# Assignment 3

## Overview

In this lab you’ll enhance the “online retailer” application as follows:

* Test components, repositories, and REST controllers
* Implement security using OAuth2
* Containerize the application
* Implement message queuing using Kafka

## IntelliJ projects

Starter project: **assignment-student/assignment3**

Solution project: **assignment-solutions/assignment3**

**Roadmap**

There are 4 exercises in this lab. Each exercise is independent of the others, so you can tackle the exercises in any order. The *solution* project contains full solutions for everything. Here is a brief summary of the tasks you will perform in each exercise; more detailed instructions follow later:

1. Testing the application
2. Implementing security
3. Containerizing the application
4. Implementing message queuing

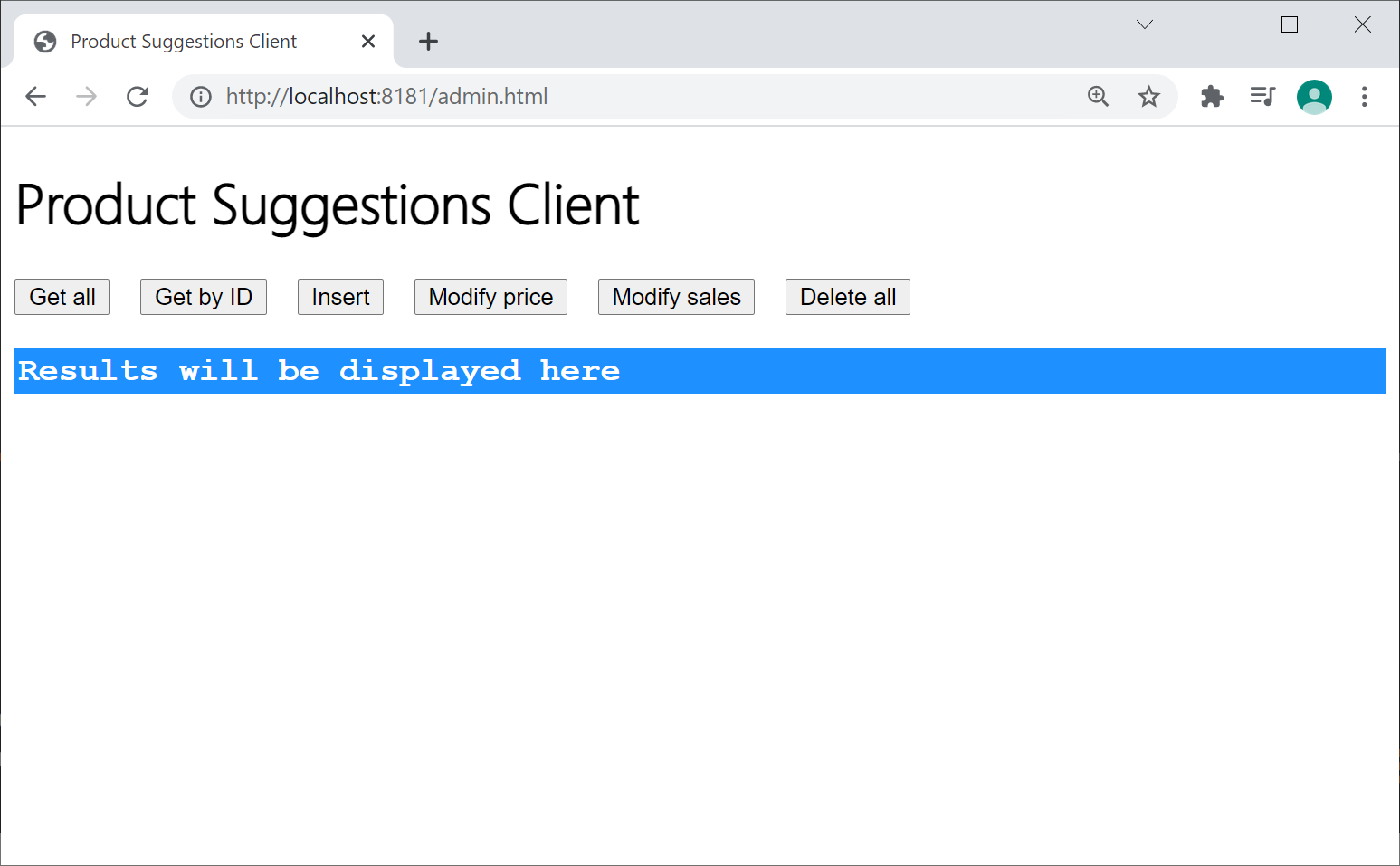
## Familiarization

Start IntelliJ and open the **assignment-student/assignment3** project. The code in this project is very similar to the *solution* project for Assignment 2, so take a moment to reacquaint yourself with the details.

There are a few differences between this project and the solution to Assignment 2:

* The web application now runs on port 8181 (rather than port 8080), as specified by the **server.port** property in the **application.properties** file. We made this change to ensure this application doesn't interfere in any way with the one from Assignment 2, if you have them both running at the same time.
* We've implemented a simple HTML web client for the "product suggestions" service, to make it easier to add, remove, and modify product suggestions. If you're interested in seeing the code details, take a look in the **src/main/resources/static** folder and explore **admin.html**, **script.js**, and **styles.css**.

To get a feel for how the web client works, run the **Application.java** class and then open a browser and browse to <http://localhost:8181/admin.html>. The web page looks like this:



Each button calls a REST endpoint on the back-end "product suggestions" controller in the Spring Boot application. For example, click *Insert* to insert a new product suggestion; you'll be prompted to enter a product description, price, and sales. Now click *Get all* to verify the item was inserted successfully. Try all the buttons on the web page and verify they all work as expected.

## Exercise 1: Testing the application

In this exercise you'll write tests for the following components in the application. Each set of tests will entail different techniques and we'll explore all the details as we go along. You can do all or some of these tests, as you see fit:

* **CartRepositoryImpl** (illustrates simple unit testing)
* **CartServiceImpl** (illustrates how to mock beans)
* **ProductSuggestionRepository** (illustrates testing a CrudRepository)
* **ProductSuggestionController** (integration testing a REST controller)

## Testing CartRepositoryImpl

Take a look at **CartRepositoryImpl** in the source code folder. This class holds a **HashMap** of items in the user's shopping cart, in memory.

In the test folder, add a class named **CartRepositoryImplTests** to test the class. Here are some hints and suggestions:

* You might be tempted to autowire a **CartRepositoryImpl** bean into your test class, but this isn't a good idea because **CartRepositoryImpl** is stateful (i.e. it stores data in memory). If you use autowiring, Spring Boot will autowire the same bean for all your test methods, which means the test methods will cumulatively change the state of the bean. Thus, test method #2 would need to know what happened in test method #1, and so on. Tests are meant to be independent of each other, so this approach isn't appropriate.
* A better approach when you have a stateful bean is to manually create your object yourself, i.e., using **new**. This way, a fresh object will be created for each test, so the test methods are independent of each other.
* Write test methods in **CartRepositoryImpl** as follows:
  1. **cart\_emptyInitially()**Test that the shopping cart is empty initially.
  2. **addItems\_itemsAdded()**Test that if you add different item IDs to the cart, each item is held as a separate entry in the map.
  3. **addSameItem\_countIncremented()**Test that if you add the same item ID to the cart several times, the item quantity is incremented (rather than inserting a separate entry in the map).
  4. **removeItem\_itemRemoved()**Test that if you remove an item from the cart, the item entry is completely removed from the cart (rather than just decrementing the quantity).

## Testing CartServiceImpl

Take a look at **CartServiceImpl** in the source code folder. This class provides business methods for managing the shopping cart, e.g., calculating the total cost of the shopping cart etc. **CartServiceImpl** uses an autowired **CartRepository** to handle all persistence.

In the test folder, add a class named **CartServiceImplTests** to test the class. Here are some hints and suggestions:

* Autowire a **CartServiceImpl** bean into your test class. This is an appropriate approach now (rather than manually creating a **CartServiceImpl** object yourself) because it's a stateless bean. Spring Boot will create a **CartServiceImpl** bean and use the same bean for all your tests. This is fine now because the bean is stateless, so it's OK to reuse the same bean in all the test methods.
* When Spring Boot creates the **CartServiceImpl** bean, it needs to autowire in a **CartRepository** bean. It's important it doesn’t autowire a "real" repository bean because that might entail database access etc., which is inappropriate in a unit test. Therefore, in your test class, use the **@MockBean** annotation to create a mock **CartRepository** bean. Spring Test will automatically inject this mock bean into your **CartServiceImpl** bean.
* Write test methods in **CartServiceImpl** as follows:
  1. **addItemToCart\_itemAdded()**Call **addItemToCart()** on your cart service bean. Specify a valid item ID between 0 and 4 inclusive (these are the valid item IDs in the catalog, as defined in the **catalog** bean in **Application.java**).

If the cart service bean method is implemented correctly, it should call **add()** on the cart repository bean. To verify it did this, call the Mockito **verify()** function and verify that **add()** was invoked upon the mock cart repository bean, with the ID and quantity you specified. For example:

**verify(mockRepo).add(eq(1),eq(100));**

* 1. **addUnknownItemToCart\_noAction()**Call **addItemToCart()** on your cart service bean with an unknown item ID. In this case it's important the cart service bean method *doesn't* call **add()** on the cart repository bean. You can verify this as follows:

**verify(mockRepo,times(0)).add(anyInt(),anyInt());**

* 1. **removeItemFromCart\_itemRemoved()**Call **removeItemFromCart()** on your cart service bean with a valid item ID. The cart service bean should call **remove()** on the cart repository bean in this case. Verify this is what happens.
  2. **removeUnknownItemFromCart\_noAction()**Call **removeItemFromCart()** on your cart service bean with an unknown item ID. In this case, the cart service bean should not call **remove()** on the cart repository bean. Verify this is the case.
  3. **calculateCartCost\_correctCostReturned()**In this test you'll confirm that the cart service bean calculates the total cart cost successfully. To make this calculation, the cart service bean must first call **getAll()** on the cart repository bean. In this unit test, you're using a mock cart repository bean, so you must tell Mockito what to return when the **getAll()** function is called. You can do this as follows:

**Map<Integer, Integer> cart = new HashMap<>();**

**cart.put(2, 1);**

**cart.put(3, 2);**

**cart.put(4, 5);**

**when(mockRepo.getAll()).thenReturn(cart);**

Once you've done this, you can proceed to call **calculateCartCost()** on the cart service bean. Assert that it returns the correct value.

## Testing ProductSuggestionRepository

Take a look at **ProductSuggestionRepository** in the source code folder. It's a simple CrudRepository interface with a couple of queries to modify product suggestions in the database.

In the test folder, add a class named **ProductSuggestionRepositoryTests** to test the interface. Here are some hints and suggestions:

* Annotate the test class with **@DataJpaTest**. As you will recall from the chapter, this annotation sets up a minimal set of beans to help with testing an in-memory database.
* Autowire a **ProductSuggestionRepository** bean into the test class. You'll write methods to test this bean shortly.
* Also autowire a **TestEntityManager** bean into the test class. You'll use this bean to populate the database with sample data, for use in the tests.
* Write test methods in **ProductSuggestionRepository** as follows (note that Spring Boot will start/rollback a new transaction for each test method, to ensure the data in the database is unaffected after each test):
  1. **testModifyPrice()**  
     First, insert a product suggestion into the database (call **persist()** on the **TestEntityManager** bean to do this). Then call **modifyPrice()** on your repository bean, to modify its *price* value in the database. To confirm the update worked successfully, call **findById()** on your repository bean to fetch it from the database, then assert it has the new price.
  2. **testModifySales()**  
     First, insert a product suggestion into the database (call **persist()** on the **TestEntityManager** bean to do this). Then call **modifySales()** on your repository bean, to modify its *sales* value in the database. To confirm the update worked successfully, call **findById()** on your repository bean to fetch it from the database, then assert it has the new sales value.

## Testing ProductSuggestionControllerTests

Take a look at **ProductSuggestionController** in the source code folder. It's a REST controller to query and modify product suggestions.

In the test folder, add a class named **ProductSuggestionControllerTests** to perform *integration testing* on this REST controller. Here are some hints and suggestions:

* Annotate the test class with **@SpringBootTest**. On this annotation, set the property **webEnvironment=SpringBootTest.WebEnvironment.RANDOM\_PORT**. This causes Spring Boot to start a real (Tomcat) web server on a random port, which is a good test of reality in an integration test. (Note that the default value of **webEnvironment** is **SpringBootTest.WebEnvironment.MOCK**, which loads a mock web server rather than a real one; this is OK for unit testing, but not good enough for integration testing).
* Also annotate the test class with **@TestMethodOrder**. On this annotation, pass in **MethodOrderer.OrderAnnotation.class** as a parameter. This is a JUnit mechanism that indicates tests should be invoked in the order specified by **@Order** annotations. You'll see why this is important shortly…
* Autowire a **TestRestTemplate** bean into the test class. You'll use this bean to call REST endpoints in your tests.
* Write test methods in **ProductSuggestionControllerTests** as follows. Annotate each test method with **@Order(x)**, where x is sub-bullet number:
  1. **testInsertProductSuggestion()**This test will execute first, and it will verify that a product suggestion can be inserted successfully. In this test:
     + Create a sample **ProductSuggestion** object.
     + Send a POST request to the **/productSuggestions** REST endpoint to insert the product suggestion object.
     + Verify that the HTTP response is correct, i.e., the status code should be 201 and the HTTP response body should contain the product suggestion with a valid ID.
     + You'll need the ID for subsequent test methods, so stick it into a static variable (for example) so you can use it later.
  2. **testGetAllProductSuggestions()**Send a GET request to the **/productSuggestions** REST endpoint. Verify it returns a status code of 200, and the HTTP response body contains collection of all the product suggestions (the collection should just contain the single item you inserted in test 1 above).
  3. **testGetProductSuggestion()**Send a GET request to **/productSuggestions/<id>** (where <id> is the ID of the item you inserted in test 1 above). Verify it returns a status code of 200, and the HTTP response body contains the correct product suggestion details.
  4. **testModifyPrice()**Send a PUT request to a URL such as the following:

**/productSuggestions/modifyPrice/<id>?newPrice=<price>**

where <id> is the ID of the item you inserted in test 1 above, and <price> is a suitable new price value.

To test that it worked, issue a GET request to get the product suggestion back again, and verify its price value has been updated successfully.

* 1. **testModifySales()**Send a PUT request to a URL such as the following:

**/productSuggestions/modifySales/<id>?newSales=<sales>**

where <id> is the ID of the item you inserted in test 1 above, and <sales> is a suitable new sales value.

To test that it worked, issue a GET request to get the product suggestion back again, and verify its sales value has been updated successfully.

* 1. **testDeleteAll()**Send a DELETE request to **/productSuggestions**. To test that it worked, issue a GET request to get all product suggestions, and verify the collection is empty.

**Exercise 2: Implementing security**

In this exercise you'll implement security in your Spring Boot application, using OAuth2. You'll use GitHub to authorize users (if you don't yet have a GitHub account, go to <https://github.com/> now and sign up for free).

Register your Spring Boot application with GitHub as follows:

* Sign in to <https://github.com/settings/developers>
* Click *OAuth apps*, then click *New OAuth App.*
* Specify the following details for your application:
  + Application name **Online Retailer app**
  + Homepage URL [**http://localhost:8181**](http://localhost:8181)
  + Authz callback URL [**http://localhost:8181/login/oauth2/code/github**](http://localhost:8181/login/oauth2/code/github)
  + Then click *Register Application*.
* In the next screen, grab the following credentials (you'll need these in a moment):
  + **Client ID**
  + **Client secret**

Now you can enhance your application to use OAuth2 authentication, courtesy of GitHub for user authentication. Follow these steps (see the chapter notes if you need a reminder):

* Add the OAuth2 starter dependency in your pom file.
* In **application.properties**, add properties for the GitHub client ID and the GitHub client secret (use the values from above)

Your next step is to define a configuration class to specify which parts of your web app are restricted and which parts are unrestricted. Add a class named **SecurityConfig** as follows:

**@Configuration**

**public class SecurityConfig {**

**@Bean**

**public SecurityFilterChain filterChain(HttpSecurity http)**

**throws Exception {**

**return http**

**.** **authorizeHttpRequests()**

**.requestMatchers("/admin.html").authenticated()** **// Note 1**

**.requestMatchers("/\*\*").permitAll() // Note 2**

**.anyRequest().anonymous() // Note 3**

**.and()**

**.csrf().disable().cors() // Note 4**

**.and()**

**.oauth2Login() // Note 5**

**.and()**

**.build();**

**}**

**}**

Notes

1. This clause indicates that the address */admin.html* is only accessible to authenticated users. If the user isn't currently authenticated, Spring Security will redirect to GitHub (in our example). GitHub will challenge the user to authenticate themselves, and will return an access token to Spring Security, which will then allow the page to be displayed.
2. This clause indicates all other addresses are permitted, regardless of authentication info.
3. This clause indicates any request is accessible to anonymous users (i.e., unauthenticated).
4. This clause is a bit tricky. In a nutshell, it allows JavaScript code running in the browser to make cross-origin requests (e.g., to REST endpoints in the Spring Boot app).
5. This clause tells Spring Boot to use OAuth2 authentication.

Build and run the application and try browsing to the following addresses:

* <http://localhost:8181/productSuggestions> (this should proceed uninhibited)
* <http://localhost:8181/admin.html> (this should ask you to authenticate via GitHub)

## Exercise 3: Containerizing the application

In this exercise you'll containerize your Spring Boot application. Follow these steps:

* If you haven't already done so, install Docker on your computer (e.g., Docker Desktop for Windows). Ensure Docker is running, e.g., open a Command Prompt window and run **docker** **version** and make sure it succeeds.
* Build your application into a JAR file.
* Add a **Dockerfile** to the root folder in your application, and specify the build instructions as described in the chapter notes.
* Open a Command Prompt window in the root folder of your application, and do a **docker** **build** to build your Docker image.
* Still in the Command Prompt window, do a **docker** **run** to run your application in a Docker container. Remember to set the **-p** option, to map port 8181 (in the container) to an available port on your host machine (e.g., 8123).
* Open a browser and browse to a URL such as <http://localhost:8123/productSuggestions>, to ping your containerized application. Verify it works correctly.

## Exercise 4: Implementing message queuing

In this exercise you'll implement Kafka message queuing in your Spring Boot application. Follow these steps:

* If you haven't already done so, install Kafka on your computer and tweak the Zookeeper and Kafka configuration as described in the chapter. Then start Zookeeper and Kafka.
* In your Spring Boot application, add the Kafka dependency in your pom file.
* Enhance the **ProductSuggestionController** class so that it publishes messages to a topic named "product\_suggestions\_topic" whenever anything interesting happens. For example:
  + When the user inserts a new product suggestion, publish a message as follows:  
      *Key* "inserted"  
     *Value* product suggestion details
  + When the user modifies a product price value, publish a message as follows:  
      *Key* "modifiedPrice"  
     *Value* ID and new price
  + When the user modifies a product sales value, publish a message as follows:  
      *Key* "modifiedSales"  
     *Value* ID and new sales
* Define a component class named **ProductSuggestionTopicListener** that listens for all messages published to the "product\_suggestions\_topic". Whenever it receives a message, simply display the message on the screen to indicate it has been received. In a real application you would do something semantically important at this juncture.

Build and run the application in IntelliJ. Browse to *admin.html* and add some product suggestions and modify their details. Then take a look in the IntelliJ console and verify messages have been published and received on the Kafka topic, such that the IntelliJ console displays the messages received.

To wrap up this assignment, open the *solution* project in IntelliJ. We've implemented 2 versions of the "product suggestions" REST controller - one that supports Kafka messaging, and one that doesn't. We've annotated the two controller classes with **@Profile**, so that the application either uses (or doesn't) use Kafka depending on which active profile is set. Now take a look in **application.properties** to see what profile is active. See what happens if you change this property. This is a very realistic example of the use of profiles to influence how Spring Boot applications work in practice 👍.

**The end 😊**