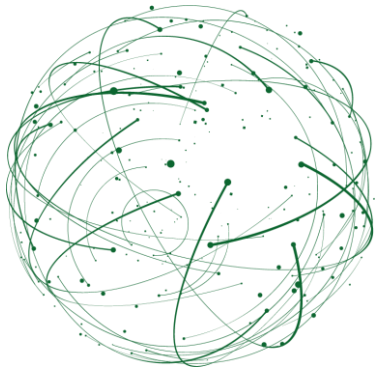
Zachodniopomorski Uniwersytet Technologiczny w Szczecinie

Wydział Informatyki

KATEDRA INŻYNIERII OPROGRAMOWANIA http://wi.zut.edu.pl/

PROJEKTOWANIE OPROGRAMOWANIA/SOFTWARE DESIGN



Aplikacje Webowe i Rozproszone

#04 : JavaScript Object Notation (JSON)
Java API for RESTful Web Services (JAX-RS)

Prowadzący:

Krzysztof Kraska

email: kkraska@zut.edu.pl

Szczecin, 4 kwietnia 2020 r.

JAVA API FOR RESTFUL WEB SERVICES (JAX-RS)

Objectives

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

- Describe the need for web services
- Design a RESTful web service and its collection of resource URLs
- Create methods that follow the prescribed rules of HTTP method behavior, including idempotence
- Create a JAX-RS resource and application classes
- Consume query and other parameter types
- Produce and consume complex data in the form of XML
- Identify and return appropriate HTTP status codes for any condition

ORACLE

Szczecin, 4 kwietnia 2020 r.

JAVA API FOR RESTFUL WEB SERVICES (JAX-RS)

Need for Web Services

A computer program calling a subroutine located on a different machine on a network is not a new development.

- Remote Procedure Call (RPC) from Sun Microsystems was an early example of cross-system execution.
- Other examples include .Net Remoting, CORBA/IIOP, and RMI.
- In all cases, the need is the same:
  - Run an operation on a remote machine (or even on the same machine but in a different address space).
- Many of the remoting technologies suffer from problems:
  - Platform-specific (CPU architecture or programming language)
  - Complexity

JAVA API FOR RESTFUL WEB SERVICES (JAX-RS)

Characteristics of Web Services

- Platform neutral (both CPU architecture and programming language independent)
  - A platform-neutral data-interchange format is needed (text instead of binary).
- Client/server architecture
  - A server has a set of available operations.
  - A client can request the execution of an operation on a server.
- HTTP
  - "Web" services are most likely to use HTTP as a transport protocol.
    - In theory, both SOAP and RESTful services are not tied to HTTP. In practice, you almost always use HTTP.
- The use of a service must be described.

ORACLE

JAVA API FOR RESTFUL WEB SERVICES (JAX-RS)

Reasons why HTTP works well for a remote method invocation protocol include:

- **HTTP libraries and utilities:** They are available for every language and platform.
- **Hardware support:** HTTP and TCP/IP are supported by all modern networking equipment.
- **Security Policies:** Corporate routers often block all but the most common types of network traffic. By using common ports and protocols, network traffic is likely to be allowed to flow across corporate networks. Some issues may arise when a web service uses all the features of HTTP, unlike normal web browsing traffic, which is typically limited to `HEAD`, `GET`, and `POST` operations.

JAVA API FOR RESTFUL WEB SERVICES (JAX-RS)

XML and JSON in Web Services

To facilitate a platform-neutral exchange of data, a general purpose data-interchange format is needed.

- **Extensible Markup Language (XML):** Used by both SOAP and REST web services
  - A large number of processing libraries exist to support XML in almost every language.
  - Java developers can use SAX, DOM, StAX, and JAXB.
  - SOAP web service rely on XML but libraries hide a large portion of the XML work.
- **JavaScript Object Notation (JSON):** Used by REST web services
  - A subset of JavaScript
  - Less verbose than XML
  - Support evolving in other languages

JAVA API FOR RESTFUL WEB SERVICES (JAX-RS)

REST Resources

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 OOA&D where you identify nouns in use-cases.

JAVA API FOR RESTFUL WEB SERVICES (JAX-RS)

Resource Design

A RESTful web service contains multiple resources.

- Resources are linked together (hypermedia).
- Design a tree of resources with a single base resource.
- It is comparable to an object graph in Java or a hierarchy of elements in an XML document.
- Resources are (usually) nouns or things.
- Each resource has a limited number of general-purpose operations that it may support (`GET`, `PUT`, `POST`, `DELETE`).
- A resource is uniquely identified by a URL:

  http://llocalhost:7001/myapp/resources/users

  http://llocalhost:7001/myapp/resources/users/matt

A collection is a resource.

A specific resource in a collection

JAVA API FOR RESTFUL WEB SERVICES (JAX-RS)

HTTP Methods

HTTP 1.1 methods are outlined by RFC 2616.

Method	Purpose
GET	Read, possibly cached
POST	Update or create without a known ID
PUT	Update or create with a known ID
DELETE	Remove
HEAD	Read headers, has version changed?
OPTIONS	List the “Allow”ed methods

ORACLE

JAVA API FOR RESTFUL WEB SERVICES (JAX-RS)

In practice, Java-based server-side UI web applications use only the `GET` and `POST` requests. JavaScript-based client-side UI web applications make AJAX Web Service requests against a JAX-RS back-end and potentially use the full range of HTTP methods.

The `GET` and `POST` requests have similar capabilities, because they both supply a request that contains a URI, and they both expect some data to be returned in the response body. The differences between `GET` and `POST` requests can best be seen in the way that form data is sent from the browser to the server. For example, if the user submits a login form by using `GET`, the browser might display the form data with the URL, as shown in the following example:

```
GET /FormExample/ReadForm?user=weblogic&password=welcome1 HTTP/1.1
```

The browser issues a `POST` request when it submits a form with a definition in HTML that contains the `method="POST"` attribute as follows:

```
<form action="ReadForm" method="POST">
```

When submitting a POST request, all form data will be placed in the request body instead of being included in the URL.

RESTful web services that use HTTP should use HTTP as the application-level protocol. This means that the HTTP methods are the only methods that clients can call on your URLs or resources. The small number of general-purpose methods is one reason why a RESTful service will have a large number of resources.

HTTP Status Codes

- **1xx – Informational:** Request received, continuing process.
- **2xx – Success:** Action successfully received, understood, and accepted.
- **3xx – Redirection:** Client must take additional action to complete the request.
- **4xx – Client Error:** Request contains bad syntax or cannot be fulfilled.
- **5xx – Server Error:** Server failed to fulfill an apparently valid request.

Idempotence

Some HTTP requests, when repeated, will have no effect, whereas other methods will.

- `GET`: Read only and idempotent. Never changes the state of the resource.
- `PUT`: An idempotent insert or update of a resource. Idempotent because it is repeatable without side effects.
- `DELETE`: Resource removal and idempotent
- `POST`: Non-idempotent, “anything goes” operation

JAVA API FOR RESTFUL WEB SERVICES (JAX-RS)

Resources

The primary component of a RESTful web service is the resource.

- A resource is a “thing” or noun.
- A resource has a unique URL.
- A resource is 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.

Resource Collections

It is very common to represent collections of resources; the collection itself is a resource.

- `GET /collection`: Return a listing of hyperlinks to the elements in the collection.
- `PUT /collection`: Replace everything.
- `POST /collection`: Create one new resource in the collection; return the generated ID of that new resource.
- `DELETE /collection/{id}`: Delete an item from the collection.
- `PUT /collection/{id}`: Update an item in the collection or create a new item with this ID.

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;
    }
}
```

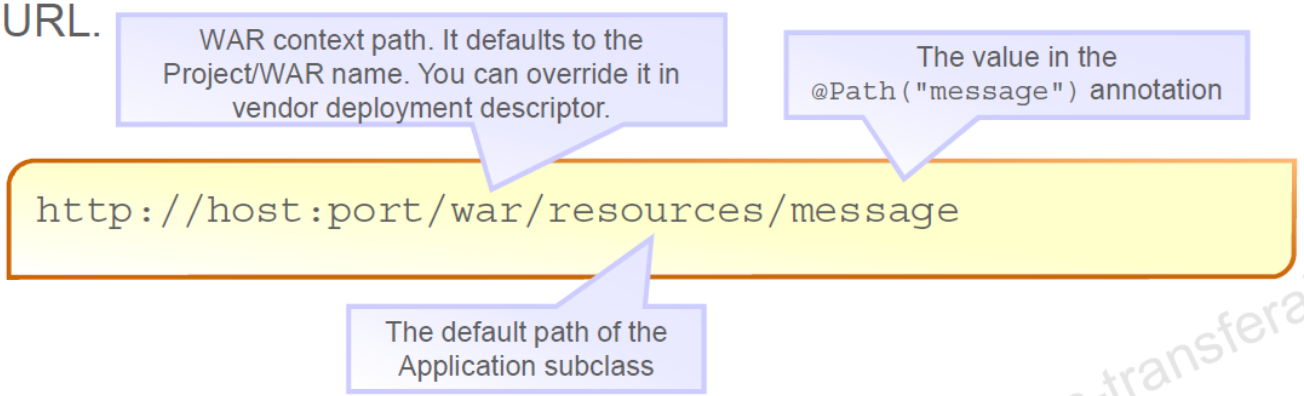
It does not matter if a path starts with a “/” or not.

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

JAVA API FOR RESTFUL WEB SERVICES (JAX-RS)

Calling a Simple Root Resource (What Is the URL?)

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;
}
```

The method return value will serve as the body of the HTTP response, also known as the entity.

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

HTTP Method Annotations

A resource method can 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 a resource
- `@HEAD` – Automatically supported; calls your `@GET` method and drops the body
- `@OPTIONS` – Automatically supported; responds with an Allowed header that shows which HTTP methods are supported for a given URL and includes the WADL in the body

Supported Entity Types

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.

```
@XmlRootElement
public class Person {}
```

The `@XmlRootElement` annotation on a class makes the class an application-supplied JAXB class.

Content Type

An HTTP client sends an Accept header to indicate the 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.

JAVA API FOR RESTFUL WEB SERVICES (JAX-RS)

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})
```

Consuming Content

Clients can place data in different areas of an HTTP request.

```
@PUT
public void setMessage(@QueryParam("name") String n,
                       String message) {
    this.message = message;
}
```

An annotation is used to extract data from other parts of the incoming HTTP message.

A single unannotated method parameter will be set to the value of the HTTP request body. This is known as the "entity".

The incoming HTTP request may contain useful data in any of the three main parts of an HTTP message:

- The request line (URL line)
- The HTTP headers
- The HTTP body (entity)

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 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) {}
```

ORACLE

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. The injected types listed in the slide typically appear as method parameters of a resource method; however, they may also appear as constructor parameters, fields, or bean properties.

Subresource Methods and `@PathParam`

A method with an `@Path("")` annotation is a subresource 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.

JAVA API FOR RESTFUL WEB SERVICES (JAX-RS)

The `@Path("")` annotation can contain any number of template parameters, for example, `@Path("{state}/{city}/mayor")`.

JAX-RS can automatically convert from text input (all HTTP is text) to:

- Primitives
- Strings
- Anything with a `Type(String s)` constructor
- Anything with a `Type.valueOf(String s)` method
- Anything with a `Type.fromString(String s)` method
- `List<T>`, `Set<T>`, `SortedSet<t>` of the above types

Subresource Locator Methods

Do not handle all URLs in a single root resource class. Delegate URLs to subresource 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() {}
}
```

JAVA API FOR RESTFUL WEB SERVICES (JAX-RS)

`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 if you do not provide one.
  - The default is to enable all resource classes.
  - You must attach the base application URL with `web.xml`.
- If you supply an `Application` subclass and annotate it with `@ApplicationPath("somepath")`, then you do not need to create a `web.xml` deployment descriptor.
- `Application` subclasses can be:
  - POJOs
  - Application- 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.

JAVA API FOR RESTFUL WEB SERVICES (JAX-RS)

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()`:
  - 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
  - It is typically used for providers instead of resources

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.

JAVA API FOR RESTFUL WEB SERVICES (JAX-RS)

javax.ws.rs.core.Response

```
@POST
@Consumes({MediaType.TEXT_PLAIN})
public Response setMessage(String message) {
    int index = counter++;
    messages.put(index, message);
    URI uri = URI.create(Integer.toString(index));
    ResponseBuilder responseBuilder =
                                Response.created(uri);
    return responseBuilder.build();
}
```

Shortcut for:
ResponseBuilder responseBuilder = Response.status(Response.Status.CREATED);
responseBuilder.location(uri);

JAVA API FOR RESTFUL WEB SERVICES (JAX-RS)

The following example shows how to return a 404 or 200 status from a `GET` method:

```
@GET
@Path("{id}")
@Produces({MediaType.TEXT_PLAIN})
public Response getMsg(@PathParam("id") Integer id) {
    String msg = messages.get(id);
    ResponseBuilder responseBuilder;
    if(msg == null) {
        responseBuilder =
Response.status(Response.Status.NOT_FOUND);
    } else {
        responseBuilder = Response.ok(msg);
    }
    return responseBuilder.build();
}
```

Web Service Errors

Web Services can experience errors in two places:

- On the server
  - In your web service an exception is thrown. How you convey that to a client depends on the type of web service (SOAP or REST).
- On the client
  - Clients receive the errors produced by a web service.
  - Clients experience error without there being any error produced by a server (networking problems, for example).

Basic Error Responses with JAX-RS

Reporting errors (exceptions) to RESTful clients is simply a matter of returning the correct HTTP status code.

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

```
ResponseBuilder rb =
Response.status(404);
return rb.build();
```

WebApplicationException

`WebApplicationException` is a runtime exception that can be thrown from any HTTP method to produce an HTTP 4XX or 5XX status response.

```
throw new WebApplicationException()
throw new WebApplicationException(int)
throw new WebApplicationException(Response.Status)
```

Validation

JAX-RS 2.0 supports the use of Bean Validation 1.1 (JSR-249) when run in an environment that supports Bean Validation.

- Validation constraints can be placed in the same location as `@MatrixParam`, `@QueryParam`, `@PathParam`, `@CookieParam`, `@HeaderParam`, and `@Context`.
- This includes resource method parameters, fields and property getters, as well as resource classes, entity parameters, and resource methods (return values).

```
@POST
@Consumes("application/x-www-form-urlencoded")
public void signup(@NotNull @Email @FormParam("email") String
email){
    //...
}
```

Built-in validator

Custom validator

ORACLE

JAVA API FOR RESTFUL WEB SERVICES (JAX-RS)

Objectives

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

- Use the JAX-RS 2 Client API
- Describe the alternatives to the JAX-RS 2 Client API:
  - `HttpURLConnection` class
  - Jersey 1.X Client API

JAVA API FOR RESTFUL WEB SERVICES (JAX-RS)

Communicating with Web Servers

- Java provides a simple mechanism for communicating with HTTP servers via `URL` objects and their associated `URLConnections`.
- Jersey 1 provides a client API for convenient access to JAX-RS web services.
- JAX-RS 2 (Java EE 7) provides a client API for convenient access to JAX-RS web services.
- Third-party libraries, such as Apache's `HttpClient`, provide finer-grained access to HTTP servers.

JAVA API FOR RESTFUL WEB SERVICES (JAX-RS)

Creating a JAX-RS Client

```
public class JaxRsClient {
    public static void main(String[] args) {
        String baseURL =
                "http://localhost:8080/JaxRsExample/resources";
        Client client = ClientBuilder.newClient();
        WebTarget target
                = client.target(baseURL);
        target = target.path("message");
        Builder builder = target.request(MediaType.TEXT_PLAIN);
        String result = builder.get(String.class);
        System.out.println("Result: " + result);
```

Continue building the URL.

WebTarget represents a URL.

A Builder is used to specify headers.

Call an HTTP method, and provide the body (entity) and the return type.

Method Chaining JAX-RS Client

```
public class JaxRsClient {
    public static void main(String[] args) {
        String baseURL =
                "http://localhost:8080/JaxRsExample/resources";
        Client client = ClientBuilder.newClient();
        WebTarget target
                = client.target(baseURL);
        String result = target
                .path("message")
                .request(MediaType.TEXT_PLAIN)
                .get(String.class);
        System.out.println("Result: " + result);
    }
}
```

Method calls are chained together.

JAVA API FOR RESTFUL WEB SERVICES (JAX-RS)

javax.ws.rs.client.WebTarget

`WebTarget` allows the application to:

- Represent a base or complete URI
  - `client.target(String)`
- Append additional path elements to the URI
  - `path(String)`
- Append Query Parameters
  - `queryParam(String, Object…)`
- Append Matrix Parameters
  - `matrixParam(String, Object…)`
- Resolve a URI template
  - `resolveTemplate(String, Object)`
- Create an `Invocation.Builder` for a content type
  - `request(MediaType)`

javax.ws.rs.client.Invocation.Builder

`Invocation.Builder` allows the application to:

- Set the Accept header
  - `accept(MediaType...)`
- Set the Accept-Language header
  - `acceptLanguage(Locale...)`
- Update the Cookie header
  - `cookie(Cookie)`
- Set any HTTP header
  - `header(String, Object)`
- Call an HTTP method synchronously
  - `get, put, post, delete, head, trace, options`
- Call an HTTP method asynchronously
  - `async()`

Obtaining the Response Entity

All the `Invocation.Builder` synchronous HTTP methods have three overloaded versions:

- Obtain the response entity by class type:

```
Person p = builder.get(Person.class);
```

- Obtain a generic response entity with `GenericType`:

```
List<User> list =
 builder.get(new GenericType<List<User>>(){});
```

- Obtain a response, analyze it, and then read the entity:

```
Response response = builder.get();
//analyze Response status or headers
Person p = response.readEntity(Person.class);
```

Sending the Request Entity

The `Invocation.Builder` synchronous HTTP `put` and `post` methods send an entity as the first argument.

```
Response msgResponse = target
        .path("message")
        .request()
        .put(Entity.text("Hello"));
```

An `Entity` object represents a message entity and associated variant information, such as the Content-Type.

Obtaining Reply Metadata

`Response` represents the complete reply message received by the client. Its API allows the application to access:

- Status code
- Message payload
- Response Headers
- Response Cookies

```
int status = msgResponse.getStatus();
Map<String,NewCookie> cookies =
  msgResponse.getCookies();
String power =
  msgResponse.getHeaderString("X-Powered-By");
Date date = msgResponse.getDate();
```

Web Service Errors

Web Services can experience errors in two places:

- On the server
  - In your web service an exception is thrown. How you convey that to a client depends on the type of web service (SOAP or REST).
- On the client
  - Clients receive the errors produced by a web service.
  - Clients experience error without there being any error produced by a server (networking problems, for example).

JAVA API FOR RESTFUL WEB SERVICES (JAX-RS)

javax.ws.rs.WebApplicationException

When not obtaining a `Response`, a `WebApplicationException` subclass is thrown. There are three direct sublcasses:

- `RedirectionException` – HTTP 3XX status codes
- `ClientErrorException` – HTTP 4XX status codes
- `ServerErrorException` – HTTP 5XX status codes

A more specific subclass will be thrown.

```
String stringResult = target
        .path("invalidpath")
        .request(MediaType.TEXT
        .get(String.class);
```

A `javax.ws.rs.NotFoundException` is thrown. `WebApplicationException` is a subclass of `RuntimeException`.

Reading `Response` Error Status

When getting a `Response`, no exception is thrown.

```
int status = msgResponse.getStatus();
if(status >= 400 && status < 500) {
    System.out.println("Client error");
} else if(status >= 500) {
    System.out.println("Server error");
} else {
    System.out.println("STATUS: " + status);
}
```

Alternative Java REST Clients

Although the JAX-RS 2 Client API (Java EE 7) is not available to you, you can still access RESTful web services in a number of ways:

- Jersey 1.X Client API
- `java.net.URL` and `java.net.HttpURLConnection`
- Third-party clients such as Apache's HttpClient

Jersey 1.X Client API

JAX-RS 1.1 does not contain a client API. The JAX-RS 1.1 reference implementation (Jersey 1.X) provides a client API.

JAX-RS 2 Client	Jersey 1 Client
ClientBuilder	com.sun.jersey.api.client.Client
Client	com.sun.jersey.api.client.Client
WebTarget	com.sun.jersey.api.client.WebResource
Response	com.sun.jersey.api.client.ClientResponse

JAVA API FOR RESTFUL WEB SERVICES (JAX-RS)

java.net.URL Client

```
1  public class SimplestClient {
2   static public void main( String[] args )
3          throws Exception {
4    String contextURL = "http://localhost:8080/jaxrs";
5    String resourcePath = "/airports";
6    String requestPath = "/numAirports";
7    String urlString =
8      contextURL + resourcePath + requestPath;
9    URL url = new URL( urlString );
10   InputStream result = (InputStream) url.getContent();
11   Scanner scanner = new Scanner( result );
12   System.out.println( "Result:_" + scanner.next() );
13  }
14 }
```

java.net.HttpURLConnection Client

```
1  public class FormParamClient {
2   static public void main( String[] args )
3         throws Exception {
4     String contextURL = "http://localhost:8080/jaxrs";
5     String resourcePath = "/airports";
6     String requestPath = "/add";
7     String code = "LGA";         // need URL-encoding
8     String name = "LaGuardia"; // need URL-encoding
9     String urlString =
10          contextURL + resourcePath + requestPath;
11    URL url = new URL( urlString );
12    HttpURLConnection connection =
13          (HttpURLConnection) url.openConnection();
```

URLConnection subclasses provide more control.

Drawbacks of the Simple Approach

- Requires explicit matching of URL rewrite rules. To avoid invalid URLs, parameters may need to be provided using URL-encoding.
- Requires some awareness of the structure of HTTP messages
- Requires low-level I/O programming

java.net.HttpURLConnection Client

```
14    connection.setRequestMethod( "POST" );
15    connection.setAllowUserInteraction( true );
16    connection.setDoOutput( true );
17    connection.setDoInput( true );
18    connection.connect();
19    OutputStream os = connection.getOutputStream();
20    PrintWriter writer = new PrintWriter( os );
21    writer.print( "code=" + code + "&name=" + name );
22    writer.close();
23    InputStream result = connection.getInputStream();
24    BufferedReader reader =
25     new BufferedReader( new InputStreamReader(result) );
26    System.out.println( "Result: " + reader.readLine() );
27   }
28 }
```

HttpUrlConnection Response Status

Use the `HttpUrlConnection.getResponseCode()` method to get the HTTP response status:

```
URL url = new
URL("http://localhost:7001/app/resources/root");
HttpURLConnection conn =
    (HttpURLConnection) url.openConnection();
conn.connect();

switch(conn.getResponseCode()) {
  case 404:
  case HttpURLConnection.HTTP_BAD_REQUEST:
  default:
}
```

JavaScript Object Notation (JSON)

JSON

- JavaScript Object Notation (JSON) is a simple format to represent objects, arrays, and values as strings.
- JavaScript includes functions to convert objects, arrays, and values to JSON strings and vice versa.

http://json.org/

JavaScript Object Notation (JSON)

JSON is a format to represent JavaScript objects, arrays, and values as strings. It is one of the preferred ways of sending and receiving data from HTML applications.

JSON enables you to represent objects as key-pair values and arrays as sequential lists of items.

Values can also be represented as JSON, allowing you to create all sorts of objects by using strings, numbers, and booleans.

JSON is often compared to XML because both can be used to represent structured data. The main difference between JSON and XML is that JSON contains only data without any schema information. Therefore, JSON has no direct validation mechanism.

Also JSON contains only string, number, and boolean value types and Object and Array data structures. You do not need to define custom data types or nodes as in XML. In JSON, data is represented as it would be inside a JavaScript object, making it extremely flexible at the exchange of validation and schema facilities.

JavaScript Object Notation (JSON)

JSON Strings

Arrays:

```
[1,2,3]
["a","b","c"]
```

Objects:

```
{"name":"john","age":31}
{"itemId":["a","b"], "total":2, "active":true}
```

- JSON is very similar to declaring JavaScript literal values.
- JSON does not have a representation for functions.
- Only values are allowed in JSON.

JavaScript Object Notation (JSON)

JSON values are represented in the following manner:

- **Numbers:** Use the number literal.
- **Strings:** Enclose the string literal inside double quotation marks.
- **Booleans:** These are either true or false.

JSON structures are represented in the following manner:

- **Arrays:** Must be enclosed in brackets "[]" and elements must be comma-separated. They can contain arrays, objects, numbers, strings, or boolean as elements.
- **Objects:** Must be enclosed in braces "{}." Properties are comma-separated key-value pairs. The key for the property must be a string; therefore, it must be enclosed in double quotation marks. Values for properties can be arrays, objects, numbers, strings, or boolean.

There is no way to represent functions in JSON because it is used to represent only data.

JavaScript Object Notation (JSON)

Java Support for JSON

- JavaScript Object Notation (JSON) is commonly used by RESTful web services and WebSocket applications.
- Just as with XML parsing, there are several ways to parse and generate JSON from within Java applications.
  - The Java API for JSON Processing (JSON-P) JSR 353:
    - Provides APIs similar to StAX and DOM for JSON
    - Is the only standard JSON API so far
  - JSON to Java Object Binding
    - EclipseLink MOXy supports binding POJO and JAXB-annotated objects to JSON.

JavaScript Object Notation (JSON)

Third-party Java JSON libraries, such as Gson, have been around for years. However, JSON APIs are now starting to become the official components of the Java platform by using the JEP/JSR process. JSON-P was released as part of Java EE 7 and several other JSON standards are under development:

- **JSR 367: Java API for JSON Binding (JSON-B):** This API will standardize JSON to Java binding similar to what JAXB does for XML. It is currently scheduled for release as a Java EE 8 component. For more, see https://jcp.org/en/jsr/detail?id=367.
- **JEP 198: Light-Weight JSON API:** This API provides a lightweight JSON library under `java.util` and is under consideration for Java SE 9. For more, see http://openjdk.java.net/jeps/198.

JavaScript Object Notation (JSON)

Java API for JSON Processing (JSON-P)

The following are the primary JSON-P classes.

- Streaming JSON classes:
  - `JsonParser` – A pull parser for reading JSON data
  - `JsonGenerator` – A JSON generator that uses method chaining
- Object-based JSON classes:
  - `JsonReader` – Reads from an `InputStream` and produces an object graph
  - `JsonWriter` – Writes a JSON-P–specific object graph to an `OutputStream`

JavaScript Object Notation (JSON)

The streaming classes parse or generate JSON as the data streams through your application eliminating the need to have all of the JSON data in memory as objects.

The object-based JSON reader or writer classes use JSON-P class types to represent JSON constructs.

- `JsonStructure` – The common JSON object supertype
- `JsonObject` – A JSON object that uses string keys to uniquely identify values
- `JsonArray` – A JSON object that uses `int` index values to uniquely identify values

JavaScript Object Notation (JSON)

JsonGenerator

```
JsonGenerator jgen = Json.createGenerator(System.out);
jgen.writeStartObject()
        .writeStartArray("persons")
            .writeStartObject()
                .write("firstName", "Sherlock")
                .write("lastName", "Holmes")
                .writeStartObject("address")
                    .write("street", "221b Baker Street")
                    .write("city", "London")
                    .write("country", "England")
                .writeEnd()
            .writeEnd()
        .writeEnd()
.writeEnd();
jgen.flush();
```

JavaScript Object Notation (JSON)

JsonParser

```
JsonParser jp =
Json.createParser(new FileInputStream("data.json"));
while(jp.hasNext()) {
    Event e = jp.next();
    switch(e) {
        case START_OBJECT:
        case END_OBJECT:
        case START_ARRAY:
        case END_ARRAY:
          break;
        case KEY_NAME:
        case VALUE_STRING:
          System.out.print(jp.getString());
    }
}
```

JavaScript Object Notation (JSON)

JsonWriter

```
JsonWriter jw = Json.createWriter(System.out);
JsonObject address = Json.createObjectBuilder()
        .add("street", "221b Baker Street")
        .build();
JsonObject sherlock = Json.createObjectBuilder()
        .add("firstName", "Sherlock")
        .add("address", address)
        .build();
JsonArray persons = Json.createArrayBuilder()
        .add(sherlock)
        .build();
JsonObject root = Json.createObjectBuilder()
        .add("persons", persons)
        .build();
jw.write(root);
```

JavaScript Object Notation (JSON)

JsonReader

```
JsonReader jr =
Json.createReader(new FileInputStream("data.json"));
JsonObject root = jr.readObject();
JsonArray persons = root.getJsonArray("persons");
if(persons != null) {
  for (JsonValue value : persons) {
    if (value.getValueType() ==
        JsonValue.ValueType.OBJECT) {
      JsonObject person = (JsonObject) value;
      String n = person.getString("firstName");
      JsonObject address =
        person.getJsonObject("address");
      if (address != null) {
        String s = address.getString("street");
}}}}
```

JavaScript Object Notation (JSON)

JAX-RS Support for JSON-P

When using JAX-RS 2 on a platform that supports JSON-P, you can leverage JSON-P to provide portable JSON support for your RESTful web services.

- Resource methods can receive or return one of the JSON object types:
  - `JsonStructure`, `JsonObject` and `JsonArray`
- Resource methods can receive and return a `java.io.InputStream`. Typically, you would receive an `InputStream` and then use a `JsonReader` in the method body to read from the `InputStream`.
- Resource methods can return a `javax.ws.rs.core.StreamingOutput`. This could be used with a `JsonWriter` or `JsonGenerator`.

JavaScript Object Notation (JSON)

Using JSON-P with JAX-RS is currently the only standards-based approach to produce and consume JSON. Many JAX-RS implementations, such as Jersey, support JSON-POJO and JSON-JAXB binding; however, support for these types of bindings is not yet standardized. You might have to add lines of code to your `Application` subclass to enable the feature- or place-specific classes on your CLASSPATH. If you are using JSON with JAX-RS and you are not using JSON-P, then you must reference the documentation for your JAX-RS implementation to understand the JSON features available to you.

JavaScript Object Notation (JSON)

JAX-RS `StreamingOutput`

The `javax.ws.rs.core.StreamingOutput` interface type can be used as a return type for JAX-RS resource methods. It enables you to stream the response body as it is created. You can use `StreamingOutput` to stream large JSON responses.

```
StreamingOutput stream = new StreamingOutput() {
  @Override
  public void write(OutputStream os) throws IOException,
    WebApplicationException {
     //JsonGenerator jgen = Json.createGenerator(os);
  }
};
return Response.ok(stream).build();
```

KONIEC

DZIĘKUJĘ ZA UWAGĘ!!!