Spring Boot

standard chartered

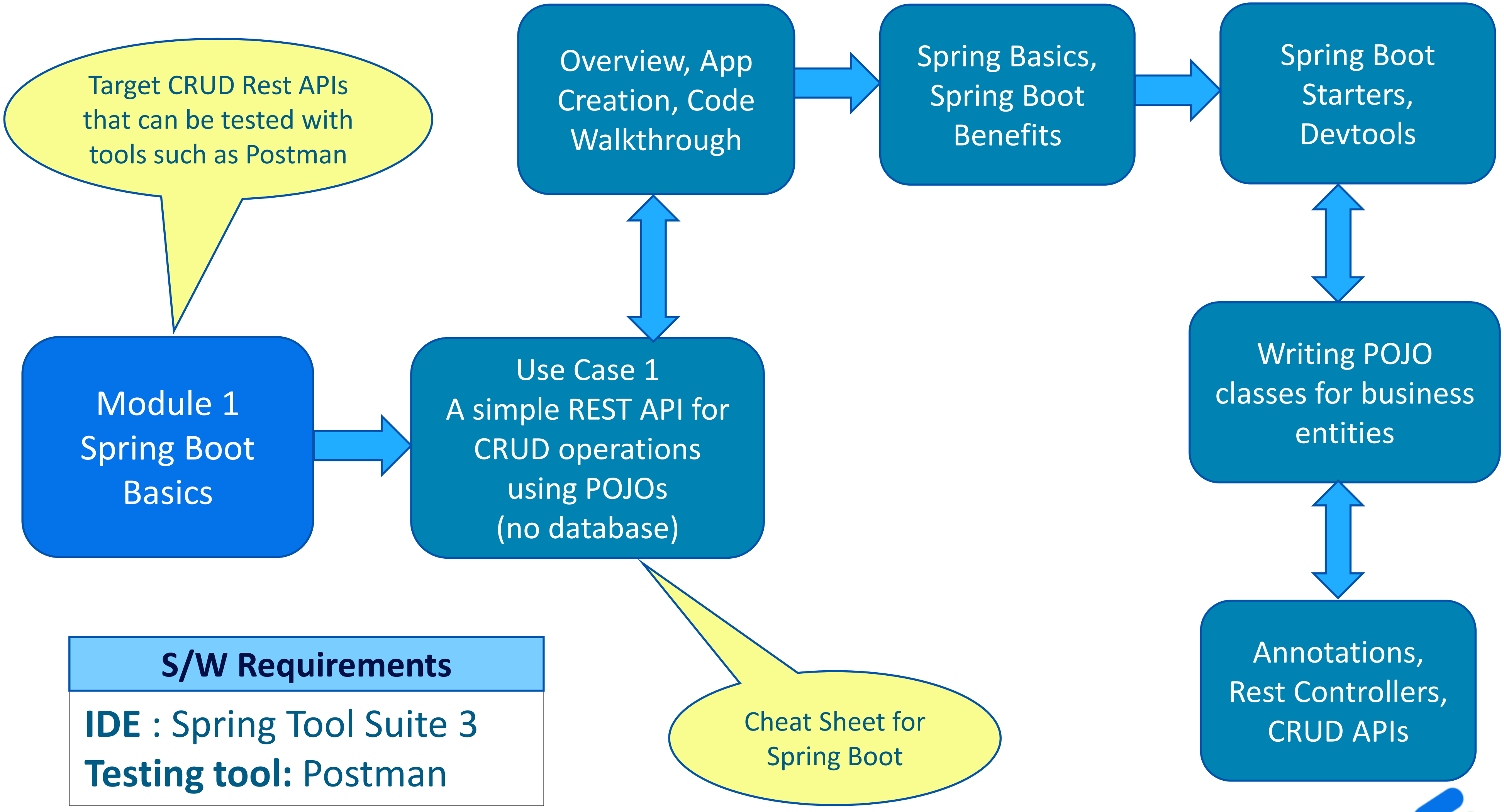
axess academy

Module 1: Spring Boot Basics

INTERNAL

Module 1 Design

axess academy


Target CRUD Rest APIs that can be tested with tools such as Postman
Overview, App Creation, Code Walkthrough
Spring Basics, Spring Boot Benefits
Spring Boot Starters, Devtools
Module 1 Spring Boot Basics
Use Case 1 A simple REST API for CRUD operations using POJOs (no database)
Writing POJO classes for business entities
Annotations, Rest Controllers, CRUD APIs
Cheat Sheet for Spring Boot

S/W Requirements
IDE : Spring Tool Suite 3 **Testing tool:** Postman

Module 1 Overview

- Overview of Spring Framework and Spring Boot
- Getting started with a Spring Boot app (Creation, Running) etc
- Code walkthrough of a hello world app
- Basics of Spring framework
- Spring Modules Overview
- Spring to Spring Boot Evolution – In Detail

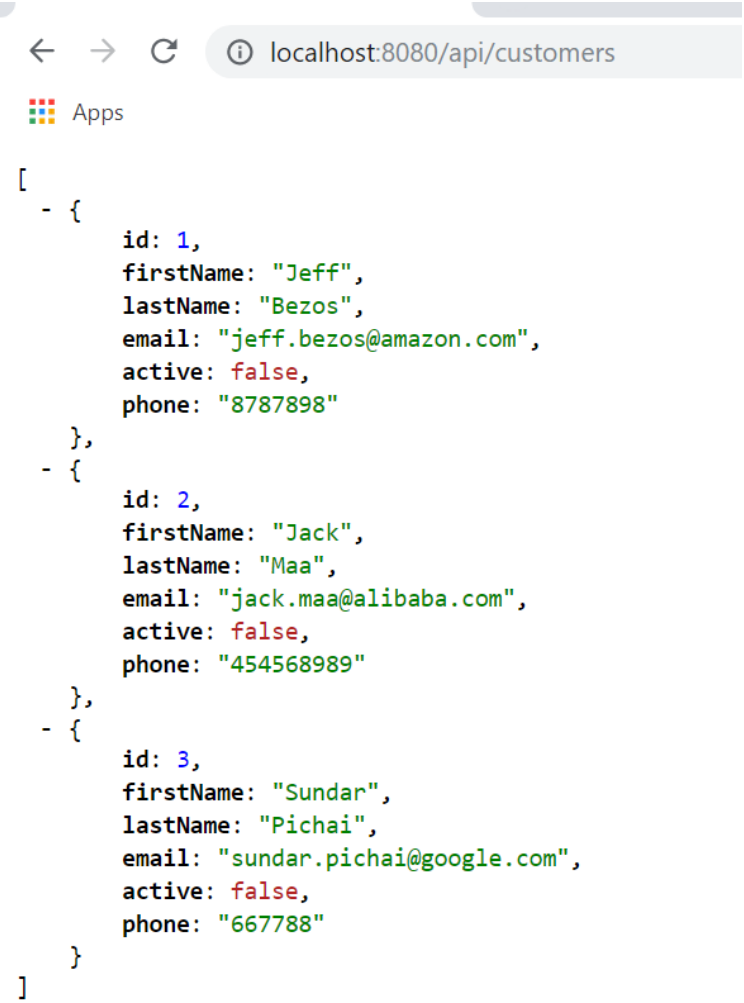
Use case 1

- As part of the use case 1, you have to build a simple CRUD based REST API for dealing with customers data
- In this uses case 1, no database need to be used
- We will only use in-memory data structures such as a arrays or list to store and manipulate data

- Use/Define a customer model/entity as shown
- There should be REST APIs built for
  - Listing all customers
  - Getting one customer details
  - Creating a new customer
  - Updating an existing customer
  - Deleting an existing customer

Customer Model
id - number
firstName - string
lastName - string
email - string
phone - string
active - string
password – string
role - string

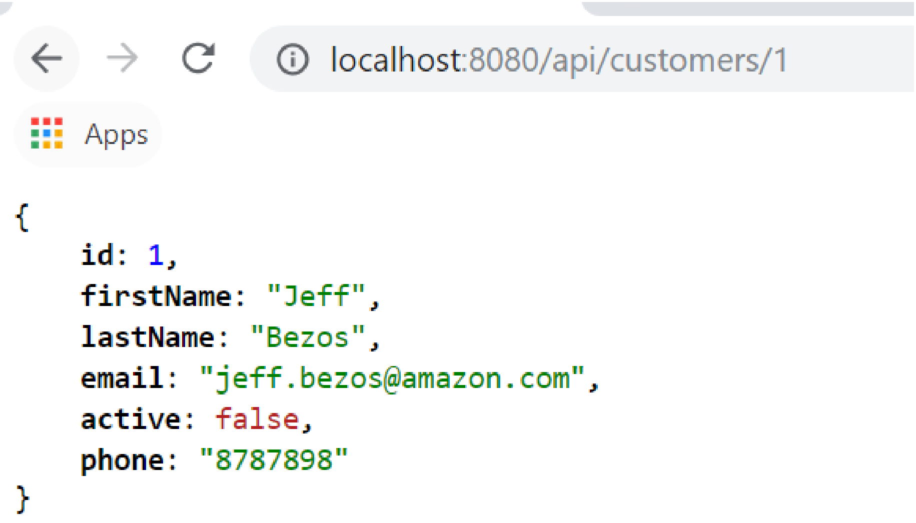
Use Case 1 – Output – GET All Customers



```
[
  - {
        id: 1,
        firstName: "Jeff",
        lastName: "Bezos",
        email: "jeff.bezos@amazon.com",
        active: false,
        phone: "8787898"
    },
  - {
        id: 2,
        firstName: "Jack",
        lastName: "Maa",
        email: "jack.maa@alibaba.com",
        active: false,
        phone: "454568989"
    },
  - {
        id: 3,
        firstName: "Sundar",
        lastName: "Pichai",
        email: "sundar.pichai@google.com",
        active: false,
        phone: "667788"
    }
]
```

- The REST API for **/api/customers** should return the output as shown in the left diagram
- This demonstrates a typical HTTP GET operation on the url **http://localhost:8080/api/customers**
- The response should be a JSON, that contains an array of customers

Use Case 1 – GET One Customer Details



```
{
    id: 1,
    firstName: "Jeff",
    lastName: "Bezos",
    email: "jeff.bezos@amazon.com",
    active: false,
    phone: "8787898"
}
```

- The REST API for **/api/customers/1** should return the output as shown in the left diagram
- This demonstrates a typical HTTP GET operation on the url **http://localhost:8080/api/customers/1**
- The response should be a JSON, that contains a single customer details

Use case 1 – Create, Update and Delete

- The previous 2 diagrams explained only the read operations (GET all, GET one)
- But the use case should also support the create, update and delete operation
- There should be REST APIs available for each one of the following
  - Create a new customer
  - Update an existing customer
  - Delete an existing customer
- The below table explains the typical Create, Update and Delete operations involved in a REST API

Action	HTTP method	Target URL	Request Body
Create	POST	/api/customers	Customer data
Update	PUT	/api/customers/123	Customer data
Delete	DELETE	/api/customers/123	None

Module 1: Spring Boot Basics

What is Spring Boot

- Spring Boot is an open-source framework that makes it easy to create stand-alone, production-grade Spring based applications
- Spring Boot takes an opinionated view of the Spring platform and 3rd party libraries
- Spring Boot is developed on top of the most successful Spring framework/ecosystem
- It is developed by the team at a company named Pivotal
- Spring Boot is used at big brands such as at big brands like Netflix, Alibaba etc. for developing backend services/microservices

Spring Boot – Background & Evolution

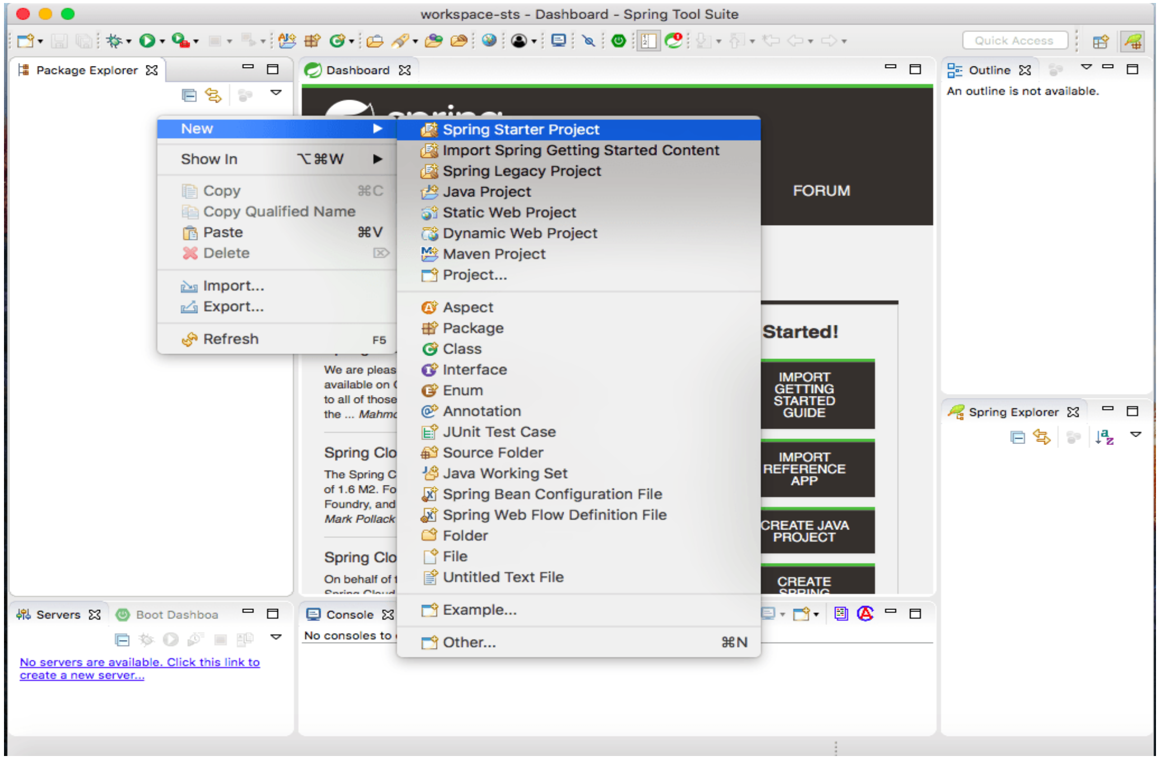
- Spring Boot is the successor to Spring framework
- Spring framework was one of the most successful enterprise application development framework
- But Spring had lots of manual configurations and wirings up that needed to be done
- So, Spring Boot was developed to simplify spring programming

Creating a Spring Boot Project

- There are 3 ways to get started with a spring boot project
  - Through IDEs such as Spring Tool Suite 3 or Eclipse with plugins or IntelliJ Idea
  - Through the web page https://start.spring.io/
  - Through the command line utilities
- The most common IDE used for creating spring boot app is STS (Spring Tool Suite)
- In the coming slides, we will see how to create a spring boot project through various means

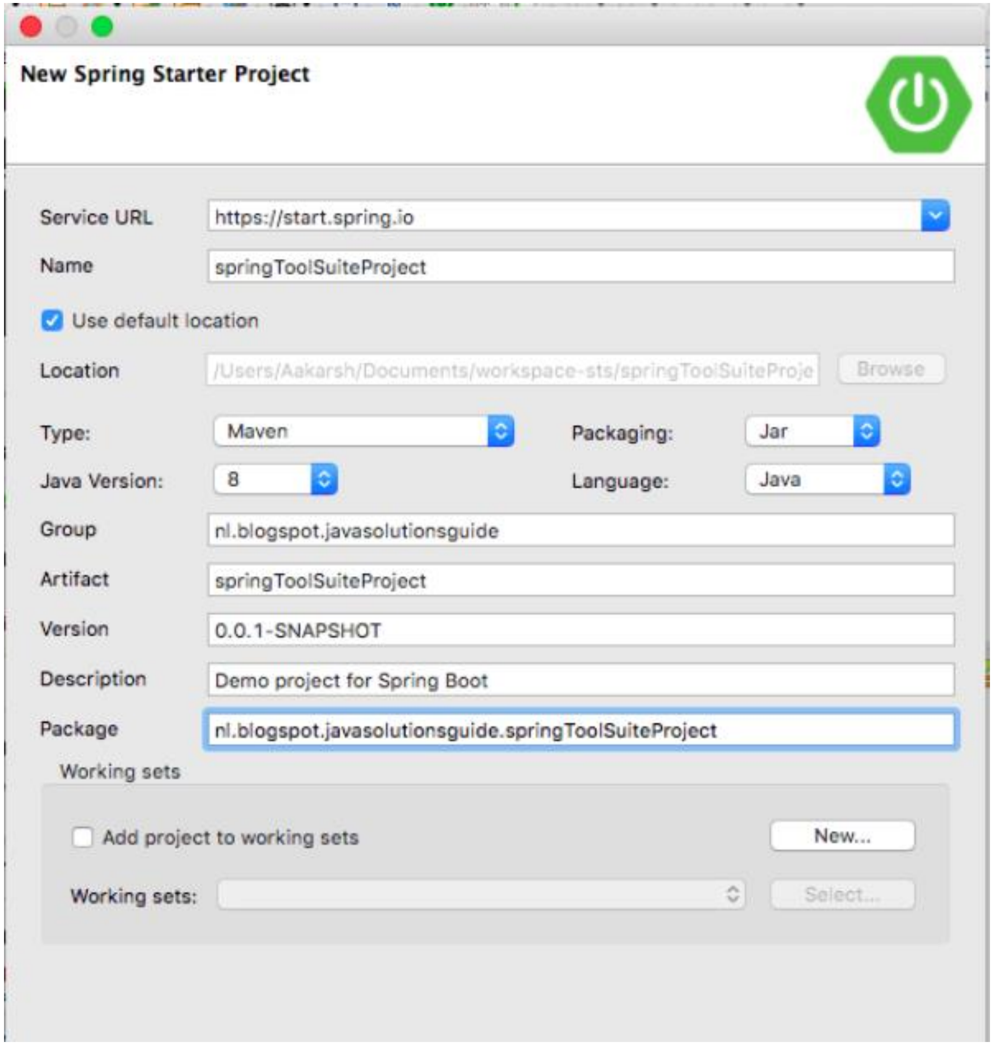
Spring Boot Project through STS

- After STS is ready, right click on package explorer, then New -> Spring Starter Project
- This can be invoked through File -> New -> Spring Starter Project also



Ref:- https://www.javacodegeeks.com/2018/07/spring-boot-project-sts.html

STS – Project details screen



New Spring Starter Project

Service URL: https://start.spring.io

Name: springToolSuiteProject

☑ Use default location

Location: /Users/Aakarsh/Documents/workspace-sts/springToolSuiteProje

Type: Maven

Packaging: Jar

Java Version: 8

Language: Java

Group: nl.blogspot.javasolutionsguide

Artifact: springToolSuiteProject

Version: 0.0.1-SNAPSHOT

Description: Demo project for Spring Boot

Package: nl.blogspot.javasolutionsguide.springToolSuiteProject

Working sets

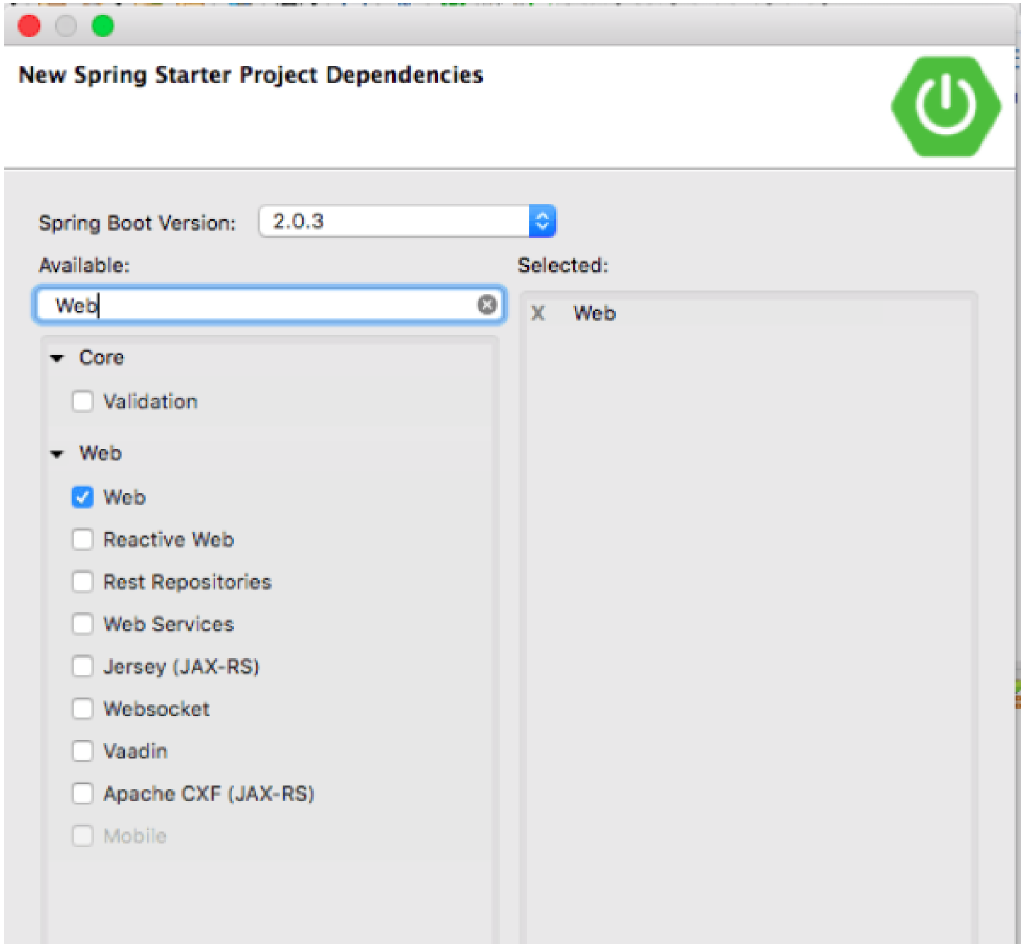
□ Add project to working sets

Working sets:

- Next, In the project details screen, enter details such as the project name, group id, artifact id and the package name
- We can also specify other details such as Java version, build tool(maven or gradle), packaging

Ref:- https://www.javacodegeeks.com/2018/07/spring-boot-project-sts.html

STS – Adding Project Dependencies



New Spring Starter Project Dependencies

Spring Boot Version: 2.0.3

Available: Web

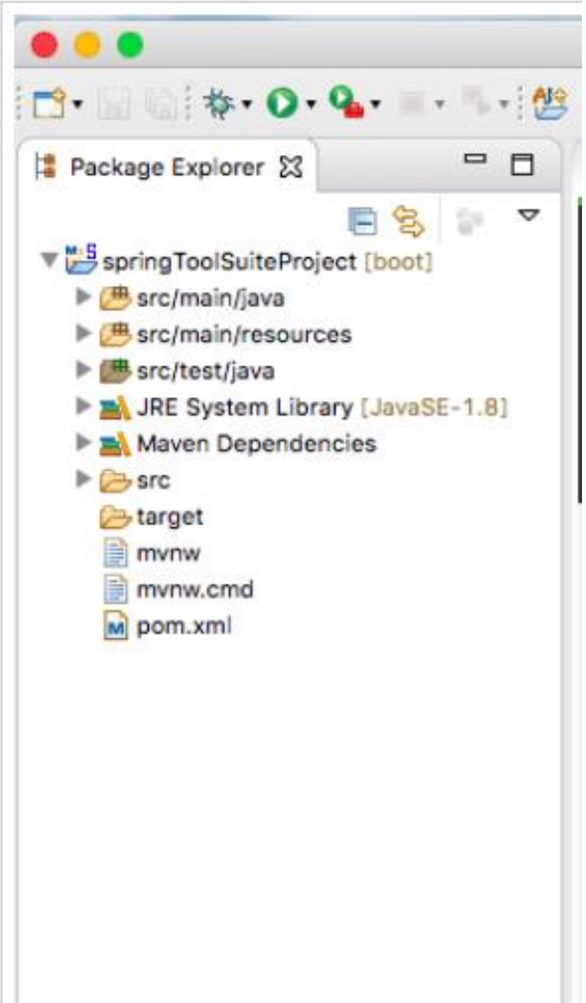
Selected:

X Web

- In the dependencies screen, we can choose all the dependencies that the project needs
- We can choose dependencies such as web, jdbc, jpa etc

Ref:- https://www.javacodegeeks.com/2018/07/spring-boot-project-sts.html

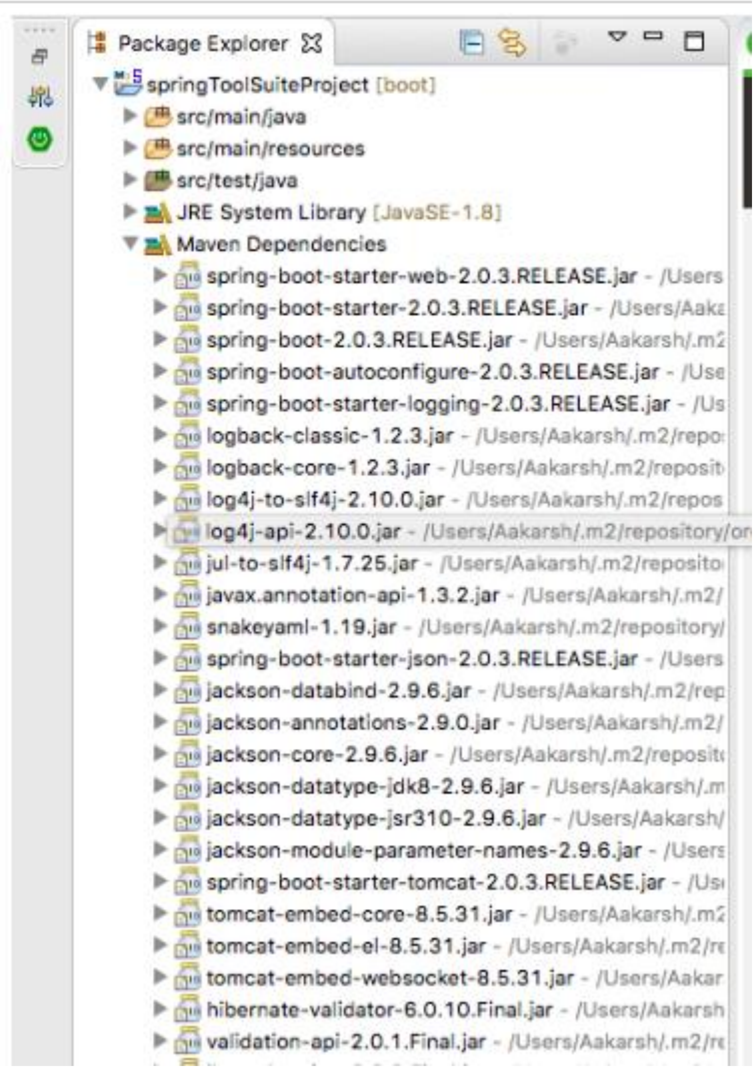
Spring Boot project structure



- After the project creation is finished, we will get a project structure as shown in the left picture
- This is a typical maven project structure which contains
  - pom.xml – maven config file
  - src – the source code of our project

Ref:- https://www.javacodegeeks.com/2018/07/spring-boot-project-sts.html

Maven dependencies Section



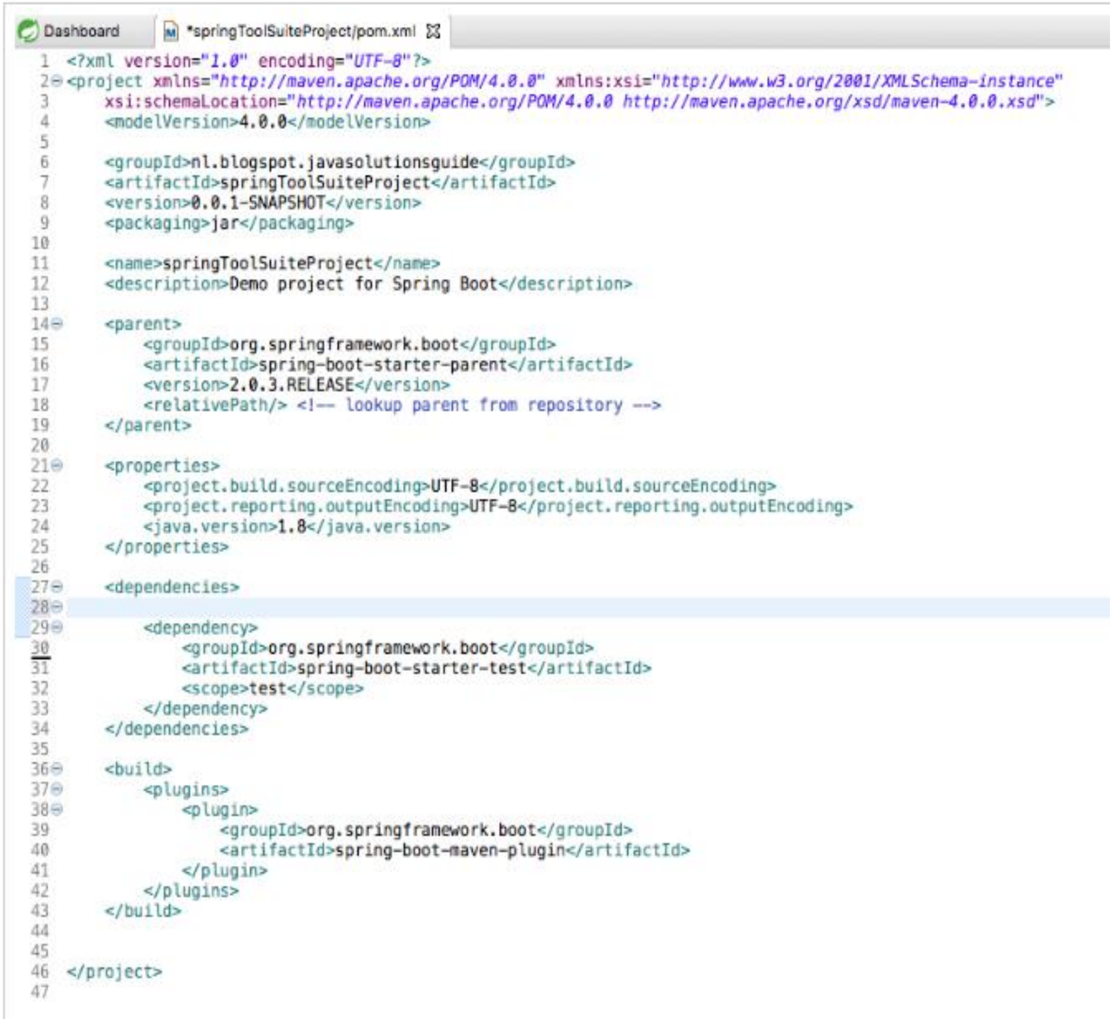
```
Package Explorer
▼ springToolSuiteProject [boot]
  ▶ src/main/java
  ▶ src/main/resources
  ▶ src/test/java
  ▶ JRE System Library [JavaSE-1.8]
  ▼ Maven Dependencies
    ▶ spring-boot-starter-web-2.0.3.RELEASE.jar - /Users
    ▶ spring-boot-starter-2.0.3.RELEASE.jar - /Users/Aaka
    ▶ spring-boot-2.0.3.RELEASE.jar - /Users/Aakarsh/.m2
    ▶ spring-boot-autoconfigure-2.0.3.RELEASE.jar - /Use
    ▶ spring-boot-starter-logging-2.0.3.RELEASE.jar - /Us
    ▶ logback-classic-1.2.3.jar - /Users/Aakarsh/.m2/repo
    ▶ logback-core-1.2.3.jar - /Users/Aakarsh/.m2/reposit
    ▶ log4j-to-slf4j-2.10.0.jar - /Users/Aakarsh/.m2/repos
    ▶ log4j-api-2.10.0.jar - /Users/Aakarsh/.m2/repository/or
    ▶ jul-to-slf4j-1.7.25.jar - /Users/Aakarsh/.m2/reposito
    ▶ javax.annotation-api-1.3.2.jar - /Users/Aakarsh/.m2/
    ▶ snakeyaml-1.19.jar - /Users/Aakarsh/.m2/repository/
    ▶ spring-boot-starter-json-2.0.3.RELEASE.jar - /Users
    ▶ jackson-databind-2.9.6.jar - /Users/Aakarsh/.m2/rep
    ▶ jackson-annotations-2.9.0.jar - /Users/Aakarsh/.m2/
    ▶ jackson-core-2.9.6.jar - /Users/Aakarsh/.m2/reposit
    ▶ jackson-datatype-jdk8-2.9.6.jar - /Users/Aakarsh/.m
    ▶ jackson-datatype-jsr310-2.9.6.jar - /Users/Aakarsh/
    ▶ jackson-module-parameter-names-2.9.6.jar - /Users
    ▶ spring-boot-starter-tomcat-2.0.3.RELEASE.jar - /Us
    ▶ tomcat-embed-core-8.5.31.jar - /Users/Aakarsh/.m2
    ▶ tomcat-embed-el-8.5.31.jar - /Users/Aakarsh/.m2/re
    ▶ tomcat-embed-websocket-8.5.31.jar - /Users/Aakar
    ▶ hibernate-validator-6.0.10.Final.jar - /Users/Aakarsh
    ▶ validation-api-2.0.1.Final.jar - /Users/Aakarsh/.m2/re
```

- In the package/project explorer, when you expand the **“Maven Dependencies”** section, we can see all the jars that are downloaded for this project

- Even in an empty spring boot project, lots of spring core jars and some specific to spring boot’s common functionalities will be downloaded

Ref:- https://www.javacodegeeks.com/2018/07/spring-boot-project-sts.html

Spring Boot – Empty Project Pom.xml



```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>

    <groupId>nl.blogspot.javasolutionsguide</groupId>
    <artifactId>springToolSuiteProject</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>springToolSuiteProject</name>
    <description>Demo project for Spring Boot</description>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.0.3.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
        <java.version>1.8</java.version>
    </properties>

    <dependencies>

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

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>


</project>

```

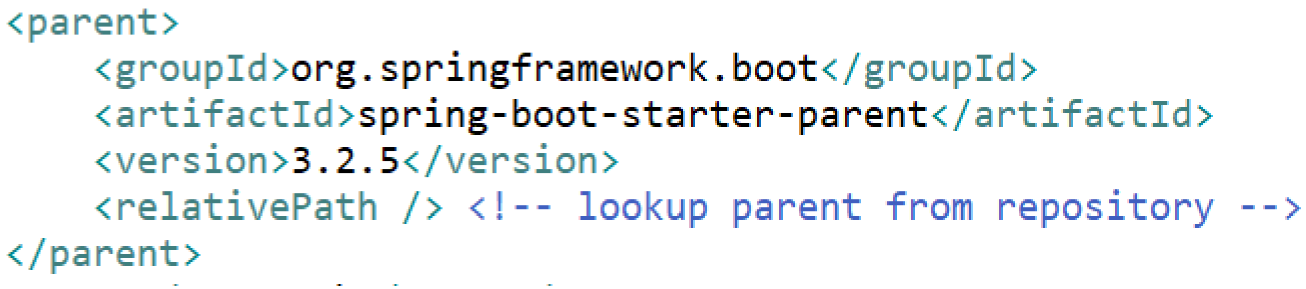
- In the left picture, we can see that the pom.xml for an empty spring boot project

- We have not chosen any dependencies(web, jdbc etc) for this project

Ref:- https://www.javacodegeeks.com/2018/07/spring-boot-project-sts.html

Pom.xml – parent section

- If you observe the **<parent>** tag in the pom.xml of our spring boot project, we can see that its referring to **spring-boot-starter-parent**, as shown below

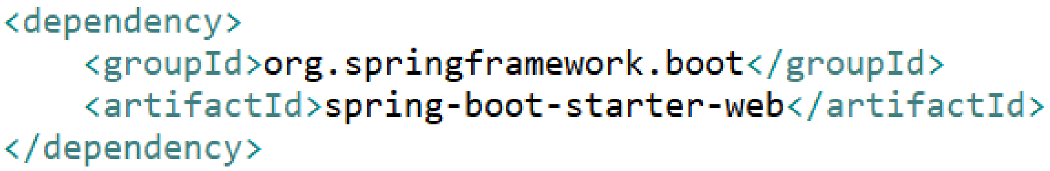


```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.2.5</version>
    <relativePath /> <!-- lookup parent from repository -->
</parent>
```

- Any maven project can inherit properties from a parent project. In a spring boot project, we are inheriting from the parent starter project named **spring-boot-starter-parent**
- The **spring-boot-starter-parent** project provides the core functionalities of a spring boot project

Spring Boot – Adding web dependency

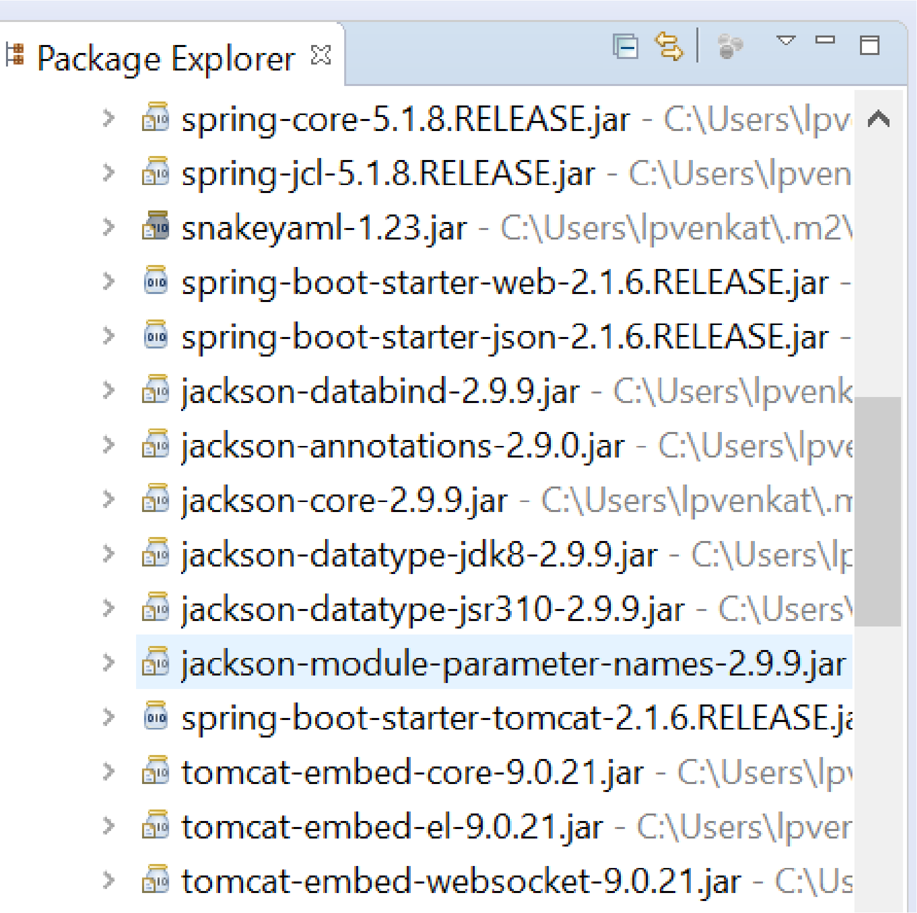
- The first project we created was a blank project which doesn't enable web functionalities
- To convert our spring boot project to a web backend API layer, we need to add the web dependencies in the pom.xml, as shown below



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

- Once we add **spring-boot-starter-web** in pom.xml, all web dependencies, such as embedded tomcat, Jackson library for JSON processing, etc

Spring Boot – Web related dependencies

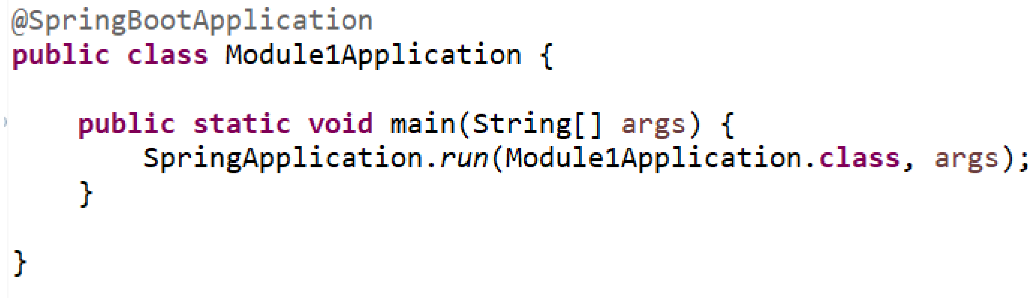


```
spring-core-5.1.8.RELEASE.jar - C:\Users\lpv
spring-jcl-5.1.8.RELEASE.jar - C:\Users\lpven
snakeyaml-1.23.jar - C:\Users\lpvenkat\.m2\
spring-boot-starter-web-2.1.6.RELEASE.jar -
spring-boot-starter-json-2.1.6.RELEASE.jar -
jackson-databind-2.9.9.jar - C:\Users\lpvenk
jackson-annotations-2.9.0.jar - C:\Users\lpve
jackson-core-2.9.9.jar - C:\Users\lpvenkat\.m
jackson-datatype-jdk8-2.9.9.jar - C:\Users\lp
jackson-datatype-jsr310-2.9.9.jar - C:\Users\
jackson-module-parameter-names-2.9.9.jar
spring-boot-starter-tomcat-2.1.6.RELEASE.ja
tomcat-embed-core-9.0.21.jar - C:\Users\lp
tomcat-embed-el-9.0.21.jar - C:\Users\lpver
tomcat-embed-websocket-9.0.21.jar - C:\Us
```

- As shown in the picture, once the **spring-boot-starter-web** is added to the pom.xml, lots of extra dependencies are downloaded
- They are
  - Embedded tomcat
  - Jackson Java parsing library

Spring Boot – The main file

- We have created a blank spring boot project and added the web dependency to it
- Spring Boot creates a main file, which looks like a standard java application, with a main method

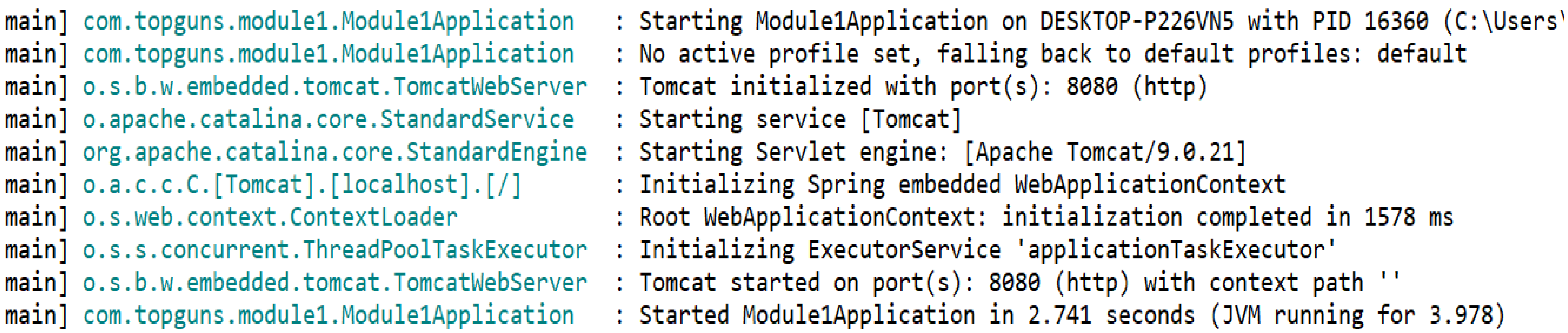


```
@SpringBootApplication
public class Module1Application {

    public static void main(String[] args) {
        SpringApplication.run(Module1Application.class, args);
    }

}
```

- The only difference here is that the main class is annotated with a **@SpringBootApplication**
- From the main method of **Module1Application**, SpringApplication class’s run method is called
- This run method ensures that the following steps are done
  - Sets up default configuration
  - Starts Spring application context
  - Performs class path scan
  - Starts tomcat server

Running the spring boot app

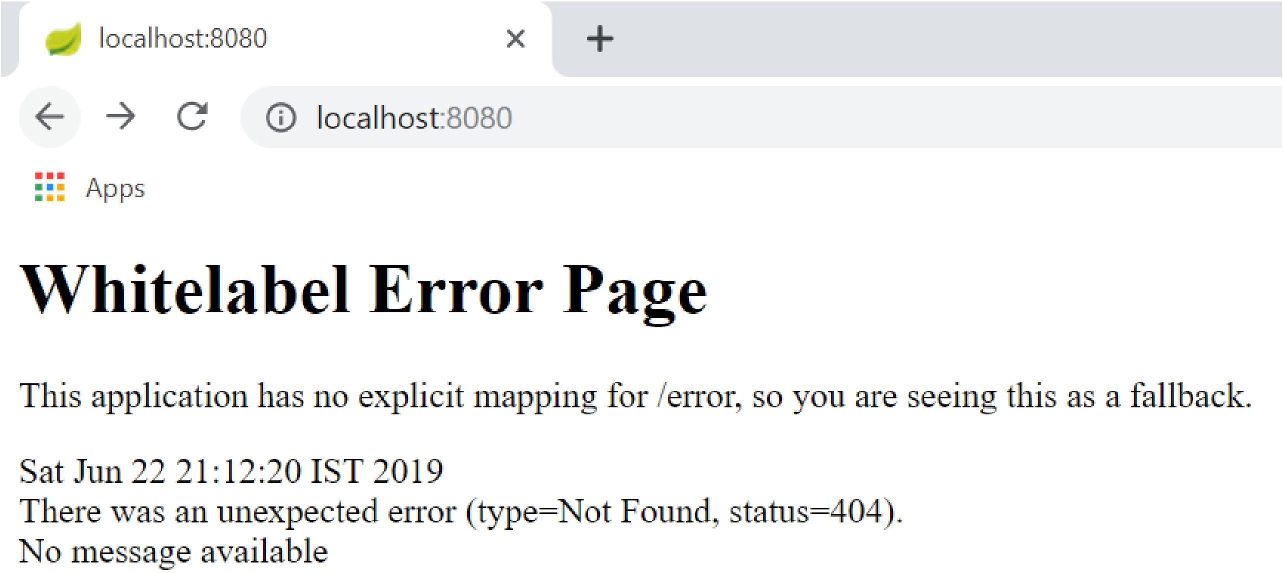
- When you right click the project and select **Run as -> Spring Boot App**, the web application should get deployed in the embedded tomcat server
- The below diagram clearly demonstrates that the embedded tomcat server is running and hosting the spring boot app in the default port



```
main] com.topguns.module1.Module1Application   : Starting Module1Application on DESKTOP-P226VN5 with PID 16360 (C:\Users\
main] com.topguns.module1.Module1Application   : No active profile set, falling back to default profiles: default
main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8080 (http)
main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
main] org.apache.catalina.core.StandardEngine  : Starting Servlet engine: [Apache Tomcat/9.0.21]
main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
main] o.s.web.context.ContextLoader            : Root WebApplicationContext: initialization completed in 1578 ms
main] o.s.s.concurrent.ThreadPoolTaskExecutor  : Initializing ExecutorService 'applicationTaskExecutor'
main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context path ''
main] com.topguns.module1.Module1Application   : Started Module1Application in 2.741 seconds (JVM running for 3.978)
```

Spring Boot – Standard output

- The default app, after it started running, when we reach the web application through the default URL, we should get the below output



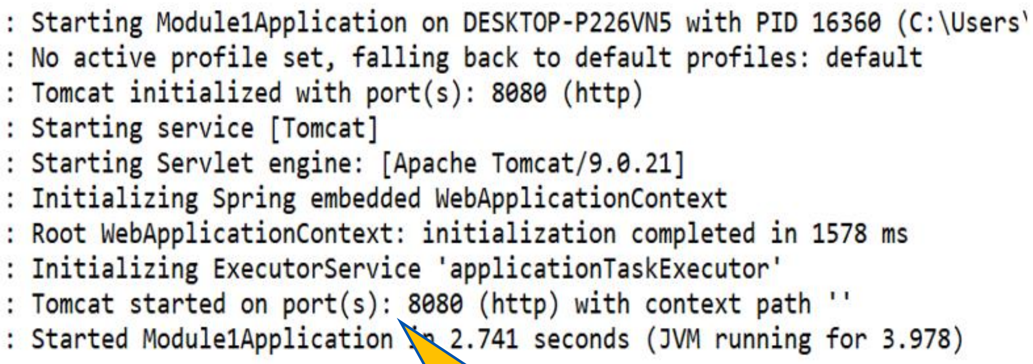
Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Sat Jun 22 21:12:20 IST 2019
There was an unexpected error (type=Not Found, status=404).
No message available

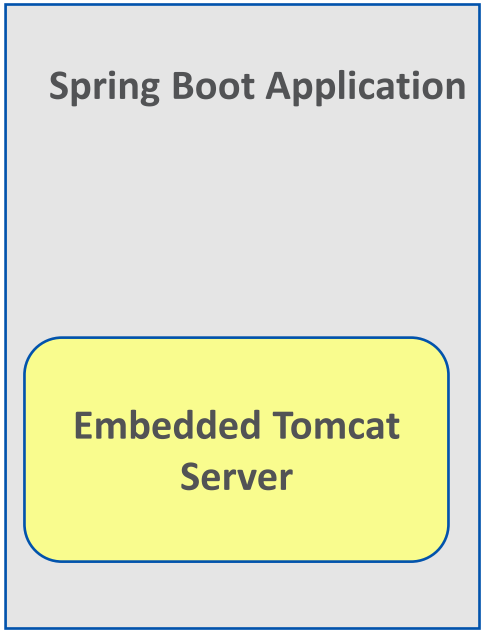
Embedded Servlet Containers

- Traditionally, Spring applications have to be hosted on a web server such as Tomcat
- Tomcat serves as the Servlet container for running spring based applications
- So, there was a separate deployment step
- But Spring Boot comes with an embedded Tomcat server, which provides
  - Convenience of deployment
  - Stand alone application
- When we run the boot app, we can see that tomcat is running



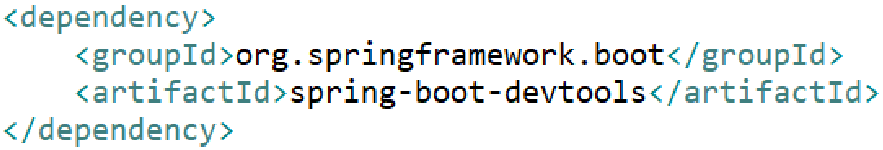
```
: Starting Module1Application on DESKTOP-P226VN5 with PID 16360 (C:\Users\
: No active profile set, falling back to default profiles: default
: Tomcat initialized with port(s): 8080 (http)
: Starting service [Tomcat]
: Starting Servlet engine: [Apache Tomcat/9.0.21]
: Initializing Spring embedded WebApplicationContext
: Root WebApplicationContext: initialization completed in 1578 ms
: Initializing ExecutorService 'applicationTaskExecutor'
: Tomcat started on port(s): 8080 (http) with context path ''
: Started Module1Application in 2.741 seconds (JVM running for 3.978)
```

Embdded Tomcat running in port 8080



DevTools

- During the development process, we have to restart the dev server every time the code changes
- This leads to a slow iterative process of code, fix and debug cycle
- Hence, the spring ecosystem introduced a new starter package called Devtools, (Spring Development Tools)
- We need to include the Devtools starter package as a dependency in the pom.xml

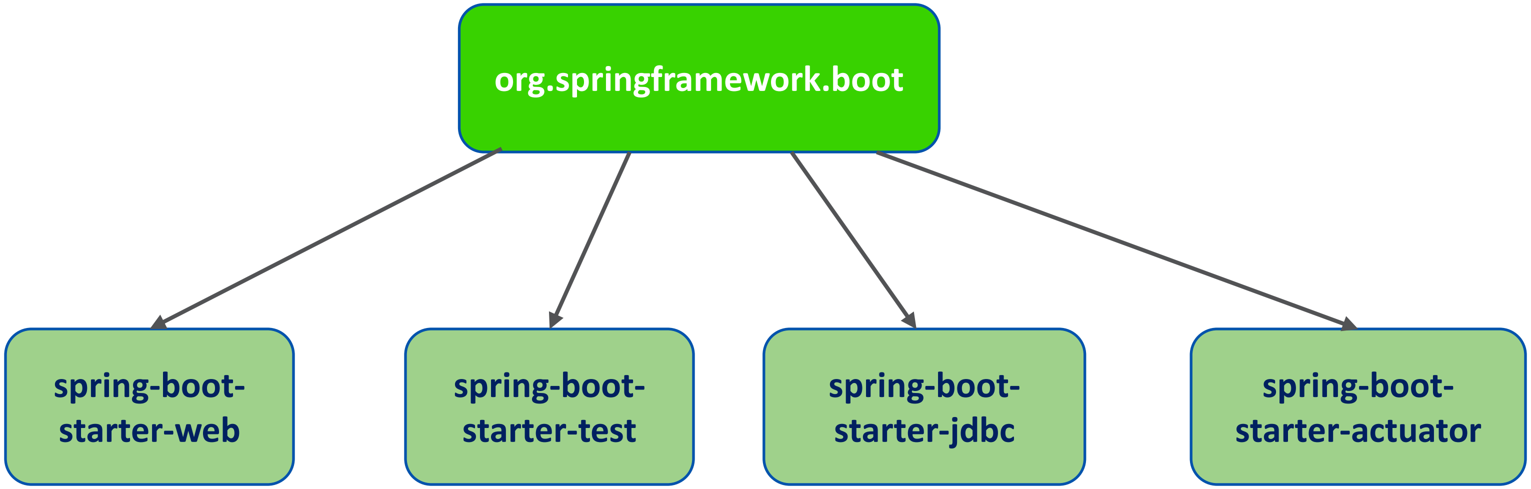


```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
</dependency>
```

- After adding Devtools, if we start the local dev server, then onwards for any code file changes, the dev server will get restarted automatically

Spring Boot Starters

- Bill of materials
- Set of convenient dependency descriptors which we can include in our application
- Makes development easier and rapid
- For e.g. for using JPA for database access, include the **spring-boot-starter-data-jpa** starter
- Most common starters for spring boot are provided under the **org.springframework.boot** group



Important Starters

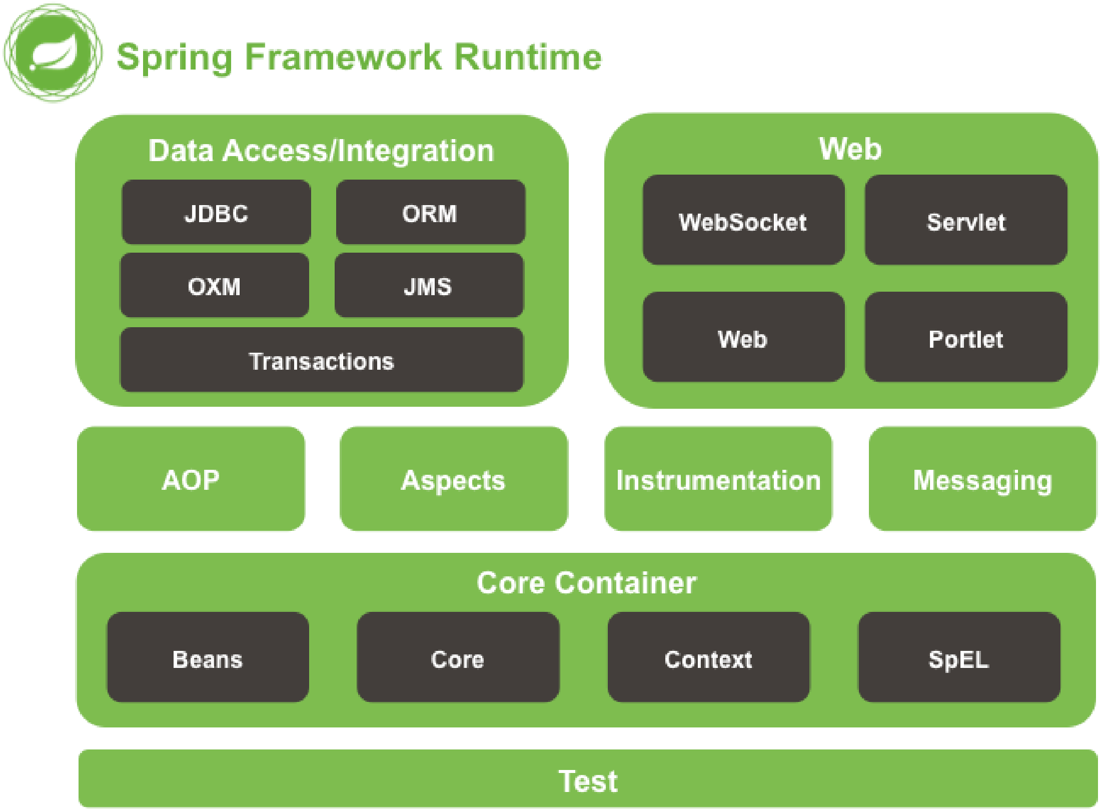
- **spring-boot-starter-web**
  - It is used for building web, including RESTful, applications using Spring MVC. Uses Tomcat as the default embedded container
- **spring-boot-starter-jdbc**
  - It is used for JDBC with the Tomcat JDBC connection pool
- **spring-boot-starter-actuator**
  - It provides production ready features to help you monitor and manage your application.
- **spring-boot-starter-test**
  - It is used to test Spring Boot applications with libraries including Junit and Mockito
- **spring-boot-starter-data-rest**
  - It is used for exposing Spring Data repositories over REST using Spring Data REST

Advantages of Spring framework

- Spring enables developers to develop enterprise applications using POJOs (Plain Old Java Object)
- Spring provides an abstraction layer on existing technologies like servlets, JDBC, ORMs, Logging etc to simplify the development process
- Spring WEB framework has a well-designed web MVC framework
- Spring framework has taken the best practice that have been proven over the years in several applications and formalized as design patterns
- Spring gives built in middleware services like Connection pooling, Transaction management

Spring Framework Modules

- Spring framework consists of features organized into about 20 modules, these modules are grouped into Core Container, Data Access/Integration, Web, Test and many more

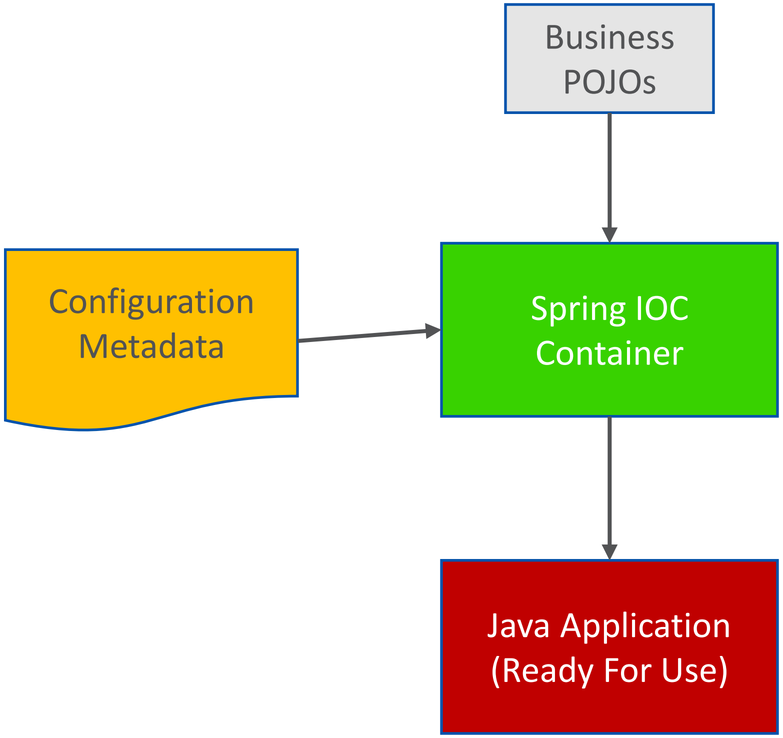


- Important Modules
  - Spring Core Container
  - Spring JDBC and DAO Module
  - ORM Module
  - Web Module

- Ref:- https://docs.spring.io/spring/docs/5.0.0.RC2/spring-framework-reference/overview.html

Spring Core Container

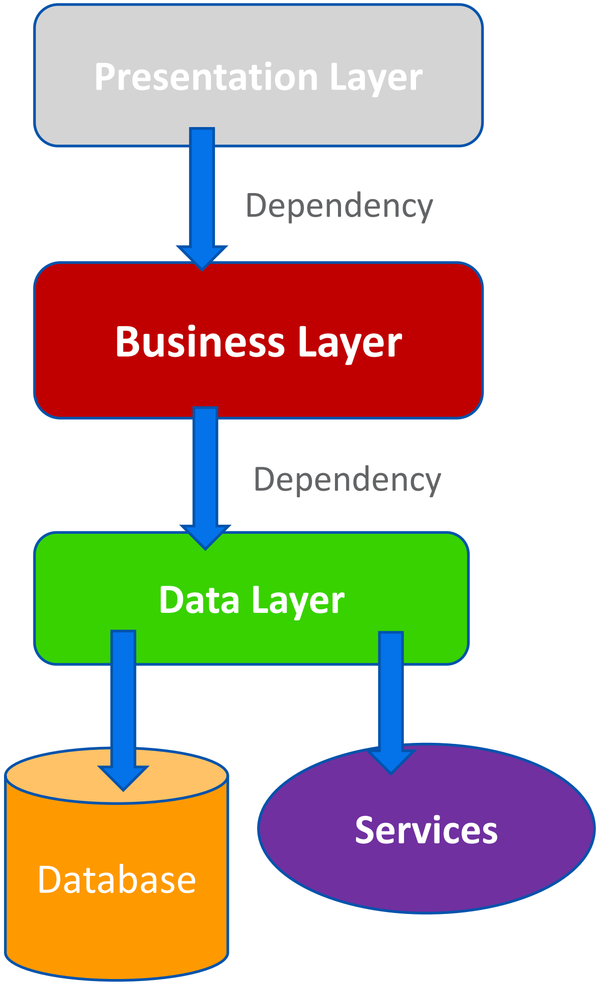
- The Spring Core container is the basis for the complete spring framework
- The Spring core container provides an implementation for IoC supporting Dependency Injection



- The Spring Container is responsible for managing the objects life cycle such as
  - Creating the Objects
  - Calling initialization methods
  - Configuring objects by writing them together

What is dependency injection?

- Before understanding dependency injection, lets try to understand the term “dependency” in our software code



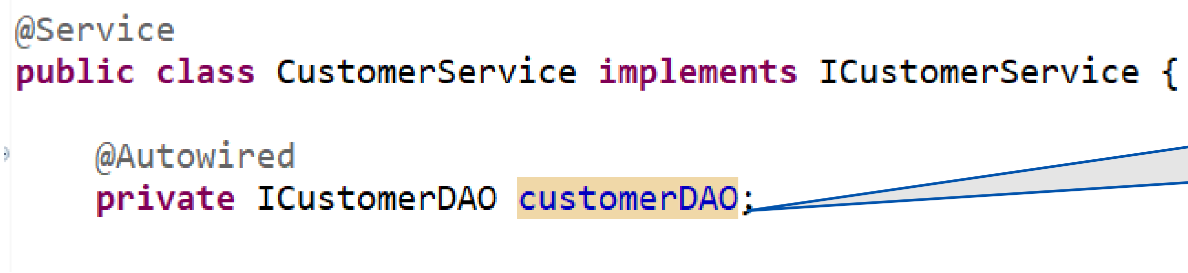
- As shown, we have the following dependencies in software layer
  - The presentation layer depends on the business layer
  - The business layer depends on the data access layer

Dependency Injection – Loosely Coupled System

- As seen in the slide earlier, the clients have to create and manage their dependencies
- When each software layer manages their own dependencies, it becomes a tightly coupled system
- Tightly coupled systems are very difficult to manage, fix errors, or test or even modify/upgrade
- Hence the modern enterprises prefer loosely coupled systems
- Dependency injection helps to build loosely coupled systems by delegating the job of managing the dependencies to a common module named injector
- The injector takes care of creating and maintaining the dependencies
- Hence the name Dependency injection
- There are so many dependency injection frameworks, but the most popular in the Java world is Spring framework
- DI is one of the functionalities of Spring framework apart from so many other features

A Sample dependency injection

- The below piece of code shows that the CustomerService class is dependent on CustomerDAO class.



```
@Service
public class CustomerService implements ICustomerService {

    @Autowired
    private ICustomerDAO customerDAO;
```

DAO object will be injected by Spring framework

- Instead of creating and managing the **customerDAO** object by itself, the customer service class delegates that dependency management to the DI framework (Spring framework)

- Hence, the service class needs to worry only about business services rather than the pain of creating and managing the object lifecycle of its dependency(**DAO class**)

- More on Spring Boot Annotations later

Spring Problems/Disadvantages

- Spring is a huge framework which has so many moving parts
- Multiple setup steps
- Multiple configuration steps
- Multiple build and deploy steps

Spring Boot – But Why?

- Spring Boot is not a whole new framework from scratch
- Spring Boot is just an abstraction layer top of Spring
- Spring Boot is opinionated
- We are no throwing away Spring, rather we are just building on Spring

- Basically, Spring Boot is built with the following things in mind
  - Very less configuration
  - Convention based framework
  - JAR incompatibilities are taken care of by using starters
  - Simplifies overall development experience
  - Less plumbing/configuration code, but more business logic code

Spring Boot Benefits

- Spring Boot takes a convention based approach and hence manual configurations are not needed
- It uses an annotation based approach which is more friendlier than the clumsy XML configurations
- Embed Tomcat, Jetty or Undertow directly (no need to deploy WAR files)
- Hence, It allows to create stand-alone Spring applications

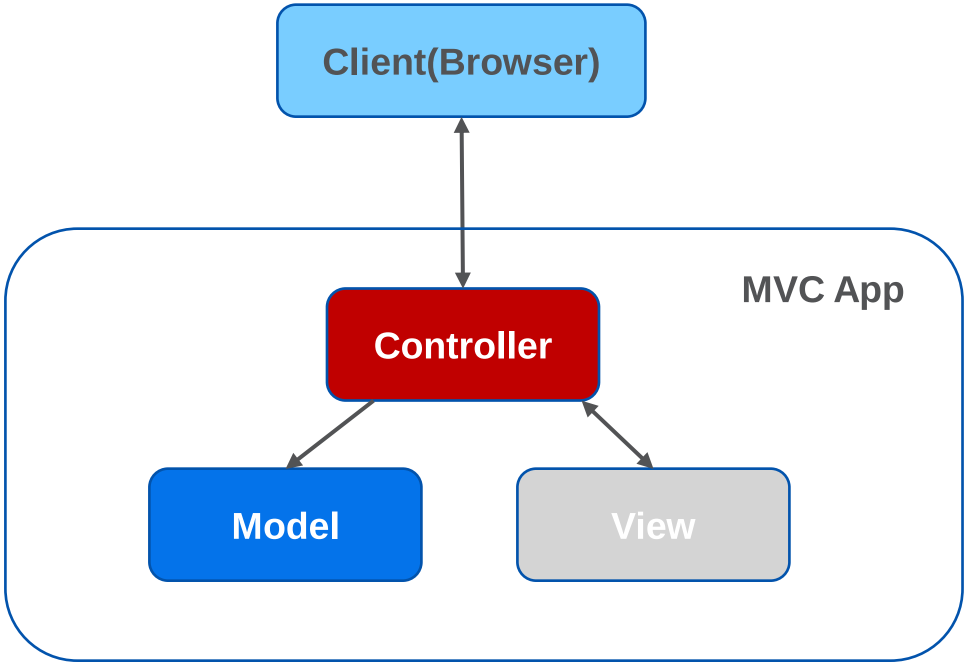
- It simplifies dependency management of our application
- Increases productivity and reduces the development time
- Provide opinionated 'starter' dependencies to simplify your build configuration
- Automatically configure Spring and 3rd party libraries whenever possible
- Provide production-ready features such as metrics, health checks and externalized configuration

API first approach – Spring Boot

- The modern enterprise application development takes an API first approach
- The APIs could be consumed by a browser app or a mobile app or a TV app or a smart watch app
- Gone are the days, where in we had only desktop apps or console only apps
- In a full stack development, the server side is taken care by Spring Boot REST APIs
- In case of web apps, the front end could be a SPA framework like React

MVC Architecture

- MVC architecture is the most common architecture used in modern web applications
- MVC stands for Model, View and Controller
- MVC provides **‘separation of concern’** concept, where each part has its own responsibility
- A typical MVC architecture is shown in the below diagram



MVC - An explanation

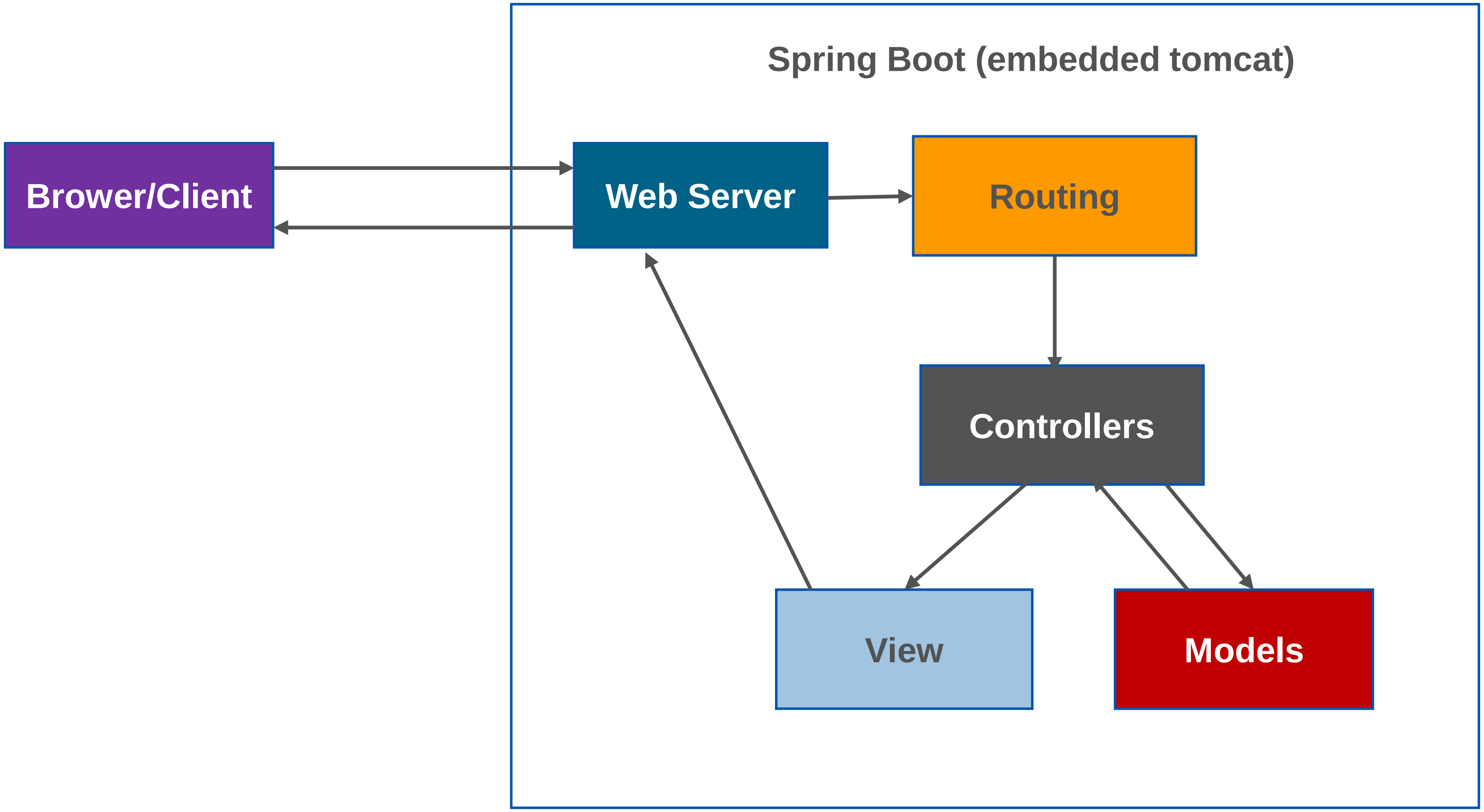
- **Models**
  - This layer typically represents the tight bonding with the databases (relational or other form of databases)
  - This layer might handle validations and association between models

- **View**
  - Presents data in a particular format, triggered by a controller's decision
  - This presentation can be of various format types such as HTML, JSON, XML or even PDFs etc
  - The final result of the view is nothing but the user interface (UI), that the user sees

- **Controllers**
  - This layer is the orchestration layer that controls the flow between the client, the models and view
  - Controllers receive the request from the client, decides which model data should be used, and then decides which view should be used to render the data to the client
  - Controllers can also do some cleanup of data received from the client

MVC – More detailed diagram

- A more detailed flow of the MVC architecture



Spring Boot Annotations

- As already discussed, Spring Boot does not use XML Configuration anymore and implements the convention over configuration principle
- Spring Boot uses what are called Annotations to adopt conventions based approach
- Annotation is a form of metadata which provides data about a program
- Annotations can be applied on a class, method, or a parameter etc
- Based on the context of where it is applied, annotations can have different side effects on the program

Spring Boot - Annotations

- There are some commonly used Spring Boot Annotations
  - **@SpringBootApplication** – enables auto configuration and component scanning
  - **@Autowired** – makes a field to be auto wired by Spring dependency injection

Controller related annotations

  - **@Controller** – makes the class as a web controller, capable of handling http requests
  - **@RestController** – makes it capable of handling REST requests (returning JSON or XML)
  - **@RequestMapping** - maps HTTP request with a path to a controller method

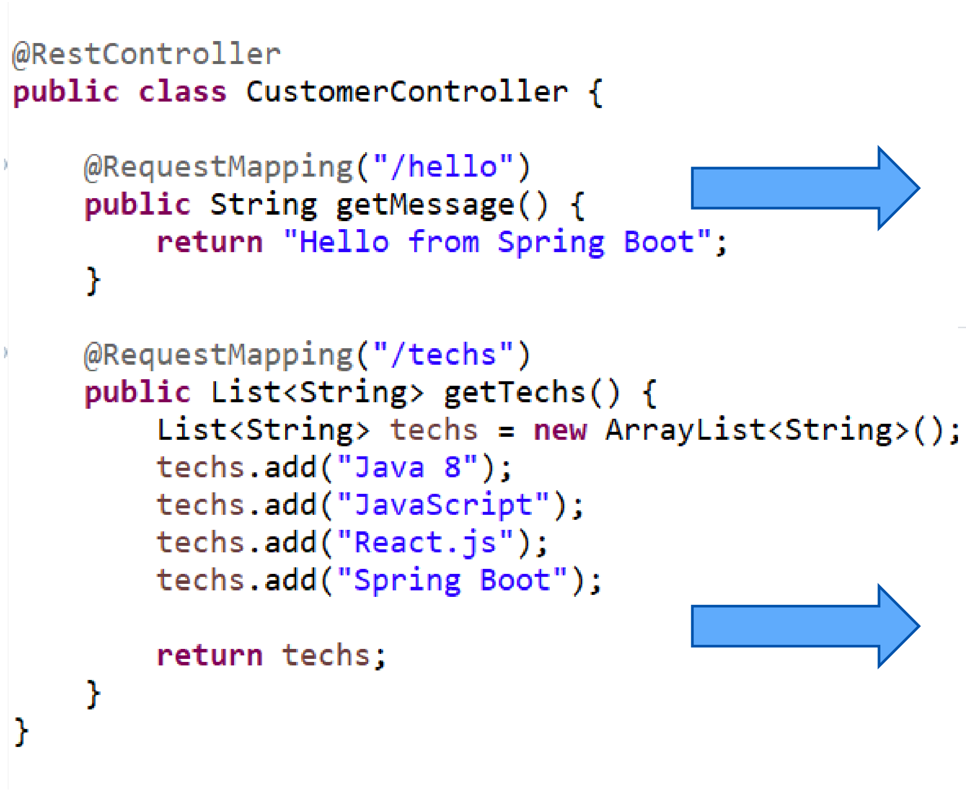
Database and Services related annotations

  - **@Repository** – indicates the class as a repository class, which is an abstraction of
  - **@Service** – to mark a class as a service class

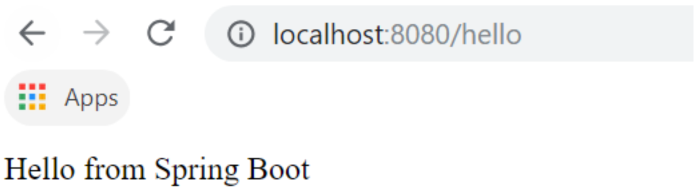
  - We will see some of these annotations in details as we progress

REST Controller

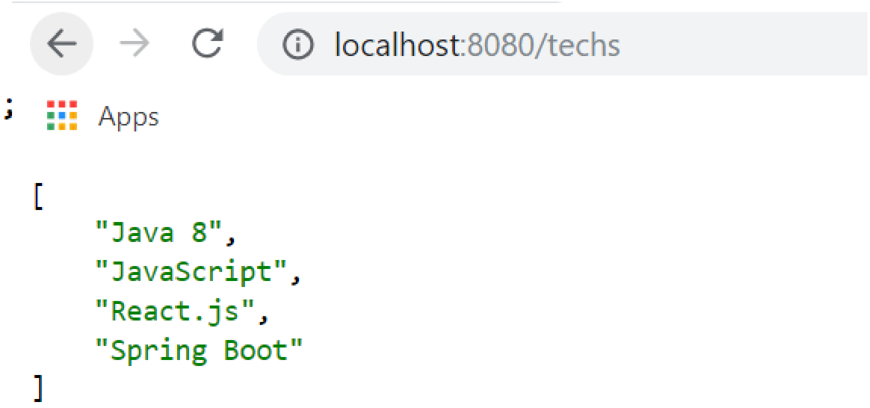
- We are using Spring Boot as a pure REST API layer
- Lets start writing a simple REST controller as below



```
@RestController
public class CustomerController {

    @RequestMapping("/hello")
    public String getMessage() {
        return "Hello from Spring Boot";
    }

    @RequestMapping("/techs")
    public List<String> getTechs() {
        List<String> techs = new ArrayList<String>();
        techs.add("Java 8");
        techs.add("JavaScript");
        techs.add("React.js");
        techs.add("Spring Boot");

        return techs;
    }
}
```



```
Hello from Spring Boot
```



```
[
    "Java 8",
    "JavaScript",
    "React.js",
    "Spring Boot"
]
```

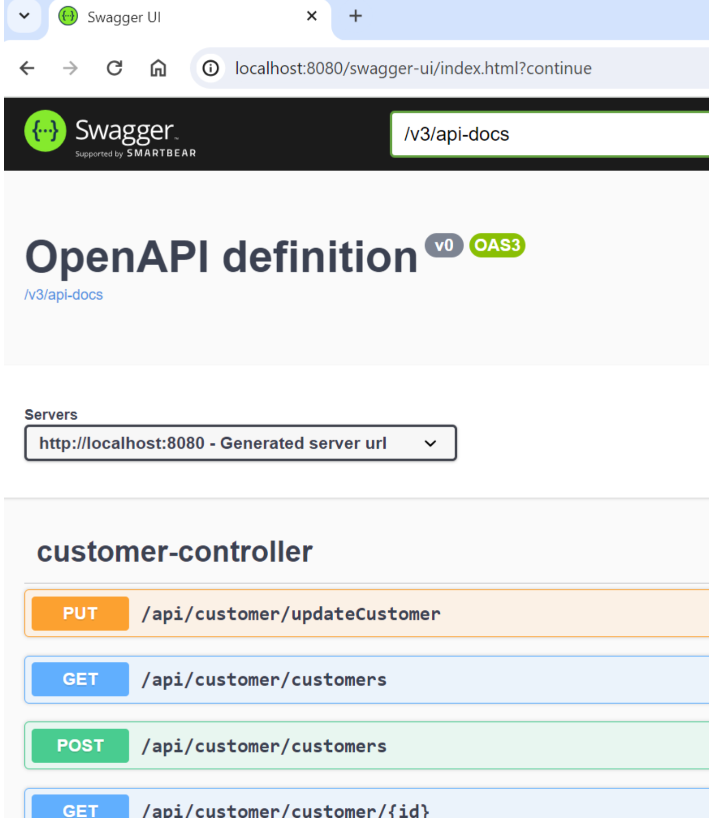
Controller Annotations explained

- In the previous slide, we wrote 2 simple APIs, one returning a simple string and other returning an array of strings
- We will also build a full fledged REST apis, where we could return objects and list of objects
- We use **@RestController** annotations for converting a class to a rest controller
- **@RestController** combines the job of **@Controller** and **@ResponseBody**
- When we use **@RestController** on any class, every request handling method of that controller class automatically serializes return objects into ***HttpResponse***
- This simplifies rest programming, as we don’t have to manually convert java objects into JSON response
- Objects to JSON conversion is internally taken care by the appropriate annotations

Swagger Integration

Swagger Integration

- Swagger UI is used to visualize and interact with the API's resources without having any of the implementation logic in place.



localhost:8080/swagger-ui/index.html?continue

/v3/api-docs

OpenAPI definition v0 OAS3

/v3/api-docs

Servers

http://localhost:8080 - Generated server url

customer-controller

PUT /api/customer/updateCustomer

GET /api/customer/customers

POST /api/customer/customers

GET /api/customer/customer/{id}

Adding the Maven Dependency

```
<dependency>
        <groupId>org.springdoc</groupId>
        <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
        <version>2.1.0</version>
</dependency>
```

Let us implement of use case

Use case 1

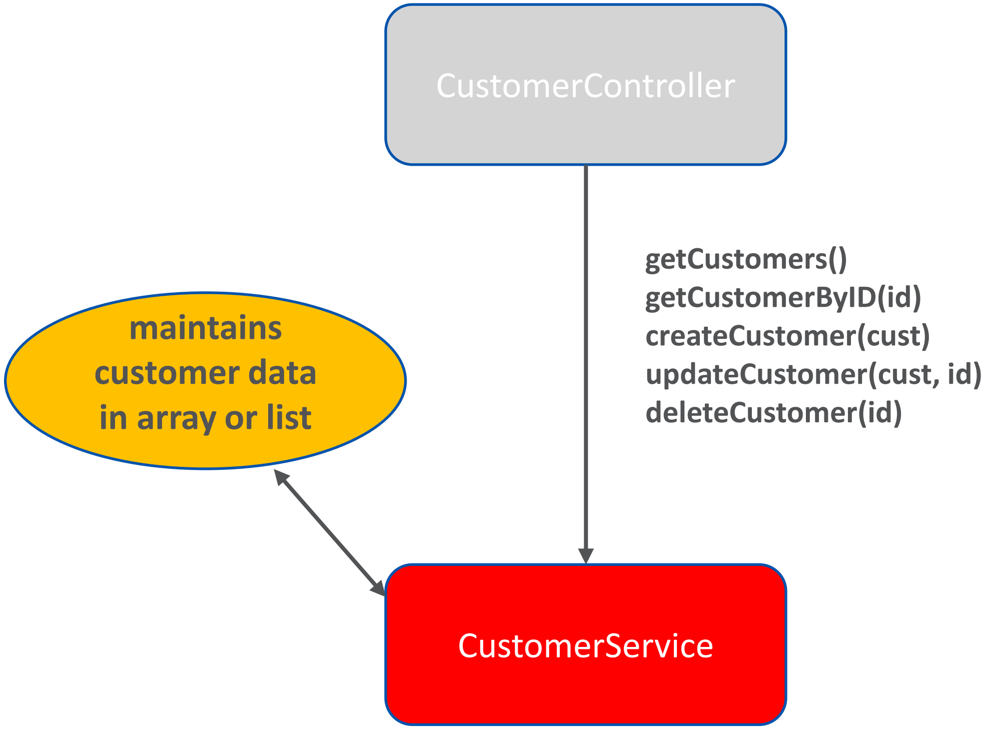
- As part of the use case 1, you have to build a simple CRUD based REST API for dealing with customers data
- In this uses case 1, no database need to be used
- We will only use in-memory data structures such as a arrays or list to store and manipulate data

- Use/Define a customer model/entity as shown
- There should be REST APIs built for
  - Listing all customers
  - Getting one customer details
  - Creating a new customer
  - Updating an existing customer
  - Deleting an existing customer

Customer Model
id - number
firstName - string
lastName - string
email - string
phone - string
active - string
password – string
role - string

A full CRUD API

- Earlier, we just saw a simple REST APIs, returning a string and an array of strings
- But lets get closer to the real world scenarios of building a CRUD APIs
- As we already know, CRUD stands for Create, Read, Update and Delete
- In this module, we will use in memory arrays or lists only to store and manipulate data
- In this scenario, we are just going to have 2 layers
  - Service Layer (**CustomerService**)
  - Controller layer (**CustomerController**)

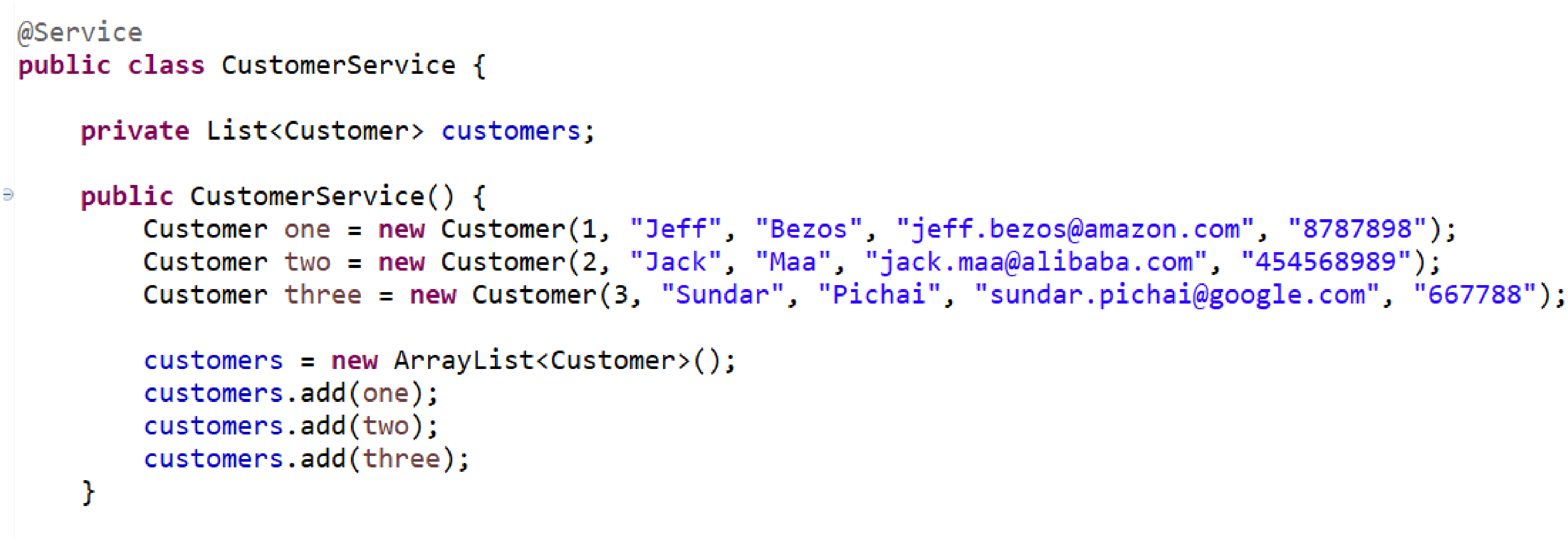


Writing a simple REST Controller

- There are lots of annotations associated with writing rest controllers or controllers in general
  - @Controller
  - @ResponseBody
  - @RestController
  - @RequestMapping
  - @PathVariable
  - @GetMapping
  - @PostMapping
  - @PutMapping

Implementing Customer Service

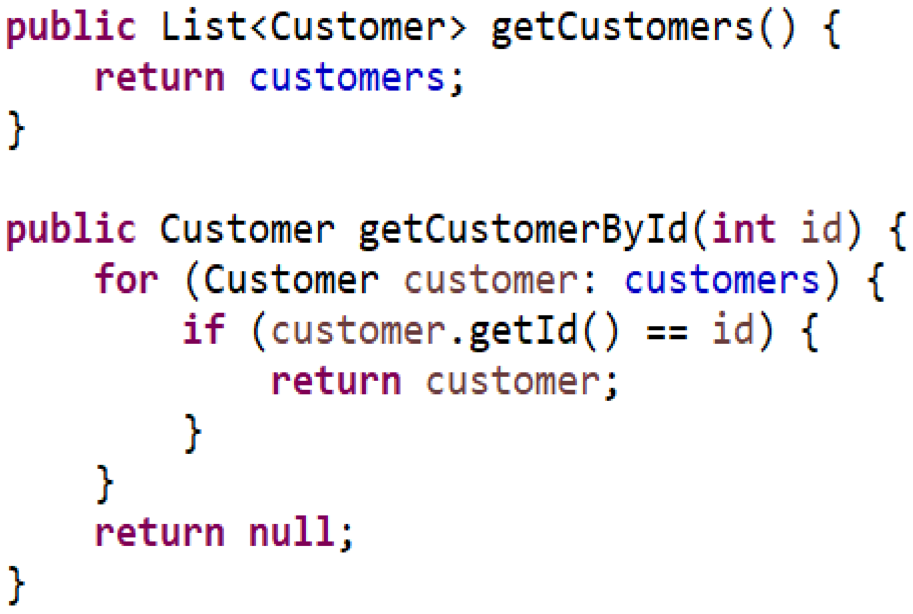
- Lets create a simple customer service class using the @Service annotation
- Spring @Service annotation is used with classes that provide some business functionalities
- Spring context will autodetect these classes, so that they can injected in controllers
- A simple CustomerService class, annotated with @Service annotation is shown below
- It creates an array list to store and manipulate the customers data for CRUD operations



```
@Service
public class CustomerService {

    private List<Customer> customers;

    public CustomerService() {
        Customer one = new Customer(1, "Jeff", "Bezos", "jeff.bezos@amazon.com", "8787898");
        Customer two = new Customer(2, "Jack", "Maa", "jack.maa@alibaba.com", "454568989");
        Customer three = new Customer(3, "Sundar", "Pichai", "sundar.pichai@google.com", "667788");

        customers = new ArrayList<Customer>();
        customers.add(one);
        customers.add(two);
        customers.add(three);
    }
```

CustomerService CRUD methods

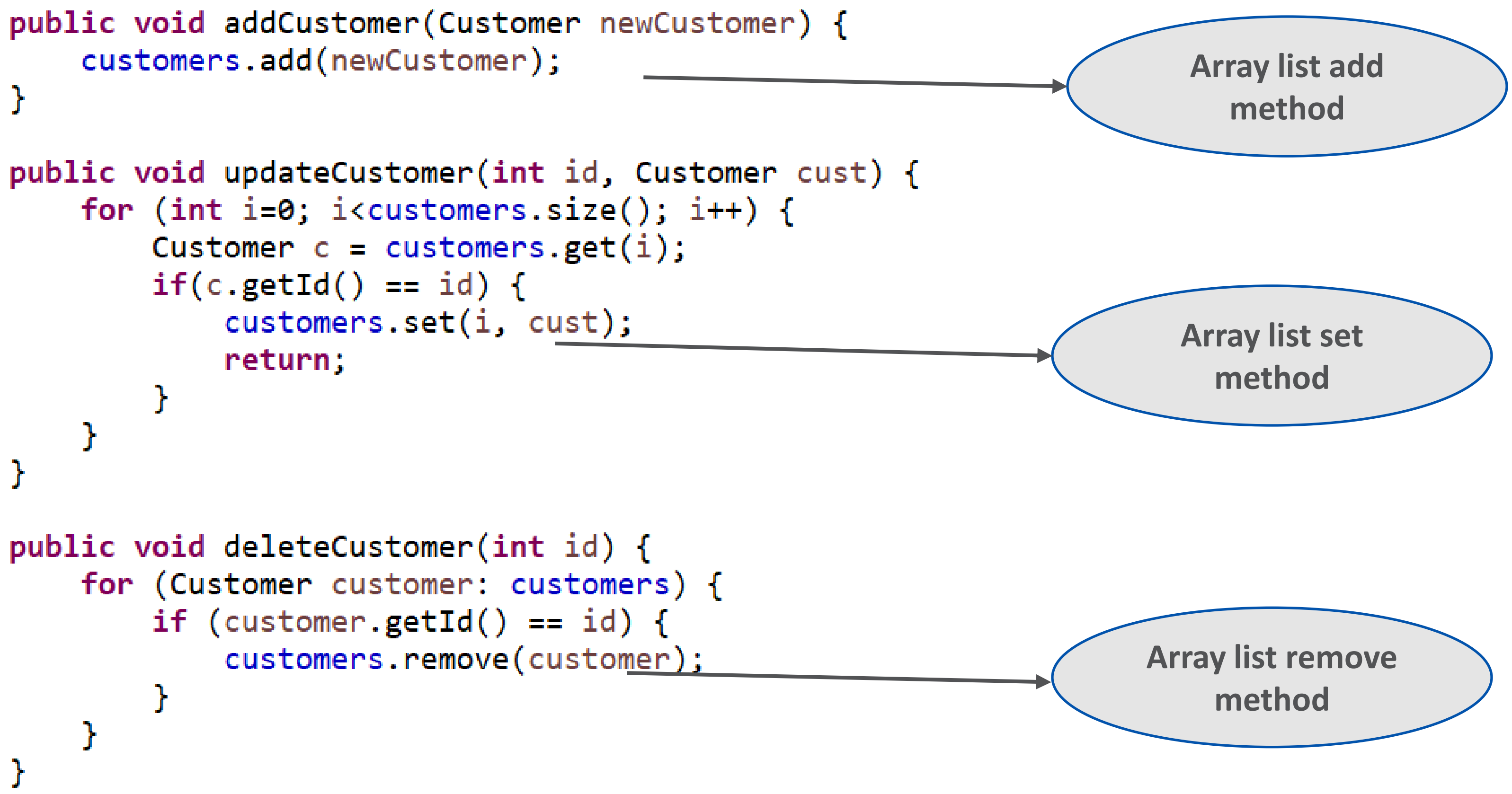
- The standard read operations are shown below
  - getCustomers() – return the entire list of customers
  - getCustomerByID() – returns a single customer with the id



```
public List<Customer> getCustomers() {
    return customers;
}

public Customer getCustomerById(int id) {
    for (Customer customer: customers) {
        if (customer.getId() == id) {
            return customer;
        }
    }
    return null;
}
```

Create, Update and Delete (CUD operation)

- The standard create, update and delete operations are shown below
- No database operations, only in memory array list is used for storing and manipulation



```
public void addCustomer(Customer newCustomer) {
    customers.add(newCustomer);
}

public void updateCustomer(int id, Customer cust) {
    for (int i=0; i<customers.size(); i++) {
        Customer c = customers.get(i);
        if(c.getId() == id) {
            customers.set(i, cust);
            return;
        }
    }
}

public void deleteCustomer(int id) {
    for (Customer customer: customers) {
        if (customer.getId() == id) {
            customers.remove(customer);
        }
    }
}
```

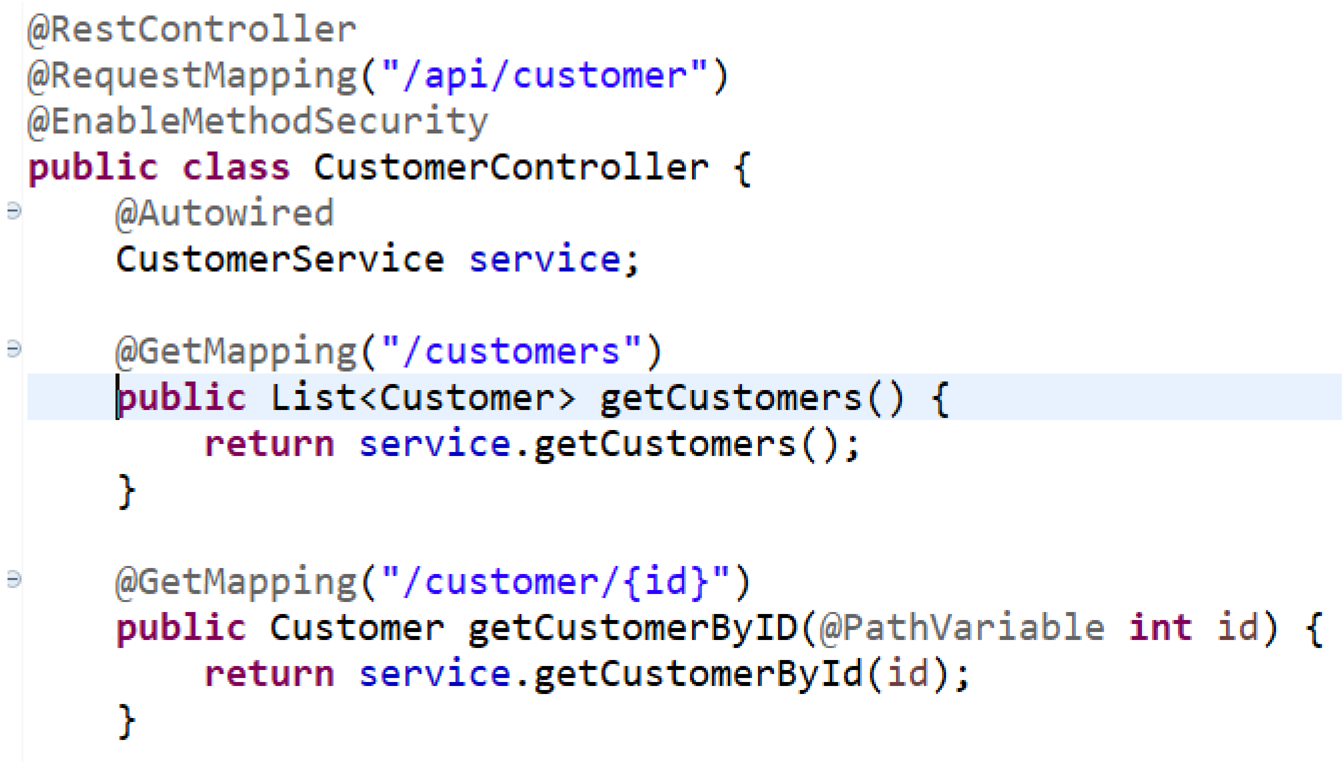
Array list add method

Array list set method

Array list remove method

The Controller – Get operation

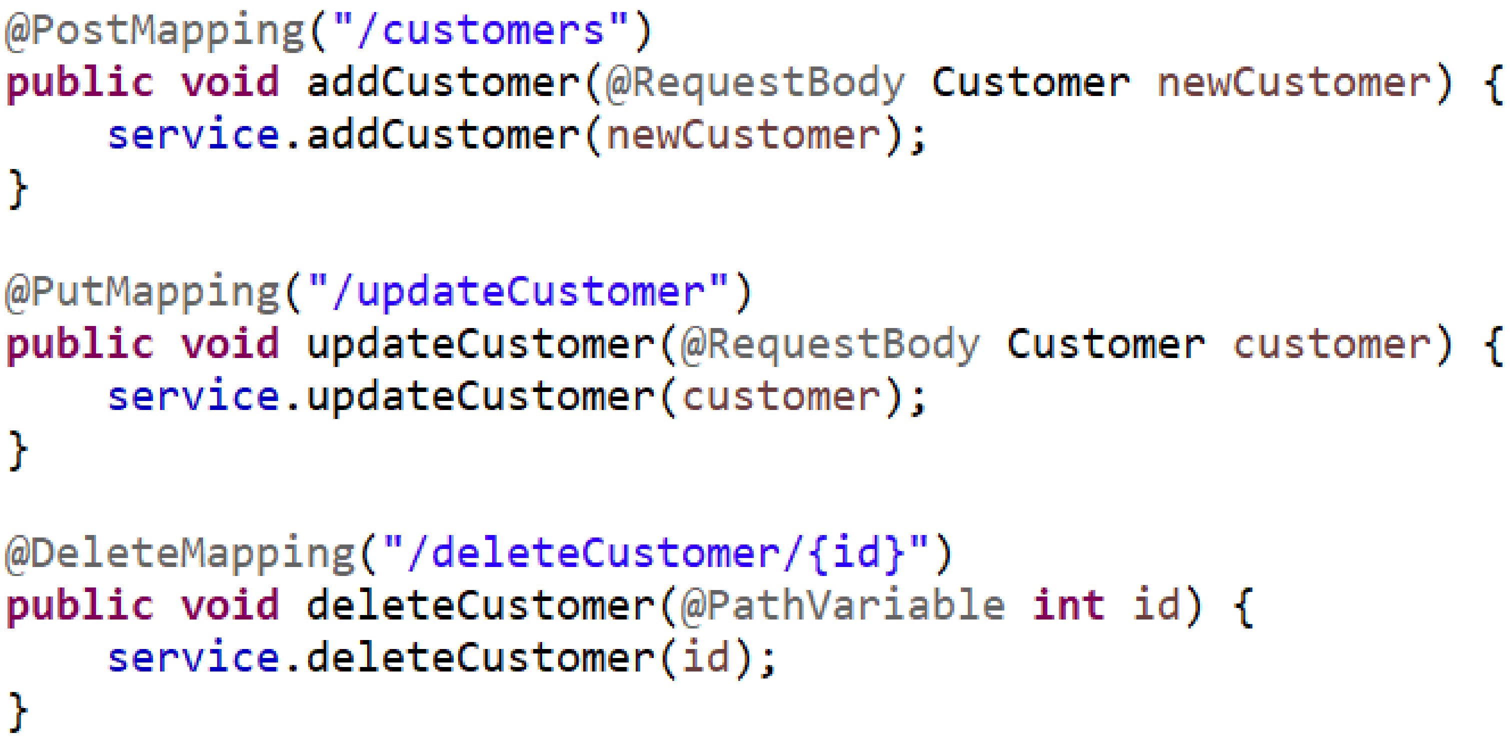
- Lets write a simple **CustomerController** class, annotated with **@RestController**
- Also, we can inject the **CustomerService** dependency using **@Autowired** annotation
- **@RequestMapping, @GetMapping, @PostMapping, @PutMapping and @DeleteMapping** annotations are used for mapping the URL, the Http method and the actual class method that would be invoked for that HTTP operation



```
@RestController
@RequestMapping("/api/customer")
@EnableMethodSecurity
public class CustomerController {
    @Autowired
    CustomerService service;

    @GetMapping("/customers")
    public List<Customer> getCustomers() {
        return service.getCustomers();
    }

    @GetMapping("/customer/{id}")
    public Customer getCustomerByID(@PathVariable int id) {
        return service.getCustomerById(id);
    }
```

Controller – Post, Put and Delete operations

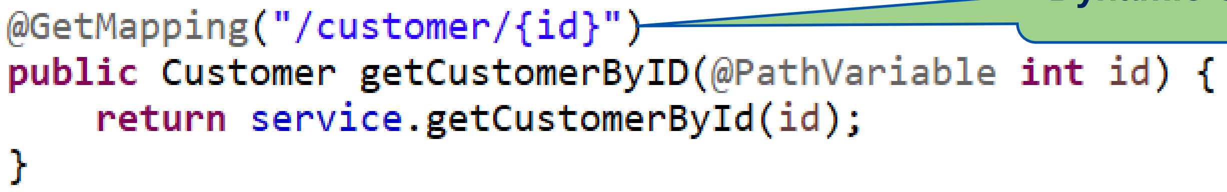
- The remaining controller methods for POST, PUT and DELETE are shown below



```
@PostMapping("/customers")
public void addCustomer(@RequestBody Customer newCustomer) {
    service.addCustomer(newCustomer);
}

@PutMapping("/updateCustomer")
public void updateCustomer(@RequestBody Customer customer) {
    service.updateCustomer(customer);
}

@DeleteMapping("/deleteCustomer/{id}")
public void deleteCustomer(@PathVariable int id) {
    service.deleteCustomer(id);
}
```

Dynamic URL handling - @PathVariable

- The REST URLs will be sometimes static URLs but most of the real world scenarios handle with dynamic URLs
- We already saw how to deal with static URL strings, but lets see how we can handle dynamic URL



```
@GetMapping("/customer/{id}")
public Customer getCustomerByID(@PathVariable int id) {
    return service.getCustomerById(id);
}
```

Dynamic URL

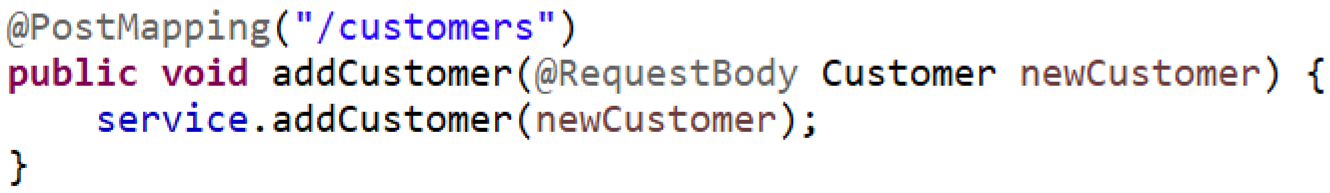
- The above handler could receive dynamic value for the id (Customer ID value)
- To capture the dynamic url parameter, we can use the **@PathVariable** annotation

REST API testing tools

- The REST APIs that we build using Spring Boot can be consumed by any application
- But during the development phase, we typically tools such as browser, postman or curl to test and work with the APIs
- Browsers can only be used to test GET operations directly
- For a full fledged REST API testing, one can use tools such Postman or curl

@RequestBody – Handling POST/PUT data

- **@RequestBody** annotation maps the **HttpRequest** body to a transfer or domain object
- It enables automatic deserialization of the inbound ***HttpRequest*** body onto a Java object



```java
@PostMapping("/customers")
public void addCustomer(@RequestBody Customer newCustomer) {
    service.addCustomer(newCustomer);
}
```

- This annotation is typically used in POST or PUT or PATCH scenarios, where we get data in the body of the request during a create or an update operation

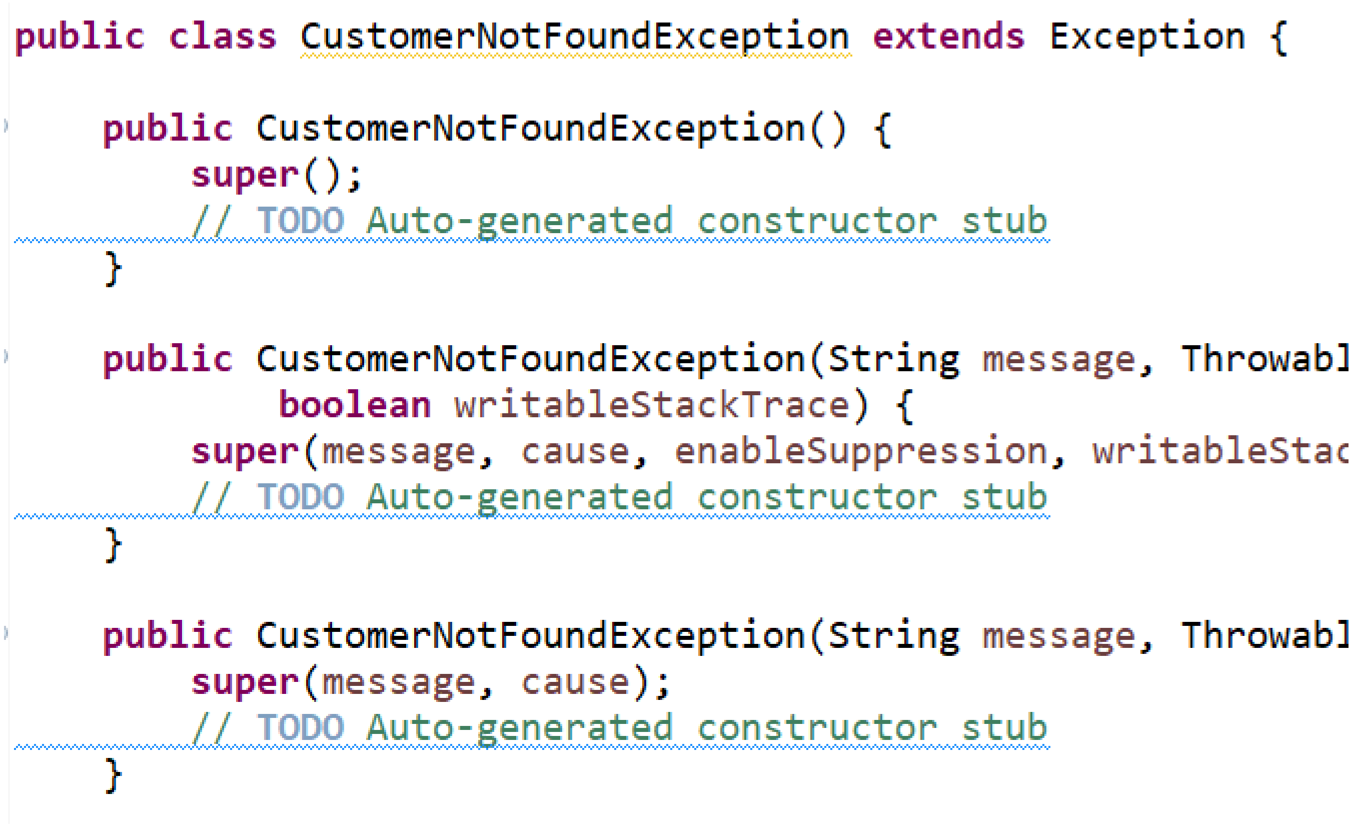
Exception Handling

Exception Handling

- Creating Custom Exception
- Throwing An Exception
- Handling The Thrown Exception

Creating Custom Exception

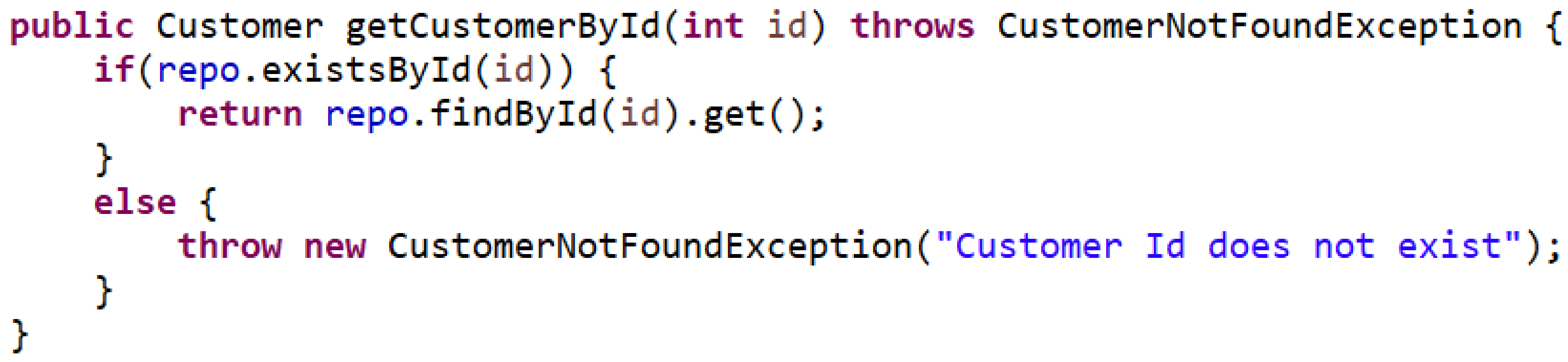
- The custom exception class must inherit the RuntimeException class.
- Inside the constructor, we need to call the super class constructor with an error message string.



```
public class CustomerNotFoundException extends Exception {

    public CustomerNotFoundException() {
        super();
        // TODO Auto-generated constructor stub
    }

    public CustomerNotFoundException(String message, Throwabl
            boolean writableStackTrace) {
        super(message, cause, enableSuppression, writableStac
        // TODO Auto-generated constructor stub
    }

    public CustomerNotFoundException(String message, Throwabl
        super(message, cause);
        // TODO Auto-generated constructor stub
    }
```

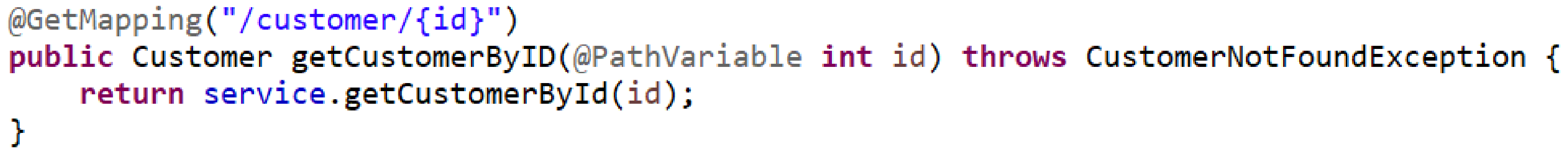
Throwing an Exception

- Now we need to throw an exception whenever something bad happens in our controller.
- CustomerService.java



```
public Customer getCustomerById(int id) throws CustomerNotFoundException {
    if(repo.existsById(id)) {
        return repo.findById(id).get();
    }
    else {
        throw new CustomerNotFoundException("Customer Id does not exist");
    }
}
```

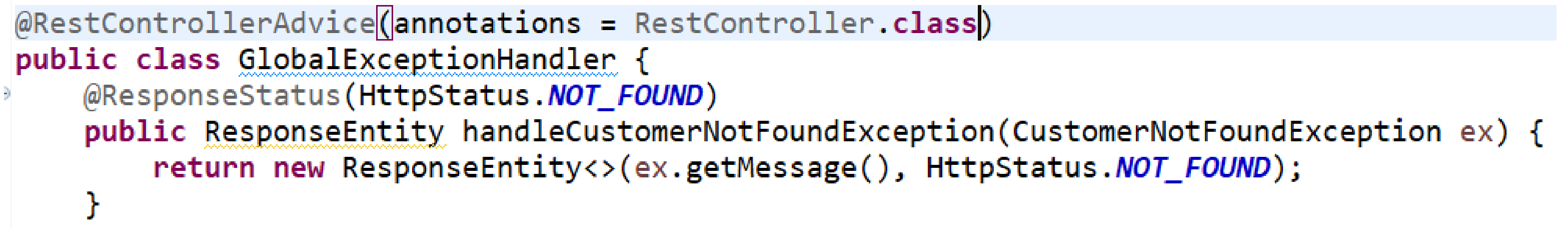
- CustomerController.java



```
@GetMapping("/customer/{id}")
public Customer getCustomerByID(@PathVariable int id) throws CustomerNotFoundException {
    return service.getCustomerById(id);
}
```

Handling the Exception

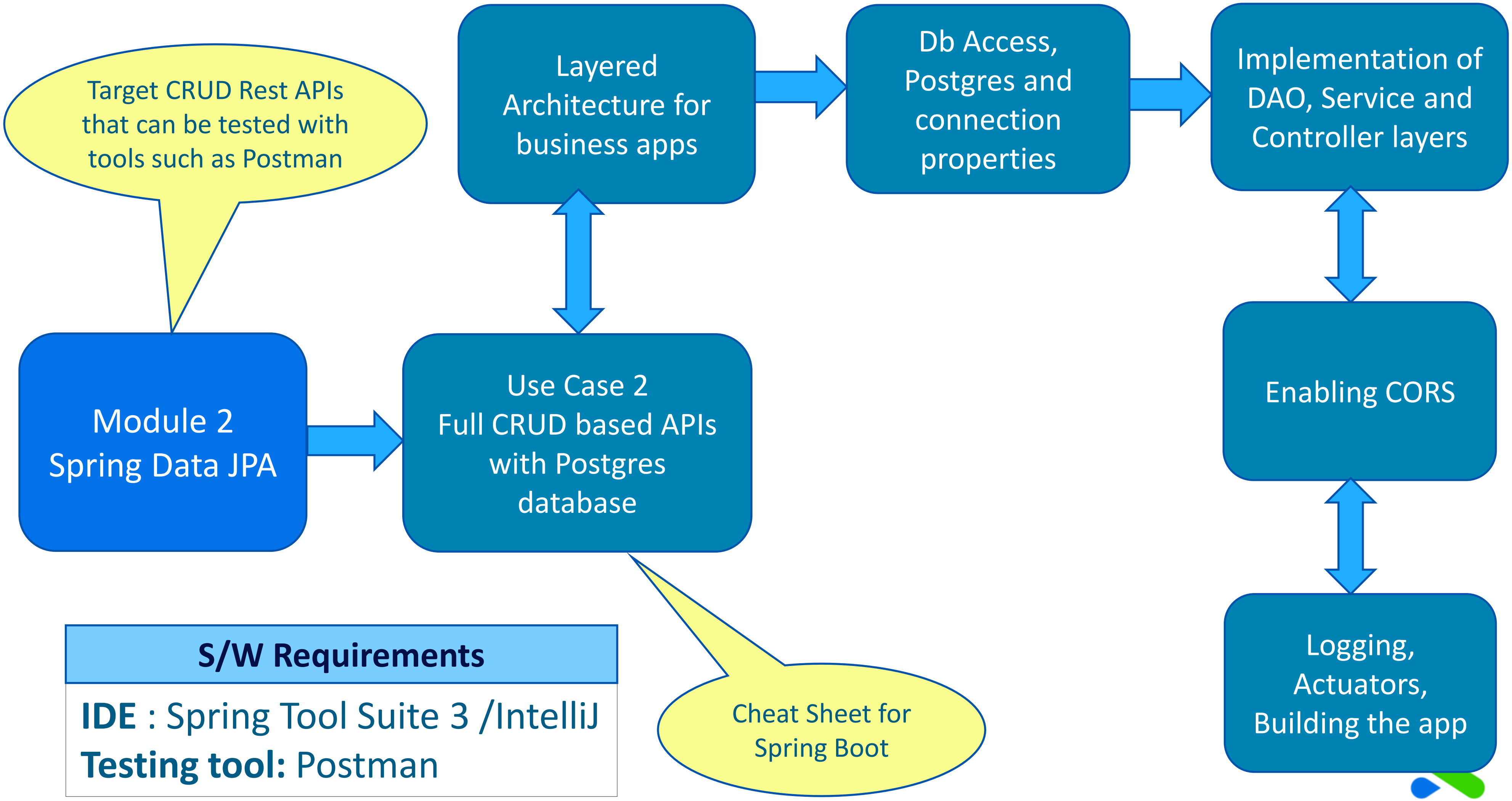
- **@RestControllerAdvice**
  - This annotation tells Spring that the class is designed to provide centralized exception handling for RESTful controllers.
- **@ExceptionHandler(CustomerNotFoundException.class):**
  - This annotation is used on a method to declare that it will handle exceptions of the specified type (CustomerNotFoundException in this case).
  - When a CustomerNotFoundException occurs in any controller method, this method will be invoked to handle it.
- **@ResponseStatus(HttpStatus.NOT_FOUND):**
  - This annotation specifies the HTTP response status code to be set when the exception is handled.
  - In this case, it's set to 404 (NOT FOUND) because the exception is a CustomerNotFoundException.



```java
@RestControllerAdvice(annotations = RestController.class)
public class GlobalExceptionHandler {
    @ResponseStatus(HttpStatus.NOT_FOUND)
    public ResponseEntity handleCustomerNotFoundException(CustomerNotFoundException ex) {
        return new ResponseEntity<>(ex.getMessage(), HttpStatus.NOT_FOUND);
    }
```

Module 2 :Spring Data JPA

Module 2 Design


Target CRUD Rest APIs that can be tested with tools such as Postman
Layered Architecture for business apps
Db Access, Postgres and connection properties
Implementation of DAO, Service and Controller layers
Module 2 Spring Data JPA
Use Case 2 Full CRUD based APIs with Postgres database
Enabling CORS
Cheat Sheet for Spring Boot
Logging, Actuators, Building the app

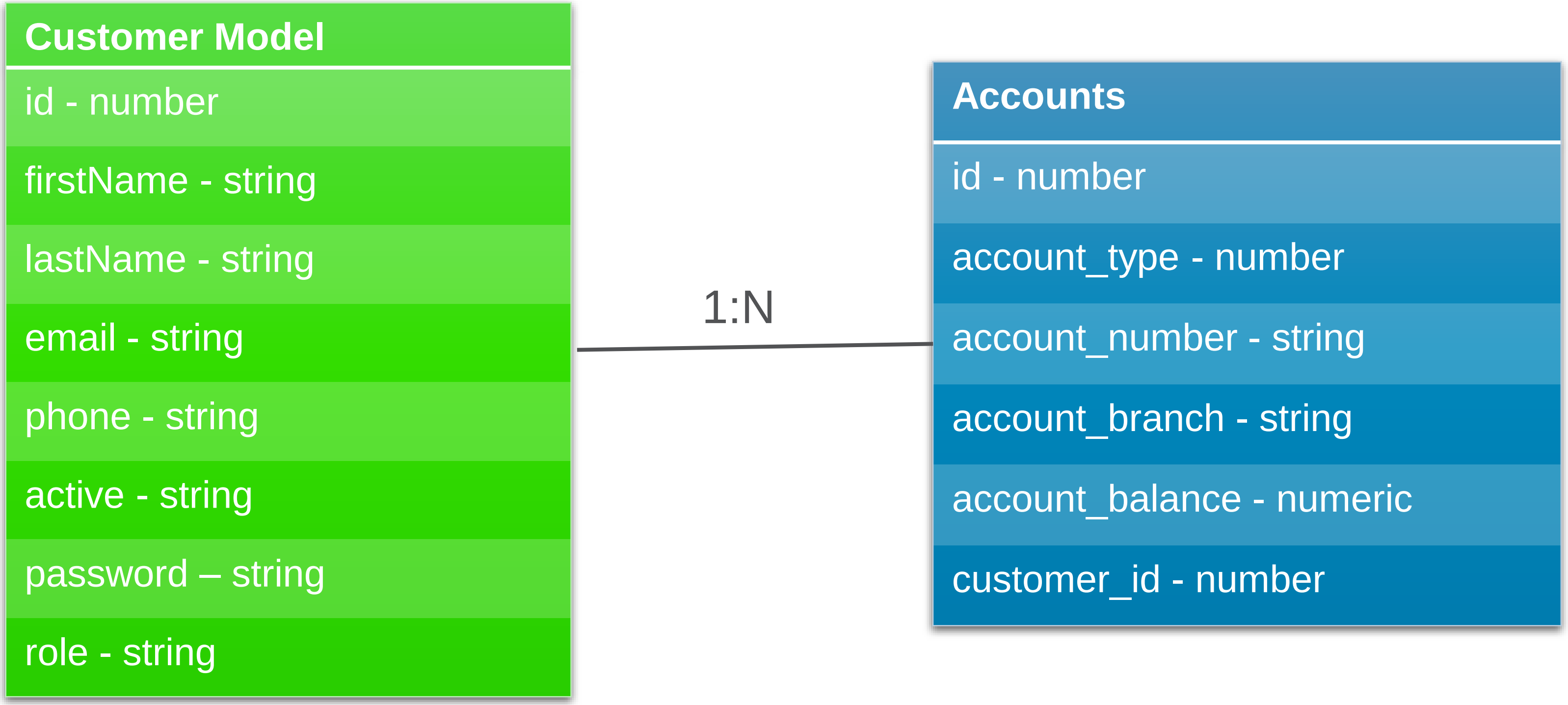
S/W Requirements
IDE : Spring Tool Suite 3 /IntelliJ **Testing tool:** Postman

Module 2 Overview

- Introduction to layered architecture of building enterprise apps/server backends
- Introduction to database access in Spring Boot
- Setting up **Postgres** connection, connection pooling etc
- Writing a full fledged **DAO(Data access layer)** using **JPA**
- Handling http responses in controllers using **ResponseEntity** objects
- Introduction Cross Origin Request (**CORS**)
- Enabling CORS globally or in controllers/methods
- More on **application.properties** to configure various spring boot properties
- Logging and Actuators

Use case 2

- All of the operations should exactly work similar to Use case1
- But, Customer data should be fetched from the postgres database compared to in-memory
- 2 tables should be created in the database (**Customers** and **Accounts**)
- Accounts table should have a foreign key relationship with the Customers table through (**customer_id**) as the foreign key as demonstrated in the below diagram



Customer Model
id - number
firstName - string
lastName - string
email - string
phone - string
active - string
password – string
role - string

1:N

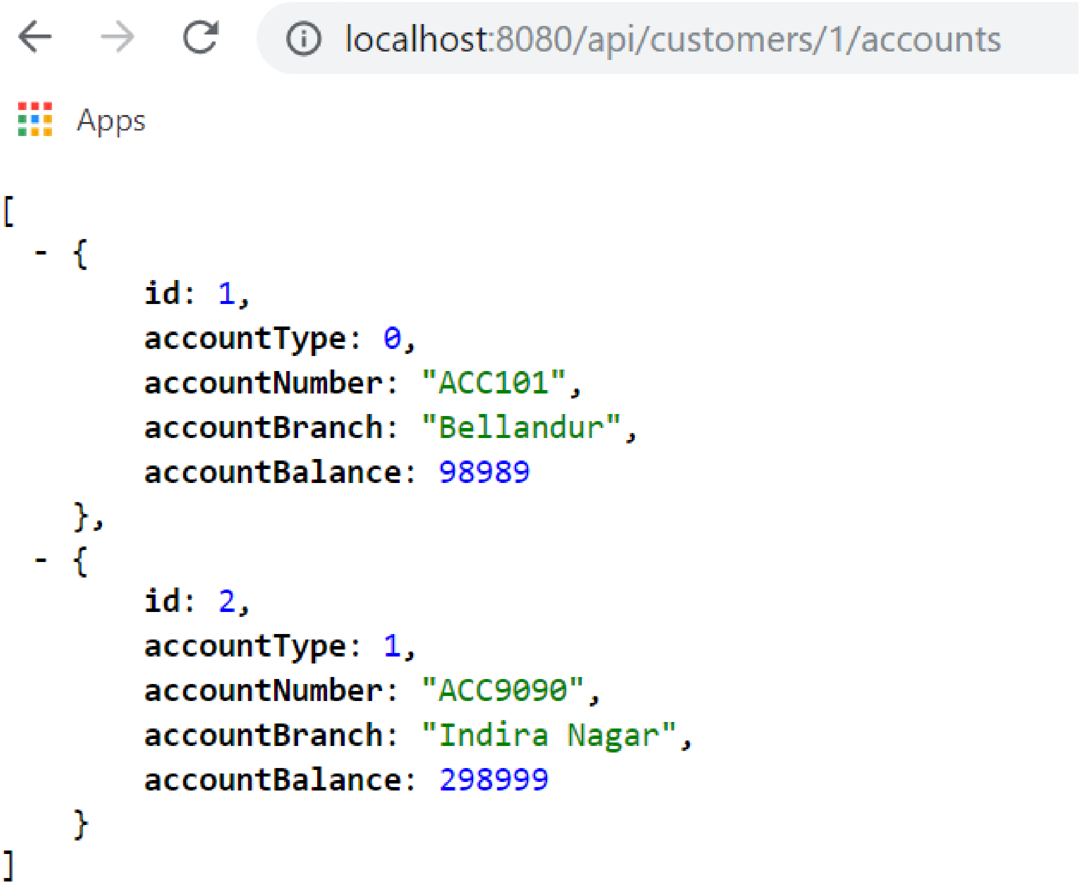
Accounts
id - number
account_type - number
account_number - string
account_branch - string
account_balance - numeric
customer_id - number

Use case 2 explained

- The system should have 2 entities named **Customers** and **Accounts**
- There should be an one to many association between customers and accounts i.e a customer can have multiple accounts
- Apart from all the REST apis that are supported in the use case 1, there should be a separate API for accounts as well as shown below
- Basically the REST client should be able to consume the customer based apis as well as the accounts based apis
- A customer can have multiple accounts and hence the backend should provide an api to retrieve all the accounts of a particular customer
- This is demonstrated in the next slide through a diagram

Use case 2 – Accounts REST API

- GET **/api/customers/1/accounts** – should fetch the list of accounts of the customer with id 1



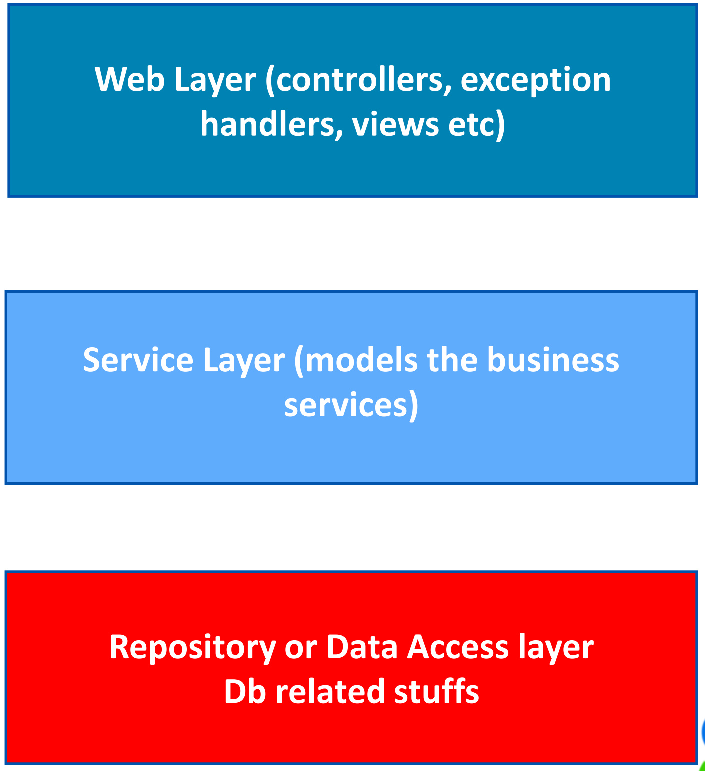
```
[
  - {
        id: 1,
        accountType: 0,
        accountNumber: "ACC101",
        accountBranch: "Bellandur",
        accountBalance: 98989
    },
  - {
        id: 2,
        accountType: 1,
        accountNumber: "ACC9090",
        accountBranch: "Indira Nagar",
        accountBalance: 298999
    }
]
```

Layered Architecture

Layered Architecture

- The modern enterprise applications are layered in a 2-tier or a 3-tire architecture based on the complexity of the application
- The most common is the 3-tier architecture as shown below

- As seen, the most common layers are
  - Web layer
  - Service layer
  - DAO or Repository layer



Web Layer (controllers, exception handlers, views etc)

Service Layer (models the business services)

Repository or Data Access layer
Db related stuffs

Layered Architecture

- **Web Layer**
  - It is the uppermost layer of a web application
  - responsible of processing user's input and returning the correct response back to the user
  - The web layer must also handle the exceptions thrown by the other layers
  - it must take care of authentication and act as a first line of defense against unauthorized users

- **Service Layer**
  - It is the plumbing between the UI and the backend systems
  - It is in charge of managing the business rules of transforming and translating data between the two layers (Web and DAL)

- **DAL(Data access layer) or Repository Layer**
  - It is the lowermost layer of a web application
  - It is responsible for communicating with the data storage, filesystems etc

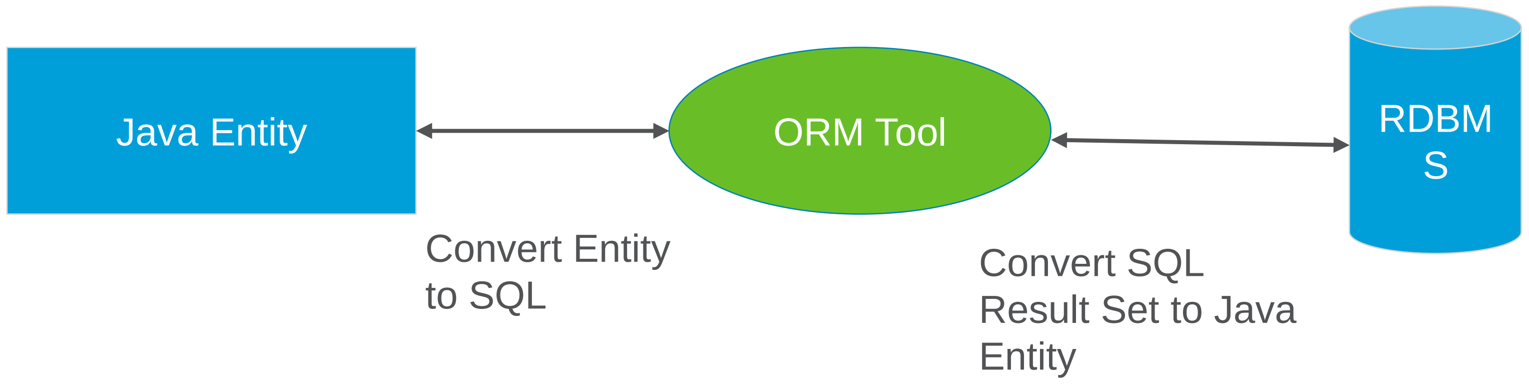
Adding a DAL (Data access Layer)

- In the first module, we created POJO classes and also created simple service class
- The service class was responsible for creating in-memory objects
- But in real world apps, the actual data should come from the database
- The Data access layer or the repository layer takes care of interacting with the database and returning that data to the service layer
- We are going to use JDBC method of connecting to the database

ORM and JPA

What is ORM?

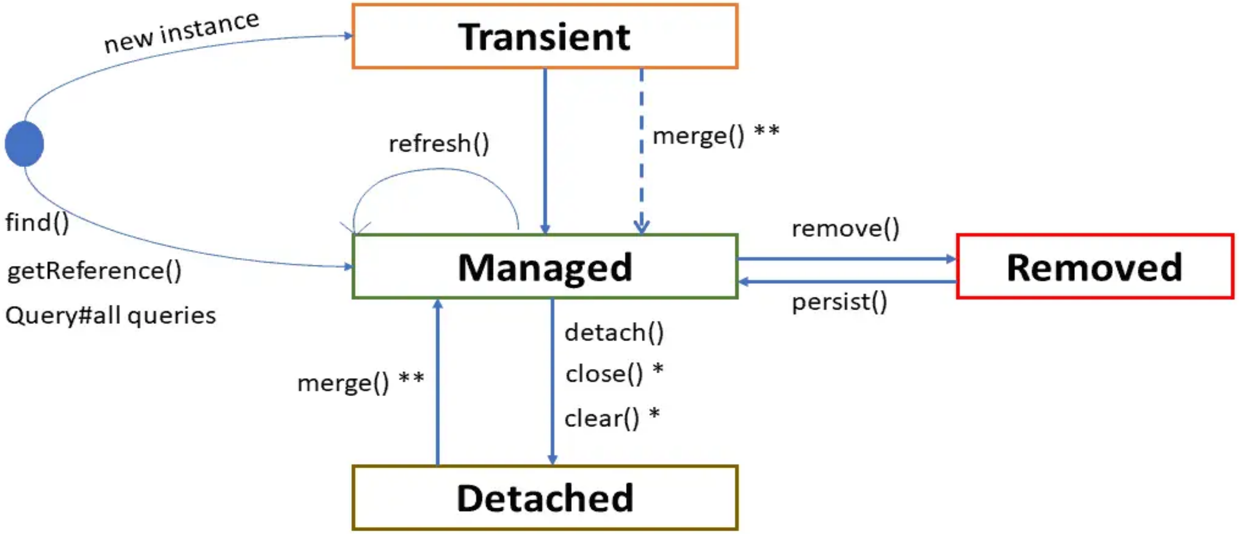
Object-Relational Mapping (ORM) is the process of converting Java objects to database tables.
In other words, this allows us to interact with a relational database without any SQL.



What is JPA?

- JPA stands for Java Persistent API. It is an official specification from Java
- That dictates the APIs to be used to persist Entities (Java Objects) in the database.
- This specification is implemented by many ORM Tools
- There are several ORM tools that implement their own API.
- This becomes a portability problem if you want to switch between them for any reason.
- This is where JPA helps you by providing an API layer on top of ORM tools.

JPA- Entity Life Cycle Management



* → Effects all instances in the Persistence Context

** → Merge returns a managed instance, original remains in same state

JPA- Entity Life Cycle Management

Transient State :

- When an entity object is initially created, it's state is **New or Transient**. In this state, the object is not yet associated with an EntityManager and has no representation in the database.

Persistent/Managed State :

- An entity object becomes **Managed or Persistent** when it is persisted to the database via an EntityManager's ***persist()*** method, which must be invoked within an active transaction.

- If we are changing persistent State object values those are synchronized automatically with the database while committing the transaction.

JPA- Entity Life Cycle Management

Detached State :

- The state, **Detached**, represents entity objects that have been disconnected from the EntityManager.

Removed State :

- A persistent state entity object can also be marked for deletion, by using the EntityManger’s remove() with in an active transaction.

- Then entity object changes its state from **Managed** to **Removed,** and physically deleted from the database during commit.

What is Persistence Context?

- The persistence context is the collection of all persistent (managed) objects of an EntityManager.

- If we try to retrieve an entity, if the entity is existed in persistence context, then managed entity object returned from persistence context without accessing the database.

- The main role of the persistence context is to make sure that a database entity object is represented by no more than one in-memory entity object within the same EntityManager.

- Every EntityManager manages it’s own persistence context.

- Persistence context act as a local cache for a given EntityManager.

Spring Data JPA

What is Spring Data JPA?

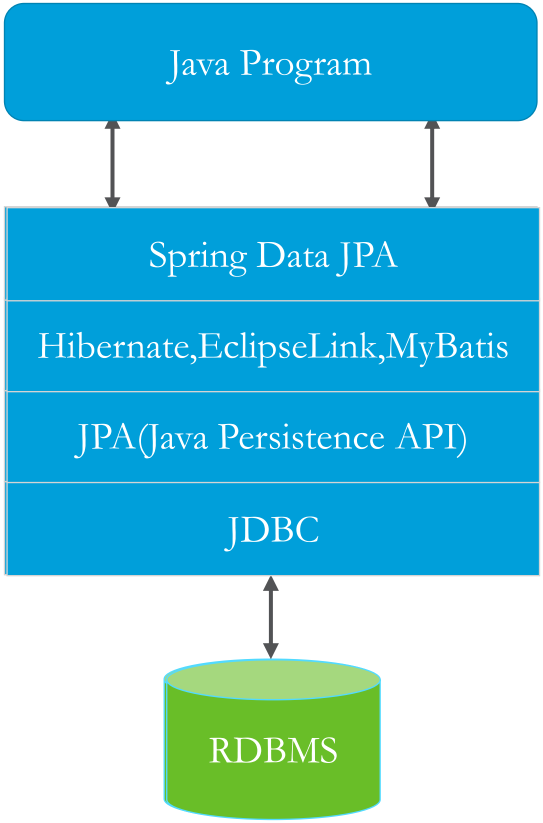
- Spring Data JPA focuses on using JPA to store and manage data in a relational database.

- To perform simple operation needs to write too much boilerplate code to execute simple queries.

- Spring Data JPA aims to significantly improve the implementation of data access layers by reducing the effort to the amount that’s needed.

- As a developer you write your repository interfaces, including custom finder methods, and Spring will provide the implementation automatically.

Spring Data JPA



Java Program

Spring Data JPA

Hibernate,EclipseLink,MyBatis

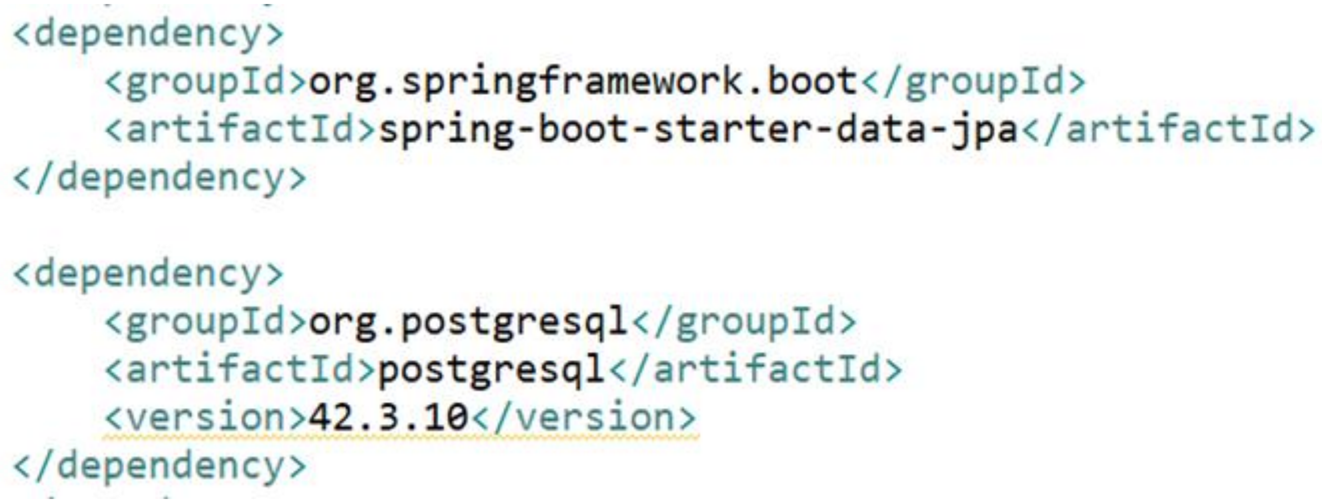
JPA(Java Persistence API)

JDBC

RDBMS

JPA Dependency using Maven

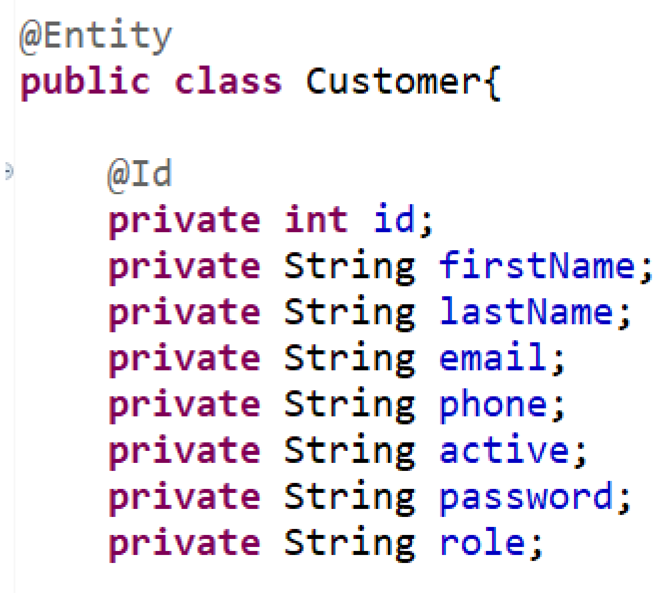
- **spring-boot-starter-data-jpa** spring boot starters
- We can use the either of the following maven dependencies



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

<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <version>42.3.10</version>
</dependency>
```

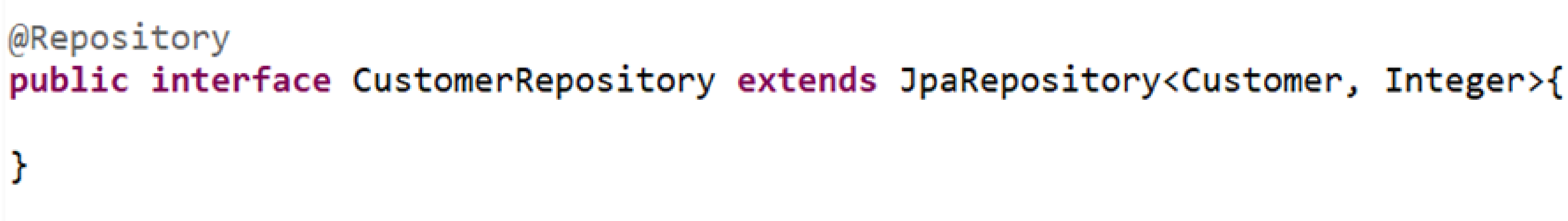
Entity of Customer and Repository

- The Customer class is annotated with @Entity, indicating that it is a JPA entity..
- The Customer object’s id property is annotated with @Id so that JPA recognizes it as the object’s ID/primary key



```java
@Entity
public class Customer{

    @Id
    private int id;
    private String firstName;
    private String lastName;
    private String email;
    private String phone;
    private String active;
    private String password;
    private String role;
```

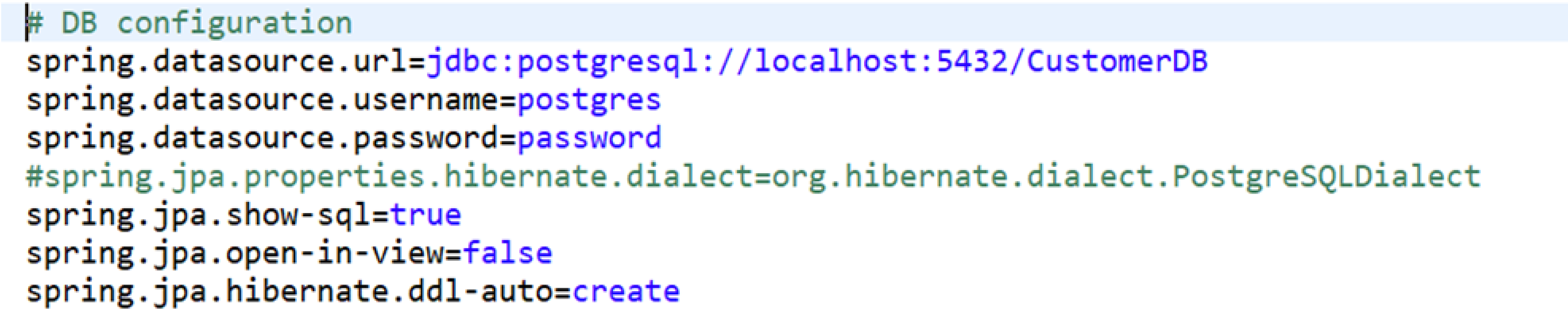
Customer Repository



```java
@Repository
public interface CustomerRepository extends JpaRepository<Customer, Integer>{

}
```

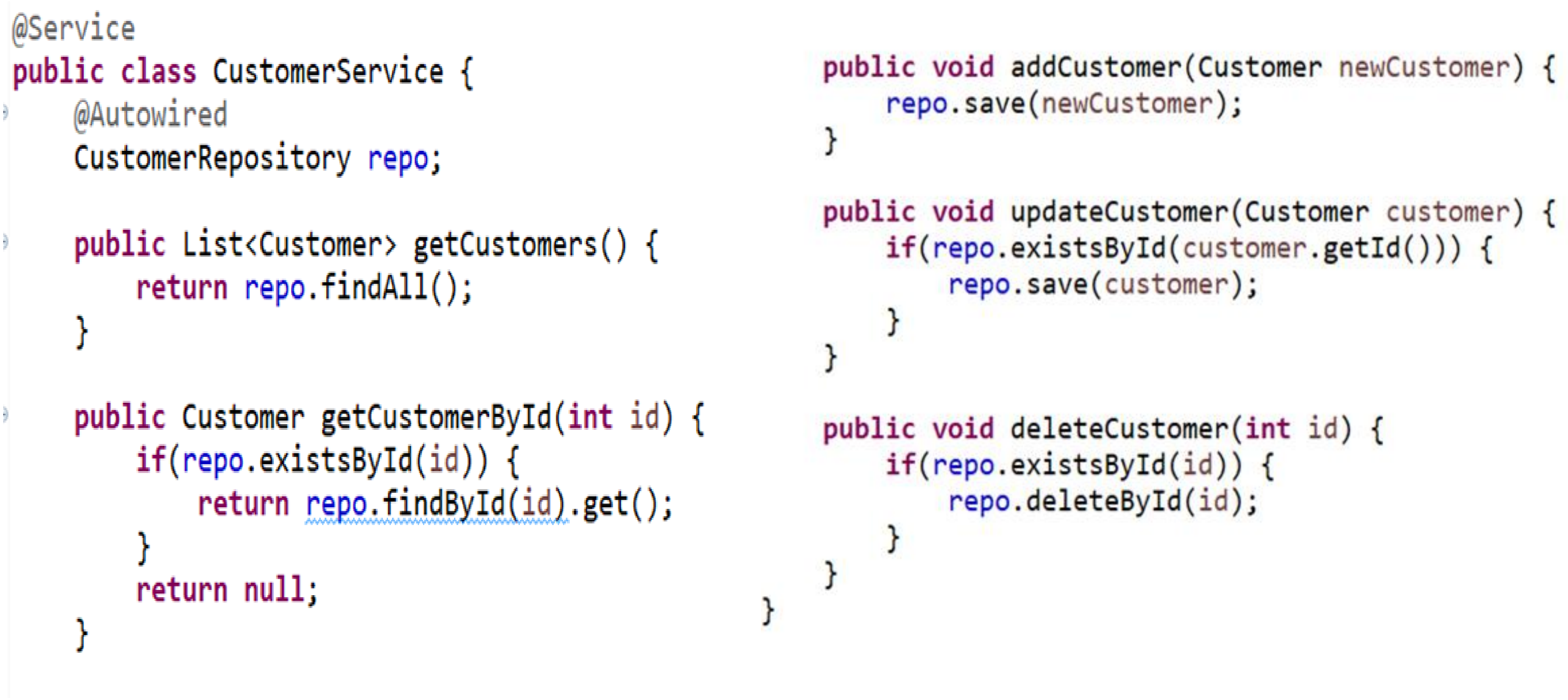
Datasource and connection properties

- DataSource and Connection Pool are configured in **application.properties** file using prefix **spring.datasource**
- In our scenario, where we have to use postgres, the following settings can be added in application.properties file
- The application.properties is typically located in the resources subfolder inside the app folder
- Spring Boot uses tomcat connection pooling by default for performance and concurrency



```
# DB configuration
spring.datasource.url=jdbc:postgresql://localhost:5432/CustomerDB
spring.datasource.username=postgres
spring.datasource.password=password
#spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
spring.jpa.show-sql=true
spring.jpa.open-in-view=false
spring.jpa.hibernate.ddl-auto=create
```

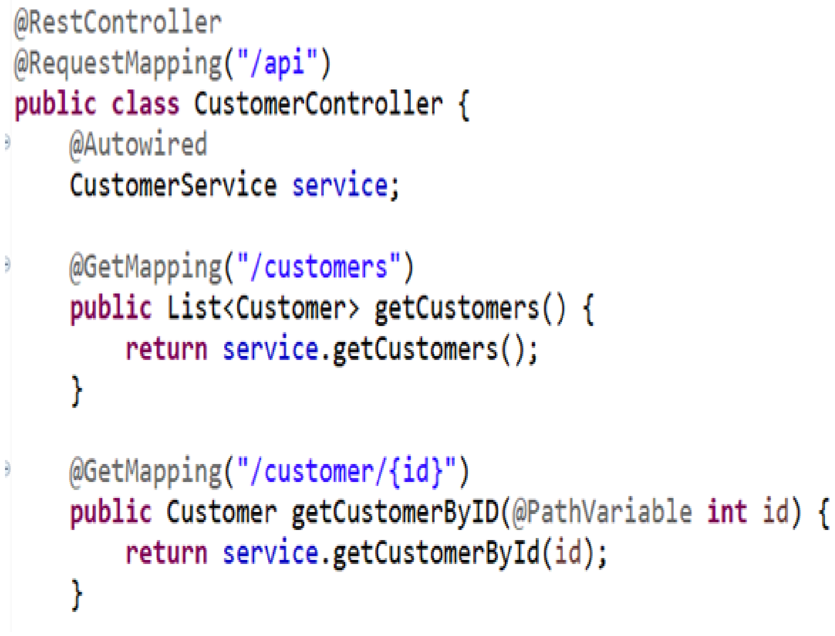
CustomerService



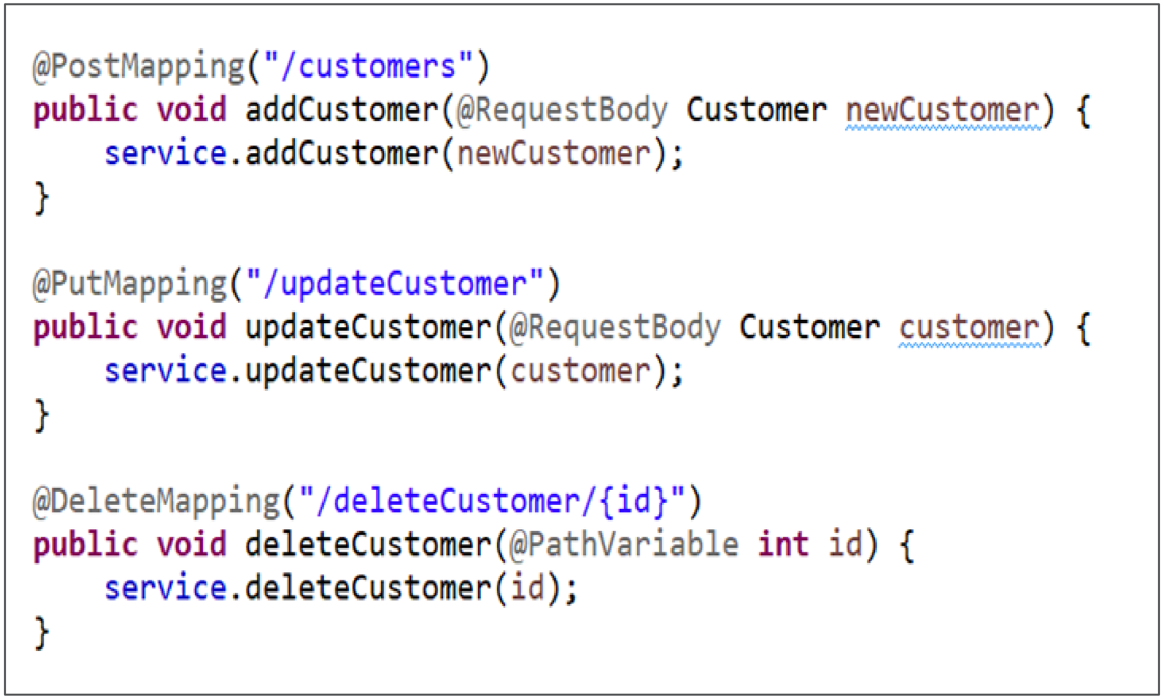
```
@Service
public class CustomerService {
    @Autowired
    CustomerRepository repo;

    public List<Customer> getCustomers() {
        return repo.findAll();
    }

    public Customer getCustomerById(int id) {
        if(repo.existsById(id)) {
            return repo.findById(id).get();
        }
        return null;
    }

    public void addCustomer(Customer newCustomer) {
        repo.save(newCustomer);
    }

    public void updateCustomer(Customer customer) {
        if(repo.existsById(customer.getId())) {
            repo.save(customer);
        }
    }

    public void deleteCustomer(int id) {
        if(repo.existsById(id)) {
            repo.deleteById(id);
        }
    }
}
```

Controller for @RequestMapping

- We can autowire in the classes annotated with spring stereotypes such as **@Component, @Service, @Repository and @Controller**



```
@RestController
@RequestMapping("/api")
public class CustomerController {
    @Autowired
    CustomerService service;

    @GetMapping("/customers")
    public List<Customer> getCustomers() {
        return service.getCustomers();
    }

    @GetMapping("/customer/{id}")
    public Customer getCustomerByID(@PathVariable int id) {
        return service.getCustomerById(id);
    }
```



```
@PostMapping("/customers")
public void addCustomer(@RequestBody Customer newCustomer) {
    service.addCustomer(newCustomer);
}

@PutMapping("/updateCustomer")
public void updateCustomer(@RequestBody Customer customer) {
    service.updateCustomer(customer);
}

@DeleteMapping("/deleteCustomer/{id}")
public void deleteCustomer(@PathVariable int id) {
    service.deleteCustomer(id);
}
```

Annotations

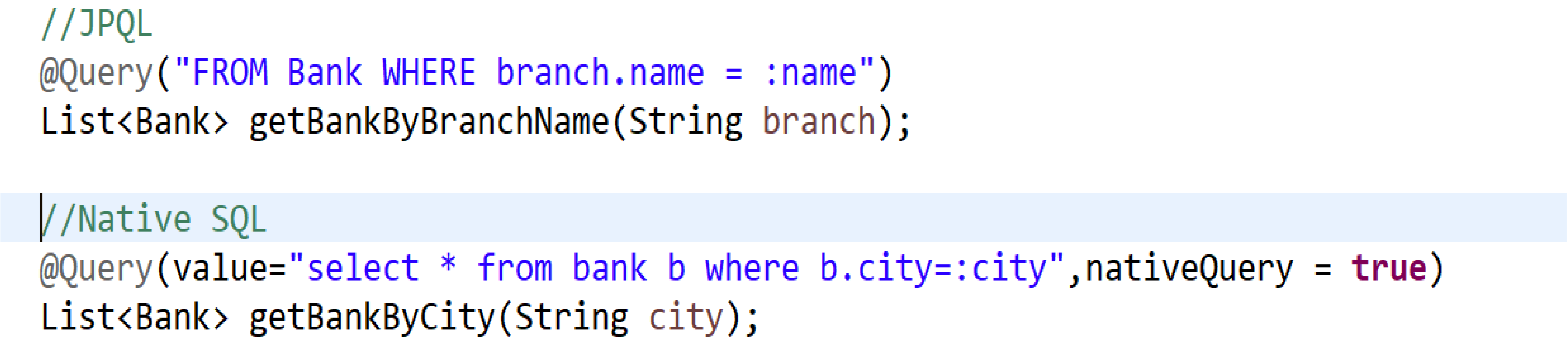
- **@Entity** - This annotation defines that a class can be mapped to a table.
- **@Table(name = "STUDENT")** -The name of the database table to be used for mapping.
- **@Id** - This annotation specifies the primary key of the entity.
- **@GeneratedValue**- This annotation is used to specify the primary key generation strategy to use.
- **@Column(name = "id")** – adding name of the column in the database.
- **@Embedded** - used to embed a type into another entity.
- **@Embeddable -** to declare that a class will be embedded by other entities.
- **@Transient** - By using @Transient we can exclude the specific field or method from being persisted in our database.

axess academy

Custom Query

- Spring Data provides many ways to define a query that we can execute. One of these is the @Query annotation.

- In spring data JPA @Query annotation will help us to to execute both
  - **JPQL**
  - **Native Query**.



```
//JPQL
@Query("FROM Bank WHERE branch.name = :name")
List<Bank> getBankByBranchName(String branch);

//Native SQL
@Query(value="select * from bank b where b.city=:city",nativeQuery = true)
List<Bank> getBankByCity(String city);
```

JPQL(Java Persistence Query language)

- JPQL is an object-oriented query language which is used to perform database operations on persistence entities.
- Instead of database table, JPQL uses entity object model to operate the SQL queries.
- Benefits of using JPQL
  - JPQL is portable across implementer and database
  - No manual conversion of row data into Object and vice-versa
  - Queries can be stored in cache to provide better performance

Association Mapping

Association Mapping

- ➢Association mapping represents relationship between entities.
- ➢A java class can contain an object of another class or a set of another class.
- ➢There is no directionality involved in relational world, Its just a matter of writing a query. But there is notion of directionality which is possible in java.
- ➢Hence association are classified as
  - Unidirectional
  - Bidirectional

Association Mapping

Employee – Passport	One To One	One Employee can have one passport
Bank – Branch	One To Many	One Bank can have multiple branch
Branch – Bank	Many To One	Many Branches belongs to One Bank
Account – Customer	Many To Many	One Account belongs to many customer and one customer can have one many account

One To One

```
@Entity
public class Customer {
        @Id
        private long customerId;
        private String name;
        @OneToOne(cascade=CascadeType.ALL)
        @JoinColumn(name = "addressId")
        private Address address;
}
```

```
@Entity
public class Address {
        @Id
        private long addressId;
        private String street;
        private String city;
        private String state;
        private String country;
}
```

One-To-Many

```
@Entity
public class Bank {
        @Id
        private String code;
        private String name;
        @OneToMany(mappedBy =
"bank")
        private Set<Branch> branch;
}
```

```
@Entity
public class Branch {
        @Id
        private long branchCode;
        private String name;
        private String address;
}
```

Many-To One

```
@Entity
public class Bank {
        @Id
        private String code;
        private String name;
}
```

```
@Entity
public class Branch {
        @Id
        private long branchCode;
        private String name;
        private String address;
        @ManyToOne
        @JoinColumn(name =
"bank_code")
        private Bank bank;
}
```

What Is Cascading?

When we perform some action on the target entity, the same action will also be applied to the associated entity.

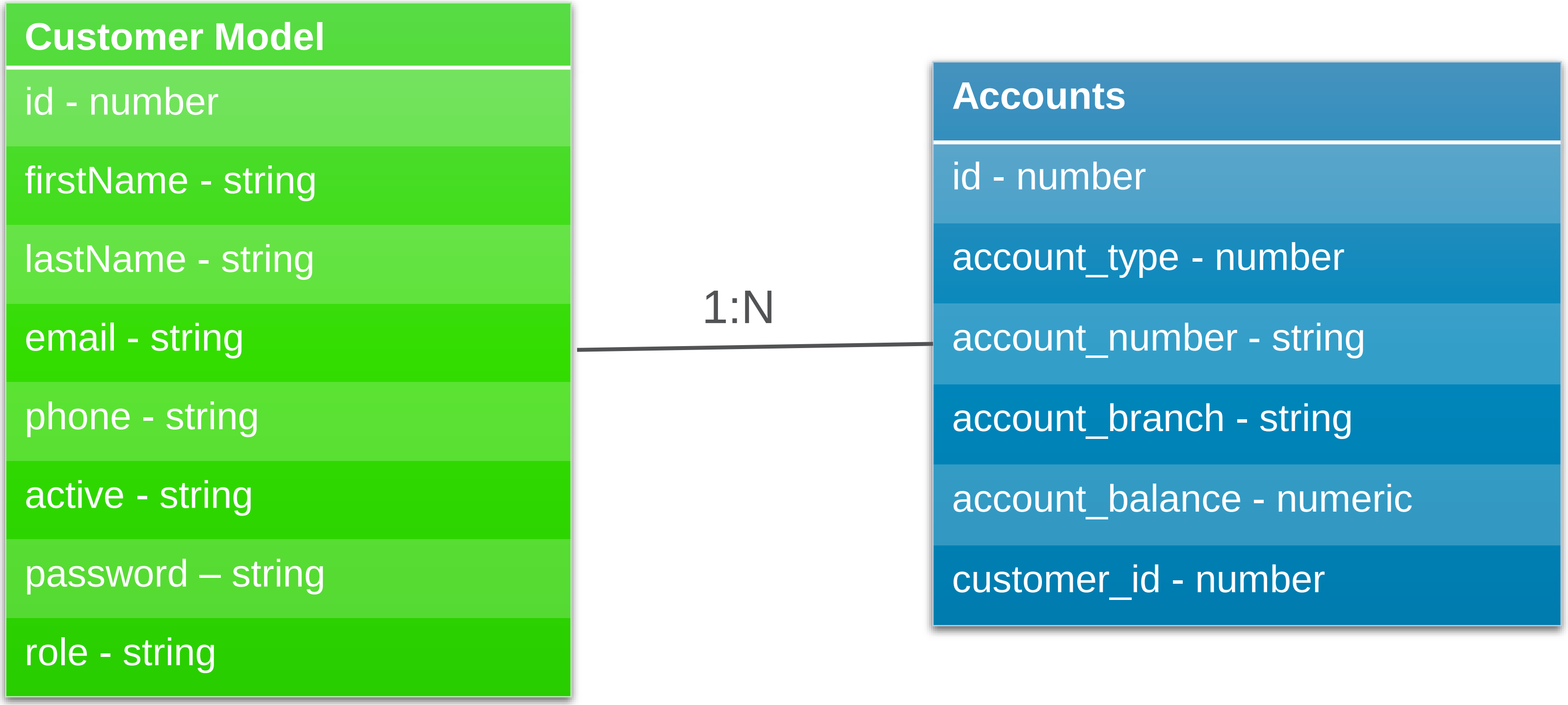
Cascade Type :

- PERSIST - propagates the Persist operation from a parent to the following child entity.
- MERGE - propagates the merge operation from a parent to the following child entity.
- REMOVE - propagates the removal operation from parent to following child entity.
- REFRESH - the child entity also gets reloaded from the database whenever the parent entity is also being refreshed.
- DETACH - - the child entity will eventually get removed from the persistent context.
- ALL - propagate all the operations - from a parent to a child entity.

Let us implement use case

Use case 2

- All of the operations should exactly work similar to Use case 1 (slides 4 to 7)
- But, Customer data should be fetched from the postgres database compared to in-memory
- 2 tables should be created in the database (**Customers** and **Accounts**)
- Accounts table should have a foreign key relationship with the Customers table through (**customer_id**) as the foreign key as demonstrated in the below diagram



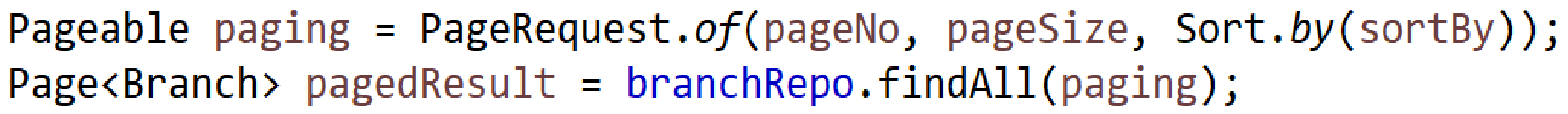
Customer Model
id - number
firstName - string
lastName - string
email - string
phone - string
active - string
password – string
role - string

Accounts
id - number
account_type - number
account_number - string
account_branch - string
account_balance - numeric
customer_id - number

Pagination and Transaction

Pagination and Sorting

- Pagination is the process of displaying the data on multiple pages rather than showing them on a single page.
- Also, we often need to sort that data by some criteria while paging.
- You usually do pagination when there is a database with numerous records. Dividing those records increases the readability of the data.
- It can retrieve this data as per the user’s requests.



```
Pageable paging = PageRequest.of(pageNo, pageSize, Sort.by(sortBy));
Page<Branch> pagedResult = branchRepo.findAll(paging);
```

Transaction

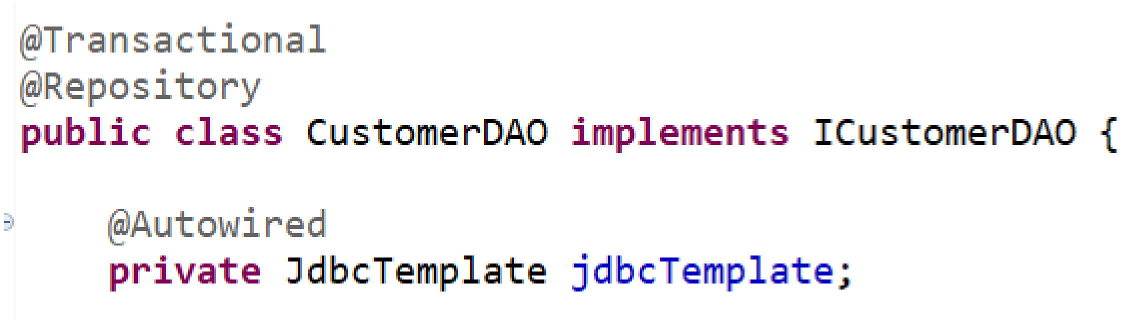
➢ **A transaction is a unit of work**. We generally group the related work within one transaction so that if part of the work is failed then entire transaction should be failed. A transaction can be described by ACID properties (Atomicity, Consistency, Isolation and Durability).

- Atomicity : The entire transaction takes place at once or doesn’t happen at all.
- Consistency: The database must be consistent before and after the transaction.
- Isolation: Multiple Transaction occur independently without interference.
- Durability: The changes of successful transaction occurs even if the system failure occur.

➢ **@Transactional annotation** provides an easy way to declare your transaction handling in Spring.

Transaction Management

- **@Transactional** annotation is used for managing transactions in jdbc queries
- It can be applied in class level or method level where in we run the **JdbcTemplate** queries



```
@Transactional
@Repository
public class CustomerDAO implements ICustomerDAO {

    @Autowired
    private JdbcTemplate jdbcTemplate;
```

- In this we have attached **@Transactional** annotation with the entire CustomerDAO class, which deals with the CRUD operations
- Hence every jdbc query in the methods will be carried over as transactions