Technische Universität München
Fakultät für Informatik

Exercise Sheet 9: Message-oriented Architectures

Welcome to the ninth week of Advanced Topics of Software Engineering. In this week's theoretical exercise, we will practice concepts from the message-oriented architectures. First, we will explore different messaging models that use message queues. Then, we will work with message brokers to optimize the recommendation module of an online book store.

For the practical exercise, we will see how to obtain code coverage and create Unit, Integration, and System tests for an example project. We will also setup a Continuous Integration (CI) pipeline in Gitlab so that each code commit must pass the tests in order to be pushed.

Theoretical Exercise

Exercise 1: Messaging Models

In the lecture, you have learned about different messaging models (patterns) that use message queues. As you know, message queues store the messages sent by Message-oriented middleware (MOM) clients and dispatch them when the recipient is ready to process the new message.

1. One of the patterns that use message queues is the *point-to-point* (p2p) pattern. In this pattern, only a single consumer (receiver) retrieves the message sent by the producer (sender), although the same message queue can be listened to by multiple receivers. Come up with two possible scenarios where using p2p pattern makes sense. Explain why this pattern fits them. (Figure 1 shows a possible p2p model.)

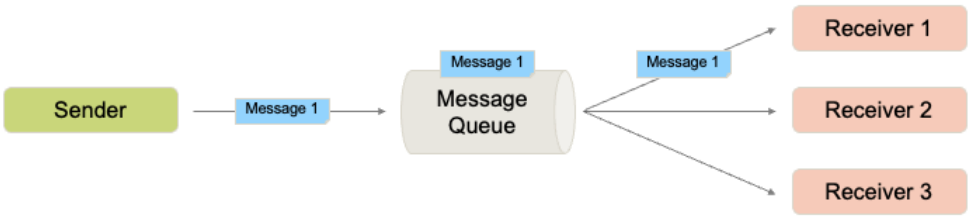

Figure 1: An example point-to-point messaging model figure.

2. Another pattern that use message queues, called "topics" now, is the *publish/subscribe* (pub-sub) pattern. In this pattern, consumers (subscribers) can listen to multiple producers (publishers), and consume all messages that are published by them. Come up with two possible scenarios where using pub-sub pattern makes sense. Explain why this pattern fits them. (Figure 2 shows a possible pub-sub model.)

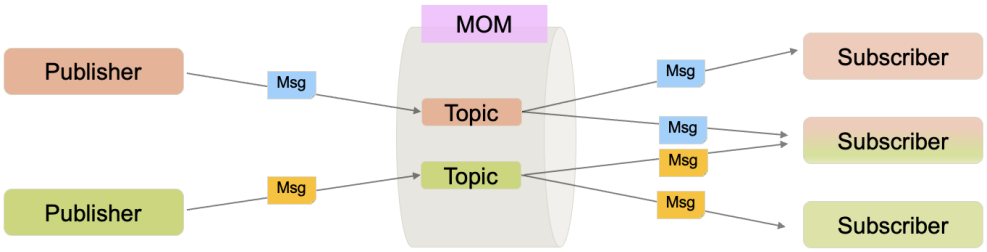

Figure 2: An example publish/subscribe messaging model figure.

Exercise 2: Message Brokers

You are a software developer at *Kafkaesque*, an online book store that is known for its bizarre user experience. After countless complaints from customers regarding getting irrelevant book recommendations about huge insects, your boss Mr. Samsa, finally decided to make some changes. He asks you to:

- optimize the integration between the web application and the real-time customer behavior analysis module such that customers receive relevant book recommendations while browsing the website,
- create a new module that runs overnight, analyzing the customer behavior information gathered by the web application, and sending an email containing *Top 10 most visited books of the day!* to customers who subscribed to the newsletter.

You know from your college days that you can use *message brokers* for creating a communication infrastructure that allows the integration of different system entities.

1. Identify the possible applications/services in which the message broker will act as a central router in between. Explain the role of each application/service.
2. Determine which messaging model can be used and how. Draw a possible system design.

Practical Exercise

After having discussed the notion of *software testability* in the lecture, the practical and project exercises on this sheet focus on *software testing*. The purpose of software testing is to assess the quality of a piece of software, by checking whether it is producing correct/incorrect results for a specific input and an expected output (e.g. taken from the specification). Test cases can typically consist of multiple checks and are generally part of a larger entity, called a test suite. In OO languages, a test case will often be equivalent to a method, whereas a test suite is represented by a class.

Besides checking your code for functionality, tests can have a largely positive influence on the maintainability of your code. When you have thoroughly tested a specific component, it will be easier to refactor parts of the code later, as you should have a clear indication when tests are failing afterwards, that some observable behavior was changed by your code refactoring. Additionally, understandability could be increased as test cases provide insights on how to use your APIs with test data to other developers.

Talk is cheap. Show me the code.[1]

Software testing often involves a lot of tooling ranging from full-featured frameworks which facilitate preparing, writing, running or analyzing test cases and their context, over code coverage software, to continuous integration (CI) tools. Writing test code is non-trivial and there are specific techniques such as stubbing (i.e. mocking) and conventions for writing meaningful test cases that need to be practiced. Hence, in this week's practical exercises, you will learn how to write test code in Java with JUnit and Spring Test on different levels of the testing pyramid and how to run them locally and in CI environments using Maven.

What's in it for the OnlineIDE?

[1] Linus Torvalds (2000) on Linux-Kernel Mailing List, https://lkml.org/lkml/2000/8/25/132

In this week's project exercises, you will implement test cases for your compilation microservice. This is one example for how to test a component on different levels of abstraction and isolation, but there are certainly no limits to your imagination when creating test cases. Furthermore, after completion of the exercises, you will have a CI infrastructure set up to run your test cases continuously after each code change in your OnlineIDE's repository. Ultimately, this will give you an understanding of how to continue testing your other microservices, in isolation or in combination with other external entities.

Caveat: Developers tend to get frustrated and ignorant when writing tests as it can be a tedious and repetitive task and also, it is never finished. Do not pay attention to these lousy fools. They will be the ones arguing that you should simply skip this one failing test case for this one minor release.

Instead, let's get the testing started!

Exercise 3: Introduction to JUnit and JaCoCo (Code Coverage)

JUnit[2] is the de-facto standard unit testing framework for Java. First-class support for the JUnit Platform exists in popular IDEs (IntelliJ IDEA, Eclipse, Visual Studio Code, ...) and build tools (Maven, Gradle, Ant).

There are plenty of naming strategies when it comes to writing test cases. In the following, we will stick to one of the two conventions:

- Should_ExpectedBehavior_When_StateUnderTest
- test[Feature being tested]

Furthermore, there are conventions for writing tests, such as the Arrange-Act-Assert or Given-When-Then styles, which are emphasized in Test-Driven-Development (TDD) and Behavior-Driven-Development (BDD). The essential idea is to break down writing a test (or scenario) into three sections[3]:

1. Set up test data or environment and expected outcomes
2. Call method/system under test which returns actual outcomes
3. Assert that expected match actual outcomes

In this exercise you are given a Maven project `exercise_08/teststarter` in the ASE repository which uses JUnit with a class `edu.tum.ase.teststarter.SuperMath` that you have to write test cases for in `src/test/edu/tum/ase/teststarter/SuperMathTest.java`.

Your first task is to look at the `findFactors(...)` method:

```java
public static ArrayList<Integer> findFactors(long input) {
  ArrayList<Integer> factors = new ArrayList<Integer>();
  int i = 1;
  while (i <= input) {
    if (input % i == 0) {
      factors.add(i);
    }
    i++;
  }
  return factors;
}
```

and the corresponding test case:

```java
@Test
public void testFindFactors() {
  // given / arrange
  ArrayList<Integer> expectedResult = new ArrayList<Integer>();
  expectedResult.add(1);
  expectedResult.add(2);
  expectedResult.add(4);
```

[2] JUnit: https://junit.org/
[3] Given-When-Then from Martin Fowler: https://martinfowler.com/bliki/GivenWhenThen.html

```
8     expectedResult.add(5);
9     expectedResult.add(10);
10    expectedResult.add(20);
11
12    // when / act
13    ArrayList<Integer> actualResult = SuperMath.findFactors(20);
14
15    // then / assert
16    assertEquals(expectedResult, actualResult);
17  }
```

To run the test, you can either use your IDE (there are often multiple ways to do so) or you can use the Maven CLI:

```
$ mvn clean test
```

In order to obtain code coverage, you can either again use your IDE's coverage integration or add the JaCoCo[4] Maven plugin manually to the `pom.xml`:

```
1  <build>
2    <plugins>
3      <plugin>
4        <groupId>org.jacoco</groupId>
5        <artifactId>jacoco-maven-plugin</artifactId>
6        <version>0.8.4</version>
7        <executions>
8          <execution>
9            <goals>
10             <goal>prepare-agent</goal>
11           </goals>
12         </execution>
13         <!-- attached to Maven test phase -->
14         <execution>
15           <id>report</id>
16           <phase>test</phase>
17           <goals>
18             <goal>report</goal>
19           </goals>
20         </execution>
21       </executions>
22     </plugin>
23   </plugins>
24 </build>
```

The JaCoCo data will be generated at `teststarter/target/site/jacoco`, where you can just open the `index.html` to inspect it.

When you run the test cases now (again with Maven CLI or IDE), you should get full code coverage for `findFactors(...)` (see figure 3):

```
13.    public static ArrayList<Integer> findFactors(long input) {
14.        ArrayList<Integer> factors = new ArrayList<Integer>();
15.        int i = 1;
16.        while (i <= input) {
17.            if (input % i == 0) {
18.                factors.add(i);
19.            }
20.            i++;
21.        }
22.        return factors;
23.    }
```

Figure 3: Code coverage as displayed by JaCoCo

4. Why, despite the fact that we have full code coverage and the test case passes, might `findFactors(...)` still fail for some inputs? Name at least one fault that is not covered by the test case.
5. How could you (hypothetically) write a better test case that also reveals the uncovered faults?

Next, take a look at the `fullAdder(...)` method:

[4] JaCoCo: https://www.jacoco.org/jacoco/trunk/index.html

```java
public static Boolean[] fullAdder(Boolean a, Boolean b, Boolean c) {
  Boolean s, tmp;
  tmp = a != b;
  s = c != tmp;
  c = (a && b) || (tmp && c);
  return new Boolean[]{s, c};
}
```

6. This method is simple enough to be completely tested. How many test cases would you need?
7. Implement all necessary test cases to completely test this method. You may use a single Java method with `@Test` annotation.

Finally, take a look at the method `BinaryAddition(...)`:

```java
public static Boolean[] BinaryAddition(Boolean[] a, Boolean[] b) throws
    ArrayLengthsMismatchException {
  if (a.length != b.length) {
    throw new SuperMath.ArrayLengthsMismatchException();
  }
  Boolean[] result = new Boolean[a.length + 1];
  Boolean carry = false;
  int i;
  for (i = 0; i < a.length; i++) {
    Boolean[] tmp = fullAdder(a[i], b[i], carry);
    result[i] = tmp[0];
    carry = tmp[1];
  }
  result[i] = carry;

  return result;
}
```

8. Can `BinaryAddition(...)` be tested completely?
9. Correct error or exception **raising** and **handling** is often insufficiently tested. The function `BinaryAddition(...)` throws a so-called *checked* exception which is subject to *compile-time checking* denoted by the `throws ArrayLengthsMismatchException` after the method signature[5]. This is designed to reduce the number of improperly handled exceptions since these checked exceptions must be handled explicitly. Note that we are enforcing the function's preconditions as in design by contract. However, we need to distinguish throwing a checked exception from assertions (i.e. *unchecked throwables*) for handling a broken precondition.

   - Think about a test case which asserts that the function raises the correct exception if the lengths of the two input arrays are different.
   - Add the following handler code (i.e. caller of `BinaryAddition(...)`) to `TestStarterApp`:

   ```java
   public static void someCaller(Boolean[] a, Boolean[] b) {
     Boolean[] res = null;
     try {
       // give it a try
       res = SuperMath.BinaryAddition(a, b);
     } catch (SuperMath.ArrayLengthsMismatchException e) {
       // fail gracefully
       res = new Boolean[]{false, false, false};
     }
     System.out.print(Arrays.toString(res));
   }
   ```

   Think of a test case that tests if the handler's logic is working properly. Note that testing handler code is generally context-specific as depending on your application you may want to treat these cases differently.

Exercise 4: The Test Pyramid in Spring Boot

[5] Checked and Unchecked Exceptions in Java: https://www.baeldung.com/java-checked-unchecked-exceptions

This exercise will introduce you more concretely to the three layers of the test pyramid; unit tests, integration (service) tests, and system or End-to-end (E2E) tests by example. The testing pyramid suggests that the bulk of your tests are unit tests at the bottom of the pyramid. As you move up the pyramid, your tests get more complicated, interwoven and larger, but at the same time the number of tests (the width of your pyramid) gets smaller. Google often suggests a 70/20/10 split: 70% unit tests, 20% integration tests, and 10% end-to-end tests.[6]

We will use a simple Spring Boot application `springtestpyramid` which you can find in the ASE repository at `exercise_08/springtestpyramid`. The main functionality of the application we are going to test is to handle project entities (similar to your project microservice) and provide a JSON REST API for accessing these via HTTP. It consists of an in-memory H2 database with a Spring JPA `@Repository` bean and a Spring `@RestController` bean. In particular, you are going to implement one test per layer of the pyramid as follows:

- **Unit Test:** Test if a controller method for retrieving a project by its id will return an empty project if called with a project id that does not match any of the projects in the database.
- **Integration Test:** Test if a JPA repository responsible for querying projects from the database will do so correctly when it is not queried with an id, but with the project name as an identifier.
- **System Test:** Test if an up-and-running Spring Boot application that receives an HTTP request to GET a specific project by its id will return an empty JSON response to the requester in case a project with this id is not found in the database.

Note that in practice you sometimes encounter inconsistency in how people define and talk about unit/integration/service/system tests[7].

1. **Unit Tests**

   The foundation of your test suite will be made up of **unit tests**. In an object-oriented language a unit can range from a single method to an entire class. Go ahead and open the test class `src/test/java/edu/tum/ase/springtestpyramid/unit/ProjectControllerTests.java`:

```java
@RunWith(SpringRunner.class)
public class ProjectControllerTests {

  @Autowired
  private ProjectController systemUnderTest;

  @MockBean
  private ProjectRepository projectRepository;

  @Test
  public void should_ReturnExistingProject_When_ProjectNameExisting() {
    // given
    Project existingProject = new Project();
    existingProject.setId("1");
    existingProject.setName("Test-project");
    Project project = new Project();
    project.setName("Test-project");
    given(projectRepository.findByName(anyString())).willReturn(Optional.of(existingProject))
        ;

    // when
    Project result = systemUnderTest.createProject(project);

    // then
    then(result.getId()).isEqualTo(existingProject.getId());
    then(result.getName()).isEqualTo(existingProject.getName());
  }

  @TestConfiguration
  static class ProjectControllerTestsConfiguration {

    @Bean
    public ProjectController systemUnderTest() {
      return new ProjectController();
    }
```

[6] Testing Pyramid at Google: https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html
[7] Google's Test Nomenclature: https://testing.googleblog.com/2010/12/test-sizes.html

```
    }
}
```

Stubbing/Mocking[8]

We use Mockito[9] to replace the real `ProjectRepository` class with a mock for our test. We can then define canned responses the *stubbed* method should return in this test. Mocking/stubbing makes tests simpler, more predictable and allows us to easily setup test data. To make use of the `@Autowired` annotation for the system under test, we can create a respective bean in a `@TestConfiguration` static class.

Your task is to write a second unit test, which checks if the controller properly returns an empty project object when the requested `projectId` is not found in the `getProject(@PathVariable String projectId)` method.

If you want to write unit tests for controller methods which require authentication, you can make use of mock user objects (e.g., `@WithMockUser(authorities = USER)`) and the annotation `@WebMvcTest`[10].

2. **Integration Tests**

   Next, let's implement an integration test (or sometimes also called service test), which integrates with some other parts (e.g. databases, file systems, or network calls to other applications). In our case, we want to test the integration with a H2 database which we need to start before running our tests. Since Spring Data does the heavy lifting of implementing database repositories, one might argue that we are testing the framework. But it still ensures that (i) our custom method `findByName(String name)` is working as expected, and (ii) the database is wired correctly with the repository.

   Go ahead and open the test class `src/test/java/edu/tum/ase/springtestpyramid/it/ProjectRepositoryIT.java`:

```java
@RunWith(SpringRunner.class)
@DataJpaTest
public class ProjectRepositoryIT {
  @Autowired
  private ProjectRepository systemUnderTest;

  @After
  public void tearDown() throws Exception {
    // This can be used here as `deleteAll()` is generated by Spring Data
    systemUnderTest.deleteAll();
  }

  @Test
  public void should_ReturnPersistedProject_When_QueriedByExistingName() {
    // given

    // when

    // then
  }
}
```

   Note how we use the Spring and JUnit annotations `@DataJpaTest` and `@After` for controlling the test procedure.

   Implement the test which checks if `findByName(...)` behaves as expected and returns a previously persisted (use `systemUnderTest.save(...)`) project.

3. **System/E2E Tests**

   Finally, we will implement an E2E test often also referred to as system test. These tests typically use a production-like environment and are run via the application's user interface, which causes them to run pretty slowly. People often forget that a REST API or a CLI is as much of a user interface as a fancy web user interface. We will test our project's REST API in the following mimicing an actual user, a web client application, or another microservice that uses it.

   Go ahead and open the test class `src/test/java/edu/tum/ase/springtestpyramid/e2e/ProjectE2ERestTests.java`:

[8] Discussion on Differences and Commonalities of Mocks and Stubs: https://stackoverflow.com/questions/3459287/whats-the-difference-between-a-mock-stub

[9] Mockito: https://site.mockito.org/

[10] Spring Web MVC Testing: https://spring.io/guides/gs/testing-web/

```java
@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureMockMvc
public class ProjectE2ERestTests {

  private final String URL = "/project/";

  @Autowired
  private MockMvc systemUnderTest;

  @Autowired
  private ProjectRepository projectRepository;

  @Autowired
  private ObjectMapper objectMapper;

  @After
  public void tearDown() {
    projectRepository.deleteAll();
  }

  @Test
  public void should_ReturnPersistedProject_When_PostWithProject() throws Exception {
    // given
    Project project = new Project();
    project.setName("Test-project");
    Project createdProject = projectRepository.save(project);

    // when
    ResultActions result = systemUnderTest.perform(post(URL)
        .content(objectMapper.writeValueAsString(project))
        .contentType(MediaType.APPLICATION_JSON));

    // then
    result
        .andExpect(status().isOk())
        .andExpect(MockMvcResultMatchers.jsonPath("$.id").value(createdProject.getId()))
        .andExpect(MockMvcResultMatchers.jsonPath("$.name").value(createdProject.getName()));
  }

}
```

Most important about this class is that we use the `@SpringBootTest` and the `@AutoConfigureMockMvc` annotations, which tell Spring to look for a main configuration class (one with `@SpringBootApplication` for instance), and use that to start a Spring application context, but without an actual web server (i.e. mocking the MVC).

Your task is to implement a method which tests the second controller endpoint `getProject(...)`. Therefore, you ought to implement the test `should_ReturnEmptyProject_When_RequestedWithInvalidId`.

Exercise 5: Continuous Integration with GitLab CI

One of the goals of using CI is to prevent integration problems ("integration hell") especially towards the final phase of a software development project. The idea is to reduce this risk through frequent (mostly at least daily) integration of code into a shared repository and building/testing each change. Typically, this integration is verified by an automated build, test and often even deployment to detect integration errors. Often the CI execution environment is better isolated and thus a more suitable and stable infrastructure when compared to local testing environments, as they might decay and diverge over time.

There is however some (at least anecdotal) evidence where CI has been misused in large organizations as well, see the footnotes for a funny example on the evergreen CI at Volkswagen[11].

This exercise will teach you the basics of GitLab CI and how to use it with Maven projects. In GitLab you can set up any repository to make use of CI/CD features by enabling it in the repository settings and adding a `.gitlab-ci.yml` configuration file to the root directory.

1. Read up on the basics of GitLab CI configuration syntax (https://docs.gitlab.com/ee/ci/yaml/README.

[11] The evergreen CI Pipeline: https://github.com/auchenberg/volkswagen

html) and have a look at code examples (https://docs.gitlab.com/ee/ci/examples/README.html)

2. Include the three stages, *build*, *test*, and *package* in the following empty `.gitlab-ci.yml`:

```
# This template uses jdk8 for verifying and deploying images
image: maven:3.3.9-jdk-8

# TODO: implement stages
```

3. Define one job for each stage which runs the respective maven commands `compile`, `test` and `package`
4. Finally, add an environment variable `MAVEN_OPTS` to your *package* job which defines to skip tests in this stage (`-DskipTests=true`)[12]

[12] Maven Configuration: https://maven.apache.org/configure.html