Lecture Notes

MVC architecture

Session 1 - Introduction

In this session you have learnt the basic concepts related to networking as follows:

- Client-Server Architecture
- Protocols
- Ports
- IP addresses
- DNS Server
- HTTP Protocol

Further in the session, you learned important concepts related to web applications as follows:

- HTML
- Static and Dynamic websites
- Business Logic and Presentation logic
- Servlet
- JSP

You then created basic web applications using Servlet and JSP. You were introduced to the following tools:

- Maven
- Spring Boot

Based on the observations made while creating the web applications using Servlet and JSP, you have understood the disadvantages of those methods.
Further on, you learnt to create simple web applications using the following three methods:

- Servlet
- JSP
- Spring MVC

Client-Server Architecture

The client and server are programs running on machines that interact with each other to provide you facilities on the internet like rendering the website on a browser.

If you are accessing a website, then:

- A software trying to access some service over the internet is called the client. You need to type the URL of the website on the browser search bar. Hence, in this case, the client program will be your browser since clients are programs that requests data from a server
- Similarly, a software providing a service or data over the internet is called a server. In this case, the server will be a web-server which provides you with the website.

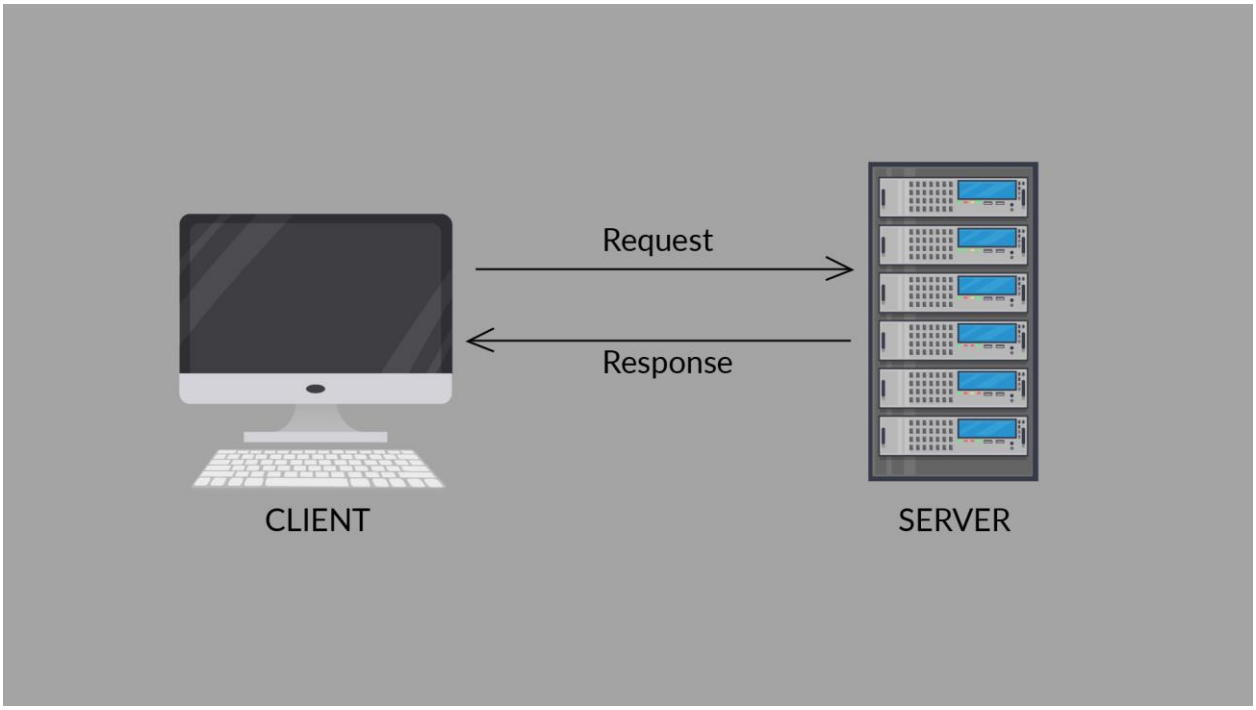

Image 1: Client-Server Architecture

Networking Terminologies

Protocols

Protocol is a set of rules that must be followed by the client and the server while communicating over the internet.

Protocols can be said as the communication language over the internet. They define a way in which information must be communicated over the internet and it is uniform for the millions of clients and servers connected to the internet and makes sure that they understand communication coming from each other. There are different protocols for different services such as email services, instant messaging or database services. When you want to access any website, you would be contacting a web-server. The web server and the client must adhere to the **Hypertext Transfer Protocol** to communicate in this case. HTTP protocol defines a uniform way or language through which every client and web-server over the internet should communicate.

IP Address

Each device connected on the internet or a network following the TCP/IP protocol are provided with a specific address which is called as IP (Internet protocol) address. This IP address will help to uniquely identify devices and help those devices communicate with each other.

IP address generally looks like this:

192.168.3.1

Ports

Many services can be hosted on the same server (i.e. a web server, an email server, and a file server can be hosted on the same computer). Ports are used to identify those specific services.

For example: the post assigned to web services is 80.

DNS Server

DNS servers act as the phonebook of the internet. This server consists of the mapping of different domain names and IP addresses of corresponding servers. When you request for a URL such as www.facebook.com (where Facebook is a domain name), your browser takes the help of a DNS server to route the request to the appropriate web server.

Hypertext Transfer Protocol (HTTP)

HTTP protocol is a protocol that web servers and clients must adhere to while communicating over the internet.

There can be different types of client requests based on the required server operations. These client requests are sent using something called “request methods”.

You have learned that the different HTTP request methods can be used to perform CRUD operations as follows,

- POST - This HTTP method is used to send a request to create new data on the server
- GET - This HTTP method is used to read or retrieve some existing data from the server
- PUT - This HTTP method is used to update some existing data in the server
- DELETE - This HTTP method is used to delete some existing data from the server.

Hypertext Markup Language (HTML)

HTML is a language that is used to create the presentation logic of web applications. It gives the browser instructions such as - this text should be displayed as bold or this text should be displayed as a link.

HTML stands for **Hypertext Markup Language.**
Hypertext simply stands for text that links to other web pages, like the text below:
Click here.

Markup languages are programming languages that are readable by both machines and humans.

HTML uses something called **tags** to define elements of a web page. Almost all HTML tags have an opening and closing tag.
Here are some important HTML tags:

- <!DOCTYPE html> indicates that the file is an HTML file and can be read by a browser.
- <html>-Everything you want to tell your browser about your webpage comes within these two <html> and </html> tags.
- The <head> and </head> tags contain metadata about the webpage. Metadata stands for “data about data”.
- The <title> tag is placed within the head tags and contains the title of the website which would be visible at the title bar.
- <body></body> tags contain the body of your webpage. These tags contain all the information about how your webpage should be including text, images, forms, animation etc.
- The tags indicate that the text written between them is a paragraph.

You created a simple webpage using HTML as follows:

```
<!DOCTYPE html>
<html>
    <head><title>Hello World!</title></head>
    <body>
        <p>Welcome to HTML, Hello World!</p>
    </body>
</html>
```

Understanding Web Applications

Websites can be broadly classified into:

1. **Static websites:** These are basic websites where the content of the web pages remains fixed and the same content is displayed to every user accessing the page.
2. **Dynamic websites:** The content of the web pages changes in real time based on the user viewing the website, or the location from where it is being accessed, and/or other differentiating criteria. For example, the Facebook news feed of a user or a Twitter account is dynamically generated for its users based on their viewing history, usage behaviour, and location.

A web application has to major components:

1. **Business Logic:** The program that intercepts a user request, processes it and generates required data is called **business logic.**
2. **Presentation Logic:** The program that renders all the assembled information into a webpage (or generally an HTML page) is called **presentation logic.**

In our case, the business logic will be implemented by Java code and the presentation logic will be implemented by HTML code.

Dynamic Website Creation

There are two popular traditional approaches to create dynamic websites:

1. **Servlet**
2. **JSP**

Servlet

Servlet is a Java class which responds to client requests and can be used to generate dynamic websites. A servlet is a program that will:

- Receive a request. (for example, a request for your list of suggested friends on Facebook)
- Processes the request and conduct business logic (Does all the processing required to find friend suggestions for you) and
- Create the presentation logic of the website based on the business logic executed in the step above, i.e. returns the list of friends as HTML.

Servlets have HTML code within JAVA code to dynamically generate web pages.
Servlet is basically a Java class that responds to HTTP requests. Using servlets spares you the work of writing the code for HTTP requests, packets and worry about the networking aspects of the code. All the worries about adhering to the HTTP protocol and the TCP/IP protocols can be left to the servlet, you can just specify the HTTP request method that you need to use and not worry about anything more.

Servlet Container

Different servlet programs can serve different purposes. Consider the example of your Facebook homepage again, a servlet class can be assigned to pick advertisements for you while another one will create and render your notifications.
A request hence firsts goes to a **servlet container**. The servlet container loads the appropriate servlet and sends the request to the servlet for processing.

You saw the example of ebooksforyou.com as follows. You learned how a server would receive a request, process it via servlets and return the response to the client.

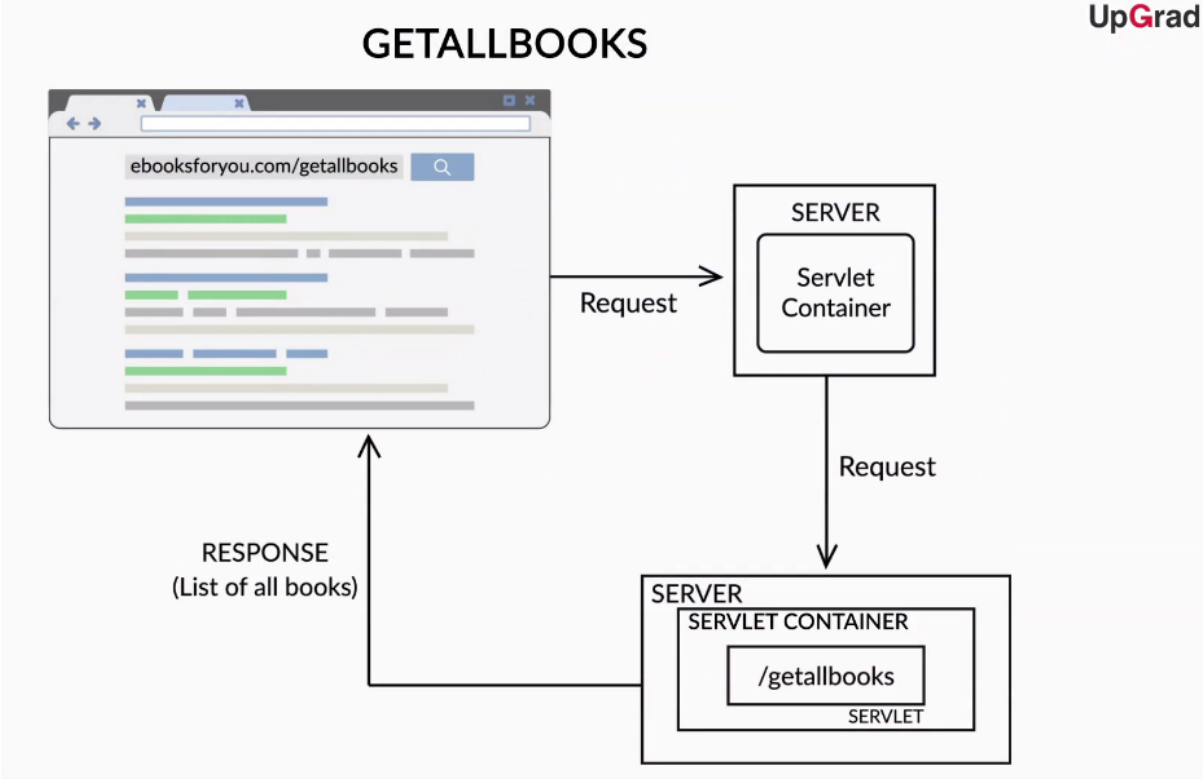

Image 2: Servlet

JSP

Java server pages (JSPs) are another way to build dynamic websites or web applications. This was introduced to ease the life of developers. It decouples the presentation logic from the business logic of generating a dynamic website. Thus, allowing the front-end developers to work separately from back-end developers.

A JSP has Java code within HTML code to dynamically generate webpages.

Maven and Spring Boot

Maven

Maven is a project management tool which will do the following for you:

- Install the required third-party library for your Java project
- Resolve dependencies- Maven will download all pre-written Java code, such as third-party libraries, that your application will use. For example, in this case, you would not want to write the code for a servlet container. Instead, you will add a dependency and Maven will download the code for the servlet container. These dependencies are present as JARs or Java Archives.
- Manage all the compiled java files as Jars (Java archives)
- Provides a pre-defined project structure template for developing web applications. You can observe the project structure in the upcoming video while creating a web application.

You must specify the following when creating a project using Maven

- **Group ID** is the team or organization that you work for. This name would be unique throughout the organization you work at.
- **Artifact ID** is the name of the project.
- **Version** simply denotes the version of the application. For example, you can also have upgraded applications or change the architecture some time down the line.

POM.xml
Maven resolves dependencies via a **POM.xml** file. This is a file where the developer, i.e. you will mention all the other code and libraries that your program will be dependent upon.
Dependencies are added via mentioning name, group ID, version and artifact ID of the dependencies in the POM file via XML.

Spring Boot
Spring Boot is simply an application development framework which simplifies the application development process by providing required libraries and default configurations. Spring Boot is used here to ease the development of web applications while using Servlet, JSP and Spring MVC. Spring Boot also provides the Tomcat server and servlet container to host the website.

Web Application Using Java Servlet

You created a web application using Spring Boot which simply hosts a webpage that say “Hello World”.

To achieve this, you did the following:

- You created a project using Maven and IntelliJ.
- You mentioned the dependencies you require in the POM.xml file.
- You wrote the required Java classes to create the web application:
    - HelloWorldApplication
    - HelloWorldServlet

The dependencies added in the POM file are as follows:

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.0.5.RELEASE</version>
</parent>

<dependencies>

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

</dependencies>
```

The details of each dependencies in the POM file are as follows:

- **Spring-boot-starter-parent:** A **parent POM** file is another POM file that a child POM can inherit. It has some default configurations required to use Spring Boot to develop any kind of an application. It also controls versions of dependencies added. So, to add any Spring Boot dependencies, you need to add the starter parent to our POM file. You can go and read more about what the Spring Boot parent POM does in the section 13.2 from the Spring documentation here.
- **Spring-boot-starter-web:** Spring Boot can be used to build various kinds of applications. In our case, we are trying to build a web application and hence you need to add this dependency to download all JARs required to build a web application using Spring Boot.

You need to create an application class to do the following:

- Start up the server and get your application up and running.
- To run application using the Tomcat server on port 8080. This means that all the functionalities that your application provides would be accessible to the user through the 8080 port of the server the application is running on.

Points to note about the HelloWorldApplication class:

- You need to add the @SpringBootApplication annotation here to indicate that this is an application class and you are using Spring Boot here to create a web application.
- You need to add @ServletComponentScan annotation in the HelloWorldApplication application class. This annotation tells your application class to look for the servlet class you have created annotated with @WebServlet.
- The SpringApplication.run () statement runs your application on the Tomcat server on the default port 8080.

The Java class is as follows:

```
package upgrad;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.web.servlet.ServletComponentScan;
import org.springframework.context.annotation.ComponentScan;

@SpringBootApplication
@ServletComponentScan
public class HelloWorldApplication {

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

}
```

We created another Java class called the HelloWorldServlet.
This class will process the client request for the hello world page.
In this class:

- The @WebServlet annotation denotes that this class is a Java web servlet.
- The servlet container invokes the service method of the appropriate servlet class based on the request. Hence, we have created the service method here.
- Earlier, this class was annotated with @ResponseBody to return a plain text response instead of HTML. This annotation denotes that the object returned by this method is to be returned to the user as HTTP response.

The servlet class is as follows:

```
package upgrad;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.PrintWriter;

@WebServlet("/helloworld")
public class HelloWorldServlet extends HttpServlet {

   public void service(HttpServletRequest request, HttpServletResponse response) throws IOException, ServletException {
      PrintWriter printWriter = response.getWriter();
     printWriter.print("<!Doctype html>");
      printWriter.print("<html>");
      printWriter.print("<head>");
      printWriter.print("<title>Hello World Servlet!</title>");
      printWriter.print("</head>");
      printWriter.print("<body>");
      printWriter.print("<p>Hello World!</p>");
      printWriter.print("</body>");
```

```
        printWriter.print("</html>");
        printWriter.close();
    }

}
```

Web Application using JSP

To implement support for JSP while using the Tomcat server, you must add the following dependency to the POM.xml file:

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

You need to create a helloworld.jsp file in main→ webapp as follows:

```
<!Doctype html>

<html>
  <head>
    <title>Hello World JSP!</title>
  </head>
  <body>
    <p>Hello World. This is your first JSP!</p>
    Current time: <%= new java.util.Date() %>
  </body>
</html>
```

Important things to note about the JSP file:

- This JSP file has a Java statement in it: **new java.util.Date()**
- You can access your website by going to the following URL: **localhost:8080/helloworld.jsp**

Servlet vs JSP

- By using JSP as the technology to build web application, work can be separated between a person responsible for developing the HTML code or the front-end and the person developing the dynamic components or the back-end.
- It decouples implementation of presentation logic (HTML code) and business logic (Java code), which makes the web application more robust and easier to maintain.

Introduction to MVC architecture

Drawbacks of using servlets/JSP

One of the major drawbacks of Servlets and JSPs is the separation of concerns.
With servlets, a front-end developer and a back-end developer would have to work together to develop the servlet classes.

Even with JSPs, a front-end or HTML developer would be worried about adding Java scriptlets to the HTML code. This would mean that an HTML developer would have to deal with Java code often and a Java developer would have to deal with HTML code often. Also, JSPs are compiled to create servlets. Due to this process, debugging becomes harder and processing is also slower.

Model-View-Controller Architecture
The MVC architecture addresses the problem of separation of concerns. It separates an application into components called model, view and controller:

- **Model:** Model represents the data layer of the application. The data is stored in the form of model object and passed to the view by the controller.
- **View:** This is the presentation layer of the application. **View** component of the web application would take information from the controller (in the form of a model) and render it on the web page to be displayed to the user.
- **Controller:** This is the business logic of the application. It involves classes and methods that process the business logic for you. Controller receives a user request, processes it and uses the model and view components to send the appropriate response to the user.

You also saw the same "ebooksforyou" example and what happens when we implement the same functionality using the MVC architecture as follows,

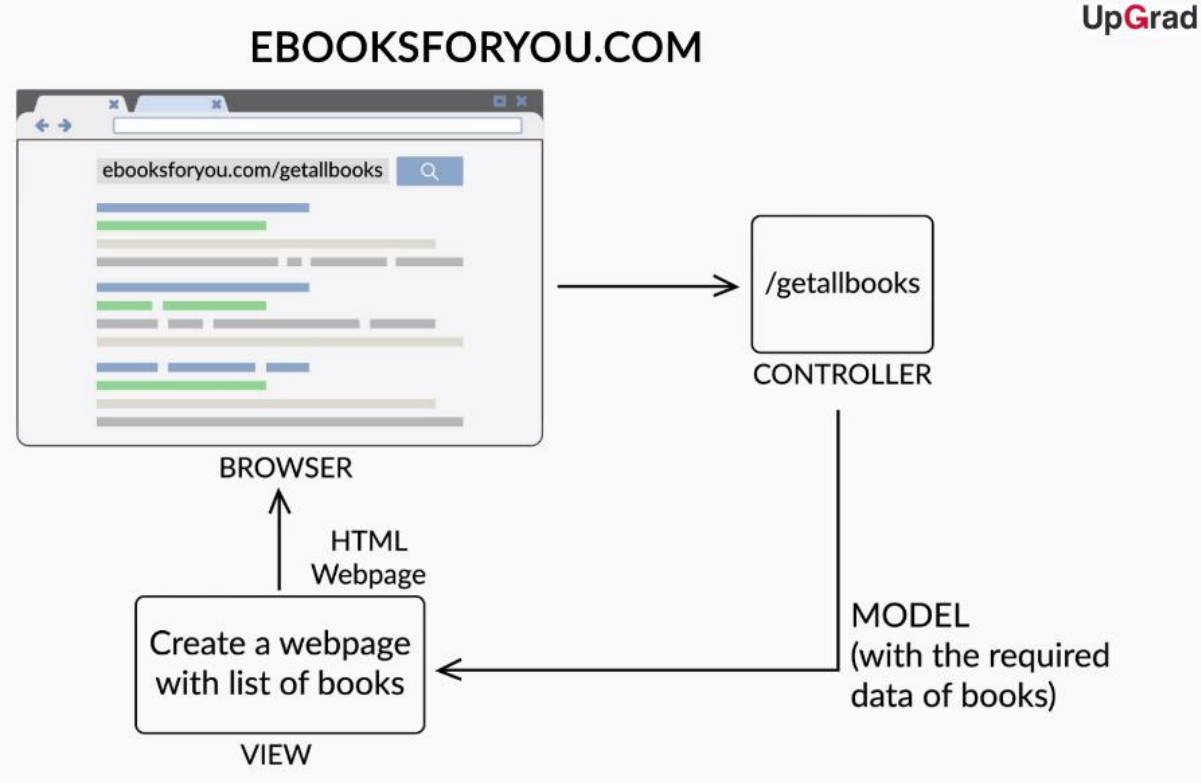

Image 3: MVC architecture

Introduction to Spring MVC

Spring MVC implements the MVC architecture through the **dispatcher servlet, handler mapping and view resolver.**
Overall, the following components are involved in creating a web application using Spring MVC:

- **Dispatcher Servlet**- This acts as a front controller. This means that a request coming to the server is first intercepted and responded to by the dispatcher servlet. The dispatcher servlet acts as a point of communication between all other components.
- **Handler Mapping**- When a request comes to the dispatcher servlet, the dispatcher servlet contacts the handler mapping to get the name of the controller to which the request should be routed. Handler mapping finds the right controller to be contacted and sends that information to the dispatcher servlet.
- **Controller**- Controller is contacted by the dispatcher servlet to process a request. A controller is responsible for processing the client request, retrieves the required data or the model, and send this model along with the view name that the model must be rendered to the dispatcher servlet.

- **Model**- The model represents the data layer of the website. All the required data is stored in the database and sent from one component to another in the form of models. Models are basically Java classes with specific attributes. For example: A model class storing data about a student in a school would have attributes like name, ID number, date of birth etc. The model layer also implements functionalities for the controller to retrieve and manipulate data in the database.
- **View Resolver**- View resolver receives view name from the controller via the dispatcher servlet. It has the mapping of view names and views. View resolver finds the appropriate view and sends it to the dispatcher servlet. For example: The view name could be helloworld and the view could be helloworld.jsp. The view resolver would receive helloworld and return the view helloworld.jsp to the dispatcher servlet.
- **View**- The view receives the business logic and/or model data from the controller via the dispatcher servlet and builds the HTML around the data that it receives. The view will build the final web page that a client will be able to see. It sends the HTML code back to the dispatcher servlet which then sends the same to the client as response.

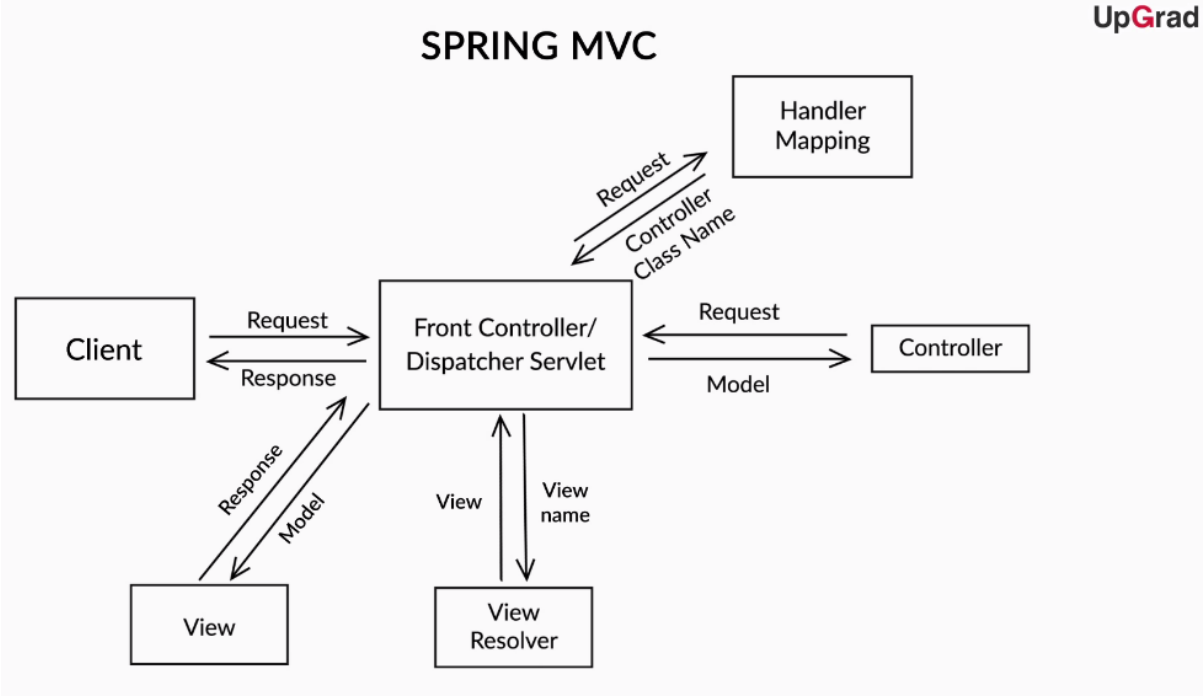

Image 4: Spring MVC

We have used Spring Boot with Spring MVC throughout this module. Following are some reasons for doing so:

- Without Spring Boot, you would have to go and configure the dispatcher servlet using an XML file and mention names and mappings of controllers etc.
- Spring Boot comes packaged with the tomcat server. This means that you don't have to download the Tomcat server program, configure it and integrate it separately with your application.
- Spring Boot uses annotations like the @SpringBootApplication annotation and bypasses a lot of xml configurations that you might have to write to integrate different programs with your application like database services.

Hello World Application, using Spring MVC

To create a simple web application using Spring MVC to host a basic HTML page, you did the following:

- Create a controller class
- Add the dependency for Thymeleaf library
- Create an html file which will be hosted via the application

Thymeleaf library provides you with a view resolver. In order to use Thymeleaf, you need to add the following dependency to the POM file:

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

Important things to note about the controller class:

- @Controller annotation tells Spring that the class is a controller class.
- @RequestMapping annotation is used to tell Spring (dispatcher servlet and handler mapping classes) that the controller method is mapped to a specific URL. In this case, it is mapped to “/hellospring”. (Similar to how you mapped the servlet to a specific request using the @WebServlet annotation, i.e. @WebServlet(“/helloworld”))

The final controller class is as follows:

```
package upgrad;

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

@Controller
public class HelloWorldController {

  @RequestMapping("/hellospring")

   public String helloSpring() {
      return “index”;
   }

}
```

MVC Architecture-Session 2

Introduction

In this session, you learnt to develop a blogging application having the following features:

- Login
- Registration
- Create blog post

You have also learned about some important concepts as follows:

- Using Thymeleaf to build dynamic webpages.
- Dependency Injection
- Best practices while developing a web application

Technical Blog Application-Landing page
Step 1 – First model

The first thing you need to do is to host a landing page via your application. The landing page shows a list of posts returned by the controller as a model.
To accomplish this, you need to do the following:

- Create a Post model class with required attributes.
- Create a controller class which adds a list of posts to a Spring Model object and returns the appropriate view name.

The following are the attributes defined for a blog post:

- ID - This variable helps to identify the specific post
- Title - This variable is to store the title of the post
- Body - This variable is to store the content of the post
- Date (date of creation) - This variable is to store the date on which the blog post is created.

The Post model class should be as follows:

```
public class Post {

    private String title;
    private String body;
    private Date date;

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getBody() {
        return body;
    }

    public void setBody(String body) {
        this.body = body;
    }

    public Date getDate() {
        return date;
    }

    public void setDate(Date date) {
        this.date = date;
    }

}
```

To host the landing page, you need to create a class called HomeController.
This class has a method called getAllPosts() which does the following for now:

- Fetch all the blog posts available in the technical blog web application.
- Store the list of posts as a Spring Model object.
- Return the Spring Model object which has the list of posts.

The controller class created is as follows:

```
@Controller
public class HomeController {

    @RequestMapping("/")
    public String getAllPosts(Model model) {

        ArrayList<Post> posts = new ArrayList<>();

        Post post1 = new Post();
        post1.setTitle("Post 1");
        post1.setBody("Post Body 1");
        post1.setDate(new Date());

        Post post2 = new Post();
        post2.setTitle("Post 2");
        post2.setBody("Post Body 2");
        post2.setDate(new Date());

        Post post3 = new Post();
        post3.setTitle("Post 3");
        post3.setBody("Post Body 3");
        post3.setDate(new Date());

        posts.add(post1);
        posts.add(post2);
        posts.add(post3);

        model.addAttribute("posts", posts);

        return "index";

    }

}
```

Spring Model Object

The MVC model class (like the Post model) is used to represent data in the application whereas the Model class provided by Spring is used to pass data between the controller and the view.

Model is similar to key-value store such in a HashTable data structure. You can add data to a Model object by calling the .addAttribute(key, value) method.

The model object is created within request scope. You have also come across three different possible scopes:

- **Request Scope:** Information which is generated and used within the context of a request is said to be in the request scope. i.e., request scope lies within the server receiving an HTTP request from a client and the server sending an HTTP response to the client. For example, information passed to the controller as part of a request is typically kept during the request scope. Some examples are information in the http request header, query parameters, etc.
- **Session Scope:** Information is always available to a user and not to any other user is said to be in the session scope. For example: you may want to keep the information about the current logged in user until the user logs out or closes his or her browser window.
- **Application Scope:** Information available to every user is said to be in the application scope. i.e. Application scope starts from the point you start the web application till the time you shut it down. For example, the number of users that are currently viewing the technical blog.

Technical Blog Application-Landing page
Step 2: Presentation Logic

We are using Thymeleaf templating engine as the view part of the MVC architecture in developing the Technical blog web application.
Thymeleaf will help you with the following:

- It will help you in creating views or dynamic websites:
    - Using Thymeleaf, you can take the model data returned by the controller class and render it in an HTML webpage.
    - You can set standard headers and footers to different webpages of your application. Thymeleaf will help in reusing the HTML code written for the headers and footers.
- Thymeleaf will also provide you with a view resolver. It will take a view name from the controller and find the appropriate view mapped to the view name.

The controller is returning the string “index”. The view resolver will take this view name and will find the corresponding view. The index.html is the view for the landing page.
The index.html file is as follows:

```
<!Doctype html>

<html xmlns:th="http://thymeleaf.org">
   <head>
      <title>Spring MVC Application</title>
   </head>
   <body>
      <h1>All Technical Posts</h1>
      <main id="posts">
         <post th:each="p : ${posts}">
            <h2 th:text="${p.title}"></h2>
            <i>Posted On: </i> <span th:text="${p.date}"></span>
            <div>
               <p th:text="${p.body}"></p>
            </div>
         </post>
      </main>
   </body>
</html>
```

The “th” namespace tells thymeleaf that the proceeding code is Thymeleaf specific code.
Here are some important concepts that you learned through the above HTML/Thymeleaf code:

- You have come across some new HTML tags like the <h1> and the tag.
    - The text within <h1></h1> tags is rendered as bold and large. The “h” denotes heading text here.
    - The tags contains text to be rendered as separate paragraphs.
    - The tags contain text to be rendered in an italic fashion like *this.*
    - The tag defines a division or a section in the webpage. Each post is contained within tags.
- The following code helps to take the model from the controller,
    - Obtain the posts data stored in the model
    - Iterate over the list of posts and display them on the web page.

```
<main id="posts">
        <post th:each="p : ${posts}">
          <h2 th:text="${p.title}"></h2>
          <i>Posted On: </i> <span th:text="${p.date}"></span>
          <div>
            <p th:text="${p.body}"></p>
          </div>
        </post>
      </main>
```

- Here, the main tag indicates that the main content of the webpage is within these tags.
- The post tag is a dummy container which does not really signify anything to HTML or Thymeleaf. This is just to make the code more readable.
- The code that really matters is **“th:each="p : ${posts}"”** - This instruction iterates over the list of posts received from the HomeController.
    - th:each acts like a for loop to iterate over the arraylist of posts received from the controller
    - ${posts} represents the array list of posts stored under the "posts" key in the Model returned by the controller.
    - p : ${posts}" means that, for each post that we are looping through in the arraylist (the posts data stored under the "post" key in the model), let's store the post in the current loop iteration under a variable called p.
- **** - This instruction takes the date from the post object (that is currently referenced by the variable p) and renders it on the page
- Similarly **“ ”** renders the post body on the web page as text.

Technical Blog Application-Landing page
Step 3: Creating the service class and dependency injection

As a best practice, a class is supposed to have only one responsibility. Here, the controller class should only be responsible for processing a user request, retrieving required data and returning the same with the appropriate view name. Hence, to address CRUD functionalities related to blog posts, we created a new Java class called PostService.

PostService takes care of all CRUD operations related to posts like retrieving all posts through the getAllPosts() method.

The PostService class is annotated with @Service. It registers the class in the Spring IOC container, and tells Spring that this class is a service class that provides business logic in the application.

The PostService class is as follows:

```
@Service
public class PostService {

    public ArrayList<Post> getAllPosts() {
        ArrayList<Post> posts = new ArrayList<>();

        Post post1 = new Post();
        post1.setTitle("Post 1");
        post1.setBody("Post Body 1");
        post1.setDate(new Date());

        Post post2 = new Post();
        post2.setTitle("Post 2");
        post2.setBody("Post Body 2");
```

```
        post2.setDate(new Date());

        Post post3 = new Post();
        post3.setTitle("Post 3");
        post3.setBody("Post Body 3");
        post3.setDate(new Date());

        posts.add(post1);
        posts.add(post2);
        posts.add(post3);

        return posts;
    }
}
```

Dependency Injection

For the controller to use a method from the service class, you need to create an instance of PostService class in the HomeController class using the new keyword.

Since you are hardcoding a dependency, this creates tight coupling between classes. Spring framework provides you with a way to make the classes loosely coupled via a technique called dependency injection. In the Spring IOC container, all component classes are registered and single object instances of all class instances are created to be used throughout the application. You can ask the Spring container for this object instance whenever required using the @Autowired annotation.

Dependency injection helps you with the following:

- Reducing the coupling between the classes and make your project code more loosely coupled. You can get an object of a component class using the @Autowired annotation.
- Dependency injection will make testing and adding mock objects easier.
- The lifecycle of object instances obtained from Spring via the @Autowired annotation is managed by Spring IOC container.

Object Lifecycle Management

An object goes through three states:

- Creation
- Management
- Destruction

The Spring IOC container manages this lifecycle of all registered objects:

- It creates an object instance when the application boots up
- It manages the object instance. It maintains an object instance in one of two states:
    - Active: The object is consuming memory resources.
    - Passive: Spring maintains passive instances of all objects as pointers to assigned memory locations until an object is required.
- The object instance is destroyed (erased from application memory) when the application is shut down.

Technical Blog Application-Landing page
Step 4: Login feature

For the implementation of login feature, you need to create the following Java classes:

- User model- It will store User information. It has two attributes: username and password.
- User controller-It will address business logic related to user related functionalities such as login and logout.
- Post Controller-To address post related functionalities like returning the posts.html page.
- UserService- To authenticate a user and register a user (addresses CRUD operations related to users)

The following HTML pages are required to implement the presentation logic:

- Layout.html- It consists of a set of reusable HTML code such as the headers for logged in and logged out users.
- Login.html- This is the page a user will see when they try to log in.
- You modified index.html to add fragments from layout.html. You added the logged-out fragment to index.html so that the user can see options to login and register
- posts.html - This is the page that a user will be directed to after logging in.

User Model

User model has the following attributes:

- Username- to store a unique username
- Password-to store the account password

The User model should look as follows:

```
public class User {

    private String username;
    private String password;

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }
}
```

User Controller

The User Controller class has methods for the following:

- Addressing user request for login and registration pages.
- Addressing the request from a user to logout.
- Addressing a user request to submit credentials from the login web page.

The UserController class is as follows:

```
@Controller
public class UserController {

    @Autowired
    private PostService postService;
    @Autowired
    private UserService userService;


    @RequestMapping("users/login")
    public String login() {
        return "users/login";
    }

    @RequestMapping("users/registration")
    public String registration() {
        return "users/registration";
    }
```

```
    @RequestMapping(value = "users/login", method=RequestMethod.POST)
    public String loginUser(User user) {
        if(userService.login(user)) {
            return "redirect:/posts";
        }
        else {
            return "users/login";
        }
    }

    @RequestMapping(value = "users/logout", method=RequestMethod.POST)
    public String logout(Model model) {
        ArrayList<Post> posts = postService.getAllPosts();

        model.addAttribute("posts", posts);
        return "index";
    }

}
```

login() method
This method is invoked when a user requests for the login page via the URL, users/login. It only returns a view name of the login page. Since the login.html page is in the users folder under templates, the method returns “users/login”

registration () method
This method returns the registration webpage to the user which is to be created later. The registration page will also be under the users folder under templates.

loginUser() method
This method currently listens to POST requests on users/login and is invoked when a user clicks on the submit button on the login page. Since a client is trying to send some data to the server, POST request method is to be used here.
This method uses login method from the UserService class to authenticate a user. In case of a successful login, it redirects a user to the posts.html page which will be the homepage of a user. Whereas, in case of an unsuccessful login, it redirects a user back to the login page.

logout() method
This method is invoked when a logged in user clicks on the logout button. This method will redirect a user back to the landing page. Since, the index.html is expecting data from the back-end, this method also assembles a list of posts via Posts service and adds them to the Spring Model object.

PostController
This controller class will address requests related to CRUD operations on posts. Later, it will have methods for addressing user requests for creating a post. Currently, this class has one method which handles the request for the posts.html page which will be the homepage for a user.

The PostController class is as follows:

```
@Controller
public class PostController {

    @Autowired
    private PostService postService;

    @RequestMapping("posts")
    public String getUserPosts(Model model) {
        ArrayList<Post> posts = postService.getOnePost();
        model.addAttribute("posts", posts);
        return "posts";
    }

}
```

This class uses the PostService class to get only one post to be shown on the posts.html page.
The corresponding PostService method is as follows:

```
public ArrayList<Post> getOnePost() {
        ArrayList<Post> posts = new ArrayList<>();

        Post post1 = new Post();
        post1.setTitle("This is your Post");
        post1.setBody("This is your Post. It has some valid content");
        post1.setDate(new Date());
        posts.add(post1);

        return posts;

    }
```

User Service

The UserService class currently has just one method with hardcoded user credentials to implement authentication. The UserService class is as follows:

```
@Service
public class UserService {

    public boolean login(User user) {
        if(user.getUsername().equals("validuser")) {
            return true;
        }
        else {
            return false;
        }
    }

}
```

layout.html

The layout.html file has two header fragments:

- logged-in
- logged-out

The logged-out fragment consists of the hyperlinks of login and registration and this hyperlinks must be provided on the landing page(index.html).

The logged in fragment will have an option for a user to log out.

The layout.html file is as follows:

```
<!Doctype html>

<html xmlns:th="http://thymeleaf.org">
   <head th:fragment="site-head"></head>

   <body>
      <header th:fragment="logged-out" th:remove="tag">
         <header>
            <a href="users/login.html" th:href="@{/users/login}">Login</a>
            <a href="users/registration.html" th:href="@{/users/registration}">Registration</a>
         </header>
      </header>

      <header th:fragment="logged-in" th:remove="tag">
         <header>
            <form method="post" th:action="@{/users/logout}">
               <input type="submit" value="Logout"/>
            </form>
         </header>
      </header>
   </body>

</html>
```

You learned about some important components while writing the layout.html file:

- The text within the <a> tags, called the anchor tag, is available as a hyperlink to the user. When clicked, this text would redirect to the link provided after href.
- The th:href keywords tells thymeleaf that the text here is to be linked with the given URL. The given URL path which is “users/login” is with reference to the current web application and when clicked, the request would be processed by the controller class and controller method mapped to the given URL in the web application.
- “**th:fragment=”logged in**”” indicates that this is a piece of HTML code named “logged-in” that can be used by other HTML files. This is not an complete HTML file in itself but just a reusable piece of HTML code.

index.html

Since this is the landing page of the website, it should have options to log-in and register. Hence, we have added the logged-out fragment to this page.

The replace keyword indicates that a fragment from layout.html is being referred in the index.html file. You need to add the following instruction in the index.html file:

```
<header th:replace="layout :: logged-out"></header>
```

login.html
This is a web page with a simple form which a user must fill to login. A user must provide the username and password here to log in.
The HTML code in the login.html file should be as follows:

```
<!Doctype html>

<html xmlns:th="http://thymeleaf.org">

   <head th:replace="layout :: site-head">
      <title>Spring MVC Application - Login</title>
   </head>

   <header th:replace="layout :: logged-out"></header>

   <h2>Please Login:</h2>

   <form method="post" th:object="${User}">
      <label for="username">Username:</label>
      <input type="text" id="username"/>
      <label for="password">Password:</label>
      <input type="password" id="password"/>
      <input type="submit" value="Login"/>
   </form>

</html>
```

Some important points to note here are:

- The data is to be sent via the POST method.
- You have come across the HTML tags for creating the form such as labels, buttons etc.
- You will be learning more about HTML forms in detail in the upcoming courses. For now, you only need to understand that each field has to be mapped to the correct name and label to make sure that you get your data in the right format.

posts.html
This is a page which a user will be redirected to after logging in.
Since a logged-in user is redirected to this page, the logged-in header fragment from layout.html is added to this page.

```
<!Doctype html>

<html xmlns:th="http://thymeleaf.org">

   <head th:replace="layout :: site-head">
      <title>Spring MVC Application - User Posts</title>
   </head>

   <header th:replace="layout :: logged-in"></header>

   <body>
      <h1>Welcome User. These are your posts</h1>
      <main id="posts">
         <post th:each="p : ${posts}">
            <h2 th:text="${p.title}"></h2>
            <i>Posted On: </i> <span th:text="${p.date}"></span>
```

```
            <div>
                <p th:text="${p.body}"></p>
            </div>
        </post>
    </main>
  </body>
</html>
```

Technical Blog Application-Landing page
Step 5: Registration feature

To implement the business logic for the registration feature, you need to do the following:

- You need to implement required methods in UserController
- You need to create a registration.html page to be shown to a user at request.

Two methods need to be added to the UserController:

- Registration method which provides the user with the registration.html page when requested. It basically listens for GET requests for the URL mapping “users/registration”.
- RegisterUser method takes the user information entered in the registration form. This method basically listens to POST requests from the users/registration URL.

The required methods added in the UserController class are as follows:

```
@RequestMapping("users/registration")
   public String registration() {
      return "users/registration";
   }
@RequestMapping(value = "users/registration", method=RequestMethod.POST)
   public String registerUser(User user) {
      return "users/login";
   }
```

registration() method

This method responds to the client’s request for the registration page. It returns the registration web page to the client.

registerUser() method

This method listens for POST requests on the URL “users/registration”. This method is invoked when a user submits information from the registration webpage.

The corresponding registration.html file is as follows:

```
<!Doctype html>

<html xmlns:th="http://thymeleaf.org">

   <head th:replace="layout :: site-head">
      <title>Spring MVC Application - User Registration</title>
   </head>

   <header th:replace="layout :: logged-out"></header>

   <h2>Please Register:</h2>
```

```
    <form method="post" th:object="${User}">
        <label for="username">Username:</label>
        <input type="text" id="username" name="username"/>
        <label for="password">Password:</label>
        <input type="password" id="password" name="password"/>
        <label for="fullname">Full Name:</label>
        <input type="text" id="fullname" name="fullName"/>
        <input type="submit" value="Register"/>
    </form>

</html>
```

Technical Blog Application-Landing page
Step 6: Create Blog Post

To implement this feature, you need to do the following:

- You need to add corresponding methods in the PostController class (since it addresses CRUD requests on blog posts)
- You need to add a corresponding create.html page

The required methods in PostController are as follows:

```
@RequestMapping("/posts/newpost")
    public String newPost() {
        return "posts/create";
    }

    @RequestMapping(value = "/posts/create", method = RequestMethod.POST)
    public String createPost(Post newPost) {
        postService.createPost(newPost);
        return "redirect:/posts";
    }
```

newPost() method:
This method responds to the user request for the create blog post webpage.

createPost() method:
This method listens to POST requests on /posts/create URL. It calls the PostService method to create store the blog post sent by the client via the HTTP POST request.
The corresponding PostService method currently does nothing since we are not storing the posts anywhere yet.

The corresponding create.html file is as follows:

```
<!Doctype html>

<html xmlns:th="http://thymeleaf.org">

<head th:replace="layout :: site-head">
    <title>Spring MVC Application - Create Post</title>
</head>
```

```
<header th:replace="layout :: logged-in"></header>

<body>
   <h1>Create New Post</h1>

   <form method="post" th:action="@{/posts/create}">
      <div>Post Title:</div>
      <div><input type="text" id="title" name="title" /></div>

      <div>Description:</div>

      <div><textarea rows="7" cols="100" name="body"></textarea></div>

      <div>
         <input type="submit" value="Submit"/>
      </div>
   </form>

</body>
</html>
```

With this, you are only left with integrating your application with the database. In the next module, you will develop features to store the data on users and blog posts as well as develop some more interesting features such as editing and deleting a blog post.