REpresentational **S**tate **T**ransfer

Web Service

Contents

List of Figures

List of Tables

1 Introduction

This article will explain the basics of Spring Data REST and show how to use it to build a simple REST API.

In general, Spring Data REST is built on top of the Spring Data project and makes it easy to build hypermedia-driven REST web services that connect to Spring Data repositories – all using HAL as the driving hypermedia type.

It takes away a lot of the manual work usually associated with such tasks and makes implementing basic CRUD functionality for web applications quite simple.

2 Maven

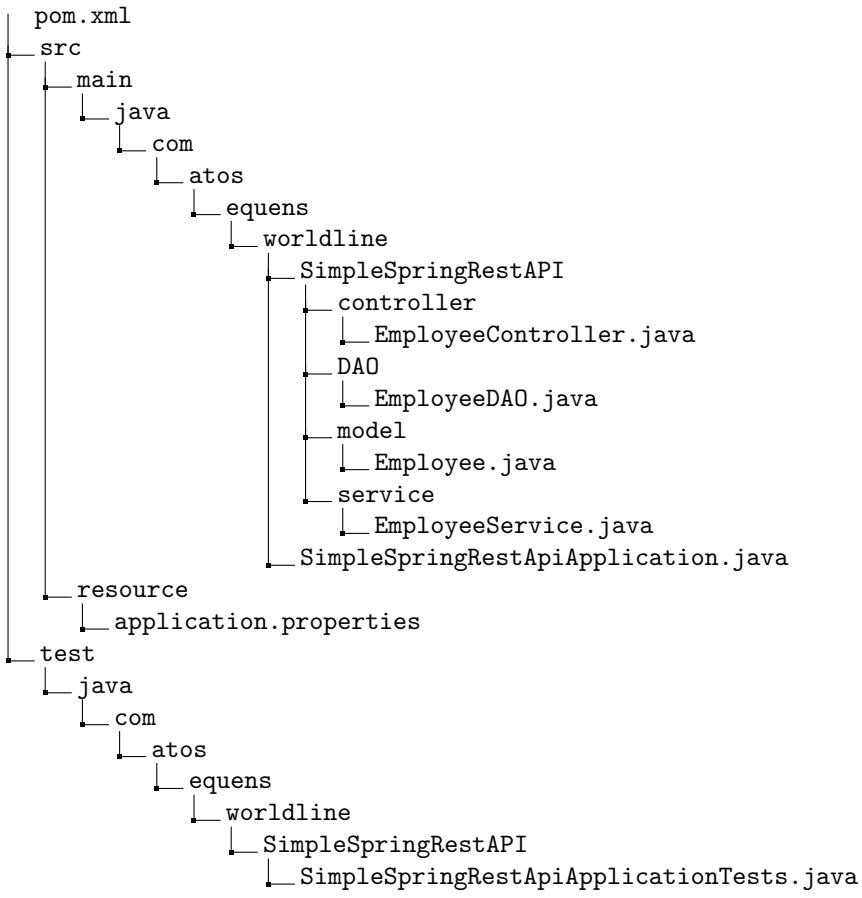
Maven is a build automation tool used primarily for Java projects.

Maven addresses two aspects of building software: first, it describes how software is built,[clarification needed] and second, it describes its dependencies. Unlike earlier tools like Apache Ant, it uses conventions for the build procedure, and only exceptions need to be written down. An XML file describes the software project being built, its dependencies on other external modules and components, the build order, directories, and required plug-ins. It comes with pre-defined targets for performing certain well-defined tasks such as compilation of code and its packaging.

Maven dynamically downloads Java libraries and Maven plug-ins from one or more repositories such as the Maven 2 Central Repository, and stores them in a local cache.[3] This local cache of downloaded artifacts can also be updated with artifacts created by local projects. Public repositories can also be updated.

3 Project Folder Structure

The following listing shows a typical layout:



4 Rest vs SOAP

Table 1: Rest vs SOAP.

	SOAP	**REST**
Meaning	Simple Object Access Protocol	Representational State Transfer
Design	Standardized protocol with predefined rules to follow.	Architectural style with loose guidelines and recommendations.
Approach	Function-driven (data available as services, e.g.: "getUser")	Data-driven (data available as resources, e.g. "user").
Statefulness	Stateless by default, but it's possible to make a SOAP API stateful.	Stateless (no server-side sessions).
Statefulness	Stateless by default, but it's possible to make a SOAP API stateful.	Stateless (no server-side sessions).
Caching	API calls cannot be cached.	API calls can be cached.
Security	WS-Security with SSL support. Built-in ACID compliance.	Supports HTTPS and SSL.
Performance	Requires more bandwidth and computing power.	Requires fewer resources.
Message format	Only XML.	Plain text, HTML, XML, JSON, YAML, and others.
Transfer protocol(s)	HTTP, SMTP, UDP, and others.	Only HTTP
Recommended for	Enterprise apps, high-security apps, distributed environment, financial services, payment gateways, telecommunication services.	Public APIs for web services, mobile services, social networks.
Advantages	High security, standardized, extensibility.	Scalability, better performance, browser-friendliness, flexibility.
Disadvantages	Poorer performance, more complexity, less flexibility.	Poorer performance, more complexity, less flexibility.

5 HTTP Method

The primary or most-commonly-used HTTP verbs (or methods, as they are properly called) are POST, GET, PUT, PATCH, and DELETE. These correspond to create, read, update, and delete (or CRUD) operations, respectively. There are a number of other verbs, too, but are utilized less frequently. Of those less-frequent methods, OPTIONS and HEAD are used more often than others.

5.0 GET

The HTTP GET method is used to **read** (or retrieve) a representation of a resource. In the "happy" (or non-error) path, GET returns a representation in XML or JSON and an HTTP response code of 200 (OK). In an error case, it most often returns a 404 (NOT FOUND) or 400 (BAD REQUEST).

According to the design of the HTTP specification, GET (along with HEAD) requests are used only to read data and not change it. Therefore, when used this way, they are considered safe. That is, they can be called without risk of data modification or corruption—calling it once has the same effect as calling it 10 times, or none at all. Additionally, GET (and HEAD) is idempotent, which means that making multiple identical requests ends up having the same result as a single request.

Do not expose unsafe operations via GET—it should never modify any resources on the server.

Examples:

- GET http://www.example.com/customers/12345
- GET http://www.example.com/customers/12345/orders
- GET http://www.example.com/buckets/sample

5.1 POST

The POST verb is most-often utilized to **create** new resources. In particular, it's used to create subordinate resources. That is, subordinate to some other (e.g. parent) resource. In other words, when creating a new resource, POST to the parent and the service takes care of associating the new resource with the parent, assigning an ID (new resource URI), etc.

On successful creation, return HTTP status 201, returning a Location header with a link to the newly-created resource with the 201 HTTP status.

POST is neither safe nor idempotent. It is therefore recommended for non-idempotent resource requests. Making two identical POST requests will most-likely result in two resources containing the same information.

Examples:

- POST http://www.example.com/customers
- POST http://www.example.com/customers/12345/orders

5.2 PUT

PUT is most-often utilized for **update** capabilities, PUT-ing to a known resource URI with the request body containing the newly-updated representation of the original resource.

However, PUT can also be used to create a resource in the case where the resource ID is chosen by the client instead of by the server. In other words, if the PUT is to a URI that contains the value of a non-existent resource ID. Again, the request body contains a resource representation. Many feel this is convoluted and confusing. Consequently, this method of creation should be used sparingly, if at all.

Alternatively, use POST to create new resources and provide the client-defined ID in the body representation—presumably to a URI that doesn't include the ID of the resource (see POST below).

On successful update, return 200 (or 204 if not returning any content in the body) from a PUT. If using PUT for create, return HTTP status 201 on successful creation. A body in the response is optional—providing one consumes more bandwidth. It is not necessary to return a link via a Location header in the creation case since the client already set the resource ID.

PUT is not a safe operation, in that it modifies (or creates) state on the server, but it is idempotent. In other words, if you create or update a resource using PUT and then make that same call again, the resource is still there and still has the same state as it did with the first call.

If, for instance, calling PUT on a resource increments a counter within the resource, the call is no longer idempotent. Sometimes that happens and it may be enough to document that the call is not idempotent. However, it's recommended to keep PUT requests idempotent. It is strongly recommended to use POST for non-idempotent requests.

Examples:

- PUT http://www.example.com/customers/12345
- PUT http://www.example.com/customers/12345/orders/98765
- PUT http://www.example.com/buckets/secretstuff

5.3 DELETE

DELETE is pretty easy to understand. It is used to **delete** a resource identified by a URI.

On successful deletion, return HTTP status 200 (OK) along with a response body, perhaps the representation of the deleted item (often demands too much bandwidth), or a wrapped response (see Return Values below). Either that or return HTTP status 204 (NO CONTENT) with no response body. In other words, a 204 status with no body, or the JSEND-style response and HTTP status 200 are the recommended responses.

HTTP-spec-wise, DELETE operations are idempotent. If you DELETE a resource, it's removed. Repeatedly calling DELETE on that resource ends up

the same: the resource is gone. If calling DELETE say, decrements a counter (within the resource), the DELETE call is no longer idempotent. As mentioned previously, usage statistics and measurements may be updated while still considering the service idempotent as long as no resource data is changed. Using POST for non-idempotent resource requests is recommended.

There is a caveat about DELETE idempotence, however. Calling DELETE on a resource a second time will often return a 404 (NOT FOUND) since it was already removed and therefore is no longer findable. This, by some opinions, makes DELETE operations no longer idempotent, however, the end-state of the resource is the same. Returning a 404 is acceptable and communicates accurately the status of the call.

Examples:

- DELETE http://www.example.com/customers/12345
- DELETE http://www.example.com/customers/12345/orders
- DELETE http://www.example.com/bucket/sample

6 HTTP Status Codes

The Status-Code element in a server response, is a 3-digit integer where the first digit of the Status-Code defines the class of response and the last two digits do not have any categorization role. There are 5 values for the first digit:

Table 2: HTTP Status Codes.

SL.NO	Code	Description
1	1XX	It means the request has been received and the process is continuing.
2	2XX	It means the action was successfully received, understood, and accepted.
3	3XX	It means further action must be taken in order to complete the request.
4	4XX	It means the request contains incorrect syntax or cannot be fulfilled.
5	5XX	It means the server failed to fulfill an apparently valid request.

Top 10" HTTP Status Code. More REST service-specific information is contained in the entry.

6.0 200 OK

Standard response for successful HTTP requests. The actual response will depend on the request method used. In a GET request, the response will contain

an entity corresponding to the requested resource. In a POST request the response will contain an entity describing or containing the result of the action.

6.1 201 Created

The request has been fulfilled and resulted in a new resource being created. The newly created resource can be referenced by the URI(s) returned in the entity of the response, with the most specific URI for the resource given by a Location header field. The response SHOULD include an entity containing a list of resource characteristics and location(s) from which the user or user agent can choose the one most appropriate. The entity format is specified by the media type given in the Content-Type header field.

6.2 204 No Content

The server has fulfilled the request but does not need to return an entity-body, and might want to return updated metainformation. The response MAY include new or updated metainformation in the form of entity-headers, which if present SHOULD be associated with the requested variant.

6.3 304 Not Modified

If the client has performed a conditional GET request and access is allowed, but the document has not been modified, the server SHOULD respond with this status code. The 304 response MUST NOT contain a message-body, and thus is always terminated by the first empty line after the header fields.

6.4 400 Bad Request

The request could not be understood by the server due to malformed syntax. The client SHOULD NOT repeat the request without modifications.

6.5 401 Unauthorized

The request requires user authentication. The response MUST include a WWW-Authenticate header field (section 14.47) containing a challenge applicable to the requested resource. The client MAY repeat the request with a suitable Authorization header field (section 14.8). If the request already included Authorization credentials, then the 401 response indicates that authorization has been refused for those credentials. If the 401 response contains the same challenge as the prior response, and the user agent has already attempted authentication at least once, then the user SHOULD be presented the entity that was given in the response, since that entity might include relevant diagnostic information. HTTP access authentication is explained in "HTTP Authentication: Basic and Digest Access Authentication".

6.6 403 Forbidden

The server understood the request, but is refusing to fulfill it. Authorization will not help and the request SHOULD NOT be repeated. If the request method was not HEAD and the server wishes to make public why the request has not been fulfilled, it SHOULD describe the reason for the refusal in the entity. If the server does not wish to make this information available to the client, the status code 404 (Not Found) can be used instead.

6.7 404 Not Found

The server has not found anything matching the Request-URI. No indication is given of whether the condition is temporary or permanent. The 410 (Gone) status code SHOULD be used if the server knows, through some internally configurable mechanism, that an old resource is permanently unavailable and has no forwarding address. This status code is commonly used when the server does not wish to reveal exactly why the request has been refused, or when no other response is applicable.

6.8 405 Method Not Allowed

The method specified in the Request-Line is not allowed for the resource identified by the Request-URI. The response MUST include an Allow header containing a list of valid methods for the requested resource.

6.9 409 Conflict

The request could not be completed due to a conflict with the current state of the resource. This code is only allowed in situations where it is expected that the user might be able to resolve the conflict and resubmit the request. The response body SHOULD include enough information for the user to recognize the source of the conflict. Ideally, the response entity would include enough information for the user or user agent to fix the problem; however, that might not be possible and is not required.

6.10 500 Internal Server Error

The server encountered an unexpected condition which prevented it from fulfilling the request.

7 HATEOAS

8 Project Object Model

```
\\pom.xml
  <?xml version="1.0" encoding="UTF-8"?>
  <project xmlns="http://maven.apache.org/POM/4.0.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
        http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-parent</artifactId>
      <version>2.1.6.RELEASE</version>
      <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.atos.equens.worldline</groupId>
    <artifactId>Messenger</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>MessengerRestSpring</name>
    <description>Messenger Spring Boot Rest API project for Spring
        Boot</description>
    <dependencies>
      <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-jersey</artifactId>
      </dependency>
      <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
      </dependency>
      <dependency>
        <groupId>com.oracle</groupId>
        <artifactId>ojdbc14</artifactId>
        <version>10.2.0.4.0</version>
      </dependency>
      <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
      </dependency>
    </dependencies>
  </project>
```

The above pom.xml describes a spring boot maven application with minimum dependencies with which we can write the simple spring boot appliaction which illustrates the usange of Rest Web Services and its method. Let's us study the pom.xml file in brief just get to the purpose of each dependencies why being used.

8.0 Parent

Parent tag in above pom.xml differentiates the spring boot application from a simple maven appliaction. It tells maven that it is Spring boot appliaction asual as asks for transitive dependencies to be avalble.

```
<parent>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-parent</artifactId>
   <version>2.1.6.RELEASE</version>
   <relativePath/> <!-- lookup parent from repository -->
</parent>
```

If you are running spring-boot application for the first time, required dependencies which are mentioned in pom.xml will be downloaded from the maven repository (.m2/repository). Downloaded dependencies will be stored in local repository which will be easy to use next time withought having to download every time you create new application.

8.1 Dependency

Dependency tag describes the required maven dependencies. In above pom.xml,I tried to use minimum dependencies which required to illustrate the standard Rest Api using Spring Boot. Let us go through each dependencies and purpose of being used.

8.1.0 Jersey Library

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-jersey</artifactId>
</dependency>
```

Jersey is an open source framework for developing RESTful Web Services in Java. It is a reference implementation of the Java API for RESTful Web Services (JAX-RS) specification.

8.1.1 Hibernate/JPA

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

The Java Persistence API (JPA) is a Java application programming interface specification that describes the management of relational data in applications

using Java Platform, Standard Edition and Java Platform, Enterprise Edition. Persistence in this context covers three areas:

- the API itself, defined in the javax.persistence package
- the API itself, defined in the javax.persistence package
- object/relational metadata

8.1.2 JDBC Driver

```
<dependency>
   <groupId>com.oracle</groupId>
   <artifactId>ojdbc14</artifactId>
   <version>10.2.0.4.0</version>
</dependency>
```

8.1.3 Junit/Spring Test

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-test</artifactId>
   <scope>test</scope>
</dependency>
```

Spring Boot provides a number of utilities and annotations to help when testing your application. Test support is provided by two modules: spring-boot-test contains core items, and spring-boot-test-autoconfigure supports auto-configuration for tests. Most developers use the spring-boot-starter-test "Starter", which imports both Spring Boot test modules as well as JUnit, AssertJ, Hamcrest, and a number of other useful libraries.

Exception : Due to Oracle license restrictions, the Oracle JDBC driver is not available in the public Maven repository. To use the Oracle JDBC driver with Maven, you have to download and install it into your Maven local repository manually.

9 application.properties

Properties files are used to keep 'N' number of properties in a single file to run the application in a different environment. In Spring Boot, properties are kept in the application.properties file under the classpath.

The application.properties file is located in the src/main/resources directory. The code for this project's application.properties file is given below.

```
## use create when running the app for the first time
## then change to "update" which just updates the schema when
    necessary
spring.jpa.hibernate.ddl-auto=create
spring.jpa.database-platform=org.hibernate.dialect.Oracle10gDialect
spring.datasource.url= jdbc:oracle:thin:@localhost:1521:XE
spring.datasource.username=SYSTEM
spring.datasource.password=system
spring.datasource.driver-class-name=oracle.jdbc.driver.OracleDriver
## this shows the sql actions in the terminal logs
spring.jpa.show-sql=true
##optional, but just in case another application is listening on
    your default port (8080)
server.port = 8035
```

10 Annotations

10.0 JPA Annotations

- @Entity The @Entity annotation is used to specify that the class is an entity.

  ### Attributes:

  Name: The Name attribute is used to specify the entity name. It is an optional attribute. Defaults to the unqualified name of the entity class. It is contained in the javax.persistence package.

- @Table

  The Table annotation is used to specify the primary table for the annotated entity. Additional tables may be specified using SecondaryTable or SecondaryTables annotation.

```
@Entity
@Table(name = "EMPLOYEE")
public class Employee {
   @Id @GeneratedValue
   @Column(name = "id")
   private int id;
}
```

Attribute:

- Name: The name of the table. It is an optional attribute.
- Schema: The schema of the table. It is an optional attribute.
- UniqueConstraints: Unique constraints that are to be placed on the table. It is an optional attribute.
- Catalog: The catalog of the table. It is an optional attribute.
- Indexes: Indexes for the table. It is an optional attribute.

- @Column

  The Column annotation is used to specify the mapped column for a persistent property or field. If no Column annotation is specified, the default value will be applied.

  ### Attribute:

  - Name: The name of the column.
  - length: The column length.
  - nullable: Whether the database column is nullable.
  - Table: The name of the table that contains the column.
  - Unique: Whether the column is a unique key.
  - updatable: Whether the column is included in SQL UPDATE statements generated by the persistence provider.
  - Precision: The precision for a decimal (exact numeric) column.
  - insertable: Whether the column is included in SQL INSERT statements generated by the persistence provider.
  - columnDefinition: The SQL fragment that is used when generating the DDL for the column.

- @Id

  The @Id annotation is used to specify the primary key of an entity. The field or property to which the Id annotation is applied should be one of the following types:

  - 1. Any Java primitive type
  - 2. Any primitive wrapper type.
  - 3. String
  - 4. java.util.Date
  - 5. java.sql.Date
  - 6. java.math.BigDecimal
  - 7. java.math.BigInteger

- @GeneratedValue

  The @GeneratedValue annotation provides the specification of generation strategies for the primary keys values.
  Example :

```
@Id @GeneratedValue(strategy=GenerationType.TABLE ,
    generator="student_generator")
```

  - 1. Strategy: The strategy attribute is used to specify the primary key generation strategy that the persistence provider must use to generate the annotated entity primary key. It is an optional attribute. Strategy values are defined in javax.persistence.GeneratorType enumeration which are as follows:
    - 1. AUTO: Based on the database's support for primary key generation framework decides which generator type to be used.
    - 2. IDENTITY: In this case database is responsible for determining and assigning the next primary key.
    - 3. SEQUENCE: A sequence specify a database object that can be used as a source of primary key values. It uses @SequenceGenerator.
    - 4. TABLE: It keeps a separate table with the primary key values. It uses @TableGenerator. Note: Default value of Strategy attribute is AUTO.
  - 2. Generator: The Generator attribute is used to specify the name of the primary key generator to use as specified in the SequenceGenerator or TableGenerator annotation. It is an optional attribute. Please follow and like us: Save

10.1 Spring Framework Annotations

- @Repository

  We can use more suitable annotation that provides additional benefits specifically for DAOs i.e. @Repository annotation. The @Repository annotation is a specialization of the @Component annotation with similar use and functionality. In addition to importing the DAOs into the DI container, it also makes the unchecked exceptions (thrown from DAO methods) eligible for translation into Spring DataAccessException.

- @Autowired

- @Service

  The @Service annotation is also a specialization of the component annotation. It doesn't currently provide any additional behavior over the @Component annotation, but it's a good idea to use @Service over @Component

in service-layer classes because it specifies intent better. Additionally, tool support and additional behavior might rely on it in the future.

- @RequestController
- @RequestMapping
- @RequestMethod
- @PathVariable

11 Java Code

11.0 Problem Statement

Write a spring boot application to implement following methods on Employee object having ID(int),First Name(String) and Last Name(String) as fields using database.

- getAllEmployee - to fetch all employees.
- getEmployee - to fetch one employee by his ID,
- updateEmployee - to update fisrt name or last name
- deleteEmployee - to delete an employee by his ID,
- deleteAllEmployee- to delete all employee

11.1 Model

```
\\Employee.java
package com.atos.equens.worldline.SimpleSpringRestAPI.model;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "Employee")
public class Employee {

   @Column(name = "ID")
   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   private int id;
```

```
    @Column(name = "FirstName")
    private String firstName;
    @Column(name = "LastName")
    private String lastName;
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

11.2 DAO

```
\\EmployeeDAO.java
package com.atos.equens.worldline.SimpleSpringRestAPI.DAO;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import com.atos.equens.worldline.SimpleSpringRestAPI.model.Employee;

@Repository
public interface EmployeeDAO extends JpaRepository<Employee, Integer>{

}
```

11.3 Servcie

```
\\EmployeeService.java
package com.atos.equens.worldline.SimpleSpringRestAPI.service;

import java.util.List;
import java.util.Optional;
```

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.atos.equens.worldline.SimpleSpringRestAPI.DAO.EmployeeDAO;
import com.atos.equens.worldline.SimpleSpringRestAPI.model.Employee;

@Service
public class EmployeeService {

   @Autowired
   EmployeeDAO employeeDAO;

   public List<Employee> getAllEmployee(){
      return employeeDAO.findAll();
   }

   public Employee addEmployee(Employee employee) {
      return employeeDAO.save(employee);
   }

   public Optional<Employee> getEmployeeByID(int id) {
      return employeeDAO.findById(id);
   }

   public Employee updateEmployee(Employee employee) {
      return employeeDAO.save(employee);
   }

   public void deleteEmployeeById(int id) {
      employeeDAO.deleteById(id);
   }

   public void deleteAllEmployee() {
      employeeDAO.deleteAll();
   }
}
```

11.4 Controller

```
\\EmployeeController.java
package com.atos.equens.worldline.SimpleSpringRestAPI.controller;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;
```

```
import com.atos.equens.worldline.SimpleSpringRestAPI.model.Employee;
import
    com.atos.equens.worldline.SimpleSpringRestAPI.service.EmployeeService;

@RestController
@RequestMapping("/employee")
public class EmployeeController {

   @Autowired
   EmployeeService employeeService;

   @RequestMapping(value = "/getallemployees",
       method=RequestMethod.GET)
   public List<Employee> getAllEmployee(){
      return employeeService.getAllEmployee();
   }


   @RequestMapping(
      value = "/addemployee",
      method = RequestMethod.POST,
      consumes = MediaType.APPLICATION_JSON_VALUE,
      produces = MediaType.APPLICATION_JSON_VALUE
   )
   public Employee addEmployee(@RequestBody Employee employee){
      return employeeService.addEmployee(employee);
   }

   @RequestMapping(
      value = "/updateemployee",
      method = RequestMethod.PUT,
      consumes = MediaType.APPLICATION_JSON_VALUE,
      produces = MediaType.APPLICATION_JSON_VALUE
   )
   public Employee updateEmployee(@RequestBody Employee employee) {
      return employeeService.updateEmployee(employee);
   }

   @RequestMapping(value = "/{id}", method = RequestMethod.GET)
      public Optional<Employee> getEmployeeById(@PathVariable int id){
         return employeeService.getEmployeeByID(id);
   }

   @RequestMapping(value = "/deleteallemployee", method =
       RequestMethod.DELETE)
   public void deleteAllEmployee(){
      employeeService.deleteAllEmployee();
   }
```

```
    @RequestMapping(value = "/{id}", method = RequestMethod.DELETE)
    public void deleteEmployeeById(@PathVariable int id){
       employeeService.deleteEmployeeById(id);
    }

}
```

---

12 Testing with Rest Client [ARC]

13 Spring Boot Framework Plug-in Eclipse

14 Assignments for practice