Table of Contents

Step 1: Profiling in spring

1. Profiles in Spring boot is a way to define different sets of configurations for your application depending on the environment it is being run in. For example, you might have one set of configurations for your development environment and another set of configurations for your production environment. These configurations might include things like database settings (I want to use a database for tests and another for dev purposes), Bean Creation (ex: I want a bean to be created only if I'm in the development process it's possible with profiles)

2. Profiles can be defined using property files, YAML files, or even Java code. By default, Spring Boot will use the "default" profile if no other profile is specified. To activate a profile, you can set the "spring.profiles.active" property to the name of the profile you want to use.

3. The entire code is available in bitbucket repo:
   https://bitbucket.org/shalini-ws/springbootrestrepo/src/main/
   Below are just the steps followed to implement profiling in spring. Look at respective classes in the project.

a. Classes that provide different messages based on the active profile:
   EnvProfile , DevEntity , ProdEntity, DefaultEntity.
b. Controller that uses messages based on profile set:
   ProfileRestController
c. Properties files for different profiles in resources folder.
d. Add below property in application.properties file to change the active profile. If you do not give anything it is "default" profile by default. If you update then it changes to the one specified
   spring.profiles.active=production
   # spring.profiles.active=dev
e. Add below class:

```
@Configuration
public class AppConfig {

  @Bean
  @Profile("dev")
  public DevEntity devEntity(){
    System.out.println("development");
    return new DevEntity();
  }
  @Bean
  @Profile("production")
  public ProdEntity prodEntity(){
    System.out.println("production");
    return new ProdEntity();
  }
  @Bean
  @Profile("default")
  public DefaultEntity defaultEntity(){
    System.out.println("default");
    return new DefaultEntity();
  }
}
```

```
java -jar -Dspring.profiles.active=prod SpringBootRESTRepo-0.0.1-SNAPSHOT.jar
```

Step 2: Validations in spring boot

1. Add below dependency

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

2. Add below annotations on the entity class:

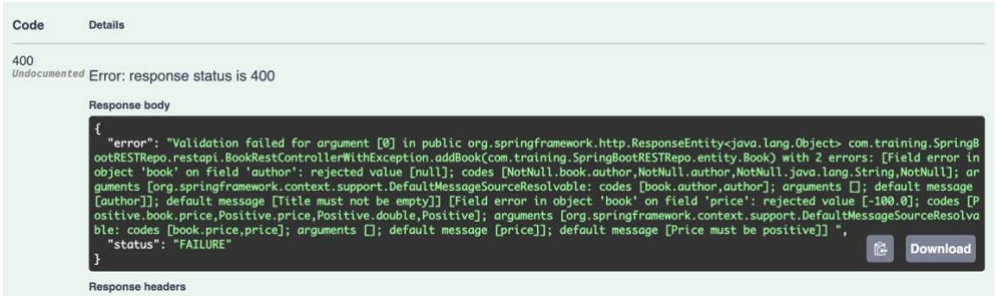
```
@NotNull(message = "Author must not be empty")
private String author;
@Column(columnDefinition = "decimal(10,2) default 100.0")
@Positive(message = "Price must be positive")
private double price;
```

3. Update the controller as follows:
   public ResponseEntity<Object> addBook(**@Valid** @RequestBody Book book)

4. Now run the code and post below json:

```
{
  "bookid": 0,
  "title": "string",
  "desc": "string",
  "price": -100
}
```

Should get an exception as follows:



5. To provide custom exception handler, add below code:

```
@ExceptionHandler({Exception.class, MethodArgumentNotValidException.class})
   public ResponseEntity<Object> handleException(Exception ex){
      Map<String, Object> map = new HashMap<>();
      map.put(AppConstants.STATUS, Status.FAILURE);

      if(ex instanceof MethodArgumentNotValidException){
         String msg = ((MethodArgumentNotValidException) ex).getAllErrors()
               .stream().map(ObjectError::getDefaultMessage)
               .collect(Collectors.joining(","));
         map.put("error",msg);
         return ResponseEntity.badRequest().body(map);
      }
      System.out.println("general exception");
      System.out.println(ex.getMessage());
      map.put("error",ex.getMessage());
      return ResponseEntity.badRequest().body(map);
   }
```

Reference:

Available annotations to use
In above example, we used only few annotations such as @NotEmpty and @Positive. There are more such annotations to validate request data. Check them out when needed.

@AssertFalse
The annotated element must be false.

@AssertTrue
The annotated element must be true.

@DecimalMax
The annotated element must be a number whose value must be lower or equal to the specified maximum.

@DecimalMin
The annotated element must be a number whose value must be higher or equal to the specified minimum.

@Future
The annotated element must be an instant, date or time in the future.

@Max
The annotated element must be a number whose value must be lower or equal to the specified maximum.

@Min
The annotated element must be a number whose value must be higher or equal to the specified minimum.

@Negative
The annotated element must be a strictly negative number.

@NotBlank
The annotated element must not be null and must contain at least one non-whitespace character.

@NotEmpty
The annotated element must not be null nor empty.

@NotNull
The annotated element must not be null.

@Null
The annotated element must be null.

@Pattern
The annotated CharSequence must match the specified regular expression.

@Positive
The annotated element must be a strictly positive number.

@Size
The annotated element size must be between the specified boundaries (included).

Step 3: Versioning in spring

What is Rest API Versioning ?

1. When we want to add extra functionality to the existing exposed api then we go for versioning of that api to provide new functionality.
2. There are some different ways to provide an API versioning in your application. The most popular of them are though :
    a. URI Path / Path Param – you include the version number in the URL path of the endpoint, for example /v1/books

    ```
    @GetMapping("/v1/books")
    @GetMapping("/v2/books")
    ```

    b. Query Parameters – you pass the version number as a query parameter with specified name, for example /books?version=1

    ```
    @GetMapping(value = "/books" , params="version=1")
    @GetMapping(value = "/books" , params="version=2")
    ```

    c. Custom HTTP Headers – you define a new header that contains the version number in the request

    ```
    @GetMapping(value = "/books" , headers="version=1")
    @GetMapping(value = "/books" , headers="version=2")
    ```

    d. Content Negotiation – the version number is included to the “Accept” header together with accepted content type.

    ```
    @GetMapping(value = "/books" , produces="application/app-v1+json")
    @GetMapping(value = "/books" , produces="application/app-v2+json")
    ```

Step 4: Logging in spring

1. In Spring Boot, the dependency spring-boot-starter-logging includes the logging frameworks. Since many Spring Boot starters include the spring-boot-starter-logging automatically, we unlikely need to add it manually.
   For example, adding the dependency web spring-boot-starter-web will automatically include the spring-boot-starter-logging.
2. By default, Spring Boot uses Logback for the logging, and the loggers are pre-configured to use console output with optional file output.
3. To use logging, check the **CharacterController** class where Logger is used.
4. To send logs to a file. Add below in application.properties as follows:
   application.properties
   logging.file.name=logs/app.log
   Log files rotate when they reach 10 MB
5. Log Levels
   TRACE
   DEBUG
   INFO
   WARN
   ERROR
   OFF

   When we set the log level to INFO (default), it logs the INFO, WARN, and ERROR log events.
   When we set the log level to DEBUG, it logs the DEBUG, INFO, WARN, and ERROR log events.
   When we set the log level to ERROR, it logs only the ERROR log events.
6. We can define the log levels in the application.properties.
   # root level
   logging.level.root=error

   # package level logging
   logging.level.org.springframework.web=debug
   logging.level.org.hibernate=error
   logging.level.com.mkyong=error
7. Read more here:
   https://mkyong.com/spring-boot/spring-boot-logging-example/
8. In details:
   https://docs.spring.io/spring-boot/reference/features/logging.html

Step 5: Testing in spring boot

There is application.properties file added within resource folder of test folder. Also data-test.sql file

Controller Layer:

1. **Mockito**
    a. Mockito is a mocking framework. It is a Java-based library used to create simple and basic test APIs for performing unit testing of Java applications.
    b. The Mockito framework's main purpose is to simplify the development of a test by mocking external dependencies and using them in the test code.

2. **Hamcrest framework**
    a. Hamcrest is the well-known framework used for unit testing in the Java ecosystem. It's bundled in JUnit and simply put, it uses existing predicates – called matcher classes – for making assertions.
    b. Hamcrest is commonly used with JUnit and other testing frameworks for making assertions. Specifically, instead of using JUnit's numerous assert methods, we only use the API's single assertThat statement with appropriate matchers.

3. @**WebMvcTest** Controller layer:
    a. SpringBoot provides @WebMvcTest annotation to test Spring MVC Controllers. Also, @WebMvcTest based test run faster because it will load only the specified controller and its dependencies only without loading the entire application.
    b. Spring Boot instantiates only the web layer rather than the whole application context. In an application with multiple controllers, you can even ask for only one to be instantiated by using, for example, @WebMvcTest(CharacterController.class).
4. @MockBean:
    a. Used to add mock objects to the Spring application context. The mock will replace any existing bean of the same type in the application context.
    b. It allows us to mock a class or an interface and record & verify its behaviors.
    c. The mock will replace any existing single bean of the same type defined in the context. If no existing bean is defined a new one will be added.
5. MockMvc:
    a. The Spring MVC Test framework, also known as MockMvc, provides support for testing Spring MVC applications. It performs full Spring MVC request handling but via mock request and response objects instead of a running server.
    b. MockMvc can be used on its own to perform requests and verify responses.
    c. To know more: https://docs.spring.io/spring-framework/reference/testing/spring-mvc-test-framework.html

Service Layer:

1. MockitoExtension
    a. is a JUnit 5 extension provided by the Mockito library. It allows you to use the Mockito framework to create and inject mocked objects into your JUnit 5 test classes.
2. Mockito @Mock
    a. annotation is useful when we want to use the mocked object at multiple places.
3. @InjectMocks:
    a. When we want to inject a mocked object into another mocked object, we can use @InjectMocks annotation. @InjectMock creates the mock object of the class and injects the mocks that are marked with the annotations @Mock into it.

DAO Layer:

1. @DataJpaTest
    a. Sometimes we might want to test the persistence layer components of our application, which doesn't require the loading of many components like controllers, security configuration, and so on.

    b. So Spring Boot provides the @DataJpaTest annotation to test the only repository/persistence layer components of our Spring boot application (The @DataJpaTest annotation doesn't load other Spring beans (@Components, @Controller, @Service, and annotated beans) into ApplicationContext).

    c. Spring Boot provides the @DataJpaTest annotation to test the persistence layer components that will autoconfigure in-memory embedded databases and scan for @Entity classes and Spring Data JPA repositories.

2. **@AutoConfigureTestDatabase**

a. As we are using the MySQL Database from TestContainers, we have to tell to spring test framework that it should not try to replace our database. We can do that by using the @AutoConfigureTestDatabase(replace=AutoConfigureTestDatabase.Replace.NONE annotation.