urlencoded request body

`<base url>/Resource/acme`	`@PathParam("x")`
`<base url>/Resource?x=acme`	`@QueryParam("x")`
`<base url>/Resource;x=acme`	`@MatrixParam("x")`
`HTTP Header x=acme`	`@HeaderParam("x")`
`Cookie x=acme`	`@CookieParam("x")`
`application/x-www-form-urlencoded x=acme`	`@FormParam("x")`

- Declarations of parameters typically appear as method parameters, constructor parameters, fields, or bean properties
- A Default value can be provided with any @*Param annotation, when the input element is missing

```
@DefaultValue(10)
```

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You can combine any number of parameter types (including an unannotated body param) when declaring method arguments.

The @DefaultValue("value") annotation can be used in combination with the @*Param annotations to supply a default value when the input element is missing.

Declarations of parameters typically appear as method parameters of a resource method; however, they may also appear as constructor parameters, fields, or bean properties.

Validating Values

JAX-RS 2.0 supports the use of Bean Validation 1.1 (JSR-249)

- Validation constraints applied to resource method parameters, fields and property getters, as well as resource classes, entity parameters, and resource methods (return values)
- REST Services may use @Valid annotation to enable bean validations when representing an Entity that already uses bean validation annotations

```
@POST
@PUT("{id}/{price}")
@Consumes({MediaType.APPLICATION_JSON})
public void updatePrice(@NotNull
                        @PathParam("id") Integer id,
                        @NotNull @DecimalMax(value="999.99")
                                 @DecimalMin(value="0.99")
                        @PathParam("price") Integer price) {...}

@POST
@Consumes({MediaType.APPLICATION_JSON})
public void create(@Valid Product product){...}
```

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You can use Bean Validation API with JAX-RS Services in exactly same way as you used it with JPA Entities

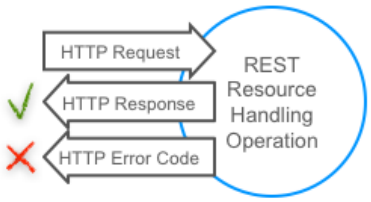
When a validation error occurs, a javax.validation.ValidationException or subclass is thrown. JAX-RS implementations must include an exception mapper, which maps a ValidationException to either a 400 (Bad Request) or 500 (Internal Server Error).

Handling Web Service Errors

When Handling Errors in REST Services

- Return HTTP status code to the client to indicate the nature of the error
- Operation may return a response containing an HTTP error code

```
ResponseBuilder rb = Response.status(Response.Status.NOT_FOUND);
return rb.build();
```



- Or throw a WebApplicationException to indicate HTTP error code to the client

```
throw new WebApplicationException(Response.Status.NOT_FOUND);
```

- Or throw one of the more specific exceptions (subclasses of the WebApplicationException)

```
throw new NotFoundException();
```

- Any other exceptions will require an ExceptionMapper (see notes)

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Status.NOT_FOUND represents an HTTP 404 error code. Therefore this code returns the same status:

ResponseBuilder rb = Response.status(404);

return rb.build();

or

throw new WebApplicationException(404);

or

throw new WebApplicationException(Response.Status.NOT_FOUND);

or

throw new NotFoundException();

NotFoundException is a subclass of the WebApplicationException. Other subclasses of the WebApplicationException class include:

BadRequestException	400	Malformed message
NotAuthorizedException	401	Authentication failure
ForbiddenException	403	Not permitted to access
NotFoundException	404	Couldn't find resource
NotAllowedException	405	HTTP method not supported
NotAcceptableException	406	Client media type requested not supported
NotSupportedException	415	Client posted media type not supported
InternalServerErrorException	500	General server error
ServiceUnavailableException	503	Server is temporarily unavailable or busy

Automatic mapping of Java Exceptions to Response objects is defined by the ExceptionMapper interface.

Providers implementing ExceptionMapper contract must be either programmatically registered in a JAX-RS runtime or must be annotated with @Provider annotation to be automatically discovered by the JAX-RS runtime during a provider scanning phase. It defines toResponse(E exception) operation that maps an Exception to a Response. Returning null results in a Response.Status.NO_CONTENT response. Throwing a runtime exception results in a Response.Status.INTERNAL_SERVER_ERROR response.

If in your application you have designed a custom exception (ProductNotFoundException), then it should be mapped to specific HTTP response code using a mapper:

```
@Provider
public class ProductNotFoundMapper
      implements ExceptionMapper<ProductNotFoundException> {

    public Response toResponse(ProductNotFoundException e) {
      return Response.status(Response.Status.NOT_FOUND).build();
    }
  }
```

Asynchronous REST Services

JAX-RS Web Services can process requests asynchronously

- AsyncResponse is similar to the Servlet 3.0 AsyncContext class and allows asynchronous request handling
- AsyncResponse resume operation produces the response to the client
- TimeoutHandler provides a mechanism for defining custom resolution of time-out events

```
@Path("/some")
public class AsyncService {
   @GET
   public void doThings(@Suspended final AsyncResponse ar) {
     ar.setTimeoutHandler(new TimeoutHandler() {
        public void handleTimeout(AsyncResponse ar {
           ar.resume(Response.status(
                     Response.Status.SERVICE_UNAVAILABLE)
                     .entity("Operation time out.").build());
        }));
     ar.setTimeout(20, TimeUnit.SECONDS);
     new Thread(new Runnable({
       public void run() {
          Object result = ...
          ar.resume(result);
       })).start();
   }
}
```

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Suspended annotation injects a suspended AsyncResponse into a parameter of an invoked JAX-RS resource or sub-resource method.

The injected AsyncResponse instance is bound to the processing of the active request and can be used to resume the request processing when a response is available.

By default there is no suspend timeout set and the asynchronous response is suspended indefinitely. The suspend timeout as well as a custom timeout handler can be specified programmatically using the AsyncResponse.setTimeout(long, TimeUnit) and AsyncResponse.setTimeoutHandler(TimeoutHandler) methods.

AsyncResponse has following capabilities:

Produce response that is either a normal response or an error with the fowling pair of methods:

- resume(Object response) - Resume the suspended request processing using the provided response data
- resume(Throwable response) - Resume the suspended request processing using the provided throwable

Cancel production of Response

- **cancel()** - Cancel the suspended request processing
- **cancel(Date retryAfter)** - Cancel the suspended request processing and set Retry-After header as a point in time
- **cancel(int retryAfter)** - Cancel the suspended request processing and set Retry-After header as a period of time in seconds

Set timeout and timeout handler

- **setTimeout(long time, TimeUnit unit)** - Set/update the suspend timeout.
- **setTimeoutHandler(TimeoutHandler handler)** - Set/replace a time-out handler for the suspended asynchronous response.

Find out status of the Response

- **isCancelled()** - Check if the asynchronous response instance has been cancelled
- **isDone()** - Check if the processing of a request this asynchronous response instance belongs to has finished
- **isSuspended()** - Check if the asynchronous response instance is in a suspended state

Register asynchronous processing lifecycle callback classes to receive lifecycle events for the asynchronous response based on the implemented callback interfaces

- register(Class<?> callback, Class<?>... callbacks)
- register(Object callback)
- register(Object callback, Object... callbacks)

CompletionCallback interface defines onComplete(Throwable throwable) operation that describes a completion callback notification method that will be invoked when the request processing is finished, after a response is processed and is sent back to the client or when an unmapped throwable has been propagated to the hosting I/O container.

An unmapped throwable is propagated to the hosting I/O container in case no exception mapper has been found for a throwable indicating a request processing failure. In this case a non-null unmapped throwable instance is passed to the method. Note that the throwable instance represents the actual unmapped exception thrown during the request processing, before it has been wrapped into an I/O container-specific exception that was used to propagate the throwable to the hosting I/O container. If Throwable parameter is null it indicates that the request processing has been completed and a response that has been sent to the client.

Asynchronous EJB and REST Services

Combination of EJB's @javax.ejb.Asynchronous annotation and the @Suspended AsyncResponse enables asynchronous execution of business logic with eventual notification of the interested client.

```
@Stateless
@Path("products")
public class ProductsRESTResource {
   @POST
   @Asynchronous
   public void handle(@Suspended AsyncResponse ar, Product p)
      // business logic execution
      Response response = Response.ok(p).build();
      ar.resume(response);
    }
}
```

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Invoking REST Service from JavaScript Client

JavaScript may invoke REST Service using Asynchronous JavaScript and XML API

- AJAX API handles both Synchronous and Asynchronous HTTP communications
- `XMLHttpRequest` object represents all types of http requests, regardless if they are actually XML or not
- Request URL pointing to the REST Service resource is prepared and opened
- Request is dispatched to the server via send operation, that can take an object as an argument
- When response has been received the `onreadystatechange` function is invoked
- Different JavaScript APIs are available to automate and simplify creation of such clients

❖ For more information on JavaScript and how it is used to invoke REST Services refer to "JavaScript and HTML5: Develop Web Applications" training course

```
var request = new XMLHttpRequest();
var id = document.getElementById("productId").value;
var url = "http://www.example.com/demos/YourRESTService/Products/"+id;
request.open('GET',url,true);
request.send();
request.onreadystatechange = function() {
   if (request.readyState == 4) {
      if (request.status == 200) {
         var product = JSON.parse(request.responseText);
         // handle product object
      }else{ alert("Error: " + request.responseText);}};
```

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Invoking REST Service from Java Client

Java clients may invoke REST Services using JAX-RS Jersey reference implementation library

- ClientBuilder class creates a Client object that handlers a client-side communication infrastructure
- Client object must be properly closed before being disposed to avoid leaking resources
- WebTarget object represents a REST Service resource
- WebTarget points to the resource via the HTTP URL, which can be constructed out of Path components
- JAX-RS API performs conversions between REST messages (such as JSON, XML etc.) and Java Objects
- WebTarget allows to submit GET/POST/PUT/DELETE etc. requests

```
String baseUrl = "http://www.example.com/demos/YourRESTService/";
Client client = ClientBuilder.newClient();
WebTarget webTarget = client.target(baseUrl).path("Products").path(id.toString());
Response response =
webTarget.register(Product.class).request(MediaType.APPLICATION_JSON).buildGet().invoke();
Product product = response.readEntity(Product.class);
client.close();
```

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Summary

In this lesson you should have learned how to handle communications with REST Services

- Understand REST service conventions
- Create REST services using JAX-RS API
- Consume REST service within the client tier

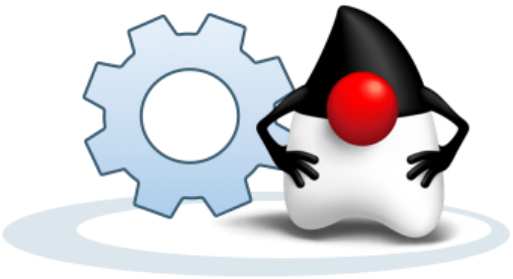
Java ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Practice

This practice covers the following tasks:

- Create REST Service to check if discount is available for a product
- Add JavaScript code to your web application to invoke Discount REST Service
- Create Java Client Application to invoke Discount REST Service



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.