INSTRUCTIONS

Spring Boot Testing

Purpose

In this lab you will gain experience with Spring Boot testing, which provides a set of annotations and utilities to help you test your Spring Boot application.

Learning Outcomes

You will learn how to:

- Enable Spring Boot testing
- Perform integration testing without having to manually run your application
- Perform Web slice testing using mocking

You will be using the *40-boot-test* project.

Estimated time to complete: 45 minutes.

Use Case

Testing is a critical part of the software development. For Spring Boot applications, writing and running test code gets easier and faster through the use of various testing utilities and annotations provided by Spring Boot Testing.

Instructions

Add Spring Boot Testing Starter

1. Note that the `pom.xml` and `build.gradle` of the parent project already has `spring-boot-starter-test`

   Note that the starter adds several libraries including Mockito framework.

Perform Integration Testing

Run integration testing without using Spring Boot Testing

Without the use of Spring Boot Testing, you will have to start your application manually before running integration tests. The existing tests in the `AccountClientTests` are examples of this.

1. Stop any application that is running on port 8080.
2. Run existing tests in the `AccountClientTests` and observe that they fail. (If you are using Gradle, remove test exclude statement from `build.gradle` befor running the test.)
3. Start the application. And run the tests again and observe that they now pass.
4. Stop the application since we will not need a manually started application.

Run integration testing using Spring Boot Testing

You are going to refactor `AccountClientTests` to use Spring Boot Testing.

TODO-01 Make this test Spring Boot test class

1. Use `@SpringBootTest` annotation with `WebEnvironment.RANDOM_PORT`.

```
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class AccountClientTests {
    ...
}
```

   Note that `@SpringBootTest` registers a `TestRestTemplate` bean.

TODO-02 Use `TestRestTemplate` bean

1. Autowire `TestRestTemplate` bean.

```
@Autowired
private TestRestTemplate restTemplate;
```

TODO-03 Update the code to use `TestRestTemplate` bean

1. Remove `RestTemplate` from the code since it is no longer needed.
2. Remove `BASE_URL` and its use in the code since `TestRestTemplate` will use relative path.
3. Run the tests and observe that the tests pass except `addAndDeleteBeneficiary` test.

TODO-04 Handle an error status from the server

1. Modify `addAndDeleteBeneficiary()` method and run the tests again.

```
@Test
public void addAndDeleteBeneficiary() {
    // perform both add and delete to avoid issues with side effects
    String addUrl = "/accounts/{accountId}/beneficiaries";
    URI newBeneficiaryLocation = restTemplate.postForLocation(addUrl, "Dav
    Beneficiary newBeneficiary = restTemplate.getForObject(newBeneficiaryL
    assertThat(newBeneficiary.getName()).isEqualTo("David");

    restTemplate.delete(newBeneficiaryLocation);

    ResponseEntity<Beneficiary> response =
        restTemplate.getForEntity(newBeneficiaryLocation, Beneficiary.clas
    assertThat(response.getStatusCode()).isEqualTo(HttpStatus.NOT_FOUND);
}
```

   TestRestTemplate is, by design, fault tolerant. This means that it does not throw exceptions when an error response (400 or greater) is received.

   This makes it easier to test error scenarios as, rather than having to catch an exception, you can simply assert that the response's status code is as expected for the scenario in question.

2. In your IDE, run all the tests in `AccountClientTests` - they should all pass.

   > Note: The tests are disabled in the maven POM, since they cannot run without a server to talk to, so you should run them from the IDE. If you are interested - look for the `maven-surefire-plugin` to see how to disable tests.

TODO-05 Observe that Tomcat server gets started as part of testing

1. Observe, in the console log, that Spring Boot Testing framework started an embedded Tomcat server on a random port.

```
INFO : o.s.b.w... - Tomcat started on port(s): 54209 (http) ...
```

Perform Web Slice Testing

TODO-06 Get yourself familiarized with various testing utility classes

To test Spring MVC Controllers, you can use `@WebMvcTest`. It auto-configures the Spring MVC infrastructure (and nothing else) for the web slice tests.

Note that the Web slice testing runs faster than an integration testing since it does not need to start a server.

In order to do this part of the lab, you need to know how to use the following API's:

- `BDDMockito`: given(..), willReturn(..), willThrow(..)
- `MockMvc`: perform(..)
- `ResultActions`: andExpect(..)
- `MockMvcRequestBuilders`: get(..), post(..), put(..), delete(..)
- `MockMvcResultMatchers`: status(), content(), jsonPath(..), header()

Create controller test class with `@WebMvcTest`

TODO-07 Write slice test code for the controller

1. Go to `AccountControllerBootTests` testing class under `accounts.web` package in `src/test/java`.

2. Use `@WebMvcTest` annotation

```java
@WebMvcTest(AccountController.class)
public class AccountControllerBootTests {
  // Test code
}
```

   Note that `@WebMvcTest` auto-configures `MockMvc` bean.

TODO-08 Autowire `MockMvc` bean

TODO-09 Create `AccountManager` mock bean using `@MockBean` annotation

Write test for `GET` request for an Account

TODO-10 Write positive test for `GET` request for an account

1. Write test for getting account detail for a valid account with `arrange/act and assert/verify` structure as shown below.

```
@Test
public void testAccountDetails() throws Exception {

    // arrange
    given(..).willReturn(..);

    // act and assert
    mockMvc.perform(get("/URL-TO-FETCH"))
           .andExpect(..);

    // verify
    verify(..).getAccount(0L);

}
```

The testing should verify the following:

- The returned Http status is 200
- The returned content type is JSON (MediaType.APPLICATION_JSON)
- The returned data contains correct `name` and `number` of an account

Make sure you are using the correct imports for `@Test` and `jsonPath`

```
import org.junit.jupiter.api.Test;
import static org.springframework.test.web.servlet.result.MockMvcResultMat
```

TODO-11 Write negative test for `GET` request for an account

1. Write test for getting account detail for a non-existent account. The testing should verify the following:
    - The returned Http status is 404

Write test for `POST` request for Account

TODO-12 Write test for `POST` request for an account

1. Write test for creating a new account. The testing should verify the following:
    - The returned Http status is 201
    - The `Location` header should contain the URI of the newly created Account

For converting an Account to JSON string, add and use the following method:

```
public static String asJsonString(final Object obj) {
    try {
        final ObjectMapper mapper = new ObjectMapper();
        final String jsonContent = mapper.writeValueAsString(obj);
        return jsonContent;
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

Experiment with @MockBean vs @Mock (Optional)

TODO-13 (Optional) Experiment with @MockBean vs @Mock

1. Change `@MockBean` to `@Mock` for the `AccountManager` dependency.
2. Run the test. Observe that the test fails.
3. Reverse the change so that the test passes again.

Summary

In this lab, you have used Spring Boot Testing framework for integration testing and Web slice testing.

Extra Part 1

If you are finished with this lab before the rest of the class, try doing some or all of the following:

Test Get All Accounts

1. Add another web slice test to `AccountControllerBootTests` that uses `MockMvc` for getting all accounts. The test should verify the following:
   - The returned Http status is 200
   - The returned content type is JSON (MediaType.APPLICATION_JSON)
   - The returned data contains the correct `name`, `number`, and contains the correct number of accounts (at least 21 - maybe more due to the POST test creating new accounts)

Test `GET`/`POST`/`DELETE` for Beneficiary

1. Write test for getting a valid beneficiary for an account. The testing should verify the following:

- The returned Http status is 200
- The returned content type is JSON (MediaType.APPLICATION_JSON)
- The returned data contains correct `name` and `allocationPercentage` of a beneficiary

2. Write test for getting an non-existent beneficiary for an account. The testing should verify the following:

   - The returned Http status is 404

3. write test for adding a new beneficiary to an account. the testing should verify the following:

   - the returned http status is 201
   - the `location` header should contain the uri of the newly added beneficiary

4. Write test for removing a beneficiary from an account. The testing should verify the following:

   - The returned Http status is 204

5. Write test for removing a non-existent beneficiary from an account. The testing should verify the following:

   - The returned Http Status is 404