# RESTFUL WEB SERVICES

# Widely Well-known "Words"

- **Internet**
  - Massive network of networks, connecting millions of computers together globally.
  - Information communication using protocols like HTTP, SMTP, FTP etc

- **World Wide Web**, or simply **Web**
  - A way of accessing information over the Internet using HTTP.

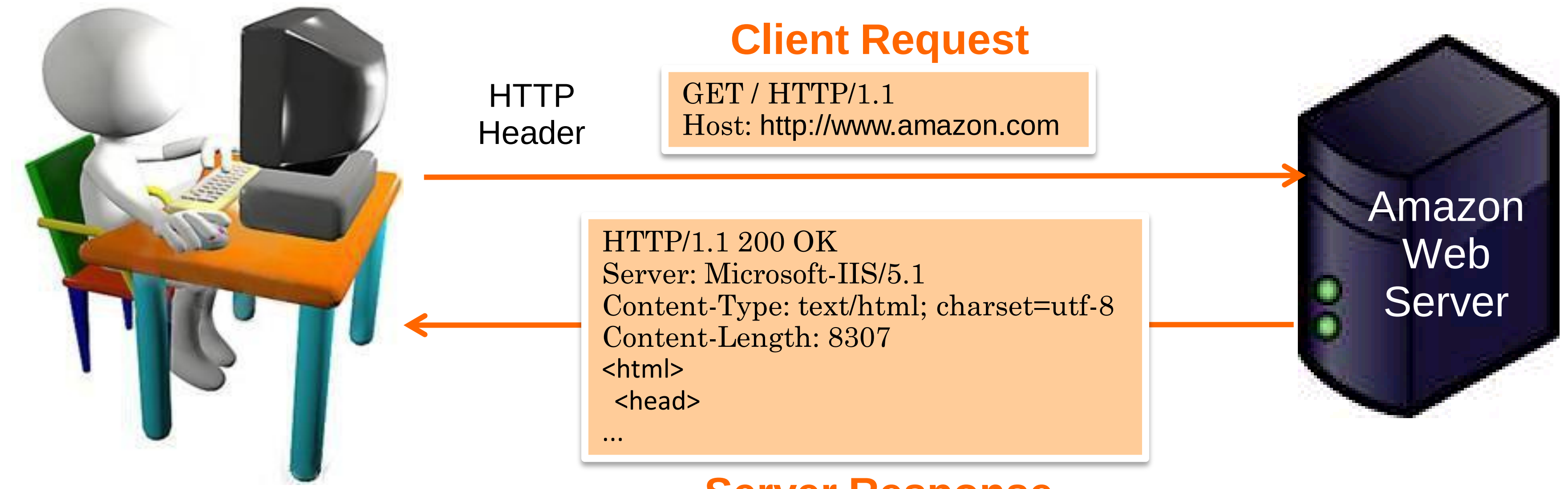
# World Wide Web

- **The Web** as defined by Tim Berners-Lee consists of three elements:
  - **URI** (Uniform Resource Identifier) - The way of uniquely identifying resources on the network.
  - **HTML** (HyperText Markup Language) - The content format of resources to be returned.
  - **HTTP** (HyperText Transfer Protocol) - The protocol used to request a resource from the network and respond to requests.

- **HTTP methods**

  All client server communication on the World Wide Web are done using the following simple HTTP methods:
  - **GET** = "give me some info" (Retrieve)
  - **POST** = "here's some info to update" (Update)
  - **PUT** = "here's some new info" (Create)
  - **DELETE** = "delete some info" (Delete)

# Retrieving Information using HTTP GET

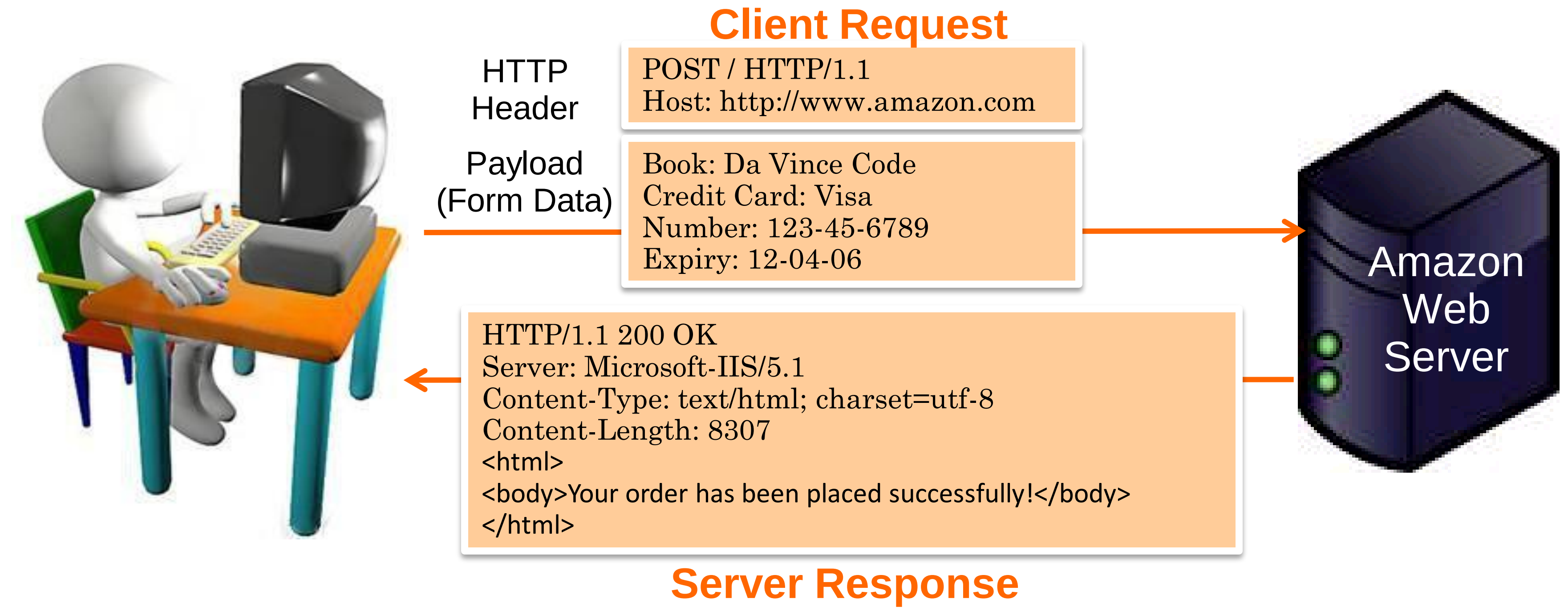


```
GET / HTTP/1.1
Host: http://www.amazon.com
```

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/5.1
Content-Type: text/html; charset=utf-8
Content-Length: 8307
<html>
 <head>
...
```

- The user types http://www.amazon.com in his browser.
- The browser software creates and sends a HTTP request with a header that holds:
  - The desired action: GET ("get me resource")
  - The target machine (www.amazon.com)
- The server responds with the requested resource which is rendered on the browser

# Updating Information using HTTP POST



```
POST / HTTP/1.1
Host: http://www.amazon.com
```

```
Book: Da Vince Code
Credit Card: Visa
Number: 123-45-6789
Expiry: 12-04-06
```

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/5.1
Content-Type: text/html; charset=utf-8
Content-Length: 8307
<html>
<body>Your order has been placed successfully!</body>
</html>
```

- The user fills in a form on the Web page and submits it.
- The browser software creates and send a HTTP request with a header and a payload comprising of the form data.
  - The HTTP header identifies:
    - The desired action: POST
    - The target machine (amazon.com)
  - The payload contains:
    - The data being POSTed (the form data)
- The server responds with the requested resource which is rendered on the browser

## Widely Well-known "Words" (Contd..)

- **Web Application**
  - Usually a collection of *dynamic* web pages
  - Usually restricted to the *intranet*
  - Can be implemented as desktop application
  - Information accessible using front end user interfaces
  - Accessed by *authorised users* only
- **Web Site**
  - Collection of *static and dynamic* web pages
  - Available on the *internet*, or an organization's *intranet*
  - Cannot be implemented as desktop application
  - Information accessible using front end user interfaces
  - Accessed by *anybody*

# Widely Well-known “Words” (Contd..)

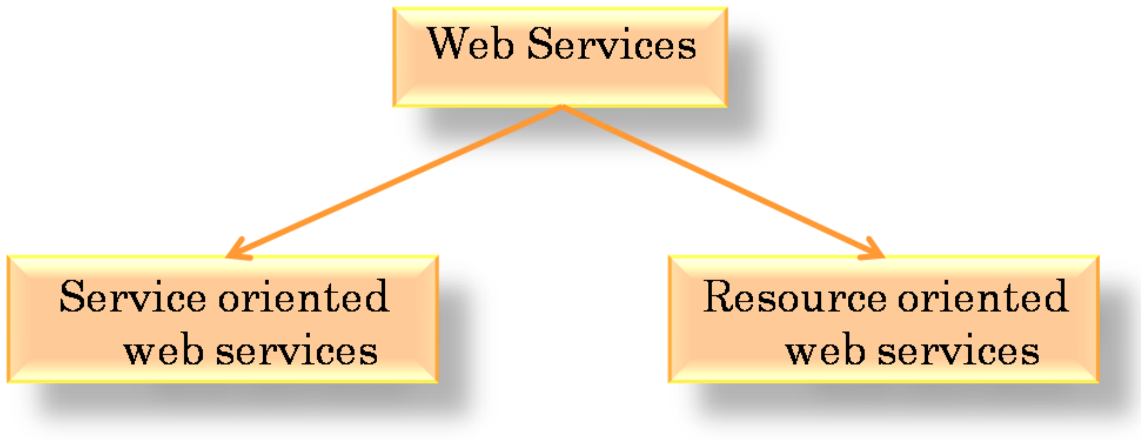
- **Web Service**
  - Application run by a web server, performing tasks and returning structured data to a calling program, rather than html for a browser.
  - Only “provides” information; does not “present” information
  - Publicly available and standardized for use by all programmers
- **Web Server**
  - Software designed to serve web pages/web sites/web services. Examples are IIS, Apache, etc.

# Web Services

- Services (usually some combination of program and data) that are made available from a Web server for access by Web users or other Web-connected programs.

- Specific business functionality exposed by a company, usually through an Internet connection, for the purpose of providing a way for another company or software program to use the service.

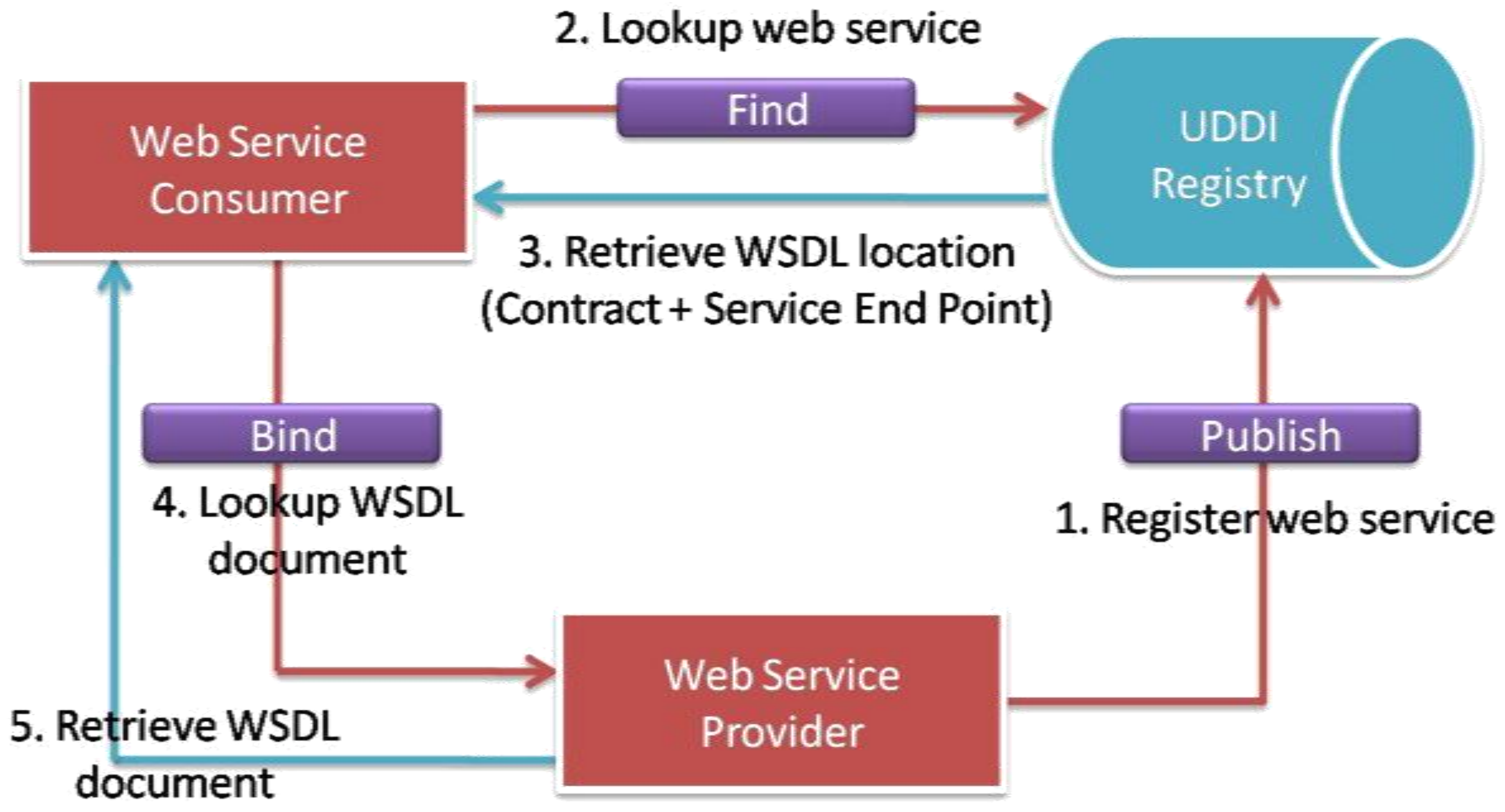
- Types of Web Services:



# Types of Web services

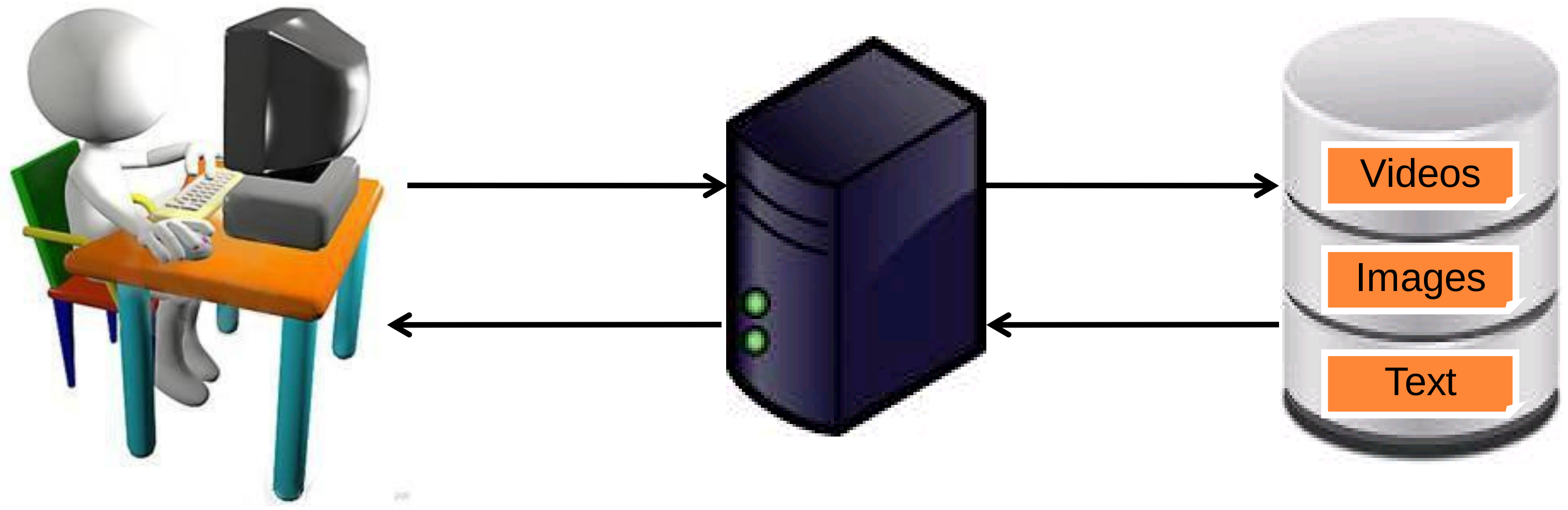
- **Service-Oriented Web Services**
  - Based on “services”
  - One service offers multiple functionalities
  - “Big” Web Services
  - JAX-WS = JAVA-API for XML-based Web Services, mainly using WSDL/SOAP
- **Resource-Oriented Web Services**
  - Based on “resources”
  - Resource - any directly accessible and distinguishable distributed component available on the network.
  - RESTful Web Services
  - JAX-RS = JAVA-API for RESTful Web Services, using only HTTP

# Service Oriented Web Services - Architecture



- ➢ **SOAP (Simple Object Access Protocol)**
- ➢ **WSDL (Web Services Definition Language)**
- ➢ **UDDI (Universal Discovery, Description and Integration)**

# Resource Oriented Web Services - Architecture

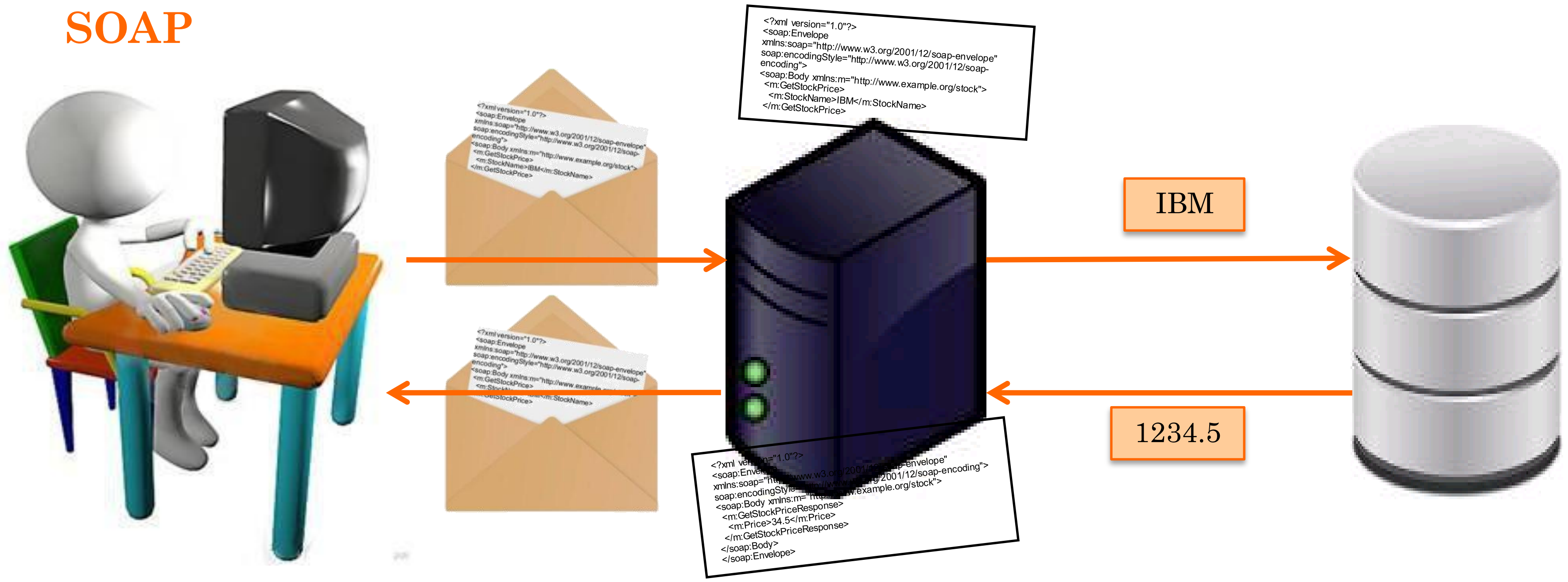


- **Resources**
  - Every distinguishable entity is a resource.
  - A resource may be a Web site, an HTML page, an XML document, a Web service, an image, a video *etc*.
- **URIs** - Every resource is uniquely identified by a URI.
- **Resource lifecycle management** using HTTP methods

| **CRUD** | **HTTP Method** |
|---|---|
| Create | PUT or POST |
| Read | GET, HEAD or OPTIONS |
| Update | PUT |
| Delete | DELETE |

# SOAP Web Service Vs RESTful Web Service

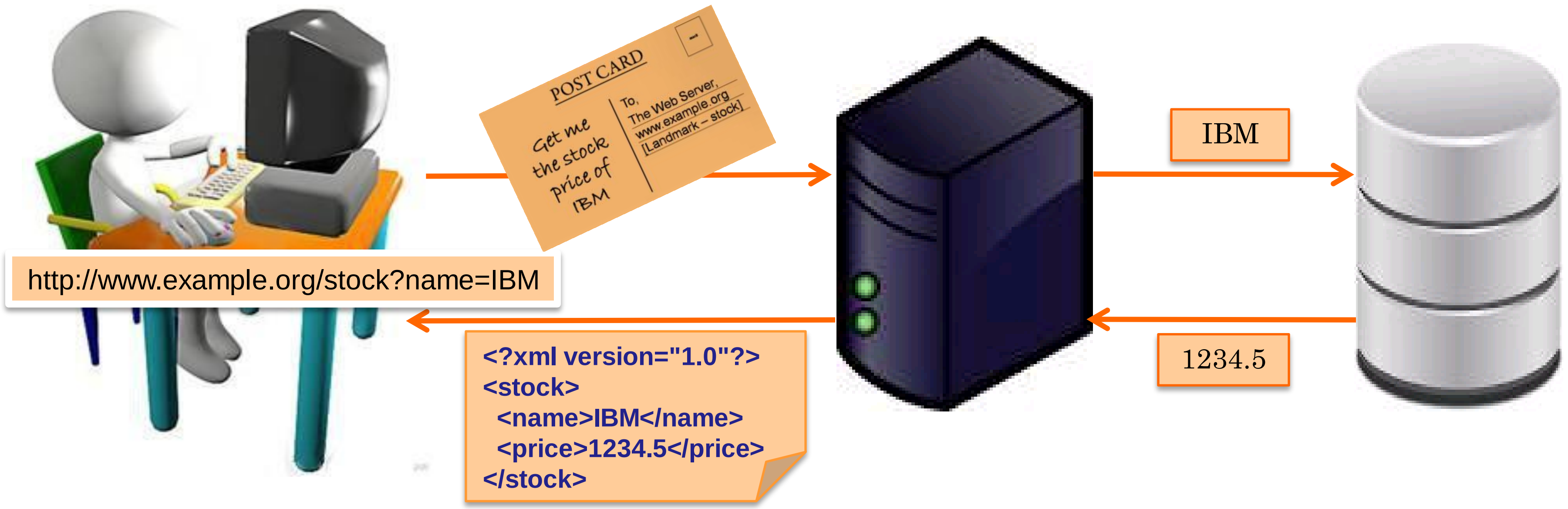
## SOAP



| SOAP Request | SOAP Response |
|---|---|
| <?xml version="1.0"?><br><soap:Envelope<br>xmlns:soap="http://www.w3.org/2001/12/soap-envelope"<br>soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding"><br>  <soap:Body<br>xmlns:m="http://www.example.org/stock"><br>      <m:GetStockPrice><br>         <m:StockName>IBM</m:StockName><br>     </m:GetStockPrice><br>  </soap:Body><br></soap:Envelope> | <?xml version="1.0"?><br><soap:Envelope<br>xmlns:soap="http://www.w3.org/2001/12/soap-envelope"<br>soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding"><br>  <soap:Body<br>xmlns:m="http://www.example.org/stock"><br>      <m:GetStockPriceResponse><br>          <m:Price>1234.5</m:Price><br>      </m:GetStockPriceResponse><br>  </soap:Body><br></soap:Envelope> |

# SOAP Web Service Vs RESTful Web Service

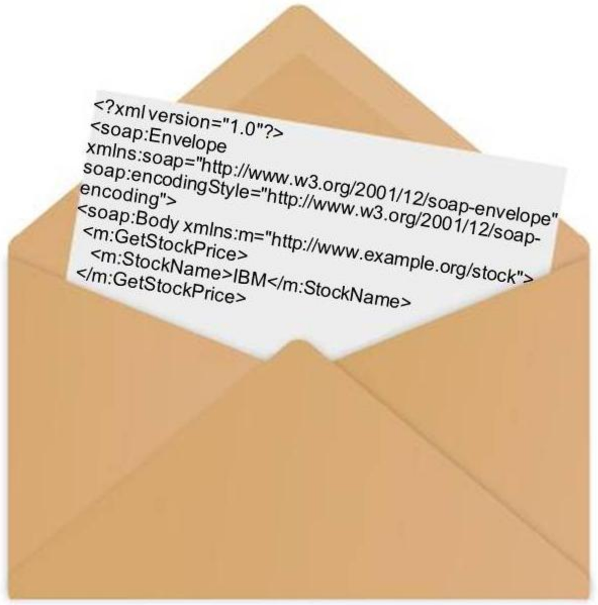
## REST



```
<?xml version="1.0"?>
<stock>
  <name>IBM</name>
  <price>1234.5</price>
</stock>
```

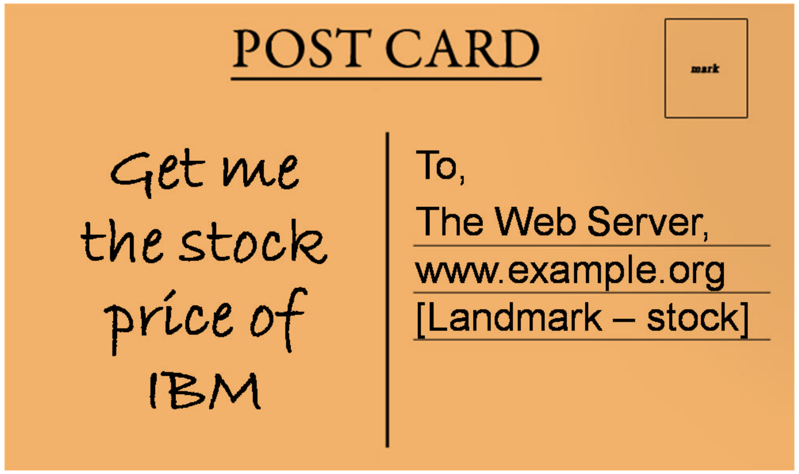
| REST – HTTP Request | REST – XML Response |
|---|---|
| http://www.example.org/stock?name=IBM | <?xml version="1.0"?> <stock> <name>IBM</name> <price>1234.5</price> </stock> |

# SOAP Vs REST



```
<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
<soap:Body xmlns:m="http://www.example.org/stock">
 <m:GetStockPrice>
  <m:StockName>IBM</m:StockName>
</m:GetStockPrice>
```

Vs

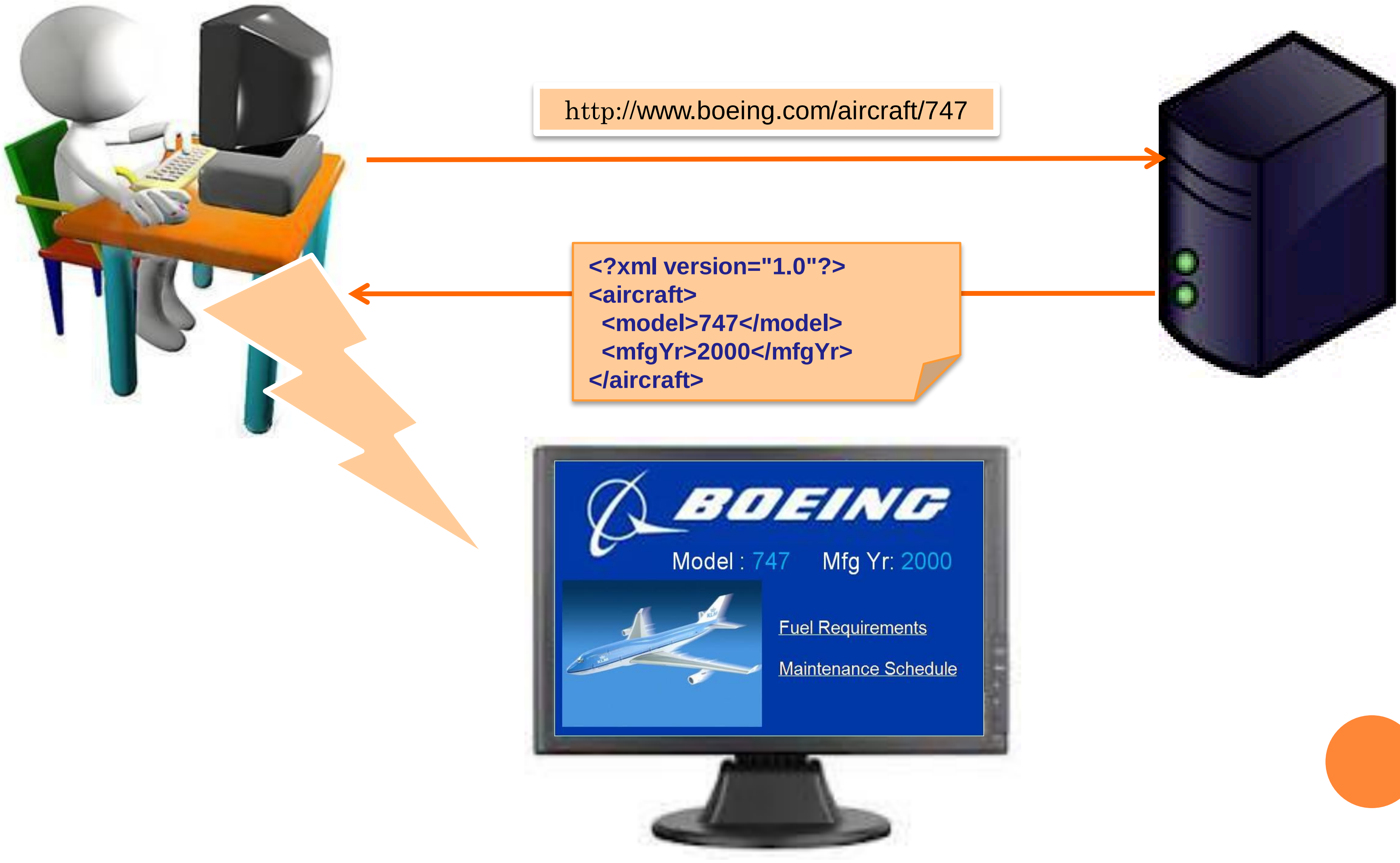


- **SOAP based web services**
  - Verbose => heavy payload
  - Suitable for enterprise web services where you need interoperability, transactions, message delivery and reliability.
- **RESTful web services**
  - Not verbose => needs less bandwidth
  - Good to use for the mobile applications.

# REST

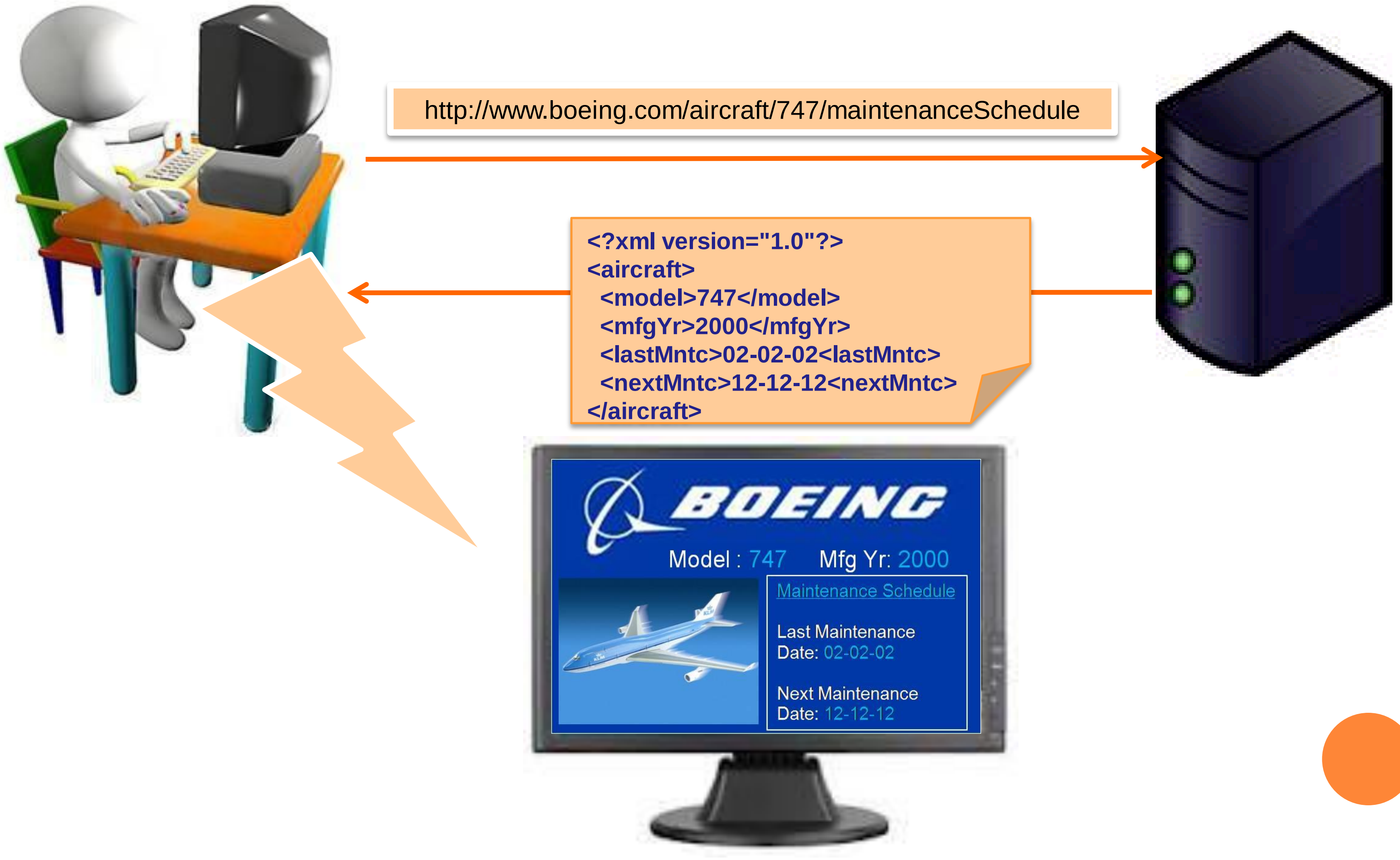
- REST = REpresentational State Transfer
- Architectural style in which clients and servers exchange representations of resources by using a standardized interface and protocol.
- Principles of REST:
  - Addressability (URI)
  - Interface Uniformity (HTTP)
  - Statelessness (HTTP)
  - Connectedness (Hypermedia)

# Why is it called "Representational State Transfer?"



```
<?xml version="1.0"?>
<aircraft>
  <model>747</model>
  <mfgYr>2000</mfgYr>
</aircraft>
```

## Why is it called "Representational State Transfer?"



```
<?xml version="1.0"?>
<aircraft>
  <model>747</model>
  <mfgYr>2000</mfgYr>
  <lastMntc>02-02-02<lastMntc>
  <nextMntc>12-12-12<nextMntc>
</aircraft>
```

## Why is it called "Representational State Transfer?"

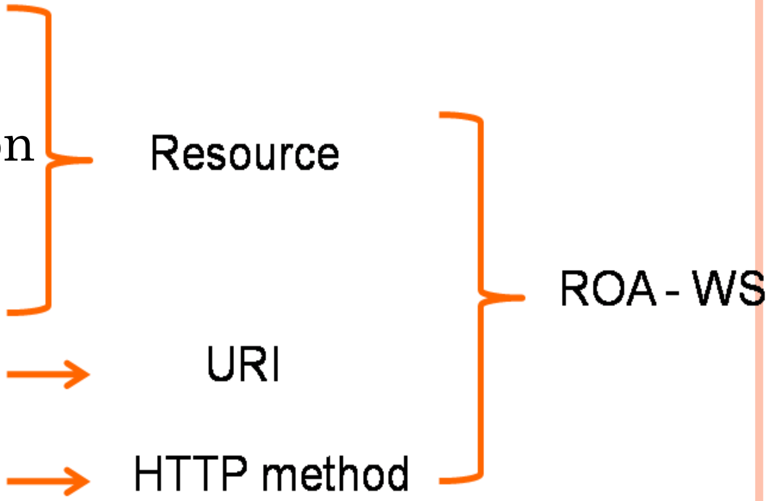
1. The Client references a Web **resource using a URL.**
2. A **representation of the resource is returned.**
3. The representation (*e.g., Boeing747.html) places the client in a new **state.***
4. When the client selects a hyperlink in Boeing747.html, it accesses another resource.
5. The new representation places the client application into yet another state.
6. Thus, the client application **transfers state with each resource representation.**

Roy T. Fielding http://roy.gbiv.com

# Building a Web Service

# Building a Web Service

- Building blocks of a web service:
  - Input(s)
  - Data retrieval and processing function
  - Output(s)
  - Access path
  - Access method



# RESTful Web Services

- Web Services (data, functionality on server side) implemented using HTTP + REST principles

- Key elements of RESTful Web service are:
  - The URI (path) of the Web Service
  - The HTTP method supported by the web service.
  - The MIME type of the request and response data supported by it.

StockCalculator.java
+ double getStockPrice(String company)
+ List<StockDTO> getStockHistory(...)

## JAX-RS

- Java API for RESTful Web Services
- Maintained through Java Specification Request – JSR311
- Has a set of annotations and associated classes and interfaces which simplify development of RESTful Web Services.
- Supports multiple data formats (JSON / XML / HTML / TEXT)

## JAX-RS Implementations

- Jersey
- Restlet
- JBoss RESTEasy
- Apache CXF
- Triaxrs
- Apache Wink
- eXo

# Important JAX-RS Annotations

| Aspect | Annotation | Scope | Description |
|---|---|---|---|
| URI | @PATH (resource_pa th) | Class, Method | Sets the path to base URL + /resource_path. The base URL is based on your application name, the servlet and the URL pattern from the web.xml" configuration file. |
| Resource Method Designators<br><br>Rules:<br>1) Only one JAX-RS method designation annotation is allowed per method in a Java class resource.<br>2) Only public methods may be exposed as resource methods | @POST | Method | Indicates that the method annotated with it will answer to a HTTP POST request |
| | @GET | Method | Indicates that the method annotated with it will answer to a HTTP GET request |
| | @PUT | Method | Indicates that the method annotated with it will answer to a HTTP PUT request |
| | @DELETE | Method | Indicates that the method annotated with it will answer to a HTTP DELETE request |
| MIME type | @Produces( MediaType [, more-types ]) | Class, Method | @Produces defines which MIME type is returned by the resource. |
| | @Consumes( MediaType [, more-types ]) | Class, Method | @Consumes defines which MIME type is consumed by this resource. |

# Configuring Jersey

1. Include the following Jar files in the web project's library:
   jersey-core.jar, jersey-server.jar, jsr311-api.jar, asm.jar and jersey-client.jar

2. Register Jersey as the servlet dispatcher for REST requests in the project's web.xml.

```
<servlet>
    <servlet-name>ServletAdaptor</servlet-name>
    <servlet-class> com.sun.jersey.spi.container.servlet.ServletContainer
    </servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>ServletAdaptor</servlet-name>
    <url-pattern>/rs/*</url-pattern>
</servlet-mapping>
```

## RESTful - Resources

- With JAX-RS,

  Annotated POJOs = RESTful Web Services a.k.a Resources.

- Root Resource = POJO (Plain Old Java Object) annotated with @Path.
- Sub Resources = Methods within the Root POJO Resource

# Sample RESTful Web Service

```
package com.example;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.POST;
import javax.ws.rs.QueryParam;
import javax.ws.rs.core.MultivaluedMap;

@Path("/customers")
public class Customers {

        // Get all customer details
        @GET
        public String getAllCustomers() {
                return "list of customers";
        }

        // Manage customer details
        @POST
        public void manageCustomers(@QueryParam("method") String method, MultivaluedMap<String,
        String> formParams) {
                if (method.equals("create")) {
                        //create new customer
                } else if (method.equals("update")) {
                        // update an existing customer
                } else if (method.equals("delete")) {
                        // delete a customer
                }
        }
}
```

**Client Request**

```
GET / HTTP/1.1
Host:
http://localhost:8081/TestRestfulService/rs/
customers
```

**Client Request**

```
POST / HTTP/1.1
Host:
http://localhost:8081/TestRestfulService/rs/
customers?method=create
```

```
Id: 12345
Name: John
```

# Accessing resources using @Path

## ❖ Root Resource

```
import javax.ws.rs.Path;

@Path("/employeeinfo")
public class EmployeeInfo {
   .......
}
```

## ❖ Root Resource Path

Syntax:

*http://your_domain:port/display-name/url-pattern/path_from_rest_class*

Example:

*http://localhost:8081/TestRestfulService/rs/employeeInfo*

# Accessing resources using @Path

- An @Path value is not required to have leading or trailing slashes (/)

  @Path(”/product/") = @Path(”/product”) = @Path("product”) = @Path(”product/”)

- Automatic encoding

  @Path("product list") = @Path("product%20list”)

- URL Pattern and path template

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

# RESTful – Resources (Optional @Path sample)

```
package com.example;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.POST;
import javax.ws.rs.FormParam;

@Path("/customers")
public class Customers {


        // Get all customer details
        @GET
        public String getAllCustomers() {
                return "list of customers";
        }



        // Create a customer
        @POST
        public void createCustomer(
                @FormParam("Id") int id,
                @FormParam("Name") String name) {
                        //create new customer
        }
}
```

**Client Request**

```
GET / HTTP/1.1
Host:
http://localhost:8081/TestRestfulService/rs/
customers
```

**Client Request**

```
POST / HTTP/1.1
Host:
http://localhost:8081/TestRestfulService/rs/
customers
```

```
Id: 12345
Name: John
```

# RESTful–Resources(Mandatory @Path sample)

```
package com.example;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.POST;
import javax.ws.rs.PathParam;

@Path("/customers")
public class Customers {

        // Get all customer details
        @GET
        public String getAllCustomers() {
                return "list of customers";
        }



        // Get specific customer details
        @GET
        @Path("{id}")
        public String getCustomer(@PathParam("id”)
                String id) {
                return "particular customer";
        }

}
```

**Client Request**

```
GET / HTTP/1.1
Host:
http://localhost:8081/TestRestfulService/rs/
customers
```

**Client Request**

```
GET / HTTP/1.1
Host:
http://localhost:8081/TestRestfulService/rs/
customers/1234
```

# Hierarchical matching using @Path

```
package com.example;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import com.thoughtworks.xstream.XStream;

@Path("/customers")
public class Customers{

        @GET
        public String getAllCustomers() {
                return "list of customers";
        }

        @GET
        @Path("/{id}")
        public String getCustomer(@PathParam("id") int id) {
                XStream xstream = new XStream();
                Customer customer = new Customer(id);
                return xstream.toXml(customer);
        }

        @GET
        @Path("/{id}/address")
        public String getAddress(@PathParam("id") int id) {
                Customer customer = new Customer(id);
                return customer.getAddress();
        }
}
```

**Client Request**

```
GET / HTTP/1.1
Host:
http://localhost:8081/TestRestfulService/rs/
customers
```

**Client Request**

```
GET / HTTP/1.1
Host:
http://localhost:8081/TestRestfulService/rs/
customers/1234
```

**Client Request**

```
GET / HTTP/1.1
Host:
http://localhost:8081/TestRestfulService/rs/
customers/1234/address
```

## RESTful – Sub Resources

- **Sub Resource Methods**
  - POJO Methods annotated with a "resource method designator" annotation and with @Path annotation.
  - Handles the HTTP request directly.
  - @Path is optional for a sub resource method under the following conditions:
    - ✓ If no. of methods per HTTP action = 1, then @Path is optional
    - ✓ If no. of methods per HTTP action > 1, then all methods or all but one method should have @Path
- **Sub Resource Locators**
  - POJO Methods annotated ONLY with @Path but NOT with any "resource method designator" annotation.
  - Returns an object of the Sub Resource Class that will handle the HTTP request.

# Examples of Sub Resources

```
package com.example;

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

@Path("/employeeinfo")
public class EmployeeInfo {

        // Subresource locator: obtains the subresource Employee from
        // the path /employeeinfo/employees/{fullName}
        @Path("/employees /{fullName}")
        public Employee getEmployee(@PathParam("fullName") String fullName) {
                Employee emp = new Employee(fullName);
                return emp;
        }

        // Subresource Method
        @GET
        @Path("/employees")
        public String getAllEmployees() {
                return "List of employees from sub resource method";
        }
}
```

# Examples of Sub Resources (contd..)

```
package com.example;

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

// Subresource class
public class Employee {

        private String fullName;

        public Employee(String fullName){
                this.fullName = fullName;
        }

        // Subresource method: returns the employee's first name
        @GET
        @Path("/firstname")
        public String getEmployeeFirstName() {
             return fullName.substring(0, fullName.indexOf(" ")==-1? fullName.length(): fullName.indexOf(" "));
        }

        // Subresource method: returns the employee's last name
        @GET
        @Path("/lastname")
        public String getEmployeeLastName() {
             if(fullName.indexOf(" ")==-1){
                return "No Last Name found";
             } else {
                return fullName.substring(fullName.indexOf(" ")+1);
             }
        }
}
```

# Examples of Sub Resources (contd..)

- Request URL:

  http://localhost:8080/TestRestfulService/rs/employeeinfo/employees

  <u>Output:</u>
  List of employees from sub resource method

- Request URL:

  http://localhost:8080/TestRestfulService/rs/employeeinfo/employees/John/firstname

  <u>Output:</u>
  John

- Request URL:

  http://localhost:8080/TestRestfulService/rs/employeeinfo/employees/John Doe/lastname

  <u>Output:</u>
  Doe

- Request URL:

  http://localhost:8080/TestRestfulService/rs/employeeinfo/employees/John/lastname

  <u>Output:</u>
  No Last Name found

# Extracting Request Parameters

**Client Request**

| | |
|---|---|
| HTTP Header | POST / HTTP/1.1<br>Host: http://www.amazon.com<br>Cookie: x=56<br>User-Agent: Mozilla |
| Payload (Form Data) | Book: Da Vince Code<br>Credit Card: Visa<br>Number: 123-45-6789 |

**Path Param:**
http://localhost:8080/TestRestfulService/rs/customer/12345

**Query Param:**
http://localhost:8080/TestRestfulService/rs/employees?id=543

**Matrix Param:**
http://localhost:8080/TestRestfulService/rs/employees;role=ITA

| Annotation | Description | Data Source | Supported HTTP Method |
|---|---|---|---|
| @QueryParam | Extracts the value of a URI query parameter. | URI | GET, POST, PUT, or DELETE |
| @PathParam | Extracts the value of a URI template parameter. | URI | GET, POST, PUT, or DELETE |
| @MatrixParam | Extracts the value of a URI matrix parameter. | URI | GET, POST, PUT, or DELETE |
| @HeaderParam | Extracts the value of a HTTP header. | Header of HTTP Request | GET, POST, PUT, or DELETE |
| @CookieParam | Extracts information from cookies declared in the cookie request header. | Header of HTTP Request | GET, POST, PUT, or DELETE |
| @FormParam | Extracts information from a request representation whose content type is application/x-www-form-urlencoded. | Payload of HTTP Request | Limited only to HTTP POST |

# Extracting Request Parameters - Example

| Annotation | HTTP Request & URL Sample | JAX-RS sample |
|---|---|---|
| @QueryParam | URL with query params:<br>http://*<host_name>*:*<port>*/*<context_root>*/*<servlet_path>*/MyService/URL?x=56<br><br>GET /MyService/URL?x=56 HTTP/1.1 | public void foo(**@QueryParam("x")** int numberX) |
| @PathParam | http://*<host_name>*:*<port>*/*<context_root>*/*<servlet_path>*/MyService/URLPattern/56<br><br>GET /MyService/URLPattern/56 HTTP/1.1 | **@Path("URLPattern/{x}")**<br>public void foo(**@PathParam("x")** int numberX) |
| @MatrixParam | URL with matrix params:<br>http://*<host_name>*:*<port>*/*<context_root>*/*<servlet_path>*/MyService/URL;x=56<br><br>GET /MyService/URL;x=56 HTTP/1.1 | public void foo(**@MatrixParam("x")** int numberX) |
| @HeaderParam | GET /MyService/URL HTTP/1.1<br>x: 56 | public void foo(**@HeaderParam("x")** int numberX) |
| @CookieParam | GET /MyService/URL HTTP/1.1<br>Cookie: x=56 | public void foo(**@CookieParam("x")** int numberX) |

# Extracting Request Parameters - Example

| Annotation | HTTP Request Sample | JAX-RS sample |
|---|---|---|
| @FormParam | The form parameters and values are encoded in the request message body like the following:<br><br>POST /MyService/URL<br>HTTP/1.1<br>x=56 | @POST<br><br>@Consumes("application/x-www-form-urlencoded")<br><br>public void post(@FormParam("x") int numberX) {<br><br>}<br><br>OR<br><br>@POST<br><br>@Consumes("application/x-www-form-urlencoded")<br><br>public void post(MultivaluedMap<String, String> formParams) {<br><br>} |

## Extracting Context Information - @Context

- To extract ServletConfig, ServletContext, HttpServletRequest and HttpServletResponse from a Web application context
- Examples:

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

        MultivaluedMap<String, String> pathParams = ui.getPathParameters();

}


@GET
public String get(@Context HttpHeaders hh) {
        MultivaluedMap<String, String> headerParams = hh.getRequestHeaders();

        Map<String, Cookie> pathParams = hh.getCookies();

}
```

# @DefaultValue

- Any ***failure to parse an input*** will result in the parameter being given whatever is the **default value for its type**: false for boolean, zero for numbers, etc.
- This can be overridden by using @DefaultValue annotation and setting preferred default value.
- This default value will be used whenever the ***expected input is missing*** – or when it is ***present but parsing fails***.
- The default value should be ***given as a String***. It will be parsed to the appropriate type of the method parameter.
- Example:

```
public void foo(@DefaultValue("123") @QueryParam("id") int id)

http://localhost:8081/TestRestfulService/rs/customer?id=56
⇒id = 56

http://localhost:8081/TestRestfulService/rs/customer
⇒id = 123

http://localhost:8081/TestRestfulService/rs/customer?id=ABC
⇒id = 123
```

# Request Parameter – Data Types

- 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.
- 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.

## Entity Parameters

- A JAX-RS service method can define any number of parameters.
- All, or all but one, of those parameters must be annotated to inform JAX-RS as to how to provide a value for it.
- The one not annotated, if present, is known as an entity, and is implicitly bound to the request body. In other words, **a *non-annotated parameter extracted from the request body*** is known as an entity.
- The JAX-RS specification does not permit more than one entity parameter per method.

# Entity Provider

- JAX-RS maps Java types to and from resource representations using entity providers.
- JAX-RS entity providers help in the mapping between entities and associated Java types.
- The two types of entity providers supported by JAX-RS are:
  - MessageBodyReader: a class that is used to map an HTTP request entity body to method parameters
  - MessageBodyWriter: a class that is used to map the return value to the HTTP response entity body.
- If a String value is used as the request entity parameter, the MessageBodyReader entity provider deserializes the request body into a new String.
- If a JAXB type is used as the return type on a resource method, the MessageBodyWriter serializes the Java Architecture for XML Binding (JAXB) object into the response body.

# Entity Provider - Example

- For a resource method to return XML content, return an instance of a JAXB class directly or return a javax.ws.rs.core.Response object with a JAXB object as the response entity.

- Suppose that BookList is a JAXB class; for example:

```
@GET
@Produces("application/xml", "text/xml")
public BookList getBookList() {
    BookList list = /* get a book list */
    return list;
}
```

Or

```
@GET
@Produces("application/xml", "text/xml")
public javax.ws.rs.core.Response getBookList() {
    BookList list = /* get a book list */
    return Response.ok(list).build();
}
```

# JAX-RS Method Return Types

- void, resulting in an HTTP 204 (no content) response
- String
- Response, a JAX-RS class that allows the programmer to provide response content and other metadata, including HTTP headers
- GenericEntity, another JAX-RS type whose job it is to represent type-parameter information for a generic entity type (think List<MyClass>) at runtime
- A valid entity type – that is to say, any other Java class will be perceived as an entity by JAX-RS and converted by the same mechanism used for entity parameters
- A sub-resource method may return any of the following types – these then have entity providers pre-registered for them:
  - byte[]
  - java.io.InputStream
  - java.io.Reader
  - java.io.File
  - javax.ws.rs.ext.StreamingOutput
  - javax.activation.DataSource

## Summary

- Web Services
- Web Service types
- REST
- JAX-RS
- JAX-RS annotations
- Jersey
- Restful Web Services
    - Root Resource
    - Sub Resources
    - Accessing resources
    - HTTP methods
    - Extracting request (input) parameters
    - Response types (output)

# Quiz

1. REST in RESTful Web Services stands for
   a. Repetitive State Transfer
   b. Representational State Transfer
   c. Representational State Transformation
   d. Representational Scope Transfer

2. Which of the following annotation indicates a method's return data type?
   a. @Accept
   b. @Produces
   c. @Consumes
   d. @POST

3. Which of the following is true about the annotation @FormParam?
   a. It can be used only for a POST Method's
   b. It can be used only for GET Method's
   c. It is used to retrieve the Query String in the URL of the HTTP Service
   d. Both B & C

# Quiz

4. How do you specify a default value for a Query Param?
   a. It is not possible to specify a default value for a Query param. It’s always null.
   b. @QueryParam("version") String version = “1”
   c. @QueryParam("version") int version @DefaultValue("1")
   d. @DefaultValue("1") @QueryParam("version") String version

5.
```
@XXX
@Path("/update")
@Produces("application/xml")
public String updateCustomer(@FormParam("data") String data)
{
...
}
```
Which method call will be replaced with XXX in the above code?
   a. GET
   b. POST
   c. Both
   d. None

# Quiz

**6.**

```
@Path("/customers")
public class Customers {

        @GET
        @Path("{id}")
        public String getCustomer(@PathParam("id”) String id) { …… }

        @POST
        @Path("{id}")
        public void deleteCustomer(@PathParam("id”) String id) { ….. }
}
```

Is this code correct? Will it compile and run?

# Quiz - Answers

1. a
2. b
3. a
4. d
5. b
6. The code is correct and will compile. Though the paths of the sub resource methods are same, their HTTP method differ and hence this is a valid code.

Thank You!