JAX-RS

JAX-RS: Java API for RESTful Web Services is a Java programming language API that provides support in creating web services according to the Representational State Transfer (REST) architectural style.

REST

REST (Representational State Transfer) is a style of architecture for building distributed applications using HTTP (Hyper Text Transfer Protocol) protocol as the transport and application protocol. This style of architecture is characterized by following key facts of the architecture style, each of which is briefly explained in the sections that follow.

- Resource
- URIs & Addressability
- Representation
- Uniform Interface
- Stateless

- Connectedness
- Cacheable
- Layered System Design

Resource
In REST architecture style everything that is referenced in the distributed application is a resource, so if I am building a system to expose employee entity for external applications to access then employee is a resource. If I am building a time entry system then assignments against which time is to be entered, time entry and employee entering time all are resources.

URIs & Addressability
URI represent the name and the address of the resource, a resource should have at least one URI. It is important to note that one URI cannot point to two resources. URI gives addressability to the resource and is an important facet of the REST architecture style.

Representation
The data exchanged between the client of the REST service and the service itself is not the resource itself but a representation of the same in a format that both parties understand example XML, JSON, etc.

Uniform Interface
HTTP exposes a few basic methods (see details below) that expose the resources exposed by REST service to its clients. They are as under.

- GET: to get representation of a resource.
- POST: to create a new resource
- PUT: to update an instance of an existing resource
- DELETE: to delete an instance of the resource
- HEAD: returns the headers of the GET request and not the representation of the resource itself, useful for writing business logic on meta-data of the resource.
- OPTIONS: returns a list of above mentioned HTTP methods supported by the
- resource

Stateless
One of the key facets of the REST architecture style is the statelessness of the REST service, i.e. the service does not maintain any state between two client requests. All state that is required by the service to service the request is carried with each request to the service; this implies that in this architecture style client is stateful if required but not the service. Since HTTP is a stateless protocol itself so it does not come as a surprise that this is a key requirement of REST.

Connectedness
The representation of the resource returned by the REST service not only contains the data about the resource only but also links to other related resources, for example a representation of the employee resource may have a link that point to its manager.

Cacheable
HTTP protocol has the concept of caches where in the client or an intermediary is allowed to cache the representations of resource returned by the service, this is a key facet of REST architecture style as well. Hence the REST service must provide instructions to the client/intermediary whether to cache the response or not and if yes then for how long. The response can be cached at the following locations.

1. Client cache (e.g. browser cache in cache of browser based client or custom client cache).
2. Proxy Server cache (this is client side proxy server, caches only public content).
3. Service Side Cache(e.g. cache on service side)

Layered Architecture
When REST clients connect to REST service they may not be connect to the service itself, the response may be returned by an intermediary which cached the response based on the instructions of the service, when it served the request for the first time. This layered architecture with stateless and cache facets increase the scalability of the service a lot.

Characteristics of REST web services

Here are the characteristics of REST:

- **Client-Server:** a pull-based interaction style: consuming components pull representations.
- **Stateless:** each request from client to server must contain all the information necessary to understand the request, and cannot take advantage of any stored context on the server.
- **Cache:** to improve network efficiency responses must be capable of being labeled as cacheable or non-cacheable.
- **Uniform interface:** all resources are accessed with a generic interface (e.g., HTTP GET, POST, PUT, and DELETE).
- **Named resources** - the system is comprised of resources which are named using a URL.
- **Interconnected resource representations** - the representations of the resources are interconnected using URLs, thereby enabling a client to progress from one state to another.
- **Layered components** - intermediaries, such as proxy servers, cache servers, gateways, etc, can be inserted between clients and resources to support performance, security, etc.

Principles of The RESTful Web Service Design

1. The key to creating Web Services in a REST network (i.e., the Web) is to identify all of the conceptual entities that you wish to expose as services. Above we saw some examples of resources: parts list, detailed part data, purchase order.
2. Create a URL to each resource. The resources should be nouns, not verbs. For example, do not use this:
   http://www.parts-depot.com/parts/getPart?id=00345
   Note the verb, getPart. Instead, use a noun:
   http://www.parts-depot.com/parts/00345
3. Categorize your resources according to whether clients can just receive a representation of the resource, or whether clients can modify (add to) the resource. For the former, make those resources accessible using an HTTP GET. For the later, make those resources accessible using HTTP POST, PUT, and/or DELETE.
4. All resources accessible via HTTP GET should be side-effect free. That is, the resource should just return a representation of the resource. Invoking the resource should not result in modifying the resource.
5. No man/woman is an island. Likewise, no representation should be an island. In other words, put hyperlinks within resource representations to enable clients to drill down for more information, and/or to obtain related information.
6. Design to reveal data gradually. Don't reveal everything in a single response document. Provide hyperlinks to obtain more details.
7. Specify the format of response data using a schema (DTD, W3C Schema, RelaxNG, or Schematron). For those services that require a POST or PUT to it, also provide a schema to specify the format of the response.
8. Describe how your services are to be invoked using either a WSDL document, or simply an HTML document.

The REST Architecture

Key components of a REST architecture:

- **Resources**, which are identified by logical URLs. Both *state* and *functionality* are represented using resources.
  - The logical URLs imply that the resources are universally addressable by other parts of the system.
  - **Resources are the key element of a true RESTful design**, as opposed to "methods" or "services" used in RPC and SOAP Web Services, respectively.

- **A web of resources**, meaning that a single resource should not be overwhelmingly large and contain too fine-grained details. Whenever relevant, a resource should contain *links* to additional information -- just as in web pages.
- The system has a **client-server**, but of course one component's server can be another component's client.
- There is **no connection state**; interaction is stateless (although the servers and resources can of course be stateful). Each new request should carry all the information required to complete it, and must not rely on previous interactions with the same client.
- Resources should be **cachable** whenever possible (with an expiration date/time).
- The protocol must allow the server to explicitly specify which resources may be cached, and for how long.

        - Since HTTP is universally used as the REST protocol, the HTTP cache control headers are used for this purpose.
        - Clients must respect the server's cache specification for each resource.
    - **Proxy servers** can be used as part of the architecture, to improve performance and scalability. Any standard HTTP proxy can be used.

When to use The RESTful Web Service

Architects and developers need to decide when this particular style is an appropriate choice for their applications. A RESTful design may be appropriate when the web services are completely stateless. A good test is to consider whether the interaction can survive a restart of the server.

- A caching infrastructure can be leveraged for performance. If the data that the web service returns is not dynamically generated and can be cached, then the caching infrastructure that web servers and other intermediaries inherently provide can be leveraged to improve performance. However, the developer must take care because such caches are limited to the HTTP GET method for most servers.

- The service producer and service consumer have a mutual understanding of the context and content being passed along. Because there is no formal way to describe the web services interface, both parties must agree out of band on the schema that describe the data being exchanged and on ways to process it meaningfully. In the real world, most commercial applications that expose services as RESTful implementations also distribute so-called value-added toolkit that describe the interfaces to developers in popular programming languages.

- Bandwidth is particularly important and needs to be limited. REST is particularly useful for limited-profile devices such as PDAs and mobile phones, for which the overhead of headers and additional layers of SOAP elements on the XML payload must be restricted.

- Web service delivery or aggregation into existing web sites can be enabled easily with a RESTful style. Developers can use technologies such as Asynchronous JavaScript with XML (AJAX) and toolkit such as Direct Web Remoting (DWR) to consume the services in their web applications. Rather than starting from scratch, services can be exposed with XML and consumed by HTML pages without significantly refactoring the existing web site architecture. Existing developers will be more productive because they are adding to something they are already familiar with, rather than having to start from scratch with new technology.

Designing a RESTful web service

The underlying RESTful web service design principles can be summarized in the following four steps:

- **Requirements gathering**—this step is similar to traditional software requirement gathering practices.
- **Resource identification**—this step is similar to OOD where we identify objects, but we don't worry about messaging between objects.
- **Resource representation definition**—because we exchange representation between clients and servers, we should define what kind of representation we need to use. Typically, we use XML, but JSON has gained popularity. That's not to say that we can't use any other form of resource representation—on the contrary, we could use XHTML or any other form of binary representation, though we let the requirements guide our choices.
- **URI definition**—with resources in place, we need to define the API, which consists of URIs for clients and servers to exchange resources' representations.

HTTP Method and URI Matching

@Path

- The @javax.ws.rs.Path annotation in JAX-RS is used to define a URI matching pattern for incoming HTTP requests.
- It can be placed upon a class or on one or more Java methods. For a Java class to be eligible to receive any HTTP requests, the class must be annotated with at least the @Path("/") expression. These types of classes are called JAX-RS root resources.
- The value of the @Path annotation is an expression that denotes a relative URI to the context root of your JAX-RS application. For example, if you are deploying into a WAR archive of a servlet container, that WAR will have a base URI that browsers and remote clients use to access it. @Path expressions are relative to this URI.
- To receive a request, a Java method must have at least an HTTP method annotation like @javax.ws.rs.GET applied to it. This method is not required to have an @Path annotation on it, though. For example:

```
@Path("/orders")
public class OrderResource {
@GET
public String getAllOrders() {
...
}
}
```

- An HTTP request of GET /orders would dispatch to the getAllOrders() method. @Path can also be applied to your Java method. If you do this, the URI matching pattern is a concatenation of the class's @Path expression and that of the method's. For example:

```
@Path("/orders")
public class OrderResource {
@GET
@Path("unpaid")
public String getUnpaidOrders() {
...
}
}
```

So, the URI pattern for getUnpaidOrders() would be the relative URI /orders/unpaid.

- @Path expressions are not limited to simple wildcard matching expressions. For example,our getCustomer() method takes an integer parameter. We can change our @Path value to only match digits:

```
@Path("/customers")
public class CustomerResource {
@GET
@Path("{id : \\d+}")
public String getCustomer(@PathParam("id") int id) {
...
}
}
```

- We have taken our {id} expression and changed it to {id : \\d+}. The id string is the identifier of the expression. We'll see in the next chapter how you can reference it with@PathParam. The \\d+ string is a regular expression. It is delimited by a ":" character.

- Regular expressions are part of a string matching language that allows you to express complex string matching patterns. In this case, \\d+ matches one or more digits. We will not cover regular expressions in this book, but you can read up on how to use them directly from your JDK's javadocs within the java.util.regex.Pattern class page.† Regular expressions are not limited in matching one segment of a URI. For example:

```
@Path("/customers")
public class CustomerResource {
@GET
@Path("{id : .+}")
public String getCustomer(@PathParam("id") String id) {
...
}
@GET
@Path("{id : .+}/address")
public String getAddress(@PathParam("id") String id) {
...
}
}
```

- We've changed getCustomer()'s @Path expression to {id : .+}. The .+ is a regular expression that will match any stream of characters after /customers. So, the GET /customers/bill/burke request would be routed to getCustomer().
  The getAddress() method has a more specific expression. It will map any stream of characters after /customers and that ends with /address. So, the GET /customers/bill/burke/address request would be routed to the getAddress() method.

HTTP methods

Every HTTP request made to a web service is handled by an appropriately annotated method in a resource class, provided that the resource class has the annotation @Path defined. JAX-RS defines five annotations that map to specific HTTP operations:

- @javax.ws.rs.GET
- @javax.ws.rs.PUT
- @javax.ws.rs.POST
- @javax.ws.rs.DELETE
- @javax.ws.rs.HEAD

@GET

An HTTP GET request is handled by an annotated method that looks as follows:

```
@Path("/users")
public class UserResource {
@GET
public String handleGETRequest() {
}
}
```

The name of the method is not important. However, we should use descriptive names that are representative of the problem we are solving.

@POST

An HTTP POST request is handled by an annotated method that looks as follows:

```
@Path("/users")
public class UserResource {
@POST
```

```
        public String handlePOSTRequest(String payload) {
        }
        }
```

The POST request has a payload that the framework intercepts and delivers to us in the parameter payload. The value of the payload could be an XML structure, so we use a String object. However, the payload could also be a binary stream of, say, MIME type image/jpg, so our object type for the payload must change accordingly (InputStream, for instance). Note that the name of the payload variable is arbitrary, just like the name of the method containing it.

@PUT

An HTTP PUT request is handled by an annotated method that looks as follows:

```
@Path("/users")
public class UserResource {
@PUT
public String handlePUTRequest(String payload) {
}
}
```

Similar to the POST request, the PUT request has a payload associated with it, which is stored in the payload variable.

@DELETE

An HTTP DELETE request is handled by an annotated method that looks as follows:

```
@Path("/users")
public class UserResource {
@DELETE
public void handleDELETERequest() {
}
}
```

For now, ignore the return type of each of these methods.

Input and output formats

We know how to handle requests made to a web service; we now need to figure out how to exchange inbound and outbound representations between clients and servers. For this, we have the following annotations.

@Consumes

- This annotation works together with @POST and @PUT.
- It tells the framework to which method to delegate the incoming request. Specifically, the client sets the *Content-Type* HTTP header and the framework delegates the request to the corresponding handling method.
- An example of this annotation together with @PUT looks as follows:

```
@Path("/users")
public class UserResource {
@PUT
@Consumes("application/xml")
public void updateUser(String representation) {
}
}
```

- In this example we're telling the framework that the updateUser() method accepts

an input stream of MIME type application/xml, which is stored in the variable representation. Therefore, a client connecting to the web service through the URI /users must send an HTTP PUT method request containing the HTTP header *Content-Type* with a value of application/xml.

- A resource can accept different types of payloads. Thus, we use the @PUT annotation with multiple methods to handle requests with different MIME types. So, we could have the following, a method to accept XML structures and a method to accept JSON structures:

```
@Path("/users")
public class UserResource {
@PUT
@Consumes("application/xml")
public void updateXML(String representation) {
}
@PUT
@Consumes("application/json")
public void updateJSON(String representation) {
}
}
```

- Again, the client must provide the MIME type it's sending to the web service. The job of the framework is to delegate the incoming HTTP request to the method matching the intended MIME type.

For @POST we use the same idea:

```
@Path("/users")
public class UserResource {
@POST
@Consumes("application/xml")
public void updateXML(String representation) {
}
@POST
@Consumes("application/json")
public void updateJSON(String representation) {
}
}
```

@Produces

- This annotation works with @GET, @POST, and @PUT.
- It lets the framework know what type of representation to send back to the client.
- Specifically, the client sends an HTTP request together with an Accept HTTP header that maps directly to the *Content-Type* the method produces. So, if the value of the HTTP header *Accept* is application/xml, the method handling the request returns a stream of MIME type application/xml. This annotation can also be used with multiple methods in the same resource class. An example that returns XML and JSON representations looks as follows:

```
@Path("/users")
public class UserResource {
@GET
@Produces("application/xml")
public String getXML(String representation) {
}
@GET
@Produces("application/json")
public String getJSON(String representation) {
```

```
}
}
```