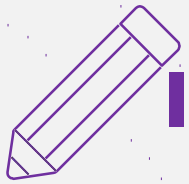
REST & RESTful WEB SERVICES

- Understanding Representational State Transfer (REST)
- Resources / Nouns vs Actions
- Breaking down the definition of REST
  - What is Representational?
  - What is State? Does it mean Session State?
  - What is transfer?
  - Understanding the definition put together
- Set of Architectural Constraints.
- Introducing JSON in REST
- Introducing HTTP Protocol



Introduction

- **Re**presentational **S**tate **T**ransfer.
- Introduced by **Roy Fielding** in 2000.
- **Architectural** style (technically not a standard).
- Uses existing standards, e.g., HTTP.
- REST is an architecture all about the **Client-Server communication**.
- **REST** is about how to **manipulate resources**.

REST

- Client requests a specific resource from the server.
- The server responds to that request by delivering the requested resource.
- Server does not have any information about any client.
- So, there is no difference between the two requests of the same client.

Resources

- REST Server provides access to resources and REST client accesses and presents the resources.
- Here each resource is identified by **URIs/ global IDs**.
- REST uses **various representations** to represent a resource like **text, JSON and XML**.

URI-Example

http://localhost:9999/restapi/books/{id}

- **GET – get the book whose id is provided**
- **POST – update the book whose id is provided**
- **DELETE – delete the book whose id is**

Resource Representation

```
{
    "id":1,
    "name":"Peter",
    "age":45,
    "profession":"Teacher"
}
```

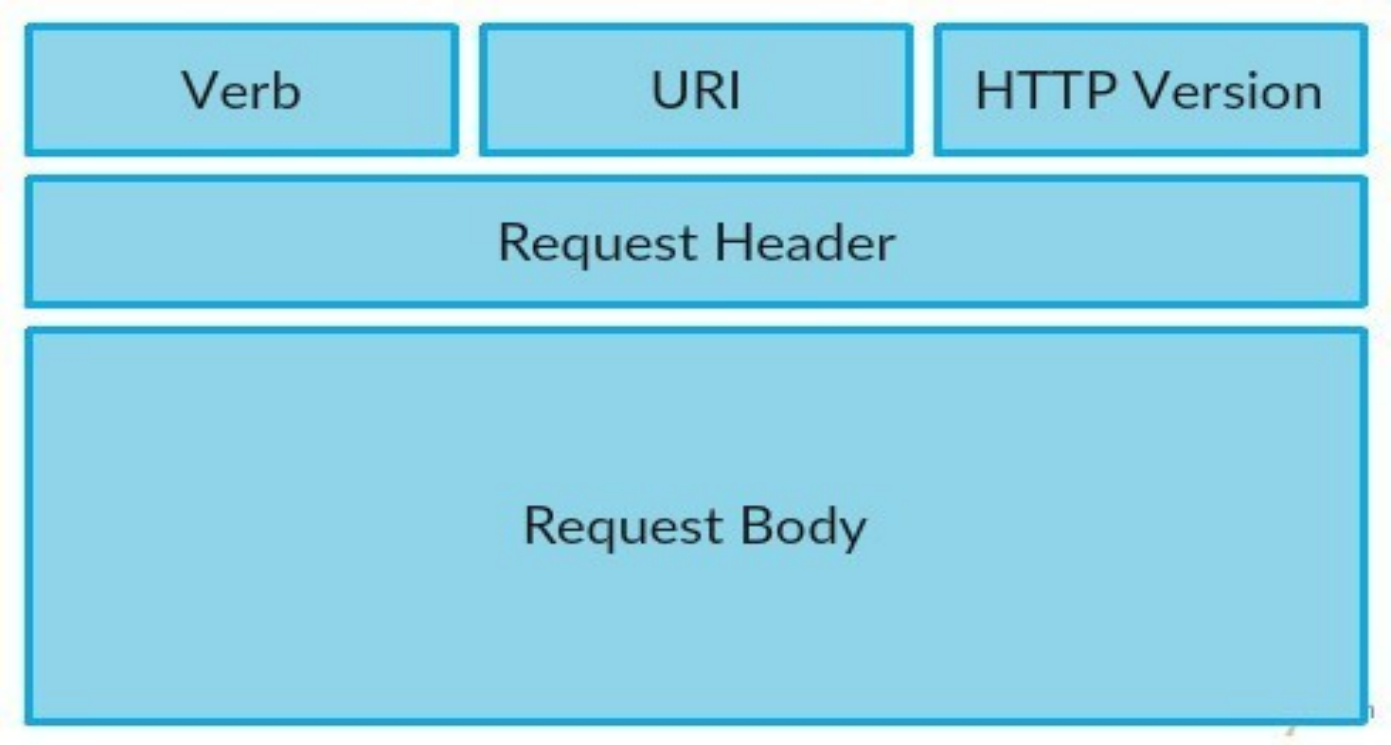
- XML

```
<person>
    <id>1</id>
    <name>Peter</name>
    <age>45</age>
    <profession>Teacher</profession>
</person>
```

Requests & Responses

- RESTful web services uses HTTP protocol as the medium to help the communication between client and server.
- Client sends HTTP Request.
- Server responds it by sending a HTTP Response.
- This is called as **messaging** as well.

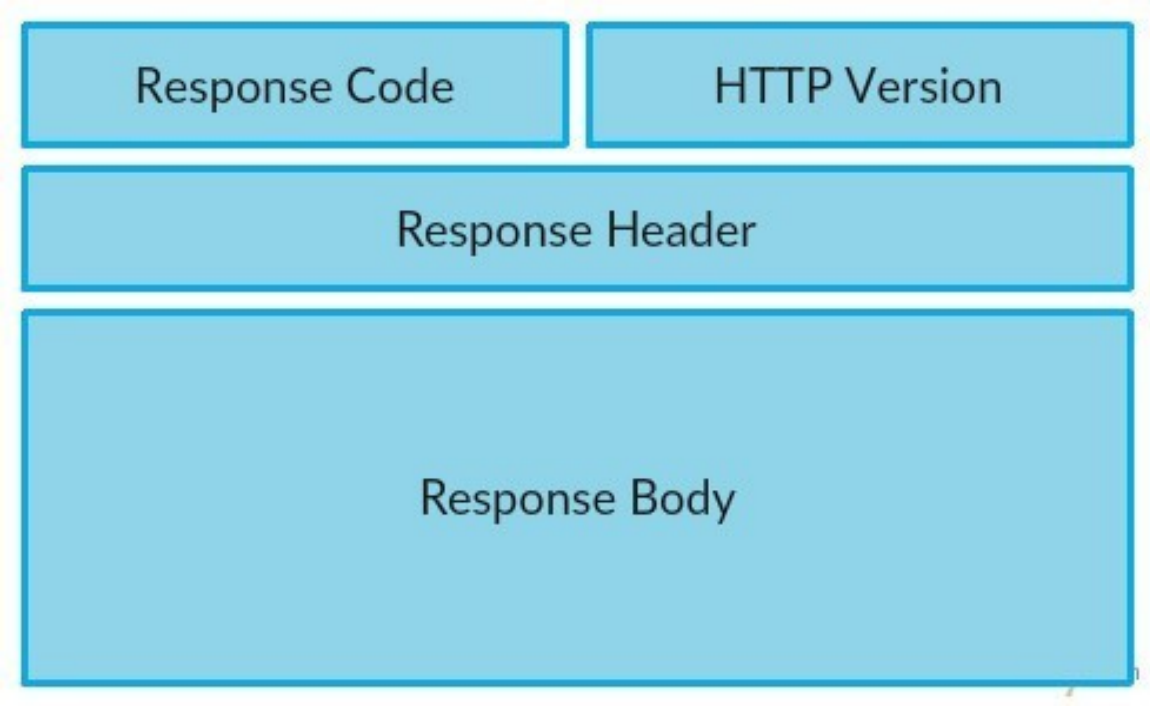
HTTP Request



HTTP Request Cont.

- **Verb**- Indicate HTTP methods such as **GET, POST, DELETE, PUT** etc.
- **URI**- Uniform Resource Identifier (URI) to **identify the resource on server**.
- **HTTP Version**- Indicate **HTTP version**.
- **Request Header**- **Contains metadata** for the HTTP Request message as key-value pairs.
- **Request Body**- **Message content or Resource** representation.

HTTP Response



HTTP Response Cont.

- **Status/Response Code**- **Indicate Server status** for the requested resource.
- **HTTP Version**- Indicate **HTTP version**, for example HTTP v1.1 .
- **Response Header**- **Contains metadata** for the HTTP Response message as key-value pairs. For example, content length, content type, response date, server type etc.
- **Response Body**- Response **message content or**

Addressing

- Addressing refers to **locating a resource or resources on the server.**
- It is analogous to locate a postal address of a person.
- Each resource in REST architecture is identified

```
<protocol>://<service-name>/<ResourceType>/<ResourceID>
```

- by its URI.

Methods

Method	URI	Description
GET (Read)	http://localhost:8080/UserManagement/rest/UserService/users http://localhost:8080/UserManagement/rest/UserService/users/1	Get list of users. Get user of id=1.
POST (Create/ Update)	http://localhost:8080/UserManagement/rest/UserService/users/2	Update user where user id=2.
PUT (Update)	http://localhost:8080/UserManagement/rest/UserService/users/2	Insert user with id=2.

Methods Cont

Method	URI	Description
Delete (Delete)	http://localhost:8080/UserManagement/rest/UserService/users/1	Delete user where user id=1.
Options	http://localhost:8080/UserManagement/rest/UserService/users	List supported web service operations.
Head	http://localhost:8080/UserManagement/rest/UserService/users	Returns HTTP header only.

- **These are the few Constraints of REST.**
  - Client-Server
  - Stateless
  - Cache
  - Uniform Interface
  - Layered System
  - Code on Demand

Client server Constraints

- This constraint states that a REST application should have a Client Server architecture.
- Advantage is Client & Server are separated
- They can evolve independently.
- Clients need not know anything about business logic / data access layer.
- Servers need not know anything about the frontend UI

Stateless Constraints

- Stateless constraint states that the Server does not store any session data.
- The communication between the Client & Server is stateless
- It means that all the information to understand a request is contained within the request.
- Improves Scalability

Cache Constraints

- Cache constraint states responses should be cacheable, if possible.
- It requires that every response should include whether a response can be cacheable or not.
- For subsequent requests, the Client can retrieve from its cache, need to send request to the Server.
- Reduces network latency.

Uniform interface Constraints

- Uniform Interface is the key differentiator between REST & Non-REST APIs.
- There are 4 elements of Uniform Interface constraint.
  - Identification of Resources (typically by an URL).
  - Manipulation of Resources through representations.
  - Self-descriptive messages for each request.
  - HATEOS (Hypermedia As The Engine Of application State)
- Promotes generality as all components interact in the same way.

Layered arch Constraints

- Allows an architecture to be composed of hierarchical layers.
- Each layer doesn't know anything beyond the immediate layer.
- Limits the amount of complexity that can be introduced at any single layer.
- Disadvantage is latency

Code on demand Constraints

- Optional constraint.
- In addition to data, the servers can provide executable code to the client.
- This constraint reduces visibility

HATEOAS

Hypermedia **A**s **T**he **E**ngine **O**f

Used to discover locations and operations.

- Link relations are used to express options.
- **Clients do not need to know URLs.**
- This controls the state.
  - e.g: **Where the user is, Instructions on user's next steps.**

HATEOAS Cont.

- Links contain
  - The target (href, mandatory).
  - A short relationship indication (rel, mandatory).
    - (e. g. “details”, “payment”, “cancel”).
  - The content type needed for the request (type, optional).
  - The HTTP method (method, optional).

HATEOAS Cont.

Sample HATEOAS-based response

```
{
    "name": "Alice",
    "links": [ {
        "rel": "self",
        "href": "http://localhost:8080/customer/1"
    } ]
}
```

JAX-RS

- **JAX-RS** stands for **JAVA API for RESTful Web Services**.
- JAX-RS is a JAVA based programming language API and specification to provide support for created RESTful Web services.
- JAX-RS makes **heavy use of annotations** to simplify development of JAVA based web services.

Some JAX-RS Annotations

Annotation	Description
@Path	Relative path of the resource class/method.
@GET	Used to fetch resource.
@POST	Used to create/update resource.
@DELETE	Used to delete resource.
@HEAD	Used to get status of method availability.
@PUT	Used to create resource.

Some JAX-RS Annotations Cont.

Annotation	Description
@PathParam	Binds the parameter passed to method to a value in path.
@QueryParam	Binds the parameter passed to method to a query parameter in path.
@FormParam	Binds the parameter passed to method to a form value.
@CookieParam	Binds the parameter passed to
@HeaderParam	Binds the parameter passed to method to a HTTP header.

Implementations

- **Apache CXF**, an open source Web service framework.
- **Jersey**, the reference **implementation** from Sun (now Oracle).
- **RESTeasy, JBoss's** implementation.
- **Restlet**.
- **WebSphere** Application Server from IBM.

Brief

Code	Description type
1XX	Informational
2XX	Success
3XX	Redirection
4XX	Client Error
5XX	Server Error

Status Codes in Brief

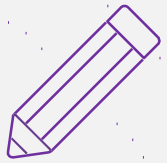
- **200 OK**
  The request has succeeded.
- **201 Created**
  The request has succeeded and a new resource has been created as a result of it.
- **301 Moved Permanently**
  URI of requested resource has been changed.
- **307 Temporary Redirect**
  Directing client to get requested resource to another URI.

Status Codes in Brief

- **308 Permanent Redirect**
  Resource is now permanently located at another URI.
- **400 Bad Request**
  Server could not understand the request due to invalid syntax.
- **403 Forbidden**
  Client does not have access rights to the content so server is rejecting to give proper response.

Status Codes in Brief

- **404 Not Found**
  Server can not find requested resource.
- **500 Internal Server Error**
  The server has encountered a situation it doesn't know how to handle.
- **503 Service Unavailable**
  The server is not ready to handle the request.
- **505 HTTP Version Not Supported**
  The HTTP version used in the request is not supported by the server.



REST

- It helps you organize even a very complex application into simple resources.
- Security: Use HTTPS.
- Performance: REST is less CPU expensive.
- Complexity: REST demands much less in terms of setup, it's just GET/POST after all. SOAP requires much more administration to maintain.

REST vs SOAP

REST	SOAP
A style.	A standard.
Proper REST: Transport must be HTTP/HTTPS.	Normally transport is HTTP/ HTTPS but can be something else.
Response data is normally transmitted as XML, can be something else.	Response data is transmitted as XML.
Request is transmitted as URI.	Request is transmitted as XML.

REST vs SOAP

REST	SOAP
Easy to be called from JavaScript.	JavaScript can call SOAP but it is hard, and not very elegant.
If JSON is returned it is very powerful.	JavaScript parsing XML is slow and the methods differ from browser to browser.
Simply calls services via URL path.	Invokes services by calling RPC method.
result is readable with is just plain XML or JSON.	Doesn't return human readable result.


Jersey
RESTful Web Services in Java

```java
import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;

@ApplicationPath("/rest")
public class AppConfig extends Application {

}
```

```java
@Path("/messages")
public class MessageResources {

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String getMessage() {
        return "hello";
    }
}
```

```java
@Path("/hello")
public class HelloWorldService {

    @GET
    @Path("/{param}")
    public Response getMessage(@PathParam("param") String message) {
        String output = "Jersey say Hello World!!! : " + message;
        return Response.status(200).entity(output).build();
    }
}
```

```java

//  /api/CustomerRest?customerId=121&customerName=raj
@Path("/CustomerRest")
public class CustomerRest {

    @GET
    @Produces(MediaType.TEXT_PLAIN)

    public String getCustomerInfo(@QueryParam("customerId") String customerId,
            @QueryParam("customerName") String customerName) {

        return customerId + " " + customerName + " processed!";
    }
}

```

```java
@Path("/books")
public class BookResources {
    private BookService dao=new BookServiceImp();

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<Book> getAllBooks(){
        return dao.getAllBooks();
    }

    @GET
    @Path("/{bookId}")
    @Produces(MediaType.APPLICATION_JSON)
    public Book getBookById(@PathParam("bookId") int bookId){
        return dao.getBookById(bookId);
    }

    @POST
    @Produces(MediaType.APPLICATION_JSON)
    @Consumes(MediaType.APPLICATION_JSON)
    public Book addBook(Book book){
        return dao.addBook(book);

    }

    @PUT
```

```java
@POST
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public Book addBook(Book book){
    return dao.addBook(book);

}

@PUT
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
@Path("/{bookId}")
public Book updateBook(@PathParam("bookId") int bookId, Book book){
    book.setId(bookId);
    dao.updateBook(book);
    return book;
}

@DELETE
@Path("/{bookId}")
public void delete(@PathParam("bookId") int bookId){
    dao.removeBook(bookId);
}
```

```java
@XmlRootElement(name="book")
@XmlType(propOrder={"id","isbn","title","author","price"})
public class Book {
    private int id;
    private String isbn;
    private String title;
    private String author;
    private double price;

}
```

Spring REST

HTTP Method	Operation Performed
GET	Get a resource (Read a resource)
POST	Create a resource
PUT	Update a resource
DELETE	Delete a resource

Spring Annotations for REST

Annotations	Usage
@Controller	mark the class as a MVC controller
@RequestMapping	Maps the request with path
@PathVariable	Map variable from the path
@RequestBody	unmarshalls the HTTP response body into a Java object injected in the method.
@ResponseBody	marshalls return value as HTTP Response
@Configuration	Spring Config as a class

Example showing Annotations

```
@Controller
@RequestMapping(value = "/ilo")
public class iLOController
{
    @RequestMapping(value = "/server/{id}", method = RequestMethod.GET)
    public @ResponseBody Book getServer(@PathVariable String id) {
            System.out.println("------Gettting Server ------"+id);
     }
        ……
        ……..
}
```

```java
@RestController//   @RestController=@Controller + @ResponseBody
public class BookResources {

    @Autowired
    private BookService service;

    @RequestMapping(value = "/api/book", method = RequestMethod.GET,
            produces = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<Collection<Book>> getAllBooks() {
        Collection<Book> greetings = service.getAllBooks();
        return new ResponseEntity<Collection<Book>>(greetings, HttpStatus.OK);
    }

    @RequestMapping(value = "/api/book/{id}", method = RequestMethod.GET,
            produces = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<Book> getAnBook(@PathVariable Integer id) {
        Book book = service.getBookById(id);
        if (book == null) {
            return new ResponseEntity<Book>(HttpStatus.NOT_FOUND);
        }

        return new ResponseEntity<Book>(book, HttpStatus.OK);
    }
```

```java
@RequestMapping(value = "/api/book", method = RequestMethod.POST,
        consumes = MediaType.APPLICATION_JSON_VALUE, produces = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<Book> createBook(@RequestBody Book book) {
    Book savedBook = service.addBook(book);
    return new ResponseEntity<Book>(savedBook, HttpStatus.CREATED);
}

@RequestMapping(value = "/api/book/{id}", method = RequestMethod.PUT,
        consumes = MediaType.APPLICATION_JSON_VALUE, produces = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<Book> updateBook(@PathVariable Integer id,
        @RequestBody Book book) {

    service.updateBook(book);

    return new ResponseEntity<Book>(HttpStatus.OK);
}

@RequestMapping(value = "/api/book/{id}", method = RequestMethod.DELETE)
public ResponseEntity<Book> deleteBook(@PathVariable("id") Integer id)
        throws Exception {

    service.removeBook(id);

    return new ResponseEntity<Book>(HttpStatus.NO_CONTENT);
}
```