Contents

[SPRING BOOT 3](#_Toc78801911)

[WHAT IS SPRING BOOT 3](#_Toc78801912)

[SPRING BOOT FEATURES 3](#_Toc78801913)

[BOOTING SPRING BOOT 3](#_Toc78801914)

[SPRING INITIALIZR 3](#_Toc78801915)

[SPRING CLI 4](#_Toc78801916)

[SPRING IDE 4](#_Toc78801917)

[SPRING BOM 4](#_Toc78801918)

[SAMLE SPRING BOOT POM.XML 5](#_Toc78801919)

[SAMPLE SPRING BOOT MAIN CLASS 5](#_Toc78801920)

[SAMLE PROJECT STRUCTURE 6](#_Toc78801921)

[REST CONTROLLER 6](#_Toc78801922)

[SERVICES 7](#_Toc78801923)

[BEAN 7](#_Toc78801924)

[SPRING BOOT – BEHIND THE SCENE 8](#_Toc78801925)

[APPLICATION.PROPERTIES 8](#_Toc78801926)

[IMPORTANT PROPERTIES 8](#_Toc78801927)

[COMPONENT SCAN - LOOKUP 8](#_Toc78801928)

[SPRING BOOT STARTER WEB 9](#_Toc78801929)

[AUTO CONFIGURATION 9](#_Toc78801930)

[SPRING JPA 9](#_Toc78801931)

[CREATING SPRING JPA PROJECT 9](#_Toc78801932)

[LOGGING IN SPRING BOOT USING SLF4J 10](#_Toc78801933)

[LOGGING IN CONSOLE 10](#_Toc78801934)

[LOGGING IN FILE 10](#_Toc78801935)

[DEPLOYING SPRING APP 11](#_Toc78801936)

[PACKAGING THE SPRING APP 11](#_Toc78801937)

[CREATING REST CLIENT 11](#_Toc78801938)

[SPRING BOOT PROFILES 12](#_Toc78801939)

[THYMELEAF 12](#_Toc78801940)

[SUBMITING THE FORM DATA 14](#_Toc78801941)

[DATABASE CACHING 15](#_Toc78801942)

[CACHING 15](#_Toc78801943)

[STEPS TO ENABLE CACHING 16](#_Toc78801944)

[SWAGGER 16](#_Toc78801945)

[STEPS TO CREATE A SWAGGER DOCUMENT 16](#_Toc78801946)

[SWAGGER UI 16](#_Toc78801947)

[VALIDATION 17](#_Toc78801948)

[ENABLING VALIDATION 17](#_Toc78801949)

# SPRING BOOT

## WHAT IS SPRING BOOT

**Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run".**

* We take an opinionated view of the Spring platform and third-party libraries so you can get started with minimum fuss. Most Spring Boot applications need minimal Spring configuration.
* ***Spring boot has an embedded Tomcat – which is bootstrap by Spring boot and deploys the code to tomcat container.***

## SPRING BOOT FEATURES

* **AUTO CONFIGURATION**
  + Spring boot does the configuration for us the type of module we are using . For example – For MVC based application – it configures the Dispatcher Servlet for us. Even for ORM module – we no longer need a to configure Data Source or Transaction Manager – as we use to do in traditional Spring Application.
* **SPRING BOOT STARTERS**
  + This is the parent project of every Spring application and it manages the dependencies.
  + It makes sure that for a particular Spring version correct version of dependent module are downloaded – rather than developer worrying about it.
* **EMBEDDED SERVLET CONTAINER**
  + Default container is Tomcat but can be switched to another container like Jetty or Undertow
* **SPRING ACTUATORS**
  + For the” Health Check” of a Spring application. Spring boot has exposed it using multiple endpoints.

## BOOTING SPRING BOOT

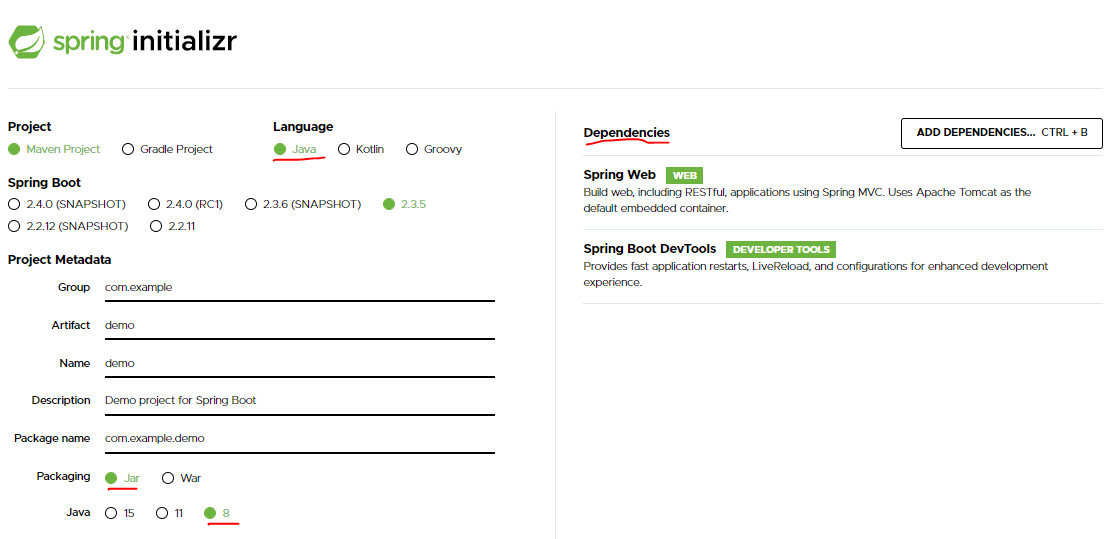
Spring boot can be created in 3 ways

### SPRING INITIALIZR

* The Spring initializr is an online UI on <https://start.spring.io/> which helps us to create a spring boot project.

The gives the configuration to select below items

* The Project Management Tool – Maven /Gradle
* Java Version (8+)
* Dependencies
  + The “Spring web” dependencies are used to create Spring MVC project.
  + “Spring boot Dev tools” helps in hot deployment of Spring App(changes deployed without restarting the server)
* Project specific properties like (Group Id, Artifacts, name)
* Packaging strategies “jar” or war



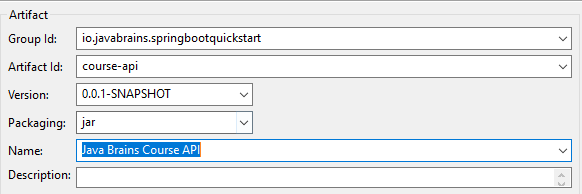
* After the above configuration we can download the zip files and import in a desired editor
* The downloaded zip project will have all the basic classes, configuration and required dependencies

### SPRING CLI

* Spring boot CLI [ <https://docs.spring.io/spring-boot/docs/current/reference/html/getting-started.html#getting-started-installing-the-cli> ]

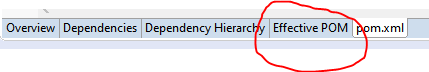
### SPRING IDE

1. File 🡪 New Maven Project 🡪 create a simple project (without archetype selection)
2. Enter the project details



## SPRING BOM

* In our project’s pom.xml we hardly define the dependency . All the default configuration and dependencies and inherited from the parent - **spring-boot-starter-parent**. The project pom just inherits the pom.xml.
* Spring has given “starter” project for each module like “spring-boot-starter-web” for Spring MVC application and ‘spring-boot-starter-data-jpa” for JPA(Hibernate) applications.
* Once these starter projects are added as dependencies – It transitively pulls all the dependent libraries. The list of Jars and its version are listed in the **spring-boot-starter-parent. This version list can be viewed in eclipse in the effective POM**



* **The spring-boot-starter-parent project – maintains the version of all the dependencies and its transitive dependencies. Hence Spring call it a BOM(Bills of Material)**

## SAMLE SPRING BOOT POM.XML

* Spring-boot-starter-parent is the parent of all Spring Boot applications.

|  |
| --- |
| <project xmlns="http://maven.apache.org/POM/4.0.0"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">  <modelVersion>4.0.0</modelVersion>  <groupId>io.javabrains.springbootquickstart</groupId>  <artifactId>course-api</artifactId>  <version>0.0.1-SNAPSHOT</version>  <name>Java Brains Course API</name>  <parent> 🡨 ***Inheritance from Parent POM***  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>2.2.2.RELEASE</version>  <relativePath /> <!-- lookup parent from repository -->  </parent>  <dependencies>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId> 🡨 **To support Spring MVC Project**  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-devtools</artifactId> 🡨 **Hot deployment**  <scope>runtime</scope>  <optional>true</optional>  </dependency>  </dependencies>  <properties>  <**java.version>1.8</java.version> 🡸 Java Version**  </properties>  <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  </plugins>  </build>  </project> |

## SAMPLE SPRING BOOT MAIN CLASS

|  |
| --- |
| import org.springframework.boot.SpringApplication;  import org.springframework.boot.autoconfigure.SpringBootApplication;  **@SpringBootApplication**  public class CourseApiApp {  public static void main(String[] args) {  SpringApplication.run(CourseApiApp.class, args);  }  } |

The above code does below things for us internally

* **SET UP THE DEFAULT CONFIGURATION**
* **START THE SPRING APPLICATION CONTEXT** 
  + Application context act as a container of all spring-based components like Controller/ Service / Repositories
  + This manages the life cycle of those components
* **PERFORM THE CLASS PATH SCAN**
  + To manage the spring-based component it must scan the files look for the Annotation which a java can has
  + It manages the components the it has been annotated. For example – the components are marked with @Service annotation are treated as service and spring always treat it as a singleton class.
* **START THE TOMCAT SERVER.**
  + Spring boot has in built tomcat container.
  + Running the Spring boot Application “main” class starts the Tomcat container too.

## SAMLE PROJECT STRUCTURE

|  |  |
| --- | --- |
|  | * The Spring Test Suite are marked by @**SpringBootTest** annotation. * It searches of for an annotation **@SpringBootApplication** – and using this class create a container – with all the beans – so that it can execute all the test cases.   Note: Due to above bean creation steps . Test methods can able to autowire the beans into the test cases. |

### REST CONTROLLER

|  |
| --- |
| **package** io.javabrains.springbootstarter.topic;  **import** java.util.List;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.web.bind.annotation.PathVariable;  **import** org.springframework.web.bind.annotation.RequestBody;  **import** org.springframework.web.bind.annotation.RequestMapping;  **import** org.springframework.web.bind.annotation.RequestMethod;  **import** org.springframework.web.bind.annotation.RestController;  @RestController  **public** **class** TopicController {  @Autowired  **private** TopicService topicService;  @RequestMapping(path = "/topics")  **public** List<Topic> getAllTopics() {  **return** topicService.getAllTopics();  }    @RequestMapping(path = "/topics/{id}")  **public** Topic getTopics(@PathVariable String id) {  **return** topicService.getTopic(id);  }    @RequestMapping(path = "/topics",method = RequestMethod.***POST***)  **public** **void** addTopic(@RequestBody Topic topic) {  topicService.addTopic(topic);  }    @RequestMapping(path = "/topics/{id}",method = RequestMethod.***PUT***)  **public** **void** updateTopic(@RequestBody Topic topic, @PathVariable String id) {  topicService.updateTopic(topic, id);  }    @RequestMapping(path = "/topics/{id}",method = RequestMethod.***DELETE***)  **public** **void** deleteTopic(@RequestBody Topic topic, @PathVariable String id) {  topicService.deleteTopic(topic, id);  }  } |

### SERVICES

If Spring Component is annotated as a service. Spring makes it a Singleton class and creates just one instance of the service which in turn used by multiple controller by auto wiring.

|  |
| --- |
| package io.javabrains.springbootstarter.topic;  import java.util.ArrayList;  import java.util.Arrays;  import java.util.List;  import org.springframework.stereotype.Service;  @Service  public class TopicService {  private List<Topic> topics = new ArrayList<>(Arrays.asList(  new Topic("spring", "Spring Framework", "Spring Framework") ,  new Topic("Java", "Core Java", "Core Java"),  new Topic("springboot", "Spring Boot", "Spring Boot"))  );  public List<Topic> getAllTopics(){  return topics;  }  public Topic getTopic(String id) {  return topics.stream().filter(t -> t.getId().equals(id)).findFirst().get();  }  public void addTopic(Topic topic) {  topics.add(topic);  }  public void updateTopic(Topic topic, String id) {  for(int topicLength = 0 ; topicLength < topics.size() ; topicLength ++) {  Topic topic2 = topics.get(topicLength);  if(topic2.getId().equals(id)) {  topics.set(topicLength, topic);  }  }  }  public void deleteTopic(Topic topic, String id) {  for(int topicLength = 0 ; topicLength < topics.size() ; topicLength ++) {  Topic topic2 = topics.get(topicLength);  if(topic2.getId().equals(id)) {  topics.remove(topicLength);  }  }  }  } |

### BEAN

|  |
| --- |
| **package** io.javabrains.springbootstarter.topic;  **public** **class** Topic {  **private** String id;  **private** String name;  **private** String description;  **public** Topic() { }  **public** Topic(String id, String name, String description) {  **super**();  **this**.id = id;  **this**.name = name;  **this**.description = description;  }  **public** String getId() {  **return** id;  }  **public** **void** setId(String id) {  **this**.id = id;  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** String getDescription() {  **return** description;  }  **public** **void** setDescription(String description) {  **this**.description = description;  }  } |

## SPRING BOOT – BEHIND THE SCENE

* **spring-boot-starter-parent:** 
  + The **Spring boot starter parent** project is parent project which gets inherited by our custom Spring boot project. It has some default configuration which will be inherited by the custom Spring boot app.
  + Due to this inheritance don’t have to do any configuration on a custom app level
* **spring-boot-starter-web**
  + The “**spring-boot-starter-web**” is dependencies we add as a dependencies of custom Spring boot Application . This will download all the web related dependencies. So now adding just one dependencies download all the related web dependencies unlike adding the dependencies individually.

The “**spring-boot-starter-web” dependencies decide what dependency to download but the version of the of those dependencies depends upon the Spring version (Spring boot starter parent)**

* **EMBEDDED TOMCAT:**

## APPLICATION.PROPERTIES

Spring boot project comes with some default configuration. We still can override those default configuration using **application.properties**

### IMPORTANT PROPERTIES

|  |  |
| --- | --- |
| spring.datasource.url  spring.datasource.username  spring.datasource.password  spring.jpa.hibernate.ddl-auto  spring.jpa.database-platform  spring.datasource.driverClassName | DATABASE PROPERTIES |
| server.port | To set custom server port numer |
| spring.jpa.show-sql | Default is FALSE. Set it to TRUE to print the SQL Query |
| server.servlet.context-path | To set the context path of the application. |
| logging.level.root | To set the log level of the application .e.g. logging.level.root=error |

* This url will have all the keys/configuration which can be used in application.properties file : <https://docs.spring.io/spring-boot/docs/current/reference/html/appendix-application-properties.html>

## COMPONENT SCAN - LOOKUP

|  |  |
| --- | --- |
| @SpringBootApplication  **@ComponentScan("com.javabrains")**  public class CourseApiApp {  public static void main(String[] args) {  SpringApplication.run(CourseApiApp.class, args);  }  } | * By default Spring scans the component ,service, controller, repository in its child package * If the Service & component do not reside in the child page. The Spring boot starter class must know the package when it must search. That can be done using @ComponentScan annotation |

## SPRING BOOT STARTER WEB

* The Spring boot started web is leverage to create Spring MVC project and Rest API
* The project has to inherit the starter web project - The inheritance is accomplished in pom.xml file using <parent> tag.

|  |
| --- |
| <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>2.2.2.RELEASE</version>  <relativePath /> <!-- lookup parent from repository -->  </parent> |

## AUTO CONFIGURATION

1. Spring Interface Injection
2. Hot reloading in+6 Spring Boot -> Add dev tools module – spring-boot-devtools
3. Deploy the Spring boot to external web server – Package it as war

## SPRING JPA

* Spring Data JPA is a separate project which lets spring with ORM based application.

### CREATING SPRING JPA PROJECT

|  |  |
| --- | --- |
| * For demo application – We will be using a in memory DB – Apache Derby * Create Spring Starter Project and select below dependencies   + Spring Data JPA   + Apache Derby or MySQL   + Spring Web |  |

#### SAMPLE CRUD JPA PROJECT

|  |  |
| --- | --- |
|  | **ENTITY**  @Entity  public class Product {  **@Id**  **@GeneratedValue(strategy = GenerationType.IDENTITY)**  private int id;  private String name;  private String description;  private Double price;  //setter & getters  } |
| **REPOSITORY INTERFACE**  **public** **interface** ProductRepository **extends** JpaRepository<Product, Integer> {  } | |
| **APPLICATION.PROPERTIES**  spring.datasource.url=jdbc:mysql://localhost:3306/springbasics?useLegacyDatetimeCode=false&serverTimezone=UTC  spring.datasource.username=root  spring.datasource.password=root  spring.jpa.hibernate.ddl-auto=update  spring.jpa.database-platform=org.hibernate.dialect.MySQL5InnoDBDialect  spring.datasource.driverClassName=  com.mysql.cj.jdbc.Driver  spring.jpa.show-sql=true  server.servlet.context-path=/productapi | |
| @RestController  **public** **class** ProductController {  @Autowired  ProductRepository productRepository;  @RequestMapping(value = "/products", method = RequestMethod.***GET***)  **public** List<Product> getProducts() {  **return** productRepository.findAll();  }  @RequestMapping(value = "/products/{id}", method = RequestMethod.***GET***)  **public** Product getProduct(@PathVariable("id") **int** id) {  **return** productRepository.findById(id).get();  }  @RequestMapping(value = "/products", method = RequestMethod.***POST***)  **public** Product createProduct(@RequestBody Product product) {  **return** productRepository.save(product);  }  @RequestMapping(value = "/products", method = RequestMethod.***PUT***)  **public** Product updateProduct(@RequestBody Product product) {  **return** productRepository.save(product);  }  @RequestMapping(value = "/products/{id}", method = RequestMethod.***DELETE***)  **public** **void** deleteProduct(@PathVariable("id") **int** id) {  productRepository.deleteById(id);  }  } | |

## LOGGING IN SPRING BOOT USING SLF4J

### LOGGING IN CONSOLE

Logging in console is the default behavior in Spring boot

|  |
| --- |
| import org.slf4j.Logger;  import org.slf4j.LoggerFactory;  @RestController  public class ProductController {  private static final Logger LOGGER = LoggerFactory.getLogger(ProductController.class) ;  @RequestMapping(value = "/products/{id}", method = RequestMethod.GET)  public Product getProduct(@PathVariable("id") int id) {  LOGGER.info("Finding the product with id="+ id); 🡨 PRINTS THE INFO LOG LEVEL IN CONSOLE  return productRepository.findById(id).get();  }  } |

### LOGGING IN FILE

|  |  |
| --- | --- |
| **OPTION 1**  logging.file.path=logs/temp/  **OPTION 2**  logging.file.name= logs/application.log | * OPTION 1: Specify the log file path using this property. Note that the log file name **is spring.log**. * OPTION 2: Specify the own log file name using this property |

## DEPLOYING SPRING APP

### PACKAGING THE SPRING APP

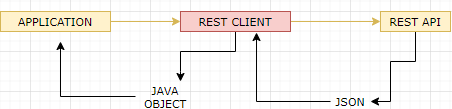
* The Spring application are standalone application which can run on its own
* **STEPS TO RUN SPRING BOOT APP**
  + Step 1: **mvn clean install** 🡪 This will create a jar/ war file in the target folder of the app
  + Step 2: java -jar <JAR\_FILE\_NAME>
* PACKAGING SPRING APP
  + The Spring app can be packaged as “jar” or “war” depending upon the pom.xml configuration

|  |  |
| --- | --- |
| PACKAGING AS JAR | PACKAGING AS WAR |
| Packaging as jar can run from command line. | Unlike jar packaging as war need a container where the app can be deployed |

## CREATING REST CLIENT

* Rest Client are the client application which are consumers of REST APIs
* We use RestTemplate Class to create Rest Clients.
* The RestTemplate class gives methods to hit the Rest APIs . These methods are specific to different HTTP methods (GET /POST etc..)- <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/client/RestTemplate.html>

|  |  |
| --- | --- |
| **HTTP METHODS** | **COMMON REST TEMPLATE METHODS** |
| GET | getForObject() |
| POST | postForObject() |
| PUT | put() |
| DELETE | delete() |



**Note: The object serialization in to JSON and de-serialization from JSON to java object happens automatically**

|  |
| --- |
| **package** com.udemy.spring.product;  **import** **static** org.junit.jupiter.api.Assertions.*assertEquals*;  **import** **static** org.junit.jupiter.api.Assertions.*assertNotNull*;  **import** org.junit.jupiter.api.Test;  **import** org.springframework.boot.test.context.SpringBootTest;  **import** org.springframework.web.client.RestTemplate;  **import** com.udemy.spring.product.entities.Product;  @SpringBootTest  **class** ProductrestapiApplicationTests {  /\* GET METHOD REST CLIENT/\*  @Test  **public** **void** testGetProduct() {  RestTemplate restTemplate = **new** RestTemplate();  Product product = restTemplate.getForObject("http://localhost:8080/productapi/products/1", Product.**class**);  *assertNotNull*(product);  System.***out***.println(product);  *assertEquals*("IPhone", product.getName());  }  /\* POST METHOD REST CLIENT/\*  @Test  **public** **void** testCreateProduct() {  RestTemplate restTemplate = **new** RestTemplate();  Product productToBeCreated = **new** Product();  productToBeCreated.setName("Laptop");  productToBeCreated.setDescription("Lenovo Laptop");  productToBeCreated.setPrice(6000d);  Product product = restTemplate.postForObject("http://localhost:8080/productapi/products", productToBeCreated, Product.**class**);  *assertNotNull*(product);  System.***out***.println(product);  *assertEquals*("Laptop", product.getName());  }  /\* PUT METHOD REST CLIENT/\*  @Test  **public** **void** testUpdateProduct() {  RestTemplate restTemplate = **new** RestTemplate();  Product productToBeUpdated = restTemplate.getForObject("http://localhost:8080/productapi/products/1", Product.**class**);  productToBeUpdated.setPrice(4000d);  restTemplate.put("http://localhost:8080/productapi/products", productToBeUpdated);  }  /\* DELETE METHOD REST CLIENT/\*  @Test  **public** **void** testDeleteProduct() {  RestTemplate restTemplate = **new** RestTemplate();  restTemplate.delete("http://localhost:8080/productapi/products/8");  }  } |

## SPRING BOOT PROFILES

* For environment (like DEV, PROD, STAGE etc.) specific configuration we use Spring Profiles.

## THYMELEAF

|  |  |
| --- | --- |
| * It’s a templating engine. * Instead of using JSP on the view for a MVC application we use HTML – which can have static and dynamic code – which in turn compiled into static HTML by Thymeleaf templating engine. * Reference : <https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html> | Java Template Engines. Introduction | by nderground | HackerNoon.com |  Medium |
| * To create a project having thymeleaf templating engine add the web and thymeleaf dependencies |  |
| * Usually for any code changes in template – We need to restart the container – it happens due to caching. * To avoid the caching add the a property in in application .properties | spring.thymeleaf.cache=false |

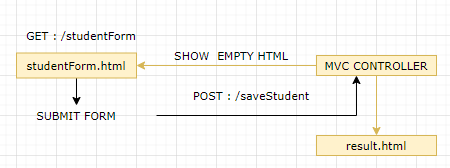
|  |  |
| --- | --- |
| import java.util.Arrays;  import java.util.List;  import org.springframework.stereotype.Controller;  import org.springframework.web.bind.annotation.ModelAttribute;  import org.springframework.web.bind.annotation.RequestMapping;  import org.springframework.web.servlet.ModelAndView;  import com.udemy.spring.thymeleafdemo.models.Student;  **@Controller**  public class HelloController {  @RequestMapping("/hello")  public String greetHello() {  return "**hello**";  } | **TEMPLATES**   * All theymeleaf templates should add the below namespace in the html tag. * In this method – we are returning “hello” , hence the file name of the template file is “***hello.html***”   <html xmlns:th=*"http://www.thymeleaf.org/"*> |
| @RequestMapping("/data")  public ModelAndView sendData() {  ModelAndView modelAndView = new ModelAndView("data");  modelAndView.addObject("message", "This is my message");  return modelAndView;  } | ***Rendering a String***   * Template file name : ***data.html***   The message is :  <div th:text=*"${message}"*></div> |
| @RequestMapping("/student")  public ModelAndView getStudent() {  ModelAndView modelAndView = new ModelAndView("**student**");  Student student = new Student();  student.setId(1);  student.setName("John");  modelAndView.addObject("student", student);  return modelAndView;  } | ***Rendering the Object***   * Template file name : ***student.html***   Name<div th:text=*"${student.name}"*></div>  Id <div th:text=*"${student.id}"*></div> |
| @RequestMapping("/students")  public ModelAndView getStudents() {  ModelAndView modelAndView = new ModelAndView("**studentlist**");  Student student = new Student();  student.setId(1);  student.setName("John");  Student student1 = new Student();  student1.setId(2);  student1.setName("Ram");  Student student2 = new Student();  student2.setId(3);  student2.setName("Alex");  List<Student> students = Arrays.asList(student,student1,student2);  modelAndView.addObject("students", students);  return modelAndView;  } | ***Rendering the list of Object***   * Template file name : ***studentlist.html*** * Below table has row – iterating the list of student data.   <table>  <th>ID<th><th>Name<th>  <tr th:each=*"student : ${students}"*> <td th:text=*"${student.id}"*></td> <td th:text=*"${student.name}"*></td></tr>  </table> |

### SUBMITING THE FORM DATA

* In the below application , the form in studentForm.html has been two way binded with Student Object and its property.

Note:

* URL are mapped using @{} – refer URL of form
* Properties are mapped using \*{} – Refer inputs field in HTML



#### CREATING EMPTY FORM

|  |
| --- |
| @RequestMapping("/studentForm")  public ModelAndView displayStudentForm() {  ModelAndView modelAndView = new ModelAndView("**studentForm**");  Student student = new Student();  modelAndView.addObject("student", student);  return modelAndView;  } |
| studentForm.html  <html xmlns:th=*"http://www.thymeleaf.org/"*>  <head>  <title>Student Data</title>  </head>  <body>  <form th:action**=*"@{/saveStudent}"*** th:object=*"${student}"* method=*"post"*>  ID: <input type=*"text"* th:field=*"\*{id}"*/>  Name: <input type=*"text"* th:field=*"\*{name}"*/>  <input type=*"submit"* value=*"Save"*/>  <input type=*"reset"* value=*"Reset"*/>  </form>  </body>  </html> |

#### DISPLAYING FORM IN RESULT HTML

|  |
| --- |
| @RequestMapping(value = "/saveStudent", method = RequestMethod.***POST***)  **public** ModelAndView saveStudent(@ModelAttribute Student student) {  ModelAndView modelAndView = **new** ModelAndView("result");  modelAndView.addObject("student", student);  **return** modelAndView;  } |
| result.html  <html xmlns:th=*"http://www.thymeleaf.org/"*>  <head>  <title>Data Saved</title>  </head>  <body>  <div th:text=*"${student.id}"*></div>  <div th:text=*"${student.name}"*></div>  </body>  </html> |

## DATABASE CACHING

* When client access our application, our application, which uses ORM tools like hibernate to read the data from the database, will execute a **select query internally** on the database table to retrieve the data.
* The **data** is then converted into an **object**, which is then passed to our application, which then sends it back to the client as required.
* Every time a client requests data from our application or ORM tool, the select statement will be executed, and the process will repeat again.
* Instead of repeatedly performing the same read operation, we employ **caching or a cache**.

### CACHING



* **Caching** stores data or objects in temporary locations. When a request comes in for the **first time**, this ORM tool or the caching framework will read the data, transform it to an object, and store it in a temporary location or on disc.
* The next time a request comes in, these ORM frameworks will check to see if the data for that request is already in the cache. If it exists, **no database select queries or database communication will be performed; simple take the object from cache, process it, and return it to the client**.
* There multiple cache provider link “**HazelCast” or “EhCache**”.
* When the objects start getting stored in the cache . if the cache size increases beyond a limit might crash the application – hence the cache needs to be cleared. This process is called “**Eviction**”. There are different policy to clear the cache like LRU, LFU, RANDOM, NONE

### STEPS TO ENABLE CACHING

1. **ADDING DEPENDENCIES**
2. **CREATE CACHE CONFIGURATION**
3. **ENABLE AND USE CACHING**
4. **EVICT CACHE**

#### ADDING MAVEN DEPEPENDCIES

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-cache</artifactId>

</dependency>

<dependency>

<groupId>com.hazelcast</groupId>

<artifactId>hazelcast</artifactId>

</dependency>

<dependency>

<groupId>com.hazelcast</groupId>

<artifactId>hazelcast-spring</artifactId>

</dependency>

#### CREATE CACHE CONFIGURATION

## SWAGGER

* It’s a way to document the Restful webservice – which show how the request -response will look like for an exposed restful webservices in JSON or YAML format.
* These documents are generated from the source code or can be handwritten using some Swagger Editor
* For handwritten use Swagger Editor : <https://editor.swagger.io/>

### STEPS TO CREATE A SWAGGER DOCUMENT

#### ADD MAVEN DEPENDENCY

* Add latest Swagger Maven dependency : <https://mvnrepository.com/artifact/io.springfox/springfox-swagger2>
* Example

<dependency>

<groupId>io.springfox</groupId>

<artifactId>springfox-swagger2</artifactId>

<version>3.0.0</version>

</dependency>

#### ENABLE SWAGGER

|  |  |
| --- | --- |
| Enable swagger from the Main Class using  @EnableSwagger2 annotation | @SpringBootApplication  @EnableSwagger2  **public** **class** ProductrestapiApplication {  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(ProductrestapiApplication.**class**, args);  }  } |

#### ACCESS SWAGGER ENDPOINT

* Access the swagger endpoint at : **/v2/api-docs**
* Example : <http://localhost:8080/productapi/v2/api-docs>

### SWAGGER UI

#### STEPS TO CREATE A SWAGGER UI DOCUMENT

#### ADD MAVEN DEPENDENCY

* Add latest Swagger Maven dependency : <https://mvnrepository.com/artifact/io.springfox/springfox-swagger-ui>
* Example

<dependency>

<groupId>org.springdoc</groupId>

<artifactId>springdoc-openapi-ui</artifactId>

<version>1.5.10</version>

</dependency>

#### ENABLE SWAGGER

|  |  |
| --- | --- |
| Enable swagger from the Main Class using  @EnableSwagger2 annotation | @SpringBootApplication  @EnableSwagger2  **public** **class** ProductrestapiApplication {  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(ProductrestapiApplication.**class**, args);  }  } |

#### ACCESS SWAGGER UI ENDPOINT

* Access the swagger endpoint at: **/swagger-ui.html**
* Example : <http://localhost:8080/productapi/swagger-api.html>

## VALIDATION

Spring boots provides annotation to validate the request data/format before it is processed.

### ENABLING VALIDATION

**ADD MAVEN DEPENDENCIES**

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-validation</artifactId>

</dependency>

# SPRING REACTIVE PROGRAMMING

|  |  |
| --- | --- |
| MAVEN  <dependencies>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>**spring-boot-starter-webflux**</artifactId>  </dependency>  <dependency>  <groupId>io.projectreactor</groupId>  <artifactId>**reactor-test**</artifactId>  <scope>test</scope>  </dependency>  </dependencies> | **ADDING DEPENDECIES in STS** |

## CONCEPT OF MONO & FLUX