REST WebService

Roy Thomas Fielding

REST WebServices

- An architectural style based on web-standards and the HTTP protocol.
- First described by Roy Fielding in 2000.
- Fast becoming defacto web services and almost replaced SOAP based web services
- Increasingly important on JEE and other web development
  - Easy to implement, its scalable, high performance,

REST Overview

- Resource Oriented Architecture
  - **Data and functionality are considered resources**
  - Accessed using **Uniform Resource Identifiers (URIs)**, typically links on the Web.
- A resource is accessed via a common interface based on the HTTP standard methods.
- **A REST server** provides access to the resources and a REST client which accesses and modify the REST resources.

Lightweight Web Services

- REST is a lightweight alternative to Web Services and RPC.
- REST services is:
  - Platform-independent
  - Language-independent
  - Runs on top of HTTP Protocol
  - Used in the Presence of firewalls too.

- REST does not offer
  - built-in security features
  - encryption
  - session management

- ***These feature can be added by building on top of HTTP:***

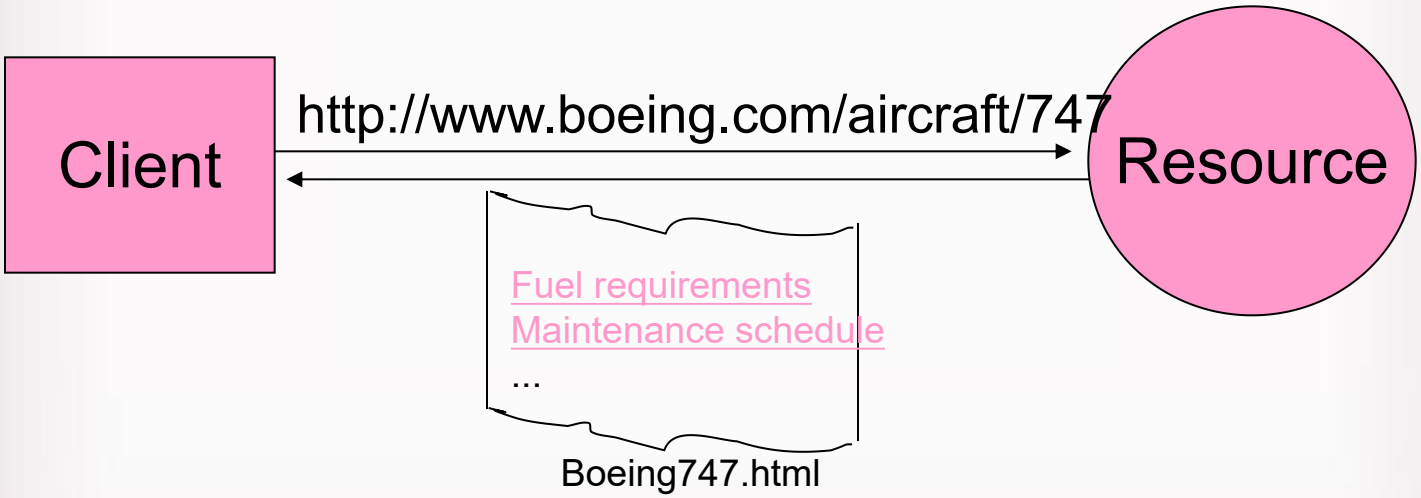
Resource Oriented Architectures

- Involve loose-coupling between client and server, due to:
  - Late binding to resource data
- Successful use revolves around the cache-ability of resource data
  - HTTP cache-control headers are used to specify resource to be cached.
- Best for "**linking and referring**" across organization boundaries
- *Interaction are Stateless , new request should carry all the information required to complete*
- *Must not rely on previous interactions with the same client.*

Web Application & REST Service

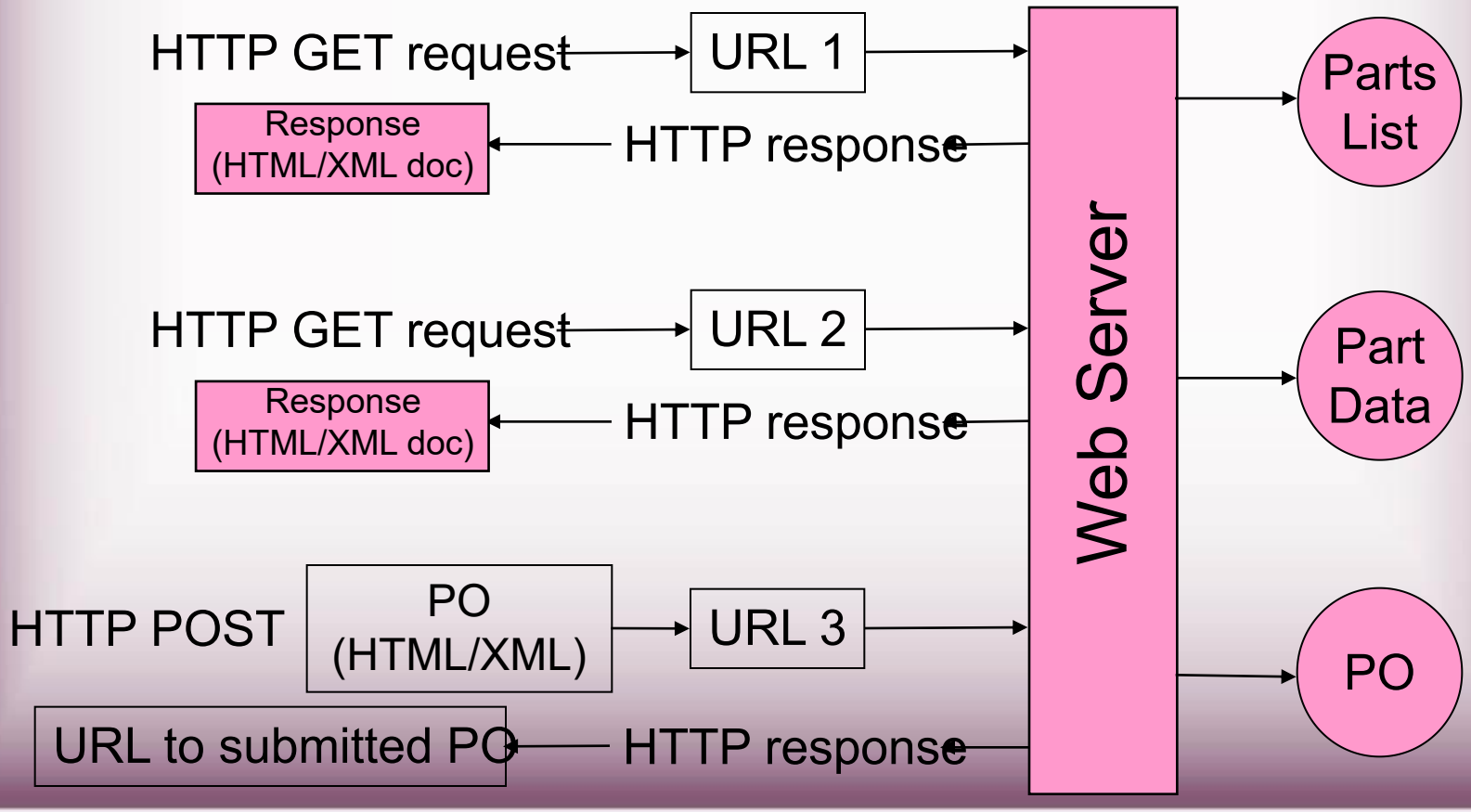
- An API that adheres to the principles of *REST* does not require the client to know anything about the structure of the API.
- Server needs to provide whatever information the client needs to interact with the service.
- The server specifies the location of the resource, and the required fields.
- The browser **doesn't know** in advance **where to submit** the information, and it **doesn't know in advance what to submit.**

"Representational State Transfer? "



- The Client references a Web resource using a URL.
- A **representation** of the resource is returned
- The representation places the client in a new **state**.
- When the client selects a hyperlink it accesses another resource.
- The client application **transfers** state with each resource representation.

The REST way of Designing the Web Services



Examples of REST Web Services

- Popular Real Life Example of REST Web services

- Twitter API
- Amazon.com
- National Digital Forecast Database (NDFD)
- Yahoo Weather Services

- http://www.thomas-bayer.com/sqlrest/CUSTOMER/
- http://www.thomas-bayer.com/sqlrest/CUSTOMER/2

JAX-RI - Implementation

- To simplify development of RESTful Web services JAX-RS API has been designed.
- **Jersey RESTful Web Services framework** is open source, production quality, framework for developing RESTful Web Services
- Provides it's own API that extend the JAX-RS toolkit with additional features and utilities to further simplify RESTful service and client development.
- Download the Reference Implementation
  - https://jersey.github.io/download.html

REST Request Methods

- **GET** - Requests a specific representation of a resource
- **PUT** - Creates or updates a resource with the supplied representation
- **DELETE** - Deletes the specified resource
- **POST** - Submits data to be processed by the identified resource

Root Resource

- *Root resource classes* are POJOs annotated with @Path
  - Can also have one method annotated with @Path
- The annotation's value is a relative URI path.
- **Resource methods**
  - Methods of a resource class annotated with a resource method designator.
  - Method designator with annotation such as @GET, @PUT, @POST, @DELETE

Main Class

– Its Auto Generated bin maven Project

```
public class Main {

public static HttpServer startServer() {
    ResourceConfig rc = new
ResourceConfig().packages("com.example.demo.rs_demo");

GrizzlyHttpServerFactory.createHttpServer(URI.create(BASE_URI), rc);
}
public static void main(String[] args) throws IOException {
     HttpServer server = startServer();
      System.in.read();
        server.stop();
    }
}
```

Root Resource

```
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.Produces;
import jakarta.ws.rs.core.MediaType;

@Path("myresource")
public class MyResource {

@GET
    @Produces(MediaType.TEXT_PLAIN)
    public String getIt() {
        return "Got it!";
    }
}
```

Response Builder

- Used to build Response instances that contain metadata instead of or in addition to an entity.
- Obtained via static methods of the Response class,
- public abstract Response **build**()
  - Create a Response instance from the current ResponseBuilder.
  - The builder is reset to a blank state equivalent to calling the ok method.

Send a JSON Response

- **import javax.json.Json;**
- **import javax.json.JsonObject;**

```
@GET
@Path("/quick")
@Produces(MediaType.APPLICATION_JSON)
public String getMessage(){
    JsonObject map =
Json.createObjectBuilder().add("ram",40).build();
        return map.toString();
}
```

@PathParam

- Annotation is used to access Value of the variable on request method as a parameter
- Binds the value of a path segment to a resource method parameter.
- Used to inject values from the URL into a method parameter.
- **getCustomerById(@PathParam("customerId") int id)**
- **Its invoked as**
- **http://localhost:2020/RestExample/webapi/customer/101**

Multiple @PathParam Annotation

```
@GET
@Path("{year}/{month}/{day}")
public Response getDate(
        @PathParam("year") int year,
        @PathParam("month") int month,
        @PathParam("day") int day) {

   String date = year + "/" + month + "/" + day;

   return Response.status(200)
    .entity("getDate is called, year/month/day : " + date)
    .build();
}
```

@QueryParam

- *Query parameters* are extracted from the request URI
  - Any java language types may be used as query parameters.
  - Can also use Default Value annotation to avoid a null pointer exception if no query parameter is passed.

```
@GET
@Path("login/{zip}")
public String login(@QueryParam("zip") String zip) {
        return "Id is " +id;   }


public String login( @DefaultValue("11111")
                        @QueryParam("zip") String zip) {

}
```

Difference between queryParam and PathPram

- @QueryParam
  - is used to access key/value pairs in the query string of the URI (the part after the ?).
  - http://example.com?q=searchterm,
  - use @QueryParam("q") to get the value of q.
- @PathParam
  - is used to match a part of the URI as a parameter.
  - http://example.com/books/{bookid},
  - use @PathParam("bookid") to get the id of a book.

@Produces Annotation

- **@Produces(MediaType.TEXT_PLAIN[, more-types])**
  - Used to specify the MIME media types or representations a resource can produce and send back to the client.
  - Can be Applied Both at the Class and Method Level
- **Class level**
  - All the methods in a resource can produce the specified MIME types by default.
- **Method Level**
  - Overrides any @Produces annotations applied at the class level.
- **406 Not Acceptable**
  - Returned when methods matching MIME type is not found in a client request,

Server Response

- A server response in REST is often an XML file; for example,

- **XML**
  - XML is easy to expand (clients should ignore unfamiliar fields) and is type-safe;
- **CSV (comma-separated values)**
  - CSV is more compact;
- **JSON (JavaScript Object Notation).**
  - Easy to parse in JavaScript clients and in other languages, too).
- **HTML**
  - Response is a human-readable document;

The @Consumes Annotation

- **@Consumes(type[, more-types])**
  - used to specify which MIME media types of representations a resource can accept, or consume, from the client.
- **Class level**
  - All the response methods accept the specified MIME types by default.
- **Method level**
  - overrides @Consumes annotations applied at the class level.
- **HTTP 415 (“Unsupported Media Type”) .**
  - Returned when methods can not respond to the requested MIME type

Java Object as JSON Response

```
@GET
@Path("{customerId}")
@Produces(MediaType.APPLICATION_JSON)
public Response  getCustomerById(@PathParam("customerId") int id) {


   Customer cust=null;
try {
cust = dao.findById(id);
} catch (SQLException e) {
e.printStackTrace();
}

 return Response.status(200).entity(cust).build();

}
```

Post Request with JSON

```
@POST
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Response addCustomer(Customer cust){
int rowAdded=0;
Customer addedCustomer = null;
try {
 rowAdded  = dao.add(cust);
} catch (SQLException e) {
e.printStackTrace();
}
if(rowAdded ==1) {
 addedCustomer = cust;
}

 return Response.status(201).entity(addedCustomer).build();
}
```

Delete Request

```
@DELETE
@Consumes(MediaType.APPLICATION_JSON)
public Response removeCustomer(Customer cust){

int rowDeleted=0;
Customer deletedCustomer = null;

    try {
rowDeleted =dao.remove(cust.getCustomerId());
} catch (SQLException e) {
e.printStackTrace();
}

 if(rowDeleted ==1) {
 deletedCustomer = cust;
}

 return Response.status(201).entity(deletedCustomer).build();
}
```

Put Method

```
@Path("updateItem/")
@PUT
 public String update(@QueryParam("itemCode") int itemCode ){

System.out.println("item Code"+itemCode);

 String message="One Item with Id
"+"value"+itemCode+"updated";

  return message;
}
```