Agenda:

1. Microservices Introduction & Architecture - Theoretical
2. ~~Introduction & Getting started with Spring Boot~~ - Practical
3. Using 12 Factors - Theory + Practical
4. ~~Giving your Microservice a REST~~ - Practical
5. Working with Spring Data - Practical
6. Creating Operations-Friendly Microservices using Spring Boot Actuator - Practical
7. Choreographing Microservices using Spring Cloud - Practical
8. Introduction to Docker - Theory
9. Building Docker images for Spring Boot Microservices - Practical

Break timings:

1. First break will be for 15 mins around 11:30 AM IST
2. Second break will for 45 mins around 01:15 PM IST
3. Third break will be for 15 mins around 03:45 PM ISt

# Microservices Introduction & Architecture

## Microservices Introduction

Micro – Small building block of a big application

Service – for easy accessibility

**Points:**

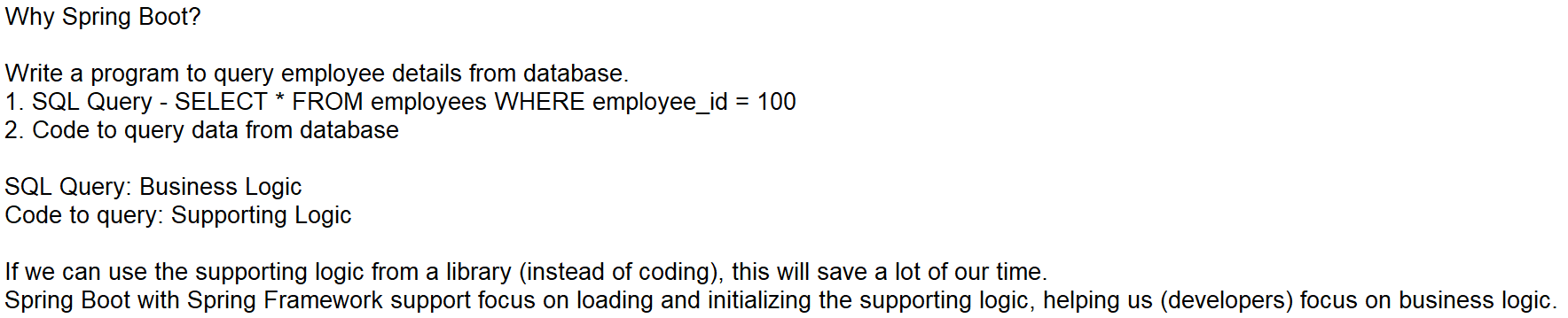
1. Microservices are based on Single Responsibility Principle (i.e. one functionality only)
2. Microservices are autonomous in nature (i.e. take care of their needs by themselves)
3. Microservices are based on polyglot architecture (i.e. different programming languages)

## Microservices Architecture

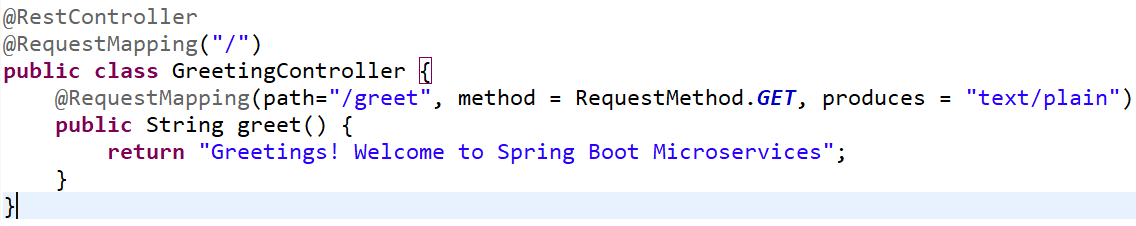
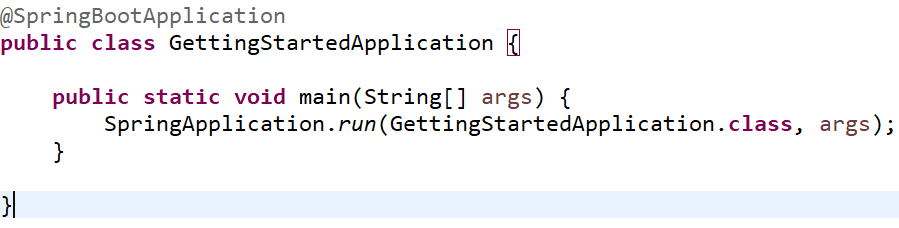
1. Data Layer
2. Conterization - Microservices reside
3. API Gateway

# 

# Spring Boot



## Getting started with Spring Boot



Execution Steps:

1. Annotation @SpringBootApplication is processed
   1. @SpringBootConfiguration → Requesting Spring Framework to create object of GettingStartedApplication class
   2. @ComponentScan → Searches for other classes with @Component annotation within current package ‘com.example.gettingstarted’
      1. The GreetingController class is loaded and instantiated because of the ‘@RestController’ annotation.  
         @RestController extends from @Controller extends from @Component  
         @RestController → @Controller → @Component
   3. @EnableAutoConfiguration → Configure the initialized objects (e.g. Embedded tomcat server is configured on 8080 port)
2. /greet URL is mapped to greet() method
3. Client sends the request to /greet URI which causes greet() method to execute and return response to client

@EnableAutoConfiguration ⇒ Job done by Spring Boot

@Component ⇒ Job done by Spring Framework

Spring Core initializes the objects

Spring Boot does the configuration of objects

## Practice 1.1 - Getting Started with Spring Boot

Usecase: Building a Greeting Service

1. Launch a browser and access <http://start.spring.io>
2. Provide the following details in the start screen

Project as Maven (Dependency management)

Language as Java

Version as 2.7.10

Group - com.example.training

Artifact - Greeting

Name - Greeting

Package name - com.example.training.greeting

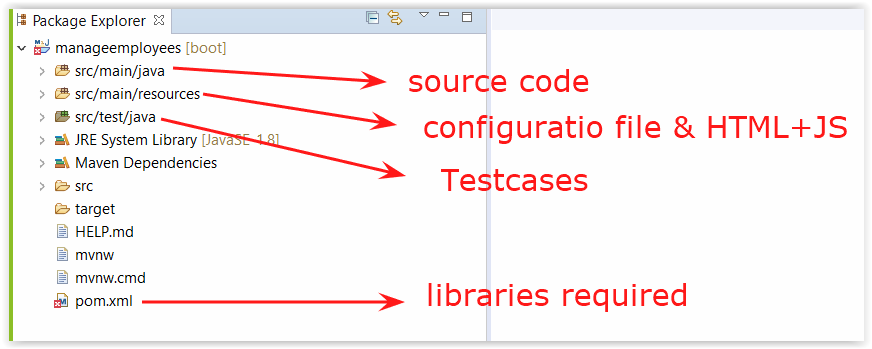
Packaging - JAR

Java - 8

1. Click on Add Dependencies button

Search for ‘Spring Web’ which includes Spring MVC and Tomcat server → Click on ‘Spring Web’

1. Click Generate
2. Extract ‘Greeting.zip’ to C:\labs\spring-labs
3. Launch the Eclipse (C:\Oracle\Softwares\eclipse) by double clicking on ‘eclipse.exe’
4. Provide the workspace location as C:\labs\spring-labs → Click Launch
5. Import Greeting project within eclipse (Please disconnect from VPN, as Maven downloads the libraries from internet)
   1. Click on File → Click Import
   2. Search for ‘Maven’ → Select ‘Existing Maven Projects’ → Click Next
   3. Click browse to locate C:\labs\spring-labs\Greeting folder → Click Select Folder
   4. Click Finish
6. Project structure is below



1. Create the GreetingController class with REST Endpoint for business methods
   1. Right click on ‘Greeting’ project within Eclipse
   2. Click New → Click Class
   3. Provide the class name as ‘GreetingController’ → Click Finish
2. Update GreetingController.java with below code

package com.example.training.greeting;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestMethod;

import org.springframework.web.bind.annotation.RestController;

@RestController

public class GreetingController {

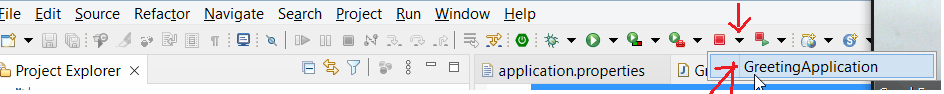
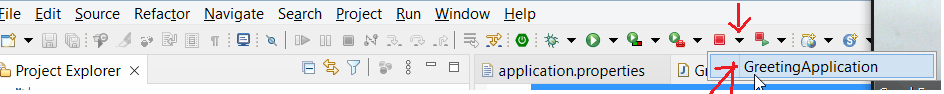
@RequestMapping(path="/greet", method = RequestMethod.GET)

public String greet() {

return "Greetings! Welcome to Spring Boot Microservice";

}

}

1. Configure Embedded Tomcat Server port as 9090
   1. Expand src\main\resources → Double click on application.properties
   2. Provide ‘server.port=9090’ (without quotes) within the file
2. Save the changes
3. Executing microservice project
   1. Approach #1:
      1. Expand ‘src\main\java\’ → Expand ‘com.example.training.greeting’
      2. Right-click on ‘GreetingApplication.java’ → Click Java Application
      3. Launch a browser and access <http://localhost:9090/greet> URL
      4. To stop the Microservice → Click on red square as described below  
         
   2. ~~Approach #2: available from Spring plugin~~
      1. ~~Right click on Greeting project → Click on Run As → Click on ‘Spring Boot App’~~
      2. ~~Launch a browser and access~~ [~~http://localhost:9090/greet~~](http://localhost:9090/greet) ~~URL~~
      3. ~~To stop the Microservice → Click on red square as described below  
         ~~
   3. Approach #3: package the application as JAR file and execute the JAR file
      1. Launch a new command prompt and execute below commands within

cd c:\labs\spring-labs\Greeting

set JAVA\_HOME="c:\oracle\softwares\jdk8"

mvnw package

cd c:\labs\spring-labs\Greeting\target

%JAVA\_HOME%\bin\java -jar Greeting-0.0.1-SNAPSHOT.jar

* + 1. Launch a browser and access the URL <http://localhost:9090/greet>
    2. To stop the microservice press Ctrl+C within command prompt
    3. Close the command prompt

## Annotations of Spring Core and Spring Boot

### Spring Boot Annotation

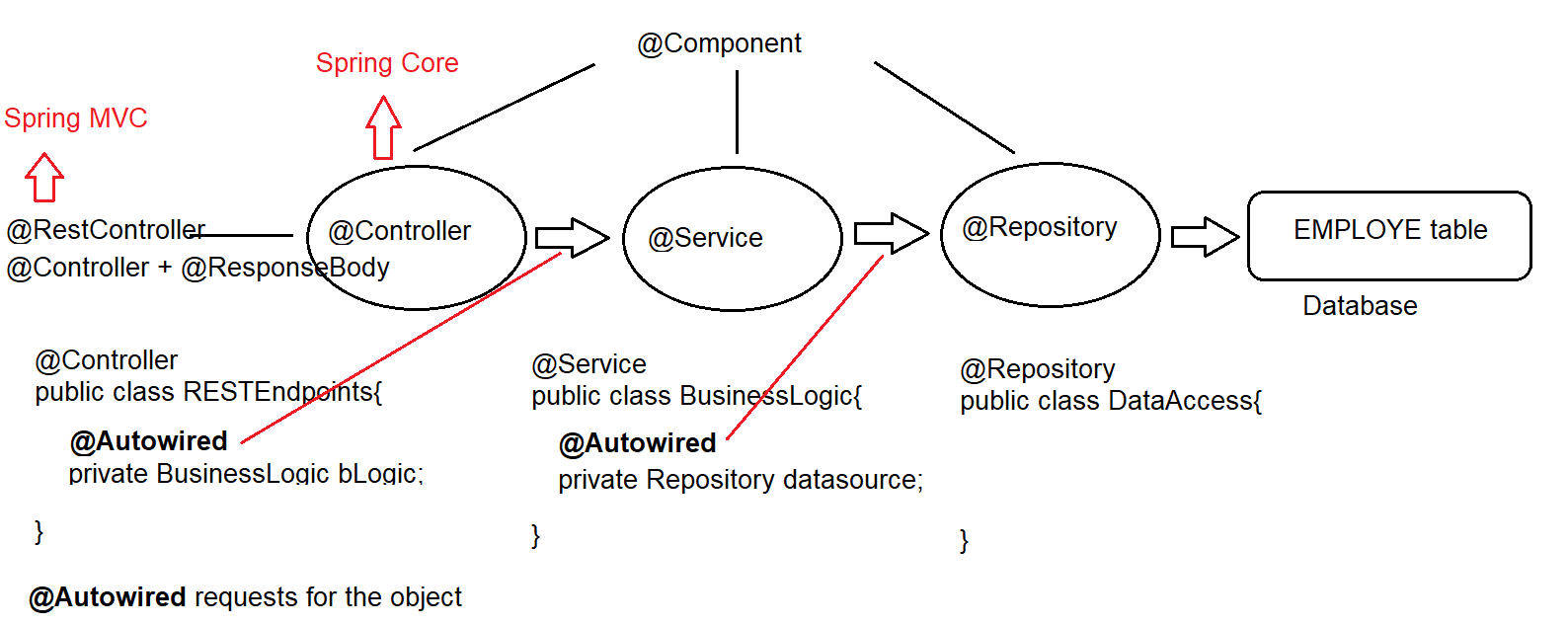
@EnableAutoConfiguration ⇒ Configure the initialized objects by using application.properties file (e.g. Configuring embedded server port as 9090)

### Spring Core Annotations

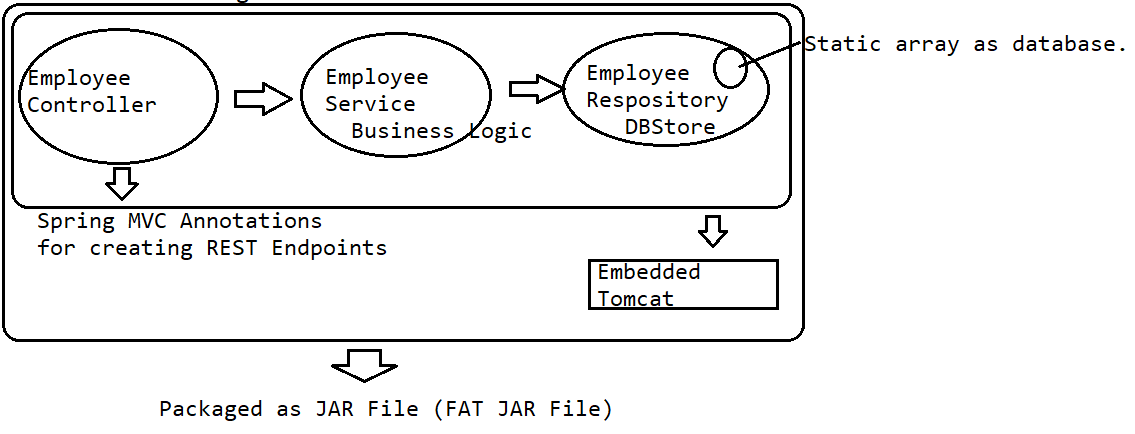
@Component / @Controller / @Service / @Repository ⇒ To create instance of current class

Usage of above annotations is mostly for documentation:

* @Component is used with any class to be instantiated by Spring Core
* @Controller is used for defining the REST Endpoints
* @Service is used for Business logic classes
* @Repository is used for data source related classes



## Practice 1.2 - ManageEmployee Service development



**Endpoints to implement (for your information):**

1. Create

* POST + <http://localhost:8080/employees>
* When creation is successful response should be “Create Successful” + HTTP Response code as 201

1. Read specific employee

* GET + <http://localhost:8080/employees>/{empId}
* When record is found then response should be Employee JSON + HTTP Response code as 200
* When record is not found then response should be “Employee Not Found” + HTTP Response code as 404

**Code Implementation:**

1. Create Employee.java
   1. Expand ‘src\main\java’ → right click on the package com.example.training.manageemployees → Click New → Click Class
   2. Enter the class name as Employee → Click Finish
   3. Update Employee.java to use below code

package com.example.training.manageemployees;

public class Employee {

private Integer empId;

private String empName;

private Double salary;

public Employee() {

}

public Employee(Integer eid, String eName, Double salary) {

this.empId = eid;

this.empName = eName;

this.salary = salary;

}

}

* 1. RIght click within java class → Click Source → Click Generate Getters and Setters
  2. Click Select All → Click Generate

1. Building EmployeeRepository.java class
   1. Expand ‘src\main\java’ → right click on the package com.example.training.manageemployees → Click New → Click Class
   2. Enter the class name as EmployeeRepository → Click Finish
   3. Update EmployeeRepository.java with below code

package com.example.training.manageemployees;

import java.util.ArrayList;

import java.util.List;

import org.springframework.stereotype.Component;

public class EmployeeRepository {

private static List<Employee> employeeData = new ArrayList<>();

static {

Employee james = new Employee(100, "James Cooper", 12345.0);

Employee steven = new Employee(200, "Steven King", 2345.0);

Employee neena = new Employee(300, "Neena Kocchar", 3456.0);

employeeData.add(james);

employeeData.add(steven);

employeeData.add(neena);

}

public boolean createEmployee(Employee newEmp) {

//Check if the employee is already present in memory array

for(Employee emp:employeeData) {

if(emp.getEmpId() == newEmp.getEmpId())

return false;//if present then return false

}

//if not present then add employee to memory & return true

employeeData.add(newEmp);

return true;

}

public Employee findEmployeeById(Integer empId) {

for(Employee emp:employeeData) {

if(emp.getEmpId().equals(empId))

return emp;

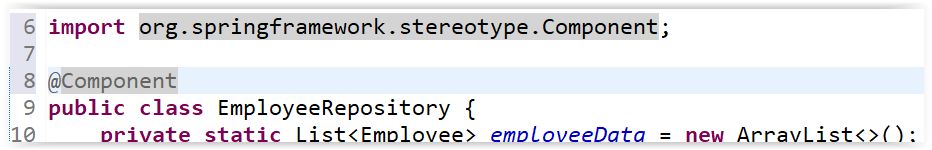
}

return null;

}

}

* 1. Instruct Spring Core to create a singleton object for EmployeeRepository class by using @Component annotation as below  
     Note you can use “@Repository” instead of @Component here.



* 1. Save the changes done to EmployeeRepository.java file

1. Create EmployeeService.java
   1. Expand ‘src\main\java’ → right click on the package com.example.training.manageemployees → Click New → Click Class
   2. Enter the class name as EmployeeService → Click Finish
   3. Update EmployeeService.java with below code

*import org.springframework.beans.factory.annotation.Autowired;*

*import org.springframework.stereotype.Service;*

*@Service //Requesting Spring Core to create object of Employee Service*

*public class EmployeeService {*

*@Autowired //Requesting Spring Core for the object of employee repository.*

*EmployeeRepository repository;*

*public boolean createEmployee(Employee newEmp) {*

*return repository.createEmployee(newEmp);*

*}*

*public Employee findEmployeeById(Integer empId) {*

*return repository.findEmployeeById(empId);*

*}*

*}*

* 1. Save the changes done to EmployeeService.java file

1. Create EmployeeController.java file
   1. Expand ‘src\main\java’ → right click on the package com.example.training.manageemployees → Click New → Click Class
   2. Enter the class name as EmployeeController → Click Finish
   3. Update EmployeeController.java with below code

*import org.springframework.beans.factory.annotation.Autowired;*

*import org.springframework.http.HttpStatus;*

*import org.springframework.http.MediaType;*

*import org.springframework.http.ResponseEntity;*

*import org.springframework.web.bind.annotation.GetMapping;*

*import org.springframework.web.bind.annotation.PathVariable;*

*import org.springframework.web.bind.annotation.PostMapping;*

*import org.springframework.web.bind.annotation.RequestBody;*

*import org.springframework.web.bind.annotation.RestController;*

***@RestController //Marking this class as Spring MVC Controller***

*public class EmployeeController {*

***@Autowired //Requesting for EmployeeService object to be injected***

***private EmployeeService service;***

*//POST + http://.../employees*

***@PostMapping(path = "/employees", consumes = MediaType.APPLICATION\_JSON\_VALUE)***

***public ResponseEntity<String> createEmployee(@RequestBody Employee emp) {***

*ResponseEntity response = null;*

*boolean createResult = service.createEmployee(emp);*

*if(createResult == true) {*

*response = new ResponseEntity("Create successful", HttpStatus.CREATED); //returning string + 200*

*return response;*

*}else {*

*response = new ResponseEntity("Create Failed", HttpStatus.BAD\_REQUEST); //returning string + 400*

*return response;*

*}*

*}*

***@GetMapping(path="/employees/{empId}")***

***public ResponseEntity getEmployee(@PathVariable("empId")Integer empId) {***

*ResponseEntity response = null;*

*Employee emp = service.findEmployeeById(empId);*

*if(emp != null) {*

*response = new ResponseEntity(emp, HttpStatus.FOUND); //returning string + 200*

*return response;*

*}else {*

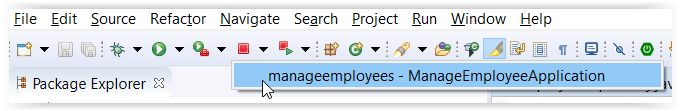
*response = new ResponseEntity("Employee Not Found", HttpStatus.NOT\_FOUND); //returning string + 200*

*return response;*

*}*

*}*

*}*

* 1. Starting the microservice
     1. Executing microservice using ‘spring-boot’ plugin  
        Right click on project → Click on Run As → Click on Spring Boot App
     2. Executing microservice using ‘Main’ class  
        Right click on ManageEmployeeApplication.java → Click on Run As → Click on Java Application  
        Note: ManageEmployeeApplication.java has a ‘public static void main’ method in it.
     3. Executing microservice by generating the JAR file
        1. Launch the command prompt
        2. Change the current directory to ‘c:\labs\spring-labs\manageemployees’  
           *cd c:\labs\spring-labs\manageemployees*
        3. Execute maven wrapper with package goal as below *c:\labs\spring-labs\manageemployees> mvnw package*  
           Notice, that FAT JAR is built in the target folder.
        4. Execute the JAR by using below commands  
           *set JAVA\_HOME=c:\Oracle\Softwares\JDK8  
           %JAVA\_HOME%\bin\java -version  
           %JAVA\_HOME%\bin\java -jar target\manageemployees-0.0.1-SNAPSHOT.jar*
  2. Testing microservice
     1. Launch the browser and access <http://localhost:8080/employees/100> to access employee with id 100.
     2. Output should be as below  
        
  3. Stopping the Microservice  
     Click on the arrow beside Red Square → Click on Microservice name  
     

# Giving Microservices a REST

REST = Java API + HTTP Protocol + Guidelines

## HTTP Protocol

Is to access the resources on the Web (Internet/Intranet). The resource can be an API or a Web Page

Terminology:

1. **HTTP URL** of the Resource: [http://hostName:portNumber/URI](about:blank)  
   Example: <https://docs.oracle.com/en/database/oracle/oracle-rest-data-services/>

https ⇒ Secure HTTP protocol

docs.oracle.com ⇒ Host name

80 as default port for HTTP protocol

/en/database/oracle/oracle-rest-data-services/ ⇒ URI for the resource

1. **HTTP URI** (Uniform Resource Identifier): a piece of string identifying the resource.  
   From above example: [/en/database/oracle/oracle-rest-data-services/](https://docs.oracle.com/en/database/oracle/oracle-rest-data-services/)

A URI can be composed of multiple URIs

* /en ⇒ indicates we are working with English language
* /database/oracle ⇒ Indicates we are accessing oracle database related feature
* /oracle-rest-data-service ⇒ Indicates the feature of Oracle database we are referring

1. **HTTP Request:** Sent by HTTP Client to the Server
   1. HTTP Request Header: Data for the server/application
   2. HTTP Request Body: Data for the application
   3. HTTP URL: The resource to access
   4. HTTP Method: The operation to be performed
      1. GET
      2. POST
      3. PUT
      4. DELETE

Example: Accessing the URL <https://docs.oracle.com/en/database/oracle/oracle-rest-data-services/> from browser

Sending HTTP GET request to access /en/data/oracle/oracle-rest-data-services resource from host docs.oracle.com accessible from 80 port.

docs.oracle.com returns the HTTP Response object.

1. **HTTP Response:** Returned by the Server as response to HTTP Client’s request.
   1. HTTP Response Header: Data for the browser
   2. HTTP Response data: Data for HTTP Client (user of browser)
   3. HTTP Response Code: 1xx, 2xx, 3xx, 4xx, 5xx  
      Example: 404 is the response code indicating resource not found
      1. 1xx ⇒ Server has returned some information to client
      2. 2xx ⇒ Server informing client that request is processed successfully
      3. 3xx ⇒ Server informing client about redirection (i.e. URI has changed)
      4. 4xx ⇒ Server informing client about invalid request (example: the URI is incorrect, and it doesn’t point to any resource)
      5. 5xx ⇒ Server informing client about an error at server side

Examples:

Client sent GET request to <http://example.com:80/employees/100>  
Consider, the server returned the response with response code as 403.

Question: Is the request processed successfully? If no, then whose mistake is it (i.e. Server or Client).

Answer: Request is not processed, because of a mistake from Client

## REST Concept

Applying HTTP protocol features on Java API, so that Java API is accessible using HTTP Protocol.

**Guidelines about HTTP Methods:**

1. HTTP GET should invoke Query API
2. HTTP POST should invoke Create API
3. HTTP PUT should invoke Update API
4. HTTP DELETE should invoke Remove API

Example:

1. **HTTP request to access employee whose ID is 200**

HTTP Method + HTTP URL (hostname, port number, URI)

HTTP Method: GET

Host Name: localhost

Port Number: 8080

URI: /employees/100

HTTP Request ⇒ <http://localhost:8080/employees/100> + HTTP GET

HTTP Response ⇒ Employee Data + HTTP Response Code

## Practice 2.1 - Developing REST Services

Location: Google Drive \ Practicals

Document: Practice - Product Micro Service.pdf

As the browser supports only GET operation, we need a new HTTP Client called ‘PostMan’

**Please use the ‘**[**http://start.spring.io**](http://start.spring.io)**’ instead of Eclipse plugin to create the project and import it as Maven project within Eclipse.**

**Use the dependencies: ‘Spring Boot DevTools’ and ‘Spring Web’. Also, use the version as 2.7.10**

**Installing POSTMan:**

1. Install the Chrome Browser
2. Install Postman Extension for Chrome (<https://chrome.google.com/webstore/detail/postman/fhbjgbiflinjbdggehcddcbncdddomop?hl=en>)
3. Click on ‘Add to Chrome’
4. Restart the browser
5. Browse the URL [http://chrome://apps](about:blank) ⇒ Click on PostMan
6. Skip the Registration → Close the first two dialogs
7. Provide the URL and choose the operation

# Working with Spring Data

Interaction with data sources

1. Structured Data storage (Relational Data Source - Database (Derby Database / Oracle Database))
2. Unstructured Data (e.g. Images, JSON files etc) - NoSQL databases (e.g. MongoDB)

## Practice 3.1 - Spring Data JPA

**Summary Steps:**

1. Step 1: Configuring Derby Database
2. Step 2: Build Employee Spring Boot service
3. Step 3: Add Spring Data JPA Support to Employee service
4. Step 4: Testing

### Step 1: Configuring Derby Database

1. Download Derby database <https://dlcdn.apache.org//db/derby/db-derby-10.14.2.0/db-derby-10.14.2.0-bin.zip>
2. Extract this zip file to​ C:\Oracle\Softwares folder
3. Start the Derby Database
   1. Open the directory using Windows explorer ‘C:\Oracle\softwares\db-derby-10.14.2.0-bin\bin’
   2. Launch the Derby Server by double clicking on ‘startNetworkServer.bat’ from C:\Oracle\softwares\db-derby-10.14.2.0-bin  
      Note: Please don’t close this command prompt as it will shutdown the database.
4. Launch SQL prompt for Derby Database
   1. Start new Command Prompt
   2. Create DERBY\_HOME environment variable

set DERBY\_HOME=C:\Oracle\softwares\db-derby-10.14.2.0-bin

set JAVA\_HOME=C:\Oracle\Softwares\JDK8

* 1. Launch the SQL prompt by executing below command

cd C:\Oracle\softwares\db-derby-10.14.2.0-bin\bin

%JAVA\_HOME%\bin\java -jar %DERBY\_HOME%\lib\derbyrun.jar ij

This should give us the ‘ij>’ prompt

* 1. Create the derby database and connect to it

CONNECT 'jdbc:derby://localhost:1527/firstdb;create=true';

### 

### Step 2: Build Employee Spring Boot service

1. Create Employees table
   1. Change focus to Derby SQL prompt ‘command window’
   2. Create EMPLOYEE table using below code

CREATE TABLE EMPLOYEES

(

EMP\_ID INT PRIMARY KEY,

EMP\_SALARY DOUBLE,

EMP\_NAME CHAR(30),

EMP\_JOB CHAR(30),

EMP\_DEPARTMENT CHAR(30)

);

* 1. Create few employee records using the below “INSERT” statement and verify the same.

INSERT INTO EMPLOYEES VALUES (100, 12345, 'James Cooper', 'Trainer', 'Training');

INSERT INTO EMPLOYEES VALUES (200, 12345, 'Steven King', 'Manager', 'Training');

SELECT \* FROM EMPLOYEES;

1. Create a new Project named ‘EmployeeMS’ (Disconnect from Oracle VPN)
   1. Browse http://start.spring.io
   2. Enter the Project name as “EmployeeMS” → Java must be 8
   3. Dependencies to be added are : ‘Spring Web’, ‘Spring Boot DevTools’ and ‘Spring Data JPA’ dependencies
      1. Spring Web → Provides us ‘Spring MVC’ and ‘Tomcat Container’
      2. Spring Data JPA → Provides us Spring Data library
      3. Spring Boot DevTools → Library auto restarts the microservice when source code changes.
   4. Click Generate
2. Create Employee.java and EmployeeController.java
   1. Create Employee.java such that it represents the database table EMPLOYEE
      1. Right click on ‘src\main\java’ → Click on New → Click Class
      2. Enter the Class name as ‘Employee’ and package as ‘com.example.training’
      3. Update the code as below

public class Employee {

private Integer empId;

private Double empSalary;

private String empName;

private String empJob;

private String empDepartment;

}

* + 1. Map the Employee.java class with EMPLOYEE table in database using JPA Annotations  
       Note: JPA annotations are from javax.persistence package.

@Entity

@Table(name = "EMPLOYEES")

public class Employee {

@Id

@Column(name = "EMP\_ID")

private Integer empId;

@Column(name = "EMP\_SALARY")

private Double empSalary;

@Column(name = "EMP\_NAME")

private String empName;

@Column(name = "EMP\_JOB")

private String empJob;

@Column(name="EMP\_DEPARTMENT")

private String empDepartment;

}

* + 1. Generate the Getters and Setters for the members of Employee.java class

Right click within Employee.java → Click on Source → Click on Generate Getters and Setters

Click Select All → Click Generate

* 1. Create the EmployeeController.java code
     1. Right click on ‘src\main\java’ → Click New → Click Class
     2. Enter Class name as ‘EmployeeController’ → Click Finish
     3. Update ‘EmployeeController.java’ to use below code

*import org.springframework.http.ResponseEntity;*

*import org.springframework.web.bind.annotation.GetMapping;*

*import org.springframework.web.bind.annotation.PathVariable;*

*import org.springframework.web.bind.annotation.RestController;*

*@RestController*

*public class EmployeeController {*

*@GetMapping("/employees")*

*public Iterable<Employee> getAllEmployees(){*

*return null;*

*}*

*@GetMapping("/employees/{empId}")*

*public ResponseEntity<Employee> getEmployeeById(@PathVariable("empId")Integer employeeId){*

*return null;*

*}*

*}*

### Step 3: Add Spring Data Support to Employee service

1. Ensure you have the dependency for ‘spring-data-jpa’ within build.gradle/pom.xml

*<dependency>*

*<groupId>org.springframework.boot</groupId>*

*<artifactId>spring-boot-starter-data-jpa</artifactId>*

*</dependency>*

1. Create EmployeeRepository interface by extending CrudRepository Interface
   1. Right click on ‘src\main\java’ → Click New → Click Interface
   2. Enter the name as ‘EmployeeRepository’ → Click on Add
   3. Search for ‘CrudRepository’ → Select ‘CrudRepository’ → Click Add → Click Ok
   4. Click Finish
   5. Within the interface EmployeeRepository.java replace T with Employee, ID with Integer
   6. Completed code is as below

*import org.springframework.data.repository.CrudRepository;*

*public interface EmployeeRepository extends CrudRepository<Employee, Integer> {*

*}*

1. Let us use EmployeeRepository inside EmployeeController to interact with the database.
   1. Open EmployeeController.java
   2. Update EmployeeController.java to add ‘EmployeeRepository’ reference with dependency injection.

@RestController

public class EmployeeController {

@Autowired

EmployeeRepository repository;

…

}

1. Update the method to use repository reference.

*import java.util.Optional;*

*import org.springframework.beans.factory.annotation.Autowired;*

*import org.springframework.http.HttpStatus;*

*import org.springframework.http.ResponseEntity;*

*import org.springframework.web.bind.annotation.GetMapping;*

*import org.springframework.web.bind.annotation.PathVariable;*

*import org.springframework.web.bind.annotation.RestController;*

*@RestController*

*public class EmployeeController {*

*@Autowired*

*EmployeeRepository repository;*

*@GetMapping("/employees")*

*public Iterable<Employee> getAllEmployees(){*

*return repository.findAll();*

*}*

*@GetMapping("/employees/{empId}")*

*public ResponseEntity<Employee> getEmployeeById(@PathVariable("empId")Integer employeeId){*

*//Check if employee exists in database for given employeeId*

*Optional<Employee> data = repository.findById(employeeId);*

*if(data.isPresent()) {*

*ResponseEntity<Employee> response = new ResponseEntity(data.get(), HttpStatus.OK);*

*return response;*

*}else {*

*ResponseEntity<Employee> response = new ResponseEntity(data.get(), HttpStatus.NOT\_FOUND);*

*return response;*

*}*

*}*

*}*

1. Configure the Spring-Data-JPA to interact with our database
   1. Open ‘src\main\resources\application.properties’, then update it to add below contents.

*server.port=9090*

*spring.datasource.driver-class-name=org.apache.derby.jdbc.ClientDriver*

*spring.datasource.url=jdbc:derby://localhost:1527/firstdb;create=false*

*spring.jpa.database=derby*

*spring.jpa.hibernate.ddl-auto=update*

*spring.jpa.generate-ddl=false*

*spring.jpa.show-sql=true*

* 1. Because we are going to rely on ‘org.apache.derby.jdbc.ClientDriver’ resource, we need its library

Hence, update build.gradle or pom.xml to add the dependency for ‘ClientDrive’ library

* 1. Incase of Gradle use below steps
     1. Update build.gradle to add below dependency  
        implementation 'org.apache.derby:derbyclient'
     2. Right on project → Click Gradle → Click Refresh Gradle Project
     3. Verify if ‘derbyclient.jar’ is added within ‘Project and External dependencies’
  2. Incase of Maven use below steps
     1. Open pom.xml and add below dependency

<dependency>

<groupId>org.apache.derby</groupId>

<artifactId>derbyclient</artifactId>

</dependency>

* + 1. Save the changes done to pom.xml
    2. Right click on ‘EmployeeMS’ project → Click Maven → Click Update Project
    3. Verify from Maven dependencies that ‘derbyclient’ jar is added.

### Step 4: Testing

1. Right click on EmployeeMS project → Click Run As → Click Spring Boot App
2. Using Postman access below resource

URL: ​http://localhost:9090/employees

Method: GET

### Step 5: Create Functionality

1. Implementing the Create functionality

@PostMapping("/employees")

public ResponseEntity<String> createEmployee(@RequestBody Employee emp){

try {

repository.save(emp);

ResponseEntity<String> response = new ResponseEntity<>("Employee Created", HttpStatus.CREATED);

return response;

} catch (Exception e) {

ResponseEntity<String> response = new ResponseEntity<>(e.getMessage(), HttpStatus.INTERNAL\_SERVER\_ERROR);

return response;

}

}

1. Test the create functionality.

## Do It Yourself

1. Implement the Update - *repository.save()* - Do this using @PutMapping(“/employees/{empid}”)

Update should update the record in database

1. Implement the Delete functionality - *repository.save(id)* - Do this using @DeleteMaping(“/employees/{empid}”)

Delete should not delete record but mark it as RESIGNED

1. Update the database EMPLOYEE table to add RESIGNED column
2. Update Employee.java accordingly
3. Update EmployeeController.java to have @DeleteMapping which marks the record as RESIGNED.

Let us conclude our discussion for today.