

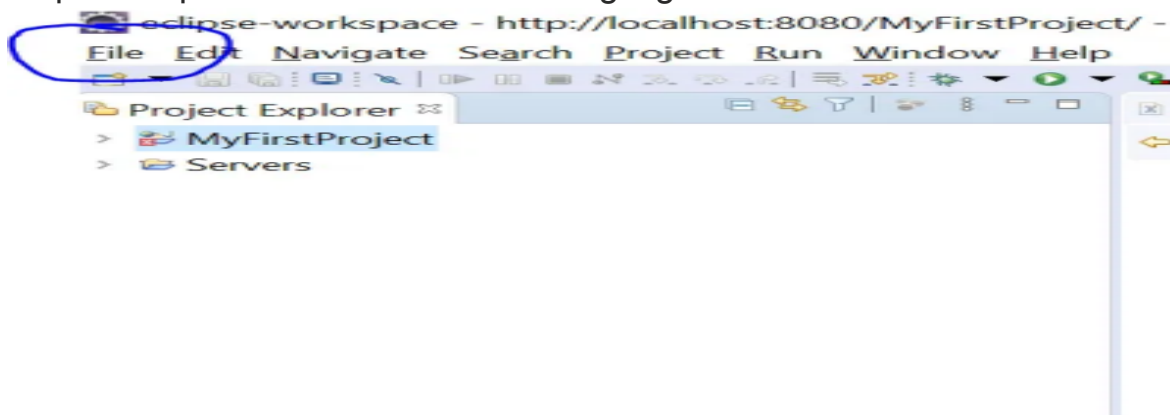
HelloWorldServlet, Apache Tomcat and eclipse setup complete guide :

Below are the steps to create a basic “Hello World” servlet project using eclipse.

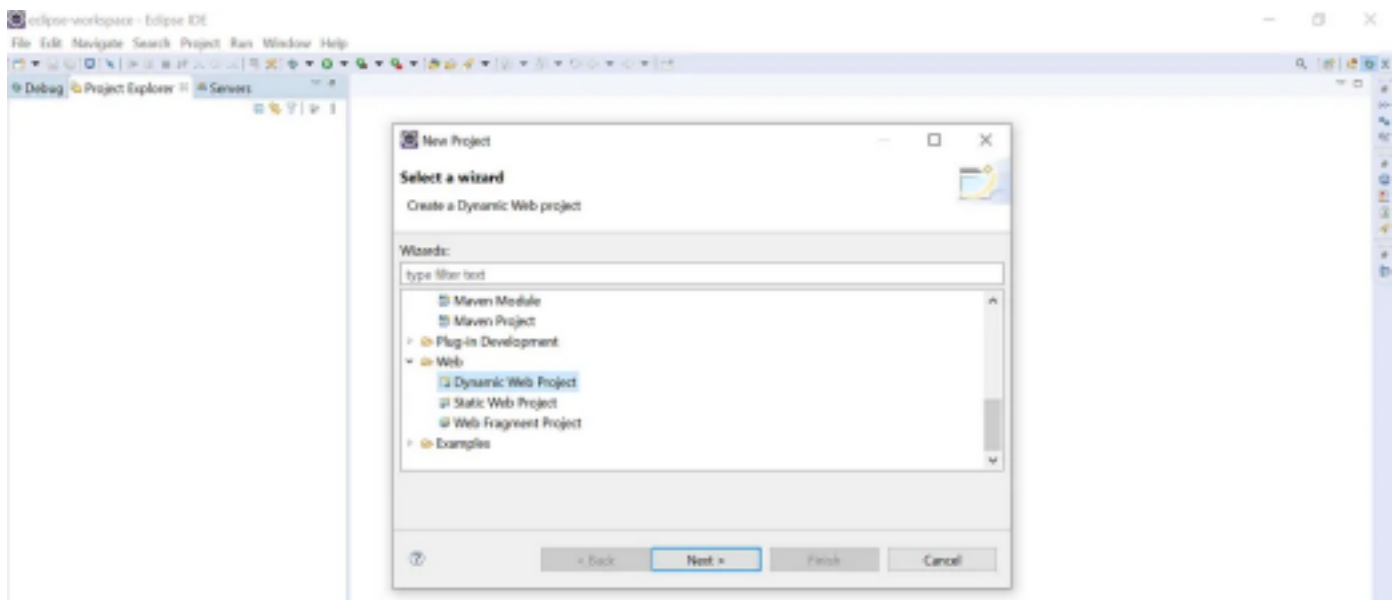
This will also help us to setup Apache tomcat with eclipse.

Same servlet project we can use to add more functionality. e.g. doPost, doDelete, cookie, session, etc.

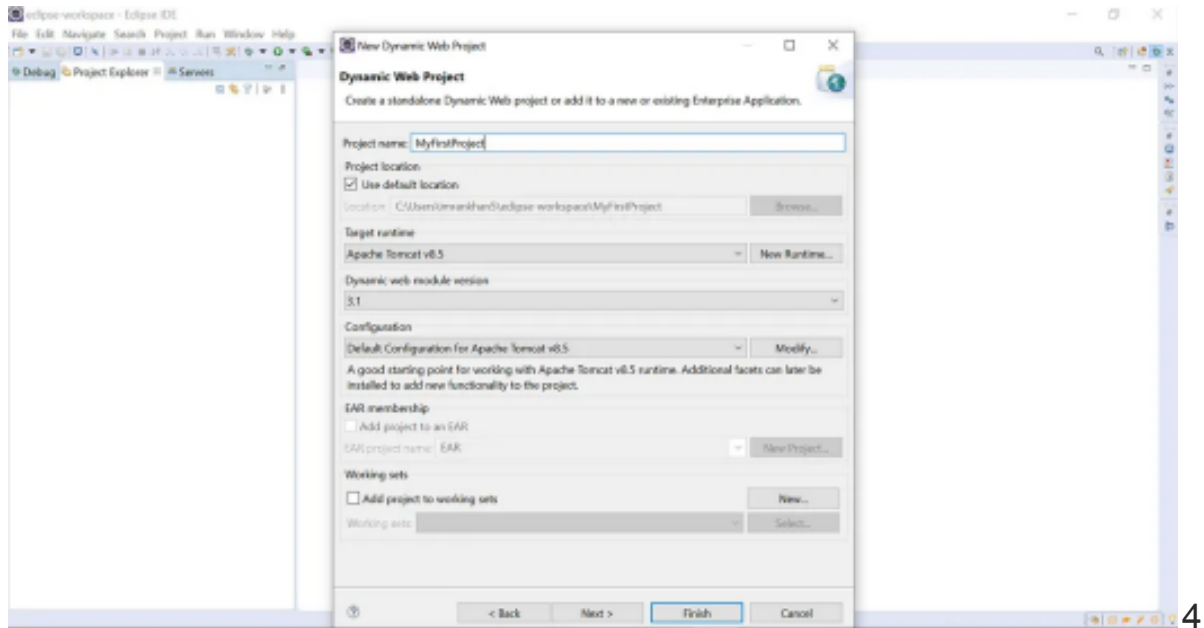
1. Open eclipse and Click on File as highlighted below. File → New → Other



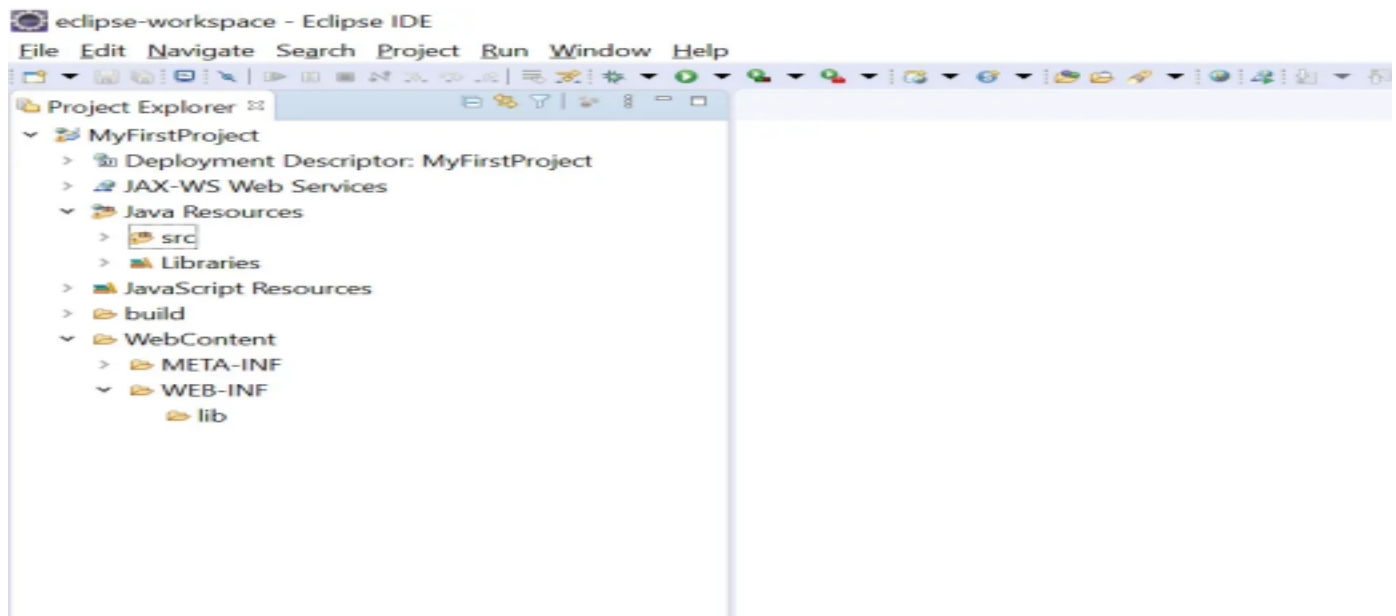
2. Select “Dynamic Web Project” and click on “next” button will help us to create project related to servlet.



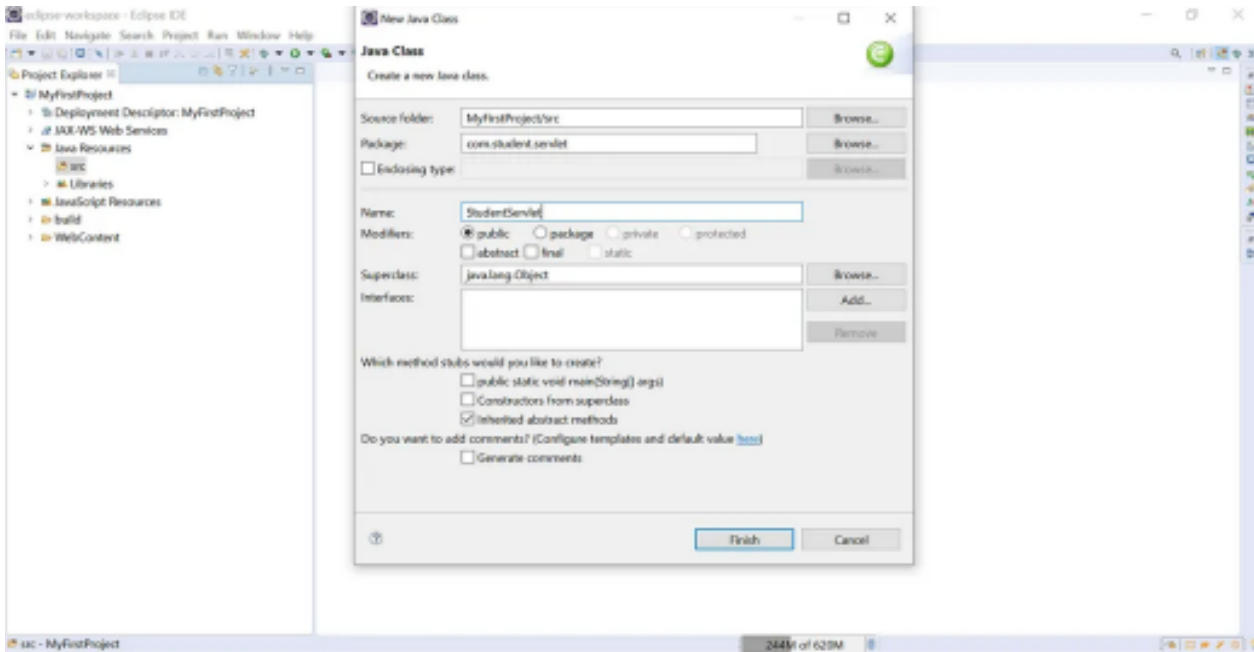
3. Provide the project name and click on finish will help us to create project. On current example we have given as “MyFirstProject”.



After clicking on finish will give below setup ready in eclipse.



4. Provide the Servlet class name and package name under which class will be lying. In current example we have given the package name as “com.student.servlet” and Servlet class name as “StudentServlet”.



5. Open Student.java created in the project and add below code inside it.

Creation of Servlet class requires extending HttpServlet class and overriding doGet() method:

```
package com.student.servlet;
```

```
import java.io.IOException;
```

```
import java.io.PrintWriter;
```

```
import javax.servlet.ServletException;
```

```
import javax.servlet.http.HttpServlet;
```

```
import javax.servlet.http.HttpServletRequest;
```

```
import javax.servlet.http.HttpServletResponse;
```

```
public class StudentServlet extends HttpServlet {
```

```
    private static final long serialVersionUID = 1L;
```

```
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
```

```
        response.setContentType("text/html"); // Set response content type
```

```
        PrintWriter out = response.getWriter();
```

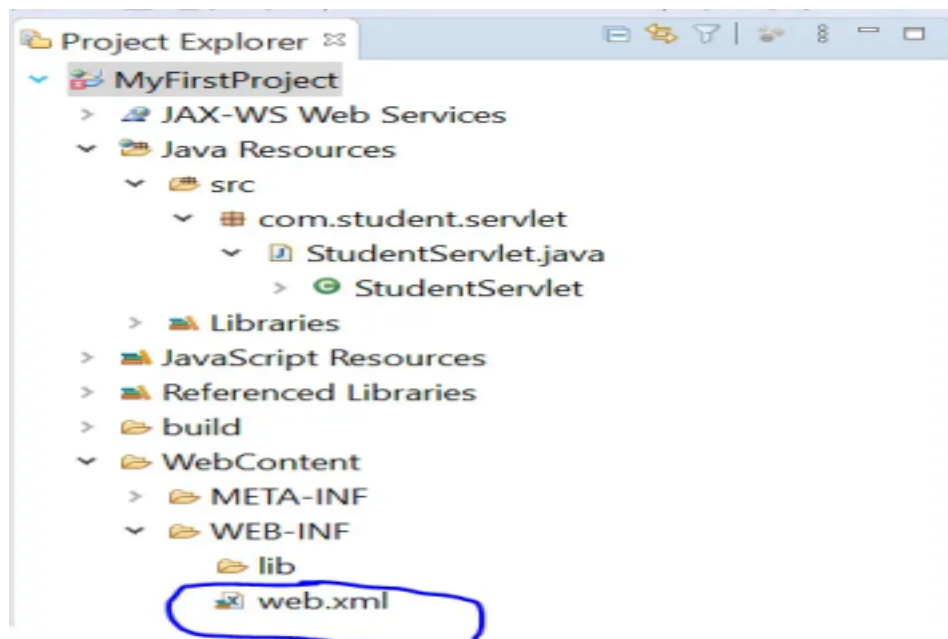
```
        out.println("<h1>Hello World</h1>");
```

```
    }
```

}

```
StudentServlet.java
1 package com.student.servlet;
2
3 import java.io.IOException;
4 import java.io.PrintWriter;
5
6 import javax.servlet.ServletException;
7 import javax.servlet.http.HttpServlet;
8 import javax.servlet.http.HttpServletRequest;
9 import javax.servlet.http.HttpServletResponse;
10
11 public class StudentServlet extends HttpServlet{
12
13     private static final long serialVersionUID = 1L;
14
15     public void doGet(HttpServletRequest request, HttpServletResponse response)
16         throws ServletException, IOException {
17
18         // Set response content type
19         response.setContentType("text/html");
20
21         // Actual logic goes here.
22         PrintWriter out = response.getWriter();
23         out.println("<h1>Hello World</h1>");
24     }
25 }
```

6. Add below web.xml file under WEB-INF folder



7. Add below content under web.xml file:

web.xml file is responsible for multiple things, it contains Servlet mapping which is responsible for calling a Servlet, welcome page as soon as servlet starts, load on startup call a servlet as soon as server starts, init params for passing a parameter to a

servlet, etc.

```
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
```

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
```

```
http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd" version="4.0"  
metadata-complete="true">
```

```
<servlet>
```

```
    <servlet-name>StudentServlet</servlet-name>
```

```
    <servlet-class>com.student.servlet.StudentServlet</servlet-class>
```

```
</servlet>
```

```
<servlet-mapping>
```

```
    <servlet-name>StudentServlet</servlet-name>
```

```
    <url-pattern>/student</url-pattern>
```

```
</servlet-mapping>
```

```
</web-app>
```

```

1 <web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
2   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3   xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
4                       http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
5   version="4.0"
6   metadata-complete="true">
7
8   <servlet>
9     <servlet-name>StudentServlet</servlet-name>
10    <servlet-class>com.student.servlet.StudentServlet</servlet-class>
11  </servlet>
12
13  <servlet-mapping>
14    <servlet-name>StudentServlet</servlet-name>
15    <url-pattern>/student</url-pattern>
16  </servlet-mapping>
17
18 </web-app>

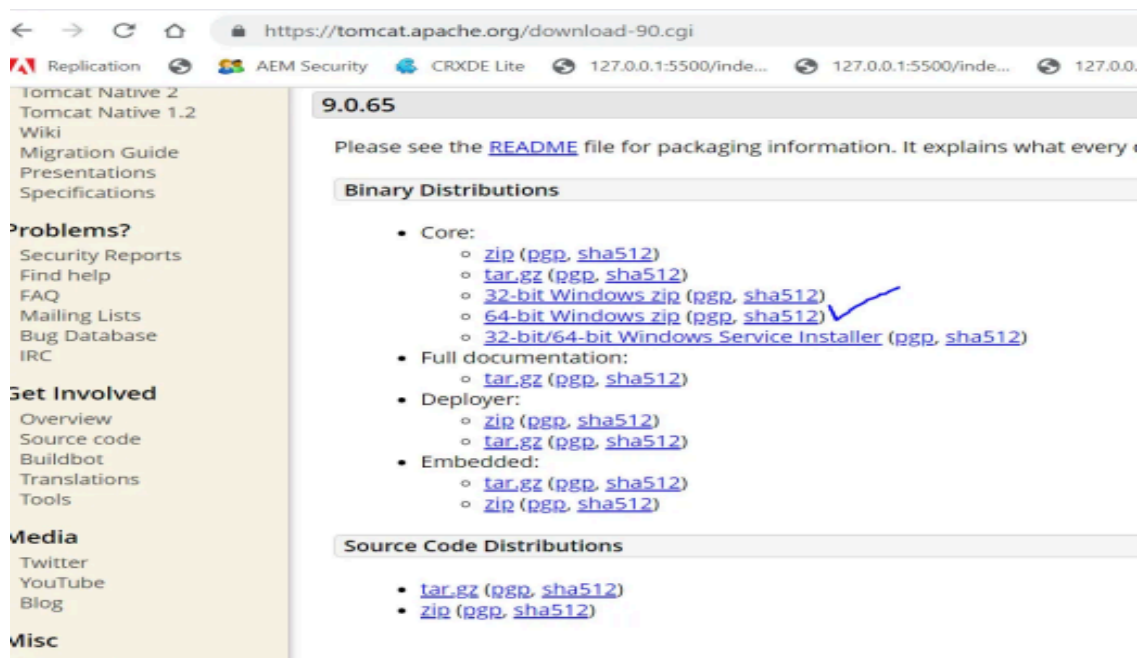
```

Apache Tomcat Setup:

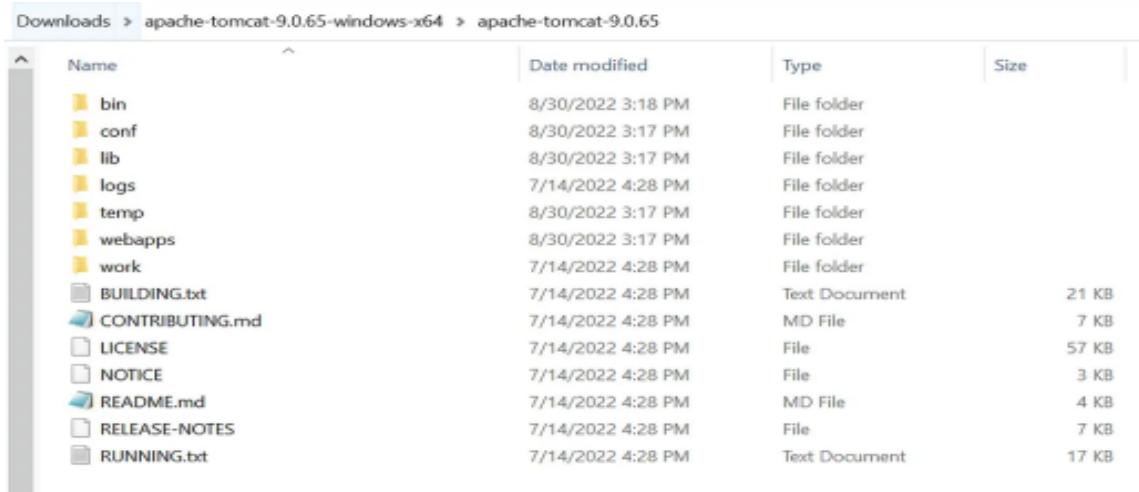
To run servlet dynamic web projects always require a web server. There are different web servers in the market such as Apache Tomcat, Jboss, etc.

For current implementation we will be using Apache Tomcat. Please follow below steps to setup apache Tomcat:

1. Download Apache tomcat from this [link](#) as show in below screen. We are setting up Apache Tomcat for windows 64 bit.

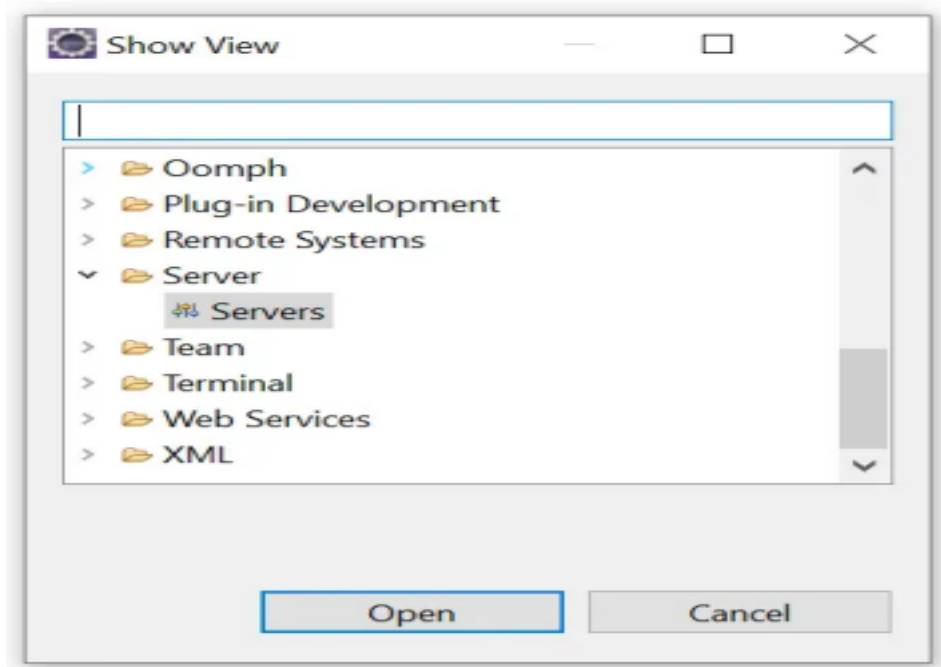


After downloading and unzipping the downloaded file in the same folder will give the below view.

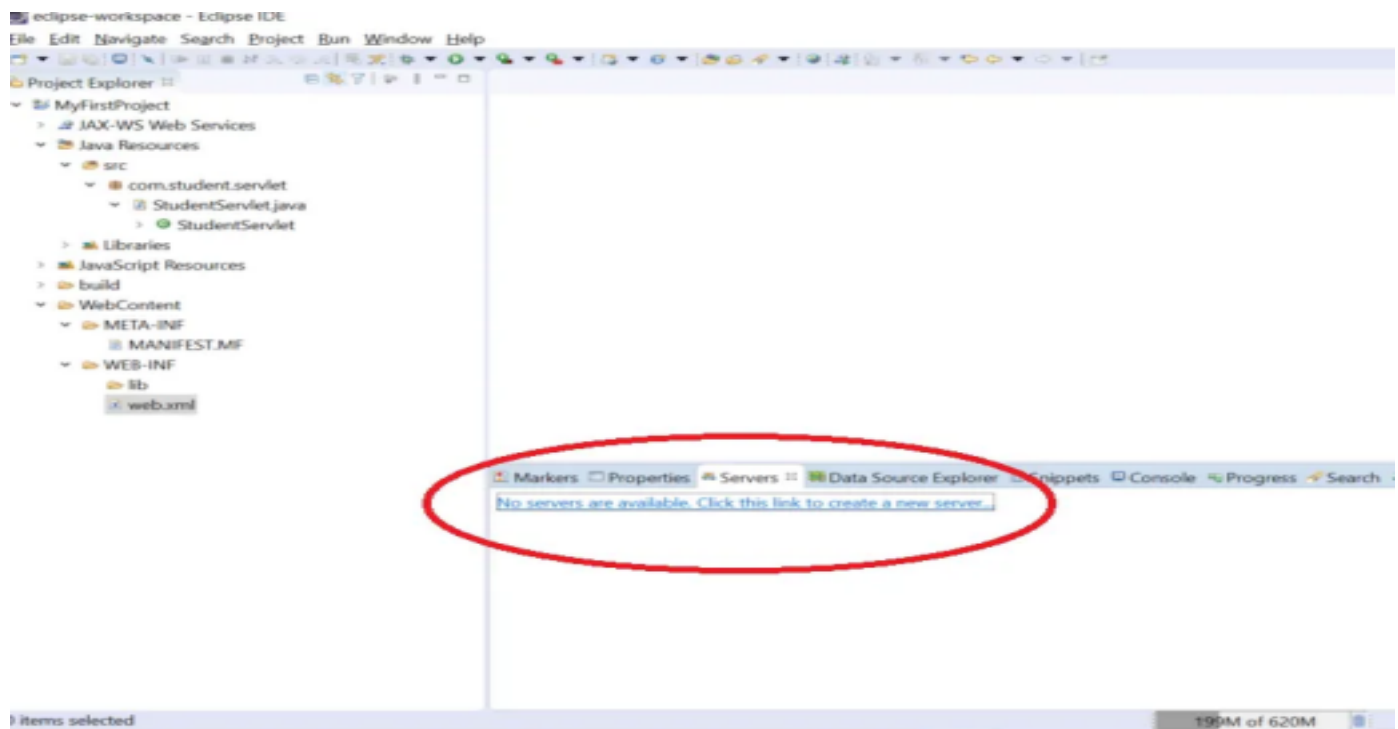


Name	Date modified	Type	Size
bin	8/30/2022 3:18 PM	File folder	
conf	8/30/2022 3:17 PM	File folder	
lib	8/30/2022 3:17 PM	File folder	
logs	7/14/2022 4:28 PM	File folder	
temp	8/30/2022 3:17 PM	File folder	
webapps	8/30/2022 3:17 PM	File folder	
work	7/14/2022 4:28 PM	File folder	
BUILDING.txt	7/14/2022 4:28 PM	Text Document	21 KB
CONTRIBUTING.md	7/14/2022 4:28 PM	MD File	7 KB
LICENSE	7/14/2022 4:28 PM	File	57 KB
NOTICE	7/14/2022 4:28 PM	File	3 KB
README.md	7/14/2022 4:28 PM	MD File	4 KB
RELEASE-NOTES	7/14/2022 4:28 PM	File	7 KB
RUNNING.txt	7/14/2022 4:28 PM	Text Document	17 KB

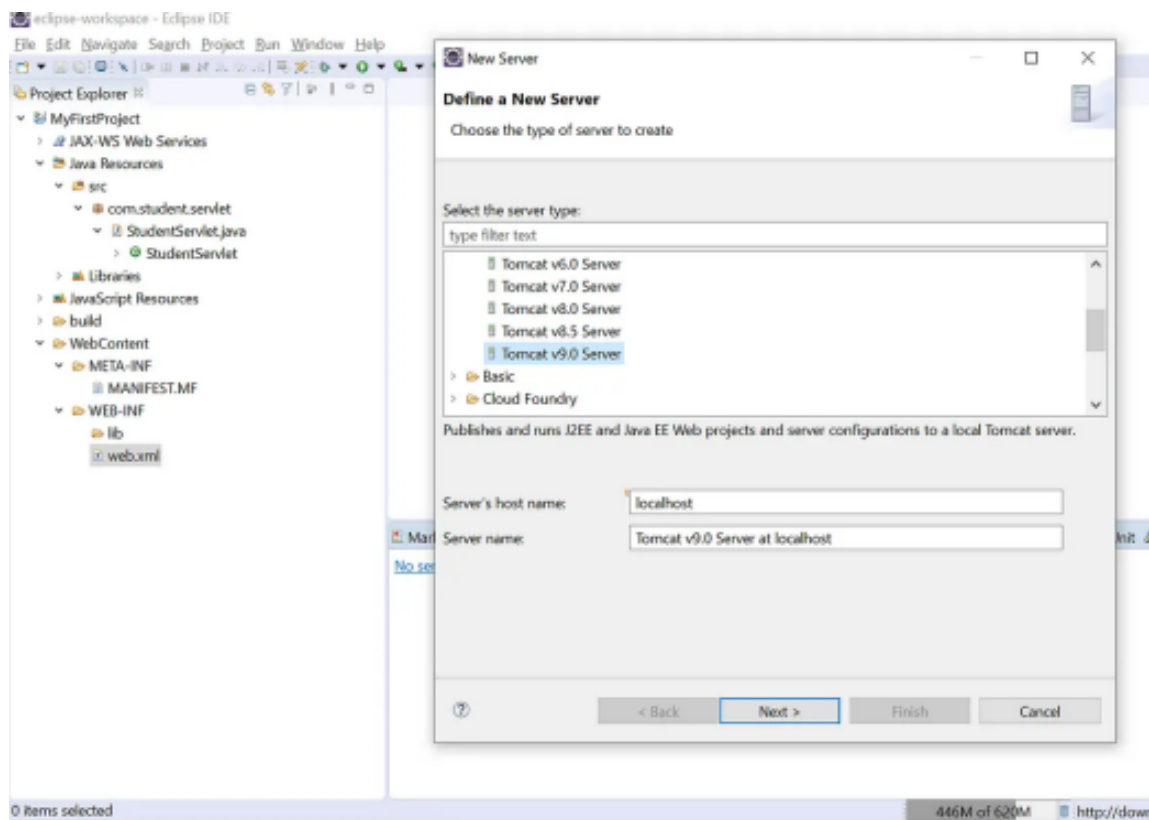
2. Go back to eclipse and click on top menu option “Window” and follow below steps: Window → Show View → Servers



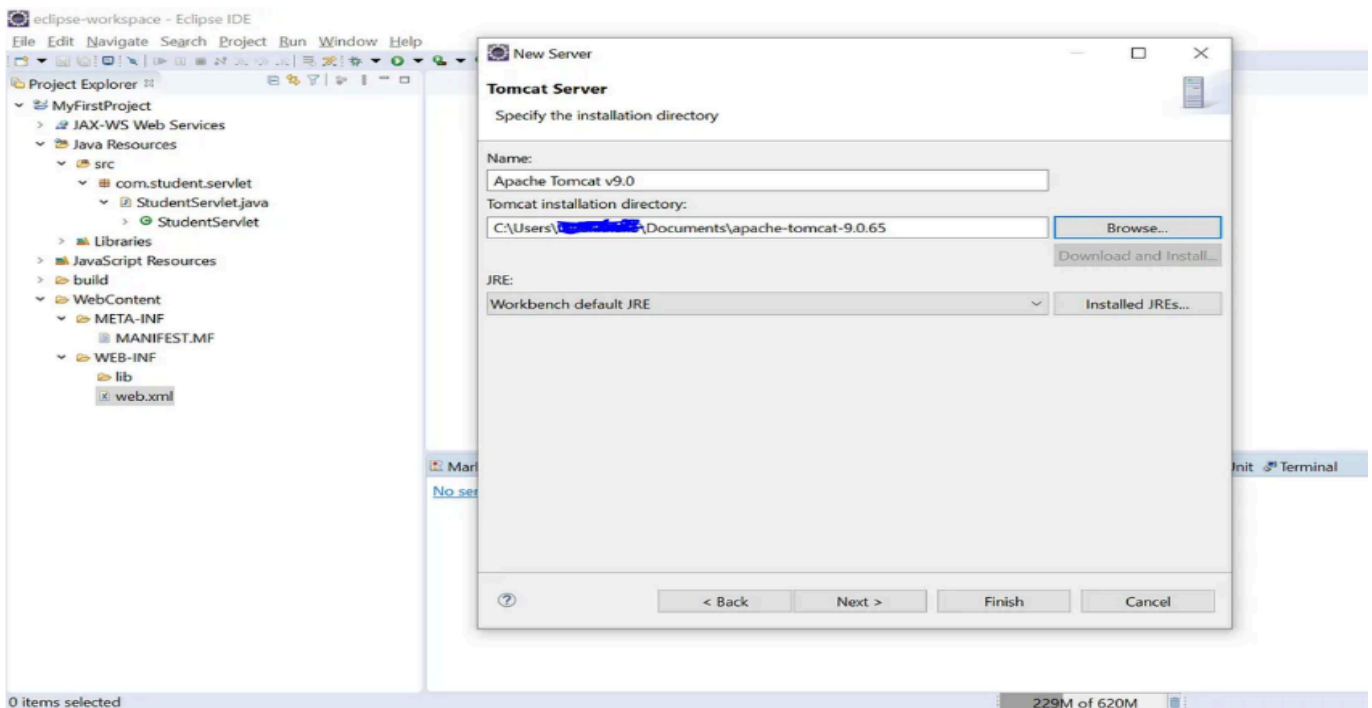
3. It will show the below screen with the selected tab as “Servers”. Select anchor link highlighted red in color.



4. Select Downloaded Tomcat Version. In our case it will be “Tomcat v9.0 Server” and click on “next” button.



5. Click on browse button and select unzip downloaded Apache Tomcat folder as mentioned in below screen. Click on Finish will complete the setup.



Now, its time to deploy our code base on Apache Tomcat as both our Servlet code base and Apache Tomcat is ready.

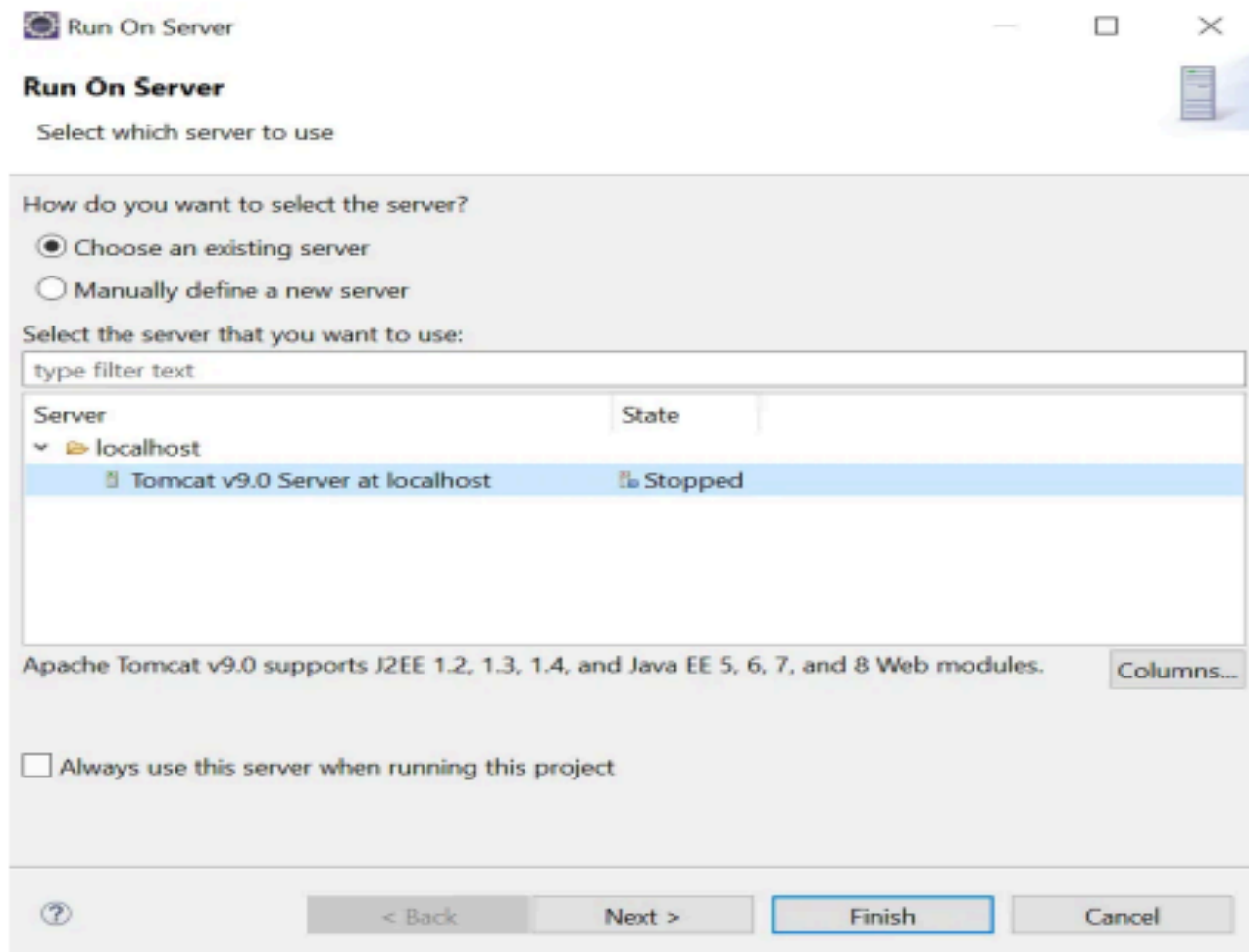
Deploy Servlet Project on Apache Tomcat

Do connect with me on LinkedIn : <https://www.linkedin.com/in/kunajun77>

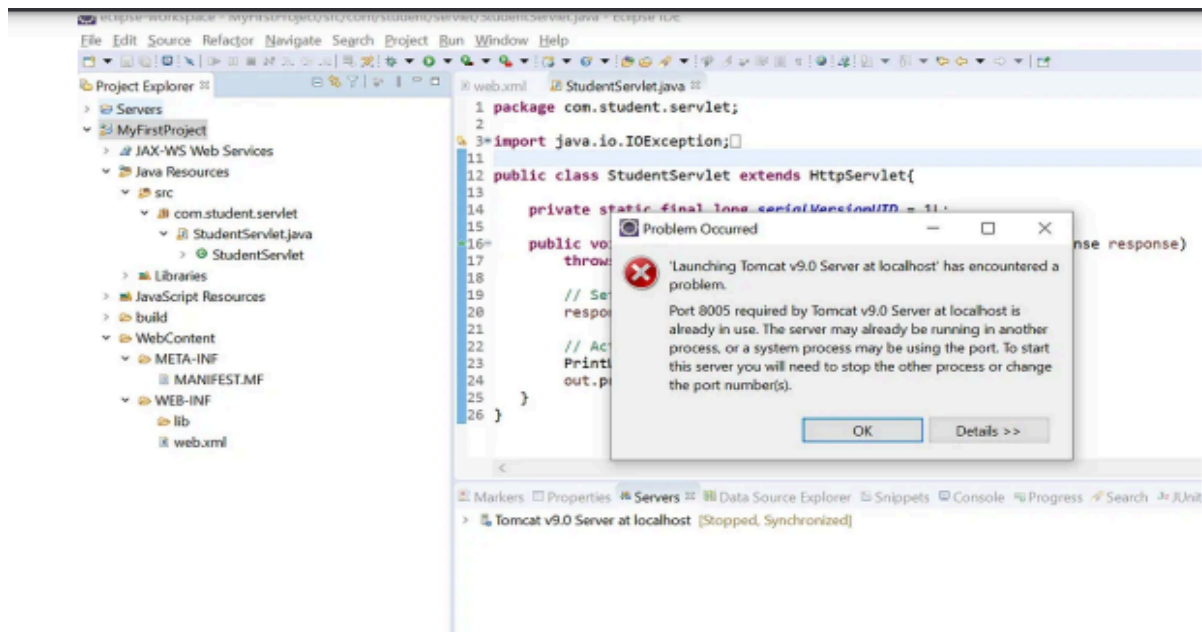
To deploy Servlet “Dynamic Web Project” follow below steps:

1. On the top menu in eclipse tool select run → Run on Server

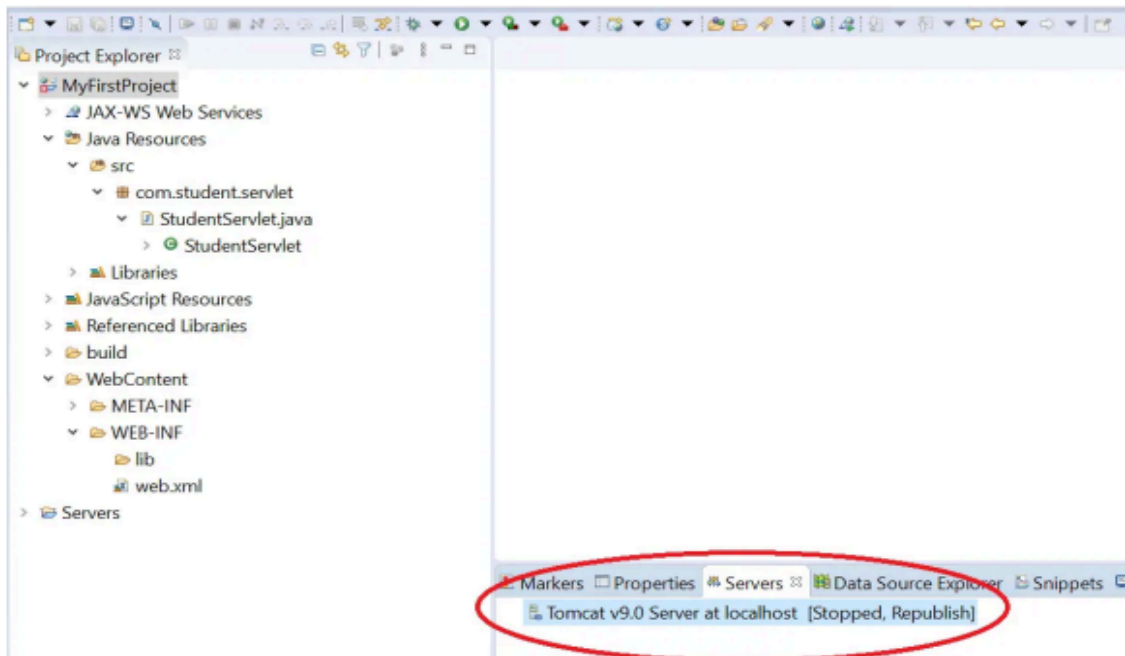
Select Tomcat as shown in the screen below and click on Finish, wait for 1 min for the server to get started.



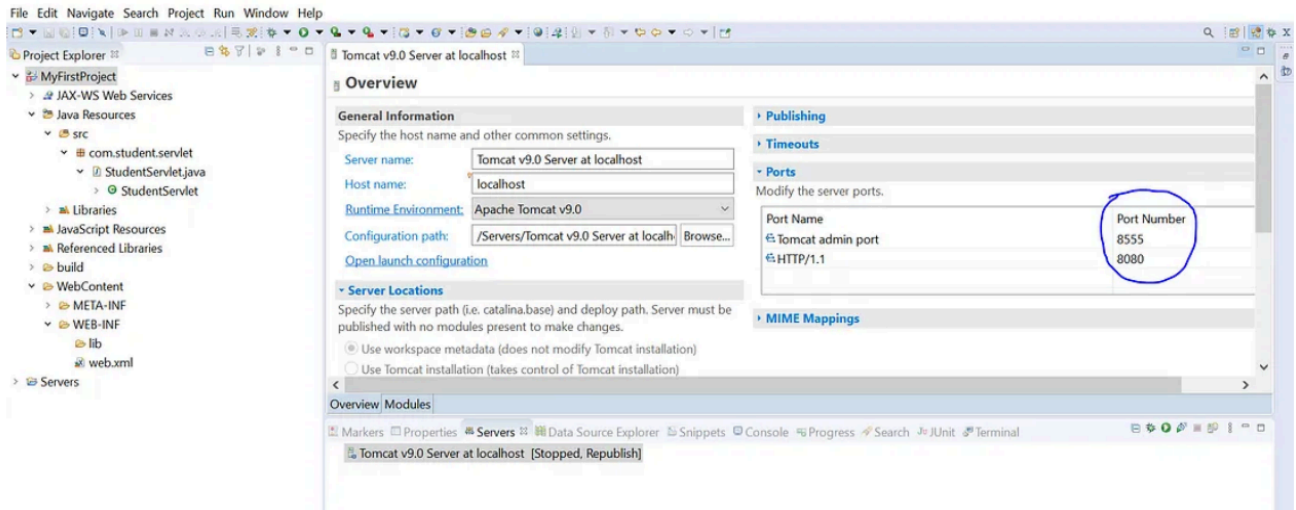
NOTE: We can face below issue as soon as we start Apache Tomcat server start. This issue occurs if the port is in use by some other application.



To resolve this issue please double click on below link highlighted in red in color.



Update the port to any four digit number which is not in use and click on save.



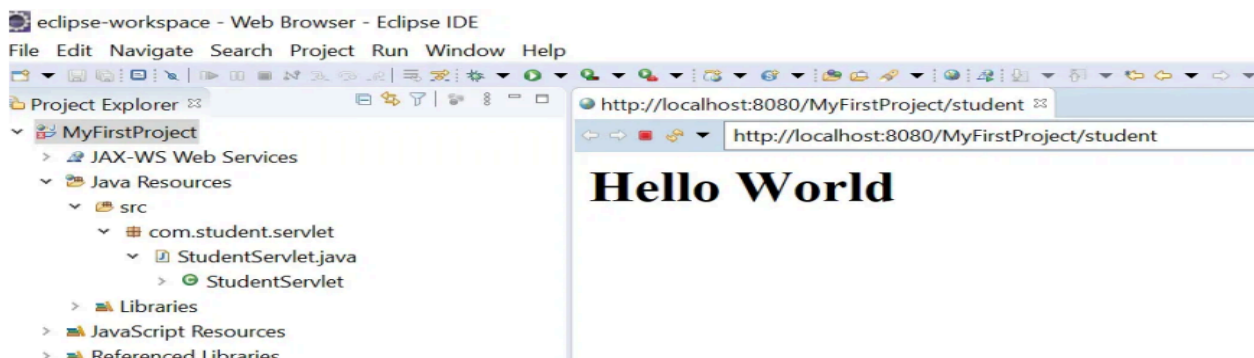
Please follow 1st step under “Deploy Servlet Project on Apache Tomcat” and redeploy the code and hit below URL to see result:

<http://localhost:8080/MyFirstProject/student>

<http://localhost:8080/MyFirstProject/student>



OUTPUT:



Do connect with me on LinkedIn : <https://www.linkedin.com/in/kunajun77>

Spring Vs Spring Core Vs Spring Boot

Spring is robust and lightweight framework us to develop enterprise applications. Spring is an open source Java platform developed by Rod Johnson in 2003.

Spring makes it very easy to develop Java Enterprise application.

Spring is supported by all JDK 8+ versions. It provides out of the box support to JDK 11.

According to java official documentation, “Spring” means different things in different contexts

We can easily create Spring project using <https://start.spring.io/>. This website will help us to create Restful web services.

There is one more open source editor named as Spring too kit is there in market to build Spring applications.

Below are the various modules supported by Spring framework:

Spring Core: It is the main feature of dependency injection and inversion of control of Spring core framework. Dependency Injection is a main feature of Spring core framework. This provides a dependency.

To understand dependency injection and inversion of control for ex: We have two classes as Class A and Class B. Dependency injection will come in picture if we want to make a relation or inject Class A in to Class B. Dependency injection can be achieved using parameterized constructor and getter setter methods. This complete process of injection is called IoC (Inversion of Control).

Spring AOP: Aspect Oriented Programming (AOP) is one of the key feature of Spring framework. AOP breaks down complete code into multiple modules.

AOP is similar to OOPS concept where it breaks code into multiple reusable

Do connect with me on LinkedIn : <https://www.linkedin.com/in/kunajun77>

modules. AOP provides the capability to dynamically add module at runtime.

Spring Web MVC (Model View Controller): Spring web is all about MVC architecture. It also allows us to create flexible and loosely coupled web application.

Spring DAO (Java Data Objects): It provides an interface which provides us the capability to connect with Spring MVC.

Spring ORM (Object relational mapping): ORM is responsible to access and manipulate data using some framework such as Java Persistence API (JPA), Java Data Objects (JDO), iBATIS SQL Maps, etc.

Spring context: Spring context is a IoC containers. It instantiates, configures, and assembles beans or pojo classes by reading XML configurations, Java annotations, and/or Java code in the configuration files.

Why is Spring Framework famous?

Spring offers other modules such as Spring Core, Security, MVC, Spring Boot, and Logging, which is very easy to integrate, loosely coupled, easy to test and helps us to develop an application with a lot more features in less time. We can test the Spring Framework using Junit. The Spring Framework has a big community.

Just for future reference:

Spring framework all together is having three different flavors known as Spring core, Spring MVC and Spring Boot. Spring MVC uses Spring core and Spring boot uses Spring MVC which internally uses Spring Core.

Spring MVC

Spring web is all about MVC architecture. It also allows us to create flexible and loosely coupled web application.

It is a very well known design pattern to develop web application where we divide complete application in to three sub parts known as Model, View and Controller.

Model → This is responsible for maintaining data. e.g. bean or Pojo class.

View → It is a representation of the end result to output or show on the on the page.

e.g. JSP, HTML etc.

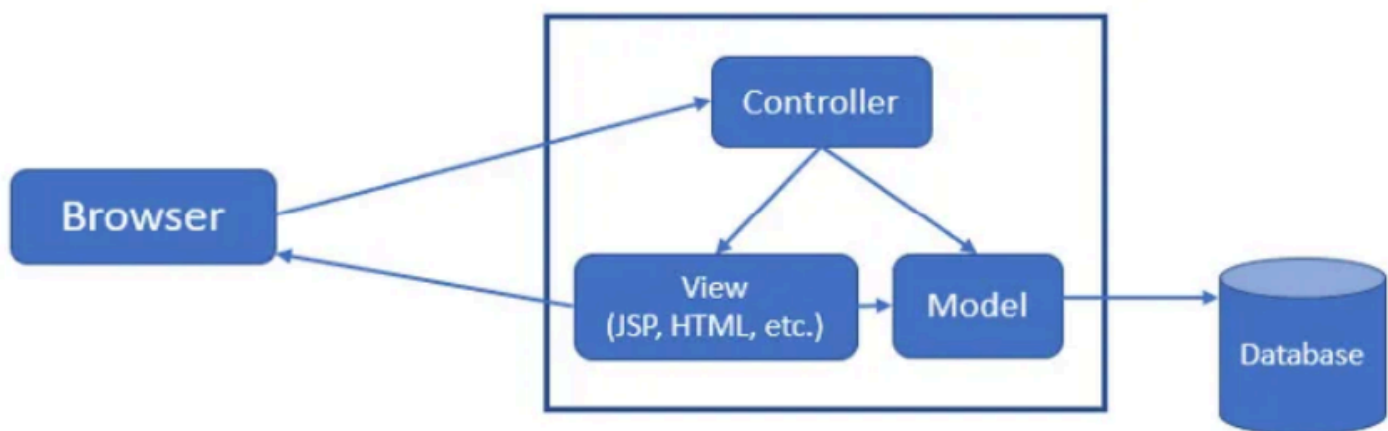
Controller → Controller is a backend code which gets trigger on performing any action or user input.

The browser will call the controller as shown in the below inline diagram. The controller will decide whether to call view or model.

1st Scenario → The browser will call the controller and, depending on input data, the model will call the database as it is tightly coupled using JPA or Hibernate. On getting a response back from the database, the model will call the view to output data on the browser output screen.

2nd Scenario → The browser will call the controller and, depending on the input data, the view will call the database as it is tightly coupled using JPA or Hibernate. On getting a response back from the database, the view will output data on the browser output screen.

Mode can get requests to return data from both View and Controller.



Spring Boot :

Spring Boot is an extended version of Spring framework which provides a capability to develop a project with more features in less time.

There are multiple things the Spring Boot project provides.

1. Spring Boot initializer which helps us to create a project in less than 10 min.
2. Spring Boot starter projects help us to define all dependencies for our projects.
3. Spring Boot DevTools helps us to deploy Spring Boot applications without server restart.

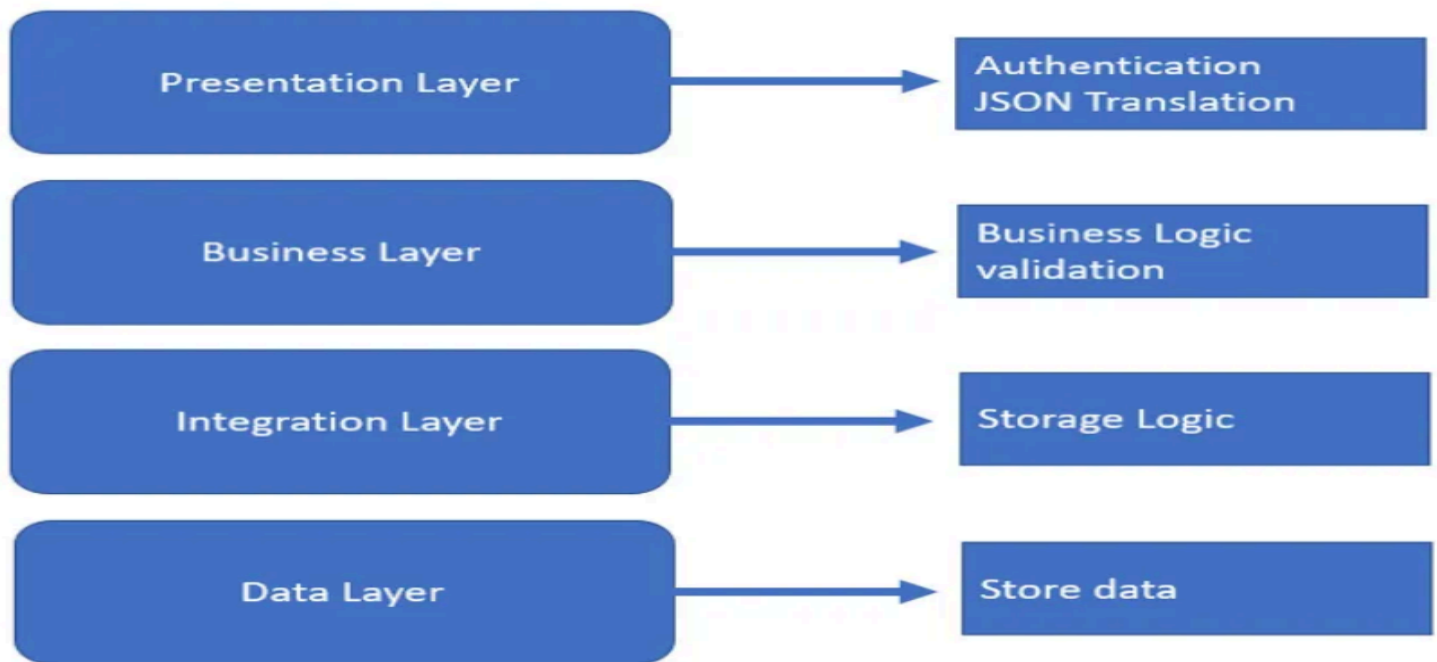
Spring Boot follows a layered architecture where each layer communicates with another required layer. There are four layers in Spring boot:

Presentation Layer: This layer is used to present JSON on the front end and authenticate end users.

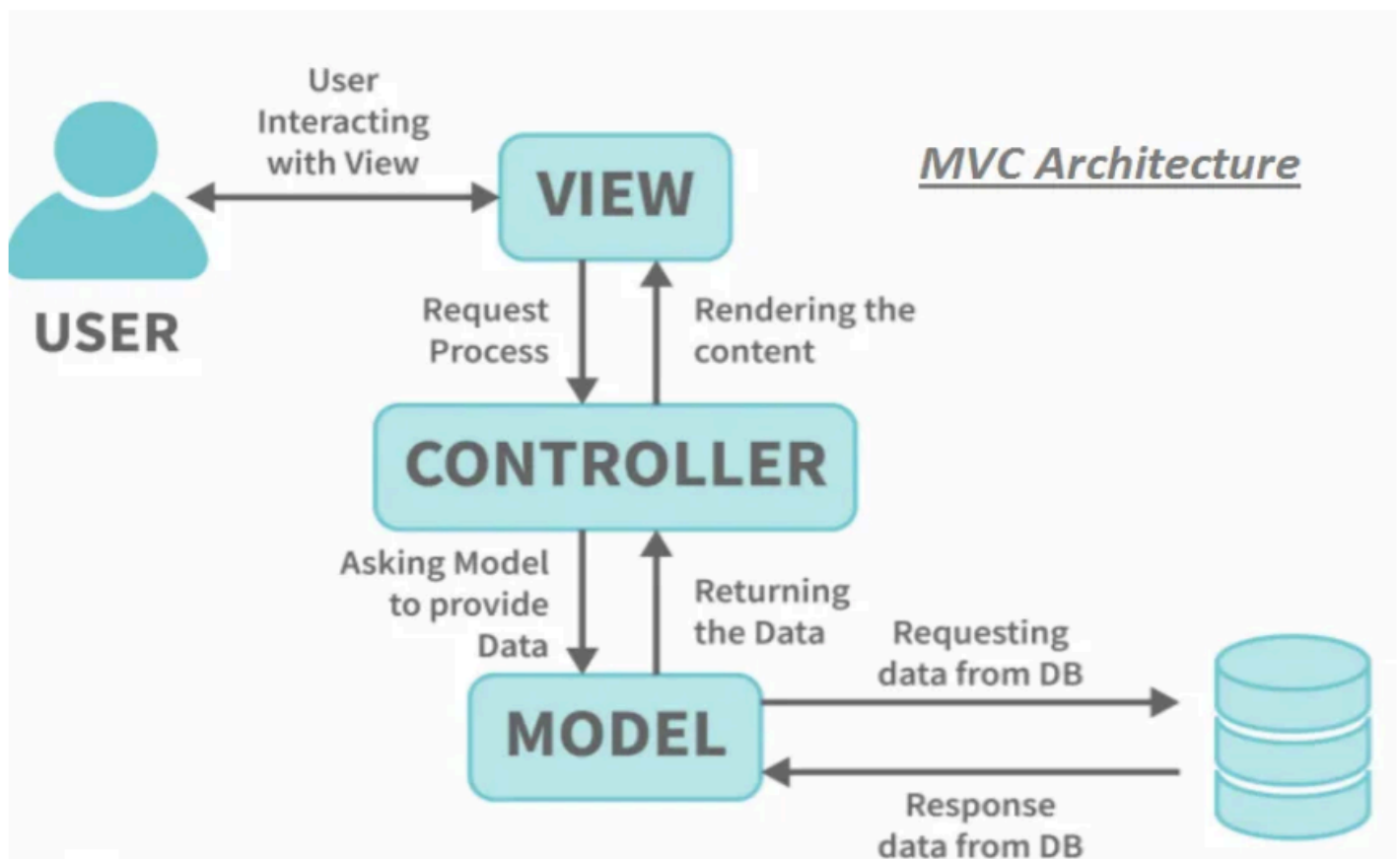
Business Layer: This layer is all about writing business logic and things related to authorization and authentication.

Integration Layer: This layer is responsible for storage logic and convert business objects from and to database rows.

Data Layer: As name suggests it is related to data and perform operations such as CRUD (create, retrieve, update, delete).



Model View Controller(MVC) Architecture with JSP example



MVC) is a software architectural pattern that separates an application into three main logical components: the model, the view, and the controller

Model: The model represents the underlying, logical structure of data in a software application and the high-level class associated with it. It represents the business logic and the underlying data.

View: The view is responsible for presenting the model to the user. It is a visual representation of the model and is rendered on the screen.

Controller: The controller is the intermediary between the model and the view. It receives input from the user, updates the model as needed, and presents the updated model to the view.

The MVC pattern helps to separate the concerns of the different parts of the application, making it easier to develop, maintain, and test.

There has been a shift in recent years away from traditional MVC frameworks towards more modern, component-based architectures such as React and Angular. These frameworks are often referred to as “single-page applications” (SPAs) because they can handle all the rendering and updating of the page on the client side, without the need for the server to send a new page with each request.

However, traditional MVC frameworks are still widely used and continue to be popular choices for building web applications. Some examples of popular MVC frameworks include ASP.NET MVC, Ruby on Rails, and Spring MVC.

It is also worth noting that the MVC pattern is not limited to web applications. It can be applied to other types of software as well, such as desktop applications and mobile apps.

MVC Example with JSP:

The following example consists of three .jsp files, one servlet and a bean class. The servlets and beans placed appropriately in the corresponding packages viz. controller and beans.

index.jsp — A page that gets input from the user.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>

<form action="ControllerServlet" method="post">
Name:<input type="text" name="name"><br>
Password:<input type="password" name="password"><br> <br>
<input type="submit" value="login">
</form>
```

ControllerServlet.java — A servlet that acts as a controller.

```
@WebServlet(urlPatterns = {"/ControllerServlet"})
public class ControllerServlet extends HttpServlet {

    protected void processRequest(HttpServletRequest request, HttpServletResponse resp throws
        ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        try (PrintWriter out = response.getWriter()) {
            String name = request.getParameter("name");
            String password = request.getParameter("password");

            LoginBean bean = new LoginBean();
            bean.setName(name);
            bean.setPassword(password);
            request.setAttribute("bean", bean);

            boolean status = bean.validate();

            if (status) {
                RequestDispatcher rd = request.getRequestDispatcher("login_success.jsp");
                rd.forward(request, response);
            }
        }
    }
}
```

```

    }
    else {
        RequestDispatcher rd = request.getRequestDispatcher("login_error.jsp")
        rd.forward(request, response);
    }
}
}

```

LoginBean.java — A java bean (POJO) with name and password as properties. It validates users with password 'admin'.

```

package beans;

public class LoginBean {
    private String name,password;

    public String getName() {
        return name;
    }

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

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public LoginBean() {
    }

    public boolean validate(){
        if(password.equals("admin")){
            return true;
        }
        else{
            return false;
        }
    }
}

```

login-success.jsp and login-error.jsp files acts as view components.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@page import="beans.LoginBean"%>

<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"> <title>Login
        Success JSP Page</title>
    </head>
    <body>
        <p>You are successfully logged in!</p>
        <%
            LoginBean bean = (LoginBean) request.getAttribute("bean");
            out.print("Welcome, " + bean.getName());
        %>
    </body>
</html>
```

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"> <title>Error
        JSP Page</title>
    </head>
    <body>
        <p>Sorry! Credentials doesn't match</p>
        <%@ include file="index.jsp" %>
    </body>
</html>
```

Separation of concerns is one of the most attractive concepts of the MVC pattern. The complexity of modern web applications can make it difficult to make changes. When the frontend and backend are separated, the application is scalable, maintainable, and easy to expand.

Thanks for reading! Happy Coding!!

Restful API with SpringBoot:

Representational State Transfer (REST) is an architectural style for designing networked applications. RESTful APIs are web services that follow REST principles, allowing clients to interact with resources through a standardized set of HTTP methods.

The main principles of RESTful APIs are:

Stateless: Each request from a client to a server must contain all the information needed to understand and process the request. The server should not store information about the client's state between requests.

Client-Server: The client and server are separate entities that communicate over a network. The client is responsible for the user interface, while the server manages the resources and handles the business logic.

Cacheable: Responses from the server can be cached by the client to improve performance and reduce the load on the server.

Layered System: The architecture can be composed of multiple layers, each providing a specific set of functionality. This allows for separating concerns and makes the system more modular and maintainable.

I will show you how to create a simple RESTful API with Spring Boot, how to add validation for the API input, and finally, how to test. Let's dive into it.

Creating controllers, routes, and handling HTTP requests In a Spring Boot application, you can create RESTful APIs by defining controllers, routes, and handling HTTP requests. To do this, follow these steps.

Create a controller

Define a class and annotate it with `@RestController`. This tells Spring Boot that this class handles HTTP requests and generates responses.

```
@RestController
public class MyController {
    // Controller methods
}
```

Define routes

Use the `@RequestMapping` annotation (or one of its shortcuts, like `@GetMapping`, `@PostMapping`, etc.) to map specific HTTP requests to your controller methods. Specify the HTTP method, the route path, and optionally, additional attributes like `consumes` or `produces`.

```
@RestController
public class MyController {
```

Do connect with me on LinkedIn : <https://www.linkedin.com/in/kunajun77>

```

    @GetMapping("/hello")
    public String hello() {
        return "Hello, World!";
    }
}

```

Handle HTTP requests

Your controller methods can accept parameters automatically populated by Spring Boot based on the incoming HTTP request. Use annotations like `@PathVariable` , `@RequestParam` , and `@RequestBody` to bind request data to your method parameters.

```

@RestController
public class MyController {

    @PostMapping("/person")
    public Person createPerson(@RequestBody Person person) {
        // Handle the request and create a new person
        return person;
    }
}

```

Implementing CRUD operations with a sample application To demonstrate how to build a RESTful API with Spring Boot, let's create a sample application for managing a list of persons. Using a simple in-memory data store, we'll implement CRUD (Create, Read, Update, Delete) operations.

Create a Person class

Define a simple Person class with a few fields (e.g., `id` , `firstName` , `lastName`).

```

public class Person {
    private Long id;
    private String firstName;
    private String lastName;
    // Getters and setters
}

```


Create a PersonController

Define a PersonController class annotated with @RestController . Implement the CRUD operations by defining controller methods and using the appropriate HTTP methods and routes.

```
@RestController
@RequestMapping("/persons")
public class PersonController {

    private final AtomicLong counter = new AtomicLong();
    private final Map<Long, Person> persons = new ConcurrentHashMap<>();

    @PostMapping
    public Person createPerson(@RequestBody Person person) {
        long id = counter.incrementAndGet();
        person.setId(id);
        persons.put(id, person);
        return person;
    }

    @GetMapping("/{id}")
    public Person getPerson(@PathVariable("id") Long id) {
        return persons.get(id);
    }

    @GetMapping
    public Collection<Person> getAllPersons() {
        return persons.values();
    }

    @PutMapping("/{id}")
    public Person updatePerson(@PathVariable("id") Long id, @RequestBody Person up Person person
        = persons.get(id);
        if (person == null) {
            throw new ResourceNotFoundException("Person not found with id: " + id) }
        person.setFirstName(updatedPerson.getFirstName());
        person.setLastName(updatedPerson.getLastName());
        persons.put(id, person);
        return person;
    }

    @DeleteMapping("/{id}")
    public void deletePerson(@PathVariable("id") Long id) {
        persons.remove(id);
    }
}
```

Do connect with me on LinkedIn : <https://www.linkedin.com/in/kunajun77>

```
}  
}
```

In this example, we've created a `PersonController` class that manages a simple in memory data store of `Person` objects. The controller provides CRUD operations through different HTTP methods and routes:

`@PostMapping` : Creates a new person using the request body, assigns a unique ID, and stores it in the `persons` map.

`@GetMapping("/{id}")` : Retrieves a person by its ID.

`@GetMapping` : Returns a collection of all persons in the data store.

`@PutMapping("/{id}")` : Updates an existing person's first and last name, identified by its ID.

`@DeleteMapping("/{id}")` : Deletes a person by its ID.

With this sample application, you've learned how to create a RESTful API with Spring Boot, define controllers, routes, and handle HTTP requests for various CRUD operations. You can now build on this foundation to create more complex APIs backed by real databases and integrate additional features like security, validation, and pagination.

Adding Validation and Error Handling to Your RESTful API To provide a robust and user-friendly API, it's essential to validate user input and handle errors gracefully. In this section, we'll show you how to add validation to your RESTful API using Spring Boot and handle any possible errors.

To add validation to your API, follow these steps

Add validation annotations

Annotate the fields in your model class (e.g., `Person`) with validation constraints from the `javax.validation.constraints` package. For example, use `@NotNull`, `@Size`, or `@Pattern` to specify constraints on your fields.

```

public class Person {
    private Long id;

    @NotNull(message = "First name must not be null")
    @Size(min = 1, max = 100, message = "First name must be between 1 and 100 char") private String
    firstName;

    @NotNull(message = "Last name must not be null")
    @Size(min = 1, max = 100, message = "Last name must be between 1 and 100 char") private String
    lastName;

    // Getters and setters
}

```

Use @Valid annotation for validation

Add the @Valid annotation to your controller method parameters to trigger validation when a request is received. If any validation errors are found, a

MethodArgumentNotValidException is thrown.

```

@PostMapping
public Person createPerson(@Valid @RequestBody Person person) {
    // ...
}

```

To handle errors and provide custom error responses, follow these steps.

Create a custom error response class

Define a class to represent your custom error response. This class will send a structured error message to the client when an error occurs.

```

public class ErrorResponse {
    private int status;
    private String message;
    private List<String> errors;

    // Getters and setters
}

```

Handle exceptions in a controller advice

Create a class annotated with `@ControllerAdvice` to handle exceptions thrown by your controllers. Use the `@ExceptionHandler` annotation to define methods that handle specific exception types.

```
@ControllerAdvice
public class ApiExceptionHandler {

    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<ErrorResponse> handleValidationExceptions(MethodArgument List<String>
        errors = ex.getBindingResult()
                                .getAllErrors().stream()
                                .map(ObjectError::getDefaultMessage)
                                .collect(Collectors.toList());

        ErrorResponse errorResponse = new ErrorResponse();
        errorResponse.setStatus(HttpStatus.BAD_REQUEST.value());
        errorResponse.setMessage("Validation error");
        errorResponse.setErrors(errors);

        return new ResponseEntity<>(errorResponse, HttpStatus.BAD_REQUEST); }

    // Other exception handlers can be added here
}
```

In this example, we handle the `MethodArgumentNotValidException` thrown when validation errors occur and return a custom error response with the validation error messages. With these additions, your RESTful API now includes validation and error handling, making it more robust and user-friendly. You can expand on this foundation by adding more advanced features such as authentication, authorization, and rate limiting to ensure your API is secure and scalable.

Testing Your API with Spring Boot Test

Spring Boot Test is a powerful testing framework supporting various testing scenarios. Here's how to write tests for your RESTful API.

Add the Spring Boot Test dependency

Add the `spring-boot-starter-test` dependency to your project's `pom.xml` or `build.gradle` file.

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

Write test cases for your controllers

Create test classes for your controllers and annotate them with `@WebMvcTest`. Use the `@Autowired` annotation to inject a `MockMvc` instance, which you can use to send HTTP requests and assert their responses.

```
@WebMvcTest(PersonController.class)
public class PersonControllerTest {

    @Autowired
    private MockMvc mockMvc;

    // Test cases
}
```

In your test cases, use the `mockMvc.perform()` method to send HTTP requests and the `andExpect()` method to assert the responses.

```
@Test
public void testGetPerson() throws Exception {
    // Prepare data and mock service behavior

    mockMvc.perform(get("/persons/1"))
        .andExpect(status().isOk())
        .andExpect(jsonPath("$.firstName").value("John"))
        .andExpect(jsonPath("$.lastName").value("Doe"));
}
```

In this example, we've written a test case for the `getPerson()` method in our

PersonController . We send a GET request to the /persons/1 path, asserting that the response has a 200 OK status and the expected JSON content.

Conclusion

Spring Boot RESTful API. We introduced the basics of RESTful APIs, creating a simple API. Then, we delved into error handling and validation to enhance the API's robustness. By following the examples and concepts presented here, you should now understand the fundamental principles and best practices for building a Spring Boot RESTful API.

As you continue to develop your API, you can explore additional Spring Boot features, such as securing, caching, asynchronous processing, and event-driven architectures, to further optimize and scale your application. With Spring Boot's rich ecosystem and strong community support, there are endless possibilities for creating powerful, flexible, and maintainable APIs for your applications.

Chapter 1

Spring overview

1.1 What is Spring?

Spring is an open source development framework for **Enterprise Java**. 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 targets to make Java EE development easier to use and promote good programming practice by enabling a **POJO based programming model**.

1.2 What are the benefits of Spring Framework?

- **Lightweight**: Spring is lightweight when it comes to size and transparency. The basic version of spring framework is around 2MB.
- **Inversion of control (IOC)**: Loose coupling is achieved in Spring, with the **Inversion of Control technique**. The objects give their dependencies instead of creating or looking for dependent objects.
- **Aspect oriented (AOP)**: **Spring supports Aspect oriented programming** and separates application business logic from system services.
- **Container**: Spring contains and manages the life cycle and configuration of application objects.
- **MVC Framework**: Spring's web framework is a well-designed **web MVC framework**, which provides a great alternative to web frameworks.
- **Transaction Management**: Spring provides a consistent transaction management interface that can scale down to a local transaction and scale up to global transactions (JTA).
- **Exception Handling**: Spring provides a convenient API to translate technology-specific exceptions (thrown by JDBC, Hibernate, or JDO) into consistent, unchecked exceptions.

1.3 Which are the Spring framework modules?

The basic modules of the Spring framework are :

- Core module
- Bean module
- Context module
- Expression Language module
- **JDBC module**
- **ORM module**
- OXM module
- Java Messaging Service(JMS) module
- Transaction module
- Web module
- Web-Servlet module
- Web-Struts module
- Web-Portlet module

1.4 Explain the Core Container (Application context) module

This is the basic Spring module, which provides the fundamental functionality of the Spring framework. BeanFactory

Do connect with me on LinkedIn : <https://www.linkedin.com/in/kunajun77>

is the heart of any spring-based application. Spring framework was built on the top of this module, which makes the Spring container.

1.5 BeanFactory - BeanFactory implementation example

A BeanFactory is an implementation of the factory pattern that applies Inversion of Control to separate the application's configuration and dependencies from the actual application code.

The most commonly used BeanFactory implementation is the XmlBeanFactory class.

1.6 XMLBeanFactory

The most useful one is org.springframework.beans.factory.xml.XmlBeanFactory, which loads its beans based on the definitions contained in an XML file. This container reads the configuration metadata from an XML file and uses it to create a fully configured system or application.

1.7 Explain the AOP module

The AOP module is used for developing aspects for our Spring-enabled application. Much of the support has been provided by the AOP Alliance in order to ensure the interoperability between **Spring and other AOP frameworks**. This module also introduces metadata programming to Spring.

1.8 Explain the JDBC abstraction and DAO module

With the **JDBC abstraction and DAO module** we can be sure that we keep up the database code clean and simple, and prevent problems that result from a failure to close database resources. It provides a layer of meaningful exceptions on top of the error messages given by several database servers. It also makes use of Spring's AOP module to provide transaction management services for objects in a Spring application.

1.9 Explain the object/relational mapping integration module

Spring also supports for using of an **object/relational mapping (ORM) tool** over straight JDBC by providing the ORM module. Spring provides support to tie into several popular ORM frameworks, including **Hibernate**, **JDO**, and **iBATIS SQL Maps**. Spring's transaction management supports each of these ORM frameworks as well as JDBC.

1.10 Explain the web module

The **Spring web module** is built on the application context module, providing a context that is appropriate for web-based applications. This module also contains support for several web-oriented tasks such as transparently handling multipart requests for file uploads and programmatic binding of request parameters to your business objects. It also contains integration support with Jakarta Struts.

1.11 Explain the Spring MVC module

MVC framework is provided by Spring for building web applications. Spring can easily be integrated with other MVC frameworks, but **Spring's MVC framework** is a better choice, since it uses IoC to provide for a clean separation of controller logic from business objects. With Spring MVC you can declaratively bind request parameters to your business objects.

1.12 Spring configuration file

Spring configuration file is an XML file. This file contains the classes information and describes how these classes are configured and introduced to each other.

1.13 What is Spring IoC container?

The Spring IoC is responsible for creating the objects, managing them (with dependency injection (DI)), wiring them together, configuring them, as also managing their complete lifecycle.

1.14 What are the benefits of IOC?

IOC or dependency injection minimizes the amount of code in an application. It makes easy to test applications, since no singletons or JNDI lookup mechanisms are required in unit tests. Loose coupling is promoted with minimal effort and least intrusive mechanism. IOC containers support eager instantiation and lazy loading of services.

1.15 What are the common implementations of the ApplicationContext?

The FileSystemXmlApplicationContext container loads the definitions of the beans from an XML file. The full path of the XML bean configuration file must be provided to the constructor. The ClassPathXmlApplicationContext container also loads the definitions of the beans from an XML file. Here, you need to set CLASSPATH properly because this container will look bean configuration XML file in CLASSPATH. The WebXmlApplicationContext container loads the XML file with definitions of all beans from within a web application.

1.16 What is the difference between Bean Factory and ApplicationContext?

Application contexts provide a means for resolving text messages, a generic way to load file resources (such as images), they can publish events to beans that are registered as listeners. In addition, operations on the container or beans in the container, which have to be handled in a programmatic fashion with a bean factory, can be handled declaratively in an application context. The application context implements MessageSource, an interface used to obtain localized messages, with the actual implementation being pluggable.

1.17 What does a Spring application look like?

- An interface that defines the functions.
- The implementation that contains properties, its setter and getter methods, functions etc.,
- Spring AOP
- The Spring configuration XML file.
- Client program that uses the function

Chapter 2 : Dependency Injection

2.1 What is Dependency Injection in Spring?

Dependency Injection, an aspect of Inversion of Control (IoC), is a general concept, and it can be expressed in many different ways. This concept says that you do not create your objects but describe how they should be created. You don't directly connect your components and services together in code but describe which services are needed by which components in a configuration file. A container (the IOC container) is then responsible for hooking it all up.

2.2 What are the different types of IoC (dependency injection)?

- **Constructor-based dependency injection:** Constructor-based DI is accomplished when the container invokes a class constructor with a number of arguments, each representing a dependency on other class.
- **Setter-based dependency injection:** Setter-based DI is accomplished by the container calling setter methods on your beans after invoking a no-argument constructor or no-argument static factory method to instantiate your bean.

2.3 Which DI would you suggest Constructor-based or setter-based DI?

You can use both Constructor-based and Setter-based Dependency Injection. The best solution is using constructor arguments for mandatory dependencies and setters for optional dependencies

Chapter 3 : Spring Beans

3.1 What are Spring beans?

The **Spring Beans** are Java Objects that form the backbone of a Spring application. They are instantiated, assembled, and managed by the Spring IoC container. These beans are created with the configuration metadata that is supplied to the container, for example, in the form of XML <bean/> definitions.

Beans defined in spring framework are singleton beans. There is an attribute in bean tag named "singleton" if specified true then bean becomes singleton and if set to false then the bean becomes a prototype bean. By default it is set to true. So, all the beans in spring framework are by default singleton beans.

3.2 What does a Spring Bean definition contain?

A Spring Bean definition contains all configuration metadata which is needed for the container to know how to create a bean, its lifecycle details and its dependencies.

3.3 How do you provide configuration metadata to the Spring Container?

There are three important methods to provide configuration metadata to the Spring Container:

- XML based configuration file.
- Annotation-based configuration

- Java-based configuration

3.4 How do you define the scope of a bean?

When defining a <bean> in Spring, we can also declare a scope for the bean. It can be defined through the scope attribute in the bean definition. For example, when Spring has to produce a new bean instance each time one is needed, the bean's scope attribute to be prototype. On the other hand, when the same instance of a bean must be returned by Spring every time it is needed, the the bean scope attribute must be set to singleton.

3.5 Explain the bean scopes supported by Spring

There are five scopes provided by the Spring Framework supports following five scopes:

- In singleton scope, Spring scopes the bean definition to a single instance per Spring IoC container.
- In prototype scope, a single bean definition has any number of object instances.
- In request scope, a bean is defined to an HTTP request. This scope is valid only in a web-aware Spring ApplicationContext.
- In session scope, a bean definition is scoped to an HTTP session. This scope is also valid only in a web-aware Spring ApplicationContext.
- In global-session scope, a bean definition is scoped to a global HTTP session. This is also a case used in a web-aware Spring ApplicationContext.

The default scope of a Spring Bean is Singleton.

3.6 Are Singleton beans thread safe in Spring Framework?

No, singleton beans are not thread-safe in Spring framework.

3.7 Explain Bean lifecycle in Spring framework

- The spring container finds the bean's definition from the XML file and instantiates the bean.
- Spring populates all of the properties as specified in the bean definition (DI).
- If the bean implements BeanNameAware interface, spring passes the bean's id to setBeanName() method.
- If Bean implements BeanFactoryAware interface, spring passes the beanfactory to setBeanFactory() method.
- If there are any bean BeanPostProcessors associated with the bean, Spring calls postProcessorBeforeInitialization() method.
- If the bean implements InitializingBean, its afterPropertySet() method is called. If the bean has init method declaration, the specified initialization method is called.
- If there are any BeanPostProcessors associated with the bean, their postProcessAfterInitialization() methods will be called.
- If the bean implements DisposableBean, it will call the destroy() method.

3.8 Which are the important beans lifecycle methods? Can you override them?

There are two important bean lifecycle methods. The first one is setup which is called when the bean is loaded in to the container. The second method is the teardown method which is called when the bean is unloaded from the

container. The bean tag has two important attributes (init-method and destroy-method) with which you can define your own custom initialization and destroy methods. There are also the corresponding annotations (@PostConstruct and @PreDestroy).

3.9 What are inner beans in Spring?

When a bean is only used as a property of another bean it can be declared as an inner bean. Spring's XML-based configuration metadata provides the use of <bean/> element inside the <property/> or <constructor-arg/> elements of a bean definition, in order to define the so-called inner bean. Inner beans are always anonymous and they are always scoped as prototypes.

3.10 How can you inject a Java Collection in Spring?

Spring offers the following types of **collection configuration elements**:

- The <list> type is used for injecting a list of values, in the case that duplicates are allowed.
- The <set> type is used for wiring a set of values but without any duplicates.
- The <map> type is used to inject a collection of name-value pairs where name and value can be of any type.
- The <props> type can be used to inject a collection of name-value pairs where the name and value are both Strings.

3.11 What is bean wiring?

Wiring, or else bean wiring is the case when beans are combined together within the Spring container. When wiring beans, the Spring container needs to know what beans are needed and how the container should use dependency injection to tie them together.

3.12 What is bean auto wiring?

The Spring container is able to **autowire relationships** between collaborating beans. This means that it is possible to automatically let Spring resolve collaborators (other beans) for a bean by inspecting the contents of the BeanFactory without using <constructor-arg> and <property> elements.

3.13 Explain different modes of auto wiring?

The autowiring functionality has five modes which can be used to instruct Spring container to use autowiring for dependency injection:

- no: This is default setting. Explicit bean reference should be used for wiring.
- byName: When autowiring byName, the Spring container looks at the properties of the beans on which autowire attribute is set to byName in the XML configuration file. It then tries to match and wire its properties with the beans defined by the same names in the configuration file.
- byType: When autowiring by datatype, the Spring container looks at the properties of the beans on which autowire attribute is set to byType in the XML configuration file. It then tries to match and wire a property if its type matches with exactly one of the beans name in configuration file. If more than one such beans exist, a fatal exception is thrown.
- constructor: This mode is similar to byType, but type applies to constructor arguments. If there is not exactly one bean of the constructor argument type in the container, a fatal error is raised.
- autodetect: Spring first tries to wire using autowire by constructor, if it does not work, Spring tries to autowire by byType.

3.14 Are there limitations with autowiring?

Limitations of autowiring are:

Do connect with me on LinkedIn : <https://www.linkedin.com/in/kunajun77>

- Overriding: You can still specify dependencies using <constructor-arg> and <property> settings which will always override autowiring.
- Primitive data types: You cannot autowire simple properties such as primitives, Strings, and Classes.
- Confusing nature: Autowiring is less exact than explicit wiring, so if possible prefer using explicit wiring.

3.15 Can you inject null and empty string values in Spring?

Yes, you can.

Chapter 4 : Spring Annotations

4.1 What is Spring Java-Based Configuration? Give some annotation example.

Java based configuration option enables you to write most of your Spring configuration without XML but with the help of few Java-based annotations. An example is the @Configuration annotation, that indicates that the class can be used by the Spring IoC container as a source of bean definitions. Another example is the @Bean annotated method that will return an object that should be registered as a bean in the Spring application context.

4.2 What is Annotation-based container configuration?

An alternative to XML setups is provided by annotation-based configuration which relies on the bytecode metadata for wiring up components instead of angle-bracket declarations. Instead of using XML to describe a bean wiring, the developer moves the configuration into the component class itself by using annotations on the relevant class, method, or field declaration.

4.3 How do you turn on annotation wiring?

Annotation wiring is not turned on in the Spring container by default. In order to use annotation based wiring we must enable it in our Spring configuration file by configuring <context:annotation-config/> element.

4.4 @Required annotation

This annotation simply indicates that the affected bean property must be populated at configuration time, through an explicit property value in a bean definition or through autowiring. The container throws BeanInitializationException if the affected bean property has not been populated.

4.5 @Autowired annotation

The `@Autowired` annotation provides more fine-grained control over where and how autowiring should be accomplished. It can be used to autowire beans on the setter method just like `@Required` annotation, on the constructor, on a property or on methods with arbitrary names and/or multiple arguments.

4.6 @Qualifier annotation

When there are more than one beans of the same type and only one is needed to be wired with a property, the `@Qualifier` annotation is used along with `@Autowired` annotation to remove the confusion by specifying which exact bean will be wired.

Chapter 5 : Spring Data Access

5.1 How can JDBC be used more efficiently in the Spring framework?

When using the Spring JDBC framework the burden of resource management and error handling is reduced. So developers only need to write the statements and queries to get the data to and from the database. JDBC can be used more efficiently with the help of a template class provided by Spring framework, which is the `JdbcTemplate` (example [here](#)).

5.2 JdbcTemplate

`JdbcTemplate` class provides many convenience methods for doing things such as converting database data into primitives or objects, executing prepared and callable statements, and providing custom database error handling.

5.3 Spring DAO support

The [Data Access Object \(DAO\) support in Spring](#) is aimed at making it easy to work with data access technologies like JDBC, Hibernate or JDO in a consistent way. This allows us to switch between the persistence technologies fairly easily and to code without worrying about catching exceptions that are specific to each technology.

5.4 What are the ways to access Hibernate by using Spring?

There are two ways to access Hibernate with Spring:

- Inversion of Control with a Hibernate Template and Callback.
- Extending `HibernateDAOSupport` and Applying an AOP Interceptor node

5.5 ORM's Spring support

Do connect with me on LinkedIn : <https://www.linkedin.com/in/kunajun77>

Spring supports the following ORM's:

- Hibernate
- iBatis
- JPA (Java Persistence API)
- TopLink
- JDO (Java Data Objects)
- OJB

5.6 How can we integrate Spring and Hibernate using HibernateDaoSupport?

Use Spring's SessionFactory called LocalSessionFactory. The integration process is of 3 steps:

- Configure the Hibernate SessionFactory
- Extend a DAO Implementation from HibernateDaoSupport
- Wire in Transaction Support with AOP

5.7 Types of the transaction management Spring support

Spring supports two types of transaction management:

- Programmatic transaction management: This means that you have managed the transaction with the help of programming. That gives you extreme flexibility, but it is difficult to maintain.
- Declarative transaction management: This means you separate **transaction management from the business code**. You only use annotations or XML based configuration to manage the transactions.

5.8 What are the benefits of the Spring Framework's transaction management?

- It provides a consistent programming model across different transaction APIs such as JTA, JDBC, Hibernate, JPA, and JDO.
- It provides a simpler API for programmatic transaction management than a number of complex transaction APIs such as JTA.
- It supports declarative transaction management.
- It integrates very well with Spring's various data access abstractions.

5.9 Which Transaction management type is more preferable?

Most users of the Spring Framework choose declarative transaction management because it is the option with the least impact on application code, and hence is most consistent with the ideals of a non-invasive lightweight container. Declarative transaction management is preferable over programmatic transaction management though it is less flexible than programmatic transaction management, which allows you to control transactions through your code.

Chapter 6

Do connect with me on LinkedIn : <https://www.linkedin.com/in/kunajun77>

Spring Aspect Oriented Programming (AOP)

6.1 Explain AOP

Aspect-oriented programming, or AOP, is a programming technique that allows programmers to modularize crosscutting concerns, or behavior that cuts across the typical divisions of responsibility, such as logging and transaction management.

6.2 Aspect

The core construct of AOP is the aspect, which encapsulates behaviors affecting multiple classes into reusable modules. It is a module which has a set of APIs providing cross-cutting requirements. For example, a logging module would be called AOP aspect for logging. An application can have any number of aspects depending on the requirement. In Spring AOP, aspects are implemented using regular classes annotated with the `@Aspect` annotation (`@AspectJ` style).

6.3 What is the difference between concern and cross-cutting concern in Spring AOP

The Concern is behavior we want to have in a module of an application. A Concern may be defined as a functionality we want to implement. The cross-cutting concern is a concern which is applicable throughout the application and it affects the entire application. For example, logging, **security** and data transfer are the concerns which are needed in almost every module of an application, hence they are cross-cutting concerns.

6.4 Join point

The join point represents a point in an application where we can plug-in an AOP aspect. It is the actual place in the application where an action will be taken using Spring AOP framework.

6.5 Advice

The advice is the actual action that will be taken either before or after the method execution. This is actual piece of code that is invoked during the program execution by the Spring AOP framework.

Spring aspects can work with five kinds of advice:

- before: Run advice before the a method execution.
- after: Run advice after the a method execution regardless of its outcome.
- after-returning: Run advice after the a method execution only if method completes successfully.
- after-throwing: Run advice after the a method execution only if method exits by throwing an exception.
- around: Run advice before and after the advised method is invoked.

6.6 Pointcut

The pointcut is a set of one or more joinpoints where an advice should be executed. You can specify pointcuts using expressions or patterns.

6.7 What is Introduction?

An Introduction allows us to add new methods or attributes to existing classes.

6.8 What is Target object?

The target object is an object being advised by one or more aspects. It will always be a proxy object. It is also referred to as the advised object.

6.9 What is a Proxy?

A proxy is an object that is created after applying advice to a target object. When you think of client objects the target object and the proxy object are the same.

6.10 What are the different types of AutoProxying?

- BeanNameAutoProxyCreator
- DefaultAdvisorAutoProxyCreator
- Metadata autoproxying

6.11 What is Weaving? What are the different points where weaving can be applied?

Weaving is the process of linking aspects with other application types or objects to create an advised object. Weaving can be done at compile time, at load time, or at runtime.

6.12 Explain XML Schema-based aspect implementation?

In this implementation case, aspects are implemented using regular classes along with XML based configuration.

6.13 Explain annotation-based (@AspectJ based) aspect implementation

This implementation case (@AspectJ based implementation) refers to a style of declaring aspects as regular Java classes annotated with Java 5 annotations.

Chapter 7 : Spring Model View Controller (MVC)

7.1 What is Spring MVC framework?

Spring comes with a **full-featured MVC framework for building web applications**. Although Spring can easily be integrated with other MVC frameworks, such as Struts, Spring's MVC framework uses IoC to provide a clean separation of controller logic from business objects. It also allows to declaratively bind request parameters to business objects.

7.2 DispatcherServlet

The Spring Web MVC framework is designed around a DispatcherServlet that handles all the HTTP requests and responses.

7.3 WebApplicationContext

The WebApplicationContext is an extension of the plain ApplicationContext that has some extra features necessary for web applications. It differs from a normal ApplicationContext in that it is capable of resolving themes, and that it knows which servlet it is associated with.

7.4 What is Controller in Spring MVC framework?

Controllers provide access to the application behavior that you typically define through a service interface. Controllers interpret user input and transform it into a model that is represented to the user by the view. Spring implements a controller in a very abstract way, which enables you to create a wide variety of controllers.

7.5 @Controller annotation

The @Controller annotation indicates that a particular class serves the role of a controller. Spring does not require you to extend any controller base class or reference the Servlet API.

7.6 @RequestMapping annotation

@RequestMapping annotation is used to map a URL to either an entire class or a particular handler method.