

Spring Boot Microservices

Chapter I: Getting Started

1.0 Objective:

- Overview.
- The Spring Framework.
- Features of Spring Boot.
- Setting Up Apache Maven.
- Building a Spring Boot application.
- Running the application of embedded Tomcat server.
- Generating and Deploying external WAR file.
- Using Spring Initializr.
- Performing Message Internationalization (different languages).
- Using Spring Tools Suite.
- Configuring Server port.
- Configuring user defined properties.

1.1 Overview

An introduction to Spring Boot framework and basic design principle of IoC which is achieved in Spring using Dependency Injection. We will see how easy it is to set up new Spring Boot applications, how to make use of starter templates, parent POM and auto-configurations allow Spring Boot apps to work. As we wind up the course, we will be able to run an embedded web server in a simple Spring Boot app, setup a new Spring Boot project using Spring Initializr and Spring tools suite, and configure properties including user defined properties for the Spring Boot application.

1.2 The Spring Framework

It is a Java framework that provides a convenient way of integrating disparate components into a working, enterprise-grade app, it relies heavily on IoC to achieve it.

Managing dependencies is vital for an app and Design Patterns are a best form of maintaining codes and managing their dependencies, while working as a team many members may fail to follow the same this is where Spring comes to play, it is made simple using IoC achieved by Dependency Injection.

Spring framework refers to an entire family of projects that support a wide range of application scenarios, such as: large enterprise applications, simple cloud-hosted servers, standalone batch and integration workloads.

When using the Spring Framework, we can configure components of the application using either Java annotations or XML configurations. A component simply has to declare its dependencies and Spring framework will take care of its access to its dependencies.

There are a large number of Spring Modules available that allow us to work with Spring where we right very simple methods to perform very complex tasks for example: to connect to a DB and

perform DB transactions Spring allows to make use of simple Java methods without the need to deal with transaction APIs, Spring makes it easy to expose HTTP endpoints from its app that can respond to HTTP request without the need to deal with servlet APIs, Spring allows us to configure message handlers that can send and receive messages between components in the app without having to deal with Java Messaging Service APIs directly, and while working with Spring framework we get monitoring support without dealing directly with JMX APIs.

Let's see how exactly the Spring framework works:



When working with Spring you're typically writing your code in business objects (POJO classes), these business objects will be managed by the Spring container. In addition you might specify additional configuration metadata for your object(via Java annotations or xml configurations). This configuration metadata will be applied to the business objects and in the end you'll get a fully configured system that is ready to use.

The core fundamental building block of the Spring is Spring Bean. Spring Beans are objects that form the backbone of your spring application, these can either be user defined beans or beans that are available as a part of the Spring framework. The basic feature of the Bean is that they are completely managed by the Spring Framework, their entire life cycle starting from the creation, the way they work and deletion is handled by the Spring. Spring Bean performs the core feature of IoC.

IoC - A process by which objects define their dependencies and an external container inject those dependencies into the object. Any Spring Bean only has to state what its dependencies are and are not responsible for creating the instances of these objects, an external container will take care of this.

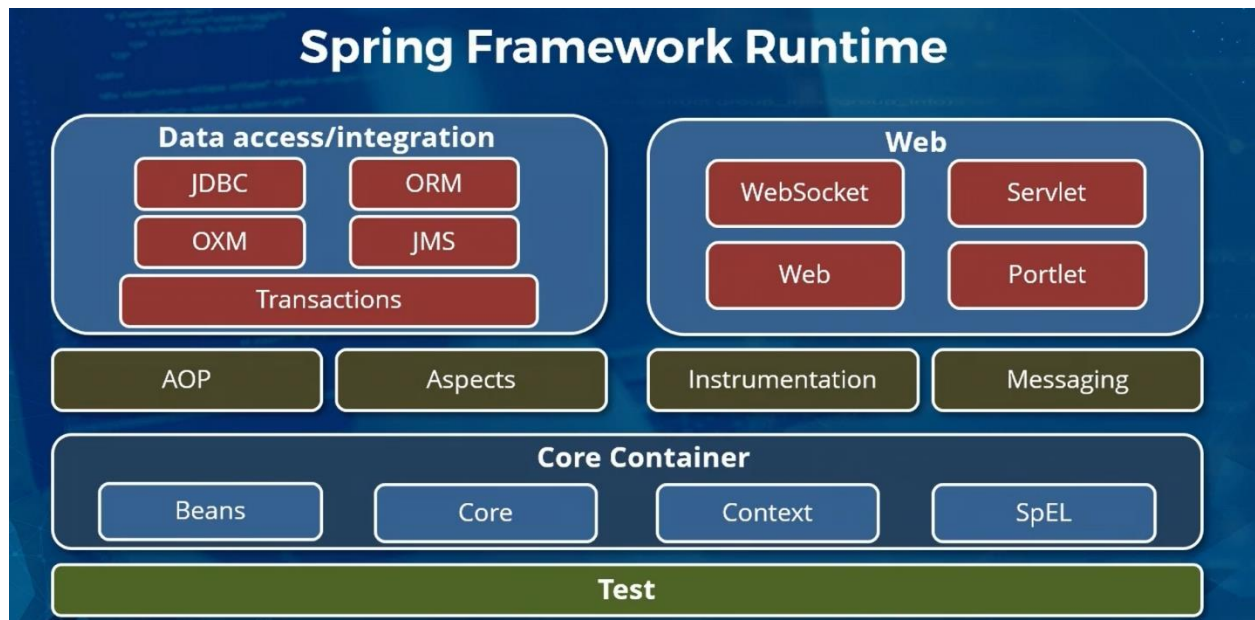
As a developer when you create your own Spring Bean components, you only need to declare what your components are dependent on. For Dependency Injection and IoC, Spring depends on these 4 components: Bean(client), Service, Injector, Injection.

Bean - An object that depends on the behavior of another object, other beans can depend on it.

Service - An object that the client bean depends upon.

Injector - Code that tells client bean which service to use.

Injection - Process of telling client bean which service to use.



Spring Framework Modules:

Core Container	Data Access
Messaging	Web
AOP & Instrumentation	Test

1.3 Features of Spring Boot

Spring Boot is not an entirely new framework it is an extension of the Spring Framework. It simplifies Spring web development by automatically pre-configuring virtually all third-party libraries by default.

How Spring Boot simplifies Spring

- Auto-Configuration (Configures automatically based on the JAR files added)
- Parent pom.xml (Instead of explicitly defining the dependencies required we use Parent POM)
- Opinionated Approach (Intentionally reduces flexibility by making choices on behalf of Dev)

Spring Boot starter dependency templates:

Built-in dependency descriptors that can be included in a Spring Boot app, will pull in and manage all dependencies including transitive dependencies.

Spring Boot: Starter Dependencies examples

spring-boot-starter-web (for web based)

spring-boot-starter-test (for testing)

spring-boot-starter-thymeleaf (for rendering views) (server-side java template engine)

1.4 Setting Up Apache Maven

Download from www.maven.apache.org/download.cgi.

Download by clicking on the 'Binary zip archive' link.

Save the file and once downloaded 'Extract All'.

Navigate to the unzipped folder and open bin.

Now copy the directory path up to bin, for example: c:/apache-maven-3.6.2/bin.

Type 'Environment variables' on the windows search menu and hit 'Enter'.

Open the 'Environment variables' tab, under 'System variables' select 'Path' and click on 'Edit'.

Click on 'New' and paste the path copied earlier, for-example: c:/apache-maven-3.6.2/bin.

Once done, click 'OK'.

To check if maven is installed, type "mvn --version" and hit Enter.

1.5 Building a Spring Boot Application

In cmd type:

```
mvn archetype:generate -DgroupId=com.solution.springboot \
    -DartifactId=Spring BootHelloApp \
    -DarchetypeArtifactId=maven-archetype-quickstart \
    -DinteractiveMode=false
```

This command will create a new quickstart maven project as specified in archetypeartifactId template specified. The groupId uniquely identifies the project, the artifact Id is the name of the Jar file that you will build using the Java project. Import the generated Maven project to Eclipse IDE.

First, we can see that there are three annotations in the default code:

```
@Spring BootConfiguration
@EnableAutoConfiguration
@ComponentScan
```

these three annotations have their own specific role to play, and it is important to understand the use of each annotation as they play a vital role in forming the core of Spring Bot framework.

Getting to know annotations:

@Spring BootConfiguration - Class level annotation that is a part of Spring Boot framework and it indicates that a class provides application configuration. In Spring Boot, you typically use Java annotation-based configuration specification rather than XML files. As a result, the @SpringBootConfiguration annotation is the primary source of configuration within your applications. Generally, this annotation is applied to the class that contains the main method, the entry point for the Spring Boot application. The use of this annotation in the class that is the entry-point for the application allows configuration to be automatically located.

@EnableAutoConfiguration - This annotation auto-configures the beans present in the Java classpath. Any dependency available in the classpath of the application set up using the starter descriptors will be injectable automatically as a bean within the app. These libraries can be referenced in a very straight forward manner rather than jumping through hoops. This annotation does a great work in guessing the beans needed for the app and inject them. Say for example we have a JAR file for Tomcat Server in our classpath, the annotation will inject the Tomcat Servlet Container Bean Factory to configure the Tomcat Server, and this is done automatically.

@ComponentScan - This annotation tells the Spring Boot that it needs to scan the current package, which is `com.solution.springboot` in our case and all of its sub packages within this current package for injectable beans. Spring Boot will take care of performing the scan and injecting all of the beans that are our dependencies.

While working with Spring instead of using these annotations separately we can use `@SpringBootApplication` annotation to encompass the work of all three annotations.

1.6 Running on the Embedded Tomcat Server

Let's create a Spring MVC app using Spring Boot. What we need to do here is to make changes to the `POM.xml` file. MVC stands for Model View Controller, the Model works with DB and underlying data that is represented; the View is what is rendered to the User, and the Controller has the business logic of the app.

We will define a simple `HelloController` which is a simple `RestController`, denoted by `@RestController` annotation. A `@RestController` essentially is what you'll use to build your Rest APIs, CRUD Operations. The code is:

```
@RestController
public class HelloController {
    @RequestMapping(value="/", method=RequestMethod.GET)
    public String index() {
        return "This is the Main Page";
    }
    @RequestMapping(value="/welcome",
method=RequestMethod.GET)
    public String welcome() {
        return "Welcome to Spring Boot!";
    }
    @RequestMapping(value="/hello",
method=RequestMethod.GET)
    public String hello() {
        return "Hello Spring Boot!";
    }
}
```

When we tag a class with `@RestController` annotation Spring MVC essentially knows that whatever response you send from the controller methods have been mapped are essentially web responses that need to be rendered to the user as such. `@RequestMapping` are mapping handler methods that map to a URL-path specified with the `RequestMethod` that defines the type (GET, PUT, etc..)

of request. To run this application we would not need to create a WAR file or so, rather run this as a Java application and a Tomcat server will be spun up automatically and will host the app in any available local-port which can be found on the console. To use any other embedded web server instead of tomcat, exclude Tomcat in the POM file and add the dependency for the desired web server.

1.7 Generating and Deploying external WAR file

To be able to generate a WAR file or to execute the application on an external server we just need to make two changes, 1. POM file and 2. The way application is set-up.

Step 1: Specify the packaging tag as `war`.

Step 2: Create a starter-tomcat dependency and add ``<scope>provided</scope>`` within the dependency. This will tell Spring Boot that an external server will be provided to run the application.

Step 3: To make sure that the WAR files are executable, include:

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

Step 4: To create a deployable WAR file, you need to make a few changes to the application's code. Override the configure method as:

```
@Override
protected SpringApplicationBuilder
configure(SpringApplicationBuilder application) {
    return application.sources(App.class);
}
```

That is all for the changes the main method and the RestController remains untouched.

Step 5: Instead of creating a WAR file explicitly, create one using the maven build and while doing so add `clean verify` as Goals and click Ok. This will generate the WAR file.

Step 6(Optional): Creating a Tomcat server, go to tomcat.apache.org/download-90.cgi. Download the latest version(9 recommended) as tar.gz file. Unzip the file and place it the desired folder.

Step 7(Optional): Go to the target folder of the application, here we can find the WAR file. Move the WAR file to the webapps directory in the tomcat folder.

Step 8: Open CMD and change the path to the bin folder of Tomcat server. Enter the bin folder in file explorer and execute the `startup.bat` file. This will start the tomcat server.

Step 9: Go to the web browser and enter the URL `localhost:8080/Spring BootHelloApp-1.0-SNAPSHOT/`

We have successfully deployed our WAR package and executed it using an external Tomcat server. To quit/stop the server hold ctrl+c and it will stop the server. To check if the port is freed up in cmd type the command `netstat -ano | findstr:8080`, if there is any process then we can force quit using `taskkill/pid <processid>`.

1.8 Using Spring Initializr

As in the previous topics we can create a Spring Boot starter app using CLI but it is both hard to remember every line of the command and to describe the dependencies. So to reduce the tedious process we can make use of Spring Initializr(<http://start.spring.io>) a web-based tool useful for setting up project structure for Spring Boot. Here all you have to do is select the type of Java application and the dependencies you need, this will generate a Spring Boot starter application with the requested specifications and download it as a Spring Boot app that you can work on.

1.9 Performing Message Internationalization

In this section we will see how you can configure different messages in your web application based on the locale (country specific like language).

Step 1: Open the POM file we have the Spring Boot Starter dependency as usual, we will add a new dependency for thymeleaf as:

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

Thymeleaf is a java based template engine used to transform data displayed in web applications. There will be no change in the entry point of the Java application.

Step 2: A simple message Controller class will be created with @Controller annotation. The @Controller annotation tags this as one, which handles incoming web requests mapped to specific parts.

However, the value returned by the handler methods is not considered as a web response, instead the return value of these methods is interpreted as a logical view name. The Thymeleaf template will then map this view name to a physical view which is then rendered by the browser.

The physical view corresponding to the logical view name will be placed within the template folder under src/main/resources. Expand the templates folder to view the .html file for the view name.

The html files are not pure html files, they are Thymeleaf based html files that refer to variables defined within `messages.properties` file.

Step 3: Create .html Thymeleaf based files for logical view names created.

Step 4: Create `messages.properties` file with the names specified in html files with locales.

Step 5: Create a new package `com.solution.springinitializr.config` and create a class file within as `InternationalizationService`.

File: "InternationalizationService.java"

```
@Configuration
public class InternationalizationService implements WebMvcConfigurer{

@Bean
public LocaleResolver localeResolver(){
    SessionLocaleResolver localeResolver = new
                                                SessionLocaleResolver();
    localeResolver.setDefaultLocale(Locale.US);
}
```



```

        return localeResolver;
    }

@Bean
public LocaleChangeInterceptor localeChangeInterceptor() {
    LocaleChangeInterceptor localeChangeInterceptor = new
        LocaleChangeInterceptor();
    localeChangeInterceptor.setParamName("lang");
    return localeChangeInterceptor;
}

@Override
public void addInterceptors(InterceptorRegistry registry) {
    registry.addInterceptor(localeChangeInterceptor());
}
}

```

The @Configuration indicates that a class declares one or more bean methods.

The first Bean localeResolver is used to set default Locale which is US

The second Bean LocaleChangeInterceptor is used to Intercept the incoming request and change as per locale.

Step 6: Create additional .properties file (E.g.: messages_french.properties) for each language we want to change.

Step 7: Open the browser and type "http://localhost:8080/home?lang=french".

1.10 Using Spring Suite

While working with Eclipse IDE we can install a set of tools called STS(Spring Tools Suite) in order to Spring application development easier.

To install go to Help->Eclipse Marketplace and search for "Spring Tools 4", install the one that is by VM Ware. When installation is successful IDE restarts.

After installation go to File->New->Other search for Spring Boot, select SpringStarter project and click next. Configure the project settings and you're ready to go.

1.11 Configuring Server Port

In this section we can see how we can change the port on which our web application run. By default the server port is 8080.

Method 1:

Step 1: Right Click on the project, select Run As and choose Run Configurations.

Step 2: Select Spring Boot application on the left pane and click on Arguments tab.

Under Program arguments type `--server.port=7070`, click on apply and click Run.

Method 2:

Step 1: Open `application.properties` type `server.port=7070` and save.

1.12 Configuring User Defined Properties

In this section we will see how to specify our own user defined properties in `application.properties` file and have Spring automatically identify and pickup these properties as a part of the application

Step 1: Create a GreetingController to greet the user, user @RequestParam annotation to get values to greet each user uniquely.

Step 2: Send the greeting message to the model to be displayed by the view resolver

Step 3: In the browser type "localhost:8080/greeting?name=John".

Step 4: Create a MessageProperties.java file to configure the messages to be sent.

Chapter II: Asynchronous Methods, Schedulers and Forms

2.0 Objectives:

- Implement Asynchronous Methods on background threads
- Schedule tasks at a fixed rate with a fixed delay
- Schedule asynchronous tasks using multiple threads
- Extract request parameters and path elements
- Use forms to get user inputs
- Validate forms using built-in validators

2.1 Implementing Asynchronous Methods

In this section we will check out how we can make an asynchronous background request within our Spring Boot application. We will make use of futures within Spring Boot to run asynchronous operations.

Now open the Eclipse IDE and open POM file from the starter project, we will have starter-web dependency which will give access to REST template that we can use to make URL requests. Now open the main class file(AsynchronousApplication.java), here we have made use of @SpringBootApplication annotation that we are familiar with that denotes the main class but we also have @EnableAsync annotation. The @EnableAsync annotation turns on the Spring's ability to run asynchronous methods in a background thread pool, rather than instantiating the threads directly this annotation uses an executor. The executor to be used can be specified using the task executor method (as on line 20). If no task executor is assigned, then the @EnableAsync annotation will make use of a simple one by default.

As a part of Asynchronous call, we will look up GitHub pages (Browse: api.github.com/users/spring-security) and as we can notice the information retrieved are in JSON format, we need a way to convert this JSON response to an actual Java object.

We will define a user class which represents the response from a specific GitHub user, Spring has a number of different ways to convert a JSON response to a Java object representation. By default, Spring Boot uses Jackson library. Notice that the user class has a specific annotation @JsonIgnoreProperties this indicates Spring that the JSON response we get from the URL will contain properties. The properties that we are interested in, and it should simply ignore those properties, the

properties we are interested in are Name, Blog, Type and URL. The member variables (names of props in user class file) that we specify for this user object should match the JSON keys in the web response. Spring will use reflection to access the names of these member variables and look up the corresponding JSON properties in the web response. At the end of User class we will override the toString method to printout the details of a single user.

The actual Asynchronous method call made to look up the GitHub users information is done within the LookupService class file. The LookupService class is annotated using the @Service annotation, this annotation indicates that this is a Spring managed component (A Spring bean whose life cycle is taken care of by the Spring framework). The @Service states that this class holds business logic. This LookupService will perform a look up operation, looking up a particular GitHub user. The URL to be hit is specified in GITHUB_USERS_URL, the actual look up for a specific user is done by using Spring's restTemplate. The restTemplate is injected into the constructor of the look up service (line 23-27). The important part of the code is the findUser method which is annotated using @Async annotation which tells the Spring Boot application that code for this method needs to be run on a separate thread (not on the main thread but on the background thread).

We can see that the return value for the findUser method is a completable future which holds a user object (We need to know that returning a future is a requirement for any async operation), the completable future will hold the user object representing our GitHub user once the asynchronous method invocation is complete and the result is available. The input argument to the findUser method is the user name in the form of a string, we then generate the URL for the specified user using String.format (on line 33), we then use restTemplate.getForObject to make a get request to this URL and we make use of User.class as an input argument (an object of the user class is what will hold the JSON response that we get from this URL) then we call a Thread.sleep() to make the async operations run slowly once the response for the user is got we CompletableFuture.completedFuture and return the result.

Moving over to the LookupAppRunner.java file, the LookupAppRunner is the class that executes and invokes the LookupService to Lookup GitHub users. The class has @Component annotation which indicates that it is a Spring managed object, it is Spring's job to take care of instantiating this object and also executing the same. But, how do we know if it is executable? this is because the LookupAppRunner implements CommandLineRunner (indicates a bean should be run or executed when contained within a Spring application), hence once this bean class is instantiated, it should be run or executed. As soon as the Spring Boot application starts up the LookupAppRunner will be instantiated and then run, the method that will be invoked to run the application is the run method (on line 23). In order to make the asynchronous method calls the LookupAppRunner needs LookupService which is provided by the @Autowired annotation on the LookupService member variable. The run method takes in command line arguments in the form of an array which is not necessary, we get the info for a total of 6 different users. The findUser methods will all run asynchronously, we have a CompletableFuture.allOf.join (on line 34) which will ensure that all of the lookup operations are complete.

2.2 Scheduling Tasks

In Spring it is possible to run tasks at scheduled intervals and this is possible using `@Scheduled` annotation. In the POM file the only dependency descriptor we specify is the `spring-boot-starter`, our application that will run tasks in a scheduled interval will be a simple Spring Boot application, therefore we will not be using web dependency, Thymeleaf or any other dependency.

Open the `SchedulerApplication.java` which is the main entry point for our application, within this file we will have two annotations `@SpringBootApplication` and `@EnableScheduling`. The `@EnableScheduling` annotation is what tells Spring that this application will run tasks at scheduled intervals, this annotation will bring up a task executor to run the scheduled tasks on a background thread (not on the main thread).

Note in the previous section while running the application we called `close()` function immediately after running to stop the execution, but here we don't because the application has to be up and running at all points in time. The scheduled task will run at fixed intervals or at fixed delays based on our specification.

Setting up a scheduled task is straight forward and will only require the use of `@Scheduled` annotation, move over to `SimpleScheduler.java` file. This class is tagged with the `@Component` annotation indicating that this is a component managed by the Spring runtime, within the component we have the `ScheduledLookup` method which is tagged using `@Scheduled` annotation. Anything with `@Scheduled` annotation, Spring will invoke this method at pre-specified times based on a schedule, the `@Scheduled` annotation will have arguments that specify how often the method will be invoked and when. In the code we use `fixedRate=3000` as the argument, which means that this method will be invoked every 3 seconds. The `fixedRate` is typically used when every invocation of this method is an independent process, subsequent invocations don't depend on previous invocations. There is exactly one background thread that will be running the tasks which means that unless the previous invocation of the method is complete, the next invocation will not be started (irrespective of the `fixedRate` value). There is another argument called `fixedDelay` which can be used like `fixedRate`, but the difference is that the value passed will be the time between the completion of previous process and the start of next process (each process starts only after 3 seconds have elapsed after the previous process was completed).

`@Scheduled` annotation arguments possible:

- `fixedRate`
- `fixedDelay`
- `initialDelay`

2.3 Scheduling Asynchronous Tasks

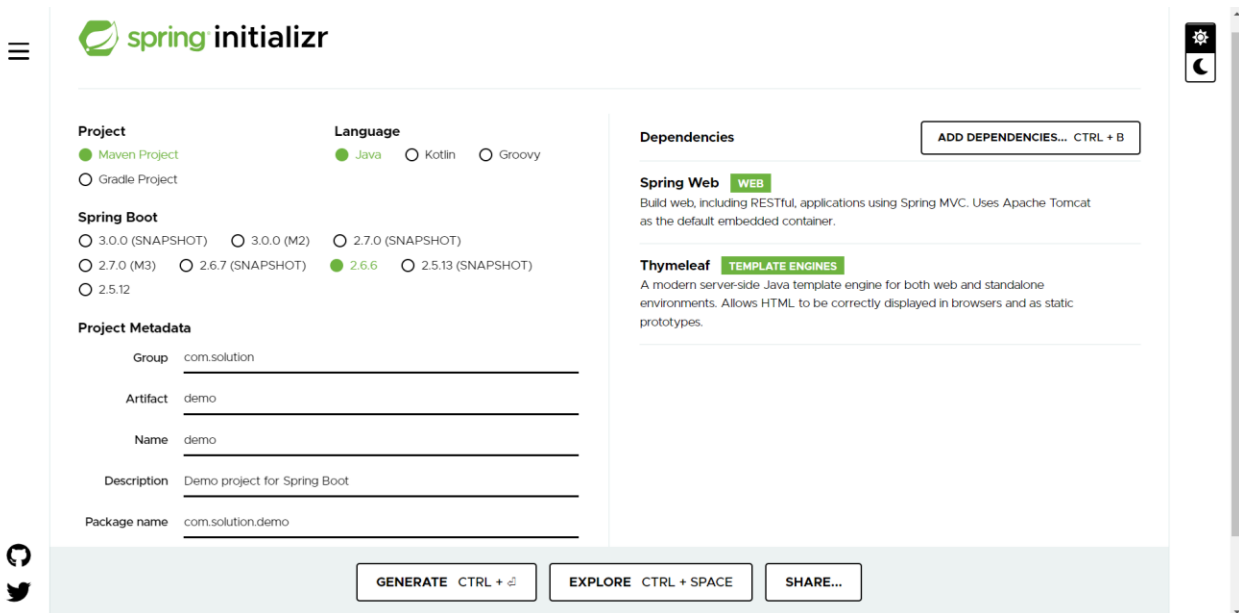
In this section we will see how to run scheduled tasks asynchronously on different background threads. In the POM file we will need `spring-boot-starter-web` dependency to use REST template to make an asynchronous request to GitHub URLs. We will have multiple web requests run asynchronously on different background threads.

Moving on to the main entry point of the application `AsynchronousApplication.java` file, this file has several annotations `@SpringBootApplication`, `@EnableAsync` and `@EnableScheduling`. We will make use of `taskExecutor` method to execute tasks, the `taskExecutor` returns an object of the executor

interface which allows us to use a thread pool executor to run our tasks on different threads. We have instantiated a new thread pool executor with `CorePoolSize` and `MaxPoolSize` as 1, that is we have configured the thread pool to have exactly one thread, which means our method invocations will run on just one thread. All other files remain the same as in section 1 of Chapter 2, but in `LookupController` we will add a `scheduledTasks` method to perform the look up service in the scheduled time slots.

2.4 Using Request Parameters and Dynamic Paths

In this section we will discuss on how we can work with values associated with the request parameters in a URL. Let's start off by creating a new project from Spring Initializr:



The screenshot shows the Spring Initializr web interface. On the left, there's a sidebar with a hamburger menu and social media icons. The main area is divided into sections: **Project** (Maven Project selected), **Language** (Java selected), **Spring Boot** (2.6.6 selected), and **Project Metadata** (Group: com.solution, Artifact: demo, Name: demo, Description: Demo project for Spring Boot, Package name: com.solution.demo). On the right, there's a **Dependencies** section with a button 'ADD DEPENDENCIES... CTRL + B'. Below it, 'Spring Web' is selected under the 'WEB' category, and 'Thymeleaf' is selected under the 'TEMPLATE ENGINES' category. At the bottom, there are three buttons: 'GENERATE CTRL + G', 'EXPLORE CTRL + SPACE', and 'SHARE...'.

Unzip the downloaded package and open with EclipseIDE we can see that we have the usual presets, navigate to `com.solution.test.controller` package and open `WebServiceController.java` file, here we can find the annotation `@RestController`, when we tag the controller class as a `RestController` any value we return from the handler method is automatically considered a response body (handler method response are assumed to be tagged with `@ResponseBody`), that is it is considered to be a web response that is rendered to the user; the return values from the handler methods are not logical views rather they are response bodies to be directly rendered rather than mapped to a physical view. This controller is where we specify the methods used to handle the incoming web requests. We also have an `@RequestMapping` annotation, which is typically used if all the URLs for individual methods in this controller have the same prefix. In our code all the URLs will have ``organization`` prefix. Within the class we have a single method handler the name of the method is "myName" and it's tagged using the `GetMapping/info`, this method is invoke if the path `<host:port>/organization/info` is used. Run the code and check the browser to find that the parameter given on `"localhost:8080/organization/info?name=Excaliber"` we can see the Parameter Excaliber was passed successfully. Remember that the variable name at `@RequestParam` annotation and the variable name of the URL must match for Spring to match the values (In our code "name" is the variable name).

Moving on to Dynamic paths, return to EclipseIDE we will extract the name this time using the dynamic URL path this is achieved by using `@PathVariable` annotation. To get the name we will make certain changes to the code and the first one is done to `@GetMapping` annotation which is now changed to `"/info/{name}"` this indicates to Spring that the name portion of the URL is actually dynamic and can be any value, no matter what the value on the name portion of this path is map that particular path to the `myName` handler method, because this dynamic portion will be extracted as an input argument to the method. The `@PathVariable` annotation is configured to extract the name portion of the dynamic path and make it available in the name variable. Now run the program and head to the browser, type the URL "localhost:8080/organization/info/Excaliber", now the dynamic path works.

Let's make some changes to the `WebServiceController.java` file to use form to extract the data, first we will make use of `@Controller` annotation where the response from the handler method will be treated as a logical view by default unless the handler method has been explicitly tagged using the `@ResponseBody` annotation. We did it because we will have a new handler method called `formPage` to receive an input. Now go to `src/main/resources` there is a `templates` folder with `editName.html` file which contains a form we can use to fill in the name that will be passed as a request parameter to our handler methods.

2.5 Using Forms

In this section we will work with forms in Spring Boot, while working with the forms it involves a user interface, some business logic and data. This means we will extensively make use of Spring MVC module which gives us the Model View Controller paradigm.

Let's look at the Model object that is in `SupportForm.java` file within the model package, it is a POJO class that will bind to the form it will have the variables we need to use in our forms. When working with forms in Spring we instantiate a model object and then bind that model to the form, the inputs entered in the form's fields will be used to set values on this model object.

Moving on to the controller in the `SupportFormController.java` file, the controller specifies the methods that are invoked corresponding to incoming requests. We will have two handlers one to render the view (`@GetMapping`) and the other to retain the data from the form (`@PostMapping`). The `complaintForm` will take an input argument the model (`SupportForm.java`) of the application, the views in our application will render the data represented in the model. We will also have a `membershipList` to be passed as a dropdown we set it up in the form of list and use `model.addAttribute` to pass the `membershipList` down to the form view, we get all the model details needed for the form we render support logical view which maps to a physical view `support.html`. We have a `submitComplaint` method handler it has input arguments to this method are `SupportForm` and `Model` object, we can notice that the `SupportForm` that is injected into this method has `@ModelAttribute` annotation which essentially means that this is the bound object that we get from our form on the client sent back to the server, the fields on the `SupportForm` object are filled based on the values that the user has specified on the client form (all the user details will be available in the `SupportForm` object). The details are then added using the `Model` (in args).

2.6 Performing Form validation

In the previous section we came across how to use forms with Spring Boot now we will see how we can work with validating the forms while submitting. To start with go to POM file and add spring-

boot-starter-validation dependency, this starter dependency in Spring Boot allows us to use the JavaX validation API and the Hibernate validator. All the code part remains same except for the Model(supportForm.java) notice that we have all the same variables but each variable has an annotation denoting the condition to be followed for validation. The common annotation used is @NotNull, the next change to be done is on SupportFormController.java file to reject forms that are not valid. The complaintForm remains the same all the changes are to be done in submitComplain method handler, notice that we have a new @Valid annotation and a new argument of bindingResult object and it is mandatory that it should be followed by the Model object bound to; we will add an if condition to check if there are any error and if it's true will return the support page and if no errors found it will be submitted. Now let's go to support.html form make changes to let the user know the errors encountered (on lines 40,46,52)

Chapter III: Building RESTful API Services

3.0 Objectives:

- Installing Advanced REST Client
- Performing Read Operation
- Performing Create Operation
- Performing Update Operation
- Performing Delete Operation
- Installing MySQL and MySQL workbench
- Integrating with MySQL
- Performing CRUD operations with a database
- CRUD operations using a Web UI
- Caching using @Cacheable
- Clearing cache using @CacheEvict

3.1 Installing Advanced REST Client

The Advanced REST Client is an API testing tool that you can use to test your API with Get, Put, Post, Delete and other HTTP requests.

Step 1: In the browser enter "<https://install.advancedrestclient.com/install>" and hit enter

Step 2: Under the download button make sure the OS is the OS corresponding to your system and hit download.

Step 3: Once the download is completed, open the downloaded file and click on Run.

Step 4: Once installed it will open the Advanced REST Client window, you can choose to walk through or skip if needed.

Step 5: Now after the walkthrough/skip we will arrive at the page where we can type the URL and make requests to it.

3.2 Performing Read Operation

Unlike other chapters, in this chapter from start to end we will use the same Spring Boot project for performing all the different operations. We will be iteratively adding RESTful API that we build using

Spring Boot, initially we will set up the APIs for the CRUD operations create, read, update and delete. We will then wire up the web application to a MySQL database using JPA and Hibernate to work with that database, then we will add a few web pages to the application in order to perform these CRUD operations and finally we'll see how we can implement caching in our web app.

Create a new Spring boot application with MVC architecture

FILESTRUCTURE GOES HERE

Open the Product.java file under the model package of the application, the product class is a POJO class where every product has an Id, Name and Category. The class has two constructors (on line 9 and 12) where one is a default no argument constructor and another takes in input arguments, the remaining code is all just getters and setters for the member variables.

In the first iteration of the web service, we'll set up a handler mapping for a get request for a read operation, open the ProductController.java file where you can find that the class is annotated with `@RestController` indicating that the responses from the handler method should be treated as response bodies, or web responses rendered to the user. We also have a `getAllProducts` method which returns a list of product objects, it is annotated using `@GetMapping("/products")` in this we will not be retrieving products from the database rather we have hard coded a list of products which will be returned to the user, every product has its own id, name and category. Run the application and switch to Advanced REST Client, select GET as method and enter the URL "http://localhost:8080/products" and click on send, we will get the response with data that we have hardcoded and a status code "200 OK" stating that URL hit was successful.

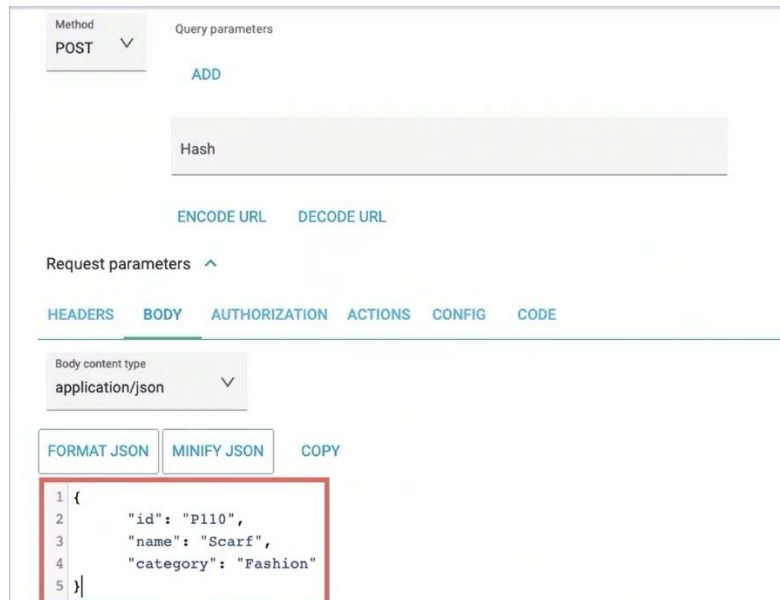
Now to improvise move to ProductService.java file in the service package and add a service class called ProductService annotated with `@Service` annotation, the service layer in any web application is what contains the business logic of our code. This file will now contain the product list that was available in our ProductController.java file earlier, so now the ProductController.java file is simplified and all it will do is get the list from the ProductService class. In the ProductService class let's add a new method called `getProduct` that will retrieve a single product with a specified Id. We have a list of products by invoking the stream function on this list, we will convert the list into a stream and perform filter operation to retrieve the product with the specified Id. We'll have a handler mapping method `getProduct` to the ProductController class as well and this handler mapping uses a path variable with `@GetMapping("/products/{pid}")` where {pid} is the dynamic path variable. With these changes in place run the application and move over to the ARC with GET as method enter the URL "http://localhost:8080/products/P105" and hit send we will get the product with the P105 as its Id.

3.3 Performing Create Operation

In this section we will further enhance the code from the previous section to be able to add a new product by sending a POST request to our API service. The change to be made first is in the ProductService class we have added a new method called `addProduct` that takes an object of the Product as an argument, we need to expose this ability to add a new product via the ProductController class so we create a method handler called `addProduct` in the ProductController class with `@PostMapping` this will handle the incoming Post requests, the input argument to this controller method is an instance that we want to add to our list of products and this is tagged using `@RequestBody` which means that the incoming request will contain the details to instantiate a new

product object. Spring will ensure that the request body is converted to a product instance and this product will be added to our list of products by invoking the productService.addProduct method.

To ensure it works properly run the application and switch to ACR, this time change the method to POST and enter the URL "http://localhost:8080/products" we are not done yet to send data click on the down arrow next to the URL and click on the body tag under request parameter and enter the product details with application/json as Body content type and now hit Send this will make a post request to the server. To ensure the data was saved send a GET request to the URL "http://localhost:8080/products".



Method: POST

Query parameters: ADD

Hash

ENCODE URL DECODE URL

Request parameters: ^

HEADERS BODY AUTHORIZATION ACTIONS CONFIG CODE

Body content type: application/json

FORMAT JSON MINIFY JSON COPY

```
1 {
2   "id": "P110",
3   "name": "Scarf",
4   "category": "Fashion"
5 }
```

3.4 Performing Update Operation

As for now we have done both read and create operations, for this section we will add one more operation to the APIs that our web service provides the ability to update a particular product. The first change will be made in the ProductService class we will add a method called updateProduct which takes in two input arguments the Id and a new instance of the product with updated details to do this we have a for loop to iterate over the products to check if the id matches with the id of any available products, if a product with the Id was found we will set a new product instance at that index this will effectively get rid of old product instance and the new product instance with the updated detail will become a part of our web app. Now move to ProductController class to set method handlers for the update operation, we will add updateProduct with @PutMapping annotation this responds to HTTP put requests (Update operations are typically Put request rather than Post request) notice that we make use of dynamic path to get the Id and we use @RequestBody to store the updated product. Now we can run this application to ensure that the code is running properly, start the application and switch to ARC. In the ARC change the method to PUT and enter the URL "http://localhost:8080/products/P101" and click on body under the request parameters section to give the new information before sending the request set all the configurations as the same for what we did with read and just type the information to be updated.

3.5 Performing Delete Operation

As for now we have implemented all the CRUD operations except for the delete operation, so let's get started. Let's make the very first change in ProductService class add a deleteProduct method that takes Id as the input argument for the product to be deleted, we then use products.removeIf() operation to remove if any such product was found in the list of products. Now we have to expose the delete operation via the ProductController class and create a handler method for the operation we will create a deleteProduct method with @DeleteMapping annotation, notice that we have a dynamic path to get the Id of the product to be deleted. Now run the application and switch to ARC, select Delete as the method and enter the URL "http://localhost:8080/products/P101" and hit send to delete the product. To ensure the product was deleted create a GET request for all details to check of the product specified was deleted.

3.6 Installing MySQL and MySQL workbench

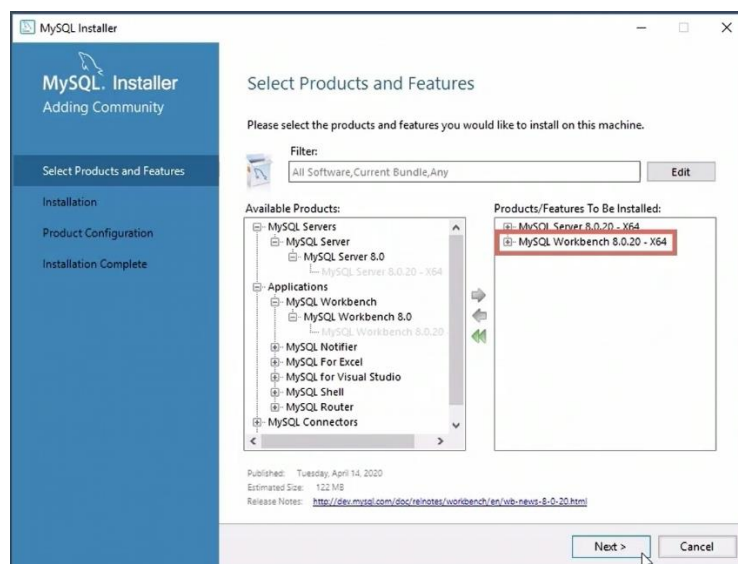
Now as we have seen how we can perform CRUD operations, so now let's go ahead and see how we can do the same with database without the need for us to hard code the data within our application. In this section we will see how to install MySQL and MySQL workbench followed by integration on the next section. Follow the steps for the installation process:

Step 1: Open the browser and head over to "https://dev.mysql.com/downloads", choose MySQL installer for windows.

Step 2: We are redirected to a page where we can download the installer for MySQL server and the MySQL workbench user interface that we will use to work with, download the MySQL installer.

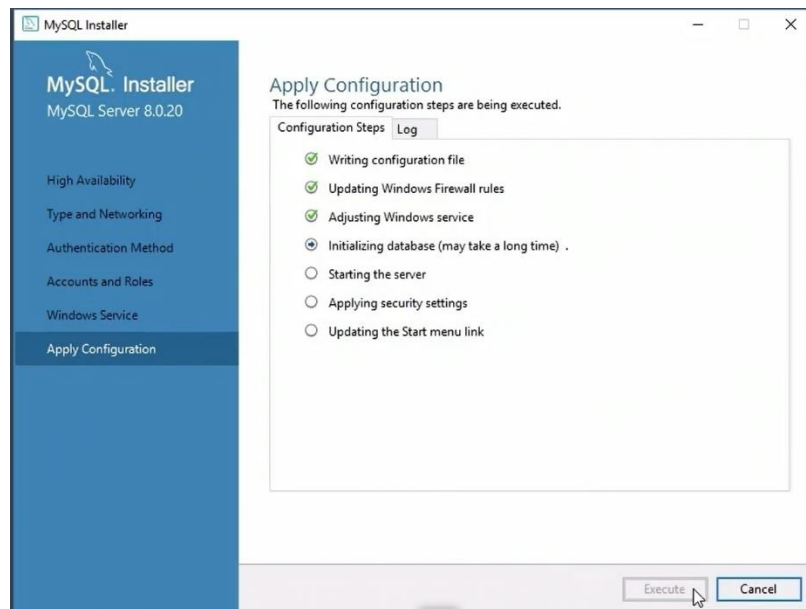
Step 3: Once the installer is downloaded, run the installer. This will have a walkthrough phase of what are the options available and what we wish to download.

Step 4: Select the MySQL server and MySQL workbench under Applications and click on Next. Leave the rest as default and if any error occurs click on Ok.



Step 5: Once the processes are over click on Execute to install the selected files. Once the installation is done click on Next, in the upcoming screens all can be set as default but note the ports (default port :3306) and other details as they are crucial.

Step 6: Set the password for root user, configure the setup and click on Finish.



Step 7: Open the MySQL workbench, right click on the local instance and select Editconnection. Change the port to the port MySQL was installed to and next to password click on "Store to vault" and enter the password to root user.

Step 8: Click on test connect to check if connection was successful.

Step 9: Double click on the MySQL local instance it will take you to the UI where we can run queries.

3.7 Integrating with MySQL

As of now we have installed the MySQL server and MySQL Workbench, now let's integrate it with our application.

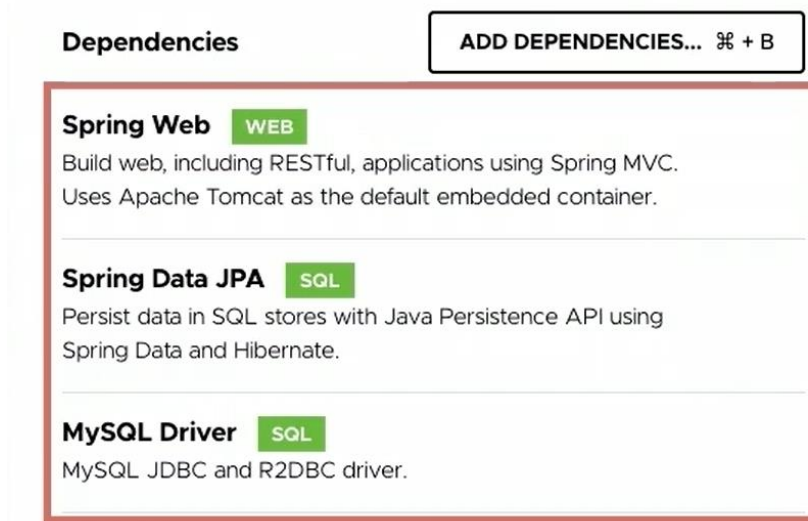
Execute the following commands in the MySQL Workbench

```
`CREATE DATABASE OnlineShoppingDB;`
```

```
`USE OnlineShoppingDB;`
```

this will create a database and use it as the DB for upcoming queries

As we are going to work with DB we need to create a new Spring project with additional dependencies (you can also add new dependencies to POM file of existing project and run "Update Maven" to get the dependencies). While creating a new Spring project we will keep everything same except we will add new dependencies.



To know more about JPA and JDBC, go to Java repository on my Github and give it read. They are simple concepts and are easy to follow. Once done generate the project structure and import it to the EclipseIDE, we will copy some of the code from the previous sections and we will do some enhancements to make our application communicate with the underlying DB.

Head over to Product.java file which was previously a POJO class, there are certain changes made which is now an entity and can be stored as a record in the underlying database. The @Entity annotation is a part of the Java Persistence API, it tags this class as mapped to an underlying database table and the name of the table is "products", hence every object of this class(every product instance) in our application maps to a record or a row in the products table. Every entity in JPA has to have a primary key and here the primary key is denoted by @Id annotation and @GeneratedValue annotation means that JPA and Hibernate will automatically generate this primary key.

Now head over to the ProductRepository.java file under the repository package, the product repository is user the Repository namespace. The repository namespace contains classes which interact directly with our underlying database, these are the data access classes. We can see that the code is relatively less but remember this is an advantage of using Spring Boot, the @Repository states that it is a Spring component for data access and the ProductRepository interface extends the CrudRepository interface which is a generic interface with two generic parameters. This means that the CrudRepository interface will expose create, read, update, and delete operations for the product entity that we have defined, and this product entity has a primary key of type Long. Now that we have the interface all set what about the implementation, Spring Data module will auto generated the implementation since we have specified the type of the primary key and type of the entity already. The implementation for all of the methods exposed by the CrudRepository interface will be automatically available (E.g.:findById(),findAll(),save(),saveAll(),delete(),deleteAll(),existsById(),...).

Now let's see the ProductService class, what happens now is that instead of storing values in memory we will store it to the database. We inject the ProductRepository into this ProductService using @Autowired and all operations of the product service will be delegated to the ProductRepository. We have an enhancement as well that is the exceptions, we have added two new classes ProductNotFoundException class and ProductNotFound response class. The ProductController class

remains unchanged, and the last change is made in the main entry point (Spring BootApplication.java) file, we directly inject ProductRepository to Spring Boot application and also make the application implement CommandLineRunner interface indicating that this is an executable. We have also overridden the underlying class and saved a bunch of products to the underlying database table.

Now we have all the coding part set but we will add some things to our `application.properties` file which makes it easier for our application to connect with the database.

3.8 Performing CRUD operations with a database

Now that we have integrated the Database to our application let's move forward with performing CRUD operations within the database. Run the code we developed in the previous section and check the console for the operations performed. Everything we did earlier can work the same and be utilized but remember the id should not be given by us as the Id is auto generated, everything else is provided by us.

3.9 CRUD Operations using a Web UI

As for now we have all the backend part of our application set, but how about an UI instead to perform operations directly on the application rather than a client. We will start using Thymeleaf template to create UI for various CRUD operations we have performed earlier, in this section we will create Web UI for Create and Read operations. To do this we will be adding Thymeleaf dependency on to our POM file, another change that we'll make is to add an additional controller class. We already have a ProductController which is basically the controller class that exposes our REST API and we have another controller called ProductWebController. The ProductWebController will handle incoming web request from the user interface that we set up, notice that this controller class is annotated using @Controller annotation indicating that the response from our handler should be considered as a logical view. The ProductWebController does not query the database directly instead we'll reference the ProductController and use the ProductController to perform CRUD operations on the DB. We will form handlers on ProductWebController to establish connection between the buttons on the html page and the handlers of ProductController which in turn performs the actual operation itself.

We create a Thymeleaf template-based html file to show the list to the user and to ensure that the CRUD operations are possible we will add buttons to add, update and delete each item available as a product. The explanations for Thymeleaf code is present in the code available in the same repository as this file, refer to it for clarifications.

3.10 Caching using @Cacheable

Now if we were to retrieve products each time it is requested from the database the app can get slow and also if these products are listed on an ecommerce site the most common operations are retrieval and read of products, one way to speed up the display of products is by caching. In Spring Boot caching is very easy where we can cache products in memory or with some kind of in-memory database using just annotations, in order to utilize this we'll work with spring-boot-starter-cache dependency descriptor and add it to the POM file.

We will make a few changes in the Spring BootApplication.java file by adding a new annotation @EnableCaching this tells the Spring application that caching is enabled for this app, now we can choose which layer we want the caching to be performed on. We'll explicitly chose the service layer for caching

so that not only the REST API can take advantage of caching but also the web UI can make use of it (we have two different controller classes). Note: Always be sure to specify the caching annotation in a layer that is common to all the clients.

In the ProductService class we will use @Cacheable annotation with getAllProducts method to cache the data of the response for this method. The @Cacheable annotation works in a way that the response to this method is saved in the cache and when it is requested again it is sent directly from the cache itself rather than querying to understand the working we'll add a try catch block that performs sleep operation to the thread. But as there is an advantage with cache, we will also have a disadvantage, when we perform other CRUD operations it will not be reflected on the result as the result is from the cache and not from the database itself, so in order to overcome this we will perform cache clearing to ensure that we get the correct results as we need them.

3.11 Clearing Caches Using @CacheEvict

To keep the cache updated we need the cache to be cleared every time an operation other than retrieval was performed, to do this we will make use of @CacheEvict annotation. The @CacheEvict annotation clears the available cache every time the annotated method is called, when the method with @Cacheable annotation is called a new cache is formed and saved.

Chapter IV: Advanced Microservices & Securing Web Applications

4.0 Objectives

- Sending Emails using JavaMailSender API
- Using Interceptors
- Setting up the Zuul Edge Service
- Routing and Filtering with Zuul
- Setting Messages and Making Phone calls
- Setting up Default Login Page
- Configuring In-memory users
- Configuring Login Roles
- Making up a well developed Microservice

4.1 Sending Emails using JavaMailSender API

In this Spring Boot demo, we'll see how we can configure our Spring Boot service to send emails(with and without attachments) using JavaMailSender API. Create a new Spring Boot project using Spring Initializr, keep all the configurations as we used before and for the dependencies add the following dependencies:

Spring Web
Thymeleaf
Java Mail Sender
and click on generate.

Dependencies

ADD DEPENDENCIES... % + B

Spring Web WEB
Build web, including RESTful, applications using Spring MVC.
Uses Apache Tomcat as the default embedded container.

Thymeleaf TEMPLATE ENGINES
A modern server-side Java template engine for both web and standalone environments. Allows HTML to be correctly displayed in browsers and as static prototypes.

Java Mail Sender I/O
Send email using Java Mail and Spring Framework's JavaMailSender.

To send mail we need an email account, for this demo we will create a Gmail account called "meluha@gmail.com". Login to the account and click on settings (gear icon) -> select forwarding and POP/IMAP -> IMAP Access -> Enable IMAP and save changes, IMAP is what JavaMailSender uses to send mails. Open a new tab and type "myaccount.google.com/lesssecureapps?pli=1" on the browser and hit enter, turn on Allow Less Secure apps "ON".

Switch to Eclipse and set-up the application.properties tab and set up the mail properties (refer application.properties in GitHub code) for JavaMailSender API. We will write the actual code to send email in the MailController.java file.

In the MailController.java file, we make use of @Configuration annotation indicating that it specifies the configuration properties for our application. For this demo we make the MailController implement CommandLineRunner interface, meaning that this particular method will be considered an executable by Spring. Once the Spring application starts up, an object of this controller class will be instantiated and executed by invoking its run method, this makes this bean an executable bean. We have tagged the JavaMailSender variable using @Autowired annotation indicating it to be automatically injected, we also need to implement the run method that takes a variable number of arguments to call the sendEmail method and finish the process. The logic to send email is in the sendEmail function, here we will instantiate a SimpleMailMessage (because it has no attachments) and configure the mail fields (To, Subject, Text) and invoke the JavaMailSender.send to send the email. Now, run the code and we will be able to send mail from "meluha@gmail.com" to our receiver.

Now as we saw how to do without attachment, we will see how we shall do the same but with attachment. First, add the attachment you want to the project under src/main/resources and switch to the MailController.java file and update the code to send email with an attachment. We can notice that the basic components of the MailController class remains the same we will create a function called sendEmailWithAttachment and invoke it. Now move down to the sendEmailWithAttachment the first thing we can notice is that the object we work with, the MimeMessage is a part of JavaMailSender API. MimeMessage along with MimeMessageHelper helps us to send emails with attachments and inline resources. We will create an object of MimeMessage and use it to instantiate the MimeMessageHelper, we will configure the rest as we did earlier, notice that at the end we have made use of addAttachment

to add the attachment to the mail and the `ClassPathResource` looks for the attachment in the `src/main/resources` package to attach, finally we send the email using `javaMailSender.send()` method. Now run the application and check the email for the message.

4.2 Using Interceptors

In this section we will learn how to make use of Interceptors using Spring, Interceptors are an integral part of the Spring MVC framework. When working with MVC any incoming request is sent to the dispatcher servlet, which then looks up the handler mappings that we have configured to find the right method to handle this particular incoming request, Interceptors in Spring allow you to intercept the client requests to handler mappings and process these requests before they are handled by the controller methods. These interceptors can be used to add arguments to request before sending to the controller, background process and so on.

To respond to incoming web requests, we will set-up a simple in-memory application that returns a few books to the user. Switch to `Book.java` file which will be the model, for every book we have an id, name and `authorName` and some getters and setters.

All the interesting part of the code will be in the `BookHandlerInterceptor.java` file, the `BookHandlerInterceptor` implements `HandlerInterceptor` comes with three methods that we have to implement, the first method is executed before a request is sent to a handler, the second method after the request is sent to the handler and the third method after the completion of the request when the view is rendered. The first method that we implement from the `HandlerInterceptor` interface is the `preHandle` method, it takes in three arguments: the HTTP request, the response and the object handler, as the name suggests this method is executed before the actual handler method is executed, our `preHandle` method will check to see if the `bookId` parameter is set on the incoming request and logs all the requests to every individual books in the library via email. The next method we implement from the `HandlerInterceptor` Interface is the `postHandle` method, it takes in the arguments: the HTTP request, the response, the object handler and the `modelAndView`, the `postHandle` method is invoked after the handler code has been executed but before the view is rendered to the user, in this method we will check whether the `bookId` parameter is present in the request and will send a book access mail that the book access is now complete. Next we have the third method that the `HandlerInterceptor` interface is the `afterCompletion` method, it takes as an input argument: the request, the response, the handler object as well as any exception that may have been thrown by our handler mapping, this method is executed after the handler method is executed and the request is complete, when a response is complete an email is sent denoting that it was complete.

Before the Interceptor becomes a part of the Spring application it must be declared explicitly as such and we'll do this within the `WebAppConfiguration.java` file, the `WebAppConfiguration` implements the `WebMvcConfigurer` indicating that some external configuration properties are present in here. The class is tagged with `@Component` indicating it's managed by Spring, we inject the `bookInterceptor` here using `@Autowired` and then we implement the `addInterceptors` method. Within `addInterceptors` you have access to the `InterceptorRegistry` which is where we register all of the interceptors that we have in our code, we can configure more than one but we have just the `bookInterceptor` here.

We have just one last file to be worked with that is the `BookController.java` file, we use the `@RestController` annotation over the class indicating that all the return values from our handler

methods are web responses to be directly rendered in the browser. To keep things lite we will not use a database but we use a static in-memory map (bookstore), a map of integer to book. Lastly we will set up two handler mappings to access the application. Run the application using "http://localhost:8080/book?bookId=456" and we can see in the console that the interceptors were executed and that we have received from every interceptor regarding an activity.


4.3 Setting up the Zuul Edge Service

If we are working within an organization, it is possible that we are working over an enterprise-grade web application that provides a RESTful API for its services. Now, in that case we might need some features which are beyond basic interceptors from the previous section, we might want to set-up an API proxy for our services. What is an API proxy, it is a service that sits in front of our APIs, rather than our users consuming the APIs directly, the users will send their request to this API proxy and this API proxy will thus control access to our APIs. We can use the API proxy then to enhance the security that we provide to our APIs, we can also use it for monetization by tracking how many times API requests are made by specific users, general access tracking is possible as well.

In this section we will set-up a simple REST API service, which is essentially our actual API and will live behind a proxy. In this application we will have just one file that is the entry point to our application, in this file we utilize two annotations `@RestController` and `@SpringBootApplication` confirming the class to be a `RestController` and the entry point as well. When we want to run an API proxy and the application, we need to make use of different ports for each of them, so we configure our standalone application with port 7070 and run it.

Now for the API proxy create a new Spring Boot project with the dependencies as shown

Dependencies

ADD DEPENDENCIES...  B

Spring Web WEB
Build web, including RESTful, applications using Spring MVC.
Uses Apache Tomcat as the default embedded container.

Zuul [Maintenance] SPRING CLOUD ROUTING
Intelligent and programmable routing with Spring Cloud Netflix Zuul.

download the project and open it with EclipseIDE, navigate to `application.properties` file and set `"zuul.routes.products.URL="http://localhost:7070"` which means that any request to the `"/products"` path in this application should be rerouted to port 7070 that is our API service, this is the routing portion of the edge service where the actual request made to the API will be satisfied by the API service and not the routing service. Now navigate to the `com.solution.test.filter` package where we have various filters that can be utilized, remember all the filters must extend the base class `ZuulFilter`. The first one is the `PreFilter` which is run before the request is actually routed to the service, how to be sure that it is a prefilter? we can see that we have used `@Override` annotation on the method `filterType` that returns `"pre"` indicating it is a `PreFilter`, when we make use of multiple `PreFilter` methods we might need to

specify a certain order for the filters and that is done by overriding the `filterOrder` method, based on the incoming requests we may or may not want to apply the filters which is done by overriding `shouldFilter` method that returns a Boolean value if the filter should be used, the actual filtering operation is performed by the overridden run method which throws a ZuulException. The other types of filter are: ErrorFilter, PostFilter, RouteFilter.

Navigate to the `TestApplication` file we can notice that we have a new annotation called @EnableZuulProxy indicating that this is an Edge service application routing and filtering, in order to get the Zuul filters run we need to explicitly instantiate these filters as beans that the Spring manages.

Now that we have setup our application as well as the API proxy we can run our code if it works fine, in order to do that we need to run our applications one by one (first the application then the API proxy).

4.4 Routing and Filtering with Zuul

In the previous section we learnt how to integrate Zuul to our Spring Boot application and how to assign routers and filters, in this section we will put them to play and see how they work. While we run the API and hit the URL we can notice that we were rerouted to our app and we can see in the console that our filters were called successfully denoting that the filters did what they were intended to do.

ADD CONSOLE O/P FOR SUCCESS AND URL NOT HIT

4.5 Sending Messages and Making Phone calls

In this section we will go ahead and try sending messages and making phone calls using our Spring Boot Application, to do so we make use of twilio cloud platform and the twilio SDK to make and receive calls from our Spring Boot application. To get started SignUp for a free Twilio account at www.twilio.com/try-twilio, after creating a free account skip to dashboard. Once at dashboard click on "Get trial number" and click on "Choose this number" and this number will be associated with your twilio account.

Create a new SpringBoot application with started dependencies, import the project to Eclipse IDE and add twilio.sdk to the POM file. Now move to the main "TestApplication" file, we can notice that the main class implements ApplicationRunner interface (just like the CommandLineRunner from before this tells Spring that this application is executable and this class should be executed or run as soon as it's instantiated). As we make use of twilio we need three bits of information: the account set, the auth id and the Twilio number, all of these information are present in the Twilio dashboard itself. After entering the details required when we run the application we can see that we are receiving a call (can take upto 30 seconds) we can notice that this number is the number from our twilio account.

4.6 Setting up Default Login Page

For this section we will create a new project using the SpringInitializer with the following dependencies as shown below

Dependencies

ADD DEPENDENCIES... % + B

Spring Web

WEB

Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Thymeleaf

TEMPLATE ENGINES

A modern server-side Java template engine for both web and standalone environments. Allows HTML to be correctly displayed in browsers and as static prototypes.

Spring Security

SECURITY

Highly customizable authentication and access-control framework for Spring applications.

We will keep the main entry point to our Spring Boot Application as it is, let's switch on to our `LoginController` file whose class is annotated using `@Controller` annotation and had a mapping that return "home", the view for the home is available as home.html file that displays a simple message.

Run the application, within the console window we can notice a "Using Generated Security Password" this is because we haven't explicitly specified any user for our application, hence Spring Security will auto configure a user for us behind the scenes and this will be the password we can use to login as that user.

CONSOLE O/P GOES HERE

Now in the browser hit the URL "http://localhost:8080" we can see that a sign-in page is available, enter the username as "user" and then enter the password generated and then click on "Sign in". Now if we want to configure our own user name and password instead of the default we can assign values in the `application.properties` file where the user name and password can be denoted as:

44 GOES HERE

4.7 Configuring In-memory users

In the previous section we saw how we can setup a user login page for users to access, now let's see how we can setup login for various users that belong to different roles and configure certain pages that can be accessed by the users as per their roles, all of these are possible with Spring Security. In order to generate passwords for our In-memory users we will make use of `PasswordGenerator` class file, while working with passwords it is always a best practice to use encryption while saving these passwords (we use `BCryptPasswordEncoder`), we will encode two passwords for the two users that we will set up for our app. Now right-click on the PasswordGenerator file and select Run As->Java Application this will generate the encoded passwords and prints them to the console, leave it for later and move on to other parts of the application. Open the `LoginController` file that sets various Mappings to roles, we will then move to `SecurityConfig` file where we configure the security settings for our web application, we extend `WebSecurityConfigurerAdapter` this allows us to configure specific security settings for our application by overriding few of its methods, notice the annotations on this class `@Configuration` indicates that the class has configuration properties and `@EnableWebSecurity` enables the Spring Security's web security support and also provides integration with the Spring MVC module that we use to construct our web app, as we scroll down we can see the method called `UserDetailsService` where we will set up the usernames and passwords of the users as per their roles

and at the end we specify what are the URL paths that need to be secure and the paths that can be accessed without the user logging in. We'll have 4 webpages where the welcome and login pages are webpages that can be used freely, admin is a page the admin can only access and home is for the user. Now run the application and hit the URL in the browser.

4.8 Configuring Login Roles

In the last section we saw how we can setup login roles and the views for respective user, now we will see how they work and how views are rendered as per the user. As soon as we run the application and hit the URL "http://localhost:8080/" we will see the home page of our application, click on the first link to login we can notice that we are directed to the login page as a Security measure and this opens our "/home" only if we logged in as a user else we won't be able to view it, same applies for the next link but it can only be used by using the admin credentials not by the user login.

4.9 Making up a well-developed Microservice

In this section we use a single Spring Boot Application that will have many of the concepts we have seen throughout this PDF, we'll secure the pages of our web app using Spring Security. This time we will provide an option to Signup as a new user using an email address and will send a confirmation mail to the user which they can use to set up their password, the mail will have a special link with a confirmation token that allows the users to set password for their login, all the registered users will be stored to the MySQL database (includes registered users and users who set passwords as well) and users who have successfully set up their passwords are allowed to login.

Let's start with the MySQL Workbench/ Heidi SQL and do the following:

Step 1: Create a new database called SpringBootDB (Command: "Create database SpringBootDB;")

Step 2: Use the database created (Command: "Use SpringBootDB;").

We will make use of a new Spring Boot Application created using Spring Initializr with dependencies for: Data JPA, Security, validation-api, Thymeleaf, Web, Mail, MySQL.

PIT A DEMO PIC HERE

Switch to the `com.solution.test.model` package and open `User.java` file this has an @Entity annotation denoting that it is related to MySQL, this class represents a JPA entity which means objects of this class represent records or rows in the underlying "users" table and then we define the member variables (columns) of the table, the users table maps to an underlying database table where we store the users. Spring Security cannot work directly with these objects hence we use a wrapper class `CurrentUserDetails.java` which implements the UserDetails interface, any class which implements the UserDetails interface is what Spring Security uses to get core user information, this class is serializable hence we make use of `serialVersionUID` and we store an instance of the user object, a constructor to the CurrentUserDetails takes in the user object and the rest of the code are simply the implementations of the methods in the UserDetails interface.

Now we have setup the database and way to retrieve and use the user's information, now we need a way to perform CRUD operations on the database as we need to add new users and retrieve user data. Open the `UserRepository.java` file where we setup operations within the repository namespace by extending the CrudRepository interface, we make use of findByConfirmationToken and findByEmail to find the users within the database.

At this point we have the model and the basic DAO (Data Access Object) layer setup, we still need to setup the service layer where the business logic for our application resides. The service layer will expose methods allowing us to register a new user, send mail to a user and authenticate an already registered user. Let's start with the `UserService` class which implements the `userDetailsService`, we need this class to implement the user detail service so that Spring Security can then use an instance of this class to authenticate logged-in users (Spring Security will use an implementation of the `userDetailsService` interface to retrieve user related data in order to authenticate users), notice that this class is annotated using `@Service` annotation indicating that this code contains the business logic and is a part of the service layer, and then we make Spring inject the `userRepository` into the `UserService` method and within this class we expose certain methods that will be useful for our application, similar with the `EmailService` class we inject the `JavaMailSender` API that sends a simple message to the user with the link to setup password.

Let's move towards the controller classes, `LoginController` where we specify the handler mappings for the incoming web request (has two mappings one to the home page and the other to the login page). Moving on to the `RegistrationController` this is fairly complex but understanding the workflow makes it simple, notice the services and the objects that we inject into the `RegistrationController` we have the `BCryptPasswordEncoder`, `UserService` and `EmailService`, the first method (`showRegistrationPage`) is a handler mapping for the registration page for our application, the next method we have is the `processRegistrationForm` which posts the user to the database if the user hasn't registered previously, if registered it'll send an error message that the user already exists and if not user is saved as a new user.

Now we will look at how we can set up the security configuration settings for our application so that Spring Security will authenticate users from our underlying database table, we'll specify these settings in the `SecurityConfig` class which extends the `WebSecurityConfigurerAdapter` as before, the `@EnableWebSecurity` annotation turns on Spring Security and integrates with the Spring MVC now this security configuration requires the `userService` which we auto-inject to the class, we create a password encoder method to encrypt the user passwords before submission, we want only the users who registered to us and has set a password to be able to access so we need to specify the authentication provider to the Spring Security. To setup the Spring Security with the authentication provider we will use the `SecurityConfig` file where we override the methods we need to configure Spring Security to use. Now that we have completed the coding part move to the `application.properties` file to setup the credentials and configurations for the application to access the database and to send email.

Appendix

Form Annotations:

@NotNull
@NotEmpty
@Min(value=<value>,message=<error_message>)
@Size(min=<value>,max=<value>)
@Email
@Pattern(regex=<pattern>,message=<error_message>)
@GetMapping
@PostMapping
@PutMapping
@DeleteMapping
@Service
@Controller
@RestController
@Entity
@Id
@GeneratedValue
@Repository
@ControllerAdvice
@ExceptionHandler
@ResponseStatus
@EnableCaching
@Cacheable

Acronyms:

IoC - Inversion of Control
POJO - Plain Old Java Object
POM - Project Object Model
ARC - Advanced REST Client