Maven Set up 👍

https://phoenixnap.com/kb/install-maven-windows

Spring Introduction

- Spring framework is an open source Java platform that provides comprehensive infrastructure support for developing robust Java applications very easily and very rapidly.
- Spring framework was initially written by Rod Johnson and was first released under the Apache 2.0 license in June 2003.
- This tutorial has been written based on Spring Framework version 4.1.6 released in Mar 2015.
- Current Spring version : 6.1.2 -> September 2023

Why to Learn Spring?

- Spring is the most popular application development framework for enterprise Java.
- Millions of developers around the world use Spring Framework to create high performing, easily testable, and reusable code.
- Spring is lightweight when it comes to size and transparency.
- The basic version of Spring framework is around 2MB.
- The core features of the Spring Framework can be used in developing any Java application, but there are extensions for building web applications on top of the Java EE platform.
- Spring framework aims to make J2EE development easier to use and promotes good programming practices by enabling a POJO-based programming model.

Features of Spring

Following is the list of few of the great benefits of using Spring Framework -

- **POJO Based** - Spring enables developers to develop enterprise-class applications using POJOs.
- The benefit of using only POJOs is that you do not need an EJB container product such as an application server but you have the option of using only a robust servlet container such as Tomcat or some commercial product.
- **Modular** - Spring is organized in a modular fashion.
- Even though the number of packages and classes are substantial, you have to work **Integration with existing frameworks** - Spring does not reinvent the wheel, instead it truly makes use of some of the existing technologies like several ORM frameworks, logging frameworks, JEE, Quartz and JDK timers, and other view technologies.
- **Testability** - Testing an application written with Spring is simple because environment-dependent code is moved into this framework.
- Furthermore, by using JavaBeanstyle POJOs, it becomes easier to use dependency injection for injecting test data.
- **Web MVC** - Spring's web framework is a well-designed web MVC framework, which provides a great alternative to web frameworks such as Struts or other over-engineered or less popular web frameworks.
- **Central Exception Handling** - Spring provides a convenient API to translate technology-specific exceptions (thrown by JDBC, Hibernate, or JDO, for example) into consistent, unchecked exceptions.

- **Lightweight** - Lightweight IoC containers tend to be lightweight, especially when compared to EJB containers, for example.
- This is beneficial for developing and deploying applications on computers with limited memory and CPU resources.
- **Transaction management** - Spring provides a consistent transaction management interface that can
- scale down to a local transaction (using a single database, for example) and
- scale up to global transactions (using JTA (Java Transaction API), for example).

Audience

- This Notes is designed for Java programmers with a need to understand the Spring framework in detail along with its architecture and actual usage.
- These Notes will bring you to an intermediate level of expertise, from where you can take yourself to higher levels of expertise.

Prerequisites

- Before proceeding with these Notes, you should have a good understanding of Java programming language.
- A basic understanding of Eclipse IDE is also required because all the examples have been compiled using Eclipse IDE.

Dependency Injection (DI)

- The technology that Spring is most identified with is the Dependency Injection (DI) flavor of Inversion of Control.
- The Inversion of Control (IoC) is a general concept, and it can be expressed in many different ways.
- Dependency Injection is merely one concrete example of Inversion of Control.
- When writing a complex Java application, application classes should be as independent as possible of other Java classes to increase the possibility to reuse these classes and to test them independently of other classes while unit testing.
- **Dependency Injection helps in gluing these classes together and at the same time keeping them independent.

What is dependency injection exactly?

- Let's look at these two words separately.
- Here the dependency part translates into an association between two classes.
- For example, class A is dependent on class B.
- Now, let's look at the second part, injection.
- All this means is, class B will get injected into class A by the IoC.
- Dependency injection can happen in the way of

  passing parameters to the constructor or

By post-construction using setter methods.

- As Dependency Injection is the heart of Spring Framework.

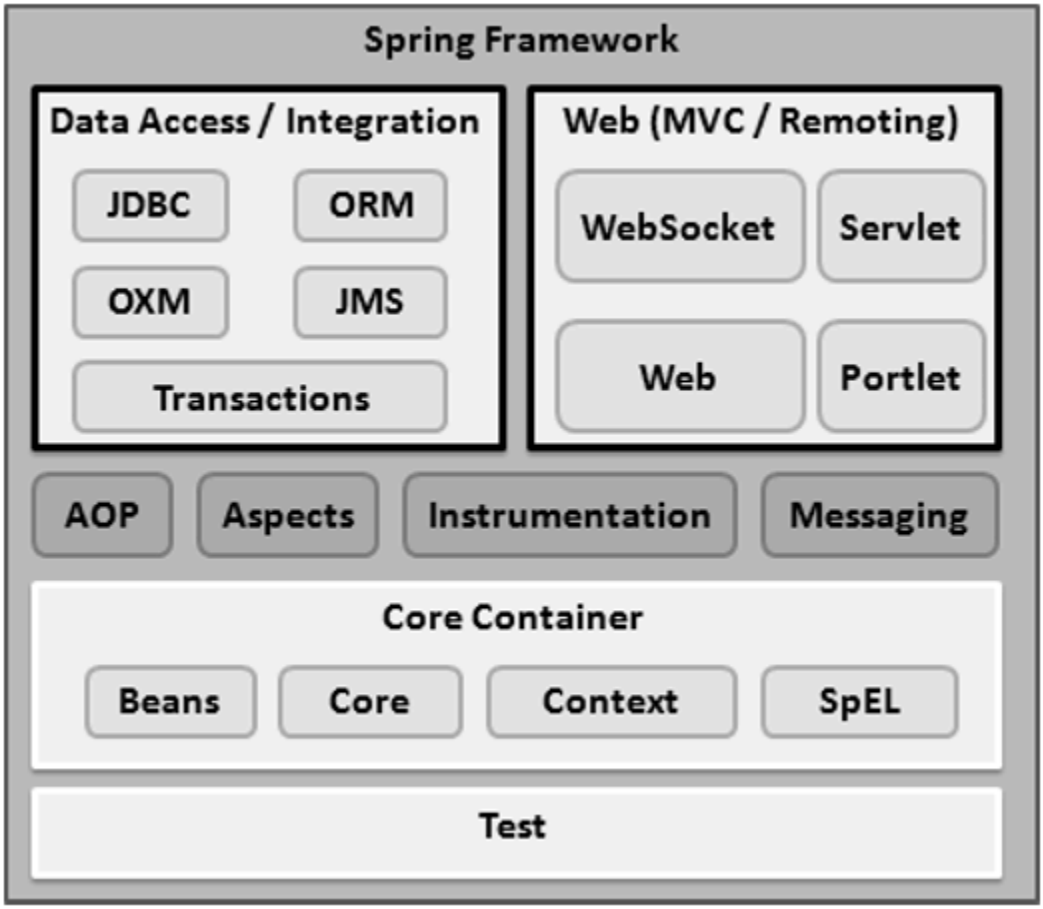
Aspect Oriented Programming (AOP)

- One of the key components of Spring is the Aspect Oriented Programming (AOP) framework.
- The functions that span multiple points of an application are called cross-cutting concerns and these cross-cutting concerns are conceptually separate from the application's business logic.
- There are various common good examples of aspects including logging, declarative transactions, security, caching, etc.
- The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the aspect(a particular part or feature of something.).
- DI helps you decouple your application objects from each other, while AOP helps you decouple cross-cutting concerns from the objects that they affect.
- The AOP module of Spring Framework provides an aspect-oriented programming implementation allowing you to define method-interceptors and pointcuts to cleanly decouple code that implements functionality that should be separated.

Spring Framework - Architecture

- Spring could potentially be a one-stop shop for all your enterprise applications.
- However, Spring is modular, allowing you to pick and choose which modules are applicable to you, without having to bring in the rest.
- The following section provides details about all the modules available in Spring Framework.

The Spring Framework provides about 20 modules which can be used based on an application requirement.



Core Container

The Core Container consists of the

Core, Beans, Context, and Expression Language modules the details of which are as follows -

- The Core module provides the fundamental parts of the framework, including the IoC and Dependency Injection features.
- The Bean module provides BeanFactory, which is a sophisticated implementation of the factory pattern.
- The Context module builds on the solid base provided by the Core and Beans modules and it is a medium to access any objects defined and configured.
- The ApplicationContext interface is the focal point of the Context module.
- The SpEL module provides a powerful expression language for querying and manipulating an object graph at runtime.

Data Access/Integration

The Data Access/Integration layer consists of the JDBC, ORM, OXM, JMS and Transaction modules whose detail is as follows -

- **The JDBC module provides a JDBC-abstraction layer that removes the need for tedious JDBC related coding.**
- **The ORM module provides integration layers for popular object-relational mapping APIs, including JPA, JDO, Hibernate, and iBatis.**
- **The OXM module provides an abstraction layer that supports Object/XML mapping implementations for JAXB, Castor, XMLBeans, JiBX and XStream.**
- **The Java Messaging Service JMS module contains features for producing and consuming messages.**
- **The Transaction module supports programmatic and declarative transaction management for classes that implement special interfaces and for all your POJOs.**

Web

The Web layer consists of the Web, Web-MVC, Web-Socket, and Web-Portlet modules the details of which are as follows –

- **The Web module provides basic web-oriented integration features such as multipart file-upload functionality and the initialization of the IoC container using servlet listeners and a web-oriented application context.**
- **The Web-MVC module contains Spring's Model-View-Controller (MVC) implementation for web applications.**

- **The Web-Socket module provides support for WebSocket-based, two-way communication between the client and the server in web applications.**
- **The Web-Portlet module provides the MVC implementation to be used in a portlet environment and mirrors the functionality of Web-Servlet module.**

Miscellaneous

There are few other important modules like AOP, Aspects, Instrumentation, Web and Test modules the details of which are as follows -

- **The AOP module provides an aspect-oriented programming implementation allowing you to define method-interceptors and pointcuts to cleanly decouple code that implements functionality that should be separated.**
- **The Aspects module provides integration with AspectJ, which is again a powerful and mature AOP framework.**
- **The Instrumentation module provides class instrumentation support and class loader implementations to be used in certain application servers.**
- **The Messaging module provides support for STOMP as the WebSocket sub-protocol to use in applications. It also supports an annotation programming model for routing and processing STOMP messages from WebSocket clients.**
- **The Test module supports the testing of Spring components with JUnit or TestNG frameworks.**

Step 1 - Setup Java Development Kit (JDK)

→ You can download the latest version of SDK from Oracle's Java site - Java SE Downloads.

→ You will find instructions for installing JDK in downloaded files, follow the given instructions to install and configure the setup.

→ Finally, set PATH and JAVA_HOME environment variables to refer to the directory that contains java and javac, typically java_install_dir/bin and java_install_dir respectively.

→ If you are running Windows and have installed the JDK in C:\jdk1.6.0_15, you would have to put the following line in your C:\autoexec.bat file.

Alternatively, on Windows NT/2000/XP, you will have to right-click on My Computer, select Properties → Advanced → Environment Variables.

Then, you will have to update the PATH value and click the OK button.

On Unix (Solaris, Linux, etc.), if the SDK is installed in /usr/local/jdk1.6.0_15 and you use the C shell, you will have to put the following into your .cshrc file.

```
PATH /usr/local/jdk1.6.0_15/bin:$PATH

setenv JAVA_HOME /usr/local/jdk1.6.0_15
```

Alternatively, if you use an Integrated Development Environment (IDE) like Borland JBuilder, Eclipse, IntelliJ IDEA, or Sun ONE Studio, you will have to

compile and run a simple program to confirm that the IDE knows where you have installed Java. Otherwise, you will have to carry out a proper setup as given in the document of the IDE.

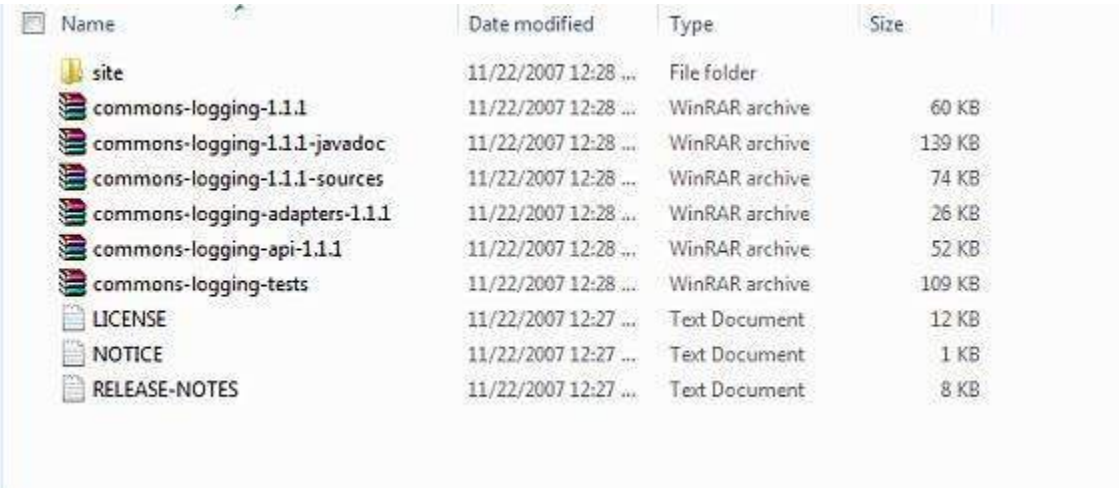
Step 2 - Install Apache Common Logging API

You can download the latest version of Apache Commons Logging API from https://commons.apache.org/logging/.

Once you download the installation, unpack the binary distribution into a convenient location.

For example, in C:\commons-logging-1.1.1 on Windows, or /usr/local/commons-logging-1.1.1 on Linux/Unix.

This directory will have the following jar files and other supporting documents, etc.



Make sure you set your CLASSPATH variable on this directory properly otherwise you will face a problem while running your application.

Step 3 - Setup Eclipse IDE

All the examples in this Notes have been written using Eclipse IDE.

So we would suggest you should have the latest version of Eclipse installed on your machine.

To install Eclipse IDE, download the latest Eclipse binaries from https://www.eclipse.org/downloads/.

Once you download the installation, unpack the binary distribution into a convenient location. For example, in C:\eclipse on Windows, or /usr/local/eclipse on Linux/Unix and finally set PATH variable appropriately.

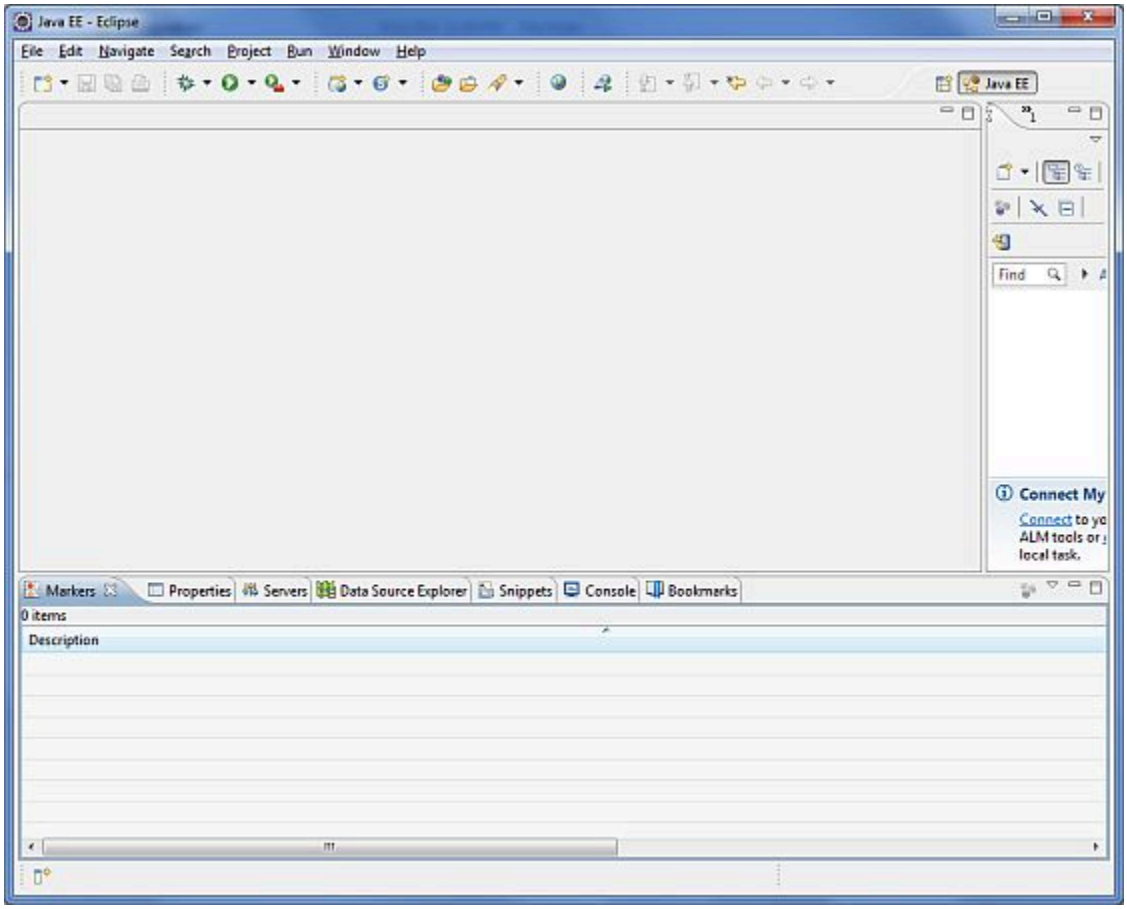
Eclipse can be started by executing the following commands on Windows machine, or you can simply double-click on eclipse.exe

```
%C:\eclipse\eclipse.exe
```

Eclipse can be started by executing the following commands on Unix (Solaris, Linux, etc.) machine -

```
$/usr/local/eclipse/eclipse
```

After a successful startup, if everything is fine then it should display the following result -

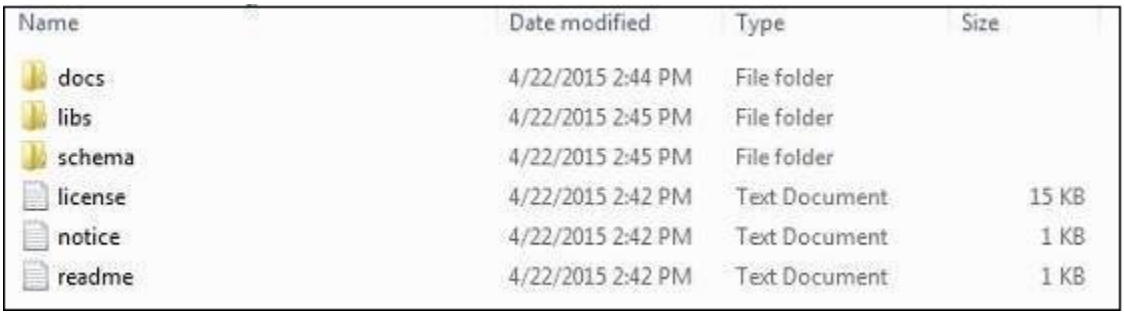


Step 4 - Setup Spring Framework Libraries

Now if everything is fine, then you can proceed to set up your Spring framework. Following are the simple steps to download and install the framework on your machine.

- Make a choice whether you want to install Spring on Windows or Unix, and then proceed to the next step to download the .zip file for Windows and .tz file for Unix.

- **Download the latest version of Spring framework binaries from https://repo.spring.io/release/org/springframework/spring.**
- **At the time of developing this notes, spring-framework-4.1.6.RELEASE-dist.zip was downloaded on Windows machine.**
- **After the downloaded file was unzipped, it gives the following directory structure inside E:\spring.**



Name	Date modified	Type	Size
docs	4/22/2015 2:44 PM	File folder	
libs	4/22/2015 2:45 PM	File folder	
schema	4/22/2015 2:45 PM	File folder	
license	4/22/2015 2:42 PM	Text Document	15 KB
notice	4/22/2015 2:42 PM	Text Document	1 KB
readme	4/22/2015 2:42 PM	Text Document	1 KB

You will find all the Spring libraries in the directory E:\spring\libs.

Make sure you set your CLASSPATH variable on this directory properly otherwise you will face a problem while running your application.

If you are using Eclipse, then it is not required to set CLASSPATH because all the settings will be done through Eclipse.

→ Once you are done with this last step, you are ready to proceed to your first Spring Example in the next chapter.

Spring - Hello World Example

Let us start actual programming with Spring Framework.

Before you start writing your first example using Spring framework, you have to make sure that you have set up your Spring environment properly as explained in Spring - Environment Setup Chapter.

We also assume that you have a bit of working knowledge on Eclipse IDE.

Now let us proceed to write a simple Spring Application, which will print "Hello World!" or any other message based on the configuration done in the

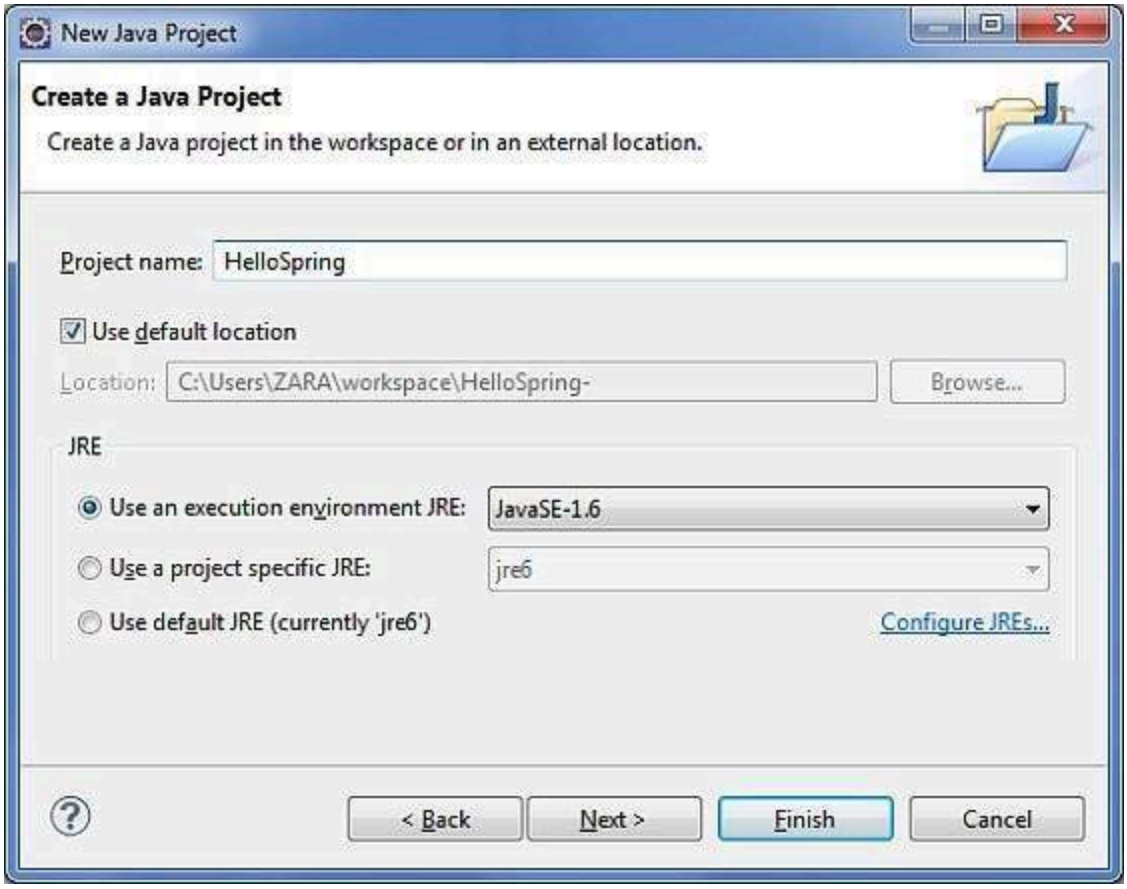
Spring Beans Configuration file.(Beans.xml)

Step 1 - Create Java Project

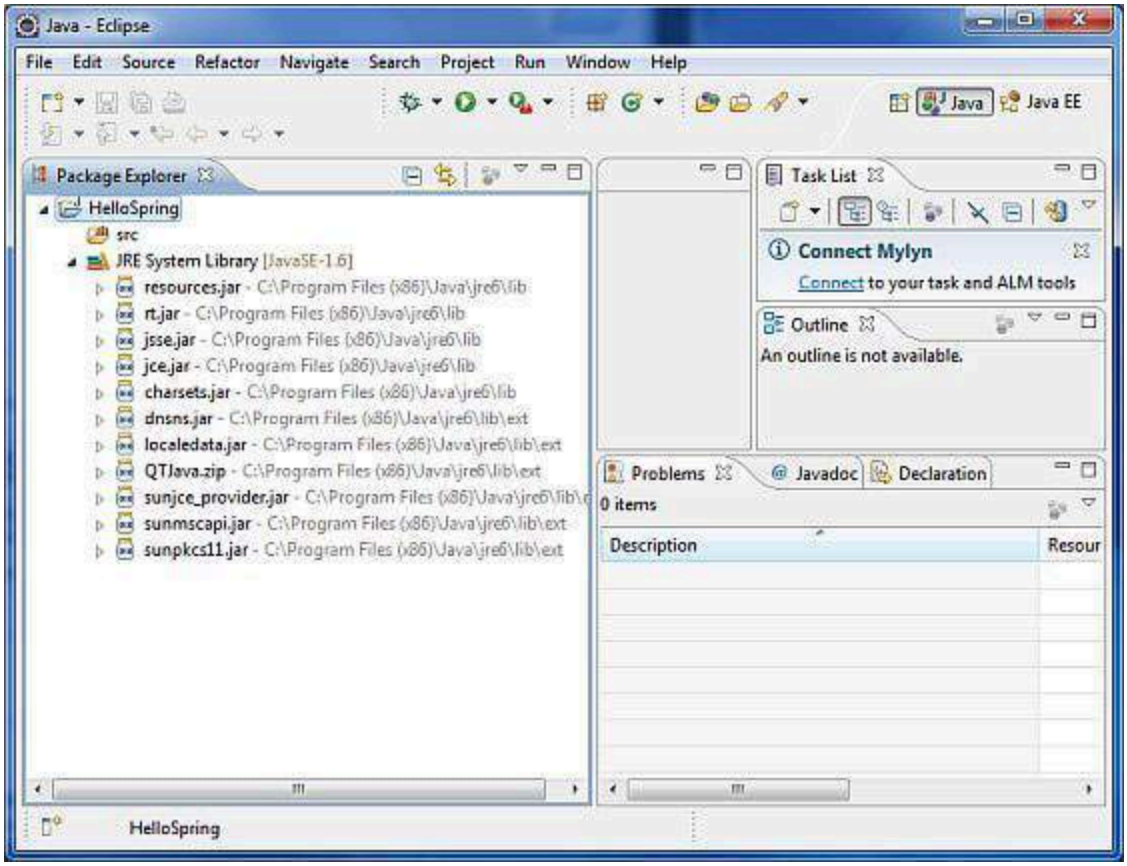
The first step is to create a simple Java Project using Eclipse IDE.

Follow the option File → New → Project and finally select Java Project wizard from the wizard list.

Now name your project as HelloSpring using the wizard window as follows -



Once your project is created successfully, you will have the following content in your Project Explorer –

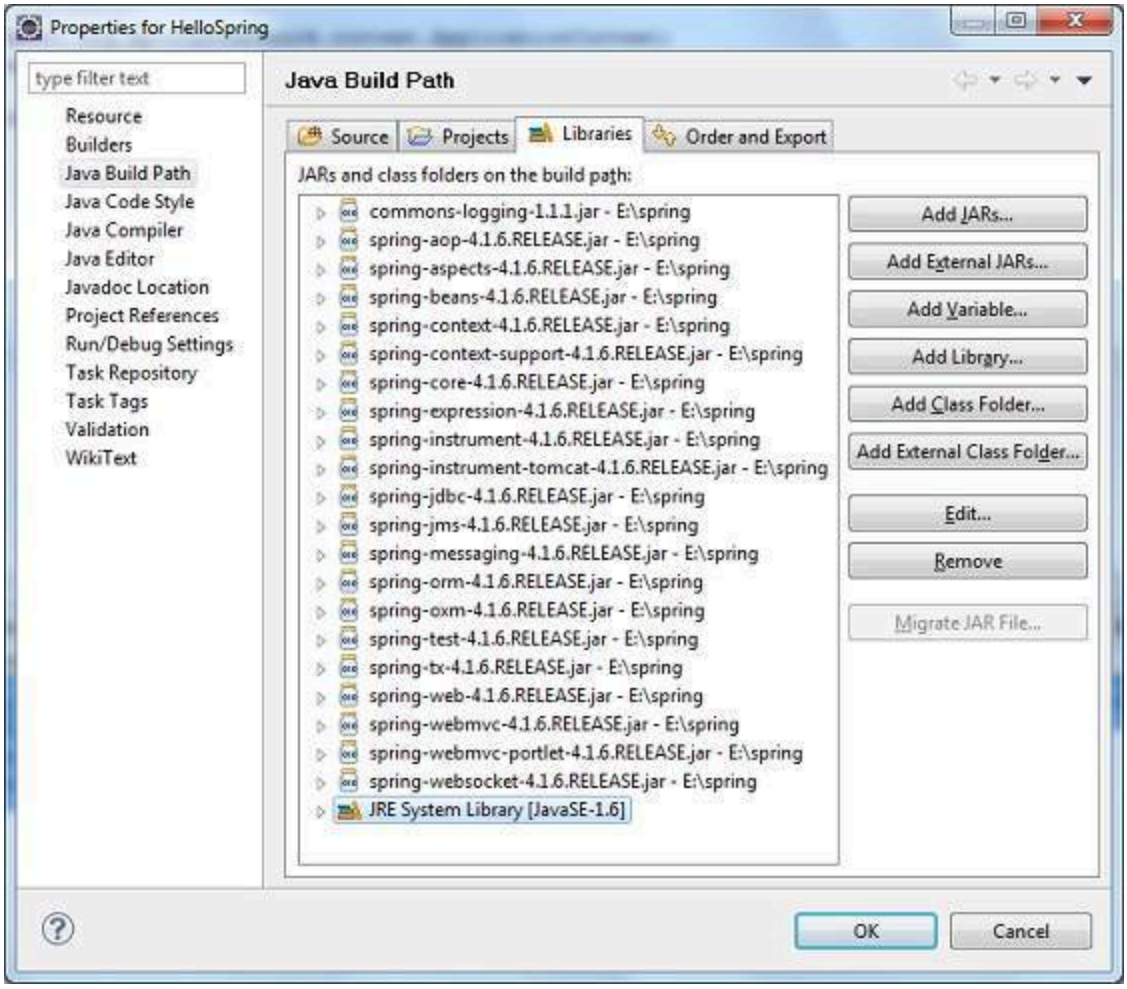


Step 2 - Add Required Libraries

As a second step let us add

Spring Framework and common logging API libraries in our project.

To do this, right-click on your project name HelloSpring and then follow the following option available in the context menu - Build Path → Configure Build Path to display the Java Build Path window as follows -



Now use Add External JARs button available under the Libraries tab to add the following core JARs from Spring Framework and Common Logging installation directories -

- **commons-logging-1.1.1**
- **spring-aop-4.1.6.RELEASE**
- **spring-aspects-4.1.6.RELEASE**
- **spring-beans-4.1.6.RELEASE**
- **spring-context-4.1.6.RELEASE**
- **spring-context-support-4.1.6.RELEASE**
- **spring-core-4.1.6.RELEASE**

- spring-expression-4.1.6.RELEASE
- spring-instrument-4.1.6.RELEASE
- spring-instrument-tomcat-4.1.6.RELEASE
- spring-jdbc-4.1.6.RELEASE
- spring-jms-4.1.6.RELEASE
- spring-messaging-4.1.6.RELEASE
- spring-orm-4.1.6.RELEASE
- spring-oxm-4.1.6.RELEASE
- spring-test-4.1.6.RELEASE
- spring-tx-4.1.6.RELEASE
- spring-web-4.1.6.RELEASE
- spring-webmvc-4.1.6.RELEASE
- spring-webmvc-portlet-4.1.6.RELEASE
- spring-websocket-4.1.6.RELEASE

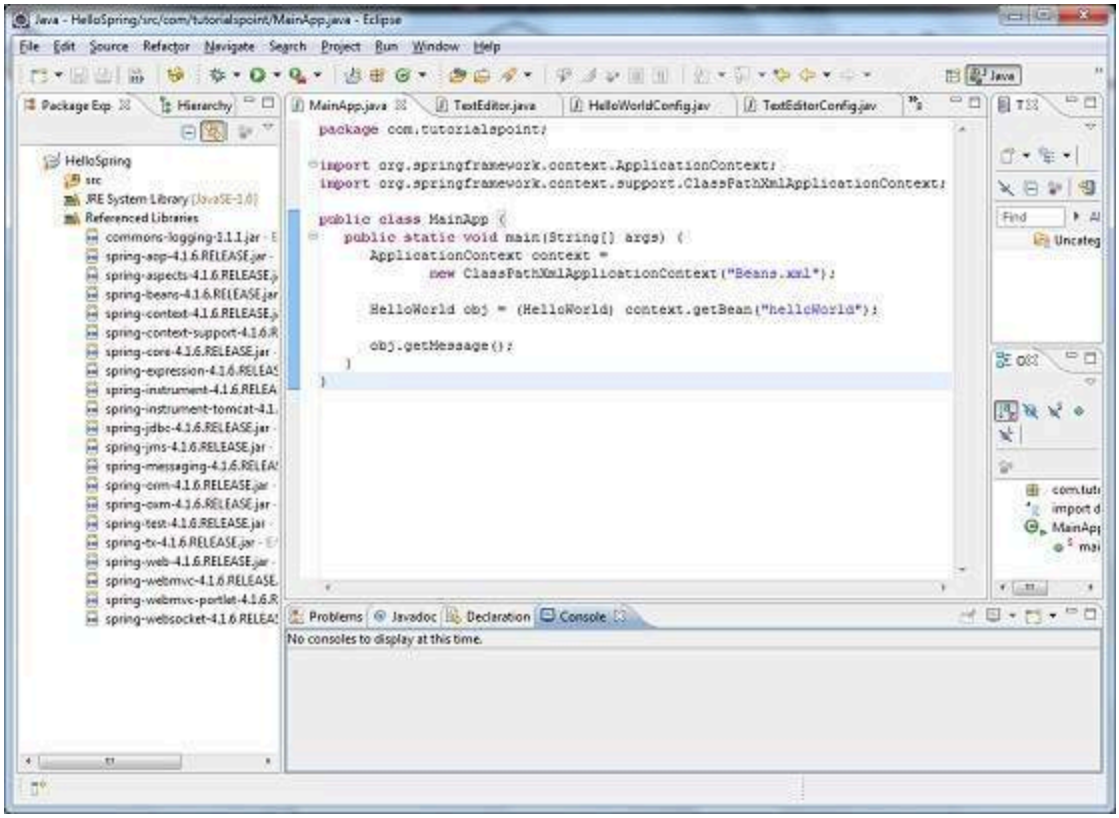
Step 3 - Create Source Files

Now let us create actual source files under the HelloSpring project.

First we need to create a package called com.pack1.

To do this, right click on src in the package explorer section and follow the option - New → Package.

Next we will create HelloWorld.java and MainApp.java files under the com.pack1 package.



Here is the content of HelloWorld.java file -

```
package com.hello;

public class HelloWorld {
     private String message;

     public void setMessage(String message) {
          this.message = message;
     }

     public void getMessage() {
          System.out.println("Your Message : " + message);
     }
}
```

Following is the content of the second file MainApp.java -

```
package com.hello;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

     public static void main(String[] args) {

          ApplicationContext context = new
ClassPathXmlApplicationContext("beans.xml");
          HelloWorld obj = (HelloWorld) context.getBean("helloWorld");
          obj.getMessage();
          System.out.println("something wrong...");
     }

}
```

Following two important points are to be noted about the main program -

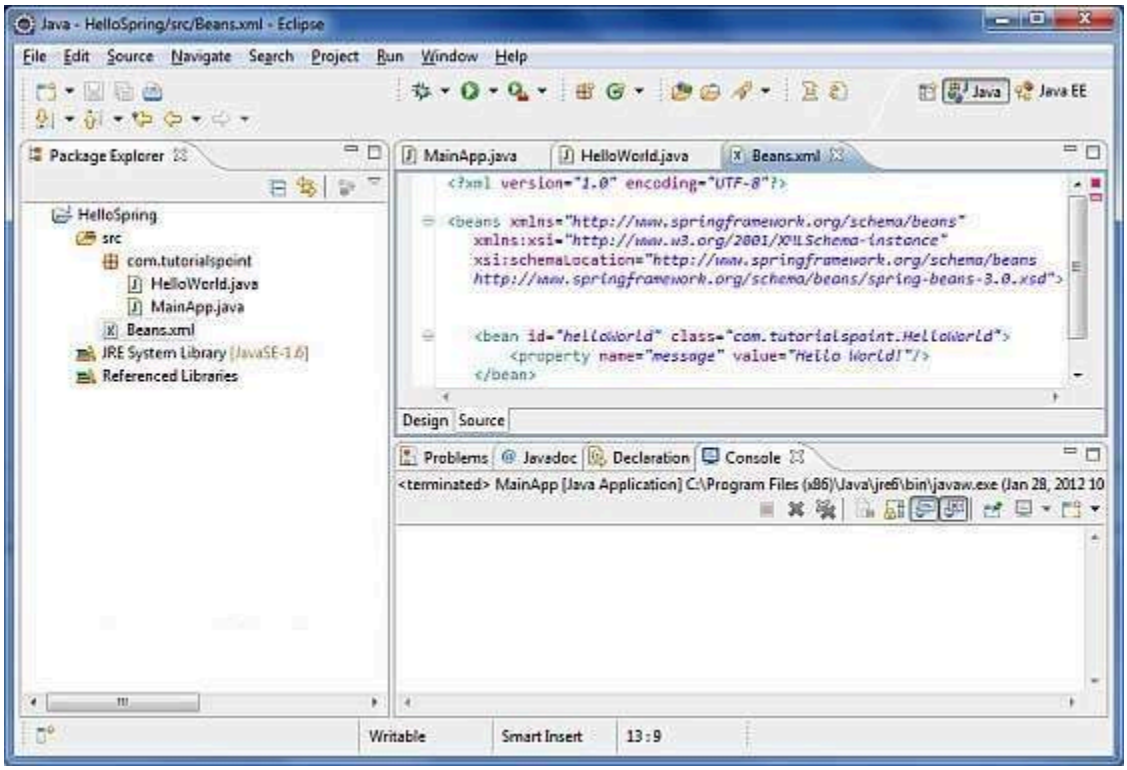
- The first step is to create an application context where we use the framework API ClassPathXmlApplicationContext().
- This API loads a beans configuration file and eventually based on the provided API, it takes care of creating and initializing all the objects, i.e. beans mentioned in the configuration file.
- The second step is used to get the required bean using getBean() method of the created context.

- This method uses bean ID to return a generic object, which finally can be casted to the actual object.
- Once you have an object, you can use this object to call any class method.

Step 4 - Create Bean Configuration File

You need to create a Bean Configuration file which is an XML file and acts as a cement that glues the beans, i.e. the classes together.

This file needs to be created under the src directory as shown in the following screenshot -



Usually developers name this file as Beans.xml, but you are independent to choose any name you like.

You have to make sure that this file is available in CLASSPATH and use the same name in the main application while creating an application context as shown in MainApp.java file.

The Beans.xml is used to assign unique IDs to different beans and to control the creation of objects with different values without impacting any of the Spring source files. For example, using the following file you can pass any value for "message" variable and you can print different values of message without impacting HelloWorld.java and MainApp.java files. Let us see how it works -

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd">

     <bean id="helloWorld" class="com.hello.HelloWorld">
          <property name="message" value="Hello World!" />
     </bean>
</beans>
```

When Spring application gets loaded into the memory, Framework makes use of the above configuration file to create all the beans defined and assigns them a unique ID as defined in <bean> tag.

You can use <property> tag to pass the values of different variables used at the time of object creation.

Step 5 - Running the Program

Once you are done with creating the source and beans configuration files, you are ready for this step, which is compiling and running your program.

To do this, keep MainApp.Java file tab active and use either the Run option available in the Eclipse IDE or use Ctrl + F11 to compile and run your MainApp application.

If everything is fine with your application, this will print the following message in Eclipse IDE's console -

```
Your Message : Hello World!
```

Congratulations, you have successfully created your first Spring Application. You can see the flexibility of the above Spring application by changing the value of "message" property and keeping both the source files unchanged.

Spring - IoC Containers

The Spring container is at the core of the Spring Framework.

The container will create the objects, wire them together, configure them, and manage their complete life cycle from creation till destruction.

The Spring container uses DI to manage the components that make up an application.

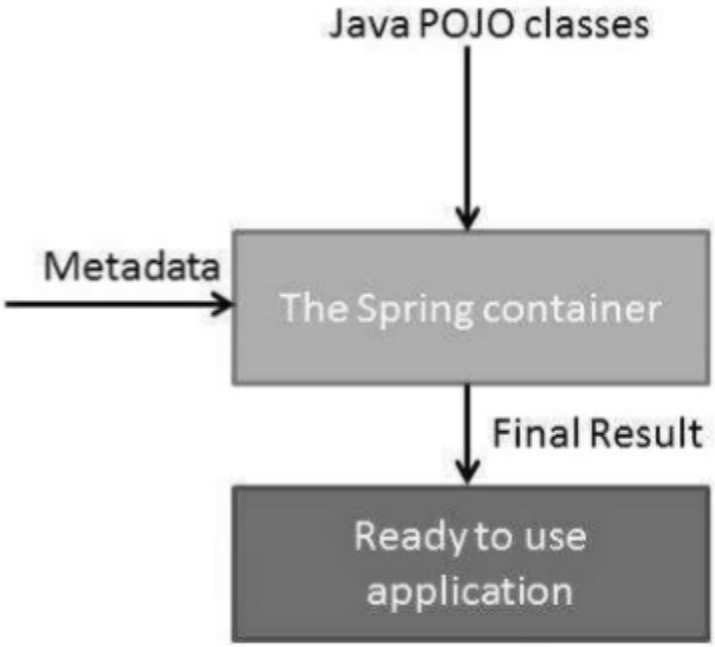
These objects are called Spring Beans.

The container gets its instructions on what objects to instantiate, configure, and assemble by reading the configuration metadata provided.

The configuration metadata can be represented either by XML, Java annotations, or Java code.

The following diagram represents a high-level view of how Spring works.

The Spring IoC container makes use of Java POJO classes and configuration metadata to produce a fully configured and executable system or application.



Spring provides the following two distinct types of containers.

Sr.No.	Container & Description
1	**Spring BeanFactory Container** **This is the simplest container providing the basic support for DI and is defined by the *org.springframework.beans.factory.BeanFactory* interface.** **The BeanFactory and related interfaces, such as** **BeanFactoryAware, InitializingBean, DisposableBean, are still present in Spring for the purpose of backward compatibility with a large number of third-party frameworks that integrate with Spring.**
2	**Spring ApplicationContext Container** **This container adds more enterprise-specific functionality such as the ability to resolve textual messages from a properties file and the ability to publish application events to interested event listeners.**

	This container is defined by the *org.springframework.context.ApplicationContext* interface.

The *ApplicationContext* container includes all functionality of the *BeanFactory*container, so it is generally recommended over *BeanFactory*.

BeanFactory can still be used for lightweight applications like mobile devices or applet-based applications where data volume and speed is significant.

HelloWorld App 02

HelloWord.java

```
package com.pack1;

public class HelloWorld {
     private String message;

     public void setMessage(String message) {
          this.message = message;
     }

     public void getMessage() {
          System.out.println("Your Message : " + message);
     }
}
```

MainApp.java

```
package com.pack1;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

     public static void main(String[] args) {

          ApplicationContext context = new
ClassPathXmlApplicationContext("beans.xml");
          HelloWorld obj = (HelloWorld) context.getBean("helloWorld");
          obj.getMessage();
     }

}
```

Beans.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:context="http://www.springframework.org/schema/context"
     xsi:schemaLocation="
   http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans.xsd
   http://www.springframework.org/schema/context
   http://www.springframework.org/schema/context/spring-context.xsd">
     <bean id="hello" class="pack1.HelloWorld">
          <property name="message" value="Hello World!" />
     </bean>
</beans>
```

Pom.xml

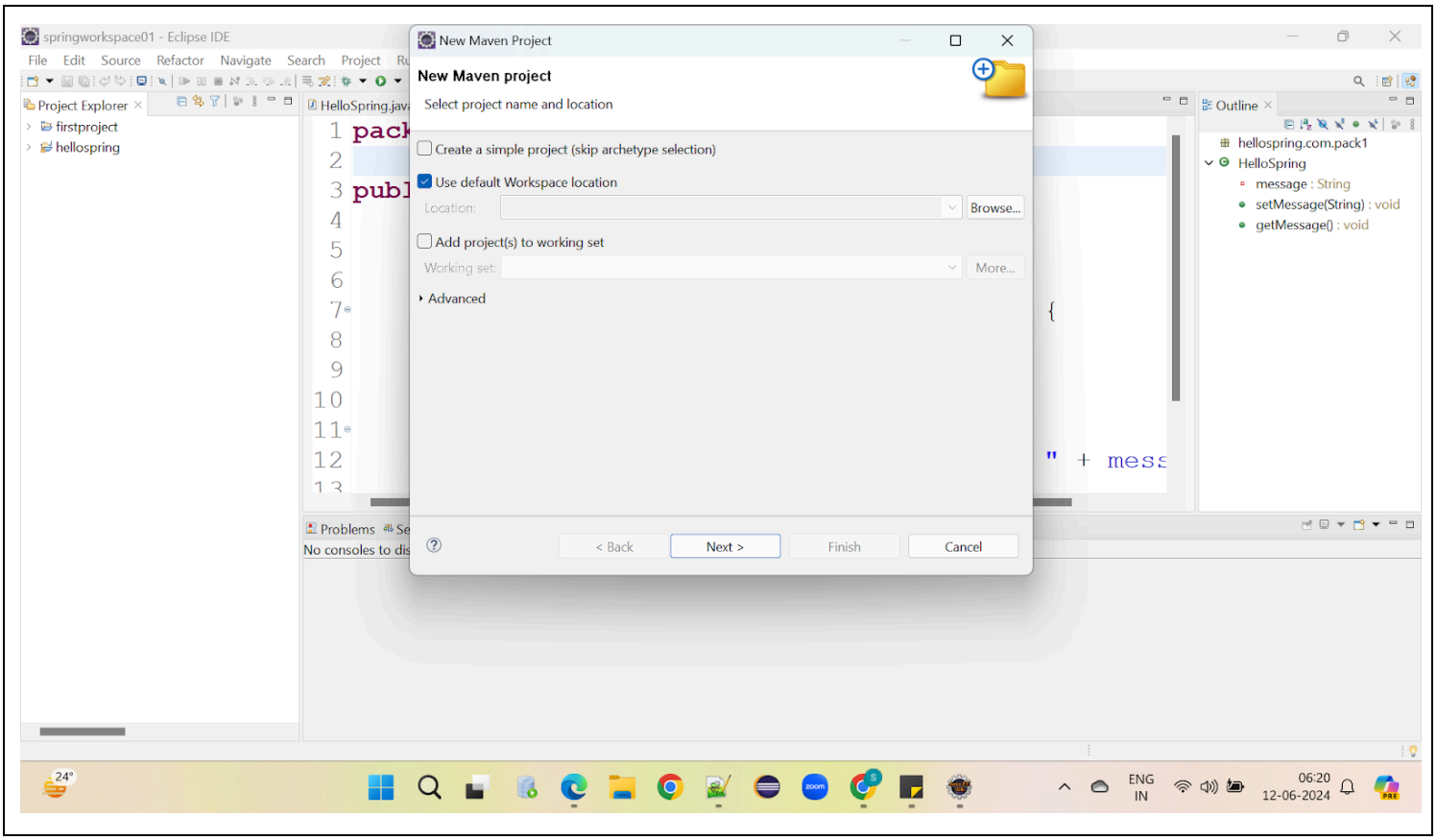
```
<project xmlns="http://maven.apache.org/POM/4.0.0"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
     <modelVersion>4.0.0</modelVersion>
     <groupId>com.srikanth</groupId>
     <artifactId>springdemo03</artifactId>
     <version>0.0.1-SNAPSHOT</version>
     <name>SpringDemo03</name>
     <description>SpringDemo03</description>

     <properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
          <maven.compiler.source>1.7</maven.compiler.source>
          <maven.compiler.target>1.7</maven.compiler.target>
     </properties>

     <dependencies>
          <!--
https://mvnrepository.com/artifact/org.springframework/spring-core -->
          <dependency>
               <groupId>org.springframework</groupId>
               <artifactId>spring-core</artifactId>
               <version>5.2.16.RELEASE</version>
          </dependency>
          <!--
https://mvnrepository.com/artifact/org.springframework/spring-context
-->
          <dependency>
               <groupId>org.springframework</groupId>
               <artifactId>spring-context</artifactId>
               <version>5.2.16.RELEASE</version>
```

```
        </dependency>

        <dependency>
             <groupId>junit</groupId>
             <artifactId>junit</artifactId>
             <version>4.11</version>
             <scope>test</scope>
        </dependency>
    </dependencies>

</project>
```

01 Setter vs Constructor

Spring Demo with Maven

```
Step1
New Maven Project
Select Project name and location
```



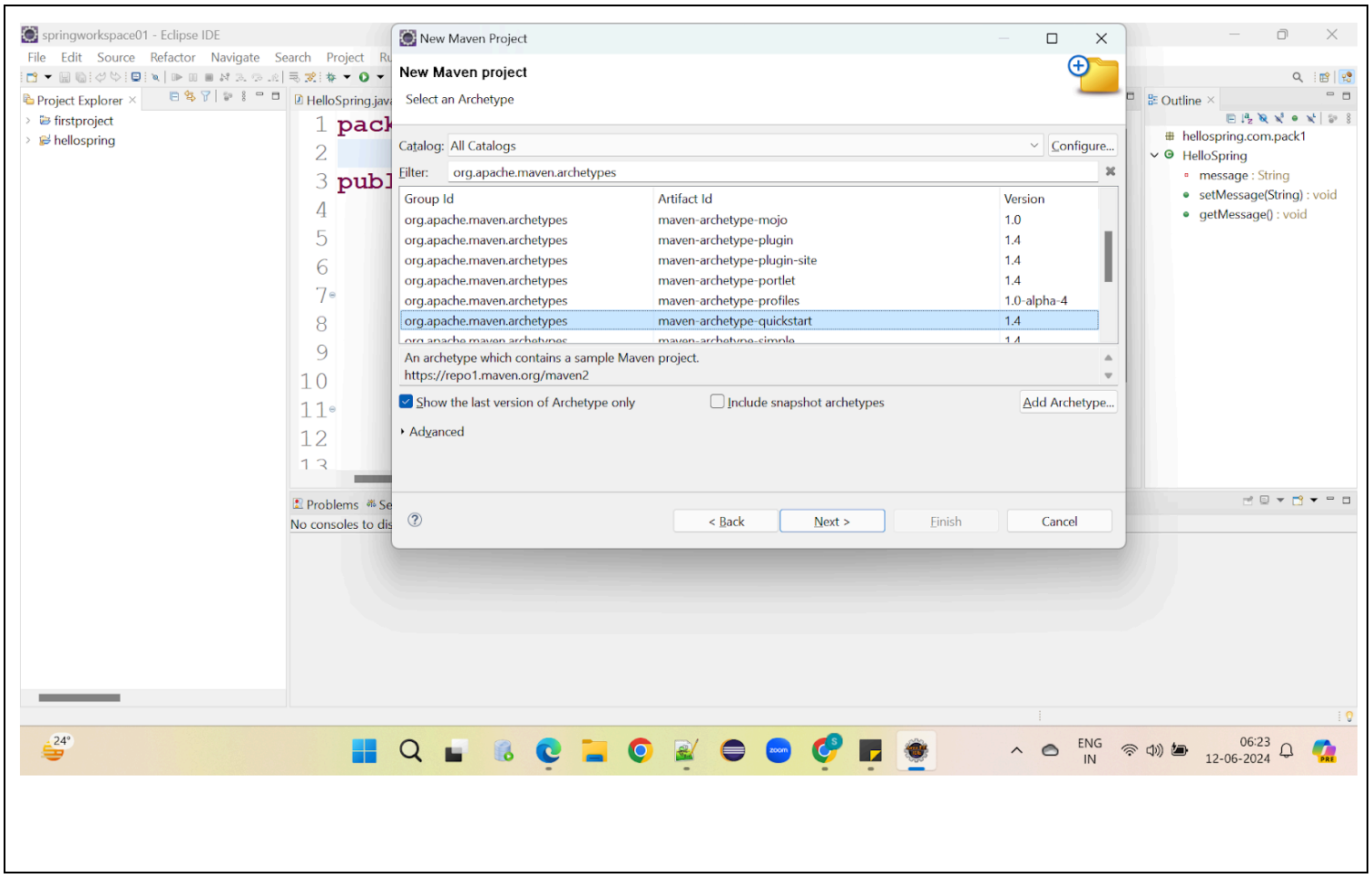
Step2

Select an archetype

org.apache.maven.archetypes - maven-archetype-quickstart 1.4

An archetype which contains a sample Maven project.

https://repo1.maven.org/maven2



Step3

New Maven Project

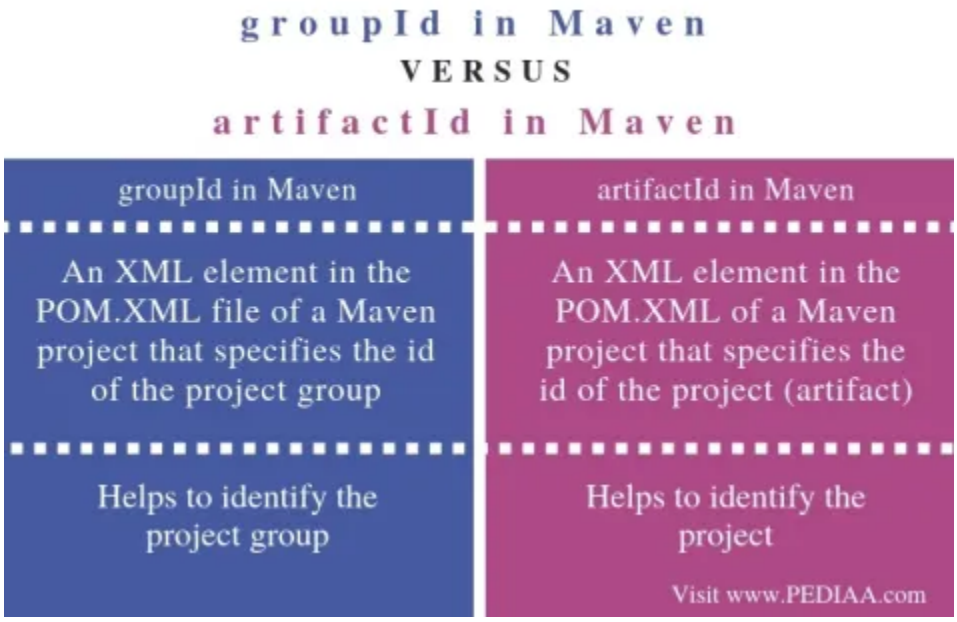
Specify Archetype Parameters

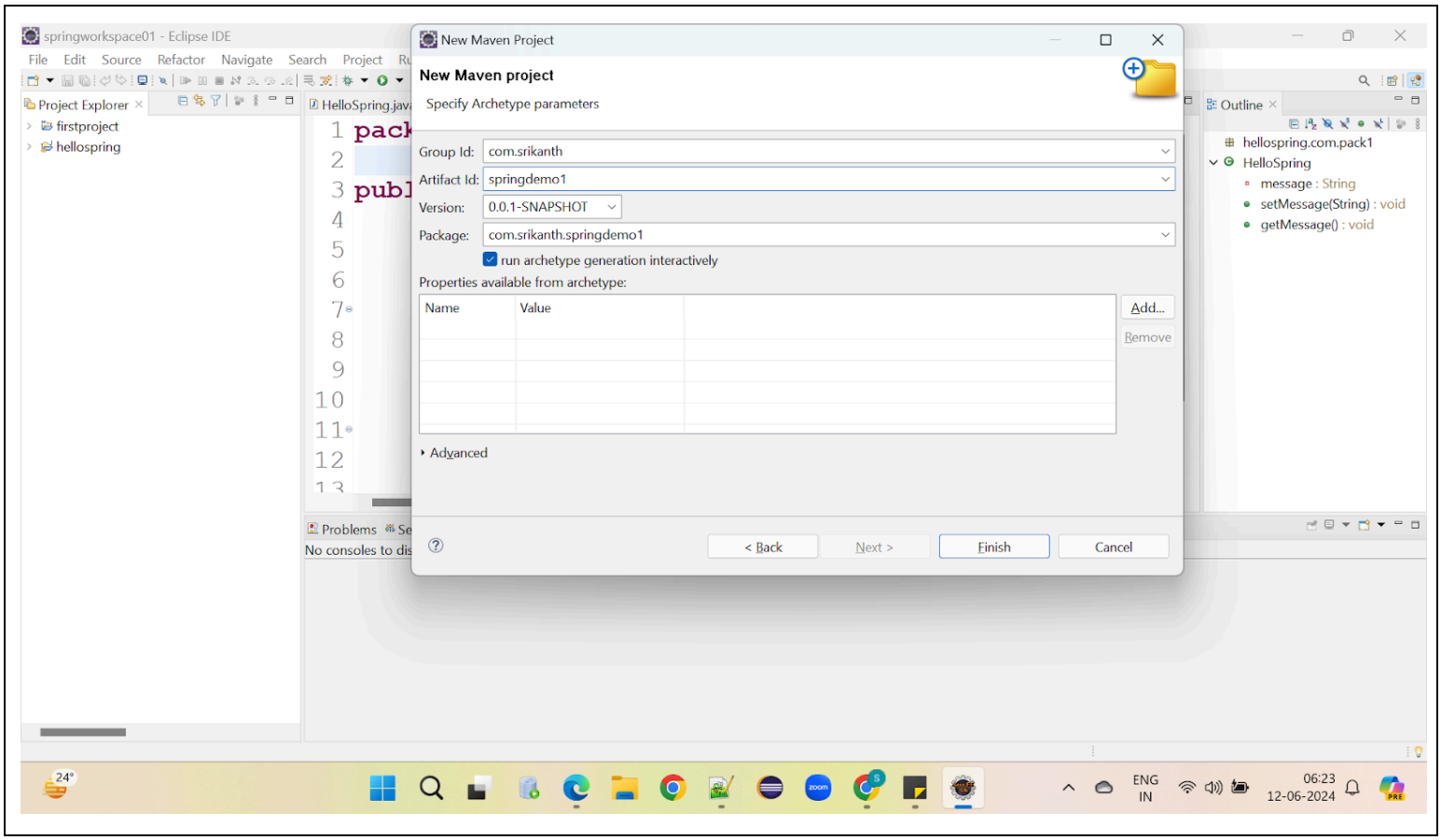
Group ID : com.srikanth
Artifact ID : springdemo1
Version : 0.0.1.SNAPSHOT
Package: com.srikanth.springdemo1

- **groupId uniquely identifies your project across all projects.**
- **A group ID should follow Java's package name rules.**
- **This means it starts with a reversed domain name you control.**

- **artifactId is the name of the jar without version.**

- If you created it, then you can choose whatever name you want with lowercase letters and no strange symbols.
- If it's a third party jar, you have to take the name of the jar as it's distributed.





Step 4

Add dependencies for Core Module

```
                    <!--
https://mvnrepository.com/artifact/org.springframework/spring-core -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>5.2.16.RELEASE</version>
        </dependency>
        <!--
https://mvnrepository.com/artifact/org.springframework/spring-context -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.2.16.RELEASE</version>
```

```
        </dependency>
```

Step 5

Beans.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:context="http://www.springframework.org/schema/context"
     xsi:schemaLocation="
   http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans.xsd
   http://www.springframework.org/schema/context
   http://www.springframework.org/schema/context/spring-context.xsd">
     <bean id="st1" class="com.srikanth.springdemo1.Student">
          <property name="sid" value="123" />
          <property name="sname" value="Kishore" />
     </bean>
     <bean id="st2" class="com.srikanth.springdemo1.Student">
          <constructor-arg index="0" value="124"></constructor-arg>
          <constructor-arg index="1" value="Sri"></constructor-arg>
     </bean>
     <bean id="st3" class="com.srikanth.springdemo1.Student">
          <constructor-arg index="0" value="Sri"></constructor-arg>
     </bean>

     <bean id="st4" class="com.srikanth.springdemo1.Student">
          <constructor-arg index="0" value="123"
type="int"></constructor-arg>
     </bean>
</beans>
```

Step6

Create a POJO class

Student.java

```java
package com.srikanth.springdemo1;
public class Student {
     int sid;
     String sname;
     public void show() {
          System.out.println("ID : " + sid);
          System.out.println("Name : " + sname);
     }
     public Student() {
          System.out.println("default constructor !!");
     }
     public Student(int sid) {
          System.out.println("sid arg constructor !!");
          this.sid = sid;
     }
     public Student(String sname) {
          System.out.println("sname arg constructor !!");
          this.sname = sname;
     }
     public Student(int sid, String sname) {
          System.out.println("sid and sname arg constructor !!");
          this.sid = sid;
          this.sname = sname;
     }
     public int getSid() {
          return sid;
     }
     public void setSid(int sid) {
          this.sid = sid;
     }
     public String getSname() {
          return sname;
     }
     public void setSname(String sname) {
          this.sname = sname;
```

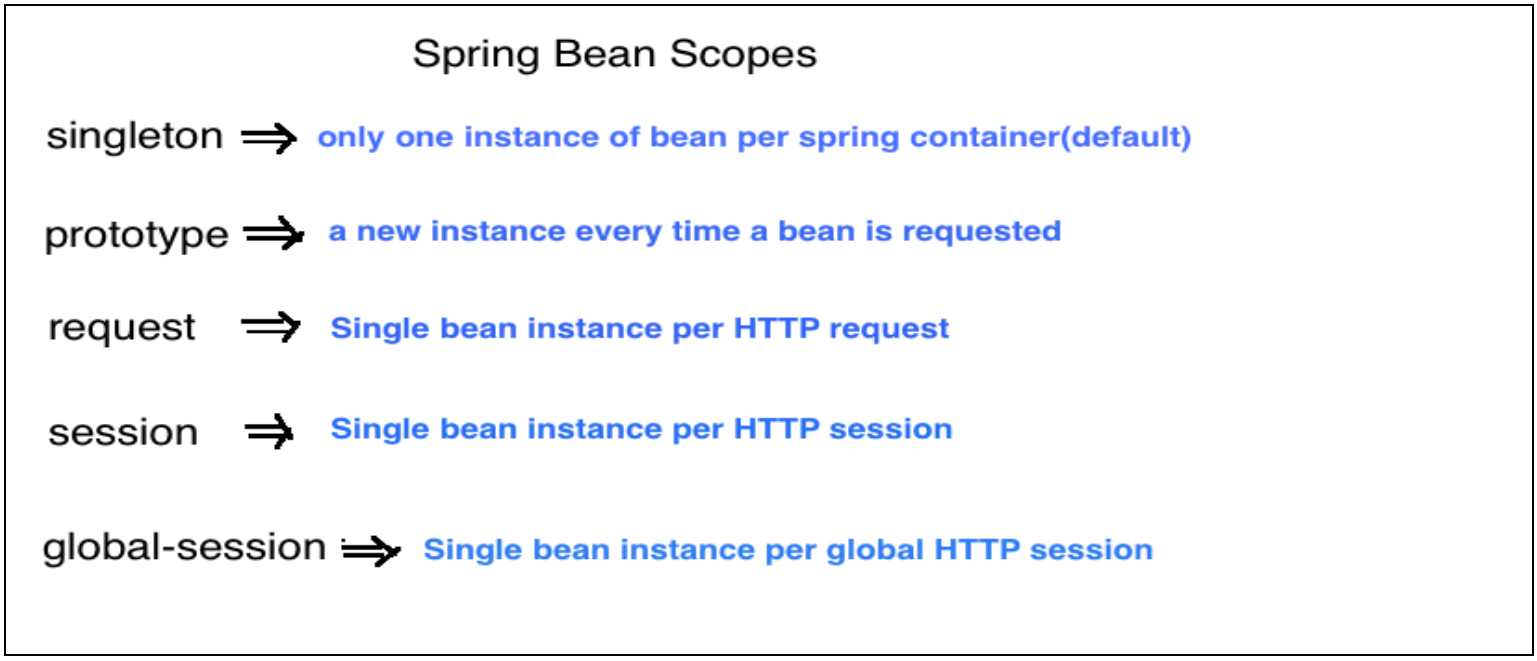
```
    }
}
```

Step7

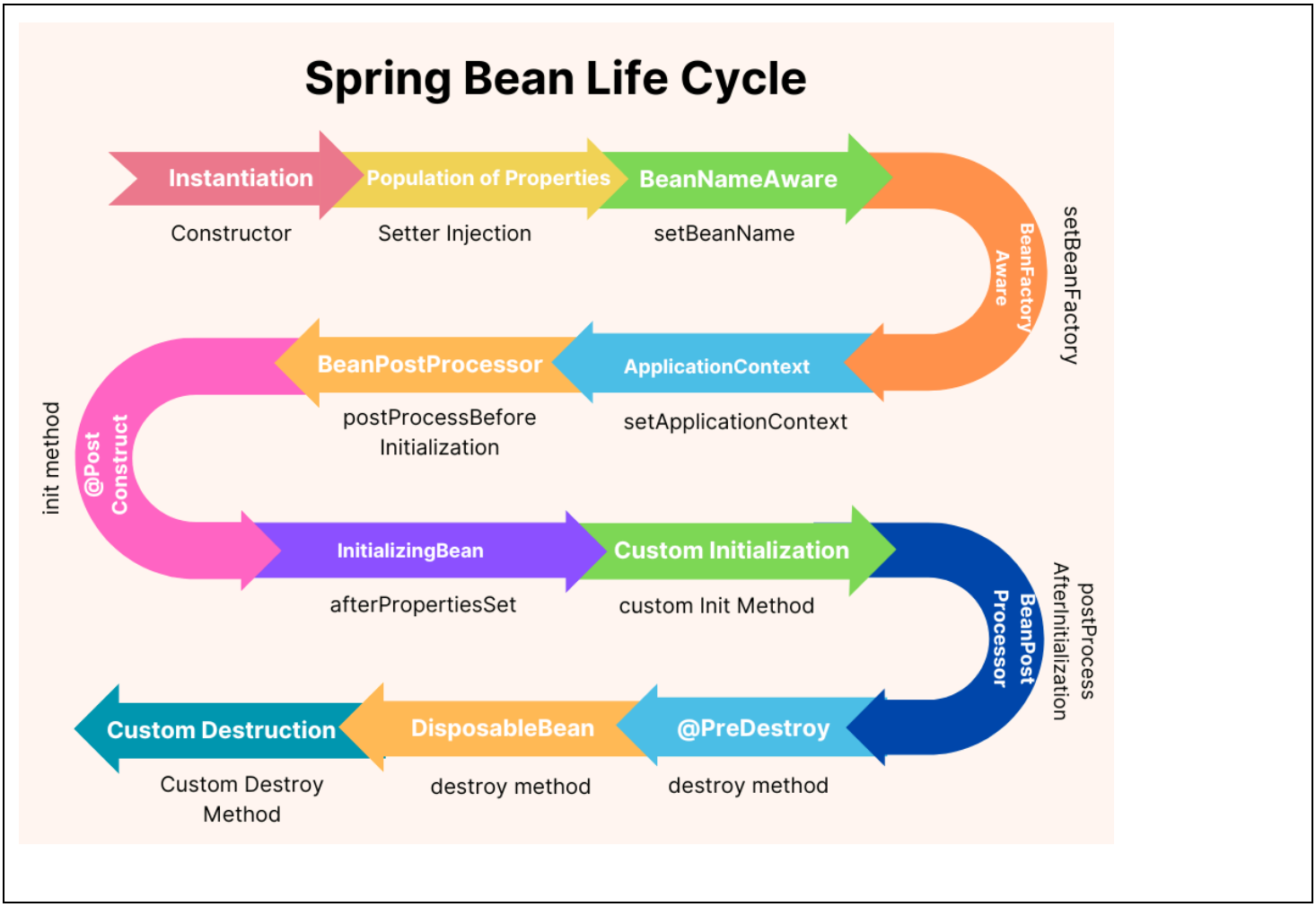
App.java

```
package com.srikanth.springdemo1;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class App {
    public static void main(String[] args) {
        System.out.println("Hello World!");

        ApplicationContext context = new
ClassPathXmlApplicationContext("com/srikanth/springdemo1/beans.xml");
        Student st1 = (Student) context.getBean("st1");
        System.out.println(st1.getSid());
        System.out.println(st1.getSname());

        System.out.println("**************");
        Student st2 = (Student) context.getBean("st2");
        st2.show();
        System.out.println("**************");
        Student st3 = (Student) context.getBean("st3");
        st3.show();
        System.out.println("**************");
        Student st4 = (Student) context.getBean("st4");
        st4.show();

    }
}
```

Spring Bean Scopes



Spring Bean lifecycle



Spring Bean Lifecycle CheatSheet

Spring Bean - POJO (Plain Old Java Object) managed by the IoC container (Spring)

Bean annotations:
@Bean **@Component** **@Service**
@Controller **@RestController** **@Repository**

Bean Definitions are declared

Bean Definitions are declared in XML | annotations with package scanning | annotations with a @Configuration-annotated class | Groovy configuration

Bean Definitions are loaded

BeanDefinitionReader

parses the configuration and **creates** BeanDefinition objects.

Bean Definitions are procesed

BeanFactoryPostProcessor

tweaks BeanDefinitions before actual bean creation. Implementing *BeanFactoryPostProcessor* interface grants access to the created BeanDefinitions and allows modifications.

Beans are instantiated

BeanFactory

invokes the constructor of each bean. If needed, it delegates this to custom FactoryBean instances.

Dependencies are injected (setters are called)

If dependencies are injected in the constructor, dependent beans are created first, followed by those that depend on them.
Avoid cyclic dependencies: use constructor Injection | @Lazy | Refactor code

Beans are Post Processed (before initialisation)

BeanPostProcessor

adjusts beans in the first round (after the object is fully created but before initialisation). Called **before the init-method** (if defined) and returns the bean.

Beans are initialised

The **@PostConstruct** init method of the bean is triggered, and the bean gets initialised

Bean scopes:
Singletone (default) | Prototype | Request | Session | Application | WebSocket | Custom Scopes

Beans are Post Processed (after initialisation)

BeanPostProcessor

makes their **second pass** on the bean, usually needed if you need to wrap a proxy around the bean, or in case of circular dependencies

Beans live in Application Context

Fully created bean is stored in the context and accessible

Injection annotations
@Autowired
@Setter
@Qualifier (to specify name when autowiring)
@Primary (primary bean of the same type to inject)

Bans are destroyed

When the context is being destroyed or bean is disposed, the **@PreDestroy** destroy method of the bean is called.

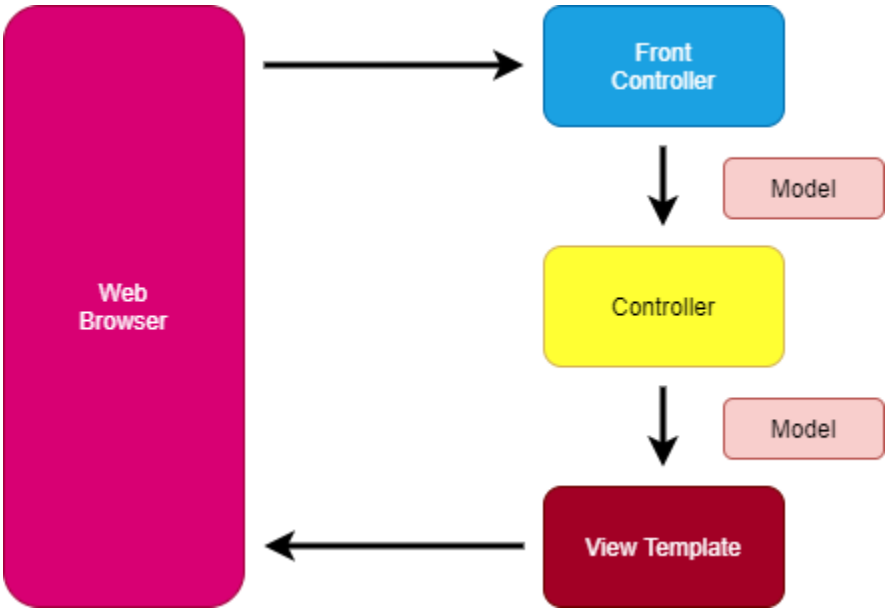
Spring MVC 👋

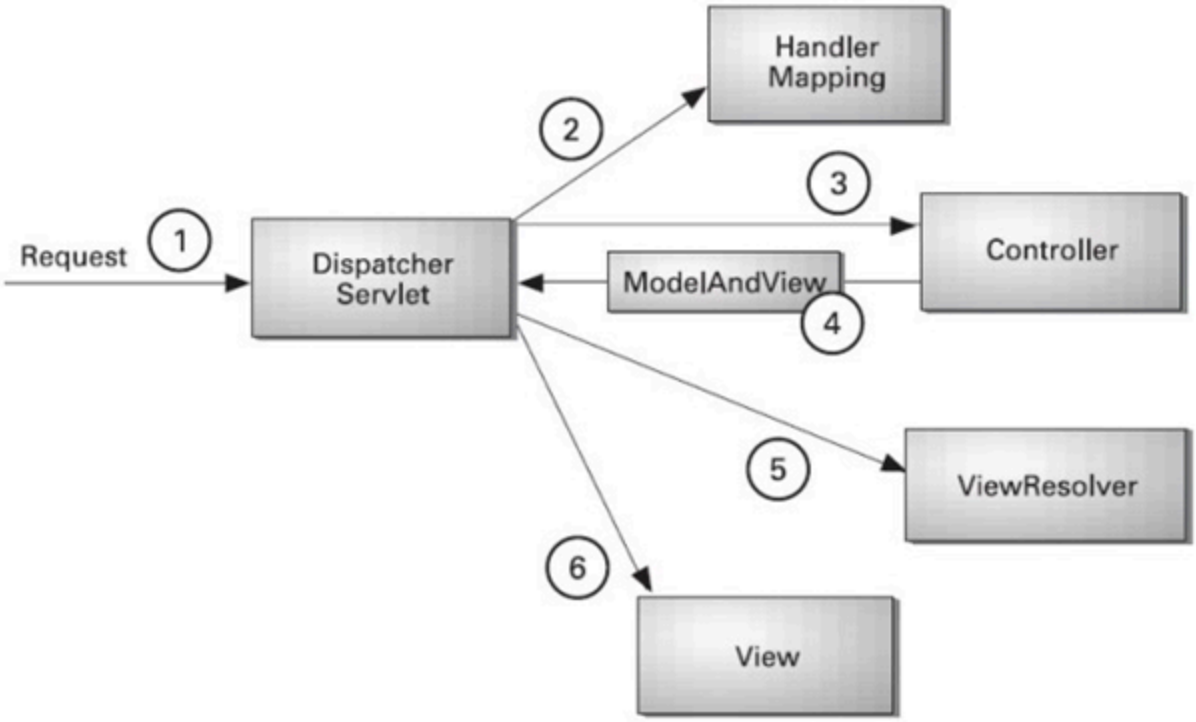
What is Spring MVC?

- Spring MVC is a framework for building web applications in Java.
- It is based on the Model-View-Control design pattern.
- It leverages features of the core Spring framework such as Inversion of control and dependency injection.

Model-View-Controller

Below is the architectural flow diagram of Spring MVC





- **Browser sends a request to the server. Server takes that request, processes it and sends it to the Front controller.**

- **Front controller is a Dispatcher Servlet, its job is to send the request to the appropriate controller.**

- **Controller code is the one written by developers, which contains the business logic.**

- **Controller will prepare the model and send it to the Front controller for further processing.**

- **Front controller then sends that model to the view template, which contains the HTML code and data which needs to be displayed in the web page.**

Controller

- **Controller classes are created by developers.**
- **Contains business logic and it handles the request.**
- **It stores and retrieves the data from the database and other services.**
- **It will write the data to the model.**
- **It will send the data to the appropriate view template with the help of Front Controller.**

Model

- **Model is just a java object containing the data.(POJO or DTO)**
- **It will help us to store and retrieve the data from a database or web service.**

View Template

- **Spring MVC is flexible, it supports many view templates.**
- **Most commonly used view templates are JSP and Thymeleaf.**
- **Developer creates the view template to display the data present in the Model.**

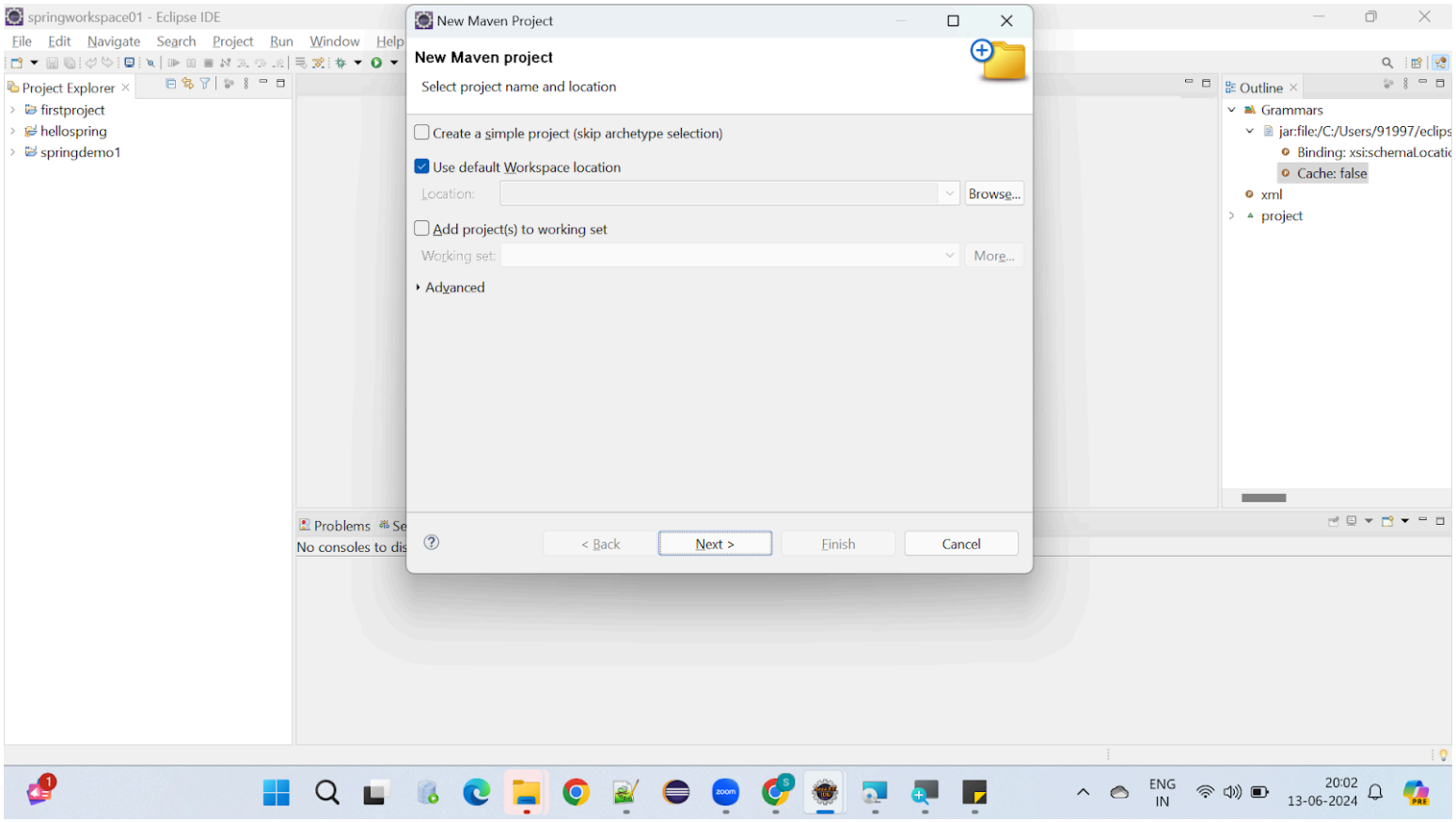
Spring MVC Benefits

- **It is the best way of building web applications in Java.**
- **It leverages a set of reusable UI components.**
- **It will help us to manage application state for web requests.**
- **It processes the form data by validating, converting etc…**
- **It is very flexible to configure the view layer with other view templates.**

Spring MVC Example :

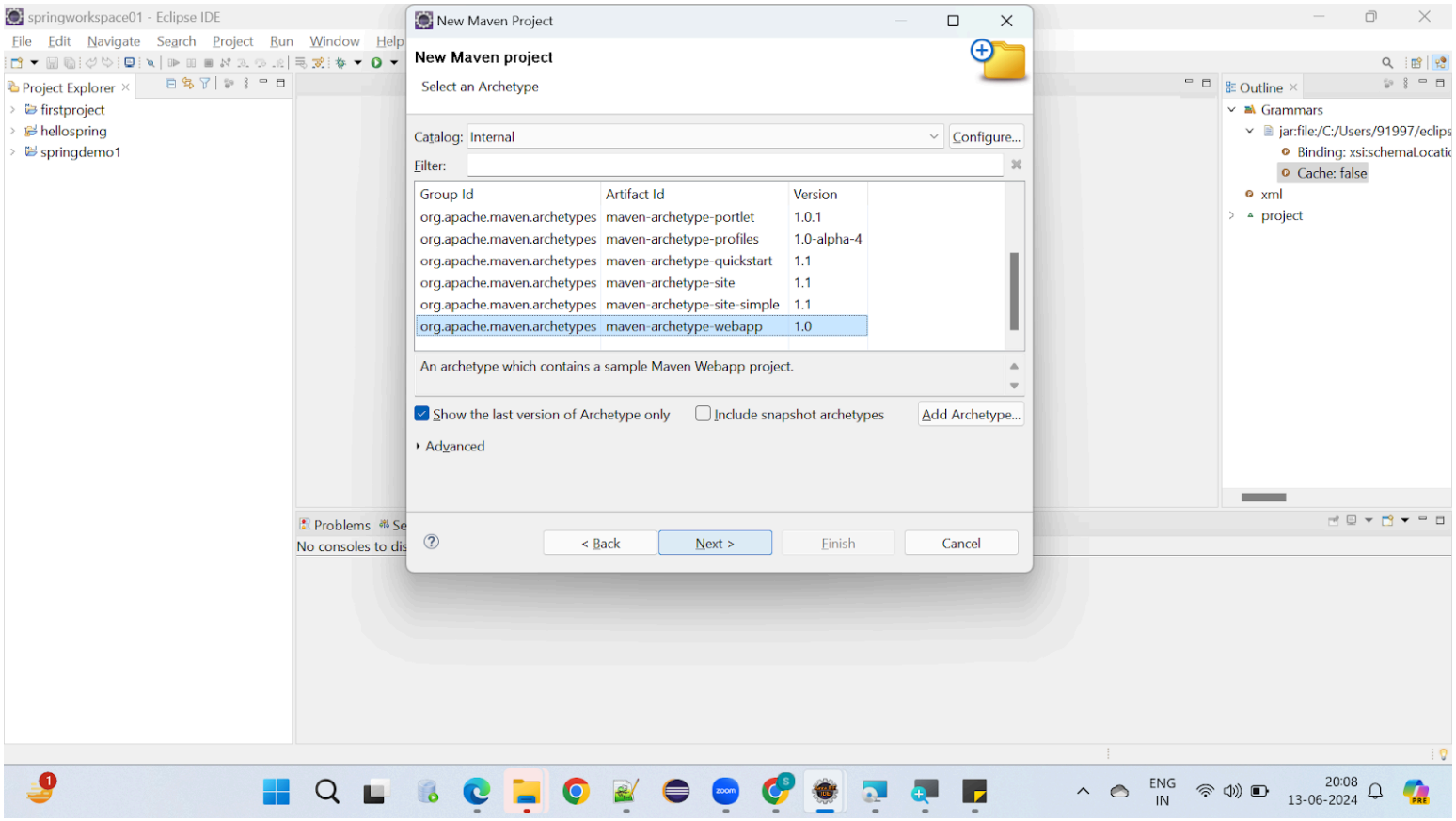
Step1 :

Select project name and location



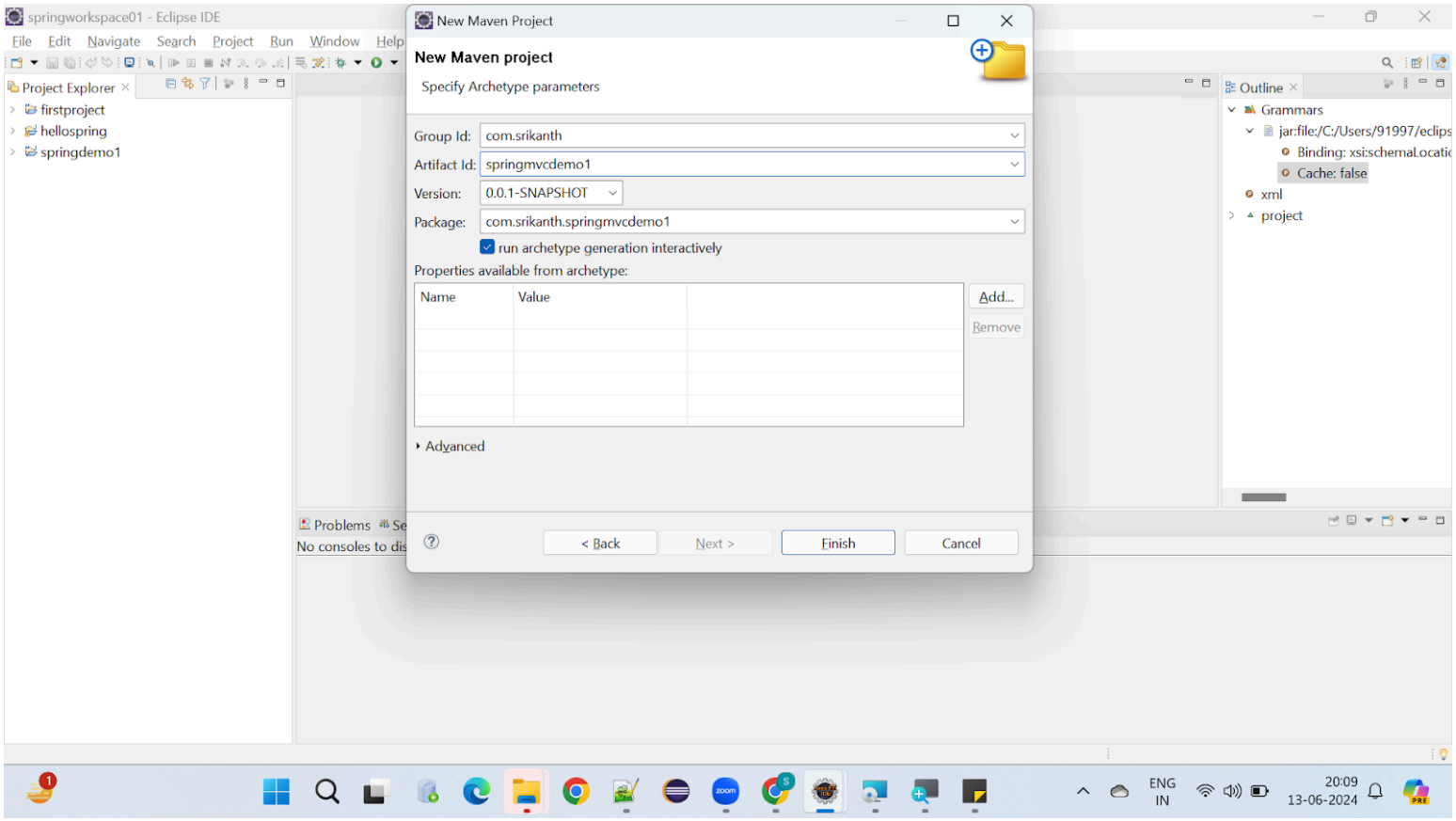
Step 2:

Select an Archetype



Step3

Specify Archetype parameters

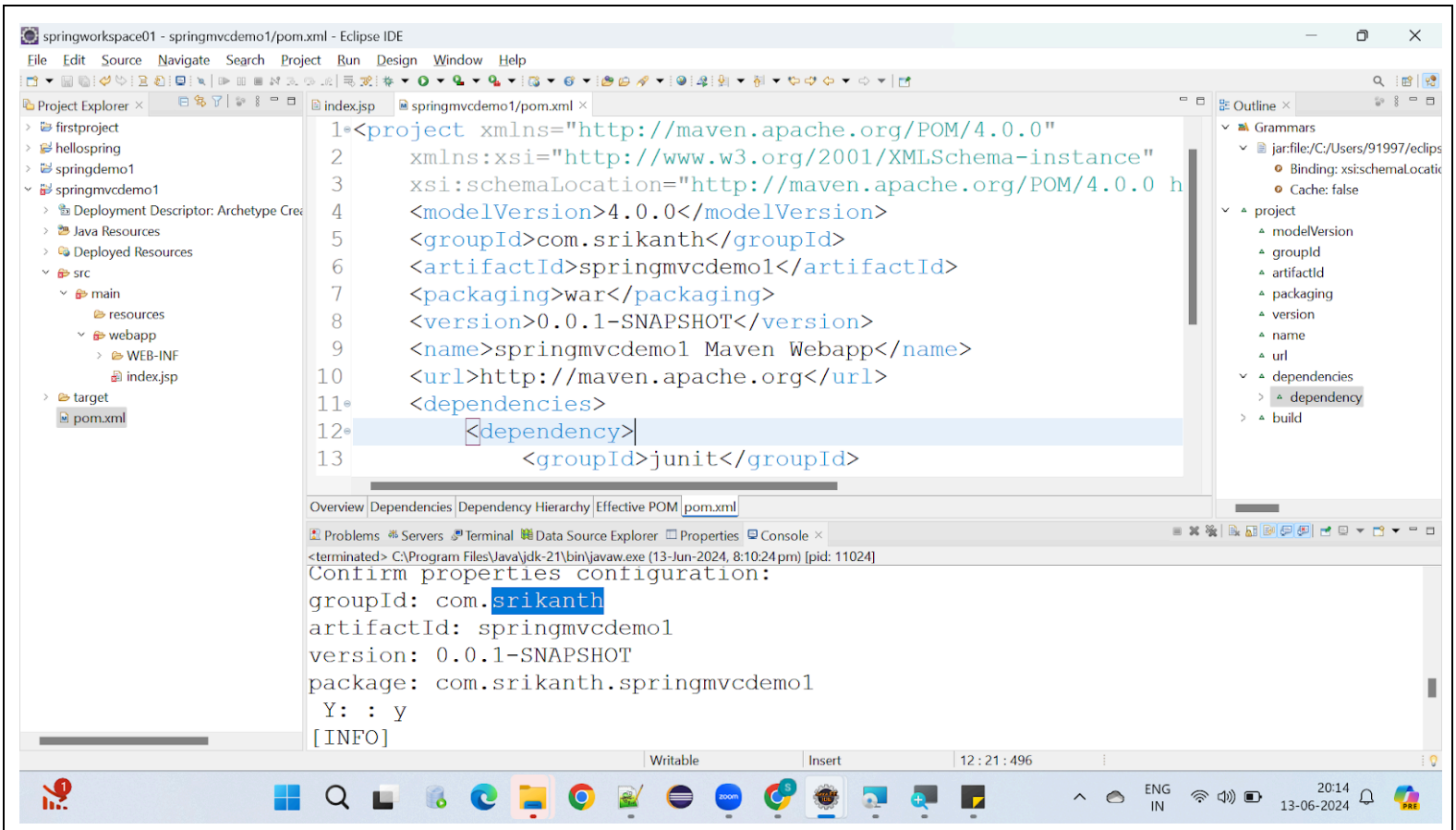


Step 4

Check POM.xml

Add below dependency

```
<dependency>
		<groupId>org.springframework</groupId>
		<artifactId>spring-webmvc</artifactId>
		<version>6.1.0</version>
	</dependency>
```



Step 5

Configure Dispatcher Servlet in web.xml

Spring MVC DispatcherServlet as Front Controller

```
<!DOCTYPE web-app PUBLIC
"-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd" >
<web-app>
     <display-name>Archetype Created Web Application</display-name>
     <servlet>
          <servlet-name>srikanth</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-clas
```

```
s>
     </servlet>
     <servlet-mapping>
          <servlet-name>srikanth</servlet-name>
          <url-pattern>/</url-pattern>
     </servlet-mapping>
</web-app>
```

Spring MVC Example Bean Configuration File

Srikanth-servlet.xml

And configuration for ***InternalResourceViewResolver***

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
     xmlns:ctx="http://www.springframework.org/schema/context"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:context="http://www.springframework.org/schema/context"
     xsi:schemaLocation="
  http://www.springframework.org/schema/mvc
  http://www.springframework.org/schema/mvc.xsd
  http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/context
  http://www.springframework.org/schema/context/spring-context.xsd">
     <ctx:component-scan base-package="com.srikanth">
     </ctx:component-scan>
     <ctx:annotation-config />
     <bean

class="org.springframework.web.servlet.view.InternalResourceViewResolver">
          <property name="prefix" value="/"></property>
          <property name="suffix" value=".jsp"></property>
     </bean>
```

```
</beans>
```

Customer.java

```
package com.srikanth.springmvcdemo1;
public class Customer {
	private int cid;
	private String cname;
	@Override
	public String toString() {
		return "Customer [cid=" + cid + ", cname=" + cname + "]";
	}
	public int getCid() {
		return cid;
	}
	public void setCid(int cid) {
		this.cid = cid;
	}
	public String getCname() {
		return cname;
	}
	public void setCname(String cname) {
		this.cname = cname;
	}
}
```

index.page

```
<%@page language="java"%>
<html>
<head>
</head>
```

```
</body>
<h2>Index Page</h2>
<form action="addCustomer">
     <label for="cid">Enter Id :</label>
     <input type="text" id="cid" name="cid"><br>

     <label for="cname">Enter Name :</label>
     <input type="text" id="cname" name="cname"><br>

     <input type="submit" value="Submit">
</form>
</body>
</html>
```

HomeController.java

```
package com.srikanth.springmvcdemo1;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
@Controller
public class HomeController {
     @ModelAttribute("course")
     public String courseName() {
          return "Java";
     }
     @RequestMapping("/")
     public String home() {
          System.out.println("Home method called");
          return "index";
     }
     @RequestMapping("addCustomer")
     public String addCustomer(Customer cust) {
          // public String addCustomer(@ModelAttribute("customer1") Customer
cust) {
          return "result";
     }
```

```
}
```

result.jsp

```
<%@page language="java" isELIgnored="false"%>
<html>
<head>
</head>
</body>
<h2>Welcome To VCube</h2>
<p>${cust}</p>
<p>Welcome to the ${course} World</p>
</body>
</html>
```

Spring Annotations

Spring Annotations allows us to configure dependencies and implement dependency injection through java programs.

Spring Annotations
Spring framework implements and promotes the principle of control inversion (IOC) or dependency injection (DI) and is in fact an IOC container.

Traditionally, Spring allows a developer to manage bean dependencies by using XML-based configuration.

There is an alternative way to define beans and their dependencies. This method is a Java-based configuration.

Unlike the XML approach, Java-based configuration allows you to manage bean components programmatically.

That's why Spring annotations were introduced.

In this article we will explore the most commonly used Spring Annotations and also look at some example programs.

Spring Annotations List
Some of the spring core framework annotations are:

@Configuration:

- Used to indicate that a class declares one or more @Bean methods.
- These classes are processed by the Spring container "to generate bean definitions and service requests for those beans at runtime."

@Bean:

- Indicates that a method produces a bean to be managed by the Spring container.
- This is one of the most used and important spring annotations.
- @Bean annotation also can be used with parameters like name, initMethod and destroyMethod.

- name – allows you to give the name for bean.
- initMethod – allows you to choose a method which will be invoked on the context register.
- destroyMethod – allows you to choose a method which will be invoked on context shutdown.

For example:

```
@Configuration
public class AppConfig {

    @Bean(name = "comp", initMethod = "turnOn", destroyMethod = "turnOff")
Computer computer(){
        return new Computer();
    }
}
public class Computer {

    public void turnOn(){
        System.out.println("Load operating system");
```

```
    }
    public void turnOff(){
        System.out.println("Close all programs");
    }
}
```

@PreDestroy and @PostConstruct are alternative ways for bean initMethod and destroyMethod.

It can be used when the bean class is defined by us. For example;

```
 public class Computer {

    @PostConstruct
    public void turnOn(){
        System.out.println("Load operating system");
    }

    @PreDestroy
public void turnOff(){
        System.out.println("Close all programs");
    }
}
```

@ComponentScan:

- Configures component scanning directives for use with @Configuration classes.
- Here we can specify the base packages to scan for spring components.

@Component:

- Indicates that an annotated class is a “component”.
- Such classes are considered as candidates for auto-detection when using annotation-based configuration and classpath scanning.

@PropertySource:

- provides a simple declarative mechanism for adding a property source to Spring’s Environment.
- There is a similar annotation for adding an array of property source files i.e @PropertySources.

@Service:

- **Indicates that an annotated class is a "Service".**
- **This annotation serves as a specialization of @Component, allowing for implementation classes to be autodetected through classpath scanning.**

@Repository: -

- **Indicates that an annotated class is a "Repository".**
- **This annotation serves as a specialization of @Component and advisable to use with DAO classes.**

@Autowired:

- **Spring @Autowired annotation is used for automatic injection of beans.**
- **Spring @Qualifier annotation is used in conjunction with Autowired"**
- **to avoid confusion when we have two or more beans configured for the same type."**

==========

Spring MVC Annotations
Some of the important Spring MVC annotations are:

@Controller :

- The @Controller stereotype **allows for auto-detection and is aligned with Spring general support for detecting @Component classes** in the classpath and auto-registering bean definitions for them.
- It also acts as a stereotype for the annotated class, indicating its **role as a web component.**
-

@RequestMapping

- The @RequestMapping annotation **is used to map requests to controllers methods**.
- It has various attributes to match by URL, HTTP method, request parameters, headers, and media types.
- You can use it at the class level to express shared mappings or at the method level to narrow down to a specific endpoint mapping.

@PathVariable

- The @PathVariable annotation **is used to retrieve data from the URL path.**
- By defining placeholders in the request mapping URL, you can bind those placeholders to method parameters annotated with @PathVariable.
- This allows you to access dynamic values from the URL and use them in your code.

@RequestParam

- @RequestParam annotation **enables spring to extract input data that may be passed as a query**, form data, or any arbitrary custom data.
- Here, we will see how we can use @RequestParam when building RESTful APIs for a web-based application.

@ModelAttribute

- Basically, **@ModelAttribute** methods are invoked before the controller methods annotated with **@RequestMapping** are invoked.
- This is because the **model object** has to be created before any processing starts inside the controller methods.
- It's also important that we annotate the respective class as @ControllerAdvice.
-

@RequestBody and @ResponseBody

- @RequestBody and @ResponseBody annotations are used **to bind the HTTP request/response** body with a domain object in the method parameter or return type.
- Behind the scenes, this annotation uses HTTP Message converters to convert the body of HTTP request/response to domain objects.

@RequestHeader and @ResponseHeader

- You can use the @RequestHeader annotation **to bind a request header to a method argument in a controller.**
- The following example gets the value of the Accept-Encoding and Keep-Alive headers: Java.

Spring Transaction Management Annotations

@Transactional is the spring declarative transaction management annotation, read more at Spring MVC Hibernate.

Spring Security Annotations

@EnableWebSecurity is used with @Configuration class to have the Spring Security configuration defined, read more at Spring Security Example.

⇒ **end Spring annotations**

Spring Boot

Spring Boot = Advanced Spring or Extension of Spring.

Micro Service is an Architecture only.

- **Spring Boot is an Open Source java based framework.**
- **By using this we can develop micro service applications.**
- **It is developed by a pivotal team and is used to build stand alone and production ready spring applications.**

- ****Micro Service is an Architecture that allows the developers to develop and deploy services independently.**

- **Each Service is running has its own process and this achieves the lightweight model to support business applications.**

- **Advantages of Micro Services**

1) **Easy Deployment.**
2) **simple Scalability.**
3) **Compatible with containers**
4) **Minimum Configuration.**
5) **Lesser production time.**

SpringBoot 👍

- SpringBoot provides a good platform for Java developers to develop applications.
- SpringBoot is using very Minimum Configuration.
- Anyone can understand easily and develop the applications in SpringBoot.
- Increases productivity.
- Reduces the development time.

Goals :

- Spring Boot is mainly to avoid complex XML configuration in Spring.
- Spring Boot is helping to develop production ready code in an easier way.
- Spring Boot is helping to reduce the development time and run the applications independently.
- Spring Boot offers an easier way of getting started with the applications.

Why Spring Boot ?

- It provides flexible way to configure java beans, (XML Configurations and Database Transactions)
- It provides a powerful batch processing and manages REST endpoints.
- In SB everything is auto configured; No manual configuration is needed.
- It offers annotation based spring applications.
- Eases the dependency Management.
- It includes an Embedded Servlet Container.
- It includes Embedded Server.

How does it work ..?

@EnableAutoConfiguration

- Spring Boot automatically configures your application based on the dependencies you have added to the project by using this annotation.

Ex: Oracle DB is on your classpath, we did not configure any DB connection, Then SB auto configures an in memory database.

@SpringBootApplication

- **This annotation is the entry point of the spring boot application the class contains.**

@ComponentScan

- **By using this annotation Spring Boot automatically scans all the components included in the project.**

Spring Boot Starters

- **Handling dependency management is a difficult task for big projects.**
- **Spring Boot resolves this problem by providing a set of dependencies for developers convenience.**
- **For example, if you want to use Spring and JPA for database access, it is sufficient if you include spring-boot-starter-data-jpa dependency in your project.**

Note that all Spring Boot starters follow the same naming pattern spring-boot-starter- *, where * indicates that it is a type of the application.

Examples

Look at the following Spring Boot starters explained below for a better understanding.

Spring Boot Starter Actuator dependency is used to monitor and manage your application.

Its code is shown below -

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

Spring Boot Starter Security dependency is used for Spring Security.

Its code is shown below -

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

Spring Boot Starter web dependency is used to write Rest Endpoints.

Its code is shown below -

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

Spring Boot Starter Thyme Leaf dependency is used to create a web application.

Its code is shown below -

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

Spring Boot Starter Test dependency is used for writing Test cases.

Its code is shown below -

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

Auto Configuration

- Spring Boot Auto Configuration automatically configures your Spring application based on the JAR dependencies you added in the project.
- For example, if MySQL database is on your classpath, but you have not configured any database connection, then Spring Boot auto configures an in-memory database.
- For this purpose, you need to add @EnableAutoConfiguration annotation or @SpringBootApplication annotation to your main class file.
- Then, your Spring Boot application will be automatically configured.

Observe the following code for a better understanding -

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;

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

Spring Boot Application

- The entry point of the Spring Boot Application is the class containing @SpringBootApplication annotation.
- This class should have the main method to run the Spring Boot application.
- @SpringBootApplication annotation includes
- Auto- Configuration, Component Scan, and Spring Boot Configuration.
- If you added @SpringBootApplication annotation to the class, you do not need to add the

  @EnableAutoConfiguration,

  @ComponentScan and

  @SpringBootConfiguration annotation.

- The @SpringBootApplication annotation includes all other annotations.

Observe the following code for a better understanding -

```
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);
   }
}
```

Component Scan

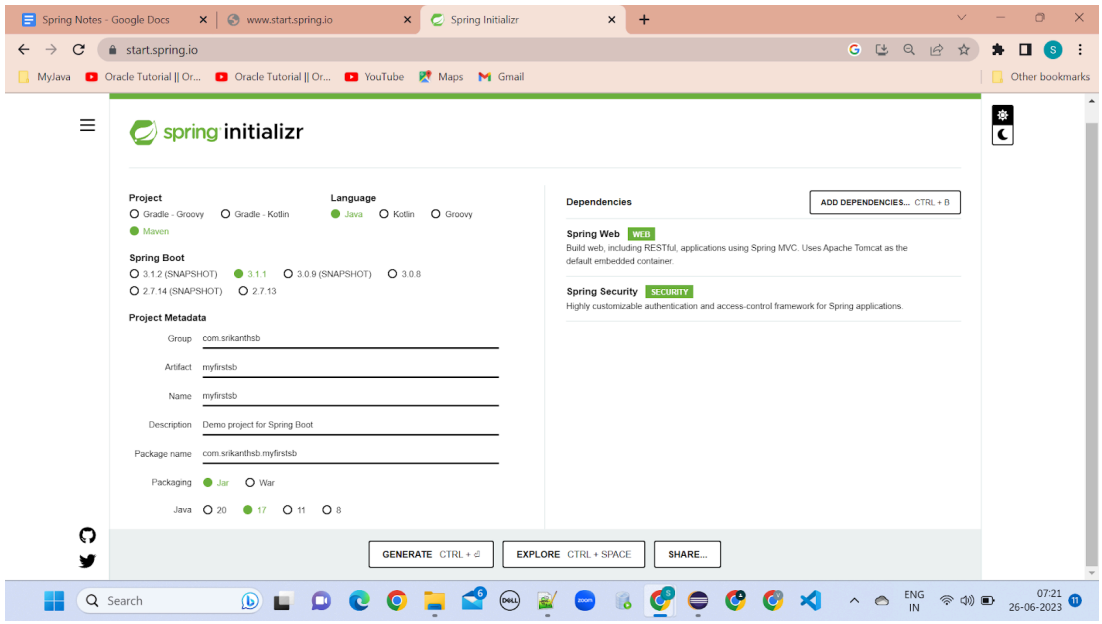
- Spring Boot application scans all the beans and package declarations when the application initializes.
- You need to add the @ComponentScan annotation for your class file to scan your components added in your project.

Observe the following code for a better understanding -

```
import org.springframework.boot.SpringApplication;
import org.springframework.context.annotation.ComponentScan;

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

Spring Boot - Bootstrapping

- One of the ways to Bootstrapping a Spring Boot application is by using Spring Initializer.
- To do this, you will have to visit the Spring Initializer web page (https://start.spring.io/) and choose your Build, Spring Boot Version and platform. Also, you need to provide a Group, Artifact and required dependencies to run the application.



- **Observe the following screenshot that shows an example where we added the spring-boot-starter-web dependency to write REST Endpoints.**
- **Once you provide the Group, Artifact, Dependencies, Build Project, Platform and Version, click the Generate Project button. The zip file will download and the files will be extracted.**
- **This section explains the examples by using both Maven and Gradle.**

***** Steps to Create Spring Boot Application :**

Step 1: Open the Spring initializr https://start.spring.io.

Step 2: Provide the Group and Artifact name.

We have provided the

- Group name com.srikanthspring and
- Artifact myfirstsb.

Step 3: Now click on the Generate button.

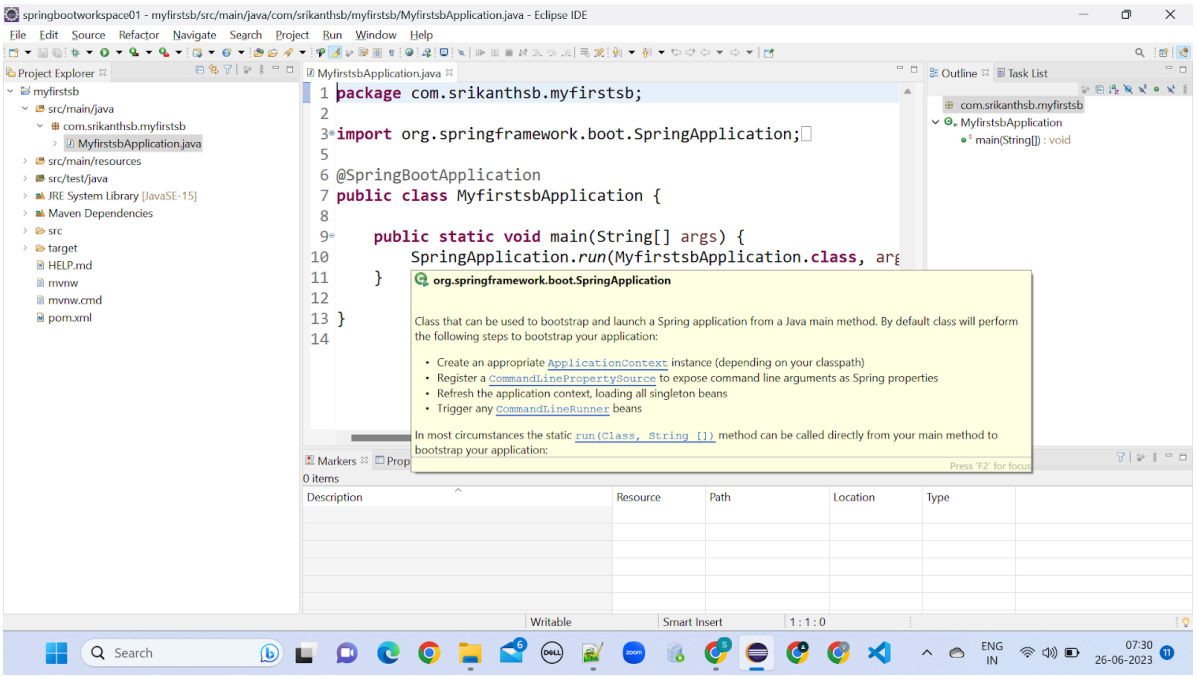
When we click on the Generate button, it starts packing the project in a .jar file and downloads the project.

Step 4: Extract the Jar file.

Step 5: Import the folder.

File -> Import -> Existing Maven Project -> Next -> Browse -> Select the project -> Finish

- It takes some time to import the project.
- When the project imports successfully, we can see the project directory in the Package Explorer.
- The following image shows the project directory:



SpringBootExampleApplication.java

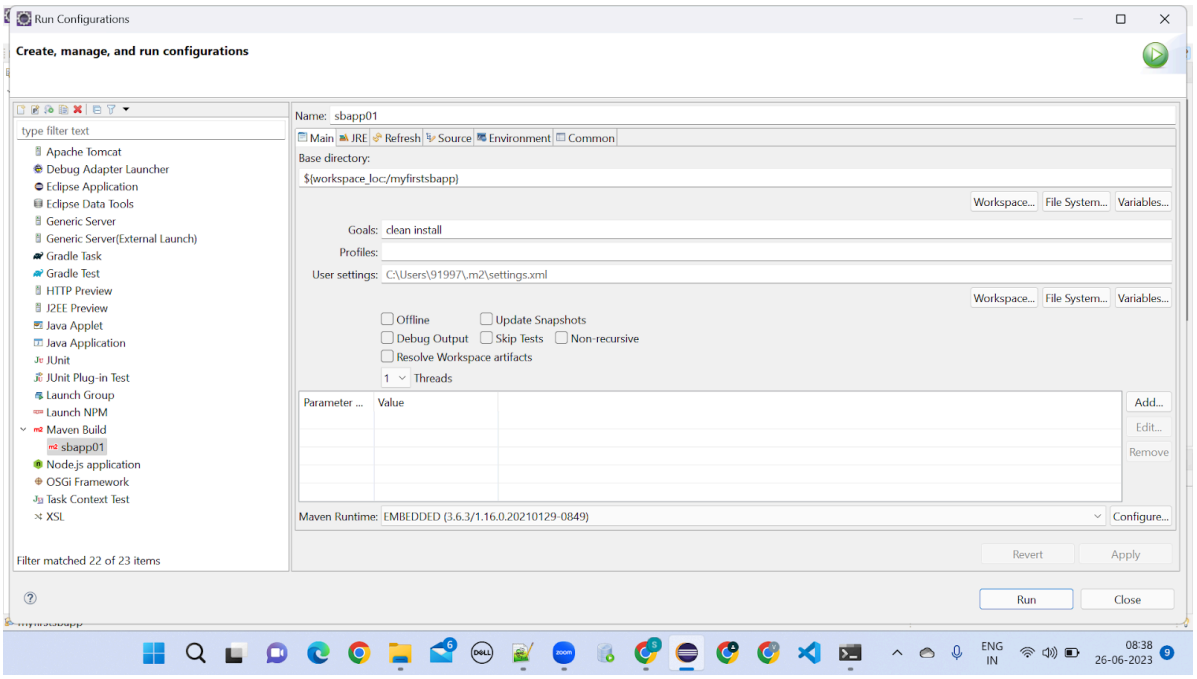
```
package com.srikanthsb.myfirstsb;

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

@SpringBootApplication
public class MyfirstsbApplication {

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

}
```

POM.xml

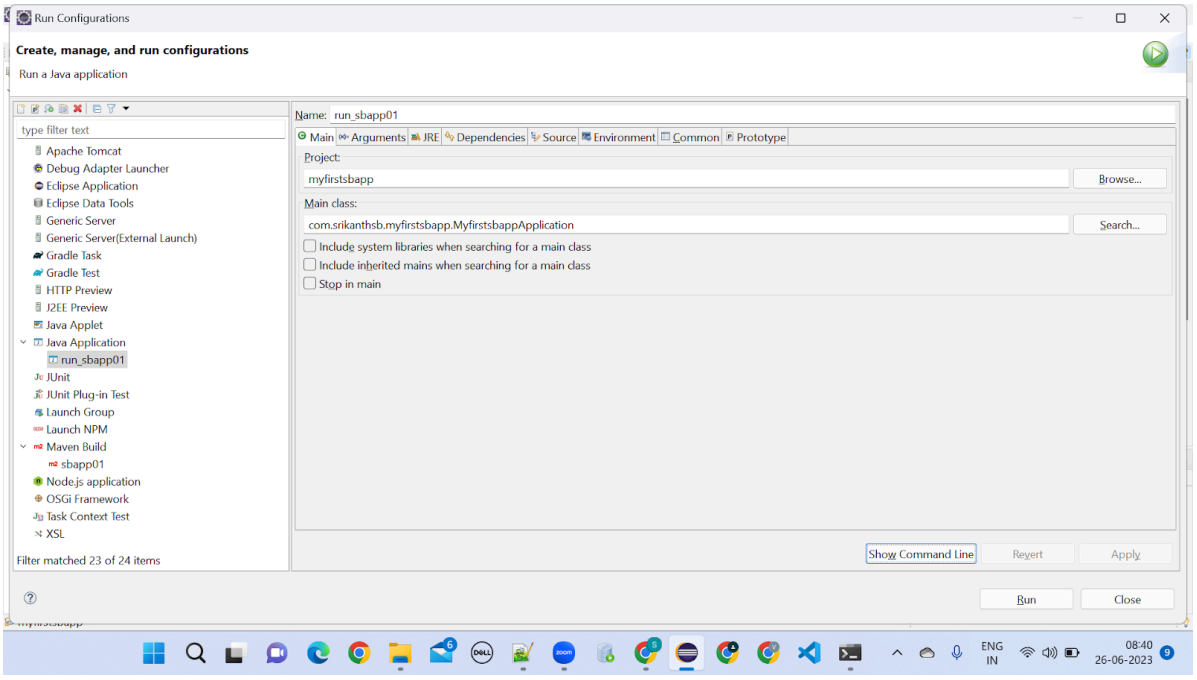
```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
	<modelVersion>4.0.0</modelVersion>
	<parent>
		<groupId>org.springframework.boot</groupId>
		<artifactId>spring-boot-starter-parent</artifactId>
		<version>3.1.1</version>
		<relativePath/> <!-- lookup parent from repository -->
	</parent>
	<groupId>com.srikanthsb</groupId>
	<artifactId>myfirstsb</artifactId>
	<version>0.0.1-SNAPSHOT</version>
	<name>myfirstsb</name>
	<description>Demo project for Spring Boot</description>
	<properties>
		<java.version>17</java.version>
	</properties>
```

```
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter</artifactId>
        </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>
            </plugin>
        </plugins>
    </build>

</project>
```

Step 5 : We need build and run the application
Build the application.



Run the application.



***SpringBoot Annotation Demo

Spring Annotation Example

```
package com.srikanth.springbootdemo1;

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

@SpringBootApplication
public class Springbootdemo1Application {

     public static void main(String[] args) {
          ApplicationContext context =
SpringApplication.run(Springbootdemo1Application.class, args);

          Employee obj1 = context.getBean(Employee.class);
          System.out.println(obj1.getAge());
          obj1.code();
     }

}
```

```
package com.srikanth.springbootdemo1;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;
@Component
public class Employee {
     @Value("21")
     int age;
     Computer computer;
     public int getAge() {
          return age;
     }
```

```
	public void setAge(int age) {
		this.age = age;
	}
	public Computer getComputer() {
		return computer;
	}
	@Autowired
	@Qualifier("laptop")
	public void setComputer(Computer computer) {
		this.computer = computer;
	}
	public void code() {
		computer.doSomething();
	}
}
```

```
package com.srikanth.springbootdemo1;
public interface Computer {
	 void doSomething();
}
```

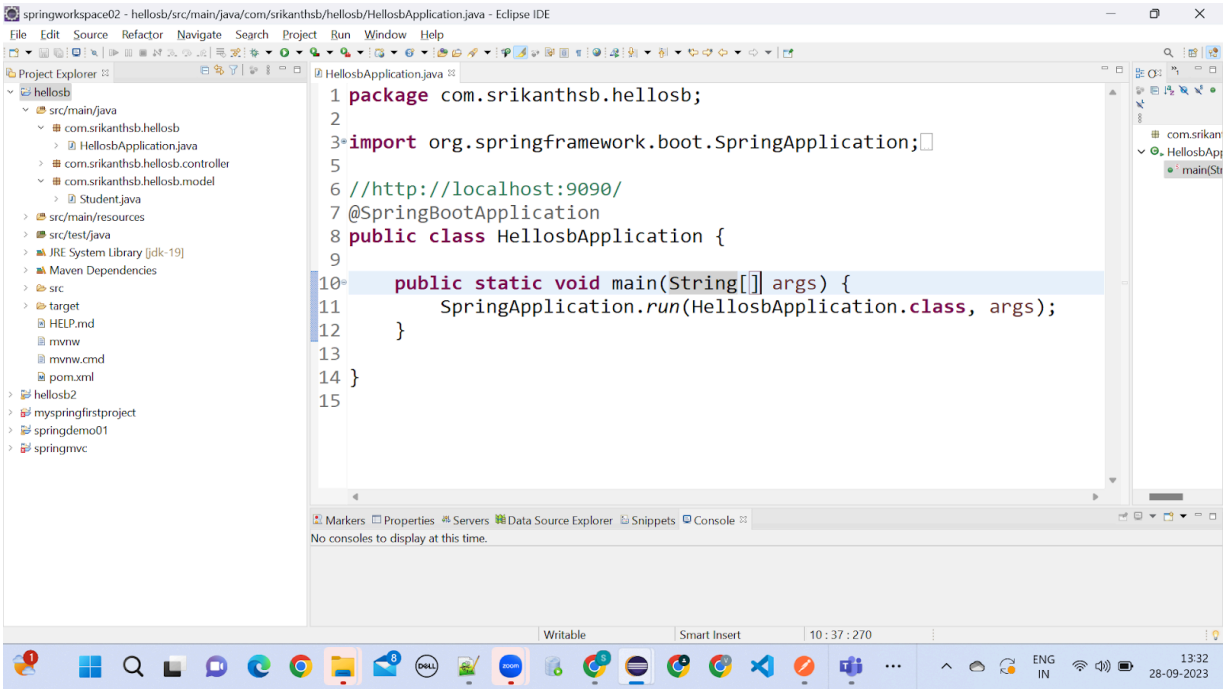
```
package com.srikanth.springbootdemo1;
import org.springframework.context.annotation.Primary;
import org.springframework.stereotype.Component;
@Component
@Primary
public class Desktop implements Computer {
	@Override
	@Primary
	public void doSomething() {
		System.out.println("Hello Desktop !!");
	}
```

```
}
```

```
package com.srikanth.springbootdemo1;
import org.springframework.stereotype.Component;
@Component
public class Laptop implements Computer {
     @Override
     public void doSomething() {
          System.out.println("Hello Laptop !!");
     }
}
```

***01 SBApplication 01 (hellosbapplication)

Small REST API



HellosbApplication .java

```
package com.srikanthsb.hellosb;

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

//http://localhost:9090/
@SpringBootApplication
public class HellosbApplication {

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

}
```

HelloSbController .java

```
package com.srikanthsb.hellosb.controller;

import java.util.ArrayList;
import java.util.List;

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

import com.srikanthsb.hellosb.model.Student;

@RestController
public class HelloSbController {

     @RequestMapping
     String helloSpringBoot() {
          String str = "Hello Spring Boot Application !!";
          return str;
     }

//http://localhost:9090/sri
     @GetMapping("/sri")
     public String hello() {
          return "Hello Spring Boot app or Micro Sercices by
Srikanth!!";
     }

     @GetMapping("/student")
     Student getStudent() {
          return new Student("Abdul", "Razak ");
     }

     @GetMapping("/students")
     List<Student> getEmployees() {
          List<Student> empList = new ArrayList<Student>();
          empList.add(new Student("Aditya", "A"));
```

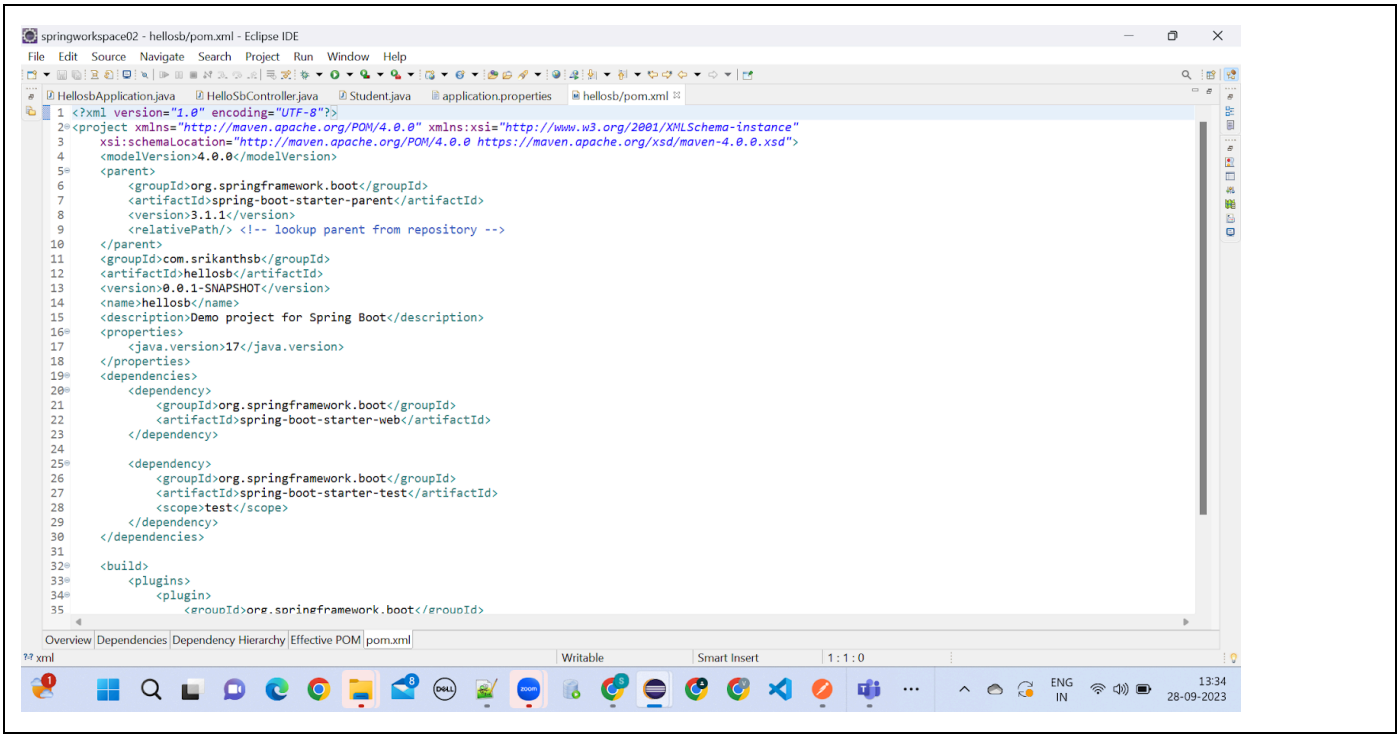
```
          empList.add(new Student("Kruthik", "k"));
          empList.add(new Student("Naina", "Shaik"));
          empList.add(new Student("Zaheer", "s"));
          empList.add(new Student("Manga", "m"));
          return empList;
     }
}
```

Student.java

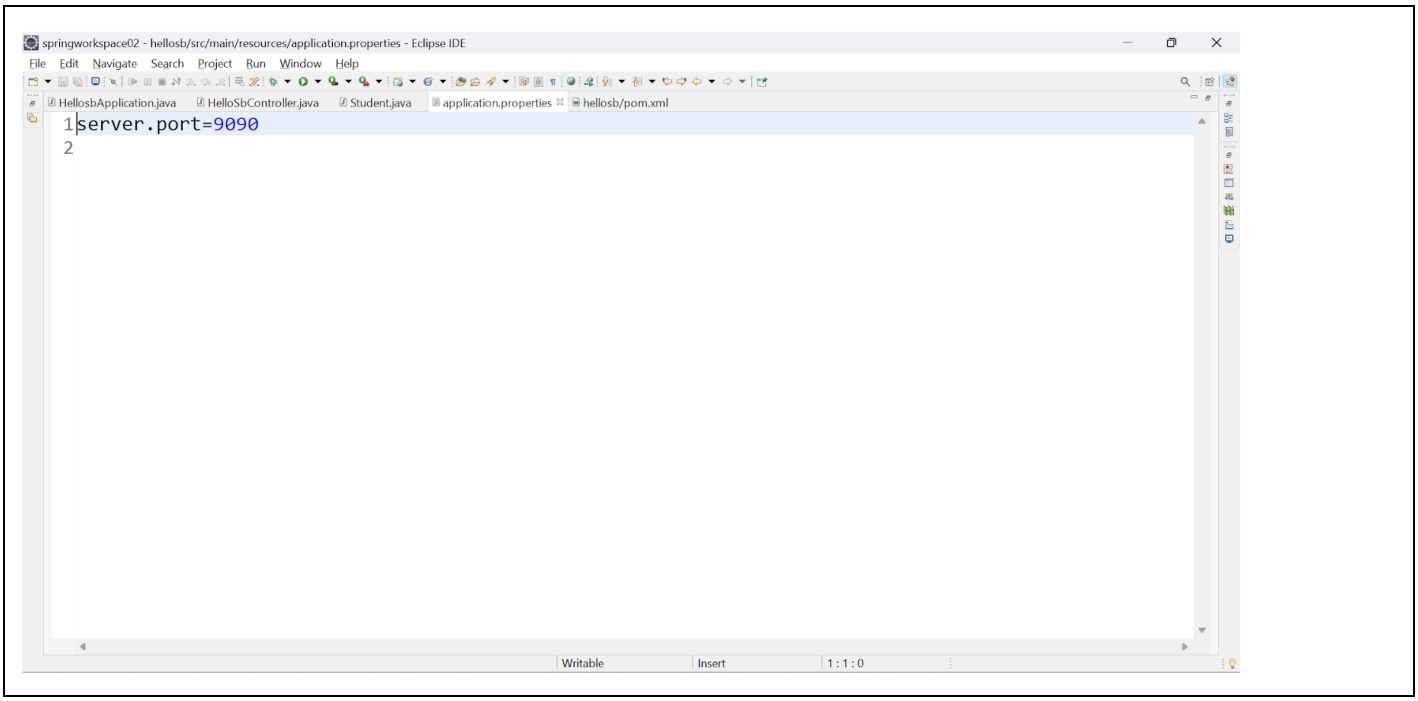
```
package com.srikanthsb.hellosb.model;

public class Student {

     private String firstname;
     private String lastname;

     public Student(String firstname, String lastname) {
          this.firstname = firstname;
          this.lastname = lastname;
     }

     public String getFirstname() {
          return firstname;
     }

     public void setFirstname(String firstname) {
          this.firstname = firstname;
     }

     public String getLastname() {
          return lastname;
     }
```

```
	public void setLastname(String lastname) {
		this.lastname = lastname;
	}

}
```

Pom.xml



application.properties



-----End of application 01----------

02 Spring Boot Web application 01

Step 1:

- **Navigate to https://start.spring.io/**
- **This service pulls in all the dependencies you need for an application and does most of the setup for you.**

Step 2:

- **Choose Maven and the language is Java.**

Step 3:

- **Click Dependencies and select Spring Web and Thymeleaf.**
- **Click Generate.**

Step 4:

- **Download the resulting ZIP file, which is an archive of a web application that is configured with your choices.**

Step 5:

- **Create an Unsecured Web Application**
- **Before you can apply security to a web application, you need a web application to secure it.**
- **This section walks you through creating a simple web application.**
- **Then you will secure it with Spring Security in the next section.**
- **The web application includes two simple views: a home page and a "Hello, World" page.**
- **The home page is defined in the following Thymeleaf template (from src/main/resources/templates/home.html):**

- **This simple view includes a link to the /hello page, which is defined in the following Thymeleaf template (from src/main/resources/templates/hello.html):**

home.html

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="https://www.thymeleaf.org">
<head>
     <title>Spring Security Example</title>
</head>
<body>
     <h1>Welcome!</h1>
     <p>Click <a th:href="@{/hello}">here</a> </a> to see a greeting.</p>
</body>
</html>
```

hello.html

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="https://www.thymeleaf.org">
<head>
     <title>Hello World!</title>
</head>
```

```
<body>
     <h1 th:inline="text">Hello
          <span th:remove="tag" sec:authentication="name">thymeleaf</span>!
     </h1>
     <form th:action="@{/logout}" method="post">
          <input type="submit" value="Sign Out" />
     </form>
</body>
</html>
```

- The web application is based on Spring MVC. As a result, you need to configure Spring MVC and set up view controllers to expose these templates.
- The following listing (from src/main/java/com/example/securingweb/MvcConfig.java) shows a class that configures Spring MVC in the application

MvcConfig.java

```
package com.sb.srikanth.webapp.sbwebapp01.securingweb;
import org.springframework.context.annotation.Configuration;
import
org.springframework.web.servlet.config.annotation.ViewControllerRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
@Configuration
public class MvcConfig implements WebMvcConfigurer {
     public void addViewControllers(ViewControllerRegistry registry) {
          registry.addViewController("/home").setViewName("home");
          registry.addViewController("/").setViewName("home");
          registry.addViewController("/hello").setViewName("hello");
          registry.addViewController("/login").setViewName("login");
     }
}
```

- The addViewControllers() method

(which overrides the method of the same name in WebMvcConfigurer) adds four view controllers.

Two of the view controllers reference the view whose name is home (defined in home.html), and another references the view named hello (defined in hello.html).

The fourth view controller references another view named login. You will create that view in the next section.

At this point, you could jump ahead to “Run the Application” and run the application without having to log in to anything.

Now that you have an unsecured web application, you can add security to it.

WebSecurityConfig.java

```
package com.sb.srikanth.webapp.sbwebapp01.securingweb;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import
org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.config.annotation.web.configuration.EnableWebSecur
ity;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.provisioning.InMemoryUserDetailsManager;
import org.springframework.security.web.SecurityFilterChain;
@Configuration
@EnableWebSecurity
public class WebSecurityConfig {
     @Bean
     public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {
          http.authorizeHttpRequests(
```

```
                    (requests) -> requests.requestMatchers("/",
"/home").permitAll().anyRequest().authenticated())
                    .formLogin((form) ->
form.loginPage("/login").permitAll()).logout((logout) -> logout.permitAll());
          return http.build();
     }
     @Bean
     public UserDetailsService userDetailsService() {
          UserDetails user =
User.withDefaultPasswordEncoder().username("Srikanth").password("Spring123").ro
les("USER")
                    .build();
          return new InMemoryUserDetailsManager(user);
     }
}
```

login.html

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="https://www.thymeleaf.org">
<head>
     <title>Spring Security Example </title>
</head>
<body bgcolor="lightgreen">
     <div th:if="${param.error}">
          Invalid username and password.
     </div>
     <div th:if="${param.logout}">
          You have been logged out.
     </div>
     <form th:action="@{/login}" method="post">
          <div><label> User Name : <input type="text" name="username" />
</label></div>
          <div><label> Password: <input type="password" name="password" />
</label></div>
          <div><input type="submit" value="Sign In" /></div>
     </form>
</body>
</html>
```

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>3.2.2</version>
          <relativePath /> <!-- lookup parent from repository -->
     </parent>
     <groupId>com.sb.srikanth.webapp</groupId>
     <artifactId>sbwebapp01</artifactId>
     <version>0.0.1-SNAPSHOT</version>
     <name>sbwebapp01</name>
     <description>Demo project for Spring Boot</description>
     <properties>
          <java.version>21</java.version>
     </properties>
     <dependencies>
          <dependency>
               <groupId>org.springframework.boot</groupId>
               <artifactId>spring-boot-starter-thymeleaf</artifactId>
          </dependency>
          <dependency>
               <groupId>org.springframework.boot</groupId>
               <artifactId>spring-boot-starter-web</artifactId>
          </dependency>
          <dependency>
               <groupId>org.springframework.boot</groupId>
               <artifactId>spring-boot-starter-test</artifactId>
               <scope>test</scope>
          </dependency>
```

```
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-security</artifactId>
		</dependency>
		<dependency>
			<groupId>org.thymeleaf.extras</groupId>
			<artifactId>thymeleaf-extras-springsecurity6</artifactId>
			<!-- Temporary explicit version to fix Thymeleaf bug -->
		</dependency>
		<dependency>
			<groupId>org.springframework.security</groupId>
			<artifactId>spring-security-test</artifactId>
			<scope>test</scope>
		</dependency>
	</dependencies>
	<build>
		<plugins>
			<plugin>
				<groupId>org.springframework.boot</groupId>
				<artifactId>spring-boot-maven-plugin</artifactId>
			</plugin>
		</plugins>
	</build>
</project>
```

-----End of application 02----------

03 Spring Boot Web application 02 using Database with JPA 01

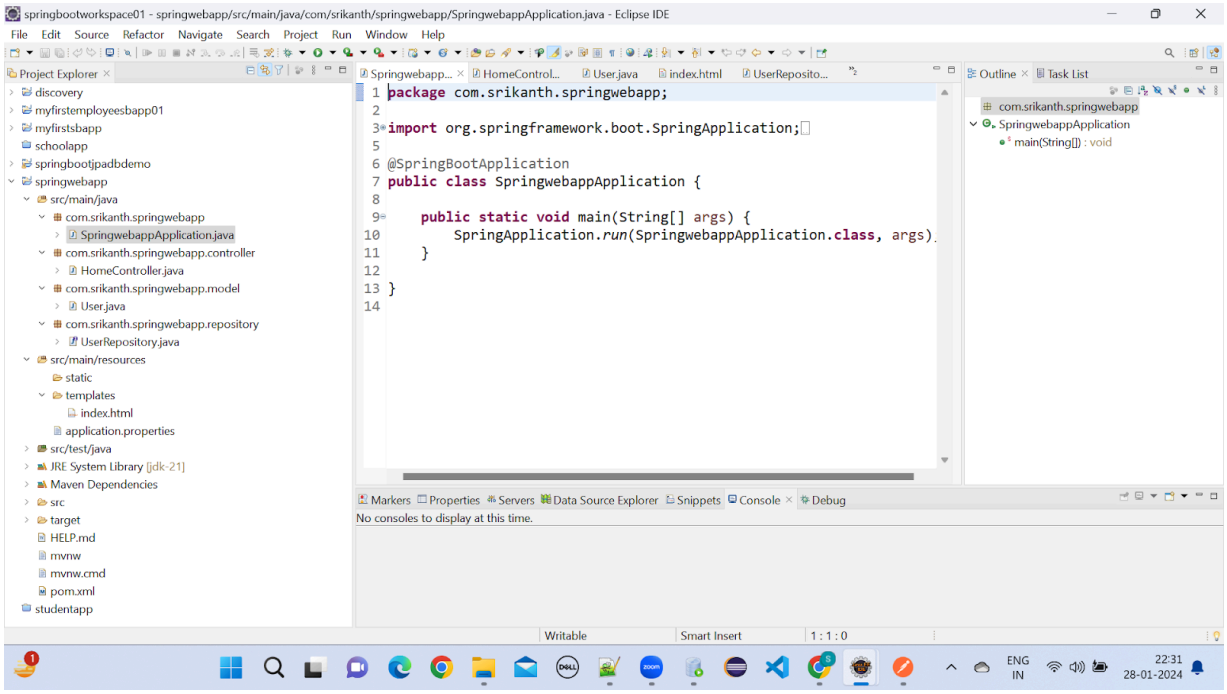
POM.xml

```
<?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>3.1.1</version>
         <relativePath /> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.srikanth</groupId>
    <artifactId>springwebapp</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>springwebapp</name>
    <description>Demo project for Spring Boot</description>
    <properties>
         <java.version>17</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>org.springframework.boot</groupId>
              <artifactId>spring-boot-starter-thymeleaf</artifactId>
         </dependency>
         <dependency>
              <groupId>com.h2database</groupId>
              <artifactId>h2</artifactId>
              <scope>runtime</scope>
         </dependency>
         <dependency>
              <groupId>com.oracle.database.jdbc</groupId>
              <artifactId>ojdbc8</artifactId>
              <scope>runtime</scope>
         </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>
            </plugin>
        </plugins>
    </build>
</project>
```

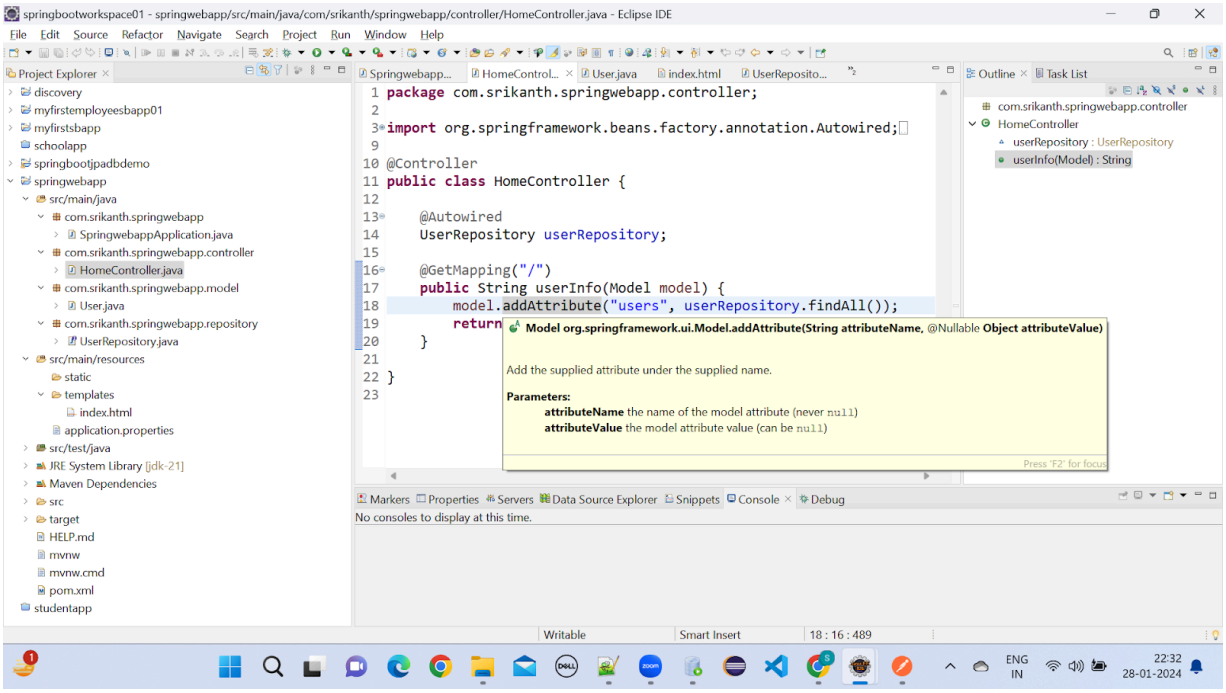
SpringwebappApplication.java




```
package com.srikanth.springwebapp;

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

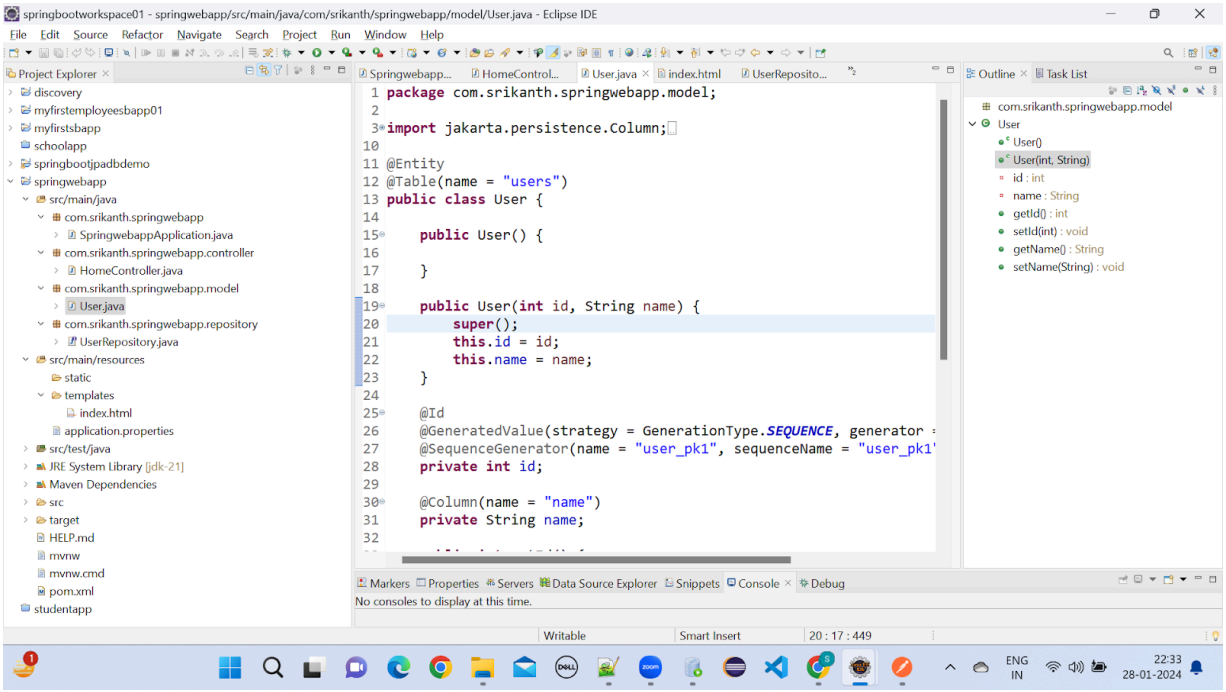
```
@SpringBootApplication
public class SpringwebappApplication {

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

}
```

HomeController .java




```java
package com.srikanth.springwebapp.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;

import com.srikanth.springwebapp.repository.UserRepository;

@Controller
public class HomeController {

	@Autowired
	UserRepository userRepository;

	@GetMapping("/")
	public String userInfo(Model model) {
		model.addAttribute("users", userRepository.findAll());
		return "index";
	}

}
```

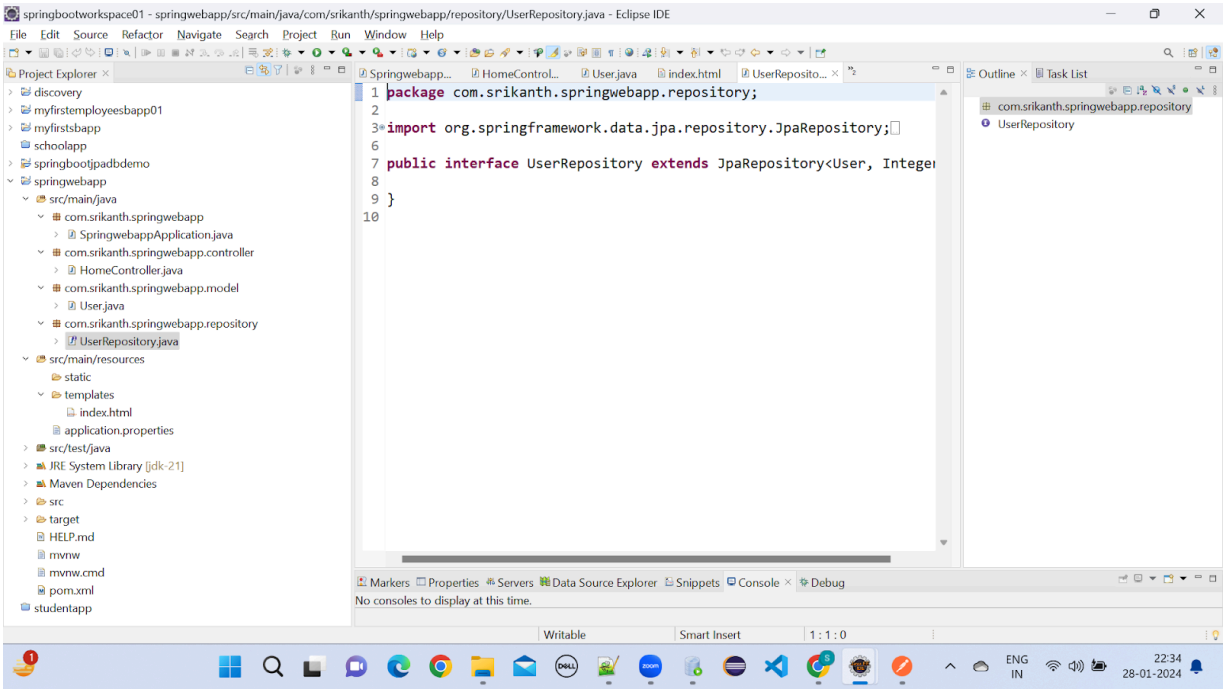
User.java




```
package com.srikanth.springwebapp.model;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.SequenceGenerator;
import jakarta.persistence.Table;

@Entity
@Table(name = "users")
public class User {

	public User() {

	}
```

```
    public User(int id, String name) {
        super();
        this.id = id;
        this.name = name;
    }

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator =
"user_pk1")
    @SequenceGenerator(name = "user_pk1", sequenceName = "user_pk1",
allocationSize = 1)
    private int id;

    @Column(name = "name")
    private String name;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

}
```

UserRepository.java




```
package com.srikanth.springwebapp.repository;

import org.springframework.data.jpa.repository.JpaRepository;

import com.srikanth.springwebapp.model.User;

public interface UserRepository extends JpaRepository<User,
Integer> {

}
```

index.html

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="utf-8"/>
<title>Spring Boot Web App</title>
</head>
```

```
<body>
     <h1>Spring Boot Web App</h1>
     <hr />
     <table>
          <thead>
               <tr>
                    <th>ID</th>
                    <th>Name</th>
               </tr>
          </thead>
          <tbody>
               <tr th:each="user :${users}">
                    <td th:text="${user.id}">Id</td>
                    <td th:text="${user.name}">Name</td>
               </tr>
          </tbody>
     </table>
</body>
</html>
```

application.properties

```
#create and drop tables
spring.jpa.hibernate.ddl.auto=update
spring.jpa.show-sql=true
#Oracle Settings
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:xe
spring.datasource.username=SYSTEM
spring.datasource.password=Oracle2
spring.datasource.driver-class-name=oracle.jdbc.driver.OracleDr
iver
#server configuration
server.port=9090
server.error.whitelabel.enabled=false
```

-----End of application 03----------

04 Spring Boot REST API using JPA 02

Create Table

```
CREATE TABLE employee6 (
    ID int,
    LastName varchar(255),
    FirstName varchar(255),
    email varchar(255),
    primary key (ID)
);
```

Create SEQUENCE

```
CREATE SEQUENCE sequence_6
start with 1
increment by 1
minvalue 0
maxvalue 100
cycle;
```

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>3.1.1</version>
         <relativePath /> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.srikanthsb</groupId>
    <artifactId>springbootjpadbdemo</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>springbootjpadbdemo</name>
    <description>Demo project for Spring Boot</description>
    <properties>
         <java.version>17</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.oracle.database.jdbc</groupId>
              <artifactId>ojdbc8</artifactId>
              <scope>12.2.0.1</scope>
         </dependency>
         <dependency>
              <groupId>org.springframework.boot</groupId>
              <artifactId>spring-boot-starter-test</artifactId>
              <scope>test</scope>
         </dependency>
         <dependency>
              <groupId>org.springframework.boot</groupId>
              <artifactId>spring-boot-devtools</artifactId>
              <scope>runtime</scope>
         </dependency>
    </dependencies>
```

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

SpringbootjpadbdemoApplication .java

```
package com.srikanthsb.springbootjpadbdemo;

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

@SpringBootApplication
public class SpringbootjpadbdemoApplication {

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

}
```

EmployeeController .java

```
package com.srikanthsb.springbootjpadbdemo.controller;

import java.util.HashMap;
import java.util.List;
import java.util.Map;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
```

```
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import com.srikanthsb.springbootjpadbdemo.exception.ResourceNotFoundException;
import com.srikanthsb.springbootjpadbdemo.model.Employee;
import com.srikanthsb.springbootjpadbdemo.repository.EmployeeRepository;

@RestController
@RequestMapping("/api/v1")
public class EmployeeController {

    @Autowired
    EmployeeRepository employeeRepository;

    /*
     * URL for Create or insert employee Object into DB
     * http://localhost:9090/api/v1/employees
     *
     * @RequestMapping(method = RequestMethod.POST).
     *
     */
    @PostMapping("/employees")
    public Employee createEmployee(@RequestBody Employee employee) {
        return employeeRepository.save(employee);
    }

    /*
     * GET All Employees http://localhost:9090/api/v1/employees
     */ @GetMapping("/employees")
    public List<Employee> getAllEmployees() {
        return employeeRepository.findAll();
    }
```

```
    @GetMapping("/employees/{id}")
    public ResponseEntity<Employee> getEmployeeById(@PathVariable(value = "id")
Long employeeId)
            throws ResourceNotFoundException {

        Employee employee = employeeRepository.findById(employeeId)
                .orElseThrow(() -> new ResourceNotFoundException("Employee
Not found for this id : " + employeeId));

        return ResponseEntity.ok().body(employee);
    }

    @PutMapping("/employees/{id}")
    public ResponseEntity<Employee> updateEmployee(@PathVariable(value = "id")
Long employeeId,
            @RequestBody Employee employeeDetails) throws
ResourceNotFoundException {

        Employee employee = employeeRepository.findById(employeeId)
                .orElseThrow(() -> new ResourceNotFoundException("Employee
Not found for this id : " + employeeId));

        employee.setFirstName(employeeDetails.getFirstName());
        employee.setLastName(employeeDetails.getLastName());
        employee.setEmailId(employeeDetails.getEmailId());

        final Employee updatedEmployee = employeeRepository.save(employee);
        return ResponseEntity.ok(updatedEmployee);

    }
```

Employee.java

```
package com.srikanthsb.springbootjpadbdemo.model;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.SequenceGenerator;
import jakarta.persistence.Table;

//Data Model
@Entity
@Table(name = "employee3")
public class Employee {

     private long id;
     private String firstName;
     private String lastName;
     private String emailId;

     public Employee() {
          super();
     }

     public Employee(long id, String firstName, String lastName, String emailId)
{
          super();
          this.id = id;
          this.firstName = firstName;
          this.lastName = lastName;
          this.emailId = emailId;
     }

//   @GeneratedValue(strategy = GenerationType.AUTO)
     @Id
     @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "emp_pk1")
     @SequenceGenerator(name = "emp_pk1", sequenceName = "emp_pk1",
allocationSize = 1)
     public long getId() {
          return id;
```

```
    }

    public void setId(long id) {
        this.id = id;
    }

    @Column(name = "firstname", nullable = false)
    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    @Column(name = "lastname", nullable = false)
    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    @Column(name = "emailid", nullable = false)
    public String getEmailId() {
        return emailId;
    }

    public void setEmailId(String emailId) {
        this.emailId = emailId;
    }
}
```

EmployeeRepository.java

```java
package com.srikanthsb.springbootjpadbdemo.repository;

import org.springframework.stereotype.Repository;
import com.srikanthsb.springbootjpadbdemo.model.Employee;
import org.springframework.data.jpa.repository.JpaRepository;

@Repository
public interface EmployeeRepository extends JpaRepository<Employee, Long> {

}
```

ResourceNotFoundException.java

```java
package com.srikanthsb.springbootjpadbdemo.exception;
public class ResourceNotFoundException extends Exception {
     public ResourceNotFoundException(String string) {
          super(string);
     }
}
```

application.properties

```
#create and drop tables
spring.jpa.hibernate.ddl.auto=create-drop
#Oracle Settings
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:xe
spring.datasource.username=SYSTEM
spring.datasource.password=Oracle3
spring.datasource.driver-class-name=oracle.jdbc.driver.OracleDriver
#server configuration
```

```
server.port=9292
```

----- End of application 04 ----------

05 Spring Boot Rest API without JPA

```
package com.srikanth.springbootrestapi;

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

@SpringBootApplication
public class SpringbootRestApiApplication {

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

}
```

```
package com.srikanth.springbootrestapi.bean;
public class Student {
	private int id;
	private String firstName;
	private String lastName;
	public Student(int id, String firstName, String lastName) {
		super();
```

```
            this.id = id;
            this.firstName = firstName;
            this.lastName = lastName;
      }
      public int getId() {
            return id;
      }
      public void setId(int id) {
            this.id = id;
      }
      public String getFirstName() {
            return firstName;
      }
      public void setFirstName(String firstName) {
            this.firstName = firstName;
      }
      public String getLastName() {
            return lastName;
      }
      public void setLastName(String lastName) {
            this.lastName = lastName;
      }
}
```

```
package com.srikanth.springbootrestapi.controller;

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

//Controller should be a suffix for a controller class.

//We use @Controller annotation to make a Java class as a Spring MVC Controller.
//The @ResponseBody annotation tells a controller that the object
//returned is automatically serialized into JSON and passed back into
HttpResponse Object.

//Spring 4.0 Released @RestController and this annotation is combination of
```

```
//@Controller and @ResponseBody

@RestController
public class HelloWorldController {

     // HTTP Get Request
     // We use @GetMapping annotation to map HTTP GET request onto specific
Handler
     // method.
     // http://localhost:8080/hello-world

     @GetMapping("hello-world")
     public String helloWorld() {
          return "Hello World with SB App !!";

     }

}
```

```
package com.srikanth.springbootrestapi.controller;

import java.util.ArrayList;
import java.util.List;

import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;

import com.srikanth.springbootrestapi.bean.Student;
```

```
@RestController
public class StudentController {

	/**
	 * Spring Boot API Returns Java Bean that returns JSON to the client.
	 * http://localhost:8181/student
	 *
	 *
	 * The client got the response JSON as Java Bean
	 *
	 * @return
	 */
	@GetMapping("student")
	public Student getStudent() {
		Student student = new Student(1, "Srikanth", "C");
		return student;
	}

	/**
	 * Create Spring Boot REST API returns List(JSON Format)
	 * http://localhost:8080:student
	 *
	 * @return
	 */
	@GetMapping("students")
	public List<Student> getStudents() {
		List<Student> students = new ArrayList<>();

		students.add(new Student(1, "Srikanth", "C"));
		students.add(new Student(2, "Advaith", "C"));
		students.add(new Student(3, "venkat", "C"));
		students.add(new Student(4, "Dhoni", "M"));
		students.add(new Student(5, "Sachin", "tendulkar"));

		return students;
	}

	/**
	 * @param studentId
```

```
     * @param firstName
     * @param lastName
     * @return
     *
     *         Create Spring Boot Rest API with PathVariable @PathVariable.
     * @PathVariable annotation used on a method argument
     * to bind it to the value of a URI template Variable URL :
     *               http://localhost:8080/students/1/sri/c
     *
     */
    @GetMapping("students/{id}/{first-name}/{last-name}")
    public Student getStudentByPathVariable(
              @PathVariable("id") int studentId,
              @PathVariable("first-name") String firstName,
              @PathVariable("last-name") String lastName) {
         return new Student(studentId, firstName, lastName);
    }

    /**
     * @param id
     * @param firstName
     * @param lastName
     * @return
     *
     *         Create Spring Boot Rest API with RequestParam (@RequestParam)
     *
http://localhost:8080/students/hello?id=1&firstName=Srikanth&lastName=C
     *         (Query Parameters )
     *
     *         Difference Between @RequestParam and @PathVariable
     * @RequestParam : To Extract the value of Query parameters in a Request
     * @PathVariable : To Bind the value of URI Template Variable into method
     *               variable
     *
     */
    @GetMapping("students/hello")
    public Student getStudentByRequestParam(
              @RequestParam int id,
              @RequestParam String firstName,
              @RequestParam String lastName) {
```

```
          return new Student(id, firstName, lastName);
     }

     /**
      * @param student
      * @return
      *
      *         Create Spring REST API that handles HTTP POST Request -
Creating a
      *         new Resource
      *
      *         http://localhost:8080/students/create
      *
      * @PostMapping annotation is used for mapping HTTP POST request onto
Specific
      *              Handler method
      * @RequestBody annotation is responsible for retrieving the HTTP request
body
      *              and automatically converting it to the Java Object.
      * @ResponseStatus annotation is used to response status.
      *
      *                 ResponseEntiry is to create the Response for Rest API
      *
      *
      * @PostMapping and @RequestBody
      */
     @PostMapping("student/create")
     @ResponseStatus(HttpStatus.CREATED) // 201 Created
     public Student createStudent(@RequestBody Student student) {

          return student;
     }

     /**
      * @param student
      * @param studentId
      * @return
      *
      *         Create Spring REST API that handles HTTP PUT Request - Updating
      *         Existing Resource
```

```
      *
      *         http://localhost:8181:students/3/update
      *
      * @PutMapping annotation is used for mapping HTTP PUT request onto
specific
      *             handler method
      * @RequestBody
      *
      *
      */
     @PutMapping("students/{id}/update")
     public Student updateStudent(@RequestBody Student student,
@PathVariable("id") int studentId) {

          return student;
     }

     /**
      * @param studentId
      * @return
      *
      *         Create Spring REST API that handles HTTP Delete Request -
Deleting
      *         the Existing Resource
      * @DeleteMapping annotation is used for mapping HTTP DELETE request onto
      *                specific handler method
      *                http://localhost:8080/students/3/delete
      *
      *
      */
     @DeleteMapping("students/{id}/delete")
     public String deleteStudent(@PathVariable("id") int studentId) {
          return "Student deleted Successfully !!";
     }

     //using Spring ResponseEntity to manipulate the HTTP Response.
     //ResponseEntity represents the whole HTTP response: status code


}
```

```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>3.3.0-M1</version>
          <relativePath/> <!-- lookup parent from repository -->
     </parent>
     <groupId>com.srikanth</groupId>
     <artifactId>springboot-rest-api</artifactId>
     <version>0.0.1-SNAPSHOT</version>
     <name>springboot-rest-api</name>
     <description>Demo project for Spring Boot REST API </description>
     <properties>
          <java.version>21</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-test</artifactId>
               <scope>test</scope>
          </dependency>
     </dependencies>
     <build>
          <plugins>
```

```
                <plugin>
                     <groupId>org.springframework.boot</groupId>
                     <artifactId>spring-boot-maven-plugin</artifactId>
                </plugin>
           </plugins>
     </build>
     <repositories>
           <repository>
                <id>spring-milestones</id>
                <name>Spring Milestones</name>
                <url>https://repo.spring.io/milestone</url>
                <snapshots>
                     <enabled>false</enabled>
                </snapshots>
           </repository>
     </repositories>
     <pluginRepositories>
           <pluginRepository>
                <id>spring-milestones</id>
                <name>Spring Milestones</name>
                <url>https://repo.spring.io/milestone</url>
                <snapshots>
                     <enabled>false</enabled>
                </snapshots>
           </pluginRepository>
     </pluginRepositories>
</project>
```

--- End of Application 05---------

Micro Services

What are Microservices?

- Microservices are a software architectural style in which a large application is built as a collection of small, independent services that communicate with each other over a network.

- Each service is a self-contained unit of functionality that can be developed, tested, and deployed independently of the other services.

- This allows for **more flexibility and scalability** than a monolithic architecture, where all the functionality is contained in a single, large codebase.

- Microservices can be written in different programming languages and use different technologies, as long as they can communicate with each other through a common API.

- They are designed to be loosely coupled, meaning that changes to one service should not affect the other services. This makes it easier **to update, maintain, and scale the application.**

- Microservices architecture is best suited for large and complex applications that need to handle a high volume of traffic and need to be scaled horizontally.

Challenges of Monolith Architecture

1. Application is too large and complex (Large code base)
2. Parts are more tightly coupled each other
3. You can only scale the entire app, instead of a specific service or feature (higher infrastructure costs)
4. Difficult if services need different **dependency versions**
5. **Release process takes longer** - On every change, entire application needs to be tested and entire application needs to be built and deployed
6. Bug in one module can potentially bring down the entire application

Key Components of a Microservices Architecture

Key components of a microservices architecture include:

1. **Core Services:** Each service is a self-contained unit of functionality that can be **developed**, **tested**, and **deployed** independently of the other services.
2. **Service registry**: A service registry is a database of all the services in the system, along with their locations and capabilities.
3. It allows services to discover and communicate with each other.
4. **API Gateway**: An API gateway is a single entry point for all incoming requests to the microservices.

   It acts as a reverse proxy, routing requests to the appropriate service and handling tasks such as authentication and rate limiting.

5. **Message bus:** A message bus is a messaging system that allows services to communicate asynchronously with each other.

   This can be done through protocols like HTTP, RabbitMQ, or Kafka.

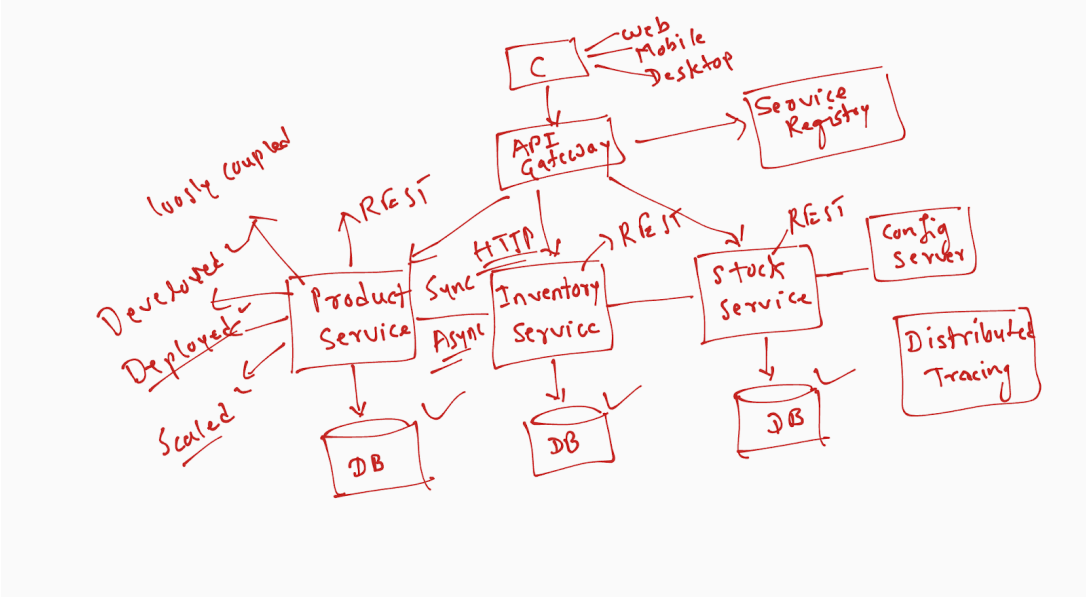
6. **Monitoring and logging**: Monitoring and logging are necessary to track the health of the services and troubleshoot problems.

7. **Service discovery and load balancing**: This component is responsible for discovering service instances and directing traffic to the appropriate service instances based on load and availability.

8. **Continuous integration and continuous deployment (CI/CD)**: To make the development and deployment process of microservices as smooth as possible, it is recommended to use a tool such as Jenkins, TravisCI, or CircleCI to automate the process of building, testing, and deploying microservices.

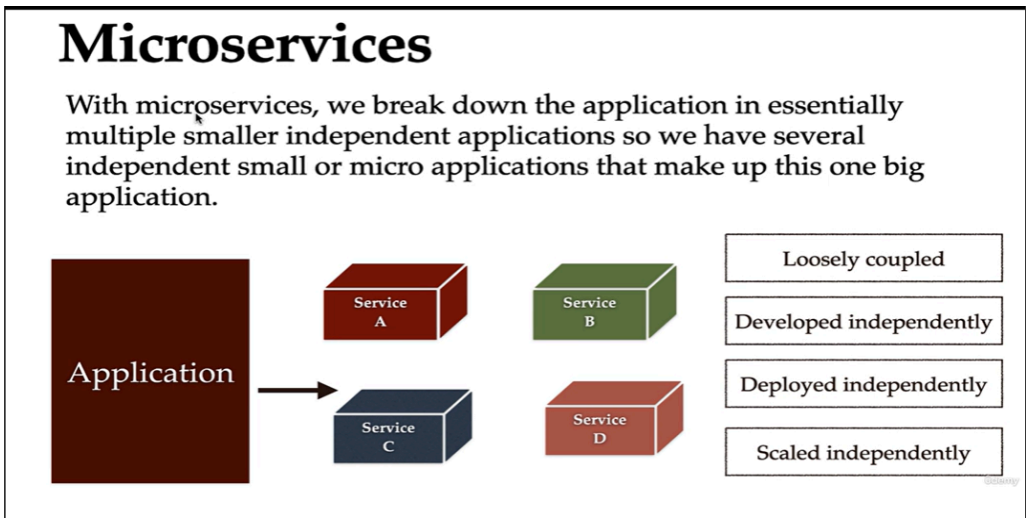
What are Microservices or Microservice Architecture?

- Well, a microservice architecture enables large teams to build scalable applications that are composed of many loosely coupled services.
- Here is what a typical microservice architecture looks like.



- For example, consider this microservice architecture for a simple shopping cart application.
- It has different services like product service, inventory service, and stock service, and these are the independent and loosely coupled services in the microservices projects.
- Each microservice has its own database.
- For example, product service has its own database, inventory service has its own database, and stock service has its own database.
- In the microservices project, all the microservices are loosely coupled.
- So loosely coupled, meaning all the services in a microservices project are independent of each other and each microservice should be **developed** independently and each microservice should be **deployed** independently and each microservice should be **scaled** independently.

So basically Microservice following characteristics:



- Each microservice can have its own database.
- Each microservice should be developed independently
- Each microservice should be deployed independently
- Each microservice should be scaled independently

In microservices projects, the services can communicate with each other.

For example,

- ❖ product service can communicate with inventory service and
- ❖ inventory service can communicate with stock service.
- ❖ Microservice can communicate with multiple services as well.

Well, there are two types of communication styles.

One is **synchronous** and another is **asynchronous**.

In the case of **synchronous**,

- ❖ we can use the HTTP protocol to make an HTTP request from one microservice to the microservice.
- ❖ RestTemplate, Open Feign Client

And in the case of **asynchronous** communication,

- We have to use a **message broker** for asynchronous communication between multiple microservices.
- For example, we can use a **RabbitMQ** or **Apache Kafka** as a message broker in order to make an asynchronous communication between multiple microservices and each microservice in a microservices project can expose REST API's.

How to Implement Microservices in Java?

Spring Boot and Spring Cloud frameworks are the de-facto standards for building Microservices in Java.

Spring Boot REST API Basics - Important Annotations

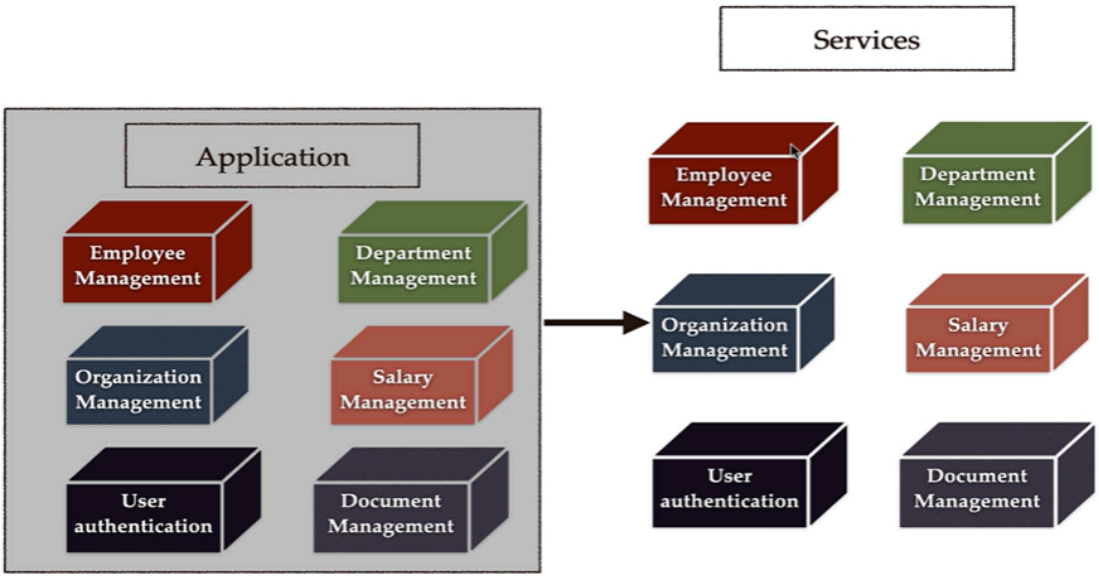
- Create Simple Spring Boot REST API - @RestController @GetMapping
- Spring Boot Rest API That returns Java Bean as JSON
- Create Spring Boot REST API that returns List as JSON
- Spring Boot REST API with Path Variable - @PathVariable
- Spring Boot REST API with Request Param - @RequestParam
- Spring Boot POST REST API - @PostMapping @RequestBody
- Spring Boot PUT REST API - @PutMapping @RequestBody
- Spring Boot Delete REST API - @DeleteMapping
- Using Spring ResponseEntity to Manipulate the HTTP Response.
- Define Base URL for REST API’s in Spring MVC Controller - @RequestMapping

Spring Boot Rest API endpoints !!

- HelloWorld ResAPI

- ❖ Spring Boot API Returns Java Bean that returns JSON to the client.
- ❖ Create Spring Boot REST API returns List(JSON Format)
- ❖ Create Spring Boot Rest API with PathVariable @PathVariable.

Microservices

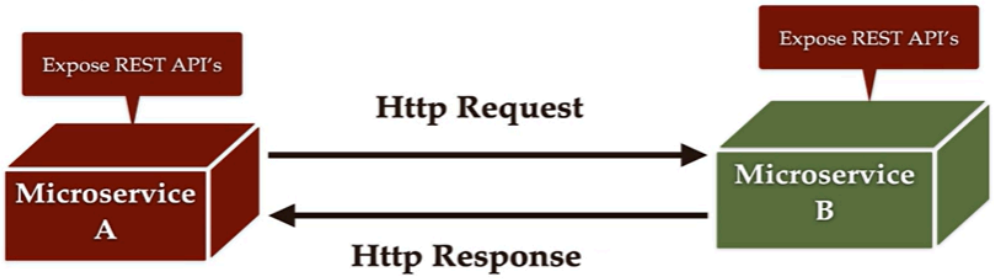


Microservices Architecture

1. The best practice is to break down the application into components or into microservices based on the **business functionalities** and not technical functionalities.
2. Separation of concerns: 1 Service for 1 specific job (Single Responsibility principle)
3. A very important characteristic of each microservice is that they should be self-contained and independent from each other this means each service must be able to be developed, deployed and scaled separately without any tight dependencies on any other services even though they are part of the same application.
4. Release process don't take longer time
5. Can independently scale up highly used services
6. Technology updates/rewrites become simpler

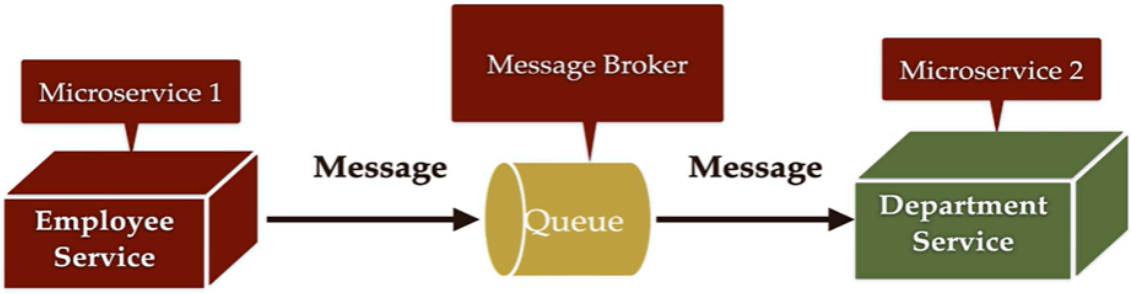
Synchronous Communication

1. The client sends a request and waits for a response from the service.
2. The important point here is that the protocol (HTTP/HTTPS) is synchronous and the client code can only continue its task when it receives the HTTP server response.
3. RestTemplate, WebClient and Spring Cloud Open Feign library



Asynchronous Communication

1. The client sends a request and does not wait for a response from the service.
2. The client will continue executing it's task - It don't wait for the response from the service.
3. RabbitMQ or Apache Kafka



Java Spring Boot Microservices Example Communication

Developing Employee-Service Step by Step

Step 1: Create a New Spring Boot Project in Spring Initializr

- Project: Maven
- Language: Java
- Packaging: Jar
- Java: 21
- Spring Boot DevTools
- Spring Data JPA
- Oracle Driver
- Spring Web

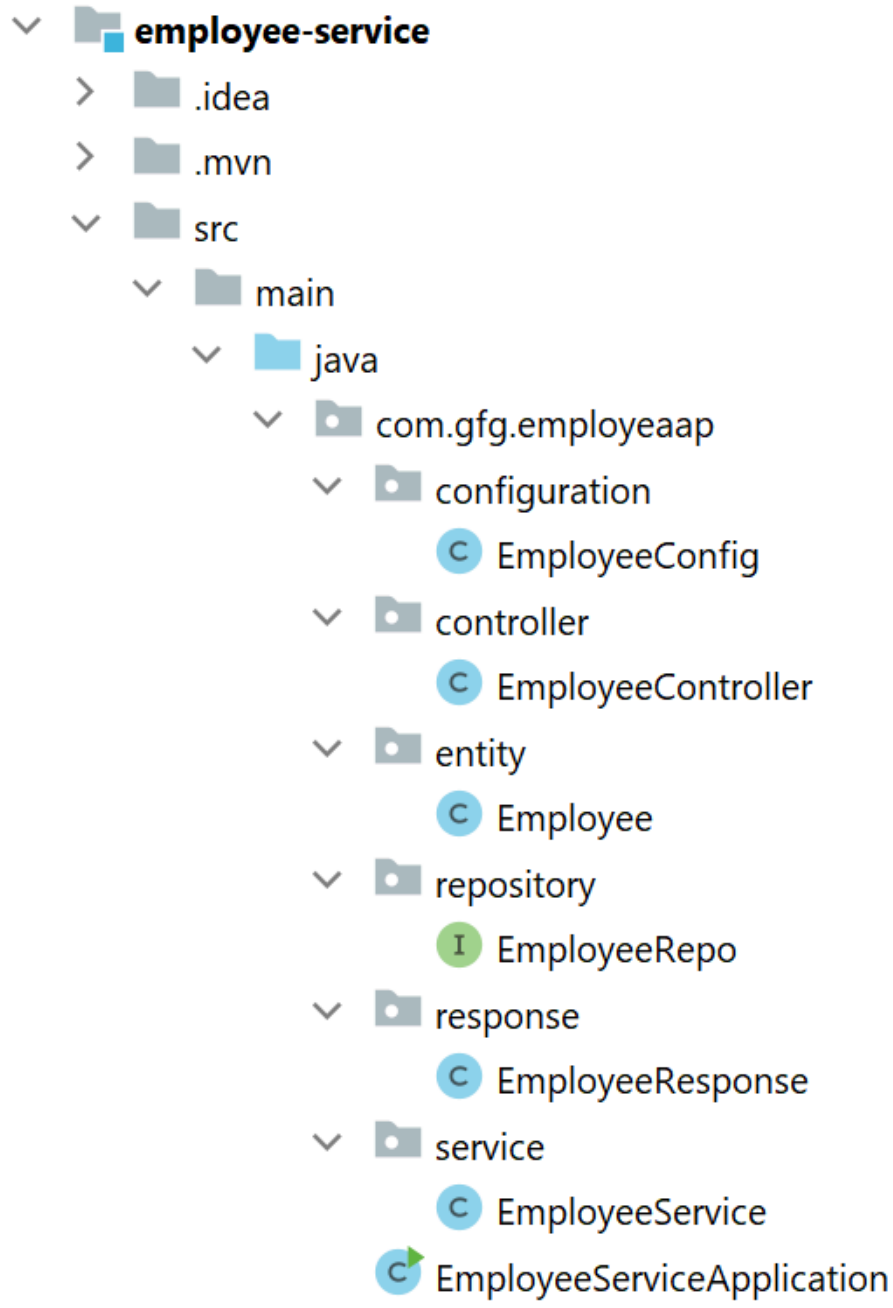
Step 2: Create Schema in Oracle SQL Developer and Put Some Sample Data.

Go to your Oracle Workbench and create a schema name **oracleDb1** and inside that create a table called **employee** and put some sample data.

Here we have created 4 columns and put some sample data.

1) Id 2)name 3) email 4) age

Now we are going to fetch Employee Data from the Employee Table in our Spring Boot project.
To do it refer to the following steps.
Before moving to Eclipse IDE let's have a look at the complete project structure for our Microservices.


employee-service
.idea
.mvn
src
main
java
com.gfg.employeeaap
configuration
EmployeeConfig
controller
EmployeeController
entity
Employee
repository
EmployeeRepo
response
EmployeeResponse
service
EmployeeService
EmployeeServiceApplication

Step 3: Make Changes in Your application.properties File

```
#server configuration
spring.application.name=employeeservice
server.port = 8181
#create and drop tables
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.datasource.url=jdbc:mysql://localhost:3306/sbdata
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
```

Step 4: Create Your Entity/Model Class

Go to the **src > main > java > entity** and create a class Employee and put the below code. This is our model class.

```
import jakarta.persistence.*;

@Entity
@Table(name = "employee2")
public class Employee {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
    @Column(name = "id")
    private int id;

    @Column(name = "name")
    private String name;

    @Column(name = "email")
    private String email;

    @Column(name = "age")
    private String age;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
```

```
        this.email = email;
    }

    public String getAge() {
        return age;
    }

    public void setAge(String age) {
        this.age = age;
    }
}
```

Step 5: Create Your Repository Interface

Go to the **src > main > java > repository** and create an interface EmployeeRepo and put the below code. This is our repository where we write code for all the database-related stuff.

```
import org.springframework.data.jpa.repository.JpaRepository;

public interface EmployeeRepo extends JpaRepository<Employee, Integer> {

}
```

Step 6: Create an EmployeeResponse Class

Go to the **src > main > java > response** and create a class EmployeeResponse and put the below code.

```
public class EmployeeResponse {

    private int id;
    private String name;
    private String email;
    private String age;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
```

```
        this.email = email;
    }

    public String getAge() {
        return age;
    }

    public void setAge(String age) {
        this.age = age;
    }
}
```

Step 7: Create Your Service Class

Go to the **src > main > java > service** and create a class EmployeeService and put the below code. This is our service class where we write our business logic.

```
import com.gfg.employeeaap.entity.Employee;
import com.gfg.employeeaap.repository.EmployeeRepo;
import com.gfg.employeeaap.response.EmployeeResponse;
import org.modelmapper.ModelMapper;
import org.springframework.beans.factory.annotation.Autowired;

import java.util.Optional;

public class EmployeeService {

    @Autowired
    private EmployeeRepo employeeRepo;
```

```
    @Autowired
    private ModelMapper mapper;

    public EmployeeResponse getEmployeeById(int id) {
        Optional<Employee> employee = employeeRepo.findById(id);
        EmployeeResponse employeeResponse = mapper.map(employee,
EmployeeResponse.class);
        return employeeResponse;
    }

}
```

Step 8: Create an Employee Controller

Go to the **src > main > java > controller** and create a class EmployeeController and put the below code. Here we are going to create an endpoint "***/employees/{id}***" to find an employee using id.

```
import com.gfg.employeeaap.response.EmployeeResponse;
import com.gfg.employeeaap.service.EmployeeService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class EmployeeController {

    @Autowired
```

```
    private EmployeeService employeeService;

    @GetMapping("/employees/{id}")
    private ResponseEntity<EmployeeResponse> getEmployeeDetails(
@PathVariable("id") int id)
{
        EmployeeResponse employee = employeeService.getEmployeeById(id);
        return ResponseEntity.status(HttpStatus.OK).body(employee);
    }

}
```

Step 9: Create a Configuration Class

Go to the **src > main > java > configuration** and create a class EmployeeConfig and put the below code.

```
import com.gfg.employeeaap.service.EmployeeService;
import org.modelmapper.ModelMapper;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class EmployeeConfig {

    @Bean
    public EmployeeService employeeBean() {
        return new EmployeeService();
    }

    @Bean
    public ModelMapper modelMapperBean() {
```

```
        return new ModelMapper();
    }

}
```

Before running the Microservice below is the complete **pom.xml** file. Please cross-verify if you have missed some dependencies

```
<?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>3.0.2</version>
          <relativePath/> <!-- lookup parent from repository -->
     </parent>
     <groupId>com.gfg.employeaap</groupId>
     <artifactId>employee-service</artifactId>
     <version>0.0.1-SNAPSHOT</version>
     <name>employee-service</name>
     <description>Employee Service</description>
     <properties>
          <java.version>17</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>org.springframework.boot</groupId>
            <artifactId>spring-boot-devtools</artifactId>
            <scope>runtime</scope>
            <optional>true</optional>
        </dependency>
        <dependency>
            <groupId>com.oracle.database.jdbc</groupId>
            <artifactId>ojdbc11</artifactId>
            <scope>runtime</scope>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.modelmapper</groupId>
            <artifactId>modelmapper</artifactId>
            <version>3.1.1</version>
        </dependency>
    </dependencies>

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

</project>
```

Step 10: Run Your Employee Microservice

To run your Employee Microservice **src > main > java > EmployeeServiceApplication** and click on the Run button. If everything goes well then you may see **Started EmployeeServiceApplication.**

Step 11: Test Your Endpoint in Postman

Now open Postman and hit the following URL:

GET: http://localhost:8181/employees/1

And you can see the following response:


```
{
   "id": 1,
   "name": "Srikanth",
   "email": "Srkntjva@gmail.com",
   "age": "30"
}
```


This is how we have built our Employee Microservice with the help of Java and Spring Boot.

Developing Address-Service Step by Step

Step 1: Create a New Spring Boot Project in Spring Initializr

- **Project: Maven**
- **Language: Java**
- **Packaging: Jar**

- **Java: 17**
- **Spring Boot DevTools**
- **Spring Data JPA**
- **MySQL Driver**
- **Spring Web**

Note: We have used the Oracle database in this project.

Step 2: Create Schema in Oracle SQL DEVELOPER Workbench and Put Some Sample Data

Go to your **Oracle SQL DEVELOPER Workbench** and create a schema name **Oracledb1** and inside that create a table called **address** and put some sample data.

Address Table:

Here we have created 4 columns and put some sample data.

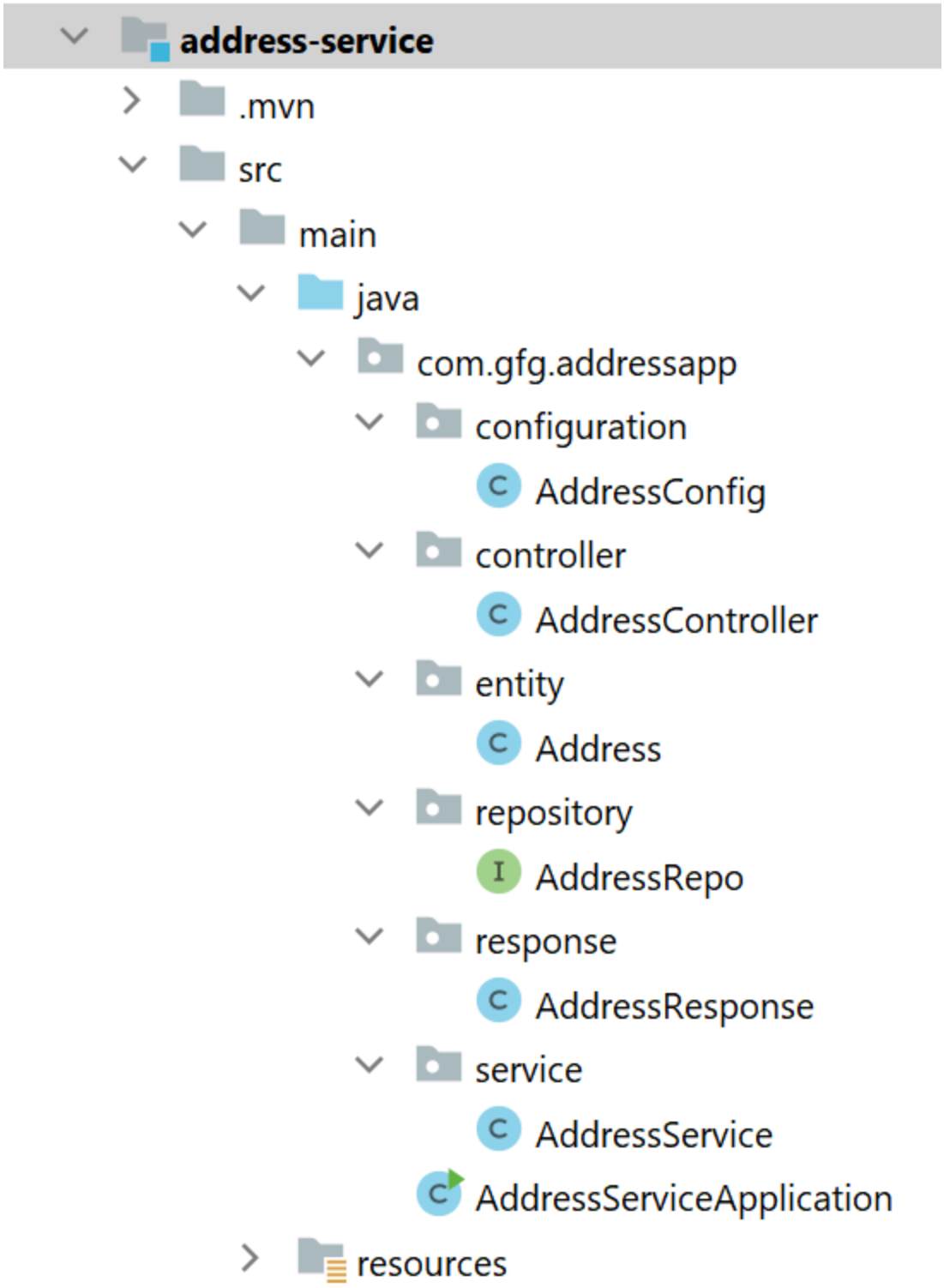
1. id
2. city
3. state
4. employee_id

Note : Note: In the Address table, employee_id is a foreign key so create it accordingly. We are going to perform a SQL JOIN Operation in our native SQL query. So create tables carefully.

```
CREATE Table address(id number, city varchar2(50),
state varchar2(50), empid number,
```

```
foreign key(id) references employee4(id));

insert into address values(1,'Hyderabad','TG',1);
insert into address values(3,'Guntur','AP',3);

DESC employee4;

ALTER TABLE employee4 ADD Primary key(id);

select * from address;
```

Before moving to Eclipse IDE let's have a look at the complete project structure for our Microservices.



Step 3: Make Changes in Your application.properties File

```
#server configuration
server.port=9191
server.error.whitelabel.enabled=false
#create and drop tables
spring.jpa.hibernate.ddl.auto=update
spring.jpa.show-sql=true

spring.application.name=address-service

server.servlet.context-path=/address-service


#Oracle Settings
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:xe
spring.datasource.username=SYSTEM
spring.datasource.password=Oracle3
spring.datasource.driver-class-name=oracle.jdbc.driver.OracleDriver
```

Step 4: Create Your Entity/Model Class

Go to the src > main > java > entity and create a class Address and put the below code. This is our model class.

```
import jakarta.persistence.*;

@Entity
@Table(name = "address")
public class Address {

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

```
    @Column(name = "city")
    private String city;

    @Column(name = "state")
    private String state;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getCity() {
        return city;
    }

    public void setCity(String city) {
        this.city = city;
    }

    public String getState() {
        return state;
    }

    public void setState(String state) {
        this.state = state;
    }
}
```

Step 5: Create Your Repository Interface

Go to the src > main > java > repository and create an interface AddressRepo and put the below code.

This is our repository where we write code for all the database-related stuff.

```
import com.gfg.addressapp.entity.Address;
import java.util.Optional;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;

@Repository
public interface AddressRepo extends JpaRepository<Address, Integer> {

    @Query(
        nativeQuery = true,
        value
        = "SELECT ea.id, ea.city, ea.state FROM address ea join employee e on
e.id = ea.employee_id where ea.employee_id=:employeeId")
       Optional<Address> findAddressByEmployeeId(@Param("employeeId") int
employeeId);
}
```

Step 6: Create an AddressResponse Class

Go to the src > main > java > response and create a class AddressResponse and put the below code.

```java
public class AddressResponse {

    private int id;
    private String city;
    private String state;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getCity() {
        return city;
    }

    public void setCity(String city) {
        this.city = city;
    }

    public String getState() {
        return state;
    }

    public void setState(String state) {
        this.state = state;
    }
}
```

Step 7: Create Your Service Class

Go to the src > main > java > service and create a class AddressService and put the below code. This is our service class where we write our business logic.

```
import com.gfg.addressapp.entity.Address;
import com.gfg.addressapp.repository.AddressRepo;
import com.gfg.addressapp.response.AddressResponse;
import org.modelmapper.ModelMapper;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.Optional;

@Service
public class AddressService {

    @Autowired
    private AddressRepo addressRepo;

    @Autowired
    private ModelMapper mapper;

    public AddressResponse findAddressByEmployeeId(int employeeId) {
        Optional<Address> addressByEmployeeId =
addressRepo.findAddressByEmployeeId(employeeId);
        AddressResponse addressResponse = mapper.map(addressByEmployeeId,
AddressResponse.class);
        return addressResponse;
    }

}
```

Step 8: Create an Address Controller

Go to the src > main > java > controller and create a class AddressController and put the below code. Here we are going to create an endpoint "***/address/{employeeId}***" to find the address using employee_id. That's why we have created a foreign key in the Address table and we have performed the SQL join operation in the native query to get our desired result.

```
import com.gfg.addressapp.response.AddressResponse;
import com.gfg.addressapp.service.AddressService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class AddressController {

    @Autowired
    private AddressService addressService;

    @GetMapping("/address/{employeeId}")
    public ResponseEntity<AddressResponse>
getAddressByEmployeeId(@PathVariable("employeeId") int employeeId) {
        AddressResponse addressResponse =
addressService.findAddressByEmployeeId(employeeId);
        return ResponseEntity.status(HttpStatus.OK).body(addressResponse);
    }

}
```

Step 9: Create a Configuration Class

Go to the src > main > java > configuration and create a class AddressConfig and put the below code.

```
import com.gfg.addressapp.service.AddressService;
import org.modelmapper.ModelMapper;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class AddressConfig {

    @Bean
    public ModelMapper modelMapperBean() {
        return new ModelMapper();
    }

}
```

Before running the Microservice below is the complete **pom.xml** file. Please cross-verify if you have missed some dependencies

```
<?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>3.0.2</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.gfg.addressapp</groupId>
    <artifactId>address-service</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>address-service</name>
    <description>Address Service</description>
    <properties>
        <java.version>17</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>org.springframework.boot</groupId>
            <artifactId>spring-boot-devtools</artifactId>
            <scope>runtime</scope>
            <optional>true</optional>
        </dependency>
        <dependency>
            <groupId>com.oracle.database.jdbc</groupId>
            <artifactId>ojdbc11</artifactId>
            <scope>runtime</scope>
        </dependency>

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

```
            <artifactId>modelmapper</artifactId>
            <version>3.1.1</version>
        </dependency>
    </dependencies>

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

</project>
```

Step 10: Run Your Address Microservice

To run your Address Microservice src > main > java > AddressServiceApplication and click on the Run button. If everything goes well then you may see **Started AddressServiceApplication.**

Step 11: Test Your Endpoint in Postman

Now open Postman and hit the following URL

GET: http://localhost:8182/address-service/address/2


```
{
   "id": 1,
   "city": "HYD",
   "state": "TELANGANA"
```

```
}
```

Microservices Communication using RestTemplate

Now let's communicate between two microservices using RestTemplate.

So we are going to get the address data by the employeeId of an employee and when we communicate we are going to get a response like the below. So let's implement it.


```
{
   "id": 1,
   "name": "SRIKANTH",
   "email": "Srikanth@gmail.com",
   "age": "30",
   "addressResponse": {
       "id": 1,
       "city": "HYD",
       "state": "Telangana"
   }
}
```

Here employee-service is going to consume data from the address-service.

So let's write the logic in the employee-service.

Step 1: Create an AddressResponse Class

Go to the employee-service > src > main > java > response and create a class AddressResponse and put the below code.

```
public class AddressResponse {

    private int id;
    private String city;
    private String state;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getCity() {
        return city;
    }

    public void setCity(String city) {
        this.city = city;
    }

    public String getState() {
```

```
        return state;
    }

    public void setState(String state) {
        this.state = state;
    }
}
```

Step 2: Modify EmployeeResponse Class

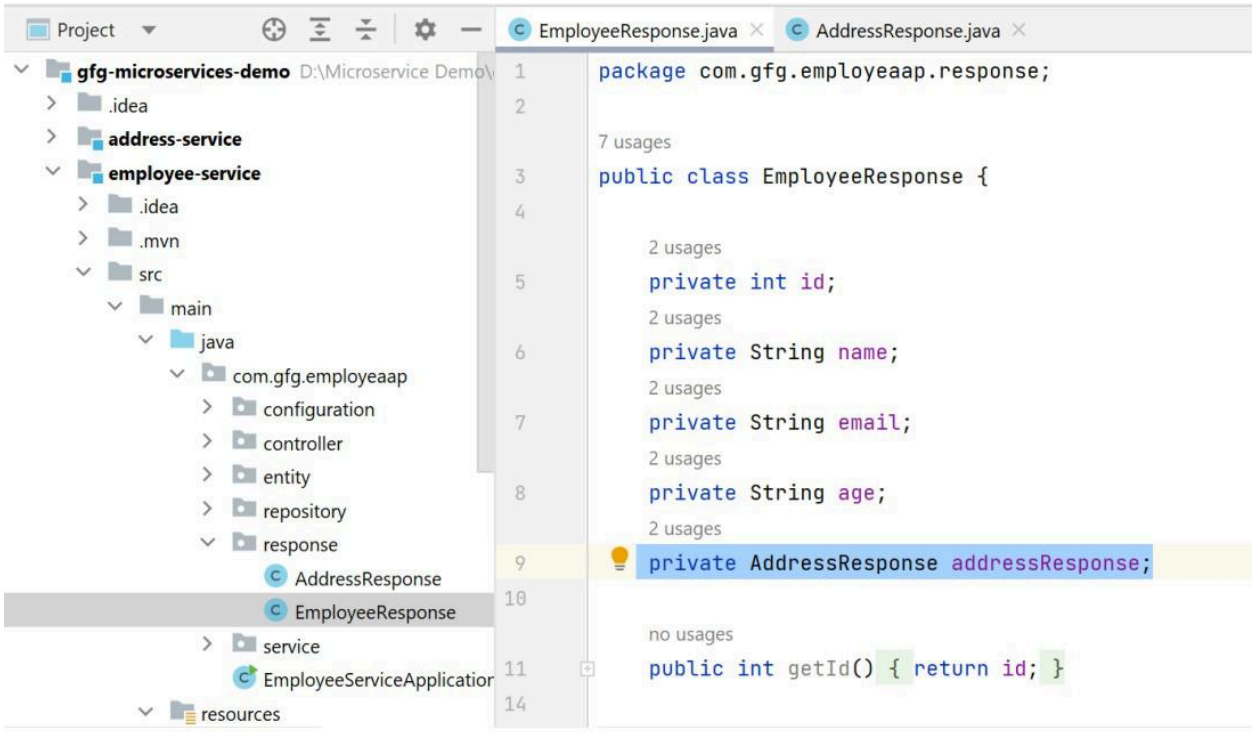
Also, go to the employee-service > src > main > java > response > EmployeeResponse and modify the EmployeeResponse class as below.

```
public class EmployeeResponse {

    private int id;
    private String name;
    private String email;
    private String age;

    // Add AddressResponse Here
    private AddressResponse addressResponse;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
```

```
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getAge() {
        return age;
    }

    public void setAge(String age) {
        this.age = age;
    }

    public AddressResponse getAddressResponse() {
        return addressResponse;
    }

    public void setAddressResponse(AddressResponse addressResponse) {
        this.addressResponse = addressResponse;
    }
```

```
}
```

Please refer to the below image.



Step 3: Modify EmployeeService Class

Now go to the employee-service > src > main > java > service > EmployeeService and modify the EmployeeService class as below. Add the below code inside this class.

```
@Autowired
private RestTemplate restTemplate;
AddressResponse addressResponse =
```

```
restTemplate.getForObject("http://localhost:8182/address-service/address/{id}"
, AddressResponse.class, id);

employeeResponse.setAddressResponse(addressResponse);
```

Below is the complete code for EmployeeService Class.

```
import com.gfg.employeeaap.entity.Employee;
import com.gfg.employeeaap.repository.EmployeeRepo;
import com.gfg.employeeaap.response.AddressResponse;
import com.gfg.employeeaap.response.EmployeeResponse;
import org.modelmapper.ModelMapper;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.client.RestTemplate;

import java.util.Optional;

@Service
public class EmployeeService {

    @Autowired
    private EmployeeRepo employeeRepo;

    @Autowired
    private ModelMapper mapper;

    @Autowired
    private RestTemplate restTemplate;

    public EmployeeResponse getEmployeeById(int id) {
```

```
        Optional<Employee> employee = employeeRepo.findById(id);
        EmployeeResponse employeeResponse = mapper.map(employee,
EmployeeResponse.class);

        AddressResponse addressResponse =
restTemplate.getForObject("http://localhost:8182/address-service/address/{id}", AddressResponse.class, id);
        employeeResponse.setAddressResponse(addressResponse);

        return employeeResponse;
    }

}
```

Step 4: Modify EmployeeConfig Class

```
@Bean
public RestTemplate restTemplateBean() {
       return new RestTemplate();
}
```

Below is the complete code for EmployeeConfig Class.

```
import com.gfg.employeeaap.service.EmployeeService;
import org.modelmapper.ModelMapper;
import org.springframework.context.annotation.Bean;
```

```java
import org.springframework.context.annotation.Configuration;
import org.springframework.web.client.RestTemplate;

@Configuration
public class EmployeeConfig {

    @Bean
    public ModelMapper modelMapperBean() {
        return new ModelMapper();
    }

    @Bean
    public RestTemplate restTemplateBean() {
        return new RestTemplate();
    }

}
```

Step 5: Run Your Both Address and Employee Microservices

Now run your both Address and Employee Microservices.

If everything goes well then you may see the **started EmployeeServiceApplication**.

Step 11: Test Your Endpoint in Postman

Now open Postman and hit the following URL

GET: http://localhost:8181/employee-service/employees/2

And you can see the following response

```
{
   "id": 2,
   "name": "Srikanth",
   "email": "Srkntjva@gmail.com",
   "age": "30",
   "addressResponse": {
       "id": 1,
       "city": "HYD",
       "state": "Telangana"
   }
}
```

Conclusion

So this is how the communication between two microservices happens using RestTemplate.

Note that as of now **RestTemplate** is deprecated but still many legacy projects are using RestTemplate so it's good to have knowledge of RestTemplate.

The replacement for RestTemplate is **Feign Client** and Spring 5 WebClient, but Feign Client is very popular nowadays.