RESTful Web Services are basically REST Architecture based Web Services. In REST Architecture everything is a resource. RESTful web services are light weight, highly scalable and maintainable and are very commonly used to create APIs for web-based applications. This tutorial will teach you the basics of RESTful Web Services and contains chapters discussing all the basic components of RESTful Web Services with suitable examples.

What is REST architecture?

REST stands for REpresentational State Transfer. REST is web standards based architecture and uses HTTP Protocol. It revolves around resource where [every component is a resource and a resource is accessed by a common interface using HTTP standard methods]. REST was first introduced by Roy Fielding in 2000.

In REST architecture, a REST Server simply provides access to resources and REST client accesses and modifies the resources. Here each resource is identified by URIs. REST uses various representation to represent a resource like text, JSON, XML. JSON is the most popular one.

HTTP methods

Following four HTTP methods are commonly used in REST based architecture.

HTTP Verb	CRUD	Entire Collection (e.g. /customers)	Specific Item (e.g. /customers/{id})
POST	Create	201 (Created), 'Location' header with link to /customers/{id} containing new ID.	404 (Not Found), 409 (Conflict) if resource already exists..
GET	Read	200 (OK). list of customers. Use pagination, sorting and filtering to navigate big lists.	200 (OK), single customer. 404 (Not Found), if ID not found or invalid.
PUT	Update/Replace	405 (Method Not Allowed), unless you want to update/replace every resource in the entire collection.	200 (OK) or 204 (No Content). 404 (Not Found), if ID not found or invalid.
PATCH	Update/Modify	405 (Method Not Allowed), unless you want to modify the collection itself.	200 (OK) or 204 (No Content). 404 (Not Found), if ID not found or invalid.
DELETE	Delete	405 (Method Not Allowed), unless you want to delete the whole collection—not often desirable.	200 (OK). 404 (Not Found), if ID not found or invalid.

Introduction to RESTFul web services

A web service is a collection of open protocols and standards used for exchanging data between applications or systems. Software applications written in various programming languages and running on various platforms can use web services to exchange data over computer networks like the Internet in a manner similar to inter-process communication on a single computer. This interoperability (e.g., between Java and Python, or Windows and Linux applications) is due to the use of open standards.

Web services based on REST Architecture are known as RESTful web services. These web services uses HTTP methods to implement the concept of REST architecture. A RESTful web service usually defines

1. a URI, Uniform Resource Identifier
2. a service, provides resource representation such as JSON and
3. Set of HTTP Methods.

Creating Restful Web service

In next chapters, we'll create a web service say user management with following functionalities –

Sr.No.	URI	HTTP Method	POST body	Result
1	/UserService/users	GET	empty	Show list of all the users.
2	/UserService/addUser	POST	JSON String	Add details of new user.
3	/UserService/getUser/:id	GET	empty	Show details of a user.

Create a resource representation class

Now that you've set up the project and build system, you can create your web service.

Begin the process by thinking about service interactions.

The service will handle `GET` requests for `/greeting`, optionally with a `name` parameter in the query string. The `GET` request should return a `200 OK` response with JSON in the body that represents a greeting. It should look something like this:

```
{
    "id": 1,
    "content": "Hello, World!"
}
```

The `id` field is a unique identifier for the greeting, and `content` is the textual representation of the greeting.

To model the greeting representation, you create a resource representation class. Provide a plain old java object with fields, constructors, and accessors for the `id` and `content` data:

`src/main/java/hello/Greeting.java`

```
package hello;

public class Greeting {

    private final long id;
    private final String content;

    public Greeting(long id, String content) {
        this.id = id;
        this.content = content;
    }

    public long getId() {
        return id;
    }

    public String getContent() {
        return content;
    }
}
```

> As you see in steps below, Spring uses the Jackson JSON library to automatically marshal instances of type `Greeting` into JSON.

Next you create the resource controller that will serve these greetings.

Create a resource controller

In Spring's approach to building RESTful web services, HTTP requests are handled by a controller. These components are easily identified by the `@RestController` annotation, and the `GreetingController` below handles `GET` requests for `/greeting` by returning a new instance of the `Greeting` class:

`src/main/java/hello/GreetingController.java`

```java
package hello;

import java.util.concurrent.atomic.AtomicLong;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class GreetingController {

    private static final String template = "Hello, %s!";
    private final AtomicLong counter = new AtomicLong();

    @RequestMapping("/greeting")
    public Greeting greeting(@RequestParam(value="name", defaultValue="World") String
        return new Greeting(counter.incrementAndGet(),
                            String.format(template, name));
    }
}
```

This controller is concise and simple, but there's plenty going on under the hood. Let's break it down step by step.

The `@RequestMapping` annotation ensures that HTTP requests to `/greeting` are mapped to the `greeting()` method.

> The above example does not specify `GET` vs. `PUT`, `POST`, and so forth, because `@RequestMapping` maps all HTTP operations by default. Use `@RequestMapping(method=GET)` to narrow this mapping.

`@RequestParam` binds the value of the query string parameter `name` into the `name` parameter of the `greeting()` method. If the `name` parameter is absent in the request, the `defaultValue` of "World" is used.

The implementation of the method body creates and returns a new `Greeting` object with `id` and `content` attributes based on the next value from the `counter`, and formats the given `name` by using the greeting `template`.

A key difference between a traditional MVC controller and the RESTful web service controller above is the way that the HTTP response body is created. Rather than relying on a view technology to perform server-side rendering of the greeting data to HTML, this RESTful web service controller simply populates and returns a `Greeting` object. The object data will be written directly to the HTTP response as JSON.

This code uses Spring 4's new `@RestController` annotation, which marks the class as a controller where every method returns a domain object instead of a view. It's shorthand for `@Controller` and `@ResponseBody` rolled together.

The `Greeting` object must be converted to JSON. Thanks to Spring's HTTP message converter support, you don't need to do this conversion manually. Because Jackson 2 is on the classpath, Spring's `MappingJackson2HttpMessageConverter` is automatically chosen to convert the `Greeting` instance to JSON.

Otherwise you need to use @XmlRootElement(name = "user") and @XmlElement annotations from javax.xml.bind.annotation package.

Make the application executable

Although it is possible to package this service as a traditional WAR file for deployment to an external application server, the simpler approach demonstrated below creates a standalone application. You package everything in a single, executable JAR file, driven by a good old Java `main()` method. Along the way, you use Spring's support for embedding the Tomcat servlet container as the HTTP runtime, instead of deploying to an external instance.

`src/main/java/hello/Application.java`

```
package hello;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

`@SpringBootApplication` is a convenience annotation that adds all of the following:

- `@Configuration` tags the class as a source of bean definitions for the application context.
- `@EnableAutoConfiguration` tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings.
- Normally you would add `@EnableWebMvc` for a Spring MVC app, but Spring Boot adds it automatically when it sees **spring-webmvc** on the classpath. This flags the application as a web application and activates key behaviors such as setting up a `DispatcherServlet`.
- `@ComponentScan` tells Spring to look for other components, configurations, and services in the `hello` package, allowing it to find the controllers.

The `main()` method uses Spring Boot's `SpringApplication.run()` method to launch an application. Did you notice that there wasn't a single line of XML? No **web.xml** file either. This web application is 100% pure Java and you didn't have to deal with configuring any plumbing or infrastructure.

Build an executable JAR

You can run the application from the command line with Gradle or Maven. Or you can build a single executable JAR file that contains all the necessary dependencies, classes, and resources, and run that. This makes it easy to ship, version, and deploy the service as an application throughout the development lifecycle, across different environments, and so forth.

If you are using Gradle, you can run the application using `./gradlew bootRun`. Or you can build the JAR file using `./gradlew build`. Then you can run the JAR file:

```
java -jar build/libs/gs-rest-service-0.1.0.jar
```

If you are using Maven, you can run the application using `./mvnw spring-boot:run`. Or you can build the JAR file with `./mvnw clean package`. Then you can run the JAR file:

```
java -jar target/gs-rest-service-0.1.0.jar
```

> The procedure above will create a runnable JAR. You can also opt to build a classic WAR file instead.

Logging output is displayed. The service should be up and running within a few seconds.

Test the service

Now that the service is up, visit http://localhost:8080/greeting, where you see:

```
{"id":1,"content":"Hello, World!"}
```

Provide a `name` query string parameter with http://localhost:8080/greeting?name=User. Notice how the value of the `content` attribute changes from "Hello, World!" to "Hello, User!":

```
{"id":2,"content":"Hello, User!"}
```

This change demonstrates that the `@RequestParam` arrangement in `GreetingController` is working as expected. The `name` parameter has been given a default value of "World", but can always be explicitly overridden through the query string.

Notice also how the `id` attribute has changed from `1` to `2`. This proves that you are working against the same `GreetingController` instance across multiple requests, and that its `counter` field is being incremented on each call as expected.

What is a Resource?

REST architecture treats every content as a resource. These resources can be Text Files, Html Pages, Images, Videos or Dynamic Business Data. REST Server simply provides access to resources and REST client accesses and modifies the resources. Here each resource is identified by URIs/ Global IDs. REST uses various representations to represent a resource where Text, JSON, XML. The most popular representations of resources are XML and JSON.

Representation of Resources

A resource in REST is a similar Object in Object Oriented Programming or is like an Entity in a Database. Once a resource is identified then its representation is to be decided using a standard format so that the server can send the resource in the above said format and client can understand the same format.

For example, in RESTful Web Services - First Application chapter, a user is a resource which is represented using the following XML format –

```
<user>
   <id>1</id>
   <name>Mahesh</name>
   <profession>Teacher</profession>
</user>
```

The same resource can be represented in JSON format as follows –

```
{
   "id":1,
   "name":"Mahesh",
   "profession":"Teacher"
}
```

Note:

With Java 8 streams the best part is that the above rearrangement of workflow can happen while the code stays almost exactly the same — no part of it must migrate to another layer. For the case of Hibernate queries, compare

```java
public List<UserDto> userList() {
  return sf.getCurrentSession()
      .createQuery("select new UserDto(name, email) from User").list();
}
```

with

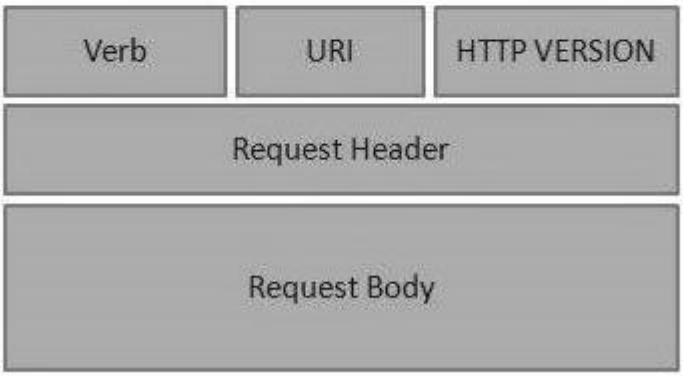
```java
public Stream<UserDto> userStream() {
  return resultStream(sf.getCurrentSession()
      .createQuery("select new UserDto(name, email) from User"));
}
```

The code looks almost exactly the same, but the workflow implied by it changed from eager to lazy. As always, the stream can be automatically parallelized at the flick of a switch; just keep in mind that this pays off only if your transformation logic is substantial enough to make CPU the bottleneck, as opposed to I/O.

RESTful Web Services – Messages

RESTful Web Services make use of HTTP protocols as a medium of communication between client and server. A client sends a message in form of a HTTP Request and the server responds in the form of an HTTP Response. This technique is termed as Messaging. These messages contain message data and metadata i.e. information about message itself. Let us have a look on the HTTP Request and HTTP Response messages for HTTP 1.1.

HTTP Request

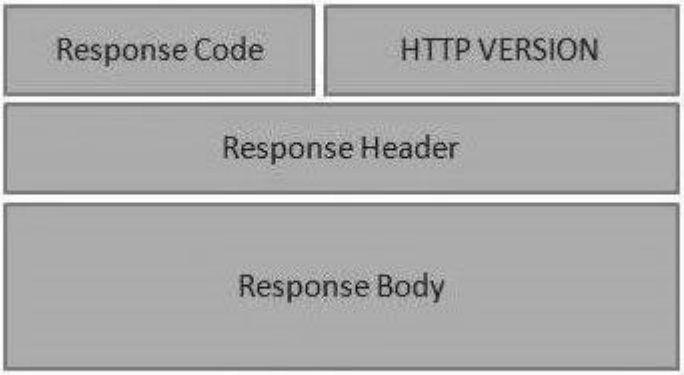


HTTP Request

An HTTP Request has five major parts –

- **Verb** – Indicates the HTTP methods such as GET, POST, DELETE, PUT, etc.
- **URI** – Uniform Resource Identifier (URI) to identify the resource on the server.
- **HTTP Version** – Indicates the HTTP version. For example, HTTP v1.1.
- **Request Header** – Contains metadata for the HTTP Request message as key-value pairs. For example, client (or browser) type, format supported by the client, format of the message body, cache settings, etc.
- **Request Body** – Message content or Resource representation.

HTTP Response



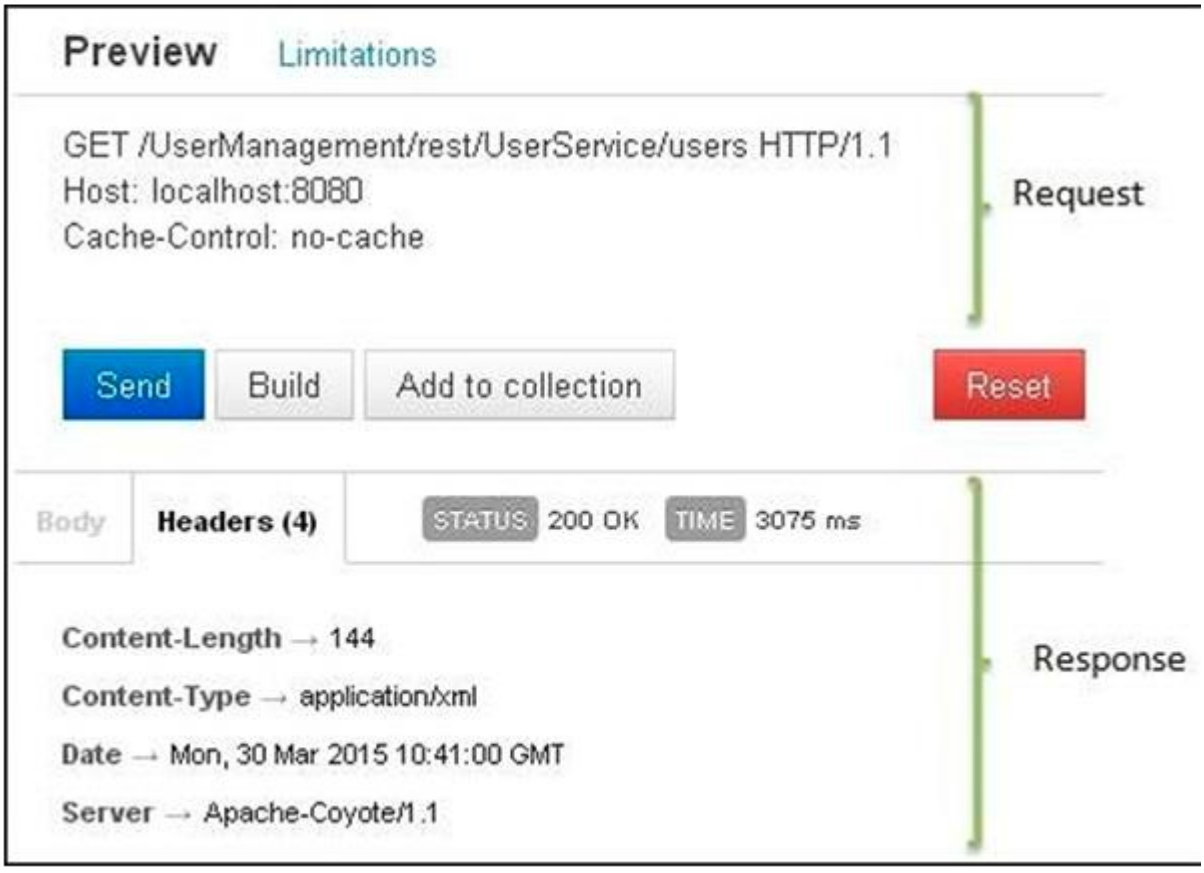
HTTP Response

An HTTP Response has four major parts –

- **Status/Response Code** – Indicates the Server status for the requested resource. For example, 404 means resource not found and 200 means response is ok.
- **HTTP Version** – Indicates the HTTP version. For example HTTP v1.1.
- **Response Header** – Contains metadata for the HTTP Response message as keyvalue pairs. For example, content length, content type, response date, server type, etc.
- **Response Body** – Response message content or Resource representation.

Example

As we have explained in the RESTful Web Services - First Application chapter, let us put http://localhost:8080/UserManagement/rest/UserService/users in the POSTMAN with a GET request. If you click on the Preview button which is near the send button of Postman and then click on the Send button, you may see the following output.



RESTful Web Services - Addressing

Addressing refers to locating a resource or multiple resources lying on the server. It is analogous to locate a postal address of a person.

Each resource in REST architecture is identified by its URI (Uniform Resource Identifier). A URI is of the following format –

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

Purpose of an URI is to locate a resource(s) on the server hosting the web service. Another important attribute of a request is VERB which identifies the operation to be performed on the resource. For example, in RESTful Web Services - First Application chapter, the URI is **http://localhost:8080/UserManagement/rest/UserService/users** and the VERB is GET.

Note: @Path annotation is for <ResourceType> and @RequestMapping annotation is for <ResourceID>

Example

Following is an example of a poor URI to fetch a user.

```
http://localhost:8080/UserManagement/rest/UserService/getUser/1
```

Following is an example of a good URI to fetch a user.

```
http://localhost:8080/UserManagement/rest/UserService/users/1
```

Idempotent REST API:

> *An idempotent HTTP method is an HTTP method that can be called many times without different outcomes. It would not matter if the method is called only once, or ten times over. The result should be the same. It essentially means that the result of a successfully performed request is independent of the number of times it is executed. For example, in arithmetic, adding zero to a number is idempotent operation.*

If you follow REST principles in designing API, you will have automatically **idempotent REST APIs** for GET, PUT, DELETE, HEAD, OPTIONS and TRACE HTTP methods. Only `POST` APIs will not be idempotent.

1. `POST` is NOT idempotent.
2. `GET`, `PUT`, `DELETE`, `HEAD`, `OPTIONS` and `TRACE` are idempotent.

HTTP GET, HEAD, OPTIONS and TRACE

`GET`, `HEAD`, `OPTIONS` and `TRACE` methods NEVER change the resource state on server. They are purely for retrieving the resource representation or meta data at that point of time. So invoking multiple requests will not have any write operation on server, so **GET, HEAD, OPTIONS and TRACE are idempotent.**

HTTP PUT

Generally – not necessarily – `PUT` APIs are used to update the resource state. If you invoke a `PUT` API N times, the very first request will update the resource; then rest N-1 requests will just overwrite the same resource state again and again – effectively not changing anything. Hence, **PUT is idempotent.**

HTTP DELETE

When you invoke N similar `DELETE` requests, first request will delete the resource and response will be `200 (OK)` or `204 (No Content)`. Other N-1 requests will return `404 (Not Found)`. Clearly, the response is different from first request, but there is no change of state for any resource on server side because original resource is already deleted. So, **DELETE is idempotent.**

Using @Consumes and @Produces to Customize Requests and Responses

The `@Produces` Annotation

The `@Produces` annotation is used to specify the MIME media types or representations a resource can produce and send back to the client. If `@Produces` is applied at the class level, all the methods in a resource can produce the specified MIME types by default. If it is applied at the method level, it overrides any `@Produces` annotations applied at the class level.

The `@Consumes` Annotation

The `@Consumes` annotation is used to specify which MIME media types of representations a resource can accept, or consume, from the client. If `@Consumes` is applied at the class level, all the response methods accept the specified MIME types by default. If `@Consumes` is applied at the method level, it overrides any `@Consumes` annotations applied at the class level.

Default is `*/*`

At Server Side

At server side if you have not specified Content-Type it can accept any content-type provided by client. However if you have specific format JSON, XML etc, you need to specify the Content-Type so that CXF can invoke corresponding providers. In some cases where you have same REST path with different content-Type then it can select corresponding method based on Content-Type. In GET method if Content-Type is mismatched it will be ignored, but not in POST, it will throw 415 error.

At Client Side

Same applies to the client side as well, however generally we specify the accept type in case server can send multiple content-type, in this case we specify what content-type we can accept. so that server can send the data in specified content-type, generally when we expose REST we expose with JSON and xml type, during this case we need to send accept type to server, so that server can send either JSON or XML.

```
@GET
@Path("/users")
@Produces(MediaType.APPLICATION_XML)
public List<User> getUsers(){
   return userDao.getAllUsers();
}

@GET
@Path("/users/{userid}")
@Produces(MediaType.APPLICATION_XML)
public User getUser(@PathParam("userid") int userid){
   return userDao.getUser(userid);
}

@POST
@Path("/users")
@Produces(MediaType.APPLICATION_XML)
@Consumes(MediaType.APPLICATION_FORM_URLENCODED)
public String createUser(@FormParam("id") int id,
   @FormParam("name") String name,
   @FormParam("profession") String profession,
   @Context HttpServletResponse servletResponse) throws IOException{
   User user = new User(id, name, profession);
   int result = userDao.addUser(user);
   if(result == 1){
      return SUCCESS_RESULT;
   }
   return FAILURE_RESULT;
}

@PUT
@Path("/users")
@Produces(MediaType.APPLICATION_XML)
@Consumes(MediaType.APPLICATION_FORM_URLENCODED)
public String updateUser(@FormParam("id") int id,
   @FormParam("name") String name,
   @FormParam("profession") String profession,
   @Context HttpServletResponse servletResponse) throws IOException{
```

RESTful Web Services - Statelessness

As per the REST architecture, a RESTful Web Service should not keep a client state on the server. This restriction is called Statelessness. It is the responsibility of the client to pass its context to the server and then the server can store this context to process the client's further request. For example, session maintained by server is identified by session identifier passed by the client.

RESTful Web Services should adhere to this restriction. We have seen this in the RESTful Web Services - Methods chapter, that the web service methods are not storing any information from the client they are invoked from.

Advantages of Statelessness

Following are the benefits of statelessness in RESTful Web Services –

- Web services can treat each method request independently.
- Web services need not maintain the client's previous interactions. It simplifies the application design.
- As HTTP is itself a statelessness protocol, RESTful Web Services work seamlessly with the HTTP protocols.

Disadvantages of Statelessness

Following are the disadvantages of statelessness in RESTful Web Services –

- Web services need to get extra information in each request and then interpret to get the client's state in case the client interactions are to be taken care of.

RESTful Web Services – Caching

Caching refers to storing the server response in the client itself, so that a client need not make a server request for the same resource again and again. A server response should have information about how caching is to be done, so that a client caches the response for a time-period or never caches the server response.

Following are the headers which a server response can have in order to configure a client's caching –

Sr.No.	Header & Description
1	**Date** Date and Time of the resource when it was created.
2	**Last Modified** Date and Time of the resource when it was last modified.
3	**Cache-Control** Primary header to control caching.
4	**Expires** Expiration date and time of caching.
5	**Age** Duration in seconds from when resource was fetched from the server.

Cache-Control Header

Following are the details of a Cache-Control header –

Sr.No.	Directive & Description
1	**Public** Indicates that resource is cacheable by any component.
2	**Private** Indicates that resource is cacheable only by the client and the server, no intermediary can cache the resource.
3	**no-cache/no-store** Indicates that a resource is not cacheable.
4	**max-age** Indicates the caching is valid up to max-age in seconds. After this, client has to make another request.
5	**must-revalidate** Indication to server to revalidate resource if max-age has passed.

Best Practices

- Always keep static contents like images, CSS, JavaScript cacheable, with expiration date of 2 to 3 days.
- Never keep expiry date too high.
- Dynamic content should be cached for a few hours only.

RESTful Web Services - Security

2. Enable Spring Security

The architecture of Spring Security is based entirely on Servlet Filters. Therefore, this comes before Spring MVC in regards to the processing of HTTP requests.

The simplest option to register the Spring security filter is by annotating our config class with *@EnableWebSecurity*:

```
@Config
@EnableWebSecurity
public class SecurityJavaConfig extends WebSecurityConfigurerAdapter {

    // ...
}
```

Now, let's create users with different roles in *SecurityJavaConfig* that we will be using to authenticate our API endpoints:

```
@Override
protected void configure(final AuthenticationManagerBuilder auth) throws Exception {
    auth.inMemoryAuthentication()
        .withUser("admin").password(encoder().encode("adminPass")).roles("ADMIN")
        .and()
        .withUser("user").password(encoder().encode("userPass")).roles("USER");
}

@Bean
public PasswordEncoder  encoder() {
    return new BCryptPassowordEncoder();
}
```

Next, let's configure security for our API endpoints:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
    .csrf().disable()
    .exceptionHandling()
    .authenticationEntryPoint(restAuthenticationEntryPoint)
    .and()
    .authorizeRequests()
    .antMatchers("/api/foos").authenticated()
    .antMatchers("/api/admin/**").hasRole("ADMIN")
    .and()
    .formLogin()
    .successHandler(mySuccessHandler)
    .failureHandler(myFailureHandler)
    .and()
    .logout();
}
```

RESTful Web Services - Java (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. Its 2.0 version was released on the 24th May 2013. JAX-RS uses annotations available from Java SE 5 to simplify the development of JAVA based web services creation and deployment. It also provides supports for creating clients for RESTful Web Services.

13) Mention what is JAX-WS and JAX-RS?

Both JAX-WS and JAX-RS are libraries (APIs) for doing communication in various ways in Java. JAX-WS is a library that can be used to do SOAP communication in JAVA, and JAX-RS lets you do the REST communication in JAVA.

15) Mention what is the difference between SOAP and REST?

SOAP	REST
• SOAP is a protocol through which two computer communicates by sharing XML document • SOAP permits only XML • SOAP based reads cannot be cached • SOAP is like custom desktop application, closely connected to the server • SOAP is slower than REST • It runs on HTTP but envelopes the message	• Rest is a service architecture and design for network-based software architectures • REST supports many different data formats • REST reads can be cached • A REST client is more like a browser; it knows how to standardized methods and an application has to fit inside it • REST is faster than SOAP • It uses the HTTP headers to hold meta information

RESTFul Web Services serves JSON that is faster to parse than XML