
SPRING
INITIALIZR

What is Spring Initializr?

1

Spring initializr is a website or web-based tool that can be used to set up a Spring Boot project.

2

Of course, you can set up a Spring Boot project without using the Spring initializr, but the advantage of using the Spring initializr is that it speeds up the process and does most of the groundwork for you.

3

All you have to do is to go to start.spring.io and add the spring boot starter dependencies that you want, (eg: web, JPA and H2) and generate the project!

Spring initialzr will do the following:

- Creates the build file with all the required dependencies, plugins and also takes care of the versions.
- Creates the project structure or the folder structure for your project.

Now you can start implementing, and yes you can do any customizations to the build file or the project setup as much as you like.


spring
boot

What is Spring Boot?

Spring Boot is an opinionated, easy to get-started addition to the Spring platform – highly useful for creating stand-alone, production-grade applications with minimum effort.

Spring Boot is a project that is built on the top of the Spring Framework. It provides an easier and faster way to set up, configure, and run both simple and web-based applications.

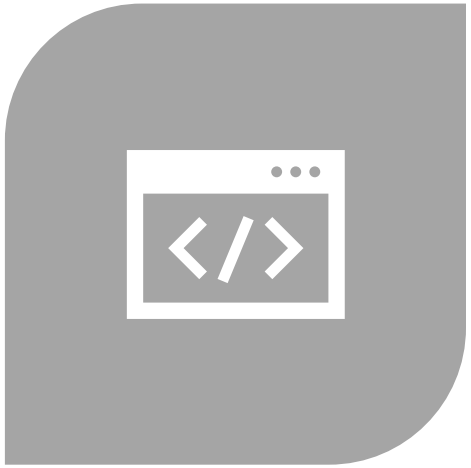
What is Spring Boot?

It is a Spring module that provides the **RAD (*Rapid Application Development*)** feature to the Spring Framework. It is used to create a stand-alone Spring-based application that you can just run because it needs minimal Spring configuration.

Spring Boot is the combination of **Spring Framework** and **Embedded Servers**.

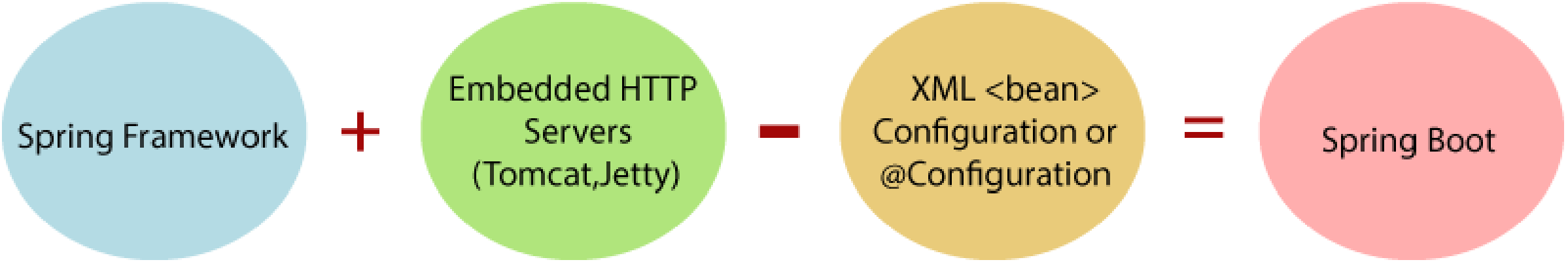
What is Spring Boot?

IN SPRING BOOT, THERE IS NO REQUIREMENT FOR XML CONFIGURATION (DEPLOYMENT DESCRIPTOR).



IT USES ***CONVENTION OVER CONFIGURATION*** SOFTWARE DESIGN PARADIGM THAT MEANS IT DECREASES THE EFFORT OF THE DEVELOPER.

What is Spring Boot?



Advantages Of Spring Boot

It creates **stand-alone** Spring applications that can be started using Java **-jar**.

It tests web applications easily with the help of different **Embedded** HTTP servers such as **Tomcat, Jetty,** etc. We don't need to deploy WAR files.

It provides opinionated '**starter**' POMs to simplify our Maven configuration.

Advantages Of Spring Boot

It provides **production-ready** features such as **metrics, health checks,** and **externalized configuration**.

There is no requirement for **XML** configuration.

It offers a **CLI** tool for developing and testing the Spring Boot application.

Advantages Of Spring Boot

It offers the number of **plug-ins**.

It also minimizes writing multiple **boilerplate codes** (the code that has to be included in many places with little or no alteration), XML configuration, and annotations.

It **increases productivity** and reduces development time.

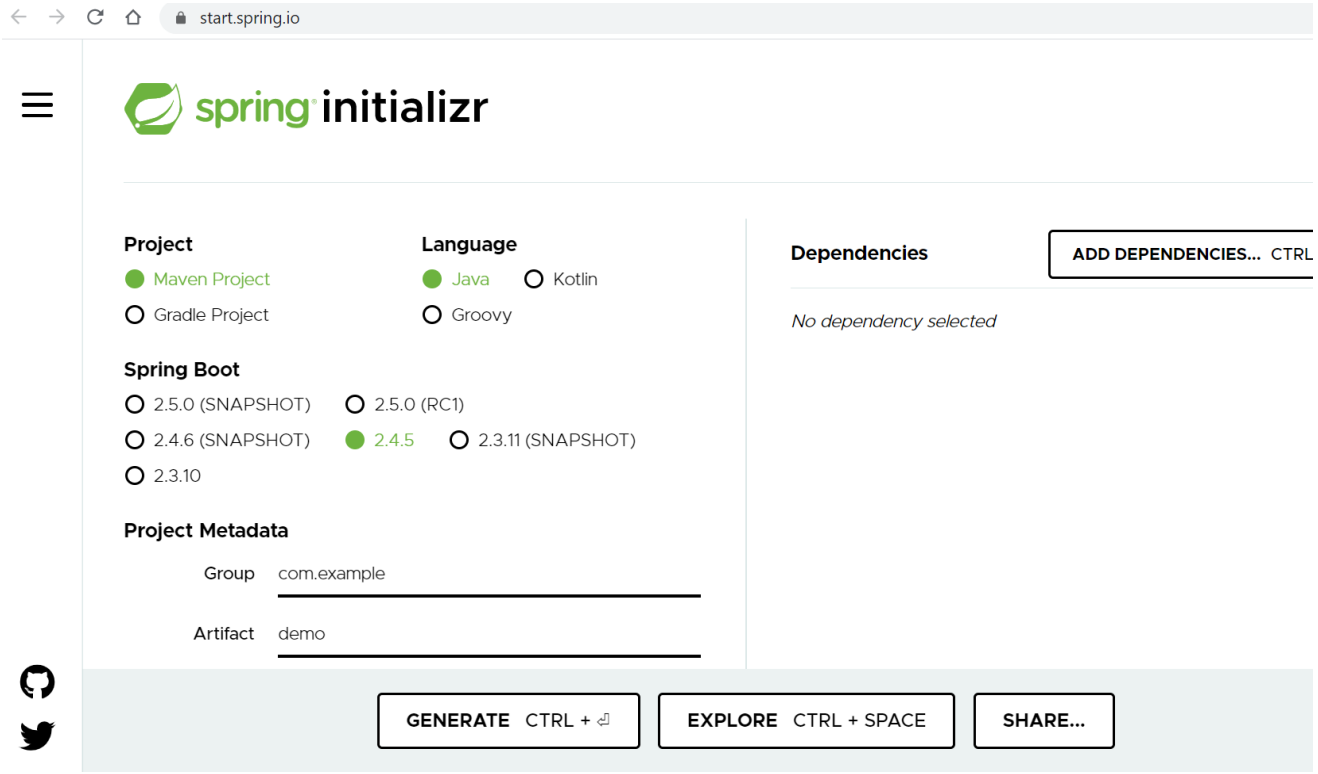
Limitations of Spring Boot

Spring Boot can use dependencies that are not going to be used in the application.

These dependencies increase the size of the application.

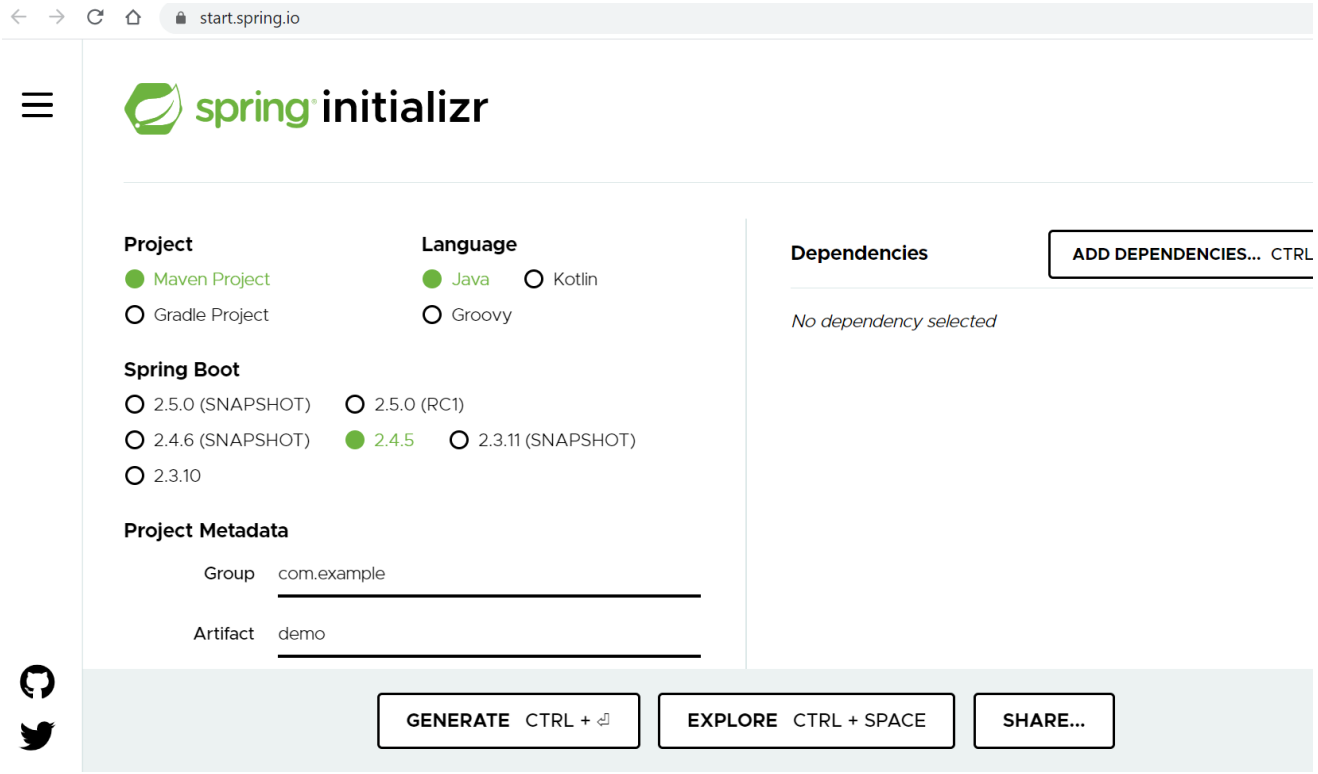
How to use the Spring Initializr to set up a Spring Boot Project

Spring Initializr – Project Setup



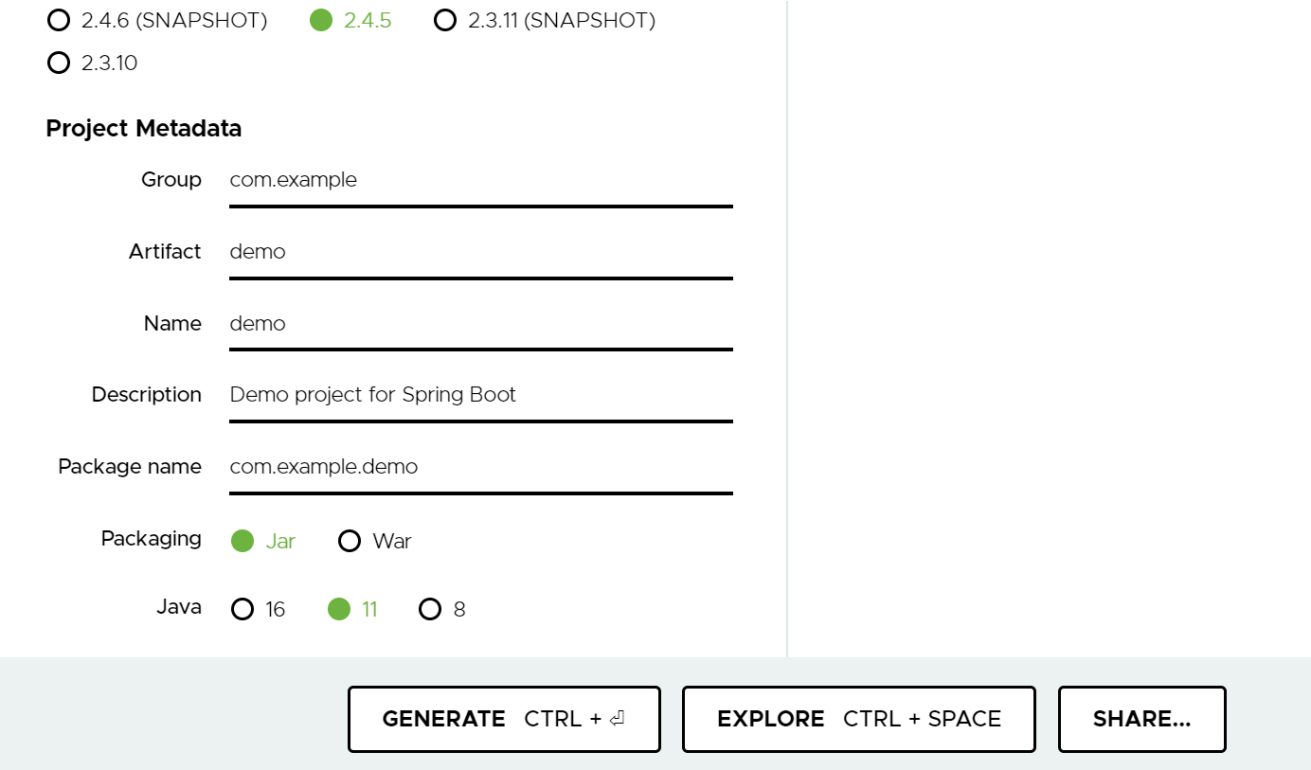
- Go to the [Spring Initializr website](). Here's what it looks like at the moment.

Spring Initializr – Project Setup



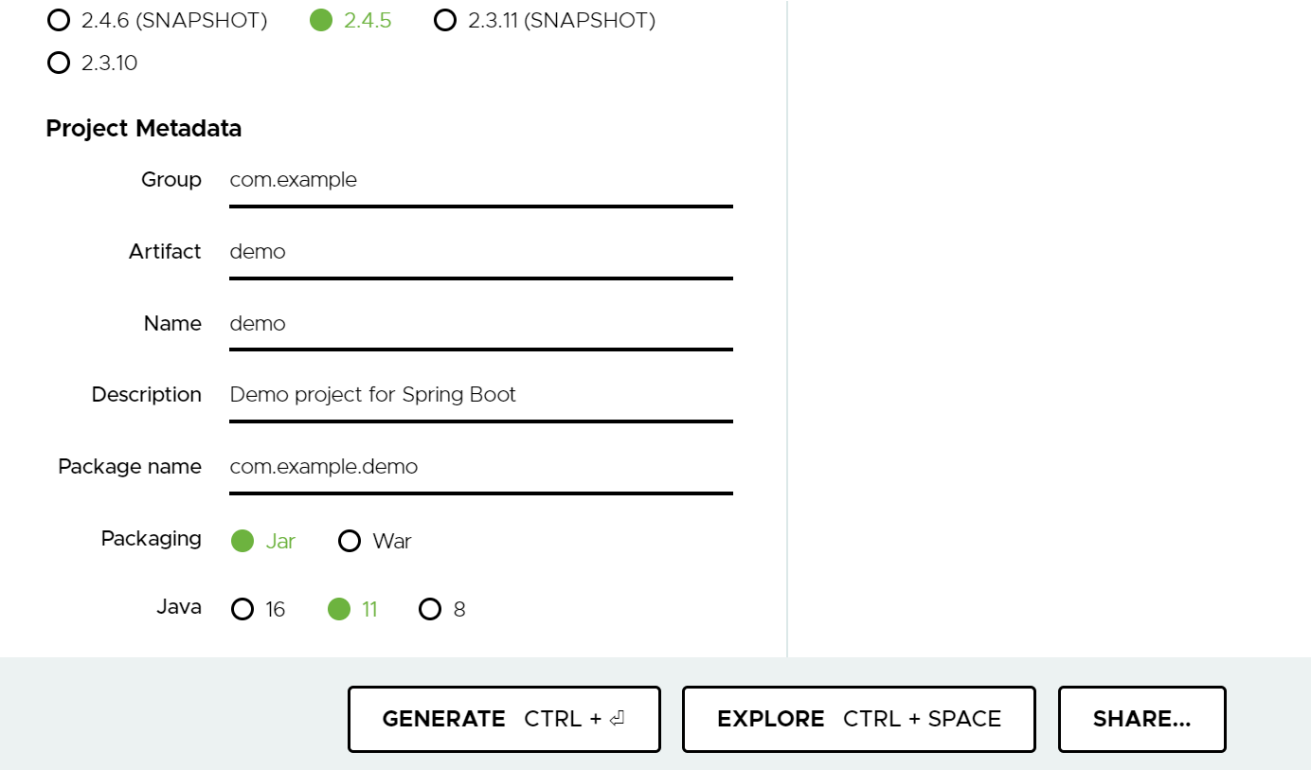
- Select the build tool you want to use. Maven is selected by default.

Spring Initializr – Project Setup



Then you can select the **programming language** you use (Java is selected by default), the **Spring Boot version** you want to use (the latest release version is selected by default), the **project metadata** (you can keep these as it is and change later), the **packaging** of the project (i.e. either jar of war), and a specific **Java version** if you want (Java 11 is selected by default).

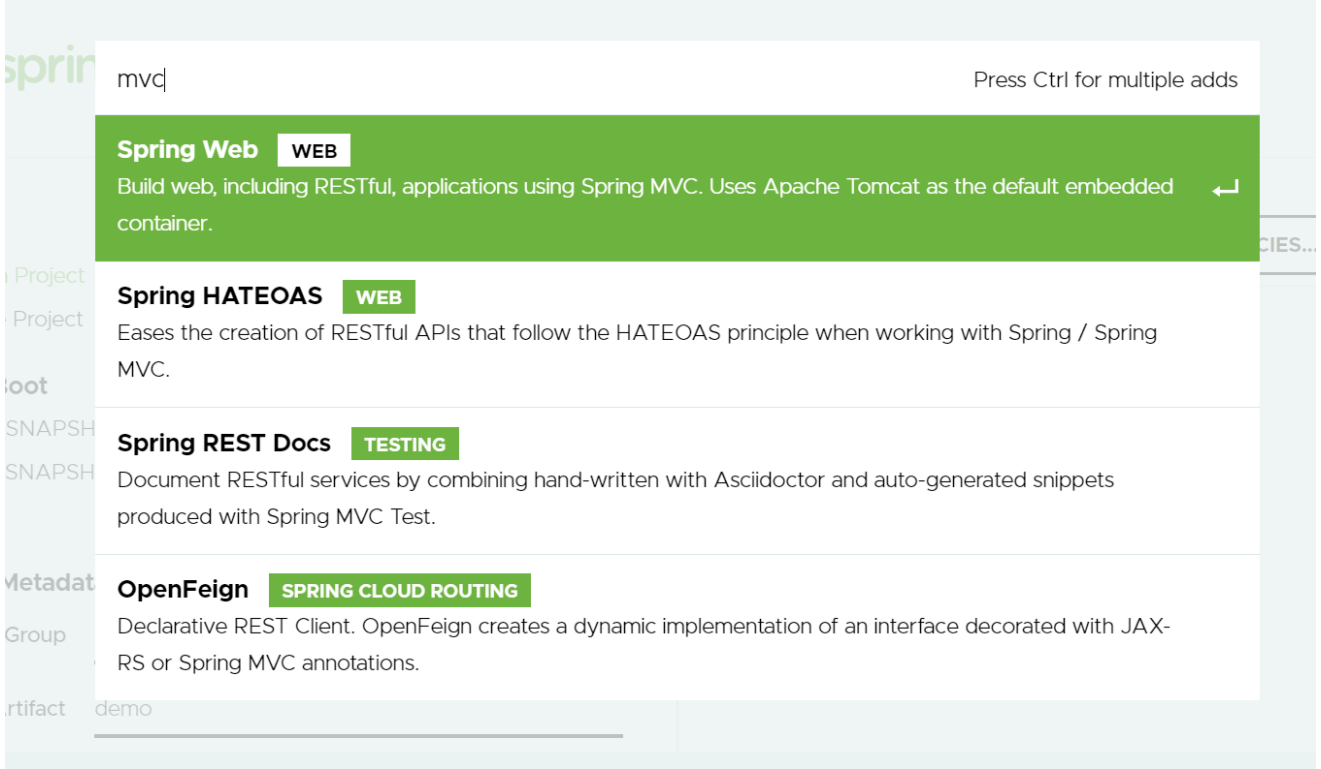
Spring Initializr – Project Setup



Add dependencies - Click on the Add dependencies button and a popup list of dependencies will appear.

At the top you can search for dependencies you want. As you type in, a list of matching dependencies will appear down below.

Spring Initializr – Project Setup



For example, if I want to create a MVC app, I would search for MVC and the first dependency which appears is **Spring Web**. That's what we want in this case.

Spring Initializr – Project Setup

You can select the one you want, and press enter or click on it to add it to the project.

Spring Initializr – Project Setup

Generate Project - Click on the Generate button to download the zip file.

Lab – Create a spring boot project

- Create a spring boot project using spring initializr.
- With Maven as build tool.
- Java as language.
- Jar as packaging.
- With ***mvc*** and ***jpa*** and ***thymleaf*** as dependencies.
- (Google out what ***jpa*** and ***thymleaf*** dependencies does?)

Spring View Technologies

- The Spring web framework is built around the MVC (Model-View-Controller) pattern, which makes it easier to separate concerns in an application. This allows for the possibility to use different view technologies, from the well established **JSP** technology to a variety of **template engines**.

Spring View Technologies

- Given that concerns in a Spring MVC application are cleanly separated switching from one view technology to another is primarily a matter of configuration.
- To render each view type, we need to define a ***ViewResolver*** bean corresponding to each technology.
- This means that we can then return the view names from ***@Controller*** mapping methods in the same way we usually return **JSP** files.

Spring View Technologies

- Main template engines that can be used with Spring: ***Thymeleaf***, ***Groovy***, ***FreeMarker, Jade***.
- In the following sections, we're going to go over more traditional technologies like ***Java Server Pages***, as well as the widely used template engine: ***Thymeleaf***.
- For each of these, we will go over the configuration necessary in an application built using standard Spring and *Spring Boot*.

Java Server Pages - JSP

- JSP is one of the most popular view technologies for Java applications, and it is supported by Spring out-of-the-box. For rendering JSP files, a commonly used type of *ViewResolver* bean is *InternalResourceViewResolver*:

```
@EnableWebMvc
@Configuration
public class ApplicationConfiguration implements WebMvcConfigurer {
  @Bean
  public ViewResolver jspViewResolver() {
    InternalResourceViewResolver bean = new InternalResourceViewResolver();
    bean.setPrefix("/WEB-INF/views/");
    bean.setSuffix(".jsp");
    return bean;
  }
}
```

- Next, we can start creating JSP files in the */WEB-INF/views* location

Java Server Pages - JSP

- If we are adding the files to a ***Spring Boot*** application, then instead of in the *ApplicationConfiguration* class, we can define the following properties in an *application.properties* file:

    - spring.mvc.view.prefix: /WEB-INF/views/
    - spring.mvc.view.suffix: .jsp

- Based on these properties, *Spring Boot* will auto-configure the necessary *ViewResolver*.

Thymeleaf

- *Thymeleaf* is a Java template engine which can process HTML, XML, text, JavaScript or CSS files.
- Unlike other template engines, *Thymeleaf* allows using templates as prototypes, meaning they can be viewed as static files.

Thymeleaf – Maven Dependencies

- To integrate Thymeleaf with Spring, we need to add the *thymeleaf* and *thymeleaf-spring5* dependencies:

```
<dependency>
  <groupId>org.thymeleaf</groupId>
  <artifactId>thymeleaf</artifactId>
  <version>3.0.11.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.thymeleaf</groupId>
  <artifactId>thymeleaf-spring5</artifactId>
  <version>3.0.11.RELEASE</version>
</dependency>
```

Thymeleaf – Spring Config

- Next, we need to add the configuration which requires a *SpringTemplateEngine* bean, as well as a *TemplateResolver* bean that specifies the location and type of the view files.

Thymeleaf – Spring Config

```
@Bean
public SpringTemplateEngine templateEngine() {
  SpringTemplateEngine templateEngine = new
SpringTemplateEngine();


templateEngine.setTemplateResolver(thymeleafTemplateResol
ver());
  return templateEngine;
}

@Bean
public SpringResourceTemplateResolver
thymeleafTemplateResolver() {
  SpringResourceTemplateResolver templateResolver
          = new SpringResourceTemplateResolver();
  templateResolver.setPrefix("/WEB-INF/views/");
  templateResolver.setSuffix(".html");
  templateResolver.setTemplateMode("HTML5");
  return templateResolver;
}
```

Thymeleaf – Spring Config

- Also, we need a *ViewResolver* bean of type *ThymeleafViewResolver*:

```
@Bean
public ThymeleafViewResolver thymeleafViewResolver() {
  ThymeleafViewResolver viewResolver = new ThymeleafViewResolver();
  viewResolver.setTemplateEngine(templateEngine());
  return viewResolver;
}
```

- Now we can add an HTML file in the *WEB-INF/views* location

Thymeleaf – Spring Boot Configuration

- *Spring Boot* will provide auto-configuration for *Thymeleaf* by adding the *spring-boot-starter-thymeleaf* dependency:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-thymeleaf</artifactId>
  <version>2.3.3.RELEASE</version>
</dependency>
```

- No explicit configuration is necessary. By default, HTML files should be placed in the *resources/templates* location.

JSP vs Thymeleaf

Both of them are view layers of Spring MVC. Firstly, the very basic difference is the file extensions (.jsp and .html).

JSP is not a template engine. It's compiled to the servlet and then the servlet is serving web content. On the other hand, Thymeleaf is a template engine which takes the ***HTML*** file, parses it and then produces web content which is being served.

JSP vs Thymeleaf

Thymeleaf is more like an HTML-ish view when you compare it with ***JSP*** views.

We can use prototype code in thymeleaf i.e we can view it in browser directly.

Since it is more HTML-ish code, thymeleaf codes are more readable (of course you can disrupt it and create unreadable codes, but at the end, it will be more readable when you compare it with ***JSP*** files.

JSP vs Thymeleaf

JSP uses **JSP Expression language**.

Thymeleaf uses **Spring Standard Dialect**(Also an expression language).

Standard Dialect is much more powerful than JSP Expression Language

BUT what was JSTL which we used in Earlier?

JSTL is an acronym that stands for JSP Standard Tag Library. JSTL allows you to program JSP pages using tags rather than the scriptlet code.

Following is the syntax to include the JSTL Core library in your JSP –

- <%@ taglib prefix = "c" uri = "http://java.sun.com/jsp/jstl/core" %>

```
<body>
   <c:set var = "salary" scope = "session" value = "${2000*2}"/>
   <c:if test = "${salary > 2000}">
      <p>My salary is:  <c:out value = "${salary}"/><p>
   </c:if>
</body>
```

```
<c:import var = "data" url = "http://www.tutorialspoint.com"/>
<c:out value = "${data}"/>
```

```
<c:redirect url = "http://www.photofuntoos.com"/>
```

```
<c:forEach var = "i" begin = "1" end = "5">
   Item <c:out value = "${i}"/><p>
</c:forEach>
```

Jstl - examples

Jstl - examples

- There are other tags as well, such as formatting tags.

```
<h3>Number Format:</h3>
<c:set var = "balance" value = "120000.2309" />

<p>Formatted Number (1): <fmt:formatNumber value = "${balance}"
   type = "currency"/></p>

<p>Formatted Number (2): <fmt:formatNumber type = "number"
   maxIntegerDigits = "3" value = "${balance}" /></p>
```

```
<h3>Number Format:</h3>
<c:set var = "now" value = "<% = new java.util.Date()%>" />

<p>Formatted Date (1): <fmt:formatDate type = "time"
   value = "${now}" /></p>
```

```
<fmt:setLocale value = "es_ES"/>
<fmt:bundle basename = "com.tutorialspoint.Example">
   <fmt:message key = "count.one"/><br/>
```

Want to know more?? Here are few helpful links.

- https://www.informit.com/articles/article.aspx?p=30334#:~:text=JSTL%20is%20an%20acronym%20that,JSP%20up%20to%20this%20point.
- https://www.tutorialspoint.com/jsp/jsp_standard_tag_library.htm
- https://www.javatpoint.com/EL-expression-in-jsp
- https://www.thymeleaf.org/doc/articles/standarddialect5minutes.html

VSCode Setup For Spring Boot

Open VS Code

Install *Spring Boot Extension Pack*.

Or install the following extensions:

Spring Boot Tools
Spring Boot Dashboard
Spring Initializr Java Support

VS Code Setup For Spring Boot

- Once the extensions are installed, you will get support for Spring Boot out of the box.
- We will use some of the features during our next lab, but feel free to try it out yourself.

Lab – Spring MVC App using Spring Boot

Code along

Lab – Spring MVC App using Spring Boot

We would be creating a product inventory app.

It will show all the products save in database.

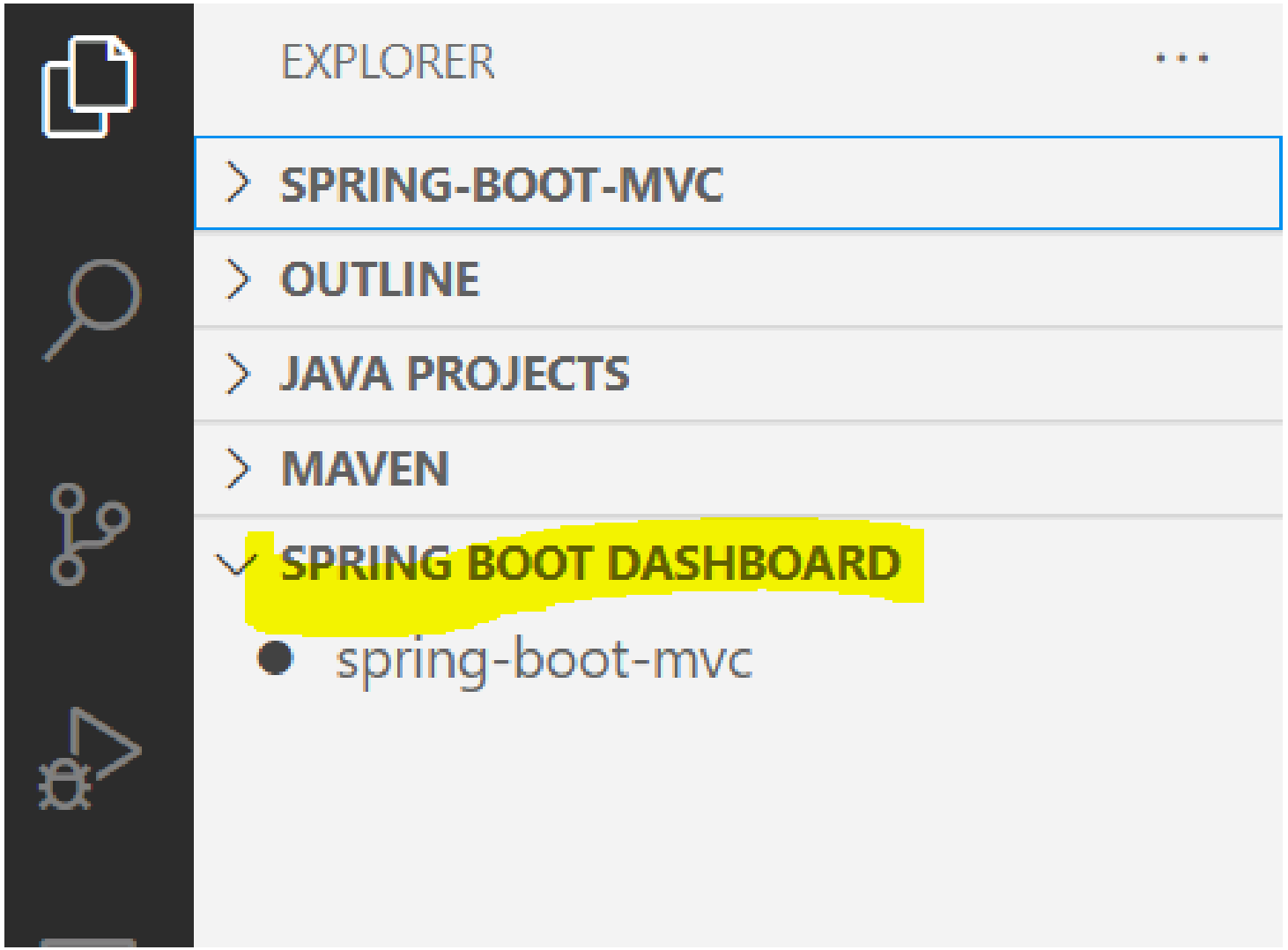
User will be able to Add/Edit/Delete products.

Lab – Spring MVC App using Spring Boot

- Make sure Tomcat is NOT running at all or at least not running on port 8080.
- Make sure MySQL service is up and running.

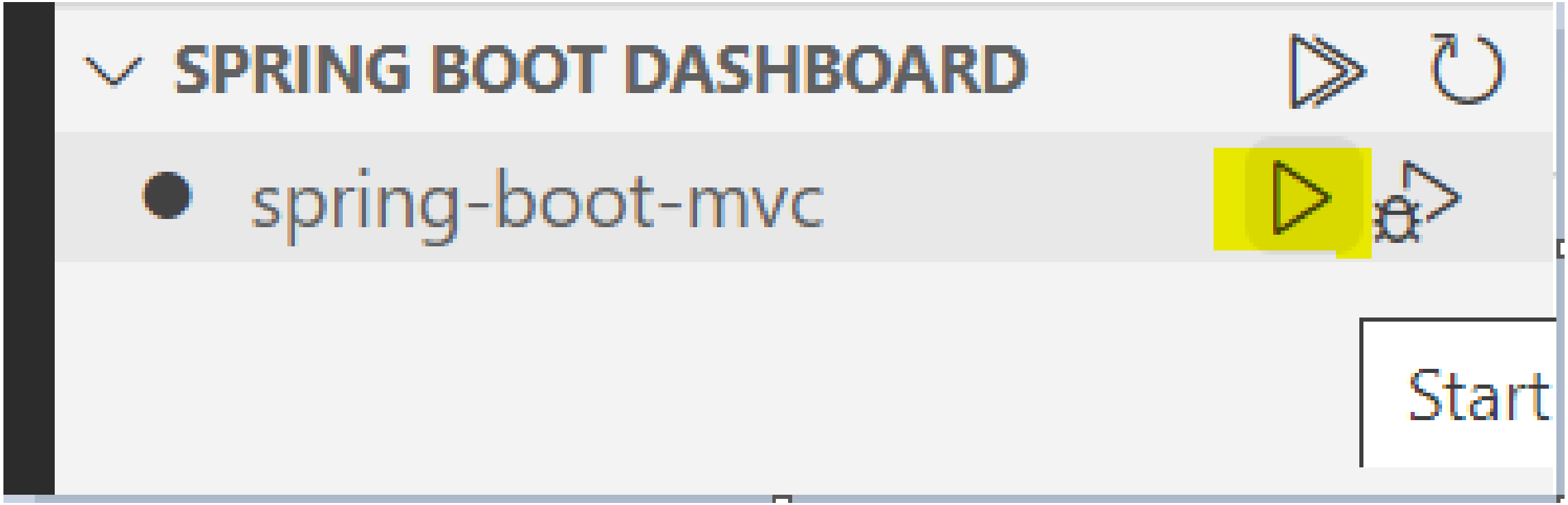
Lab – Spring MVC App using Spring Boot

- Extract the zip(for spring boot project) generated in Lab1.
- Open the project with vscode.
- Once project is imported to vs code, you should be able to observe '**spring boot dashboard**' tab at left side of vscode.



Lab – Spring MVC App using Spring Boot

- Under '**spring boot dashboard**' tab your project should appear.
- Now click on 'start' button (play icon) present on your project name.



Lab – Spring MVC App using Spring Boot

- Let's observe the Logs.
- At the end of logs, you should see something like below.

```
***************************
APPLICATION FAILED TO START
***************************

Description:

Failed to configure a DataSource: 'url' attribute is not specified and no embedded datasource could be configured.

Reason: Failed to determine a suitable driver class
```

Lab – Spring MVC App using Spring Boot

- Our Application has failed to start.
- Under description it says “Failed to configure a Datasource…”
- Explanation :-
  - This is because we have added ‘jpa’ as a dependency while creating our project.
  - Remember? We had configured a ***DataSource bean*** in our spring-mvc lab.
  - Similarly, the application is expecting some kind of configuration for datasource.

Lab – Spring MVC App using Spring Boot

- Before we go ahead and configure our datasource.
- Let's observe some more logs. Look out for logs as shown below. You should be able to find it somewhere in middle of logs.

```
o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8080 (http)
o.apache.catalina.core.StandardService   : Starting service [Tomcat]
org.apache.catalina.core.StandardEngine  : Starting Servlet engine: [Apache Tomcat/9.0.45]
o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed
```

Lab – Spring MVC App using Spring Boot

- What ???
- Tomcat server initialized ??
- Explanation :
  - This is magic of spring-boot.
  - No more mvn clean package
  - No more creating war files.
  - No more copying war into tomcat and restarting the server.
  - Spring boot uses embedded server to deploy apps.
  - Just click ‘start/run’ button and spring boot will do everything for you.

Lab – Spring MVC App using Spring Boot

- Let’s Configure our DataSource.
- Open ‘application.properties’ file under ‘src/main/resources’.
- Set the following properties provided by spring. These are standard properties which spring boot uses internally. Vscode should assist you as you type.

  - spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
  - spring.datasource.url=jdbc:mysql://localhost:3306/test
  - spring.datasource.username=root
  - spring.datasource.password=root
  - spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Dialect

Lab – Spring MVC App using Spring Boot

- That’s it. No more @Bean annotation and java code.
- Based on the values of these properties spring boot will create a bean for you automatically.
- Let’s start our application now.

Lab – Spring MVC App using Spring Boot

- It FAILED again ☹
- Let’s take a look a t logs. You should see something like below.

```
Caused by: org.springframework.beans.factory.BeanCreationException: Error creating bean with name 'dataSource' defined in
class path resource [org/springframework/boot/autoconfigure/jdbc/DataSourceConfiguration$Hikari.class]: Bean instantiatio
n via factory method failed; nested exception is org.springframework.beans.BeanInstantiationException: Failed to instanti
ate [com.zaxxer.hikari.HikariDataSource]: Factory method 'dataSource' threw exception; nested exception is java.lang.Ille
galStateException: Cannot load driver class: com.mysql.cj.jdbc.Driver
```

Lab – Spring MVC App using Spring Boot

- So, this time spring boot finds datasource properties and tries to create a bean.
- But the bean creation process failed because spring boot could not find the driver class we defined in our application.properties file.
- Explanation:-
  - Spring boot does not know which database we will use in our app.
  - Some may use mysql, some may use postgres and so on.
  - So we have to add connector dependency for the database we will use in pom.xml.
  - Spring boot does not do this automatically because then it will have to add dependency for all the databases which exists in the market. Which is not a good thing to do.
  - The driver class we provided belongs to ‘mysql-connector-java’ dependency. Let’s add it.

Lab – Spring MVC App using Spring Boot

- Add this to pom.xml.

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.25</version>
    <scope>runtime</scope>
</dependency>
```

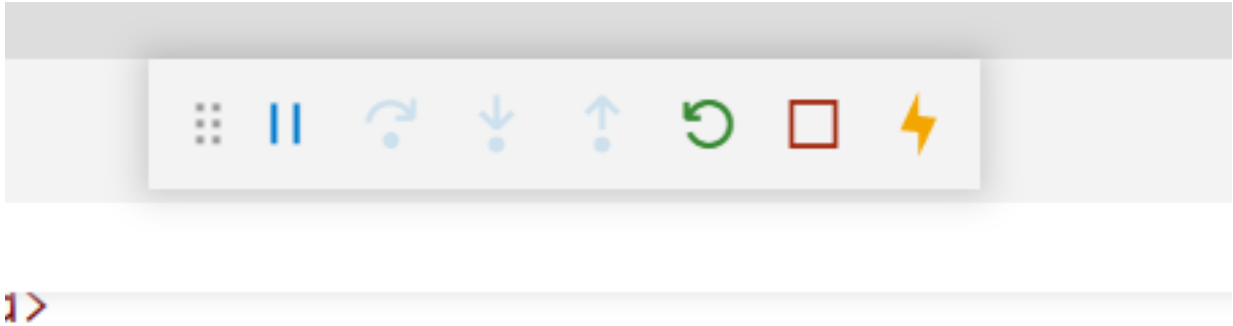
Lab – Spring MVC App using Spring Boot

- Let’s Start the App.
- Application should start successfully.
- You should see following success logs at the end.

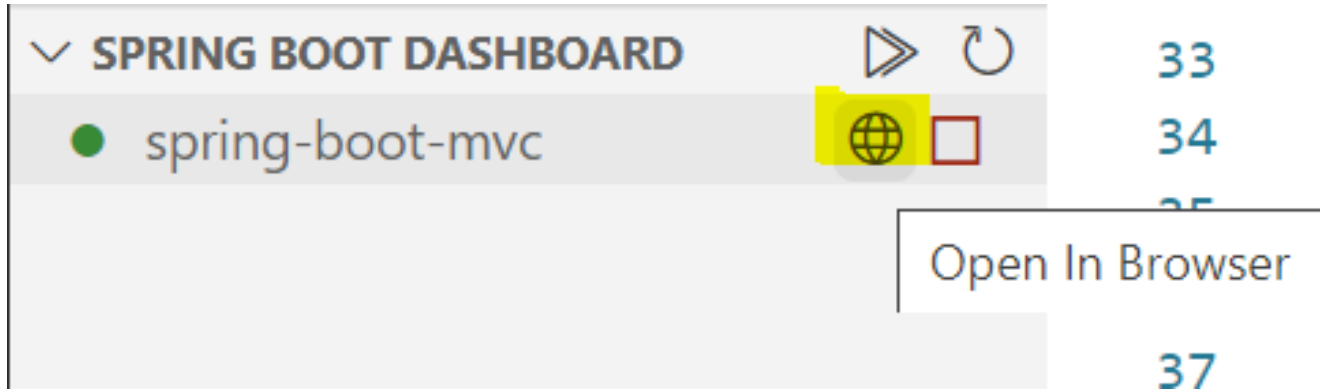
```
main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context p

main] c.d.s.SpringBootMvcApplication           : Started SpringBootMvcApplication in 2.563 seconds (JV
```

Lab – Spring MVC App using Spring Boot

- A spring boot toolbar will appear at the top with some handy buttons.

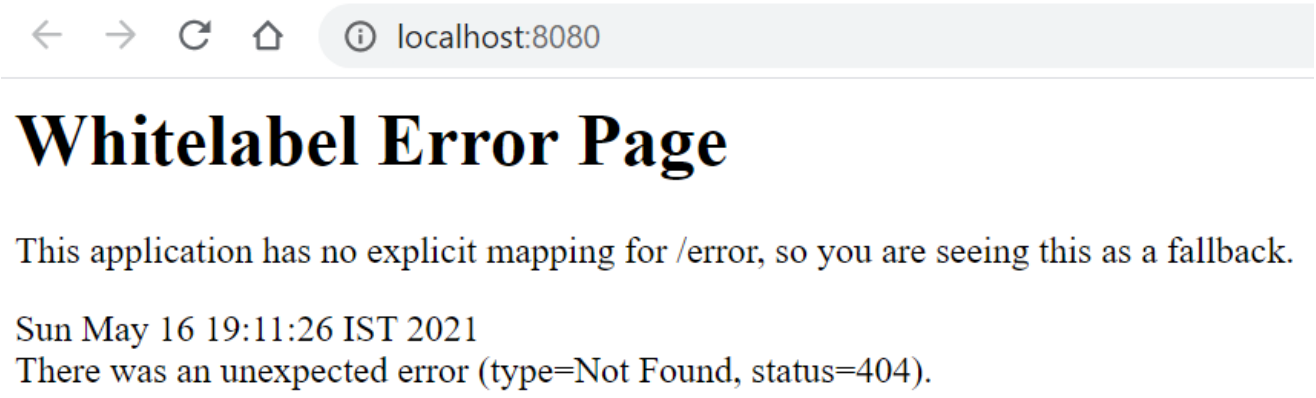


- Click on ‘Open in Browser’ button.



Lab – Spring MVC App using Spring Boot

- You will see a ‘Whitelabel Error Page’.
- Don’t worry. Since we haven't setup our mappings/views, spring-boot defaults to this page.
- Great!! Our app is up and running.



Lab – Spring MVC App using Spring Boot

- Add a products.html file in ‘templates’ folder under ‘src/main/resources’.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <h1>Hello Spring boot</h1>
</body>
</html>
```

- Remember to add `xmlns:th=http://www.thymeleaf.org` inside html tag.

Lab – Spring MVC App using Spring Boot

- Create a package ‘controllers’ and create a *controller* class ‘ProductController.java’ in it.
- Return the name of html file we created in last step.

```
@Controller
public class ProductController {
    @GetMapping("/")
    public String viewHomePage() {
        return "products";
    }
}
```

Lab – Spring MVC App using Spring Boot

`@GetMapping("/")` translates to `@RequestMapping(value = "/", method = RequestMethod.GET)`.

Basically, we are telling spring to render ‘products.html’ at root url i.e. localhost:8080.

Now hit the reload button from spring-boot toolbar at the top of screen.

Reload the browser. You should see “Hello Spring Boot” on the screen.

Lab – Spring MVC App using Spring Boot

- Let’s create a class to represent our Products.
- Create a package ‘entities’ and inside that create a class Product.java.
- Create following fields inside it.

```
public class Product {
    private Long id;
    private String name;
    private String brand;
    private String madein;
    private float price;

    //getters n setters
```

Lab – Spring MVC App using Spring Boot

Next, we'll add some Products in our db.

Before that let's learn some spring-data-jpa concepts.

Entity

Entities in JPA are nothing but POJOs representing data that can be persisted to the database.

An entity represents a table stored in a database.

Every instance of an entity represents a row in the table.

The Entity Annotation @Entity

Let's say we have a POJO called *Student* which represents the data of a student, and we would like to store it in the database.

In order to do this, we should define an entity using *@Entity* annotation so that JPA is aware of it.

We must specify this annotation at the class level.

We must also ensure that the entity has a no-arg constructor and a primary key.

The Entity Annotation @Entity

- The entity name defaults to the name of the class. We can change its name using the *name* element.

- Because various JPA implementations will try subclassing our entity in order to provide their functionality, **entity classes must not be declared *final*.**

```
• @Entity(name="student")
• public class Student {
• // fields, getters and setters
• }
```

The ID Annotation @Id

Each JPA entity must have a primary key which uniquely identifies it.

The *@Id* annotation defines the primary key.

We can generate the identifiers in different ways which are specified by the *@GeneratedValue* annotation.

We can choose from four id generation strategies with the *strategy* element. **The value can be *AUTO, TABLE, SEQUENCE,* or *IDENTITY.***

```
@Entity public class Student {
@Id
@GeneratedValue(strategy=GenerationType.AUTO) private Long id;
```

@Table

In most cases, the name of the table in the database and the name of the entity will not be the same.

In these cases, we can specify the table name using the *@Table* annotation.

We can also mention the schema using the *schema* element. Schema name helps to distinguish one set of tables from another,

If we do not use the *@Table* annotation, the name of the entity will be considered the name of the table.

@Column

Just like the *@Table* annotation, we can use the *@Column* annotation to mention the details of a column in the table.

The *@Column* annotation has many elements such as *name, length, nullable, and unique*.

The *name* element specifies the name of the column in the table. The *length* element specifies its length. The *nullable* element specifies whether the column is nullable or not, and the *unique* element specifies whether the column is unique.

If we don't specify this annotation, the name of the field will be considered the name of the column in the table.

@Table @Column Example

```
@Entity
@Table(name="STUDENT")
public class Student {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private Long id;

    @Column(name="STUDENT_NAME", length=50,
nullable=false, unique=false)
    private String name;

    // other fields, getters and setters
}
```

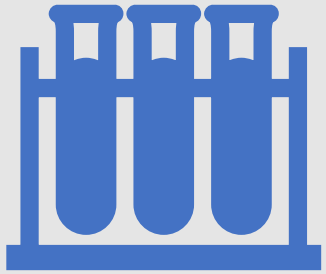
Repository

- Most production codebases have some kind of DAO layer.
- Most likely, there is a one-to-one relation between the DAOs and the entities in the system.
- Spring Data **makes it possible to remove the DAO implementations entirely**. The interface of the DAO is now the only artifact that we need to explicitly define.

Repository

- In order to start leveraging the Spring Data programming model with JPA, a DAO interface needs to extend the JPA specific *Repository* interface – *JpaRepository*.
- This will enable Spring Data to find this interface and automatically create an implementation for it.
- By extending the interface we get the most relevant CRUD methods for standard data access available in a standard DAO.

Let’s continue the Lab



Lab – Spring MVC App using Spring Boot

- Let’s convert our Product model into a Spring Entity like below.

```
@Entity
public class Product {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String brand;
    private String madein;
    private float price;
```

Lab – Spring MVC App using Spring Boot

- Create a package ‘repositories’ and create a DAO (data access object) interface ‘ProductRepository.java’ inside the package.
- Now extend it with *JpaRepository* interface so that spring can implement it internally and provide some basic methods out of the box.
- JpaRepository<arg1, arg2> : arg1 is *type* Of Entity and arg2 is *type* of Primary Key.

```
public interface ProductRepository extends JpaRepository<Product, Lo
ng> {}
```

Lab – Spring MVC App using Spring Boot

- Now let's create a *new-product.html* in *templates* folder and add a form to submit the product details.
- Create a mapping in controller "/new" for creating new product. This will return 'new-product' view.
- Also let's add a button 'Create New Product' on product.html which will send request to mapping created above.

New-product.html

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
     xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="utf-8" />
<title>Create New Product</title>
</head>
<body>
     //create form here
</body>
</html>
```

New-product.html – inside <body>

```
<div align="center">
      <h1>Create New Product</h1><br />
      <form action="#" th:action="@{/save}" th:object="${product}"
          method="post">

          <table border="0" cellpadding="10">
              <tr>
                  <td>Product Name:</td>
                  <td><input type="text" th:field="*{name}" /></td>
              </tr>
              <tr>
                  <td>Brand:</td>
                  <td><input type="text" th:field="*{brand}" /></td>
              </tr>
              <tr>
                  <td>Made In:</td>
                  <td><input type="text" th:field="*{madein}" /></td>
              </tr>
              <tr>
                  <td>Price:</td>
                  <td><input type="text" th:field="*{price}" /></td>
              </tr>
              <tr>
                  <td colspan="2"><button type="submit">Save</button> </td>
              </tr>
          </table>
      </form>
  </div>
```

Update ProductController.java

```
@GetMapping("/new")
public String showNewProductPage(Model model) {
    Product product = new Product();
    model.addAttribute("product", product);
    return "new-product";
}
```

Update products.html

```html
<body>
    <h1>Product List</h1>
    <a href="new">Create New Product</a>
</body>
```

Lab – Spring MVC App using Spring Boot

Hit *Restart* button.

Head over to browser and *refresh* to see changes.

View After Clicking ‘Create New Product’

localhost:8080/new

Create New Product

Product Name:

Brand:

Made In:

Price: 0.0

Save

Lab – Spring MVC App using Spring Boot

- Now we have our form to enter product details.

- Observe ‘new-product.html’, we already have “/save” defined in form action.
  - `<form action="#" th:action="@{/save}" th:object="${product}" method="post">`

- Let’s create a “/save” mapping in controller when the ***save*** button in clicked.

Update Controller

- Inject ProductRepository.

```
@Autowired
private ProductRepository productRepository;
```

- Add mapping for save.

```
@PostMapping(value = "/save")
public String saveProduct(@ModelAttribute("product") Product product) {
  productRepository.save(product);
  return "redirect:/";
}
```

Lab – Spring MVC App using Spring Boot

Hit *Restart* button.

Head over to browser and *refresh* to see changes.

Enter the product details and hit ‘save’.

Go to workbench and see if the product is added to database.

Understanding productRepository.save(product)

- ‘save’ is standard method on repositiory which inserts the given entity into database.
- Repository also created a table for us if one does not exist.
- So, things like “creating a table” and “writing query for inserting record into db” that we did in spring-mvc Lab are all done for us.

Lab – Spring MVC App using Spring Boot

- Now we have some products in our db.
- Let’s update our controller to fetch all products from db.
- Add add some code in ‘products.html’ to show the list of products.

Update Controller

```
@GetMapping("/")
public String viewHomePage(Model model) {
  List<Product> products = productRepository.findAll();
  model.addAttribute("products", products);
  return "products";
}
```

- “findAll” is again a standard method from spring-data repositories which fetches all the records from given table.

Update products.html

```
<body>
    <h1>Product List</h1>
    <a href="new">Create New Product</a>
    <br/><br/>
<table border="1" cellpadding="10">
    <thead>
        <tr>
            <th>Product ID</th>
            <th>Name</th>
            <th>Brand</th>
            <th>Made In</th>
            <th>Price</th>
            <th>Actions</th>
        </tr>
    </thead>
```

Update products.html

```
    <tbody>
        <tr th:each="product : ${products}">
            <td th:text="${product.id}">Product ID</td>
            <td th:text="${product.name}">Name</td>
            <td th:text="${product.brand}">Brand</td>
            <td th:text="${product.madein}">Made in</td>
            <td th:text="${product.price}">Price</td>
            <td>
                <a th:href="@{'/edit/' + ${product.id}}">Edit</a>
                   
                <a th:href="@{'/delete/' + ${product.id}}">Delete</a>
            </td>
        </tr>
    </tbody>
</table>
```

Lab – Spring MVC App using Spring Boot

Hit *Restart* button.

Head over to browser and *refresh* to see changes.

Lab – Spring MVC App using Spring Boot

- You should be able to see the product list like below :

localhost:8080

Product List

Create New Product

Product ID	Name	Brand	Made In	Price	Actions
1	laptop	Mac	USA	400.0	Edit Delete
2	Laptop	Dell	China	200.0	Edit Delete

In Next Session :-

We will implement Edit and Delete.

Do some Exception Handling.

See how we can make our UI look better.