



# SPRING BOOT

---

## QUICK START

Version: 1.0

Date: 03.2021

[www.ivoronline.com](http://www.ivoronline.com)

<b>1 QUICK START .....</b>	<b>8</b>
<b>1.1 Install.....</b>	<b>9</b>
1.1.1 Java.....	10
1.1.2 IntelliJ IDEA.....	12
<b>1.2 Create Project with IntelliJ .....</b>	<b>16</b>
1.2.1 Community Edition (free).....	17
1.2.2 Ultimate Edition (paid).....	19
<b>1.3 Application .....</b>	<b>21</b>
1.3.1 Run .....	22
1.3.2 Deploy .....	23
<b>2 MAIN TERMS .....</b>	<b>26</b>
<b>2.1 Project.....</b>	<b>27</b>
2.1.1 Spring Boot Starters .....	28
2.1.2 Spring Initializer .....	29
2.1.3 Spring Boot Project .....	30
<b>2.2 Object Types.....</b>	<b>32</b>
2.2.1 Entity Object (DO) .....	35
2.2.2 Data Access Object (DAO) .....	37
2.2.3 Data Transfer Object (DTO) .....	38
2.2.4 Data Converter Object (DCO) .....	40
2.2.5 Controller Object (CO) .....	41
<b>2.3 Controller .....</b>	<b>42</b>
2.3.1 @Controller .....	43
2.3.2 @RestController .....	45
<b>2.4 Endpoint.....</b>	<b>47</b>
2.4.1 @RequestMapping .....	48
2.4.2 @GetMapping .....	50
2.4.3 @PostMapping .....	52
2.4.4 @RequestParam .....	54
2.4.5 @PathVariable.....	56
2.4.6 Return - Data - Text.....	58
2.4.7 Return - Data - JSON .....	60
2.4.8 Return - View - HTML.....	63
2.4.9 Return - View - Thymeleaf.....	65
2.4.10 Return - View - JSP .....	67
2.4.11 Download - Excel .....	70
2.4.12 Download - Excel - With Header.....	75
<b>2.5 Entity.....</b>	<b>80</b>
2.5.1 Properties - Public.....	81
2.5.2 Properties - Private - Getters & Setters .....	84
2.5.3 Properties - Private - Getters & Constructor.....	87
2.5.4 @Entity.....	90
2.5.5 @Id.....	95
2.5.6 @IdClass.....	100
2.5.7 @EmbeddedId .....	106
2.5.8 @Table .....	112

2.5.9	@Column.....	116
2.5.10	@Transient .....	120
2.5.11	Recommended Annotations.....	125
<b>2.6</b>	<b>Databases.....</b>	<b>128</b>
2.6.1	H2.....	129
2.6.2	MySQL .....	133
2.6.3	PostgreSQL .....	137
2.6.4	MongoDB.....	141
<b>2.7</b>	<b>Relationships.....</b>	<b>144</b>
2.7.1	@OneToOne - Join Table .....	145
2.7.2	@OneToOne - Foreign Key.....	150
2.7.3	@OneToMany - Join Table .....	155
2.7.4	@OneToMany - Foreign Key .....	160
2.7.5	@ManyToMany - Join Table.....	166
<b>2.8</b>	<b>Service.....</b>	<b>172</b>
2.8.1	Business Logic - Inside Controller .....	173
2.8.2	Business Logic - Inside Service.....	176
2.8.3	Instantiate Service - Using Class .....	179
2.8.4	Instantiate Service - Using Class - @Autowired .....	182
2.8.5	Instantiate Service - Using Interface.....	185
2.8.6	Instantiate Service - Using Interface - @Autowired .....	188
<b>2.9</b>	<b>@Autowired .....</b>	<b>191</b>
2.9.1	Manually.....	192
2.9.2	Location - Property .....	195
2.9.3	Location - Setter .....	198
2.9.4	Location - Constructor .....	201
2.9.5	Interface - @Primary.....	204
2.9.6	Interface - @Qualifier .....	207
2.9.7	Interface - @Profile.....	210
<b>2.10</b>	<b>DTO - Serialize .....</b>	<b>214</b>
2.10.1	@ResponseBody .....	215
2.10.2	@JsonSerialize .....	217
<b>2.11</b>	<b>DTO - Deserialize .....</b>	<b>221</b>
2.11.1	From Request Parameters - Using Setters .....	225
2.11.2	From Request Parameters - Using Setters - Customized.....	228
2.11.3	From Request Parameters - Using Map.....	231
2.11.4	From Request Parameters - Using Map - Customize Setter .....	234
2.11.5	From JSON - @RequestBody - Properties .....	237
2.11.6	From JSON - @RequestBody - Setters .....	241
2.11.7	From JSON - @RequestBody - Constructor.....	245
2.11.8	From JSON - @RequestBody - Constructor - Custom .....	250
2.11.9	From JSON - @JsonProperty .....	254
2.11.10	From JSON - @JsonFormat.....	258
2.11.11	From JSON - @JsonDeserialize .....	262
2.11.12	From JSON Array.....	267
<b>2.12</b>	<b>DTO to Entity .....</b>	<b>271</b>
2.12.1	Manually.....	272
2.12.2	JMapper - Using Annotations .....	276
<b>2.13</b>	<b>Profiles .....</b>	<b>281</b>

2.13.1	Assign Profile - To Class .....	282
2.13.2	Assign Profile - To application.properties .....	283
2.13.3	Specify Active Profile - Through Run Configuration .....	286
2.13.4	Specify Active Profile - Through application.properties .....	287
<b>2.14</b>	<b>application.properties.....</b>	<b>288</b>
2.14.1	Read Property - From Controller .....	289
2.14.2	Read Property - From HTML.....	291
2.14.3	Built-in Properties .....	293
<b>2.15</b>	<b>DB Queries .....</b>	<b>294</b>
2.15.1	Native Query.....	297
2.15.2	JPQL.....	304
<b>3</b>	<b>DEMO APPLICATIONS.....</b>	<b>311</b>
<b>3.1</b>	<b>Simple Applications.....</b>	<b>312</b>
3.1.1	Request - Store Entity from JSON .....	313
3.1.2	Request - Store Entities from JSON Array .....	317
3.1.3	Response - Store Entities from JSON Array .....	321
3.1.4	Use Filter to Log HTTP Parameters into DB.....	326
<b>3.2</b>	<b>Authors &amp; Books .....</b>	<b>331</b>
3.2.1	Step 1: Create Project .....	333
3.2.2	Step 2: Entity - Author.....	337
3.2.3	Step 3: Entity - Book.....	342
3.2.4	Step 4: Relationship @OneToMany.....	347
3.2.5	Step 5: Service - Add Author .....	353
3.2.6	Step 6: Service - Add Book .....	357
3.2.7	Step 7: Service - Get Books.....	361
3.2.8	Step 8: Service - Add Author & Book .....	365
<b>4</b>	<b>APPENDIX .....</b>	<b>371</b>
<b>4.1</b>	<b>Lombok.....</b>	<b>372</b>
4.1.1	Lombok API - Use .....	373
4.1.2	Lombok Plugin - Install for IntelliJ.....	376
4.1.3	Lombok Plugin - Lombok .....	377
4.1.4	Lombok Plugin - Delombok .....	378
<b>4.2</b>	<b>IntelliJ .....</b>	<b>380</b>
4.2.1	Copy Project .....	381
4.2.2	DB Tools.....	386
<b>4.3</b>	<b>IntelliJ - Developer Tools .....</b>	<b>389</b>
4.3.1	Install Chrome Extension: Live Reload .....	390
4.3.2	IntelliJ Registry: compiler.automake .....	391
4.3.3	Run Configuration: Update classes and resources .....	392
4.3.4	Settings: Build Project Automatically .....	394
4.3.5	Create Spring Boot Project .....	396
<b>4.4</b>	<b>Database - H2 .....</b>	<b>399</b>
4.4.1	Application Example .....	401
4.4.2	DB: Specify Name.....	406
4.4.3	Console: Enable .....	407
4.4.4	Console: Open .....	408
4.4.5	Console: Show Records .....	409
4.4.6	Console: Security .....	410

<b>4.5 Database - MySQL .....</b>	<b>413</b>
4.5.1 Create Schema/Database.....	414
<b>4.6 Mapper - Map Struct .....</b>	<b>416</b>
4.6.1 DTO to Entity - Map Struct .....	417
<b>4.7 Mapper - JMapper .....</b>	<b>423</b>
4.7.1 DTO to Entity - JMapper - Annotations.....	424
<b>4.8 Mapper - Model Mapper .....</b>	<b>429</b>
4.8.1 Example .....	430
4.8.2 Matching Strategy.....	432
4.8.3 DTO to Entity - Model Mapper - Default.....	433
4.8.4 DTO to Entity - Model Mapper - Custom .....	438
4.8.5 DTO to Entity - Model Mapper - Exclude .....	443
<b>4.9 Mapper - Model Mapper vs JMapper vs Map Struct .....</b>	<b>448</b>
<b>5 ERRORS.....</b>	<b>449</b>
<b>5.1 Controller .....</b>	<b>450</b>
5.1.1 Circular view path .....	451
<b>5.2 Database .....</b>	<b>452</b>
5.2.1 Cannot Resolve Table.....	453
5.2.2 Records are not inserted into DB .....	455
5.2.3 Database Tool Window can't see H2 Tables .....	456

# ABOUT THIS BOOK

## This is first Book in the series

[Spring Boot - Quick Start](#)

[Spring Boot - Accessories](#)

[Spring Boot - Security](#)

## Content

Intention of this Book is to quickly get you started using Spring Boot to develop Web Applications. Book covers basic functionalities like: Controllers, Endpoints, Entities, DTOs, Services, Repositories, etc.

## Standalone Tutorials

The core of this book are standalone tutorials that explain different functionalities of Spring Boot.

Each tutorial contains minimum amount of code needed to explain specific functionality. Tutorials have minimum amount of encompassing text that explains related theory and different parts of the code. This approach allows students to grasp presented concepts in a very fast and efficient manner. Full code, which can also be downloaded from GitHub, prevents any time being wasted trying to make the code work. Simple examples allow for full understanding of the functionality without any unnecessary distractions.

## Theoretical Background

Where needed tutorials are preceded by chapters focusing on theoretical background. This way reader can fully understand functionalities explained in the subsequent chapters. But such chapters are in minority and of secondary importance because the main focus is on practical applications.

## Demo Applications

Book contains demo Applications that show how to combine functionalities covered in previous tutorials. More complex Application "Authors & Books" is broken into multiple steps showing how to add additional functionalities at each step.

## **WHY TUTORIALS?**

**"Things are only as complicated as they are badly explained."**

Proper documentation is essential to avoid struggle and frustration when working with simple things that only seem complicated by not being properly documented and explained.

## **WHAT KIND OF TUTORIALS?**

**"Working example is worth thousand words."**

Just like the picture is worth thousand words the same goes for the working example. Documentation in the form of working examples is proved to be the fastest and the most effective way of transferring knowledge. Sometimes an example is all you need to get the things done. And if there are some accompanying comments that explain what is going on even better. This approach is used in this book. This results in fast learning and the ability to apply tutorials when you need them in the spirit of Just In Time Support.

**I wish you rapid learning!**



# 1 Quick Start

## Info

---

- Following tutorials show how to prepare development environment that will be used to create Spring Boot Application.
  - [Install Java](#) (Programming Language. Java can be installed as part of IntelliJ IDEA installation)
  - [Install IntelliJ IDEA](#) (free Community or paid Ultimate Edition with free 30 day trial)
- Then we show how to [Create Spring Boot Project](#) that will be used as a starting point for creating our applications
  - [Using IntelliJ Community Edition \(free\)](#)
  - [Using IntelliJ Ultimate Edition \(paid\)](#)

# 1.1 Install

## Info

---

- Following tutorials show how to install applications which will be used to create Spring Boot Application
  - [Java](#) (Programming Language. Java can be installed as part of IntelliJ IDEA installation)
  - [Install IntelliJ IDEA](#) (IDE - Integrated Development Environment)

# 1.1.1 Java

## Info

- This tutorial shows how to install JAVA by using Windows installer program.
- Environment variables will be automatically set.
- You can **skip** this step since Java can also be installed as part of [IntelliJ IDEA](#) installation

## Download & Execute

- <http://java.sun.com/javase/downloads/index.jsp>
- Oracle JDK
- **jdk-15.0.1\_windows-x64\_bin.exe**
- I reviewed ...: CHECK
- Download jdk-15.0.1\_windows-x64\_bin.exe
- Save as: D:\Downloads\jdk-15.0.1\_windows-x64\_bin.exe
- Execute jdk-15.0.1\_windows-x64\_bin.exe

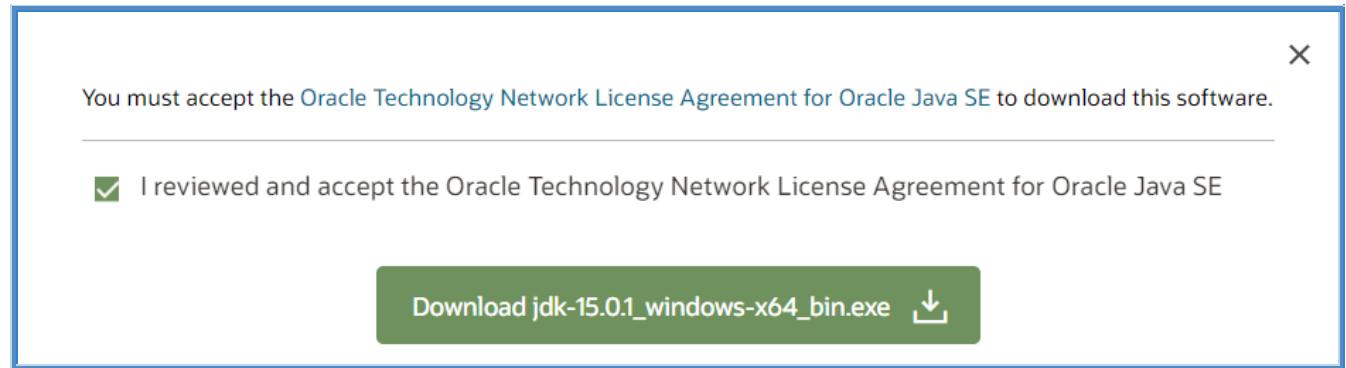
<http://java.sun.com/javase/downloads/index.jsp> - Oracle JDK

The screenshot shows a web browser window with the URL [oracle.com/java/technologies/javase-downloads.html](http://oracle.com/java/technologies/javase-downloads.html). The page title is "Java SE 15". It features a navigation bar with links for Products, Resources, Support, and Events. Below the navigation bar, there is a section titled "Java SE 15" with the subtext "Java SE 15.0.1 is the latest release for the Java SE Platform". To the left, there is a list of links: Documentation, Installation Instructions, Release Notes, and Oracle License. To the right, there is a section titled "Oracle JDK" with two download options: "JDK Download" and "Documentation Download".

*jdk-15.0.1\_windows-x64\_bin.exe*

The screenshot shows a web browser window with the URL [oracle.com/java/technologies/javase-jdk15-downloads.html](http://oracle.com/java/technologies/javase-jdk15-downloads.html). The page title is "Java SE Development Kit 15 Downloads". It features a navigation bar with links for Products, Resources, Support, and Events, and a "View Accounts" button. Below the navigation bar, there is a breadcrumb trail "Java / Technologies / Java SE 15 - Downloads" and two buttons: "Java SE Downloads" and "Java SE Subscriptions". The main content area is titled "Java SE Development Kit 15 Downloads" and lists two download options: "macOS Compressed Archive" (176.53 MB) and "Windows x64 Installer" (159.69 MB). Each option has a download link icon next to it.

*Download jdk-15.0.1\_windows-x64\_bin.exe*



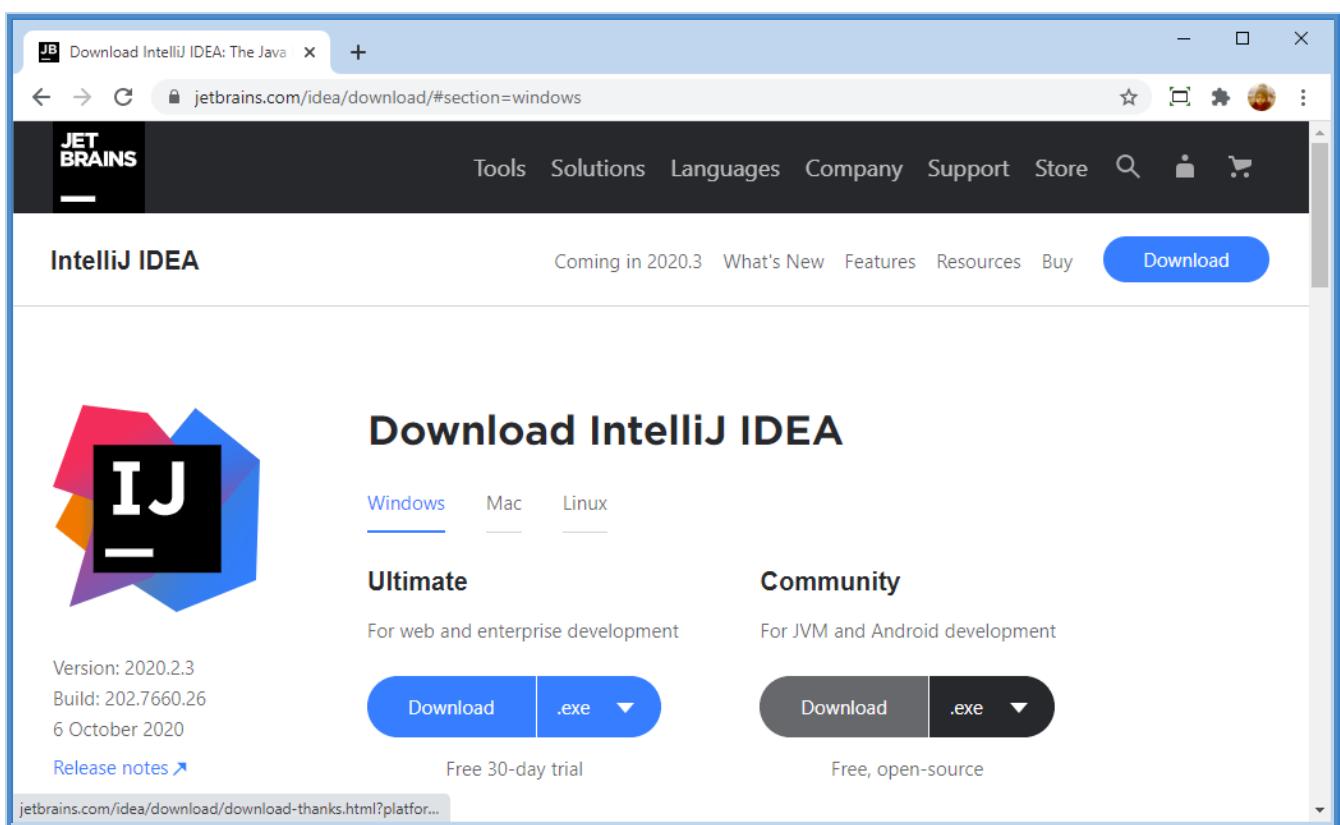
# 1.1.2 IntelliJ IDEA

## Info

- This tutorial shows how to install IntelliJ IDEA Integrated Development Environment (IDE)
  - [Download IntelliJ IDEA](#) (free Community or paid Ultimate Edition with free 30 day trial)
  - [Install IntelliJ IDEA](#)
  - [Download Java JDK](#) (if JDK is not already installed on the PC)
  - [Customize](#) (select dark or light interface)

## Download IntelliJ IDEA

- <https://www.jetbrains.com/idea>
- Download
- Community
- Download
- Save as: D:\Downloads\idealC-2020.2.3.exe



## Install IntelliJ IDEA

- Execute idealC-2020.2.3.exe
- Run
- 3\*Next

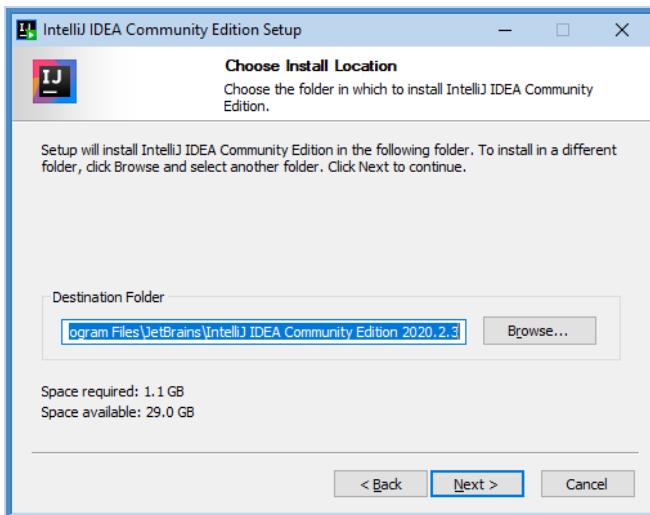
Run



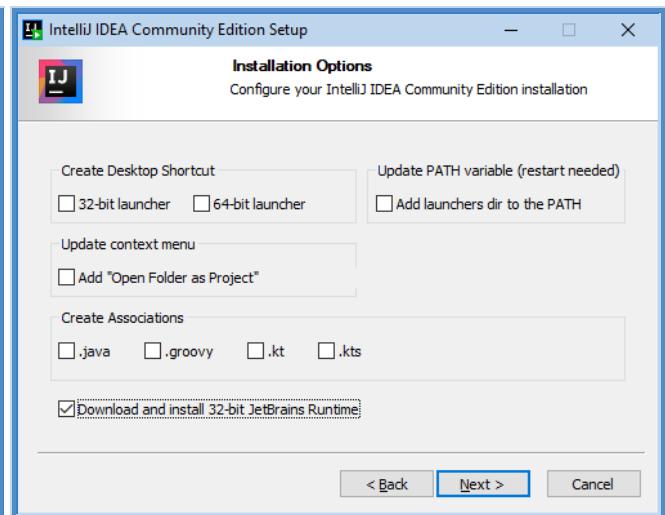
Next



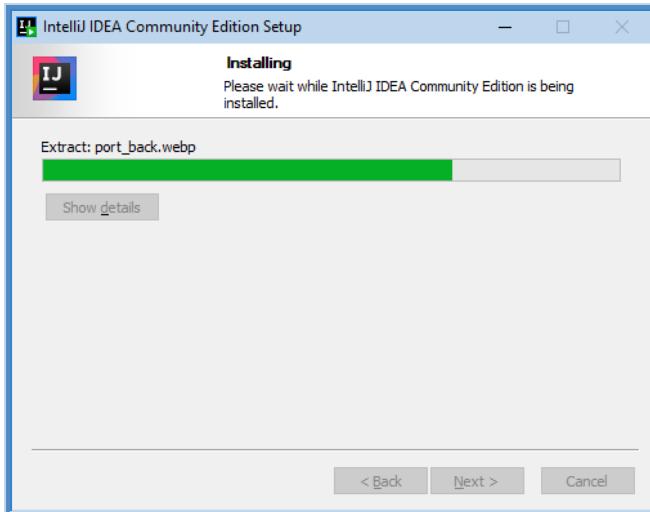
Next



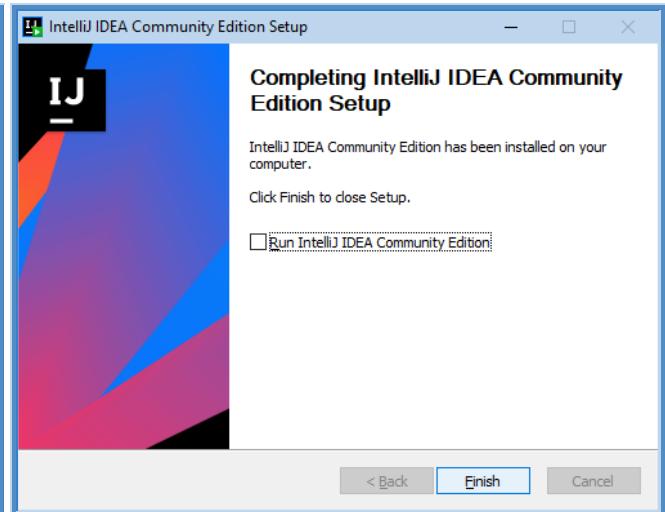
Next



Progress



Finish



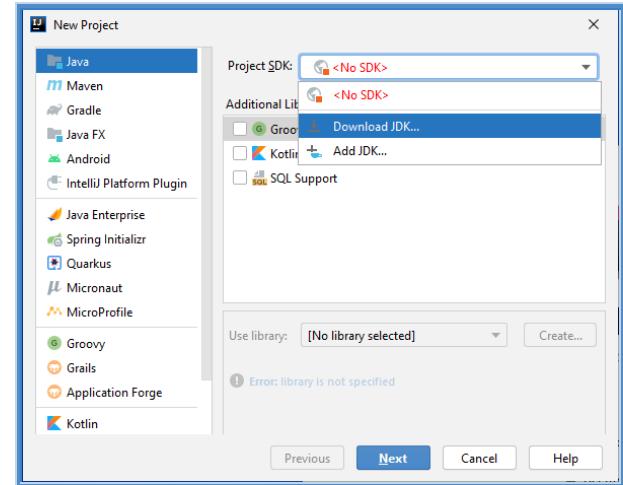
## Download Java JDK

- Start IntelliJ IDEA
- New Project
- Java
- Download JDK

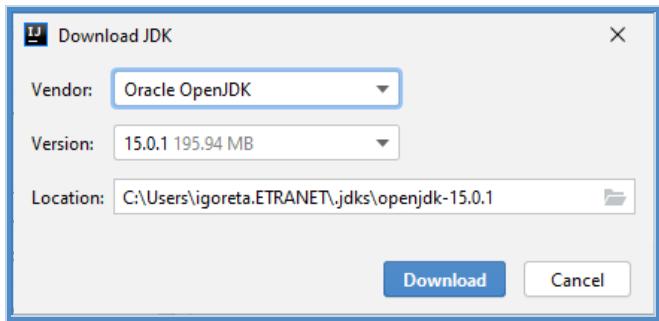
### New Project



### Download JDK



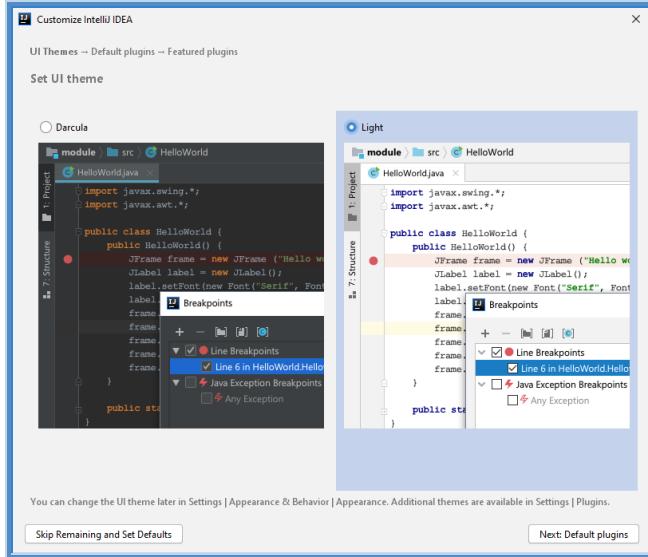
### Download



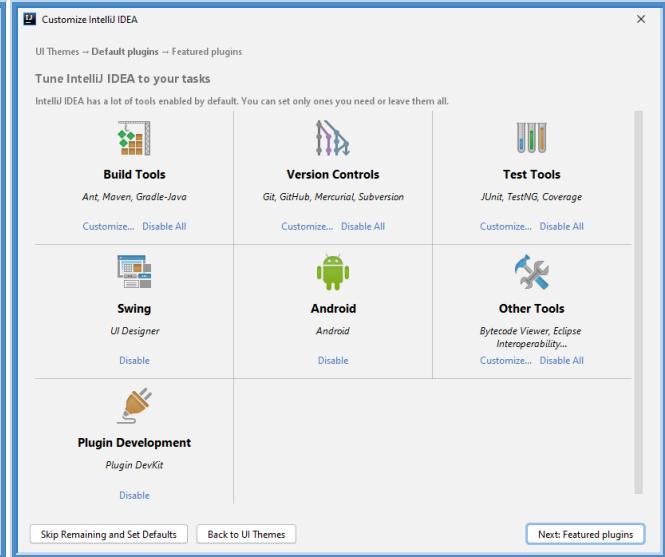
## Customize

- Start IntelliJ IDEA
- Light
- 2\*Next
- Start

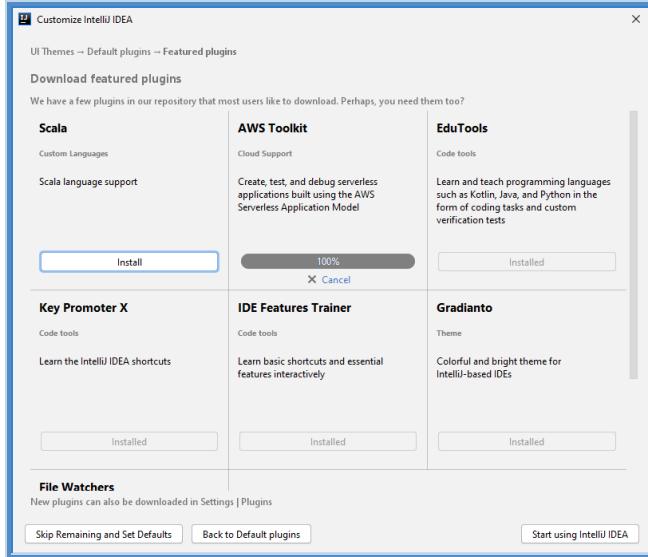
### Light



### Next



### Start

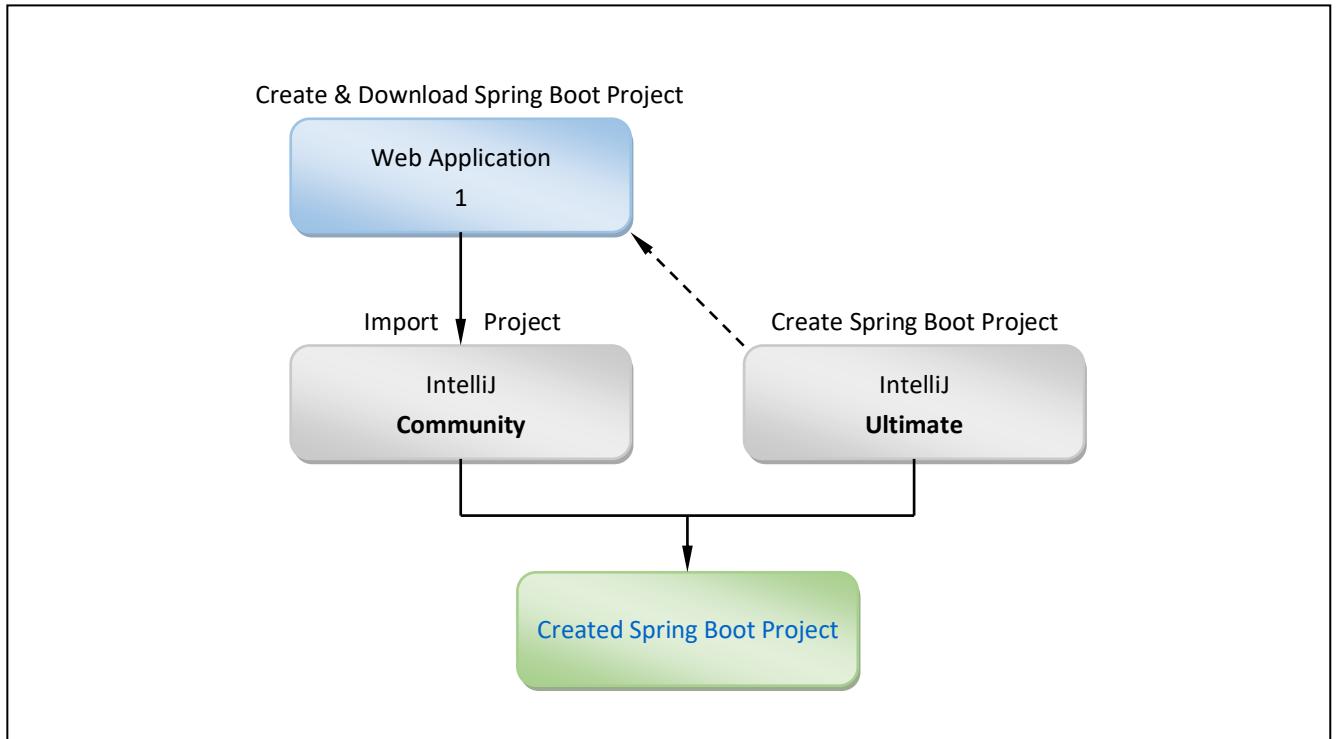


# 1.2 Create Project with IntelliJ

## Info

- Following tutorial show how to create Spring Boot Project using IntelliJ IDEA Integrated Development Environment (IDE).
- Depending on the edition of IntelliJ IDEA, Spring Boot Project can be created in two ways
  - Using IntelliJ Community Edition** (use <https://start.spring.io> Web Application to create & download project)
  - Using IntelliJ Ultimate Edition** (IntelliJ uses <https://start.spring.io> Web Application in the background)
- In both cases we end up with the same [Spring Boot Project](#).

### Create Spring Boot Project



# 1.2.1 Community Edition (free)

## Info

- This tutorial shows how to Create [Spring Boot Project](#) using IntelliJ free **Community** edition
  - [Create Spring Boot Project](#) (use <https://start.spring.io/> Web App that implements [Spring Initializer](#))
  - [Import Spring Boot Project into IntelliJ](#) (Maven loads dependency JARs)
- Community edition doesn't support functionality to create Spring Project directly from within IntelliJ IDE.  
Therefore we use <https://start.spring.io/> to select [Spring Boot Starters](#) (Maven dependencies).  
Downloaded Spring Project will not contain any Java Classes. Maven downloads them after importing Project to IntelliJ.

## Create Spring Boot Project

- <https://start.spring.io/>
- Project: Maven Project
- Language: Java
- Spring Boot: 2.4.1
- Project Metadata
  - Group: com.ivoronline.springboot
  - Artifact: **demo** (Project Name)
  - Packaging: Jar
  - Java: 11
- Dependencies
  - Spring Web
  - Spring Data JPA
  - H2 Database
- Generate
- Save as: C:\Downloads\demo.zip
- Extract demo.zip

## Generate Spring Project

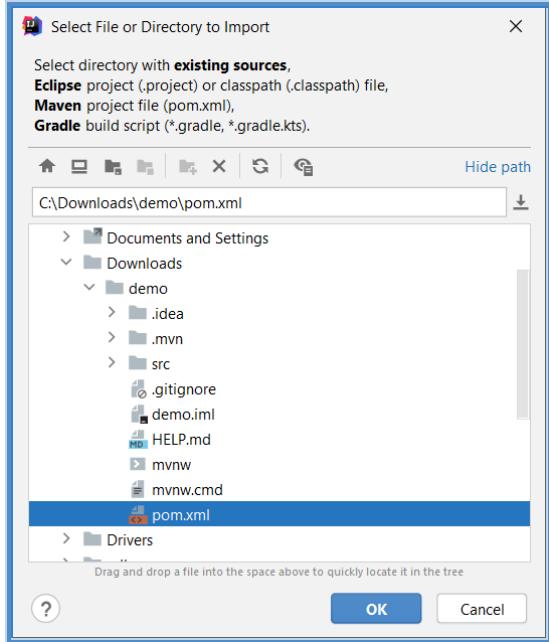
The screenshot shows the Spring Initializr interface. The 'Project' section is set to 'Maven Project'. The 'Language' section has 'Java' selected. Under 'Spring Boot', version '2.4.1' is chosen. In the 'Project Metadata' section, the 'Group' field is set to 'com.ivoronline.springboot' and the 'Artifact' field is set to 'demo', both of which are highlighted with a red rectangle. The 'Name' field is 'demo', 'Description' is 'Demo project for Spring Boot', 'Package name' is 'com.ivoronline.springboot.demo', 'Packaging' is 'Jar', and 'Java' is '11'. The 'Dependencies' section includes 'Spring Web' (selected) and 'H2 Database'. At the bottom, there are 'GENERATE' and 'SHARE...' buttons.

## Import Spring Boot Project into IntelliJ

---

- Start IntelliJ
- File - New - **Project from existing sources**
- C:\Downloads\demo\pom.xml
- (wait for it to resolve Maven dependencies - download JARs with Java Classes)

*Project from existing sources - pom.xml*



# 1.2.2 Ultimate Edition (paid)

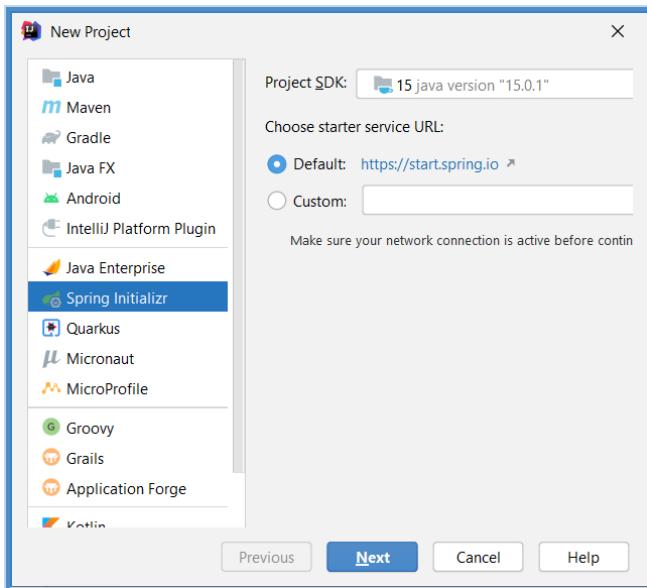
## Info

- This tutorial shows how to Create [Spring Boot Project](#) using IntelliJ paid **Ultimate** Edition (free 30 day trial).
- In the background IntelliJ uses <https://start.spring.io/> Web Application.
- That Web Application is also used in [Create Spring Project - Using start.spring.io.](#)

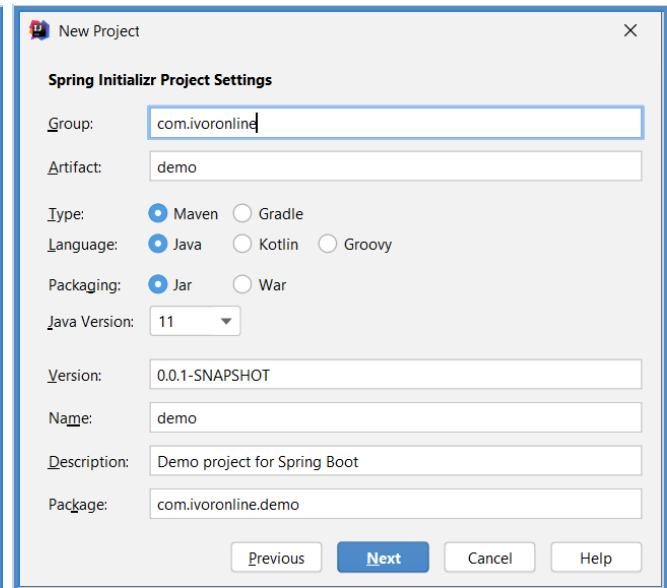
## Create Spring Boot Project

- Start IntelliJ
- File - New - Project
- Spring Initializr - Next
- Project Settings
  - Group: com.ivoronline.springboot
  - Artifact: **demo** (Project Name)
  - Type: Maven
  - Language: Java
  - Packaging: Jar
  - Java Version: 11
  - Next
- Dependencies
  - Web: Spring Web
  - SQL: Spring Data JPA
  - SQL: H2 Database
  - Next
- Project name: demo
- Project location: C:\Projects\demo
- Finish
- (wait for it to resolve Maven dependencies - download JARs with Java Classes)

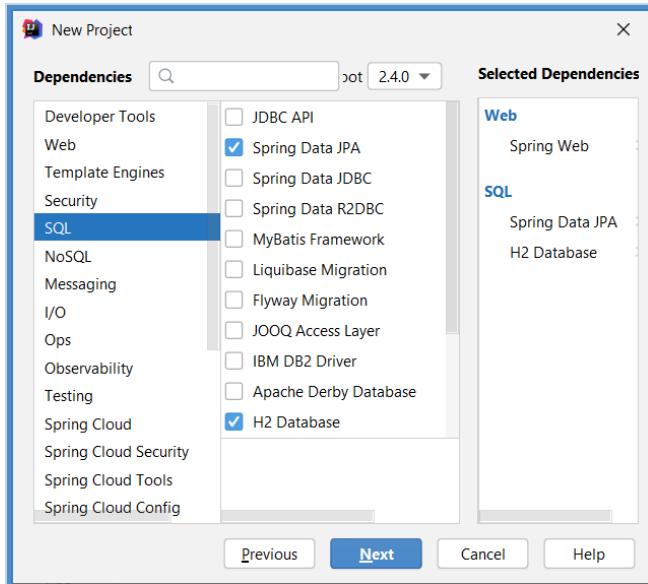
File - New - Project - Spring Initializr



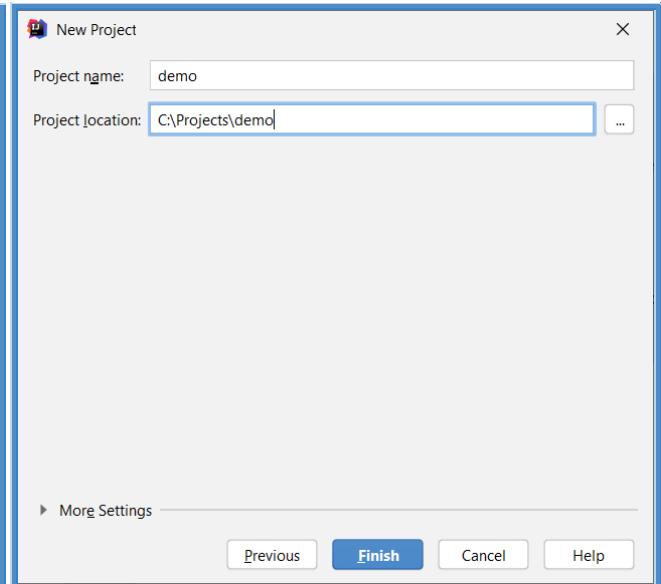
Project Settings



## Dependencies



## Project name



# 1.3 Application

## Info

---

- Following tutorials show how to Run and Deploy Application.

# 1.3.1 Run

## Info

- This tutorial shows how to run Spring Boot Application.
- We will and display a message in the Console just to make sure everything is working properly.

## Run Application

- Create Spring Boot Project (no additional dependencies needed)
- Edit Java Class: TestSpringBootApplication.java (insert highlighted line)
- Run Application

ConsoleApplication.java

```
package com.ivoronline.console_application;

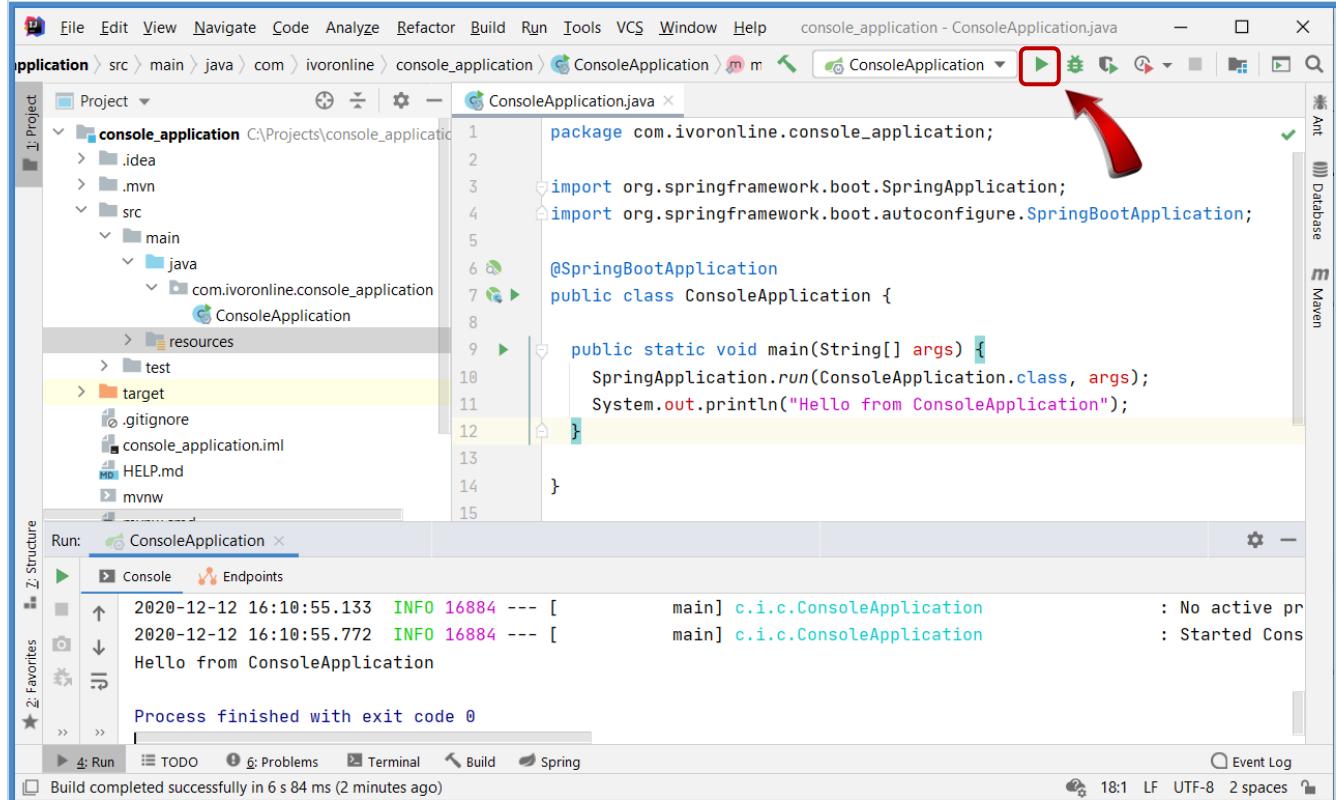
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class ConsoleApplication {

    public static void main(String[] args) {
        SpringApplication.run(ConsoleApplication.class, args);
        System.out.println("Hello from ConsoleApplication");
    }

}
```

Application



Console

```
Hello from ConsoleApplication
```

## 1.3.2 Deploy

### Info

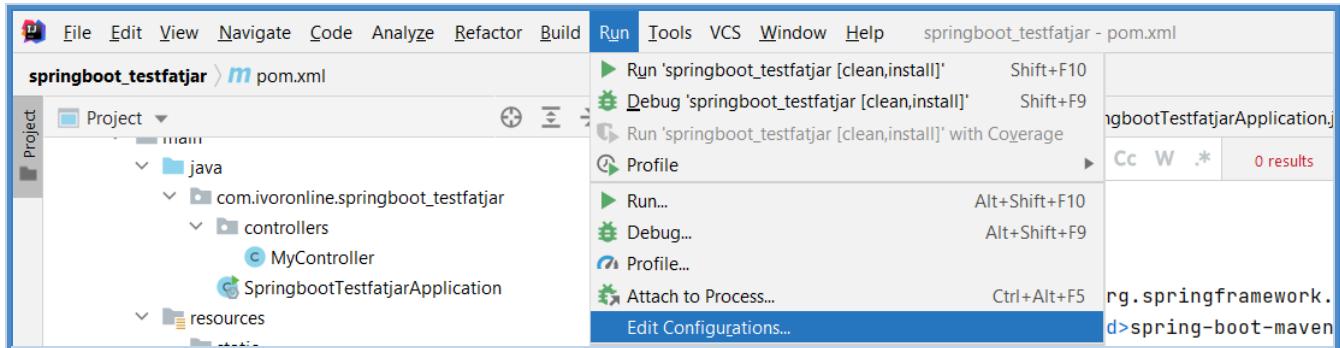
[R]

- This tutorial shows how to deploy Spring Boot Application by creating **Fat Jar**.
- Fat Jar will include everything needed for the application to run (including Tomcat Server).
- You can then simply copy **Fat Jar** (using FTP) to target Server (with Java installed) and start it with Java.
- To demonstrate how this works first create simple Spring Boot Application as shown in [Return - Text](#) and then
  - [Maven Run Configuration - Create](#)    `clean install`
  - [Maven Run Configuration - Run](#)    creates `springboot_testfatjar-0.0.1-SNAPSHOT.jar`
  - [Run Fat Jar](#)    `java -jar myfatapp.jar`
  - [Results](#)    `http://localhost:8085/hello`

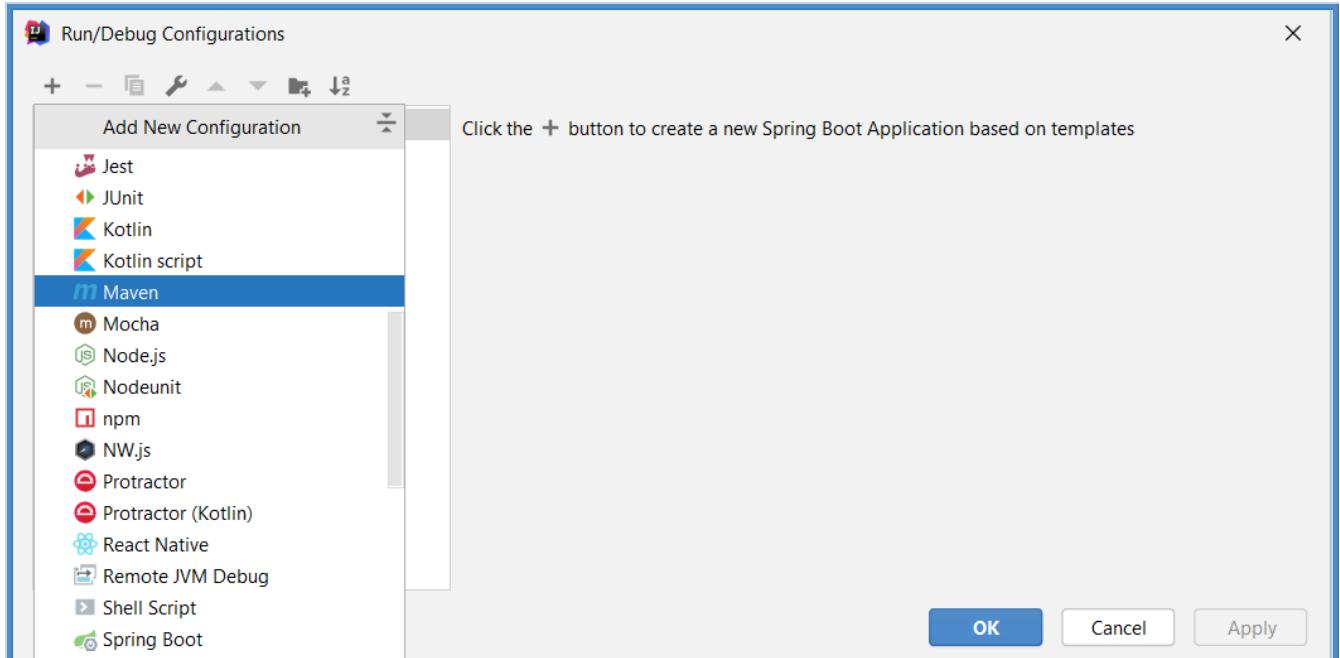
### Maven Run Configuration - Create

- Run
- Edit Configurations
- Add New Configuration
- Maven
- Working Directory: C:/Projects/springboot\_testfatjar
- Command line: clean install
- OK

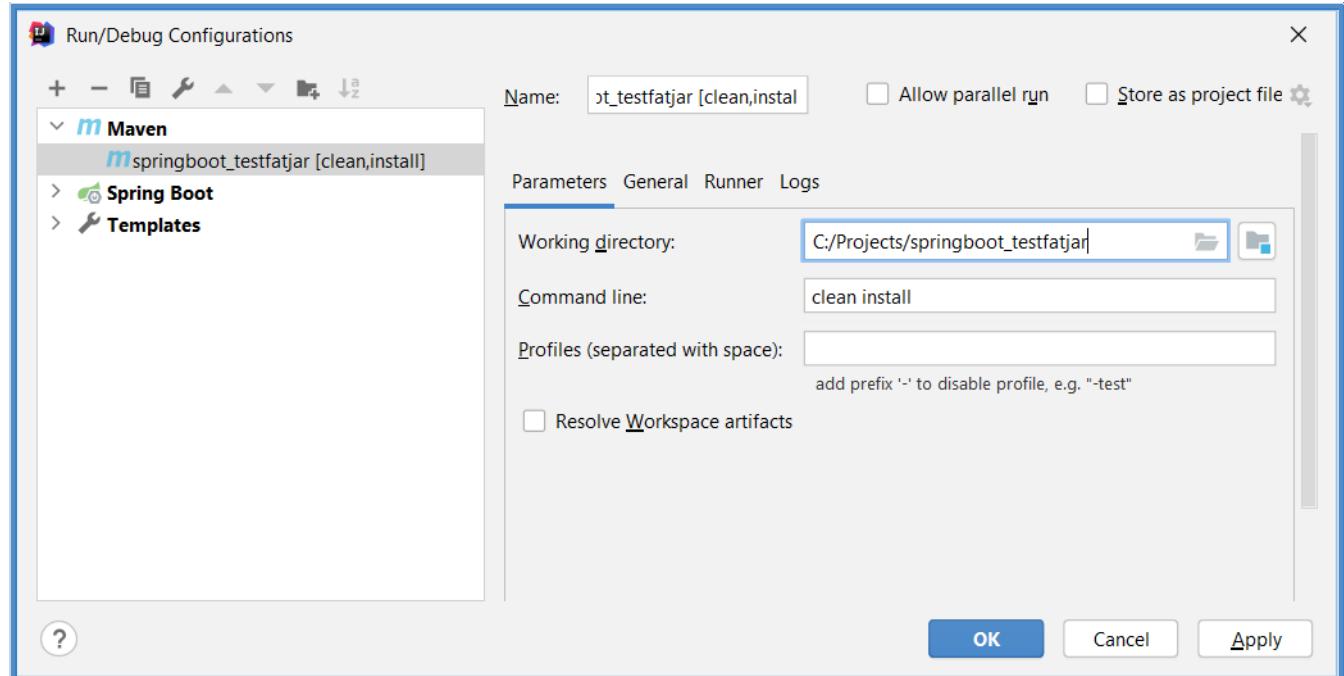
#### Run - Edit Configurations



#### Add New Configuration - Maven

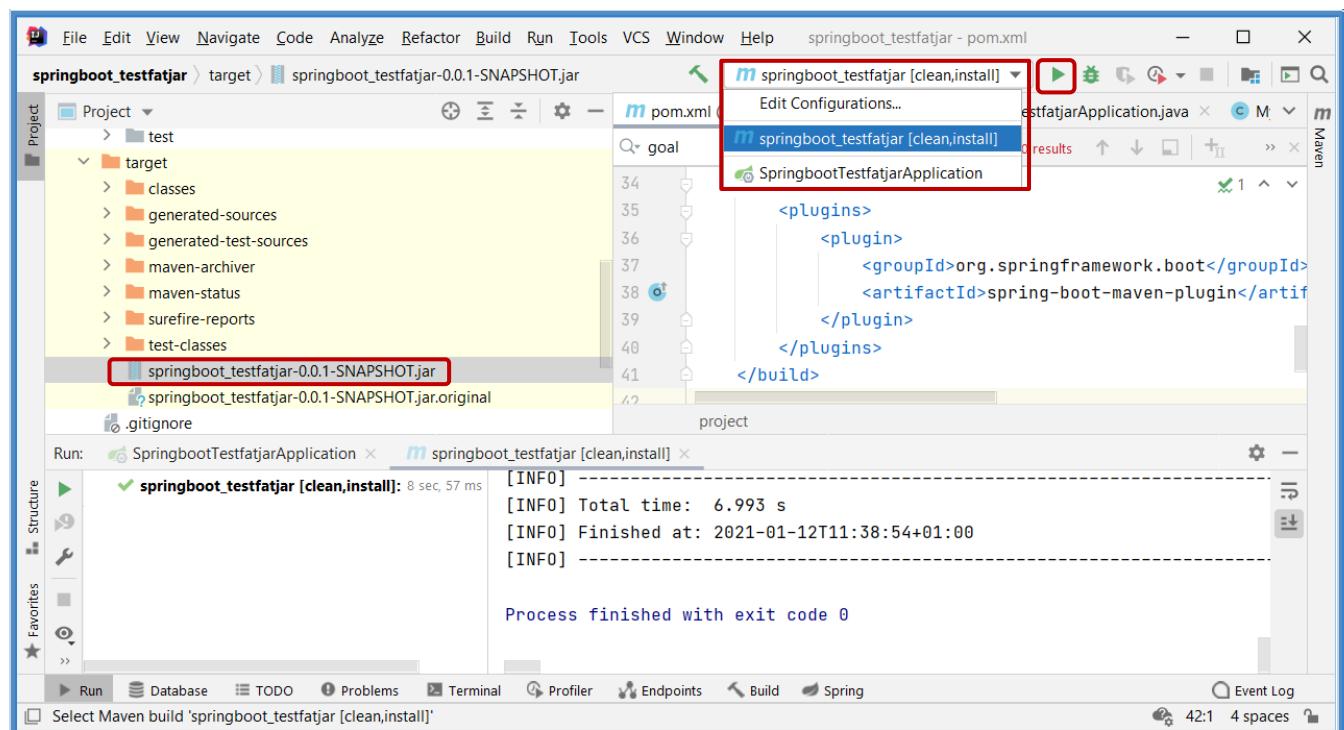


## Command line: clean install



## Maven Run Configuration - Run

- Select Maven Run Configuration: springboot\_testfatjar [clean,install]
- Run (creates springboot\_testfatjar-0.0.1-SNAPSHOT.jar)



## Run Fat Jar

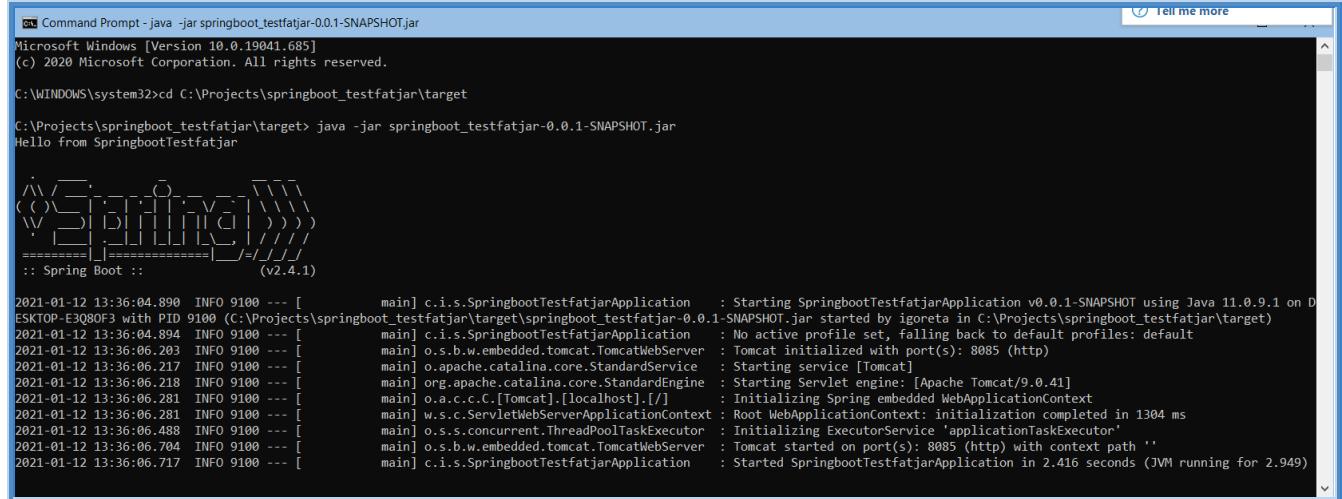
- Start Command Prompt
- (execute below commands)

### Command Prompt

```
cd C:\Projects\springboot_testfatjar\target  
java -jar springboot_testfatjar-0.0.1-SNAPSHOT.jar
```

## Results

### Command Prompt



```
cmd Command Prompt - java -jar springboot_testfatjar-0.0.1-SNAPSHOT.jar
Microsoft Windows [Version 10.0.19041.685]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\WINDOWS\system32>cd C:\Projects\springboot_testfatjar\target

C:\Projects\springboot_testfatjar\target> java -jar springboot_testfatjar-0.0.1-SNAPSHOT.jar
Hello from SpringbootTestfatjar

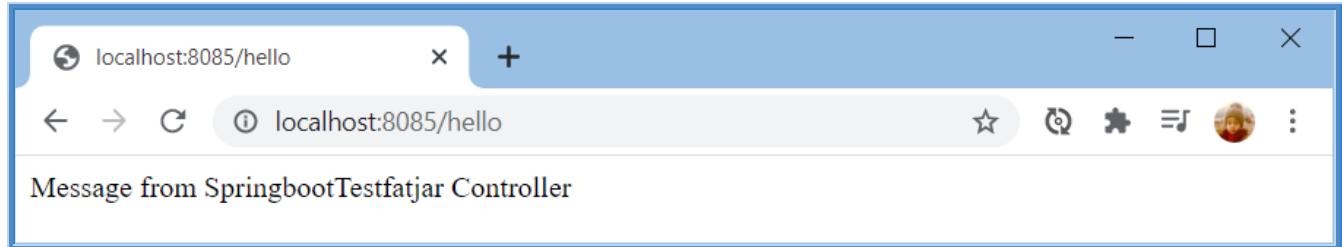
:: Spring Boot ::   (v2.4.1)

2021-01-12 13:36:04.890  INFO 9100 --- [           main] c.i.s.SpringbootTestfatjarApplication : Starting SpringbootTestfatjarApplication v0.0.1-SNAPSHOT using Java 11.0.9.1 on DESKTOP-E3Q80F3 with PID 9100 (C:\Projects\springboot_testfatjar\target\springboot_testfatjar-0.0.1-SNAPSHOT.jar started by igoreta in C:\Projects\springboot_testfatjar\target)
2021-01-12 13:36:04.894  INFO 9100 --- [           main] c.i.s.SpringbootTestfatjarApplication : No active profile set, falling back to default profiles: default
2021-01-12 13:36:06.203  INFO 9100 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8085 (http)
2021-01-12 13:36:06.217  INFO 9100 --- [           main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2021-01-12 13:36:06.218  INFO 9100 --- [           main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.41]
2021-01-12 13:36:06.281  INFO 9100 --- [           main] o.a.c.c.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2021-01-12 13:36:06.281  INFO 9100 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 1304 ms
2021-01-12 13:36:06.488  INFO 9100 --- [           main] o.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
2021-01-12 13:36:06.704  INFO 9100 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8085 (http) with context path ''
2021-01-12 13:36:06.717  INFO 9100 --- [           main] c.i.s.SpringbootTestfatjarApplication : Started SpringbootTestfatjarApplication in 2.416 seconds (JVM running for 2.949)
```

### Command Prompt Output

```
Hello from SpringbootTestfatjar
Tomcat initialized with port(s): 8085 (http)
```

<http://localhost:8085/hello>



# 2 Main Terms

## Info

---

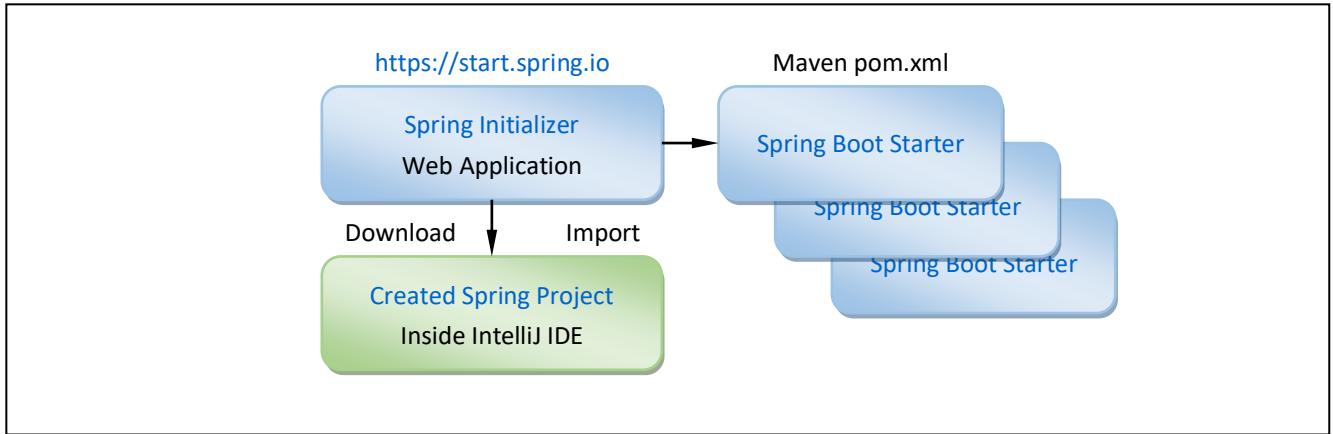
- Following tutorials explain main terms related to Spring Boot Development.
- They will be explained as stand-alone applications where each one is used to demonstrate specific functionality.
- Applications are designed to be as simple as possible to demonstrate functionality in question.
- That means that they might not comply to best coding practices because simplicity takes higher priority.
- How to combine all these functionalities in a proper way will be demonstrated in later chapter with [Demo Applications](#).

## 2.1 Project

### Info

- Following tutorials explain main terms related to creation of Spring Boot Project.

#### Spring Boot Starters



## 2.1.1 Spring Boot Starters

### Info

[R]

- ➊ Spring Boot Starter
  - is [pom.xml](#) or [build.gradle](#) file that contains Maven or Gradle dependencies needed for certain functionality
  - is Maven or Gradle dependency descriptor that can be included in your [pom.xml](#) or [build.gradle](#) file
- ➋ [pom.xml](#) or [build.gradle](#) file specifies which JAR files are needed for specific functionality.  
They make sure that compatible versions of JAR files are downloaded together.  
JAR files contain Java classes that implement functionality that you want to include inside your Project.
- ➌ You can
  - simply copy these Maven or Gradle dependencies inside your [pom.xml](#) or [build.gradle](#) file
  - or use [Spring Initializer](#) to automatically create [pom.xml](#) or [build.gradle](#) file from selected [Spring Boot Starters](#)

#### *Spring Boot Starter: Data JPA*

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

#### *Spring Boot Starter: Web*

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

#### *Spring Boot Starter: H2*

```
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
</dependency>
```

## 2.1.2 Spring Initializer

### Info

[R]

- Spring Initializer is [Web Application](#) that creates Spring Boot Project from the selected [Spring Boot Starters](#) by
  - populating `pom.xml` or `build.gradle` file with Maven or Gradle dependencies for selected [Spring Boot Starters](#)
  - performing some additional setup with configuration files
  - creating directory structure for your Spring Project
- You can then download such empty Spring Project as ZIP File and use it to create Spring Project inside your IDE. Spring Project will not contain any JARs or Java Classes. They will be downloaded once we import Project into IntelliJ.
- Spring Initializer implementation code is located at <https://github.com/spring-io/initializr>. <https://start.spring.io> is example of Web Application that implements Spring Initializer.
- We are using Spring Initializer to [Create Spring Boot Project](#) in following tutorials
  - [Using IntelliJ Community](#)
  - [Using IntelliJ Ultimate](#)

This will Results in [Created Spring Boot Project](#).

<https://start.spring.io> implements Spring Initializer

The screenshot shows the start.spring.io web application interface. At the top, there's a navigation bar with a logo, a search bar containing 'start.spring.io', and a user icon. On the left, there's a sidebar with a menu icon and social sharing icons for GitHub and Twitter. The main content area has a header 'spring initializr'. It contains several configuration sections:

- Project**:
  - Maven Project
  - Gradle Project
- Language**:
  - Java
  - Kotlin
  - Groovy
- Spring Boot**:
  - 2.4.1(SNAPSHOT)
  - 2.4.0
  - 2.3.7(SNAPSHOT)
  - 2.3.6
  - 2.2.12(SNAPSHOT)
  - 2.2.11
- Project Metadata**:

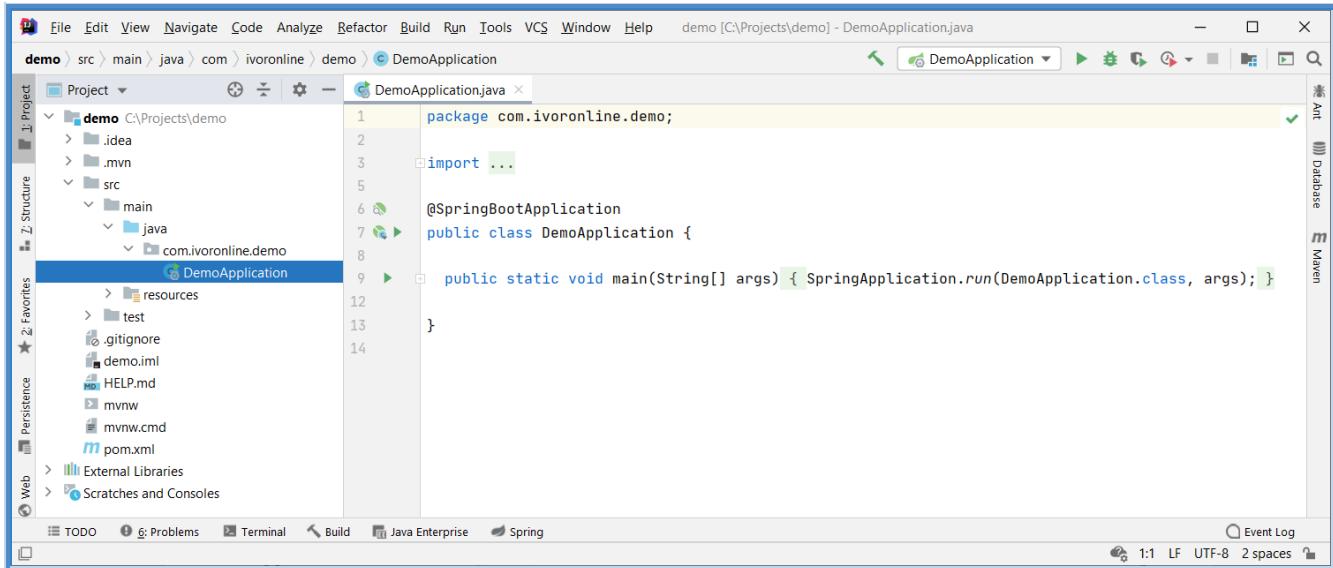
Group	com.ivoronline
Artifact	demo
Name	demo
Description	Demo project for Spring Boot
- Dependencies**:
  - Spring Web** [WEB]: Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.
  - Spring Data JPA** [SQL]: Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.
  - H2 Database** [SQL]: Provides a fast in-memory database that supports JDBC API and R2DBC access, with a small (2mb) footprint. Supports embedded and server modes as well as a browser based console application.
- Buttons**:
  - GENERATE CTRL + ↵
  - EXPLORE CTRL + SPACE
  - SHARE...

## 2.1.3 Spring Boot Project

### Info

- Created Spring Boot Project will contain
  - [Directory Structure](#)
  - [DemoApplication.java](#)
    - (entry point to our application - Java Class with main method)
  - [pom.xml](#)
    - (with selected dependencies: Spring Web, Spring Data JPA, H2 Database)
- Depending on the edition of IntelliJ IDEA you can [Create Spring Boot Project](#) in two ways
  - [Using IntelliJ Community Edition](#)
    - (use <https://start.spring.io> Web Application to create & download project)
  - [Using IntelliJ Ultimate Edition](#)
    - (IntelliJ uses <https://start.spring.io> Web Application in the background)

### Directory Structure



### DemoApplication.java

```
package com.ivoronline.demo;

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

@SpringBootApplication
public class DemoApplication {

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

}
```

```
<?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 https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.2.11.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>

  <groupId>com.ivoronline</groupId>
  <artifactId>demo</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>demo</name>
  <description>Demo project for Spring Boot</description>

  <properties>
    <java.version>11</java.version>
  </properties>

  <dependencies>

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

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

    <dependency>
      <groupId>com.h2database</groupId>
      <artifactId>h2</artifactId>
      <scope>runtime</scope>
    </dependency>

  </dependencies>

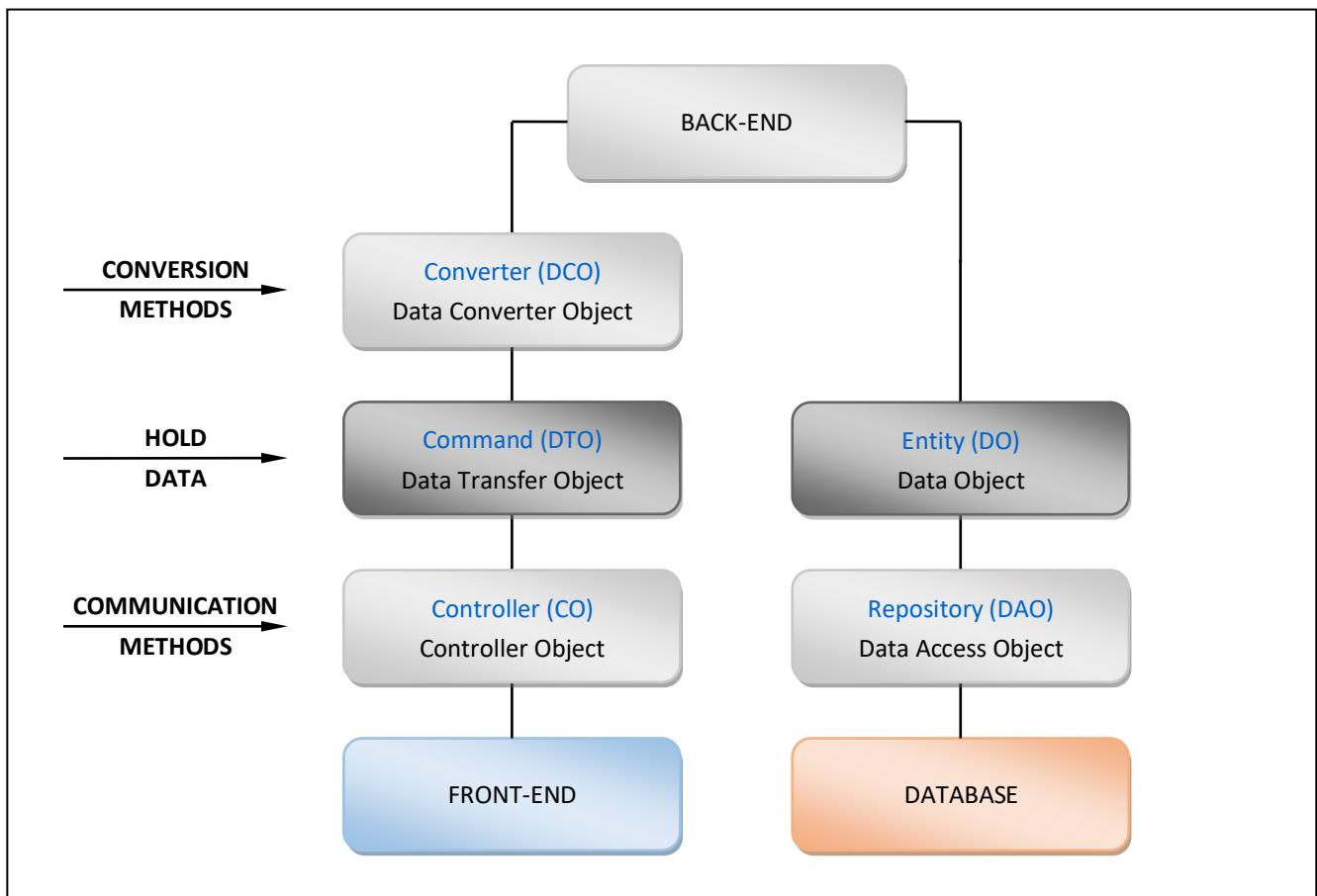
</project>
```

## 2.2 Object Types

### Info

- Following tutorials explain main terms related to data flow like Objects that are used to
  - **store data:** Entity (DO), Command (DTO) (thick border)
  - **transfer data:** Repository (DAO), Controller (CO) (thin border)
  - **convert data:** Converter (DCO) (thin border)
- Below schema shows three main parts of the Web Application that work together
  - **front-end** application might be Web Browser or Mobile Application (colored in blue)
  - **back-end** application is what we are building with Spring Boot (colored in gray)
  - **database** application is used to store data (colored in red)
- Back-end application
  - works with Entity (DO) Objects (each represents a record in the DB)
  - doesn't work with DB Tables
  - doesn't know where data comes from: DB, File, URL, Array, ...
- When back-end needs
  - to get data from DB it calls method from Repository (DAO) which returns Entity (DO) Object with retrieved data
  - to store data into DB it calls method from Repository (DAO) and provides Entity (DO) Object with data to be stored
  - to send data to Frontend it stores data in Command (DTO) Object and sends thatIn the process it uses Controller (CO) to convert Entity (DO) Object into Command (DTO) Object

Relations between Objects



## Terms

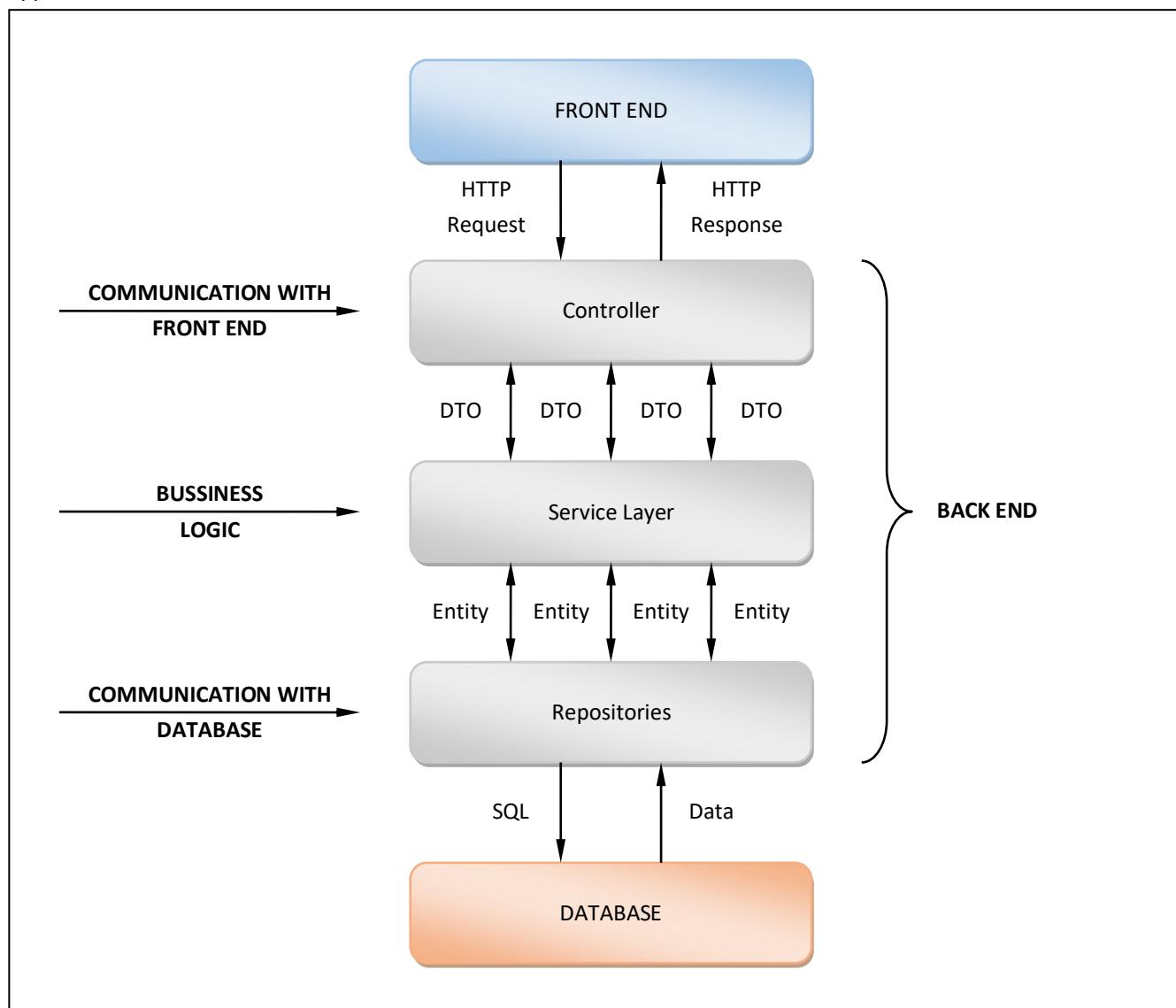
NAME	DESCRIPTION
Entity (DO) Data Object	<ul style="list-style-type: none"> <li>➊ <b>Entity</b> Object represents record in DB Table and contains           <ul style="list-style-type: none"> <li>● <b>Properties</b> (that are equivalent to DB Table Columns)</li> <li>● Helper Methods (setters, getters, equals, toString)</li> </ul> </li> </ul>
Command (DTO) Data Transfer Object	<ul style="list-style-type: none"> <li>➋ <b>Command</b> Object is used to transfer data between application layers and contains           <ul style="list-style-type: none"> <li>● <b>Properties</b> (subset of Entity properties)</li> <li>● Helper Methods (setters, getters, equals, toString)</li> </ul> </li> </ul>
Repository (DAO) Data Access Object	<ul style="list-style-type: none"> <li>➌ <b>Repository</b> Object is used to store/retrieve Entities into/from DB and contains           <ul style="list-style-type: none"> <li>● Entity Property (that holds data)</li> <li>● <b>Methods</b> for accessing DB (for storing and retrieving data)</li> </ul> </li> </ul>
Controller (CO) Controller Object	<ul style="list-style-type: none"> <li>➍ <b>Controller</b> Object contains           <ul style="list-style-type: none"> <li>● <b>Methods</b> that accept HTTP Requests (end points)</li> </ul> </li> </ul>
Converter (DCO) Data Converter Object	<ul style="list-style-type: none"> <li>➎ <b>Converter</b> Object contains           <ul style="list-style-type: none"> <li>● <b>Methods</b> for converting between <b>Command (DTO)</b> and <b>Entity (DO)</b> Objects</li> </ul> </li> </ul>

## Application Structure

---

- ➏ Main part of any Application is **Service Layer** which consists of **Service Objects** which implement Business Logic.
- ➐ Every other part of the application is just plumbing allowing Service Objects to perform their tasks
  - On one side we have **Presentation Layer** which allows Service Layer to **communicate with outside world**
  - On the other side we have **Persistence Layer** which allows Service Layer to **memorize things** (persist them in DB)
- ➑ Service Layer uses
  - **Data Transfer Objects (DTO)** to encapsulate data that is exchanged with **Presentation Layer**
  - **Entities** to encapsulate data that is exchanged with **Persistence Layer**
- ➒ Since the above Data Objects just hold pure data (without any methods/logic) Service Layer uses
  - **Controller** to communicate with **Presentation Layer** (can be only one)
  - **Repositories** to communicate with **Persistence Layer** (one for every Entity)
- ➓ **Controller**
  - receives **HTTP Request**
  - creates DTO from the data in HTTP Request
  - forwards DTO to specific Service Object (that is mapped to that type of HTTP Request)
  - receives DTO returned by the Service Object
  - forwards returned DTO back to the front-end (as the final response to the HTTP Request)
- ➔ **Service Object**
  - receives DTO from the Controller
  - creates Entities from DTO
  - performs **CRUD Requests** toward Persistence Layer (Create Read Update Delete Entities)
  - performs its business logic using Entities (those from DTO and those from Persistence Layer)
  - creates DTO from Entities
  - returns DTO back to the Controller

## *Application Schema*



## 2.2.1 Entity Object (DO)

### Info

[R] [R]

- Entity Object, DO, PO & POJO are synonyms.
  - DO stands for **Data Object** or **Domain Object**
  - PO stands for **Persistence Object**
  - POJO stands for **Plain Old/Ordinary Java Object**
- **Entity**
  - Class represents DB Table
  - Object/Instance represents record in DB Table
  - allows you to treat DB Record as an Object
- **Entity Class** should only have
  - Properties that are equivalent to Table Columns
  - Optional helper Methods
  - Optional Constructor
- **Command (DTO)** as equivalent purpose as **Entity (DO)**
  - **Command (DTO)** is used when back-end wants to send/retrieve data to **front-end**
  - **Entity (DO)** is used when back-end wants to send/retrieve data to **DB**

*PersonEntity.java*

```
package com.ivoronline.test_spring_boot.model;

public class PersonEntity {

    //PROPERTIES
    private Long id;
    private String name;
    private Integer age;

    //CONSTRUCTORS
    public PersonEntity() { }

    //SETTERS
    public void setId (Long id) { this.id = id; }
    public void setName(String name) { this.name = name; }
    public void setAge (Integer age) { this.age = age; }

    //GETTERS
    public Long getId () { return id; }
    public String getName() { return name; }
    public Integer getAge () { return age; }

}
```

## Entity

---

- Generally speaking **Entity is something that exists.**

Entity refers to anything that exists and has properties that differentiate it from other stuff.

We give different names to different Entities so that we could know what we are referring to while communicating.

Examples of Entities are: Tree, Dog, Table, Cloud.

- Generally speaking Entity has set of Properties (name, color, size) and set of behaviors (run, sing, calculate).

In OOP Languages Entities are modeled using Classes which have Properties and Methods.

- But inside Spring Boot

- Entity has a more restricted meaning and includes only static Properties (without behaviors).
- Entity Class is equivalent to DB Table.
- Instance of Entity Class is equivalent to single Record in DB Table.

## POJO

---

[R]

- POJO is Plain Old/Ordinary Java Object that has following characteristics

- Constructor is optional
- there is no restriction on access-modifiers of fields (Fields don't have to be private)
- Fields/Properties can be accessed directly (they don't need to be accessed through constructors, getters or setters)
- it should not extend prespecified classes: `class NotPOJO extends HttpServlet`
- it should not implement prespecified interfaces: `class NotPOJO implements EntityBean`
- it should not contain prespecified annotations: `@Entity class NotPOJO`

## Java Bean

---

[R]

- Java Bean is a special type of POJO that has following restrictions

- it must implement Serializable interface
- it must have no-arg Constructor (Constructor that accepts no arguments)
- direct access to Fields is forbidden
  - all Fields/Properties must be private
  - all Fields/Properties must have getters or setters or both
  - Fields/Properties can be accessed through constructors

## Domain & Domain Object

---

- Domain is subject/purpose of your business logic.
- Domain Object is an Object/Entity that is related to that Domain.

- So Domain is what the application is about and Domain Objects are Objects that will be used inside application.

- For example

- `Domain = selling stuff` & Domain Objects = car, receipt, customer
- `Domain = growing vegetables` & Domain Objects = tomato, worker, storage
- `Domain = healing pet` & Domain Objects = pet, owner, appointment

## Lombok - Hiding Helper Methods

---

- The only useful part of the Entity Class (from a business logic perspective) are its static Properties.
- Helper Methods (like setters and getters) are just part of the plumbing/implementation so that Entity Class could fulfill its role of store some data/properties.
- Since these helper methods are the same for all Entities, and therefore do not provide any useful information, they are in our way obfuscating the true purpose of Entity Class.
- Therefore it is useful to hide helper Methods so that we can focus on different Properties as we move between Entities.
- For that purpose we can use **Lombok API** which implements Annotations that provide such service.
- Lombok's Annotations inject helper Methods behind the scene allowing us to ignore them inside Entity Class.

## 2.2.2 Data Access Object (DAO)

### Info

---

- Repository Object and DAO are synonyms.
  - DAO stands for **Data Access Object** (in Spring Boot where they refer to the same thing)  
(Class/Object that is used to store and retrieve data)
- Repository
  - contains methods for accessing database (for storing and retrieving data)
  - contains **Entity** Property (which contains actual data that should be stored or was retrieved)
- Term Repository might be misleading since generally repository means "a **place** where something is stored".  
In contrast Repository Object is not "a **place** where something is stored".  
Instead it is a collection of Methods that are used to communicate with "a **place** where something is stored".  
Having this in mind DAO as Data **Access** Object represents the purpose of this Object much better.  
Because it says it is something that allows you to **access** data and not "a **place** where data is stored".
- Some use DAO instead of **Entity** by having **Entity** Properties, Setters and Getters inside DAO itself.  
This way DAO is used to both represent DB Table/Record and to encapsulate Methods for communicating with DB.  
In Spring Boot we will have clear distinction between **Entity** and Repository (DAO).

## 2.2.3 Data Transfer Object (DTO)

Info

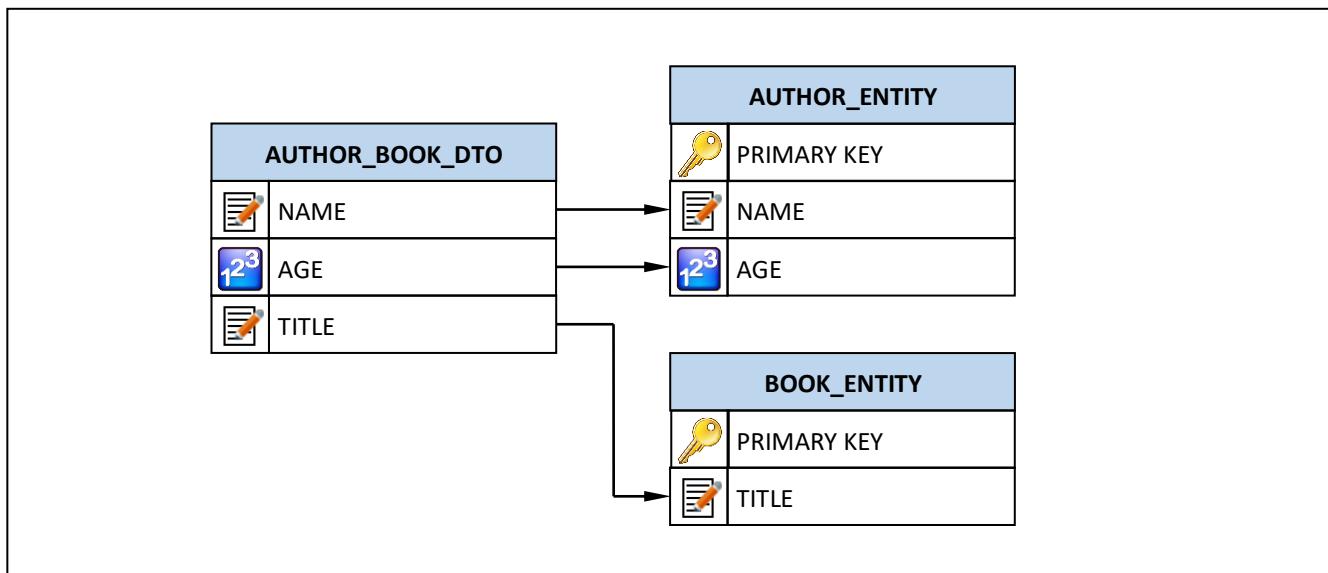
[R]

- **Data Transfer Object (DTO)**
  - contains data that should be transferred between application layers (but not between application and DB)
  - is used as a replacement for [Entity](#) when not all [Entity](#) Properties need to be sent
- Just like [Entity](#), Command Class should only have
  - Properties that are equivalent to Table Columns
  - Setters and getters for setting and reading these Properties
  - Optional Constructor for settings all of these properties at once
- Command Class should not have any logic (just pure data to transfer over layers/network).  
It also shouldn't contain any methods for converting between Command and [Entity](#) Objects.  
These methods should be implemented inside Converter Object.
- For example, if Table has 100 Columns, then corresponding [Entity](#) will have 100 Properties.  
But if we need to display only 10 Properties on our interface, these can be encapsulated inside the Command Object.  
Now our backend can send Command Object with only 10 Properties to our front end (instead of sending Entity with 100 Properties).
- **Command (DTO) serves similar purpose to Entity (DO)**
  - [Command \(DTO\)](#) is used when back-end wants to send/retrieve data to **front-end**
  - [Entity \(DO\)](#) is used when back-end wants to send/retrieve data to **DB**
- If your [Entity](#) Object is small enough and if it doesn't contain any sensitive data, it can be sent directly to the front-end.  
In that case you wouldn't need neither [Converter \(DCO\)](#) nor [Command \(DTO\)](#).

### DTO vs Entity

- The best way to understand difference between DTO and Entity is to create DTO that contains data for multiple Entities.  
Service will then extract data from DTO in order to create distinct Entities that can be stored into DB.
- For instance HTML Form might be used to enter both Author and Book data.  
Inside Controller such HTTP Request is converted into DTO and forwarded to Service.  
In order to save Author and Book as separate Entities Service must extract relevant data from DTO.
- DTO can contain data for one or more Entities which are used to create those Entities.  
This means that DTO should be converted into Entities.  
Conversion is usually done through Mapper Class that maps DTO Properties into Entity Properties with the same name.  
Properties that are missing in DTO but exist in Entity remain empty in Entity.

#### DTO conversion into Entities



- DTO **decouples** front-end from the Domain Model.  
With DTO changes to Domain Model don't need to influence front-end since DTO might remain the same.  
Also DTO can contain **data from multiple Entities** making it easier to send all of that data in one go.
- Exposing Domain Objects to the UI is a bad idea because as the application grows the **needs** for
  - **UI** change and force your Domain Objects to accommodate those changes
  - **Domain Objects** change and force your UI to accommodate those changesIn other words your Domain Objects end up **serving 2 masters** which address different needs of the application
  - User Interface (UI) which has the function of communicating with the user
  - Domain Model which holds business rules and stores data
- If you return part of your Domain Model to the front-end then Domain Model becomes part of a **contract** with front-end.  
This is problematic because a contract is hard to change (since things outside of your context depend on it).  
Therefore you would be making part of your Domain Model hard to change.  
Because every change would need to be propagated all the way to the front-end.  
Very important aspect of Domain Model is that it is easy to change.  
This makes us flexible to the domain's changing requirements.

## When to use Entity (instead DTO)

- At the beginning of the Project you might decide to use only Entities.  
This way you don't have to change two things every time a new change is requested.
- Also during development you will probably use test data so there is no need to protect sensitive data.
- Even after the Project is finished you don't have to have DTO for every Entity.  
If you need to return single Entity which is small and has no sensitive data you don't need to create DTO for it.
- If at later time front-end and back-end requirements start to diverge you can simply introduce DTO which will look like the current Entity and then add changes to the Entity so that they can continue to live independently.
- But no matter if you use Entity or DTO for communication between front-end and back-end you will always need Services.  
This is because Service Layer contains business logic and it is always good to have it encapsulated separately.

## 2.2.4 Data Converter Object (DCO)

### Info

---

- Converter Object and DCO are synonyms (in Spring Boot where they refer to the same thing)
  - DCO stands for **Data Conversion Object**
- Converter Class has methods for converting between **Command** and **Entity** Objects.  
Spring Boot will instantiate single instance of this Class for us to use.
- So when backend wants to send some data to the front end it will use Converter to convert **Entity** (with 100 Properties) into **Command Object** (with 10 Properties) which will then be sent to the front end to display it.
- If your **Entity** Object is small enough and if it doesn't contain any sensitive data, it can be sent directly to the front-end.  
In that case you wouldn't need either **Converter (DCO)** nor **Command (DTO)**.

## 2.2.5 Controller Object (CO)

### Info

---

- Controller Object and CO are synonyms (in Spring Boot where they refer to the same thing)
  - CO stands for **Controller Object**
- Controller Class has methods (also known as Endpoints) for accepting HTTP Requests.  
Spring Boot will instantiate single instance of this Class for us to use.
- Back-end uses controller to receive requests from Front-end.

## 2.3 Controller

### Info

---

- Following tutorials explain different Controller Annotations.
- By using Controller Annotations you are telling Spring that Class contains Endpoints (Methods called for specific URLs).
- Spring needs to collect these Methods to properly configure Application Server (like Tomcat).
- Application Server needs to know for which URLs which Method needs to be called.

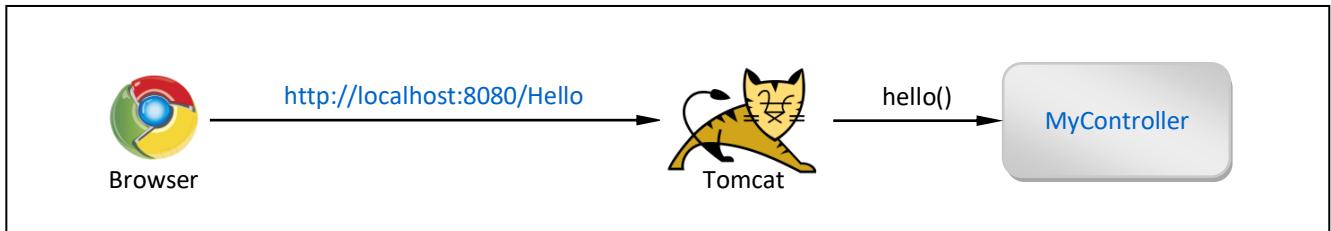
## 2.3.1 @Controller

### Info

- `@Controller` Annotation tells Spring that Class is Controller. (it contains Endpoints)
- Endpoints in `@Controller` Class by default return a View. (HTML or Thymeleaf)
- If you want Endpoint to return actual data you can Annotated Endpoint with `@ResponseBody`. (as is done in this tutorial)

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables <code>@RequestMapping</code> . Includes Tomcat HTTP Server.

### Procedure

- Create Project: controller\_returns\_text (add Spring Boot Starters from the table)
- Create Package: controllers (inside main package)
  - Create Class: `MyController.java` (inside controllers package)

### MyController.java

```
package com.ivoronline.controller_returns_text.controllers;

import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ResponseBody;

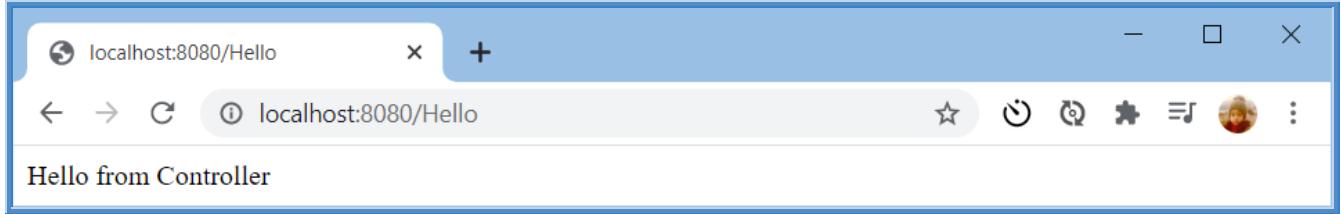
@Controller
public class MyController {

    @ResponseBody
    @RequestMapping("/Hello")
    public String hello() {
        return "Hello from Controller";
    }
}
```

## Results

---

<http://localhost:8080>Hello>



pom.xml

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

## 2.3.2 @RestController

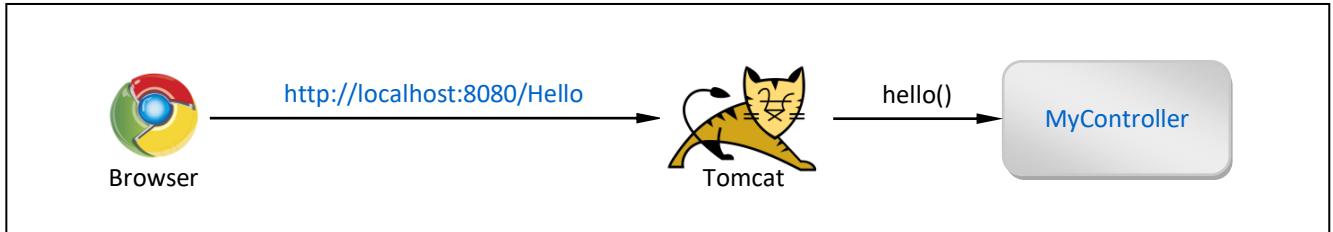
### Info

[G] [R]

- `@RestController` Annotation tells Spring that Class is Controller. (it contains Endpoints)
- Endpoints in `@RestController` Class can only return **data**. (they can't return Views like HTML or Thymeleaf)
- Using `@RestController` is the same as using `@Controller` in combination with `@ResponseBody`.

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables: <code>@RestController</code> , <code>@RequestMapping</code> , Tomcat Server

### Procedure

- **Create Project:** `springboot_controller_annotation_restcontroller` (add Spring Boot Starters from the table)
- **Create Package:** controllers (inside main package)
  - **Create Class:** `MyController.java` (inside controllers package)

### MyController.java

```
package com.ivoronline.springboot_controller_annotation_restcontroller.controllers;

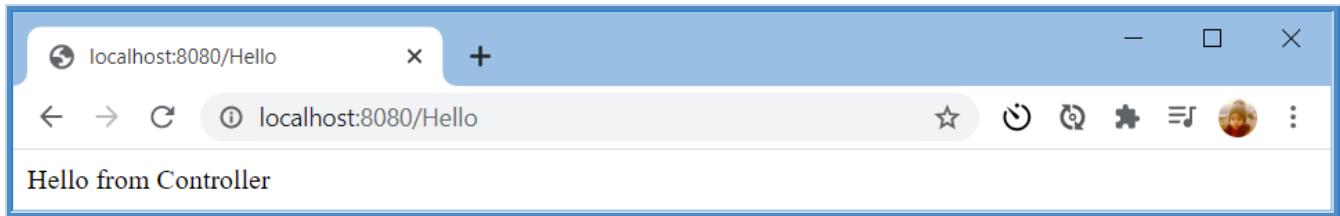
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class MyController {

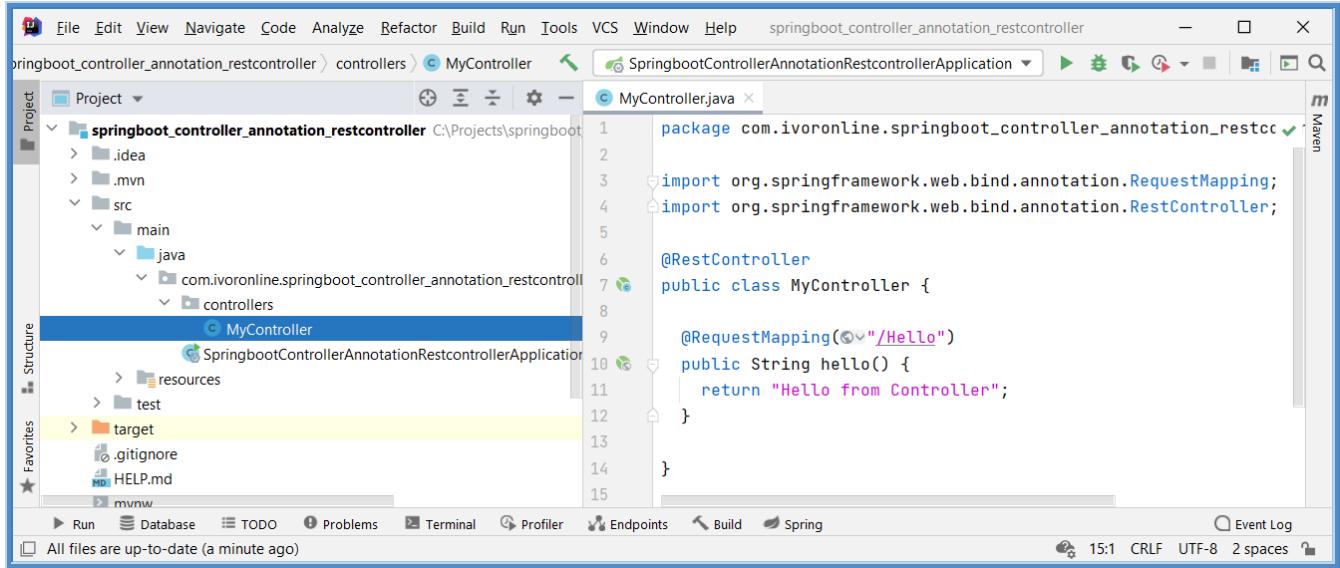
    @RequestMapping("/Hello")
    public String hello() {
        return "Hello from Controller";
    }
}
```

## Results

<http://localhost:8080>Hello>



## Application Structure



## pom.xml

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

## 2.4 Endpoint

### Info

---

- Following tutorials explain how to work with Endpoints.

Endpoint is Method called for specific URL.

Spring collects Endpoints from Controllers to properly configure Application Server (like Tomcat).

Application Server needs to know for which URL which Method needs to be called.

- Endpoint can return

- **View** (HTML, JSP, Thymeleaf)
- **Data** (Text, JSON)

- In the `@RestController` Endpoint can only return Data.

- In the `@Controller` Endpoint returns

- View by default
- Data if it is Annotated with `@ResponseBody`

- Endpoint Annotations define

- for which URL Endpoint should be called ("Hello")
- for which type of HTTP Request Endpoint should be called (GET, POST, UPDATE, DELETE)

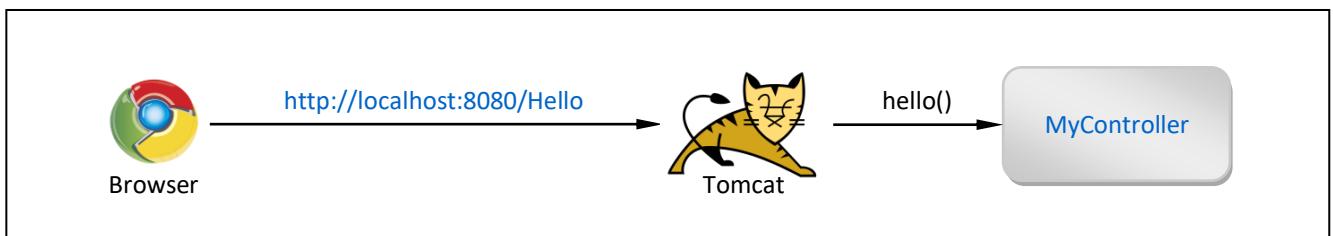
## 2.4.1 @RequestMapping

### Info

- `@RequestMapping` tells Spring for which URL to call the Endpoint.  
Endpoint will be called for all types of HTTP Requests: GET, POST, UPDATE, DELETE.
- To specify that Endpoint should be called only for specific types of HTTP Requests you can combine Annotations: `@GetMapping`, `@PostMapping`.

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables: <code>@Controller</code> , <code>@ResponseBody</code> , <code>@RequestMapping</code> , Tomcat Server

### Procedure

- Create Project: controller\_returns\_text (add Spring Boot Starters from the table)
- Create Package: controllers (inside main package)
  - Create Class: `MyController.java` (inside controllers package)

### MyController.java

```
package com.ivoronline.controller_returns_text.controllers;

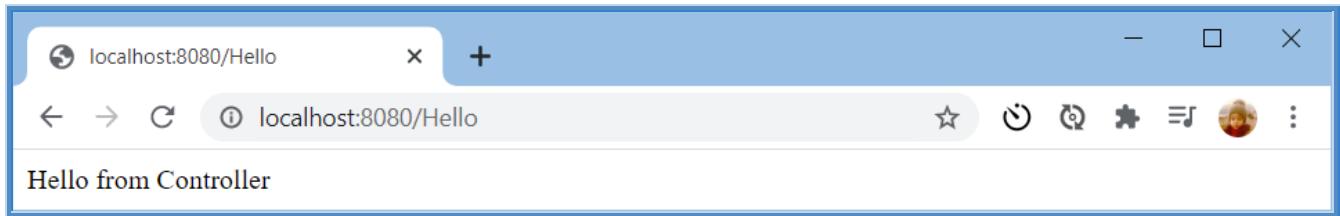
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class MyController {

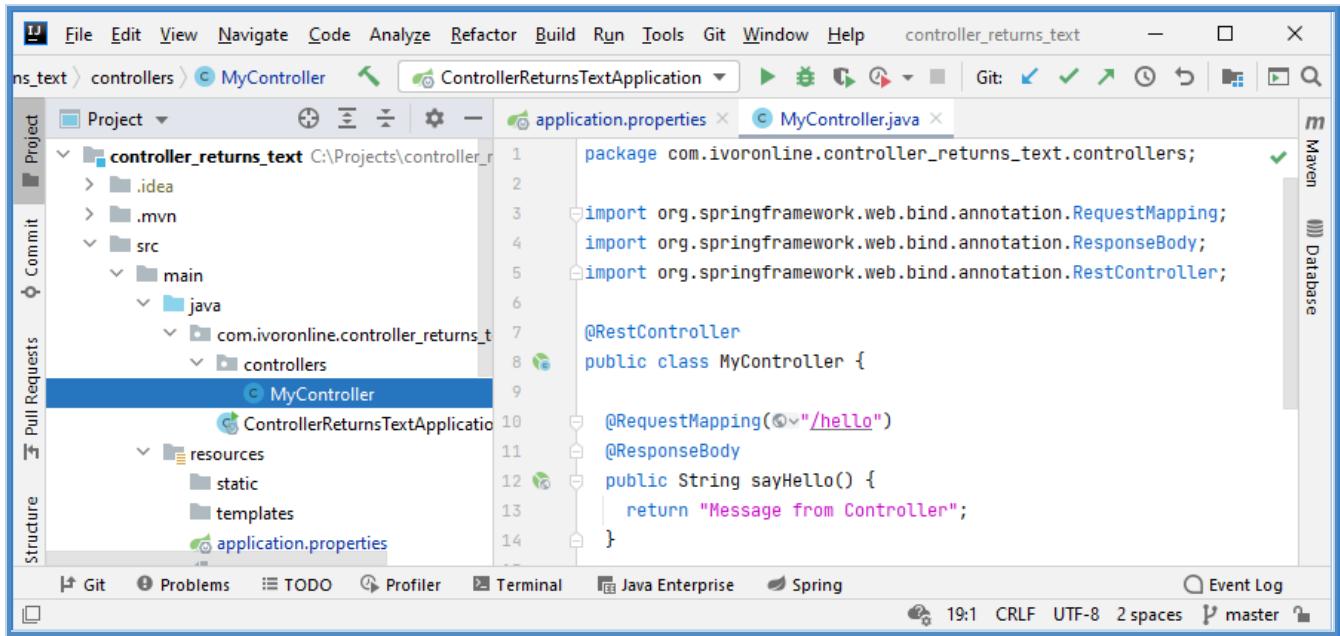
    @ResponseBody
    @RequestMapping("/Hello")
    public String hello() {
        return "Hello from Controller";
    }
}
```

## Results

<http://localhost:8080>Hello>



## Application Structure



## pom.xml

```
<dependencies>

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

</dependencies>
```

## 2.4.2 @GetMapping

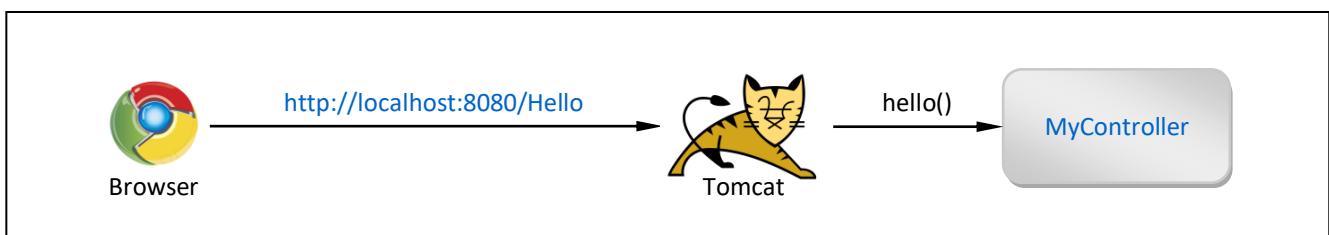
### Info

[G]

- `@GetMapping` Annotation tells Spring for which URL to call this Endpoint.  
But Endpoint will be called only for HTTP **GET** Requests.
- You can combine it with other Annotations of this type.  
For instance if you also use `@PostMapping` on the same Endpoint, it will get called for both HTTP **GET** and **POST** Requests.

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables: <code>@Controller</code> , <code>@ResponseBody</code> , <code>@GetMapping</code> , Tomcat Server

### Procedure

- Create Project: `springboot_endpoint_annotation_getmapping` (add Spring Boot Starters from the table)
- Create Package: `controllers` (inside main package)
  - Create Class: `MyController.java` (inside controllers package)

### MyController.java

```
package com.ivoronline.springboot_endpoint_annotation_getmapping.controllers;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ResponseBody;

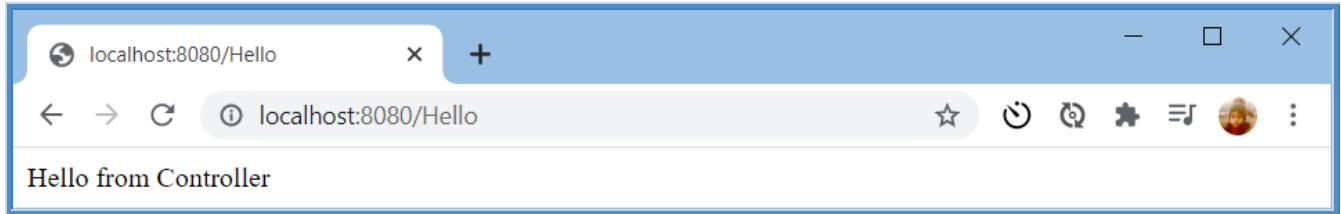
@Controller
public class MyController {

    @ResponseBody
    @GetMapping("/Hello")
    public String hello() {
        return "Hello from Controller";
    }
}
```

## Results

---

<http://localhost:8080>Hello>



pom.xml

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

## 2.4.3 @PostMapping

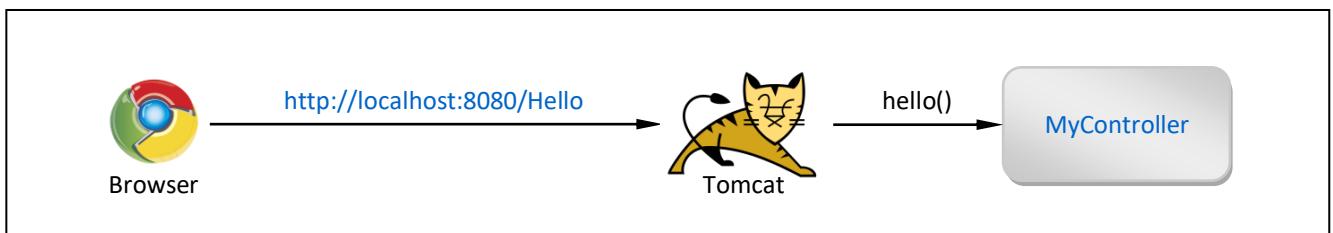
### Info

[G]

- `@PostMapping` Annotation tells Spring for which URL to call this Endpoint.  
But Endpoint will be called only for HTTP **POST** Requests.
- You can combine it with other Annotations of this type.  
For instance if you also use `@GetMapping` on the same Endpoint, it will get called for both HTTP **GET** and **POST** Requests.

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables: <code>@Controller</code> , <code>@ResponseBody</code> , <code>@PostMapping</code> , Tomcat Server

### Procedure

- `Create Project:` controller\_returns\_text (add Spring Boot Starters from the table)
- `Create Package:` controllers (inside main package)
  - `Create Class:` `MyController.java` (inside controllers package)

### MyController.java

```
package com.ivoronline.springboot_endpoint_annotation_postmapping.controllers;

import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ResponseBody;

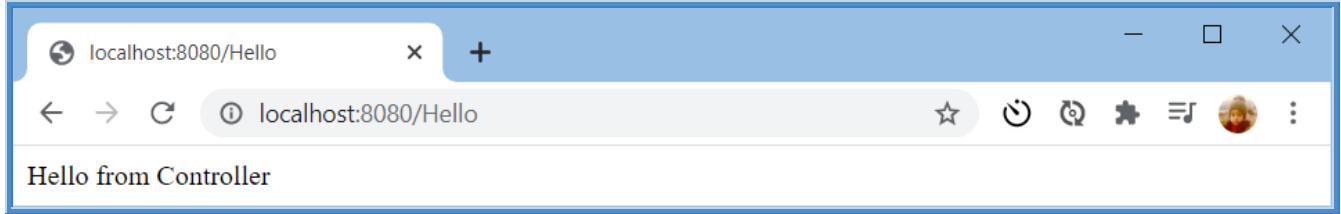
@Controller
public class MyController {

    @ResponseBody
    @PostMapping("/Hello")
    public String hello() {
        return "Hello from Controller";
    }
}
```

## Results

---

<http://localhost:8080>Hello>



pom.xml

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

## 2.4.4 @RequestParam

### Info

[G]

#### • @RequestParam Annotation

- is used to read HTTP Request Parameters
- is assigned to end-point Input Parameter - to Input Parameter of a @RequestMapping Method inside Controller
- stores HTTP Request Parameter that has the same as that Input Parameter (into that Input Parameter)

*MyController.java*

```
@ResponseBody  
@RequestMapping("/Hello")  
public String sayHello(@RequestParam String name) {  
    return "Hello " + name;  
}
```

*Application Schema*

[Results]



Browser

http://localhost:8080/Hello?name=John



Tomcat

hello()

MyController

*Spring Boot Starters*

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables @Controller and @RequestMapping. Includes Tomcat Server.

### Procedure

- Create Project: springboot\_httprequest\_requestparam (add Spring Boot Starters from the table)
- Create Package: controllers (inside main package)
  - Create Class: MyController.java (inside package controllers)

*MyController.java*

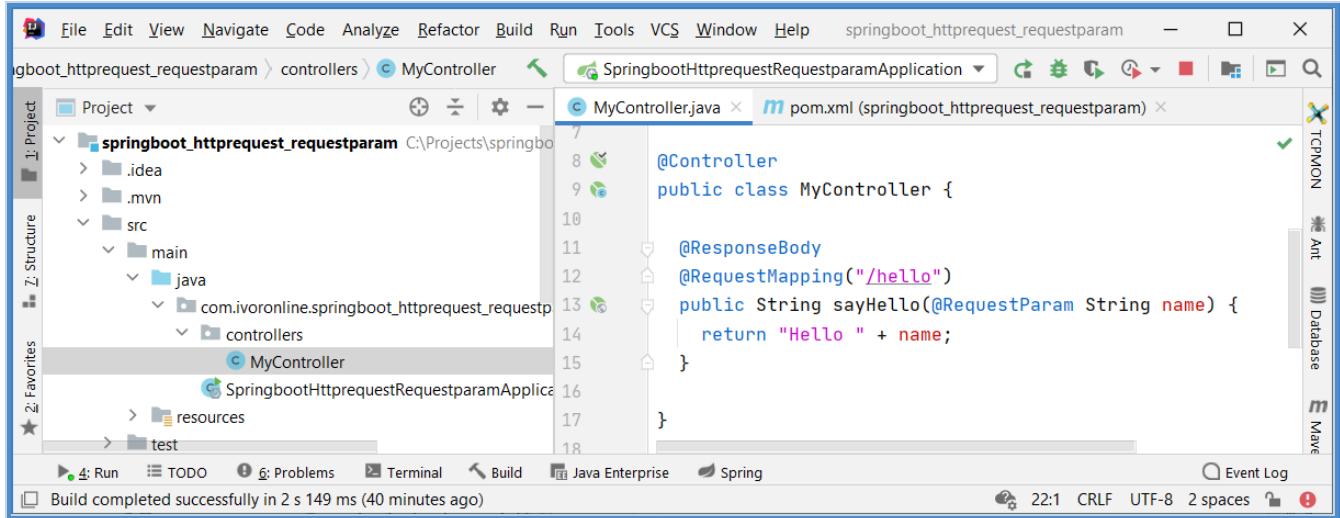
```
package com.ivoronline.springboot_httprequest_requestparam.controllers;  
  
import org.springframework.stereotype.Controller;  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.RequestParam;  
import org.springframework.web.bind.annotation.ResponseBody;  
  
@Controller  
public class MyController {  
  
    @ResponseBody  
    @RequestMapping("/Hello")  
    public String hello(@RequestParam String name) {  
        return "Hello " + name;  
    }  
}
```

## Results

<http://localhost:8080>Hello?name=John>



## Application Structure



## pom.xml

```
<dependencies>

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

</dependencies>
```

## 2.4.5 @PathVariable

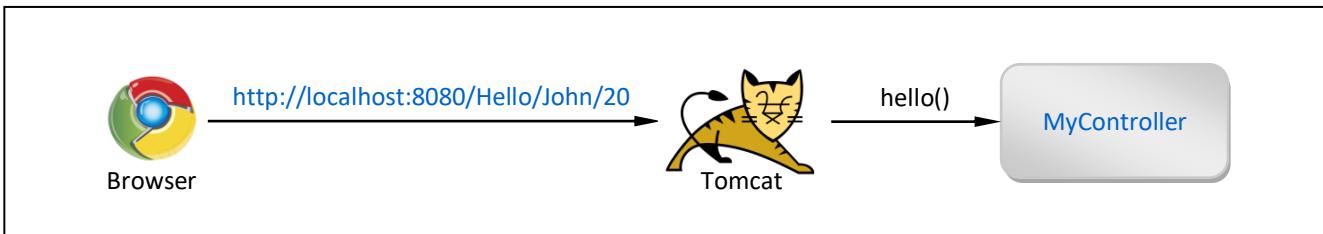
### Info

[G]

- `@PathVariable` stores Path Component into Endpoint's Input Parameter.

*Application Schema*

[Results]



*Spring Boot Starters*

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables <code>@Controller</code> and <code>@RequestMapping</code> . Includes Tomcat Server.

### Procedure

- Create Project: `springboot_endpoint_annotation_pathvariable` (add Spring Boot Starters from the table)
- Create Package: `controllers` (inside main package)
  - Create Class: `MyController.java` (inside package `controllers`)

*MyController.java*

```
package com.ivoronline.springboot_endpoint_annotation_pathvariable.controllers;

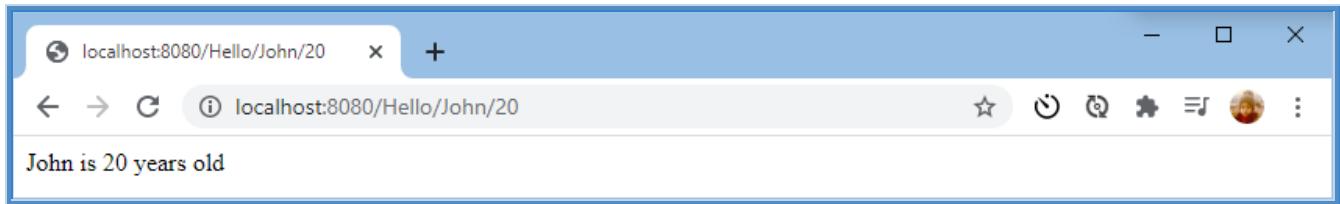
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class MyController {

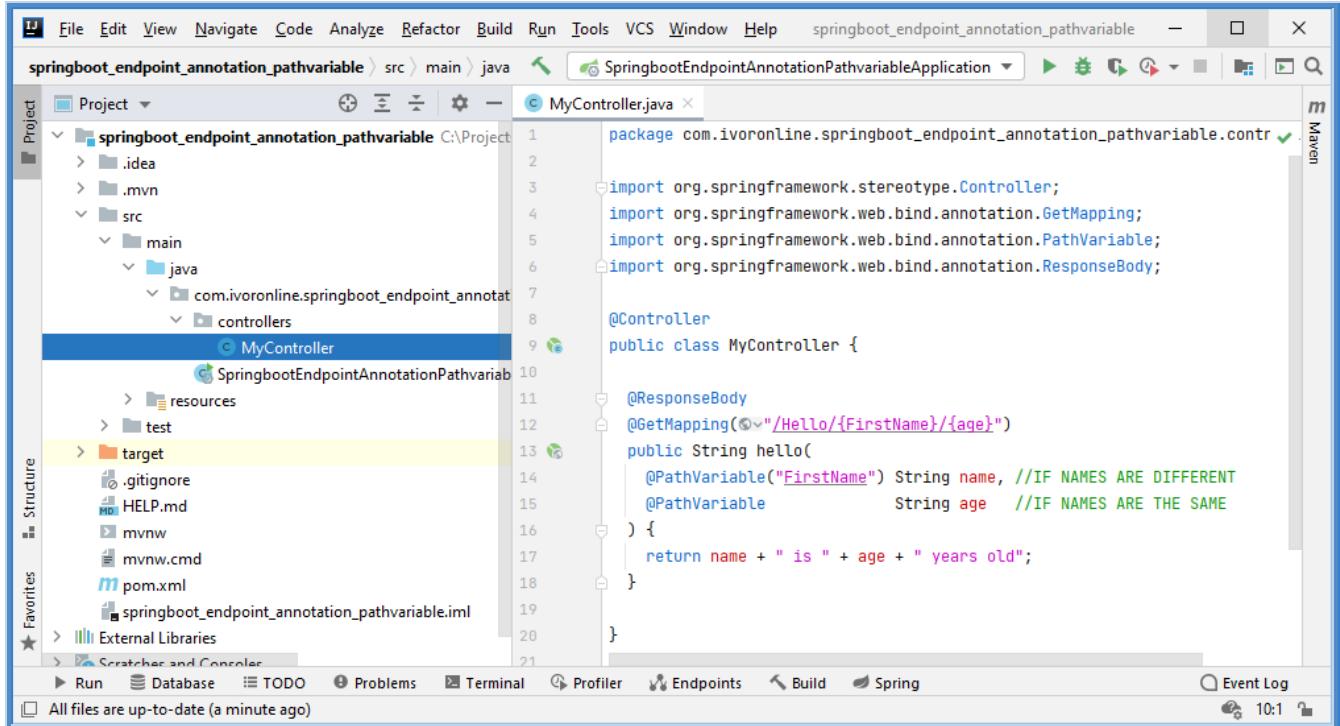
    @ResponseBody
    @GetMapping("/Hello/{FirstName}/{age}")
    public String hello(
        @PathVariable("FirstName") String name, //IF NAMES ARE DIFFERENT
        @PathVariable String age //IF NAMES ARE THE SAME
    ) {
        return name + " is " + age + " years old";
    }
}
```

## Results

<http://localhost:8080>Hello/John/20>



## Application Structure



## pom.xml

```
<dependencies>

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

</dependencies>
```

## 2.4.6 Return - Data - Text

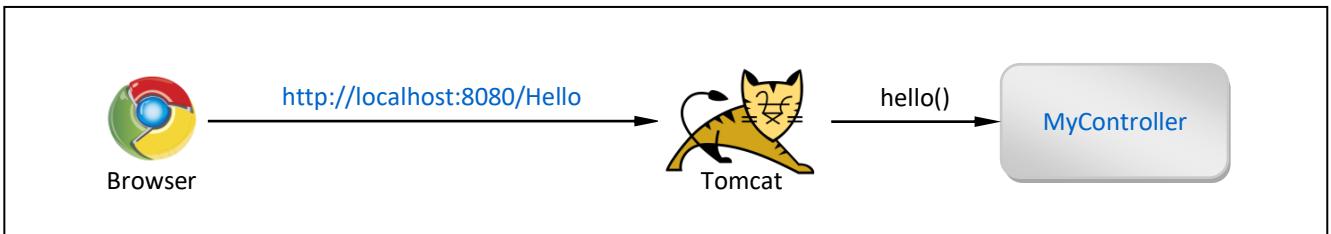
### Info

[G] [R]

- This tutorial shows how to use `@ResponseBody` to return Text from the Controller.

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables: <code>@RequestMapping</code> , Tomcat

### Procedure

- Create Project: controller\_returns\_text (add Spring Boot Starters from the table)
- Create Package: controllers (inside main package)
  - Create Class: `MyController.java` (inside controllers package)

### MyController.java

```
package com.ivoronline.controller_returns_text.controllers;

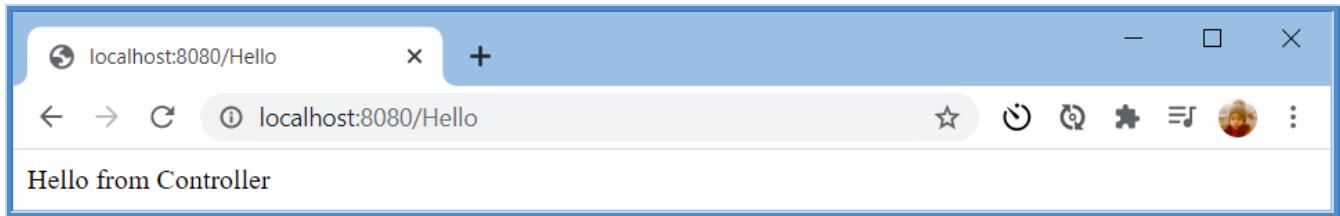
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class MyController {

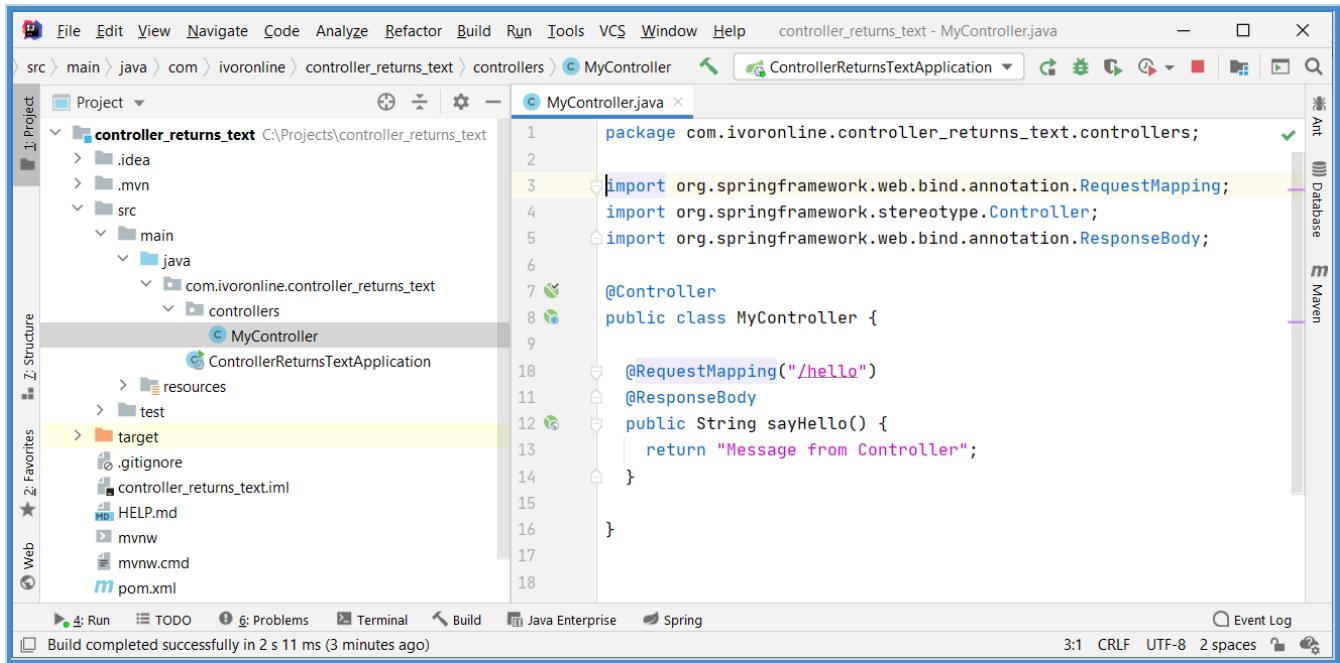
    @ResponseBody
    @RequestMapping("/Hello")
    public String hello() {
        return "Hello from Controller";
    }
}
```

## Results

<http://localhost:8080>Hello>



## Application Structure



## pom.xml

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

## 2.4.7 Return - Data - JSON

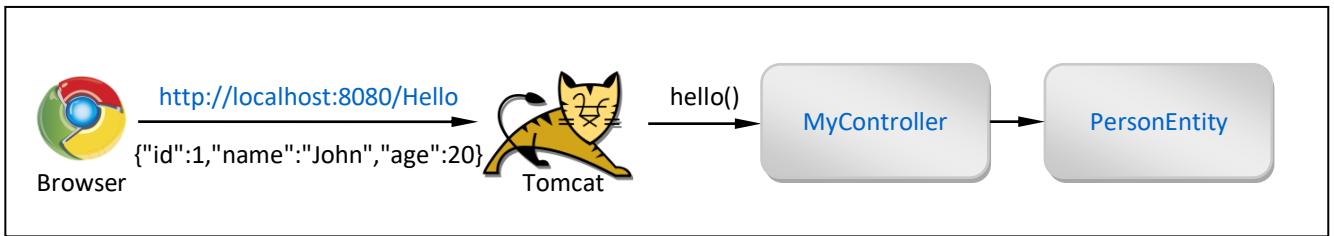
### Info

[G] [R]

- This tutorial shows how to use `@ResponseBody` to return instance of `PersonEntity` from the Controller as JSON String.

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables <code>@Controller</code> , <code>@RequestMapping</code> and Tomcat HTTP Server.

## Procedure

---

- [Create Project:](#) controller\_returns\_text\_json (add Spring Boot Starters from the table)
- [Create Package:](#) entities (inside main package)
- [Create Interface:](#) PersonEntity.java (inside package entities)
- [Create Package:](#) controllers (inside main package)
- [Create Class:](#) MyController.java (inside controllers package)

PersonEntity.java

```
package com.ivoronline.springboot.controller_returns_text_json.entities;

public class PersonEntity {
    public Integer id;
    public String name;
    public Integer age;
}
```

MyController.java

```
package com.ivoronline.springboot.controller_returns_text_json.controllers;

import com.ivoronline.springboot.controller_returns_text_json.entities.PersonEntity;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class MyController {

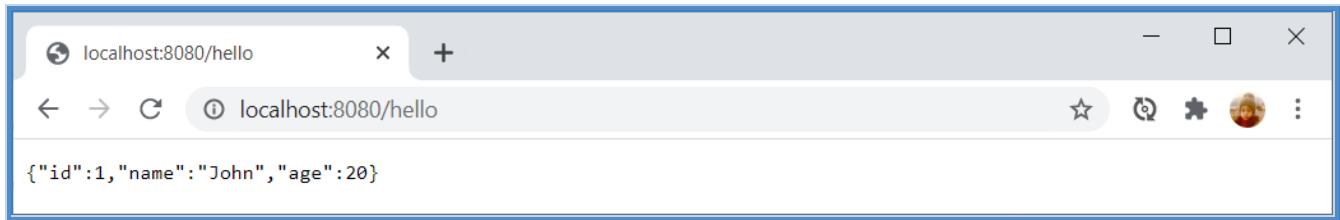
    @ResponseBody
    @RequestMapping("/Hello")
    public PersonEntity hello() {

        //CREATE INSTANCE
        PersonEntity personEntity = new PersonEntity();
        personEntity.id = 1;
        personEntity.name = "John";
        personEntity.age = 20;

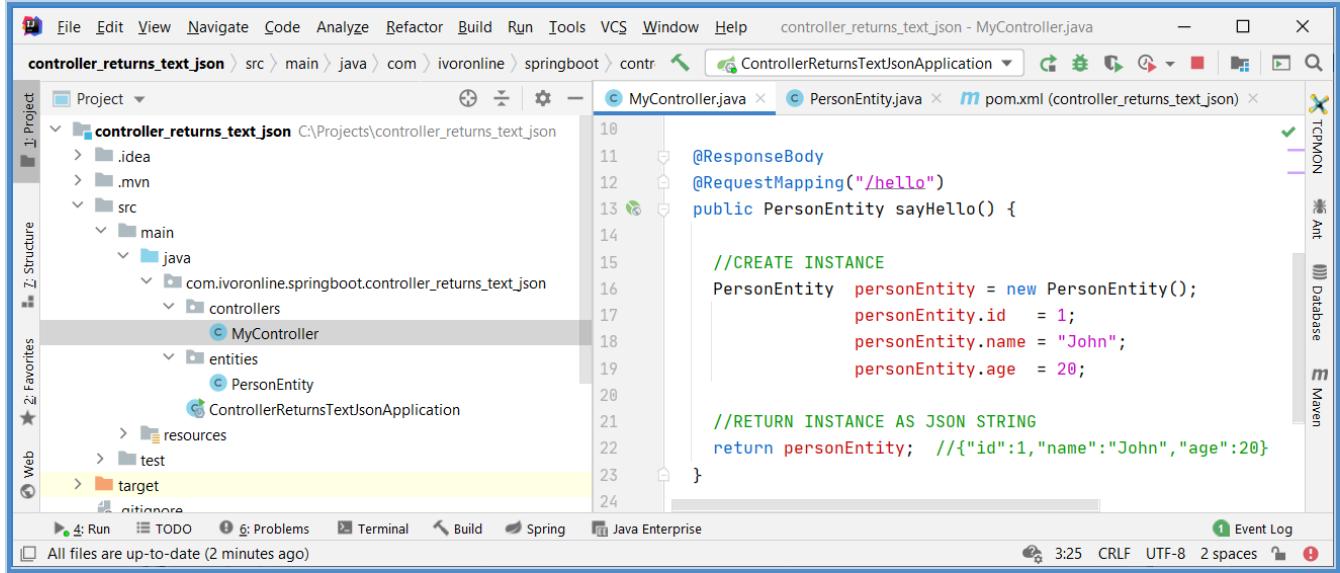
        //RETURN INSTANCE AS JSON STRING
        return personEntity; //{"id":1,"name":"John","age":20}
    }
}
```

## Results

<http://localhost:8080>Hello>



## Application Structure



## pom.xml

```
<dependencies>

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

</dependencies>
```

## 2.4.8 Return - View - HTML

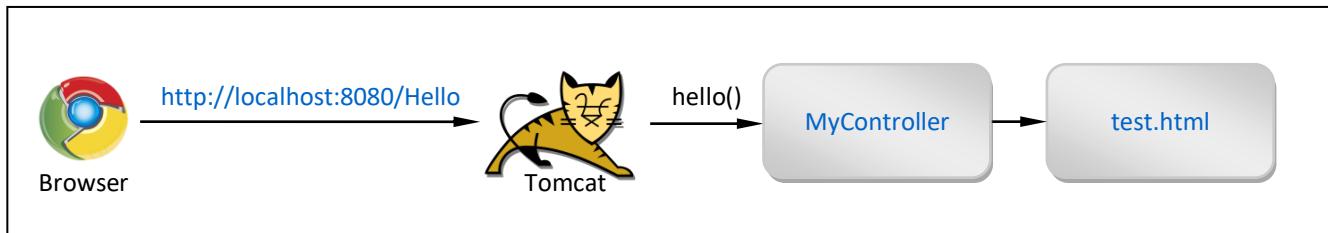
### Info

[G]

- In this tutorial we will build simple Spring Boot Application that only has Controller that returns static HTML page.

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables @Controller, @RequestMapping and Tomcat Server.
Template Engines	Thymeleaf	Enables Controller to return reference to HTML file test.html

### Procedure

- Create Project: controller\_application (add Spring Boot Starters from the table)
- Create Package: controllers (inside package com.ivoronline.test\_spring\_boot)
- Create Class: MyController.java (inside package controllers)
- Create HTML File: test.html (inside directory resources/templates)

### MyController.java

```
package com.ivoronline.controller_application.controllers;

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

@Controller
public class MyController {

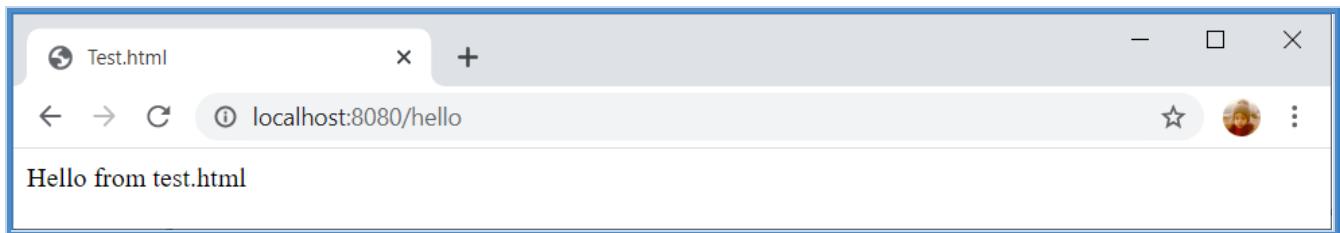
    @RequestMapping("/Hello")
    public String hello() {
        System.out.println("Hello from Controller");
        return "test";
    }
}
```

### test.html

```
<title>Test.html</title>
Hello from test.html
```

## Results

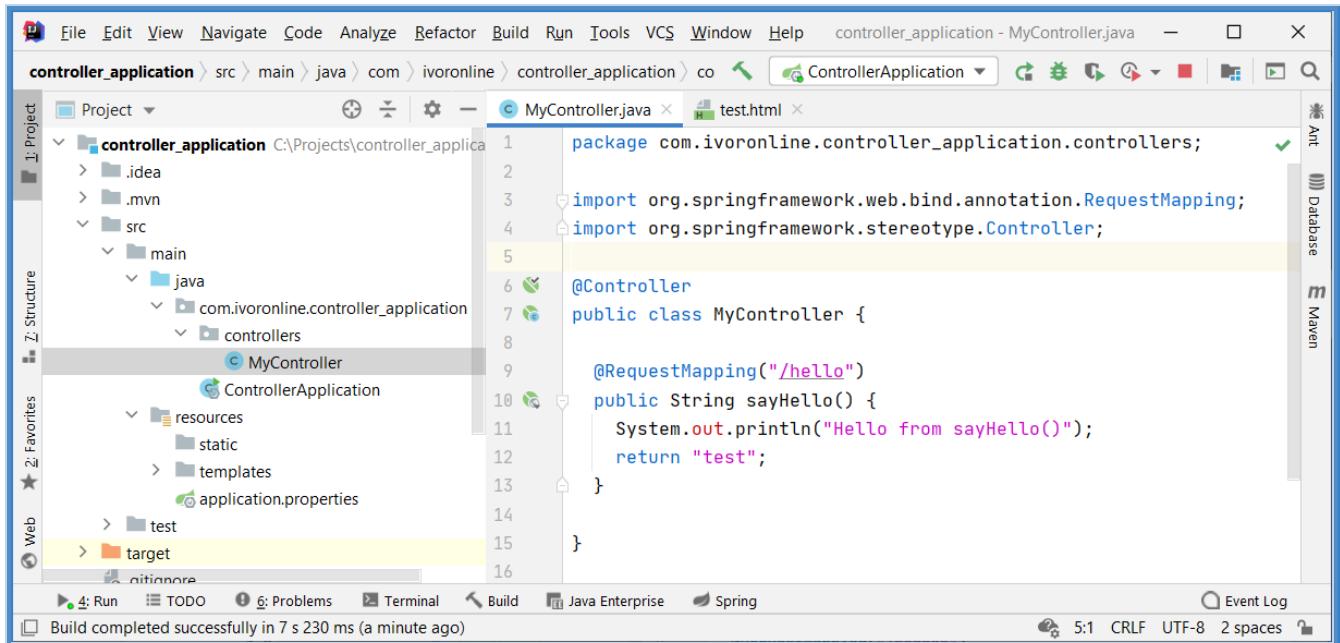
<http://localhost:8080>Hello>



## Console

Hello from Controller

## Application Structure



## pom.xml

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

## 2.4.9 Return - View - Thymeleaf

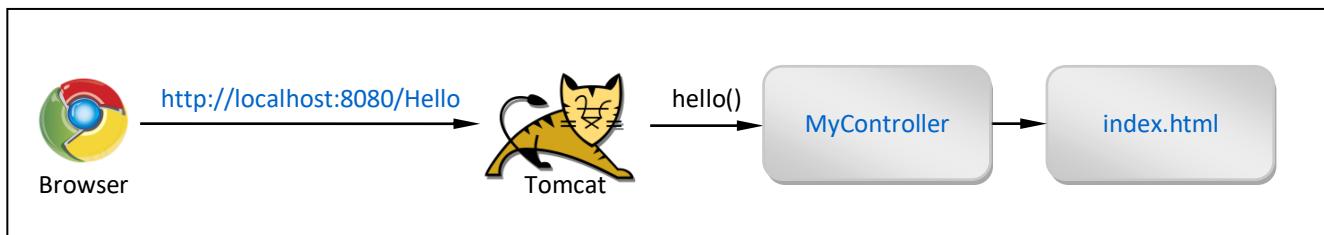
### Info

[G]

- In this tutorial we will build simple Spring Boot Application that only has Controller that returns Thymeleaf page.

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables @Controller, @RequestMapping and Tomcat Server.
Template Engines	Thymeleaf	Enables Controller to return reference to Thymeleaf template index.html

### Procedure

- Create Project:** controller\_returns\_thymeleaf (add Spring Boot Starters from the table)
- Create Package:** controllers (inside package com.ivoronline.test\_spring\_boot)
- **Create Class:** MyController.java (inside package controllers)
- Create Thymeleaf template:** index.html (inside directory resources/templates)

### MyController.java

```
package com.ivoronline.controller_application.controllers;

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

@Controller
public class MyController {

    @RequestMapping("/Hello")
    public String hello() {
        System.out.println("Hello from Controller");
        return "index";
    }
}
```

### index.html

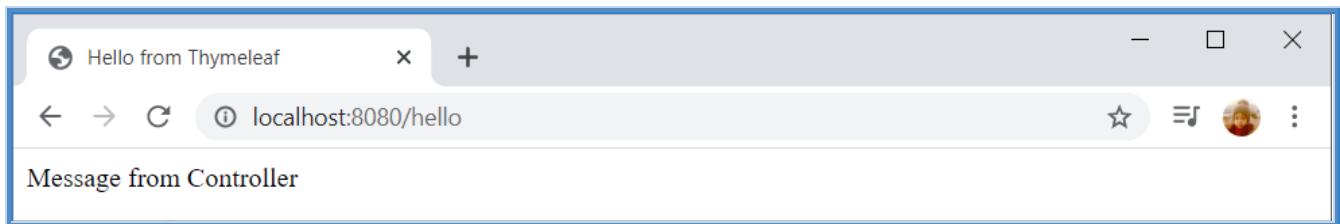
(Thymeleaf template)

```
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<title>Hello from Thymeleaf</title>
<p th:text="${message}">/>
```

## Results

<http://localhost:8080>Hello>

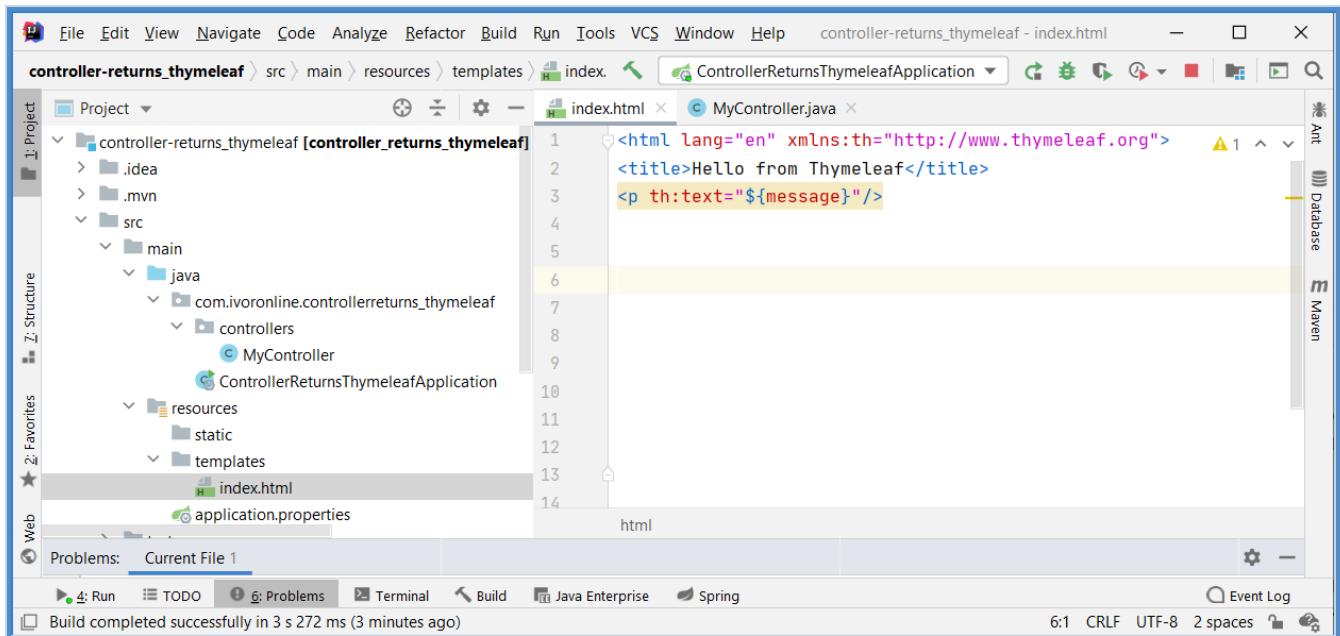
(Web Browser)



## Console

Hello from Controller

## Application Structure



## pom.xml

```
<dependencies>

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

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

</dependencies>
```

## 2.4.10 Return - View - JSP

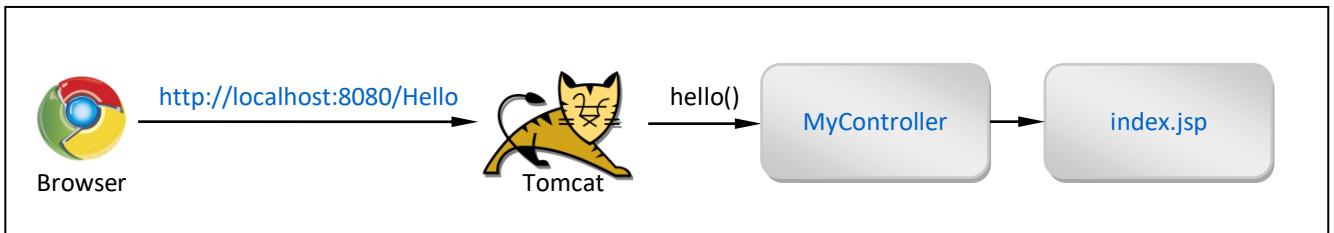
### Info

[G] [R]

- This tutorial shows how to return JSP from the Controller.

*Application Schema*

[Results]



*Spring Boot Starters*

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables @RequestMapping. Includes Tomcat HTTP Server.

## Procedure

– Create Project:	controller_returns_jsp	(add Spring Boot Starters from the table)
– Edit File:	pom.xml	(manually add jasper dependency to be able to compile JSP files)
– Edit File:	application.properties	(specify location of JSP files)
– Create Package:	controllers	(inside main package)
– Create Class:	MyController.java	(inside controllers package)
– Create Directories:	webapp/WEB-INF/jsp	(inside src/main/ directory)
– Create JSP file:	index.jsp	(inside jsp directory)

pom.xml

(manually add jasper dependency)

```
<dependency>
    <groupId>org.apache.tomcat.embed</groupId>
    <artifactId>tomcat-embed-jasper</artifactId>
    <scope>provided</scope>
</dependency>
```

application.properties

(Directory: src/main/resources)

```
# JSP
spring.mvc.view.prefix = /WEB-INF/jsp/
spring.mvc.view.suffix = .jsp
```

MyController.java

(Directory: src/main/java/.../controllers)

```
package com.ivoronline.controller_returns_jsp.controllers;

import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.stereotype.Controller;

@Controller
public class MyController {

    @RequestMapping("/Hello")
    public String hello(Model model) {
        model.addAttribute("message", "Hello from Controller");
        return "index";
    }

}
```

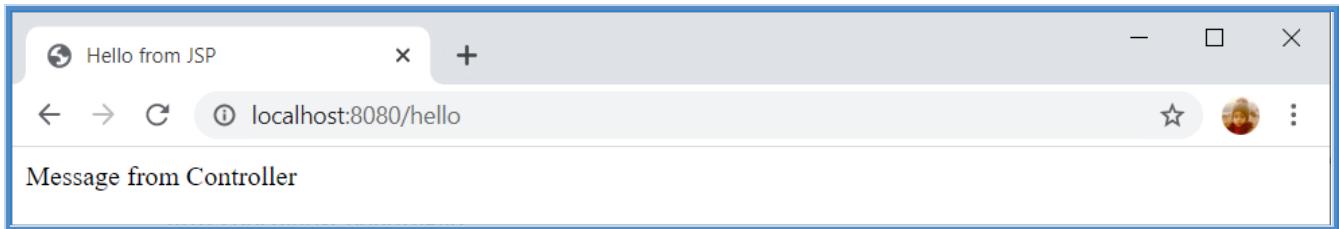
index.jsp

(Directory: src/main/webapp/WEB-INF/jsp)

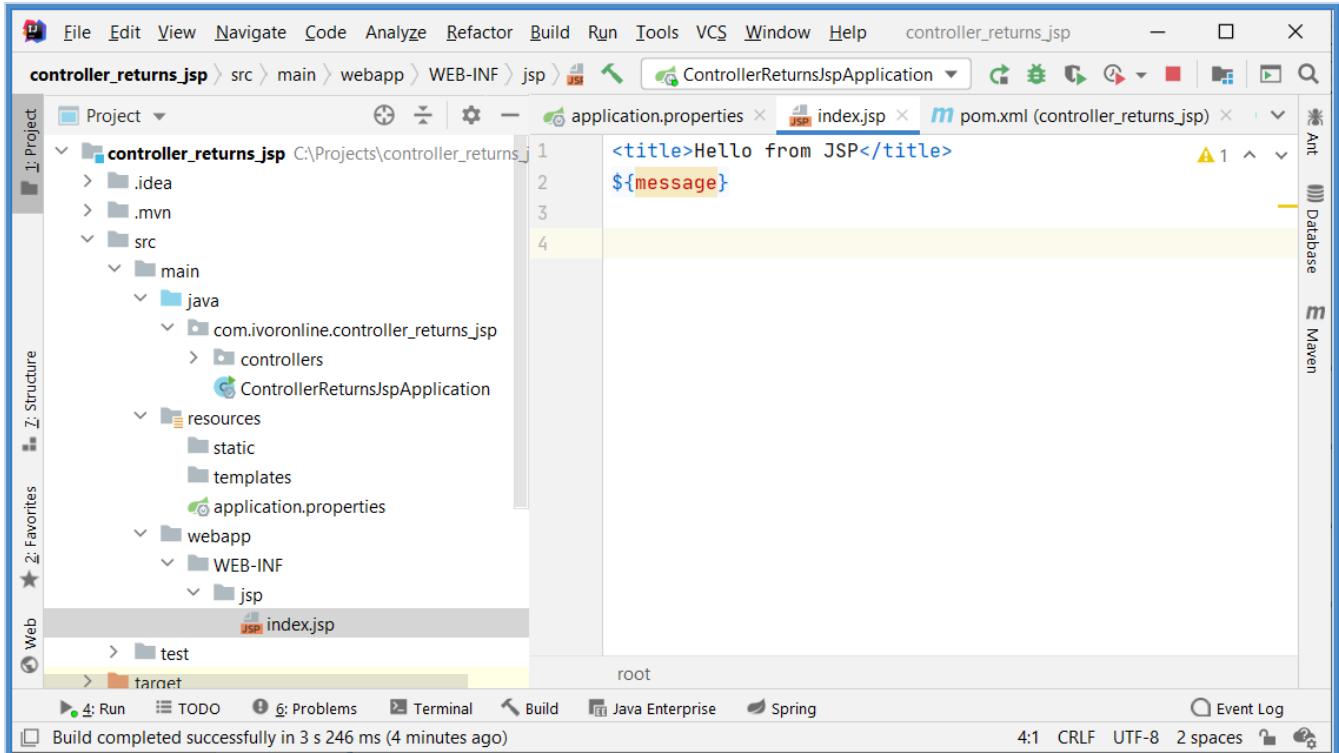
```
<title>Hello from JSP</title>
${message}
```

## Results

<http://localhost:8080>Hello>



## Application Structure



pom.xml

(manually add jasper dependency)

```
<dependencies>

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

    <dependency>
        <groupId>org.apache.tomcat.embed</groupId>
        <artifactId>tomcat-embed-jasper</artifactId>
        <scope>provided</scope>
    </dependency>

</dependencies>
```

## 2.4.11 Download - Excel

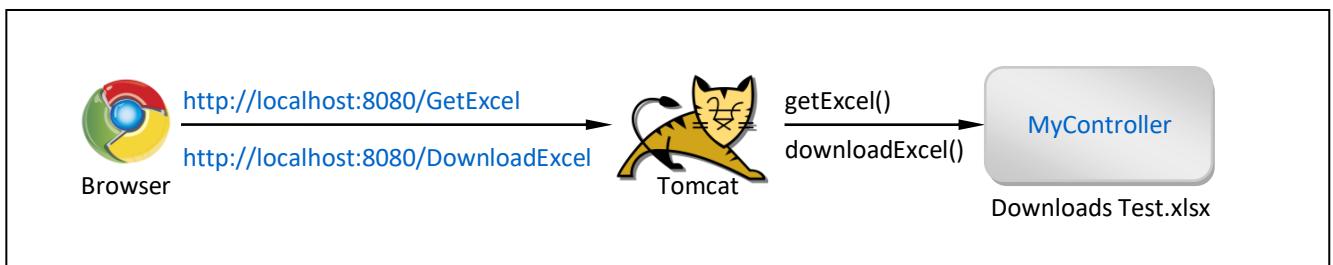
### Info

[G] [R]

- This tutorial shows how to use  `ResponseEntity` to download `Excel` from the Controller.
- You can either call
  - `http://localhost:8080/DownloadExcel` to directly download Excel (Page Content is not changed)
  - `http://localhost:8080/GetExcel` to display download link

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables @Controller, @RequestMapping and Tomcat HTTP Server.

## Procedure

- **Create Project:** `springboot_endpoint_return_excel` (add Spring Boot Starters from the table)
- **Edit File:** `pom.xml` (add poi Maven dependencies)
- **Create Package:** `controllers` (inside main package)
  - **Create Class:** `MyController.java` (inside controllers package)

*pom.xml*

```
<dependency>
    <groupId>org.apache.poi</groupId>
    <artifactId>poi</artifactId>
    <version>3.15</version>
</dependency>

<dependency>
    <groupId>org.apache.poi</groupId>
    <artifactId>poi-ooxml</artifactId>
    <version>3.15</version>
</dependency>
```

*MyController.java*

```
package com.ivoronline.springboot_endpoint_return_excel.controllers;

import org.apache.poi.ss.usermodel.Cell;
import org.apache.poi.ss.usermodel.Row;
import org.apache.poi.ss.usermodel.Sheet;
import org.apache.poi.ss.usermodel.Workbook;
import org.apache.poi.xssf.usermodel.XSSFWorkbook;
import org.springframework.http.HttpHeaders;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.servlet.mvc.method.annotation.StreamingResponseBody;

@Controller
public class MyController {

    //=====
    // GET EXCEL
    //=====

    @ResponseBody
    @GetMapping("/GetExcel")
    public String getExcel() {
        return "<a href='DownloadExcel'> Download Excel </a>";
    }

    //=====
    // DOWNLOAD EXCEL
    //=====

    @ResponseBody
    @GetMapping("/DownloadExcel")
    public ResponseEntity<StreamingResponseBody> DownloadExcel() {

        //CREATE EXCEL FILE
        Workbook workBook = new XSSFWorkbook();

        //CREATE TABLE
        Sheet sheet = workBook.createSheet("My Sheet");
        sheet.setColumnWidth(0, 10 * 256);           //10 characters wide
    }
}
```

```
//CREATE ROW
Row      row = sheet.createRow(0);

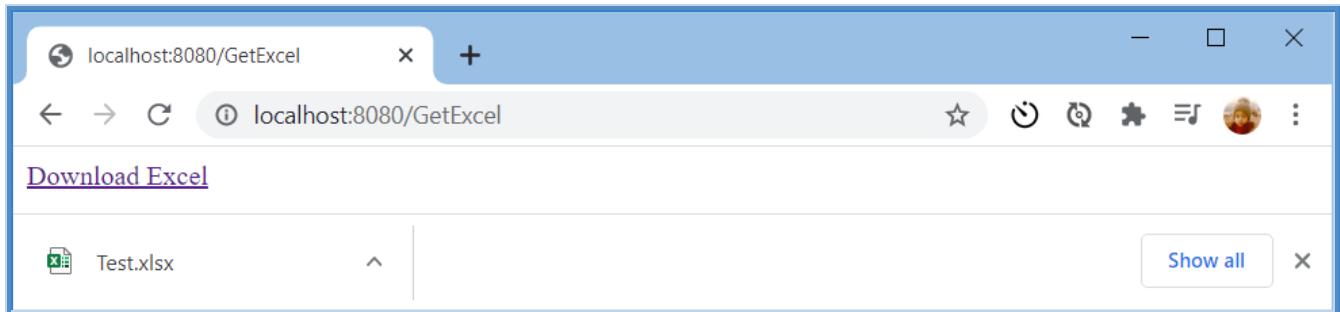
//CREATE CELL
Cell      cell = row.createCell(0);
cell.setCellValue("Hello World");

//DOWNLOAD EXCEL
return ResponseEntity
.ok()
.contentType(MediaType.APPLICATION_OCTET_STREAM)
.header(HttpHeaders.CONTENT_DISPOSITION, "inline;filename=\"Test.xlsx\"")
.body(workBook::write);
}

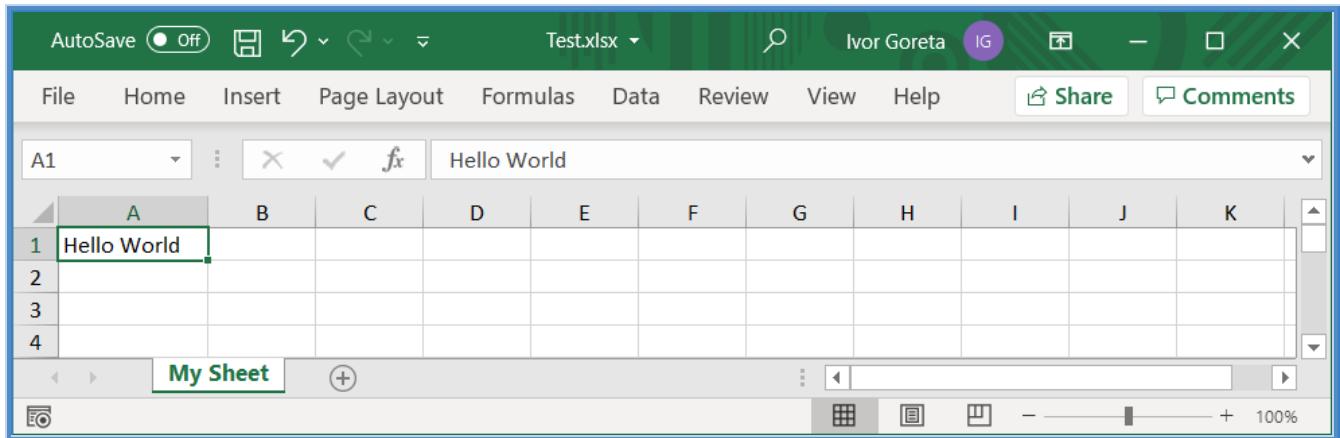
}
```

## Results

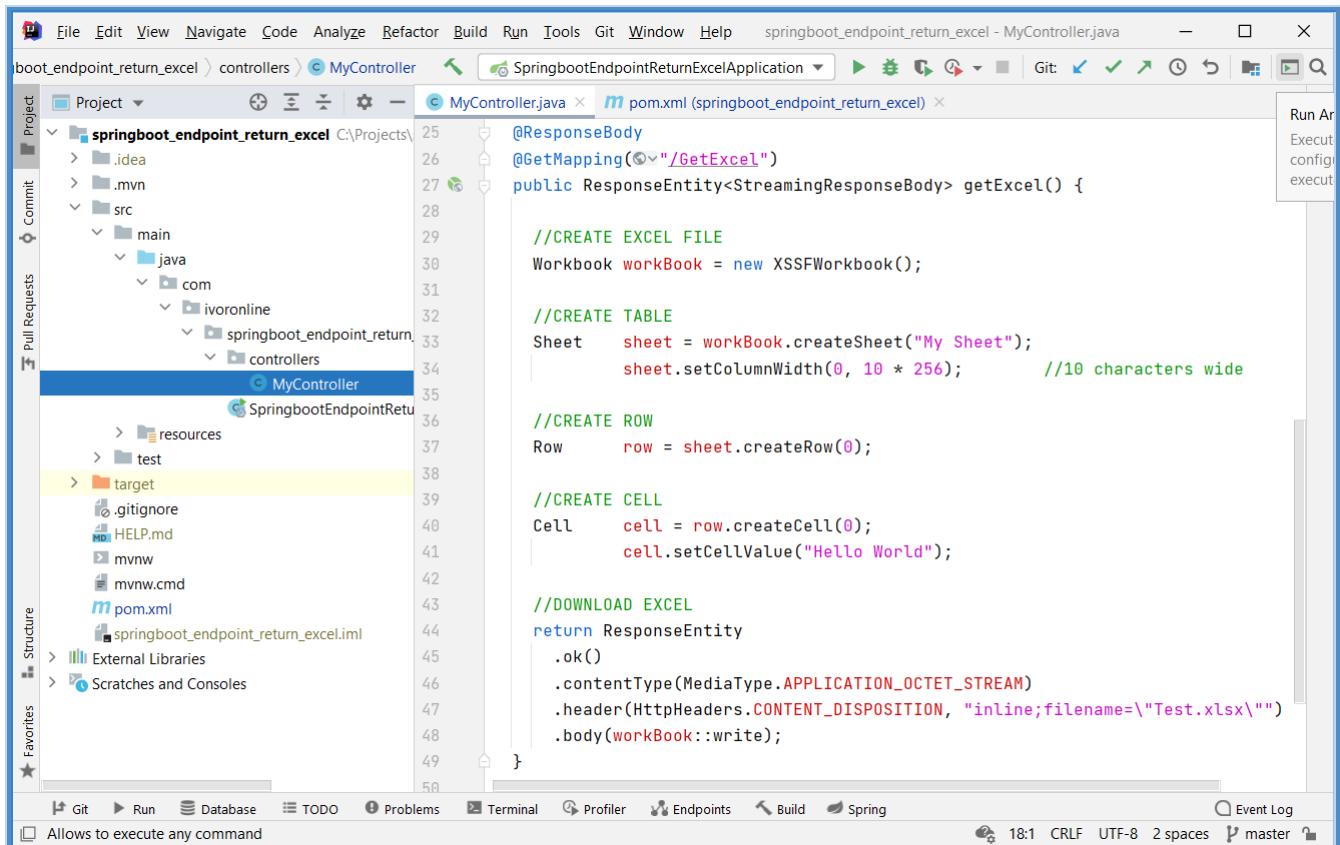
<http://localhost:8080/GetExcel>



Test.xlsx



## Application Structure



```
<dependencies>

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

    <dependency>
        <groupId>org.apache.poi</groupId>
        <artifactId>poi</artifactId>
        <version>3.15</version>
    </dependency>

    <dependency>
        <groupId>org.apache.poi</groupId>
        <artifactId>poi-ooxml</artifactId>
        <version>3.15</version>
    </dependency>

</dependencies>
```

## 2.4.12 Download - Excel - With Header

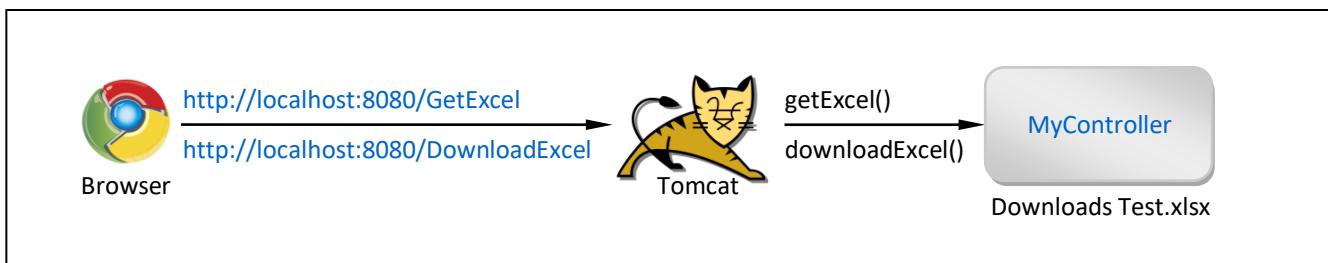
### Info

[G]

- This tutorial shows how to use `ResponseEntity` to download `Excel` from the Controller.  
And how to use first column as Header. (to specify Column Names)
- You can either call
  - `http://localhost:8080/DownloadExcel` to directly download Excel (Page Content is not changed)
  - `http://localhost:8080/GetExcel` to display download link

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables @Controller, @RequestMapping and Tomcat HTTP Server.

## Procedure

- **Create Project:** `springboot_endpoint_return_excel_header` (add Spring Boot Starters from the table)
- **Edit File:** `pom.xml` (add poi Maven dependencies)
- **Create Package:** `controllers` (inside main package)
- **Create Class:** `MyController.java` (inside controllers package)

*pom.xml*

```
<dependency>
    <groupId>org.apache.poi</groupId>
    <artifactId>poi</artifactId>
    <version>3.15</version>
</dependency>

<dependency>
    <groupId>org.apache.poi</groupId>
    <artifactId>poi-ooxml</artifactId>
    <version>3.15</version>
</dependency>
```

*MyController.java*

```
package com.ivoronline.springboot_endpoint_return_excel_header.controllers;

import org.apache.poi.ss.usermodel.Cell;
import org.apache.poi.ss.usermodel.Row;
import org.apache.poi.ss.usermodel.Sheet;
import org.apache.poi.ss.usermodel.Workbook;
import org.apache.poi.xssf.usermodel.XSSFWorkbook;
import org.springframework.http.HttpHeaders;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.servlet.mvc.method.annotation.StreamingResponseBody;

@Controller
public class MyController {

    //=====
    // GET EXCEL
    //=====

    @ResponseBody
    @GetMapping("/GetExcel")
    public String getExcel() {
        return "<a href='DownloadExcel'> Download Excel </a>";
    }

    //=====
    // DOWNLOAD EXCEL
    //=====

    @ResponseBody
    @GetMapping("/DownloadExcel")
    public ResponseEntity<StreamingResponseBody> DownloadExcel() {

        //CREATE EXCEL FILE
        Workbook workBook = new XSSFWorkbook();

        //CREATE TABLE
        Sheet sheet = workBook.createSheet("My Sheet");
        sheet.setColumnWidth(0, 10 * 256);           //10 characters wide
    }
}
```

```

//CREATE HEADER
Row    header = sheet.createRow(0);
header.createCell(0).setCellValue("NAME");
header.createCell(1).setCellValue("AGE");

//CREATE ROW 1
Row    row1 = sheet.createRow(1);
row1.createCell(0).setCellValue("John");
row1.createCell(1).setCellValue("20");

//CREATE ROW 2
Row    row2 = sheet.createRow(2);
row2.createCell(0).setCellValue("Bill");
row2.createCell(1).setCellValue("50");

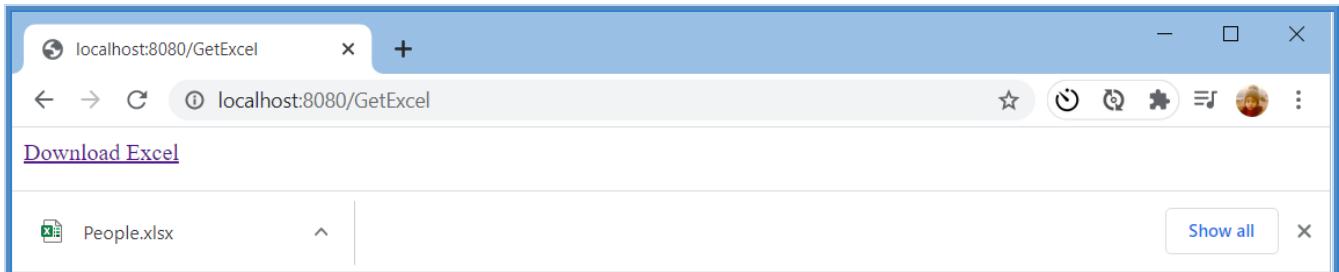
//DOWNLOAD EXCEL
return ResponseEntity
.ok()
.contentType(MediaType.APPLICATION_OCTET_STREAM)
.header(HttpHeaders.CONTENT_DISPOSITION, "inline;filename=\"People.xlsx\"")
.body(workBook::write);
}

}

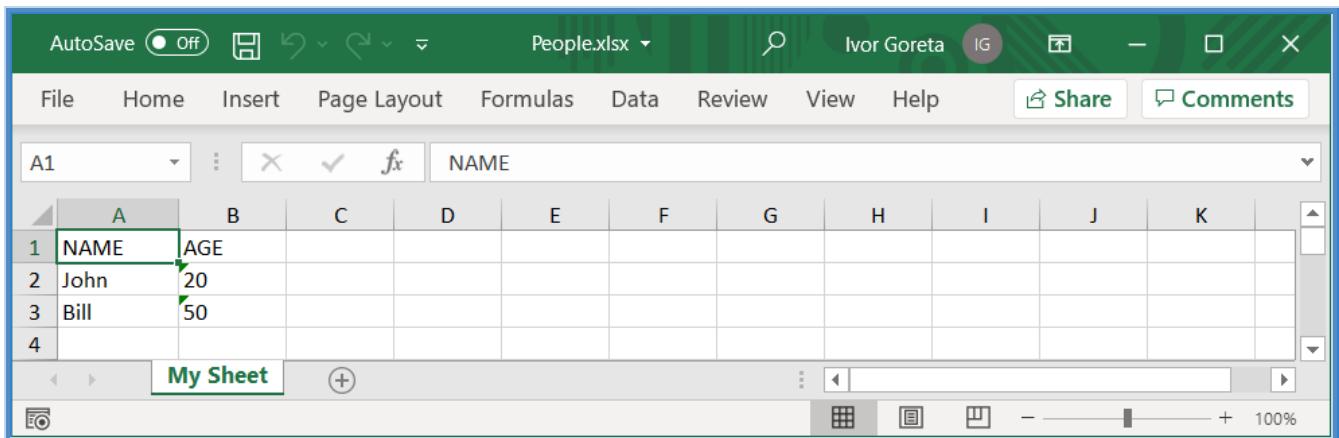
```

## Results

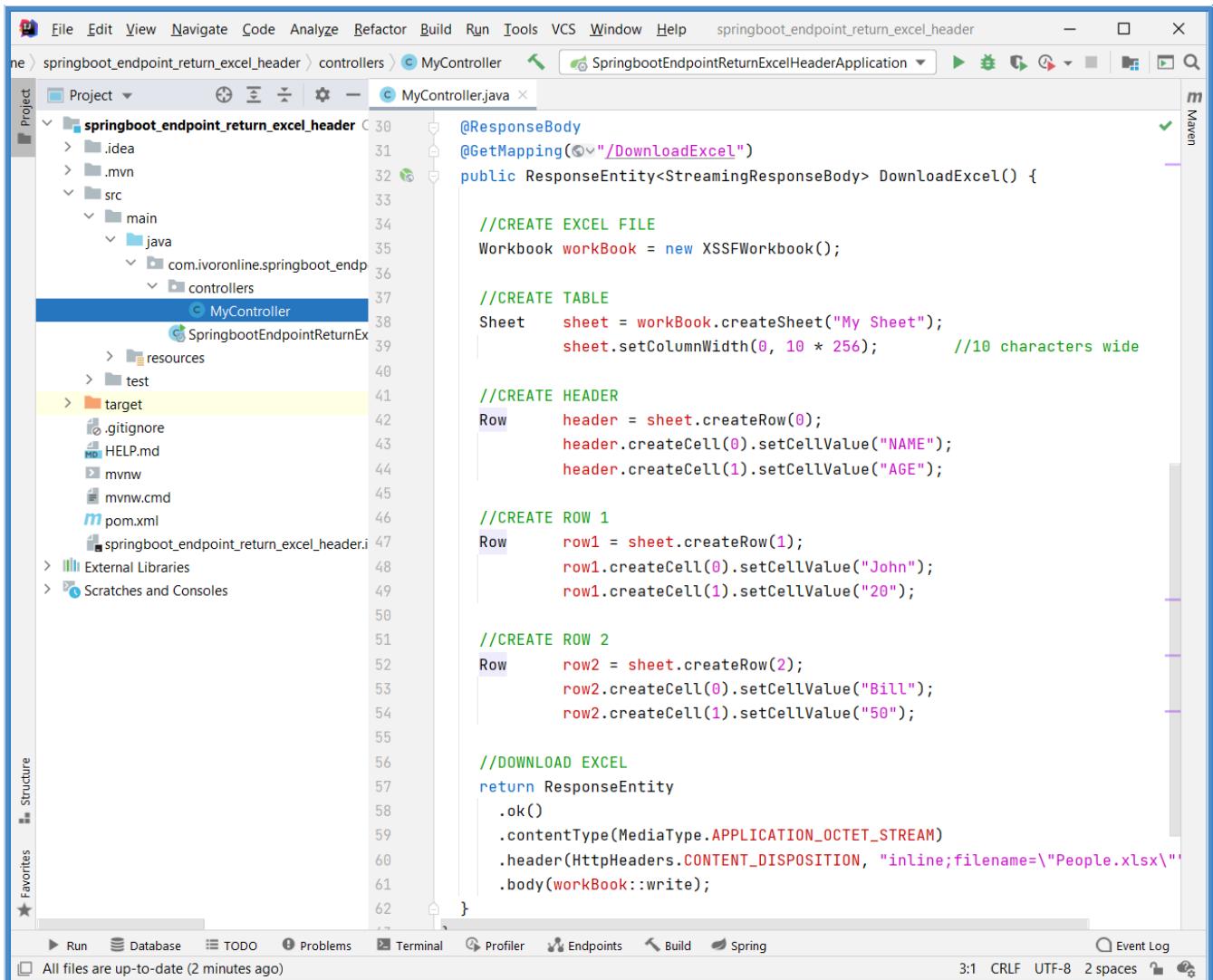
<http://localhost:8080/GetExcel>



Test.xlsx



Application Structure



*pom.xml*

```
<dependencies>

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

    <dependency>
        <groupId>org.apache.poi</groupId>
        <artifactId>poi</artifactId>
        <version>3.15</version>
    </dependency>

    <dependency>
        <groupId>org.apache.poi</groupId>
        <artifactId>poi-ooxml</artifactId>
        <version>3.15</version>
    </dependency>

</dependencies>
```

## 2.5 Entity

### Info

---

- Following tutorials show how to work with Entities.

## 2.5.1 Properties - Public

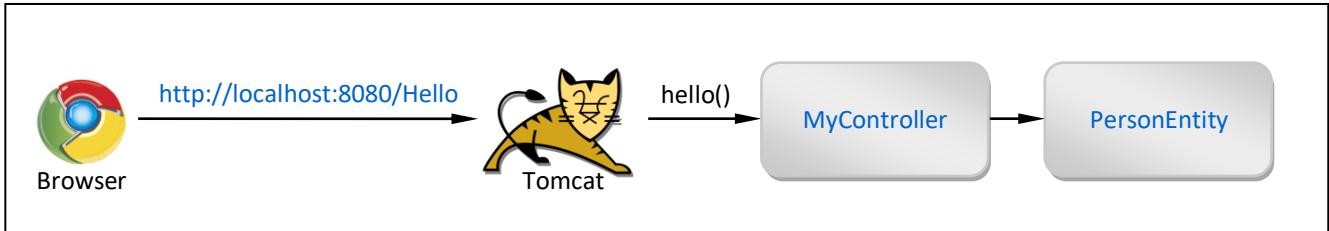
### Info

[G]

- This tutorial shows how to create Entity that only has **public** Properties.
- This way Entity is made very simple since there is no need for getters and setters.
- Instead we can directly access Properties when we want to read them or set their values.

*Application Schema*

[Results]



*Spring Boot Starters*

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables @RequestMapping. Includes Tomcat Server.

## Procedure

---

- Create Project: entity\_application (add Spring Boot Starters from the table)
- Create Package: entities (inside main package)
  - Create Class: PersonEntity.java (inside package entities)
- Create Package: controllers (inside main package)
  - Create Class: MyController.java (inside package controllers)

PersonEntity.java

```
package com.ivoronline.entity_application.entities;

public class PersonEntity {
    public Long id;
    public String name;
    public Integer age;
}
```

MyController.java

```
package com.ivoronline.entity_application.controllers;

import com.ivoronline.entity_application.entities.PersonEntity;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RequestMapping;

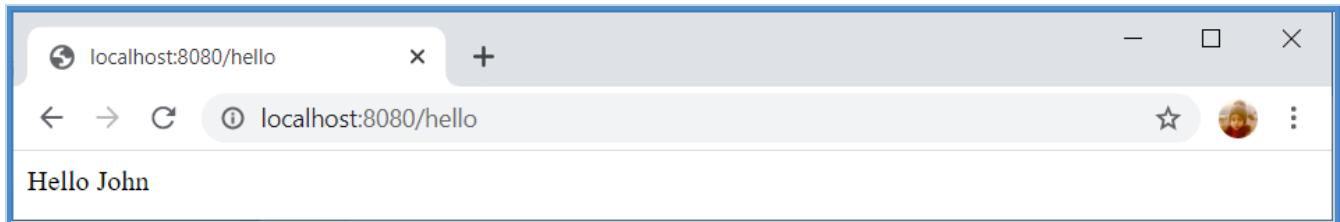
@Controller
public class MyController {

    PersonEntity personEntity = new PersonEntity();

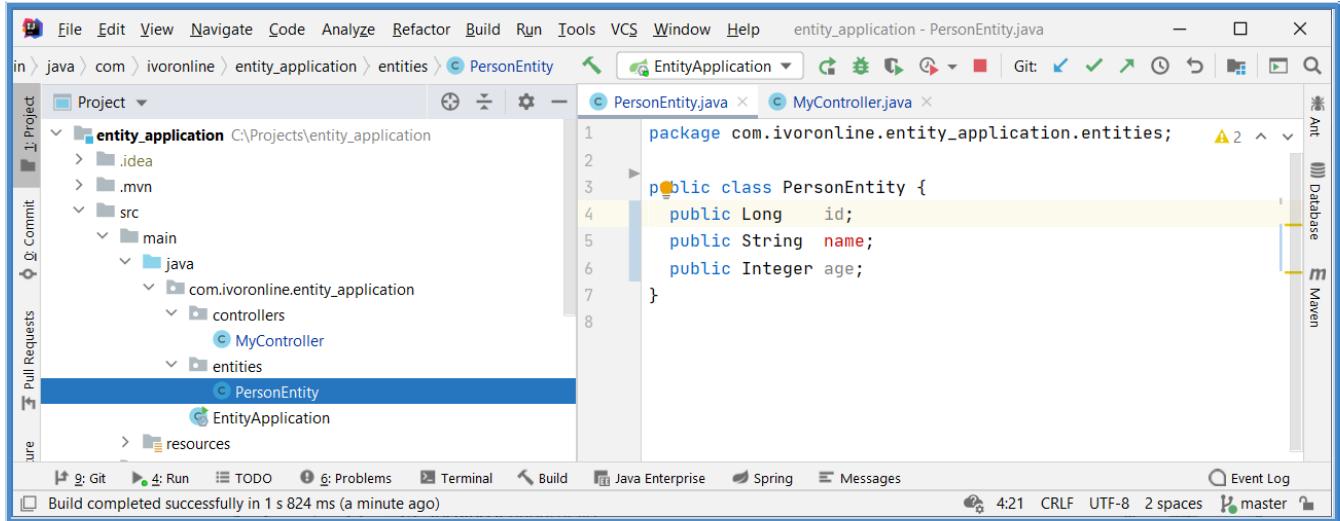
    @ResponseBody
    @RequestMapping("/Hello")
    public String hello() {
        personEntity.name = "John";
        String name = personEntity.name;
        return "Hello " + name;
    }
}
```

## Results

<http://localhost:8080>Hello>



## Application Structure



## pom.xml

```
<dependencies>

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

</dependencies>
```

## 2.5.2 Properties - Private - Getters & Setters

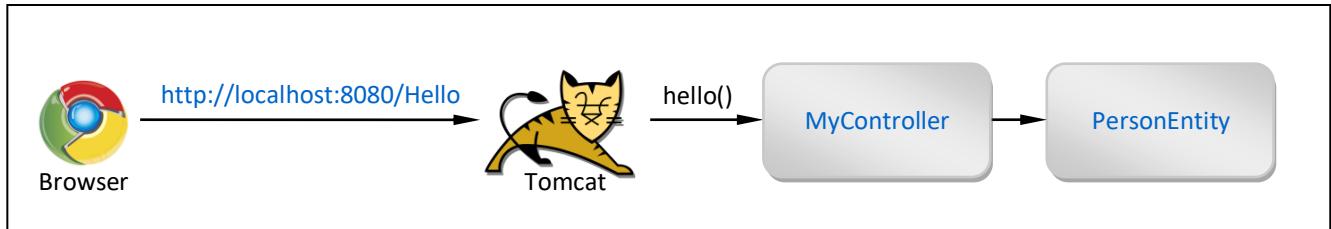
### Info

[G]

- This tutorial shows how to create Entity that has **private** Properties.
- In order to access them we need to create additional **setter** and **getter** Methods.
- You can combine Constructor and Setters where
  - Constructor** is used to set **Required** or **Immutable** Properties
  - Setters** are used to set **Optional** or **Mutable** Properties

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables: @RequestMapping, Tomcat Server

## Procedure

---

- Create Project: entity\_entity (add Spring Boot Starters from the table)
- Create Package: entities (inside main package)
  - Create Class: PersonEntity.java (inside package entities)
- Create Package: controllers (inside main package)
  - Create Class: MyController.java (inside package controllers)

### PersonEntity.java

```
package com.ivoronline.entity_entity.entities;

public class PersonEntity {

    //PROPERTIES
    private Long id;
    private String name;
    private Integer age;

    //SETTERS
    public void setId (Long id) { this.id = id; }
    public void setName(String name) { this.name = name; }
    public void setAge (Integer age) { this.age = age; }

    //GETTERS
    public Long getId () { return id; }
    public String getName() { return name; }
    public Integer getAge () { return age; }

}
```

### MyController.java

```
package com.ivoronline.entity_entity.controllers;

import com.ivoronline.entity_entity.entities.PersonEntity;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class MyController {

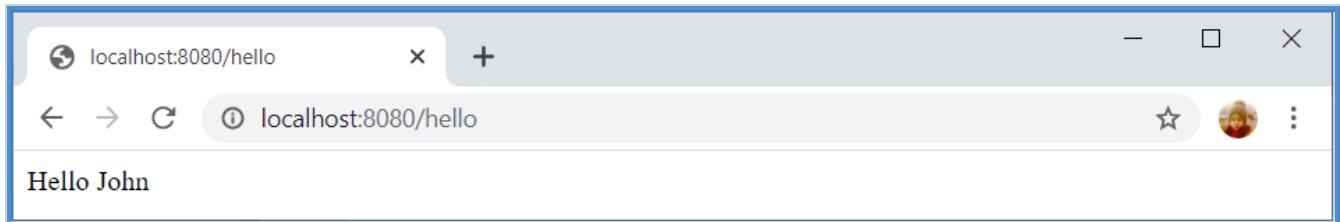
    PersonEntity personEntity = new PersonEntity();

    @ResponseBody
    @RequestMapping("/Hello")
    public String hello() {
        personEntity.setName("John");
        String name = personEntity.getName();
        return "Hello " + name;
    }

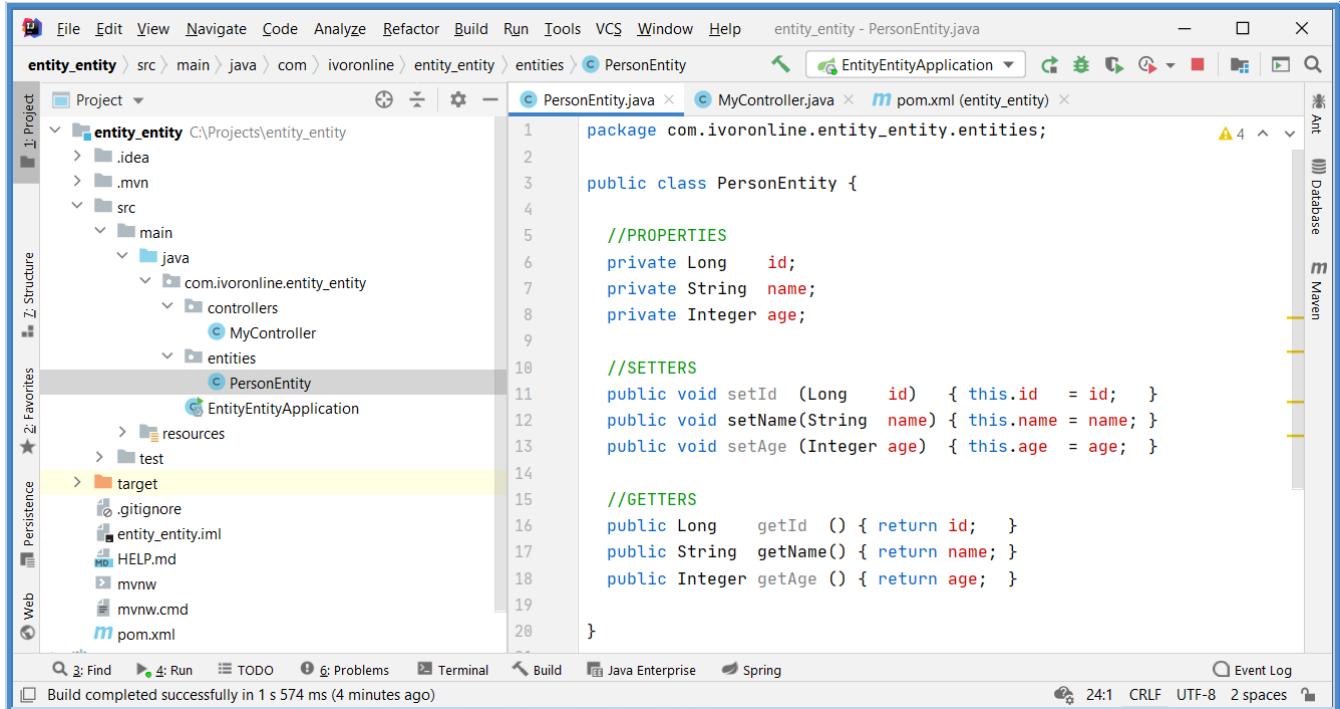
}
```

## Results

<http://localhost:8080>Hello>



## Application Structure



## pom.xml

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

## 2.5.3 Properties - Private - Getters & Constructor

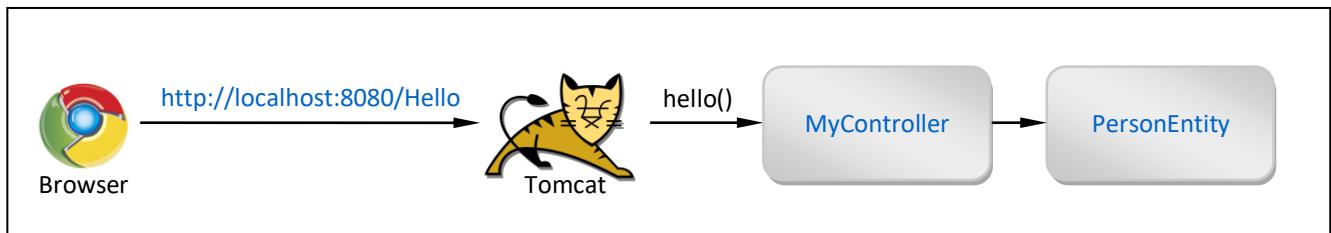
### Info

[G]

- This tutorial shows how to create Entity that has **private** Properties but uses **Constructor** instead of **Setters**.
- You can combine Constructor and Setters where
  - Constructor** is used to set **Required** or **Immutable** Properties
  - Setters** are used to set **Optional** or **Mutable** Properties

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables: @RequestMapping, Tomcat Server

## Procedure

---

- Create Project: `springboot_entity_constructor` (add Spring Boot Starters from the table)
- Create Package: `entities` (inside main package)
  - Create Class: `PersonEntity.java` (inside package entities)
- Create Package: `controllers` (inside main package)
  - Create Class: `MyController.java` (inside package controllers)

*PersonEntity.java*

```
package com.ivoronline.springboot_entity_constructor.entities;

public class PersonEntity {

    //PROPERTIES
    private long id;
    private String name;
    private Integer age;

    //CONSTRUCTOR
    public PersonEntity(long id, String name, Integer age) {
        this.id = id;
        this.name = name;
        this.age = age;
    }

    //GETTERS
    public long getId () { return id; }
    public String getName() { return name; }
    public Integer getAge () { return age; }

}
```

*MyController.java*

```
package com.ivoronline.springboot_entity_constructor.controllers;

import com.ivoronline.springboot_entity_constructor.entities.PersonEntity;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RequestMapping;

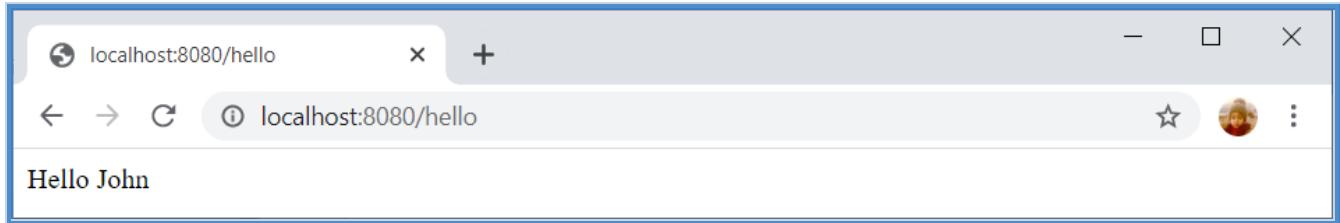
@Controller
public class MyController {

    @ResponseBody
    @RequestMapping("/Hello")
    public String hello() {
        PersonEntity personEntity = new PersonEntity(1, "John", 20);
        return "Hello " + personEntity.getName();
    }

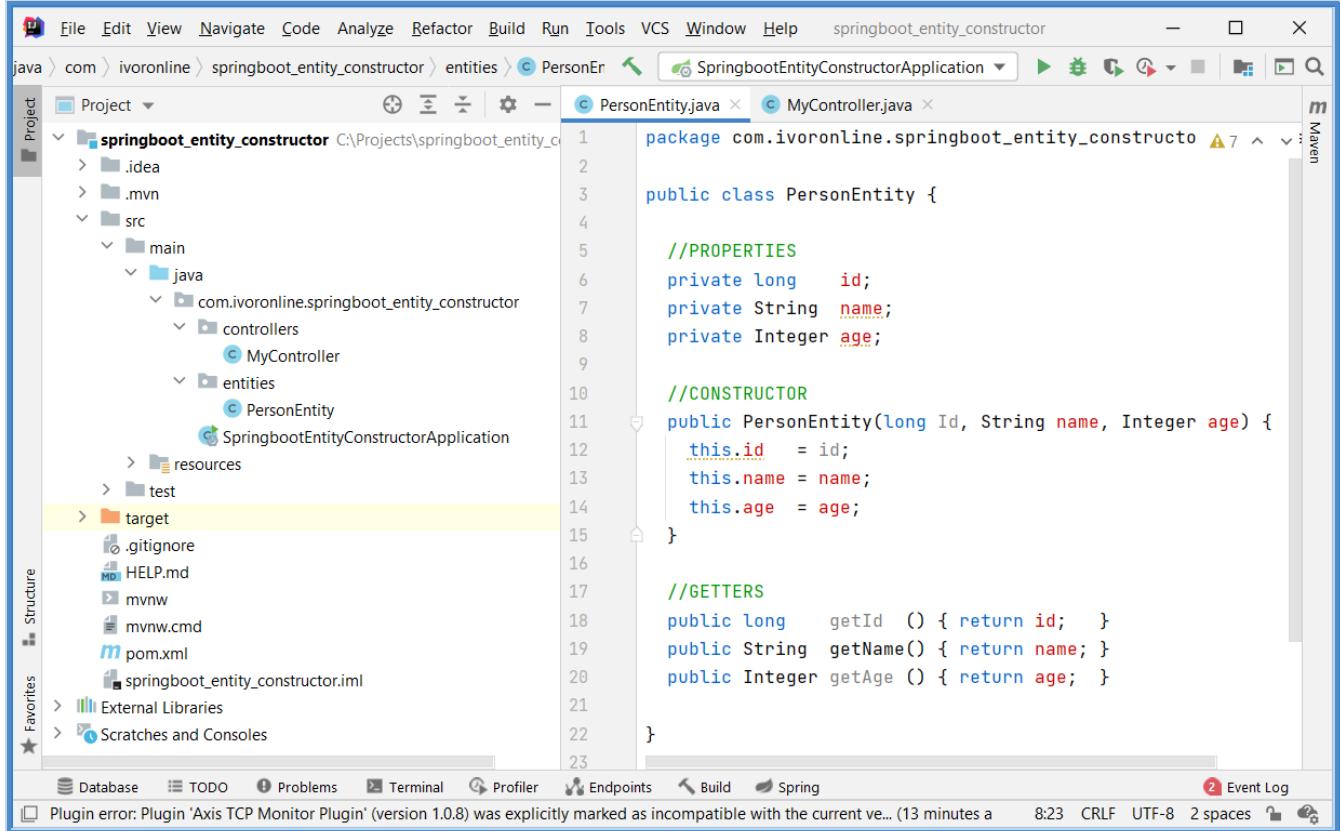
}
```

## Results

<http://localhost:8080>Hello>



## Application Structure



## pom.xml

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

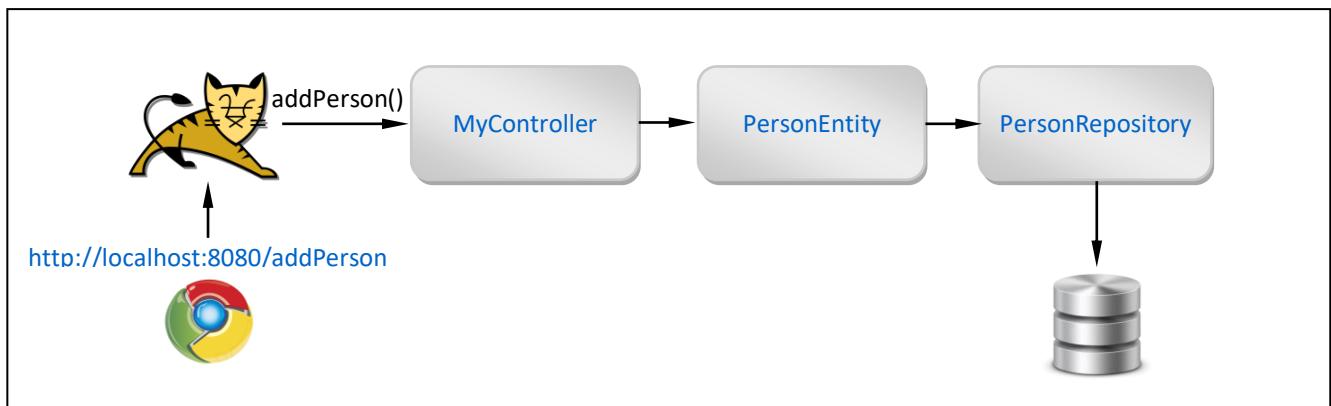
## 2.5.4 @Entity

### Info

- In this tutorial we will use JPA's **@Entity** Annotation to indicate that instances of this Class should be stored in DB. Without this Hibernate will not be able to pick up this Class and do its magic with it by automatically creating Tables, Columns and then taking care of persistent its instances in selected DB.  
Classes annotated with **@Entity** must have Property that is declared as Primary Key by using **@Id** Annotation.  
As reminder: **JPA API** is interface that defines **@Entity** and **Hibernate API** is actual implementation that stores Entity in DB.
- For each **@Entity** Class Hibernate creates DB Table
  - that has the same name as the name of the Class
  - that has Columns with the same names as Class Properties
- When you include JPA Spring Boot Starter you also need to select some DB Spring Boot Starter: H2, MySQL, PostgreSQL. Otherwise Spring Boot will not be able to initialize JPA and your Application would not run.  
This tutorial includes H2 in-memory DB for which no additional installation and configuration is needed.
- JPA Annotations can only be used on SQL DBs (H2, MySQL, Oracle) and can't be used on NoSQL DBs (MongoDB).

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables <b>@Controller</b> , <b>@RequestMapping</b> and Tomcat Server
SQL	Spring Data JPA	Enables <b>@Entity</b> and <b>@Id</b>
SQL	H2 Database	Enables in-memory H2 DB

## Procedure

- Create Project: entity\_entity (add Spring Boot Starters from the table)
- Edit: application.properties (specify H2 DB name & enable H2 Web Console)
- Create Package: entities (inside main package)
  - Create Class: PersonEntity.java (inside package entities)
- Create Package: repositories (inside main package)
  - Create Interface: PersonRepository.java (inside package repositories)
- Create Package: controllers (inside main package)
  - Create Class: MyController.java (inside package controllers)

*application.properties*

```
# H2 DATABASE
spring.datasource.url      = jdbc:h2:mem:testdb
spring.h2.console.enabled = true
```

*PersonEntity.java*

```
package com.ivoronline.entity_entity.entities;

import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class PersonEntity {

    @Id
    public Integer id;
    public String name;
    public Integer age;

}
```

*PersonRepository.java*

```
package com.ivoronline.entity_entity.repositories;

import com.ivoronline.entity_entity.entities.PersonEntity;
import org.springframework.data.repository.CrudRepository;

public interface PersonRepository extends CrudRepository<PersonEntity, Integer> { }
```

### MyController.java

```
package com.ivorononline.entity_entity.controllers;

import com.ivorononline.entity_entity.entities.PersonEntity;
import com.ivorononline.entity_entity.repositories.PersonRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class MyController {

    @Autowired
    PersonRepository personRepository;

    @ResponseBody
    @RequestMapping("/addPerson")
    public String addPerson() {

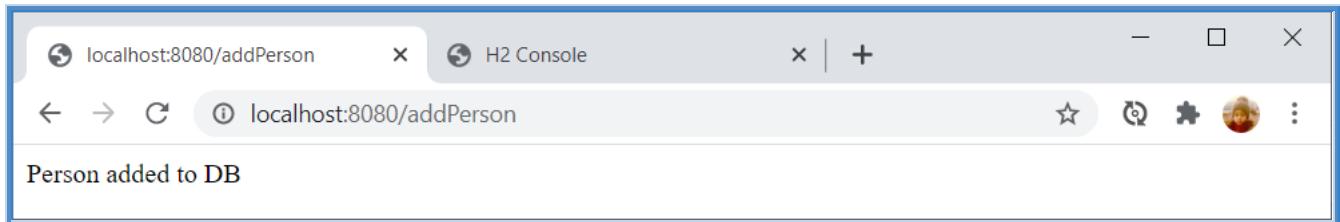
        //CREATE ENTITY OBJECT
        PersonEntity personEntity      = new PersonEntity();
        personEntity.id     = 1;
        personEntity.name  = "John";
        personEntity.age   = 20;

        //STORE ENTITY OBJECT INTO DB
        personRepository.save(personEntity);

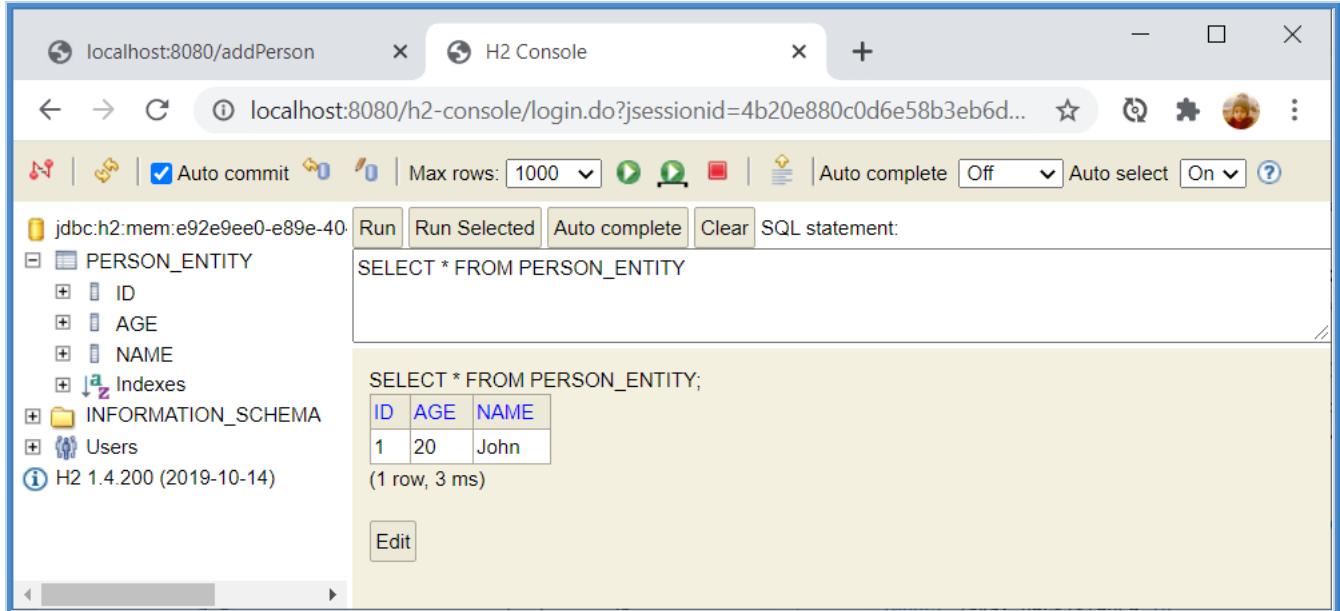
        //RETURN SOMETHING TO BROWSER
        return personEntity.name + " was stored into DB";
    }
}
```

## Results

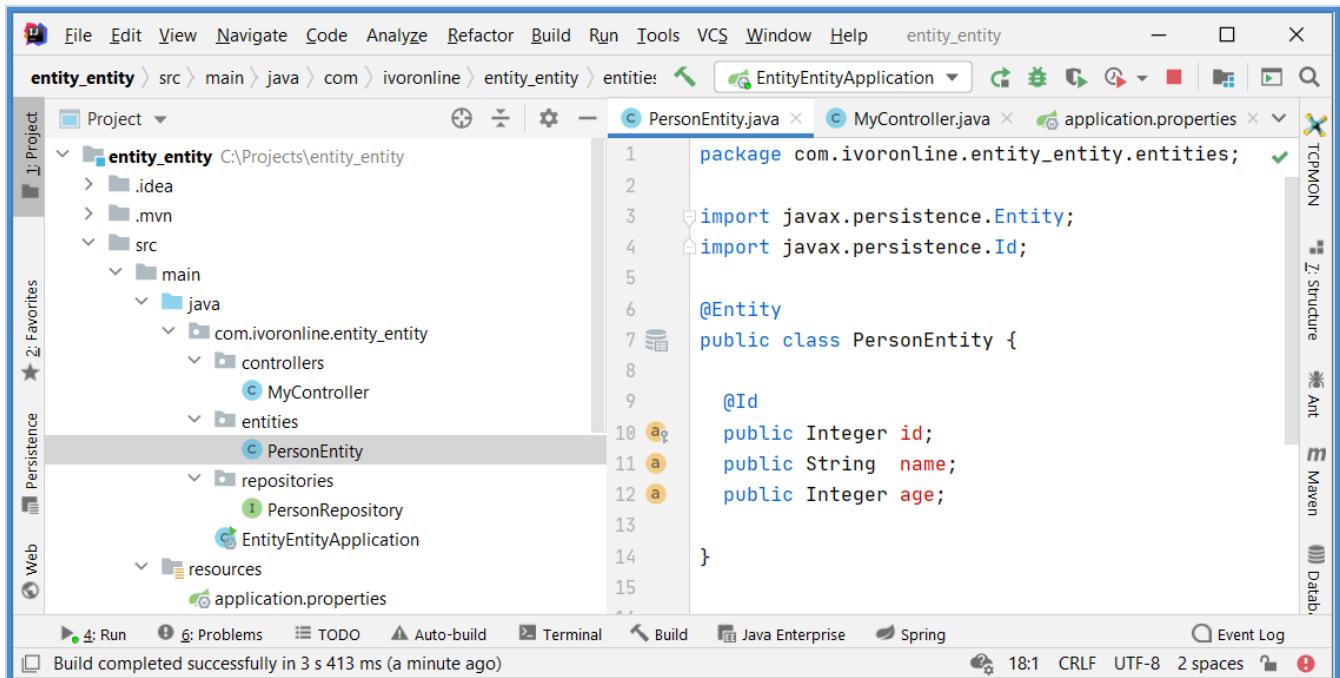
<http://localhost:8080/addPerson>



<http://localhost:8080/h2-console> - Connect - PERSON\_ENTITY - Run (H2 Console)



## Application Structure



```
<dependencies>

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

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

    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
    </dependency>

</dependencies>
```

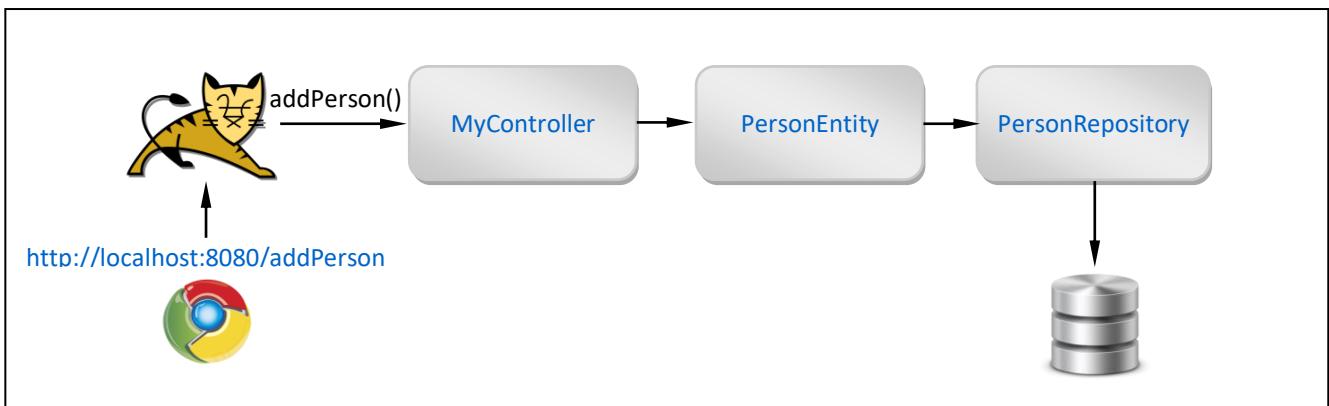
## 2.5.5 @Id

### Info

- In this tutorial we will use JPA's `@Id` Annotation to indicate that Property represents Primary Key.
- Hibernate needs this information to be able to store Entities as Records in DB.
- You can manually set id when creating Entities but in this example ids are automatically generated by DB Sequence.
- Compared to previous example we will only
  - add following line to `PersonEntity.java` `@GeneratedValue(strategy = GenerationType.IDENTITY)`
  - remove following line from `MyController.java` `personEntity.id = 1;`
- JPA Annotations can only be used on SQL DBs (H2, MySQL, Oracle) and can't be used on NoSQL DBs (MongoDB).

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables <code>@Controller</code> , <code>@RequestMapping</code> and Tomcat Server
SQL	Spring Data JPA	Enables <code>@Entity</code> and <code>@Id</code>
SQL	H2 Database	Enables in-memory H2 DB

## Procedure

- Create Project: entity\_entity (add Spring Boot Starters from the table)
- Edit File: application.properties (specify H2 DB name & enable H2 Web Console)
- Create Package: entities (inside main package)
  - Create Class: PersonEntity.java (inside package entities)
- Create Package: repositories (inside main package)
  - Create Interface: PersonRepository.java (inside package repositories)
- Create Package: controllers (inside main package)
  - Create Class: MyController.java (inside package controllers)

*application.properties*

```
# H2 DATABASE
spring.datasource.url      = jdbc:h2:mem:testdb
spring.h2.console.enabled = true
```

*PersonEntity.java*

```
package com.ivoronline.springboot.entity_annotation_id.entities;

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

@Entity
public class PersonEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    public Integer id;
    public String name;
    public Integer age;

}
```

*PersonRepository.java*

```
package com.ivoronline.springboot.entity_annotation_id.repositories;

import com.ivoronline.entity_entity.entities.PersonEntity;
import org.springframework.data.repository.CrudRepository;

public interface PersonRepository extends CrudRepository<PersonEntity, Integer> { }
```

### *MyController.java*

```
package com.ivorononline.springboot.entity_annotation_id.controllers;

import com.ivorononline.entity_entity.entities.PersonEntity;
import com.ivorononline.entity_entity.repositories.PersonRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class MyController {

    @Autowired
    PersonRepository personRepository;

    @ResponseBody
    @RequestMapping("/addPerson")
    public String addPerson() {

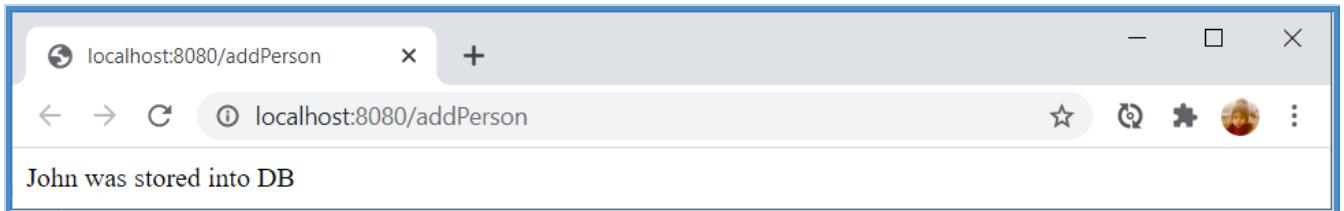
        //CREATE ENTITY OBJECT
        PersonEntity personEntity      = new PersonEntity();
        personEntity.name = "John";
        personEntity.age = 20;

        //STORE ENTITY OBJECT INTO DB
        personRepository.save(personEntity);

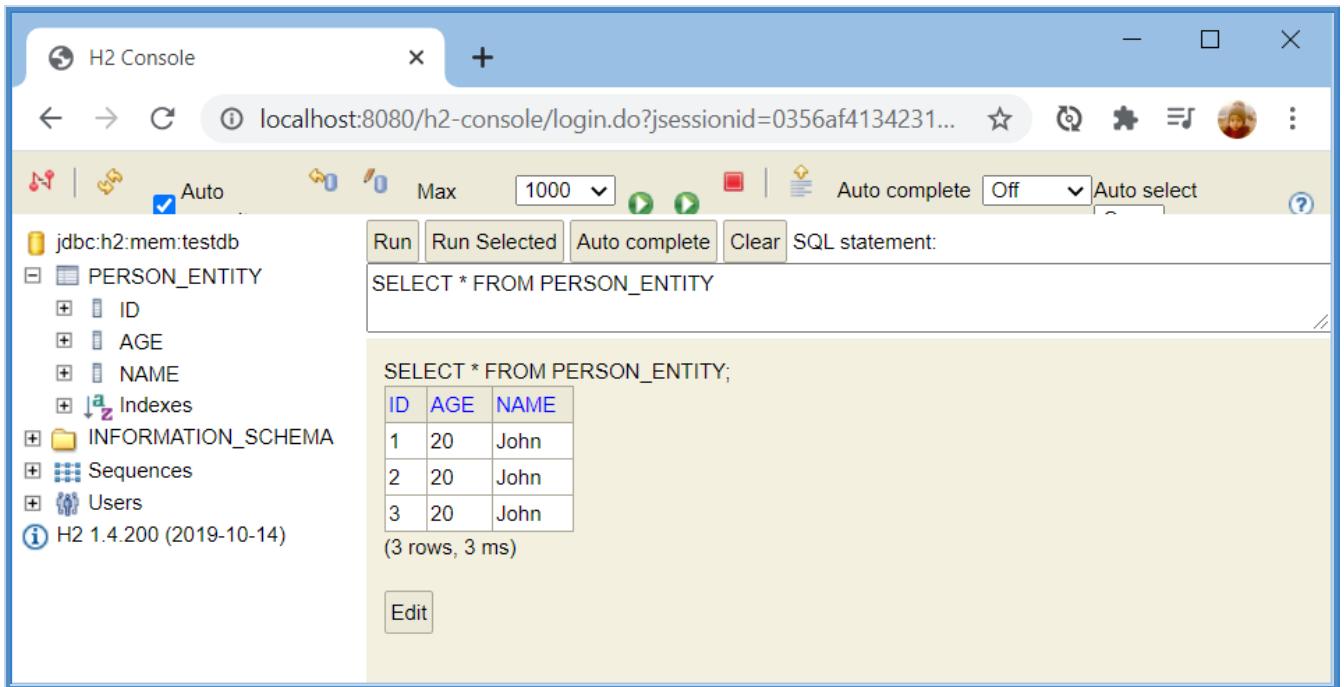
        //RETURN SOMETHING TO BROWSER
        return personEntity.name + " was stored into DB";
    }
}
```

## Results

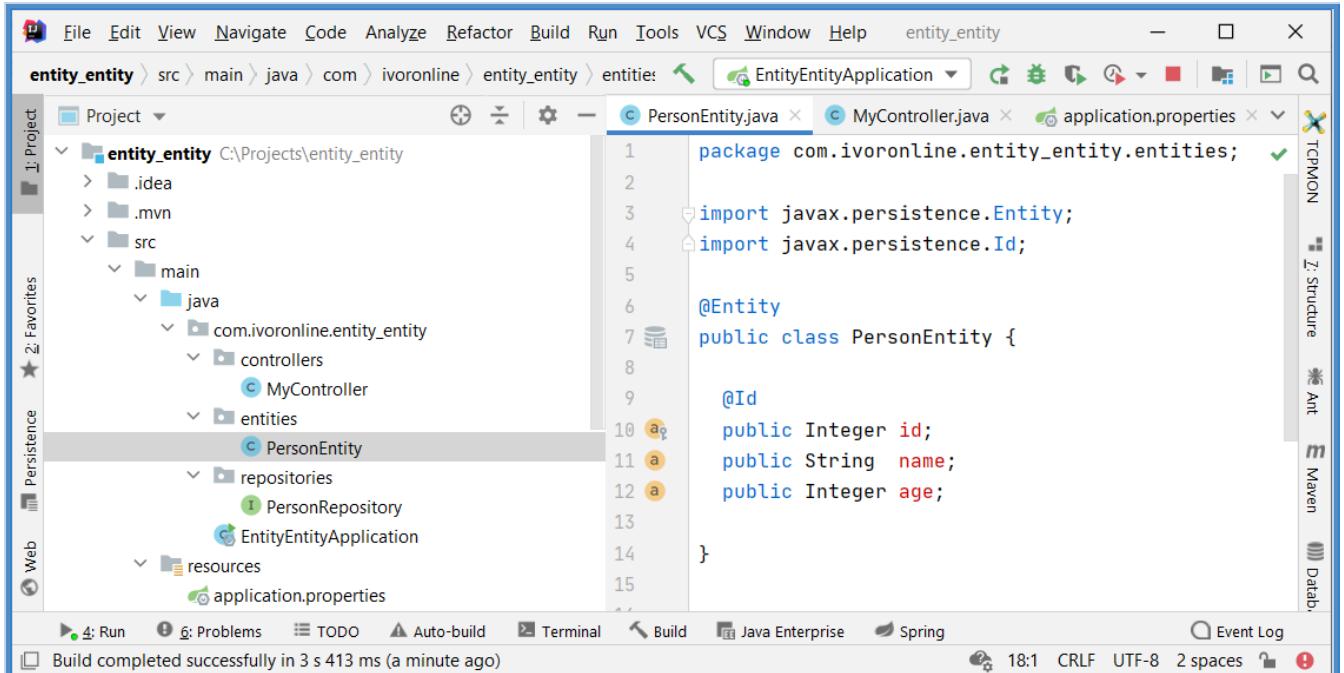
<http://localhost:8080/addPerson>



Open H2 Console: <http://localhost:8080/h2-console> - Connect - PERSON\_ENTITY - Run



## Application Structure



```
<dependencies>

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

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

    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
    </dependency>

</dependencies>
```

## 2.5.6 @IdClass

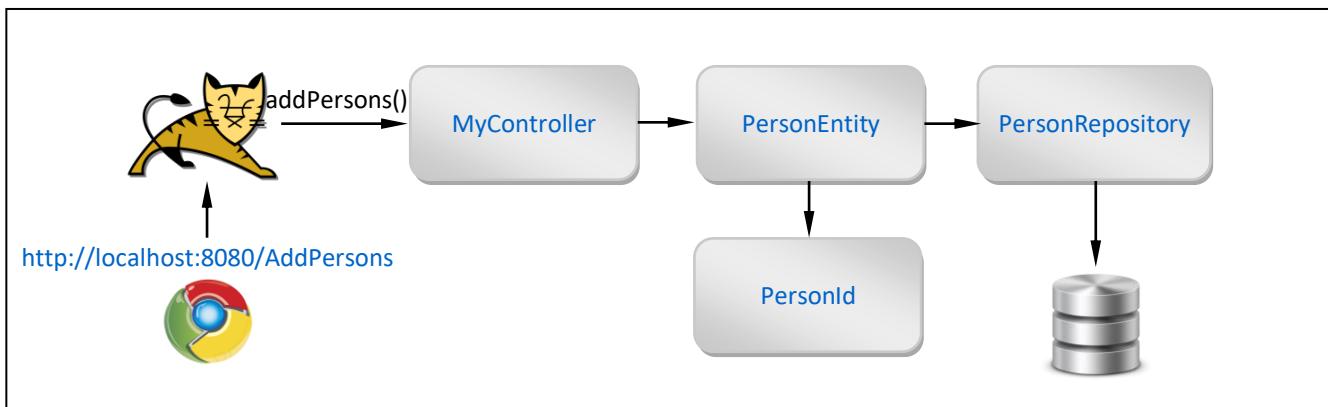
### Info

[G] [R]

- In this tutorial we will use JPA's `@IdClass` Annotation to create Composite Primary Key from `name` and `age`.
- `@IdClass` keeps all Properties at the same level so that we can use `@RequestBody` to map JSON Properties from Request.

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables <code>@Controller</code> , <code>@RequestMapping</code> and Tomcat Server
SQL	Spring Data JPA	Enables <code>@Entity</code> and <code>@Id</code>
SQL	H2 Database	Enables in-memory H2 DB

### Overview

- We will create `PersonId` Class to hold Properties from which Composite Primary Key will be created.
- Inside `PersonEntity` we will use `@IdClass(PersonId.class)` to specify that `PersonId` Class should be used as Id.
- inside `PersonRepository` we also need to specify that `PersonId` Class should be used as Id.

### PersonId.java

```
public class PersonId implements Serializable {  
    private String name;  
    private Integer age;
```

### PersonEntity.java

```
@Entity  
@IdClass(PersonId.class)  
public class PersonEntity {  
    @Id public String name;  
    @Id public Integer age;
```

### PersonRepository.java

```
public interface PersonRepository extends CrudRepository<PersonEntity, PersonId> { }
```

## Procedure

- Create Project: entity\_entity (add Spring Boot Starters from the table)
- Edit File: application.properties (specify H2 DB name & enable H2 Web Console)
- Create Package: entities (inside main package)
  - Create Class: PersonId.java (inside package entities)
  - Create Class: PersonEntity.java (inside package entities)
- Create Package: repositories (inside main package)
  - Create Interface: PersonRepository.java (inside package repositories)
- Create Package: controllers (inside main package)
  - Create Class: MyController.java (inside package controllers)

### application.properties

```
# H2 DATABASE
spring.datasource.url      = jdbc:h2:mem:testdb
spring.h2.console.enabled = true
```

### PersonId.java

```
package com.ivoronline.springboot_primarykey_idclass.entities;

import java.io.Serializable;

public class PersonId implements Serializable {

    //COMPOSITE PRIMARY KEY
    private String name;
    private Integer age;

    //REQUIRED NO ARGS CONSTRUCTOR
    public PersonId() {}

    //CONSTRUCTOR FOR findById(BookId)
    public PersonId(String name, Integer age) {
        this.name = name;
        this.age = age;
    }

}
```

### PersonEntity.java

```
package com.ivoronline.springboot_primarykey_idclass.entities;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.IdClass;

@Entity
@IdClass(PersonId.class)
public class PersonEntity {

    //COMPOSITE PRIMARY KEY
    @Id public String name;
    @Id public Integer age;

    //OTHER PROPERTIES
    public String book;

}
```

### PersonRepository.java

```
package com.ivorononline.springboot_primarykey_idclass.repositories;

import com.ivorononline.springboot_primarykey_idclass.entities.PersonEntity;
import com.ivorononline.springboot_primarykey_idclass.entities.PersonId;
import org.springframework.data.repository.CrudRepository;

public interface PersonRepository extends CrudRepository<PersonEntity, PersonId> { }
```

### MyController.java

```
package com.ivorononline.springboot_primarykey_idclass.controllers;

import com.ivorononline.springboot_primarykey_idclass.entities.PersonEntity;
import com.ivorononline.springboot_primarykey_idclass.entities.PersonId;
import com.ivorononline.springboot_primarykey_idclass.repositories.PersonRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class MyController {

    @Autowired
    PersonRepository personRepository;

    //=====
    // ADD PERSONS
    //=====

    @ResponseBody
    @RequestMapping("/AddPersons")
    public String addPersons() {

        //CREATE BOOK1
        PersonEntity personEntity1 = new PersonEntity();
        personEntity1.name = "John";
        personEntity1.age = 20;
        personEntity1.book = "Book about dogs";

        //CREATE BOOK2
        PersonEntity personEntity2 = new PersonEntity();
        personEntity2.name = "John";
        personEntity2.age = 50;
        personEntity2.book = "Book about cats";

        //STORE BOOKS
        personRepository.save(personEntity1);
        personRepository.save(personEntity2);

        //RETURN SOMETHING TO BROWSER
        return "Persons added to DB";
    }

    //=====
    // GET PERSON
    //=====

    @ResponseBody
    @RequestMapping("/GetPerson")
    public PersonEntity getPerson() {
        PersonEntity personEntity = personRepository.findById(new PersonId("John", 20)).get();
        return personEntity;
    }
}
```

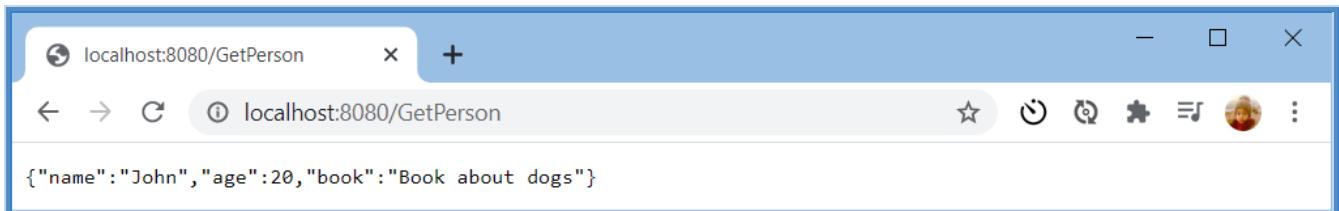
```
 }  
 }
```

## Results

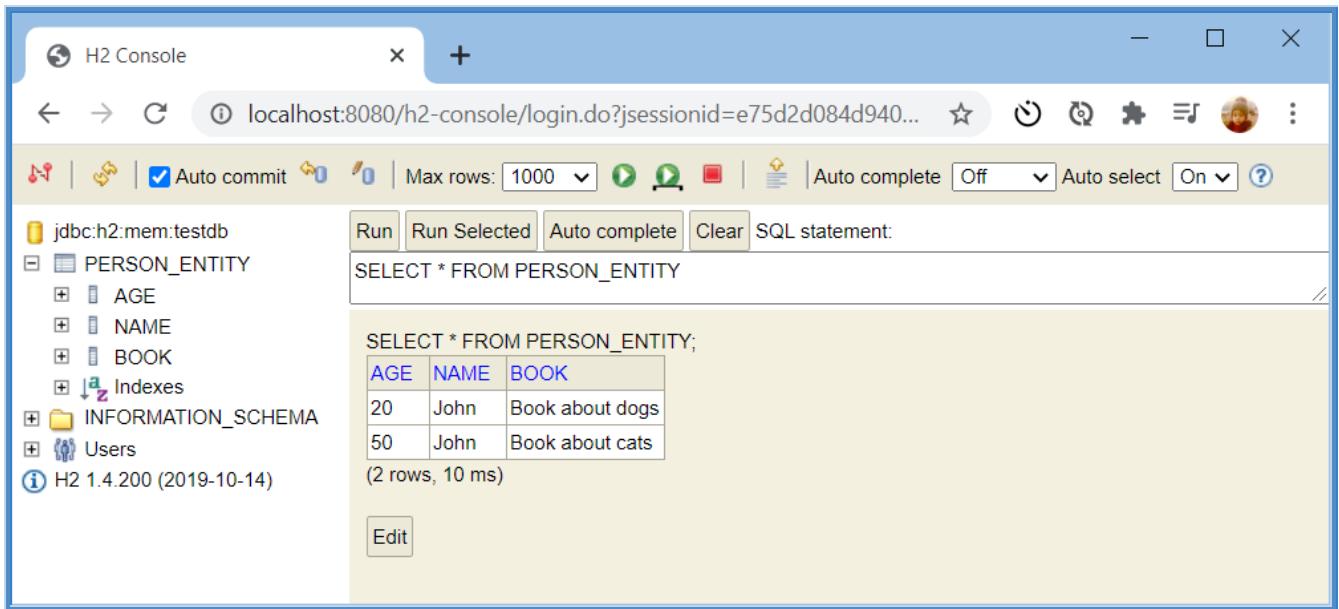
<http://localhost:8080/AddPersons>



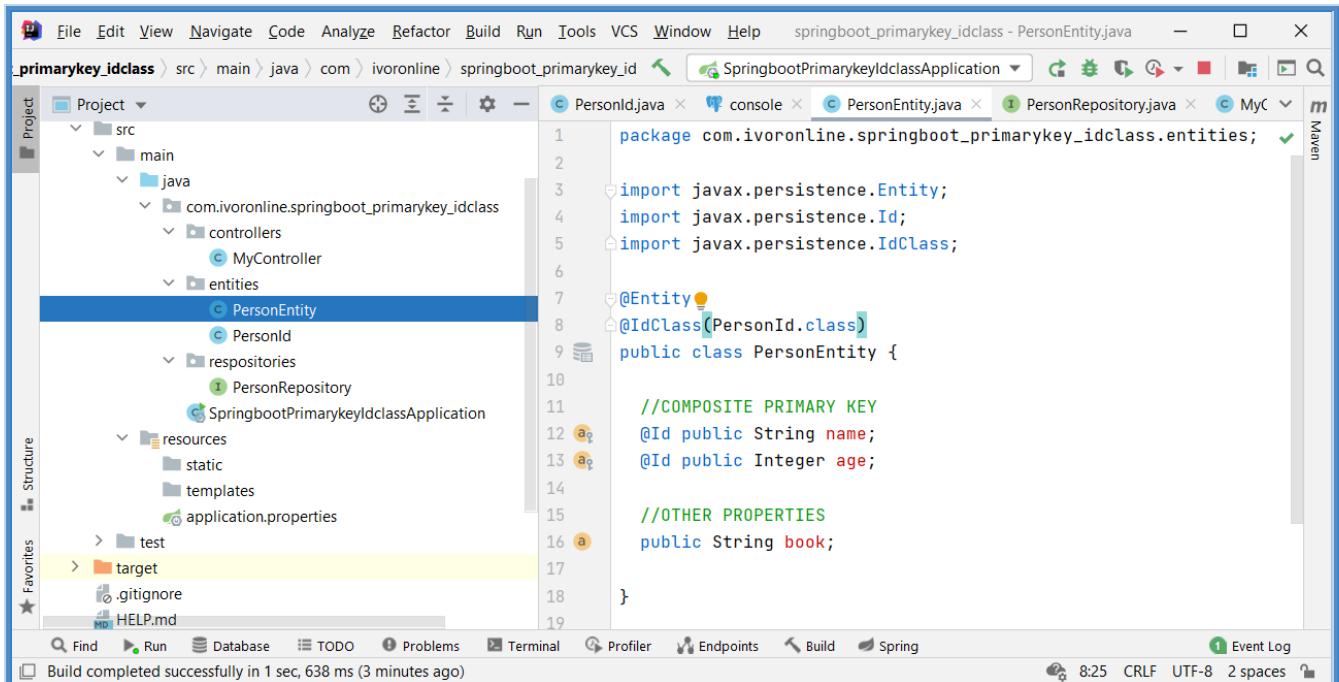
<http://localhost:8080/GetPerson>



Open H2 Console: <http://localhost:8080/h2-console> - Connect - PERSON\_ENTITY - Run



## Application Structure



```
<dependencies>

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

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

    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
    </dependency>

</dependencies>
```

## 2.5.7 @EmbeddedId

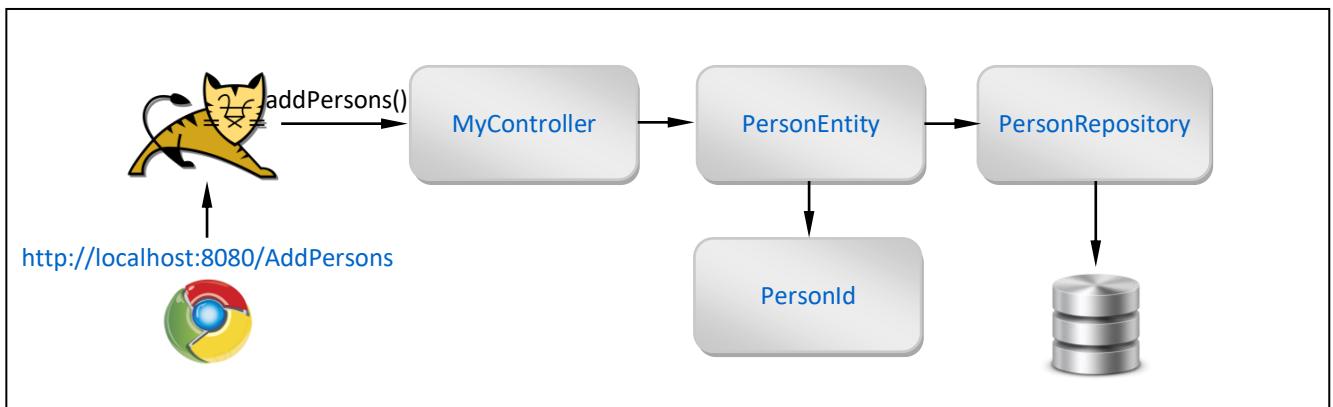
### Info

[G] [R]

- In this tutorial we will use JPA's **@EmbeddedId** Annotation to create Composite Primary Key from **name** and **age**.
- @EmbeddedId** hides Primary Key Properties from the **Entity** so we can't use **@RequestBody** to map JSON Properties.

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables <code>@Controller</code> , <code>@RequestMapping</code> and Tomcat Server
SQL	Spring Data JPA	Enables <code>@Entity</code> and <code>@Id</code>
SQL	H2 Database	Enables in-memory H2 DB

### Overview

- We will create `PersonId` Class to hold Properties from which Composite Primary Key will be created.
- Inside `PersonEntity` we will use `@EmbeddedId` to specify that `PersonId` Class should be used as Id.
- inside `PersonRepository` we also need to specify that `PersonId` Class should be used as Id.

### PersonId.java

```
@Embeddable
public class PersonId implements Serializable {
    private String name;
    private Integer age;
```

### PersonEntity.java

```
@EmbeddedId
public PersonId personId;
```

### PersonRepository.java

```
public interface PersonRepository extends CrudRepository<PersonEntity, PersonId> { }
```

## Procedure

- Create Project: entity\_entity (add Spring Boot Starters from the table)
- Edit File: application.properties (specify H2 DB name & enable H2 Web Console)
- Create Package: entities (inside main package)
  - Create Class: PersonId.java (inside package entities)
  - Create Class: PersonEntity.java (inside package entities)
- Create Package: repositories (inside main package)
  - Create Interface: PersonRepository.java (inside package repositories)
- Create Package: controllers (inside main package)
  - Create Class: MyController.java (inside package controllers)

### application.properties

```
# H2 DATABASE
spring.datasource.url      = jdbc:h2:mem:testdb
spring.h2.console.enabled = true
```

### PersonId.java

```
package com.ivoronline.springboot_db_primarykey_embeddedid.entities;

import javax.persistence.Embeddable;
import java.io.Serializable;

@Embeddable
public class PersonId implements Serializable {

    //PROPERTIES FOR COMPOSITE PRIMARY KEY
    private String name;
    private Integer age;

    //REQUIRED NO ARGS CONSTRUCTOR
    public PersonId() {}

    //CONSTRUCTOR FOR findById(BookId)
    public PersonId(String name, Integer age) {
        this.name = name;
        this.age = age;
    }

}
```

### PersonEntity.java

```
package com.ivoronline.springboot_db_primarykey_embeddedid.entities;

import javax.persistence.EmbeddedId;
import javax.persistence.Entity;

@Entity
public class PersonEntity {

    //COMPOSITE PRIMARY KEY
    @EmbeddedId
    public PersonId personId;

    //OTHER PROPERTIES
    public String book;

}
```

### PersonRepository.java

```
package com.ivorononline.springboot_primarykey_idclass.repositories;

import com.ivorononline.springboot_primarykey_idclass.entities.PersonEntity;
import com.ivorononline.springboot_primarykey_idclass.entities.PersonId;
import org.springframework.data.repository.CrudRepository;

public interface PersonRepository extends CrudRepository<PersonEntity, PersonId> { }
```

### MyController.java

```
package com.ivorononline.springboot_db_primarykey_embeddedid.controllers;

import com.ivorononline.springboot_db_primarykey_embeddedid.entities.PersonEntity;
import com.ivorononline.springboot_db_primarykey_embeddedid.entities.PersonId;
import com.ivorononline.springboot_db_primarykey_embeddedid.repositories.PersonRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class MyController {

    @Autowired
    PersonRepository personRepository;

    //=====
    // ADD PERSONS
    //=====

    @ResponseBody
    @RequestMapping("/AddPersons")
    public String addPersons() {

        //CREATE BOOK1
        PersonEntity personEntity1 = new PersonEntity();
        personEntity1.personId = new PersonId("John", 20);
        personEntity1.book = "Book about dogs";

        //CREATE BOOK2
        PersonEntity personEntity2 = new PersonEntity();
        personEntity1.personId = new PersonId("John", 50);
        personEntity2.book = "Book about cats";

        //STORE BOOKS
        personRepository.save(personEntity1);
        personRepository.save(personEntity2);

        //RETURN SOMETHING TO BROWSER
        return "Persons added to DB";
    }

    //=====
    // GET PERSON
    //=====

    @ResponseBody
    @RequestMapping("/GetPerson")
    public PersonEntity getPerson() {
        PersonEntity personEntity = personRepository.findById(new PersonId("John", 20)).get();
        return personEntity;
    }
}
```

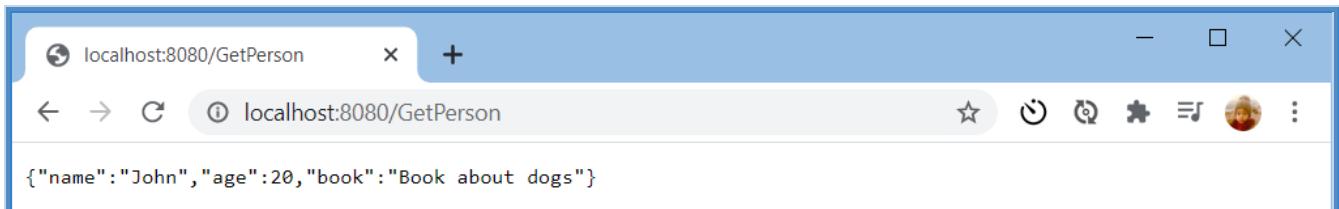


## Results

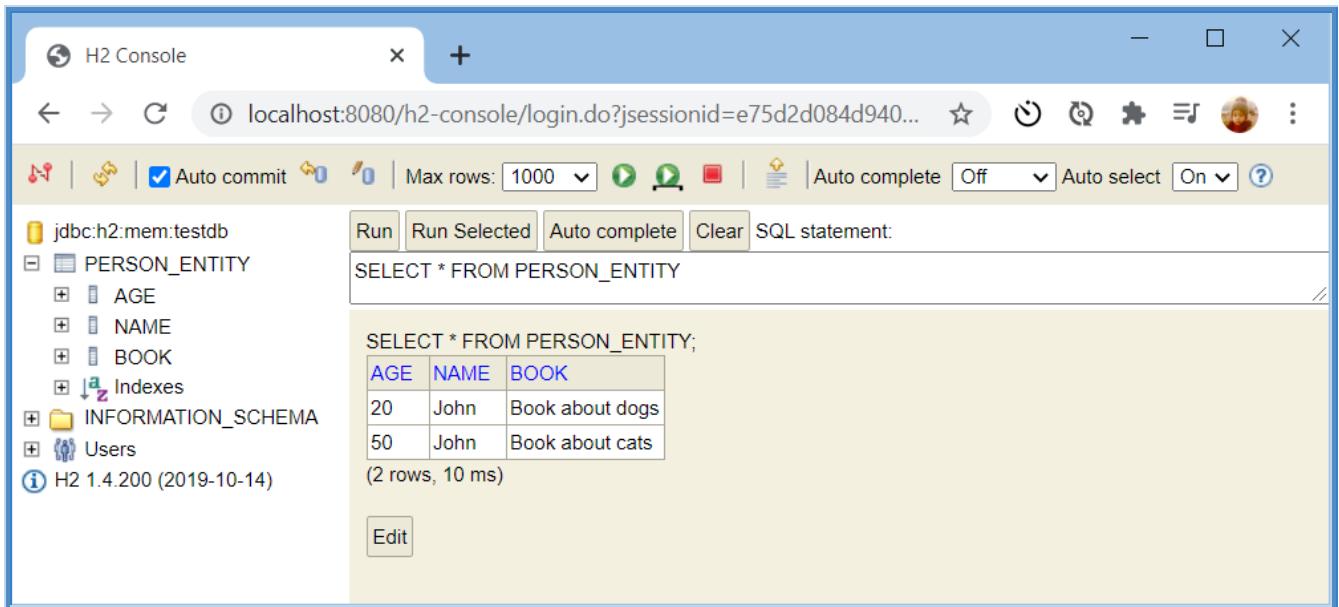
<http://localhost:8080/AddPersons>



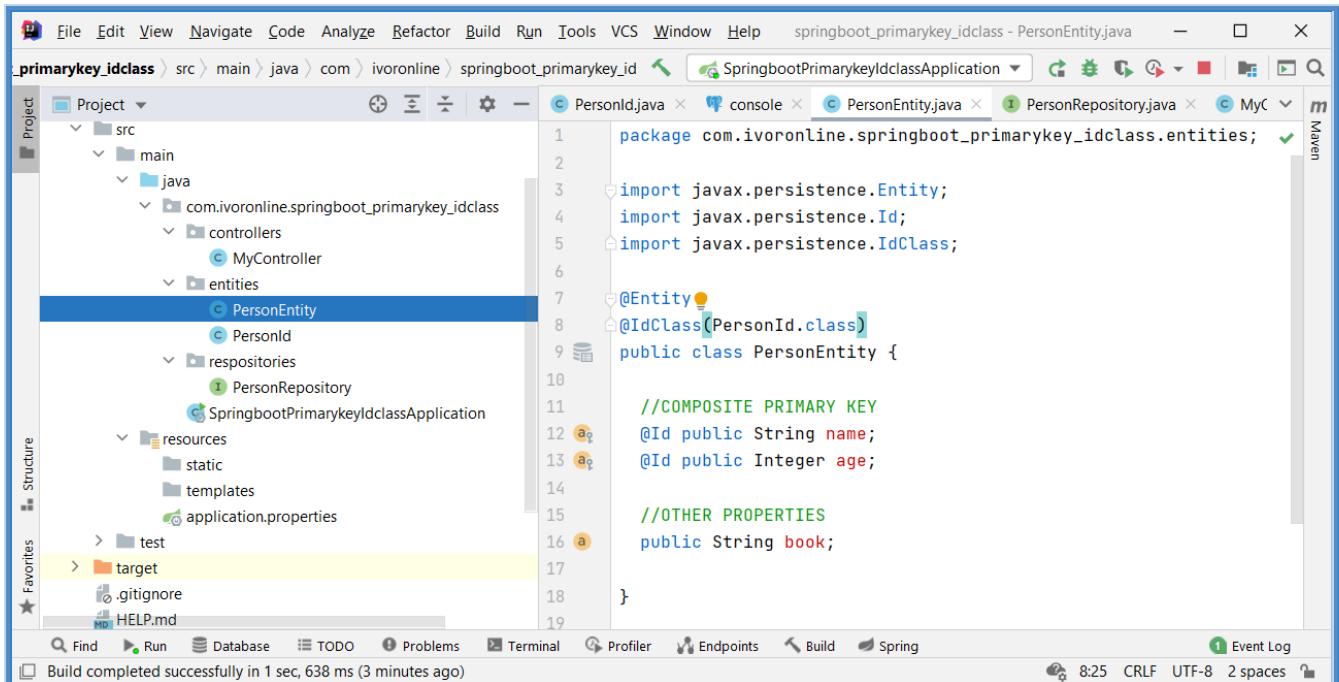
<http://localhost:8080/GetPerson>



Open H2 Console: <http://localhost:8080/h2-console> - Connect - PERSON\_ENTITY - Run



## Application Structure



```
<dependencies>

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

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

    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
    </dependency>

</dependencies>
```

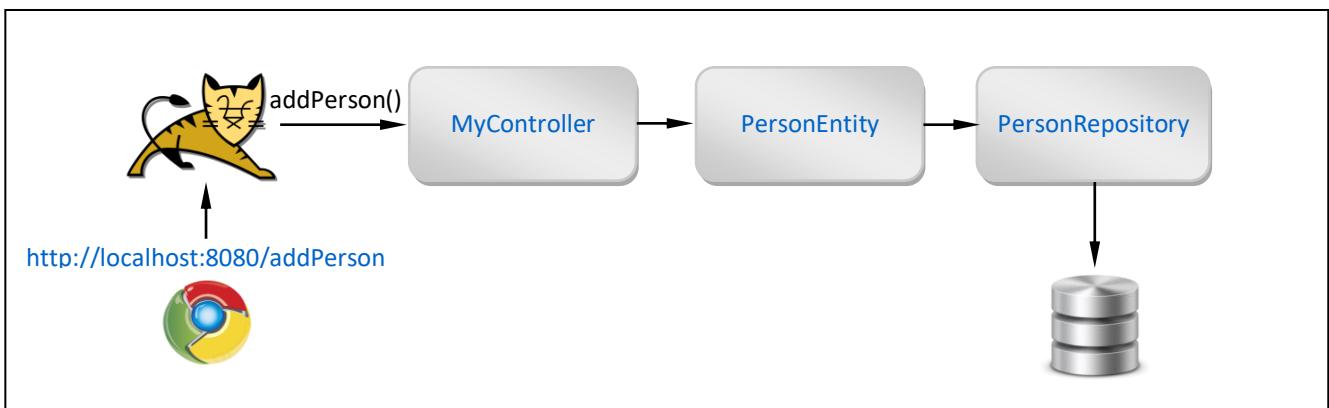
## 2.5.8 @Table

### Info

- In this tutorial we will use JPA's **@Table** Annotation to explicitly specify Table Name (by default Class Name is used).
- Compared to previous example we will only
  - add following line to `@Table(name = "Person")`
- JPA Annotations can only be used on SQL DBs (H2, MySQL, Oracle) and can't be used on NoSQL DBs (MongoDB).

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables <code>@Controller</code> , <code>@RequestMapping</code> and Tomcat Server
SQL	Spring Data JPA	Enables <code>@Entity</code> and <code>@Id</code>
SQL	H2 Database	Enables in-memory H2 DB

### Procedure

- Create Project:** `entity_entity` (add Spring Boot Starters from the table)
- Edit:** `application.properties` (specify H2 DB name & enable H2 Web Console)
- Create Package:** `entities` (inside main package)
  - Create Class:** `PersonEntity.java` (inside package entities)
- Create Package:** `repositories` (inside main package)
  - Create Interface:** `PersonRepository.java` (inside package repositories)
- Create Package:** `controllers` (inside main package)
  - Create Class:** `MyController.java` (inside package controllers)

### `application.properties`

```
# H2 DATABASE
spring.datasource.url      = jdbc:h2:mem:testdb
spring.h2.console.enabled = true
```

### PersonEntity.java

```
package com.ivoronline.springboot.entity_annotation_table.entities;

import javax.persistence.*;

@Entity
@Table(name = "Person")
public class PersonEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    public Integer id;
    public String name;
    public Integer age;

}
```

### PersonRepository.java

```
package com.ivoronline.springboot.entity_annotation_table.repositories;

import com.ivoronline.entity_entity.entities.PersonEntity;
import org.springframework.data.repository.CrudRepository;

public interface PersonRepository extends CrudRepository<PersonEntity, Integer> { }
```

### MyController.java

```
package com.ivoronline.springboot.entity_annotation_table.controllers;

import com.ivoronline.entity_entity.entities.PersonEntity;
import com.ivoronline.entity_entity.repositories.PersonRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class MyController {

    @Autowired
    PersonRepository personRepository;

    @ResponseBody
    @RequestMapping("/addPerson")
    public String addPerson() {

        //CREATE ENTITY OBJECT
        PersonEntity personEntity      = new PersonEntity();
        personEntity.name = "John";
        personEntity.age = 20;

        //STORE ENTITY OBJECT INTO DB
        personRepository.save(personEntity);

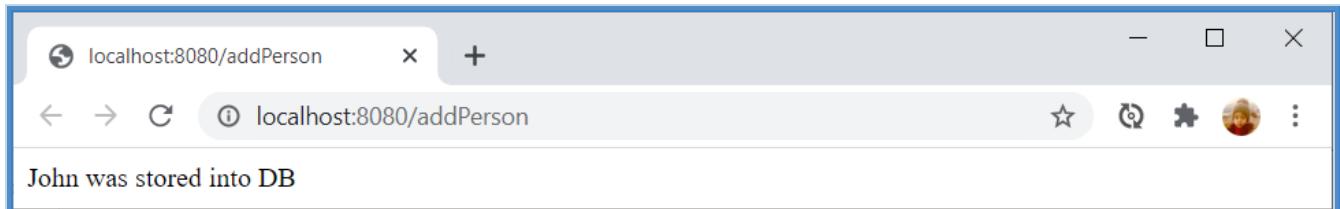
        //RETURN SOMETHING TO BROWSER
        return personEntity.name + " was stored into DB";

    }

}
```

## Results

<http://localhost:8080/addPerson>



<http://localhost:8080/h2-console> - Connect - PERSON\_ENTITY - Run

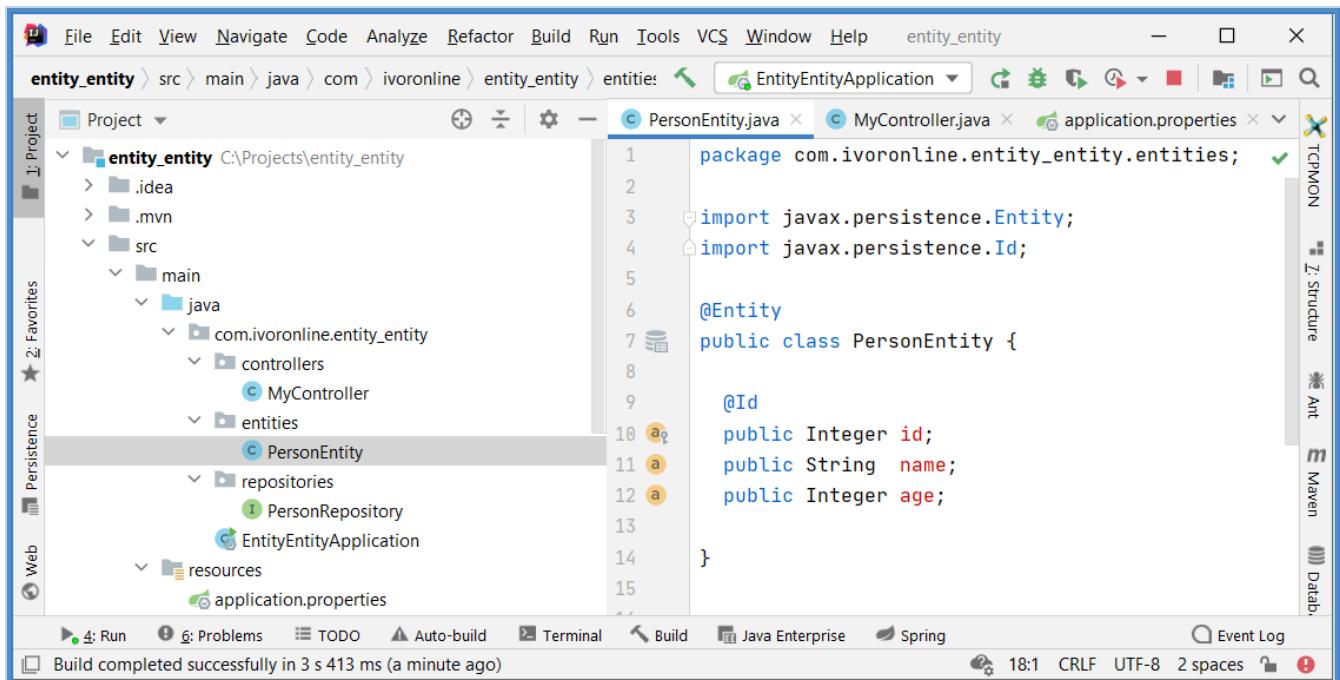
(Open H2 Console)

A screenshot of the H2 Console interface. The left sidebar shows the database schema with tables 'PERSON' and 'INFORMATION\_SCHEMA'. The 'PERSON' table has columns 'ID', 'AGE', and 'NAME'. A query 'SELECT \* FROM PERSON;' is run, resulting in the following table:

ID	AGE	NAME
1	20	John
2	20	John

(2 rows, 2 ms)

## Application Structure



```
<dependencies>

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

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

    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
    </dependency>

</dependencies>
```

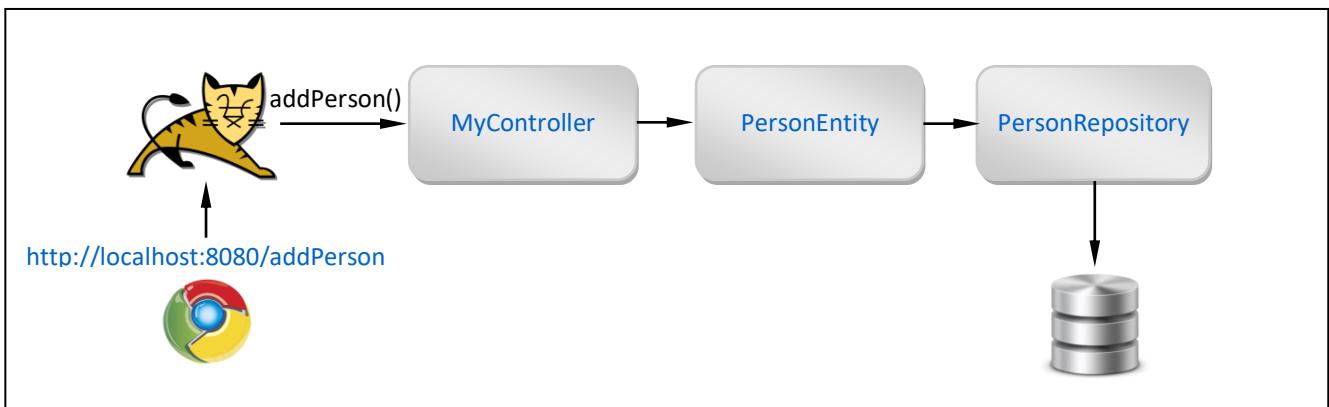
## 2.5.9 @Column

### Info

- This tutorial shows how to use JPA's **@Column** to explicitly specify Column Name (by default Property Name is used).
- JPA Annotations can only be used on SQL DBs (H2, MySQL, Oracle) and can't be used on NoSQL DBs (MongoDB).

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables <code>@Controller</code> , <code>@RequestMapping</code> and Tomcat Server
SQL	Spring Data JPA	Enables <code>@Entity</code> and <code>@Id</code>
SQL	H2 Database	Enables in-memory H2 DB

### Procedure

- Create Project:** entity\_entity (add Spring Boot Starters from the table)
- Edit:** application.properties (specify H2 DB name & enable H2 Web Console)
- Create Package:** entities (inside main package)
  - Create Class:** PersonEntity.java (inside package entities)
- Create Package:** repositories (inside main package)
  - Create Interface:** PersonRepository.java (inside package repositories)
- Create Package:** controllers (inside main package)
  - Create Class:** MyController.java (inside package controllers)

### application.properties

```
# H2 DATABASE
spring.datasource.url      = jdbc:h2:mem:testdb
spring.h2.console.enabled = true
```

### *PersonEntity.java*

```
package com.ivoronline.springboot.entity_annotation_column.entities;

import javax.persistence.*;

@Entity
public class PersonEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    public Integer id;

    @Column(name="PERSON_NAME")
    public String name;

    public Integer age;

}
```

### *PersonRepository.java*

```
package com.ivoronline.springboot.entity_annotation_column.repositories;

import com.ivoronline.entity_entity.entities.PersonEntity;
import org.springframework.data.repository.CrudRepository;

public interface PersonRepository extends CrudRepository<PersonEntity, Integer> { }
```

### *MyController.java*

```
package com.ivoronline.springboot.entity_annotation_column.controllers;

import com.ivoronline.entity_entity.entities.PersonEntity;
import com.ivoronline.entity_entity.repositories.PersonRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class MyController {

    @Autowired
    PersonRepository personRepository;

    @ResponseBody
    @RequestMapping("/addPerson")
    public String addPerson() {

        //CREATE ENTITY OBJECT
        PersonEntity personEntity      = new PersonEntity();
        personEntity.name = "John";
        personEntity.age = 20;

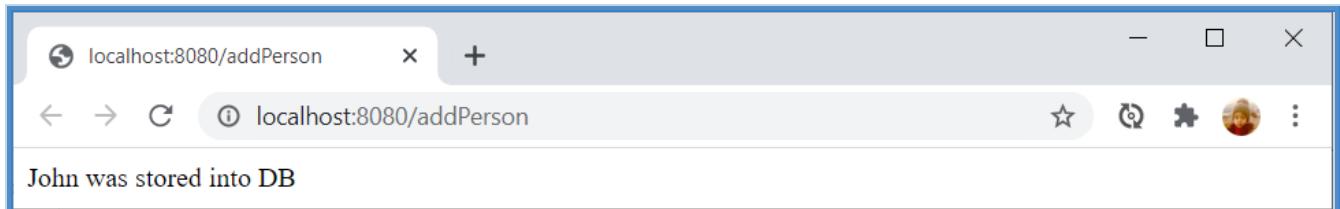
        //STORE ENTITY OBJECT INTO DB
        personRepository.save(personEntity);

        //RETURN SOMETHING TO BROWSER
        return personEntity.name + " was stored into DB";

    }
}
```

## Results

<http://localhost:8080/addPerson>



<http://localhost:8080/h2-console> - Connect - PERSON\_ENTITY - Run

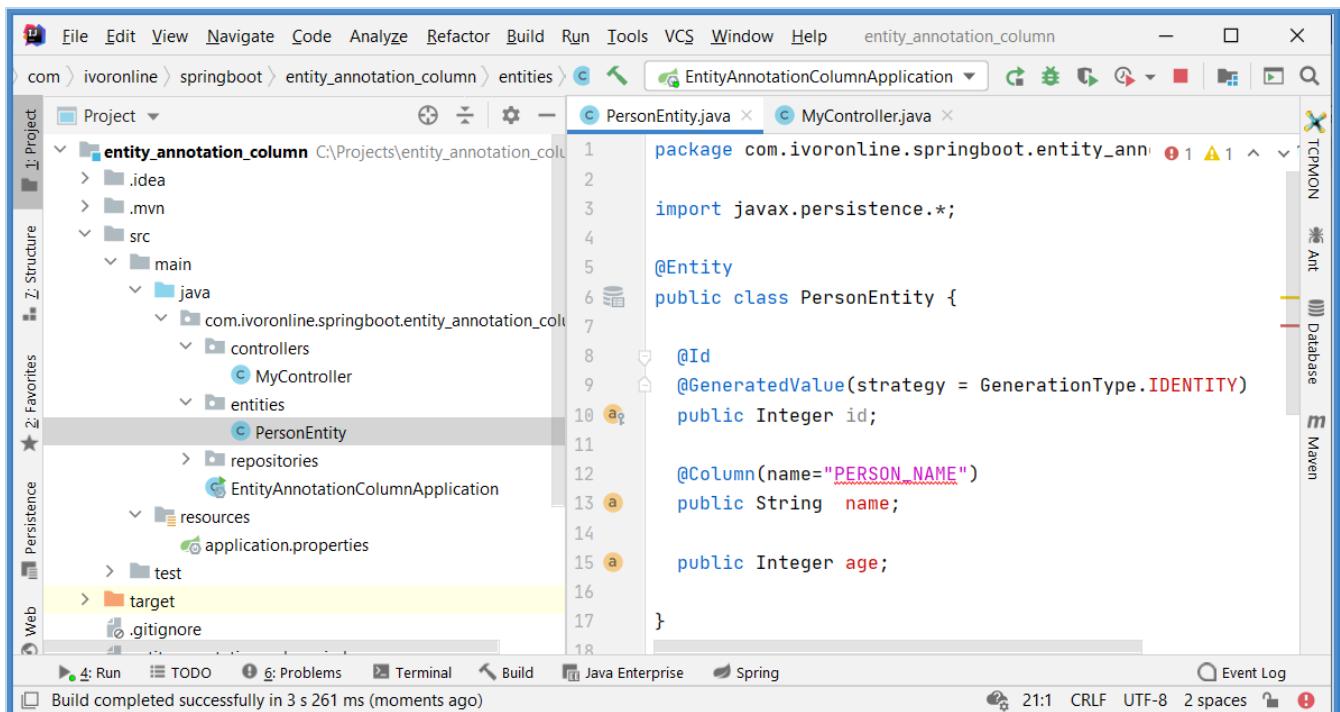
(Open H2 Console)

A screenshot of the H2 Console interface. On the left, the database schema is shown with tables like PERSON\_ENTITY, INFORMATION\_SCHEMA, and Sequences. The PERSON\_ENTITY table has columns ID, AGE, and PERSON\_NAME. The PERSON\_NAME column is highlighted with a red box. A SQL query "SELECT \* FROM PERSON\_ENTITY;" is run, and the result shows one row: ID 1, AGE 20, and PERSON\_NAME John. The entire result table is highlighted with a red box.

ID	AGE	PERSON_NAME
1	20	John

(1 row, 2 ms)

## Application Structure



```
<dependencies>

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

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

    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
    </dependency>

</dependencies>
```

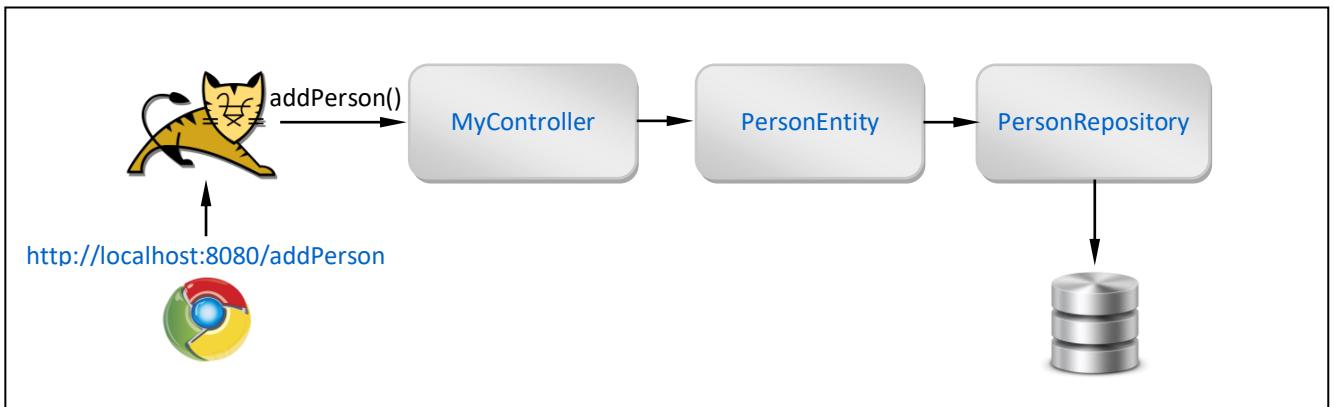
## 2.5.10 @Transient

### Info

- JPA's **@Transient** Annotation specifies that Property should not be stored in the DB (non-persistent).
- JPA Annotations can only be used on SQL DBs (H2, MySQL, Oracle) and can't be used on NoSQL DBs (MongoDB).

Application Schema

[Results]



Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables <code>@Controller</code> , <code>@RequestMapping</code> and Tomcat Server
SQL	Spring Data JPA	Enables <code>@Entity</code> and <code>@Id</code>
SQL	H2 Database	Enables in-memory H2 DB

## Procedure

- Create Project: entity\_entity (add Spring Boot Starters from the table)
- Edit: application.properties (specify H2 DB name & enable H2 Web Console)
- Create Package: entities (inside main package)
- Create Class: PersonEntity.java (inside package entities)
- Create Package: repositories (inside main package)
- Create Interface: PersonRepository.java (inside package repositories)
- Create Package: controllers (inside main package)
- Create Class: MyController.java (inside package controllers)

### application.properties

```
# H2 DATABASE
spring.datasource.url      = jdbc:h2:mem:testdb
spring.h2.console.enabled = true
```

### PersonEntity.java

```
package com.ivoronline.springboot.entity_annotation_transient.entities;

import javax.persistence.*;

@Entity
public class PersonEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    public Integer id;

    public String name;

    public Integer age;

    @Transient
    public String message;

}
```

### PersonRepository.java

```
package com.ivoronline.springboot.entity_annotation_transient.repositories;

import com.ivoronline.entity_entity.entities.PersonEntity;
import org.springframework.data.repository.CrudRepository;

public interface PersonRepository extends CrudRepository<PersonEntity, Integer> { }
```

### MyController.java

```
package com.ivorononline.springboot.entity_annotation_transient.controllers;

import com.ivorononline.springboot.entity_annotation_transient.entities.PersonEntity;
import com.ivorononline.springboot.entity_annotation_transient.repositories.PersonRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class MyController {

    @Autowired
    PersonRepository personRepository;

    @ResponseBody
    @RequestMapping("/addPerson")
    public String addPerson() {

        //CREATE ENTITY OBJECT
        PersonEntity personEntity = new PersonEntity();
        personEntity.name = "John";
        personEntity.age = 20;
        personEntity.message = "John is 20 years old";

        //STORE ENTITY OBJECT INTO DB
        personRepository.save(personEntity);

        //RETURN SOMETHING TO BROWSER
        return personEntity.message;
    }

}
```

## Results

<http://localhost:8080/addPerson>

localhost:8080/addPerson

localhost:8080/addPerson

John is 20 years old

<http://localhost:8080/h2-console> - Connect - PERSON\_ENTITY - Run

(Open H2 Console)

localhost:8080/addPerson

localhost:8080/h2-console/login.do?jsessionid=d1e3112a1477a74a50a...

Auto commit | Max rows: 1000 | Run | Run Selected | Auto complete | Clear | SQL statement: SELECT \* FROM PERSON\_ENTITY

jdbc:h2:mem:testdb

PERSON\_ENTITY

- + ID
- + AGE
- + NAME
- + Indexes

INFORMATION\_SCHEMA

Sequences

Users

H2 1.4.200 (2019-10-14)

SELECT \* FROM PERSON\_ENTITY;

ID	AGE	NAME
1	20	John

(1 row, 3 ms)

## Application Structure

File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help entity\_annotation\_transient

Project

entity\_annotation\_transient

- .idea
- .mvn
- src
  - main
    - java
      - com.ivoronline.springboot.entity\_annotation\_transient.controllers
      - entities

Structure

PersonEntity.java

```
package com.ivoronline.springboot.entity_annotation_transient;

import javax.persistence.*;

@Entity
public class PersonEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    public Integer id;

    public String name;

    public Integer age;

    @Transient
    public String message;
}
```

MyController.java

EntityAnnotationTransientApplication

Resource

Test

target

.gitignore

entity\_annotation\_transient.iml

HELP.md

mvnw

mvnw.cmd

Run TODO Problems Terminal Build Java Enterprise Spring Event Log

All files are up-to-date (2 minutes ago)

```
<dependencies>

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

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

    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
    </dependency>

</dependencies>
```

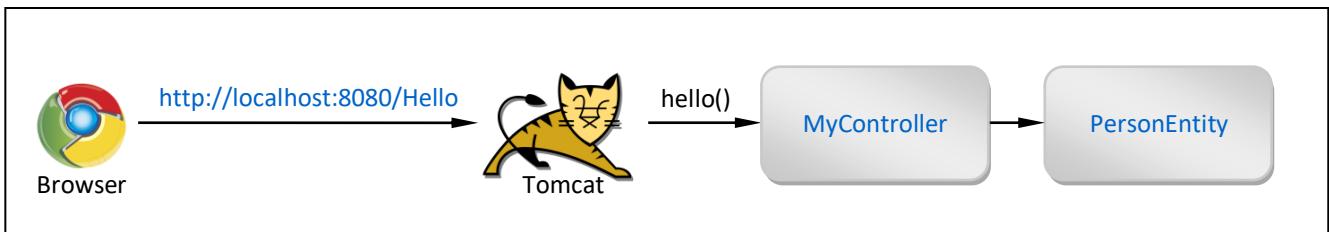
## 2.5.11 Recommended Annotations

### Info

- This tutorial show how to create Entity that uses all of the previously discussed functionalities
  - [Entity - Private Properties](#) (protect Properties by declaring them private)
  - [Entity - @Component \(Spring\)](#) (use @Component annotation so that Spring can use DI to Instantiate Class)
  - [Entity - @Data \(Lombok\)](#) (use @Data annotation so that Lombok can create helper methods)
  - [Entity - @Entity \(JPA\)](#) (use @Entity so that JPA can remind you to declare Primary Key with @Id)

### Application Schema

[[Results](#)]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables @RequestMapping (includes Tomcat Server)
Developer Tools	Lombok	Enables @Data (generates helper methods: setters, getters, ...)
SQL	Spring Data JPA	Enables @Entity and @Id
SQL	H2 Database	Some DB is needed to initialize JPA

## Procedure

---

- Create Project: entity\_recommended (add Spring Boot Starters from the table)
- Create Package: entities (inside main package)
  - Create Class: PersonEntity.java (inside package entities)
- Create Package: controllers (inside main package)
  - Create Class: MyController.java (inside package controllers)

PersonEntity.java

```
package com.ivoronline.springboot.entity_recommended.entitites;

import lombok.Data;
import org.springframework.stereotype.Component;
import javax.persistence.Entity;
import javax.persistence.Id;

@Data
@Entity
@Component
public class PersonEntity {

    @Id
    private Long id;
    private String name;
    private Integer age;

}
```

MyController.java

```
package com.ivoronline.springboot.entity_recommended.controllers;

import com.ivoronline.springboot.entity_recommended.entitites.PersonEntity;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class MyController {

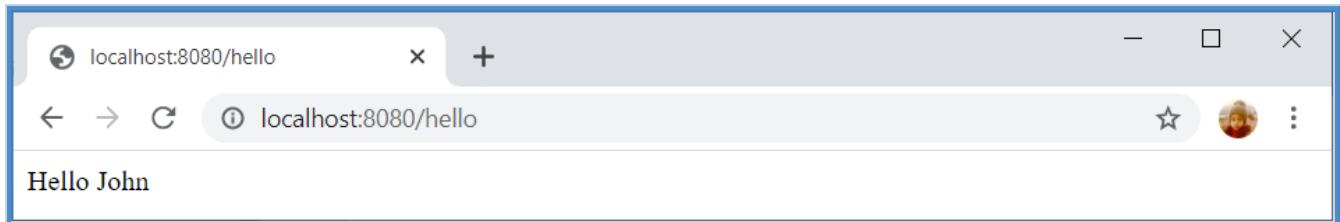
    @Autowired
    PersonEntity personEntity;

    @ResponseBody
    @RequestMapping("/Hello")
    public String hello() {
        personEntity.setName("John");
        String name = personEntity.getName();
        return "Hello " + name;
    }

}
```

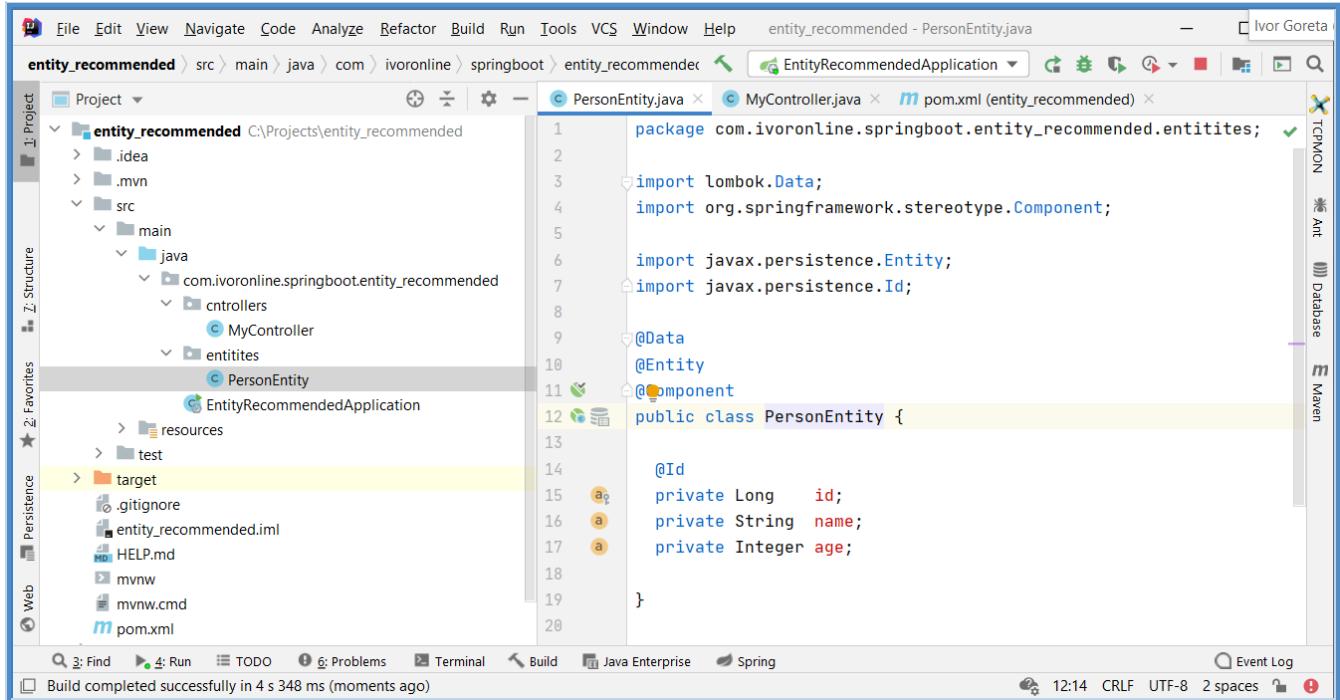
## Results

<http://localhost:8080>Hello>



## Application Structure

(Lombok generated methods are shown under Structure View)



## pom.xml

```
<dependencies>

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

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

    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
    </dependency>

    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <optional>true</optional>
    </dependency>

</dependencies>
```

## 2.6 Databases

### Info

---

- Following tutorials show how to work with different Databases: [H2](#), [MySQL](#), [PostgreSQL](#), [MongoDB](#).

## 2.6.1 H2

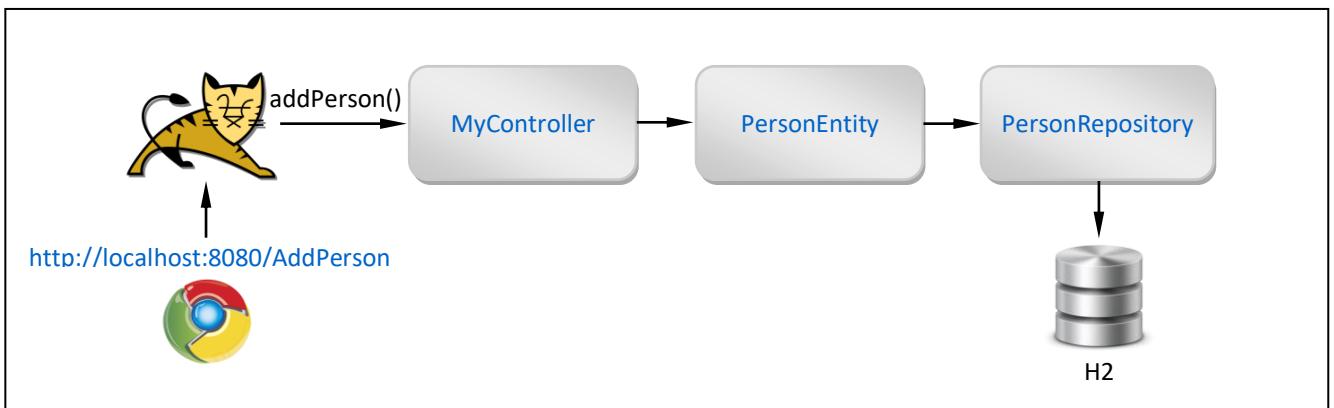
### Info

[G]

- This tutorial shows how to use Repository that works with H2 Database.

#### Application Schema

[Results]



#### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables @Controller and @RequestMapping. Includes Tomcat Server.
SQL	Spring Data JPA	Enables @Entity and @Id
SQL	H2 Database	Enables in-memory H2 Database

### Procedure

- Create Project:** `springboot_db_h2` (add Spring Boot Starters from the table)
- Edit:** `application.properties` (specify H2 DB name & enable H2 Web Console)
- Create Package:** `entities` (inside main package)
  - Create Class:** `PersonEntity.java` (inside package `entities`)
- Create Package:** `repositories` (inside main package)
  - Create Interface:** `PersonRepository.java` (inside package `repositories`)
- Create Package:** `controllers` (inside main package)
  - Create Class:** `MyController.java` (inside package `controllers`)

#### application.properties

```
# H2 DATABASE
spring.h2.console.enabled = true
spring.datasource.url      = jdbc:h2:mem:testdb
```

### *PersonEntity.java*

```
package com.ivoronline.springboot_db_h2.entities;

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

@Entity
public class PersonEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    public Integer id;
    public String name;
    public Integer age;

}
```

### *PersonRepository.java*

```
package com.ivoronline.springboot.repository_store_entity.respositories;

import com.ivoronline.springboot.repository_store_entity.entities.PersonEntity;
import org.springframework.data.repository.CrudRepository;

public interface PersonRepository extends CrudRepository<PersonEntity, Integer> { }
```

### *MyController.java*

```
package com.ivoronline.springboot_db_h2.controllers;

import com.ivoronline.springboot_db_h2.entities.PersonEntity;
import com.ivoronline.springboot_db_h2.repositories.PersonRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class MyController {

    @Autowired
    PersonRepository personRepository;

    @ResponseBody
    @RequestMapping("/AddPerson")
    public String addPerson() {

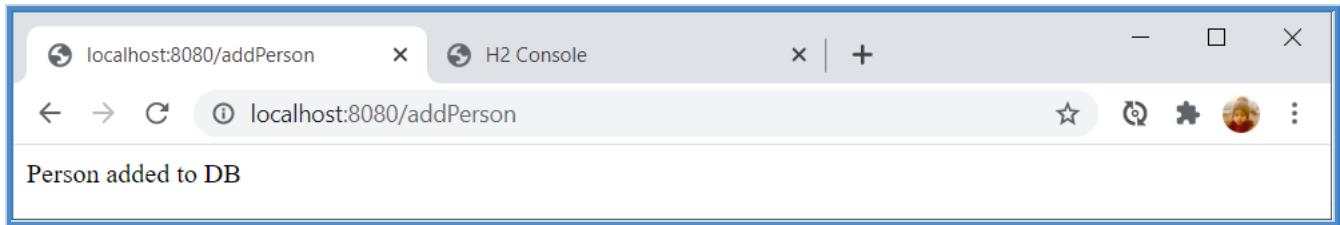
        //CREATE PERSON
        PersonEntity personEntity      = new PersonEntity();
        personEntity.name = "John";
        personEntity.age = 20;

        //STORE PERSON
        personRepository.save(personEntity);

        //RETURN SOMETHING
        return "Hello " + personEntity.name;
    }
}
```

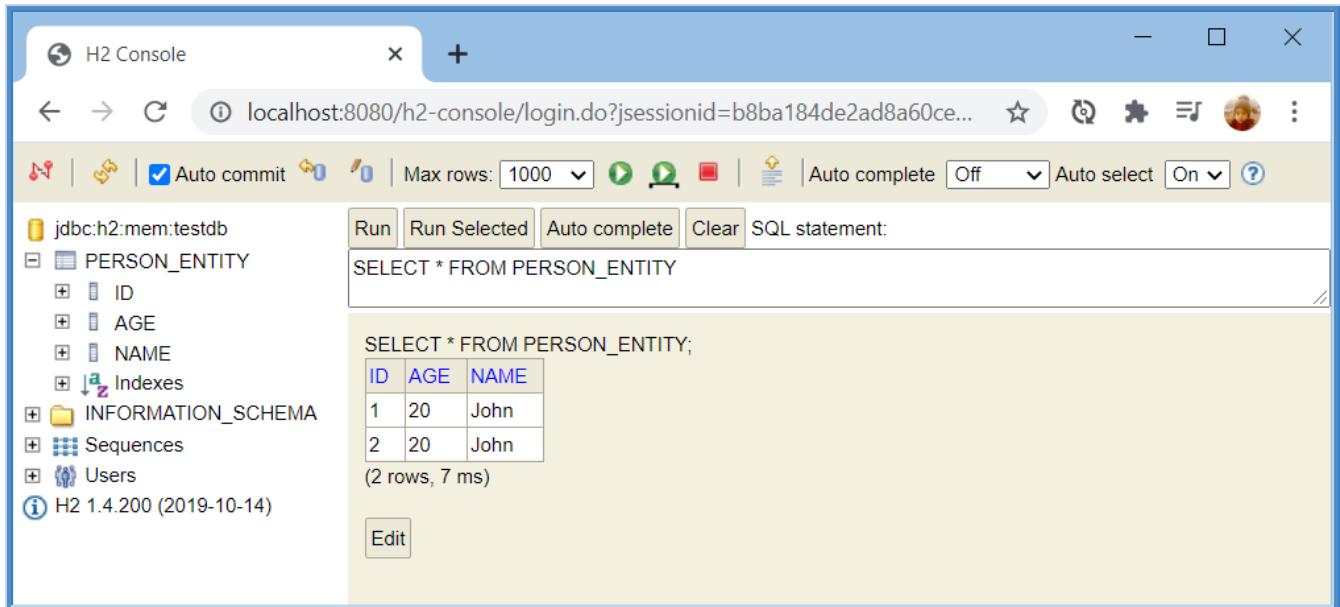
## Results

<http://localhost:8080/AddPerson>

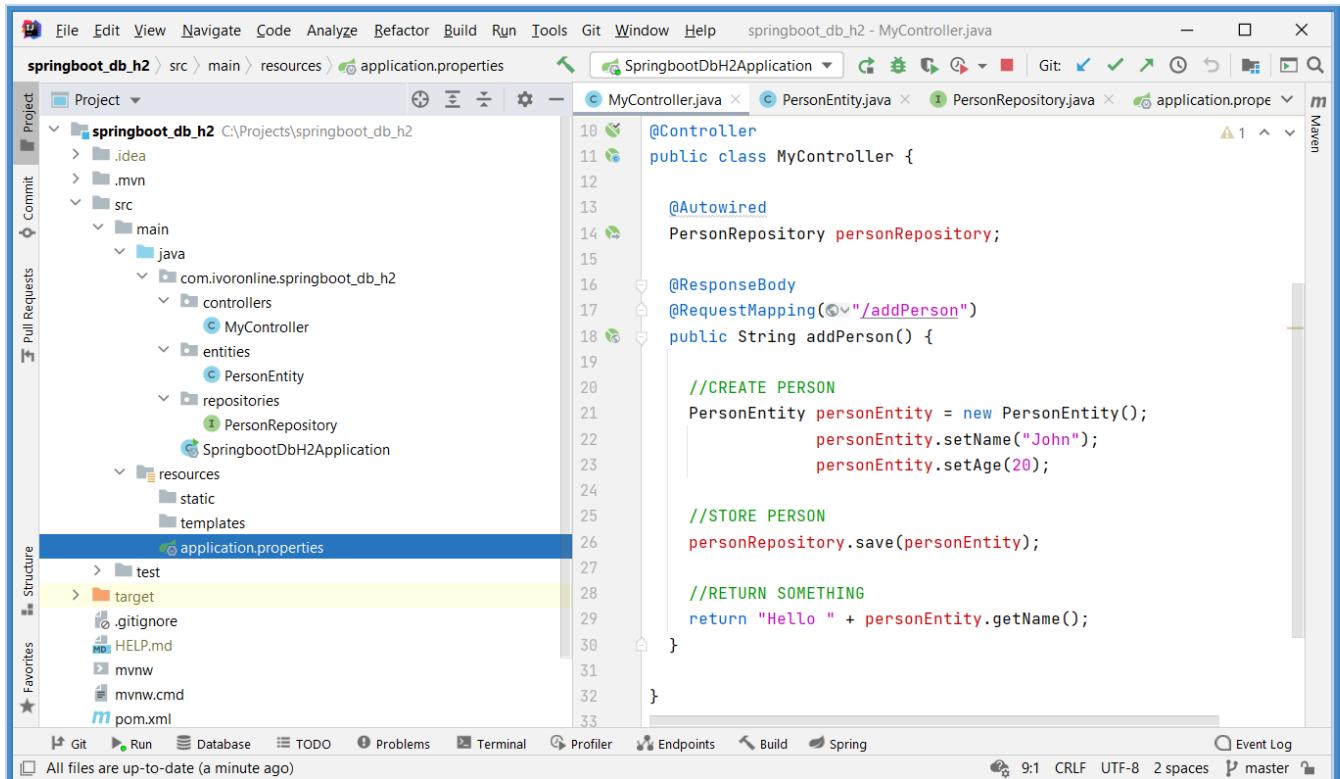


<http://localhost:8080/h2-console>

(Open H2 Console)



## Application Structure



```
<dependencies>

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

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

    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
    </dependency>

    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <optional>true</optional>
    </dependency>

</dependencies>
```

## 2.6.2 MySQL

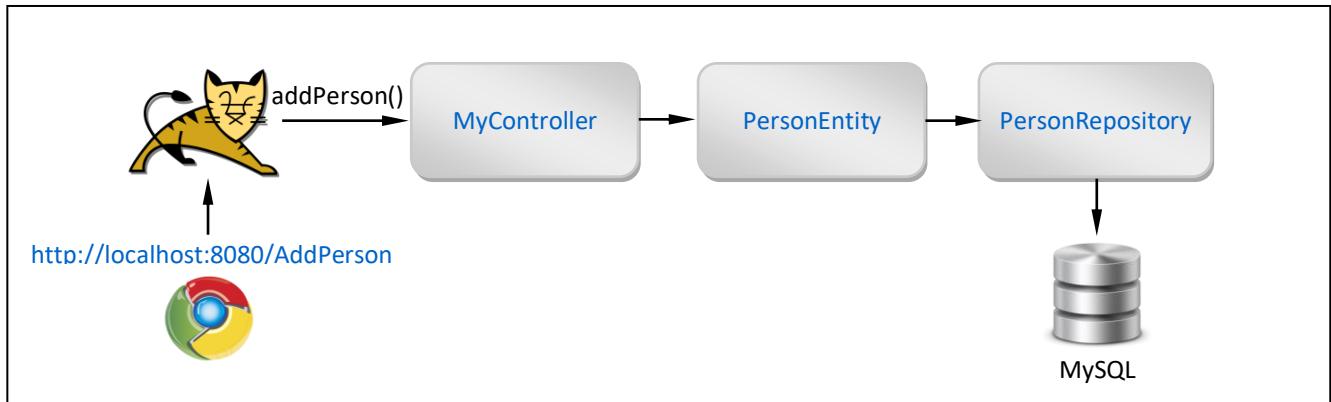
### Info

[R]

- This tutorial show how to use Repository to work with MySQL Database.
- When you install MySQL DB it will be installed with default **world** database/schema which we will use in our example.
- To use different Database/Schema you have to manually [Create Schema/Database](#) before connecting with Hibernate.

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables <code>@RequestMapping</code> (includes Tomcat Server)
SQL	Spring Data JPA	Enables <code>@Entity</code> and <code>@Id</code>
SQL	MySQL Database	Enables Hibernate to work with MySQL DB

### Procedure

- Create Project:** `springboot_db_mysql` (add Spring Boot Starters from the table)
- Edit File:** `application.properties` (enter DB connection parameters)
- Create Package:** `entities` (inside main package)
  - Create Class:** `PersonEntity.java` (inside package `entities`)
- Create Package:** `repositories` (inside main package)
  - Create Interface:** `PersonRepository.java` (inside package `repositories`)
- Create Package:** `controllers` (inside main package)
  - Create Class:** `MyController.java` (inside package `controllers`)

### `application.properties`

```
# MYSQL DATABASE
spring.datasource.url      = jdbc:mysql://localhost:3306/world?serverTimezone=UTC
spring.datasource.username  = root
spring.datasource.password  = admin
spring.jpa.database-platform = org.hibernate.dialect.MySQL5Dialect
spring.datasource.driver-class-name = com.mysql.jdbc.Driver

# JPA / HIBERNATE
spring.jpa.hibernate.ddl-auto = create-drop
spring.jpa.show-sql           = true
```

### *PersonEntity.java*

```
package com.ivoronline.springboot_db_mysql.entities;

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

@Entity
public class PersonEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    public Integer id;
    public String name;
    public Integer age;

}
```

### *PersonRepository.java*

```
package com.ivoronline.springboot_db_mysql.repositories;

import com.ivoronline.springboot.repository_mysql.entities.PersonEntity;
import org.springframework.data.repository.CrudRepository;

public interface PersonRepository extends CrudRepository<PersonEntity, Integer> { }
```

### *MyController.java*

```
package com.ivoronline.springboot_db_mysql.controllers;

import com.ivoronline.springboot.repository_mysql.entities.PersonEntity;
import com.ivoronline.springboot.repository_mysql.repositories.PersonRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class MyController {

    @Autowired
    PersonRepository personRepository;

    @ResponseBody
    @RequestMapping("/AddPerson")
    public String addPerson() {

        //CREATE PERSON ENTITY
        PersonEntity personEntity      = new PersonEntity();
        personEntity.name = "John";
        personEntity.age  = 20;

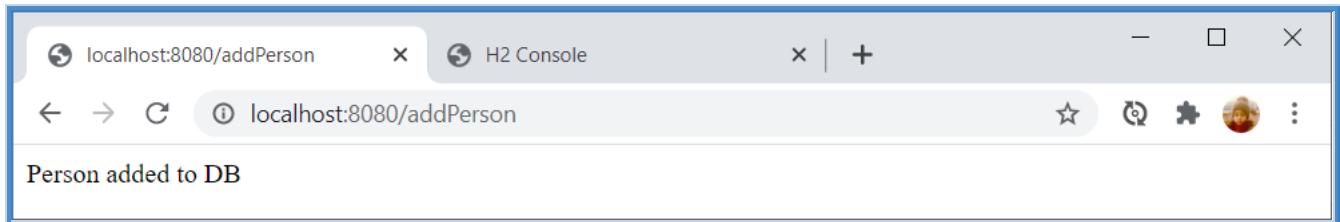
        //STORE PERSON ENTITY
        personRepository.save(personEntity);

        //RETURN SOMETHING TO BROWSER
        return "Person added to DB";

    }
}
```

## Results

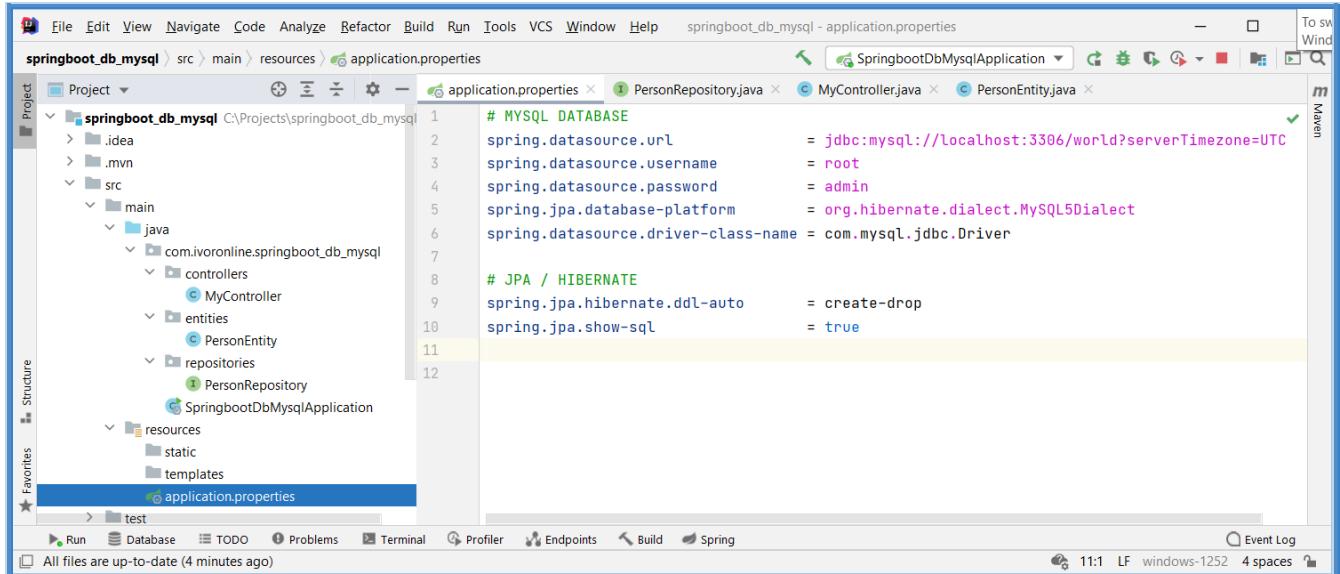
<http://localhost:8080/AddPerson>



## MySQL Workbench

A screenshot of the MySQL Workbench application. The left sidebar shows the 'Schemas' tree with 'world' selected, containing 'Tables' like 'city', 'country', 'countrylanguage', and 'person\_entity'. The central pane shows a SQL editor with the query 'SELECT \* FROM world.person\_entity;' and a result grid displaying two rows of data: id=1, age=20, name='John' and id=2, age=20, name='John'. The right pane contains a message about context help being disabled and a log of recent actions.

## Application Structure



```
<dependencies>

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

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

    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <scope>runtime</scope>
    </dependency>

</dependencies>
```

## 2.6.3 PostgreSQL

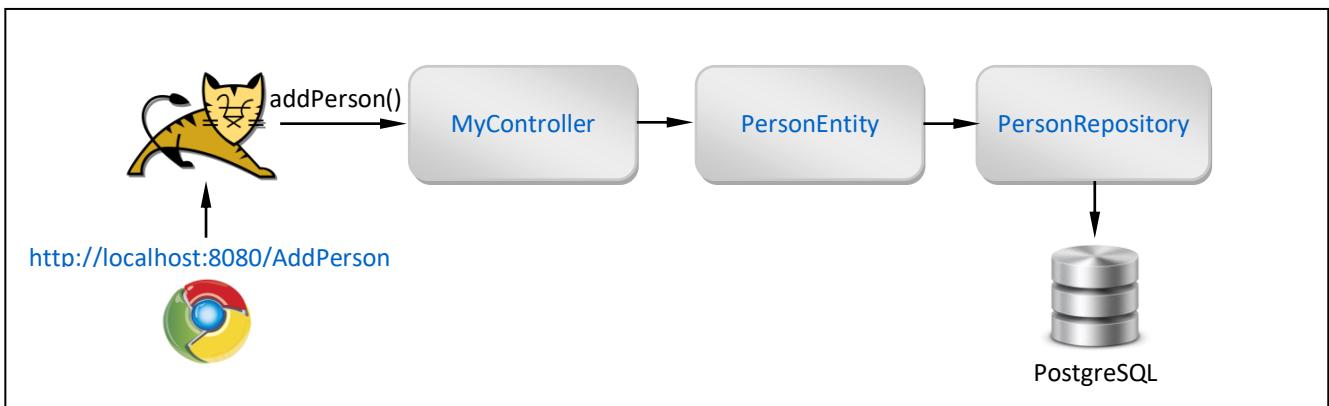
### Info

[G]

- This tutorial show how to use Repository to work with PostgreSQL Database.

*Application Schema*

[Results]



*Spring Boot Starters*

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables @RequestMapping (includes Tomcat Server)
SQL	Spring Data JPA	Enables @Entity and @Id
SQL	PostgreSQL Database	Enables Hibernate to work with PostgreSQL DB

### Procedure

- Create Project: `springboot_db_postgresql` (add Spring Boot Starters from the table)
- Edit File: `application.properties` (enter DB connection parameters)
- Create Package: `entities` (inside main package)
  - Create Java Class: `PersonEntity.java` (inside package entities)
- Create Package: `repositories` (inside main package)
  - Create Java Interface: `PersonRepository.java` (inside package repositories)
- Create Package: `controllers` (inside main package)
  - Create Java Class: `MyController.java` (inside package controllers)

*application.properties*

```
# POSTGRES DATABASE
spring.datasource.url      = jdbc:postgresql://localhost:5432/postgres
spring.datasource.username   = postgres
spring.datasource.password   = letmein
spring.datasource.driver-class-name = org.postgresql.Driver

# JPA / HIBERNATE
spring.jpa.hibernate.ddl-auto = create-drop
```

### *PersonEntity.java*

```
package com.ivoronline.springboot_db_postgresql.entities;

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

@Entity
public class PersonEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    public Integer id;
    public String name;
    public Integer age;

}
```

### *PersonRepository.java*

```
package com.ivoronline.springboot_db_postgresql.repositories;

import com.ivoronline.springboot_db_postgresql.entities.PersonEntity;
import org.springframework.data.repository.CrudRepository;

public interface PersonRepository extends CrudRepository<PersonEntity, Integer> { }
```

### *MyController.java*

```
package com.ivoronline.springboot_db_postgresql.controllers;

import com.ivoronline.springboot_db_postgresql.entities.PersonEntity;
import com.ivoronline.springboot_db_postgresql.repositories.PersonRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class MyController {

    @Autowired
    PersonRepository personRepository;

    @ResponseBody
    @RequestMapping("/AddPerson")
    public String addPerson() {

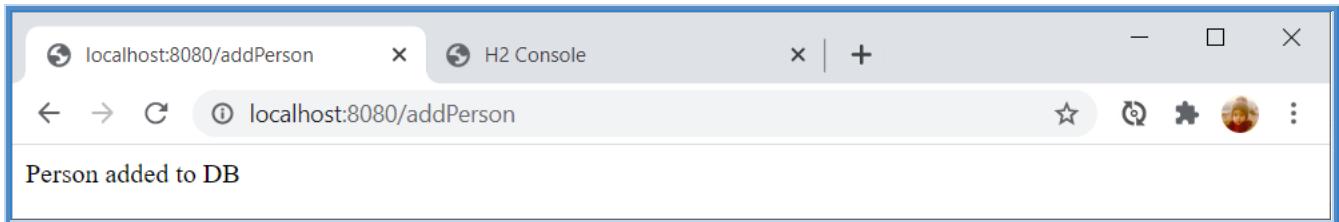
        //CREATE PERSON ENTITY
        PersonEntity personEntity      = new PersonEntity();
        personEntity.name = "John";
        personEntity.age = 20;

        //STORE PERSON ENTITY
        personRepository.save(personEntity);

        //RETURN SOMETHING TO BROWSER
        return "Person added to DB";
    }
}
```

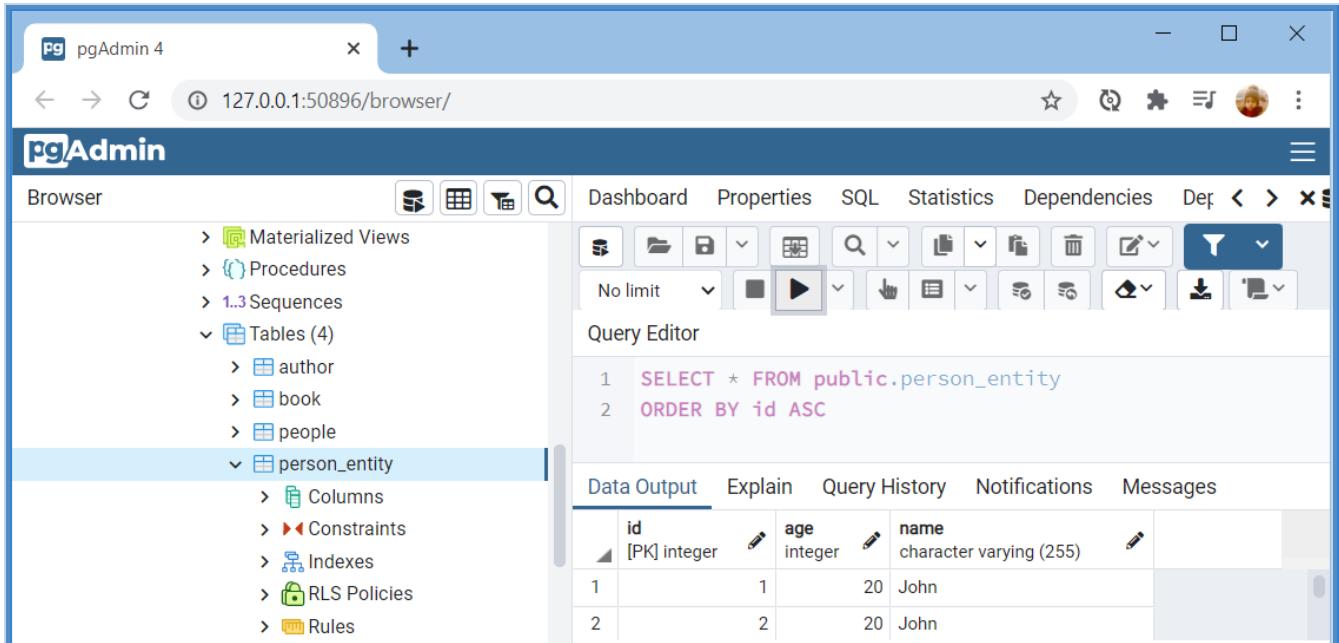
## Results

<http://localhost:8080/AddPerson>

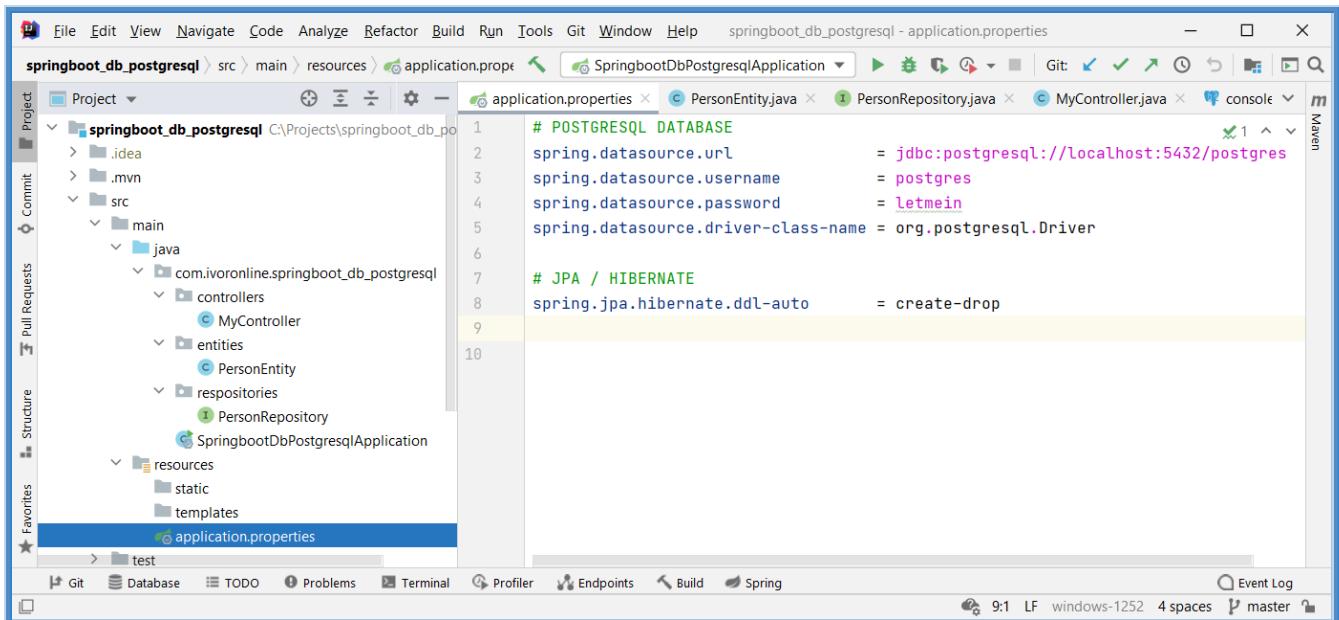


pgAdmin4

(<http://127.0.0.1:50043/browser>)



## Application Structure



```
<dependencies>

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

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

    <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <scope>runtime</scope>
    </dependency>

</dependencies>
```

## 2.6.4 MongoDB

### Info

[G]

- This tutorial show how to use Repository to work with MongoDB Database.
- No configuration** is needed.

By default Hibernate will use `anonymous` access to connect to `localhost` at port `27017` and create `test` database.

If your connection parameters differ you can explicitly set them in `application.properties` file as shown below.

If you specify database name which doesn't exist, Hibernate will create it for you.

- When using MongoDB you have to remove **JPA** from `pom.xml` (it only works for SQL DBs like: H2, MySQL, Oracle). This means that you can't use JPA Annotations like: `@Entity`, `@Table`, `@Column`, `@Transient`, `@Id`, `@GeneratedValue`.

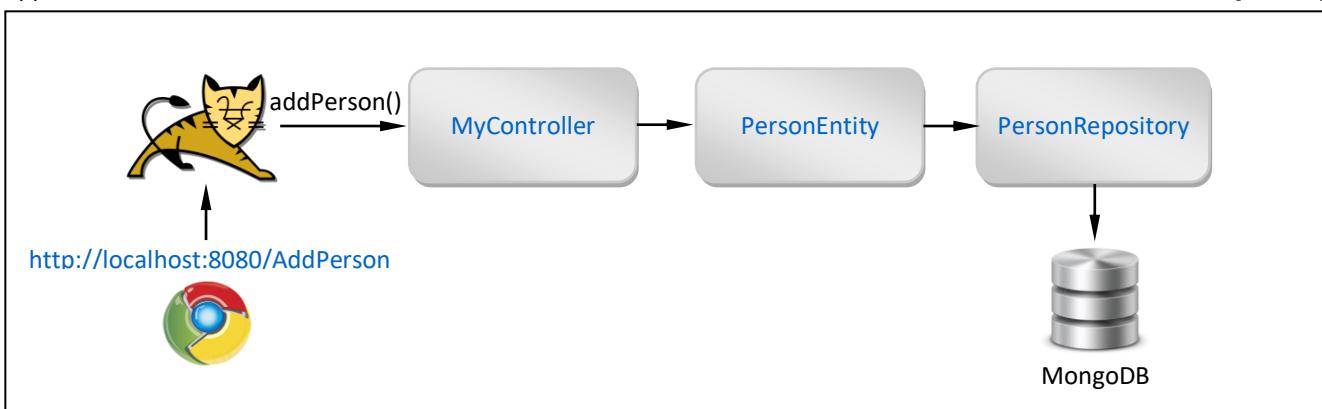
`application.properties`

(choose one if needed)

```
spring.data.mongodb.url=mongodb://localhost:27017  
spring.data.mongodb.uri=mongodb://localhost:27017/mydatabase  
spring.data.mongodb.uri=mongodb://myuser:mypassword@localhost:27017/mydatabase
```

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables <code>@RequestMapping</code> (includes Tomcat Server)
NoSQL	Spring Data MongoDB	Enables working with MongoDB

### Procedure

- Create Project:** `springboot_db_mongodb` (add Spring Boot Starters from the table)
- Create Package:** `entities` (inside main package)
  - Create Interface:** `PersonEntity.java` (inside package entities)
- Create Package:** `repositories` (inside main package)
  - Create Class:** `PersonRepository.java` (inside package repositories)
- Create Package:** `controllers` (inside main package)
  - Create Class:** `MyController.java` (inside package controllers)

### `PersonEntity.java`

```
package com.ivoronline.springboot_db_mongodb.entities;

public class PersonEntity {
    public Integer id;
    public String name;
    public Integer age;
}
```

### PersonRepository.java

```
package com.ivorononline.springboot_db_mongodb.repositories;

import com.ivorononline.springboot_db_mongodb.PersonEntity;
import org.springframework.data.repository.CrudRepository;

public interface PersonRepository extends CrudRepository<PersonEntity, Integer> { }
```

### MyController.java

```
package com.ivorononline.springboot_db_mongodb.controllers;

import com.ivorononline.springboot_db_mongodb.PersonEntity;
import com.ivorononline.springboot_db_mongodb.PersonRepository;
import org.springframework.stereotype.Controller;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class MyController {

    @Autowired
    PersonRepository personRepository;

    @ResponseBody
    @RequestMapping("/AddPerson")
    public String addPerson() {

        //CREATE PERSON ENTITY
        PersonEntity personEntity = new PersonEntity();
        personEntity.id = 1;
        personEntity.name = "John";
        personEntity.age = 20;

        //STORE PERSON ENTITY
        personRepository.save(personEntity);

        //RETURN SOMETHING TO BROWSER
        return "Person added to DB";
    }
}
```

## Results

<http://localhost:8080/AddPerson>

A screenshot of a web browser window. The address bar shows 'localhost:8080/addPerson'. The main content area displays the text 'Person added to DB'.

## Compass Client

(MongoDB)

A screenshot of the MongoDB Compass client interface. On the left, the sidebar shows 'Local' with '3 DBS' and '1 COLLECTIONS'. The 'test' database is selected, and its 'personEntity' collection is highlighted. The main panel shows the 'test.personEntity' collection details: DOCUMENTS 0, TOTAL SIZE 0B, AVG. SIZE 0B, INDEXES 1, TOTAL SIZE 36.0KB, AVG. SIZE 36.0KB. Below this, the 'Documents' tab is active, showing a table with one document: \_id Int32, name String, age Int32, \_cls String. The document data is: 1, "John", 20, "com.". There are buttons for 'ADD DATA', 'VIEW', 'FILTER', 'OPTIONS', 'FIND', 'RESET', and 'REFRESH'.

## Application Structure

A screenshot of the IntelliJ IDEA IDE. The left side shows the 'Project' structure with a 'springboot\_repository\_mongodb' module containing 'src/main/java/com.ivoronline.springboot.springboot\_repository\_mongodb/controllers', 'entities', 'repositories', and 'SpringbootRepositoryMongodbApplication'. The right side shows the code editor with 'PersonEntity.java' open, displaying the following code:

```
package com.ivoronline.springboot.springboot_repository_mongodb.entities;

public class PersonEntity {
    public Integer id;
    public String name;
    public Integer age;
}
```

## pom.xml

```
<dependencies>

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

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

</dependencies>
```

# 2.7 Relationships

## Info

---

- Following tutorials show how to create different relationships between Entities/Tables
  - [@OneToOne](#)
  - [@OneToMany](#)
  - [@ManyToMany](#)
- Tables relationships are quite a complex topic since relationships can be implemented in different ways
  - inside **DB** by using **Foreign-Key** or **Join-Table**
  - inside **Code** by using **unidirectional** or **bidirectional** references

## @OneToOne

---

- **@OneToMany** relationship is used when one Entity can be related to only one Entity of another type.
- **@OneToOne** is used when you want to split data into multiple Tables.

For instance instead of having single Table with 20 columns you can have 2 Tables with 10 columns each.

It allows you to logically **group data**

- first Table can contain Customer's basic information like: FirstName, LastName, Age
- second Table can contain Customer's contact information like: Email, Phone, Address
- **@OneToOne** relationship can be implemented either by using
  - **Foreign Key** in which case one Table will have an additional column that points to a related Record in the second Table
  - **Join Table** which holds information which Record are related

## @OneToMany

---

- **@OneToMany** relationship is used when single Entity can be related to multiple Entities of another type.  
For instance single Author can write multiple Books.
- **@OneToMany** relationship can be implemented either by using
  - **Foreign Key** in which case one Table will have an additional column that points to a related Record in the second Table
  - **Join Table** which holds information which Record from first Table is related to which Records from the second Table

## @ManyToMany

---

- **@ManyToMany** relationship is used when one Entity can be related to multiple Entities of another type and vice versa.  
For instance single Author can write multiple Books.  
But at the same time Book can be written by multiple Authors.
- **@OneToMany** relationship can be implemented only by using
  - **Join Table** which holds information which Records from first Table are related to which Records from the second Table
- Note that by default Hibernate creates two Join Tables and in order to have only one you have to specify owning side.

## 2.7.1 @OneToOne - Join Table

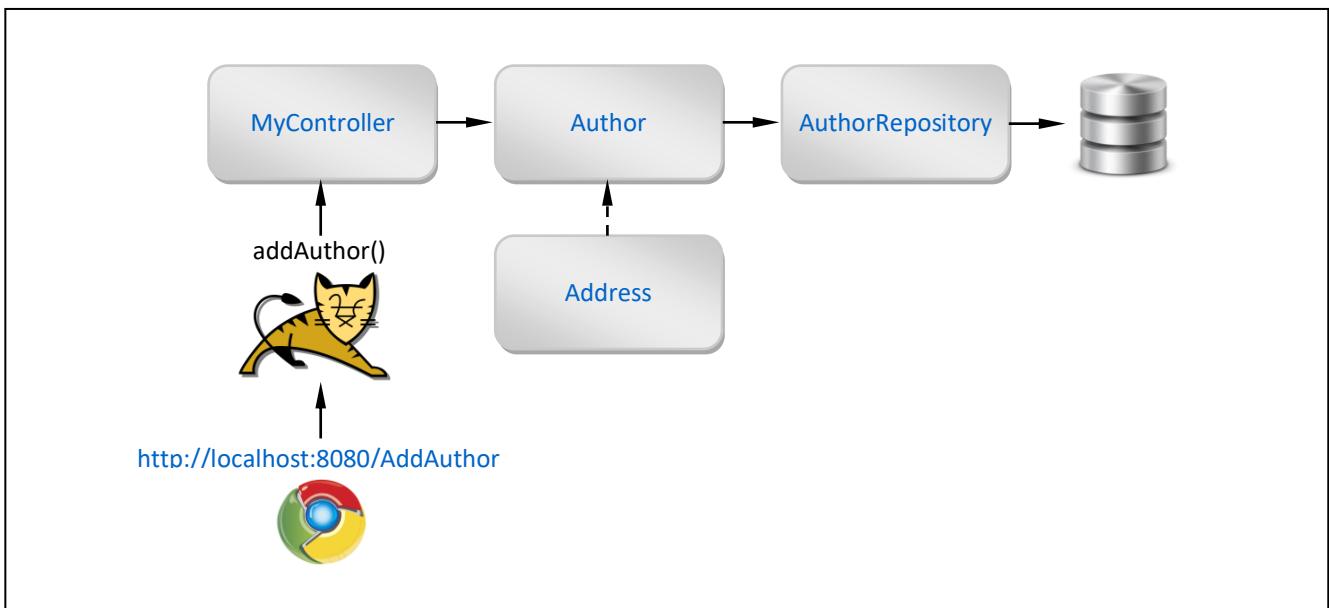
Info

[G] [R]

- This tutorial shows how to implement **@OneToOne** relationship by using **Join Table**.

Application Schema

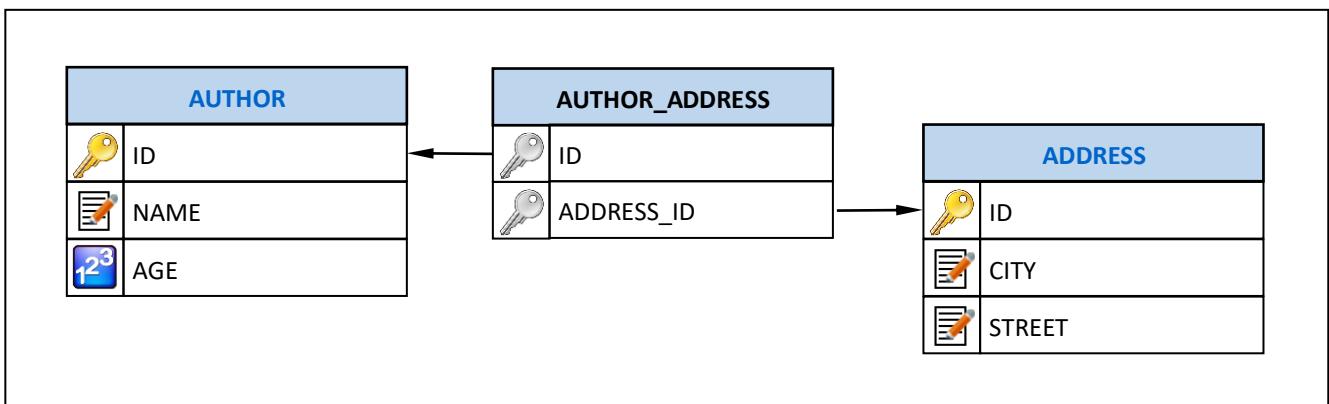
[Results]



Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables <code>@Controller</code> and <code>@RequestMapping</code> . Includes Tomcat Server.
SQL	Spring Data JPA	Enables <code>@Entity</code> and <code>@Id</code>
SQL	H2 Database	Enables in-memory H2 Database

DB Schema



Syntax

```
@OneToOne(cascade = CascadeType.ALL)
@JoinTable(name = "AUTHOR_ADDRESS",
    joinColumns      = { @JoinColumn(name = "AUTHOR_ID", referencedColumnName = "id") },
    inverseJoinColumns = { @JoinColumn(name = "ADDRESS_ID", referencedColumnName = "id") }
)
public Address address;
```

## Procedure

— Create Project:	springboot_relationships_onetoone_jointable	(add Spring Boot Starters from the table)
— Edit File:	application.properties	(specify H2 DB name & enable H2 Web Console)
— Create Package:	entities	(inside main package)
— Create Class:	Author.java	(inside package entities)
— Create Class:	Address.java	(inside package entities)
— Create Package:	repositories	(inside main package)
— Create Interface:	AuthorRepository.java	(inside package repositories)
— Create Package:	controllers	(inside main package)
— Create Class:	MyController.java	(inside package controllers)

### application.properties

```
spring.datasource.url      = jdbc:h2:mem:testdb
spring.h2.console.enabled = true
```

### Author.java

```
package com.ivoronline.springboot_relationships_onetoone_jointable.entities;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.JoinTable;
import javax.persistence.OneToOne;
import javax.persistence.CascadeType;

@Entity
public class Author {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    public Integer id;
    public String name;
    public Integer age;

    @OneToOne(cascade = CascadeType.ALL)
    @JoinTable(name = "AUTHOR_ADDRESS")
    public Address address;

}
```

### Address.java

```
package com.ivoronline.springboot_relationships_onetoone_jointable.entities;

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

@Entity
public class Address {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    public Integer id;
    public String city;
    public String street;
}
```

### *AuthorRepository.java*

```
package com.ivorononline.springboot_relationships_onetoone_jointable.repositories;

import com.ivorononline.springboot_relationships_onetoone_jointable.entities.Author;
import org.springframework.data.repository.CrudRepository;

public interface AuthorRepository extends CrudRepository<Author, Integer> { }
```

### *MyController.java*

```
package com.ivorononline.springboot_relationships_onetoone_jointable.controllers;

import com.ivorononline.springboot_relationships_onetoone_jointable.entities.Author;
import com.ivorononline.springboot_relationships_onetoone_jointable.entities.Address;
import com.ivorononline.springboot_relationships_onetoone_jointable.repositories.AuthorRepository;
import org.springframework.stereotype.Controller;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class MyController {

    @Autowired
    AuthorRepository authorRepository;

    @ResponseBody
    @RequestMapping("/AddAuthor")
    public String addAuthor() {

        //CREATE ADDRESS ENTITY
        Address address = new Address();
        address.city = "London";
        address.street = "Piccadilly";

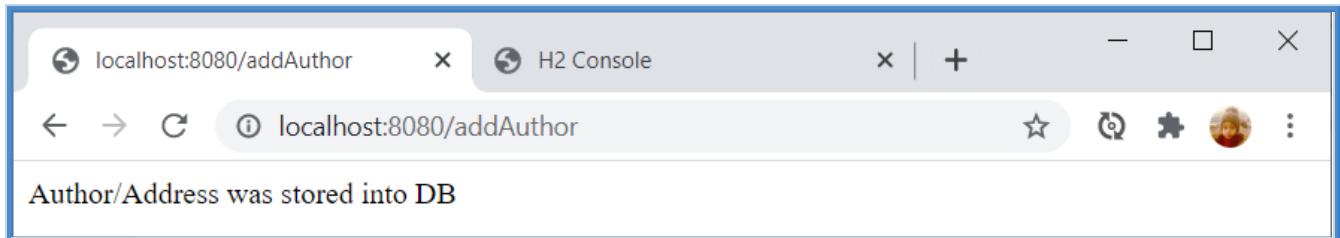
        //CREATE AUTHOR ENTITY
        Author author = new Author();
        author.name = "John";
        author.age = 20;
        author.address = address;

        //STORE AUTHOR/ADDRESS ENTITY INTO DB
        authorRepository.save(author);

        //RETURN SOMETHING TO BROWSER
        return "Author/Address was stored into DB";
    }
}
```

## Results

<http://localhost:8080/AddAuthor>



AUTHOR

The screenshot shows the H2 Console interface for the AUTHOR schema. The left sidebar shows tables: ADDRESS, AUTHOR, and AUTHOR\_ADDRESS. The right pane shows the results of the query `SELECT * FROM AUTHOR;`, which returned one row:

ID	AGE	NAME
20	20	John

(1 row, 3 ms)

ADDRESS

The screenshot shows the H2 Console interface for the ADDRESS schema. The left sidebar shows tables: ADDRESS, AUTHOR, and AUTHOR\_ADDRESS. The right pane shows the results of the query `SELECT * FROM ADDRESS;`, which returned one row:

ID	CITY	STREET
1	London	Piccadilly

(1 row, 5 ms)

(<http://localhost:8080/h2-console>)

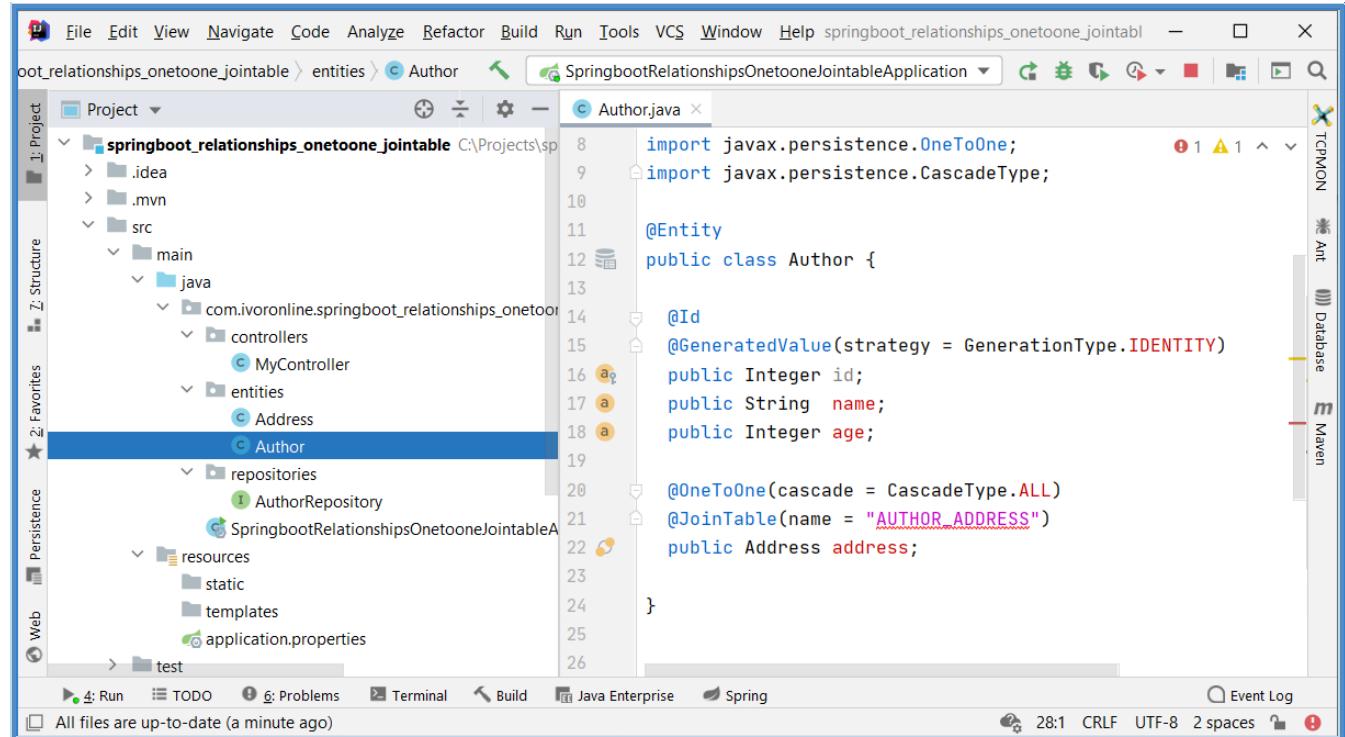
AUTHOR\_ADDRESS

The screenshot shows the H2 Console interface for the AUTHOR\_ADDRESS schema. The left sidebar shows tables: ADDRESS\_ID and ID. The right pane shows the results of the query `SELECT * FROM AUTHOR_ADDRESS;`, which returned one row:

ADDRESS_ID	ID
1	1

(1 row, 3 ms)

## Application Structure



## pom.xml

```
<dependencies>

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

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

    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
    </dependency>

</dependencies>
```

## 2.7.2 @OneToOne - Foreign Key

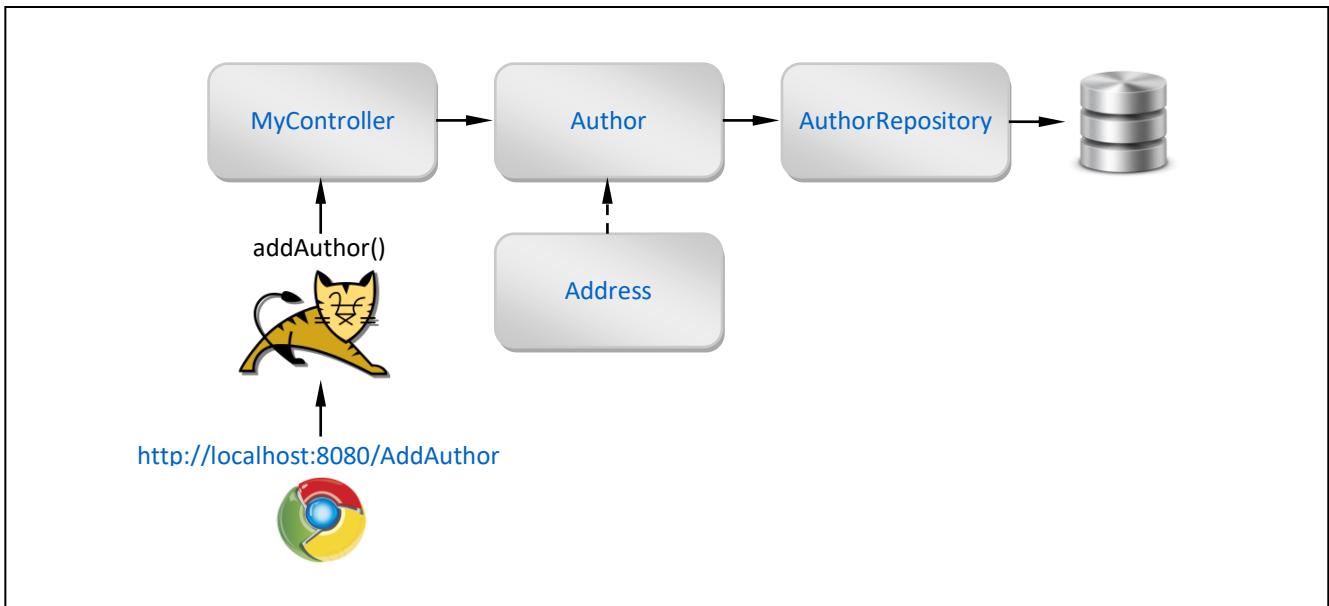
Info

[G] [R]

- **@OneToOne** relationship is used when you want to split data into multiple Tables.  
For instance instead of having single Table with 20 columns you can have 2 Tables with 10 columns each.  
One Table will have an additional column that points to a related Record in the second Table.
- **@OneToOne** relationship can be useful when you want to logically **group data**
  - first Table can contain Customer's basic information like: FirstName, LastName, Age
  - second Table can contain Customer's contact information like: Email, Phone, Address

Application Schema

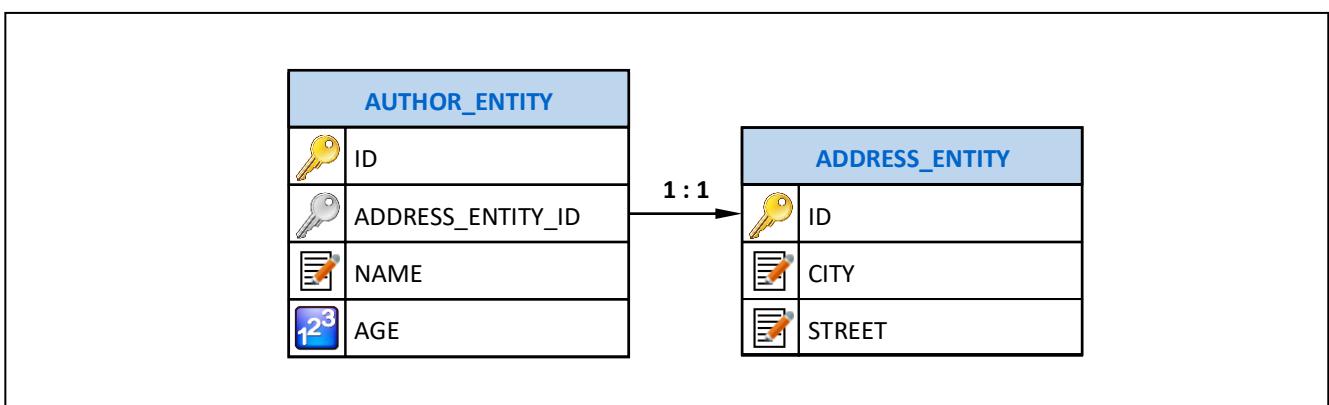
[Results]



Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables @Controller and @RequestMapping. Includes Tomcat Server.
SQL	Spring Data JPA	Enables @Entity and @Id
SQL	H2 Database	Enables in-memory H2 Database

DB Schema



Syntax

```
@OneToOne(cascade = CascadeType.ALL)  
public AddressEntity addressEntity;
```

## Procedure

- Create Project: relationships\_onetoone (add Spring Boot Starters from the table)
- Edit File: application.properties (specify H2 DB name & enable H2 Web Console)
- Create Package: entities (inside main package)
  - Create Class: Author.java (inside package entities)
  - Create Class: Address.java (inside package entities)
- Create Package: repositories (inside main package)
  - Create Interface: AuthorRepository.java (inside package repositories)
- Create Package: controllers (inside main package)
  - Create Class: MyController.java (inside package controllers)

*application.properties*

```
spring.datasource.url      = jdbc:h2:mem:testdb
spring.h2.console.enabled = true
```

*Author.java*

```
package com.ivoronline.relationships_onetoone.entities;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.OneToOne;
import javax.persistence.CascadeType;

@Entity
public class Author {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    public Integer id;
    public String name;
    public Integer age;

    @OneToOne(cascade = CascadeType.ALL)
    public Address address;

}
```

*Address.java*

```
package com.ivoronline.relationships_onetoone.entities;

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

@Entity
public class Address {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    public Integer id;
    public String city;
    public String street;

}
```

### *AuthorRepository.java*

```
package com.ivorononline.relationships_onetoone.repositories;

import com.ivorononline.relationships_onetoone.entities.Author;
import org.springframework.data.repository.CrudRepository;

public interface AuthorRepository extends CrudRepository<Author, Integer> { }
```

### *MyController.java*

```
package com.ivorononline.relationships_onetoone.controllers;

import com.ivorononline.relationships_onetoone.entities.Author;
import com.ivorononline.relationships_onetoone.entities.Address;
import com.ivorononline.relationships_onetoone.repositories.AuthorRepository;
import org.springframework.stereotype.Controller;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class MyController {

    @Autowired
    AuthorRepository authorRepository;

    @ResponseBody
    @RequestMapping("/AddAuthor")
    public String addAuthor() {

        //CREATE ADDRESS ENTITY
        Address address = new Address();
        address.city = "London";
        address.street = "Piccadilly";

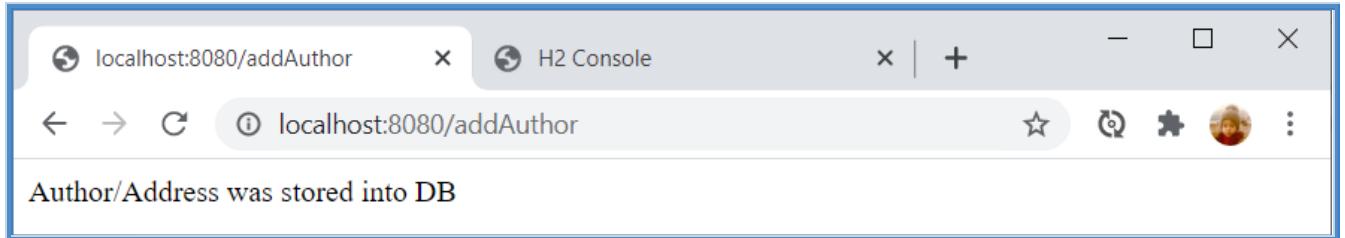
        //CREATE AUTHOR ENTITY
        Author author = new Author();
        author.name = "John";
        author.age = 20;
        author.address = address;

        //STORE AUTHOR/ADDRESS ENTITY INTO DB
        authorRepository.save(author);

        //RETURN SOMETHING TO BROWSER
        return "Author/Address was stored into DB";
    }
}
```

## Results

<http://localhost:8080/AddAuthor>



<http://localhost:8080/h2-console>

(Open H2 Console)

Two side-by-side screenshots of the H2 Console interface. Both windows show the same database structure and data.

**Left Window:**

- SQL Editor: `SELECT * FROM AUTHOR_ENTITY;`
- Result Set:

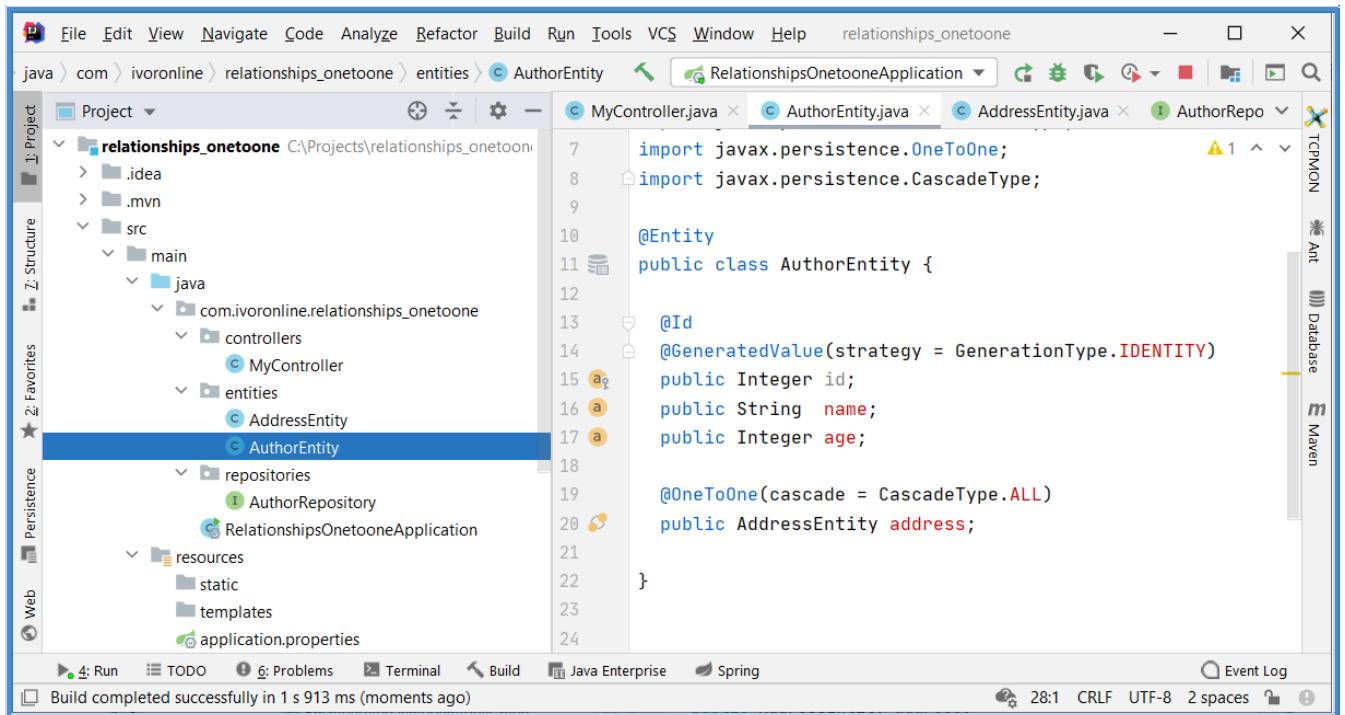
ID	AGE	NAME	ADDRESS_ID
1	20	John	1

**Right Window:**

- SQL Editor: `SELECT * FROM ADDRESS_ENTITY;`
- Result Set:

ID	CITY	STREET
1	London	Piccadilly

## Application Structure



```
<dependencies>

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

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

    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
    </dependency>

</dependencies>
```

## 2.7.3 @OneToMany - Join Table

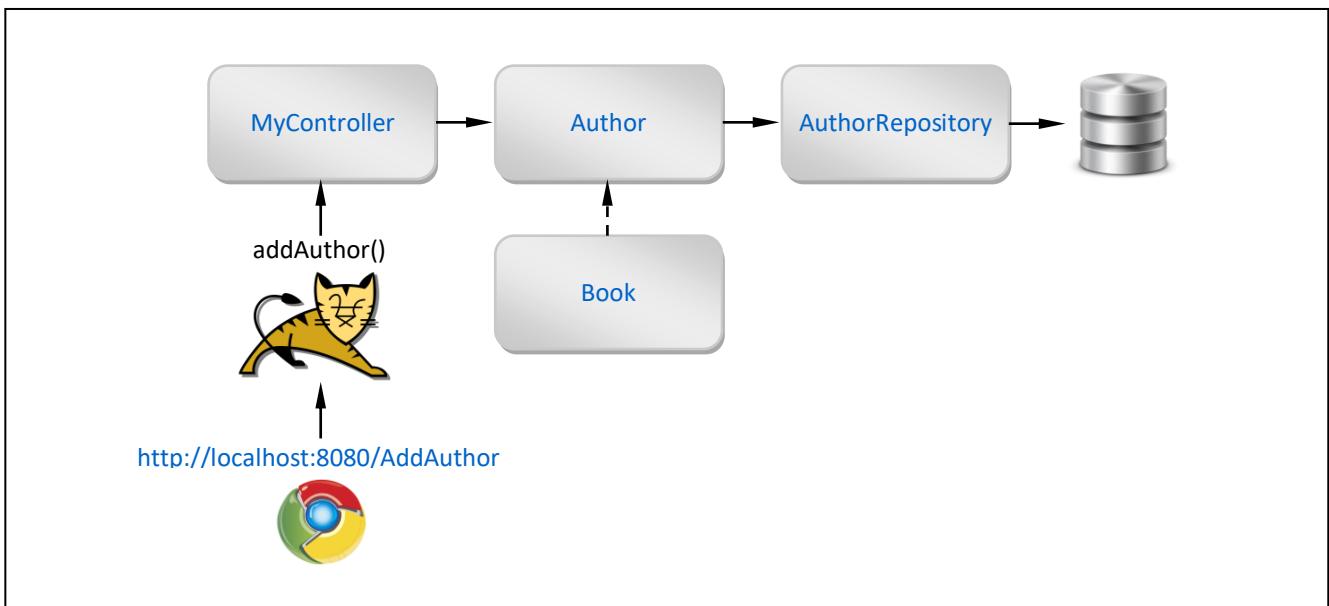
### Info

[G] [R]

- This tutorial shows how to implement **@OneToMany** relationship by using **Join Table AUTHOR\_BOOKS**.
- This table will hold information which Author has written which Books.

### Application Schema

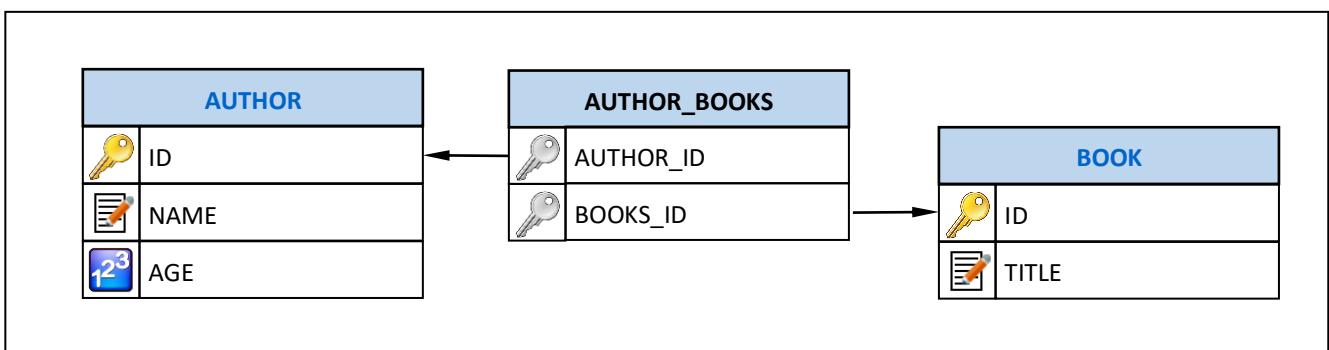
[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables @Controller and @RequestMapping. Includes Tomcat Server.
SQL	Spring Data JPA	Enables @Entity and @Id
SQL	H2 Database	Enables in-memory H2 Database

### DB Schema



### Syntax

```
@OneToMany(cascade = CascadeType.ALL)
public Address address;
```

## Procedure

- **Create Project:** relationships\_onetomany (add Spring Boot Starters from the table)
- **Edit File:** `application.properties` (specify H2 DB name & enable H2 Web Console)
- **Create Package:** entities (inside main package)
  - **Create Class:** `Author.java` (inside package entities)
  - **Create Class:** `Book.java` (inside package entities)
- **Create Package:** repositories (inside main package)
  - **Create Interface:** `AuthorRepository.java` (inside package repositories)
- **Create Package:** controllers (inside main package)
  - **Create Class:** `MyController.java` (inside package controllers)

*application.properties*

```
spring.datasource.url      = jdbc:h2:mem:testdb
spring.h2.console.enabled = true
```

*Author.java*

```
package com.ivoronline.relationships_onetomany.entities;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.OneToMany;
import javax.persistence.CascadeType;
import java.util.Set;

@Entity
public class Author {

    //PRIMARY KEY
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    public Integer id;

    //RELATIONSHIPS
    @OneToMany(cascade = CascadeType.ALL)
    public Set<Book> books;

    //DATA
    public String name;
    public Integer age;

}
```

*AuthorRepository.java*

```
package com.ivoronline.relationships_onetomany.repositories;

import com.ivoronline.relationships_onetomany.entities.Author;
import org.springframework.data.repository.CrudRepository;

public interface AuthorRepository extends CrudRepository<Author, Integer> { }
```

### *Book.java*

```
package com.ivoronline.relationships_onetomany.entities;

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

@Entity
public class Book {

    //PRIMARY KEY
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    public Integer id;

    //DATA
    public String title;

}
```

### *MyController.java*

```
package com.ivoronline.relationships_onetomany.controllers;

import com.ivoronline.relationships_onetomany.entities.Author;
import com.ivoronline.relationships_onetomany.entities.Book;
import com.ivoronline.relationships_onetomany.repositories.AuthorRepository;
import org.springframework.stereotype.Controller;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RequestMapping;

import java.util.HashSet;

@Controller
public class MyController {

    @Autowired
    AuthorRepository authorRepository;

    @ResponseBody
    @RequestMapping("/AddAuthor")
    public String addAuthor() {

        //CREATE BOOK ENTITIES
        Book book1 = new Book();
        book1.title = "Book about dogs";

        Book book2 = new Book();
        book2.title = "Book about cats";

        //CREATE AUTHOR ENTITY
        Author author = new Author();
        author.name = "John";
        author.age = 20;
        author.books = new HashSet<Book>();
        author.books.add(book1);
        author.books.add(book2);

        //STORE AUTHOR/BOOKS ENTITIES
        authorRepository.save(author);
    }
}
```

```

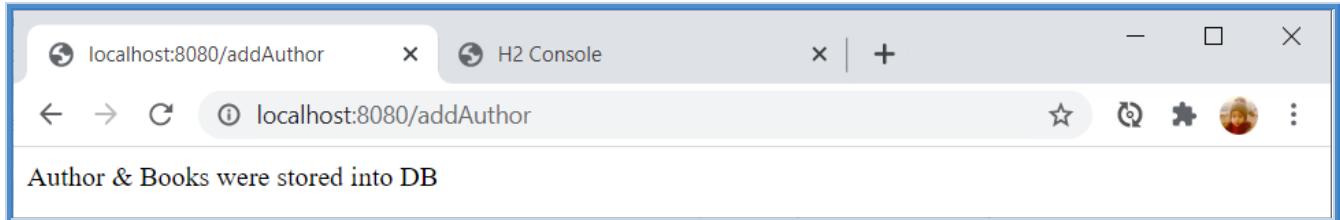
    //RETURN SOMETHING TO BROWSER
    return "Author & Books were stored into DB";

}
}

```

## Results

<http://localhost:8080/AddAuthor>



### AUTHOR

A screenshot of the H2 Console interface for the AUTHOR schema. The left sidebar shows tables: AUTHOR, AUTHOR\_BOOKS, and BOOK. The right panel shows the results of the query `SELECT * FROM AUTHOR;`, which returns one row: ID 1, AGE 20, NAME John.

ID	AGE	NAME
1	20	John

### BOOK

A screenshot of the H2 Console interface for the BOOK schema. The left sidebar shows tables: AUTHOR, AUTHOR\_BOOKS, and BOOK. The right panel shows the results of the query `SELECT * FROM BOOK;`, which returns two rows: ID 1, TITLE Book about dogs; ID 2, TITLE Book about cats.

ID	TITLE
1	Book about dogs
2	Book about cats

<http://localhost:8080/h2-console>

### AUTHOR\_BOOKS

A screenshot of the H2 Console interface for the AUTHOR\_BOOKS schema. The left sidebar shows tables: AUTHOR, AUTHOR\_BOOKS, and BOOK. The right panel shows the results of the query `SELECT * FROM AUTHOR_BOOKS;`, which returns two rows: AUTHOR\_ID 1, BOOKS\_ID 1; AUTHOR\_ID 1, BOOKS\_ID 2.

AUTHOR_ID	BOOKS_ID
1	1
1	2

## Application Structure

The screenshot shows the IntelliJ IDEA interface. The left sidebar displays the project structure under 'relationships\_onetomany'. The code editor on the right shows the Java file `Author.java` with the following content:

```
11  @Entity
12  public class Author {
13
14      //PRIMARY KEY
15      @Id
16      @GeneratedValue(strategy = GenerationType.IDENTITY)
17      public Integer id;
18
19      //RELATIONSHIPS
20      @OneToMany(cascade = CascadeType.ALL)
21      public Set<Book> books;
22
23      //DATA
24      public String name;
25      public Integer age;
26
27 }
```

The code editor has syntax highlighting and several annotations are underlined with yellow circles, indicating potential issues or warnings.

## pom.xml

```
<dependencies>

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

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

    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
    </dependency>

</dependencies>
```

## 2.7.4 @OneToMany - Foreign Key

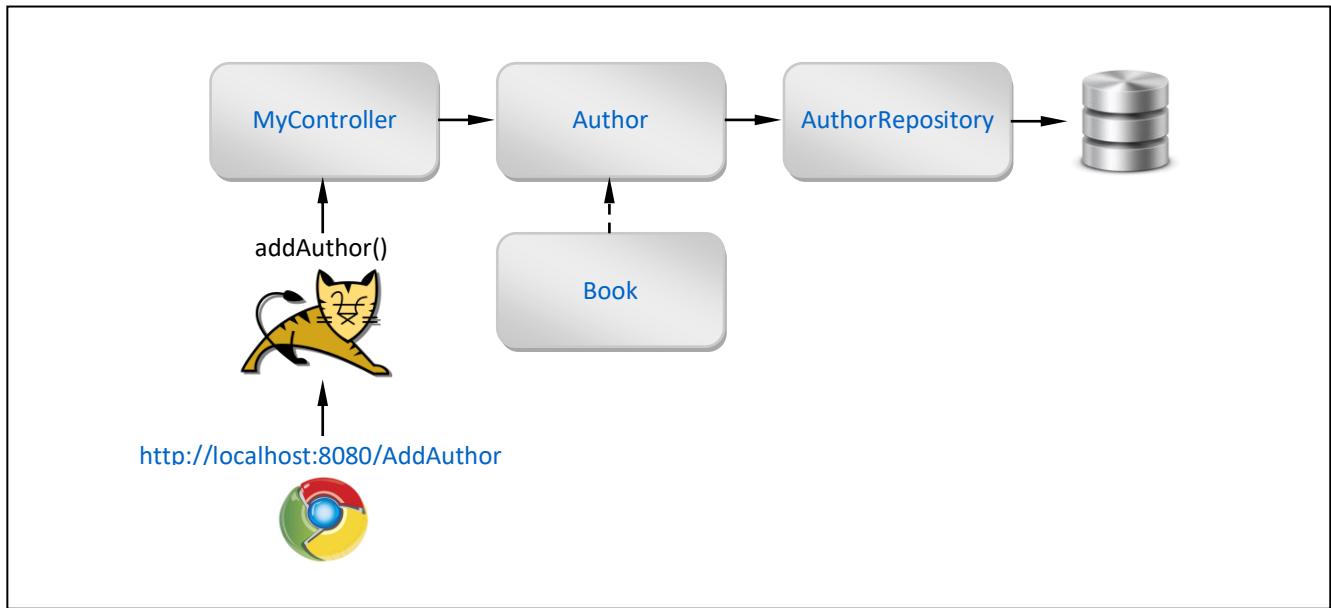
### Info

[G] [R]

- This tutorial shows how to implement **@OneToMany** relationship (between Author and Books).
- Single Author can be related to many Books - by adding Foreign Key Column **authorId** to Book Table.
- Compared to [@OneToMany - Join Table](#) we have only add following two lines
  - In `Book.java` `public Integer authorId;` (add Column to store Foreign Key to Author)
  - In `Author.java` `@JoinColumn(name = "authorId")` (specify which Book Column to use as Foreign Key)

### Application Schema

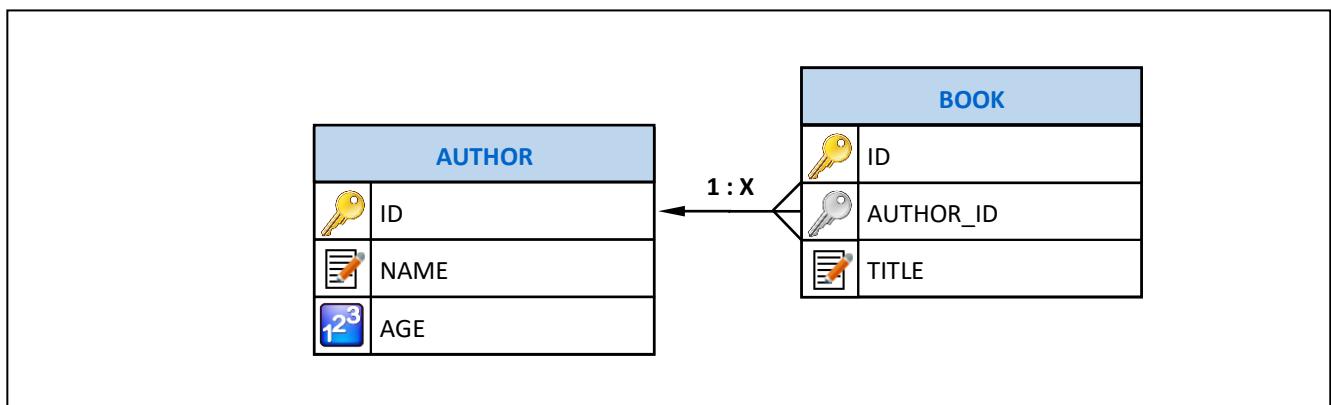
[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables <code>@Controller</code> and <code>@RequestMapping</code> . Includes Tomcat Server.
SQL	Spring Data JPA	Enables <code>@Entity</code> and <code>@Id</code>
SQL	H2 Database	Enables in-memory H2 Database

### DB Schema



### *Book.java*

```
public Integer authorId; //Add Column to Book to store Foreign Key to Author
```

### *Author.java*

```
@OneToMany(cascade = CascadeType.ALL) //Specify Relationship (By default uses Join-Table)
@JoinColumn(name = "authorId") //Specify Foreign Key Column in Book (Overrides default behaviour)
public Set<Book> books; //Store related Books when getting Author from DB
```

## Procedure

- Create Project: [springboot\\_relationships\\_onetoone\\_foreignkey](#) (add Spring Boot Starters from the table)
- Edit File: [application.properties](#) (specify H2 DB name & enable H2 Web Console)
- Create Package: [entities](#) (inside main package)
  - Create Class: [Author.java](#) (inside package entities)
  - Create Class: [Book.java](#) (inside package entities)
- Create Package: [repositories](#) (inside main package)
- Create Interface: [AuthorRepository.java](#) (inside package repositories)
- Create Package: [controllers](#) (inside main package)
- Create Class: [MyController.java](#) (inside package controllers)

### *application.properties*

```
spring.datasource.url      = jdbc:h2:mem:testdb
spring.h2.console.enabled = true
```

### *Author.java*

```
package com.ivoronline.springboot_relationships_onetomany_foreignkey.entities;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.JoinColumn;
import javax.persistence.OneToMany;
import javax.persistence.CascadeType;
import java.util.Set;

@Entity
public class Author {

    //PRIMARY KEY
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    public Integer id;

    //RELATIONSHIPS
    @OneToMany(cascade = CascadeType.ALL)
    @JoinColumn(name = "authorId")
    public Set<Book> books;

    //DATA
    public String name;
    public Integer age;

}
```

### *AuthorRepository.java*

```
package com.ivorononline.relationships_onetomany.repositories;

import com.ivorononline.relationships_onetomany.entities.Author;
import org.springframework.data.repository.CrudRepository;

public interface AuthorRepository extends CrudRepository<Author, Integer> { }
```

### *Book.java*

```
package com.ivorononline.springboot_relationships_onetomany_foreignkey.entities;

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

@Entity
public class Book {

    //PRIMARY KEY
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    public Integer id;

    //FOREIGN KEY
    public Integer authorId;

    //DATA
    public String title;

}
```

### MyController.java

```
package com.ivorononline.relationships_onetomany.controllers;

import com.ivorononline.relationships_onetomany.entities.Author;
import com.ivorononline.relationships_onetomany.entities.Book;
import com.ivorononline.relationships_onetomany.repositories.AuthorRepository;
import org.springframework.stereotype.Controller;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RequestMapping;

import java.util.HashSet;

@Controller
public class MyController {

    @Autowired
    AuthorRepository authorRepository;

    @ResponseBody
    @RequestMapping("/AddAuthor")
    public String addAuthor() {

        //CREATE BOOK ENTITIES
        Book book1 = new Book();
        book1.title = "Book about dogs";

        Book book2 = new Book();
        book2.title = "Book about cats";

        //CREATE AUTHOR ENTITY
        Author author = new Author();
        author.name = "John";
        author.age = 20;
        author.books = new HashSet<Book>();
        author.books.add(book1);
        author.books.add(book2);

        //STORE AUTHOR/BOOKS ENTITIES
        authorRepository.save(author);

        //RETURN SOMETHING TO BROWSER
        return "Author & Books were stored into DB";
    }
}
```

## Results

<http://localhost:8080/AddAuthor>

A screenshot of a browser window titled "localhost:8080/AddAuthor". The address bar shows "localhost:8080/addAuthor". The main content area displays the message "Author & Books were stored into DB".

### AUTHOR

A screenshot of the H2 Console showing the "AUTHOR" table. The table has columns ID, AGE, and NAME. There is one row: ID 1, AGE 20, NAME John.

ID	AGE	NAME
1	20	John

### BOOK

A screenshot of the H2 Console showing the "BOOK" table. The table has columns ID, AUTHOR\_ID, and TITLE. There are two rows: ID 1, AUTHOR\_ID 1, TITLE Book about dogs; ID 2, AUTHOR\_ID 1, TITLE Book about cats.

ID	AUTHOR_ID	TITLE
1	1	Book about dogs
2	1	Book about cats

<http://localhost:8080/h2-console>

## Application Structure

A screenshot of the IntelliJ IDEA IDE. The left side shows the project structure for "springboot\_relationships\_onetomany\_foreignkey". The "entities" package contains "Author.java" and "Book.java". The code editor shows the "Author.java" file:

```
12  @Entity
13  public class Author {
14
15      //PRIMARY KEY
16      @Id
17      @GeneratedValue(strategy = GenerationType.IDENTITY)
18      public Integer id;
19
20      //RELATIONSHIPS
21      @OneToMany(cascade = CascadeType.ALL)
22      @JoinColumn(name = "authorId")
23      public Set<Book> books;
24
25      //DATA
26      public String name;
27      public Integer age;
28
29  }
```

```
<dependencies>

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

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

    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
    </dependency>

</dependencies>
```

## 2.7.5 @ManyToMany - Join Table

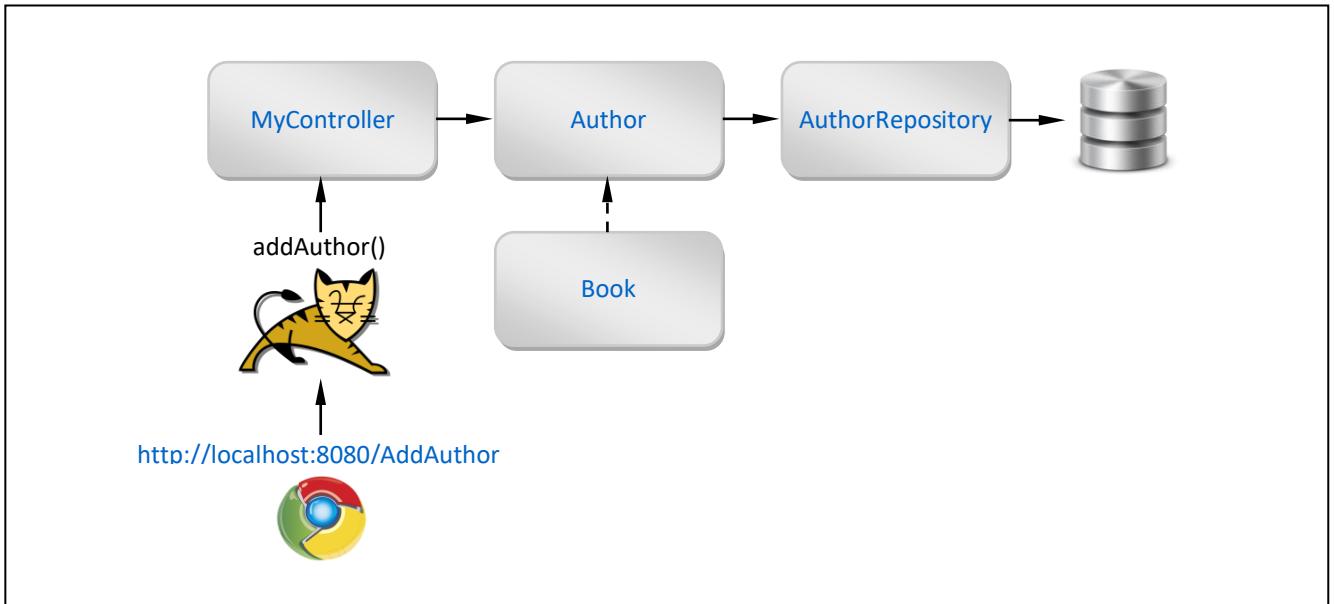
### Info

[G] [R]

- This tutorial shows how to implement **@ManyToMany** relationship by using **Join-Table**.
- @ManyToMany** relationship can't be implemented using Foreign-Key approach.

### Application Schema

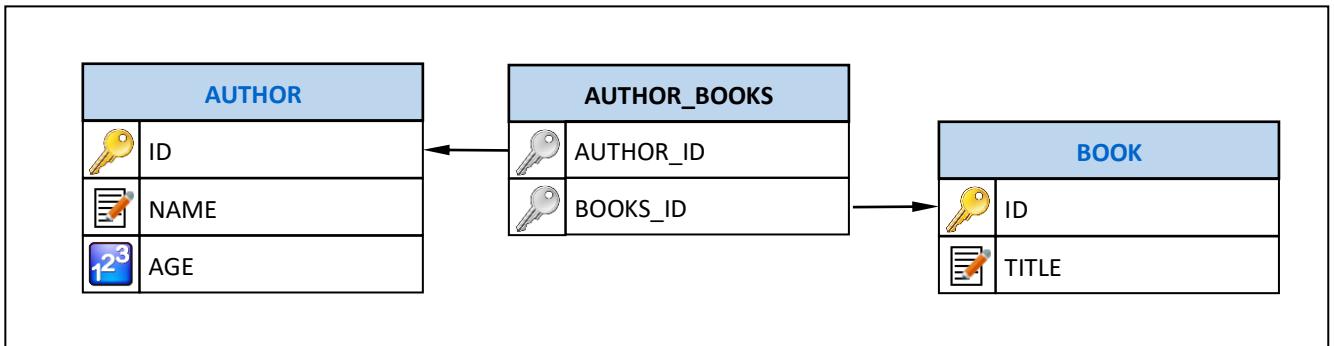
[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables <code>@Controller</code> and <code>@RequestMapping</code> . Includes Tomcat Server.
SQL	Spring Data JPA	Enables <code>@Entity</code> and <code>@Id</code>
SQL	H2 Database	Enables in-memory H2 Database

### DB Schema



### Syntax

```
@OneToOne(cascade = CascadeType.ALL) //By default uses Join-Table
@JoinColumn(name = "authorId") //Overrides default behaviour by using Foreign-Key in Book
public Set<Book> books;
```

## Procedure

- Create Project: [springboot\\_relationships\\_manytoone\\_jointable](#) (add Spring Boot Starters from the table)
- Edit File: [application.properties](#) (specify H2 DB name & enable H2 Web Console)
- Create Package: entities (inside main package)
  - Create Class: [Author.java](#) (inside package entities)
  - Create Class: [Book.java](#) (inside package entities)
- Create Package: repositories (inside main package)
  - Create Interface: [AuthorRepository.java](#) (inside package repositories)
- Create Package: controllers (inside main package)
  - Create Class: [MyController.java](#) (inside package controllers)

*application.properties*

```
spring.datasource.url      = jdbc:h2:mem:testdb
spring.h2.console.enabled = true
```

*AuthorRepository.java*

```
package com.ivoronline.springboot_relationships_manytoone_jointable.repositories;

import com.ivoronline.springboot_relationships_manytoone_jointable.entities.Author;
import org.springframework.data.repository.CrudRepository;

public interface AuthorRepository extends CrudRepository<Author, Integer> { }
```

### *Author.java*

```
package com.ivoronline.springboot_relationships_manytoone_jointable.entities;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.ManyToMany;
import javax.persistence.CascadeType;
import java.util.Set;

@Entity
public class Author {

    //PRIMARY KEY
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    public Integer id;

    //RELATIONSHIP
    @ManyToMany(cascade = CascadeType.ALL)
    public Set<Book> books;

    //DATA
    public String name;
    public Integer age;

}
```

### *Book.java*

```
package com.ivoronline.springboot_relationships_manytoone_jointable.entities;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.ManyToMany;
import java.util.Set;

@Entity
public class Book {

    //PRIMARY KEY
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    public Integer id;

    //RELATIONSHIP
    @ManyToMany(mappedBy="books")
    public Set<Author> authors;

    //DATA
    public String title;

}
```

### MyController.java

```
package com.ivorononline.springboot_relationships_manytoone_jointable.controllers;

import com.ivorononline.springboot_relationships_manytoone_jointable.entities.Author;
import com.ivorononline.springboot_relationships_manytoone_jointable.entities.Book;
import com.ivorononline.springboot_relationships_manytoone_jointable.repositories.AuthorRepository;
import org.springframework.stereotype.Controller;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RequestMapping;

import java.util.HashSet;

@Controller
public class MyController {

    @Autowired
    AuthorRepository authorRepository;

    @ResponseBody
    @RequestMapping("/AddAuthor")
    public String addAuthor() {

        //CREATE BOOKS
        Book book1 = new Book();
        book1.title = "Book about dogs";

        Book book2 = new Book();
        book2.title = "Book about cats";

        //CREATE AUTHORS
        Author author1 = new Author();
        author1.name = "John";
        author1.age = 20;

        Author author2 = new Author();
        author2.name = "Bill";
        author2.age = 30;

        //ADD BOOKS TO AUTHORS
        author1.books = new HashSet<Book>();
        author1.books.add(book1);
        author1.books.add(book2);

        author2.books = new HashSet<Book>();
        author2.books.add(book1);
        author2.books.add(book2);

        //STORE AUTHORS & BOOKS
        authorRepository.save(author1);
        authorRepository.save(author2);

        //RETURN SOMETHING TO BROWSER
        return "Author & Books were stored into DB";
    }
}
```

## Results

<http://localhost:8080/AddAuthor>

A screenshot of a browser window with two tabs: 'localhost:8080/addAuthor' and 'H2 Console'. The main content area displays the message 'Author & Books were stored into DB'.

AUTHOR

A screenshot of the H2 Console showing the AUTHOR schema. The left sidebar lists tables: AUTHOR, AUTHOR\_BOOKS, BOOK, INFORMATION\_SCHEMA, Sequences, and Users. The right panel shows the results of the query 'SELECT \* FROM AUTHOR;' with two rows:

ID	AGE	NAME
1	20	John
2	30	Bill

(2 rows, 7 ms)

BOOK

A screenshot of the H2 Console showing the BOOK schema. The left sidebar lists tables: AUTHOR, AUTHOR\_BOOKS, BOOK, INFORMATION\_SCHEMA, Sequences, and Users. The right panel shows the results of the query 'SELECT \* FROM BOOK;' with two rows:

ID	TITLE
1	Book about cats
2	Book about dogs

(2 rows, 4 ms)

<http://localhost:8080/h2-console>

AUTHOR\_BOOKS

A screenshot of the H2 Console showing the AUTHOR\_BOOKS schema. The left sidebar lists tables: AUTHOR, AUTHOR\_BOOKS, BOOK, INFORMATION\_SCHEMA, Sequences, and Users. The right panel shows the results of the query 'SELECT \* FROM AUTHOR\_BOOKS;' with four rows:

AUTHORS_ID	BOOKS_ID
1	1
1	2
2	1
2	2

(4 rows, 5 ms)

## Application Structure

The screenshot shows the IntelliJ IDEA interface with the following details:

- Project Tree:** The project is named "springboot\_relationships\_manytomany\_jointable". It contains .idea, .mvn, and src folders. src/main/java/com.ivoronline.springboot\_relationships contains controllers (MyController), entities (Author, Book), and repositories (AuthorRepository). src/resources contains static, templates, application.properties, and banner.txt. src/test and target are also visible.
- Code Editor:** The file "Author.java" is open. The code defines an Entity class "Author" with an @Id annotated with @GeneratedValue(strategy = GenerationType.IDENTITY) and a @ManyToMany(cascade = CascadeType.ALL) annotated with @JoinTable(name = "book\_authors"). It also includes fields for name and age.
- Toolbars and Menus:** Standard IntelliJ menus like File, Edit, View, Navigate, Code, Analyze, Refactor, Build, Run, Tools, Git, Window are at the top. A toolbar with icons for file operations is on the right.
- Status Bar:** Shows "Plugin error: Plugin 'Axis TCP Monitor Plugin' (version 1.0.8) was explicitly marked as incompatible ... (a minute ago)" and other build-related information.

## pom.xml

```
<dependencies>

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

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

    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
    </dependency>

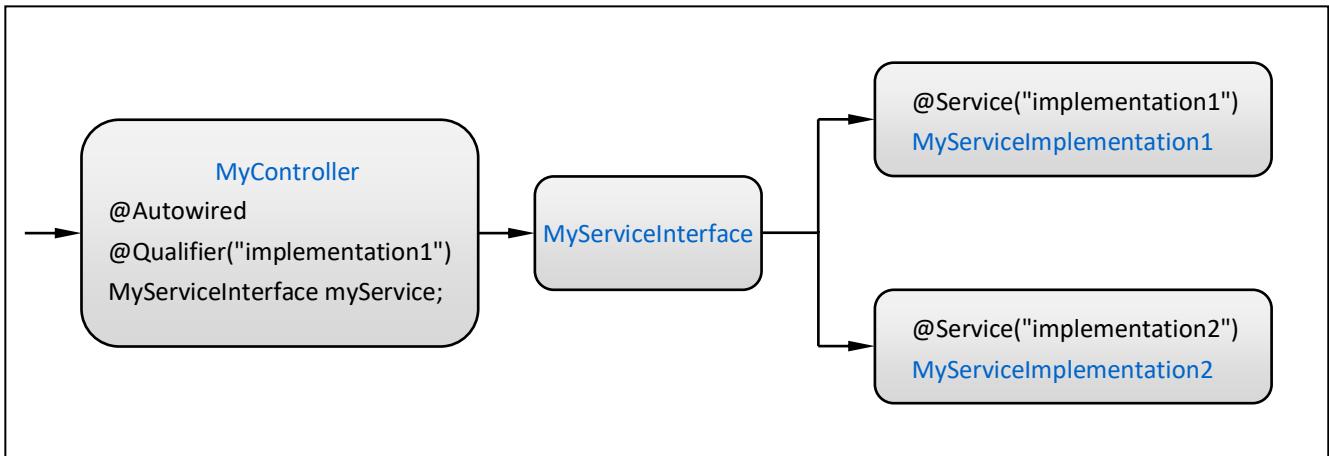
</dependencies>
```

## 2.8 Service

### Info

- Following tutorials explain why and how to use Services (Service Layer).
- Service is Class that implements business logic.
- It is called by Controller and it uses Entities to do its work.

Final Application Schema



## 2.8.1 Business Logic - Inside Controller

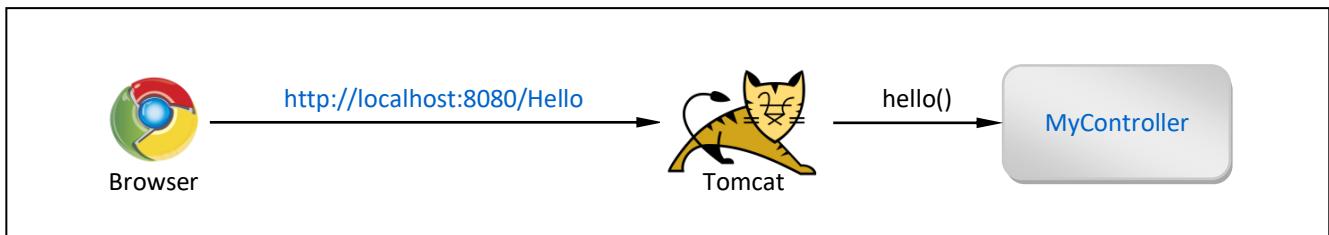
### Info

[G]

- We will start by making Controller that doesn't use Service.
  - Instead business logic is implemented inside the Controller itself.
  - In our example business logic is to return "Hello" String.
  - Such approach is not ideal because each Class should do one thing and one thing only.
- But in this case Controller does two things
  - accepts incoming HTTP Requests
  - implements business logic
- This causes following problems
  - Application becomes harder to maintain and test since everything is in one Class
  - Controller becomes big and unreadable by having all of the business logic placed inside its end points.
  - To change business logic we need to change the Controller potentially introducing bugs which might
    - affect its main purpose of routing HTTP Requests
    - affect other parts of business logic if Controller becomes corrupted

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables: @Controller, @RequestMapping, Tomcat Server

## Procedure

---

- **Create Project:** `springboot_service_controller` (add Spring Boot Starters from the table)
- **Create Package:** `controllers` (inside main package)
- **Create Class:** `MyController.java` (inside package controllers)

`MyController.java`

```
package com.ivoronline.springboot_service_controller.controllers;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class MyController {

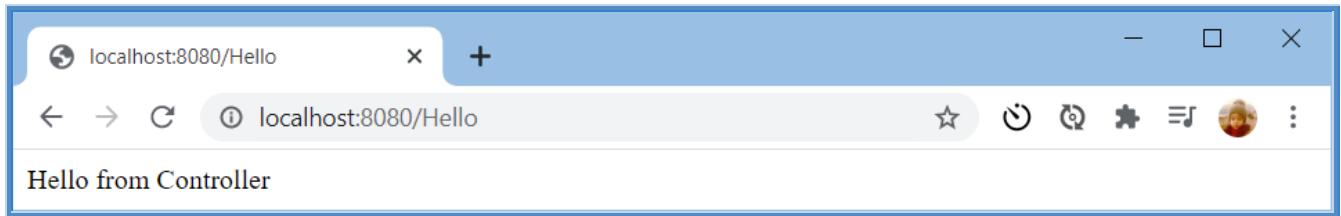
    @ResponseBody
    @RequestMapping("/Hello")
    public String hello() {

        //BUSINESS LOGIC
        String result = "Hello from Controller";

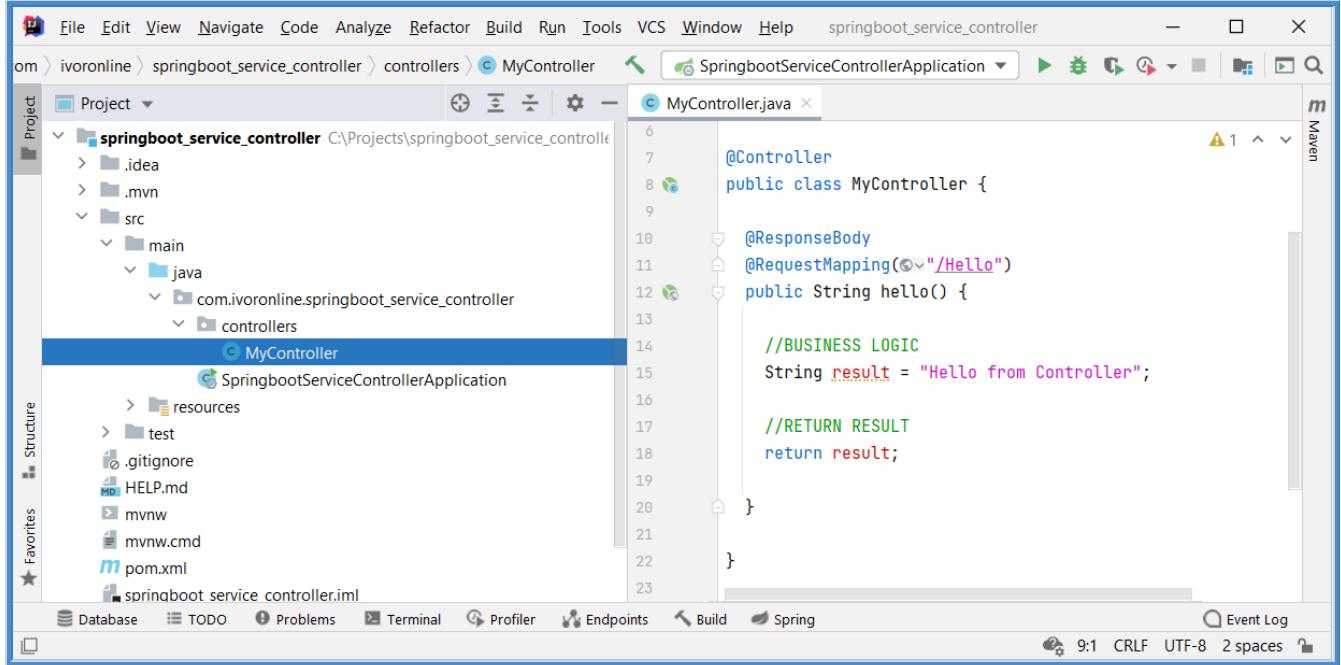
        //RETURN RESULT
        return result;
    }
}
```

## Results

<http://localhost:8080>Hello>



## Application Properties



## pom.xml

```
<dependencies>

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

</dependencies>
```

## 2.8.2 Business Logic - Inside Service

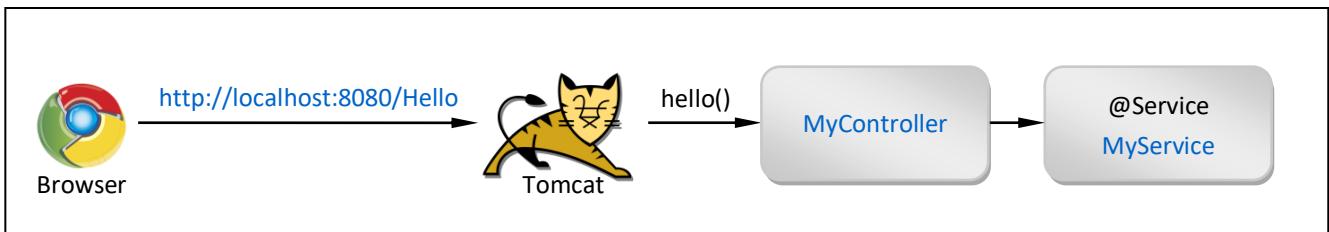
### Info

[G]

- The above problems can be solved by creating separate Service Class for each part of business logic.
- That way we get multiple smaller more manageable Classes each focusing on a specific part of business logic.
- With this approach
  - Controller now has a **single task** of routing incoming HTTP Requests to appropriate Service
  - to change business logic we don't need to change the Controller potentially introducing bugs in it

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables @Controller and @RequestMapping. Includes Tomcat Server.

## Procedure

---

- **Create Project:** `springboot_service` (add Spring Boot Starters from the table)
- **Create Package:** `controllers` (inside main package)
  - **Create Class:** `MyController.java` (inside package controllers)
- **Create Package:** `services` (inside main package)
  - **Create Class:** `MyService.java` (inside package controllers)

### *MyService.java*

```
package com.ivoronline.springboot_service.services;

import org.springframework.stereotype.Service;

@Service
public class MyService {

    public String hello() {
        return "Hello from Service";
    }

}
```

### *MyController.java*

```
package com.ivoronline.springboot_service.controllers;

import com.ivoronline.springboot_service.services.MyService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class MyController {

    @Autowired MyService myService;

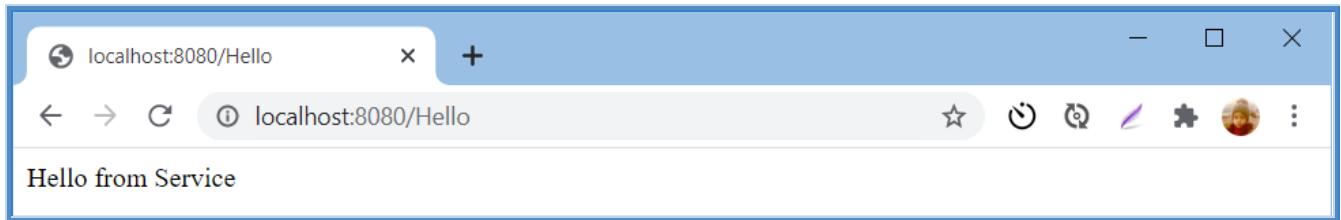
    @ResponseBody
    @RequestMapping("/Hello")
    public String hello() {

        //CALL SERVICE
        String result = myService.hello();

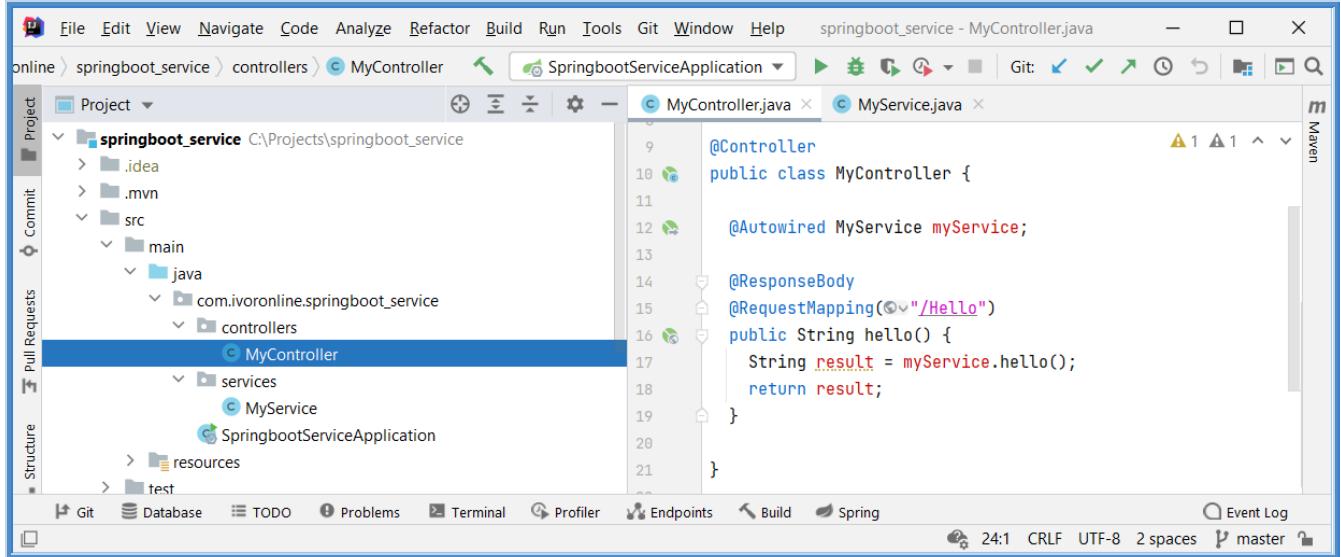
        //RETURN RESULT
        return result;
    }
}
```

## Results

<http://localhost:8080>Hello>



## Application Structure



## 2.8.3 Instantiate Service - Using Class

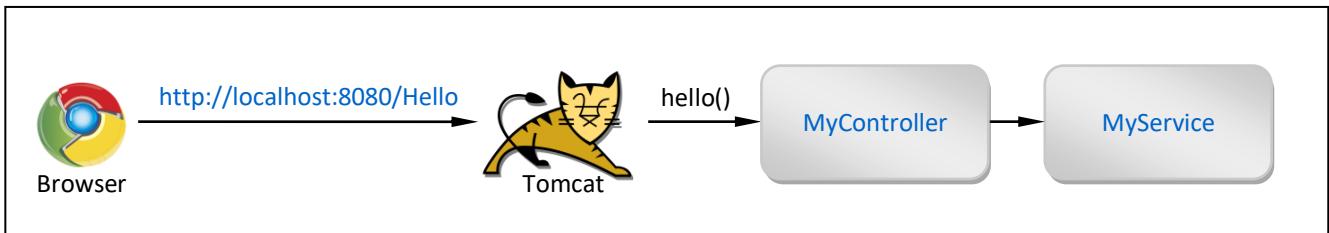
### Info

[G]

- In this example we will instantiate `MyService` Class as Property inside `MyController`.

Application Schema

[Results]



Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables: @Controller, @RequestMapping, Tomcat Server

## Procedure

---

- **Create Project:** `springboot_service_class` (add Spring Boot Starters from the table)
- **Create Package:** `controllers` (inside main package)
  - **Create Class:** `MyController.java` (inside package controllers)
- **Create Package:** `services` (inside main package)
  - **Create Class:** `MyService.java` (inside package controllers)

### *MyService.java*

```
package com.ivoronline.springboot_service_class.services;

import org.springframework.stereotype.Service;

@Service
public class MyService {

    public String hello() {
        return "Hello from Service";
    }
}
```

### *MyController.java*

```
package com.ivoronline.springboot_service_class.controllers;

import com.ivoronline.springboot_service_class.services.MyService;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class MyController {

    MyService myService = new MyService();

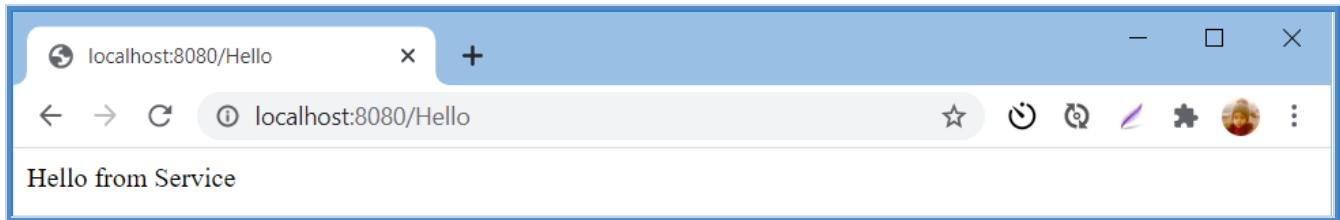
    @ResponseBody
    @RequestMapping("/Hello")
    public String hello() {

        //CALL SERVICE
        String result = myService.hello();

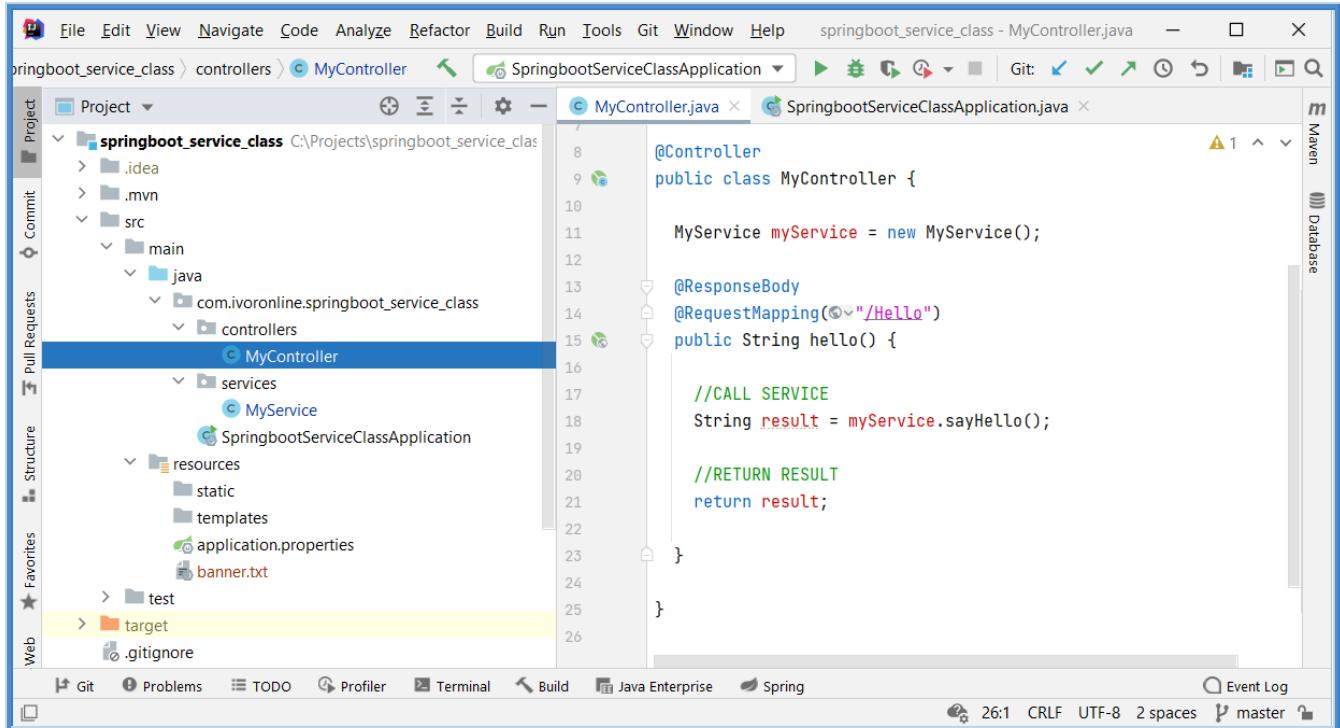
        //RETURN RESULT
        return result;
    }
}
```

## Results

<http://localhost:8080>Hello>



## Application Structure



## pom.xml

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

## 2.8.4 Instantiate Service - Using Class - @Autowired

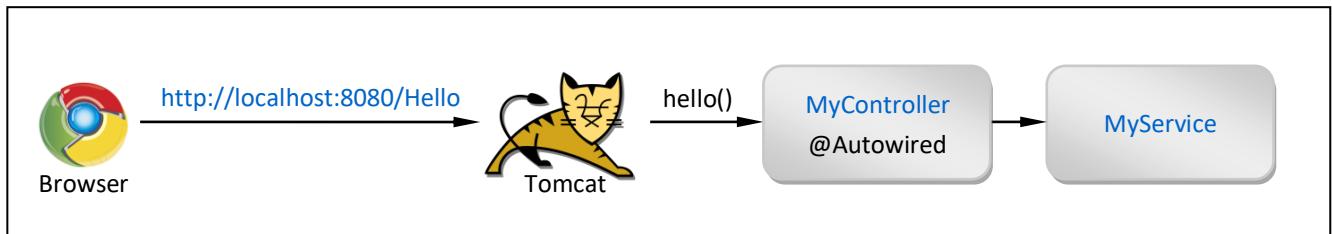
### Info

[G]

- Previous example can be simplified by using `@Autowired` to tell Spring to create Service Instance.  
This way we don't need to manually create Service Instance inside the Controller's endpoint.
- But with this approach we are tightly coupling Controller with specific Service implementation.  
This means that if we want to use different Service Class that implements this part of business logic we would still need to make changes to the Controller which increases our workload and potentially introduces bugs in it.  
So a more elegant solution would be to use Service Interface inside the Controller.

Application Schema

[Results]



Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables <code>@Controller</code> and <code>@RequestMapping</code> . Includes Tomcat Server.

## Procedure

---

- Create Project: `springboot_service_class` (add Spring Boot Starters from the table)
- Create Package: `controllers` (inside main package)
  - Create Class: `MyController.java` (inside package controllers)
- Create Package: `services` (inside main package)
  - Create Class: `MyService.java` (inside package controllers)

### *MyService.java*

```
package com.ivoronline.springboot_service_class_awared.services;

import org.springframework.stereotype.Service;

@Service
public class MyService {

    public String hello() {
        return "Hello from Service";
    }
}
```

### *MyController.java*

```
package com.ivoronline.springboot_service_class_awared.controllers;

import com.ivoronline.springboot_service_class_awared.services.MyService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class MyController {

    @Autowired MyService myService;

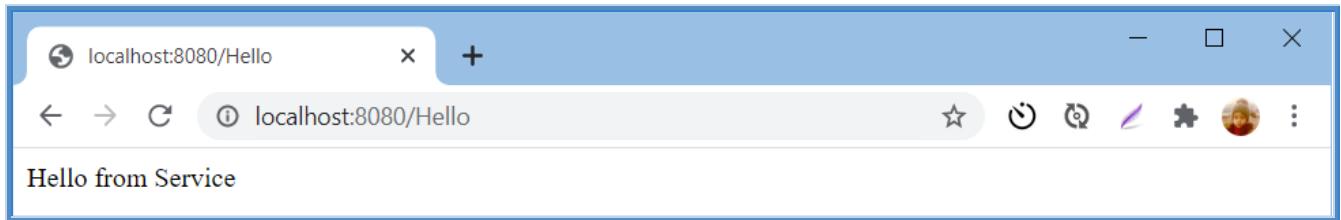
    @ResponseBody
    @RequestMapping("/Hello")
    public String hello() {

        //CALL SERVICE
        String result = myService.hello();

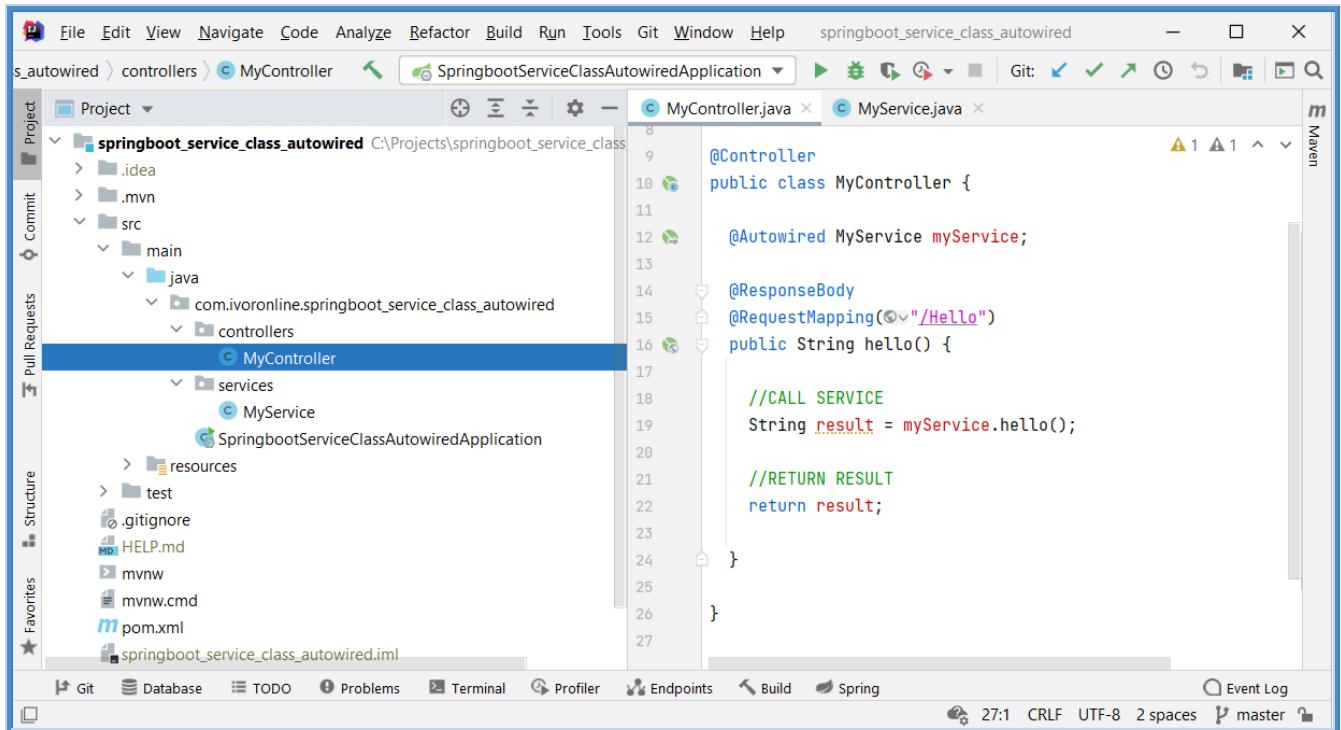
        //RETURN RESULT
        return result;
    }
}
```

## Results

<http://localhost:8080>Hello>



## Application Structure



## pom.xml

```
<dependencies>

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

</dependencies>
```

## 2.8.5 Instantiate Service - Using Interface

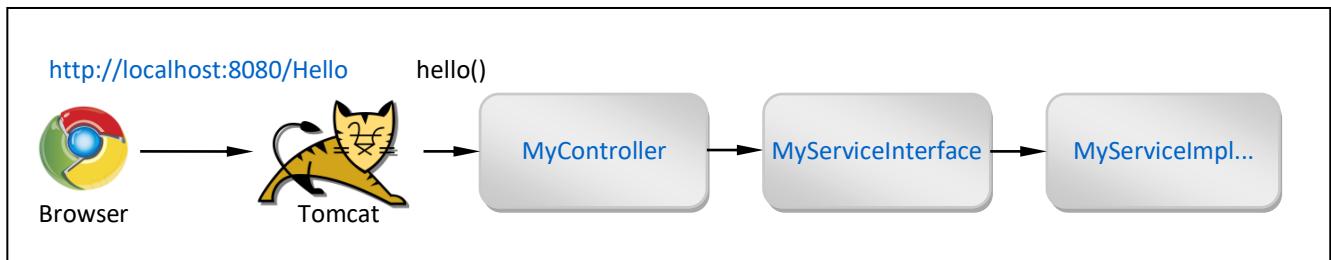
### Info

[G]

- In this step we will introduce Service Interface to decouple Controller from specific Service implementation.
- Inside Controller we will just reference this interface and not any specific implementation of it.
- In other words Spring will create an Object from a Service Class that implements this Service Interface.
- Since we will have only one Service Class that implements this Service Interface Spring will know which one to use.

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables: @Controller, @RequestMapping, Tomcat Server

## Procedure

---

- **Create Project:** `springboot_service_interface` (add Spring Boot Starters from the table)
- **Create Package:** `controllers` (inside main package)
  - **Create Class:** `MyController.java` (inside package controllers)
- **Create Package:** `services` (inside main package)
  - **Create Class:** `MyServiceInterface.java` (inside package controllers)
  - **Create Class:** `MyServiceImplementation.java` (inside package controllers)

*MyServiceInterface.java*

```
package com.ivoronline.springboot_service_interface.services;

public interface MyServiceInterface {
    public String hello();
}
```

*MyServiceImplementation.java*

```
package com.ivoronline.springboot_service_interface.services;

import org.springframework.stereotype.Service;

@Service
public class MyServiceImplementation implements MyServiceInterface {

    public String hello() {
        return "Hello from Service";
    }
}
```

*MyController.java*

```
package com.ivoronline.springboot_service_interface.controllers;

import com.ivoronline.springboot_service_interface.services.MyServiceImplementation;
import com.ivoronline.springboot_service_interface.services.MyServiceInterface;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class MyController {

    MyServiceInterface myService = new MyServiceImplementation();

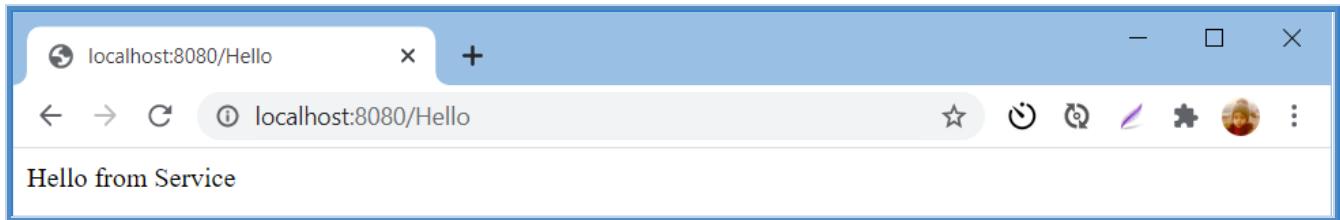
    @ResponseBody
    @RequestMapping("/Hello")
    public String hello() {

        //CALL SERVICE
        String result = myService.hello();

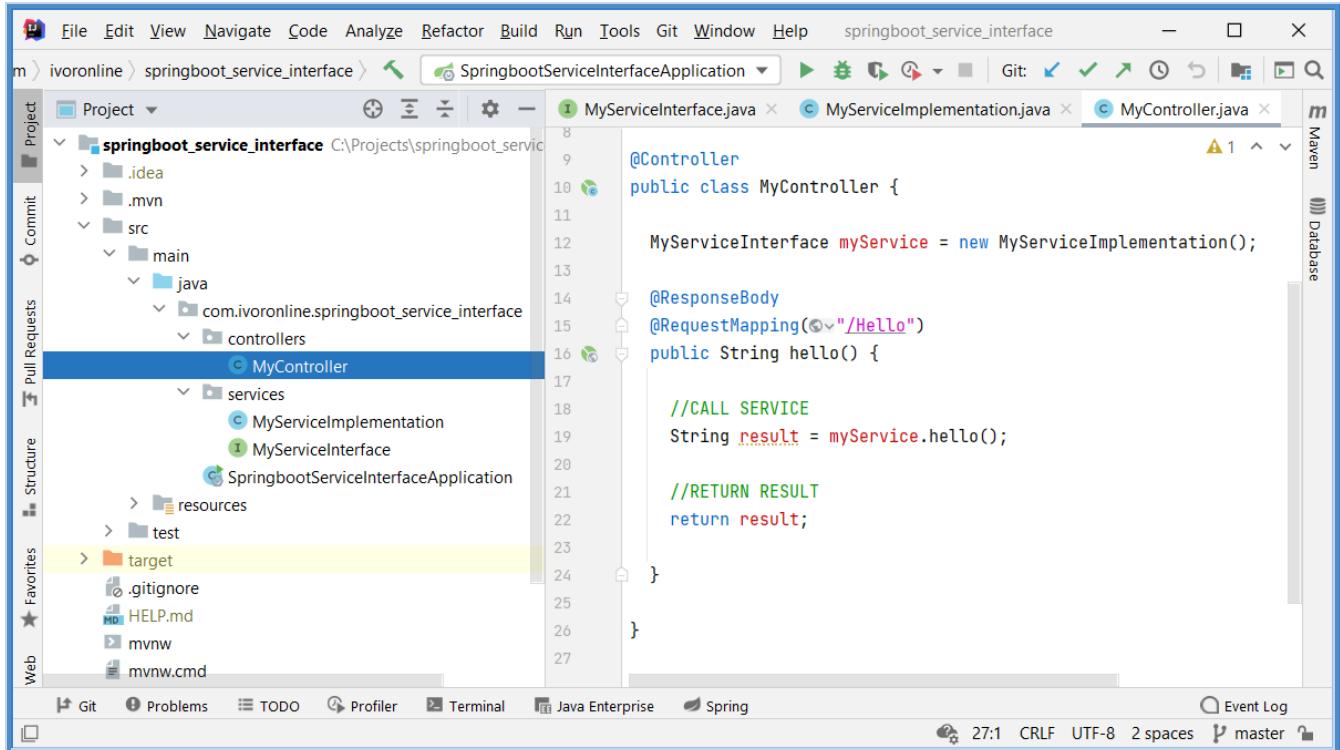
        //RETURN RESULT
        return result;
    }
}
```

## Results

<http://localhost:8080>Hello>



## Application Properties



## pom.xml

```
<dependencies>

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

</dependencies>
```

## 2.8.6 Instantiate Service - Using Interface - @Autowired

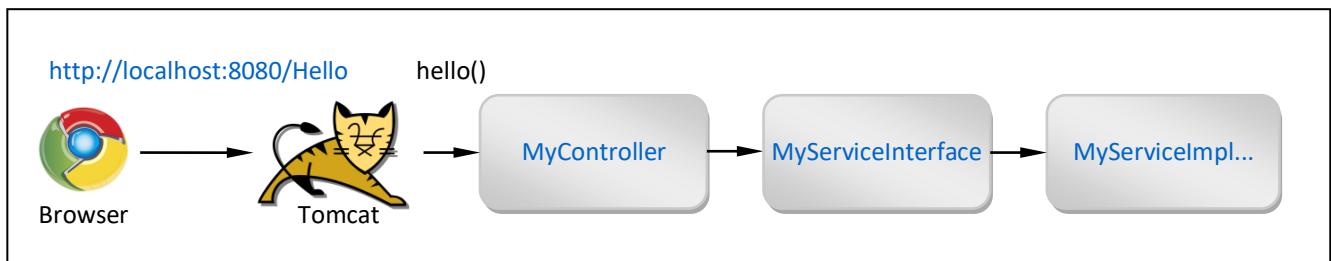
### Info

[G]

- In this step we will introduce Service Interface to decouple Controller from specific Service implementation.
- Inside Controller we will just reference this interface and not any specific implementation of it.
- In other words Spring will create an Object from a Service Class that implements this Service Interface.
- Since we will have only one [Service Class](#) that implements this [Service Interface](#) Spring will know which one to use.
- If there are multiple Classes that Implements the same Interface then you can use [@Primary](#), [@Qualifier](#) or [@Profile](#) to tell Spring from which Implementation to create instance.

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables: @Controller, @RequestMapping, Tomcat Server

## Procedure

---

- **Create Project:** `springboot_service_interface_awtowired` (add Spring Boot Starters from the table)
- **Create Package:** `controllers` (inside main package)
- **Create Class:** `MyController.java` (inside package controllers)
- **Create Package:** `services` (inside main package)
- **Create Class:** `MyServiceInterface.java` (inside package controllers)
- **Create Class:** `MyServiceImplementation.java` (inside package controllers)

*MyServiceInterface.java*

```
package com.ivoronline.springboot_service_interface_awtowired.services;

public interface MyServiceInterface {
    public String hello();
}
```

*MyServiceImplementation.java*

```
package com.ivoronline.springboot_service_interface_awtowired.services;

import org.springframework.stereotype.Service;

@Service
public class MyServiceImplementation implements MyServiceInterface {

    public String hello() {
        return "Hello from Service";
    }
}
```

*MyController.java*

```
package com.ivoronline.springboot_service_interface_awtowired.controllers;

import com.ivoronline.springboot_service_interface_awtowired.services.MyServiceInterface;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class MyController {

    @Autowired MyServiceInterface myService;

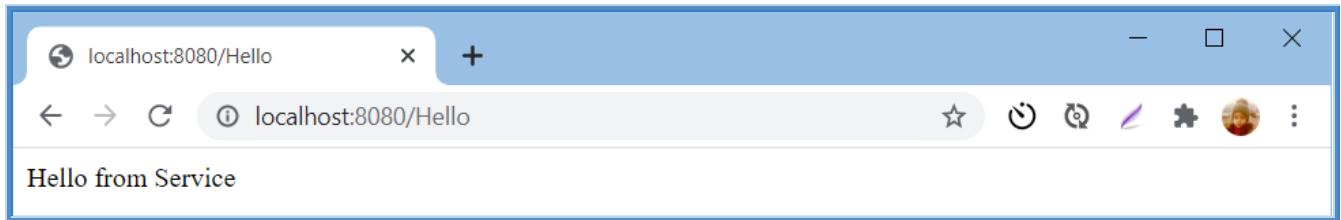
    @ResponseBody
    @RequestMapping("/Hello")
    public String hello() {

        //CALL SERVICE
        String result = myService.hello();

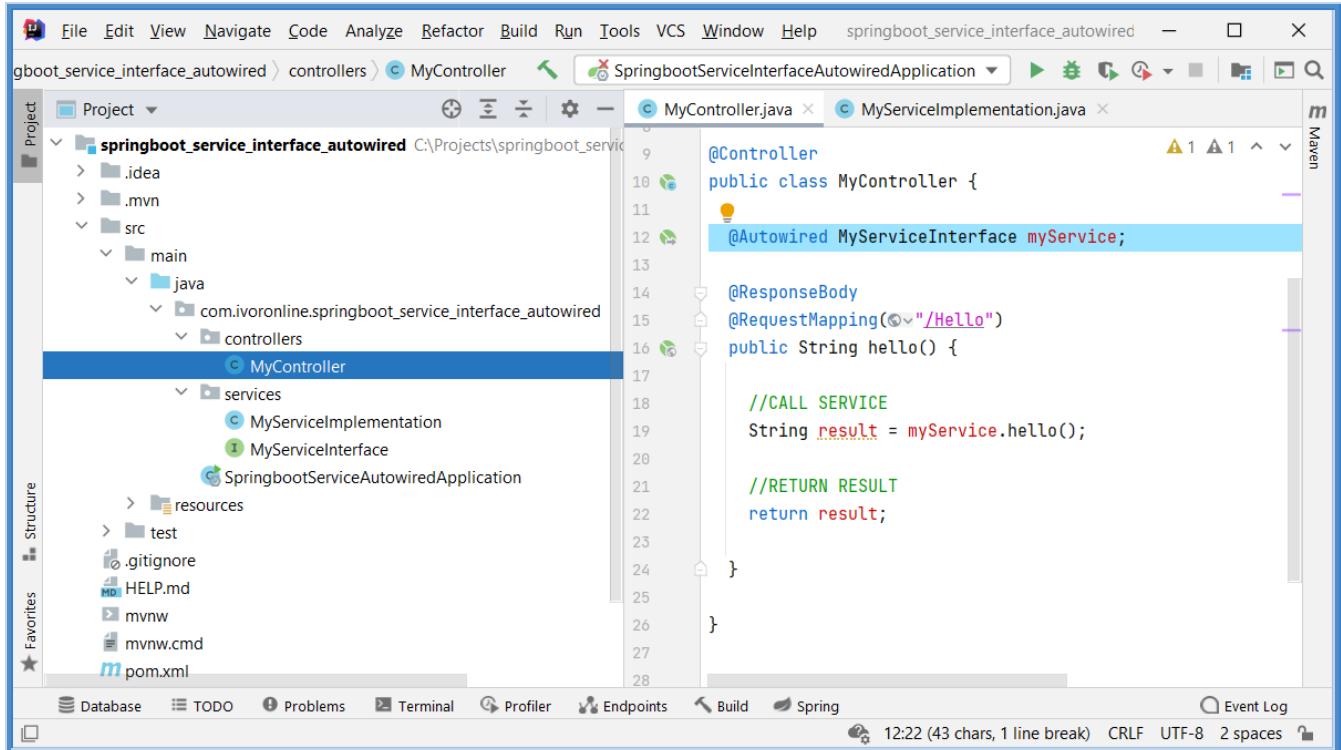
        //RETURN RESULT
        return result;
    }
}
```

## Results

<http://localhost:8080>Hello>



## Application Structure



## pom.xml

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

## 2.9 @Autowired

### Info

---

- `@Autowired` Annotation tells Spring Boot to create/inject Instance of Class into Variable by specifying either
  - **Class**      `@Autowired MyService myService;`
  - **Interface**    `@Autowired MyServiceInterface myService;`
- In both cases Class must be Spring Component/Bean in order for Spring Boot to be able to find it.  
That means that Class must be Annotated with either: `@Controller`, `@Component`, `@Service`.
- `@Autowired` can inject Instance of a Class into
  - **Property**      `@Autowired MyService myService;`
  - **Setter** Param    `@Autowired public void setMyService(MyService myService) { this.myService = myService; }`
  - **Constructor** Param `@Autowired public MyController(MyService myService) { this.myService = myService; }`
- If you specify **Interface** (and there is more than one Class that implements that Interface) you need to tell Spring which Class to instantiate by using any of the following Annotations (or their combination)
  - `@Primary`      - only one Class can be `@Primary` per `@Profile`
  - `@Qualifier("impl2")` - selects named Component `@Service("impl2")`
  - `@Profile("Profile1")` - specify active Profile in `application.properties` with `spring.profiles.active = Profile1`

### Usage

---

- For example `@Autowired` can be used
  - inside **Controller** to instantiate **Services**
  - inside **Service** to instantiate **Entities** and **Repositories**

## 2.9.1 Manually

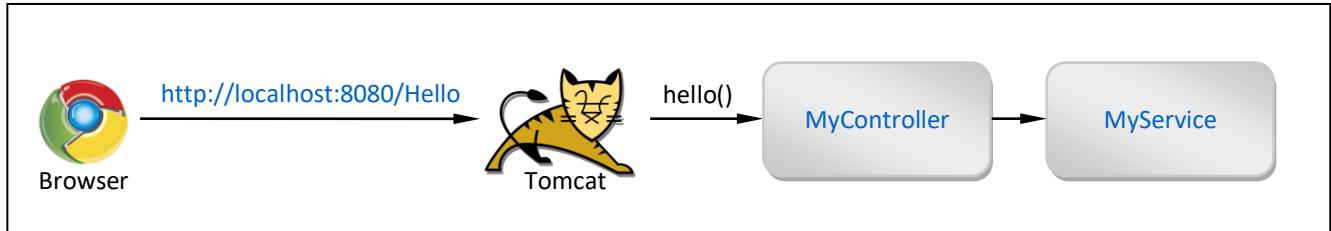
### Info

[G]

- This tutorial shows how to manually instantiate a Class without using `@Autowired`.
- This is useful to know to get better understanding of what is Spring Boot doing in the background.

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables <code>@Controller</code> and <code>@RequestMapping</code> . Includes Tomcat Server.

## Procedure

---

- **Create Project:** `springboot_awtowired_manually` (add Spring Boot Starters from the table)
- **Create Package:** `controllers` (inside main package)
  - **Create Class:** `MyController.java` (inside package controllers)
- **Create Package:** `services` (inside main package)
  - **Create Class:** `MyService.java` (inside package controllers)

### *MyController.java*

```
package com.ivoronline.springboot_awtowired_manually.controllers;

import com.ivoronline.springboot_awtowired_manually.services.MyService;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class MyController {

    // @Autowired
    MyService myService = new MyService();

    @ResponseBody
    @RequestMapping("/Hello")
    public String hello() {
        String Results = myService.sayHello();
        return Results;
    }

}
```

### *MyService.java*

```
package com.ivoronline.springboot_awtowired_manually.services;

import org.springframework.stereotype.Service;

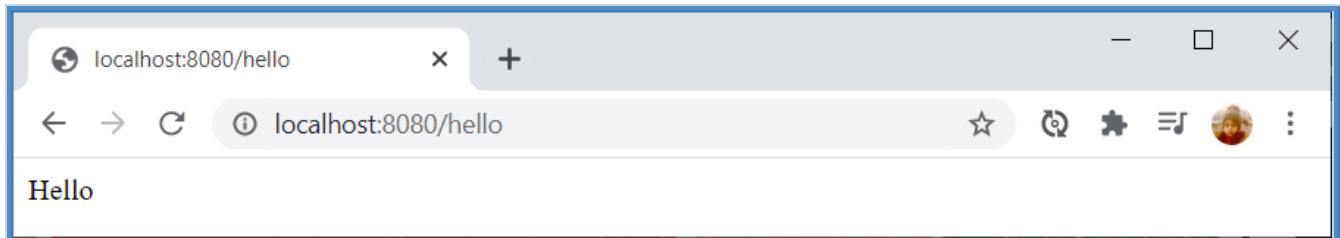
@Service
public class MyService {

    public String sayHello() {
        return "Hello";
    }

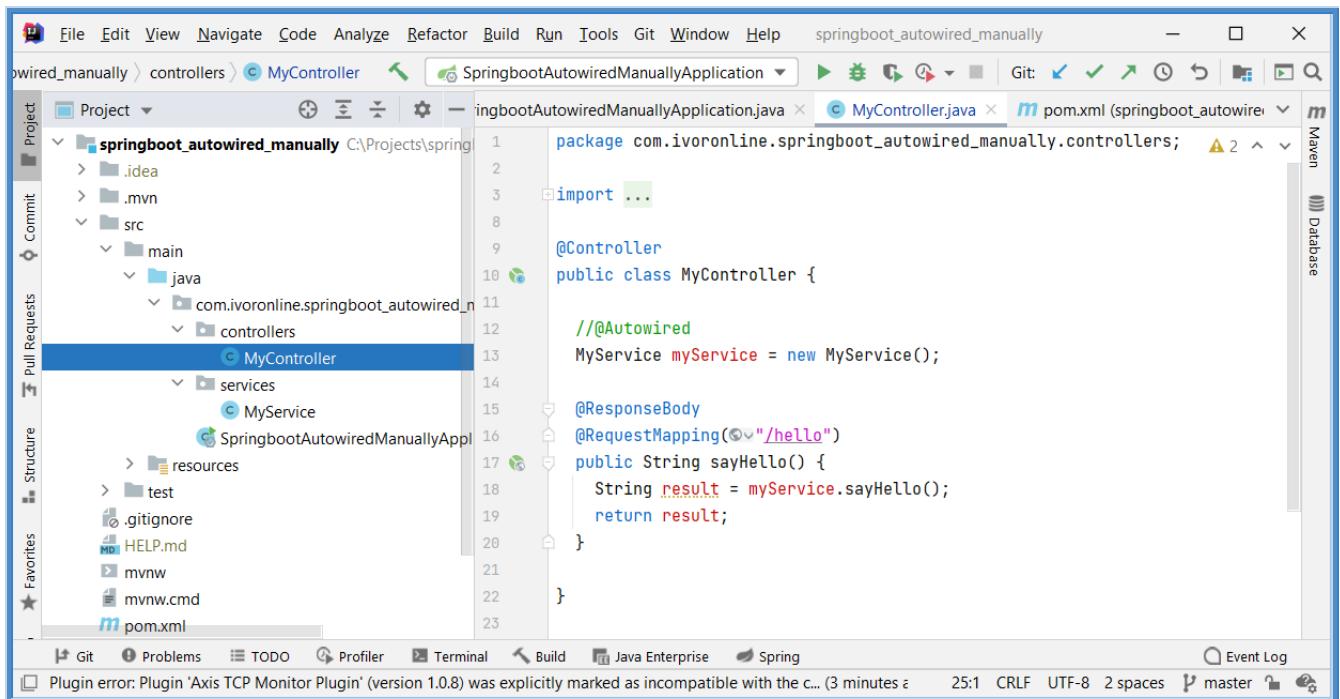
}
```

## Results

<http://localhost:8080>Hello>



## Application Structure



## pom.xml

```
<dependencies>

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

</dependencies>
```

## 2.9.2 Location - Property

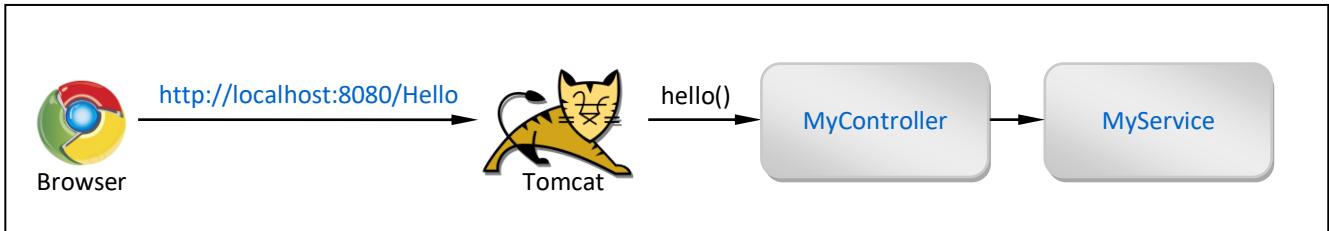
### Info

[G]

- This tutorial shows how to use `@Autowired` to inject instance into Class Property.

Application Schema

[Results]



Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables <code>@Controller</code> and <code>@RequestMapping</code> . Includes Tomcat Server.

## Procedure

---

- **Create Project:** `springboot_autowired_property` (add Spring Boot Starters from the table)
- **Create Package:** `controllers` (inside main package)
  - **Create Class:** `MyController.java` (inside package controllers)
- **Create Package:** `services` (inside main package)
  - **Create Class:** `MyService.java` (inside package controllers)

### *MyController.java*

```
package com.ivoronline.springboot_autowired_property.controllers;

import com.ivoronline.springboot_autowired_property.services.MyService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class MyController {

    @Autowired MyService myService;

    @ResponseBody
    @RequestMapping("/Hello")
    public String hello() {
        String Results = myService.sayHello();
        return Results;
    }
}
```

### *MyService.java*

```
package com.ivoronline.springboot_autowired_property.services;

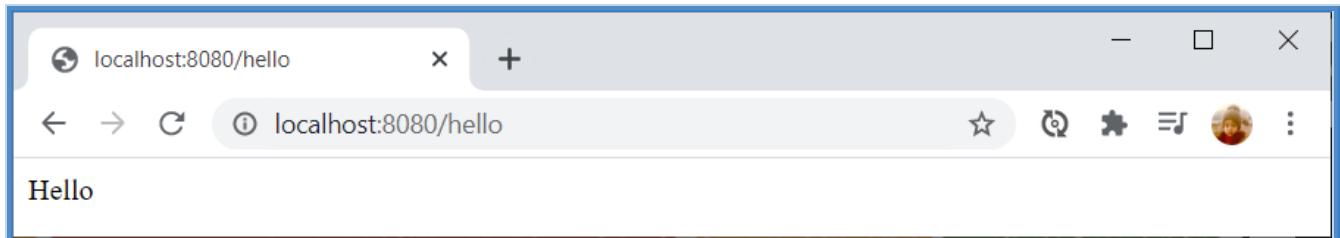
import org.springframework.stereotype.Service;

@Service
public class MyService {

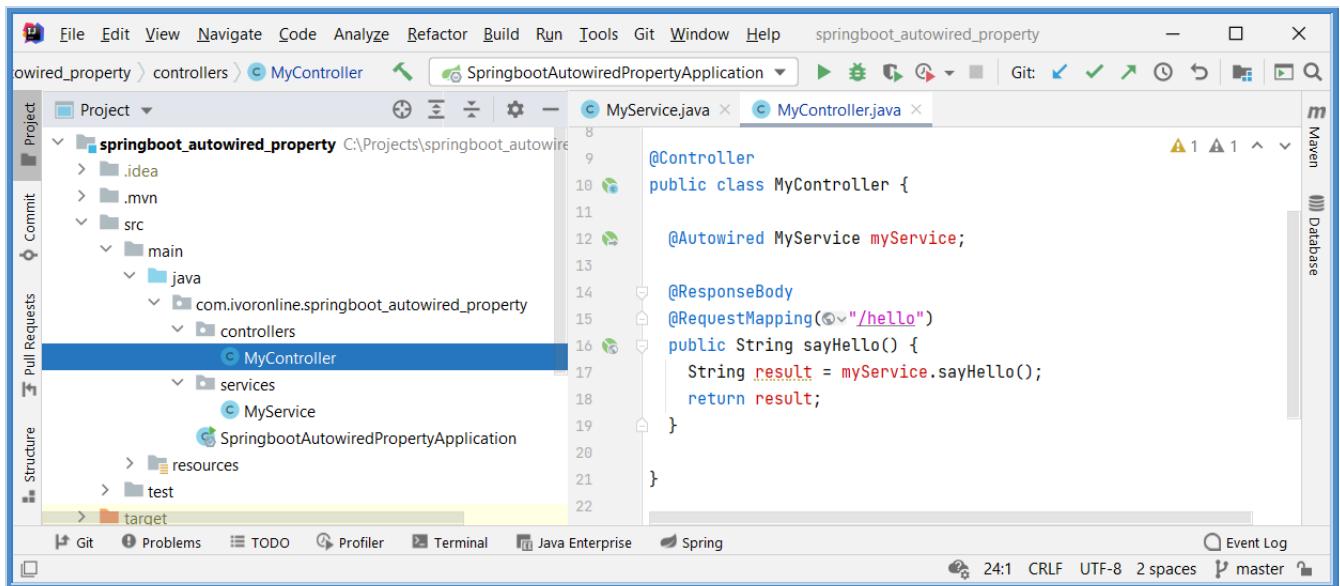
    public String sayHello() {
        return "Hello";
    }
}
```

## Results

<http://localhost:8080>Hello>



## Application Structure



## pom.xml

```
<dependencies>

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

</dependencies>
```

## 2.9.3 Location - Setter

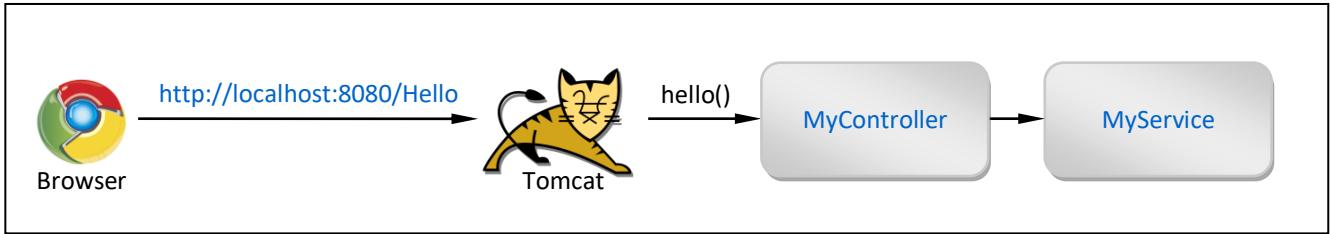
### Info

[G] [R] [R]

- This tutorial shows how to use `@Autowired` to inject instance into Setter's Input Parameter.
- When `@Component` Class is loaded Spring automatically calls all Methods Annotated with `@Autowired`.
- Then it creates instances of Classes/Interfaces that this Method accepts as Input Parameters.

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables <code>@Controller</code> and <code>@RequestMapping</code> . Includes Tomcat Server.

## Procedure

---

- Create Project: `springboot_autowired_setter` (add Spring Boot Starters from the table)
- Create Package: `controllers` (inside main package)
  - Create Class: `MyController.java` (inside package controllers)
- Create Package: `services` (inside main package)
  - Create Class: `MyService.java` (inside package controllers)

### `MyController.java`

```
package com.ivoronline.springboot_autowired_setter.controllers;

import com.ivoronline.springboot_autowired_setter.services.MyService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class MyController {

    MyService myService;

    @Autowired
    public void setMyService(MyService myService) {
        this.myService = myService;
    }

    @ResponseBody
    @RequestMapping("/Hello")
    public String hello() {
        String Results = myService.sayHello();
        return Results;
    }
}
```

### `MyService.java`

```
package com.ivoronline.springboot_autowired_setter.services;

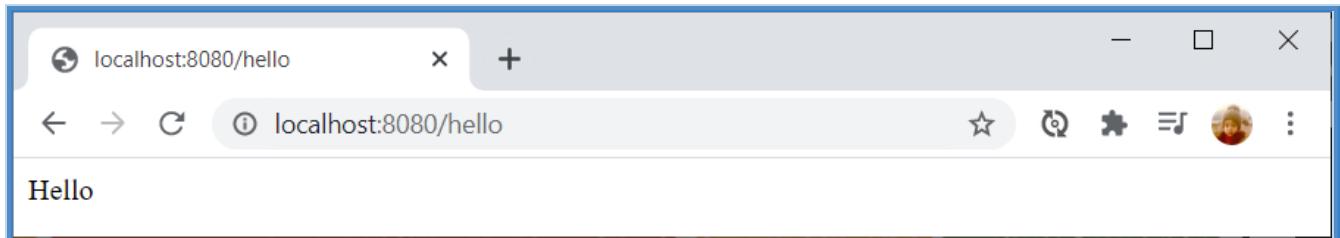
import org.springframework.stereotype.Service;

@Service
public class MyService {

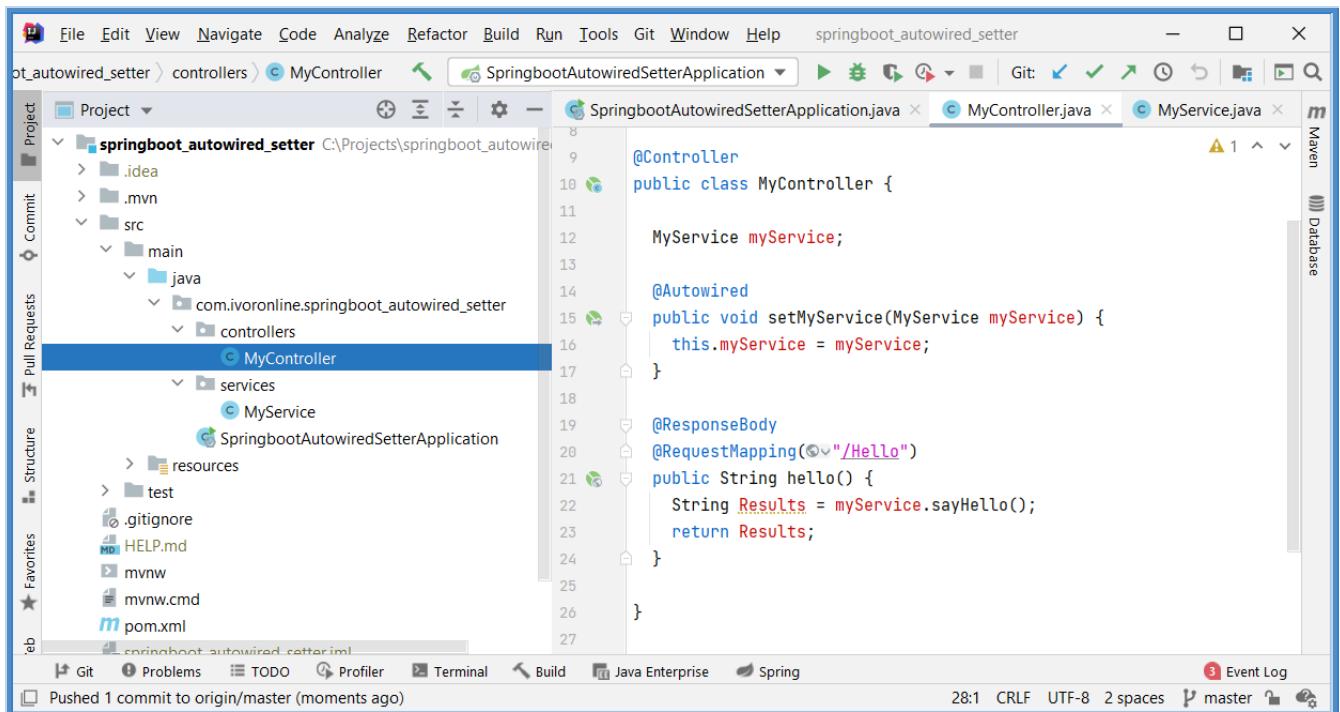
    public String sayHello() {
        return "Hello";
    }
}
```

## Results

<http://localhost:8080>Hello>



## Application Structure



## pom.xml

```
<dependencies>

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

</dependencies>
```

## 2.9.4 Location - Constructor

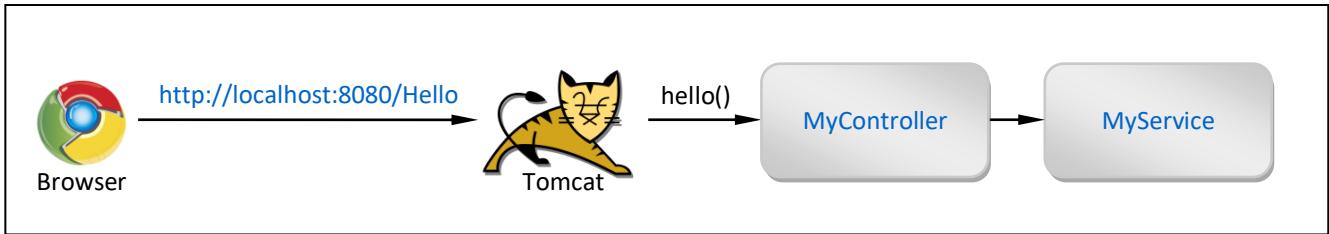
### Info

[G] [R] [R]

- This tutorial shows how to use `@Autowired` to inject instance into Constructor's Input Parameter.
- When `@Component` Class is instantiated Constructor is called (just like when instantiating any other Java Class).
- If Constructor accepts Spring Component, Spring Boot will automatically create an Instance of it.
- `@Autowired` Annotation before the Constructor is optional, in other words
  - only Methods with `@Autowired` Annotation will get its Parameters injected
  - but Constructor Parameters get injected by default (even if Constructor is not Annotated with `@Autowired`)

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables <code>@Controller</code> and <code>@RequestMapping</code> . Includes Tomcat Server.

## Procedure

---

- [Create Project](#): `springboot_autowired_constructor` (add Spring Boot Starters from the table)
- [Create Package](#): `controllers` (inside main package)
  - [Create Class](#): `MyController.java` (inside package controllers)
- [Create Package](#): `services` (inside main package)
  - [Create Class](#): `MyService.java` (inside package controllers)

### *MyController.java*

```
package com.ivoronline.springboot_autowired_constructor.controllers;

import com.ivoronline.springboot_autowired_constructor.services.MyService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.*;

@Controller
public class MyController {

    MyService myService;

    @Autowired
    public MyController(MyService myService) {
        this.myService = myService;
    }

    @ResponseBody
    @RequestMapping("/Hello")
    public String hello() {
        String Results = myService.sayHello();
        return Results;
    }
}
```

### *MyService.java*

```
package com.ivoronline.springboot_autowired_constructor.services;

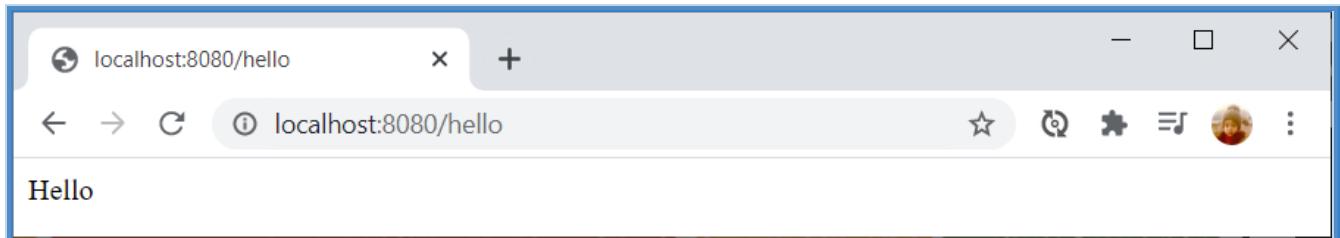
import org.springframework.stereotype.Service;

@Service
public class MyService {

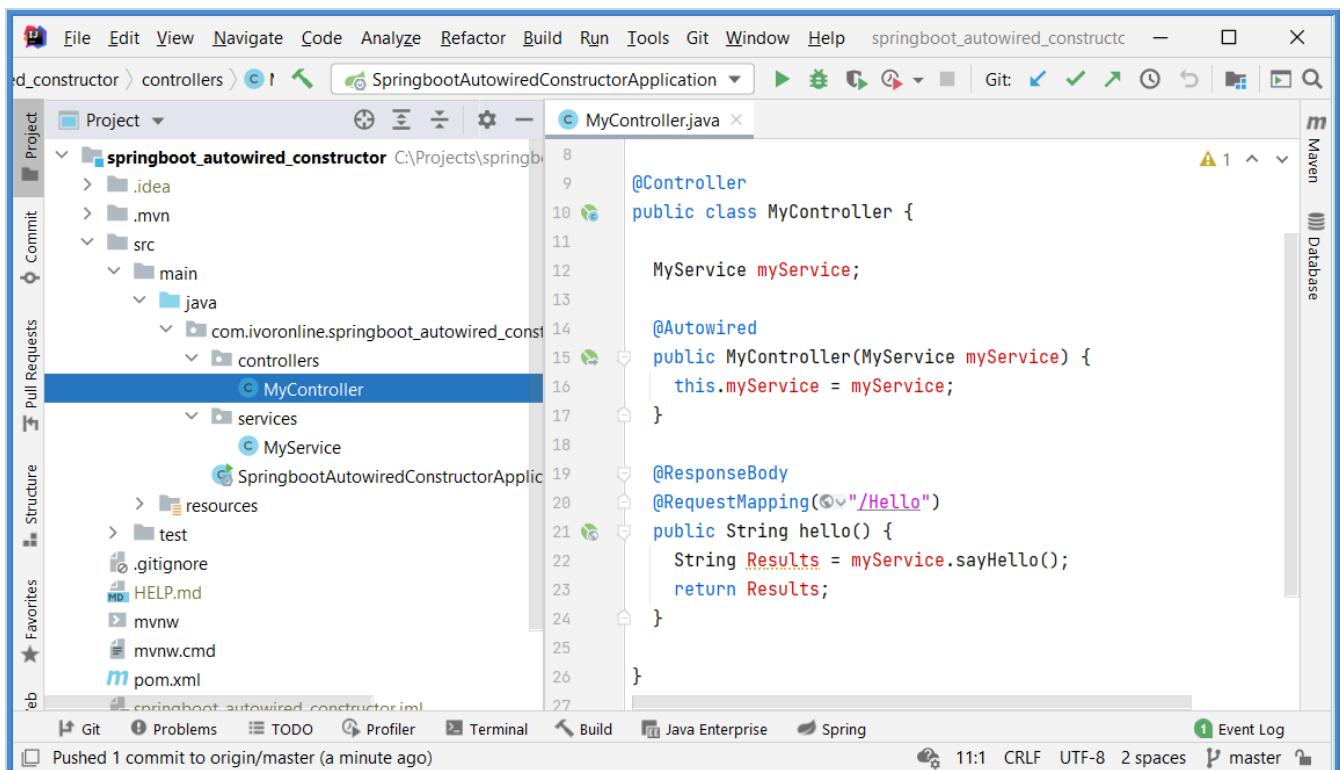
    public String sayHello() {
        return "Hello";
    }
}
```

## Results

<http://localhost:8080>Hello>



## Application Structure



## pom.xml

```
<dependencies>

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

</dependencies>
```

## 2.9.5 Interface - @Primary

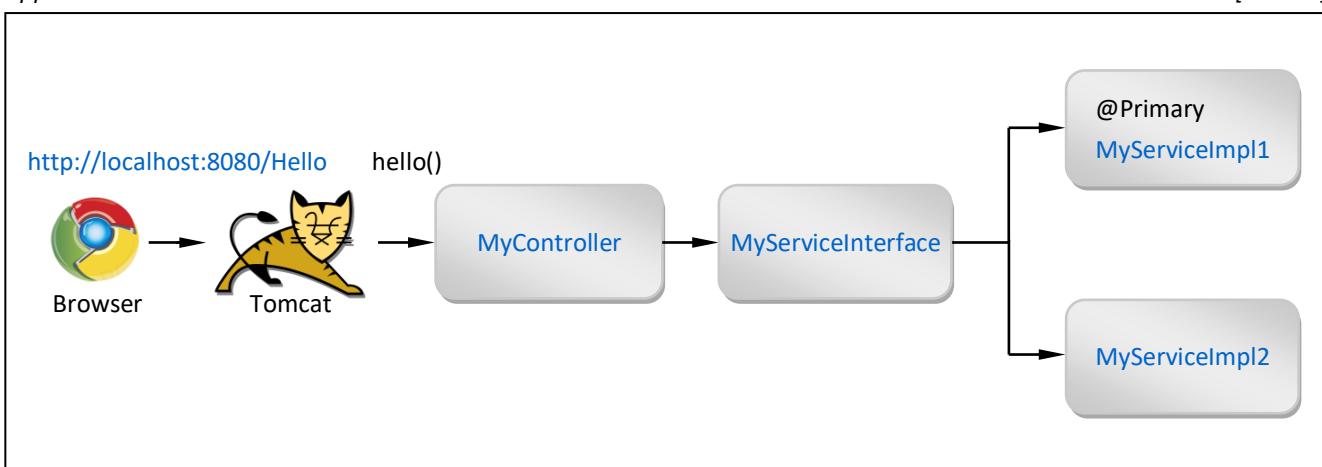
### Info

[G]

- In previous step we had a single Service Class that implements Service Interface.  
This way Spring knew which Service Class to auto wire.
- But if you have multiple Service Classes that implement the same Service Interface then you have to specify which to use.  
There are two ways to instruct Spring which Implementation to inject inside the Controller
  - @Primary** Annotation is attached to the Class that implements Interface
  - @Qualifier** Annotation is used in combination with `@Autowired`
- In this step we will create two Service Classes and use `@Primary` Annotation to tell Spring which one to inject
  - `MyServiceImplementation1` that returns "Hello"
  - `MyServiceImplementation2` that returns "Hello World"

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables: <code>@Controller</code> , <code>@RequestMapping</code> , Tomcat Server

### Procedure

- **Create Project:** `springboot_awtowired_primary` (add Spring Boot Starters from the table)
- **Create Package:** `controllers` (inside main package)
  - **Create Class:** `MyController.java` (inside package controllers)
- **Create Package:** `services` (inside main package)
  - **Create Interface:** `MyServiceInterface.java` (inside package controllers)
  - **Create Class:** `MyServiceImplementation1.java` (inside package controllers)
  - **Create Class:** `MyServiceImplementation2.java` (inside package controllers)

### *MyServiceInterface.java*

```
package com.ivoronline.springboot_awutowired_primary.services;

public interface MyServiceInterface {
    public String sayHello();
}
```

### *MyServiceImplementation1.java*

```
package com.ivoronline.springboot_awutowired_primary.services;

import org.springframework.context.annotation.Primary;
import org.springframework.stereotype.Service;

@Service
@Primary
public class MyServiceImplementation1 implements MyServiceInterface {
    public String sayHello() {
        return "Hello";
    }
}
```

### *MyServiceImplementation2.java*

```
package com.ivoronline.springboot_awutowired_primary.services;

import org.springframework.stereotype.Service;

@Service
public class MyServiceImplementation2 implements MyServiceInterface {
    public String sayHello() {
        return "Hello World";
    }
}
```

### *MyController.java*

```
package com.ivoronline.springboot_awutowired_primary.controllers;

import com.ivoronline.springboot_awutowired_primary.services.MyServiceInterface;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

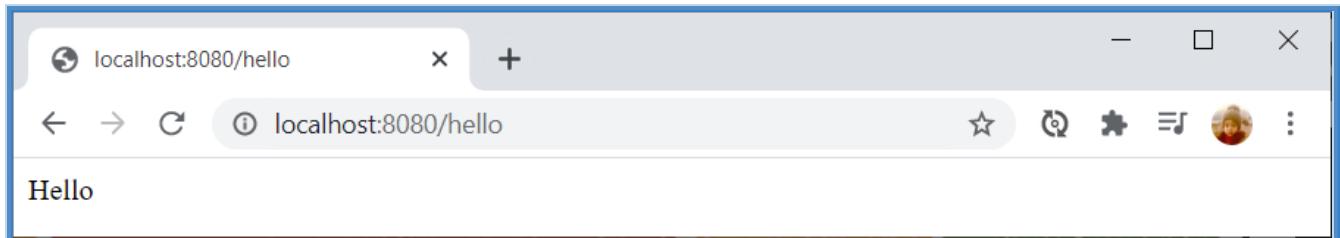
@Controller
public class MyController {

    @Autowired
    MyServiceInterface myService;

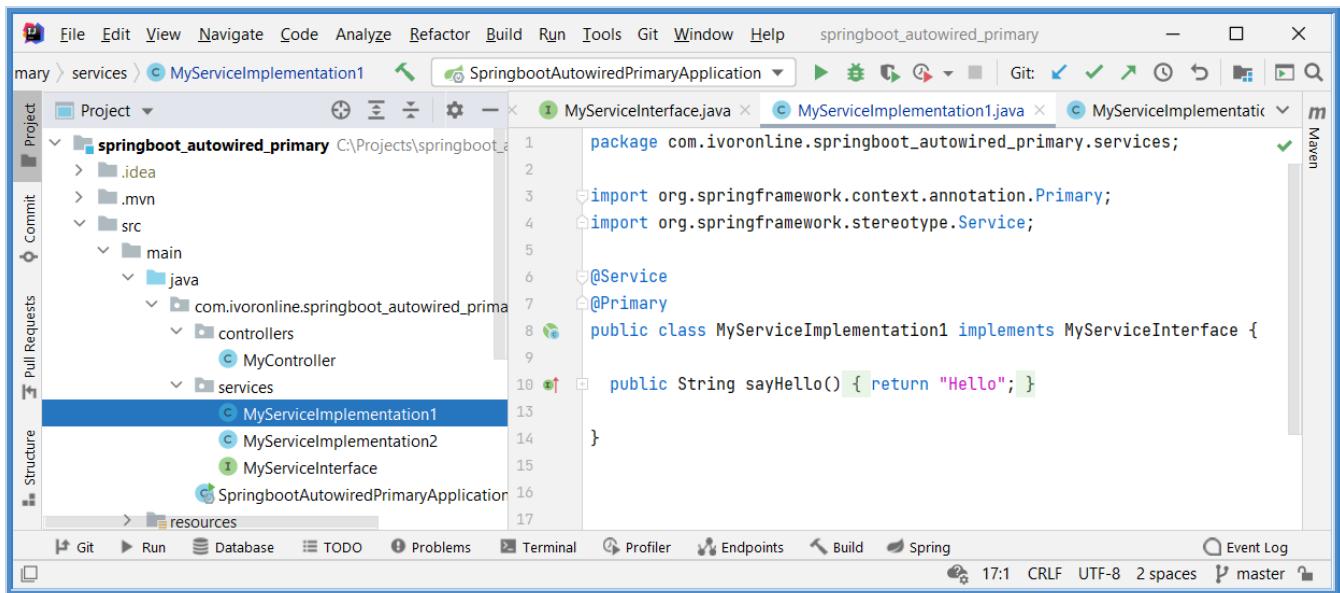
    @ResponseBody
    @RequestMapping("/Hello")
    public String hello() {
        String Results = myService.sayHello();
        return Results;
    }
}
```

## Results

<http://localhost:8080>Hello>



## Application Structure



## pom.xml

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

## 2.9.6 Interface - @Qualifier

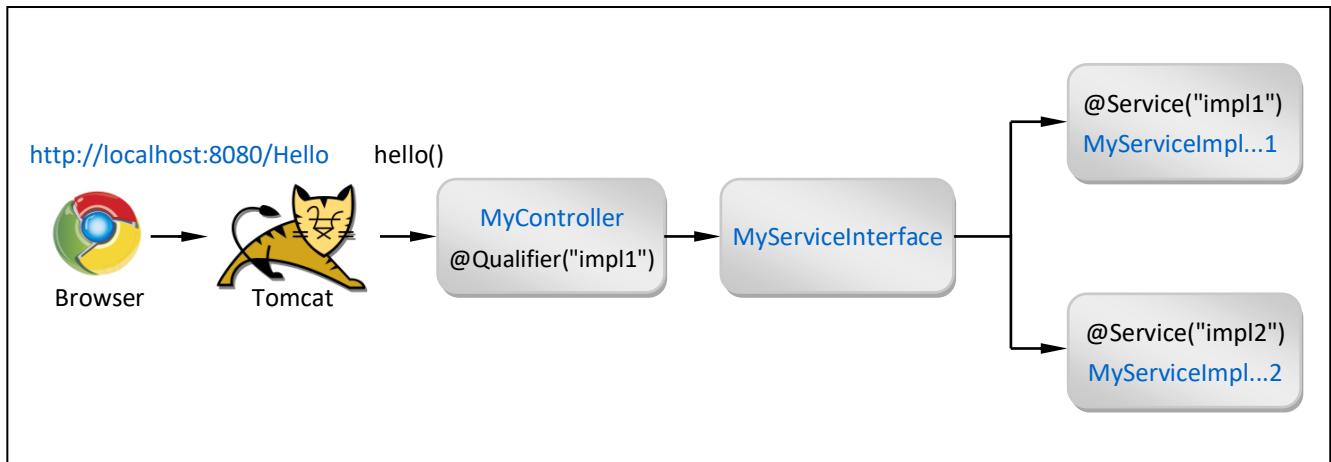
### Info

[G] [R]

- Another way to tell Spring which implementation to inject is to use **@Qualifier** Annotation.
- @Qualifier** is used by using
  - `@Service(name = "implementation1")` to give a name to the Spring Component
  - `@Autowired @Qualifier("implementation1")` to reference that Spring Component where it needs to be injected
- @Primary** and **@Qualifier** Annotations can be used at the same time.  
If `@Autowired` Annotation doesn't specify `@Qualifier`, `@Primary` Implementation will be used.

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables <code>@Controller</code> and <code>@RequestMapping</code> . Includes Tomcat Server.

### Procedure

- Create Project:** `springboot_autowired_qualifier` (add Spring Boot Starters from the table)
- Create Package:** `controllers` (inside main package)
  - Create Class:** `MyController.java` (inside package controllers)
- Create Package:** `services` (inside main package)
  - Create Interface:** `MyServiceInterface.java` (inside package controllers)
  - Create Class:** `MyServiceImpl1.java` (inside package controllers)
  - Create Class:** `MyServiceImpl2.java` (inside package controllers)

### *MyServiceInterface.java*

```
package com.ivoronline.springboot_awarded_qualifier.services;

public interface MyServiceInterface {
    public String sayHello();
}
```

### *MyServiceImplementation1.java*

```
package com.ivoronline.springboot_awarded_qualifier.services;

import org.springframework.stereotype.Service;

@Service("impl1")
public class MyServiceImplementation1 implements MyServiceInterface {
    public String sayHello() {
        return "Hello";
    }
}
```

### *MyServiceImplementation2.java*

```
package com.ivoronline.springboot_awarded_qualifier.services;

import org.springframework.stereotype.Service;

@Service("impl2")
public class MyServiceImplementation2 implements MyServiceInterface {
    public String sayHello() {
        return "Hello World";
    }
}
```

### *MyController.java*

```
package com.ivoronline.springboot_awarded_qualifier.controllers;

import com.ivoronline.springboot_awarded_qualifier.services.MyServiceInterface;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

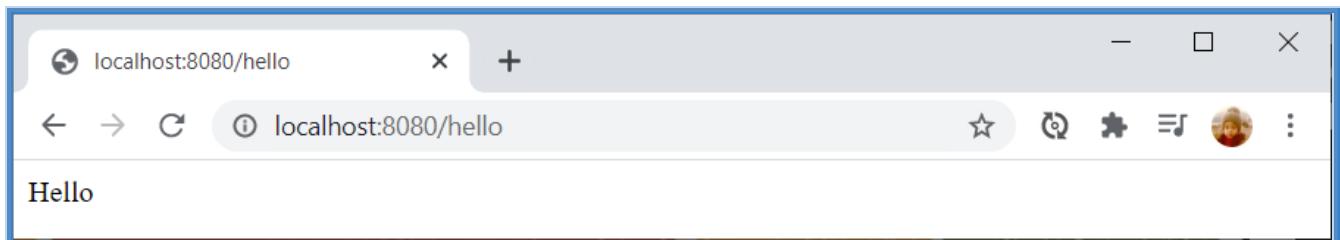
@Controller
public class MyController {

    @Autowired
    @Qualifier("impl1")
    MyServiceInterface myService;

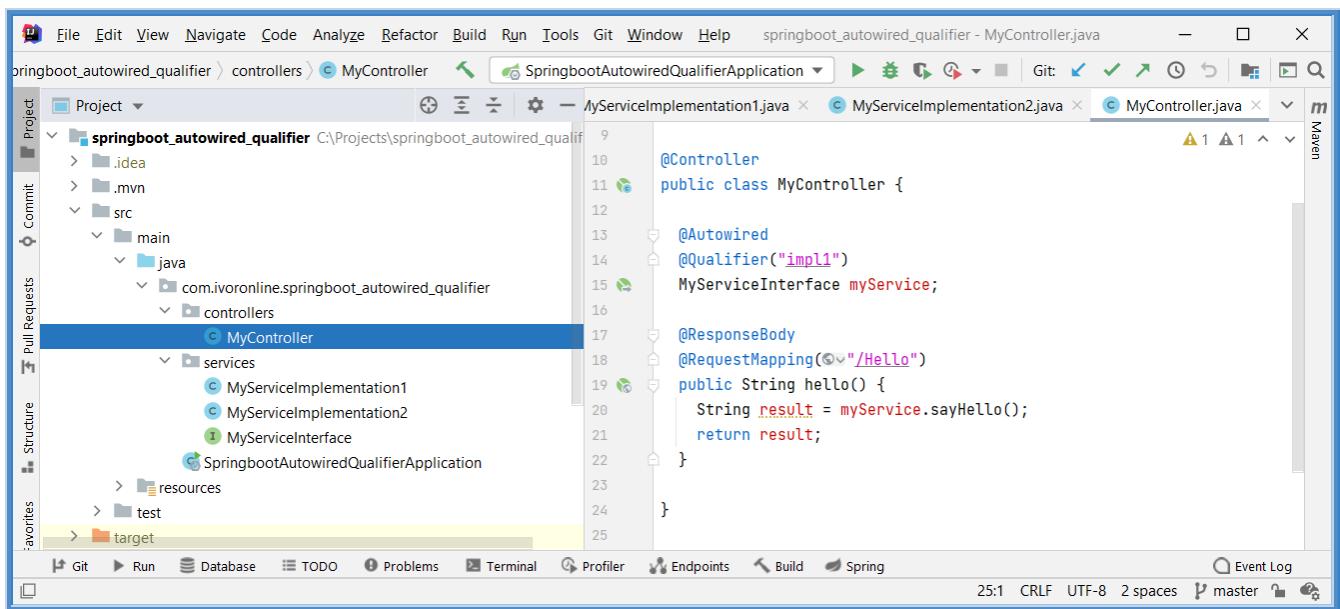
    @ResponseBody
    @RequestMapping("/Hello")
    public String hello() {
        String Results = myService.sayHello();
        return Results;
    }
}
```

## Results

<http://localhost:8080>Hello>



## Application Structure



## pom.xml

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

## 2.9.7 Interface - @Profile

### Info

[G]

- Another way to tell Spring which Implementation/Class to inject is to use **@Profile** Annotation.
- Profiles define which Class will be used in your application.
- You can assign one or more Profiles to a Class.
- Then in the configuration file you define **Active Profile**.
- When you start application only Classes that belong to the **Active Profile** (or have no @Profile) will be used.

### Syntax

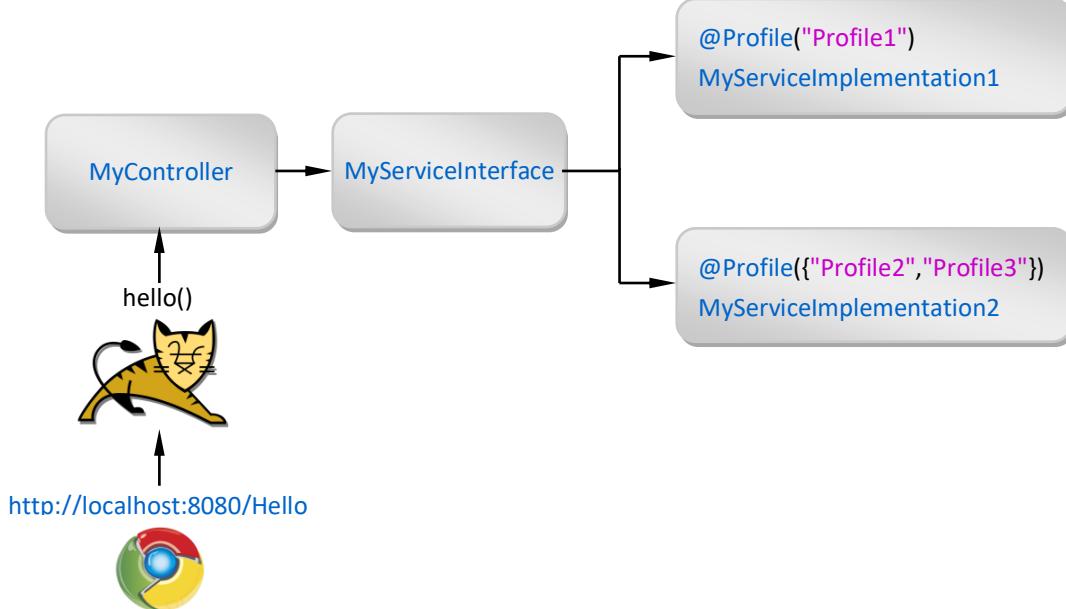
```
@Profile( "Profile1" )
@Profile({"Profile2","Profile3"})
public class MyServiceImplementation1 implements MyServiceInterface { ... }
```

### application.properties

```
spring.profiles.active = Profile1
```

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables @Controller and @RequestMapping. Includes Tomcat Server.

## Procedure

- Create Project: springboot\_awutowired\_profile (add Spring Boot Starters from the table)
- Edit File: application.properties (spring.profiles.active = Profile1)
- Create Package: controllers (inside main package)
  - Create Class: MyController.java (inside package controllers)
- Create Package: services (inside main package)
  - Create Interface: MyServiceInterface.java (inside package controllers)
  - Create Class: MyServiceImplementation1.java (inside package controllers)
  - Create Class: MyServiceImplementation2.java (inside package controllers)

application.properties

```
spring.profiles.active = Profile1
```

MyServiceInterface.java

```
package com.ivoronline.springboot_awutowired_profile.services;

public interface MyServiceInterface {
    public String sayHello();
}
```

MyServiceImplementation1.java

```
package com.ivoronline.springboot_awutowired_profile.services;

import org.springframework.context.annotation.Profile;
import org.springframework.stereotype.Service;

@Service
@Profile("Profile1")
public class MyServiceImplementation1 implements MyServiceInterface {

    public String sayHello() {
        return "Hello";
    }
}
```

MyServiceImplementation2.java

```
package com.ivoronline.springboot_awutowired_profile.services;

import org.springframework.context.annotation.Profile;
import org.springframework.stereotype.Service;

@Service
@Profile({"Profile2","Profile3"})
public class MyServiceImplementation2 implements MyServiceInterface {

    public String sayHello() {
        return "Hello World";
    }
}
```

### *MyController.java*

```
package com.ivoronline.springboot_awutowired_profile.controllers;

import com.ivoronline.springboot_awutowired_profile.services.MyServiceInterface;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class MyController {

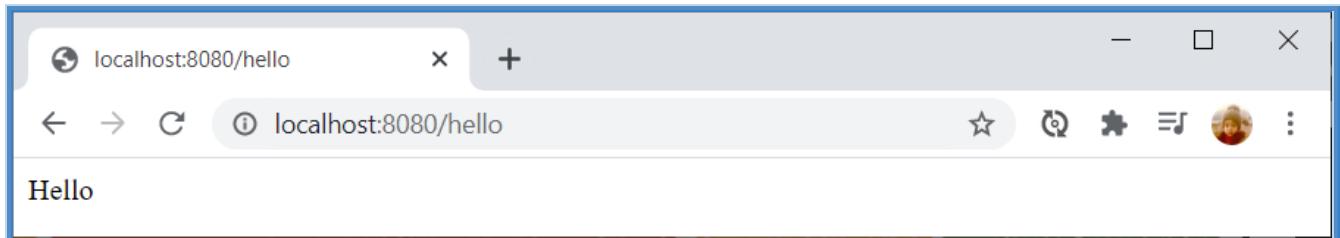
    @Autowired
    MyServiceInterface myService;

    @ResponseBody
    @RequestMapping("/Hello")
    public String hello() {
        String Results = myService.sayHello();
        return Results;
    }

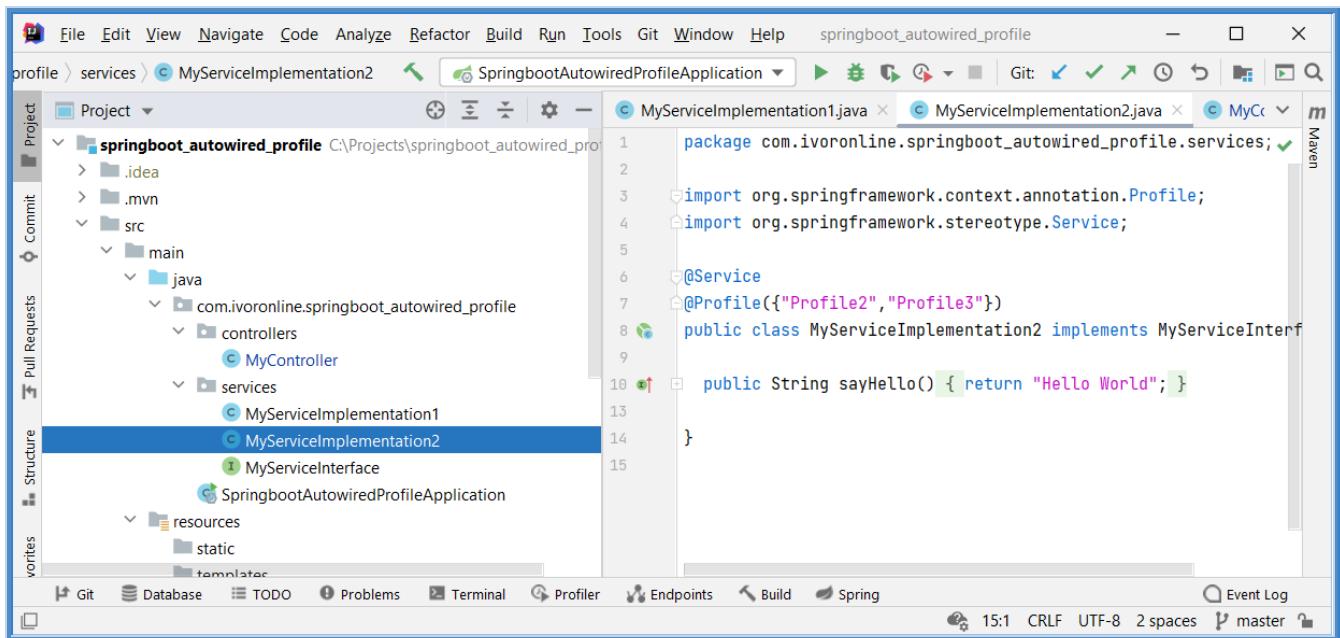
}
```

## Results

<http://localhost:8080>Hello>



## Application Structure



## pom.xml

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

## 2.10 DTO - Serialize

### Info

---

- Following tutorials show how to Serialize DTO - convert it into JSON for the purpose of returning it back to Frontend.

## 2.10.1 @ResponseBody

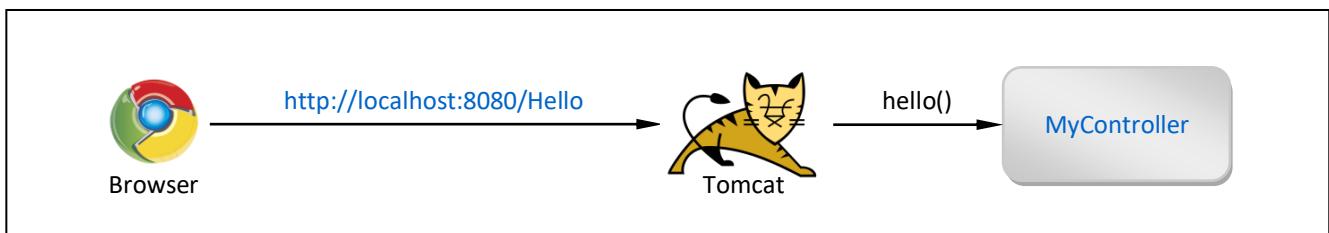
### Info

[G]

- `@ResponseBody` defines that Endpoint returns Data.
  - This means that we are not feeding return value into some Template.
  - Instead return value should be returned directly through HTTP Response.
  - If Endpoint returns an Object it gets Deserialized.
- (this tutorial is identical to "Return - Text")  
(like HTML, JSP, Thymeleaf)  
(Text or Object Serialized as JSON)  
(It is returned as JSON)

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables: @RequestMapping, Tomcat Server

### Procedure

- **Create Project:** controller\_returns\_text (add Spring Boot Starters from the table)
- **Create Package:** controllers (inside main package)
  - **Create Class:** `MyController.java` (inside controllers package)

### MyController.java

```
package com.ivoronline.controller_returns_text.controllers;

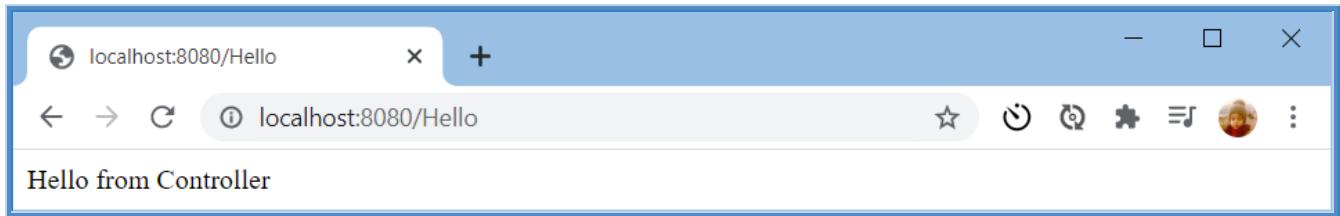
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class MyController {

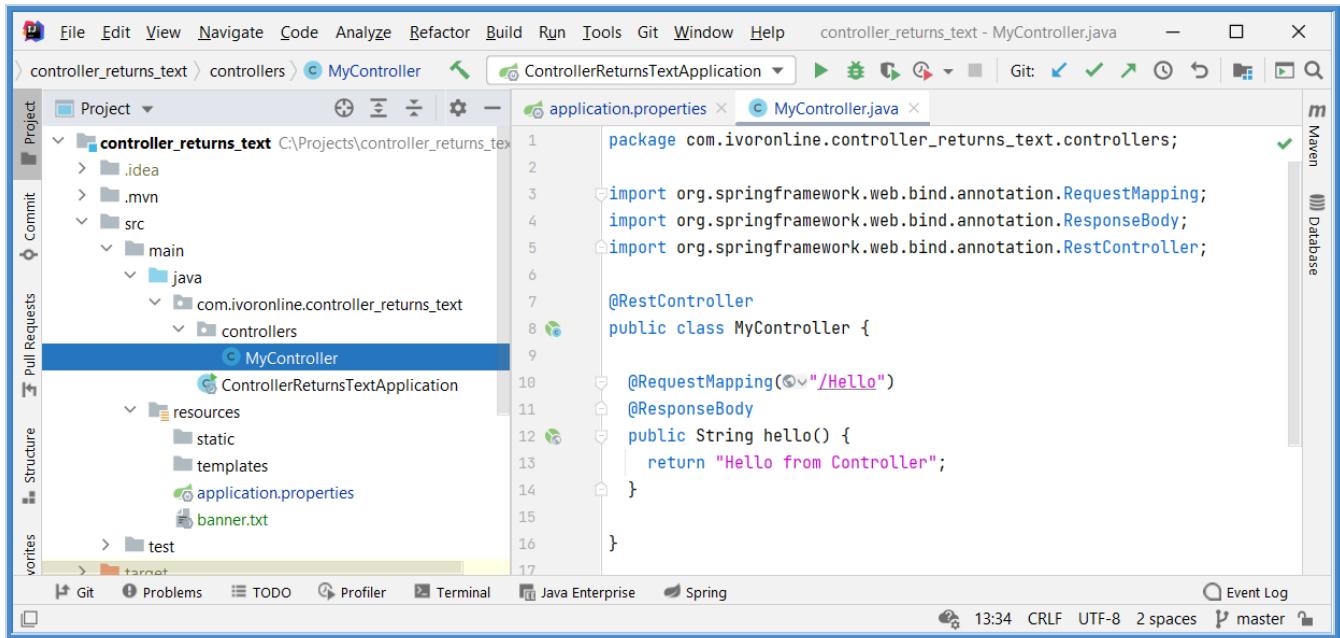
    @ResponseBody
    @RequestMapping("/Hello")
    public String hello() {
        return "Hello from Controller";
    }
}
```

## Results

<http://localhost:8080>Hello>



## Application Structure



## pom.xml

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

## 2.10.2 @JsonSerialize

### Info

[G] [R]

- This tutorial shows how to create custom Serializer to return DTO as **JSON** from Controller.
- Serializer is
  - implemented as `class PersonDTOSerializer extends JsonSerializer<PersonDTO>`
  - attached to PersonDTO with `@JsonSerialize(using = PersonDTOSerializer.class)`
  - called when Controller returns PersonDTO

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables: @Controller, @RequestMapping, Tomcat Server

## Procedure

---

- [Create Project](#): `springboot_json_serializer` (add Spring Boot Starters from the table)
- [Create Package](#): `DTO` (inside main package)

  - [Create Class](#): `PersonDTOSerializer.java` (inside package `DTO`)
  - [Create Class](#): `PersonDTO.java` (inside package `DTO`)

- [Create Package](#): `controllers` (inside main package)

  - [Create Class](#): `MyController.java` (inside package `controllers`)

### *PersonDTOSerializer.java*

```
package com.ivoronline.springboot_json_serializer.DTO;

import com.fasterxml.jackson.core.JsonGenerator;
import com.fasterxml.jackson.databind.JsonSerializer;
import com.fasterxml.jackson.databind.SerializerProvider;
import java.io.IOException;

public class PersonDTOSerializer extends JsonSerializer<PersonDTO> {

    @Override
    public void serialize(PersonDTO personDTO, JsonGenerator jsonGenerator, SerializerProvider provider)
throws IOException {
        jsonGenerator.writeStartObject();
        jsonGenerator.writeStringField("First Name", personDTO.name);
        jsonGenerator.writeNumberField("Age" , personDTO.age);
        jsonGenerator.writeEndObject();
    }

}
```

### *PersonDTO.java*

```
package com.ivoronline.springboot_json_serializer.DTO;

import com.fasterxml.jackson.databind.annotation.JsonSerialize;

@JsonSerialize(using = PersonDTOSerializer.class)
public class PersonDTO {
    public String name;
    public Integer age;
}
```

### MyController.java

```
package com.ivoronline.springboot_json_serializer.controllers;

import com.ivoronline.springboot_json_serializer.DTO.PersonDTO;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class MyController {

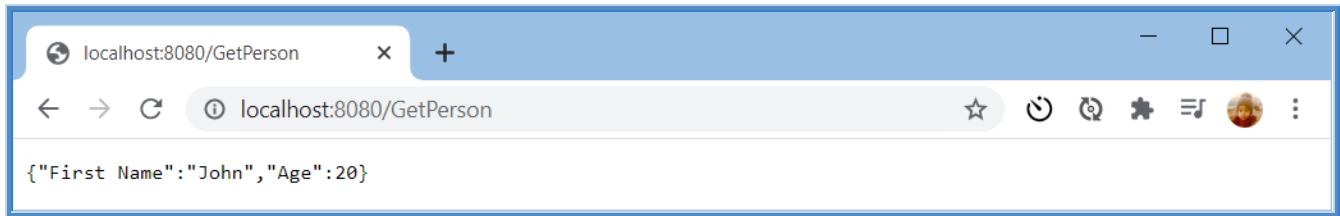
    @ResponseBody
    @RequestMapping("/GetPerson")
    public PersonDTO addPerson() {

        //CREATE PERSON
        PersonDTO PersonDTO = new PersonDTO();
        PersonDTO.name = "John";
        PersonDTO.age = 20;

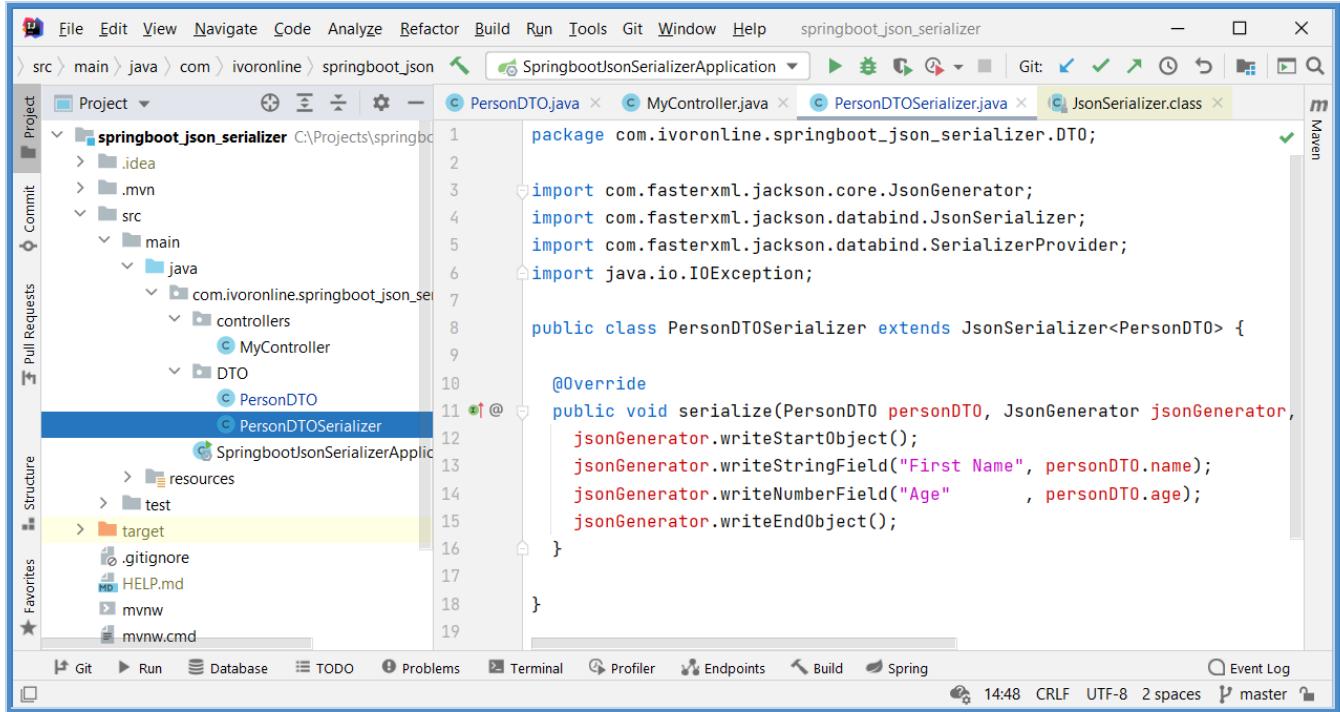
        //RETURN PERSON AS JSON (SERIALIZED)
        return PersonDTO;
    }
}
```

## Results

<http://localhost:8080/GetPerson>



## Application Structure



## pom.xml

```
<dependencies>

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

</dependencies>
```

## 2.11 DTO - Deserialize

### Info

- Following tutorials show how to work with Data Transfer Objects (DTO) which are used to **transfer data** between
  - **Applications** (Frontend and Backend)
  - **Layers** of the same Application (Controller and Service Layer)
- In certain situations instead of DTO you can **reuse Entity** to transfer data.  
But if data from multiple Entities needs to be transferred then DTO is used.
- For each pair of HTTP Request and Response you can have two DTOs
  - **RequestDTO** to Deserialize data received from the Frontend
  - **ResponseDTO** to Serialize data returned to the Frontend

### Content

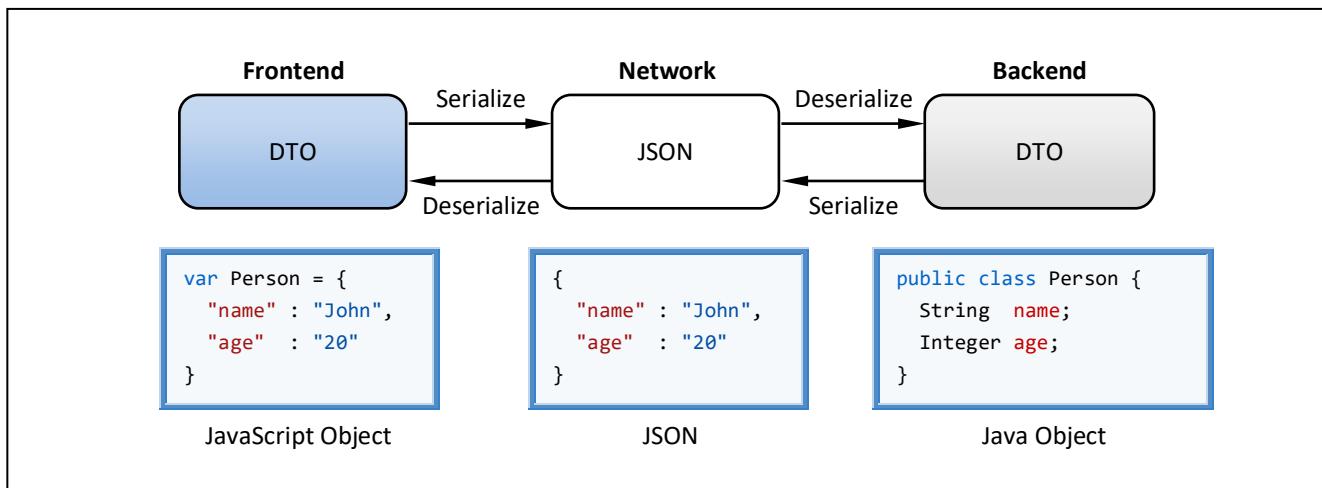
[DTO usage between Frontend and Backend](#)  
[DTO usage between Application Layers](#)  
[Jackson Library](#)  
[Model Mapper](#)

### Related Previous Tutorials

<a href="#">Data Transfer Object (DTO)</a>	Explains theory behind using DTO between Controller and Service Layer
<a href="#">Annotation - (Spring) @RequestBody</a>	Explains how to convert HTTP Request Parameters into DTO

- General process looks like this
  - Frontend stores data into DTO (JavaScript Object)
  - Frontend Serializes DTO into JSON
  - Frontend sends JSON (Postman with JSON Body)
  - Controller receives JSON
  - Controller Deserializes JSON into DTO (@RequestBody PersonDTO personDTO)
  - Controller works with DTO (Java Object)
- **Serialization** is process of turning an **Object** into a **Structured Data Format** like: **JSON, XML**.  
Serialization can be done to **store** or **send** Object.
- **Deserialization** is reverse process of turning **Structured Data Format** back into an **Object**.

### Schema



### MyController.java

```
@Controller  
public class MyController {  
  
    @ResponseBody  
    @RequestMapping("/AddPerson")  
    public PersonDTO addPerson(@RequestBody PersonDTO personDTO) {  
        //Controller works with Deserialized personDTO Object  
        return personDTO; //Controller returns Serialized PersonDTO as JSON  
    }  
  
}
```

### Jackson Library

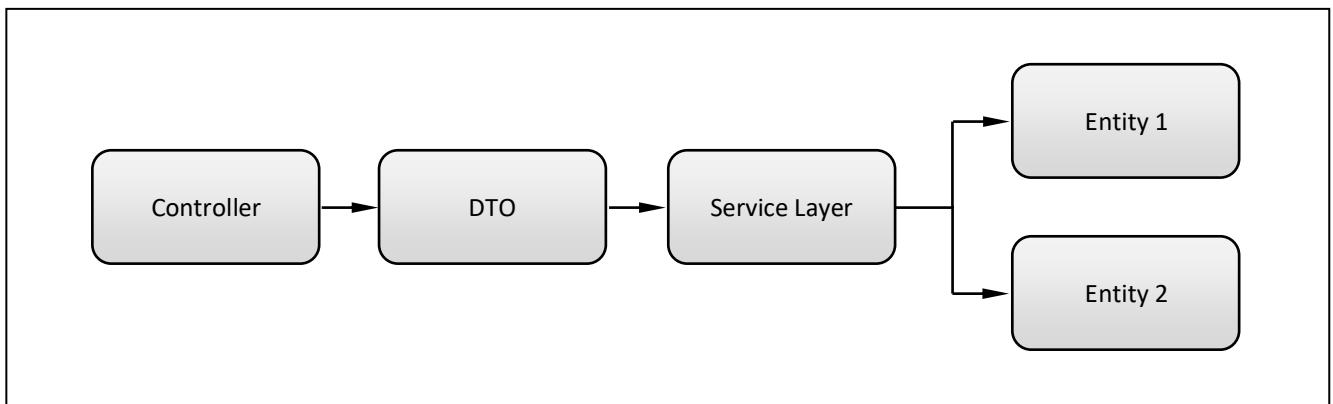
- Spring Boot uses **Jackson Library** to Serialize/Deserialize JSON/Object.
- Jackson Library uses **reflection** to access **private** Properties/Setters/Getters.
- To Deserialize JSON into Object Jackson will use
  - Constructor (if present)
  - Setters (if Constructor is not present)
  - Properties (if neither Setters nor Constructor are present)
- Following Annotations can be used above Properties/Setters/Getters or for Constructor Parameters
  - **@JsonProperty** specifies which JSON Property to map with Annotated Property/Setter/Getter/Constructor Parameter
  - **@JsonFormat** specifies in which format is Value of JSON Property  
(so that it could be properly converted into Value that can be stored into Object Property)

## DTO usage between Application Layers

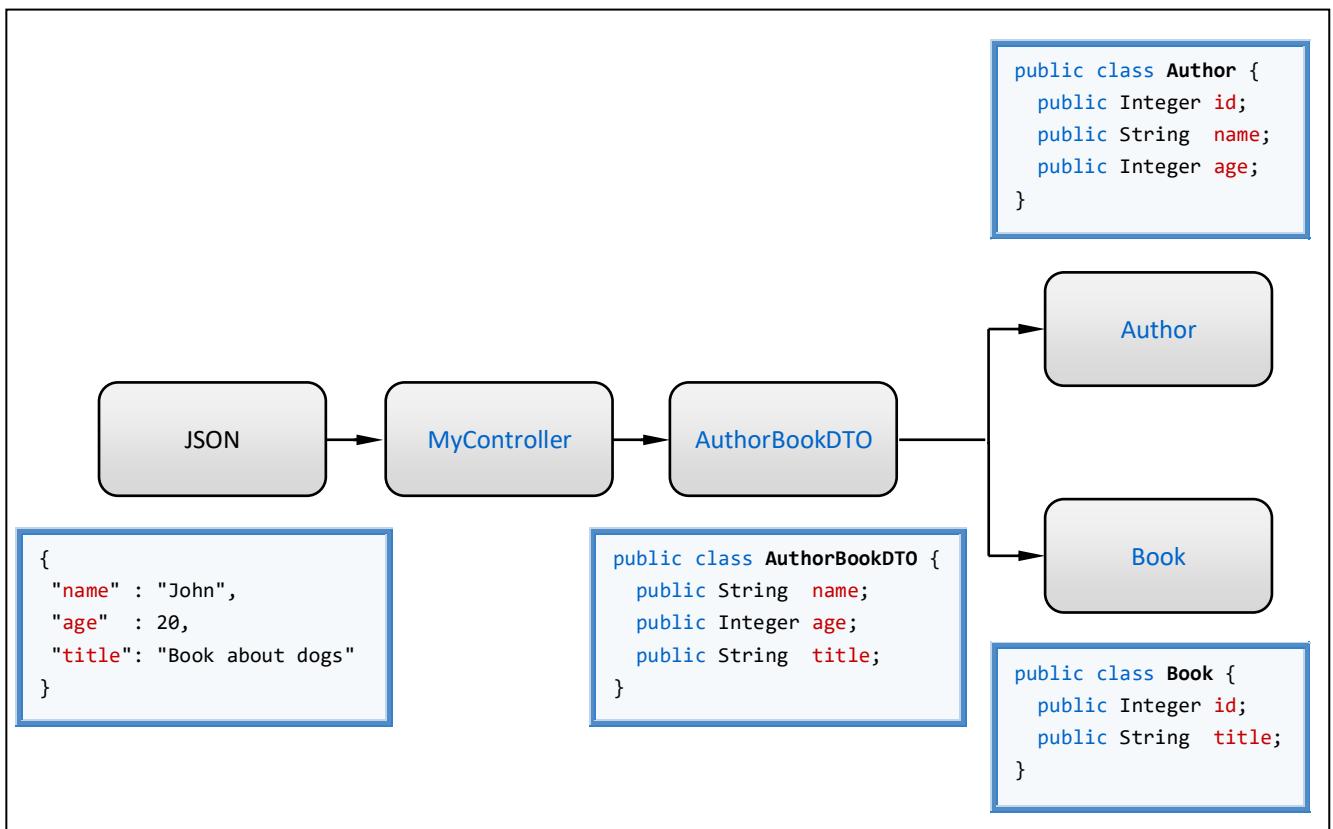
- General process looks like this
  - Controller sends DTO to Service Layer (after it Deserializing DTO from JSON received by Frontend)
  - Service Layer converts DTO into Entities (using Object Mappers)
  - Service Layer uses Entities to perform Business Logic
  - Service Layer stores resulting Data into DTO
  - Service Layer return DTO to Controller (which Serializes DTO into JSON and returns it to Frontend)
- Object Mappers are used to convert one Object into another - in our case to convert DTO into Entities.  
There are different Libraries that implement Object Mapping in different ways.

*Controller sends DTO to Service Layer*

*Service Layer converts DTO into Entities*



*AuthorBookDTO is mapped into Author and Book Entities*



- Often there is a need to convert DTO into Entity and vice versa which is done using Model Mapper Class that
  - maps DTO into Entity and vice versa
  - by default matches Properties with the same name
  - transfers matching Properties between DTO and Entity by using their **setters and getters**  
(both DTO and Entity must have setters and getters since Properties aren't referenced directly even if they are public)
- There is **no Spring Boot Starter** for Model Mapper so Maven Dependency has to be manually added to [pom.xml](#).

*pom.xml*

[R]

```
<dependency>
    <groupId>org.modelmapper</groupId>
    <artifactId>modelmapper</artifactId>
    <version>2.3.9</version>
</dependency>
```

## 2.11.1 From Request Parameters - Using Setters

### Info

[G]

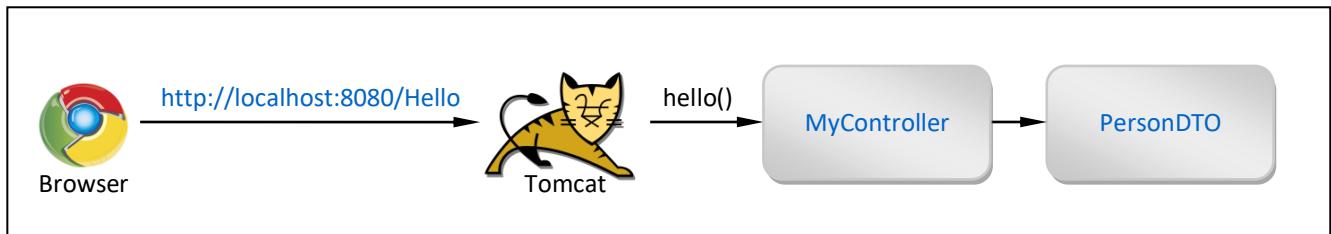
- This tutorial shows how to **Deserialize HTTP Request Parameters** into **DTO**. (how to automatically load them into DTO)
- Unfortunately there is no way to map HTTP Request Parameters to DTO Parameters if they have different names.  
To do so you can use workaround described in [From Request Parameters - Using Map](#) where
  - HTTP Request Parameters are first loaded into `Map<String, Object>` and then
  - `ObjectMapper` loads the Map into `PersonDTO`

### Syntax

```
@RequestMapping("/Hello")
public String hello(PersonDTO personDTO) { ... }
```

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables: Controller Annotations, Tomcat Server

## Procedure

---

- [Create Project](#): `springboot_deserialize_requestparameters` (add Spring Boot Starters from the table)
- [Create Package](#): `DTO` (inside main package)
- [Create Class](#): `PersonDTO.java` (inside package controllers)
- [Create Package](#): `controllers` (inside main package)
- [Create Class](#): `MyController.java` (inside package controllers)

*PersonDTO.java*

```
package com.ivoronline.springboot_deserialize_requestparameters.DTO;

public class PersonDTO {

    //PROPERTIES
    public String name;
    public Integer age;

    //SETTERS (used for deserialization)
    public void setName(String name) { this.name = name; }
    public void setAge (Integer age ) { this.age = age; }

}
```

*MyController.java*

```
package com.ivoronline.springboot_deserialize_requestparameters.controllers;

import com.ivoronline.springboot_deserialize_requestparameters.DTO.PersonDTO;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

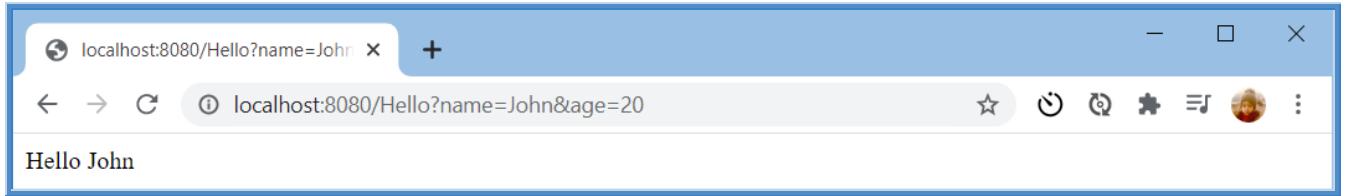
@Controller
public class MyController {

    @ResponseBody
    @RequestMapping("/Hello")
    public String hello(PersonDTO personDTO) {
        return "Hello " + personDTO.name;
    }

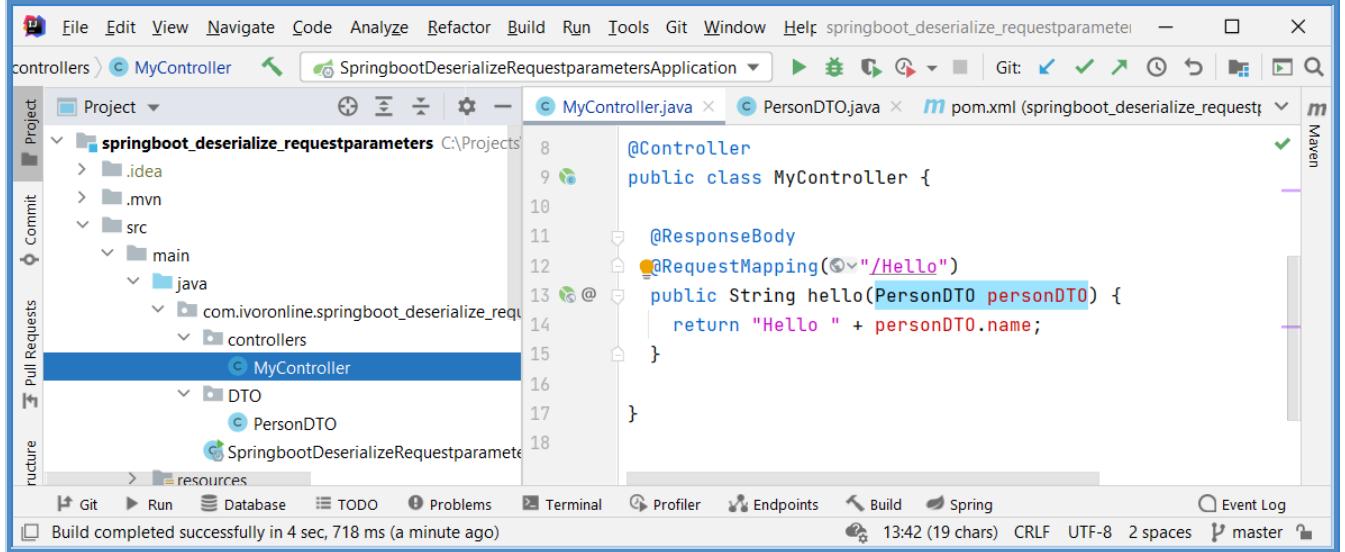
}
```

## Results

<http://localhost:8080>Hello?name=John&age=20>



## Application Structure



## pom.xml

```
<dependencies>

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

</dependencies>
```

## 2.11.2 From Request Parameters - Using Setters - Customized

### Info

[G]

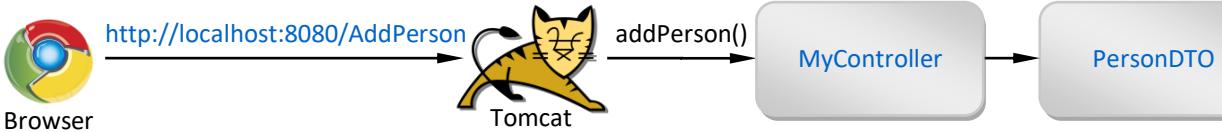
- This tutorial shows how to implement **Customized** Deserialization of HTTP Request Parameters into DTO. Since this type of Deserialization is done through **Setters**, you just need to add your custom logic to them. Also make note that there is no way to map HTTP Request Parameters to DTO Parameters if they have different names.
- In this tutorials Request Parameter `height` uses comma separated value `1,67` which can't automatically loaded into DTO. Therefore first we load it into `String height`, replace comma with dot and then convert it into to Float `heightFloat` Property

### Syntax

```
@RequestMapping("/Hello")
public String hello(PersonDTO personDTO) { ... }
```

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables: Controller Annotations, Tomcat Server

## Procedure

- **Create Project:** `springboot_deserialize_requestparameters_custom` (add Spring Boot Starters from the table)
- **Create Package:** `DTO` (inside main package)
- **Create Class:** `PersonDTO.java` (inside package controllers)
- **Create Package:** `controllers` (inside main package)
- **Create Class:** `MyController.java` (inside package controllers)

### *PersonDTO.java*

```
package com.ivoronline.springboot_deserialize_requestparameters_custom.DTO;

public class PersonDTO {

    //PROPERTIES
    public String name;
    public String height;        //"1,67"
    public Float heightFloat;   // 1.67

    //SETTERS (used for deserialization)
    public void setName (String name) { this.name = name; }

    public void setHeight (String height) {
        height = height.replace(',', '.');
        this.heightFloat = Float.parseFloat(height);
    }

}
```

### *MyController.java*

```
package com.ivoronline.springboot_deserialize_requestparameters_custom.controllers;

import com.ivoronline.springboot_deserialize_requestparameters_custom.DTO.PersonDTO;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class MyController {

    @ResponseBody
    @RequestMapping("/AddPerson")
    public String addPerson(PersonDTO personDTO) {
        return personDTO.name + " is " + personDTO.heightFloat + " meters high";
    }

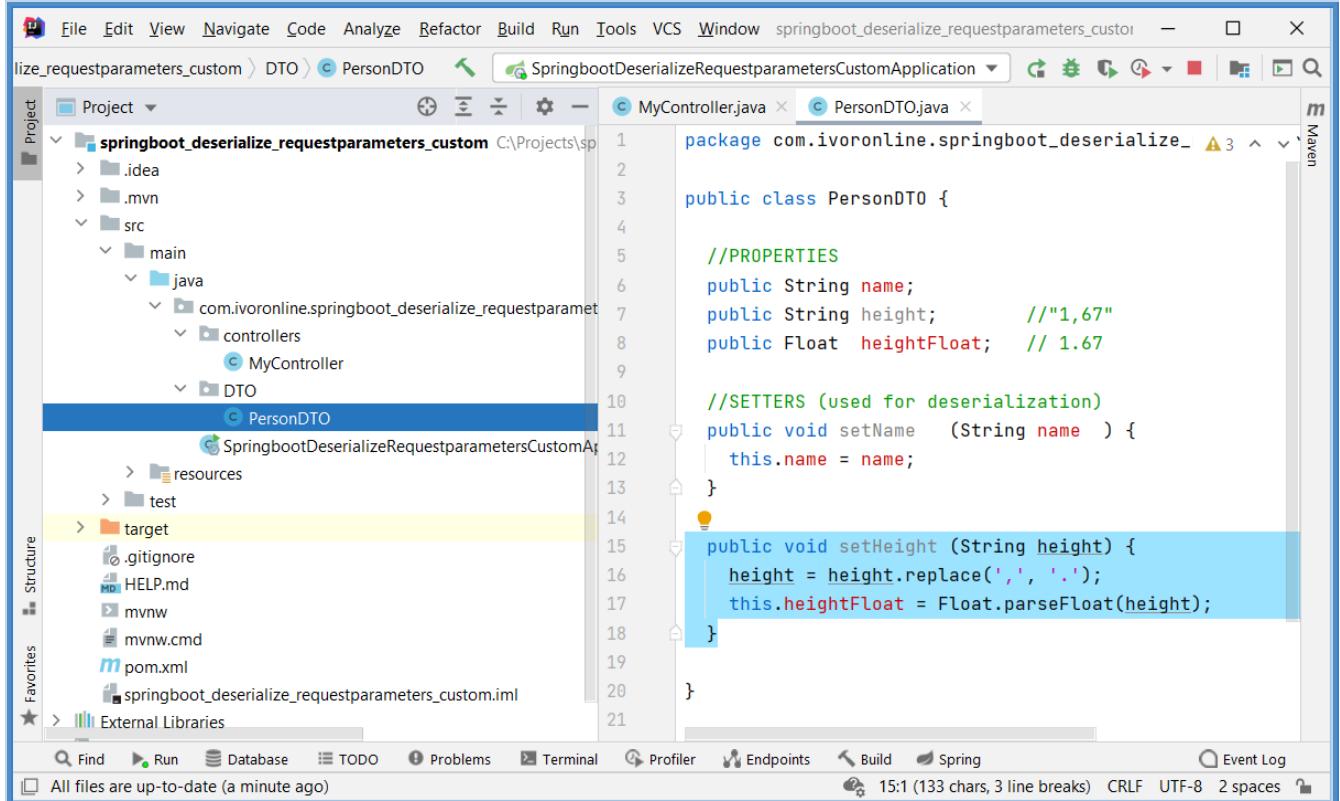
}
```

## Results

<http://localhost:8080/AddPerson?name=John&height=1,67>



## Application Structure



## pom.xml

```
<dependencies>

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

</dependencies>
```

## 2.11.3 From Request Parameters - Using Map

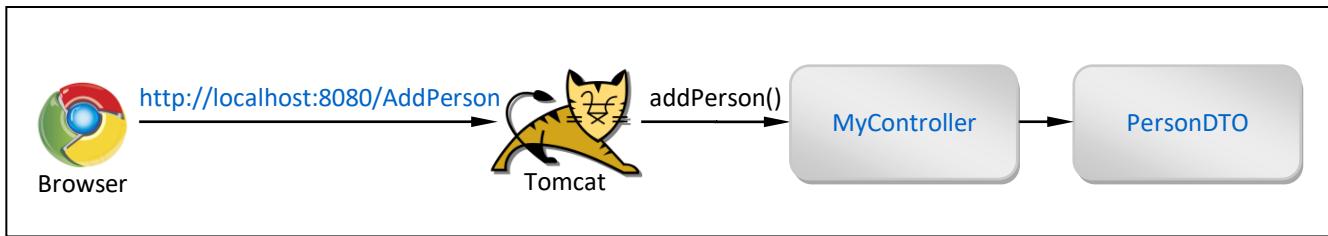
### Info

[G]

- This tutorial shows how to
  - Deserialize HTTP Request Parameters into `Map<String, Object>` (how to automatically load them into Map)
  - an then use `ObjectMapper` to load that Map into `PersonDTO`
- This allows us to use `@JsonProperty()` Annotation to map HTTP Request Parameters to DTO Properties. This is not possible when directly loading HTTP Request Parameters into DTO (as shown in [From Request Parameters - Using Setters](#)) but the downside is the additional line in the Controller to call `new ObjectMapper().convertValue()`.

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables: Controller Annotations, Tomcat Server

## Procedure

- [Create Project](#): `springboot_deserialize_requestparameters_objectmapper` (add Spring Boot Starters from the table)
- [Create Package](#): `DTO` (inside main package)
- [Create Class](#): `PersonDTO.java` (inside package `DTO`)
- [Create Package](#): `controllers` (inside main package)
- [Create Class](#): `MyController.java` (inside package `controllers`)

### *PersonDTO.java*

```
package com.ivoronline.springboot_deserialize_requestparameters_objectmapper.DTO;

import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
import com.fasterxml.jackson.annotation.JsonProperty;

@JsonIgnoreProperties(ignoreUnknown = true) //To avoid Error if Request contains additional Parameters
public class PersonDTO {

    @JsonProperty("firstName") //If Request Parameter and DTO Property have different names
    public String name;

    public String height;

}
```

### *MyController.java*

```
package com.ivoronline.springboot_deserialize_requestparameters_objectmapper.controllers;

import com.fasterxml.jackson.databind.ObjectMapper;
import com.ivoronline.springboot_deserialize_requestparameters_objectmapper.DTO.PersonDTO;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseBody;
import java.util.Map;

@Controller
public class MyController {

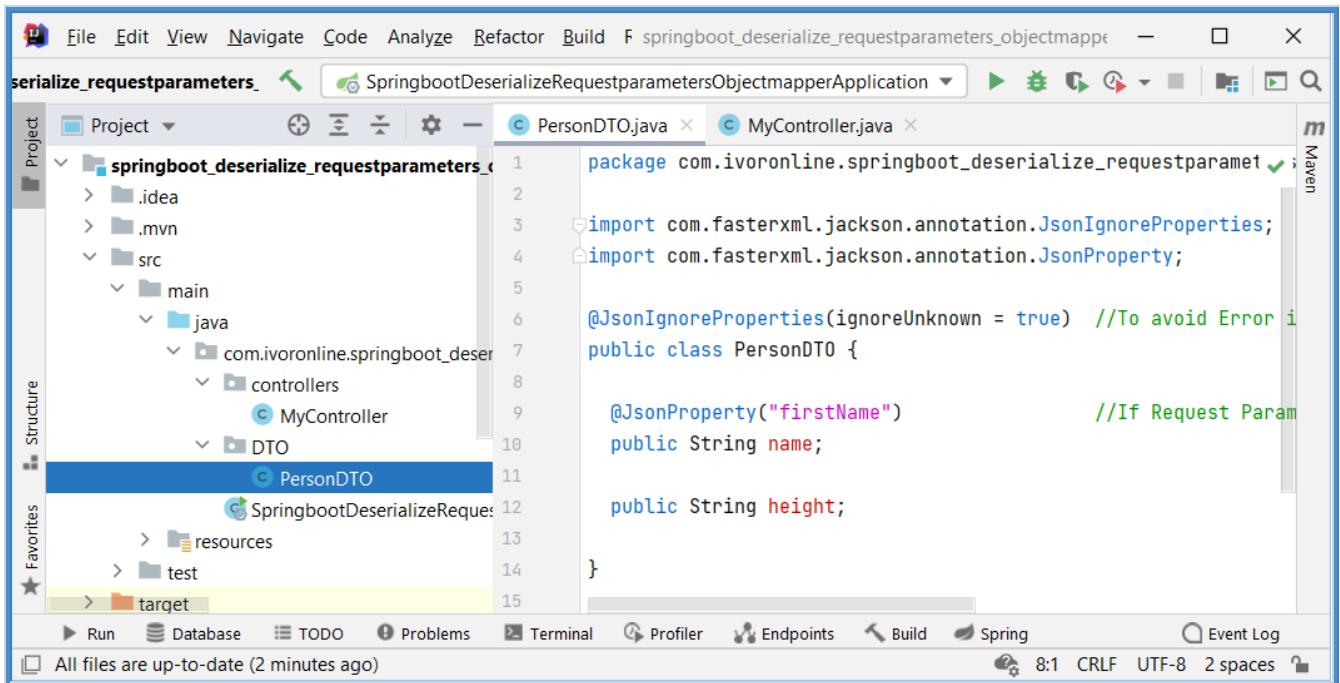
    @ResponseBody
    @RequestMapping("/AddPerson")
    public String addPerson(@RequestParam Map<String, Object> requestParameters) {
        PersonDTO personDTO = new ObjectMapper().convertValue(requestParameters, PersonDTO.class);
        return personDTO.name + " is " + personDTO.height + " meters high";
    }
}
```

## Results

<http://localhost:8080/AddPerson?firstName=John&height=1.67&age=10>



## Application Structure



## pom.xml

```
<dependencies>

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

</dependencies>
```

## 2.11.4 From Request Parameters - Using Map - Customize Setter

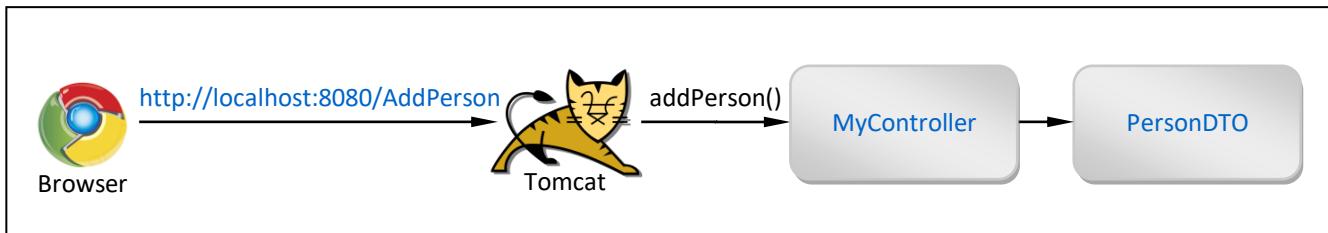
Info

[G]

- This tutorial shows how to
  - Deserialize **HTTP Request Parameters** into `Map<String, Object>` (how to automatically load them into Map)
  - use `ObjectMapper` to load that Map into `PersonDTO`
  - use `setters()` to customize values that go into `PersonDTO` Properties (converting "1,67" into "1.67")

Application Schema

[Results]



Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables: Controller Annotations, Tomcat Server

## Procedure

- [Create Project](#): `springboot_deserialize_requestparameters_objectmapper` (add Spring Boot Starters from the table)
- [Create Package](#): `DTO` (inside main package)
- Create Class: `PersonDTO.java` (inside package `DTO`)
- [Create Package](#): `controllers` (inside main package)
- Create Class: `MyController.java` (inside package `controllers`)

### *PersonDTO.java*

```
package com.ivoronline.springboot_deserialize_requestparameters_objectmapper.DTO;

import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
import com.fasterxml.jackson.annotation.JsonProperty;

@JsonIgnoreProperties(ignoreUnknown = true) //To avoid Error if Request contains additional Parameters
public class PersonDTO {

    //PROPERTIES
    public String name;
    public Float height;

    //CONVERSION SETTER
    @JsonProperty("height") //Map Request Parameter to Setter
    public void setHeight(String heightString) {
        heightString = heightString.replace(',', '.');
        this.height = Float.parseFloat(heightString);
    }

}
```

### *MyController.java*

```
package com.ivoronline.springboot_deserialize_requestparameters_objectmapper.controllers;

import com.fasterxml.jackson.databind.ObjectMapper;
import com.ivoronline.springboot_deserialize_requestparameters_objectmapper.DTO.PersonDTO;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseBody;
import java.util.Map;

@Controller
public class MyController {

    @ResponseBody
    @RequestMapping("/AddPerson")
    public String addPerson(@RequestParam Map<String, Object> requestParameters) {
        PersonDTO personDTO = new ObjectMapper().convertValue(requestParameters, PersonDTO.class);
        return personDTO.name + " is " + personDTO.height + " meters high";
    }

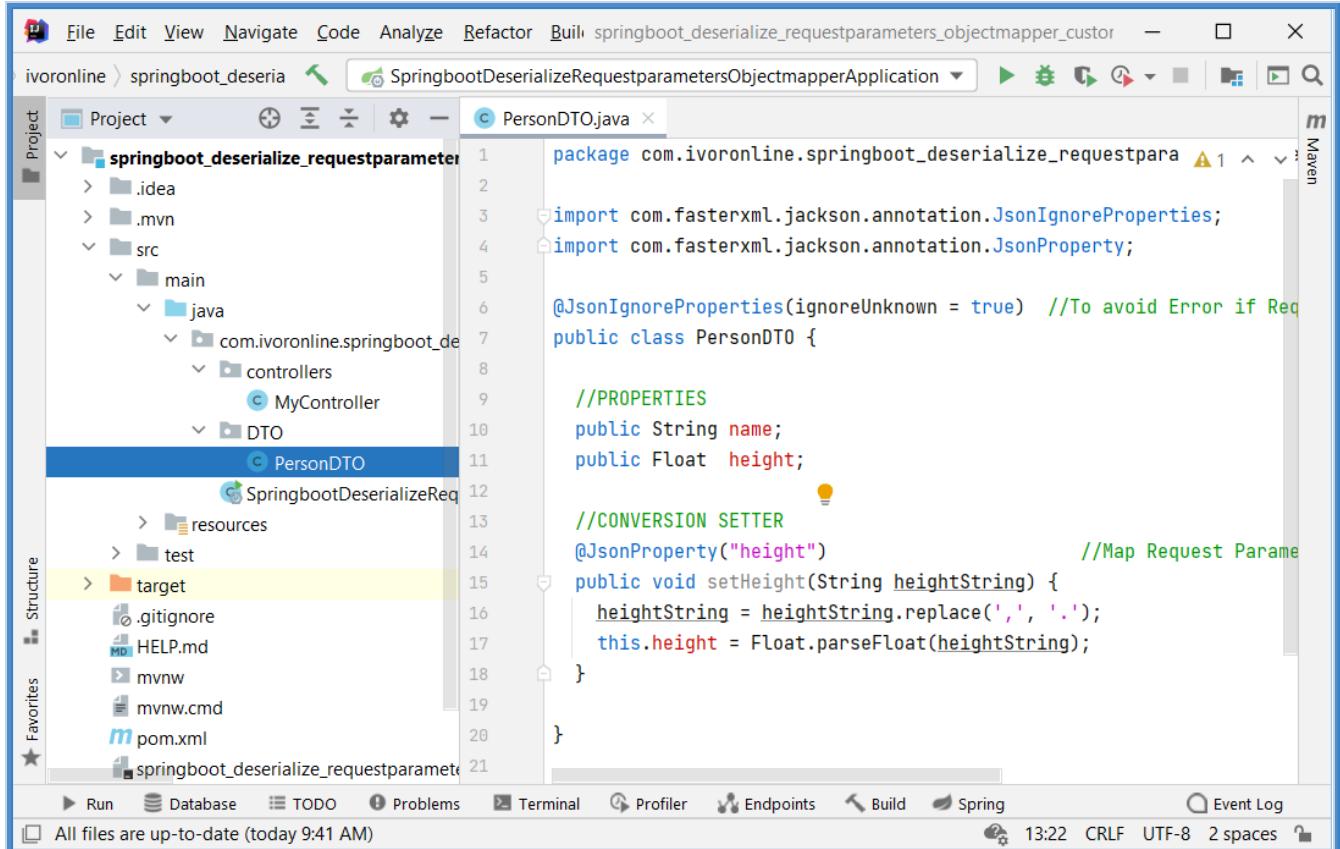
}
```

## Results

<http://localhost:8080/AddPerson?name=John&height=1,67>



## Application Structure



## pom.xml

```
<dependencies>

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

</dependencies>
```

## 2.11.5 From JSON - @RequestBody - Properties

### Info

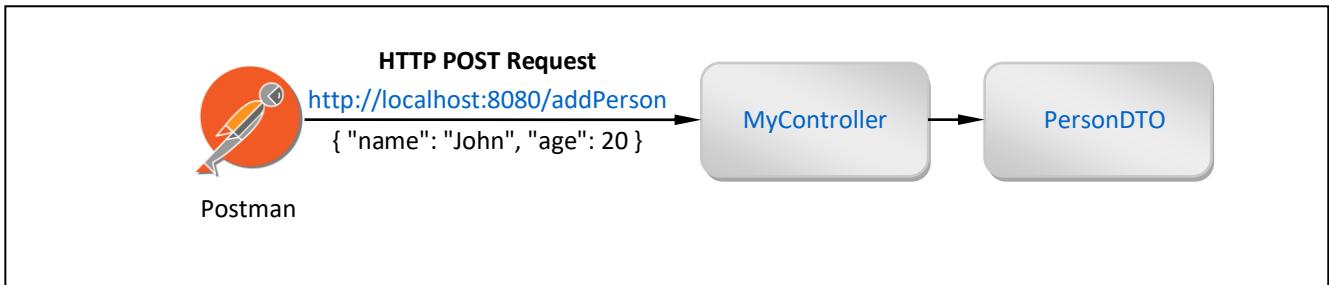
- This tutorial shows how to use `@RequestBody` to convert JSON from HTTP Request Body into Java Object (DTO).

### Syntax

```
@RequestMapping("/AddPerson")
public String addPerson(@RequestBody PersonDTO personDTO) { ... }
```

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables @Controller, @RequestMapping, Tomcat Server

## Procedure

---

- Create Project: bootspring\_http\_requestbody (add Spring Boot Starters from the table)
- Create Package: DTO (inside main package)
  - Create Class: PersonDTO.java (inside package DTO)
- Create Package: controllers (inside main package)
  - Create Class: MyController.java (inside package controllers)

### PersonDTO.java

```
package com.ivoronline.bootspring_http_requestbody.DTO;

public class PersonDTO {

    //PROPERTIES
    //Used for Deserialization if there is no Constructor or Setters
    //Jackson uses reflection to access private Properties.
    public Long id;
    public String name;
    public Integer age;

}
```

### MyController.java

```
package com.ivoronline.bootspring_http_requestbody.controllers;

import com.ivoronline.bootspring_http_requestbody.DTO.PersonDTO;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class MyController {

    @ResponseBody
    @RequestMapping("/AddPerson")
    public String addPerson(@RequestBody PersonDTO personDTO) {

        //GET DATA FROM PersonDTO
        String name = personDTO.name;
        Integer age = personDTO.age;

        //RETURN SOMETHING
        return name + " is " + age + " years old";

    }
}
```

## Results

- Start Postman

POST

```
http://localhost:8080/addPerson
```

Headers

(add Key-Value)

```
Content-Type: application/json
```

Body

(option: raw)

```
{
  "name" : "John",
  "age"   : 20
}
```

Postman

The screenshot shows the Postman application window. At the top, there's a navigation bar with File, Edit, View, Help, Home, Workspaces, Reports, Explore, and a search bar. Below the navigation is a toolbar with various icons. The main area shows a list of recent requests and a new request section labeled 'DTO / New Request'. A specific request is selected, showing a 'GET' method and the URL 'http://localhost:8080/AddPerson'. The 'Body' tab is active, displaying a JSON payload:

```
1 {
2   "name" : "John",
3   "age"   : 20
4 }
```

Below the body, the response status is shown as 200 OK with a response time of 99 ms and a size of 184 B. The response body contains the text 'John is 20 years old'. There are tabs for Body, Cookies, Headers (5), and Test Results. At the bottom, there are buttons for Pretty, Raw, Preview, Visualize, and Text.

HTTP Response Body

```
John is 20 years old
```

## Application Structure

The screenshot shows the IntelliJ IDEA interface. The left sidebar displays the project structure under 'Project'. The 'src' folder contains 'main' and 'test' packages. 'main/java' has 'com.ivoronline.bootstrapping\_http\_requestbody' and 'controllers' packages. 'controllers' contains 'MyController.java'. The code editor shows the following Java code:

```
8  @Controller
9  public class MyController {
10
11     @ResponseBody
12     @RequestMapping("/addAuthor")
13     public String addAuthor(@RequestBody PersonDTO personDTO) {
14
15         //GET DATA FROM PersonDTO
16         String name = personDTO.name;
17         Integer age = personDTO.age;
18
19         //RETURN SOMETHING
20         return name + " is " + age + " years old";
21     }
22
23 }
24
25 }
```

The 'pom.xml' tab is visible at the top, and the status bar at the bottom indicates 'All files are up-to-date (a minute ago)'.

## pom.xml

```
<dependencies>

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

</dependencies>
```

## 2.11.6 From JSON - @RequestBody - Setters

### Info

[G]

- This tutorial shows how to use `@RequestBody` Annotation to convert JSON from HTTP Request into Java Object (DTO).

### Syntax

```
@RequestMapping("/AddPerson")
public String addPerson(@RequestBody PersonDTO personDTO) { ... }
```

### Application Schema

[Results]



#### HTTP POST Request

<http://localhost:8080/addPerson>

{ "name": "John", "age": 20 }

Postman

MyController

PersonDTO

### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables: <code>@Controller</code> , <code>@RequestMapping</code> , Tomcat Server

## Procedure

---

- Create Project: `springboot_dto_json_object_setters` (add Spring Boot Starters from the table)
- Create Package: `DTO` (inside main package)
  - Create Class: `PersonDTO.java` (inside package controllers)
- Create Package: `controllers` (inside main package)
  - Create Class: `MyController.java` (inside package controllers)

### *PersonDTO.java*

```
package com.ivoronline.springboot_dto_json_object_setters.DTO;

public class PersonDTO {

    //PROPERTIES
    //Not used for Deserialization if there is Constructor or Setters
    public String name;
    public Integer age;

    //SETTERS
    //Used for Deserialization if there is no constructor
    //Jackson uses reflection to access private setters
    private void setName(String name) { this.name = name; }
    private void setAge (Integer age ) { this.age = age; }

}
```

### *MyController.java*

```
package com.ivoronline.springboot_dto_json_object_setters.controllers;

import com.ivoronline.springboot_dto_json_object_setters.DTO.PersonDTO;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class MyController {

    @ResponseBody
    @RequestMapping("/AddPerson")
    public String addPerson(@RequestBody PersonDTO personDTO) {

        //GET DATA FROM PersonDTO
        String name = personDTO.name;
        Integer age = personDTO.age;

        //RETURN SOMETHING
        return name + " is " + age + " years old";

    }
}
```

## Results

- Start Postman

POST

```
http://localhost:8080/addPerson
```

Headers

(add Key-Value)

```
Content-Type: application/json
```

Body

(option: raw)

```
{
  "name" : "John",
  "age"   : 20
}
```

Postman

The screenshot shows the Postman application window. In the top navigation bar, 'File', 'Edit', 'View', and 'Help' are visible. The main interface shows a 'DTO / New Request' section with a 'GET' method and URL 'http://localhost:8080/AddPerson'. The 'Body' tab is selected, showing a JSON payload:

```
1 {
2   "name" : "John",
3   "age"   : 20
4 }
```

The 'Tests' tab contains the response: '1 John is 20 years old'. The status bar at the bottom indicates 'Status: 200 OK Time: 99 ms Size: 184 B'. Other tabs like 'Cookies', 'Headers', and 'Test Results' are also visible.

HTTP Response Body

```
John is 20 years old
```

## Application Structure

The screenshot shows the IntelliJ IDEA interface. The left sidebar displays the project structure under 'springboot\_dto\_json\_object\_setters'. The 'src' folder contains 'main' and 'test' packages. 'main/java/com.ivoronline.springboot\_dto.controllers' contains 'MyController.java'. 'main/java/com.ivoronline.springboot\_dto.DTO' contains 'PersonDTO.java'. The code editor on the right shows the content of 'PersonDTO.java'. The code defines a class 'PersonDTO' with properties 'name' and 'age', and two private setters for them. It also includes two public getters for 'name' and 'age'. The code editor has syntax highlighting and line numbers.

```
package com.ivoronline.springboot_dto.DTO;

public class PersonDTO {

    //PROPERTIES
    private String name;
    private Integer age;

    //SETTERS (USED FOR DESERIALIZATION SINCE THERE IS NO CONSTRUCTOR)
    //JACKSON USES REFLECTION TO ACCESS PRIVATE SETTERS
    private void setName(String name) { this.name = name; }
    private void setAge (Integer age ) { this.age = age; }

    //GETTERS
    public String getName() { return name; }
    public Integer getAge () { return age; }

}
```

## pom.xml

```
<dependencies>

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

</dependencies>
```

## 2.11.7 From JSON - @RequestBody - Constructor

### Info

[G]

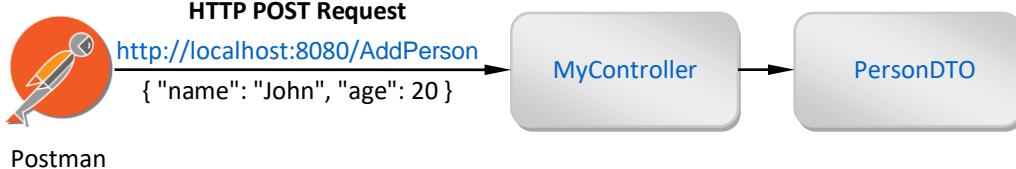
- This tutorial shows how to
  - use `@RequestBody` to convert **JSON** from HTTP Request Body into **DTO**
  - by calling DTO **Constructor**

### Syntax

```
@RequestMapping("/AddPerson")
public String addPerson(@RequestBody PersonDTO personDTO) { ... }
```

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables @Controller and @RequestMapping. Includes Tomcat Server.

## Procedure

- Create Project: `springboot_dto_json_object_constructor` (add Spring Boot Starters from the table)
- Create Package: `controllers` (inside main package)
- Create Class: `MyController.java` (inside package controllers)
- Create Package: `DTO` (inside main package)
- Create Class: `PersonDTO.java` (inside package controllers)

### `PersonDTO.java`

```
package com.ivoronline.springboot_dto_json_object_constructor.DTO;

public class PersonDTO {

    //PROPERTIES
    //Not used for Deserialization if there is Constructor or Setters
    private String name;
    private Integer age;

    //SETTERS
    //Not used for Deserialization if there is Constructor
    public String getName() { return name; }
    public Integer getAge () { return age; }

    //CONSTRUCTOR
    //Used for Deserialization
    PersonDTO(String name, Integer age) {
        this.name = name;
        this.age = age;
    }

}
```

### `MyController.java`

```
package com.ivoronline.springboot_dto_json_object_constructor.controllers;

import com.ivoronline.springboot_dto_json_object_constructor.DTO.PersonDTO;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class MyController {

    @ResponseBody
    @RequestMapping("/AddPerson")
    public String addPerson(@RequestBody PersonDTO personDTO) {

        //GET DATA FROM PersonDTO
        String name = personDTO.getName();
        Integer age = personDTO.getAge();

        //RETURN SOMETHING
        return name + " is " + age + " years old";

    }

}
```



## Results

- Start Postman

POST

```
http://localhost:8080/addAuthor
```

Headers

(add Key-Value)

```
Content-Type: application/json
```

Body

(option: raw)

```
{
  "name" : "John",
  "age"   : 20
}
```

Postman

The screenshot shows the Postman application window. At the top, there's a navigation bar with File, Edit, View, Help, Home, Workspaces, Reports, Explore, and a search bar. Below the navigation is a toolbar with various icons. The main area shows a list of requests under 'DTO / New Request'. A specific request is selected, showing a 'GET' method and the URL 'http://localhost:8080/AddPerson'. The 'Body' tab is active, displaying a JSON payload:

```
1 {
2   "name" : "John",
3   "age"   : 20
4 }
```

Below the body, the response status is shown as 200 OK with a response time of 99 ms and a size of 184 B. The response content is displayed as 'John is 20 years old'. There are tabs for Body, Cookies, Headers (5), and Test Results. At the bottom, there are buttons for Pretty, Raw, Preview, Visualize, and Text.

HTTP Response Body

```
John is 20 years old
```

## Application Structure

The screenshot shows the IntelliJ IDEA interface. The left sidebar displays the project structure for 'springboot\_dto\_json\_object\_constructor'. The 'src' folder contains 'main' and 'DTO' subfolders. 'main/java/com.ivoronline.springboot\_dto.controllers' contains 'MyController.java'. 'DTO' contains 'PersonDTO.java'. The code editor shows the content of PersonDTO.java:

```
package com.ivoronline.springboot_dto_json_object_constructor.DTO;

public class PersonDTO {

    //PROPERTIES ARE NOT USED FOR DESERIALIZATION IF THERE IS CONSTRUCTOR
    private String name;
    private Integer age;

    //SETTERS
    public String getName() { return name; }
    public Integer getAge () { return age; }

    //CONSTRUCTOR IS USED FOR DESERIALIZATION
    PersonDTO(String name, Integer age) {
        this.name = name;
        this.age = age;
    }
}
```

The 'target' folder is highlighted in yellow in the project tree.

pom.xml

```
<dependencies>

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

</dependencies>
```

## 2.11.8 From JSON - @RequestBody - Constructor - Custom

### Info

[G]

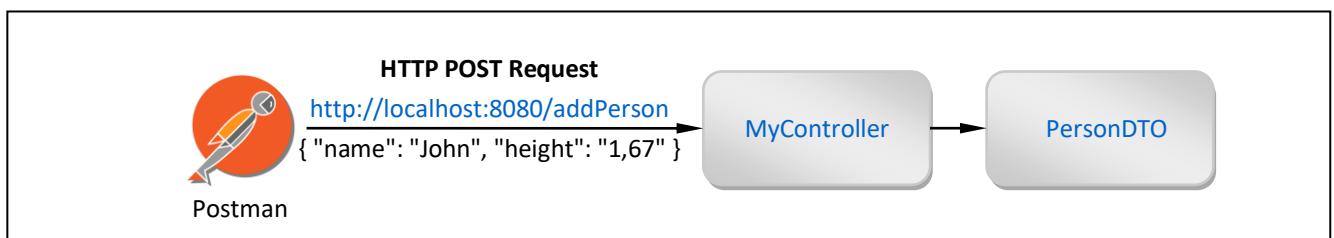
- In the previous tutorial we were using **Custom Deserializer** to convert comma separated Float "1,67" String into Float 1.67
- This tutorial shows how to achieve the same result by using **Default Deserializer** and then fixing just this one Property by
  - adding Constructor to **PersonDTO** to be used for Deserialization
  - so that it can automatically call the Method for reformatting JSON String containing Float

### Syntax

```
@RequestMapping("/AddPerson")
public String addPerson(@RequestBody PersonDTO personDTO) { ... }
```

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables: @Controller, @RequestMapping, Tomcat Server

## Procedure

- Create Project: springboot\_json\_deserializer (add Spring Boot Starters from the table)
- Create Package: DTO (inside main package)
  - Create Class: PersonDTO.java (inside package DTO)
- Create Package: controllers (inside main package)
  - Create Class: MyController.java (inside package controllers)

### PersonDTO.java

```
package com.ivoronline.springboot_json_deserializer.DTO;

public class PersonDTO {

    //PROPERTIES
    public String name;
    public Float height;           //TO STORE FORMATTED FLOAT VALUE 1.67

    //CONSTRUCTOR          //DESERIALIZATION USES CONSTRUCTOR IF AVAILABLE
    PersonDTO(String name, String height) { //STRING HEIGHT TO STORE COMMA SEPARATED JSON STRING "1,67"
        this.name = name;           //DESERIALIZED PROPERTY THAT DOESN'T NEED CUSTOM REFORMATTING
        convertHeight(height);     //PROPERTY THAT NEEDS CUSTOM DESERIALIZATION
    }

    //CONVERTERS
    public void convertHeight(String height) {
        height      = height.replace(",", "."); //HEIGHT FROM JSON: REPLACE DOT WITH COMMA
        this.height = Float.parseFloat(height); //HEIGHT FROM DTO: PARSE STRING INTO FLOAT
    }
}
```

### MyController.java

```
package com.ivoronline.springboot_json_deserializer.controllers;

import com.ivoronline.springboot_json_deserializer.DTO.PersonDTO;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class MyController {

    @ResponseBody
    @RequestMapping("/AddPerson")
    public String addPerson(@RequestBody PersonDTO personDTO) {

        //GET DATA FROM PersonDTO
        String name   = personDTO.name;
        Float height = personDTO.height;

        //RETURN SOMETHING
        return name + " is " + height + " meters high";
    }
}
```

## Results

- Start Postman

POST

```
http://localhost:8080/AddPerson
```

Headers

(add Key-Value)

```
Content-Type: application/json
```

Body

(option: raw)

```
{
  "name" : "John",
  "height" : "1,67"
}
```

Postman

The screenshot shows the Postman application window. In the top navigation bar, 'File', 'Edit', 'View', and 'Help' are visible. Below the bar, there are tabs for 'Home', 'Workspaces', 'Reports', and 'Explore'. A search bar says 'Search Postman'. On the right side of the header, there are icons for cloud storage, a gear, a bell, and a refresh symbol, followed by 'Upgrade'.

In the main workspace, a list of requests is shown: 'GET' (blue), 'POST' (orange), 'GET' (green), '[CONFL...' (red), 'GET' (green), '[CONFL...' (red), '[CONFL...' (red), and '[CONFL...' (red). Below this, a section titled 'DTO / New Request' is displayed. It shows a 'GET' request to 'http://localhost:8080/AddPerson'. The 'Body' tab is selected, showing the JSON payload:

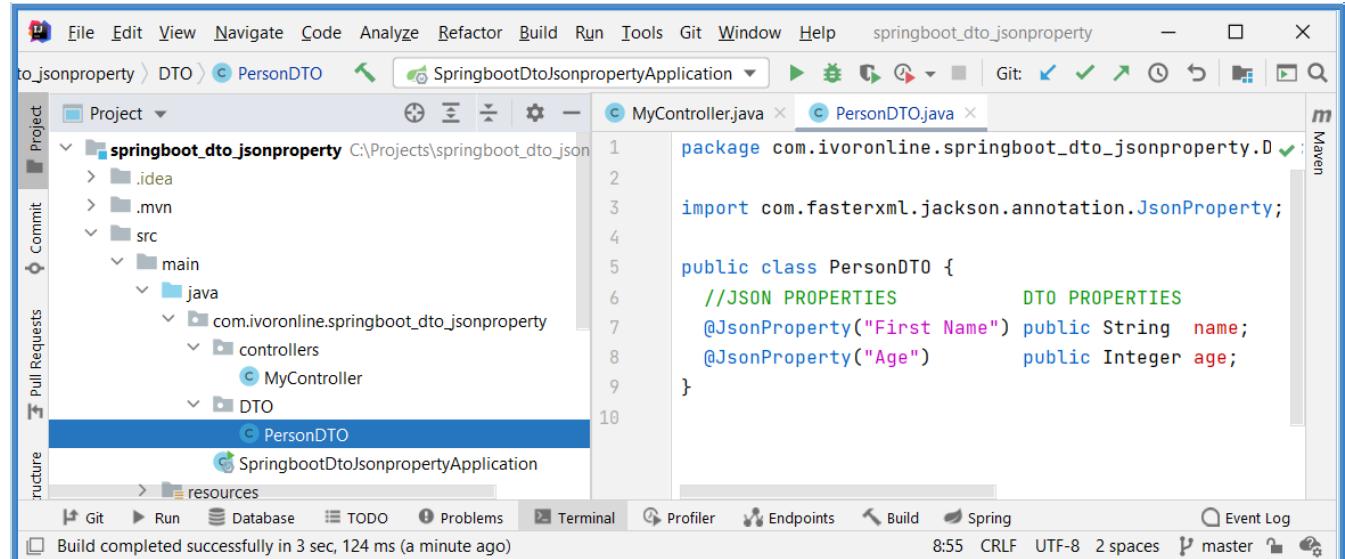
```
1 {
2   "name" : "John",
3   "height" : "1,67"
4 }
```

The 'Tests' tab shows the response: 'Status: 200 OK Time: 124 ms Size: 188 B'. Below the body, there are tabs for 'Pretty', 'Raw', 'Preview', 'Visualize', and 'Text'. The 'Text' tab is currently selected, displaying the response message: '1 John is 1.67 meters high'. At the bottom of the window, there are buttons for 'Find and Replace' and 'Console', along with environment and runner buttons.

HTTP Response Body

```
John is 1.67 meters high
```

## Application Structure



pom.xml

```
<dependencies>

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

</dependencies>
```

## 2.11.9 From JSON - @JsonProperty

### Info

[G] [R]

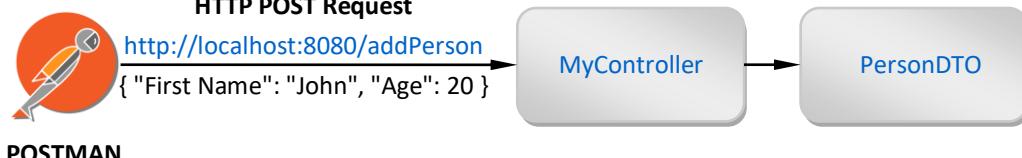
- This tutorial shows how to use `@JsonProperty` Annotation to map Properties from **JSON** HTTP Request into **DTO**. (You can also use it to map to Entity if you are not using DTO between Controller and Service Layer).
- `@JsonProperty` Annotation can be used to
  - Annotate public Properties (when public Properties are used for Deserialization in absence of Constructor)
  - Annotate Constructor Parameters (when Constructor is used for Deserialization)

### Syntax

```
@JsonProperty("First Name") //JSON PROPERTY  
public String name; //DTO PROPERTY
```

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables: <code>@Controller</code> <code>@RequestMapping</code> , Tomcat Server

## Procedure

---

- Create Project: bootspring\_http\_requestbody (add Spring Boot Starters from the table)
- Create Package: DTO (inside main package)
  - Create Class: PersonDTO.java (inside package controllers)
- Create Package: controllers (inside main package)
  - Create Class: MyController.java (inside package controllers)

### PersonDTO.java

```
package com.ivoronline.springboot_dto_jsonproperty.DTO;

import com.fasterxml.jackson.annotation.JsonProperty;

public class PersonDTO {

    //JSON PROPERTIES          //DTO PROPERTIES
    @JsonProperty("First Name") public String name;      //Completely different name (and with space)
    @JsonProperty("Age")        public Integer age;       //Uppercase A

}
```

### MyController.java

```
package com.ivoronline.springboot_dto_jsonproperty.controllers;

import com.ivoronline.springboot_dto_jsonproperty.DTO.PersonDTO;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class MyController {

    @ResponseBody
    @RequestMapping("/AddPerson")
    public String addPerson(@RequestBody PersonDTO personDTO) {

        //GET DATA FROM PersonDTO
        String name = personDTO.name;
        Integer age = personDTO.age;

        //RETURN SOMETHING
        return name + " is " + age + " years old";

    }
}
```

## Results

- Start Postman

POST

```
http://localhost:8080/AddPerson
```

Headers

(add Key-Value)

```
Content-Type: application/json
```

Body

(option: raw)

```
{
  "First Name" : "John",
  "Age"        : 20
}
```

Postman

The screenshot shows the Postman application window. At the top, there's a navigation bar with 'File', 'Edit', 'View', 'Help', and various icons. Below the bar, the main interface has tabs for 'Home', 'Workspaces', 'Reports', and 'Explore'. A search bar says 'Search Postman'. On the right side, there are several environment dropdowns and a 'Upgrade' button.

In the center, there's a 'DTO / New Request' section. It shows a 'GET' method selected, with the URL 'http://localhost:8080/AddPerson'. Below the URL, there are tabs for 'Params', 'Authorization', 'Headers (9)', 'Body', 'Pre-request Script', 'Tests', and 'Settings'. The 'Body' tab is active, showing a JSON payload:

```
1 {
2   "First Name" : "John",
3   "Age"        : 20
4 }
```

Below the body, there are tabs for 'Body', 'Cookies', 'Headers (5)', and 'Test Results'. The 'Body' tab is active, showing the response: 'Status: 200 OK Time: 105 ms Size: 184 B'. The response content is: '1 John is 20 years old'. There are also 'Pretty', 'Raw', 'Preview', 'Visualize', and 'Text' buttons. At the bottom, there are buttons for 'Find and Replace' and 'Console', along with 'Bootcamp', 'Runner', and 'Trash' links.

HTTP Response Body

```
John is 20 years old
```

## Application Structure

The screenshot shows the IntelliJ IDEA interface. The left sidebar displays the project structure under 'springboot\_dto\_jsonproperty'. The 'src' folder contains 'main' and 'java'. The 'java' folder contains 'com.ivoronline.springboot\_dto\_jsonproperty' which has 'controllers' and 'DTO' subfolders. 'MyController.java' and 'PersonDTO.java' are open in the code editor. The code for PersonDTO is as follows:

```
package com.ivoronline.springboot_dto_jsonproperty.D ✓;

import com.fasterxml.jackson.annotation.JsonProperty;

public class PersonDTO {
    //JSON PROPERTIES           DTO PROPERTIES
    @JsonProperty("First Name") public String name;
    @JsonProperty("Age")        public Integer age;
}
```

The bottom status bar shows 'Build completed successfully in 3 sec, 124 ms (a minute ago)'. The toolbar includes icons for Git, Run, Database, TODO, Problems, Terminal, Profiler, Endpoints, Build, Spring, and Event Log.

pom.xml

```
<dependencies>

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

</dependencies>
```

## 2.11.10 From JSON - @JsonFormat

### Info

[G] [R]

- This tutorial shows how to use `@JsonFormat` to specify **Date Format of JSON Property**. That way Date can be properly loaded from JSON Property into DTO Property.  
In this tutorial JSON Property `birthday 25.02.2021` is converted into `PersonDTO` Property in `LocalDate` Format `2021-02-25`.
- `@JsonFormat` Annotation can be used to
  - Annotate public Properties (when public Properties are used for Deserialization in absence of Constructor)
  - Annotate Constructor Parameters (when Constructor is used for Deserialization)

### Syntax

```
@JsonFormat(pattern="dd.MM.yyyy") //Define which date format is used by JSON Property: dd.mm.yyyy 25.02.2021  
public LocalDate birthday;           //It will be converted to LocalDate format:           yyyy-mm-dd 2021-02-25
```

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables: <code>@Controller</code> <code>@RequestMapping</code> , Tomcat Server

## Procedure

---

- [Create Project:](#) bootspring\_http\_requestbody (add Spring Boot Starters from the table)
- [Create Package:](#) DTO (inside main package)
  - [Create Class:](#) PersonDTO.java (inside package controllers)
- [Create Package:](#) controllers (inside main package)
  - [Create Class:](#) MyController.java (inside package controllers)

### PersonDTO.java

```
package com.ivorononline.springboot_dto_jsonproperty.DTO;

import com.fasterxml.jackson.annotation.JsonFormat;
import java.time.LocalDate;

public class PersonDTO {

    public String name;

    //DEFINE WHICH DATE FORMAT IS USED BY JSON PROPERTY: dd.MM.yyyy 25.02.2021
    //IT WILL BE CONVERTED TO LOCALDATE FORMAT:             yyyy-MM-dd 2021-02-25
    @JsonFormat(pattern="dd.MM.yyyy")
    public LocalDate birthday;

}
```

### MyController.java

```
import java.time.LocalDate;

@Controller
public class MyController {

    @ResponseBody
    @RequestMapping("/AddPerson")
    public String addPerson(@RequestBody PersonDTO personDTO) {

        //GET DATA FROM PersonDTO
        String name = personDTO.name;
        LocalDate birthday = personDTO.birthday;

        //RETURN SOMETHING
        return name + " is born on " + birthday;

    }
}
```

## Results

- Start Postman

POST

```
http://localhost:8080/AddPerson
```

Headers

(add Key-Value)

```
Content-Type: application/json
```

Body

(option: raw)

```
{
  "name"      : "John",
  "birthday"  : "25.02.2021"
}
```

Postman

The screenshot shows the Postman application window. The top navigation bar includes File, Edit, View, Help, Home, Workspaces, Reports, Explore, and a search bar. Below the navigation is a toolbar with various icons. The main workspace shows a list of recent requests and a 'New Request' button. A 'GET' request is selected, pointing to the URL `http://localhost:8080/AddPerson`. The 'Body' tab is active, containing the following JSON payload:

```
1 {
2   "name"      : "John",
3   "birthday"  : "25.02.2021"
4 }
```

The status bar at the bottom indicates a successful response: Status: 200 OK, Time: 113 ms, Size: 190 B, and a 'Save Response' button.

HTTP Response Body

```
John is born on 2021-02-25
```

## Application Structure

The screenshot shows the IntelliJ IDEA interface. The left sidebar displays the project structure under 'springboot\_dto\_jsonproperty'. The 'src' folder contains 'main' and 'java'. The 'java' folder contains 'com.ivoronline.springboot\_dto\_jsonproperty' which has 'controllers' and 'DTO' subfolders. 'MyController.java' and 'PersonDTO.java' are open in the code editor. The code for PersonDTO is as follows:

```
package com.ivoronline.springboot_dto_jsonproperty.D ✓;

import com.fasterxml.jackson.annotation.JsonProperty;

public class PersonDTO {
    //JSON PROPERTIES           DTO PROPERTIES
    @JsonProperty("First Name") public String name;
    @JsonProperty("Age")        public Integer age;
}
```

The bottom status bar shows 'Build completed successfully in 3 sec, 124 ms (a minute ago)'. The toolbar includes icons for Git, Run, Database, TODO, Problems, Terminal, Profiler, Endpoints, Build, Spring, and Event Log.

pom.xml

```
<dependencies>

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

</dependencies>
```

## 2.11.11 From JSON - @JsonDeserialize

### Info

[G] [R]

- This tutorial shows how to create custom Deserializer to convert **JSON** HTTP Request into DTO in Controller.
- Deserializer is used to
  - convert JSON Property `height "1,67"` (String which uses comma and can't be converted into Float)
  - into DTO Property `height "1.67"` (String which uses dot and can be converted into Float)
- Deserializer is
  - implemented as
  - attached to PersonDTO with
  - called as Controller Input

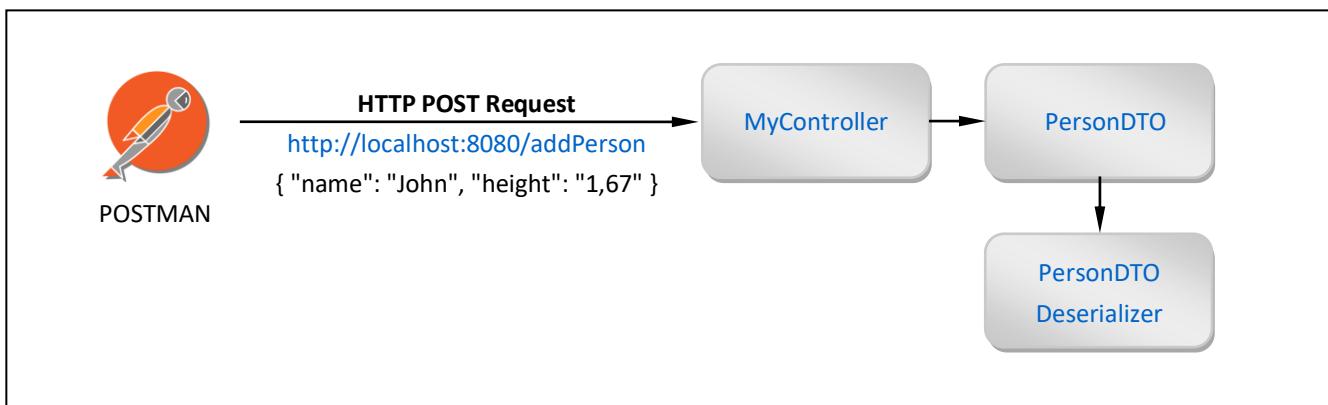
```
class PersonDTODeserializer extends JsonDeserializer<PersonDTO>
@JsonDeserialize(using = PersonDTODeserializer.class)
addPerson(@RequestBody PersonDTO personDTO)
```

### Syntax

```
@JsonDeserialize(using = PersonDTODeserializer.class)
public class PersonDTO { ... }
```

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables: @Controller, @RequestMapping, Tomcat Server

## Procedure

- Create Project: `springboot_json_deserializer` (add Spring Boot Starters from the table)
- Create Package: `DTO` (inside main package)
  - Create Class: `PersonDTODeserializer.java` (inside package DTO)
  - Create Class: `PersonDTO.java` (inside package DTO)
- Create Package: `controllers` (inside main package)
  - Create Class: `MyController.java` (inside package controllers)

### *PersonDTODeserializer.java*

```
package com.ivoronline.springboot_json_deserializer.DTO;

import com.fasterxml.jackson.core.JsonParser;
import com.fasterxml.jackson.core.ObjectCodec;
import com.fasterxml.jackson.databind.DeserializationContext;
import com.fasterxml.jackson.databind.JsonDeserializer;
import com.fasterxml.jackson.databind.JsonNode;
import java.io.IOException;

public class PersonDTODeserializer extends JsonDeserializer<PersonDTO> {

    @Override
    public PersonDTO deserialize(JsonParser jsonParser, DeserializationContext ctxt) throws IOException {

        //PREPARE JSON STUFF
        ObjectCodec objectCodec = jsonParser.getCodec();
        JsonNode node = objectCodec.readTree(jsonParser);

        //DESERIALIZE PROBLEMATIC JSON PROPERTY: HEIGHT
        String heightString = node.get("height").asText();
        heightString = heightString.replace(',', '.'); //Convert "1,67" into "1.67"
        Float height = Float.parseFloat(heightString); //Create Float from String

        //CREATE PERSONDTO
        PersonDTO personDTO = new PersonDTO();
        personDTO.name = node.get("name").asText(); //NORMAL PROPERTY
        personDTO.height = height; //PROBLEMATIC PROPERTY

        //RETURN PERSONDTO
        return personDTO;
    }
}
```

### *PersonDTO.java*

```
package com.ivoronline.springboot_json_deserializer.DTO;

import com.fasterxml.jackson.databind.annotation.JsonDeserialize;

@JsonDeserialize(using = PersonDTODeserializer.class)
public class PersonDTO {
    public String name;
    public Float height;
}
```

### MyController.java

```
package com.ivorononline.springboot_json_deserializer.controllers;

import com.ivorononline.springboot_json_deserializer.DTO.PersonDTO;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class MyController {

    @ResponseBody
    @RequestMapping("/AddPerson")
    public String addPerson(@RequestBody PersonDTO personDTO) {

        //GET DATA FROM PersonDTO
        String name = personDTO.name;
        Float height = personDTO.height;

        //RETURN SOMETHING
        return name + " is " + height + " meters high";
    }
}
```

## Results

- Start Postman

POST

```
http://localhost:8080/AddPerson
```

Headers

(add Key-Value)

```
Content-Type: application/json
```

Body

(option: raw)

```
{
  "name" : "John",
  "height" : "1,67"
}
```

Postman

The screenshot shows the Postman application window. The top navigation bar includes File, Edit, View, Help, Home, Workspaces, Reports, Explore, and a search bar. Below the navigation is a toolbar with various icons. The main workspace shows a list of requests and a 'New Request' button. A specific request is selected: a GET request to 'http://localhost:8080/AddPerson'. The 'Body' tab is active, showing a JSON payload:

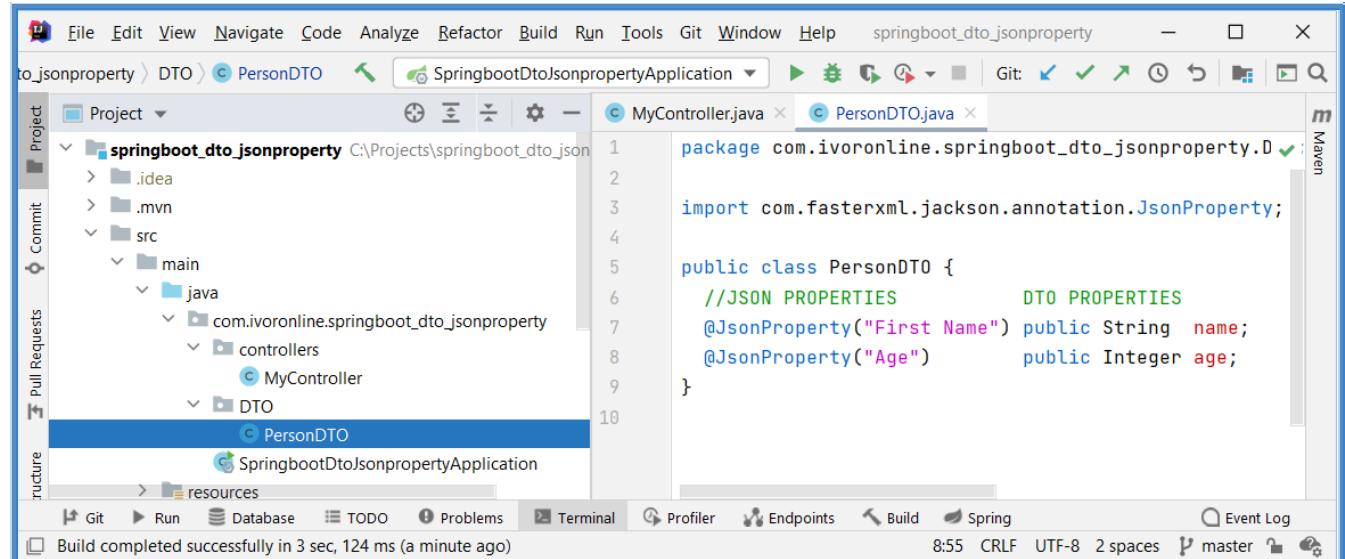
```
1 {
2   "name" : "John",
3   "height" : "1,67"
4 }
```

The status bar at the bottom indicates a successful response: Status: 200 OK, Time: 124 ms, Size: 188 B, and a 'Save Response' button.

HTTP Response Body

```
John is 1.67 meters high
```

## Application Structure



pom.xml

```
<dependencies>

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

</dependencies>
```

## 2.11.12 From JSON Array

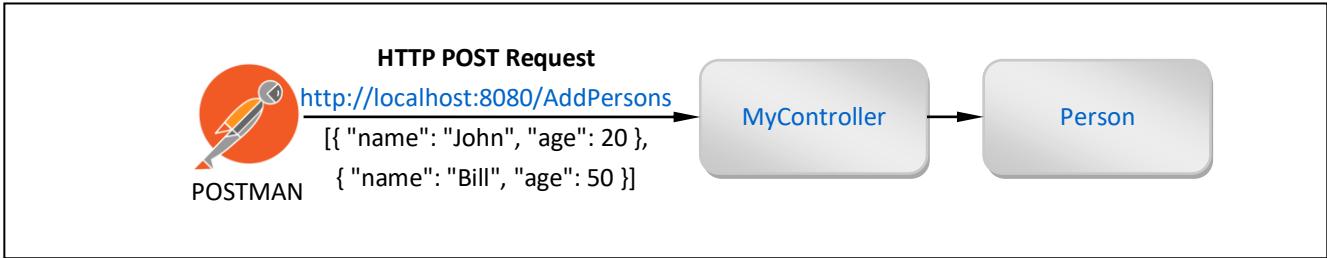
### Info

[G]

- This tutorial shows how to use `@RequestBody` Annotation to convert **JSON Array** from HTTP Request into **List** of Objects.

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables <code>@Controller</code> , <code>@RequestMapping</code> and Tomcat Server

## Procedure

- Create Project: `springboot_httprequest_requestbody_json_array` (add Spring Boot Starters from the table)
- Create Package: `controllers` (inside main package)
  - Create Class: `MyController.java` (inside package controllers)
- Create Package: `entities` (inside main package)
  - Create Class: `Person.java` (inside package controllers)

*Person.java*

```
package com.ivoronline.springboot_httprequest_requestbody_json_array.entities;

public class Person {
    public String name;
    public Integer age;
}
```

*MyController.java*

```
package com.ivoronline.springboot_httprequest_requestbody_json_array.controllers;

import com.ivoronline.springboot_httprequest_requestbody_json_array.entities.Person;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RequestMapping;
import java.util.List;

@Controller
public class MyController {

    @ResponseBody
    @RequestMapping("/AddPersons")
    public String addPersons(@RequestBody List<Person> persons) {

        //ITERATE THROUGH LIST
        String result = "";
        for (Person person : persons) {
            String name = person.name;
            Integer age = person.age;
            result += name + " is " + age + " years old \n";
        }

        //RETURN SOMETHING
        return result;
    }
}
```

## Results

- Start Postman
- POST: <http://localhost:8080/AddPersons>
- Headers: (copy from below)
- Body: (copy from below)
- Send

Headers

(add Key-Value)

```
Content-Type: application/json
```

Body

(option: raw)

```
[  
  {  
    "name" : "Jack",  
    "age" : "20"  
  },  
  
  {  
    "name" : "Bill",  
    "age" : "50"  
  }  
]
```

POST <http://localhost:8080/AddPersons>

The screenshot shows the Postman application window. The top navigation bar includes File, Edit, View, Help, Home, Workspaces, Reports, Explore, and a search bar. Below the navigation is a toolbar with various icons. The main workspace shows a collection named 'MyCollection / MyRequest' with a single POST request. The request details show the URL as 'http://localhost:8080/AddPersons'. The 'Body' tab is selected, showing the JSON array defined above. The 'Pretty' tab in the preview section displays the response: 'Jack is 20 years old' and 'Bill is 50 years old'. The status bar at the bottom indicates a 200 OK status with a time of 145 ms and a size of 208 B.

HTTP Response Body

```
Jack is 20 years old  
Bill is 50 years old
```

## Application Structure

The screenshot shows the IntelliJ IDEA interface with the following details:

- Project Tree:** Shows the project structure under "springboot\_httprequest\_requestbody\_json\_array". The "controllers" package contains a file named "MyController.java".
- Code Editor:** Displays the content of "MyController.java". The code defines a controller method "addPersons" that takes a list of "Person" objects from the request body and returns a string result.
- Code Navigation:** A tooltip for the annotation "@RequestMapping" is visible, pointing to its documentation.
- Toolbars and Status Bar:** Standard IntelliJ toolbars and a status bar at the bottom indicating the current time and file status.

pom.xml

```
<dependencies>

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

</dependencies>
```

## 2.12 DTO to Entity

Info

---

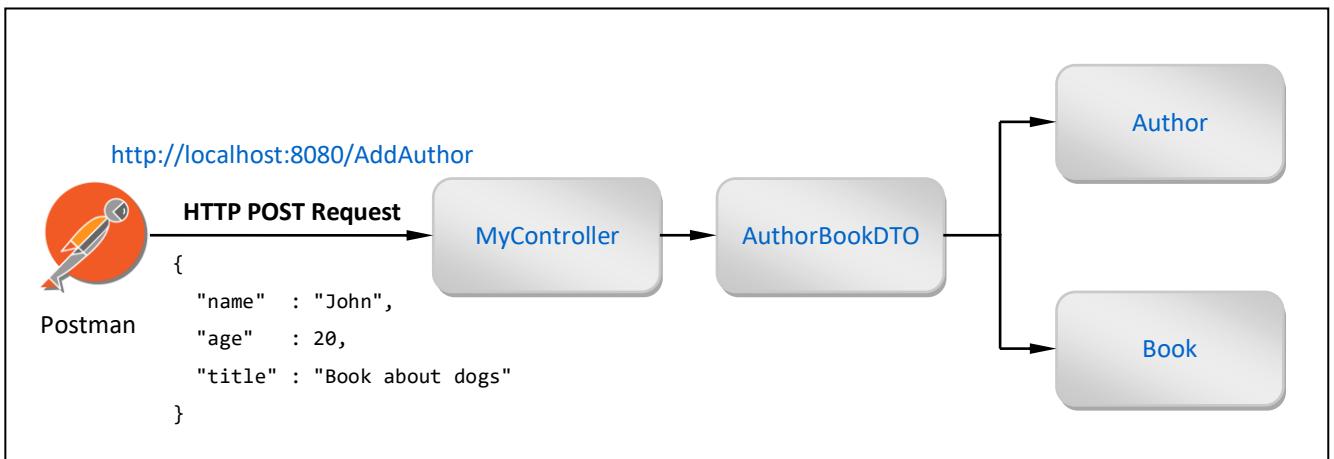
## 2.12.1 Manually

### Info

- This tutorial shows how to **manually** Convert DTO into Entities.
- Tutorial Annotation - (Spring) `@RequestBody` shows how to convert JSON data from HTTP Request into DTO.

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables <code>@Controller</code> and <code>@RequestMapping</code> . Includes Tomcat Server.

### Procedure

- **Create Project:** service (add Spring Boot Starters from the table)
- **Create Package:** entities (inside main package)
  - Create Class: `Author.java` (inside package entities)
  - Create Class: `Book.java` (inside package entities)
- **Create Package:** DTOs (inside main package)
  - Create Class: `AuthorBookDTO.java` (inside package controllers)
- **Create Package:** controllers (inside main package)
  - Create Class: `MyController.java` (inside package controllers)

### Author.java

```
package com.ivoronline.springboot_dto_modelmapper.entities;

public class Author {
    public Integer id;
    public String name;
    public Integer age;
}
```

### Book.java

```
package com.ivoronline.springboot_dto_modelmapper.entities;

public class Book {
    public Integer id;
    public String title;
}
```

### *AuthorBookDTO.java*

```
package com.ivorononline.springboot_dto_modelmapper.DTOs;

public class AuthorBookDTO {
    public String name;
    public Integer age;
    public String title;
}
```

### *MyController.java*

```
package com.ivorononline.springboot_dto_modelmapper.controllers;

import com.ivorononline.springboot_dto_modelmapper.DTOs.AuthorBookDTO;
import com.ivorononline.springboot_dto_modelmapper.entities.Author;
import com.ivorononline.springboot_dto_modelmapper.entities.Book;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class MyController {

    @ResponseBody
    @RequestMapping("AddAuthorBook")
    public String addAuthorBook(@RequestBody AuthorBookDTO authorBookDTO) {

        //CONVERT DTO TO AUTHOR ENTITY
        Author author = new Author();
        author.name = authorBookDTO.name;
        author.age = authorBookDTO.age;

        //CONVERT DTO TO BOOK ENTITY
        Book book = new Book();
        book.title = authorBookDTO.title;

        //RETURN SOMETHING
        return author.name + " has written " + book.title;
    }
}
```

## Results

- Start Postman
- POST: <http://localhost:8080/AddAuthorBook>
- Headers: (copy from below)
- Body: (copy from below)
- Send

### Headers

(add Key-Value)

```
Content-Type: application/json
```

### Body

(option: raw)

```
{
  "name" : "John",
  "age" : 20
  "title" : "Book about dogs"
}
```

### HTTP Response Body

```
John has written Book about dogs
```

### HTTP Request Headers (Bulk Edit)

The screenshot shows the Postman interface with a request titled "MyRequest". The method is set to POST, and the URL is http://localhost:8080/addAuthorBook. The "Headers" tab is selected, showing the Content-Type header set to application/json. The "Body" tab is also visible. The response body is displayed as "John has written Book about dogs".

### Request JSON Body (raw) & Response Body

The screenshot shows the Postman interface with a request titled "MyRequest". The method is set to POST, and the URL is http://localhost:8080/addAuthorBook. The "Headers" tab is selected, showing 11 headers. The "Body" tab is selected and set to "raw" mode, displaying a JSON object with three fields: name, age, and title. The response body is displayed as "John has written Book about dogs".

## Application Structure

The screenshot shows the IntelliJ IDEA interface with the following details:

- Project View:** Shows the project structure under "springboot(dto)\_modelmapper". The "src/main/java" package contains "com.ivoronline.springboot(dto)\_modelmapper/controllers" and "com.ivoronline.springboot(dto)\_modelmapper/DTOs". "MyController.java" is selected in the editor.
- Code Editor:** Displays the content of `MyController.java`. The code defines a controller method `addAuthorBook` that takes a `AuthorBookDTO` and returns a string. It uses ModelMapper to convert the DTO to an Author entity and the Book entity, then returns a concatenated string.
- Toolbars and Status Bar:** Standard IntelliJ IDEA toolbars and status bar showing file paths, file status, and system information.

## pom.xml

```
<dependencies>

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

</dependencies>
```

## 2.12.2 JMapper - Using Annotations

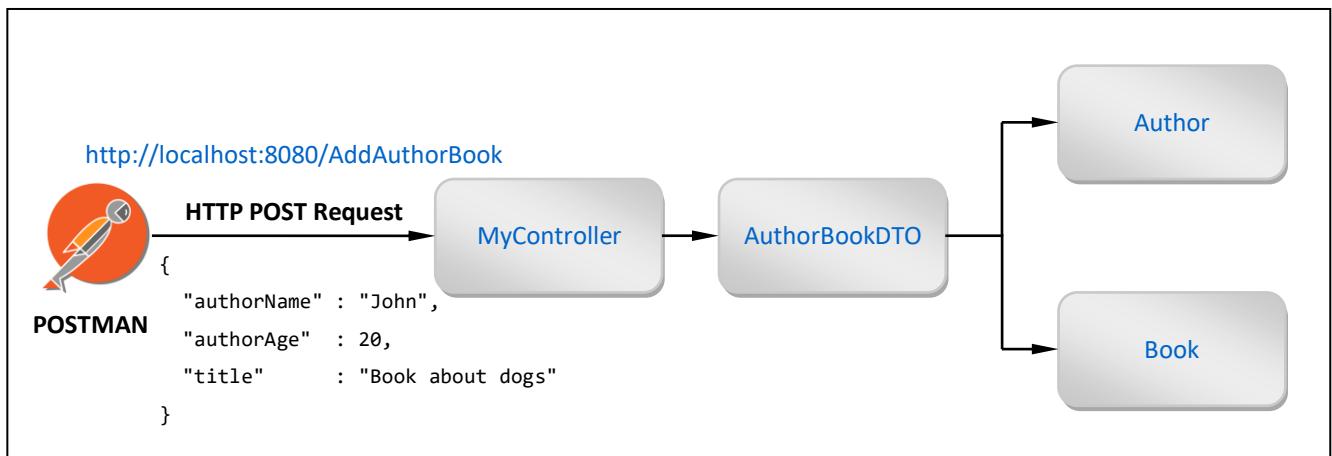
### Info

[R]

- This tutorial shows how to use JMapper's **Annotations** to convert `AuthorBookDTO` into `Author` and `Book` Entities.
- `AuthorBookDTO` is instantiated from Postman's **JSON** HTTP Request as described in [Annotation - \(Spring\) @RequestBody](#).
- Alternative to using JMapper is [Model Mapper](#).

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables <code>@Controller</code> and <code>@RequestMapping</code> . Includes Tomcat Server.
Developer Tools	Lombok	Enables <code>@Data</code> which generate helper methods (setters, getters, ...)

### Procedure

- Create Project: `springboot_dtojmapper` (add Spring Boot Starters from the table)
- Edit File: `pom.xml` (manually add JMapper dependency)
- Create Package: `entities` (inside main package)
  - Create Class: `Author.java` (inside package entities)
  - Create Class: `Book.java` (inside package entities)
- Create Package: `DTOs` (inside main package)
  - Create Class: `AuthorBookDTO.java` (inside package controllers)
- Create Package: `controllers` (inside main package)
  - Create Class: `MyController.java` (inside package controllers)

### pom.xml

```
<dependency>  
    <groupId>com.googlecode.jmapper-framework</groupId>  
    <artifactId>jmapper-core</artifactId>  
    <version>1.6.0.1</version>  
</dependency>
```

### *Author.java*

```
package com.ivoronline.springboot_dto_jmapper.entities;

import com.googlecode.jmapper.annotations.JMap;
import lombok.Data;

@Data
public class Author {

    public Integer id;      //Can't be mapped since there is no Annotation

    @JMap("authorName")    //Name of the Source Property
    public String name;

    @JMap("authorAge")     //Name of the Source Property
    public Integer age;

}
```

### *Book.java*

```
package com.ivoronline.springboot_dto_jmapper.entities;

import com.googlecode.jmapper.annotations.JMap;
import lombok.Data;

@Data
public class Book {

    public Integer id;

    @JMap           //Source Property should have the same name
    public String title;

}
```

### *AuthorBookDTO.java*

```
package com.ivoronline.springboot_dto_jmapper.DTO;

import lombok.Data;

@Data
public class AuthorBookDTO {
    public String authorName;
    public Integer authorAge;
    public String title;
}
```

### MyController.java

```
package com.ivorononline.springboot_jmapper.controllers;

import com.googlecode.jmapper.JMapper;
import com.ivorononline.springboot_jmapper.DTO.AuthorBookDTO;
import com.ivorononline.springboot_jmapper.entities.Author;
import com.ivorononline.springboot_jmapper.entities.Book;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class MyController {

    @ResponseBody
    @RequestMapping("AddAuthorBook")
    public String addAuthorBook(@RequestBody AuthorBookDTO authorBookDTO) {

        //INSTANTIATE JMAPPERS<DESTINATION, SOURCE>
        JMapper<Book, AuthorBookDTO> bookMapper = new JMapper<>(Book .class, AuthorBookDTO.class);
        JMapper<Author, AuthorBookDTO> authorMapper = new JMapper<>(Author.class, AuthorBookDTO.class);

        //MAP AuthorBookDTO TO AUTHOR & BOOK.
        Book book = bookMapper .getDestination(authorBookDTO);
        Author author = authorMapper.getDestination(authorBookDTO);

        //RETURN SOMETHING
        return author.name + " has written: " + book.title;

    }
}
```

## Results

- Start Postman
- POST: <http://localhost:8080/AddAuthorBook>
- Headers: (copy from below)
- Body: (copy from below)
- Send

### Headers

(add Key-Value)

```
Content-Type: application/json
```

### Body

(option: raw)

```
{
  "name" : "John",
  "age" : 20,
  "title" : "Book about dogs"
}
```

### HTTP Response Body

```
John has written Book about dogs
```

### HTTP Request Headers (Bulk Edit)

The screenshot shows the Postman interface with a request named "MyRequest". The method is set to POST, and the URL is http://localhost:8080/addAuthorBook. The "Headers" tab is selected, showing the following configuration:

```
Content-Type: application/json
Host: localhost
Content-Length: 100
```

The "Body" tab is also visible at the bottom.

### Request JSON Body (raw) & Response Body

The screenshot shows the Postman interface with a request named "MyRequest". The method is set to POST, and the URL is http://localhost:8080/addAuthorBook. The "Headers" tab is selected, showing the following configuration:

```
Content-Type: application/json
Host: localhost
Content-Length: 100
```

The "Body" tab is selected, showing a JSON payload:

```
1  {
2    "name": "John",
3    "age" : 20,
4    "title": "Book about dogs"
5 }
```

The "Body" tab is also visible at the bottom.

## Application Structure

The screenshot shows the IntelliJ IDEA interface with the following details:

- Project Tree:** The project is named "springboot(dto\_jmapper)". It contains a "src" directory with "main" and ".mvn" sub-directories. "main" has "java" and "resources" sub-directories. "java" contains "com.ivoronline.springboot(dto\_jmapper)" which has "controllers" and "entities" packages. "entities" contains "Author" and "Book" classes. "resources" contains "application.properties".
- Code Editor:** The editor shows the "MyController.java" file. The code defines a controller with a method "addAuthorBook" that takes an "AuthorBookDTO" and returns a string. It uses JMapper annotations to map the DTO to entities.
- Pom.xml:** The pom.xml file is visible in the project tree under "resources".
- Status Bar:** The status bar at the bottom right shows "10:30 CRLF UTF-8 2 spaces".

## pom.xml

```
<dependencies>

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

    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <optional>true</optional>
    </dependency>

    <dependency>
        <groupId>com.googlecode.jmapper-framework</groupId>
        <artifactId>jmapper-core</artifactId>
        <version>1.6.0.1</version>
    </dependency>

</dependencies>
```

## 2.13 Profiles

### Info

[R]

- Profiles define which files to include/exclude in/from your application.
- You can specify for which
  - Profile to include Class @Profile("Profile1")
  - Profile to exclude Class @Profile("!Profile1")
  - Profiles to include/exclude Class @Profile({"Profile1", "!Profile2"})
  - Profile to include `application.properties` File application-**Profile1**.properties
- Classes for which Profile is not assigned are always included in the application.  
`application.properties` file for which Profile is not assigned is also always included in the application.
- Profile names are **NOT Case Sensitive**.

## 2.13.1 Assign Profile - To Class

### Info

[R]

- This tutorial shows how to assign Profiles to Classes by specifying for which
  - Profile to include Class `@Profile("Profile1")`
  - Profile to exclude Class `@Profile("!Profile1")`
  - Profiles to include/exclude Class `@Profile({"Profile1", "!Profile2"})`
- Classes for which Profile is not assigned are always included in the application.
- Assigning Profile to a Class is used in combination with `@Autowired` on Interface to tell Spring which Class to instantiate. Such usage is demonstrated in tutorial [Interface - @Profile](#).

*application.properties*

*(specify active Profile)*

```
spring.profiles.active = Profile1
```

*MyServiceImplementation1.java*

*(assign Profile to Class)*

```
@Profile( "Profile1" )  
@Profile({ "Profile2", "Profile3" })  
public class MyServiceImplementation1 implements MyServiceInterface { ... }
```

*MyController.java*

*(instantiate included Implementation/Class)*

```
@Autowired  
MyServiceInterface myService;
```

## 2.13.2 Assign Profile - To application.properties

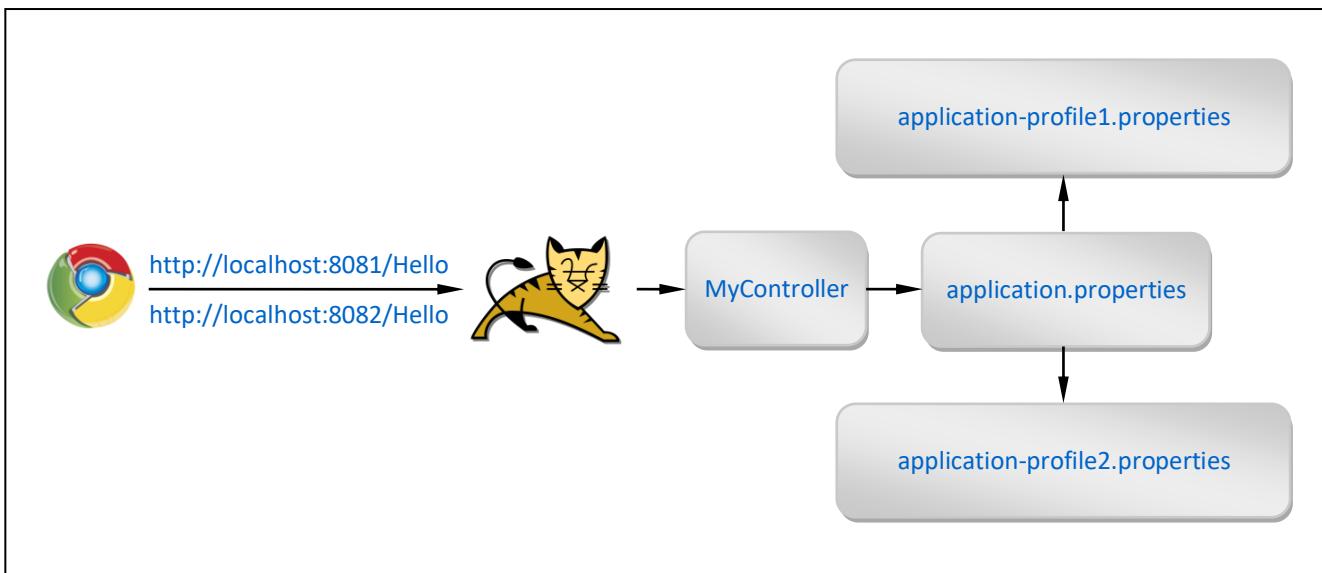
### Info

[R]

- This tutorial shows how to assign Profile to [application-profile1.properties](#) file.  
This is used to switch between test and development environments to specify different: host, port, DB.  
In this tutorial we will use different [application.properties](#) files to specify different ports [8081](#) and [8082](#).
- You can't specify multiple Profiles to a single [application.properties](#) file (like you can with Classes).  
You can't specify for which Profiles to exclude [application.properties](#) file (like you can with Classes).
- Application will always include [application.properties](#) file (for which no Profile is defined).  
In this tutorial we will use [application.properties](#) file to specify Active Profile.

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables <code>@Controller</code> and <code>@RequestMapping</code> . Includes Tomcat Server.

## Procedure

- Create Project: profiles\_applicationproperties (add Spring Boot Starters from the table)
- Edit File: application.properties (inside src/main/resources, specifies active Profile)
- Create Package: controllers (inside main package)
  - Create Class: MyController.java (inside controllers package)
- Create File: application-profile1.properties (inside src/main/resources, specifies Port 8081)
- Create File: application-profile2properties (inside src/main/resources, specifies Port 8082)

application.properties

(specifies active Profile)

```
spring.profiles.active = Profile1
```

application-profile1.properties

(specifies Port 8081)

```
server.port = 8081
```

application-profile2properties

(specifies Port 8082)

```
server.port = 8082
```

MyController.java

```
package com.ivoronlineprofiles_applicationproperties.controllers;

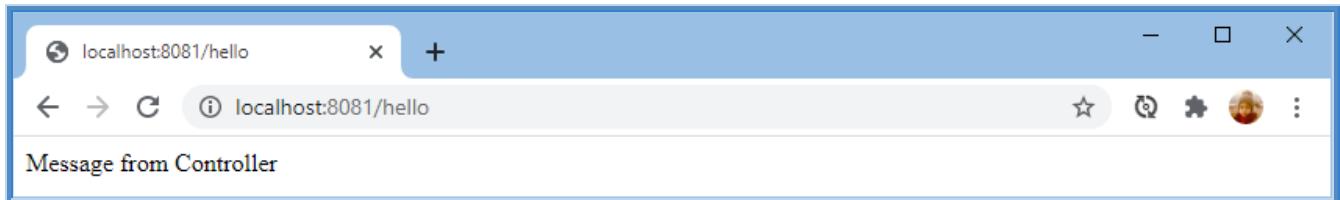
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class MyController {

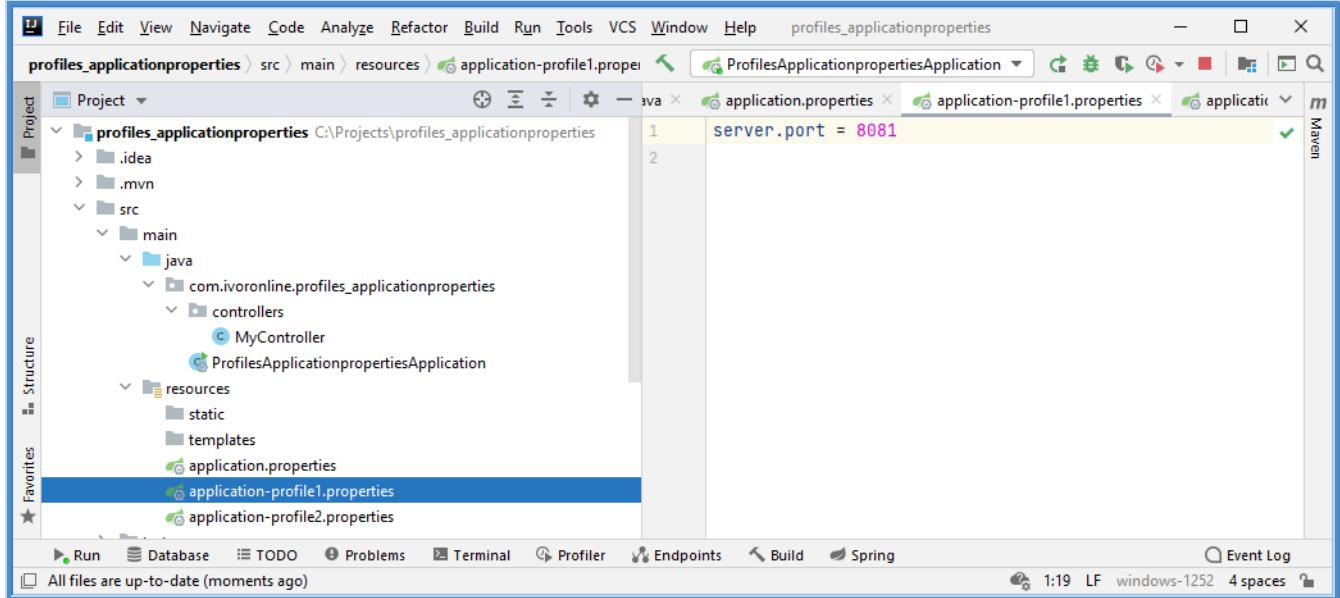
    @ResponseBody
    @RequestMapping("/Hello")
    public String hello() {
        return "Hello from Controller";
    }
}
```

## Results

<http://localhost:8081/hello>



## Application Structure



## pom.xml

```
<dependencies>

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

</dependencies>
```

## 2.13.3 Specify Active Profile - Through Run Configuration

### Info

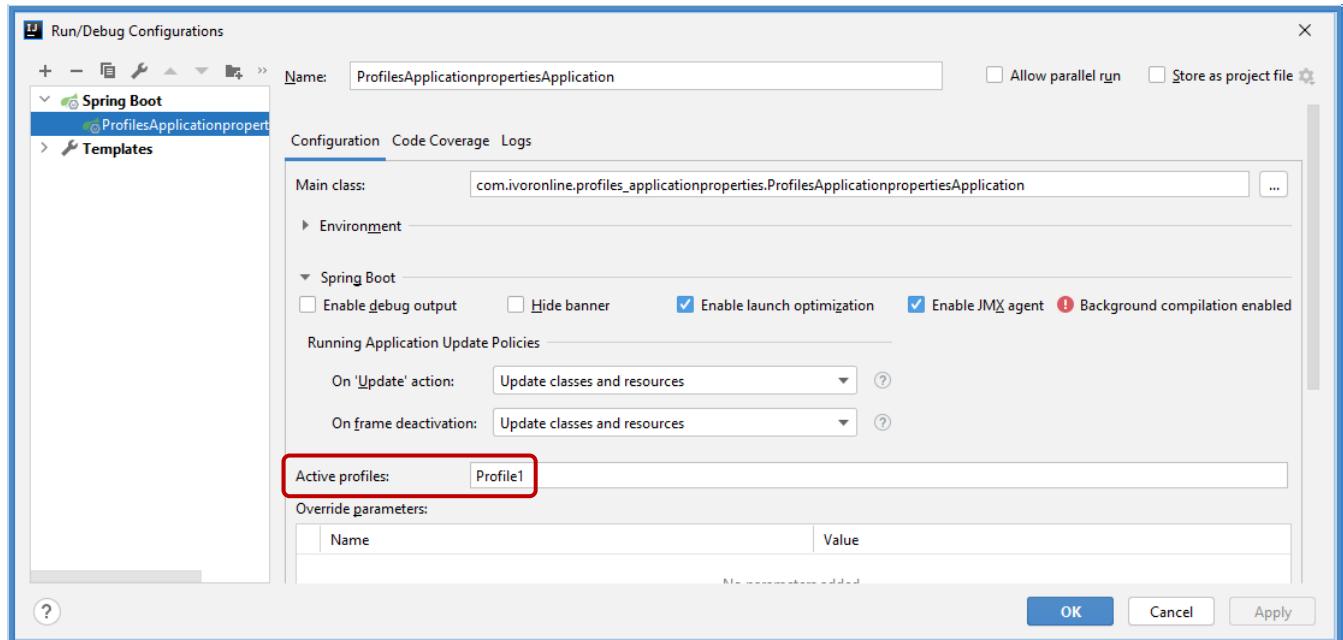
[R]

- This tutorial shows how to specify [Active Profile](#) through [Run Configuration](#).
- You can use [Assign Profile to - application.properties](#) to test changes.

### Specify Active Profile

- Run
- Edit Configurations
- Active profiles: Profile1
- OK

#### Run Configuration



## 2.13.4 Specify Active Profile - Through application.properties

### Info

[R]

- This tutorial shows how to specify Active Profile through `application.properties`.
- `application.properties` will always be included in the application (regardless of Active Profile).
- You can use `Assign Profile to - application.properties` to test changes.

### Specify Active Profile

- Edit File: `application.properties` (inside `src/main/resources`, specifies active Profile)

*application.properties*

(specifies active Profile)

```
spring.profiles.active = Profile1
```

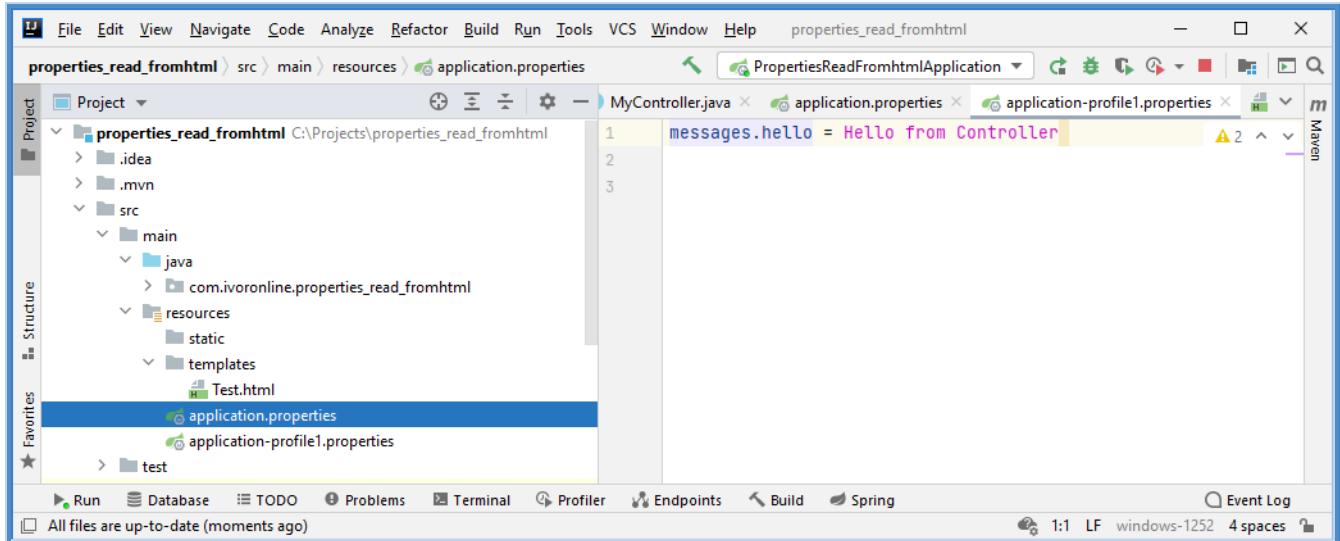
# 2.14 application.properties

## Info

- application.properties file is located in src/main/resources Directory.
- You can create additional Profile specific files as described in [Assign Profile - To application.properties](#)

src/main/resources

(application-profile1.properties)



- Properties inside application.properties file will be automatically used to configure different functionalities: DB, Port.

application.properties

```
# DATABASE
spring.datasource.url      = jdbc:postgresql://localhost:5432/postgres
spring.datasource.username   = postgres
spring.datasource.password   = letmein
spring.datasource.driver-class-name = org.postgresql.Driver

# JPA / HIBERNATE
spring.jpa.hibernate.ddl-auto = create-drop
```

- Properties inside application.properties file can be defined by using either '=' or ':'.

application.properties

```
messages.hello1 : Hello1 from Controller
messages.hello2 = Hello2 from Controller
```

- You can also add additional custom Properties.

All Properties from application.properties file can be referenced from Controller or HTML.

Reference from Controller

```
@Value("${messages.hello1}")
private String message;
```

Reference from HTML

```
<p th:text="${@environment.getProperty('messages.hello1')}">/>
```

## 2.14.1 Read Property - From Controller

### Info

[R]

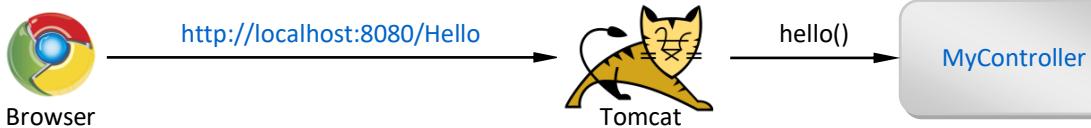
- This tutorial shows how to load Property from `application.properties` into Variable inside Controller.

### Syntax

```
@Value("${messages.hello1}")
private String message;
```

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables @RequestMapping. Includes Tomcat HTTP Server.

### Procedure

- Create Project:** properties\_read\_fromcontroller (add Spring Boot Starters from the table)
- Edit File:** `application.properties` (inside `src/main/resources`)
- Create Package:** controllers (inside main package)
  - Create Class:** `MyController.java` (inside controllers package)

### `application.properties`

```
messages.hello1 : Hello1 from Controller
messages.hello2 = Hello2 from Controller
```

### `MyController.java`

```
package com.ivoronline.properties_read_fromcontroller.controllers;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ResponseBody;

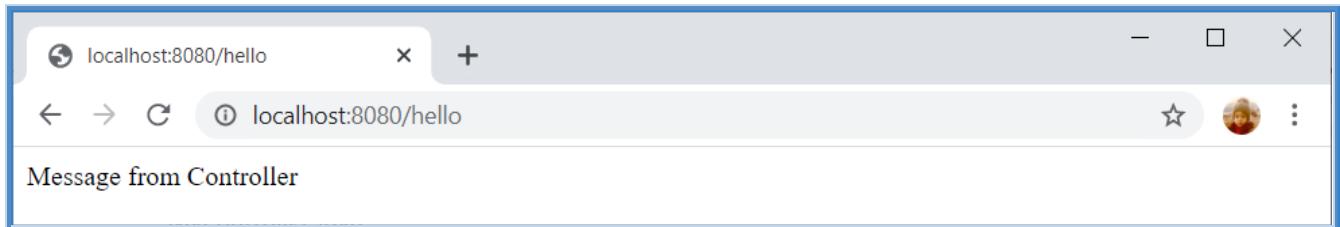
@Controller
public class MyController {

    @Value("${messages.hello1}")
    private String message;

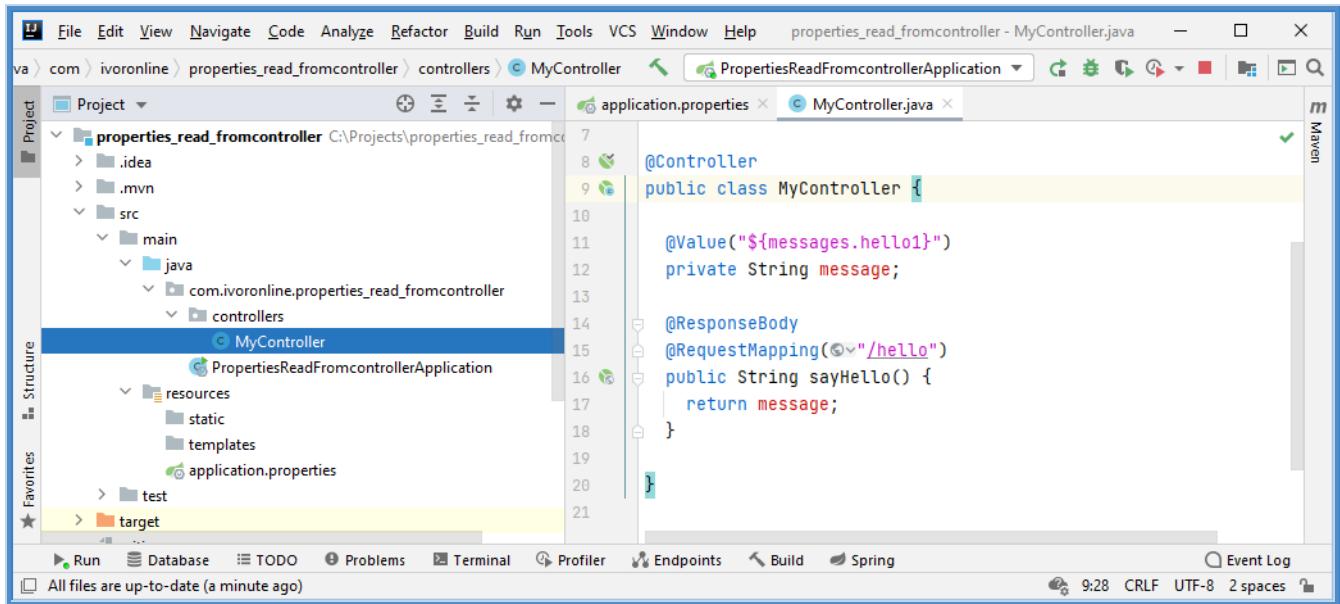
    @ResponseBody
    @RequestMapping("/Hello")
    public String hello() {
        return message;
    }
}
```

## Results

<http://localhost:8080>Hello>



## Application Structure



## pom.xml

```
<dependencies>

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

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

</dependencies>
```

## 2.14.2 Read Property - From HTML

### Info

[R]

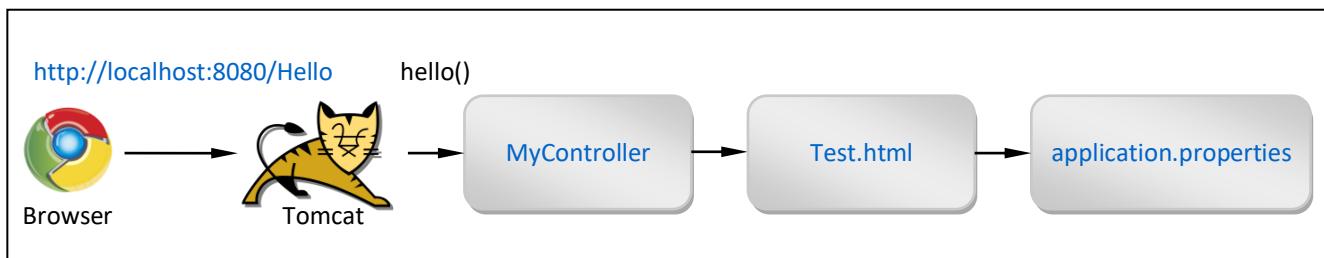
- This tutorial shows how to read Property from `application.properties` from HTML.

### Syntax

```
<p th:text="${@environment.getProperty('messages.hello')}">
```

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables <code>@Controller</code> , <code>@RequestMapping</code> and Tomcat Server.
Template Engines	Thymeleaf	Enables Controller to return reference to HTML file test.html

### Procedure

- Create Project:** properties\_read\_fromhtml (add Spring Boot Starters from the table)
- Edit File:** `application.properties` (inside src/main/resources)
- Create Package:** controllers (inside package com.ivoronline.test\_spring\_boot)
  - Create Class:** `MyController.java` (inside package controllers)
- Create HTML File:** `Test.html` (inside directory resources/templates)

### `application.properties`

```
messages.hello = Hello from Controller
```

### `MyController.java`

```
package com.ivoronline.properties_read_fromhtml.controllers;

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

@Controller
public class MyController {

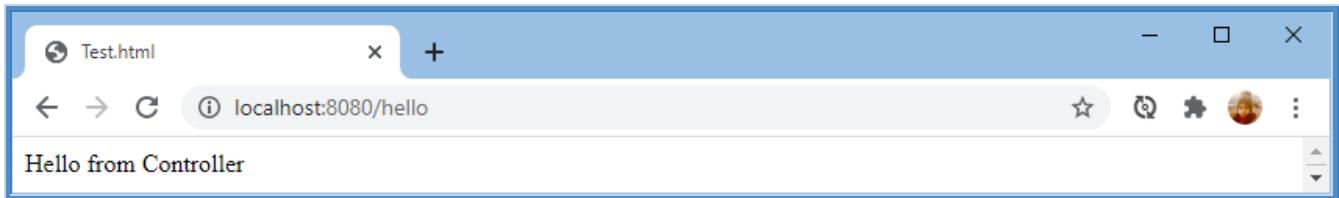
    @RequestMapping("/Hello")
    public String hello() {
        System.out.println("Hello from Controller");
        return "Test";
    }
}
```

### `Test.html`

```
<title>Test.html</title>
<p th:text="${@environment.getProperty('messages.hello')}">
<form th:attr="action=${@environment.getProperty('form.action')}" method="get">
```

## Results

<http://localhost:8080>Hello>



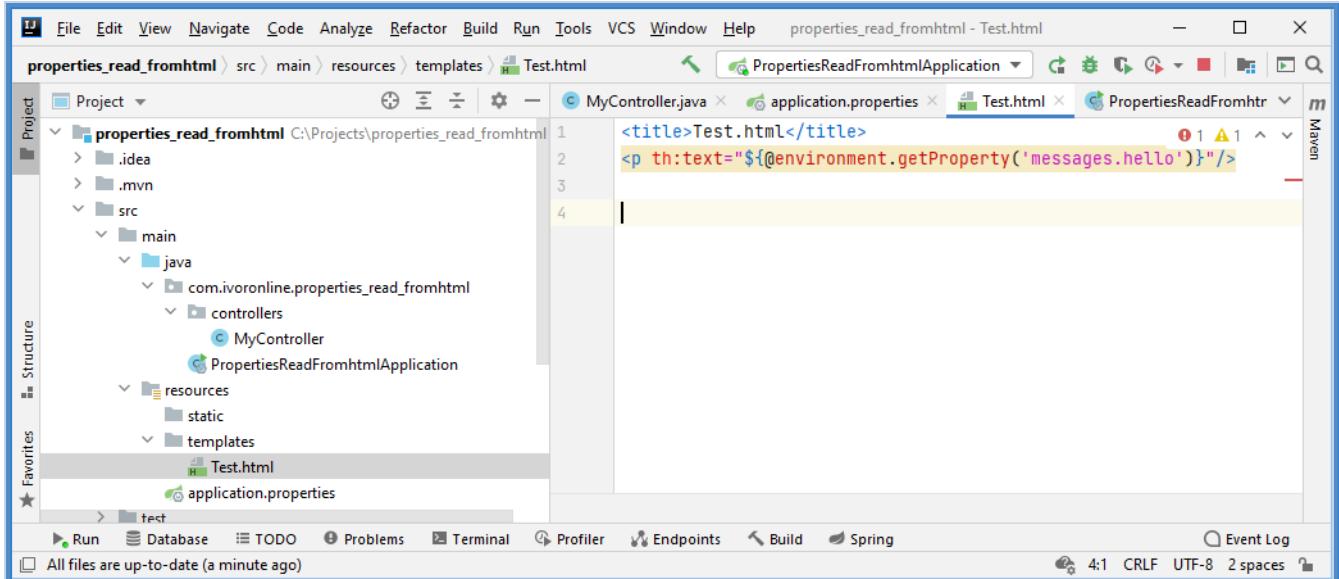
### Page Source

```
<title>Test.html</title>
<p> Hello from Controller </p>
<form action="http://localhost:8080/AddAuthor" method="get"/>
```

### Console

```
Hello from Controller
```

### Application Structure



### pom.xml

```
<dependencies>

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

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

</dependencies>
```

## 2.14.3 Built-in Properties

### Info

- This tutorial shows different built-in Properties that can be used inside `application.properties`.

#### Custom

```
messages.hello1 : Hello1 from Controller  
messages.hello2 = Hello2 from Controller
```

#### Tomcat

```
# PORT  
server.port = 8085
```

#### Profiles

```
# PROFILES  
spring.profiles.active = Profile1
```

#### Hibernate

```
# JPA / HIBERNATE  
spring.jpa.hibernate.ddl-auto = create-drop  
spring.jpa.show-sql = true
```

#### Databases

```
# H2 DATABASE  
spring.h2.console.enabled = true  
spring.datasource.url = jdbc:h2:mem:testdb  
  
# MYSQL DATABASE  
spring.datasource.url = jdbc:mysql://localhost:3306/world?serverTimezone=UTC  
spring.datasource.username = root  
spring.datasource.password = admin  
spring.jpa.database-platform = org.hibernate.dialect.MySQL5Dialect  
spring.datasource.driver-class-name = com.mysql.jdbc.Driver  
  
# POSTGRESQL DATABASE  
spring.datasource.url = jdbc:postgresql://localhost:5432/postgres  
spring.datasource.username = postgres  
spring.datasource.password = letmein  
spring.datasource.driver-class-name = org.postgresql.Driver  
  
# MONGODB DATABASE  
spring.data.mongodb.url = mongodb://localhost:27017  
spring.data.mongodb.uri = mongodb://localhost:27017/mydatabase  
spring.data.mongodb.uri = mongodb://myuser:mypassword@localhost:27017/mydatabase
```

# 2.15 DB Queries

## Info

[R]

- Following tutorials show how to use **@Query** Annotation to execute DB Queries.  
You simply add **@Query** to a Method and when Method is called it will execute specified SQL and return its result.
- **SQL** can be provided as
  - Native Query `SELECT * FROM PERSON WHERE NAME='John' AND AGE=20` (nativeQuery = true)
  - JPQL `SELECT john FROM Person john WHERE john.name = 'John' AND john.age = 20` (default)
- **Method** can then be called like this
  - `Person person = personRepository.getPersonByNameAgeIndexed(name, age);` (return single Record)
  - `List<Person> persons = personRepository.getPersonsByName(name);` (returns List of Records)
- If SQL has **Parameters**, Method will also have Input Parameters
  - Indexed Parameters `(?1, ?2) => (String name, int age)`
  - Named Parameters `(:parname, :parage) => (@Param("parname") String name, @Param("parage") int age)`

### Examples of Native Query Methods

```
//=====
// SELECT
//=====

//NO PARAMETERS
@Query(nativeQuery = true, value = "SELECT * FROM PERSON WHERE NAME = 'John' AND AGE = 20")
Person getJohn();

//INDEXED PARAMETERS
@Query(nativeQuery = true, value = "SELECT * FROM PERSON WHERE NAME = ?1 AND AGE = ?2")
Person selectPersonByNameAgeIndexed(String name, Integer age);

//NAMED PARAMETERS
@Query(nativeQuery = true, value = "SELECT * FROM PERSON WHERE NAME = :name AND AGE = :parameterAge")
Person selectPersonByNameAgeNamed(String name, @Param("parameterAge") Integer age);

//RETURN LIST
@Query(nativeQuery = true, value = "SELECT * FROM PERSON WHERE NAME = :name")
List<Person> selectPersonsByName(String name);

//=====
// UPDATE
//=====

@Modifying
@Query(nativeQuery = true, value = "UPDATE PERSON SET AGE = :newAge WHERE NAME = :name")
Integer updatePersonsByName(String name, Integer newAge);

//=====
// DELETE
//=====

@Modifying
@Query(nativeQuery = true, value = "DELETE FROM PERSON WHERE NAME = :name")
Integer deletePersonsByName(String name);

//=====
// INSERT
//=====

@Modifying
@Query(nativeQuery = true, value = "INSERT INTO PERSON (name, age) VALUES (:name, :age)")
Integer insertPerson(String name, Integer age);
```

### Examples of JPQL Methods

```
//=====
// SELECT
//=====

//NO PARAMETERS
@Query(value = "SELECT john FROM Person john WHERE john.name = 'John' AND john.age = 20")
Person selectJohn();

//INDEXED PARAMETERS
@Query(value = "SELECT person FROM Person person WHERE person.name = ?1 AND person.age = ?2")
Person selectPersonByNameAgeIndexed(String name, Integer age);

//NAMED PARAMETERS
@Query(value = "SELECT person FROM Person person WHERE person.name = :name AND person.age = :parameterAge")
Person selectPersonByNameAgeNamed(String name, @Param("parameterAge") Integer age);

//RETURN LIST
@Query(value = "SELECT person FROM Person person WHERE person.name = :name")
List<Person> selectPersonsByName(String name);

//RETURN SORTED LIST
@Query(value = "SELECT person FROM Person person WHERE person.name = :name")
List<Person> selectPersonsByNameSorted(String name, Sort sort);

//=====
// UPDATE
//=====

@Modifying
@Query(value = "UPDATE Person person SET person.age = :newAge WHERE person.name = :name")
Integer updatePersonsByName(String name, Integer newAge);

//=====
// DELETE
//=====

@Modifying
@Query(value = "DELETE FROM Person person WHERE person.name = :name")
Integer deletePersonsByName(String name);

//=====
// INSERT IS NOT SUPPORTED BY JPA
//=====
```

### Examples of Method Calls

```
Person      person      = personRepository.selectPerson (name);           //Returns single Record
List<Person> persons     = personRepository.selectPersons(name);          //Returns multiple Records
Integer    recordsUpdated = personRepository.updatePersons(name, newAge); //Returns affected number
```



## 2.15.1 Native Query

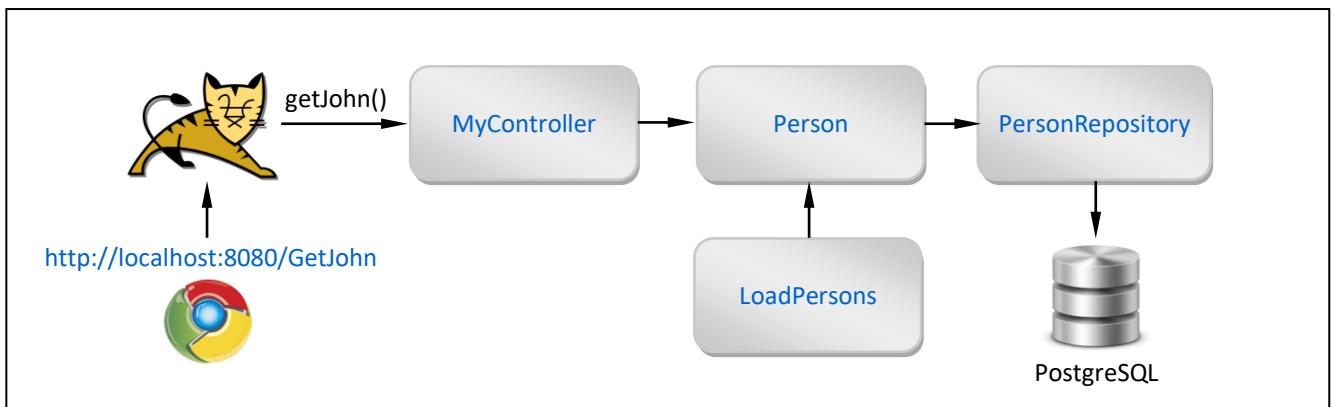
### Info

[G] [R]

- This tutorial shows how to use `@Query` Annotation to execute Native SQL.  
You simply add `@Query` to a Method and when Method is called it will execute specified SQL and return its result.  
You also need to add `nativeQuery = true` Property because by default Spring expects JPQL.
- If SQL has **Parameters**, Method will also have Input Parameters
  - Indexed Parameters (`?1, ?2`) => (`String name, int age`)
  - Named Parameters (`:parname, :parage`) => (`@Param("parname") String name, @Param("parage") int age`)
- To demonstrate this functionality we will load some Persons into DB and the use different SQLs to get them from the DB.

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables <code>@RequestMapping</code> (includes Tomcat Server)
SQL	Spring Data JPA	Enables <code>@Entity</code> and <code>@Id</code>
SQL	PostgreSQL Database	Enables Hibernate to work with PostgreSQL DB
Developer Tools	Lombok	Enables <code>@Data</code> which generate helper methods (setters, getters, ...)

## Procedure

- Create Project: `springboot_db_query_native` (add Spring Boot Starters from the table)
- Edit File: `application.properties` (enter DB connection parameters)
- Create Package: `entities` (inside main package)
  - Create Java Class: `Person.java` (inside package entities)
- Create Package: `repositories` (inside main package)
  - Create Java Interface: `PersonRepository.java` (inside package repositories)
- Create Package: `runners` (inside main package)
  - Create Java Interface: `LoadPersons.java` (inside package runners)
- Create Package: `controllers` (inside main package)
  - Create Java Class: `MyController.java` (inside package controllers)

### *application.properties*

```
# POSTGRES DATABASE
spring.datasource.url      = jdbc:postgresql://localhost:5432/postgres
spring.datasource.username   = postgres
spring.datasource.password   = letmein
spring.datasource.driver-class-name = org.postgresql.Driver

# JPA / HIBERNATE
spring.jpa.hibernate.ddl-auto = create-drop
```

### *Person.java*

```
package com.ivoronline.springboot_db_query_native.entities;

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

@Entity
public class Person {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    public Integer id;
    public String name;
    public Integer age;

    //CONSTRUCTORS
    public Person() { }                                //Forced by @Entity
    public Person(String name, Integer age) {           //To simplify PersonLoader
        this.name = name;
        this.age = age;
    }
}
```

### PersonRepository.java

```
package com.ivorononline.springboot_db_query_native.repositories;

import com.ivorononline.springboot_db_query_native.entities.Person;
import org.springframework.data.jpa.repository.Modifying;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.CrudRepository;
import org.springframework.data.repository.query.Param;
import java.util.List;

public interface PersonRepository extends CrudRepository<Person, Integer> {

    //=====
    // SELECT
    //=====

    //NO PARAMETERS
    @Query(nativeQuery = true, value = "SELECT * FROM PERSON WHERE NAME = 'John' AND AGE = 20")
    Person getJohn();

    //INDEXED PARAMETERS
    @Query(nativeQuery = true, value = "SELECT * FROM PERSON WHERE NAME = ?1 AND AGE = ?2")
    Person selectPersonByNameAgeIndexed(String name, Integer age);

    //NAMED PARAMETERS
    @Query(nativeQuery = true, value = "SELECT * FROM PERSON WHERE NAME = :name AND AGE = :parameterAge")
    Person selectPersonByNameAgeNamed(String name, @Param("parameterAge") Integer age);

    //RETURN LIST
    @Query(nativeQuery = true, value = "SELECT * FROM PERSON WHERE NAME = :name")
    List<Person> selectPersonsByName(String name);

    //=====
    // UPDATE
    //=====

    @Modifying
    @Query(nativeQuery = true, value = "UPDATE PERSON SET AGE = :newAge WHERE NAME = :name")
    Integer updatePersonsByName(String name, Integer newAge);

    //=====
    // DELETE
    //=====

    @Modifying
    @Query(nativeQuery = true, value = "DELETE FROM PERSON WHERE NAME = :name")
    Integer deletePersonsByName(String name);

    //=====
    // INSERT
    //=====

    @Modifying
    @Query(nativeQuery = true, value = "INSERT INTO PERSON (name, age) VALUES (:name, :age)")
    Integer insertPerson(String name, Integer age);

}
```

### LoadPersons.java

```
package com.ivoronline.springboot_db_query_native.runners;

import com.ivoronline.springboot_db_query_native.entities.Person;
import com.ivoronline.springboot_db_query_native.repositories.PersonRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.core.annotation.Order;
import org.springframework.stereotype.Component;
import org.springframework.transaction.annotation.Transactional;

@Component
@Order(1)
public class LoadPersons implements CommandLineRunner {

    @Autowired PersonRepository personRepository;

    @Override
    @Transactional
    public void run(String... args) throws Exception {
        personRepository.save(new Person("John" , 20));
        personRepository.save(new Person("John" , 21));
        personRepository.save(new Person("Bill" , 30));
        personRepository.save(new Person("Nancy", 40));
        personRepository.save(new Person("Susan", 50));
    }
}
```

### MyController.java

```
package com.ivoronline.springboot_db_query_native.controllers;

import com.ivoronline.springboot_db_query_native.entities.Person;
import com.ivoronline.springboot_db_query_native.repositories.PersonRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RequestMapping;
import java.util.List;

@Controller
public class MyController {

    @Autowired PersonRepository personRepository;

    //=====
    // SELECT JOHN
    //=====

    @ResponseBody
    @RequestMapping("/SelectJohn")
    public Person selectJohn() {
        Person john = personRepository.getJohn();
        return john;
    }

    //=====
    // SELECT PERSON BY NAME AGE INDEXED
    //=====

    @ResponseBody
    @RequestMapping("/SelectPersonByNameAgeIndexed")
```

```

public Person selectPersonByNameAgeIndexed(@RequestParam String name, @RequestParam Integer age) {
    Person person = personRepository.selectPersonByNameAgeIndexed(name, age);
    return person;
}

//=====
// SELECT PERSON BY NAME AGE NAMED
//=====

@RequestBody
@RequestMapping("/SelectPersonByNameAgeNamed")
public Person selectPersonByNameAgeNamed(@RequestParam String name, @RequestParam Integer age) {
    Person person = personRepository.selectPersonByNameAgeNamed(name, age);
    return person;
}

//=====
// SELECT PERSONS BY NAME
//=====

@RequestBody
@RequestMapping("/SelectPersonsByName")
public List<Person> selectPersonsByName(@RequestParam String name) {
    List<Person> persons = personRepository.selectPersonsByName(name);
    return persons;
}

//=====
// UPDATE PERSON BY NAME
//=====

@RequestBody
@Transactional
@RequestMapping("/UpdatePersonsByName")
public String updatePersonsByName(@RequestParam String name, @RequestParam Integer newAge) {
    Integer recordsUpdated = personRepository.updatePersonsByName(name, newAge);
    return recordsUpdated + " Records updated";
}

//=====
// DELETE PERSON BY NAME
//=====

@RequestBody
@Transactional
@RequestMapping("/DeletePersonsByName")
public String deletePersonsByName(@RequestParam String name) {
    Integer recordsDeleted = personRepository.deletePersonsByName(name);
    return recordsDeleted + " Records deleted";
}

//=====
// INSERT IS NOT SUPPORTED BY JPA
//=====

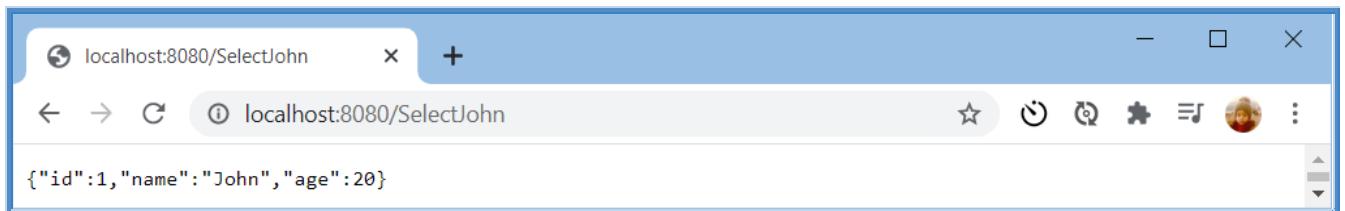
@RequestBody
@Transactional
@RequestMapping("/InsertPerson")
public String insertPerson(@RequestParam String name, @RequestParam Integer age) {
    Integer recordsDeleted = personRepository.insertPerson(name, age);
    return recordsDeleted + " Records inserted";
}

```

## Results

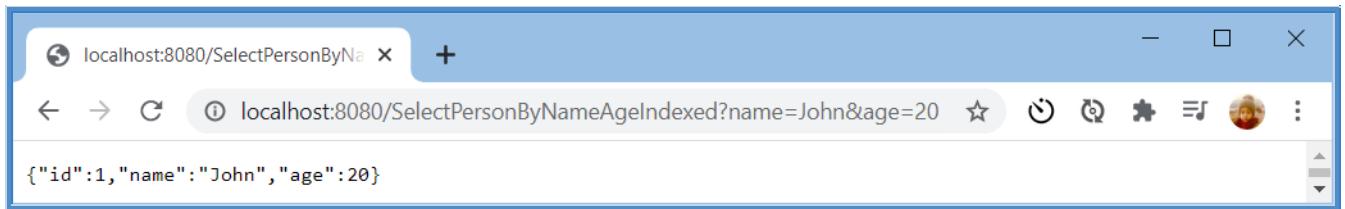
SELECT <http://localhost:8080>SelectJohn>

(No Parameters)



SELECT <http://localhost:8080>SelectPersonByNameAgeIndexed?name=John&age=20>

(Indexed Parameters)



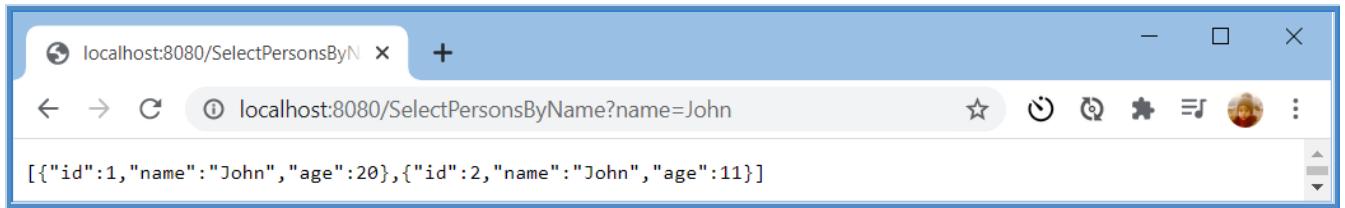
SELECT <http://localhost:8080>SelectPersonByNameAgeNamed?name=John&age=20>

(Named Parameters)



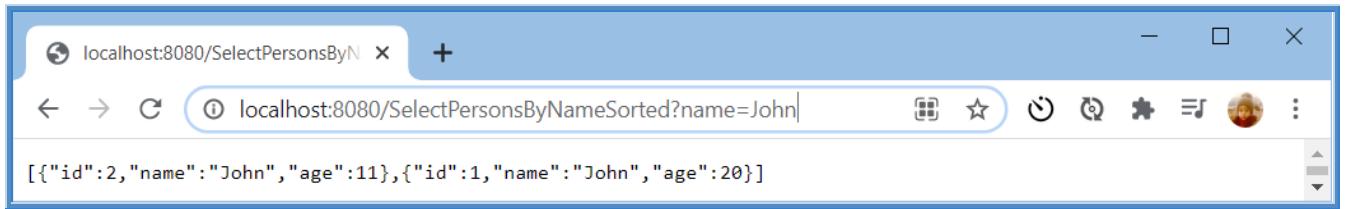
SELECT <http://localhost:8080>SelectPersonsByName?name=John>

(Return List)



SELECT <http://localhost:8080>SelectPersonsByNameSorted?name=John>

(Sort by Age)



UPDATE <http://localhost:8080/UpdatePersonsByName?name=John&newAge=100>



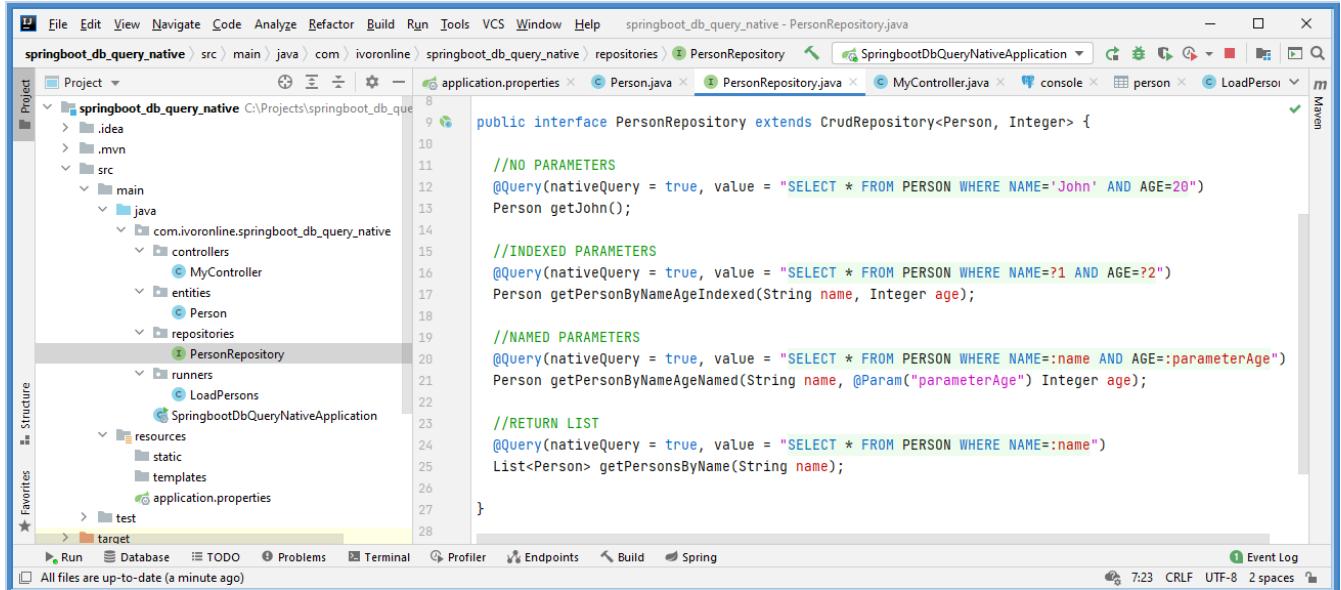
DELETE <http://localhost:8080>DeletePersonsByName?name=John>



INSERT <http://localhost:8080/InsertPerson?name=Peter&age=60>



## Application Structure



## pom.xml

```
<dependencies>

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

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

    <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <scope>runtime</scope>
    </dependency>

    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <optional>true</optional>
    </dependency>

</dependencies>
```

## 2.15.2 JPQL

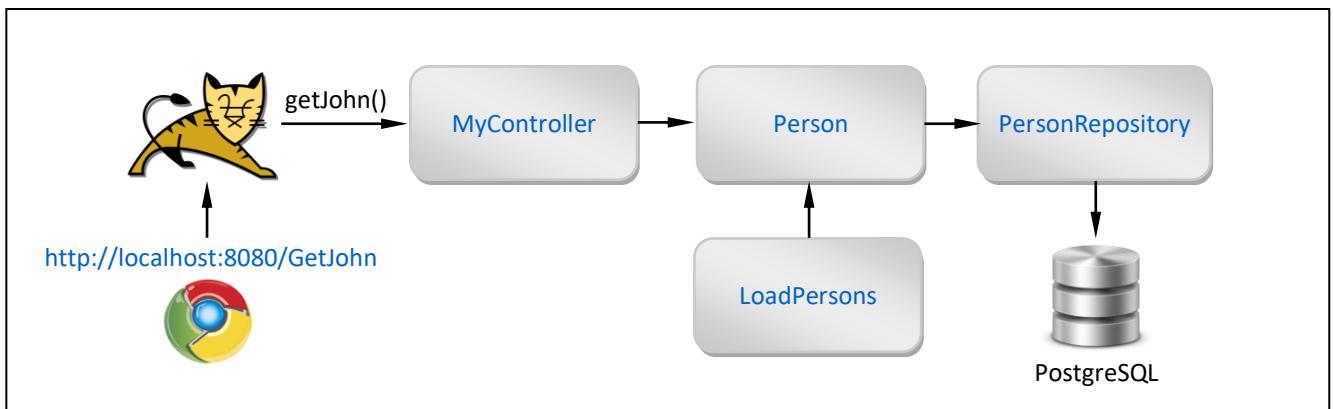
### Info

[G] [R]

- This tutorial shows how to use **@Query** Annotation to execute Java Persistence Query Language (JPQL). You simply add **@Query** to a Method and when Method is called it will execute specified SQL and return its result.
- If SQL has Parameters, Method will also have Input Parameters
  - Indexed Parameters (`?1, ?2`) => (`String name, int age`)
  - Named Parameters (`:parname, :parage`) => (`@Param("parname") String name, @Param("parage") int age`)
- To demonstrate this functionality we will load some Persons into DB and the use different SQLs to get them from the DB.

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables <code>@RequestMapping</code> (includes Tomcat Server)
SQL	Spring Data JPA	Enables <code>@Entity</code> and <code>@Id</code>
SQL	PostgreSQL Database	Enables Hibernate to work with PostgreSQL DB
Developer Tools	Lombok	Enables <code>@Data</code> which generate helper methods (setters, getters, ...)

## Procedure

- Create Project: `springboot_db_query_native` (add Spring Boot Starters from the table)
- Edit File: `application.properties` (enter DB connection parameters)
- Create Package: `entities` (inside main package)
  - Create Java Class: `Person.java` (inside package entities)
- Create Package: `repositories` (inside main package)
  - Create Java Interface: `PersonRepository.java` (inside package repositories)
- Create Package: `runners` (inside main package)
  - Create Java Interface: `LoadPersons.java` (inside package runners)
- Create Package: `controllers` (inside main package)
  - Create Java Class: `MyController.java` (inside package controllers)

### *application.properties*

```
# POSTGRES DATABASE
spring.datasource.url      = jdbc:postgresql://localhost:5432/postgres
spring.datasource.username   = postgres
spring.datasource.password   = letmein
spring.datasource.driver-class-name = org.postgresql.Driver

# JPA / HIBERNATE
spring.jpa.hibernate.ddl-auto = create-drop
```

### *Person.java*

```
package com.ivoronline.springboot_db_query_native.entities;

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

@Entity
public class Person {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    public Integer id;
    public String name;
    public Integer age;

    //CONSTRUCTORS
    public Person() { }                                //Forced by @Entity
    public Person(String name, Integer age) {           //To simplify PersonLoader
        this.name = name;
        this.age = age;
    }
}
```

### PersonRepository.java

```
package com.ivorononline.springboot_db_query_jpql.repositories;

import com.ivorononline.springboot_db_query_jpql.entities.Person;
import org.springframework.data.domain.Sort;
import org.springframework.data.jpa.repository.Modifying;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.CrudRepository;
import org.springframework.data.repository.query.Param;

import java.util.List;

public interface PersonRepository extends CrudRepository<Person, Integer> {

    //=====
    // SELECT
    //=====

    //NO PARAMETERS
    @Query(value = "SELECT john FROM Person john WHERE john.name = 'John' AND john.age = 20")
    Person selectJohn();

    //INDEXED PARAMETERS
    @Query(value = "SELECT person FROM Person person WHERE person.name = ?1 AND person.age = ?2")
    Person selectPersonByNameAgeIndexed(String name, Integer age);

    //NAMED PARAMETERS
    @Query(value = "SELECT person FROM Person person WHERE person.name = :name AND person.age = :parameterAge")
    Person selectPersonByNameAgeNamed(String name, @Param("parameterAge") Integer age);

    //RETURN LIST
    @Query(value = "SELECT person FROM Person person WHERE person.name = :name")
    List<Person> selectPersonsByName(String name);

    //RETURN SORTED LIST
    @Query(value = "SELECT person FROM Person person WHERE person.name = :name")
    List<Person> selectPersonsByNameSorted(String name, Sort sort);

    //=====
    // UPDATE
    //=====

    @Modifying
    @Query(value = "UPDATE Person person SET person.age = :newAge WHERE person.name = :name")
    Integer updatePersonsByName(String name, Integer newAge);

    //=====
    // DELETE
    //=====

    @Modifying
    @Query(value = "DELETE FROM Person person WHERE person.name = :name")
    Integer deletePersonsByName(String name);

    //=====
    // INSERT IS NOT SUPPORTED BY JPA
    //=====

}
```

### *LoadPersons.java*

```
package com.ivoronline.springboot_db_query_jpql.runners;

import com.ivoronline.springboot_db_query_jpql.entities.Person;
import com.ivoronline.springboot_db_query_jpql.repositories.PersonRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.core.annotation.Order;
import org.springframework.stereotype.Component;
import org.springframework.transaction.annotation.Transactional;

@Component
@Order(1)
public class LoadPersons implements CommandLineRunner {

    @Autowired PersonRepository personRepository;

    @Override
    @Transactional
    public void run(String... args) throws Exception {
        personRepository.save(new Person("John" , 20));
        personRepository.save(new Person("John" , 11));
        personRepository.save(new Person("Bill" , 30));
        personRepository.save(new Person("Nancy", 40));
        personRepository.save(new Person("Susan", 50));
    }
}
```

### *MyController.java*

```
package com.ivoronline.springboot_db_query_jpql.controllers;

import com.ivoronline.springboot_db_query_jpql.entities.Person;
import com.ivoronline.springboot_db_query_jpql.repositories.PersonRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Sort;
import org.springframework.stereotype.Controller;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseBody;

import java.util.List;

@Controller
public class MyController {

    @Autowired PersonRepository personRepository;

    //=====
    // SELECT JOHN
    //=====

    @ResponseBody
    @RequestMapping("/SelectJohn")
    public Person selectJohn() {
        Person john = personRepository.selectJohn();
        return john;
    }

    //=====
    // SELECT PERSON BY NAME AGE INDEXED
    //=====
```

```

@RequestBody
@RequestMapping("/SelectPersonByNameAgeIndexed")
public Person selectPersonByNameAgeIndexed(@RequestParam String name, @RequestParam Integer age) {
    Person person = personRepository.selectPersonByNameAgeIndexed(name, age);
    return person;
}

//=====
// SELECT PERSON BY NAME AGE NAMED
//=====

@ResponseBody
@RequestMapping("/SelectPersonByNameAgeNamed")
public Person selectPersonByNameAgeNamed(@RequestParam String name, @RequestParam Integer age) {
    Person person = personRepository.selectPersonByNameAgeNamed(name, age);
    return person;
}

//=====
// SELECT PERSONS BY NAME
//=====

@ResponseBody
@RequestMapping("/SelectPersonsByName")
public List<Person> selectPersonsByName(@RequestParam String name) {
    List<Person> persons = personRepository.selectPersonsByName(name);
    return persons;
}

//=====
// SELECT PERSONS BY NAME SORTED
//=====

@ResponseBody
@RequestMapping("/SelectPersonsByNameSorted")
public List<Person> selectPersonsByNameSorted(@RequestParam String name) {
    List<Person> persons = personRepository.selectPersonsByNameSorted(name, Sort.by("age"));
    return persons;
}

//=====
// UPDATE PERSON BY NAME
//=====

@ResponseBody
@Transactional
@RequestMapping("/UpdatePersonsByName")
public String updatePersonsByName(@RequestParam String name, @RequestParam Integer newAge) {
    Integer recordsUpdated = personRepository.updatePersonsByName(name, newAge);
    return recordsUpdated + " Records were updated";
}

//=====
// DELETE PERSON BY NAME
//=====

@ResponseBody
@Transactional
@RequestMapping("/DeletePersonsByName")
public String deletePersonsByName(@RequestParam String name) {
    Integer recordsDeleted = personRepository.deletePersonsByName(name);
    return recordsDeleted + " Records were deleted";
}

//=====
// INSERT IS NOT SUPPORTED BY JPA
//=====

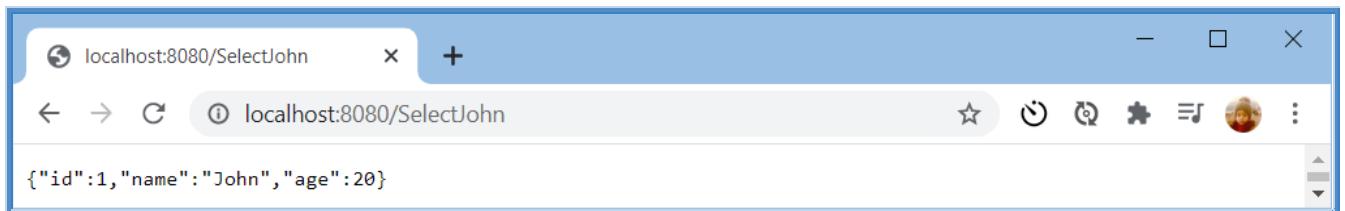
}

```

## Results

SELECT <http://localhost:8080>SelectJohn>

(No Parameters)



SELECT <http://localhost:8080>SelectPersonByNameAgeIndexed?name=John&age=20>

(Indexed Parameters)



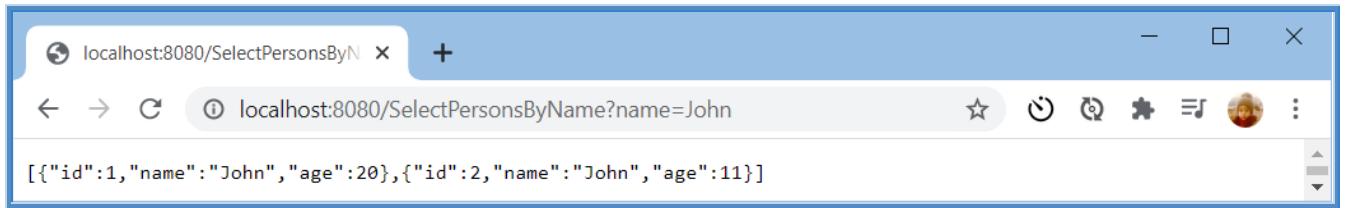
SELECT <http://localhost:8080>SelectPersonByNameAgeNamed?name=John&age=20>

(Named Parameters)



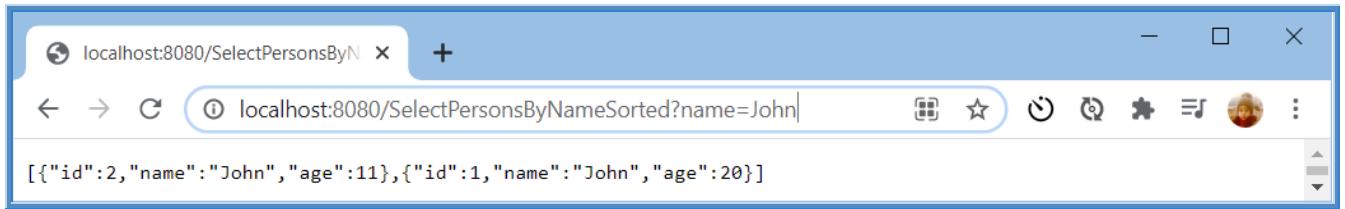
SELECT <http://localhost:8080>SelectPersonsByName?name=John>

(Return List)



SELECT <http://localhost:8080>SelectPersonsByNameSorted?name=John>

(Sort by Age)



UPDATE <http://localhost:8080/UpdatePersonsByName?name=John&newAge=100>



DELETE <http://localhost:8080>DeletePersonsByName?name=John>



## Application Structure

The screenshot shows the IntelliJ IDEA interface with the following details:

- Project Tree:** Shows the project structure under "springboot\_db\_query\_jpql". It includes ".idea", ".mvn", "src" (containing "main" and "test"), "resources", "static", "templates", and "application.properties".
- Code Editor:** Displays the file "PersonRepository.java" with JPQL queries. The code includes methods like "getJohn()", "getPersonByNameAgeIndexed()", "getPersonByNameAgeNamed()", "getPersonsByName()", and "getPersonsByNameSorted()".
- Toolbar:** Includes standard IntelliJ tools like Find, Database, TODO, Problems, Terminal, Profiler, Endpoints, Build, and Spring.
- Status Bar:** Shows "23:1 (83 chars, 1 line break) CRLF UTF-8 2 spaces".

## pom.xml

```
<dependencies>

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

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

    <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <scope>runtime</scope>
    </dependency>

    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <optional>true</optional>
    </dependency>

</dependencies>
```

# 3 Demo Applications

## Info

---

- Following tutorials contain demo applications that show how to combine functionalities covered in previous tutorials.
- Each application is broken into multiple tutorials/steps showing how to add additional functionalities.

## Balance between simplicity and coding practices

---

- In the previous tutorials main priority was simplicity, sometimes at the expense of the best coding practices.  
For instance in previous tutorials Entities were created by using **public Properties** to keep them as simple as possible by avoiding having setters and getters.
- Tutorials that follow will try to strike a balance between remaining simple while following the best coding practices.  
In case of Entities this means that we will have **private Properties** accessed through Lombok's setters and getters.  
This Results in a more complex code and more stuff to learn (like Lombok) but it is more aligned with real life practices.  
But it will be far easier to make this transition after you have already learned the basics in previous tutorials.

# 3.1 Simple Applications

## Info

---

- Following tutorials shows how to create simple applications that combine only few of previously functionalities described.
- You can use them as basis for creating more complex applications.

## 3.1.1 Request - Store Entity from JSON

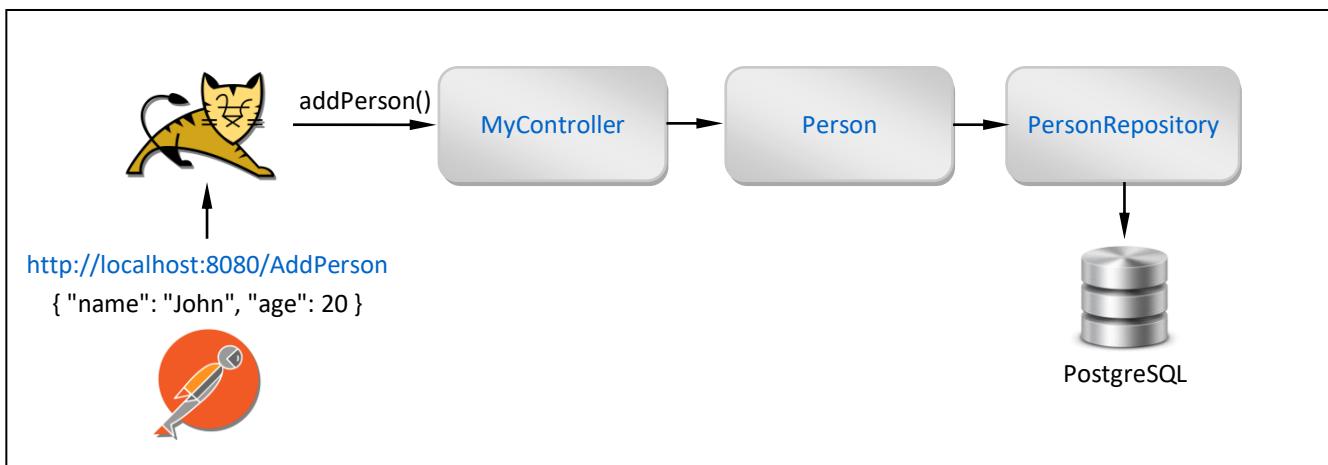
### Info

[G]

- This tutorial shows how to create App that stores Person Entity from **JSON Request** by combining following tutorials
  - [PostgreSQL](#) (Main functionality for storing hard coded Entity into DB)
  - [Annotation - \(Lombok\) @Data](#) (Edit Entity to use private Properties and Lombok Setters & Getters)
  - [Annotation - \(Spring\) @RequestBody - JSON](#) (Replace Endpoint's hard coded data with JSON from HTTP Request)

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables <code>@RequestMapping</code> (includes Tomcat Server)
SQL	Spring Data JPA	Enables <code>@Entity</code> and <code>@Id</code>
SQL	PostgreSQL Database	Enables Hibernate to work with PostgreSQL DB
Developer Tools	Lombok	Enables <code>@Data</code> which generate helper methods (setters, getters, ...)

### Procedure

- Create Project:** [springboot\\_demo\\_store\\_entity\\_from\\_json](#) (add Spring Boot Starters from the table)
- Edit File:** [application.properties](#) (enter DB connection parameters)
- Create Package:** entities (inside main package)
  - Create Java Class:** [Person.java](#) (inside package entities)
- Create Package:** repositories (inside main package)
  - Create Java Interface:** [PersonRepository.java](#) (inside package repositories)
- Create Package:** controllers (inside main package)
  - Create Java Class:** [MyController.java](#) (inside package controllers)

### application.properties

```
# POSTGRES DATABASE
spring.datasource.url      = jdbc:postgresql://localhost:5432/postgres
spring.datasource.username   = postgres
spring.datasource.password   = letmein
spring.datasource.driver-class-name = org.postgresql.Driver

# JPA / HIBERNATE
spring.jpa.hibernate.ddl-auto = create-drop
```

### Person.java

```
package com.example.springboot_demo_store_entity_from_json.entities;

import lombok.Data;

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

@Data
@Entity
public class Person {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    private String name;
    private Integer age;

}
```

### PersonRepository.java

```
package com.example.springboot_demo_store_entity_from_json.repositories;

import com.example.springboot_demo_store_entity_from_json.entities.Person;
import org.springframework.data.repository.CrudRepository;

public interface PersonRepository extends CrudRepository<Person, Integer> { }
```

### MyController.java

```
package com.example.springboot_demo_store_entity_from_json.controllers;

import com.example.springboot_demo_store_entity_from_json.entities.Person;
import com.example.springboot_demo_store_entity_from_json.repositories.PersonRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class MyController {

    @Autowired
    PersonRepository personRepository;

    @ResponseBody
    @RequestMapping("/AddPerson")
    public String addPerson(@RequestBody Person person) {

        //STORE PERSON ENTITY
        personRepository.save(person);

        //RETURN SOMETHING TO BROWSER
        return "Person added to DB";

    }

}
```

## Results

- Start Postman

POST

```
http://localhost:8080/AddPerson
```

Headers

(add Key-Value)

```
Content-Type: application/json
```

Body

(option: raw)

```
{
  "name" : "John",
  "age"   : 20
}
```

POST <http://localhost:8080/AddPerson>

The screenshot shows the Postman application window. At the top, there's a navigation bar with File, Edit, View, Help, Home, Workspaces, Reports, Explore, and a search bar. Below the navigation is a toolbar with various icons. The main area shows a collection named 'MyCollection / MyRequest' with a single POST request to 'http://localhost:8080/AddPerson'. The request details tab shows the method as POST, the URL, and the body content type as JSON. The body panel contains the following JSON:

```
1 {
2   "name" : "John",
3   "age"   : 20
4 }
```

The response panel shows a status of 200 OK, time 80 ms, size 182 B, and a message 'Person added to DB'. The bottom of the window has tabs for Find and Replace, Console, Bootcamp, Runner, and Trash.

HTTP Response Body

```
Person added to DB
```

DB Tools

	id	age	name
1	1	20	John

## Application Structure

The screenshot shows the IntelliJ IDEA interface. The left sidebar displays the project structure under 'Project'. The 'src' folder contains 'main' and 'java' subfolders. 'main' has 'controllers' and 'entities'. 'entities' contains 'Person'. 'java' contains 'com.example.springboot\_demo\_store\_entity\_from\_json.controllers' which has 'MyController'. 'repositories' contains 'PersonRepository'. 'resources' contains 'static' and 'templates'. 'application.properties' is also present. The right side shows the code editor with 'MyController.java' open. The code defines a controller named 'MyController' with a method 'addPerson' that takes a 'Person' object from the request body, saves it to the database, and returns a success message. The code editor includes syntax highlighting and various annotations like @Controller, @Autowired, @ResponseBody, and @RequestMapping.

```
11  @Controller
12  public class MyController {
13
14      @Autowired
15      PersonRepository personRepository;
16
17      @RequestMapping("/AddPerson")
18      public String addPerson(@RequestBody Person person) {
19
20          //STORE PERSON ENTITY
21          personRepository.save(person);
22
23          //RETURN SOMETHING TO BROWSER
24          return "Person added to DB";
25
26      }
27
28  }
```

## pom.xml

```
<dependencies>

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

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

    <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <scope>runtime</scope>
    </dependency>

    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <optional>true</optional>
    </dependency>

</dependencies>
```

## 3.1.2 Request - Store Entities from JSON Array

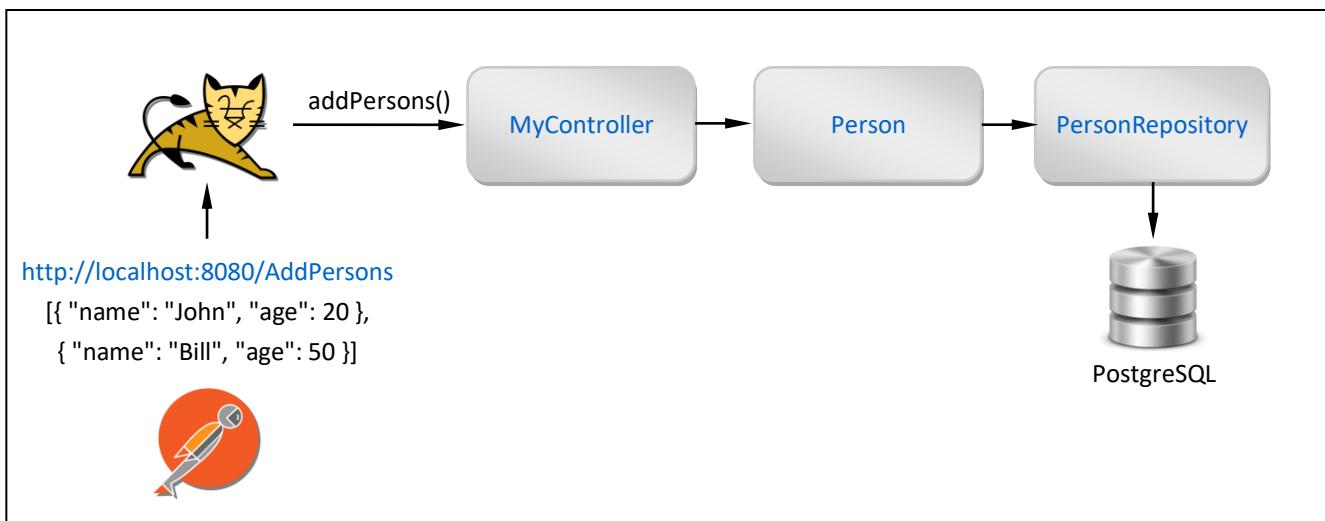
### Info

[G]

- This tutorial shows how to create App that stores Persons from **JSON Array Request** by combining following tutorials
  - PostgreSQL (Main functionality for storing hard coded Entity into DB)
  - Annotation - (Lombok) @Data (Edit Entity to use private Properties & Lombok Setters & Getters)
  - Annotation - (Spring) @RequestBody - JSON Array (Replace Endpoint hard coded data with JSON from HTTP Request)

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables @RequestMapping (includes Tomcat Server)
SQL	Spring Data JPA	Enables @Entity and @Id
SQL	PostgreSQL Database	Enables Hibernate to work with PostgreSQL DB
Developer Tools	Lombok	Enables @Data which generate helper methods (setters, getters, ...)

### Procedure

- Create Project:** `springboot_demo_store_entity_from_json_array` (add Spring Boot Starters from the table)
- Edit File:** `application.properties` (enter DB connection parameters)
- Create Package:** `entities` (inside main package)
- **Create Java Class:** `Person.java` (inside package entities)
- Create Package:** `repositories` (inside main package)
- **Create Java Interface:** `PersonRepository.java` (inside package repositories)
- Create Package:** `controllers` (inside main package)
- **Create Java Class:** `MyController.java` (inside package controllers)

### application.properties

```
# POSTGRES DATABASE
spring.datasource.url      = jdbc:postgresql://localhost:5432/postgres
spring.datasource.username  = postgres
spring.datasource.password = letmein
spring.datasource.driver-class-name = org.postgresql.Driver

# JPA / HIBERNATE
spring.jpa.hibernate.ddl-auto = create-drop
```

### Person.java

```
package com.example.springboot_demo_store_entity_from_json.entities;

import lombok.Data;

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

@Data
@Entity
public class Person {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    private String name;
    private Integer age;

}
```

### PersonRepository.java

```
package com.example.springboot_demo_store_entity_from_json.repositories;

import com.example.springboot_demo_store_entity_from_json.entities.Person;
import org.springframework.data.repository.CrudRepository;

public interface PersonRepository extends CrudRepository<Person, Integer> { }
```

### MyController.java

```
package com.example.springboot_demo_store_entity_from_json_array.controllers;

import com.example.springboot_demo_store_entity_from_json_array.entities.Person;
import com.example.springboot_demo_store_entity_from_json_array.repositories.PersonRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RequestMapping;

import java.util.List;

@Controller
public class MyController {

    @Autowired
    PersonRepository personRepository;

    @ResponseBody
    @RequestMapping("/AddPersons")
    public String addPersons(@RequestBody List<Person> persons) {

        //STORE PERSON ENTITY
        personRepository.saveAll(persons);

        //RETURN SOMETHING TO BROWSER
        return "Persons added to DB";

    }
}
```

## Results

- Start Postman

POST

```
http://localhost:8080/AddPersons
```

Headers

(add Key-Value)

```
Content-Type: application/json
```

Body

(option: raw)

```
[  
  {  
    "name" : "John",  
    "age"  : "20"  
  },  
  
  {  
    "name" : "Bill",  
    "age"  : "50"  
  }  
]
```

POST <http://localhost:8080/AddPersons>

The screenshot shows the Postman application window. The top navigation bar includes File, Edit, View, Help, Workspaces, Reports, Explore, and a search bar. Below the header is a toolbar with various icons. The main workspace shows a collection named 'MyCollection / MyRequest'. A POST request is selected with the URL 'http://localhost:8080/AddPersons'. The 'Body' tab is active, showing the JSON payload: 

```
[  
  {  
    "name" : "John",  
    "age"  : "20"  
  },  
  
  {  
    "name" : "Bill",  
    "age"  : "50"  
  }  
]
```

. The 'Pretty' option is selected under the Body tab. At the bottom, the status bar shows 'Status: 200 OK Time: 313 ms Size: 183 B' and a 'Save Response' button.

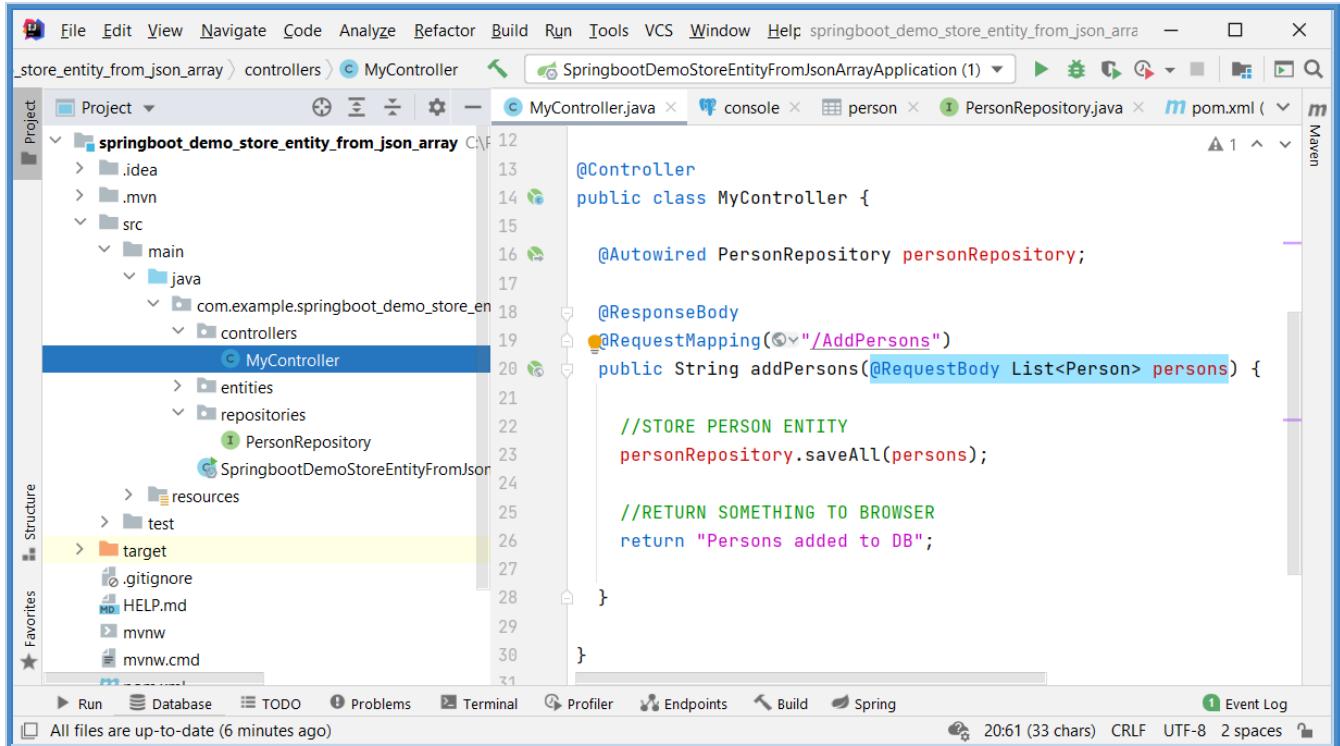
HTTP Response Body

```
Person added to DB
```

## DB Tools

	id	age	name
1	1	20	John
2	2	50	Bill

## Application Structure



## pom.xml

```
<dependencies>

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

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

    <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <scope>runtime</scope>
    </dependency>

    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <optional>true</optional>
    </dependency>

</dependencies>
```

### 3.1.3 Response - Store Entities from JSON Array

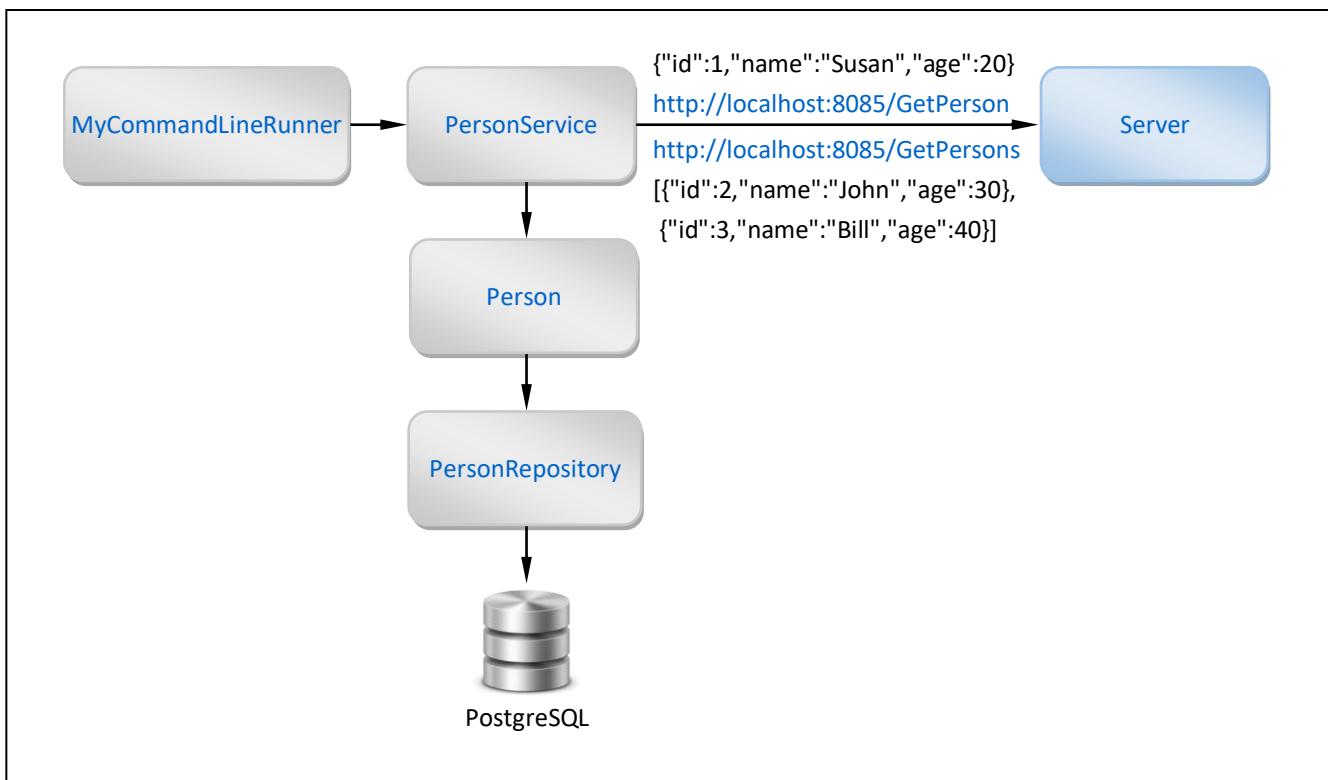
#### Info

[G]

- This tutorial shows how to create App that stores Persons from **JSON Array Response** by combining following tutorials
  - [PostgreSQL](#) (Main functionality for storing hard coded Entity into DB)
  - [Annotation - \(Lombok\) @Data](#) (Edit Entity to use private Properties & Lombok Setters & Getters)
  - [Client](#) (CommandLineRunner uses PersonService to call Server and store returned JSON)
  - [Server](#) (Create Server that returns JSON Array of Persons)

#### Application Schema

[Results]



#### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables @RequestMapping (includes Tomcat Server)
SQL	Spring Data JPA	Enables @Entity and @Id
SQL	PostgreSQL Database	Enables Hibernate to work with PostgreSQL DB
Developer Tools	Lombok	Enables @Data which generate helper methods (setters, getters, ...)

#### Procedure

- Create Project:** `springboot_demo_store_entity_from_json_array_response` (add Spring Boot Starters )
- Edit File:** `pom.xml` (add webflux dependency)
- Edit File:** `application.properties` (enter DB parameters)
- Create Package:** `entities` (inside main package)
- **Create Java Class:** `Person.java` (inside package entities)
- Create Package:** `repositories` (inside main package)
- **Create Java Interface:** `PersonRepository.java` (inside package repositories)
- Create Package:** `services` (inside main package)
- **Create Class:** `PersonService.java` (inside controllers package)
- Create Package:** `runners` (inside main package)
- **Edit Class:** `MyCommandLineRunner.java` (inside package runners)

*pom.xml*

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

*application.properties*

```
# POSTGRES DATABASE
spring.datasource.url      = jdbc:postgresql://localhost:5432/postgres
spring.datasource.username   = postgres
spring.datasource.password  = letmein
spring.datasource.driver-class-name = org.postgresql.Driver

# JPA / HIBERNATE
spring.jpa.hibernate.ddl-auto = create-drop
```

*Person.java*

```
package com.ivoronline.springboot_demo_store_entity_from_json_array_response.entities;

import lombok.Data;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Data
@Entity
public class Person {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    private String name;
    private Integer age;

}
```

*PersonRepository.java*

```
package com.ivoronline.springboot_demo_store_entity_from_json_array_response.repositories;

import com.ivoronline.springboot_demo_store_entity_from_json_array_response.entities.Person;
import org.springframework.data.repository.CrudRepository;

public interface PersonRepository extends CrudRepository<Person, Integer> { }
```

### PersonService.java

```
package com.ivorononline.springboot_demo_store_entity_from_json_array_response.services;

import com.ivorononline.springboot_demo_store_entity_from_json_array_response.entities.Person;
import com.ivorononline.springboot_demo_store_entity_from_json_array_response.repositories.PersonRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import org.springframework.web.reactive.function.client.WebClient;
import java.time.Duration;
import java.util.Arrays;

@Component
public class PersonService {

    @Autowired
    PersonRepository personRepository;

    //=====
    // GET PERSON
    //=====
    public void getPerson() {

        //GET PERSON FROM SERVER
        Person person = WebClient.create("http://localhost:8085")
            .get()
            .uri("/GetPerson")
            .retrieve()
            .bodyToMono(Person.class)
            .block(Duration.ofSeconds(3));

        //STORE PERSON
        personRepository.save(person);

        //DISPLAY PERSON
        System.out.println("getPerson ()");
        System.out.println(person.getId() + " " + person.getName() + " " + person.getAge());
    }

    //=====
    // GET PERSONS
    //=====
    public void getPersons() {

        //GET PERSON FROM SERVER
        Person[] persons = WebClient.create("http://localhost:8085")
            .get()
            .uri("/GetPersons")
            .retrieve()
            .bodyToMono(Person[].class)
            .block(Duration.ofSeconds(3));

        //STORE PERSONS
        personRepository.saveAll(Arrays.asList(persons));

        //DISPLAY PERSONS
        System.out.println("getPersons()");
        for (Person person : persons) {
            System.out.println(person.getId() + " " + person.getName() + " " + person.getAge());
        }
    }
}
```

### *MyCommandLineRunner.java*

```
package com.ivoronline.springboot_demo_store_entity_from_json_array_response.runners;

import com.ivoronline.springboot_demo_store_entity_from_json_array_response.services.PersonService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;
import org.springframework.transaction.annotation.Transactional;

@Component
public class MyCommandLineRunner implements CommandLineRunner {

    @Autowired private PersonService personService;

    @Override
    @Transactional
    public void run(String... args) throws Exception {
        personService.getPerson();
        personService.getPersons();
    }

}
```

## Results

### DB Tools

	id	age	name
1	1	30	Susan
2	2	40	John
3	3	50	Bill

### Application Structure

The screenshot shows the IntelliJ IDEA interface. The left sidebar displays the project structure under 'Project'. The 'src' directory contains 'main' and 'java'. The 'java' directory has packages like 'com.ivoronline.springboot\_demo\_store\_entity\_from\_json\_array\_response.entities' containing 'Person' and 'PersonRepository', and 'services' containing 'PersonService'. The code editor on the right shows the implementation of the 'PersonService' class. The 'getPerson()' method uses WebClient to get a person from a server, saves it to the repository, and then prints the person's details. The code editor also shows imports for WebClient, Person, UriComponentsBuilder, and Duration.

```
//=====
// GET PERSON
//=====
public void getPerson() {
    //GET PERSON FROM SERVER
    Person person = WebClient.create("http://localhost:8085")
        .get()
        .uri("/GetPerson")
        .retrieve()
        .bodyToMono(Person.class)
        .block(Duration.ofSeconds(3));

    //STORE PERSON
    personRepository.save(person);

    //DISPLAY PERSON
    System.out.println("getPerson ()");
    System.out.println(person.id + " " + person.name+ " " + person.age);
}
```

### pom.xml

```
<dependencies>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <optional>true</optional>
    </dependency>

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

</dependencies>
```

## 3.1.4 Use Filter to Log HTTP Parameters into DB

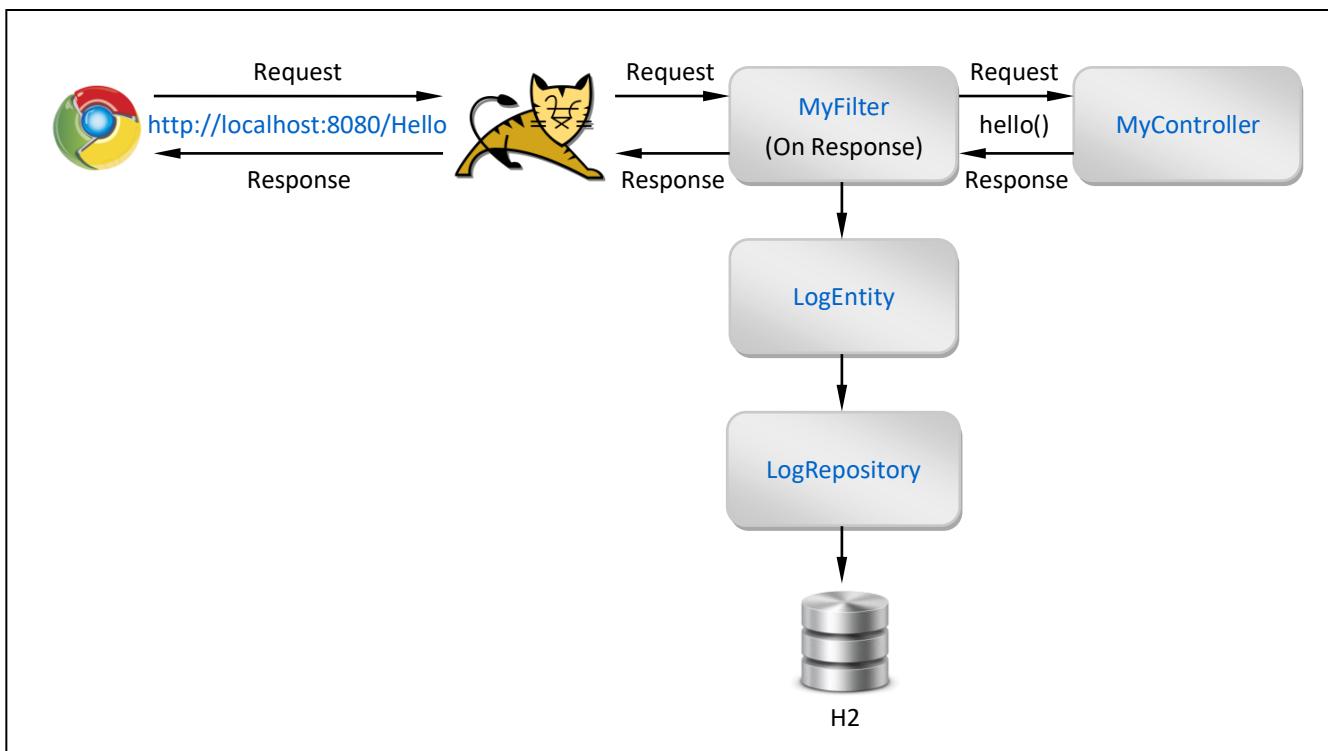
### Info

[G]

- This tutorial shows how to use Filter to log HTTP Request and Response Parameters into DB.  
Logging is done in the response part of the Filter to have access to Response Status (200, 404).
- Tutorial combines following tutorials
  - H2 (store Log Entity into H2 DB)
  - HTTP Request/Response - Get Parameters (use Filter to intercept HTTP Request/Response and get Parameters)

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables: @Controller, @RequestMapping, Tomcat Server.
SQL	Spring Data JPA	Enables: @Entity, @Id
SQL	H2 Database	Enables: in-memory H2 Database

### Procedure

- Create Project: `springboot_demo_logfilterdb` (add Spring Boot Starters from the table)
- Edit: `application.properties` (specify H2 DB name & enable H2 Web Console)
- Create Package: `entities` (inside main package)
  - Create Class: `LogEntity.java` (inside package entities)
- Create Package: `repositories` (inside main package)
  - Create Interface: `LogRepository.java` (inside package repositories)
- Create Package: `controllers` (inside main package)
  - Create Class: `MyController.java` (inside package controllers)
- Create Package: `filters` (inside main package)
  - Create Class: `MyFilter.java` (inside package controllers)

### *application.properties*

```
# H2 DATABASE
spring.h2.console.enabled = true
spring.datasource.url      = jdbc:h2:mem:testdb
```

### *LogEntity.java*

```
package com.ivoronline.springboot_demo_logfilterdb.entities;

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

@Entity
public class LogEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    public Integer id;
    public String protocol;
    public String serverName;
    public Integer serverPort;
    public String username;
    public Integer status;

}
```

### *LogRepository.java*

```
package com.ivoronline.springboot_demo_logfilterdb.repositories;

import com.ivoronline.springboot_demo_logfilterdb.entities.LogEntity;
import org.springframework.data.repository.CrudRepository;

public interface LogRepository extends CrudRepository<LogEntity, Integer> { }
```

### *MyController.java*

```
package com.ivoronline.springboot_demo_logfilterdb.controllers;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class MyController {

    @ResponseBody
    @RequestMapping("/Hello")
    public String hello() {
        System.out.println("Controller:");
        return "Hello from Controller";
    }

}
```

### MyFilter.java

```
package com.ivorononline.springboot_demo_logfilterdb.filters;

import java.io.IOException;
import javax.servlet.FilterChain;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.ivorononline.springboot_demo_logfilterdb.entities.LogEntity;
import com.ivorononline.springboot_demo_logfilterdb.repositories.LogRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import org.springframework.web.filter.OncePerRequestFilter;

@Component
public class MyFilter extends OncePerRequestFilter {

    @Autowired LogRepository logRepository;

    @Override
    public void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain chain)
        throws IOException, ServletException {

        //DIVIDES HTTP REQUEST AND RESPONSE CODE
        chain.doFilter(request, response);

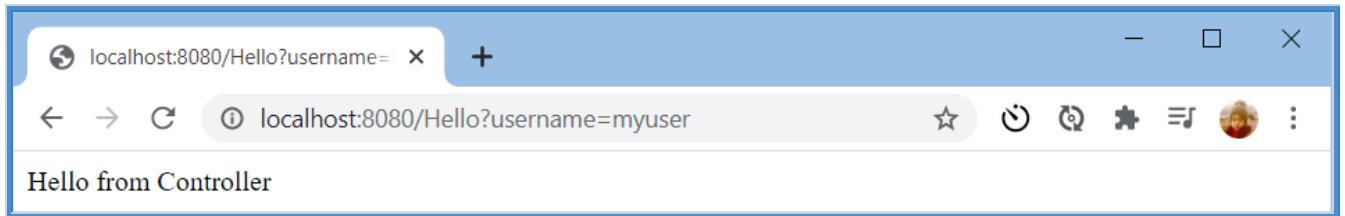
        //CREATE LOG
        LogEntity logEntity = new LogEntity();
        logEntity.protocol = request.getProtocol();
        logEntity.serverName = request.getServerName();
        logEntity.serverPort = request.getServerPort();
        logEntity.username = request.getParameter("username");
        logEntity.status = response.getStatus();

        //STORE LOG
        logRepository.save(logEntity);
    }
}
```

## Results

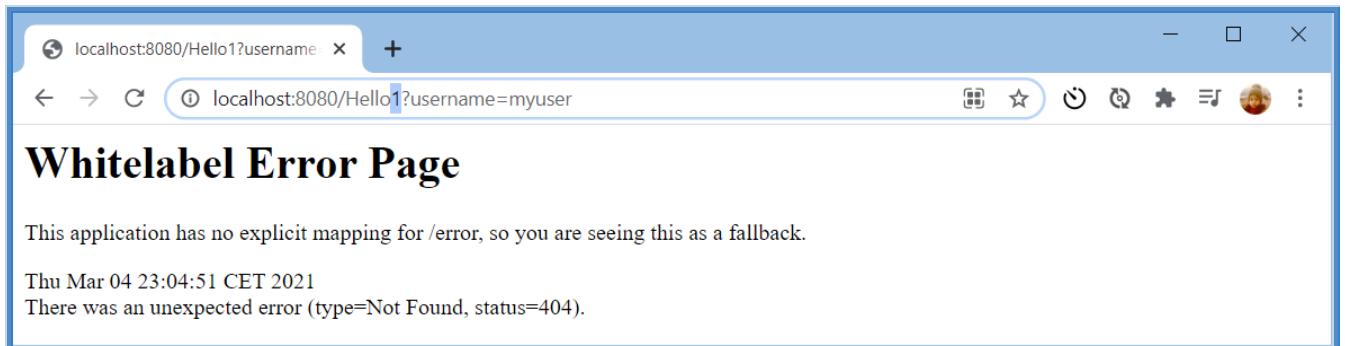
<http://localhost:8080>Hello?username=myuser>

(Response Status = 200)



<http://localhost:8080>Hello1?username=myuser>

(Response Status = 404)



<http://localhost:8080/h2-console>

(Open H2 Console)

A screenshot of the H2 Console interface. On the left, the database schema is shown with tables like LOG\_ENTITY and INFORMATION\_SCHEMA. In the center, a SQL query 'SELECT \* FROM LOG\_ENTITY WHERE USERNAME = 'myuser'' is entered, and its results are displayed in a table:

ID	PROTOCOL	SERVER_NAME	SERVER_PORT	STATUS	USERNAME
54	HTTP/1.1	localhost	8080	200	myuser
56	HTTP/1.1	localhost	8080	200	myuser
60	HTTP/1.1	localhost	8080	404	myuser

(3 rows, 3 ms)

## Application Structure

The screenshot shows the IntelliJ IDEA interface with the following details:

- Project Tree:** The project is named "springboot\_demo\_logfilterdb". It contains .idea, .mvn, and src folders. The src folder has main, test, and target sub-folders. Under main, there are java, resources, and templates. The java folder contains com.ivoronline.springboot controllers, entities, filters, and repositories. The filters folder contains MyFilter.java. The repositories folder contains LogRepository.java. The resources folder contains application.properties.
- Code Editor:** The code editor displays MyFilter.java. The code implements a OncePerRequestFilter and uses Autowiring to inject LogRepository. It overrides doFilterInternal to log request and response details. It creates a LogEntity object with protocol, serverName, serverPort, username, and status from the request and response. It then saves this entity to the LogRepository.
- Toolbars and Status Bar:** The bottom of the screen shows various toolbars like Run, Database, TODO, Problems, Terminal, Profiler, Endpoints, Build, and Spring. The status bar indicates the current time is 10:41, encoding is CRLF, file encoding is UTF-8, and two spaces are used for indentation.

## pom.xml

```
<dependencies>

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

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

    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
    </dependency>

</dependencies>
```

## 3.2 Authors & Books

### Info

[G]

- Following tutorials show how to use Spring Boot to create Web Application for entering Authors & Books.

<http://localhost:8080/AddAuthorForm>

ADD AUTHOR

Enter Author id  
Enter Author name  
Add Author

<http://localhost:8080/AddBookForm>

ADD BOOK

Enter Author id  
Enter Book id  
Enter Book title  
Add Book

<http://localhost:8080/GetBooksForm>

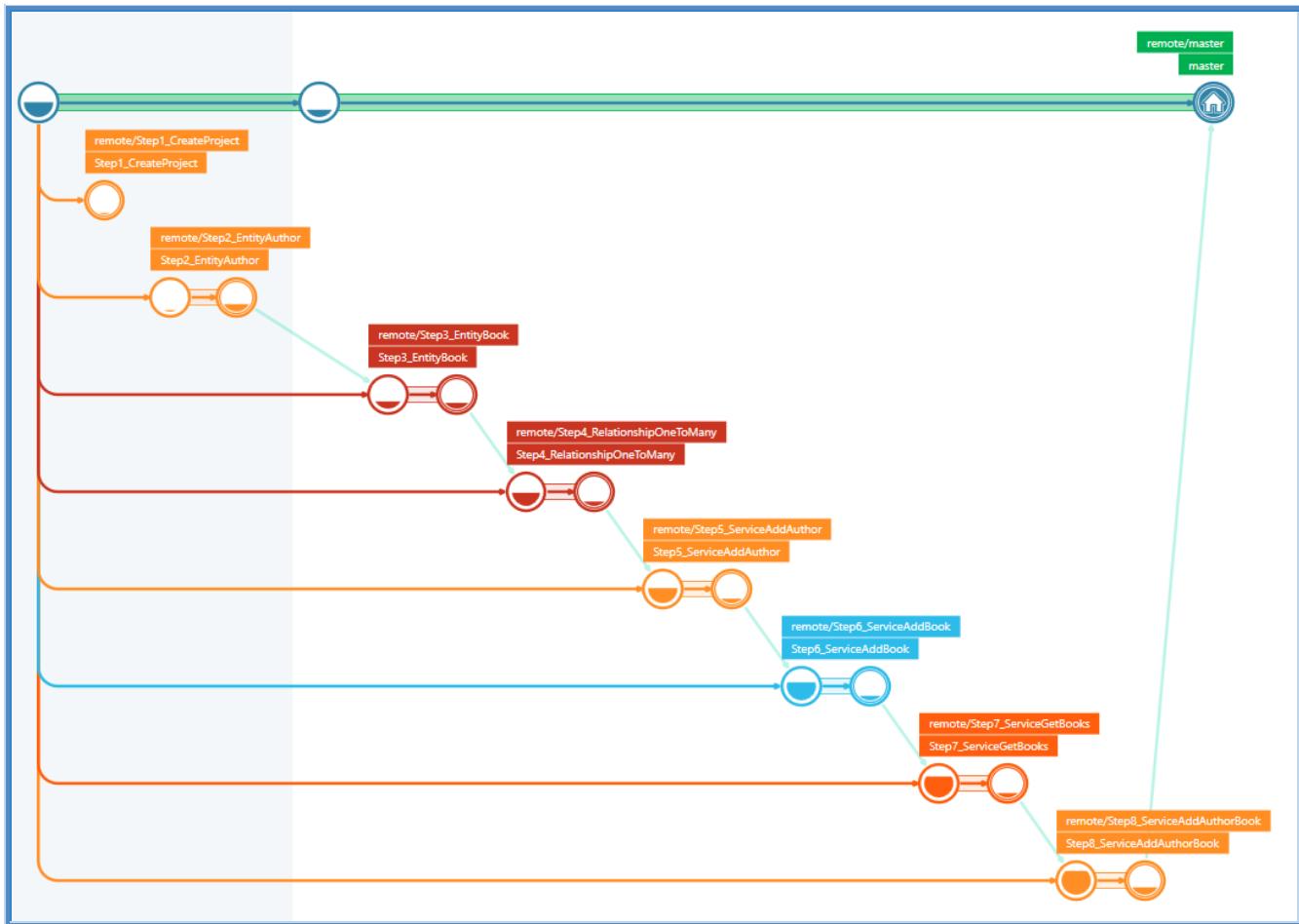
GET BOOKS

Enter Author id  
Get Books

<http://localhost:8080/getBooks?authorId=1>

Books by John

Book about dogs  
Book about birds  
Book about cats



## 3.2.1 Step 1: Create Project

### Info

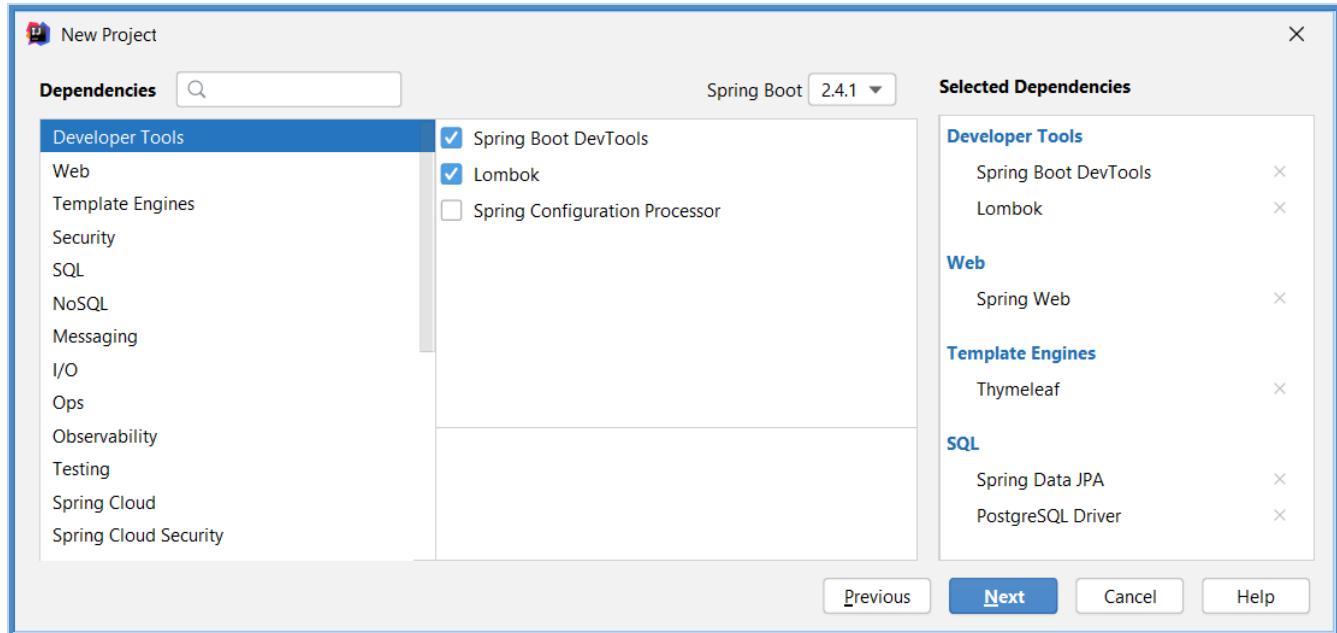
[G]

- First we will create basic Spring Boot Project.
- You should have PostgreSQL DB running and then add connection parameters to [application.properties](#).  
Or choose some other DB and then adjust connection parameters as described under [Repository](#) section for each DB.
- To verify setup we will edit [SpringbootDemoAuthorsbooksApplication.java](#) to print `Hello` to Console.

### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables <code>@RequestMapping</code> (includes Tomcat Server)
Developer Tools	Spring Boot DevTools	Enables application auto-reload
Developer Tools	Lombok	Enables <code>@Data</code> (generates helper methods: setters, getters, ...)
SQL	Spring Data JPA	Enables <code>@Entity</code> and <code>@Id</code>
SQL	PostgreSQL Database	Enables Hibernate to work with PostgreSQL DB

### Spring Boot Initializer



## Create Project

- [Create Project:](#) `springboot_demo_authorsbooks` (add Spring Boot Starters from the table)
- [Edit File:](#) `application.properties` (specify PostgreSQL DB connection)
- [Edit File:](#) `SpringbootDemoAuthorsbooksApplication.java` (print "Hello" to Console)

*application.properties*

```
# DATABASE
spring.datasource.url      = jdbc:postgresql://localhost:5432/postgres
spring.datasource.username   = postgres
spring.datasource.password   = letmein
spring.datasource.driver-class-name = org.postgresql.Driver

# JPA / HIBERNATE
spring.jpa.hibernate.ddl-auto = create-drop
```

*SpringbootDemoAuthorsbooksApplication.java*

```
package com.ivoronline.springboot_demo_authorsbooks;

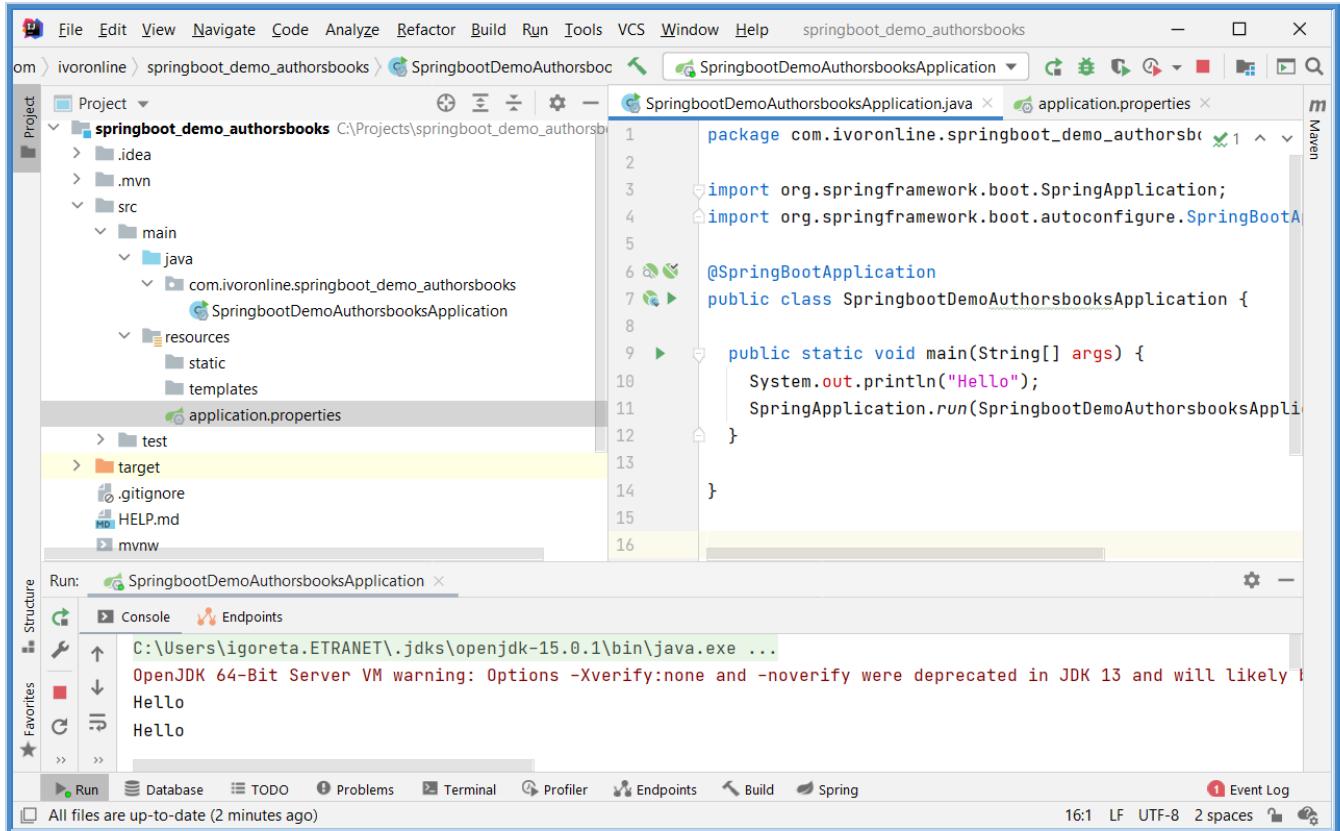
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SpringbootDemoAuthorsbooksApplication {

    public static void main(String[] args) {
        System.out.println("Hello");
        SpringApplication.run(SpringbootDemoAuthorsbooksApplication.class, args);
    }
}
```

## Results

### Application Structure



### pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.4.1</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.ivoronline</groupId>
    <artifactId>springboot_demo_authorsbooks</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>springboot_demo_authorsbooks</name>
    <description>Demo project for Spring Boot</description>

    <properties>
        <java.version>11</java.version>
    </properties>

    <dependencies>

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

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

    </dependencies>
```

```

<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <scope>runtime</scope>
</dependency>

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

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
    <optional>true</optional>
</dependency>

<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <optional>true</optional>
</dependency>

<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>
            <configuration>
                <excludes>
                    <exclude>
                        <groupId>org.projectlombok</groupId>
                        <artifactId>lombok</artifactId>
                    </exclude>
                </excludes>
            </configuration>
        </plugin>
    </plugins>
</build>

</project>

```

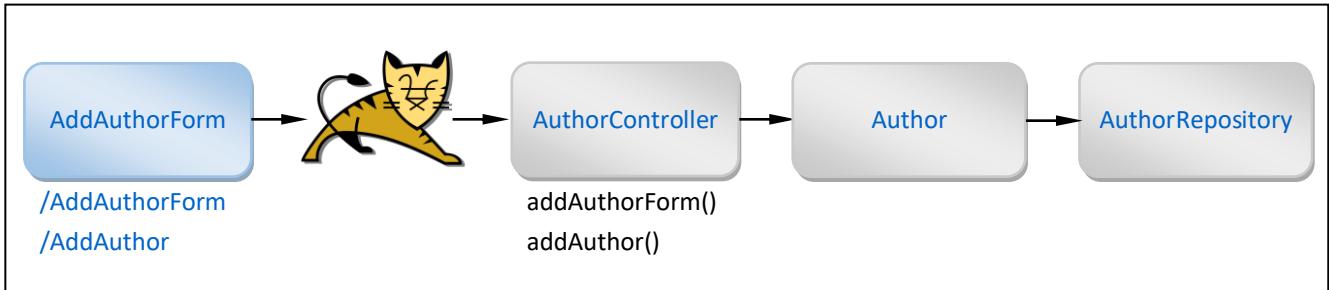
## 3.2.2 Step 2: Entity - Author

### Info

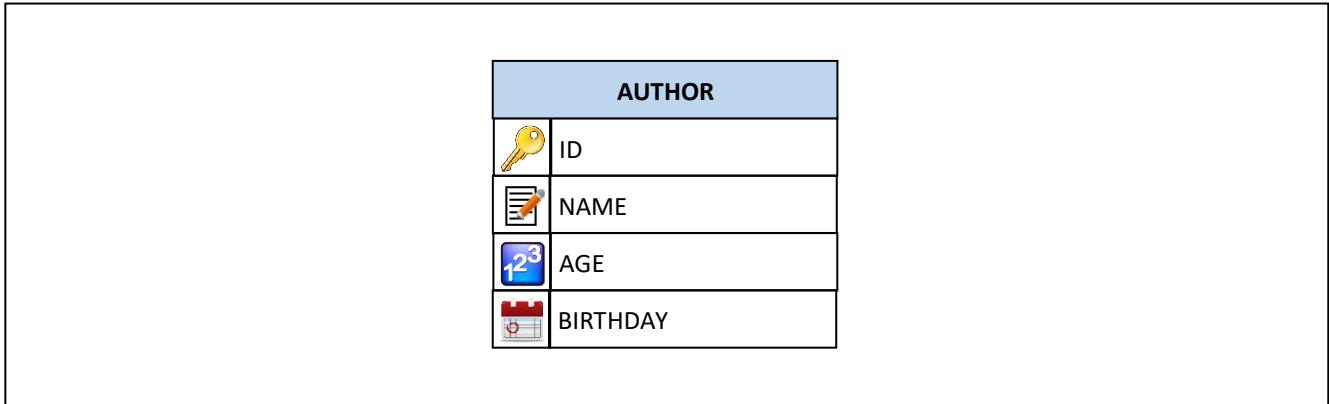
[G]

- Then we will add Author Entity and HTML Page to add new Author.

### Application Schema



### DB Schema



## Create Project

---

- Create Package: entities (inside main package)
- Create Class: [Author.java](#) (inside package entities)
- Create Package: repositories (inside main package)
- Create Interface: [AuthorRepository.java](#) (inside package repositories)
- Create Package: controllers (inside main package)
- Create Class: [AuthorController.java](#) (inside package controllers)
- Create HTML File: [AddAuthorForm.html](#) (inside directory resources/templates)

### *Author.java*

```
package com.ivoronline.springboot_demo_authorsbooks.entities;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import java.sql.Date;

@Data
@Entity
@Component
public class Author {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    private String name;
    private Integer age;
    private Date birthday;

}
```

### *AuthorRepository.java*

```
package com.ivoronline.springboot_demo_authorsbooks.repositories;

import com.ivoronline.springboot_demo_authorsbooks.entities.Author;
import org.springframework.data.repository.CrudRepository;

public interface AuthorRepository extends CrudRepository<Author, Integer> { }
```

### *AuthorController.java*

```
package com.ivorononline.springboot_demo_authorsbooks.controllers;

import com.ivorononline.springboot_demo_authorsbooks.entities.Author;
import com.ivorononline.springboot_demo_authorsbooks.repositories.AuthorRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RequestMapping;

import java.sql.Date;

@Controller
public class AuthorController {

    //=====
    // REPOSITORIES
    //=====

    @Autowired AuthorRepository authorRepository;

    //=====
    // METHOD: ADD AUTHOR FORM
    //=====

    @RequestMapping("/AddAuthorForm")
    public String addAuthorForm() {
        return "AddAuthorForm";
    }

    //=====
    // METHOD: ADD AUTHOR
    //=====

    @ResponseBody
    @RequestMapping("/AddAuthor")
    public String addAuthor(@RequestParam String name,@RequestParam Integer age,@RequestParam Date birthday) {

        //CREATE AUTHOR
        Author author = new Author();
        author.setName(name);
        author.setAge(age);
        author.setBirthday(birthday);

        //STORE AUTHOR
        authorRepository.save(author);

        //RETURN
        return "Author added to DB";
    }
}
```

```
<title> ADD AUTHOR </title>

<style type="text/css">
  div { display:flex; flex-direction:column; align-items:center; border: solid 1pt; margin: 10pt 50pt;
background-color: aliceblue }
</style>

<div>
  <h2> ADD AUTHOR </h2>
  <form method="get" action="http://localhost:8080/AddAuthor">
    <p> <input type="text" name="name" placeholder="Enter Author Name" /> </p>
    <p> <input type="text" name="age" placeholder="Enter Author Age" /> </p>
    <p> <input type="text" name="birthday" placeholder="Enter Author Birthday" /> </p>
    <p> <input type="submit" name="addAuthor" value="Add Author" style="width:100%"/> </p>
  </form>
</div>
```

## Results

<http://localhost:8080/AddAuthorForm>

**ADD AUTHOR**

<input type="text" value="Bill"/>
<input type="text" value="20"/>
<input type="text" value="2021-01-02"/>
<input type="button" value="Submit"/>

<http://localhost:8080/AddAuthorForm>

**ADD AUTHOR**

<input type="text" value="John"/>
<input type="text" value="50"/>
<input type="text" value="2004-09-12"/>
<input type="button" value="Submit"/>

<http://localhost:8080/AddAuthor?name=Bill&age=20&birthday=2021-01-02&addAuthor=Submit>

<http://localhost:8080/AddAuthor?name=John&age=20&birthday=2021-01-12&addAuthor=Submit>

A screenshot of a Microsoft Edge browser window. The address bar shows the URL "localhost:8080/AddAuthor?name=Bill&age=20&birthday=2021...". The main content area displays the message "Author added to DB".

*Author* (DB Table)

ID	NAME	AGE	BIRTHDAY
1	Bill	20	2021-01-02
2	John	50	2004-09-12

## *Application Structure*

The screenshot shows the IntelliJ IDEA interface with the following details:

- File Menu:** File, Edit, View, Navigate, Code, Analyze, Refactor, Build, Run, Tools, Git, Window, Help.
- Project Bar:** Project (selected), .idea, .mvn, src, main, java, com.ivoronline.springboot\_demo\_authorsbooks, controllers, AuthorController, entities, Author (selected), repositories, AuthorRepository, SpringbootDemoAuthorsbooksApplication, resources, static, templates, AddAuthorForm.html, application.properties, test.
- Code Editor:** Author.java (selected), AddAuthorForm.html, author\_id\_seq, application.properties.
- Toolbars:** Git, Run, Database, TODO, Problems, Terminal, Profiler, Endpoints, Build, Services, Spring, Event Log.
- Status Bar:** Connected (moments ago), 33:1 CRLF, UTF-8, 2 spaces, Authors.

```
import javax.persistence.GenerationType;
import javax.persistence.Id;
import java.sql.Date;

@Data
@Entity
@Component
public class Author {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    private String name;
    private Integer age;
    private Date birthday;
}
```

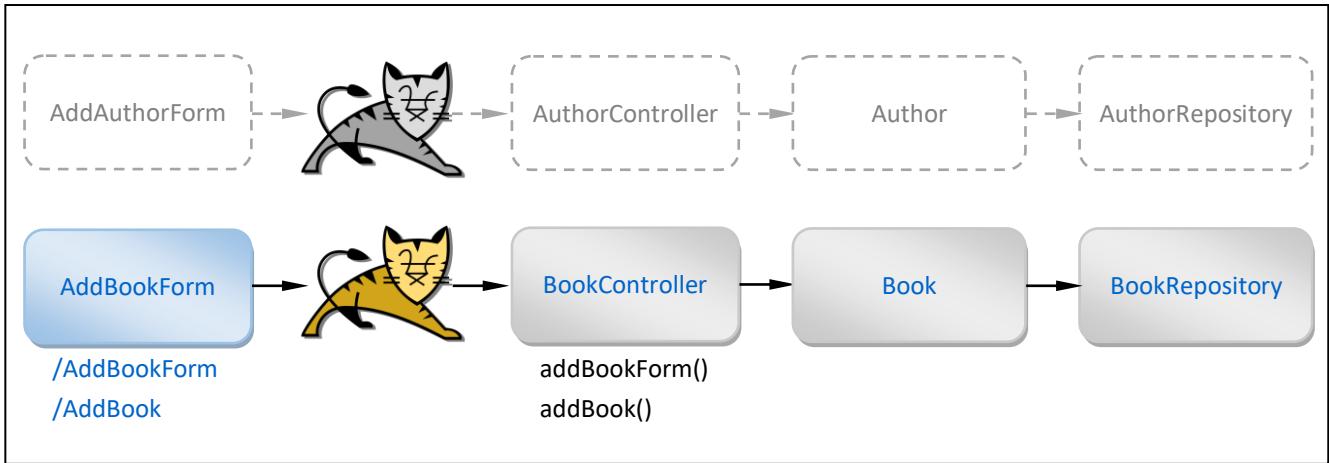
### 3.2.3 Step 3: Entity - Book

#### Info

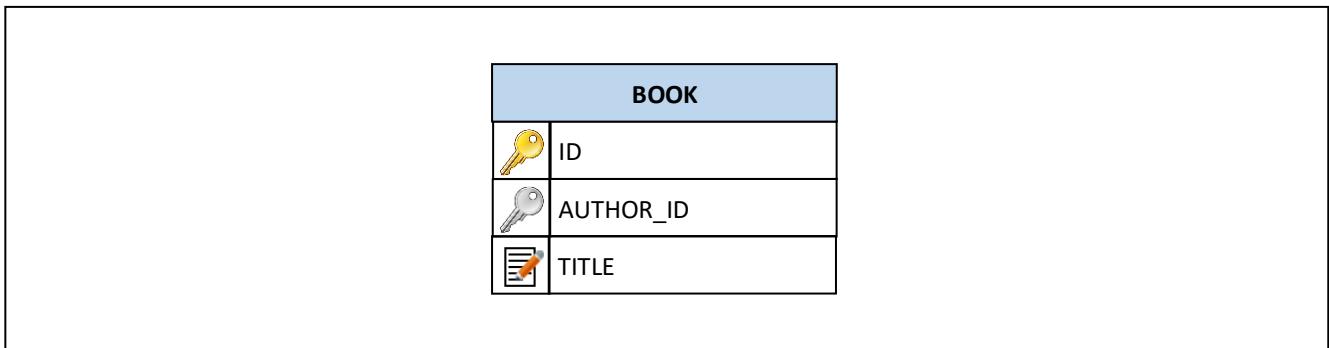
[G]

- Next we will add Book Entity and HTML Page to add new Book.
- We want to have [OneToMany](#) Relationship between Authors and Books, where one Author can have multiple Books. Therefore Entity Book needs additional Property authord that points to specific Author Entity. Therefore Table BOOK needs additional Column AUTHOR\_ID that points to specific Record in AUTHOR Table.
- Just adding Foreign Key to Book Entity will not make Hibernate aware of intended [OneToMany](#) Relationship. We will make it aware of it in the next step by using [@OneToMany](#) Annotation.

#### Application Schema



#### DB Schema



## Add Classes & HTML

---

- Create Class: [Book.java](#) (inside package: entities)
- Create Interface: [BookRepository.java](#) (inside package: repositories)
- Create Class: [BookController.java](#) (inside package: controllers)
- Create HTML File: [AddBookForm.html](#) (inside directory resources/templates)

### *Book.java*

```
package com.ivoronline.springboot_demo_authorsbooks.entities;

import lombok.Data;
import org.springframework.stereotype.Component;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Data
@Entity
@Component
public class Book {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;           //PRIMARY KEY

    private Integer authorId;     //FOREIGN KEY

    private String title;

}
```

### *BookRepository.java*

```
package com.ivoronline.springboot_demo_authorsbooks.repositories;

import com.ivoronline.springboot_demo_authorsbooks.entities.Book;
import org.springframework.data.repository.CrudRepository;

public interface BookRepository extends CrudRepository<Book, Integer> {
}
```

### *BookController.java*

```
package com.ivorononline.springboot_demo_authorsbooks.controllers;

import com.ivorononline.springboot_demo_authorsbooks.entities.Book;
import com.ivorononline.springboot_demo_authorsbooks.repositories.BookRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class BookController {

    //=====
    // REPOSITORIES
    //=====

    @Autowired BookRepository bookRepository;

    //=====
    // METHOD: ADD BOOK FORM
    //=====

    @RequestMapping("/AddBookForm")
    public String addBookForm() {
        return "AddBookForm";
    }

    //=====
    // METHOD: ADD BOOK
    //=====

    @ResponseBody
    @RequestMapping("/AddBook")
    public String addBook(@RequestParam String title) {

        //CREATE BOOK
        Book book = new Book();
        book.setTitle(title);

        //STORE BOOK
        bookRepository.save(book);

        //RETURN
        return "Book added to DB";
    }
}
```

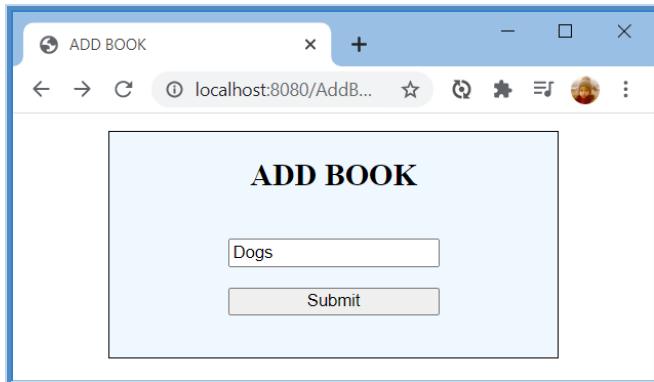
```
<title> ADD BOOK </title>

<style type="text/css">
  div { display:flex; flex-direction:column; align-items:center; border: solid 1pt; margin: 10pt 50pt;
background-color: aliceblue }
</style>

<div>
  <h2> ADD BOOK </h2>
  <form method="get" action="http://localhost:8080/AddBook">
    <p> <input type="text" name="title" placeholder="Enter Book Title"/> </p>
    <p> <input type="text" name="authorId" placeholder="Enter Author Id" /> </p>
    <p> <input type="submit" name="addBook" style="width:100%" /> </p>
  </form>
</div>
```

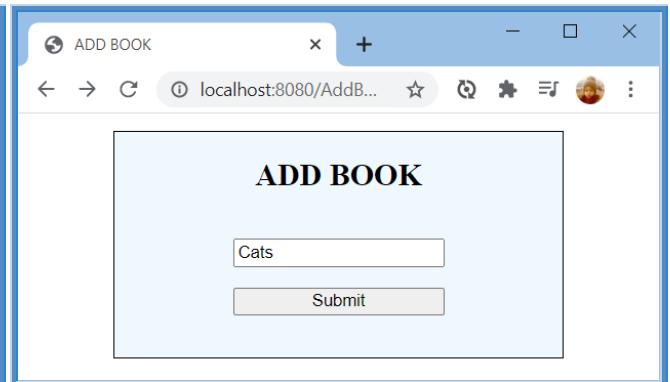
## Results

<http://localhost:8080/AddBookForm>



The screenshot shows a browser window titled "ADD BOOK". Inside, there is a form with a single input field containing the text "Dogs" and a "Submit" button below it.

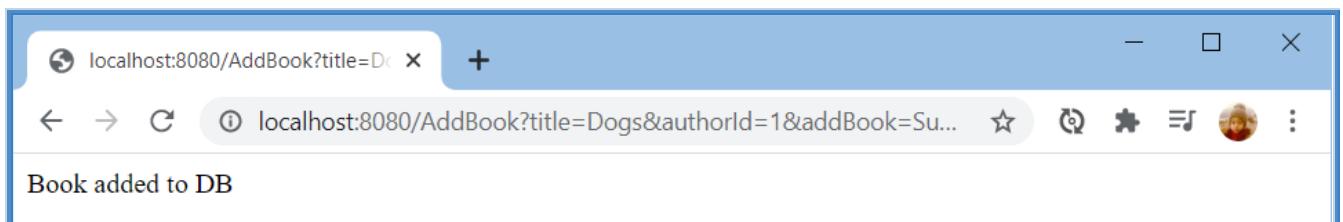
<http://localhost:8080/AddBookForm>



The screenshot shows a browser window titled "ADD BOOK". Inside, there is a form with a single input field containing the text "Cats" and a "Submit" button below it.

<http://localhost:8080/AddBook?title=Dogs&authorId=1&addBook=Submit>

<http://localhost:8080/AddBook?title=Cats&authorId=1&addBook=Submit>



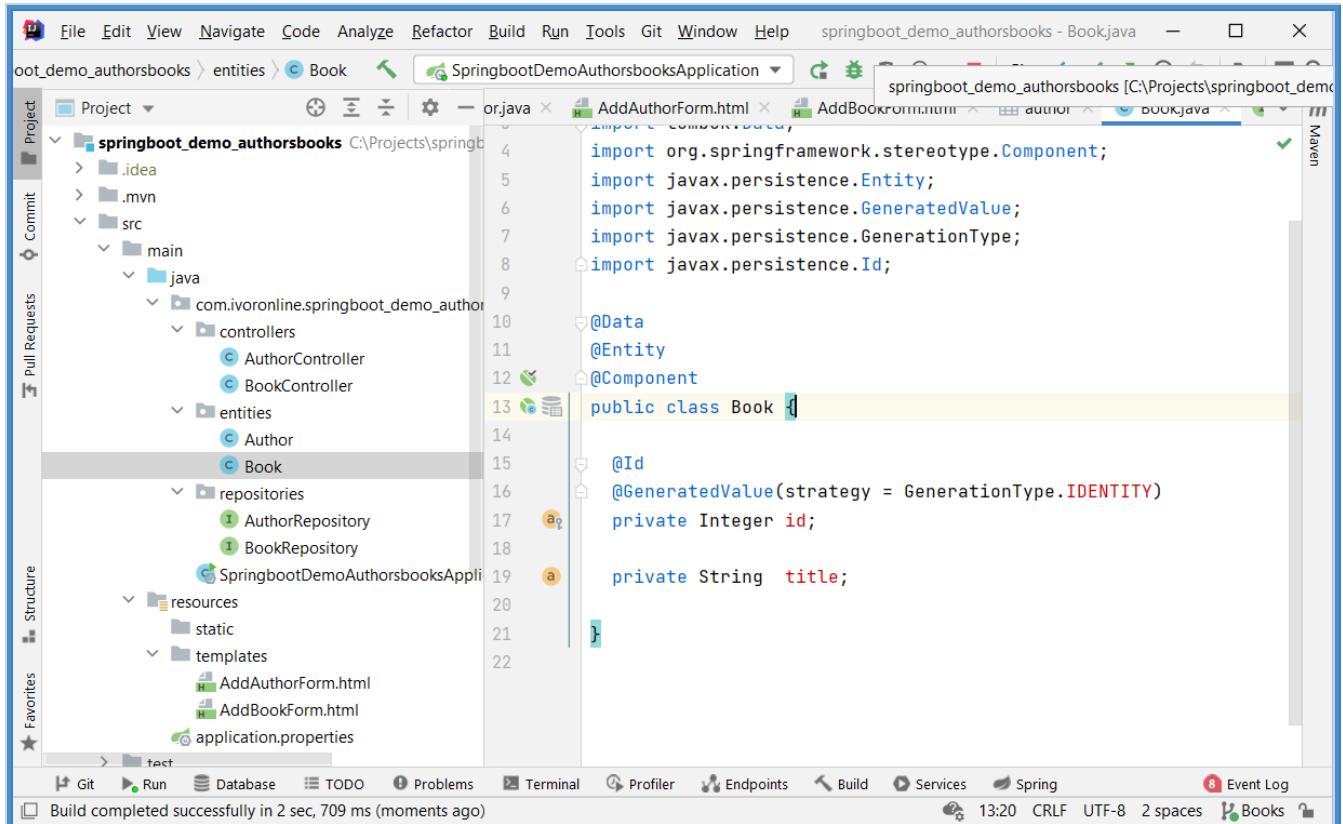
The screenshot shows a browser window with the URL "localhost:8080/AddBook?title=Dogs&authorId=1&addBook=Submit". Below the address bar, the text "Book added to DB" is displayed in a large, bold font.

Book

(DB Table)

ID	TITLE	AUTHOR_ID
1	Dogs	1
2	Cats	1

## Application Structure



The screenshot shows the IntelliJ IDEA interface with the project "springboot\_demo\_authorsbooks" open. The left sidebar displays the project structure, showing packages like "com.ivoronline.springboot\_demo\_authorsbooks" containing "AuthorController", "BookController", and "entities" (with "Author" and "Book" classes). The right side shows the code editor for the "Book.java" file, which contains the following code:

```
import com.fasterxml.jackson.annotation.JsonIgnore;
import org.springframework.stereotype.Component;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
@Data
@Entity
@Component
public class Book {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    private String title;
}
```

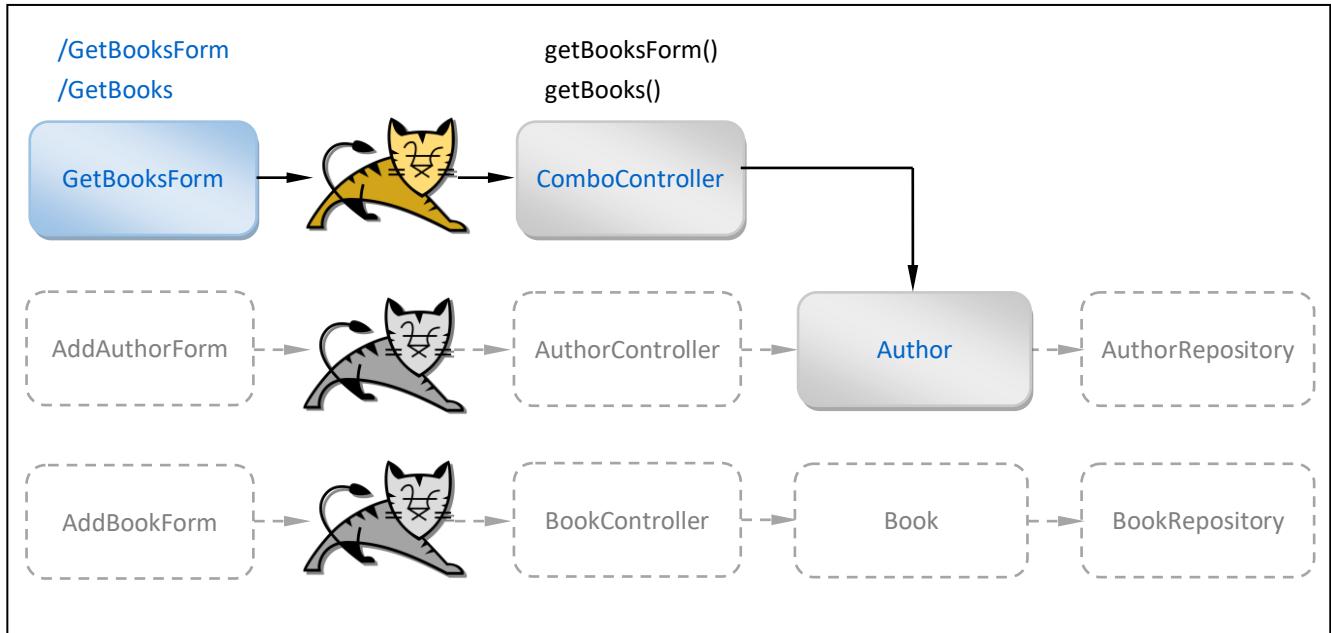
## 3.2.4 Step 4: Relationship @OneToMany

### Info

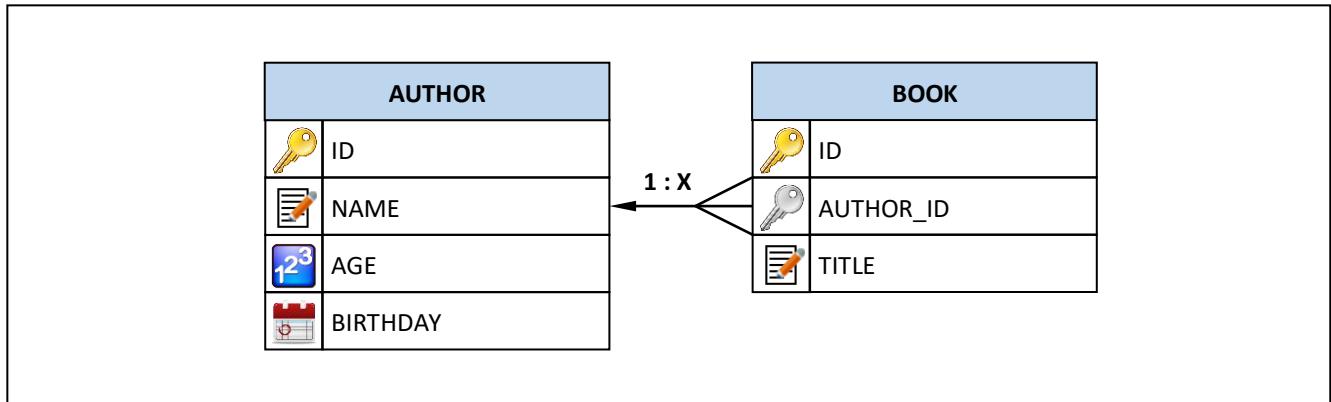
[G]

- Next we will use `@OneToMany` Annotation to make Hibernate aware of intended Relationship between Authors & Books.

### Application Schema



### DB Schema



## Add Classes & HTML

---

- Edit Class: [Author.java](#) (inside package: entities)
- Create Class: [ComboController.java](#) (inside package: controllers)
- Create HTML File: [GetBooksForm.html](#) (inside directory resources/templates)

*Author.java*

```
package com.ivoronline.springboot_demo_authorsbooks.entities;

import lombok.Data;
import org.springframework.stereotype.Component;
import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.OneToMany;
import java.sql.Date;
import java.util.Set;

@Data
@Entity
@Component
public class Author {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    private String name;
    private Integer age;
    private Date birthday;

    @OneToMany(cascade = CascadeType.ALL)
    @JoinColumn(name = "authorId")
    private Set<Book> books;
}
```

### ComboController.java

```
package com.ivorononline.springboot_demo_authorsbooks.controllers;

import com.ivorononline.springboot_demo_authorsbooks.entities.Author;
import com.ivorononline.springboot_demo_authorsbooks.entities.Book;
import com.ivorononline.springboot_demo_authorsbooks.repositories.AuthorRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseBody;

import java.util.Optional;
import java.util.Set;

@Controller
public class ComboController {

    //=====
    // REPOSITORIES
    //=====

    @Autowired AuthorRepository authorRepository;

    //=====
    // METHOD: GET BOOKS FORM
    //=====

    @RequestMapping("/GetBooksForm")
    public String getBooksForm() {
        return "GetBooksForm";
    }

    //=====
    // METHOD: GET BOOKS
    //=====

    @ResponseBody
    @RequestMapping("/GetBooks")
    public String getBooks(@RequestParam Integer authorId) {

        //GET AUTHOR & BOOKS
        Optional<Author> authorOptional = authorRepository.findById(authorId);
        Author author = authorOptional.get();
        Set<Book> books = author.getBooks();

        //DISPLAY BOOKS
        String Results = "<h2> Books by " + author.getName() + "</h2>";
        for(Book book : books) {
            Results += book.getTitle() + "<br>";
        }

        //RETURN
        return Results;
    }
}
```

*GetBooksForm.html*

```
<title> GET BOOKS </title>

<style type="text/css">
  div { display:flex; flex-direction:column; align-items:center; border: solid 1pt; margin: 10pt 50pt;
background-color: aliceblue }
</style>

<div>
  <h2> GET BOOKS </h2>
  <form method="get" action="http://localhost:8080/GetBooks">
    <p> <input type="text" name="authorId" placeholder="Enter Author Id"/> </p>
    <p> <input type="submit" name="getBooks" style="width:100%" /> </p>
  </form>
</div>
```

## Results

- To test the application store one Author and two Books and then show Author's Books.

<http://localhost:8080/AddAuthorForm>

ADD AUTHOR

Bill

20

2021-01-02

Submit

<http://localhost:8080/AddBookForm>

ADD BOOK

Dogs

1

Submit

<http://localhost:8080/AddBookForm>

ADD BOOK

Cats

1

Submit

<http://localhost:8080/GetBooksForm>

GET BOOKS

1

Submit

<http://localhost:8080/GetBooks?authorId=1&getBooks=Submit>

Books by Bill

Dogs  
Cats

Author

(DB Table)

ID	NAME	AGE	BIRTHDAY
1	Bill	20	2021-01-02
2	John	50	2004-09-12

Book

(DB Table)

ID	TITLE	AUTHOR_ID
1	Dogs	1
2	Cats	1

## Application Structure

The screenshot shows the IntelliJ IDEA interface with the project 'springboot\_demo\_authorsbooks' open. The left sidebar displays the project structure, including the 'entities' package which contains the 'Author' class. The right side shows the code editor with the 'Author.java' file open. The code defines an entity 'Author' with fields for id, name, age, and birthday, and a relationship to 'Book' via a 'Set' named 'books'. The code editor also shows annotations like @Data, @Entity, and @Component.

```
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import java.sql.Date;
import java.util.Set;

@Data
@Entity
@Component
public class Author {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    private String name;
    private Integer age;
    private Date birthday;

    @OneToMany(mappedBy = "authorId")
    private Set<Book> books;
}
```

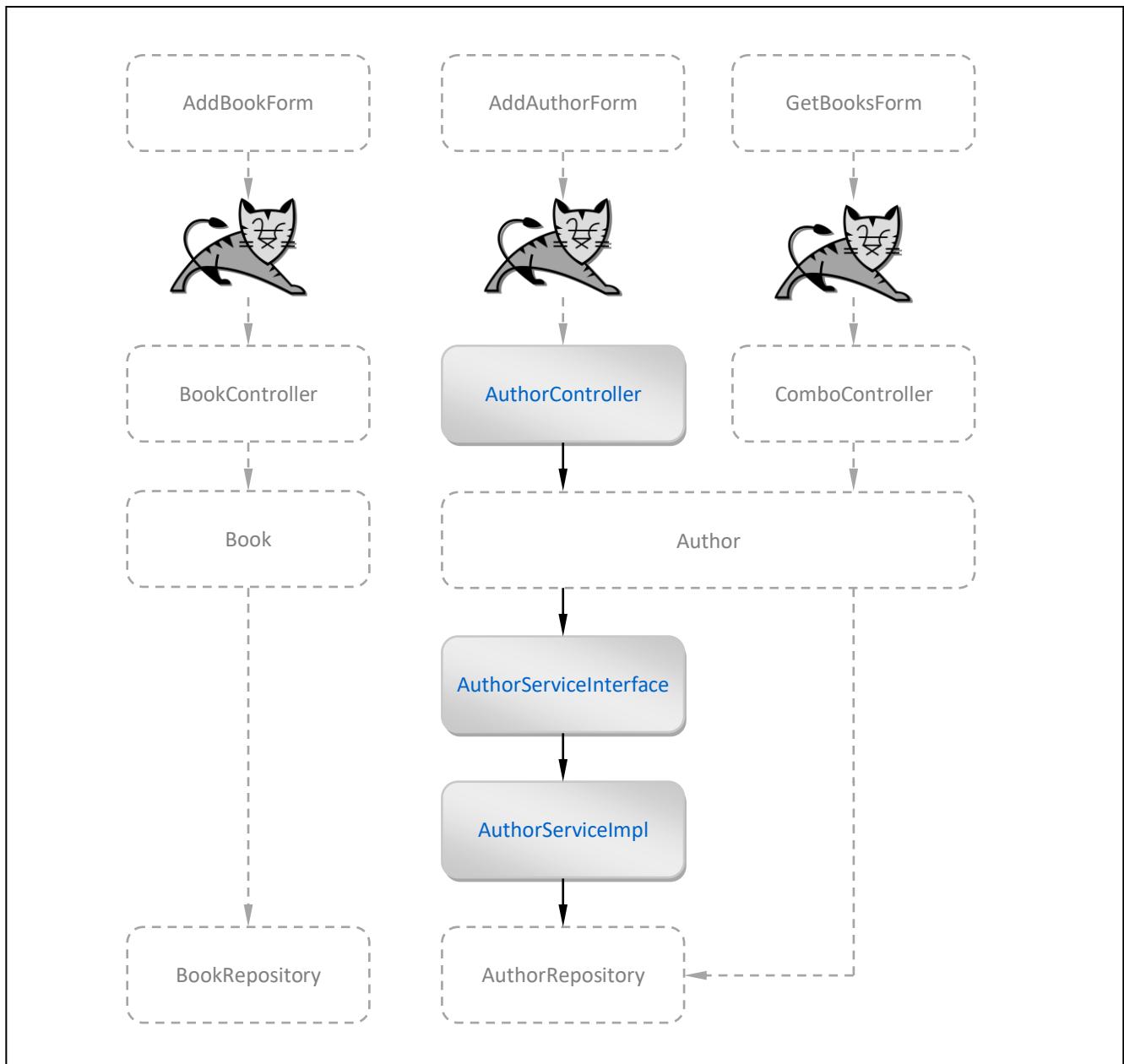
## 3.2.5 Step 5: Service - Add Author

### Info

[G]

- Next we will add Service Layer that moves business logic from `AuthorController` which
  - will use `@Autowired AuthorServiceInterface` to decouple actual Service Implementation from the Controller
  - will use `Author Entity` to communicate with the Service
  - will no longer directly access `AuthorRepository`

*Application Schema*



## Add Classes & HTML

---

- Create Interface: [AuthorServiceInterface.java](#) (inside package: services)
- Create Class: [AuthorServiceImpl.java](#) (inside package: services)
- Create Class: [AuthorController.java](#) (inside package: controllers)

### *AuthorServiceInterface.java*

```
package com.ivoronline.springboot_demo_authorsbooks.services;

import com.ivoronline.springboot_demo_authorsbooks.entities.Author;

public interface AuthorServiceInterface {
    String addAuthor(Author author);
}
```

### *AuthorServiceImpl.java*

```
package com.ivoronline.springboot_demo_authorsbooks.services;

import com.ivoronline.springboot_demo_authorsbooks.entities.Author;
import com.ivoronline.springboot_demo_authorsbooks.repositories.AuthorRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class AuthorServiceImpl implements AuthorServiceInterface {

    @Autowired
    AuthorRepository authorRepository;

    @Override
    public String addAuthor(Author author) {

        //STORE AUTHOR (BUSINESS LOGIC)
        authorRepository.save(author);

        //RETURN
        return "Author added to DB";
    }
}
```

### *AuthorController.java*

```
package com.ivorononline.springboot_demo_authorsbooks.controllers;

import com.ivorononline.springboot_demo_authorsbooks.entities.Author;
import com.ivorononline.springboot_demo_authorsbooks.services.AuthorServiceInterface;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RequestMapping;

import java.sql.Date;

@Controller
public class AuthorController {

    //=====
    // SERVICES
    //=====

    @Autowired AuthorServiceInterface authorService;

    //=====
    // METHOD: ADD AUTHOR FORM
    //=====

    @RequestMapping("/AddAuthorForm")
    public String addAuthorForm() {
        return "AddAuthorForm";
    }

    //=====
    // METHOD: ADD AUTHOR
    //=====

    @ResponseBody
    @RequestMapping("/AddAuthor")
    public String addAuthor(@RequestParam String name, @RequestParam Integer age, @RequestParam Date birthday)
    {

        //CREATE AUTHOR
        Author author = new Author();
        author.setName(name);
        author.setAge(age);
        author.setBirthday(birthday);

        //CALL SERVICE (BUSINESS LOGIC)
        String Results = authorService.addAuthor(author);

        //RETURN
        return Results;
    }
}
```

## Results

- Since Interface has not changed you can use [Results](#) from Step 3: Relationship @OneToMany to test Application.

### Application Structure

The screenshot shows the IntelliJ IDEA interface with the following details:

- Project View:** Shows the directory structure of the Spring Boot application. The `src/main/java` folder contains packages like `com.ivoronline.springboot_demo_authorsbooks.controllers`, `com.ivoronline.springboot_demo_authorsbooks.entities`, `com.ivoronline.springboot_demo_authorsbooks.repositories`, and `com.ivoronline.springboot_demo_authorsbooks.services`. It also includes `pom.xml`, `application.properties`, and various template files (`AddAuthorForm.html`, `AddBookForm.html`, `GetBooksForm.html`) under `src/main/resources/templates`.
- Code Editor:** Displays the `AuthorController.java` file. The code defines a controller for managing authors. It includes annotations for `@Controller`, `@Autowired`, and `@RequestMapping`. The logic involves creating a new author and calling a service method to add it.
- Toolbars and Status Bar:** The top bar shows the project name `SpringbootDemoAuthorsbooksApplication`. The bottom status bar indicates the current time is 29:59, the encoding is CRLF, and there are 2 spaces.

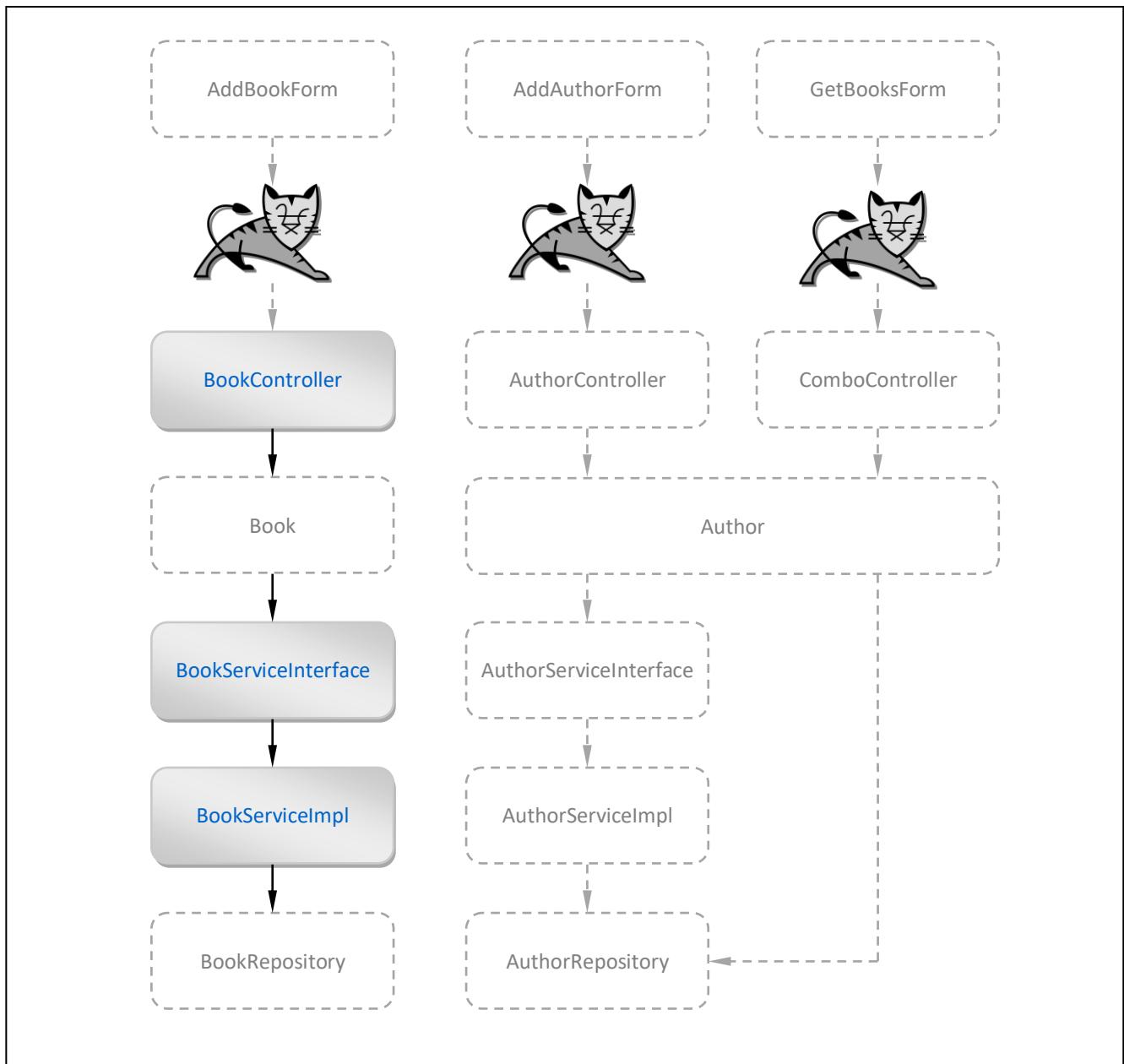
## 3.2.6 Step 6: Service - Add Book

Info

[G]

- Next we will add Service Layer that moves business logic from [AuthorController](#) which
  - will use [@Autowired AuthorServiceInterface](#) to decouple actual Service Implementation from the Controller
  - will use [Author Entity](#) to communicate with the Service
  - will no longer directly access [AuthorRepository](#)

Application Schema



## Add Classes & HTML

---

- Create Interface: [BookServiceInterface.java](#) (inside package: services)
- Create Class: [BookServiceImpl.java](#) (inside package: services)
- Create Class: [BookController.java](#) (inside package: controllers)

### *BookServiceInterface.java*

```
package com.ivoronline.springboot_demo_authorsbooks.services;

import com.ivoronline.springboot_demo_authorsbooks.entities.Book;

public interface BookServiceInterface {
    String addBook(Book book);
}
```

### *BookServiceImpl.java*

```
package com.ivoronline.springboot_demo_authorsbooks.services;

import com.ivoronline.springboot_demo_authorsbooks.entities.Book;
import com.ivoronline.springboot_demo_authorsbooks.repositories.BookRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class BookServiceImpl implements BookServiceInterface {

    @Autowired
    BookRepository bookRepository;

    public String addBook(Book book) {

        //STORE AUTHOR (BUSINESS LOGIC)
        bookRepository.save(book);

        //RETURN
        return "Book added to DB";
    }
}
```

### BookController.java

```
package com.ivorononline.springboot_demo_authorsbooks.controllers;

import com.ivorononline.springboot_demo_authorsbooks.entities.Book;
import com.ivorononline.springboot_demo_authorsbooks.repositories.BookRepository;
import com.ivorononline.springboot_demo_authorsbooks.services.BookServiceInterface;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class BookController {

    //=====
    // SERVICES
    //=====

    @Autowired BookServiceInterface bookService;

    //=====
    // METHOD: ADD BOOK FORM
    //=====

    @RequestMapping("/AddBookForm")
    public String addBookForm() {
        return "AddBookForm";
    }

    //=====
    // METHOD: ADD BOOK
    //=====

    @ResponseBody
    @RequestMapping("/AddBook")
    public String addBook(@RequestParam String title, @RequestParam Integer authorId) {

        //CREATE BOOK
        Book book = new Book();
        book.setTitle(title);
        book.setAuthorId(authorId);

        //STORE BOOK
        bookService.addBook(book);

        //RETURN
        return "Book added to DB";
    }
}
```

## Results

- Since Interface has not changed you can use [Results](#) from Step 3: Relationship @OneToMany to test Application.

### Application Structure

The screenshot shows the project structure and code editor for a Spring Boot application named "springboot\_demo\_authorsbooks".

**Project Structure:**

- src/main/java/com.ivoronline.springboot\_demo\_authorsbooks/controllers:
  - AuthorController
  - BookController
  - ComboController
- src/main/java/com.ivoronline.springboot\_demo\_authorsbooks/entities:
  - Author
  - Book
- src/main/java/com.ivoronline.springboot\_demo\_authorsbooks/repositories:
  - AuthorRepository
  - BookRepository
- src/main/java/com.ivoronline.springboot\_demo\_authorsbooks/services:
  - AuthorServiceImpl
  - AuthorServiceInterface
  - BookServiceImpl
  - BookServiceInterface
- src/main/resources:
  - static
  - templates
    - AddAuthorForm.html
    - AddBookForm.html
    - GetBooksForm.html
  - application.properties
- test
- target
- .gitignore
- HELP.md
- mvnw
- mvnw.cmd
- pom.xml
- springboot\_demo\_authorsbooks.iml

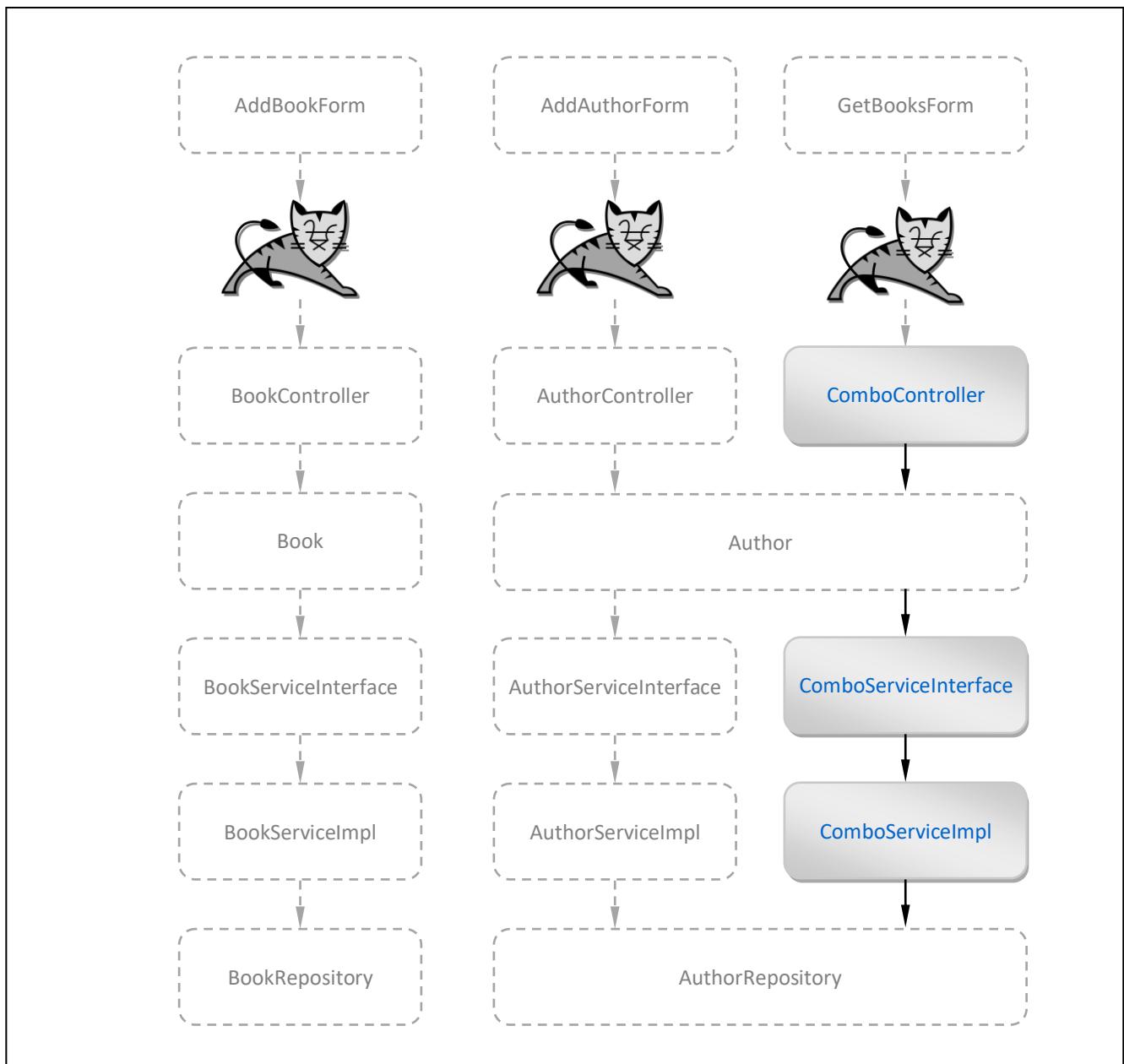
## 3.2.7 Step 7: Service - Get Books

### Info

[G]

- Next we will add Service Layer that moves business logic from `ComboController` which
  - will use `@Autowired ComboServiceInterface` to decouple actual Service Implementation from the Controller
  - will use `Author` Entity to communicate with the Service
  - will no longer directly access `AuthorRepository` to get Author and his Books
- After this step business logic will be transferred from all Controllers into respective Services.  
But Controllers will still use Entities in order to Communicate with Services.  
So in the next steps we will introduce Data Transfer Objects between Controllers and Services.

### Application Schema



## Add Classes & HTML

---

- Create Interface: [ComboServiceInterface.java](#) (inside package: services)
- Create Class: [ComboServiceImpl.java](#) (inside package: services)
- Create Class: [ComboController.java](#) (inside package: controllers)

### *ComboServiceInterface.java*

```
package com.ivoronline.springboot_demo_authorsbooks.services;

import com.ivoronline.springboot_demo_authorsbooks.entities.Author;

public interface ComboServiceInterface {
    String getBooks(Author author);
}
```

### *ComboServiceImpl.java*

```
package com.ivoronline.springboot_demo_authorsbooks.services;

import com.ivoronline.springboot_demo_authorsbooks.entities.Author;
import com.ivoronline.springboot_demo_authorsbooks.entities.Book;
import com.ivoronline.springboot_demo_authorsbooks.repositories.AuthorRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.Optional;
import java.util.Set;

@Service
public class ComboServiceImpl implements ComboServiceInterface {

    @Autowired
    AuthorRepository authorRepository;

    public String getBooks(Author author) {

        //GET AUTHOR & BOOKS
        Optional<Author> authorOptional = authorRepository.findById(author.getId());
        author = authorOptional.get();
        Set<Book> books = author.getBooks();

        //DISPLAY BOOKS
        String Results = "<h2> Books by " + author.getName() + "</h2>";
        for(Book book : books) {
            Results += book.getTitle() + "<br>";
        }

        //RETURN
        return Results;
    }
}
```

### ComboController.java

```
package com.ivorononline.springboot_demo_authorsbooks.controllers;

import com.ivorononline.springboot_demo_authorsbooks.entities.Author;
import com.ivorononline.springboot_demo_authorsbooks.services.ComboServiceInterface;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class ComboController {

    //=====
    // SERVICES
    //=====

    @Autowired
    ComboServiceInterface comboService;

    //=====
    // METHOD: GET BOOKS FORM
    //=====

    @RequestMapping("/GetBooksForm")
    public String getBooksForm() {
        return "GetBooksForm";
    }

    //=====
    // METHOD: GET BOOKS
    //=====

    @ResponseBody
    @RequestMapping("/GetBooks")
    public String getBooks(@RequestParam Integer authorId) {

        //CREATE AUTHOR
        Author author = new Author();
        author.setId(authorId);

        //CALL SERVICE (BUSINESS LOGIC)
        String Results = comboService.getBooks(author);

        //RETURN
        return Results;
    }
}
```

## Results

- Since Interface has not changed you can use [Results](#) from Step 3: Relationship @OneToMany to test Application.

### Application Structure

The screenshot shows the IntelliJ IDEA interface with the following details:

- Project View:** Shows the project structure under "Project".
  - src/main/java/com.ivoronline.springboot\_demo\_authorsbooks/controllers:** Contains `AuthorController.java`, `BookController.java`, and `ComboController.java`.
  - src/main/java/com.ivoronline.springboot\_demo\_authorsbooks/entities:** Contains `Author.java` and `Book.java`.
  - src/main/java/com.ivoronline.springboot\_demo\_authorsbooks/repositories:** Contains `AuthorRepository.java` and `BookRepository.java`.
  - src/main/java/com.ivoronline.springboot\_demo\_authorsbooks/services:** Contains `AuthorServiceImpl.java`, `AuthorServiceInterface.java`, `BookServiceImpl.java`, `BookServiceInterface.java`, `ComboServiceImpl.java`, and `ComboServiceInterface.java`.
  - src/main/resources:** Contains static files and templates.
    - `static`
    - `templates` containing `AddAuthorForm.html`, `AddBookForm.html`, and `GetBooksForm.html`.
    - `application.properties`
- Code Editor:** Displays the `ComboController.java` file with the following code:`11 @Controller
12 public class ComboController {
13
14 //=====
15 // SERVICES
16 //=====
17
18 @Autowired
19 ComboServiceInterface comboService;
20
21 //=====
22 // METHOD: GET BOOKS FORM
23 //=====
24 @RequestMapping("/GetBooksForm")
25 public String getBooksForm() { return "GetBooksForm"; }
26
27 //=====
28 // METHOD: GET BOOKS
29 //=====
30
31 @ResponseBody
32 @RequestMapping("/GetBooks")
33 public String getBooks(@RequestParam Integer authorId) {
34
35 //CREATE AUTHOR
36 Author author = new Author();
37 author.setId(authorId);
38
39 //CALL SERVICE (BUSINESS LOGIC)
40 String result = comboService.getBooks(author);
41
42 //RETURN
43 return result;
44 }
45 }`
- Toolbars and Status Bar:** Includes Git, Run, Database, TODO, Problems, Terminal, Profiler, Endpoints, Build, Services, Spring, and Event Log. The status bar shows the time as 12:14, encoding as CRLF, and file as Step6\_ServiceCombo.

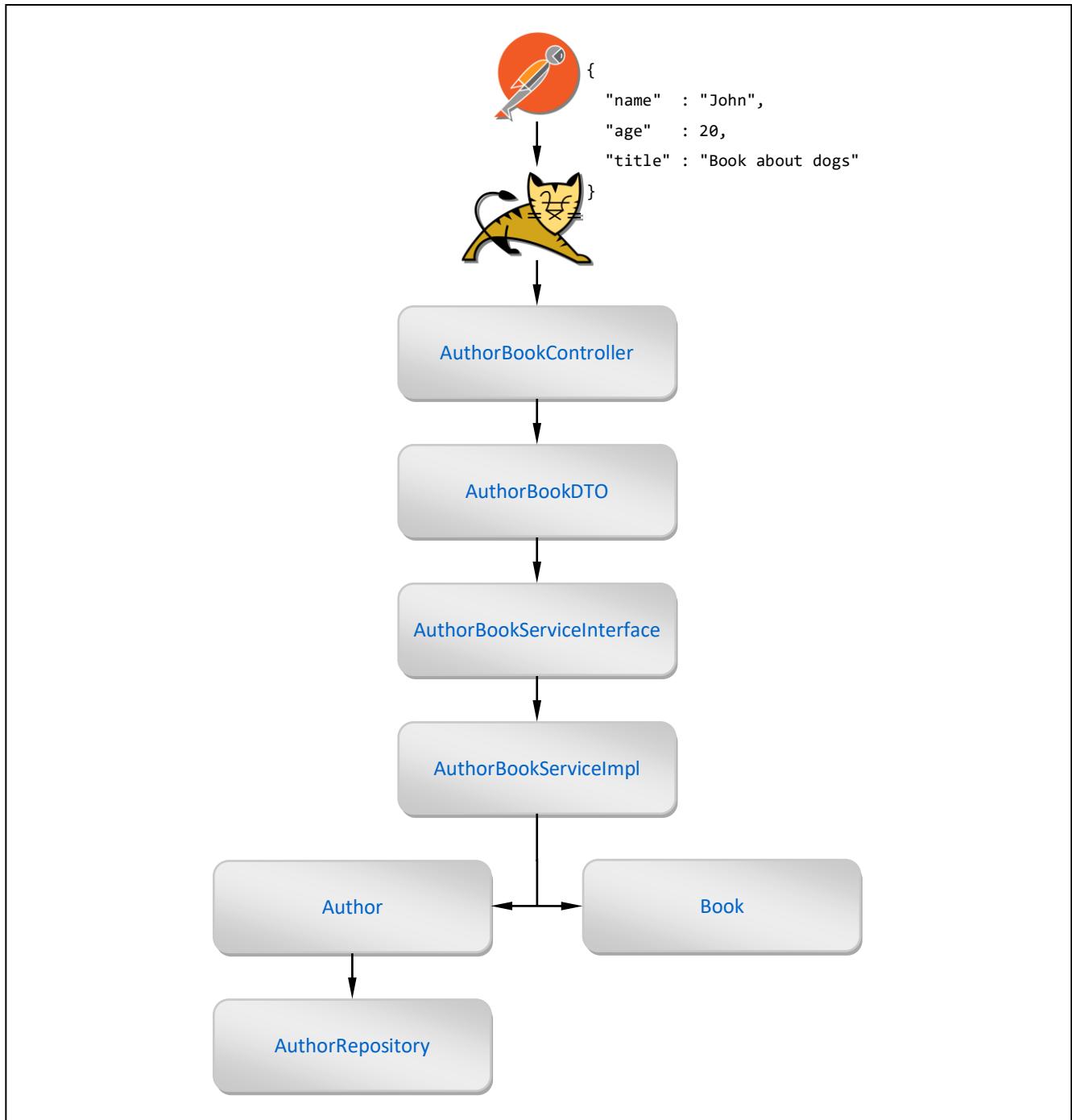
## 3.2.8 Step 8: Service - Add Author & Book

### Info

[G]

- Next we will create
  - new `AuthorBookController` which accepts **JSON** that contains both Author and his Book
  - which will be converted inside Endpoint into Data Transfer object (**DTO**) `AuthorBookDTO`
  - which will be used as input parameter to new **Service** `AuthorBookServiceImpl`
  - which will convert DTO into separate `Author` and `Book Entities`
  - which will be stored into **DB** using `AuthorRepository`
- `AuthorBookServiceImpl` returns `AuthorBookDTO` with populated Ids of inserted Author and Book. `AuthorBookController` returns this `AuthorBookDTO` in HTTP Response Body in JSON format.

*Application Schema*



## Add Classes & HTML

- Edit File: [pom.xml](#) (add dependency for modelmapper)
- Create Package: [DTO](#) (inside main package)
- Create Class: [AuthorBookDTO.java](#) (inside package DTO)
- Create Interface: [AuthorBookServiceInterface.java](#) (inside package: services)
- Create Class: [AuthorBookServiceImpl.java](#) (inside package: services)
- Create Class: [AuthorBookController.java](#) (inside package: controllers)

*pom.xml*

```
<dependency>
    <groupId>org.modelmapper</groupId>
    <artifactId>modelmapper</artifactId>
    <version>2.3.9</version>
</dependency>
```

*AuthorBookDTO.java*

```
package com.ivoronline.springboot_demo_authorsbooks.DTO;

import lombok.Data;
import org.springframework.stereotype.Component;
import java.sql.Date;

@Data
@Component
public class AuthorBookDTO {

    //AUTHOR PROPERTIES
    private Integer authorId;
    private String name;
    private Integer age;
    private Date birthday;

    //BOOK PROPERTIES
    private Integer bookId;
    private String title;

}
```

### *AuthorBookServiceInterface.java*

```
package com.ivorononline.springboot_demo_authorsbooks.services;

import com.ivorononline.springboot_demo_authorsbooks.DTO.AuthorBookDTO;

public interface AuthorBookServiceInterface {
    AuthorBookDTO addAuthorBook(AuthorBookDTO authorBookDTO);
}
```

### *AuthorBookServiceImpl.java*

```
package com.ivorononline.springboot_demo_authorsbooks.services;

import com.ivorononline.springboot_demo_authorsbooks.DTO.AuthorBookDTO;
import com.ivorononline.springboot_demo_authorsbooks.entities.Author;
import com.ivorononline.springboot_demo_authorsbooks.entities.Book;
import com.ivorononline.springboot_demo_authorsbooks.repositories.AuthorRepository;
import org.modelmapper.ModelMapper;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class AuthorBookServiceImpl implements AuthorBookServiceInterface {

    @Autowired
    AuthorRepository authorRepository;

    @Override
    public AuthorBookDTO addAuthorBook(AuthorBookDTO authorBookDTO) {

        //INSTANTIATE MODEL MAPPER
        ModelMapper modelMapper = new ModelMapper();
        modelMapper.getConfiguration().setMatchingStrategy(MatchingStrategies.STRICT);

        //CONVERT DTOs TO ENTITIES
        Book book = modelMapper.map(authorBookDTO, Book.class);
        Author author = modelMapper.map(authorBookDTO, Author.class);
        author.getBooks().add(book);

        //SAVE AUTHOR & BOOK
        authorRepository.save(author);

        //UPDATE DTO
        authorBookDTO.setAuthorId(author.getId());
        authorBookDTO.setBookId(book.getId());

        //RETURN
        return authorBookDTO;
    }
}
```

### *AuthorBookController.java*

```
package com.ivorononline.springboot_demo_authorsbooks.controllers;

import com.ivorononline.springboot_demo_authorsbooks.DTO.AuthorBookDTO;
import com.ivorononline.springboot_demo_authorsbooks.services.AuthorBookServiceInterface;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class AuthorBookController {

    //=====
    // SERVICES
    //=====

    @Autowired AuthorBookServiceInterface authorBookServiceInterface;

    //=====
    // METHOD: ADD AUTHOR BOOK
    //=====

    @ResponseBody
    @RequestMapping("addAuthorBook")
    public AuthorBookDTO addAuthorBook(@RequestBody AuthorBookDTO authorBookDTO) {

        //CALL SERVICE
        AuthorBookDTO Results = authorBookServiceInterface.addAuthorBook(authorBookDTO);

        //RETURN
        return Results;
    }
}
```

## Send HTTP Request with Postman

- Start Postman
- POST: <http://localhost:8080/addAuthorBook>
- Headers: (copy from below)
- Body: (copy from below)
- Send

### Headers

(add Key-Value)

```
Content-Type: application/json
```

### Body

(option: raw)

```
{
  "name" : "John",
  "age" : 20
  "title" : "Book about dogs"
}
```

### HTTP Response Body

```
John has written: Book about dogs
```

### HTTP Request Body (raw - JSON)

The screenshot shows the Postman interface with a request labeled 'addAuthorBook'. The 'Body' tab is selected, showing a red box around the 'Body' section. Below it, the 'raw' option is selected. The JSON payload is:

```
1 {
2   "name" : "John",
3   "age" : 20,
4   "title" : "Book about dogs"
5 }
```

### HTTP Request Headers Content-Type: application/json

The screenshot shows the Postman interface with a request labeled 'addAuthorBook'. The 'Headers' tab is selected, showing a red box around the 'Content-Type' field. The value is set to 'application/json'. Other headers listed are Accept, Accept-Encoding, and Connection.

Key	Value
Accept	/*
Accept-Encoding	gzip, deflate, br
Connection	keep-alive
Content-Type	application/json

## Results

### Application Structure

The screenshot shows the IntelliJ IDEA interface with the following details:

- Project Tree:** The project structure is displayed under the "Project" tab. It includes:
  - main
  - java:
    - com.ivoronline.springboot\_demo\_authorsbooks:
      - controllers (containing AuthorBookController, AuthorController, BookController, ComboController)
      - DTO (containing AuthorBookDTO)
      - entities (containing Author, Book)
      - repositories (containing AuthorRepository, BookRepository)
      - services (containing AuthorBookServiceImpl, AuthorBookServiceInterface, AuthorServiceImpl, AuthorServiceInterface, BookServiceImpl, BookServiceInterface, ComboServiceImpl, ComboServiceInterface)
    - SpringbootDemoAuthorsbooksApplication
  - resources
  - static
  - templates (containing AddAuthorForm.html, AddBookForm.html, GetBooksForm.html)
  - application.properties
- Code Editor:** The code editor shows the `AuthorBookController.java` file. The code is as follows:

```
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class AuthorBookController {

    //=====
    // SERVICES
    //=====

    @Autowired AuthorBookServiceInterface authorBookServiceInterface;

    //=====
    // METHOD: ADD AUTHOR BOOK
    //=====

    @ResponseBody
    @RequestMapping("addAuthorBook")
    public String addAuthorBook(@RequestBody AuthorBookDTO authorBookDTO) {
        //CALL SERVICE
        String result = authorBookServiceInterface.addAuthorBook(authorBookDTO);

        //RETURN
        return result;
    }
}
```

- Toolbars and Status Bar:** The top bar includes File, Edit, View, Navigate, Code, Analyze, Refactor, Build, Run, Tools, Git, Window, Help, and a Git status bar. The bottom bar shows tabs for Git, Run, Database, TODO, Problems, Terminal, Profiler, Endpoints, Build, Services, Spring, and Event Log. The status bar at the bottom indicates "All files are up-to-date (14 minutes ago)" and shows the current time as 12:14.

# 4 Appendix

## Info

---

- Following tutorials show how to use additional technologies that can be helpful while developing Spring Boot Application.

# 4.1 Lombok

## Info

- Following tutorials show how to use
  - Lombok API** contains Annotations that automatically generate helper methods (setters, getters, constructors)
  - Lombok Plugin**
    - prevents IntelliJ for showing errors while using Lombok (if it can't find explicit Constructor and such)
    - provides refactoring options to add/remove Lombok Annotations

## Hiding Helper Methods

- The only useful part of the **Entity Class** are its **Properties** (from a business logic perspective).
- Helper Methods (like setters and getters) are just part of the plumbing/implementation so that Entity Class could fulfill its role to store some data/properties. Since these helper methods are the same for all Entities (and therefore do not provide any useful information) they are obfuscating true purpose of Entity Class. Therefore it is useful to hide helper methods (so that we can focus only on the Properties as we move between Entities).
- For that purpose we can use **Lombok API** which contains Annotations that inject helper Methods into Entity Class. This way helper Methods are no longer visible inside the Entity Class (allowing us to ignore them and focus on Properties).

## Annotations

- @Data** Annotation is most useful since it generates typical boilerplate code for Entities.  
@Data is used as shorthand for: @Getter, @Setter, @ToString, @EqualsAndHashCode, @RequiredArgsConstructor.  
But you can still explicitly add any of these if you need to specify additional parameters (fine tune Annotation).

### @Data Annotations

ANNOTATION	DESCRIPTION
<b>@Data</b>	<ul style="list-style-type: none"><li>@Getter, @Setter, @ToString, @EqualsAndHashCode, @RequiredArgsConstructor (Constructor is not generated if one already exists)</li></ul>
<b>@Getter</b>	<ul style="list-style-type: none"><li>Creates getter Methods for all Properties</li></ul>
<b>@Setter</b>	<ul style="list-style-type: none"><li>Creates setter Methods for all non-final Properties</li></ul>
<b>@ToString</b>	<ul style="list-style-type: none"><li>Creates toString Method from Class Name and Properties<ul style="list-style-type: none"><li>Optional parameter to include Property's name</li><li>Optional parameter to call super toString Method</li></ul></li></ul>
<b>@EqualsAndHashCode</b>	<ul style="list-style-type: none"><li>Creates equals() and hashCode() Methods</li></ul>
<b>@RequiredArgsConstructor</b>	<ul style="list-style-type: none"><li>Generate Constructor for final and @NonNull Properties</li></ul>

### Additional Annotations

ANNOTATION	DESCRIPTION
<b>@NoArgsConstructor</b>	<ul style="list-style-type: none"><li>Generate Constructor with no parameters (Throws error if there are final Properties)</li></ul>
<b>@AllArgsConstructor</b>	<ul style="list-style-type: none"><li>Generate Constructor for all Properties (@NonNull Properties are checked)</li></ul>

## 4.1.1 Lombok API - Use

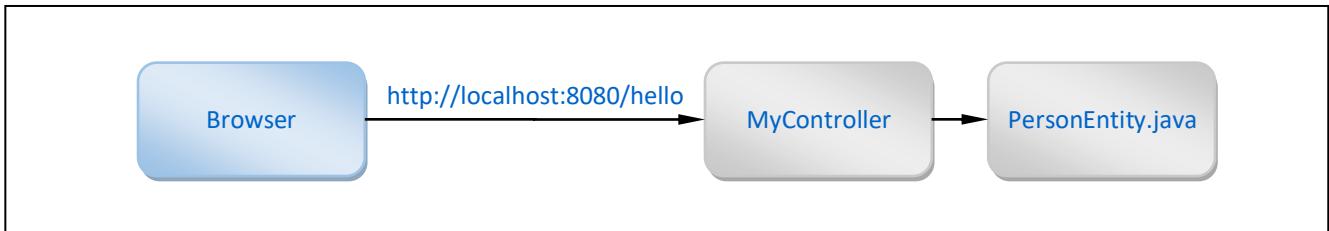
### Info

[G]

- This tutorial shows how to include Lombok API in your project by selecting its [Spring Boot Starter](#).
- Alternatively you can add Maven dependency to [your pom.xml](#).
- This tutorial is exactly the same as [Entity - @Data \(Lombok\)](#).

*Application Schema*

[[Results](#)]



*Spring Boot Starters*

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables @RequestMapping. Includes Tomcat Server.
Developer Tools	<b>Lombok</b>	Enables <b>@Data</b> which generate helper methods (setters, getters, ...)

*pom.xml*

```
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <optional>true</optional>
</dependency>
```

## Procedure

---

- Create Project entity\_lombok (add Spring Boot Starters from the table)
- Create Package: entities (inside main package)
- Create Java Class: PersonEntity.java (inside package entities)
- Create Package: controllers (inside main package)
- Create Java Class: MyController.java (inside package controllers)

### PersonEntity.java

```
package com.ivoronline.springboot.entity_lombok.entities;

import lombok.Data;
import org.springframework.stereotype.Component;

@Data
@Component
public class PersonEntity {
    private Long id;
    private String name;
    private Integer age;
}
```

### MyController.java

```
package com.ivoronline.springboot.entity_lombok.controllers;

import com.ivoronline.springboot.entity_lombok.entities.PersonEntity;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;

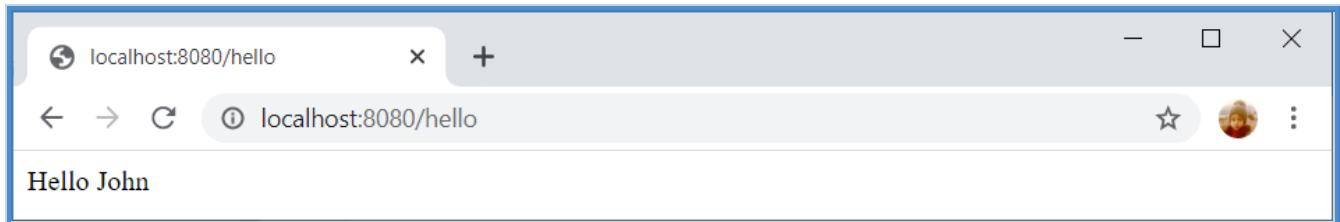
@Controller
public class MyController {

    @Autowired
    PersonEntity personEntity;

    @RequestBody
    @RequestMapping("/hello")
    public String sayHello() {
        personEntity.setName("John");
        String name = personEntity.getName();
        return "Hello " + name;
    }
}
```

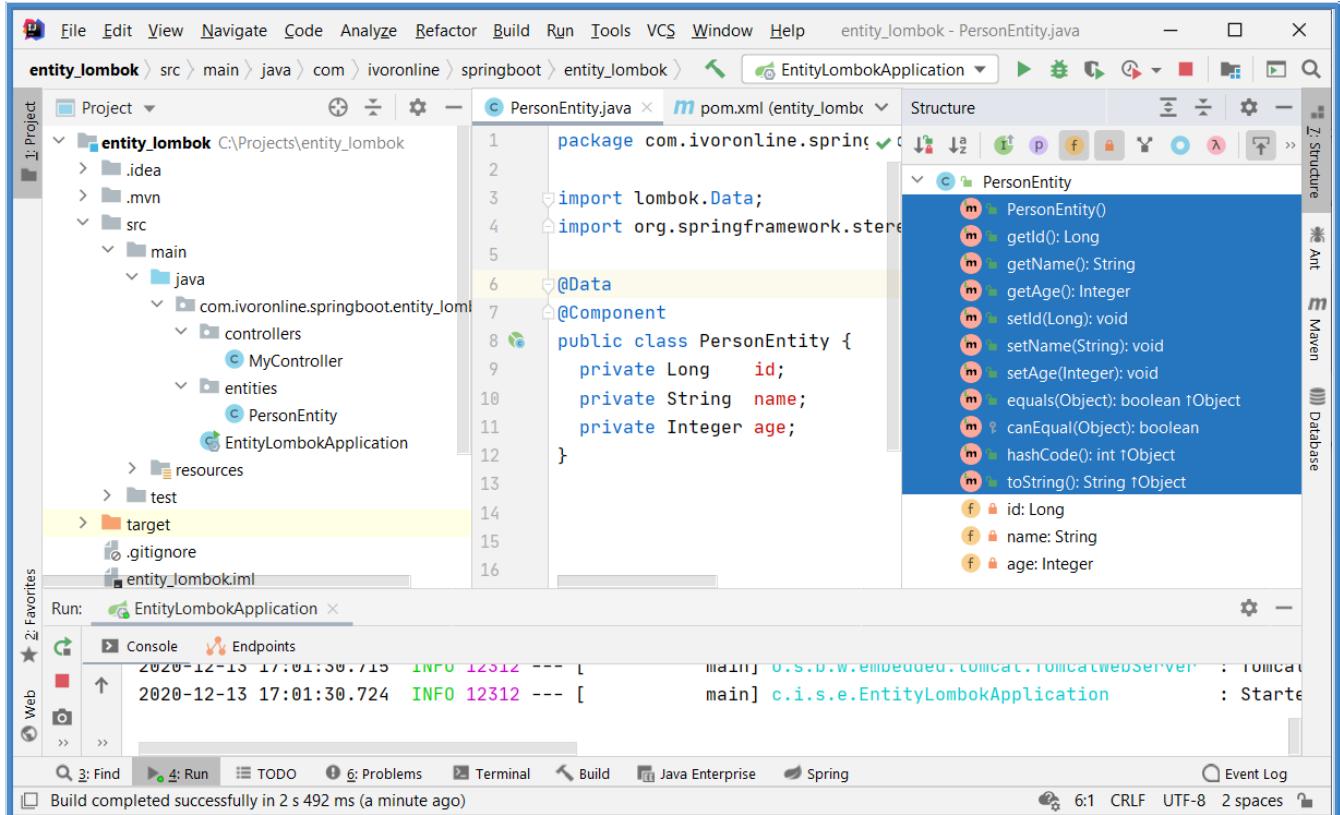
## Results

<http://localhost:8080/hello>



## Application Structure

(Lombok generated methods are shown under Structure View)



## pom.xml

```
<dependencies>

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

    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <optional>true</optional>
    </dependency>

</dependencies>
```

## 4.1.2 Lombok Plugin - Install for IntelliJ

### Info

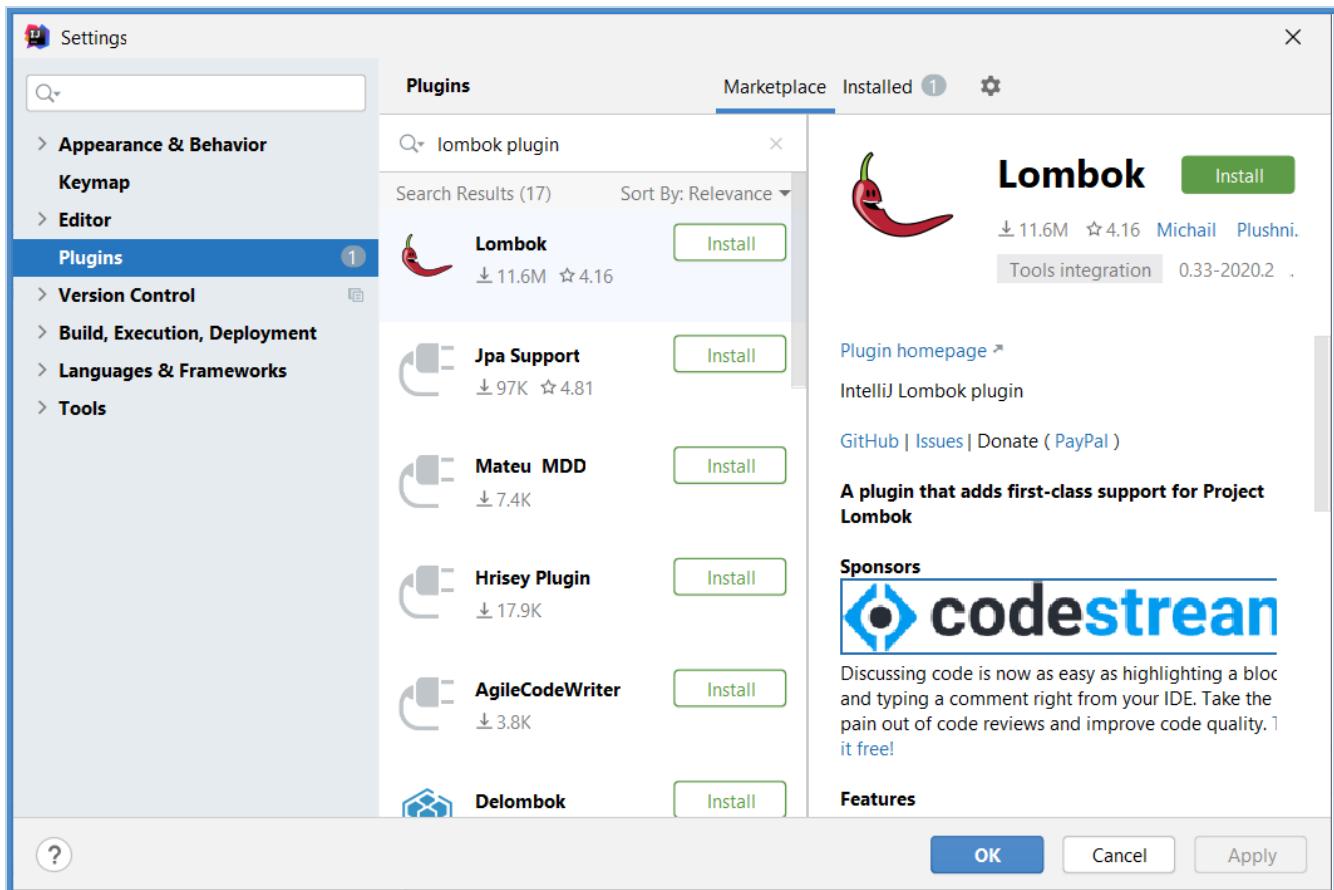
[R]

- This tutorial shows how to install Lombok **Plugin** for IntelliJ.

### Install Lombok Plugin

- Start IntelliJ
- File
- Settings
- Plugins
- Search: Lombok Plugin
- Install
- Enable Annotation Processing
- Restart IntelliJ

#### Install Lombok Plugin



#### Enable

**!** Lombok Requires Annotation Processing

Do you want to enable annotation processors? [Enable](#)

## 4.1.3 Lombok Plugin - Lombok

### Info

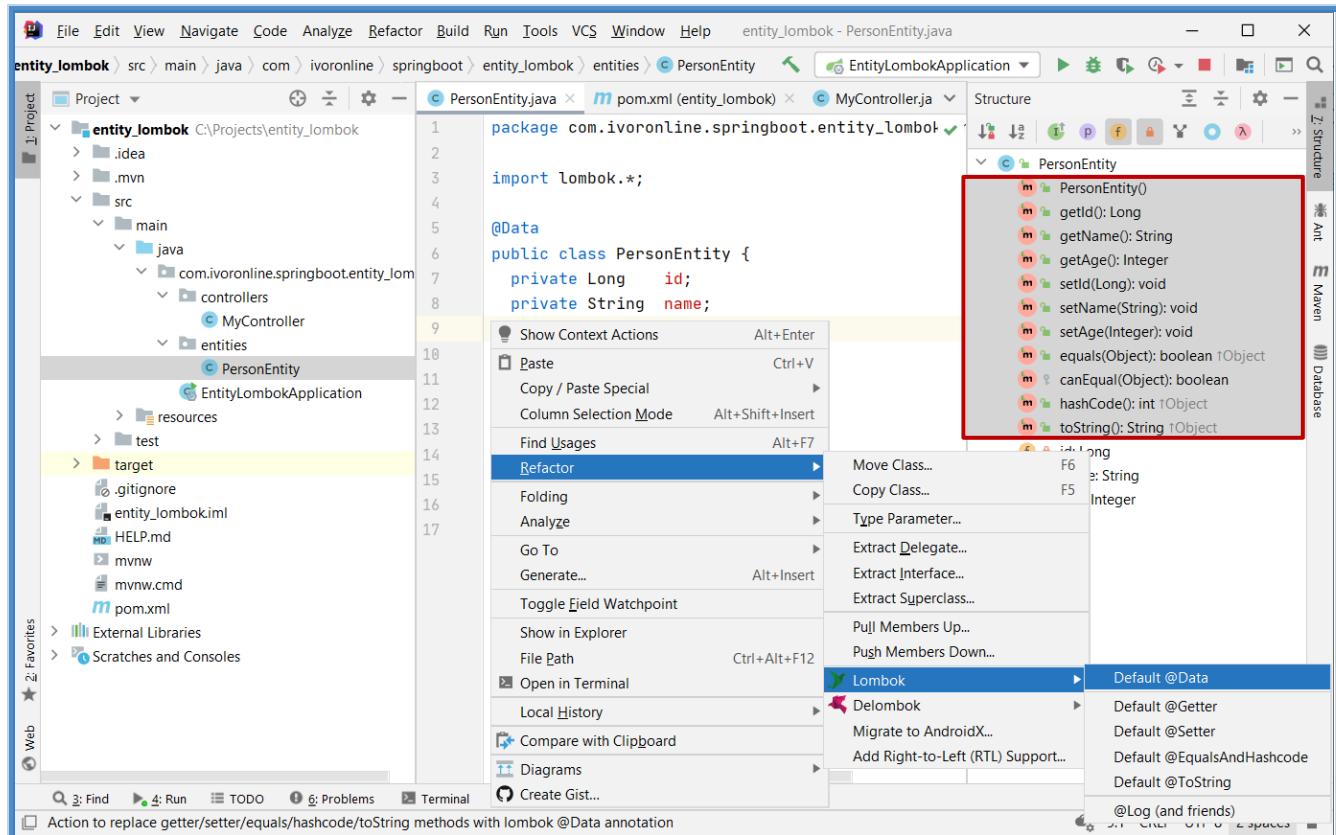
- This tutorial shows how to add Lombok Annotations through Lombok Plugin.
- Alternativly you can simply type these annotations manually in your code.
- Generated methods are visible in the Structure View: View - Tool Window - Structure
- You can use [Lombok Plugin - Delombok](#) to remove Lombok Annotations and insert these methods in the Class.

### Delombok

- RC inside Class
- Refactor
- Lombok
- Default @Data

#### Delombok @Data

(generated methods are visible under Structure tab)



#### PersonEntity.java

```
package com.ivoronline.springboot.entity_lombok.entities;

import lombok.*;

@Data
public class PersonEntity {
    private Long id;
    private String name;
    private Integer age;
}
```

## 4.1.4 Lombok Plugin - Delombok

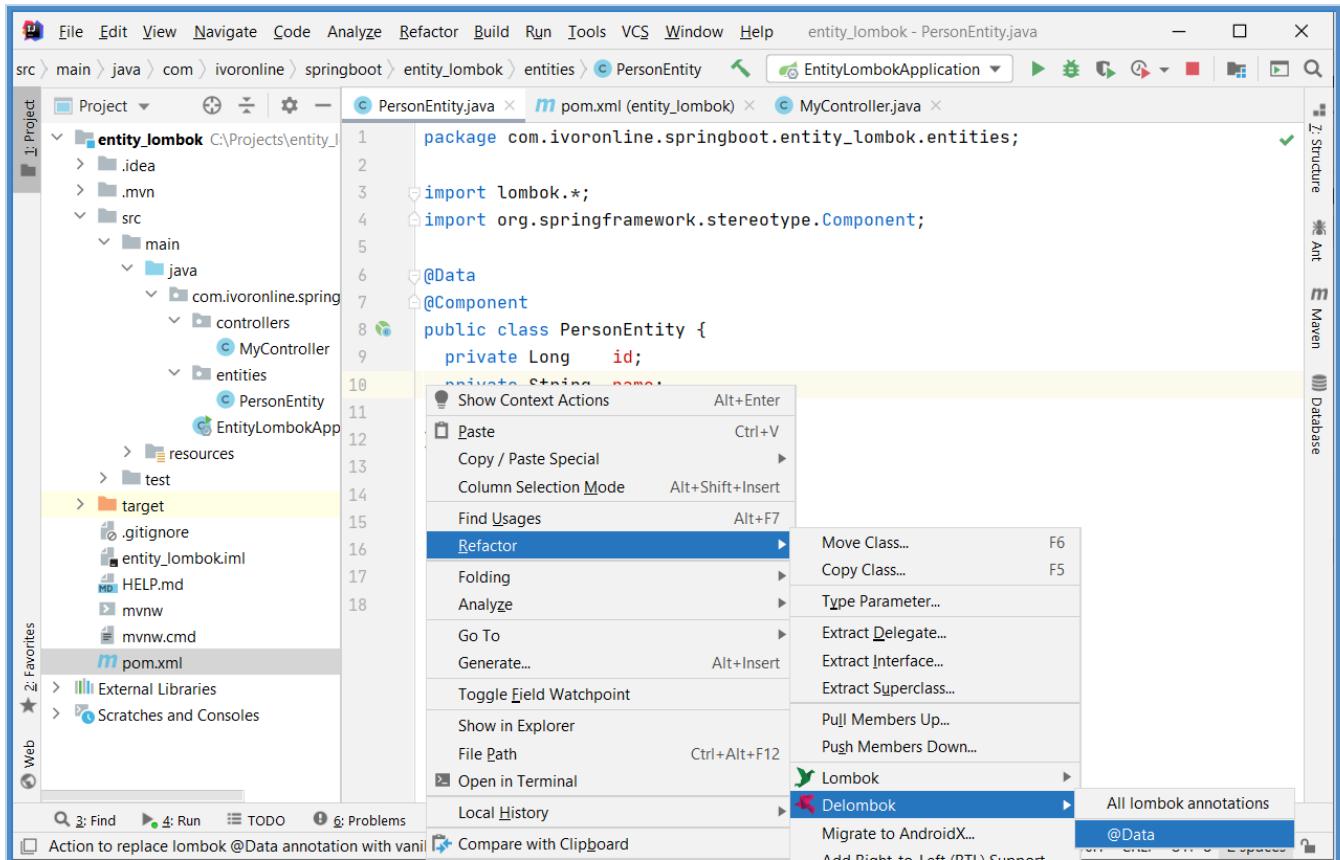
### Info

- Lombok Plugin can be used to Delombok your Class.
- This will remove selected Lombok Annotation and replace it with the Lombok generated code.
- You can use this approach to generate helper methods even if you don't want them to be hidden.

### Delombok

- RC inside Class
- Refactor
- Delombok
- @Data

#### Delombok @Data



#### PersonEntity.java

(Lomboked)

```
package com.ivoronline.springboot.entity_lombok.entities;

import lombok.*;

@Data
public class PersonEntity {
    private Long id;
    private String name;
    private Integer age;
}
```

```

package com.ivorononline.springboot.entity_lombok.entities;

public class PersonEntity {

    //PROPERTIES
    private Long id;
    private String name;
    private Integer age;

    //NO-ARG CONSTRUCTOR
    public PersonEntity() { }

    //GETTERS
    public Long getId () { return this.id; }
    public String getName() { return this.name; }
    public Integer getAge () { return this.age; }

    //SETTERS
    public void setId (Long id) { this.id = id; }
    public void setName(String name) { this.name = name; }
    public void setAge (Integer age) { this.age = age; }

    //TOSTRING
    public String toString() {
        return "PersonEntity(id=" + this.getId() + ", name=" + this.getName() + ", age=" + this.getAge() + ")";
    }

    //EQUALS
    public boolean equals(final Object o) {
        if (o == this) return true;
        if (!(o instanceof PersonEntity)) return false;
        final PersonEntity other = (PersonEntity) o;
        if (!other.canEqual((Object) this)) return false;
        final Object this$id = this.getId();
        final Object other$id = other.getId();
        if (this$id == null ? other$id != null : !this$id.equals(other$id)) return false;
        final Object this$name = this.getName();
        final Object other$name = other.getName();
        if (this$name == null ? other$name != null : !this$name.equals(other$name)) return false;
        final Object this$age = this.getAge();
        final Object other$age = other.getAge();
        if (this$age == null ? other$age != null : !this$age.equals(other$age)) return false;
        return true;
    }

    //CANEQUAL
    protected boolean canEqual(final Object other) { return other instanceof PersonEntity; }

    //HASHCODE
    public int hashCode() {
        final int PRIME = 59;
        int Results = 1;
        final Object $id = this.getId();
        Results = Results * PRIME + ($id == null ? 43 : $id.hashCode());
        final Object $name = this.getName();
        Results = Results * PRIME + ($name == null ? 43 : $name.hashCode());
        final Object $age = this.getAge();
        Results = Results * PRIME + ($age == null ? 43 : $age.hashCode());
        return Results;
    }
}

```

## 4.2 IntelliJ

### Info

---

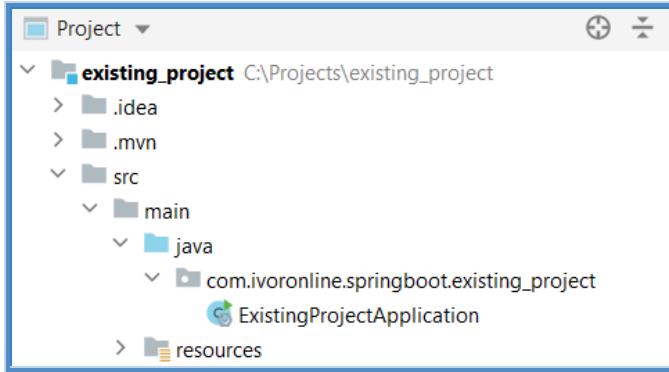
- Following tutorials show how to use IntelliJ to perform some useful tasks.

## 4.2.1 Copy Project

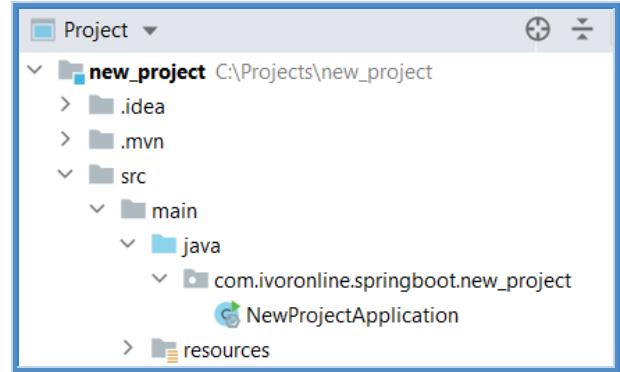
### Info

- This tutorial shows how to copy IntelliJ Project (when you want to use existing Project as starting point for new Project).
  - From Existing Project - Copy src Directory (existing\_project/src)
  - Create New Project (new\_project)
  - In New Project - Replace src Directory (delete new\_project/src, paste copied src)
  - In New Project - Rename main Package (com.ivoronline.springboot.new\_project)
  - In New Project - Rename main Class (**NewProjectApplication.java**)

*existing\_project*



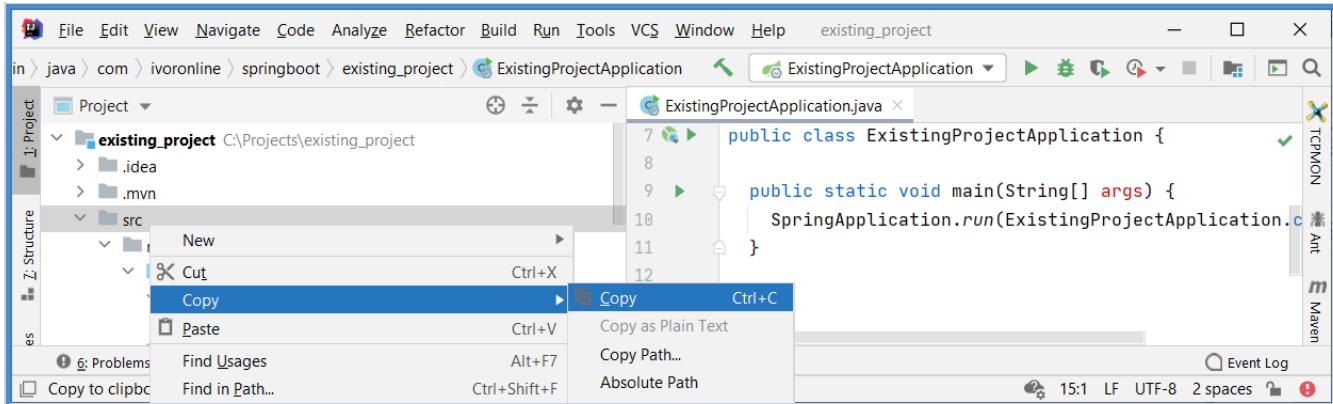
*new\_project*



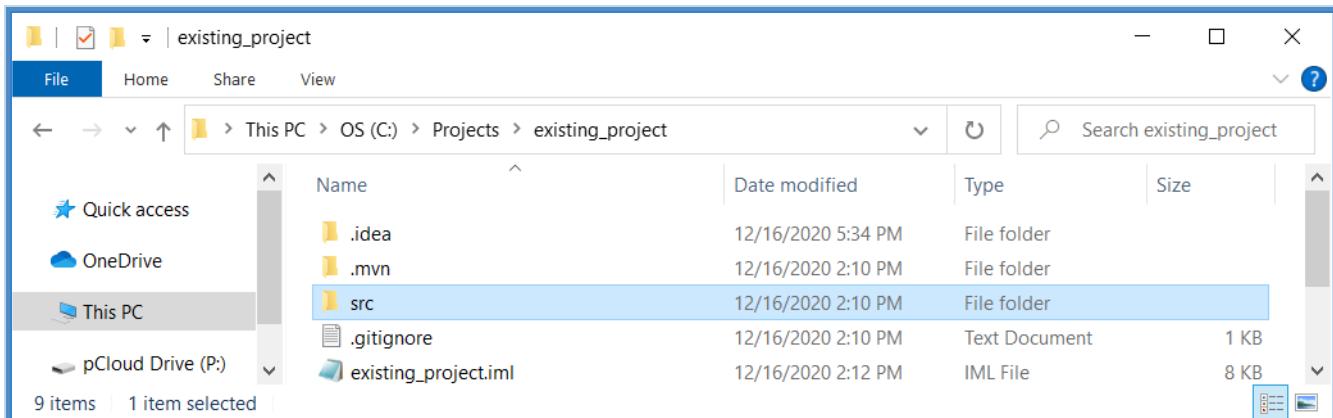
### From Existing Project - Copy src Directory

- RC on [src](#)
- Copy

*Copy using IntelliJ*



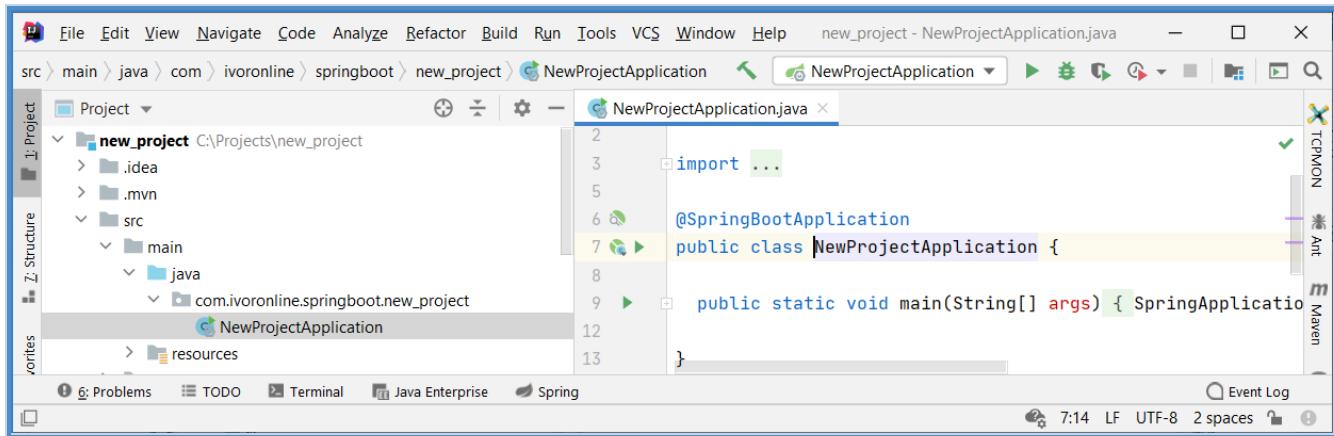
*Copy using Windows Explorer*



## Create New Project

- **Create Spring Boot Project:** new\_project (add Spring Boot Starters for the existing and new Project)

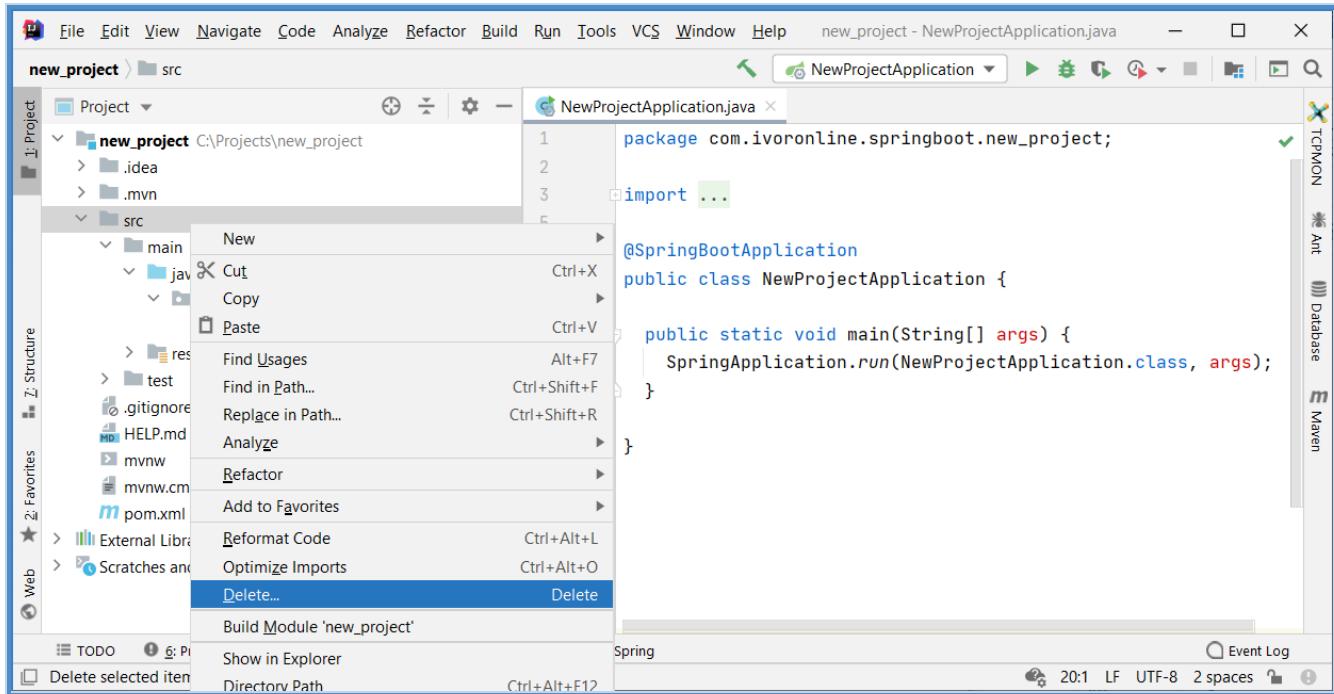
## *New Project*



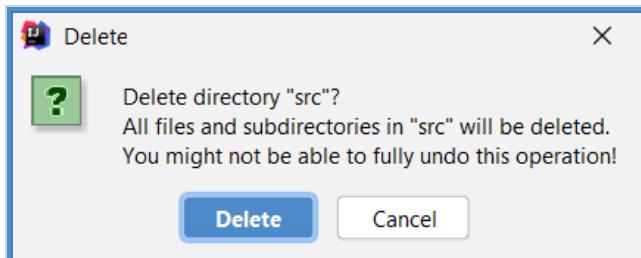
## In New Project - Replace src Directory

- Delete `src` Directory
  - Paste `src` Directory

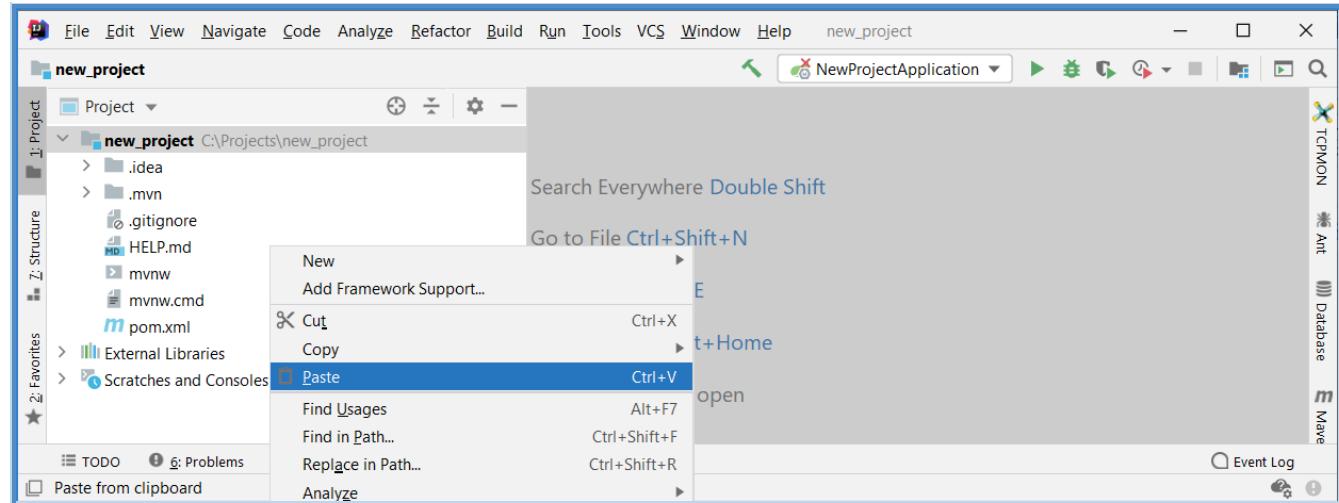
## *Delete `src` Directory*



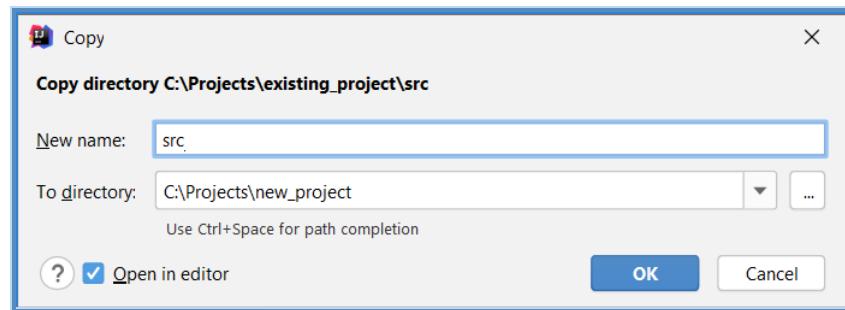
## *Delete*



## Paste `src` Directory



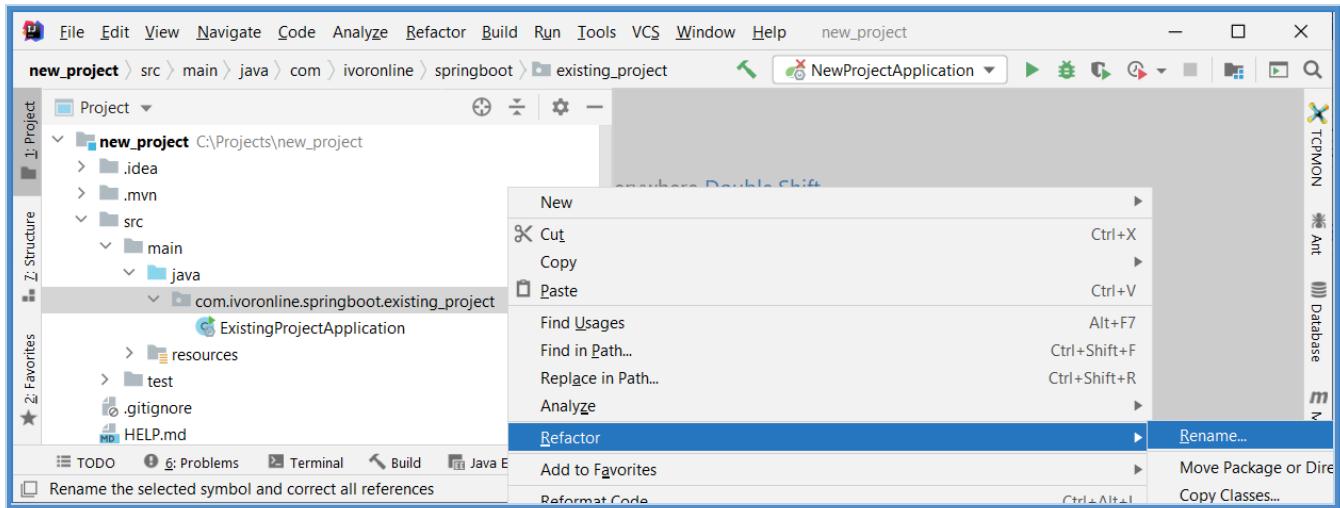
## Paste `src` Directory



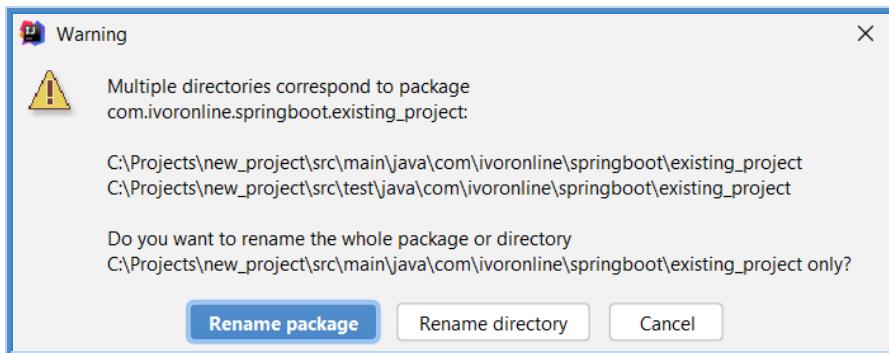
## In New Project - Rename main Package & Class

- RC on Package
- Refactor
- Rename
- Rename Package
- (enter new name)
- Refactor

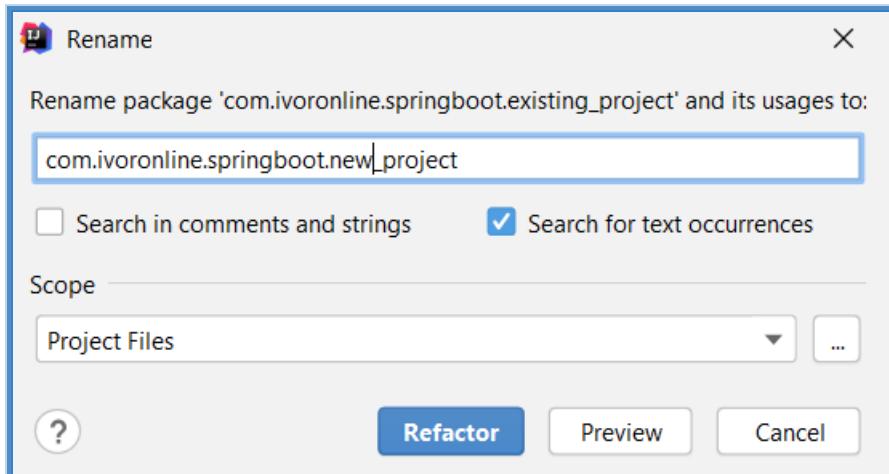
### Rename Package



### Rename Package



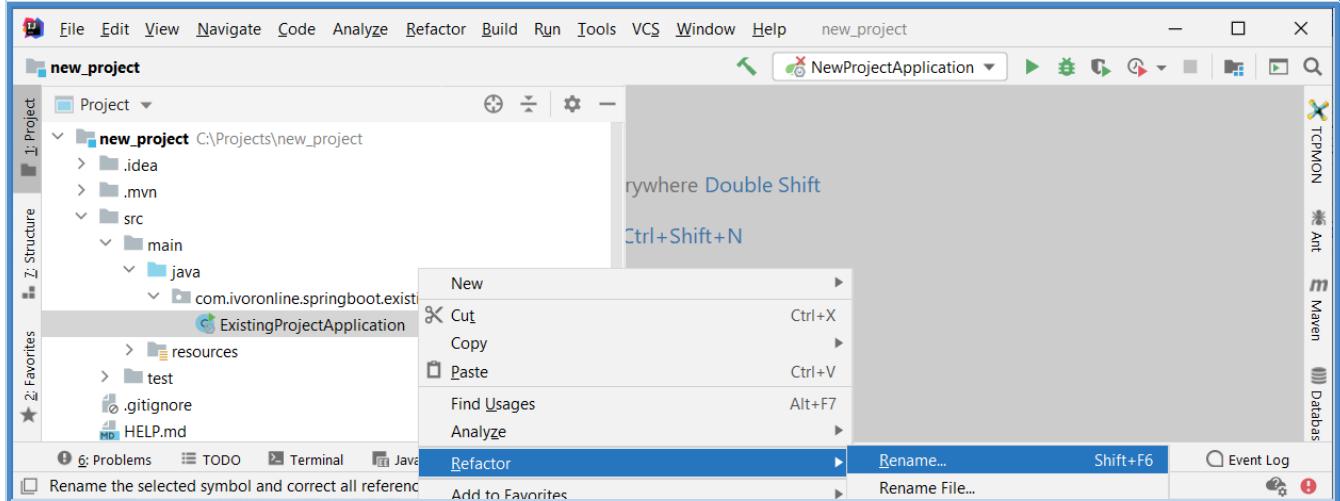
### New Package Name



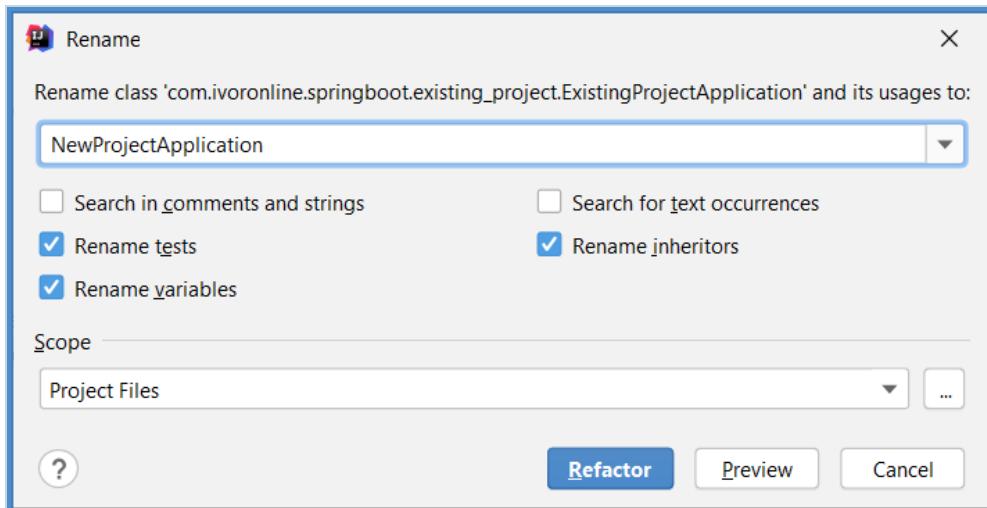
## In New Project - Rename main Class

- RC on ExistingProjectApplication.java
- Refactor
- Rename
- (enter new name)
- Refactor

### Rename Class



### Rename Class



## 4.2.2 DB Tools

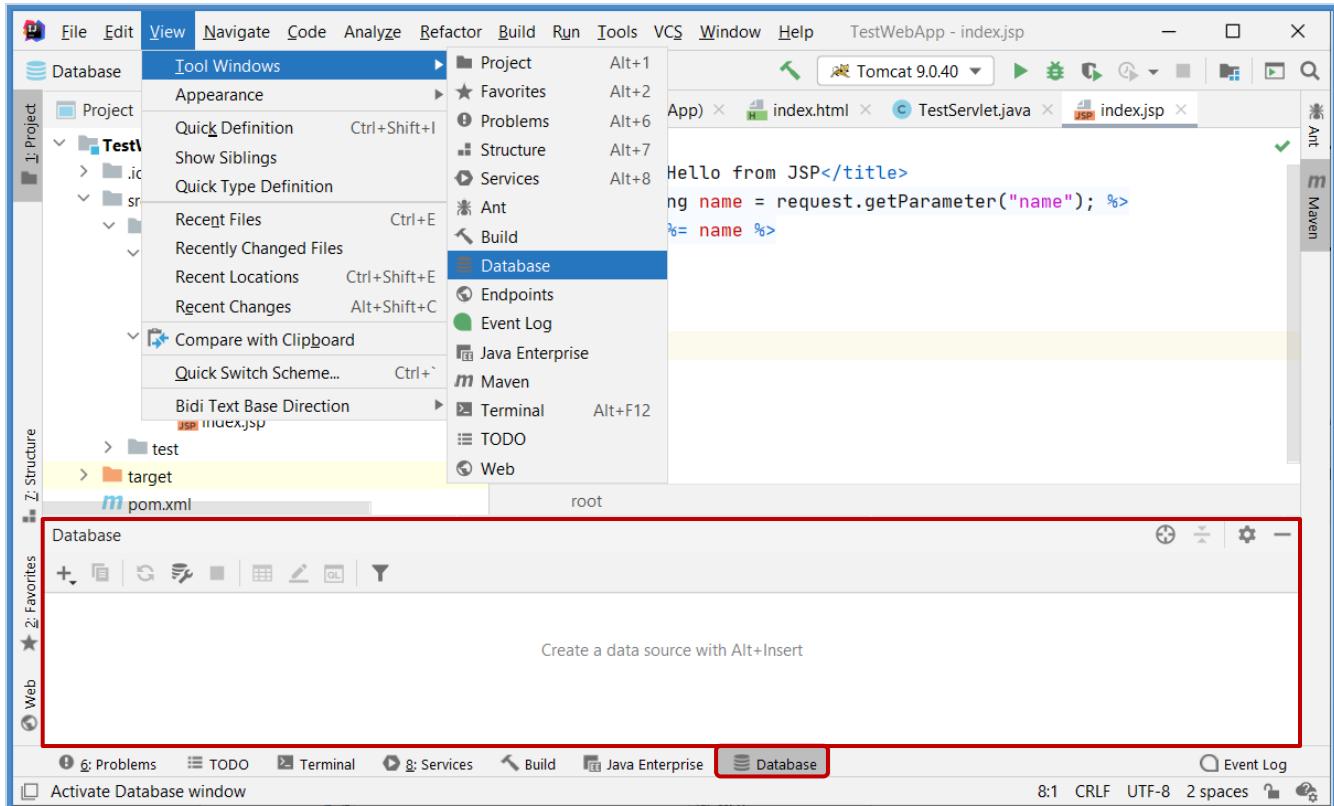
### Info

- This tutorial shows how to use DB Tools Window to query Database.

### Open DB Tools Window

- View
- Tool Windows
- Database

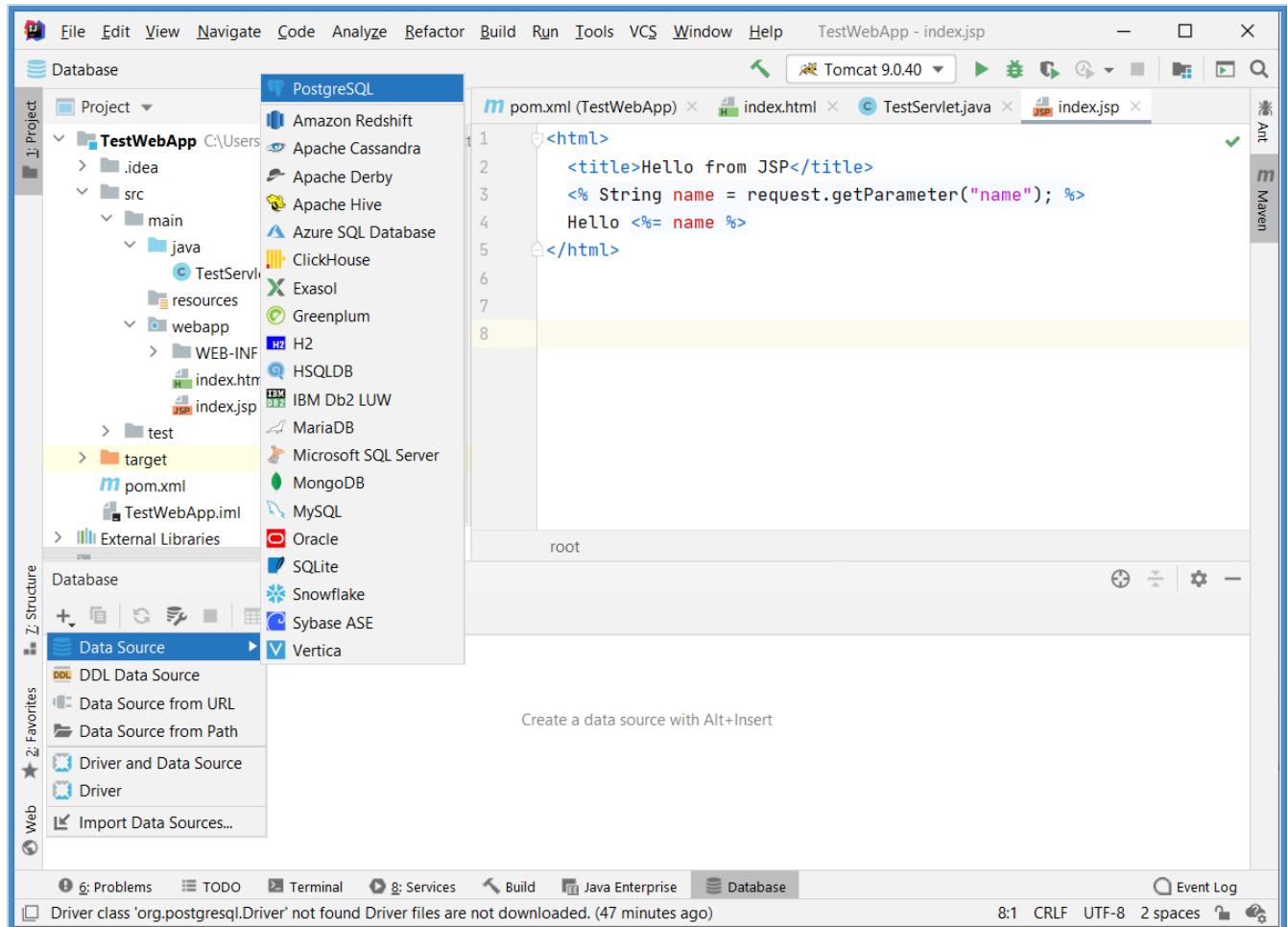
#### Database



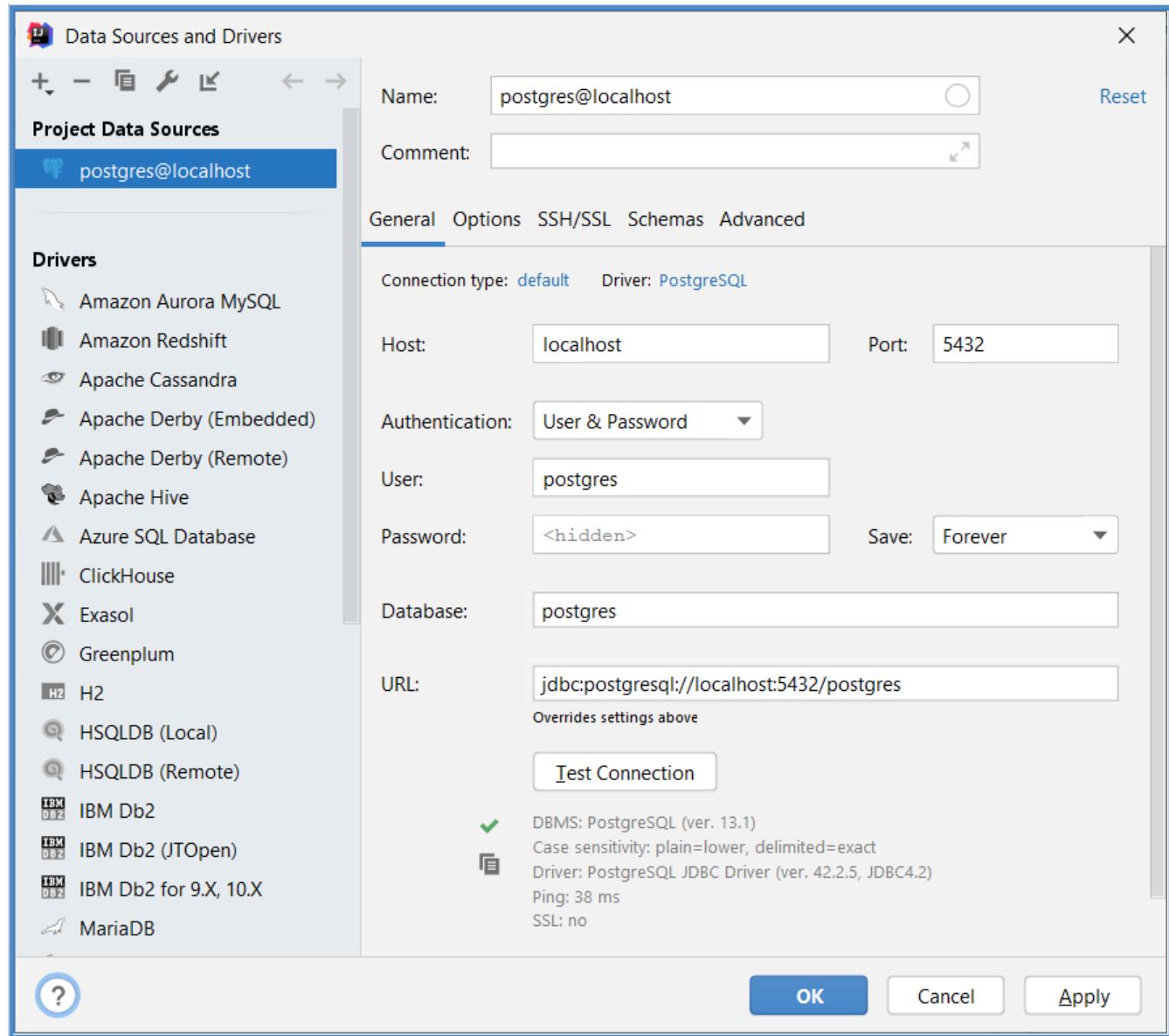
## Create connection to DB

- New - Data Source - PostgreSQL
- Download missing drivers (link at the bottom)
- User: postgres
- Password: B...5
- Test Connection
- OK

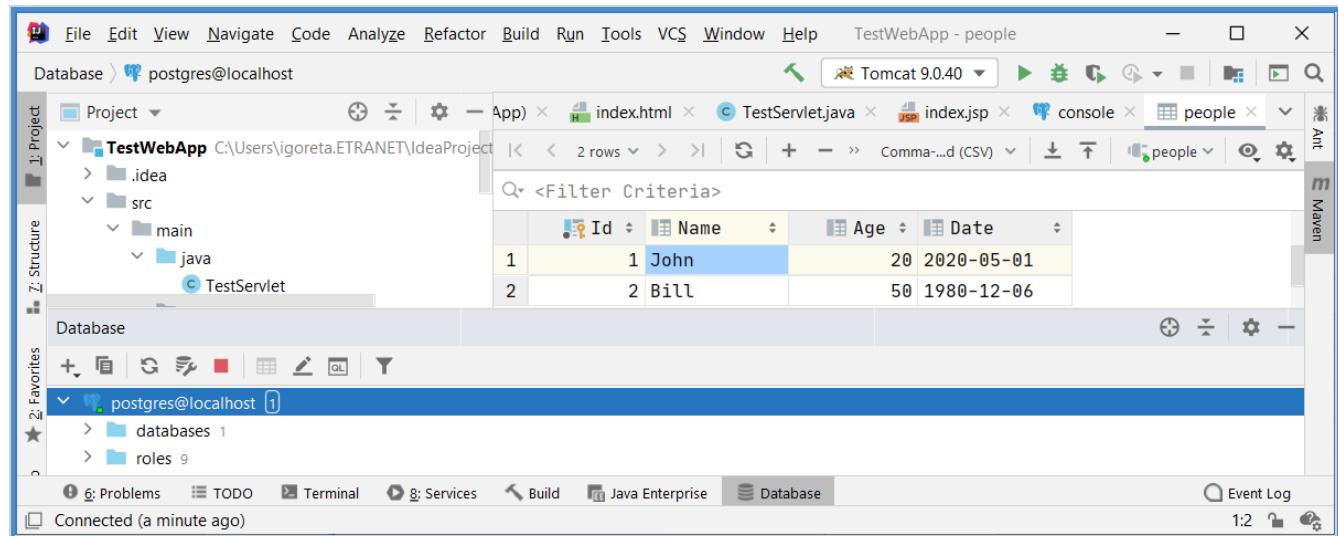
### New - Data Source



### Enter connection details



### Created Connection



## 4.3 IntelliJ - Developer Tools

### Info

[R] [R]

- This tutorial shows how to install and use Developer Tools API together with Chrome Live Reload Extension.
- This will automatically reload your application and refresh the browser when you either
  - Switch to Browser (just selecting Browser Window should be enough to trigger application reload)
  - Refresh Browser (both Console and Browser will get refreshed)
  - File - Save All (might wait a second for application to reload in the browser)

## 4.3.1 Install Chrome Extension: Live Reload

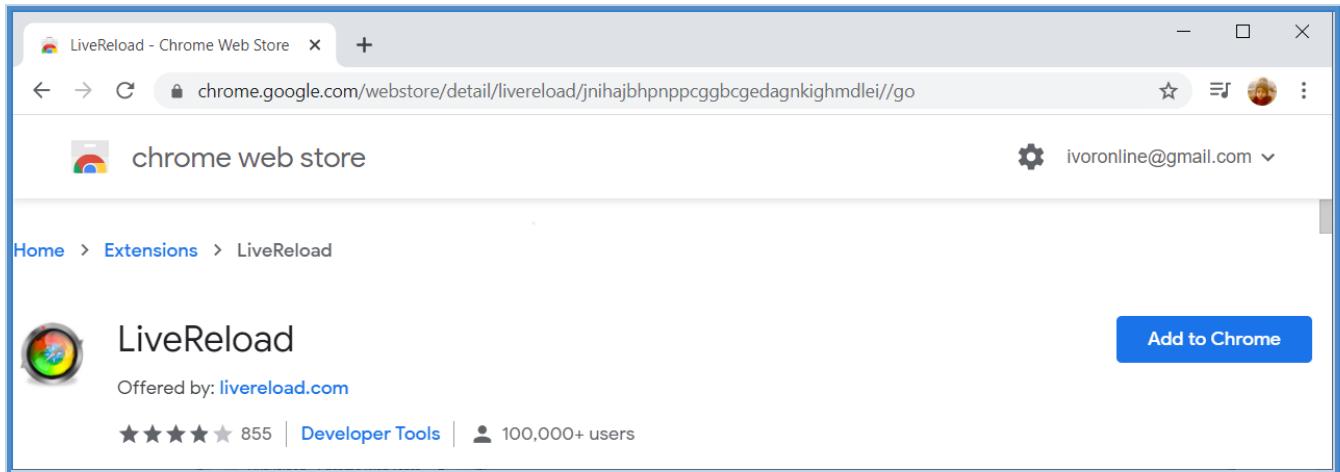
### Info

- This tutorial shows how to Install Chrome Extension Live Reload.

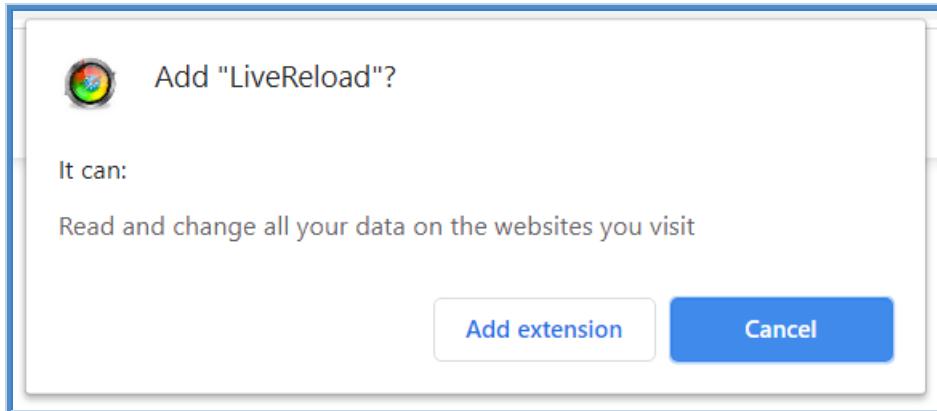
### Install Live Reload

- <https://chrome.google.com/webstore/detail/livereload/jnihajbpnppcggbcgedagnkighmdlei//go>
- Add to Chrome

#### Live Reload



#### Add extension



## 4.3.2 IntelliJ Registry: compiler.automake

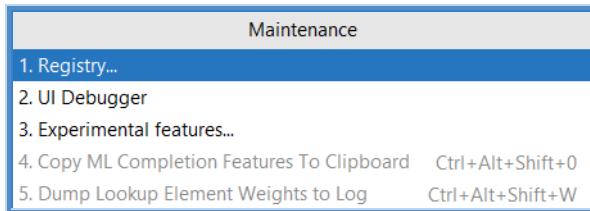
### Info

- This tutorial shows how to Edit IntelliJ Registry to support automatic compile.
- You only have to do this once and it will apply to every Project.

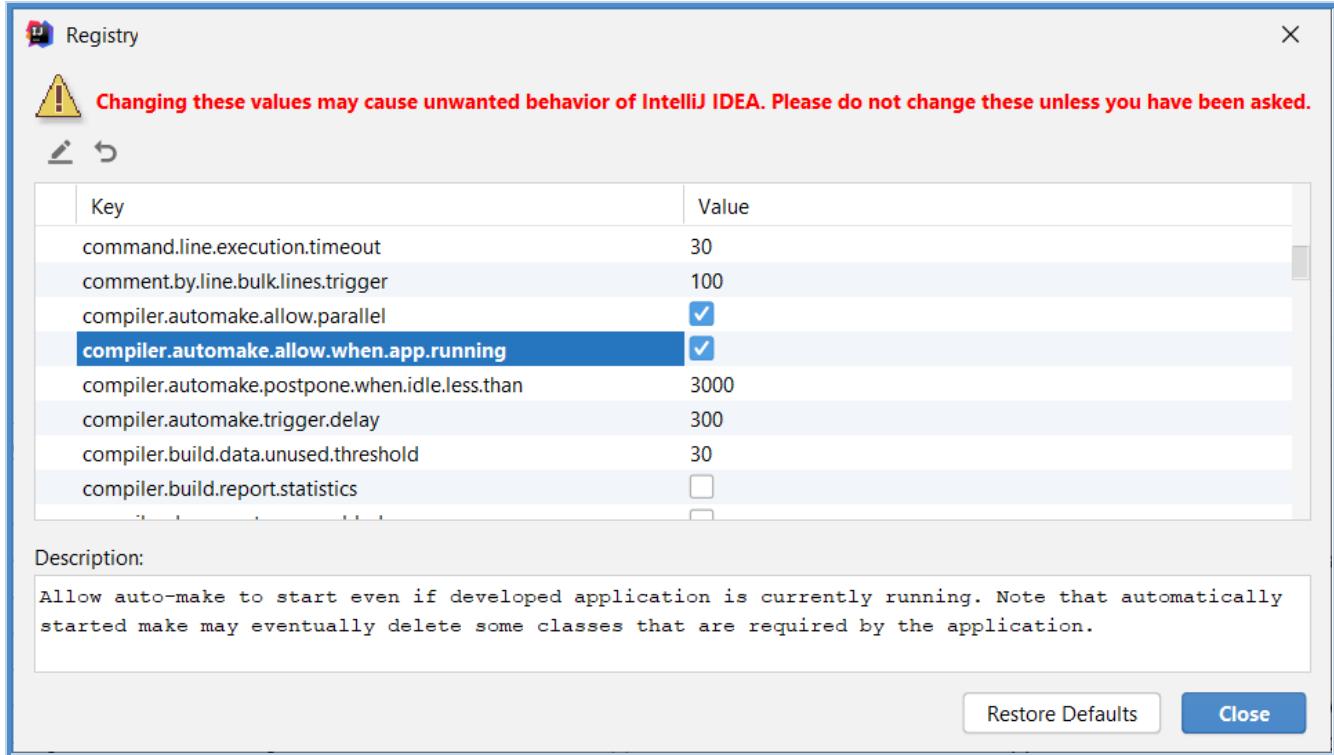
### Edit IntelliJ Registry

- <https://stackoverflow.com/questions/28415695/how-do-you-set-a-value-in-the-intellij-registry>
- Ctrl + Shift + Alt + /
- Registry...
- Start typing: automake (Search for)
- compiler.automake.allow.when.app.running: CHECK
- Close

Registry...



IntelliJ Registry



## 4.3.3 Run Configuration: Update classes and resources

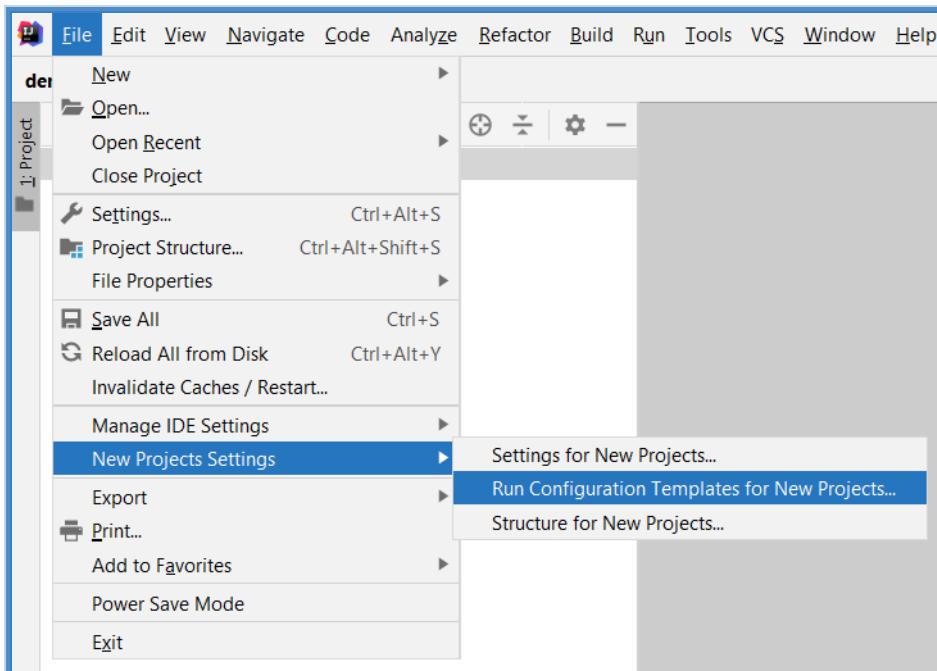
### Info

- This tutorial shows how to edit Run Configuration to Update classes and resources either for
  - For all Projects
  - For current Project

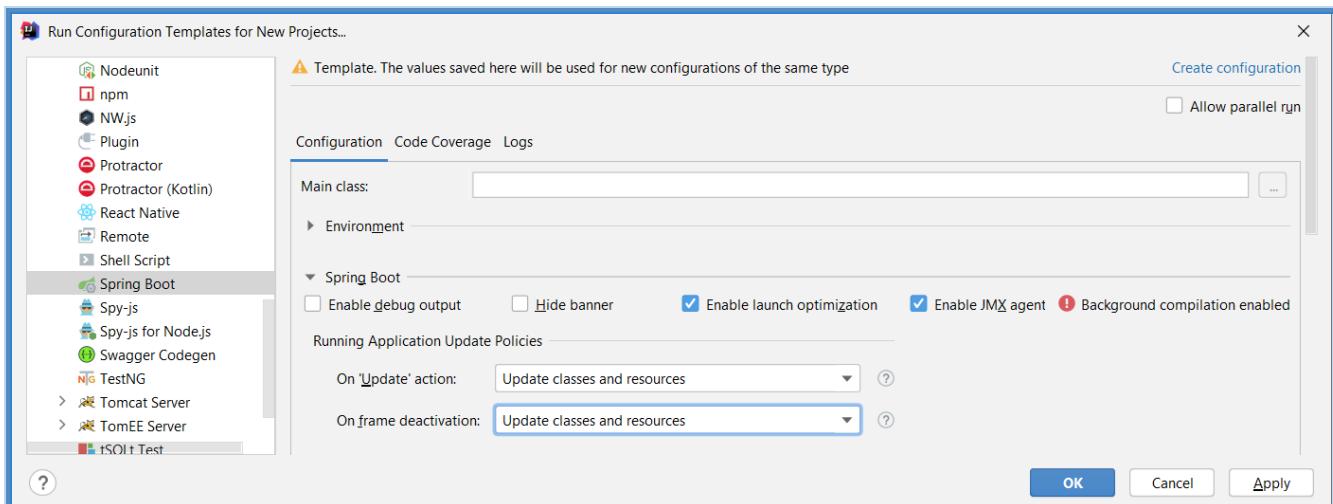
### For all Projects

- File - **New Projects Settings** - Run Configuration Templates for New Projects
- Templates: **Spring Boot**
- Expand: Spring Boot
- On Update action: Update classes and resources
- On frame deactivation: Update classes and resources
- OK

#### Run Configuration Templates for New Projects



#### Spring Boot Template

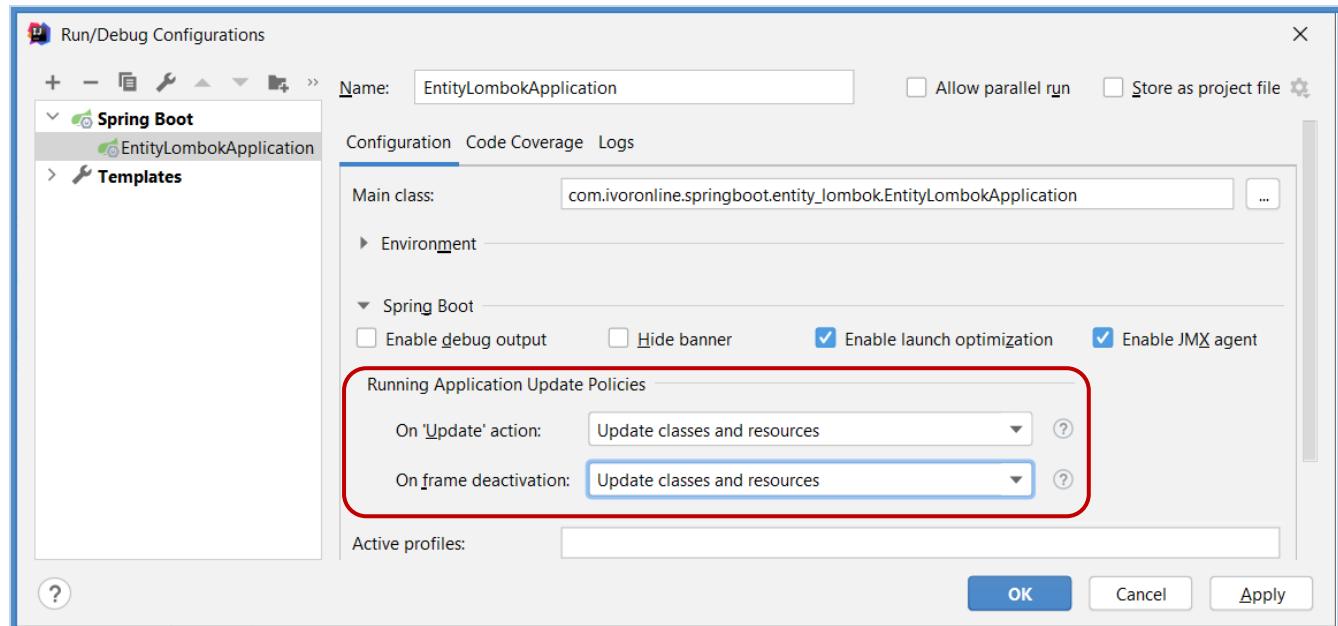


## For current Project

---

- Run
- Edit Configurations
- Expand: Spring Boot
- (Select Run Configuration)
- Expand: Spring Boot
- On Update action:      Update classes and resources
- On frame deactivation: Update classes and resources
- OK

### *Update classes and resources*



## 4.3.4 Settings: Build Project Automatically

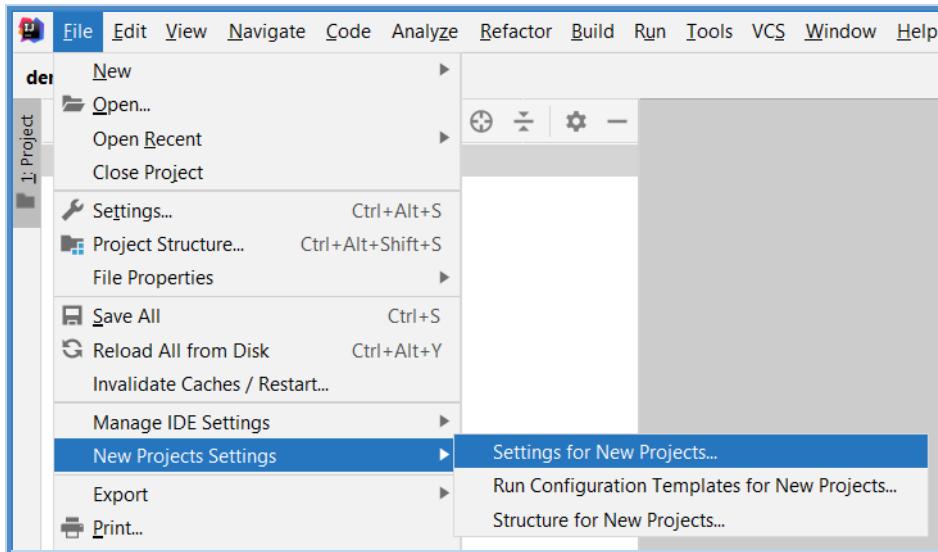
### Info

- This tutorial shows how to edit Project Settings to build Project Automatically either for
  - For all Projects
  - For current Project

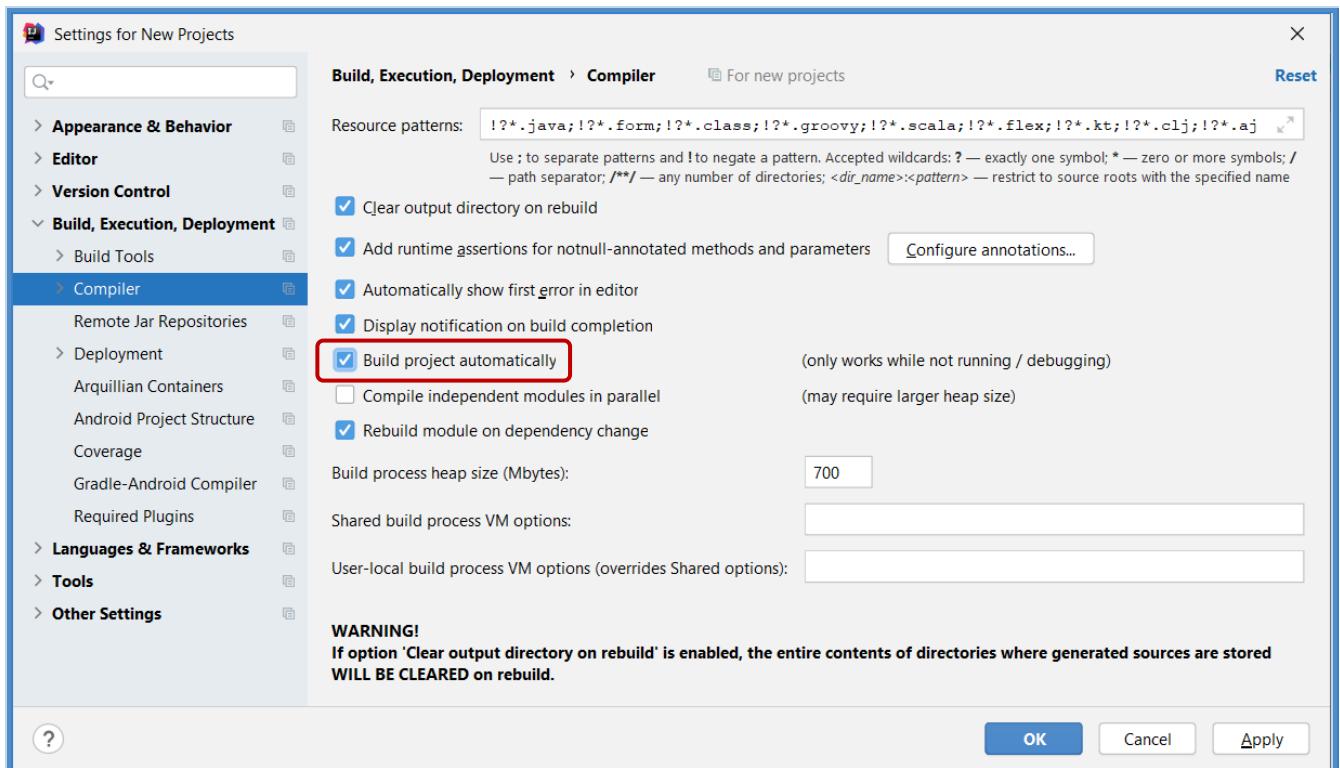
### For all Projects

- File - **New Projects Settings** - Settings for New Projects
- Build, Execution, Deployment - **Compiler**
- Build project automatically: CHCK
- OK

#### Settings for New Projects



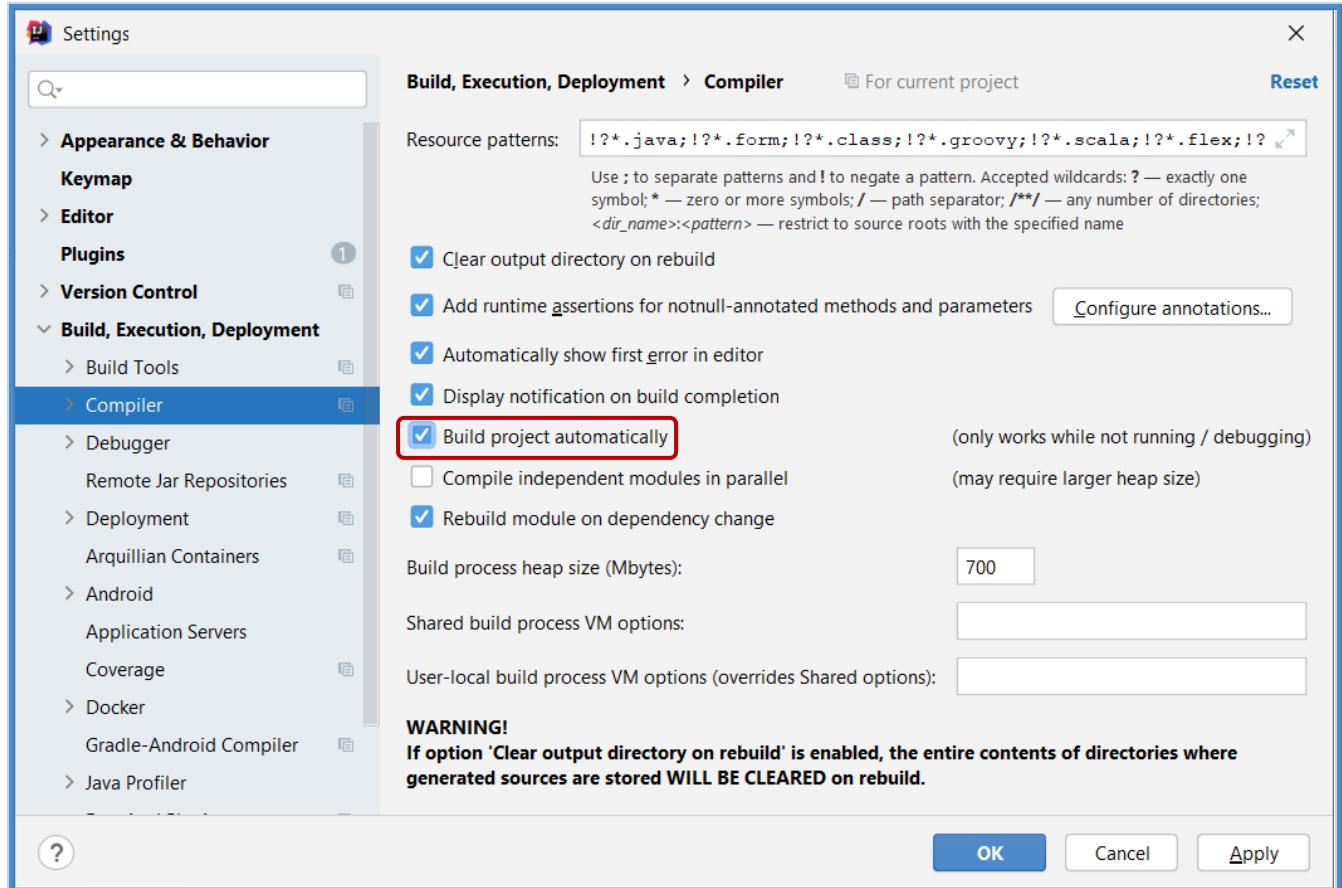
#### Build project automatically



## For current Project

- File
- Settings
- Build, Execution, Deployment
- Compiler
- Build project automatically: CHCK
- OK

### Build project automatically



## 4.3.5 Create Spring Boot Project

### Project

- This tutorial shows how to create Spring Boot Project to test functionality of Developer Tools and Live Reload.

### Create Project

- [Create Spring Boot Project](#) (add Spring Boot Starters from the table)
- Create Java Class: MyController.java (inside main package)
- Add highlighted line to ConsoleApplication.java

#### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables @RequestMapping. Includes Tomcat Server.
Developer Tools	<b>Spring Boot DevTools</b>	Enables auto reloading application

#### MyController.java

```
package com.ivoronline.springboot.developer_tools;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class MyController {

    @RequestMapping("/hello")
    @ResponseBody
    public String sayHello() {
        return "Message from Controller";
    }

}
```

#### ConsoleApplication.java

```
package com.ivoronline.console_application;

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

@SpringBootApplication
public class ConsoleApplication {

    public static void main(String[] args) {
        SpringApplication.run(ConsoleApplication.class, args);
        System.out.println("Hello from ConsoleApplication");
    }

}
```

## Test

- (add "CHANGED" to highlighted lines inside MyController.java and ConsoleApplication.java)
- Choose one (to reload application)
  - Switch to Browser
  - Refresh Browser
  - File - Save All

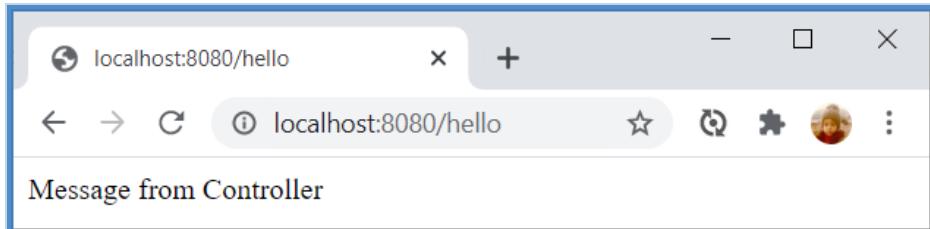
MyController.java

```
return "Message from Controller CHANGED";
```

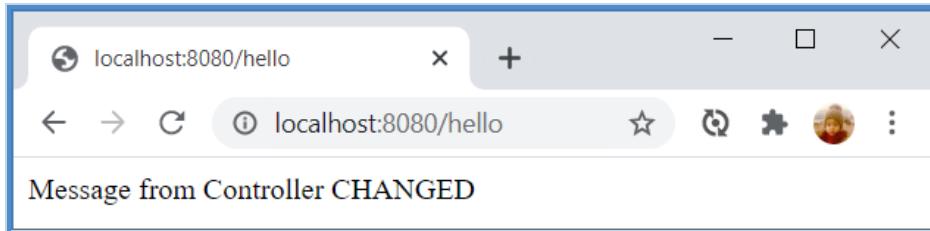
ConsoleApplication.java

```
System.out.println("Hello from ConsoleApplication CHANGED");
```

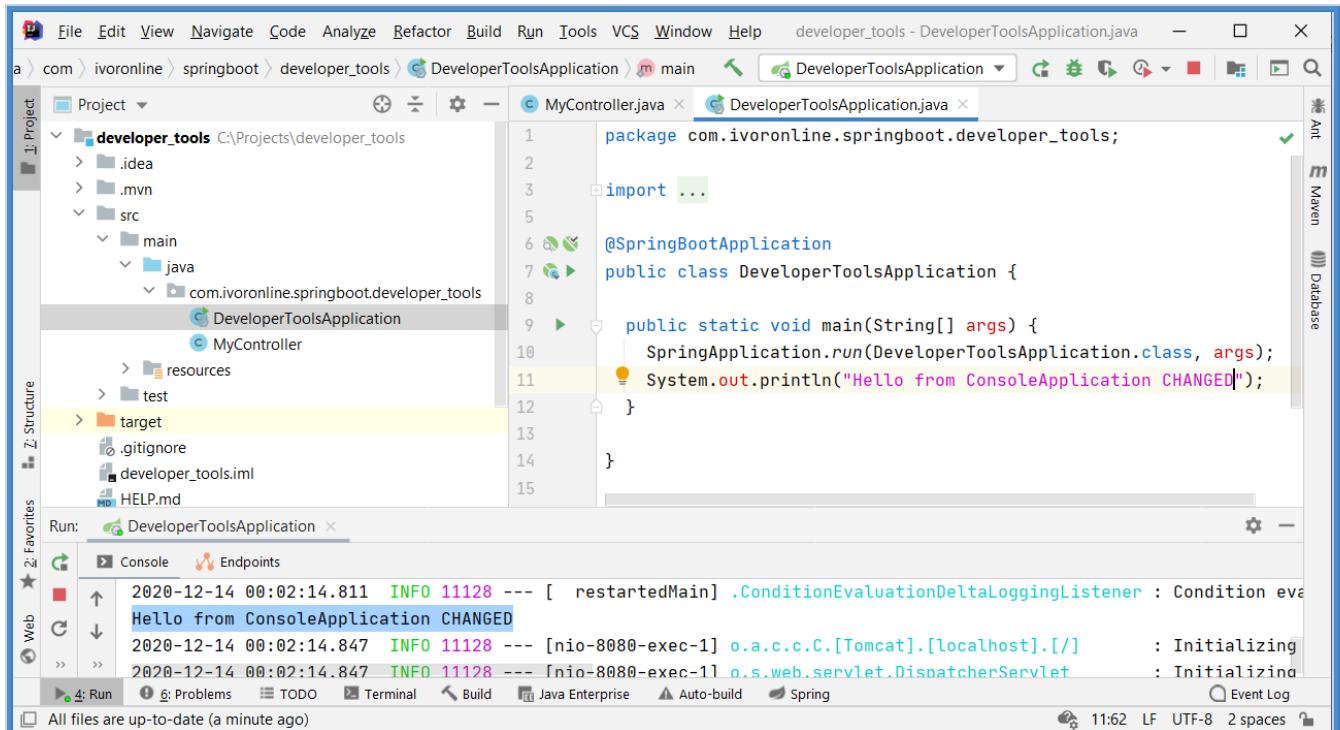
<http://localhost:8080/hello>



Just switching to Browser reloads application



Application Structure



*pom.xml*

```
<dependencies>

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

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
    <optional>true</optional>
</dependency>

</dependencies>
```

# 4.4 Database - H2

## Info

- Following tutorials show how to use H2 Database.  
H2 DB is **in memory** database (it gets created when application starts and it gets destroyed when application stops).
- To include H2 DB in the Application, select Spring Starters H2 Database (which adds Maven dependency to pom.xml).  
If you also include Spring Starter **JPA**, JPA will detect H2 DB and automatically configure application to use it.  
When application starts, inside the Console you will get information that JPA created Database (as shown below).
- By default, every time application starts, new Database is created (with new name displayed in Console as shown below).  
But you can **Specify DB Name** through **application.properties** (as shown below so that it remains the same).  
Database will still get destroyed every time application stops by you can now use the same string to connect to it.
- You can access H2 DB through its **Web Console** at [/h2-console](#) after **Enabling Console** through **application.properties**.
- Application Example** shows how to create Person Entity and then storing it into H2 DB as Record.

## Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables @Controller and @RequestMapping. Includes Tomcat Server.
SQL	Spring Data JPA	Enables @Entity and @Id
SQL	H2 Database	Enables in-memory H2 Database

pom.xml

(Spring Starters: H2, JPA)

```
<dependencies>

    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
    </dependency>

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

</dependencies>
```

application.properties

```
spring.h2.console.enabled = true
spring.datasource.url      = jdbc:h2:mem:testdb
```

Spring Console

(name of created database)

```
H2 console available at '/h2-console'.
Database available at 'jdbc:h2:mem:testdb'
Database available at 'jdbc:h2:mem:0054f1b1-1fda-45be-be3e-24110a28c109' //From application.properties
                                                               //Autogenerated
```

## H2 Console

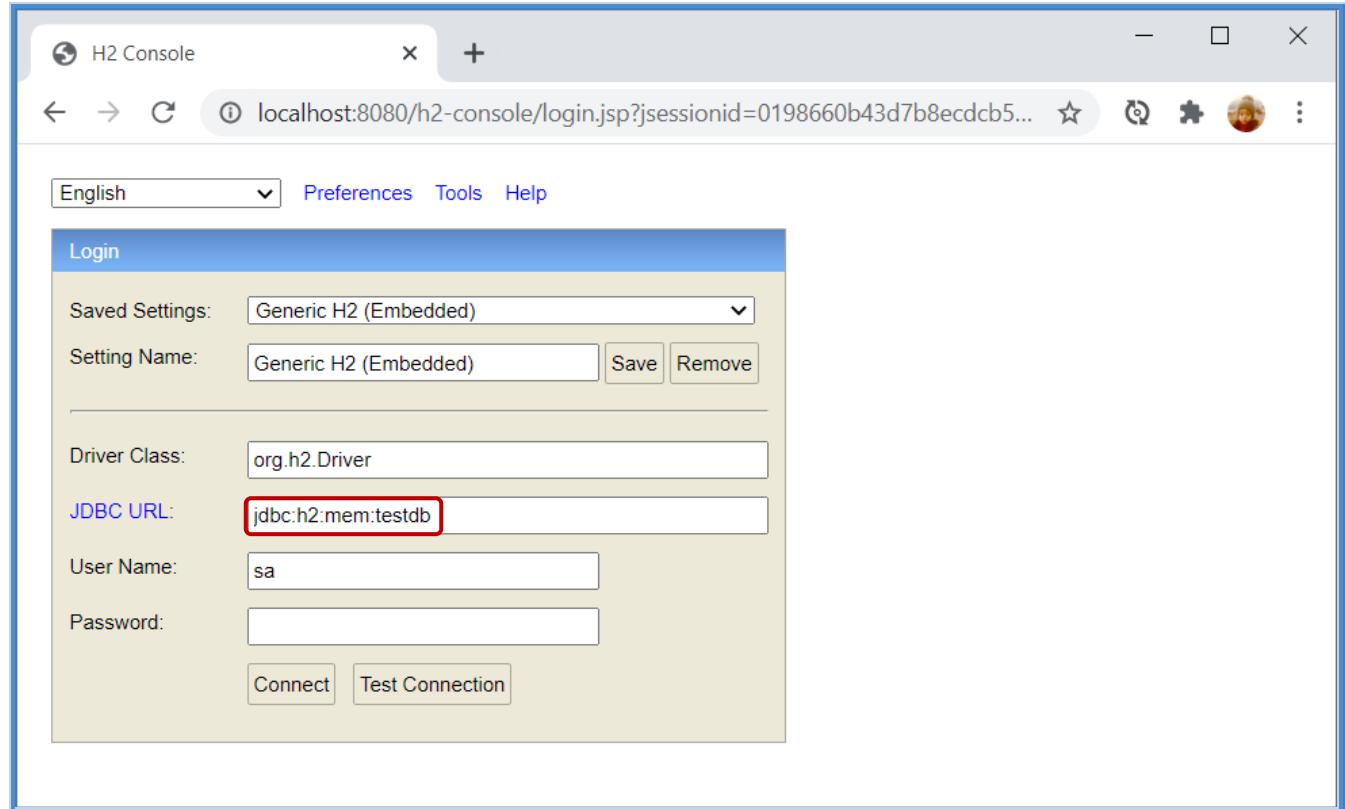
- You can access H2 DB through its [Web Console](#) at `/h2-console` after Enabling Console through `application.properties`.

`application.properties`

```
spring.h2.console.enabled = true  
spring.datasource.url      = jdbc:h2:mem:testdb
```

*H2 DB Web Console (Client)*

(<http://localhost:8080/h2-console>)



*Show Table Records*

(`SELECT * FROM PERSON_ENTITY`)



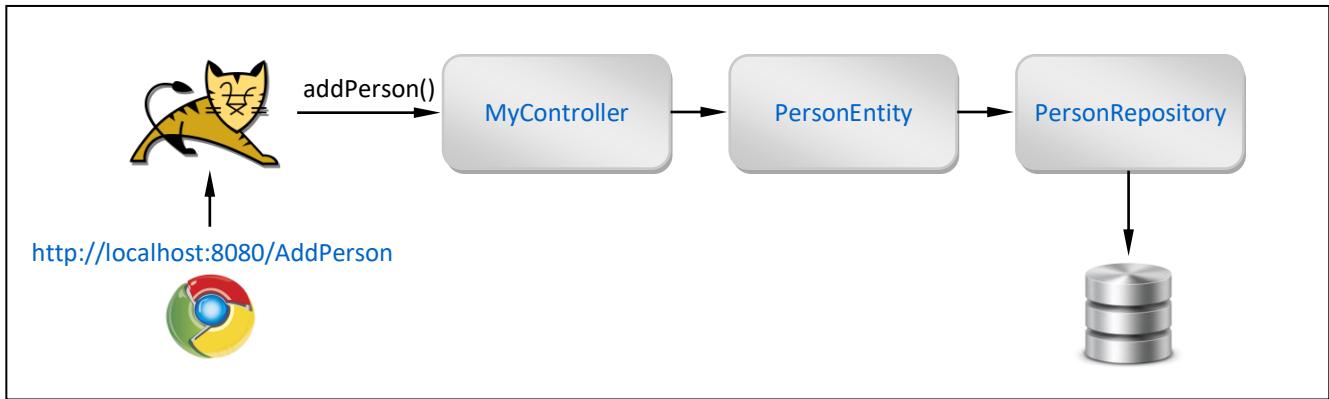
## 4.4.1 Application Example

### Info

- This tutorial show how to use H2 Database by creating a Person Entity and then storing it into DB as a Record.
- This example is exactly the same as [Repository - H2](#).

Application Schema

[Results]



Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables @Controller and @RequestMapping. Includes Tomcat Server.
Developer Tools	Spring Boot DevTools	Enables application auto-reload
SQL	Spring Data JPA	Enables @Entity and @Id
SQL	H2 Database	Enables in-memory H2 Database

## Procedure

---

- Create Project: entity\_recommended (add Spring Boot Starters from the table)
- Edit: application.properties (specify H2 DB name & enable H2 Web Console)
- Create Package: entities (inside main package)
  - Create Class: PersonEntity.java (inside package entities)
- Create Package: repositories (inside main package)
  - Create Interface: PersonRepository.java (inside package repositories)
- Create Package: controllers (inside main package)
  - Create Class: MyController.java (inside package controllers)

*application.properties*

```
spring.datasource.url      = jdbc:h2:mem:testdb
spring.h2.console.enabled = true
```

*PersonEntity.java*

```
package com.ivoronline.springboot.entity_recommended.entities;

import lombok.Data;
import org.springframework.stereotype.Component;
import javax.persistence.Entity;
import javax.persistence.Id;

@Data
@Entity
@Component
public class PersonEntity {

    @Id
    private Integer id;
    private String name;
    private Integer age;

}
```

*PersonRepository.java*

```
package com.ivoronline.springboot.repository_store_entity.repositories;

import com.ivoronline.springboot.repository_store_entity.entities.PersonEntity;
import org.springframework.data.repository.CrudRepository;

public interface PersonRepository extends CrudRepository<PersonEntity, Integer> { }
```

### MyController.java

```
package com.ivorononline.springboot.repository_store_entity.controllers;

import com.ivorononline.springboot.repository_store_entity.entities.PersonEntity;
import com.ivorononline.springboot.repository_store_entity.repositories.PersonRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class MyController {

    @Autowired
    PersonEntity personEntity;

    @Autowired
    PersonRepository personRepository;

    @ResponseBody
    @RequestMapping("/addPerson")
    public String addPerson() {

        //CREATE PERSON
        personEntity.setId(1);
        personEntity.setName("John");
        personEntity.setAge(20);

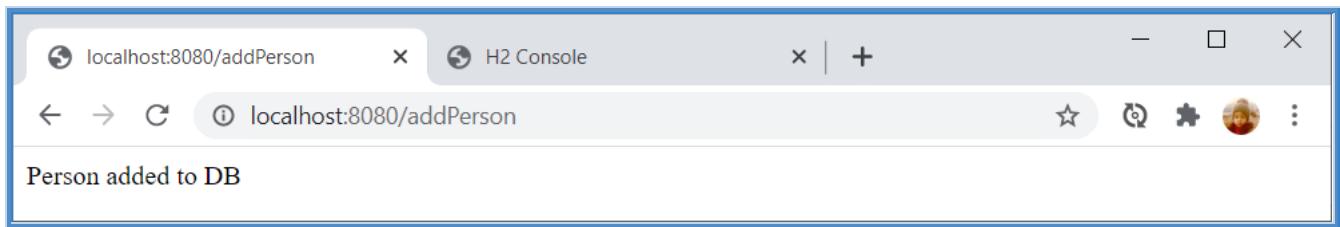
        //STORE PERSON
        personRepository.save(personEntity);

        String name = personEntity.getName();
        return "Hello " + name;
    }
}
```

## Results

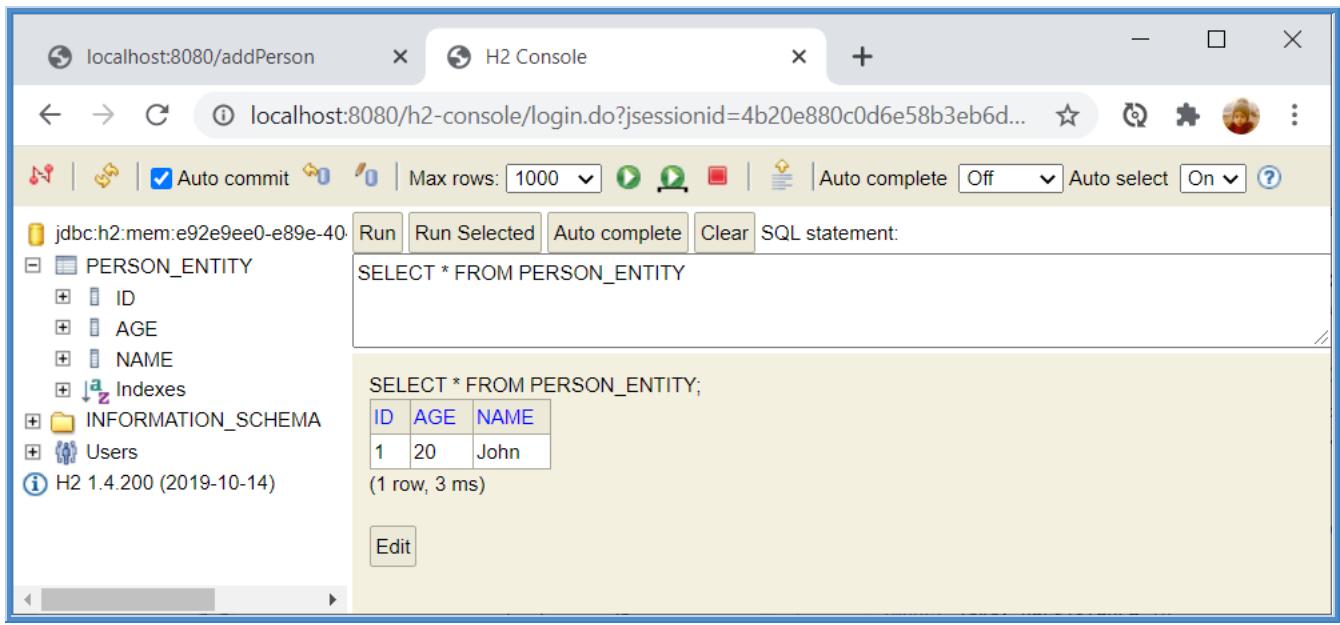
<http://localhost:8080/addPerson>

(Web Browser)

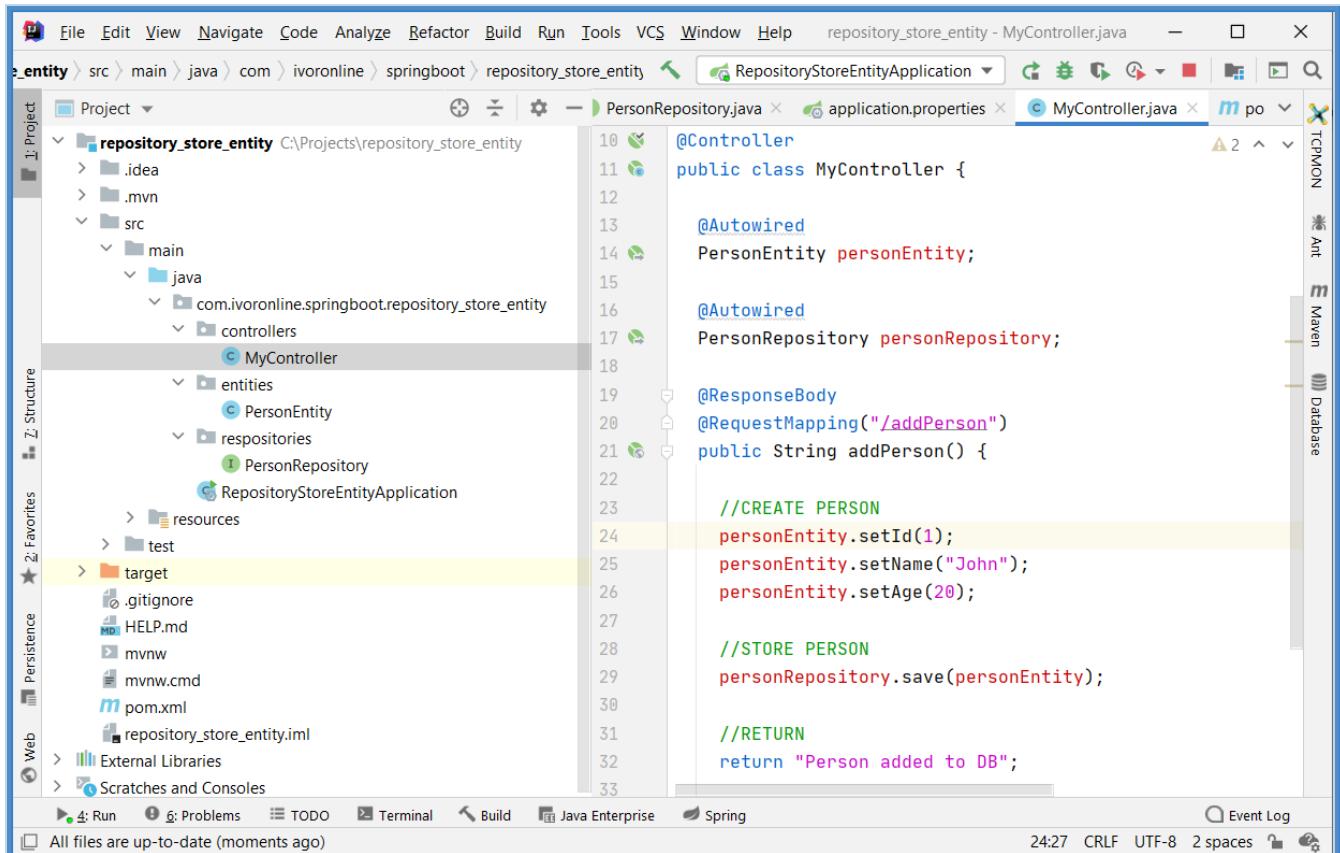


<http://localhost:8080/h2-console>

(Open H2 Console)



## Application Structure



```
<dependencies>

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

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

    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-devtools</artifactId>
        <scope>runtime</scope>
        <optional>true</optional>
    </dependency>

    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <optional>true</optional>
    </dependency>

</dependencies>
```

## 4.4.2 DB: Specify Name

### Info

---

- This tutorial shows how to specify H2 DB name by adding highlighted line to **application.properties** file.

### Specify DB Name

---

- Edit **application.properties**

*application.properties*

```
spring.datasource.url      = jdbc:h2:mem:testdb
spring.h2.console.enabled = true
```

## 4.4.3 Console: Enable

### Info

---

- This tutorial shows how to enable H2 Console by adding highlighted line to **application.properties** file.

### Enable H2 Console

---

- Edit [application.properties](#)

*application.properties*

```
spring.datasource.url      = jdbc:h2:mem:testdb
spring.h2.console.enabled = true
```

## 4.4.4 Console: Open

### Info

- This tutorial shows to open H2 Web Console (to query DB).
- First you need to [Enable H2 Console](#) for each Project.

### Open H2 Console

- <http://localhost:8080/h2-console>
- Connect

<http://localhost:8080/h2-console>

The screenshot shows the H2 Console login interface. At the top, there is a navigation bar with links for English, Preferences, Tools, and Help. Below the navigation bar is a "Login" form. The "Saved Settings" dropdown is set to "Generic H2 (Embedded)". The "Setting Name" field contains "Generic H2 (Embedded)" with "Save" and "Remove" buttons next to it. The "Driver Class" field contains "org.h2.Driver". The "JDBC URL" field contains "jdbc:h2:mem:testdb". The "User Name" field contains "sa". The "Password" field is empty. At the bottom of the form are "Connect" and "Test Connection" buttons.

## 4.4.5 Console: Show Records

### Info

- This tutorial shows how to use H2 Web Console to Table Records.

### Show Records

- [Open H2 Console](#)
- Click on Person\_Entity (constructs SQL)
- Run (runs SQL)

*SELECT \* FROM PERSON\_ENTITY*

The screenshot shows the H2 Web Console interface. On the left, the schema browser displays tables like PERSON\_ENTITY, INFORMATION\_SCHEMA, and Users. The PERSON\_ENTITY table is selected and expanded, showing columns ID, AGE, and NAME. On the right, the main panel contains the SQL query `SELECT * FROM PERSON_ENTITY;` and its result, which is a table with one row:

ID	AGE	NAME
1	20	John

(1 row, 1 ms)

## 4.4.6 Console: Security

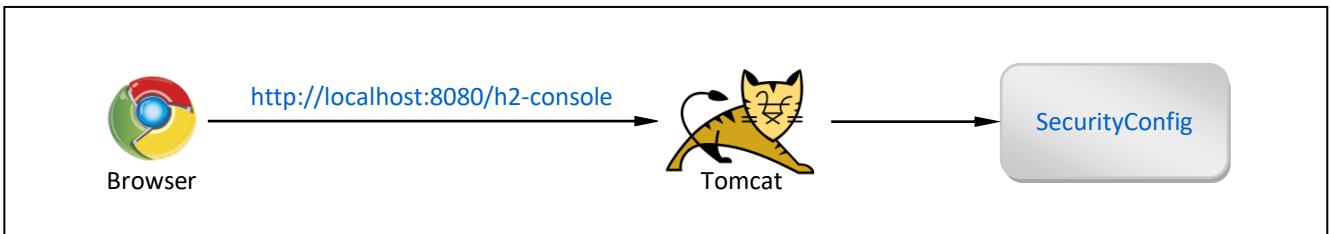
### Info

[R] [R]

- This tutorial shows how to access H2 Console without logging in when using Spring Security.

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables @Controller, @RequestMapping and Tomcat Server.
SQL	Spring Data JPA	Enables @Entity and @Id
SQL	H2 Database	Enables in-memory H2 Database
Security	Spring Security	Enables Spring Security.

### Procedure

- Edit File: `application.properties`
- Create Package: config
  - Create Class: `SecurityConfig.java`

#### `application.properties`

```
spring.h2.console.enabled = true
spring.datasource.url      = jdbc:h2:mem:testdb
```

#### `SecurityConfig.java`

```
package com.ivoronline.springboot_security_h2console.config;

import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity httpSecurity) throws Exception {

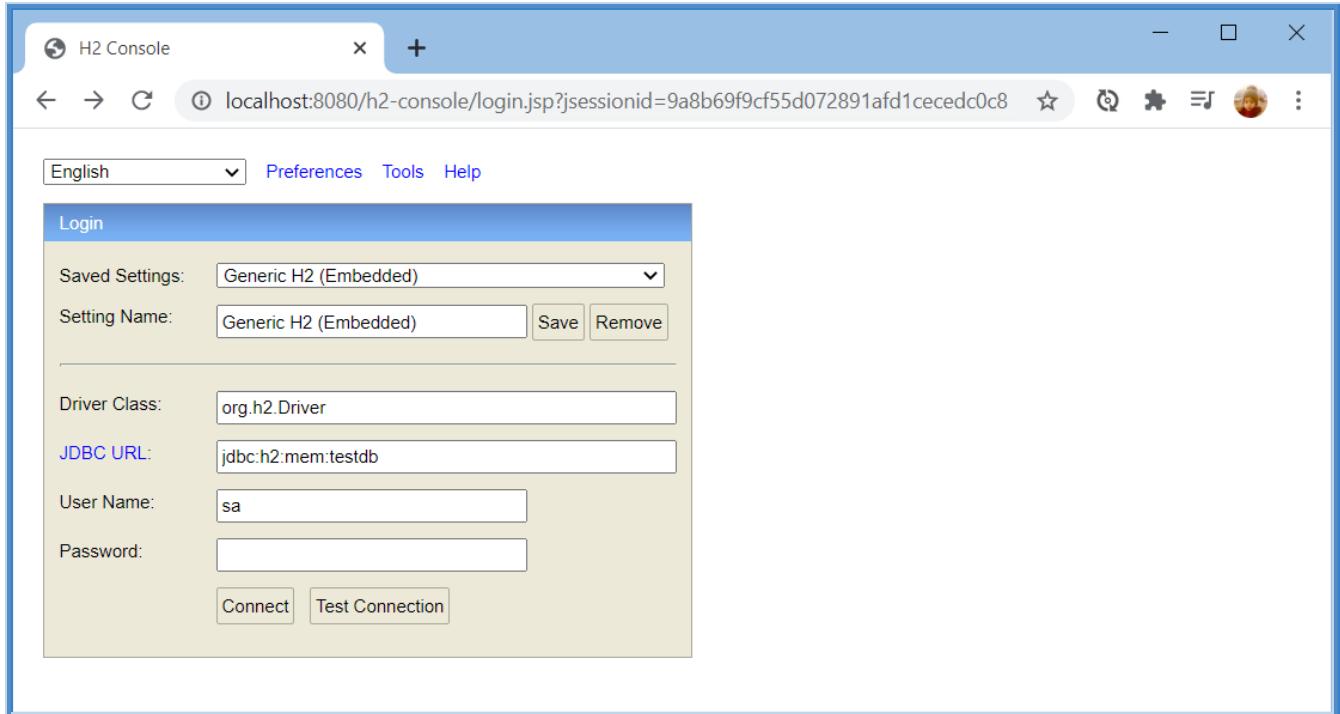
        //H2 CONSOLE
        httpSecurity.authorizeRequests(authorize -> { authorize.antMatchers("/h2-console/**").permitAll(); });
        httpSecurity.headers().frameOptions().sameOrigin();
        httpSecurity.csrf().disable();

        //EVERYTHING ELSE IS LOCKED
        httpSecurity.authorizeRequests().anyRequest().authenticated();

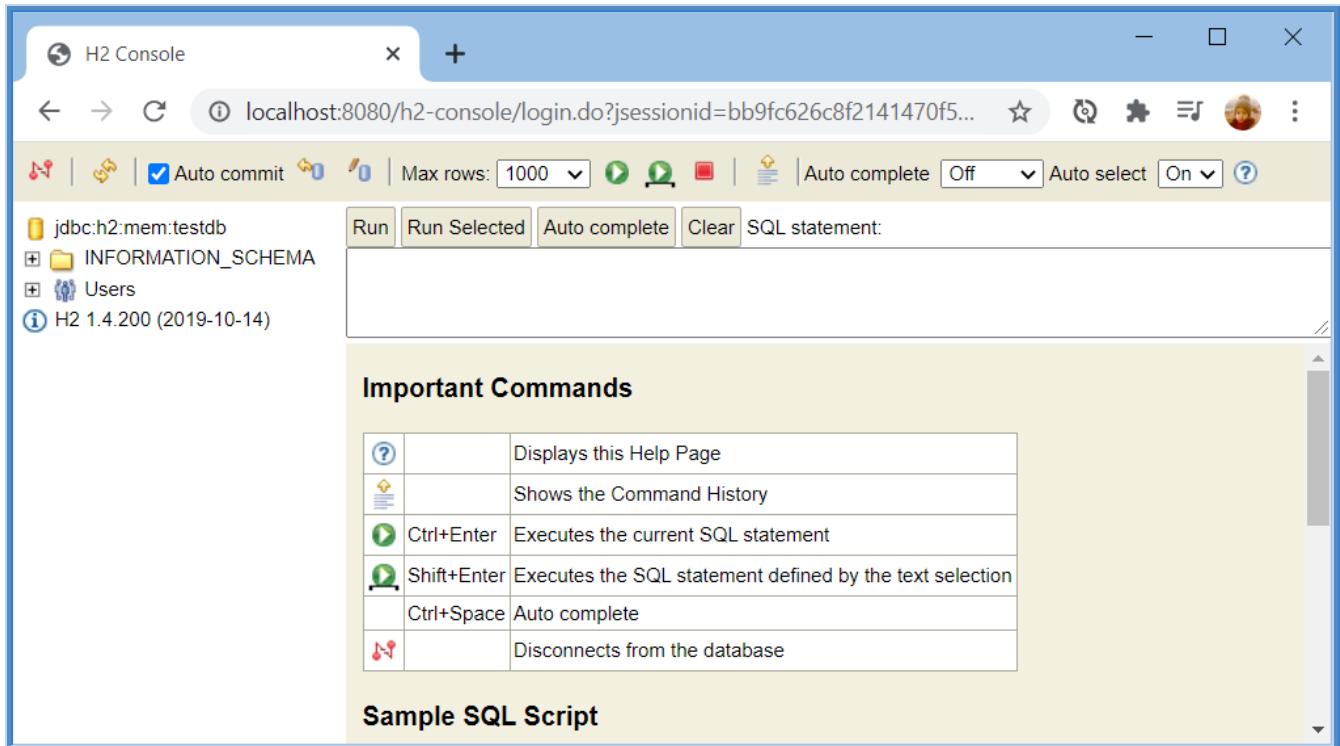
    }
}
```

## Results

<http://localhost:8080/h2-console>



<http://localhost:8080/h2-console/login.do?jsessionid=bb9fc626c8f2141470f53b1e44831c28>



```
<dependencies>

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

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

    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
    </dependency>

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

</dependencies>
```

## 4.5 Database - MySQL

### Info

- 
- Following tutorials show how to perform some of the operations related to MySQL DB.

## 4.5.1 Create Schema/Database

### Info

- This tutorial shows how to create new Schema/Database **stores** using MySQL Workbench.
- This might be needed since Hibernate needs existing Schema/Database (can't create new one).

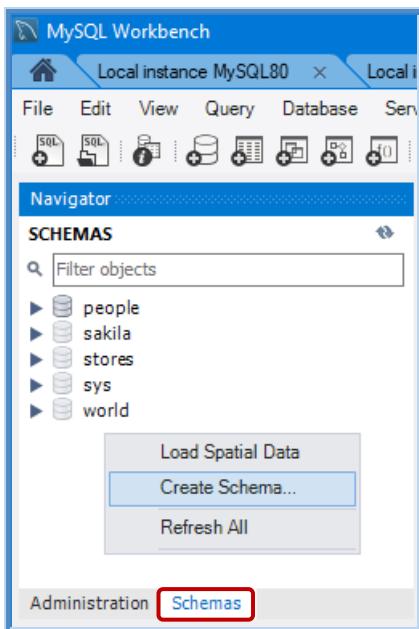
### SQL

```
CREATE SCHEMA stores
```

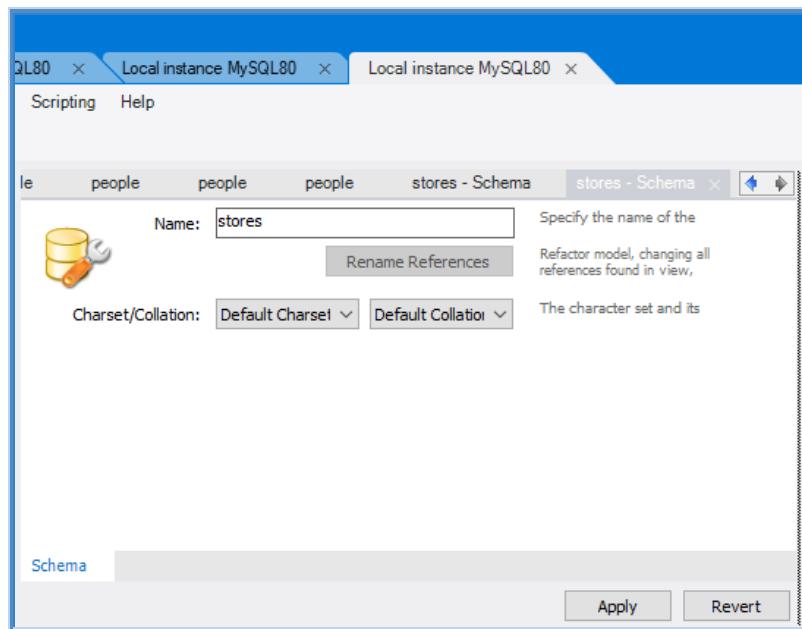
### Procedure

- Tab: Schemas
- RC on empty space (or existing Schema)
- Create Schema
- Name: stores
- Apply
- Apply
- Finish

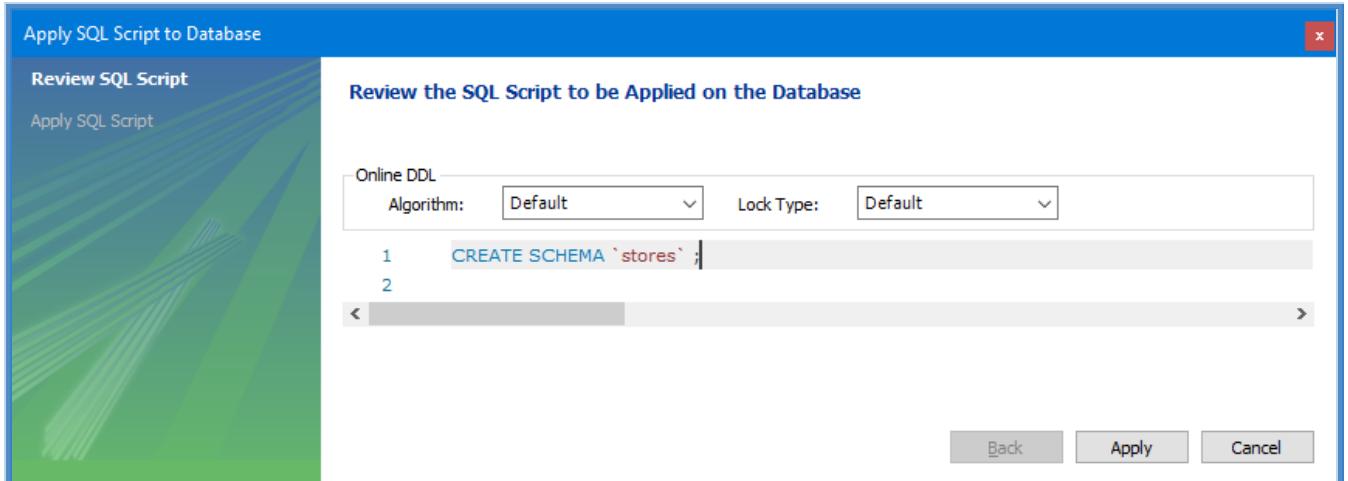
#### Create Schema



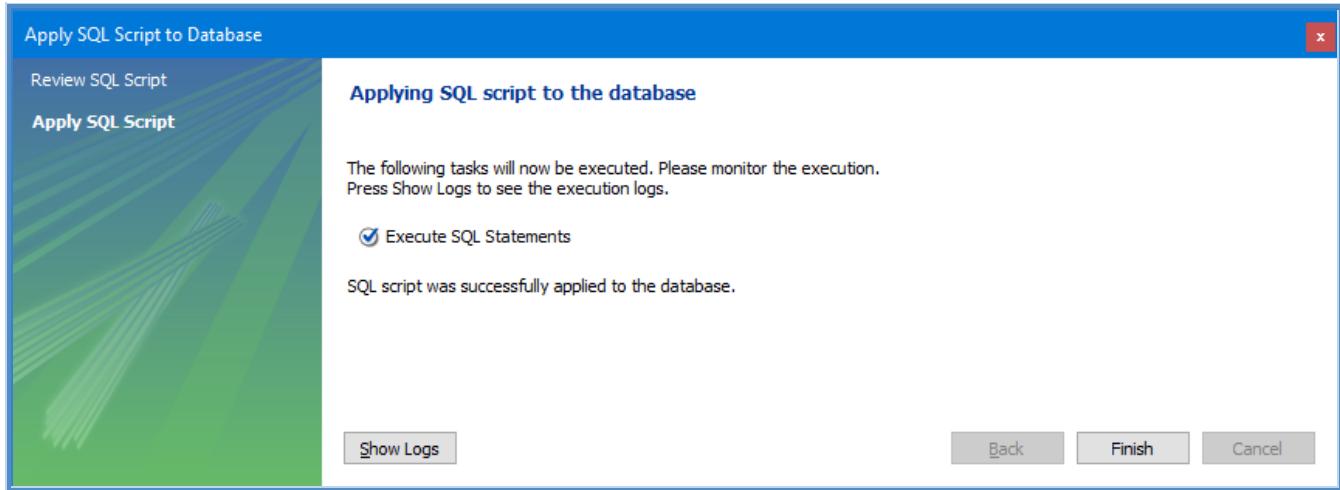
#### Name: stores



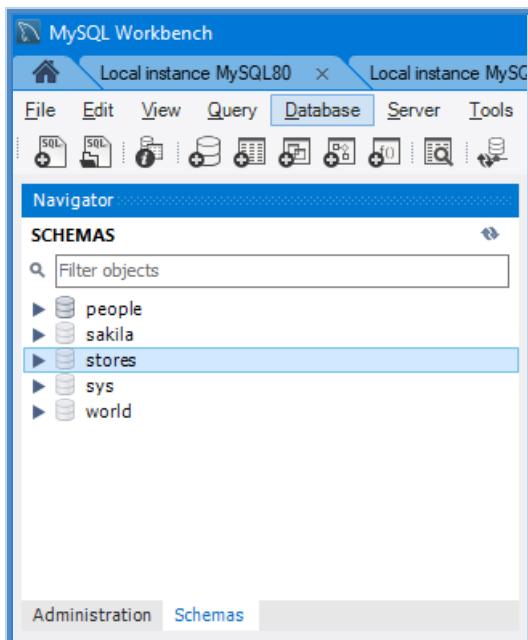
### Apply



*Finish*



*Created stores Schema/Database*



# 4.6 Mapper - Map Struct

## Info

[R]

- Map Struct is used to map Properties from one Class to another.
- It generates Class that implements Interface that you define.
- Inside the Interface you can use Annotations to simply defined additional mappings when Property names differ.
- If Property names are the same mapping will be done automatically.
- Benefit of Map Struct is that it doesn't change Entities but still uses Annotations inside Interface to define mappings.

*Author.java*

```
@Data  
public class Author {  
    private Integer id;  
    private String name;  
    private Integer age;  
}
```

*Book.java*

```
@Data  
public class Book {  
    private Integer id;  
    private String title;  
}
```

*AuthorBookDTO.java*

```
public class AuthorBookDTO {  
    public String authorName;  
    public Integer authorAge;  
    public String title;  
}
```

*AuthorBookMapper.java*

```
@Mapper  
public interface AuthorBookMapper {  
  
    AuthorBookMapper INSTANCE = Mappers.getMapper(AuthorBookMapper.class);  
  
    @Mapping(source = "authorName", target = "name") //For additional Properties with different names  
    @Mapping(source = "authorAge", target = "age") //For additional Properties with different names  
    Author mapToAuthor(AuthorBookDTO authorBookDTO);  
  
    Book mapToBook(AuthorBookDTO authorBookDTO); //It will map Properties with the same name  
  
}
```

*MyController.java*

```
//INSTANTIATE MAPPER  
AuthorBookMapper authorBookMapper = AuthorBookMapper.INSTANCE;  
  
//MAP AUTHORBOOKDTO TO AUTHOR & BOOK.  
Book book = authorBookMapper.mapToBook(authorBookDTO);  
Author author = authorBookMapper.mapToAuthor(authorBookDTO);
```

## 4.6.1 DTO to Entity - Map Struct

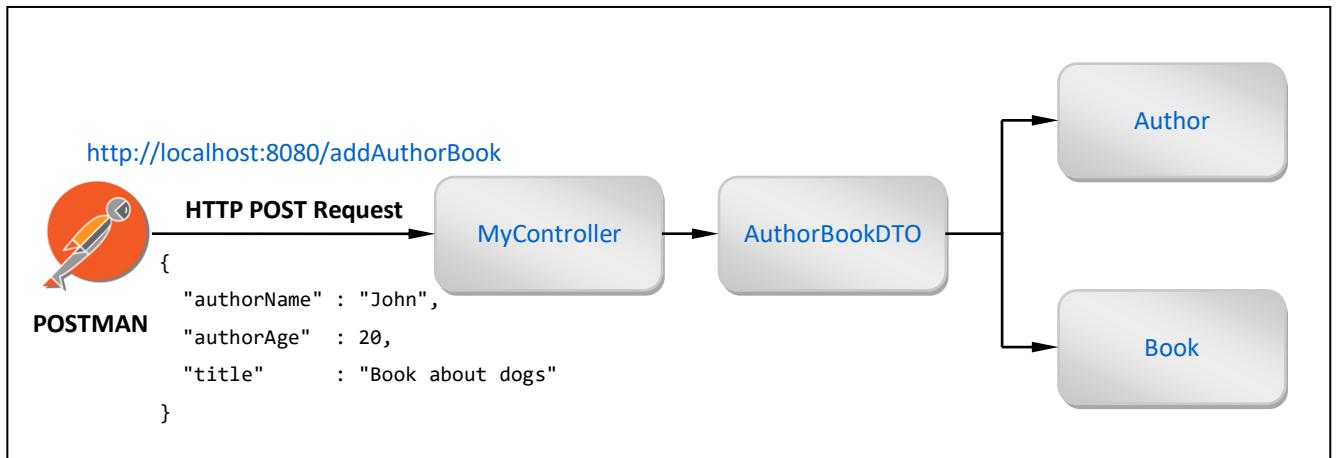
### Info

[G] [R] [R]

- This tutorial shows how to use [Map Struct](#) to convert [AuthorBookDTO](#) into [Author](#) & [Book](#) Entities.
- [AuthorBookDTO](#) is instantiated from Postman's **JSON** HTTP Request as described in [Annotation - \(Spring\) @RequestBody](#).

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables @Controller and @RequestMapping. Includes Tomcat Server.

### Procedure

- Create Project: [springboot\\_dto\\_mapstruct](#) (add Spring Boot Starters from the table)
- Edit File: [pom.xml](#) (add highlighted: <property>, <dependency>, <plugin>)
- Create Package: [entities](#) (inside main package)
  - Create Class: [Author.java](#) (inside package entities)
  - Create Class: [Book.java](#) (inside package entities)
- Create Package: [DTOS](#) (inside main package)
  - Create Class: [AuthorBookDTO.java](#) (inside package controllers)
- Create Package: [mappers](#) (inside main package)
  - Create Interface: [AuthorBookMapper.java](#) (inside package controllers)
- Create Package: [controllers](#) (inside main package)
  - Create Class: [MyController.java](#) (inside package controllers)

```

<?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 https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.4.1</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <groupId>com.ivoronline</groupId>
  <artifactId>springboot_dto_mapstruct</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>springboot_dto_mapstruct</name>
  <description>Demo project for Spring Boot</description>

  <properties>
    <java.version>11</java.version>
    <org.mapstruct.version>1.4.1.Final</org.mapstruct.version>
  </properties>

  <dependencies>

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

    <dependency>
      <groupId>org.projectlombok</groupId>
      <artifactId>lombok</artifactId>
      <optional>true</optional>
    </dependency>

    <dependency>
      <groupId>org.mapstruct</groupId>
      <artifactId>mapstruct</artifactId>
      <version>${org.mapstruct.version}</version>
    </dependency>

    <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>
        <configuration>
          <excludes>
            <exclude>
              <groupId>org.projectlombok</groupId>
              <artifactId>lombok</artifactId>
            </exclude>
          </excludes>
        </configuration>
      </plugin>

    </plugins>
  </build>

```

```

</plugin>

<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.8.1</version>
    <configuration>
        <source>1.8</source> <!-- depending on your project -->
        <target>1.8</target> <!-- depending on your project -->
        <annotationProcessorPaths>
            <path>
                <groupId>org.mapstruct</groupId>
                <artifactId>mapstruct-processor</artifactId>
                <version>${org.mapstruct.version}</version>
            </path>
            <!-- other annotation processors -->
        </annotationProcessorPaths>
    </configuration>
</plugin>

</plugins>
</build>

</project>

```

#### *Author.java*

```

package com.ivoronline.springboot_dto_mapstruct.entities;

public class Author {
    public Integer id;
    public String name;
    public Integer age;
}

```

#### *Book.java*

```

package com.ivoronline.springboot_dto_mapstruct.entities;

public class Book {
    public Integer id;
    public String title;
}

```

#### *AuthorBookDTO.java*

```

package com.ivoronline.springboot_dto_mapstruct.DTO;

public class AuthorBookDTO {
    public String authorName;
    public Integer authorAge;
    public String title;
}

```

### *AuthorBookMapper.java*

```
package com.ivoronline.springboot_dto_mapstruct.mappers;

import com.ivoronline.springboot_dto_mapstruct.DTO.AuthorBookDTO;
import com.ivoronline.springboot_dto_mapstruct.entities.Author;
import com.ivoronline.springboot_dto_mapstruct.entities.Book;
import org.mapstruct.Mapper;
import org.mapstruct.Mapping;
import org.mapstruct.factory.Mappers;

@Mapper
public interface AuthorBookMapper {

    AuthorBookMapper INSTANCE = Mappers.getMapper(AuthorBookMapper.class);

    //Without Annotations it will map Properties with the same name
    Book mapToBook(AuthorBookDTO authorBookDTO);

    //Annotations can be used to map Properties with different names
    @Mapping(source = "authorName", target = "name")
    @Mapping(source = "authorAge", target = "age")
    Author mapToAuthor(AuthorBookDTO authorBookDTO);

}
```

### *MyController.java*

```
package com.ivoronline.springboot_dto_mapstruct.controllers;

import com.ivoronline.springboot_dto_mapstruct.DTO.AuthorBookDTO;
import com.ivoronline.springboot_dto_mapstruct.entities.Author;
import com.ivoronline.springboot_dto_mapstruct.entities.Book;
import com.ivoronline.springboot_dto_mapstruct.mappers.AuthorBookMapper;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class MyController {

    @ResponseBody
    @RequestMapping("addAuthorBook")
    public String addAuthorBook(@RequestBody AuthorBookDTO authorBookDTO) {

        //INSTANTIATE MAPPER
        AuthorBookMapper authorBookMapper = AuthorBookMapper.INSTANCE;

        //MAP AUTHORBOOKDTO TO AUTHOR & BOOK.
        Book book = authorBookMapper.mapToBook(authorBookDTO);
        Author author = authorBookMapper.mapToAuthor(authorBookDTO);

        //RETURN SOMETHING
        return author.name + " has written: " + book.title;

    }
}
```

## Results

- Start Postman

POST

http://localhost:8080/addAuthorBook

Headers

(add Key-Value)

Content-Type: application/json

Body

(option: raw)

```
{  
    "authorName" : "John",  
    "authorAge" : 20,  
    "title" : "Book about dogs"  
}
```

HTTP Response Body

John has written Book about dogs

HTTP Request Headers (Bulk Edit)

Postman

File Edit View Help

+ New Import Runner My Works

POST MyRequest POST MyRequest

MyRequest

POST http://localhost:8080/addAuthorBook

Params Authorization Headers (11) Body Pre-req

Headers Hide auto-generated headers

Content-Type: application/json  
Host: localhost  
Content-Length: 100

Body Cookies Headers (3) Test Results

Pretty Raw Preview Visualize Text

1 John has written Book about dogs

Find and Replace Console

Request JSON Body (raw) & Response Body

Postman

File Edit View Help

+ New Import Runner My Works

POST MyRequest POST MyRequest

MyRequest

POST http://localhost:8080/addAuthorBook

Params Authorization Headers (11) Body Pre-req

none form-data x-www-form-urlencoded raw

1 {  
2 "name": "John",  
3 "age": 20,  
4 "title": "Book about dogs"  
5 }

Body Cookies Headers (3) Test Results

Pretty Raw Preview Visualize Text

1 John has written Book about dogs

Find and Replace Console

### AuthorBookMapperImpl.java

```
package com.ivoronline.springboot_dto_mapstruct.mappers;

import com.ivoronline.springboot_dto_mapstruct.DTO.AuthorBookDTO;
import com.ivoronline.springboot_dto_mapstruct.entities.Author;
import com.ivoronline.springboot_dto_mapstruct.entities.Book;
import javax.annotation.Generated;

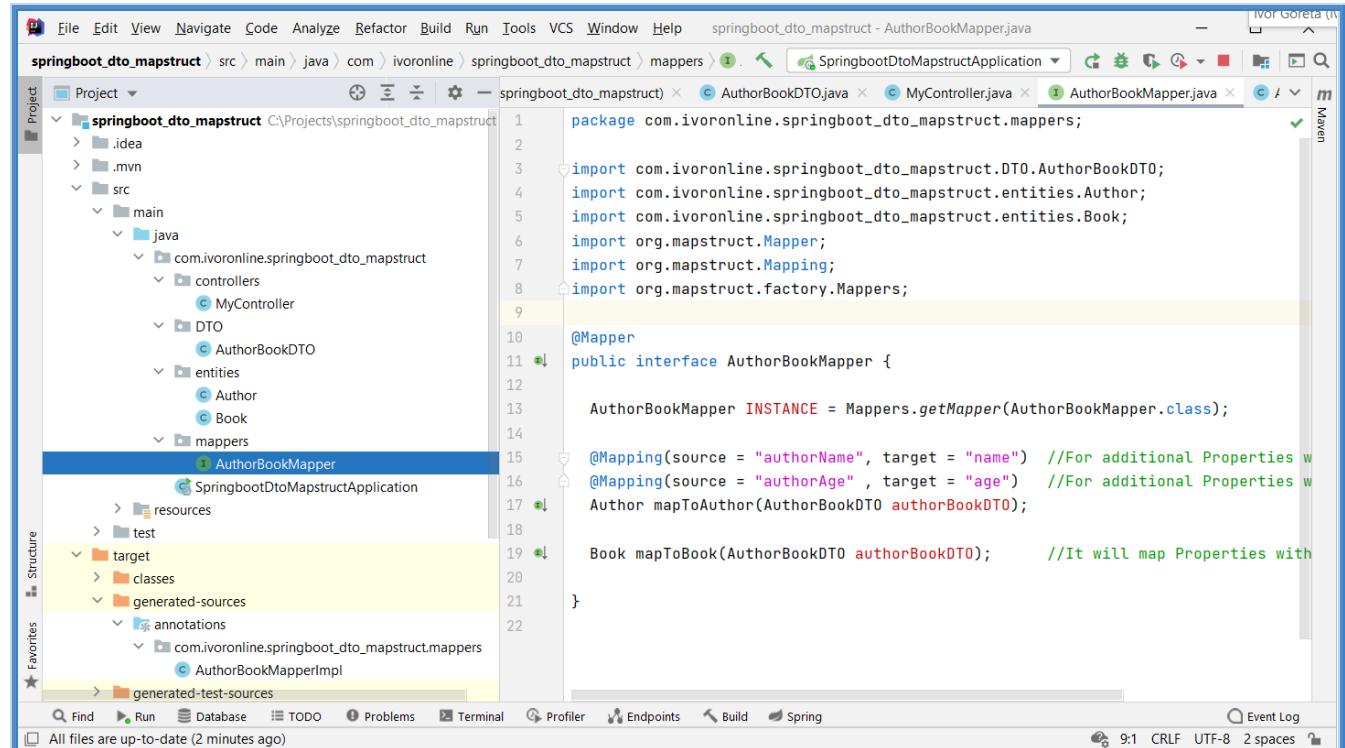
@Generated(
    value = "org.mapstruct.ap.MappingProcessor",
    date = "2021-01-13T19:30:28+0100",
    comments = "version: 1.4.1.Final, compiler: javac, environment: Java 15.0.1 (Oracle Corporation)"
)
public class AuthorBookMapperImpl implements AuthorBookMapper {

    @Override
    public Author mapToAuthor(AuthorBookDTO authorBookDTO) {
        if (authorBookDTO == null) { return null; }
        Author author = new Author();
        author.name = authorBookDTO.authorName;
        author.age = authorBookDTO.authorAge;
        return author;
    }

    @Override
    public Book mapToBook(AuthorBookDTO authorBookDTO) {
        if (authorBookDTO == null) { return null; }
        Book book = new Book();
        book.title = authorBookDTO.title;
        return book;
    }

}
```

### Application Structure



# 4.7 Mapper - JMapper

## Info

- JMapper is used to map Properties from one Class to another.  
Unlike other mappers it supports **Annotations** (making it the simplest to use: less code, more readable).
- JMapper will not automatically map Properties with the same name (like Model Mapper would).  
Instead Destination Class Properties must be Annotated (you must specify which Destination Properties should be filled).  
This allows you to have single approach for custom mapping and excluding Properties.
- Annotations can specify name of the Source Property (if it is different).
- Destination Class Annotated Properties must have both getters and setters (even when only one of them are used).
- When instantiating Generic Class **JMapper<Destination, Source>** then Destination Class is declared first.

*Author.java*

(Destination)

```
@Data //Lombok creates setters and getters
public class Author {

    public Integer id; //It will not be mapped since there is no @JMap Annotation

    @JMap //Name of the Source Property should be the same "age"
    public String name;

    @JMap("authorAge") //Name of the Source Property should be "authorAge"
    public Integer age;

}
```

*AuthorBookDTO.java*

(Source)

```
@Data //Lombok creates setters and getters
public class AuthorBookDTO {
    public Integer id;
    public String name;
    public Integer authorAge;
    public String title;
}
```

*MyController.java*

(Usage)

```
//INSTANTIATE JMAPPER<DESTINATION, SOURCE>
JMapper<Author, AuthorBookDTO> authorMapper = new JMapper<>(Author.class, AuthorBookDTO.class);

//MAP AuthorBookDTO TO Author
Author author = authorMapper.getDestination(authorBookDTO);
```

## 4.7.1 DTO to Entity - JMapper - Annotations

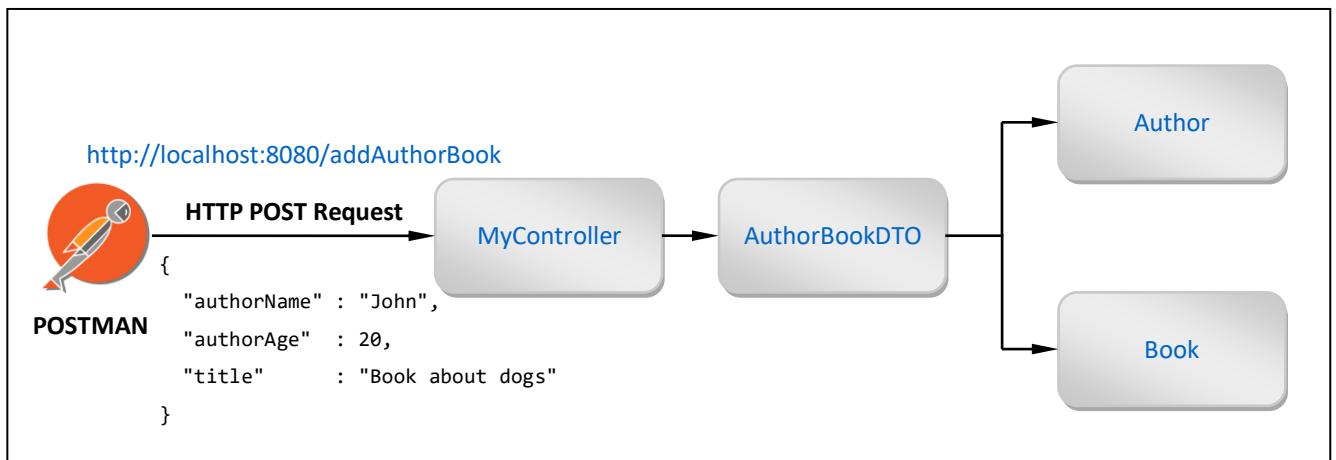
### Info

[G] [R]

- This tutorial shows how to use **JMapper's Annotations** to convert `AuthorBookDTO` into `Author` and `Book` Entities.
- `AuthorBookDTO` is instantiated from Postman's **JSON** HTTP Request as described in [Annotation - \(Spring\) @RequestBody](#).

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables <code>@Controller</code> and <code>@RequestMapping</code> . Includes Tomcat Server.
Developer Tools	Lombok	Enables <code>@Data</code> which generate helper methods (setters, getters, ...)

### Procedure

- Create Project:** `springboot_dtojmapper` (add Spring Boot Starters from the table)
- Edit File:** `pom.xml` (manually add JMapper dependency)
- Create Package:** `entities` (inside main package)
  - Create Class:** `Author.java` (inside package `entities`)
  - Create Class:** `Book.java` (inside package `entities`)
- Create Package:** `DTOs` (inside main package)
  - Create Class:** `AuthorBookDTO.java` (inside package `controllers`)
- Create Package:** `controllers` (inside main package)
  - Create Class:** `MyController.java` (inside package `controllers`)

*pom.xml*

```
<dependencies>

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

    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <optional>true</optional>
    </dependency>

    <dependency>
        <groupId>com.googlecode.jmapper-framework</groupId>
        <artifactId>jmapper-core</artifactId>
        <version>1.6.0.1</version>
    </dependency>

</dependencies>
```

*Author.java*

```
package com.ivoronline.springboot_dto_jmapper.entities;

import com.googlecode.jmapper.annotations.JMap;
import lombok.Data;

@Data
public class Author {

    public Integer id;      //Can't be mapped since there is no Annotation

    @JMap("authorName")   //Name of the Source Property
    public String name;

    @JMap("authorAge")    //Name of the Source Property
    public Integer age;

}
```

*Book.java*

```
package com.ivoronline.springboot_dto_jmapper.entities;

import com.googlecode.jmapper.annotations.JMap;
import lombok.Data;

@Data
public class Book {

    public Integer id;

    @JMap           //Source Property should have the same name
    public String title;

}
```

### *AuthorBookDTO.java*

```
package com.ivoronline.springboot_dto_jmapper.DTO;

import lombok.Data;

@Data
public class AuthorBookDTO {
    public String authorName;
    public Integer authorAge;
    public String title;
}
```

### *MyController.java*

```
package com.ivoronline.springboot_dto_jmapper.controllers;

import com.googlecode.jmapper.JMapper;
import com.ivoronline.springboot_dto_jmapper.DTO.AuthorBookDTO;
import com.ivoronline.springboot_dto_jmapper.entities.Author;
import com.ivoronline.springboot_dto_jmapper.entities.Book;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class MyController {

    @ResponseBody
    @RequestMapping("addAuthorBook")
    public String addAuthorBook(@RequestBody AuthorBookDTO authorBookDTO) {

        //INSTANTIATE JMAPPERS<DESTINATION, SOURCE>
        JMapper<Book, AuthorBookDTO> bookMapper = new JMapper<>(Book .class, AuthorBookDTO.class);
        JMapper<Author, AuthorBookDTO> authorMapper = new JMapper<>(Author.class, AuthorBookDTO.class);

        //MAP AuthorBookDTO TO AUTHOR & BOOK.
        Book book = bookMapper .getDestination(authorBookDTO);
        Author author = authorMapper.getDestination(authorBookDTO);

        //RETURN SOMETHING
        return author.name + " has written: " + book.title;

    }
}
```

## Results

- Start Postman

POST

http://localhost:8080/addAuthorBook

Headers

(add Key-Value)

Content-Type: application/json

Body

(option: raw)

```
{  
    "authorName" : "John",  
    "authorAge" : 20,  
    "title" : "Book about dogs"  
}
```

HTTP Response Body

John has written Book about dogs

HTTP Request Headers (Bulk Edit)

Postman

File Edit View Help

+ New Import Runner My Works

POST MyRequest POST MyRequest

MyRequest

POST http://localhost:8080/addAuthorBook

Params Authorization Headers (11) Body Pre-req

Headers Hide auto-generated headers

Content-Type: application/json  
Host: localhost  
Content-Length: 100

Body Cookies Headers (3) Test Results

Pretty Raw Preview Visualize Text

1 John has written Book about dogs

Find and Replace Console

Request JSON Body (raw) & Response Body

Postman

File Edit View Help

+ New Import Runner My Works

POST MyRequest POST MyRequest

MyRequest

POST http://localhost:8080/addAuthorBook

Params Authorization Headers (11) Body Pre-req

none form-data x-www-form-urlencoded raw

1 {  
2 "name": "John",  
3 "age": 20,  
4 "title": "Book about dogs"  
5 }

Body Cookies Headers (3) Test Results

Pretty Raw Preview Visualize Text

1 John has written Book about dogs

Find and Replace Console

## Application Structure

The screenshot shows the IntelliJ IDEA interface with the following details:

- Project Tree:** The project is named "springboot(dto\_jmapper)". It contains a "src" directory with "main" and "test" sub-directories. "main" contains "java" which has "com.ivoronline.springboot(dto\_jmapper)" and "controllers". "controllers" contains "MyController.java". Other packages like "DTO", "entities", "Author", "Book", and "SpringbootDtoJMapperApplication" are also visible.
- Code Editor:** The "MyController.java" file is open. The code defines a controller with a method "addAuthorBook". It uses JMapper annotations to map "AuthorBookDTO" to "Author" and "Book".

```
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/addAuthorBook")
public String addAuthorBook(@RequestBody AuthorBookDTO authorBookDTO) {
    //INSTANTIATE JMAPPER<DESTINATION, SOURCE>
    JMapper<Book, AuthorBookDTO> bookMapper = new JMapper<>(Book.class, AuthorBookDTO.class);
    JMapper<Author, AuthorBookDTO> authorMapper = new JMapper<>(Author.class, AuthorBookDTO.class);

    //MAP AuthorBookDTO TO AUTHOR & BOOK.
    Book book = bookMapper.getDestination(authorBookDTO);
    Author author = authorMapper.getDestination(authorBookDTO);

    //RETURN SOMETHING
    return author.name + " has written: " + book.title;
}
```
- Pom.xml:** The pom.xml file is shown in the bottom left, indicating the project uses Spring Boot Starter Web and Lombok.
- Bottom Bar:** Shows the status bar with "All files are up-to-date (moments ago)", the current time "10:30", and encoding "UTF-8 2 spaces".

## pom.xml

```
<dependencies>

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

    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <optional>true</optional>
    </dependency>

    <dependency>
        <groupId>com.googlecode.jmapper-framework</groupId>
        <artifactId>jmapper-core</artifactId>
        <version>1.6.0.1</version>
    </dependency>

</dependencies>
```

# 4.8 Mapper - Model Mapper

## Info

[R]

- Model Mapper
  - is Java API that allows you to convert Java Data Objects between each other
  - copies Property values from one Object into another (by matching Properties, Setters & Getters)
- We have used Model Mapper in [Authors & Books - Step 7: Service AuthorBook tutorial](#) to convert Data Transfer Objet (DTO) into Author and Book Entities .

*AuthorBookDTO.java*

```
public class AuthorBookDTO {  
    public String authorName;  
    public Integer authorAge;  
    public String title;  
}
```

*Author.java*

```
@Data  
public class Author {  
    private Integer id;  
    private String name;  
    private Integer age;  
}
```

*Book.java*

```
@Data  
public class Book {  
    private Integer id;  
    private String title;  
}
```

*MyController.java*

```
@Controller  
public class MyController {  
  
    @ResponseBody  
    @RequestMapping("addAuthorBook")  
    public String addAuthorBook(@RequestBody AuthorBookDTO authorBookDTO) {  
  
        //INSTANTIATE MODEL MAPPER  
        ModelMapper modelMapper = new ModelMapper();  
  
        //CONVERT DTOs TO ENTITIES  
        Author author = modelMapper.map(authorBookDTO, Author.class);  
        Book book = modelMapper.map(authorBookDTO, Book.class);  
  
        //RETURN SOMETHING  
        return author.name + " has written: " + book.title;  
    }  
}
```

## 4.8.1 Example

### Info

- For example we can use Model Mapper as shown in [DTO to Entity - Model Mapper - Default](#).
- There we have used Postman to send HTTP Request with JSON Body that contains data for both Author and Book.

HTTP Request Body: JSON

```
{  
    "name" : "John",  
    "age" : 20,  
    "title" : "Book about dogs"  
}
```

- Then we create Data Transfer Object in which JSON data will be loaded.

*AuthorBookDTO.java*

```
@Data  
@Component  
public class AuthorBookDTO {  
    private String name;  
    private Integer age;  
    private String title;  
}
```

- Our intention will be to use Model Mapper to automatically load these data into Author and Book Entities.

*Author.java*

```
@Data  
public class Author {  
    private Integer id;  
    private String name;  
    private Integer age;  
}
```

*Book.java*

```
@Data  
public class Book {  
    private Integer id;  
    private String title;  
}
```

- To do this, inside Controller we can Annotate Endpoint's Input Parameter with `@RequestBody` to automatically instantiate `AuthorBookDTO` and load JSON data into it. And inside Endpoint's Body we instantiate new ModelMapper and use it to transfer data from `AuthorBookDTO` into `Author` and `Book` Entities. ModelMapper will automatically match related Properties by using their names.

*MyController.java*

```
@Controller
public class MyController {

    @ResponseBody
    @RequestMapping("addAuthorBook")
    public String addAuthorBook(@RequestBody AuthorBookDTO authorBookDTO) {

        //INSTANTIATE MODEL MAPPER
        ModelMapper modelMapper = new ModelMapper();

        //CONVERT DTOs TO ENTITIES
        Author author = modelMapper.map(authorBookDTO, Author.class);
        Book book = modelMapper.map(authorBookDTO, Book.class);

        //RETURN SOMETHING
        return author.name + " has written: " + book.title;

    }
}
```

## 4.8.2 Matching Strategy

### Info

[R]

- Model Mapper can use different Matching Strategies to figure out related Properties between Objects.

#### Configure Matching Strategy

```
ModelMapper modelMapper = new ModelMapper();
modelMapper.getConfiguration().setMatchingStrategy(MatchingStrategies.STRICT);
```

#### Matching Strategies

NAME	DESCRIPTION
Standard	Default. Intelligently matches to properties.
Loose	Requires that only the last destination property in a hierarchy be matched
Strict	Properties must be strictly matched. Ensure that no unexpected mapping occurs.

## 4.8.3 DTO to Entity - Model Mapper - Default

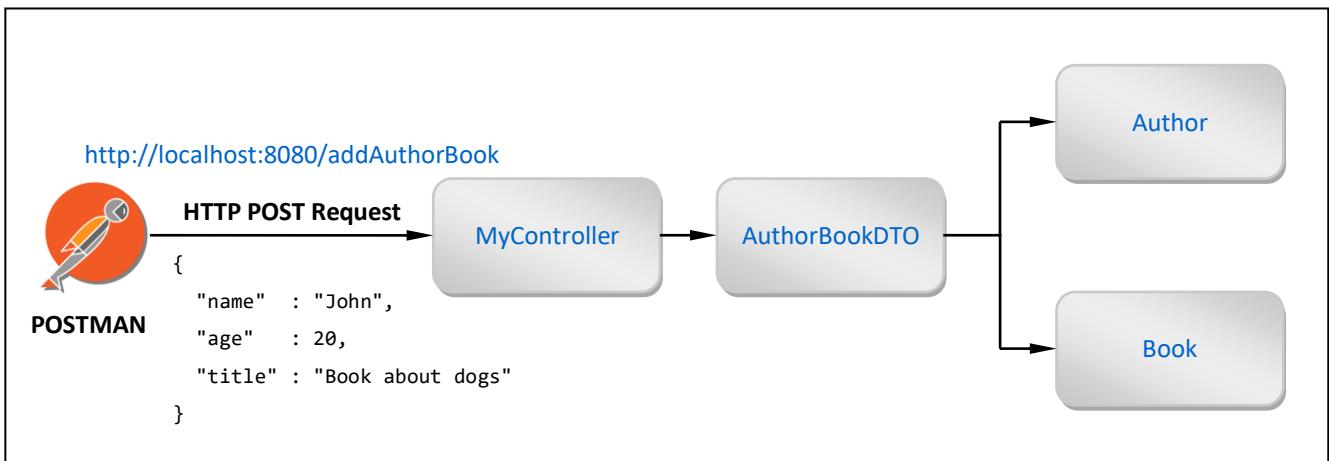
### Info

[G]

- This tutorial shows how to use [Model Mapper](#) to convert DTO into Entities.
- Tutorial [Annotation - \(Spring\) @RequestBody](#) shows how to convert JSON data from HTTP Request into DTO.

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables @Controller and @RequestMapping. Includes Tomcat Server.

### Procedure

- Create Project: [springboot\\_dto\\_modelmapper](#) (add Spring Boot Starters from the table)
- Edit File: [pom.xml](#) (manually add Model Mapper dependency)
- Create Package: [entities](#) (inside main package)
  - Create Class: [Author.java](#) (inside package entities)
  - Create Class: [Book.java](#) (inside package entities)
- Create Package: [DTOs](#) (inside main package)
  - Create Class: [AuthorBookDTO.java](#) (inside package controllers)
- Create Package: [controllers](#) (inside main package)
  - Create Class: [MyController.java](#) (inside package controllers)

### pom.xml

```
<dependency>
<groupId>org.modelmapper</groupId>
<artifactId>modelmapper</artifactId>
<version>2.3.9</version>
</dependency>
```

### *Author.java*

```
package com.ivoronline.springboot_dto_modelmapper.entities;

public class Author {

    //PROPERTIES
    public Integer id;
    public String name;
    public Integer age;

    //SETTERS
    public void setId (Integer id) { this.id = id; }
    public void setName(String name) { this.name = name; }
    public void setAge (Integer age) { this.age = age; }

}
```

### *Book.java*

```
package com.ivoronline.springboot_dto_modelmapper.entities;

public class Book {

    //PROPERTIES
    public Integer id;
    public String title;

    //SETTERS
    public void setId (Integer id) { this.id = id; }
    public void setTitle(String title) { this.title = title; }

}
```

### *AuthorBookDTO.java*

```
public class AuthorBookDTO {

    //PROPERTIES
    public String name;
    public Integer age;
    public String title;

    //GETTERS
    public String getName () { return name; }
    public Integer getAge () { return age; }
    public String getTitle() { return title; }

}
```

### MyController.java

```
package com.ivorononline.springboot_dto_modelmapper.controllers;

import com.ivorononline.springboot_dto_modelmapper.DTOs.AuthorBookDTO;
import com.ivorononline.springboot_dto_modelmapper.entities.Author;
import com.ivorononline.springboot_dto_modelmapper.entities.Book;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;
import org.modelmapper.ModelMapper;

@Controller
public class MyController {

    @ResponseBody
    @RequestMapping("addAuthorBook")
    public String addAuthorBook(@RequestBody AuthorBookDTO authorBookDTO) {

        //INSTANTIATE MODEL MAPPER
        ModelMapper modelMapper = new ModelMapper();

        //CONVERT DTOs TO ENTITIES
        Author author = modelMapper.map(authorBookDTO, Author.class);
        Book book = modelMapper.map(authorBookDTO, Book.class);

        //RETURN SOMETHING
        return author.name + " has written: " + book.title;

    }
}
```

## Results

- Start Postman

POST

http://localhost:8080/addAuthorBook

Headers

(add Key-Value)

Content-Type: application/json

Body

(option: raw)

```
{  
    "name" : "John",  
    "age" : 20,  
    "title" : "Book about dogs"  
}
```

HTTP Response Body

John has written Book about dogs

HTTP Request Headers (Bulk Edit)

Postman

File Edit View Help

+ New Import Runner My Works

POST MyRequest POST MyRequest

MyRequest

POST http://localhost:8080/addAuthorBook

Params Authorization Headers (11) Body Pre-req

Headers Hide auto-generated headers

Content-Type: application/json  
Host: localhost  
Content-Length: 100

Body Cookies Headers (3) Test Results

Pretty Raw Preview Visualize Text

1 John has written Book about dogs

Find and Replace Console

Request JSON Body (raw) & Response Body

Postman

File Edit View Help

+ New Import Runner My Works

POST MyRequest POST MyRequest

MyRequest

POST http://localhost:8080/addAuthorBook

Params Authorization Headers (11) Body Pre-req

none form-data x-www-form-urlencoded raw

1 {  
2 "name": "John",  
3 "age" : 20,  
4 "title": "Book about dogs"  
5 }

Body Cookies Headers (3) Test Results

Pretty Raw Preview Visualize Text

1 John has written Book about dogs

Find and Replace Console

## *Application Structure*

The screenshot shows the IntelliJ IDEA interface with the following details:

- File Menu:** File, Edit, View, Navigate, Code, Analyze, Refactor, Build, Run, Tools, VCS, Window, Help.
- Title Bar:** springboot\_dto\_modelmapper - MyController.java
- Toolbars:** Springboot Dto ModelMapper Application, Git.
- Project Tree (Left):** Project, springboot\_dto\_modelmapper (C:\Projects\springboot\_dto\_modelmapper), src, main, java, com, ivoro, .idea, .mvn, pom.xml (springboot\_dto\_modelmapper), controllers, MyController, DTOs, AuthorBookDTO, entities, Author, Book, SpringbootDtoModelmapperApplication, resources, test, target, .gitignore, HELP.md, mvnw, mvnw.cmd.
- Code Editor (Right):** The current file is MyController.java, which contains the following code:

```
11  @Controller
12
13  public class MyController {
14
15      @ResponseBody
16      @RequestMapping("addAuthorBook")
17      public String addAuthorBook(@RequestBody AuthorBookDTO authorBookDTO) {
18
19          //INSTANTIATE MODEL MAPPER
20          ModelMapper modelMapper = new ModelMapper();
21
22          //CONVERT DTOs TO ENTITIES
23          Author author = modelMapper.map(authorBookDTO, Author.class);
24          Book book = modelMapper.map(authorBookDTO, Book.class);
25
26          //RETURN SOMETHING
27          return author.name + " has written: " + book.title;
28
29      }
30
31  }
```
- Sidebar (Right):** TEPMON, Ant, Database, Maven.
- Bottom Navigation:** Git, Run, TODO, Problems, Terminal, Build, Java Enterprise, Spring, Event Log.
- Status Bar:** All files are up-to-date (2 minutes ago), 14:1, CRLF, UTF-8, 2 spaces, master.

pom.xml

```
<dependencies>

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

    <dependency>
        <groupId>org.modelmapper</groupId>
        <artifactId>modelmapper</artifactId>
        <version>2.3.9</version>
    </dependency>

</dependencies>
```

## 4.8.4 DTO to Entity - Model Mapper - Custom

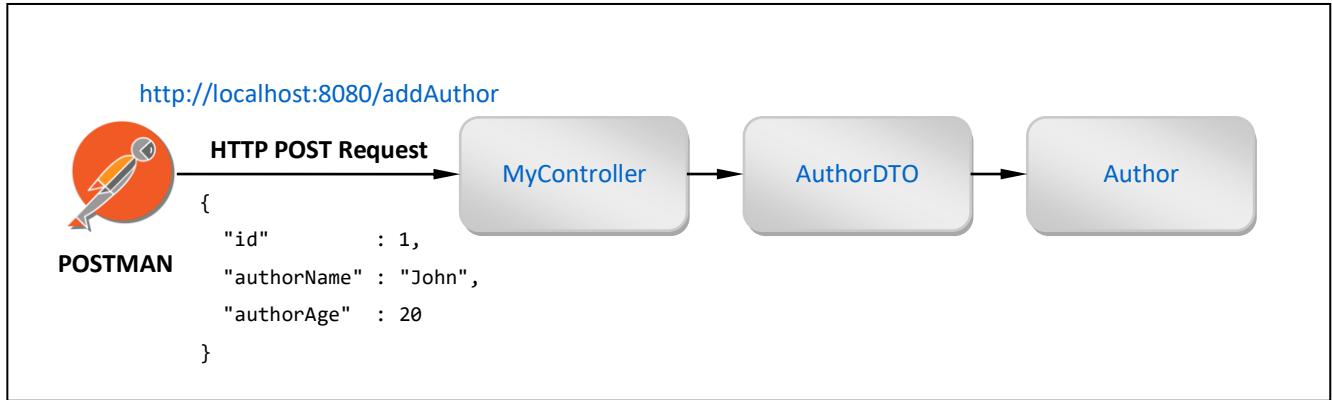
### Info

[G]

- This tutorial shows how to **customize Model Mapper** by specifying how to map Properties from DTO to Entity.
- Customization works only in **one direction** - to customize mappings from Entity to DTO you need another Model Mapper.
- Model Mapper will continue to use **default mappings for other Properties** - for which customization was not specified.

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables @Controller and @RequestMapping. Includes Tomcat Server.

## Procedure

- **Create Project:** `springboot_dto_modelmapper_custommapper` (add Spring Boot Starters from the table)
- **Edit File:** `pom.xml` (manually add Model Mapper dependency)
- **Create Package:** `entities` (inside main package)
  - **Create Class:** `Author.java` (inside package entities)
- **Create Package:** `DTOs` (inside main package)
  - **Create Class:** `AuthorDTO.java` (inside package controllers)
- **Create Package:** `controllers` (inside main package)
  - **Create Class:** `MyController.java` (inside package controllers)

*pom.xml*

```
<dependency>
  <groupId>org.modelmapper</groupId>
  <artifactId>modelmapper</artifactId>
  <version>2.3.9</version>
</dependency>
```

*Author.java*

```
package com.ivoronline.springboot_dto_modelmapper_custommapper.entities;

public class Author {

    //PROPERTIES
    public Integer id;
    public String name;
    public Integer age;

    //SETTERS
    public void setId (Integer id) { this.id = id; }
    public void setName(String name) { this.name = name; }
    public void setAge (Integer age) { this.age = age; }

}
```

*AuthorDTO.java*

```
package com.ivoronline.springboot_dto_modelmapper_custommapper.DTOs;

public class AuthorDTO {

    //PROPERTIES
    public Integer id;
    public String authorName;
    public Integer authorAge;

    //GETTERS
    public Integer getId () { return id; }
    public String getAuthorName() { return authorName; }
    public Integer getAuthorAge () { return authorAge; }

}
```

### MyController.java

```
package com.ivorononline.springboot_dto_modelmapper_customMapper.controllers;

import com.ivorononline.springboot_dto_modelmapper_customMapper.DTOs.AuthorDTO;
import com.ivorononline.springboot_dto_modelmapper_customMapper.entities.Author;
import org.modelmapper.TypeMap;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;
import org.modelmapper.ModelMapper;

@Controller
public class MyController {

    @ResponseBody
    @RequestMapping("addAuthor")
    public String addAuthor(@RequestBody AuthorDTO authorDTO) {

        //CUSTOMIZE MODEL MAPPER
        ModelMapper authorDTOEntity = new ModelMapper();
        TypeMap<AuthorDTO, Author> typeMap = authorDTOEntity.typeMap(AuthorDTO.class, Author.class);
        typeMap.addMappings(mapper -> {
            mapper.map(AuthorDTO::getAuthorName, Author::setName); //src -> src.getAuthorName()
            mapper.map(AuthorDTO::getAuthorAge, Author::setAge); //src -> src.getAuthorAge ()
        });

        //CONVERT AUTHORDTO TO AUTHOR
        Author author = authorDTOEntity.map(authorDTO, Author.class);

        //RETURN SOMETHING
        return author.id + ": " + author.name + " is " + author.age + " years old";
    }
}
```

## Results

- Start Postman

POST

```
http://localhost:8080/addAuthor
```

Headers

(add Key-Value)

```
Content-Type: application/json
```

Body

(option: raw)

```
{  
    "id" : 1,  
    "authorName" : "John",  
    "authorAge" : 20  
}
```

HTTP Response Body

```
1: John is 20 years old
```

HTTP Request Headers (Bulk Edit)

The screenshot shows the Postman interface with a request titled "MyRequest". The method is set to "POST" and the URL is "http://localhost:8080/addAuthor". The "Headers" tab is selected, showing the following headers:

```
Content-Type: application/json  
Host: localhost  
Content-Length: 100
```

The "Body" tab is selected, showing the JSON payload:

```
1: John is 20 years old
```

At the bottom, there are tabs for "Pretty", "Raw", "Preview", "Visualize", and "Text".

Request JSON Body (raw) & Response Body

The screenshot shows the Postman interface with a request titled "MyRequest". The method is set to "POST" and the URL is "http://localhost:8080/addAuthor". The "Body" tab is selected, showing the JSON payload. The "Body" dropdown menu is open, with "raw" selected.

```
1: {  
    "id" : 1,  
    "authorName" : "John",  
    "authorAge" : 20  
}
```

The "Body" tab is selected, showing the JSON payload:

```
1: John is 20 years old
```

At the bottom, there are tabs for "Pretty", "Raw", "Preview", "Visualize", and "Text".

## Application Structure

The screenshot shows the IntelliJ IDEA interface with the following details:

- Project Tree:** The project is named "springboot\_dto\_modelmapper\_custommapper". It contains a ".idea" folder, a ".mvn" folder, and a "src" folder. The "src" folder has "main" and "test" subfolders. "main/java" contains "com.ivoronline.springboot\_dto\_modelmapper.controllers" (which has "MyController.java") and "com.ivoronline.springboot\_dto\_modelmapper.entities" (which has "Author.java" and "AuthorDTO.java"). "resources" and "target" folders are also present.
- Code Editor:** The file "MyController.java" is open. The code defines a controller with a POST endpoint for adding an author. It uses ModelMapper to map AuthorDTO to Author entities. The code includes comments for customizing the model mapper and converting AuthorDTO to Author.
- Pom.xml:** The pom.xml file is visible in the project tree under "src/main/resources".
- Toolbars and Status Bar:** The status bar at the bottom shows "Build completed successfully in 1 s 923 ms (moments ago)".
- Right Panel:** The right panel shows various tool tabs: TCPMON, Ant, Database, and Maven.

## pom.xml

```
<dependencies>

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

    <dependency>
        <groupId>org.modelmapper</groupId>
        <artifactId>modelmapper</artifactId>
        <version>2.3.9</version>
    </dependency>

</dependencies>
```

## 4.8.5 DTO to Entity - Model Mapper - Exclude

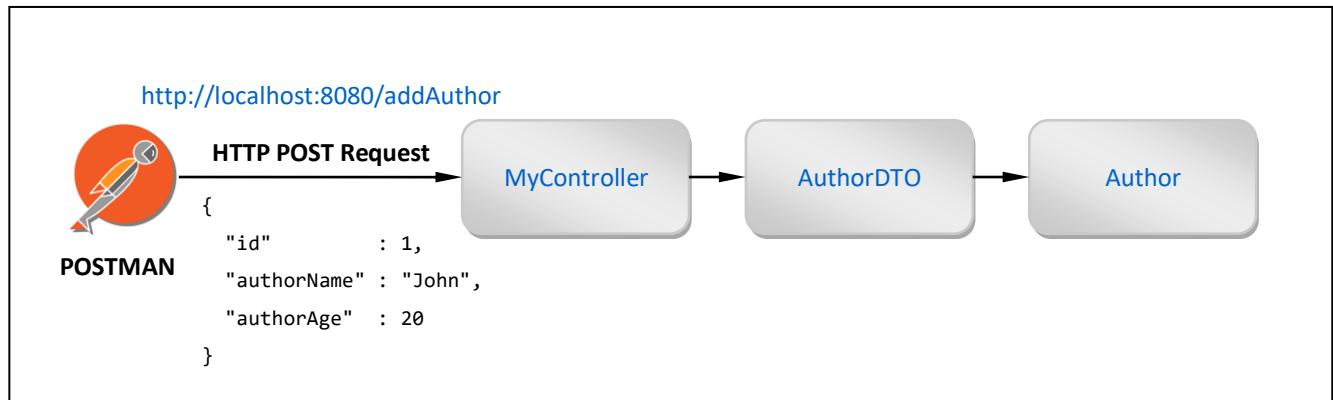
### Info

[G] [R]

- This tutorial shows how to customize Model Mapper to **skip destination Property**.
- In this tutorial we will skip settings Author's `age` Resulting in a null value when we return it from Controller.

### Application Schema

[Results]



### Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables @Controller and @RequestMapping. Includes Tomcat Server.

## Procedure

- Create Project: `springboot_dto_modelmapper_skip` (add Spring Boot Starters from the table)
- Edit File: `pom.xml` (manually add Model Mapper dependency)
- Create Package: `entities` (inside main package)
  - Create Class: `Author.java` (inside package `entities`)
- Create Package: `DTOs` (inside main package)
  - Create Class: `AuthorDTO.java` (inside package `controllers`)
- Create Package: `controllers` (inside main package)
  - Create Class: `MyController.java` (inside package `controllers`)

*pom.xml*

```
<dependency>
  <groupId>org.modelmapper</groupId>
  <artifactId>modelmapper</artifactId>
  <version>2.3.9</version>
</dependency>
```

*Author.java*

```
package com.ivoronline.springboot_dto_modelmapper_skip.entities;

public class Author {

    //PROPERTIES
    public Integer id;
    public String name;
    public Integer age;

    //SETTERS
    public void setId (Integer id) { this.id = id; }
    public void setName(String name) { this.name = name; }
    public void setAge (Integer age) { this.age = age; }

}
```

*AuthorDTO.java*

```
package com.ivoronline.springboot_dto_modelmapper_skip.DTOs;

public class AuthorDTO {

    //PROPERTIES
    public Integer id;
    public String authorName;
    public Integer authorAge;

    //GETTERS
    public Integer getId () { return id; }
    public String getAuthorName() { return authorName; }
    public Integer getAuthorAge () { return authorAge; }

}
```

### MyController.java

```
package com.ivorononline.springboot_dto_modelmapper_skip.controllers;

import com.ivorononline.springboot_dto_modelmapper_skip.DTOs.AuthorDTO;
import com.ivorononline.springboot_dto_modelmapper_skip.entities.Author;
import org.modelmapper.TypeMap;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;
import org.modelmapper.ModelMapper;

@Controller
public class MyController {

    @ResponseBody
    @RequestMapping("addAuthor")
    public String addAuthor(@RequestBody AuthorDTO authorDTO) {

        //CUSTOMIZE MODEL MAPPER
        ModelMapper authorDTOEntity = new ModelMapper();
        TypeMap<AuthorDTO, Author> typeMap = authorDTOEntity.typeMap(AuthorDTO.class, Author.class);
        typeMap.addMappings(mapper -> {
            mapper.map(AuthorDTO::getAuthorName, Author::setName); //CUSTOM MAPPING
            mapper.skip(Author::setAge); //SKIP DESTINATION PROPERTY
        });

        //CONVERT AUTHORDTO TO AUTHOR
        Author author = authorDTOEntity.map(authorDTO, Author.class);

        //RETURN SOMETHING
        return author.id + ": " + author.name + " is " + author.age + " years old";
    }
}
```

## Results

- Start Postman

*POST*

```
http://localhost:8080/addAuthor
```

*Headers*

(add Key-Value)

```
Content-Type: application/json
```

*Body*

(option: raw)

```
{
  "id" : 1,
  "authorName" : "John",
  "authorAge" : 20
}
```

*HTTP Response Body*

```
1: John is 20 years old
```

*HTTP Request Headers (Bulk Edit)*

Postman

File Edit View Help

+ New Import Runner My Works

POST MyRequest

MyRequest

POST http://localhost:8080/addAuthor

Params Authorization Headers (11) Body Pre-req

Headers Hide auto-generated headers

Content-Type: application/json  
Host: localhost  
Content-Length: 100

Body Cookies Headers (3) Test Results

Pretty Raw Preview Visualize Text

1 1: John is 20 years old

Find and Replace Console

*Request JSON Body (raw) & Response Body*

Postman

File Edit View Help

+ New Import Runner My Works

POST MyRequest

MyRequest

POST http://localhost:8080/addAuthor

Params Authorization Headers (11) Body Pre-req

none form-data x-www-form-urlencoded raw

1 {
 "id" : 1,
 "authorName" : "John",
 "authorAge" : 20
}

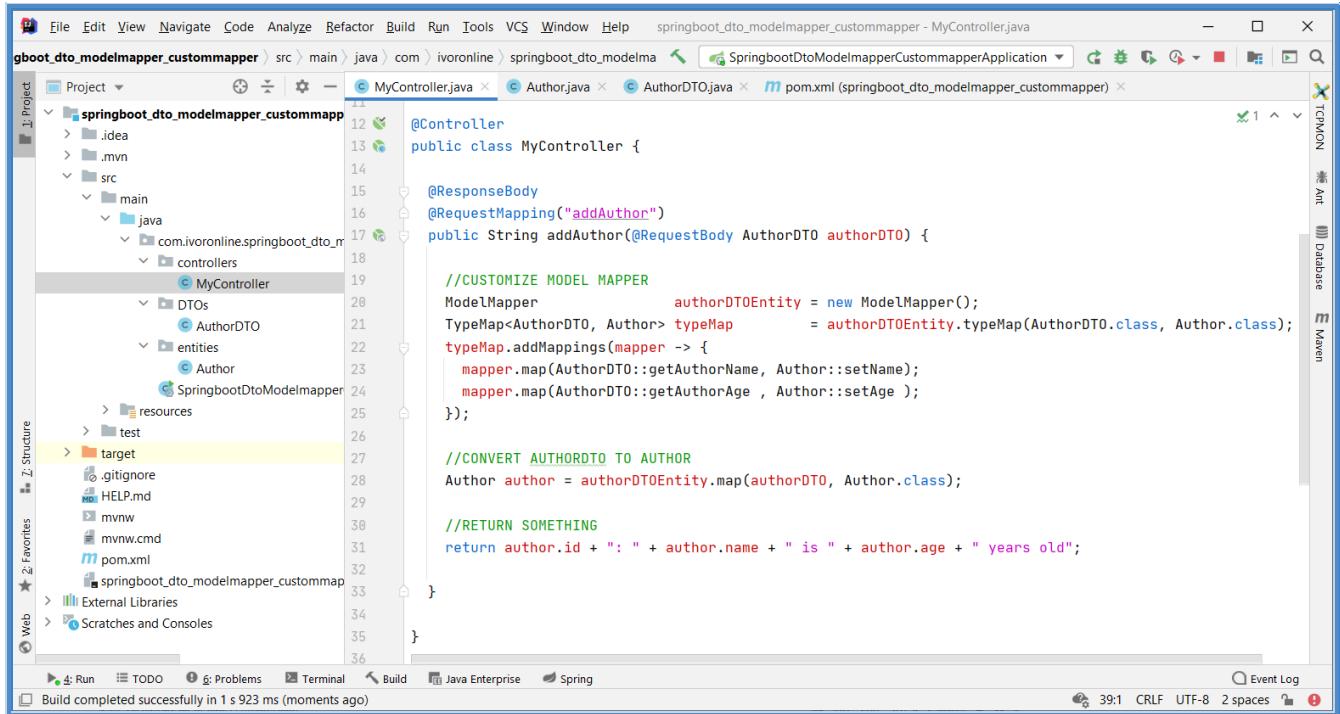
Body Cookies Headers (3) Test Results

Pretty Raw Preview Visualize Text

1 1: John is 20 years old

Find and Replace Console

## Application Structure



## pom.xml

```
<dependencies>

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

    <dependency>
        <groupId>org.modelmapper</groupId>
        <artifactId>modelmapper</artifactId>
        <version>2.3.9</version>
    </dependency>

</dependencies>
```

# 4.9 Mapper - Model Mapper vs JMapper vs Map Struct

## Info

---

- **Model Mapper** doesn't need any configuration if Properties have the same or similar names.  
But any additional customization must be done using complicated API.
- **JMapper** can't automatically map Properties by just matching their names.  
Instead Destination Properties MUST be Annotated.  
But any additional customization can be simply done again using Annotations (without coding).  
This makes JMapper easier to use as long as any customization is needed.  
JMapper also supports customization through API or XML.
- **Map Struct** generates Class that implements Interface that you define.  
Inside Interface you declare methods that map between objects.  
If Property names are the same mapping will be done automatically.  
If Property names differ you can use Annotations to simply define additional mappings.  
Map Struct benefit is that it doesn't change Entities but still uses simple Annotations inside Interface to define mappings.

# 5 Errors

## Info

---

- Following tutorials show how to resolve common Errors.

# 5.1 Controller

## Info

---

- Following tutorials show how to resolve Controller related Errors.

## 5.1.1 Circular view path

### Error Message

---

#### Error Message

```
javax.servlet.ServletException: Circular view path [GetPerson]:  
would dispatch back to the current handler URL [/GetPerson] again. Check your ViewResolver setup!  
(Hint: This may be the result of an unspecified view, due to default view name generation.)
```

### Reason

---

- This error occurs if you intend to return JSON from the Endpoint but forget to use `@ResponseBody` Annotation.

### Solution

---

- Add Green Line.

#### MyController.java

```
@Controller  
public class MyController {  
  
    @ResponseBody  
    @RequestMapping("/GetPerson")  
    public Person getPerson() {  
  
        //CREATE PERSON  
        Person person      = new Person();  
        person.id       = 1;  
        person.name     = "John";  
        person.age      = 20;  
  
        //RETURN PERSON  
        return person;  
    }  
}
```

## 5.2 Database

### Info

---

- Following tutorials show how to resolve Database related Errors.

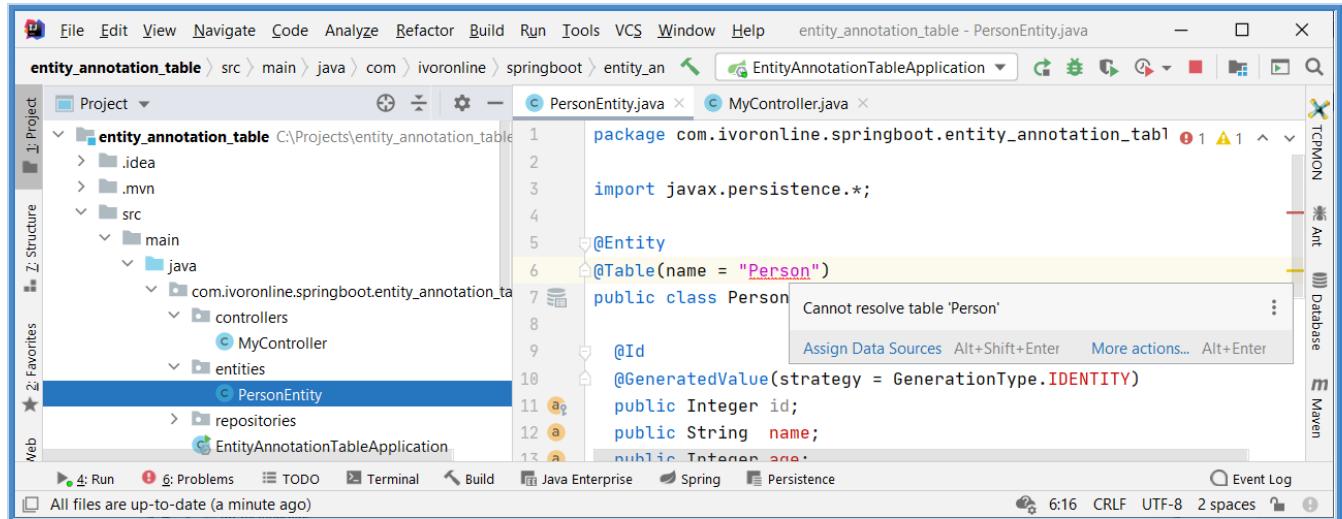
## 5.2.1 Cannot Resolve Table

### Error Message

#### Error Message

```
Cannot resolve table 'AUTHOR'
```

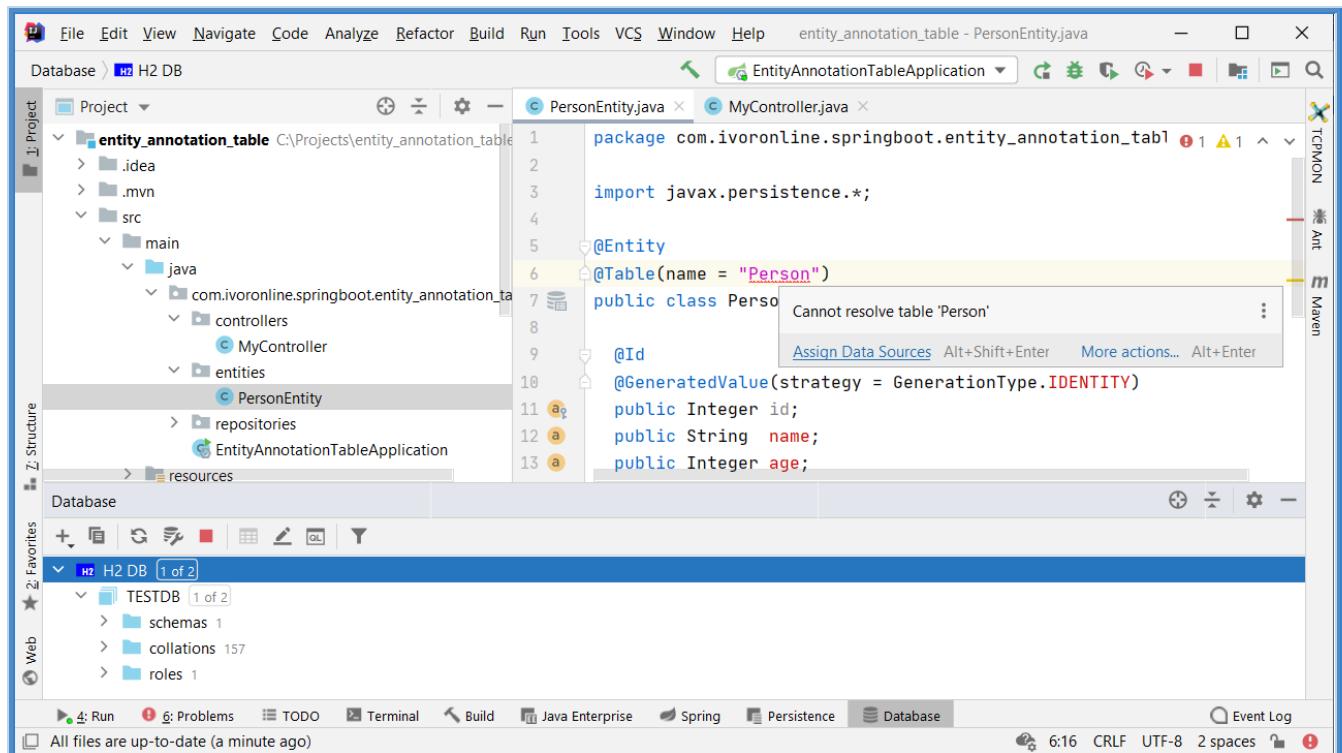
#### Error Message



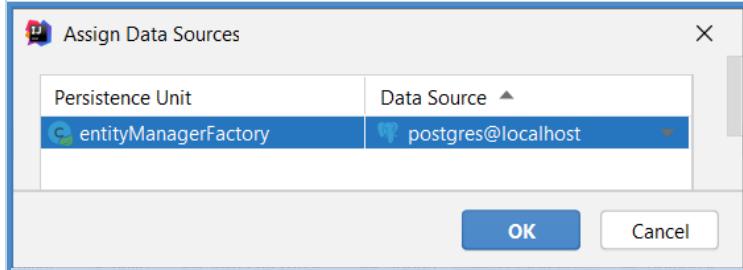
### Solution

- View
- Tool Window
- Persistence
- RC on BookEntity
- Assign Data Source
- postgres@localhost

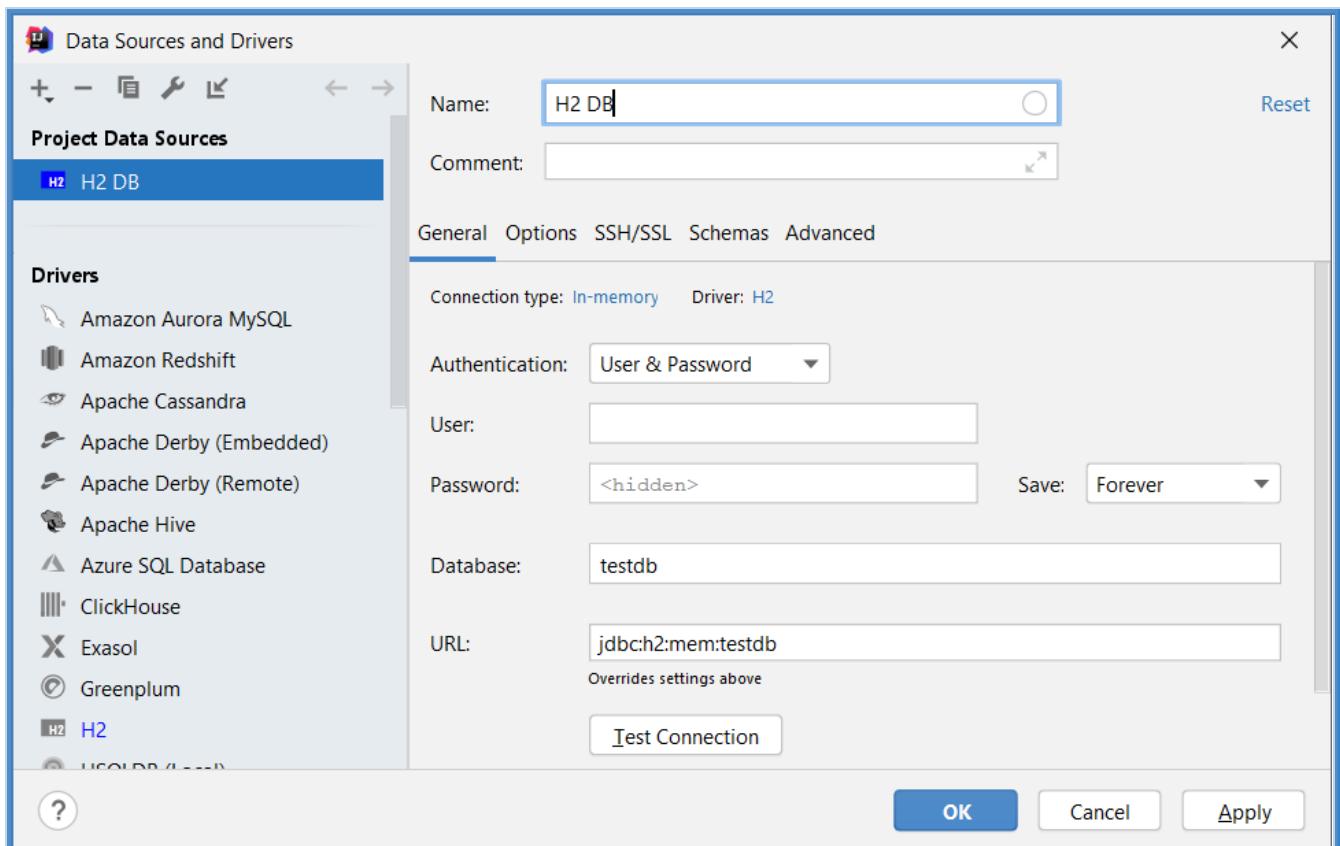
#### Assign Data Source



postgres@localhost



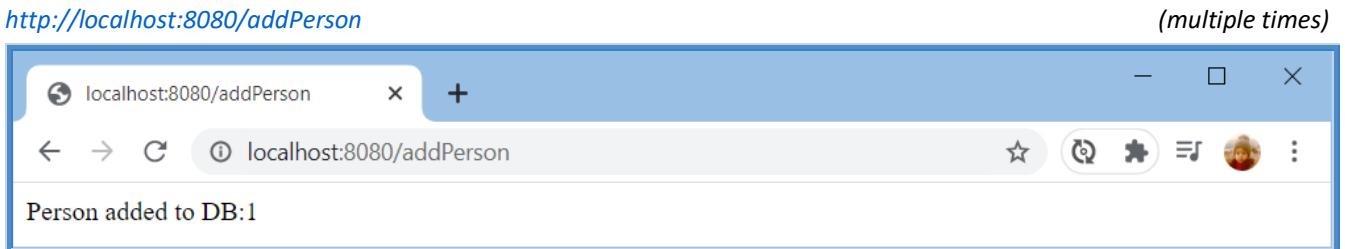
## Create DB Connection



## 5.2.2 Records are not inserted into DB

### Error Description

- When trying to insert multiple Records into the same DB Table, only the first Record is inserted.



Only first Record gets inserted

	id	age	name
1	1	20	John

### Reason

- This error occurs if you `@Autowired Entity` inside the Controller.
- This way the same Entity is always used and updated in the DB - instead of creating/inserting new one each time.

### Solution

- Remove Red Line- do not `@Autowired Entity` inside Controller/Service.
- Add Green Line - create Entity instance inside the Endpoint Method.

*MyController.java*

```
@Controller
public class AuthorController {

    @Autowired Author author;
    @Autowired AuthorRepository authorRepository;

    //=====
    // ADD AUTHOR
    //=====

    @ResponseBody
    @RequestMapping("/AddAuthor")
    public String addAuthor(@RequestParam String name, @RequestParam Integer age) {

        //CREATE AUTHOR
        Author author = new Author();
        author.setName(name);
        author.setAge(age);

        //STORE AUTHOR
        authorRepository.save(author);

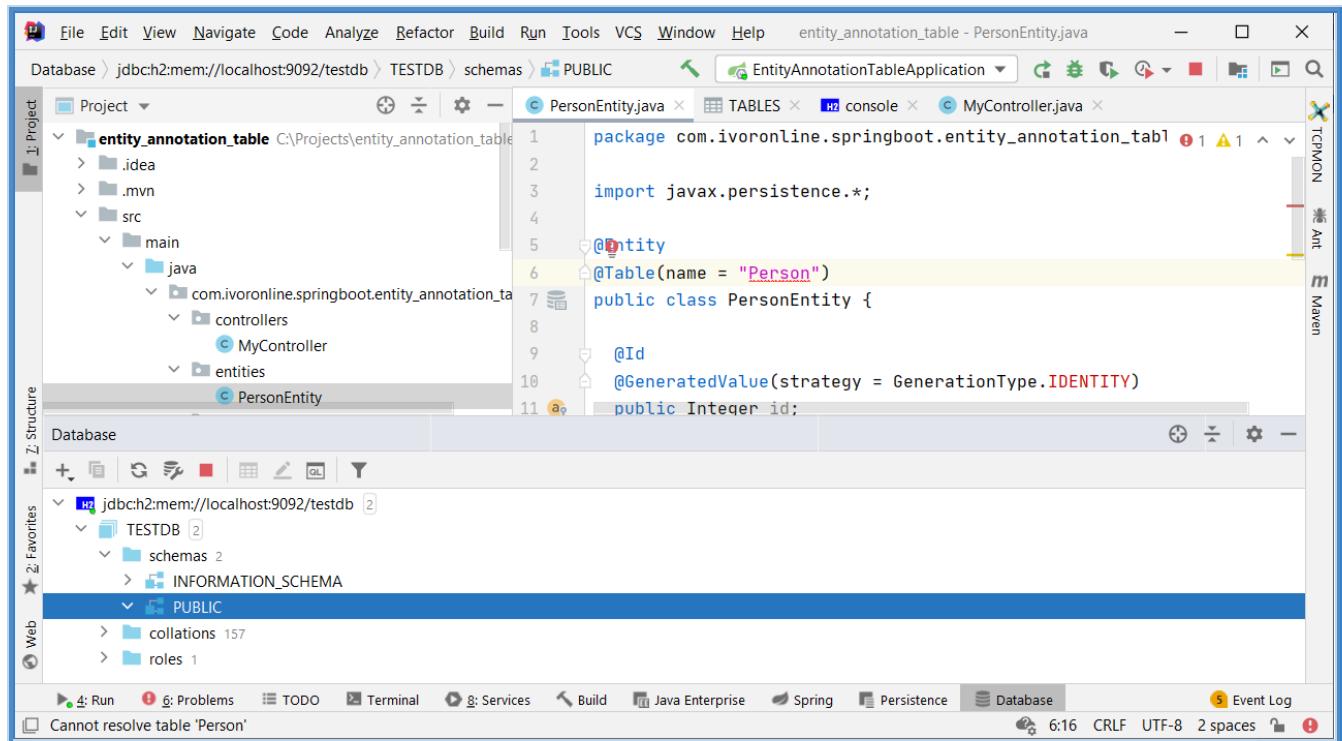
        //RETURN
        return "Author added to DB";
    }
}
```

## 5.2.3 Database Tool Window can't see H2 Tables

### Error Description

- You can create connection to H2 DB but no tables created by your Spring Boot Application will be visible.

*There are no Tables*



### Reason

[R]

- IntelliJ Database Window can't normally be used to connect to in-memory H2 Database (without additional workarounds)