10

JAX-RS RESTFul Web Services

Objectives

After completing this lesson, you should be able to do the following:

- Download, install, and configure Jersey
- Create application subclasses
- Create resource classes
- Create resource methods, sub-resource methods, and sub-resource locator methods
- Produce and consume XML and JSON content with JAX-RS

Course Roadmap

Application Development Using Webservices [SOAP and Restful]

- Lesson 1: Introduction to Web Services
- Lesson 2: Creating XML Documents
- Lesson 3: Processing XML with JAXB
- Lesson 4: SOAP Web Services Overview
- Lesson 5: Creating JAX-WS Clients

Course Roadmap

Application Development Using Webservices [SOAP and Restful]

- Lesson 6: Exploring REST Services
- Lesson 7: Creating REST Clients
- Lesson 8: Bottom Up JAX Web Services
- Lesson 9: Top Down JAX Web Services
- **Lesson 10: Implementing JAX RS Web Services**

You are here!

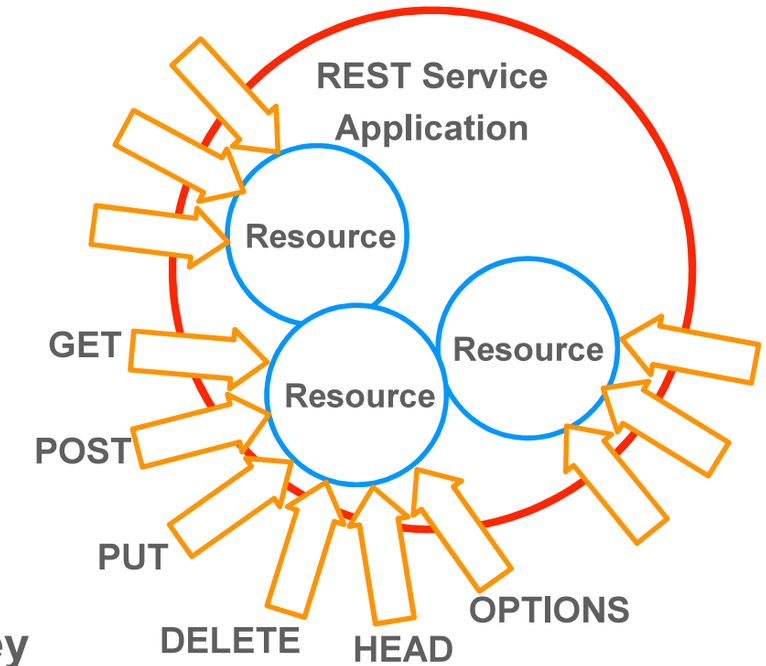
Course Roadmap

Application Development Using Webservices [SOAP and Restful]

- Lesson 11: Web Service Error Handling
- Lesson 12: Java EE Security and Securing JAX WS

REST Service Conventions and Resources

- REST Service Application represents one or more Resource
- 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.

REST Communication Model

REST Resources and REST Clients characteristics:

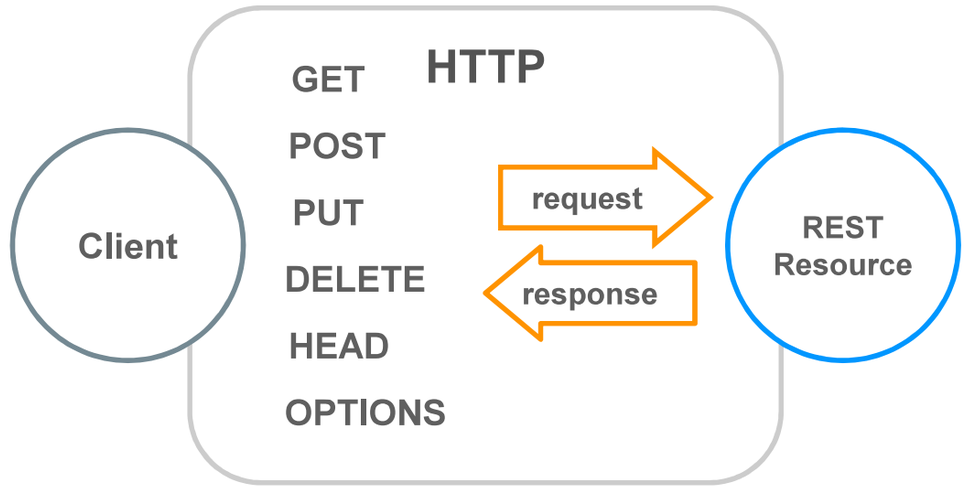
- REST Clients are often implemented by using JavaScript in Browser or Mobile Applications.
- Use HTTP as a transport layer and dispatch requests by using the GET, POST, PUT, DELETE, HEAD, and OPTIONS methods containing data that the client wants to pass to the REST Service.
- Clients handle responses 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"
}}
```

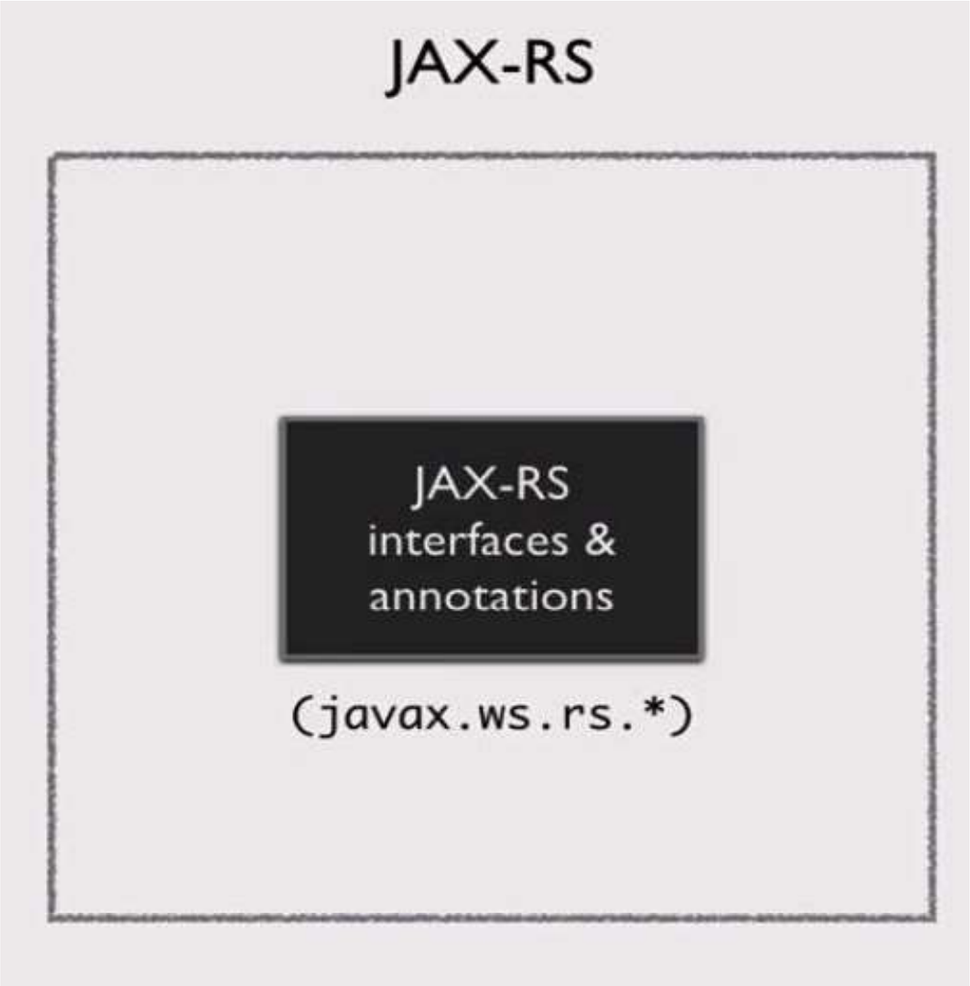


The JAX-RS Reference Implementation

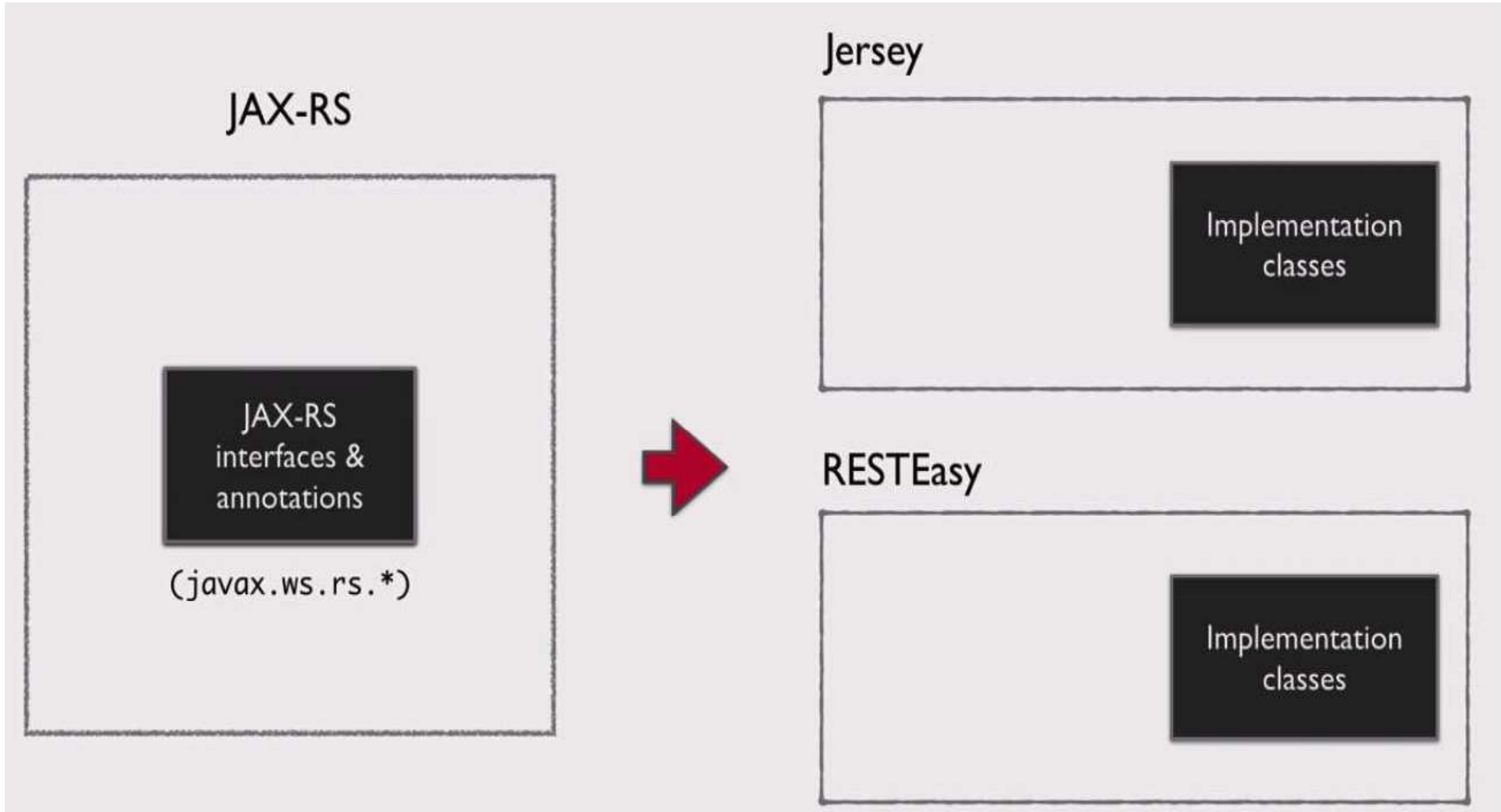
Jersey, http://jersey.java.net/, supports the implementation of RESTful web services.

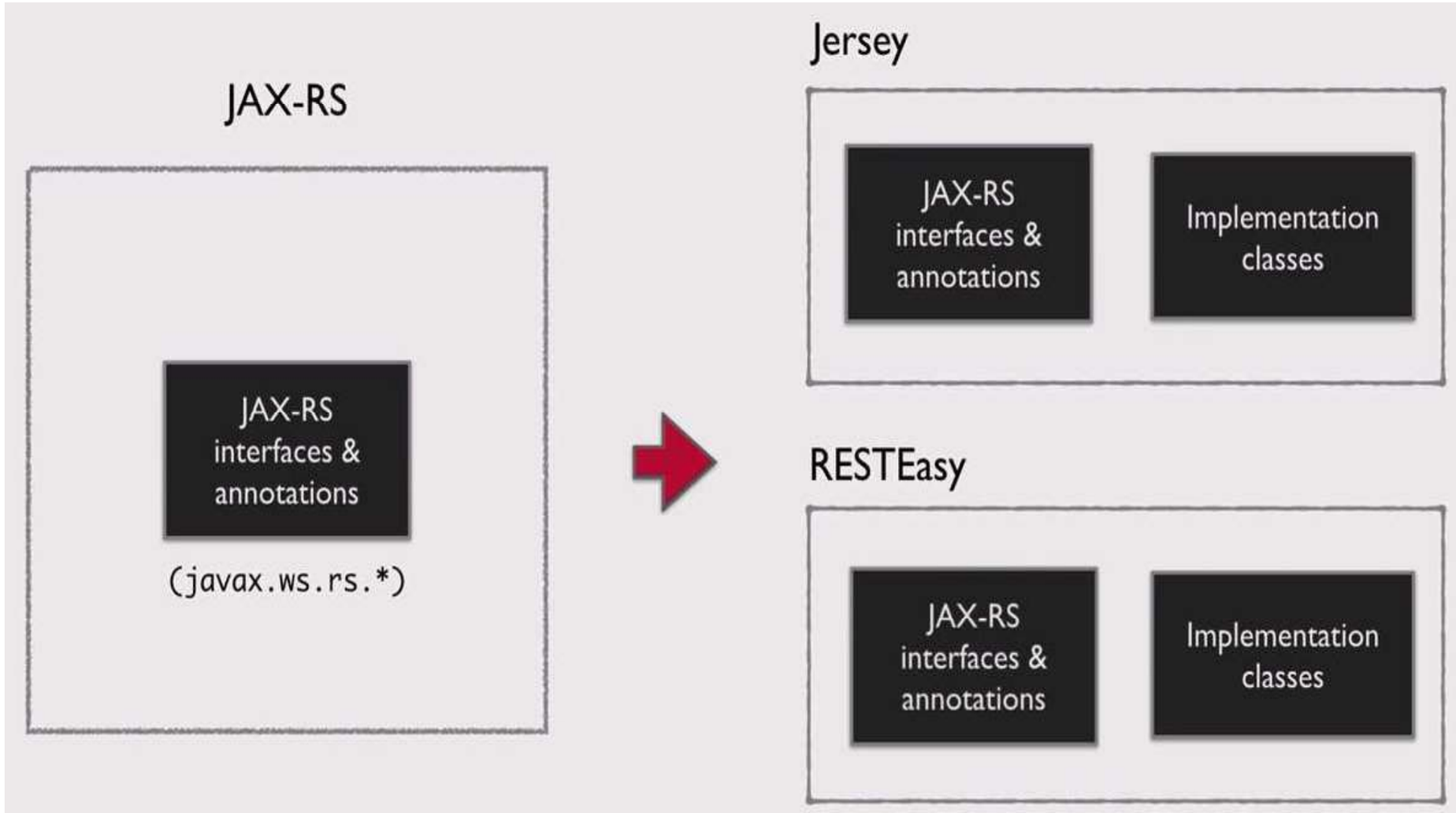
- A JAX-RS (JSR-311) Implementation. JSR-311 specifies only a server-side REST API
- Add-on for Servlet containers or Java EE Web Profile servers
- A client library, the Jersey Client API
- Extensions to the JAX-RS specification including:
  - Filters
  - OAuth
  - Model-View-Controller (MVC)

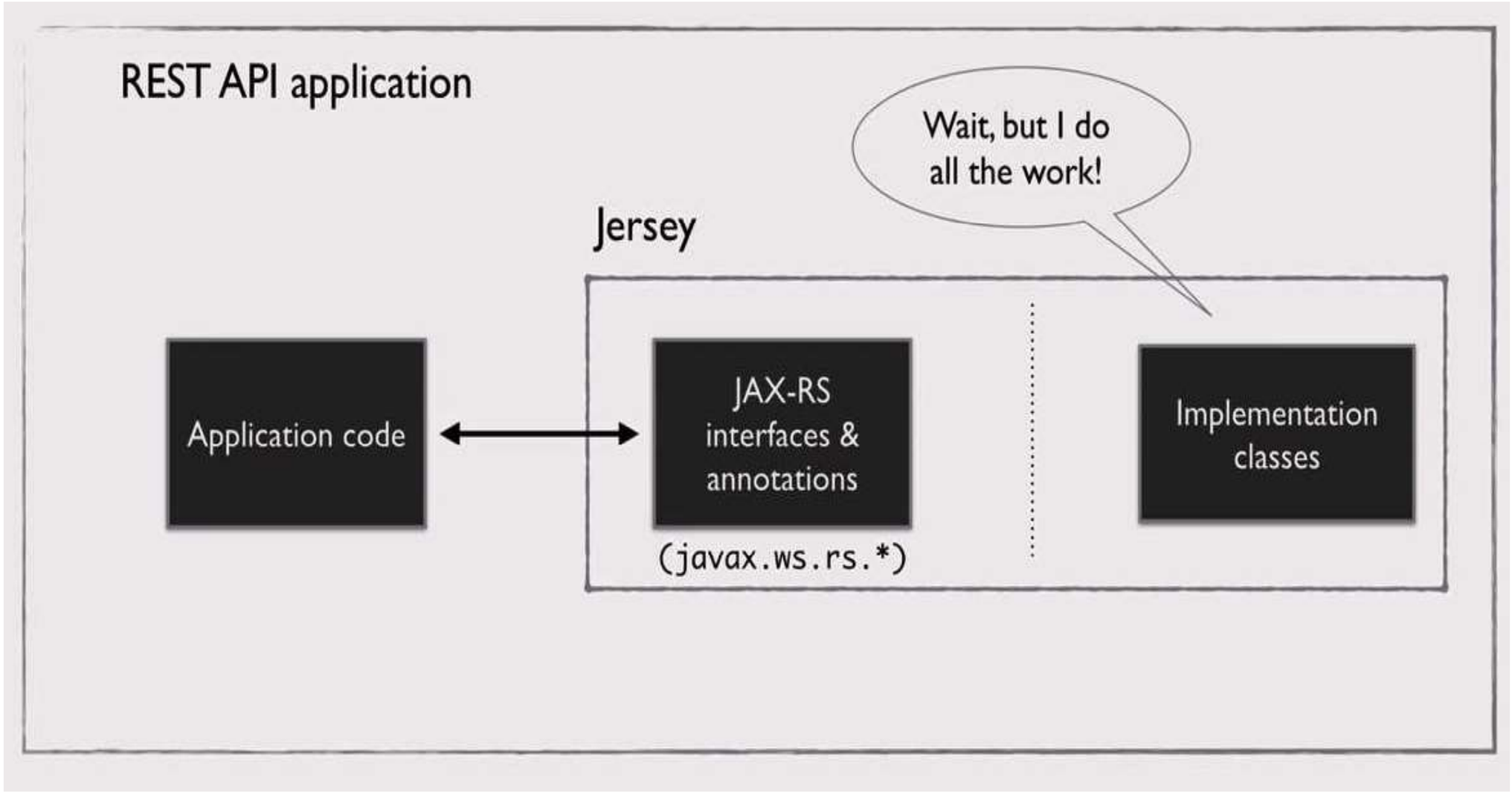
What is JAX RS ?



What is JAX RS ?




JAX-RS
JAX-RS interfaces & annotations
(javax.ws.rs.*)
Jersey
JAX-RS interfaces & annotations
Implementation classes
RESTEasy
JAX-RS interfaces & annotations
Implementation classes


REST API application
Wait, but I do all the work!
Jersey
Application code
JAX-RS interfaces & annotations
(javax.ws.rs.*)
Implementation classes

Updating the Version of Jersey in

With WebLogic Server you can use a more recent version of Jersey to get the latest bug fixes, performance improvements, and enhancements.

- WebLogic Server includes Jersey 1.9.
- Updated Jersey libraries can be packaged with the application or deployed to WLS as shared libraries.
- WLS class loading features allow applications to use their tested version of Jersey and keep Jersey 1.9 as the default.

Creating WebLogic Shared Libraries

WebLogic shared libraries can be used by any application deployed to the server. Applications request shared libraries.

1. Create an EE module such as a WAR.
2. Add library JARs to EE module.
3. Create a `MANIFEST.MF` file.

```
Manifest-Version: 1.0
Extension-Name: Jersey-1.17
Specification-Version: 1.17
Implementation-Version: 1.17.0
```

4. Configure `weblogic.xml`.

```
<prefer-application-packages>
    <package-name>com.sun.jersey.*</package-name>
</prefer-application-packages>
```

5. Deploy as a Library module using Admin Console.

Using WebLogic Shared Libraries

Assuming a WebLogic Shared Library has been deployed to the server:

- You must obtain the information from the `MAINFEST.MF` file of the shared library.
- In your application, add a `<library-ref>` element to your `weblogic.xml` file.

```
<library-ref>
    <library-name>Jersey-1.17</library-name>
    <specification-version>1.17</specification-version>
    <implementation-version>1.17.0</implementation-version>
    <exact-match>true</exact-match>
</library-ref>
```

Resources

The primary component in RESTful web services is the resource.

- A resource is a "thing" or noun.
- A resource has a unique URL.
- A resource in realized by a JAX-RS annotated class.
- A JAX-RS annotated class contains a Java method for each HTTP method supported.
- JAX-RS resources support consuming and producing XML and JSON by using JAXB.

Implementing REST Services using JAX-RS API

- REST Application class mapped to base HTTP URL
- One or more Resource may be registered with the REST Application
- Resources can be implemented as:
  - Singleton or Staleless Session EJBs
  - CDI Beans
  - Simple POJOs
- Each Resource is mapped to unique URL path
- REST Resource may register Sub-Resources using ResourceContext object
- Each sub-resource is also mapped to its own URL

<base url>/Products/Discounts

register sub-resource

http://www.example.com/demos/YourRESTService

```
@ApplicationPath("YourRESTService")
public class ApplicationConfig extends Application{
    @Override
    public Set<Class<?>> getClasses() {
        final Set<Class<?>> classes = new HashSet<>();
        classes.add(ProductRESTResource.class);
        return classes;
    }
}
```

<base url>/Products

register resource

```
@Path("Products")
public class ProductResource {
    @Context ResourceContext rc;
    @Path("/Products/Discounts")
    public DiscountResource discount(){
        return rc.initResource(new DiscountResource());
    }
}
```

A Simple Root Resource

To begin you must create a "root" resource class.

- Annotate a class with the `@Path("mypath")` annotation:

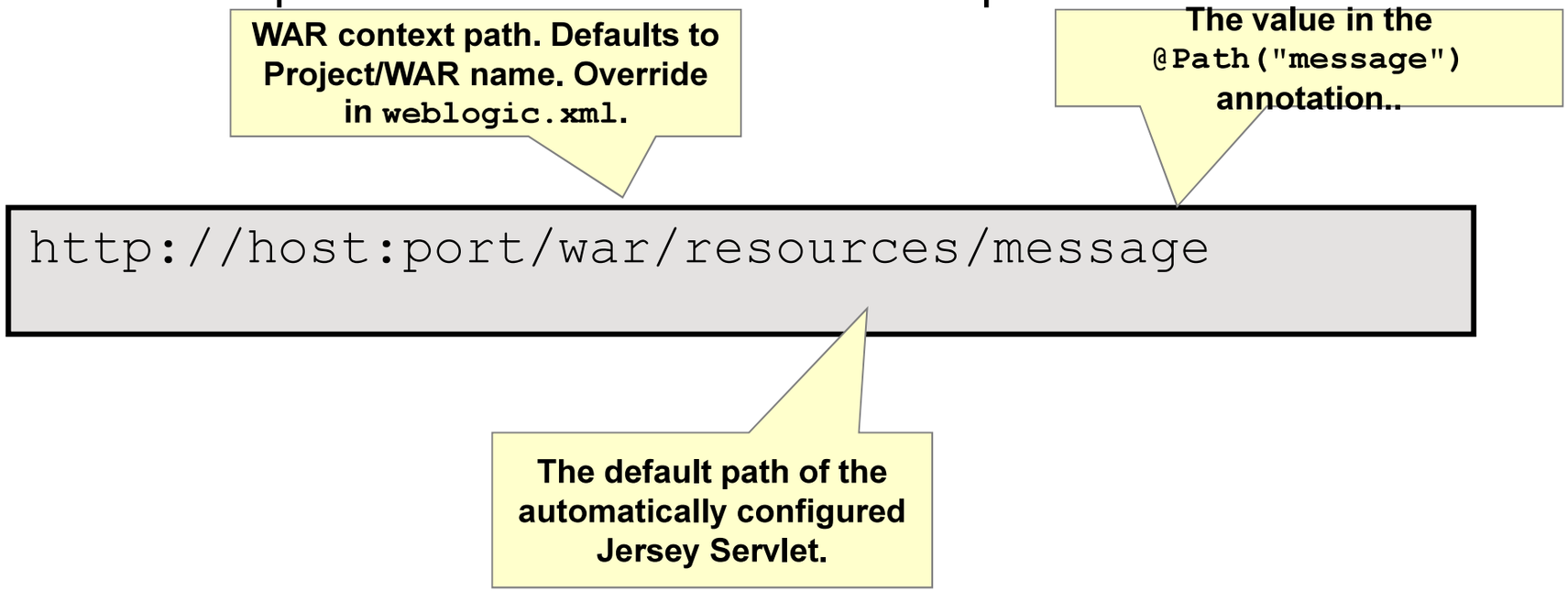
```
@Path("message")
public class Message {
    private static String message =
                                "Today's word is JAVA!";
    @GET
    public String getMessage() {
        return message;
    }
}
```

It does not matter if a path starts with a "/" or not.

JAX-RS locates methods by the HTTP method annotation, not by name.

Calling a Simple Root Resource

Resource classes are packaged into Web Application Archives (WARs). The context path of the WAR determines a part of the URL.



Resource Methods

A resource class, annotated with `@Path("")`, can contain zero or more resource methods.

- Resource methods are marked with an annotation that is named after an HTTP method.

```
@GET
public String getMessage() {
    return message;
}
@PUT
public void setMessage(String message) {
    this.message = message;
}
```

HTTP Method Annotations

A resource method may be annotated with:

- `@GET` – A "safe" and idempotent method that retrieves information
- `@PUT` – An idempotent method that adds (known identity) or updates a resource
- `@DELETE` – An idempotent method that deletes a resource
- `@POST` – A non-idempotent method that adds

Mapping REST Resource Operations

REST Resource defines operations handling different HTTP Method calls

- GET to read entities
- POST to create entities
- PUT to update entities
- DELETE to remove an entity
- HEAD to return headers
- OPTIONS to return metadata

```
@Path("Products")
@Produces({MediaType.APPLICATION_JSON})
@Consumes({MediaType.APPLICATION_JSON})
public class ProductsRESTResource {
    @GET
    public List<Products> findAll(){...}
    @GET @Path("{id}")
    public Product find(@PathParam("id") Integer id){...}
    @POST
    public void create(Product product){...}
    @PUT @Path("{id}")
    public void edit(@PathParam("id") Integer id, Product p){...}
    @DELETE @Path("{id}")
    public void remove(@PathParam("id") Integer id){...}
`
```

- POST can also be used to update an entity if id does not need to be set by the client
- PUT can also be used to create and entity with id set by the client
- JAX-RS API provides default implementation for HEAD and OPTIONS HTTP Methods

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){...}
}
```

Content-Type

A HTTP client sends an Accept header to indicate content type that it can understand. A server uses the Content-Type header to indicate the type returned.

- JAX-RS can select the resource method based on content type.

```
@GET
@Produces({MediaType.TEXT_PLAIN})
public String getMessage() {
    return message;
}
@GET
@Produces({MediaType.TEXT_HTML})
public String getHtmlMessage() {
    return "<h1>" + message + "</h1>";
}
```

Both methods are GET methods. The Accept header is used to determine which method gets called.

@Produces and @Consumes

- The `@Produces` and `@Consumes` annotations may be placed at the class level to define defaults for all class methods.
- Adding at the method level overrides the class-level default.
- A method may both produce and consume content.
- 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})
```

Consuming Content

Clients may place data in several different areas of an HTTP request.

```
@PUT
public void setMessage(String message) {
    this.message = message;
}
```

An unannotated method parameter will be set to the value of the HTTP request body.

- An HTTP body parameter
- Matrix parameters
- Query parameters
- Path parameters
- Header parameters
- Cookie parameters

Query and Other Parameter Types

- A query parameter follows a question mark in the URL.

```
http://host/resource?qname=value
public void method(@QueryParam("qname")String param) {}
```

- A Matrix parameter follows a semicolon in the URL.

```
http://host/resource;mname=value
public void method(@MatrixParam("mname")String param) {}
```

- A header parameter is an HTTP header in the request.

```
hname: value
public void method(@HeaderParam("hname")String param) {}
```

- A cookie parameter is included in the Cookie HTTP header.

```
Cookie: cname=value
public void method(@CookieParam("cname")String param) {}
```

- A form parameter is read from an application/x-www-form-urlencoded request body.

```
pname=value&pname2=value
public void method(@FormParam("pname")String param) {}
```

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)
```

Sub-Resource Methods and `@PathParam`

A method with a `@Path("")` annotation is a sub-resource method. Its URL is created by combining path values.

```
@Path("messages")
public class Messages {
    private static List<String> messages =
                                        new ArrayList<>();
    @GET
    @Path("{id}")
    @Produces({MediaType.TEXT_PLAIN})
    public String getMsg(@PathParam("id") Integer id) {
        return messages.get(id);
    }
    /* … */
}
```

A template parameter

JAX-RS can do type conversion.

A PathParam extracts a part of the URL.

Sub-Resource Locator Methods

Do not handle all URLs in a single root resource class. Delegate to sub-resource classes by using locator methods.

```
@Path("/")
public class MyClass {
  @Path("/subresource2")
  public SomeClass subResourceLocator() {
    return new SomeClass();
  }
}
```

locator method

```
public class SomeClass {
  @GET
  public String resourceMethod() {}
}
```

JAXB can be used to receive and return complex types.

- Annotate classes for JAXB use.
  - `@XmlRootElement` and so on
- Resources should produce and consume `"application/xml"`
  - `@Produces({MediaType.APPLICATION_XML})`
  - `@Consumes({MediaType.APPLICATION_XML})`
- JAX-RS has native support for JAXB object; nothing else is required.

Application Subclasses

An application class defines the resources and providers that make up your RESTful application.

- Your JAX-RS implementation (Jersey) supplies the default application class.
- To customize the resources, providers, and their life cycles, you should supply an application subclass.
- In some environments, the default behavior that scans for and activates all root resource classes may not find your classes.
- Application subclasses can be:
  - POJOs
  - Application-scoped or Singleton-scoped CDI beans
  - Stateless or Singleton Session beans

Application Subclass: Example

```
@ApplicationPath("resources")
public class MyApplication extends Application {
    @Override
    public Set<Class<?>> getClasses() {
        Set<Class<?>> classes = new HashSet<>();
        classes.add(Messages.class);
        return classes;
    }

    @Override
    public Set<Object> getSingletons() {
        return super.getSingletons();
    }
}
```

Returns root resource classes

Returns provider classes.

web.xml Configuration

```
<servlet>
  <servlet-name>
    javax.ws.rs.Application
  </servlet-name>
    <servlet-class>
com.sun.jersey.spi.container.servlet.ServletContainer
  </servlet-class>
   <init-param>
    <param-name>javax.ws.rs.Application</param-name>
    <param-value>ou.MyApplication</param-value>
   </init-param>
</servlet>
```

Your application subclass

JAXB and JSON with JAX-RS

Jersey supports using the Jackson library to marshall/unmarshall JSON using JAXB.

- With the Jackson library on the classpath it just works.
  - Mostly, there are some configuration options.
- Annotate classes for JAXB use.
  - `@XmlRootElement`, etc.
- Resources should produce and consume `"application/json"`
  - `@Produces({MediaType.APPLICATION_JSON})`
  - `@Consumes({MediaType.APPLICATION_JSON})`
- You can produce and consume JSON and XML with the same methods.

HTTP Response Status

All methods mapped to an HTTP method result in the return of a numeric status code.

- Returning void or null results in a 204 (No Content) status.
- Simple and JAXB return types result in a 200 (OK) status.
- Exceptions can result in 4XX and 5XX error codes.
- To control the response value you should use `javax.ws.rs.core.Response` as the return type of methods.
  - A large number of methods will use this return type.
  - A response can include simple and complex (JAXB) data.

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

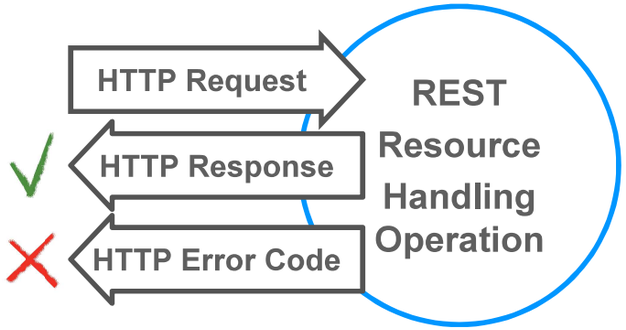
```
@POST
@PUT("{id}/{price}")
@Consumes({MediaType.APPLICATION_JSON})
public void updatePrice(@NotNull
                        @PathParam("id") Integer id,
                        @NotNull @DecimalMax(value="999.99")
                                 @DecimalMin(value="0.99")
                       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

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

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)

Life Cycle and Resource Class Types

- Root resource classes can be POJOs, CDI-managed beans, 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()`
    - Jersey 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
    - Typically used for providers instead of resources

```
<init-param>
    <param-name>com.sun.jersey.spi.container.ResourceFilters</param-name>
     <param-value>
        com.sun.jersey.api.container.filter.RolesAllowedResourceFilterFactory
     </param-value>
</init-param>
<init-param>
    <param-name>
      com.sun.jersey.spi.container.ContainerRequestFilters
    </param-name>
    <param-value>
      com.sun.jersey.api.container.filter.LoggingFilter
    </param-value>
</init-param>
<init-param>
    <param-name>
       com.sun.jersey.spi.container.ContainerResponseFilters</param-name>
    <param-value>com.sun.jersey.api.container.filter.LoggingFilter</param-value>
</init-param>
```

Called after container filters, may only filter some resources.

Inbound filter

Outbound filter

Providers

Providers extend the capabilities of your JAX-RS application. There are three types of providers:

- Entity – Support marshalling/unmarshalling new types
- Context – Support new context types
- Exception Mapping – Convert exceptions to responses

Providers:

- Must be thread-safe
- Are annotated with `@Provider`
- Can be:
  - POJOs
  - Application-scoped or singleton-scoped CDI beans
  - Stateless or singleton session beans

Context Providers

JAX-RS uses its own form of dependency injection, the `@Context` annotation. Only the listed types are required to be supported by JAX-RS.

- `@Context Application`
- `@Context UriInfo`
- `@Context HttpHeaders`
- `@Context Request`
- `@Context SecurityContext`
- `@Context Providers`
- `@Context ServletConfig`
- `@Context ServletContext`
- `@Context HttpServletRequest`
- `@Context HttpServletResponse`

Life Cycle

- Resource classes are instance-per-request, no state
- Root resource classes support common life-cycle annotations like `@PostConstruct`.
- You can perform JAX-RS inject on fields or method parameters:

```
(@QueryParam("qname")String param;
```

(except in sub-resource classes where only method parameter injection works)

- Constructor injection support is optional (don't use it).

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();
   }
}
```

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 onreadystatechage function is invoked
- Different other JavaScript APIs are available to automate and simplify creation of such clients

```
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);}};
```

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();
```

Deploying a Web Service on JavaSE

```
1  public class AirportRM {
2   // ...
3   static public void
4   main(String[] args) throws IOException {
5    String contextUrl =
6      "http://localhost:8080/jaxrs";
7     if (args.length > 0)
8      contextUrl = args[1];
9     HttpServer server =
10      HttpServerFactory.create( contextUrl );
11    server.start();
12  }
13 }
```

What are the three categories of methods that might be called to handle a HTTP GET by a JAX-RS implementation?

a. Resource
b. Nested-Resource
c. Sub-Resource
d. Factory
e. Locator

A sub-resource locator method in annotated with one or more HTTP method annotations such as @GET.

a. True
b. False

A root resource class must be annotated with:

a. @Produces
b. @GET
c. @WebService
d. @Path

Resources

Topic	Website
Jersey User Guide	http://jersey.java.net/nonav/documentation/latest/user-guide.html
Developing RESTful Web Services for Oracle WebLogic Server	http://docs.oracle.com/cd/E24329_01/web.1211/e24983/toc.htm
The Java EE 6 Tutorial - Building RESTful Web Services with JAX-RS	http://docs.oracle.com/javaee/6/tutorial/doc/giepu.html
Jersey Samples	https://maven.java.net/content/repositories/releases/com/sun/jersey/samples/

Summary

In this lesson, you should have learned how to:

- Download, install, and configure Jersey
- Create application subclasses
- Create resource classes
- Create resource methods, sub-resource methods, and sub-resource locator methods
- Produce and consume XML and JSON content with JAX-RS

Practice 10 : Overview

This practice covers the following topics:

- The Rules of Indian Rummy
- Creating the Indian Rummy Web Service Project
- Creating the Indian Rummy Game Creation REST Resources
- Using JSON as a Data Interchange Format