COMP 3095

In Class Instructions - Part 2.1

Contents

COMP 3095 – In Class Instructions

Introduction:

In this lab, we will deepen our understanding of testing and integrating MongoDB with a Spring Boot application using Docker and Postman. We will cover both manual and automated testing, focusing on CRUD operations through REST APIs. The first part of the lab involves setting up and running MongoDB containers, followed by testing REST APIs using Postman. The second part introduces automated testing with TestContainers, where we will configure a MongoDB TestContainer and write integration tests for the ProductService. By the end of the lab, you'll be equipped with the tools to test your application in both manual and automated environments, ensuring better coverage and stability of your services.

Objectives:

- Introduction to Postman
- Integration Testing of REST Services
  - GET
  - POST
  - DELETE
  - PUT

Create a new branch – Optional

1. You may find it beneficial to start a new branch at the beginning of every lesson, to track your deltas. Alternatively, you can work freely within the main branch throughout. Ultimately the choice is yours.

Pull / Run MongoDb Container

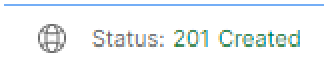
1. Run your mongo container or daemon and validate that it is listening okay (port **27017**)
   a. You can do this using the **docker cli**, **docker desktop** or even within **intellij**

Pull / Run MongoExpress

1. Run your mongo express container – GUI for MongoDB
   a. You can do this using the **docker cli**, **docker desktop** or even within **intellij**

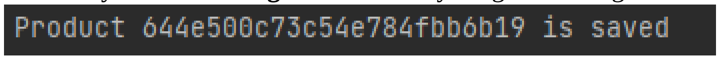
Postman - Testing the POST Rest API

1. Start your application from last lesson, and validate that your solution is running okay (port **8080**).
2. Start Postman, and create a workspace for today's tests within (call the workspace a relatable name).
3. Within your Postman workspace, create a **POST** request
   a. **http://localhost:8080/api/product**
   b. We must pass to the call data for the **ProductRequest**
      i. Postman request **Body**
      ii. **raw**, **text → JSON**
      iii. Provide data for the **ProductRequest** with the JSON body (**name**, **description**, **price**)
4. Submit the POST request, and you should have received a **201 Created** success response status code



Status: 201 Created

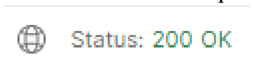
5. Validate your **console log** to ensure everything is working well.



```
Product 644e500c73c54e784fbb6b19 is saved
```

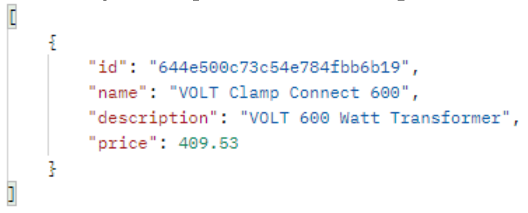
Postman - Testing the GET Rest API

1. Within your Postman workspace, create a **GET** request
   a. **http://localhost:8080/api/product**
   b. No parameters need to be passed to execute our GET
2. Submit the **GET** request, and you should have received a **201 OK** success response status code



Status: 200 OK

3. Validate your output is the set of **all products** in your database collections.



```
[
    {
        "id": "644e500c73c54e784fbb6b19",
        "name": "VOLT Clamp Connect 600",
        "description": "VOLT 600 Watt Transformer",
        "price": 409.53
    }
]
```

Postman - Testing the PUT and DELETE Rest API

1. Complete your testing using Postman for **PUT** and **DELETE**

At this point we can conclude our product-service is working well, but our testing was manually done. In realty what we want to achieve is a level of automated testing within our project.

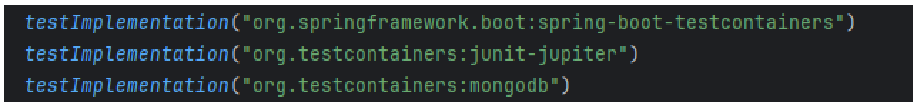
Automated Testing - TestContainers

The TestContainer library – this is a Java library that supports JUnit tests – by providing lightweight throwaway instances of common databases – or anything that can be run within a docker container. Test containers allow for the following:

- Data access layer integration tests
- Application integration tests
- UI/Acceptance tests

For now, the first (one) test container we will require is for MongoDB, so we will add test container support for mongodb to our application. Review the online TestContainers documentation for MongoDB support (**Modules → Databases →MongoDb Module**)

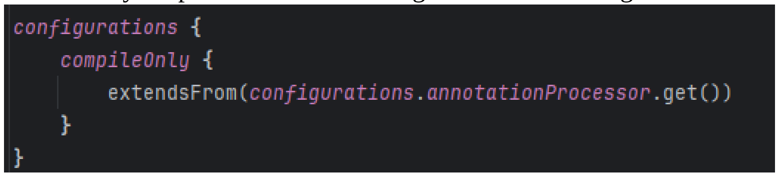
1. Add testcontainers to yourn product-service build.gradle.kts file. This should have already been added by default



```
testImplementation("org.springframework.boot:spring-boot-testcontainers")
testImplementation("org.testcontainers:junit-jupiter")
testImplementation("org.testcontainers:mongodb")
```

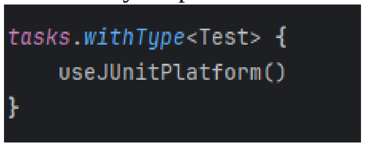
2. Add RestAssured (**io.rest-assured**) dependency to your **build.gradle.kts** file. Rest assured will allow us to make REST calls to our service, through our Integration Test.

3. Make sure your product-service build.gradle.kts has configuration of annotation preprocessing



```
configurations {
    compileOnly {
        extendsFrom(configurations.annotationProcessor.get())
    }
}
```

4. Make sure your product-service build.gradle.kts has configuration for unit testing



```
tasks.withType<Test> {
    useJUnitPlatform()
}
```

Please note every time you make changes to your **gradle.build.kts** you should load the new Gradle changes

MongoDB TestContainer: Configure for ProductService Integration Tests

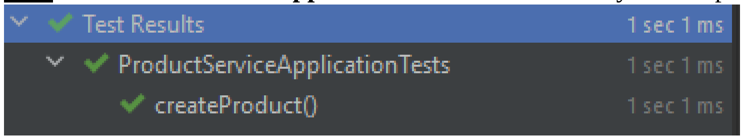
1. Edit the **ProductServiceApplicationTests** file by:
    a. Add the **@Testcontainers** annotation
    b. Instantiate an object reference to the **MongoDBContainer**, specifying the version of **mongodb docker image** you would like to use. Add **@Container** annotation
    c. Define **setProperties()** to provide **URI** dynamically at the point of integration testing.

Author Integration Test for POST ProductService

We will want to write test against each of our endpoints that we authored in our **ProductController**

1. Write a test to validate **createProduct() POST** request
   a. Create **createProduct()**
   b. Create **getProductRequest()**

2. **Run** the **ProductServiceApplicationTests** to make sure your test passes correctly.



```
✓ Test Results                              1 sec 1 ms
  ✓ ProductServiceApplicationTests          1 sec 1 ms
    ✓ createProduct()                       1 sec 1 ms
```

Update Repository

1. **Commit** and **push** all your code to your **private git repository.**

Conclusion

By completing this lab, you should now have hands-on experience in setting up and testing a Spring Boot application with MongoDB through both manual and automated testing. You've explored using Docker to run MongoDB containers, Postman to interact with REST APIs, and TestContainers for dynamic, automated integration testing. These skills are essential for creating reliable, scalable applications that can be thoroughly tested in different environments. Remember to continually practice these workflows, as they are vital components of modern software development.