Aufgabe 1 – Unit Test

In dieser Übung werden Sie die Grundlagen von Unit-Tests in Java kennenlernen. Sie werden einen Unit-Test für den Client des Demoprojekts schreiben. Die Übung zeigt Ihnen, wie Sie Testfälle definieren, diese ausführen und die Ergebnisse interpretieren. Ziel ist es, dass Sie am Ende in der Lage sind, selbstständig Unit-Tests für Ihre Java-Methoden zu schreiben.

Klonen Sie zunächst das Demoprojekts: https://github.com/PatrickATdIT/atdit-y2025-a. Machen Sie sich mit dem Code des LoginControllers und des LoginServices vertraut.

Kompilieren Sie das Programm und lassen Sie es laufen. Bei gültiger Eingabe einer Benutzer-Passwort-Kombination erscheint eine Erfolgsmeldung, andernfalls erscheint ein entsprechender Fehler. Testen Sie die Anmeldung einmal mit den gültigen Anmeldedaten Benutzer „admin“ und Passwort „admin“ und einmal mit beliebigen, etwa Benutzer „XXX“ und Passwort „YYY“, um das Prinzip nachvollziehen zu können.

Schreiben Sie nun einen UnitTest unter Benutzung von JUnit 5, der sicherstellt, dass bei fehlender Eingabe des Benutzernamens oder des Passworts die Fehlermeldung „Login failed“ erscheint.

Hilfestellungen

1. Es ist sinnvoll, den Test nicht auf der UI-Klasse (also dem LoginController) laufen zu lassen, sondern auf dem LoginService, denn hier kann der Rückgabewert der öffentlichen Methode login() geprüft werden. Die Erwartung ist, dass bei fehlendem Benutzer oder Passwort eine LoginFailedException geworfen wird.
2. Sie müssen die Dependency für JUnit 5 über Maven importieren. Empfehlung: Benutzen Sie das Aggregat aus Engine und API „junit-jupiter”. Sie finden die Koordinaten dafür auf https://mvnrepository.com/.
3. Stellen Sie sicher, dass sie eine aktuelle Version des Maven-Surefire-Plugins verwenden, damit Ihr Testfall von Maven erkannt werden kann. Die Koordinaten finden Sie ebenfalls auf https://mvnrepository.com/.
4. Die Testklasse muss im src/test/java-Ordner liegen. Der Testfall muss die Annotation org.junit.jupiter.api.Test aufweisen.
5. Erstellen Sie eine Testmethode mit beschreibenden Namen (z.B. ifPasswordIsInitial__throwLoginFailedException)
6. Prüfen Sie, ob die Login-Methode bei fehlender Eingabe von Passwort oder Benutzer eine LoginFailedException prüft, indem Sie die Methoden der org.junit.jupiter.api.Assertions-Klasse nutzen.
7. Der LoginService sollte einen Fehler haben, es wird stets eine ServiceException geworfen, wenn Benutzer oder Passwort nicht geliefert werden. Korrigieren Sie den Fehler und stellen Sie sicher, dass der Unit-Test bestanden wird.

Lösung

1. Importieren Sie die junit-jupiter-Dependency in der pom.xml mit Scope Test:

```
<dependencies>
[...]
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-slf4j2-impl</artifactId>
    <version>2.24.3</version>
  </dependency>

  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>5.12.0</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

2. Deklarieren Sie die Nutzung einer aktuellen Surefire-Plugin-Version in der pom.xml:

```
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>3.5.2</version>
      </plugin>
    </plugins>
  </build>
</project>
```

3. Erstellen Sie die Klasse LoginServiceTest im Paket atdit.y2025.client.middleware im Ordner src/test/java. Erstellen Sie die Methode ifPasswordIsInitial__throwLoginFailedException mit Test-Annotation.

```
package atdit.y2025.client.middleware;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;

public class LoginServiceTest {
 @Test
 public void ifPasswordIsInitial__throwLoginFailedException() {
 }
}
```

4. Erstellen Sie eine neue Instanz von LoginService und rufen Sie die Methode login mit beliebigen user aber ohne passwort auf.

```
 @Test
 public void ifPasswordIsInitial__throwLoginFailedException() {
  try {
   new LoginService().login( "admin", "" );
  } catch( ServiceException e ) {

  } catch( LoginFailedException e ) {

  }
 }
```

5. Die Erwartung ist, das seine Exception vom Typ LoginFailedException geworfen wird. Daher schlägt der Unit-Test fehl, wenn keine Exception geworfen wird oder eine andere, als die erwartete. Lassen Sie den Unit-Test daher mittels Assertions.fail() fehlschalgen, wenn die Zeile nach der Login-Methode erreicht wird (das heißt, keine Exception geworfen wurde) oder eine ServiceException abgefangen wurde. Im Fall einer LoginFailedException ist nichts zu tun, da dies das gewünschte Resultat ist; die Methode darf dann einfach enden.

```
public void ifPasswordIsInitial__throwLoginFailedException() {
  try {
    new LoginService().login( "admin", "" );
    Assertions.fail("Expected LoginFailedException");
  } catch( ServiceException e ) {
    Assertions.fail("Expected LoginFailedException");
  } catch( LoginFailedException e ) {
    //expected
  }
}
```

6. Lassen Sie den Unit-Test laufen und interpretieren Sie das Ergebnis.
   Der Unit-Test schlägt fehl. Es wird eine ServiceException geworfen. Dies ist unerwartet und muss korrigiert werden.
7. Fehlerursache ist, dass nirgends auf fehlenden User oder Passwort geprüft wird. Implementieren Sie diese Prüfung direkt am Anfang der Login-Methode vor dem ServiceAufruf. Wenn der übergebene User null oder leer ist, soll eine LoginFailedException geworfen werden; analog beim Passwort.

```
public User login( String user, String password ) throws ServiceException, LoginFailedException {
  logger.info( "login attempt" );
  logger.trace( "user: {}", user );
  logger.trace( "password: {}", password );

  if( user == null || password == null || user.isEmpty( ) || password.isEmpty( ) ) {
    throw new LoginFailedException( );
  }

  var response = callLoginService( user, password );
```

8. Lassen Sie den Test erneut laufen. Der Test sollte nun bestanden werden.
9. Erstellen Sie entsprechende Testfälle, die abprüfen, das keiner der Parameter null oder leer ist. Es müssen insgesamt sechs Testfälle sein:
   a. User und Passwort leer
   b. User und Passwort null
   c. User gefüllt, Passwort leer
   d. User gefüllt, Passwort null
   e. User leer, Passwort gefüllt
   f. User null, Passwort gefüllt
10. Die Lösung finden sie im Repository https://github.com/PatrickATdIT/atdit-y2025-b.
    Anmerkung: Diese enthält nur den Testfall für das leere Passwort.

Zusatzaufgabe:

Stellen Sie sicher, dass bei korrekter Eingabe von gültigen Anmeldeinformationen ein entsprechendes User-Objekt erstellt wird.

Aufgabe 2 – Abhängigkeiten auflösen

Nach der ersten Übung wissen Sie nun, wie man Unit-Tests mit JUnit 5 schreibt. In dieser Aufgabe beschäftigen wir uns mit Abhängigkeiten. Haben Sie die Zusatzaufgabe der Aufgabe 1 angeschaut und eine Lösung, wie diese gefunden?

```
@Test
void ifCorrectCredentials__createUserObject( ) {
  var server = new Server( new LoginRequestHandler( ) );
  server.start( ); //this makes it an integration test, but not a unit test!!!

  User user = null;

  try {
    var cut = new LoginService( );
    user = cut.login( "admin", "admin" );
  } catch( ServiceException | LoginFailedException e ) {
    Assertions.fail("No exception expected");
  }

  Assertions.assertNotNull( user );
}
```

Herzlichen Glückwunsch. Sie haben einen Integrationstest geschrieben, aber keinen Unit-Test. Denn zwar prüft der Test darauf, dass bei korrekten Anmeldedaten ein User-Objekt erstellt wird, aber es wird nicht nur die Middleware (LoginService-Klasse) sondern auch den Service auf dem Server selbst. Der Server ist eine Abhängigkeit, die nicht unter unserer Kontrolle steht. Die Abhängigkeit muss aufgelöst werden.

Ihre Aufgabe lautet also: Modifizieren Sie die LoginService-Klasse so, dass Sie sie isoliert vom Server testen können. Stellen Sie sicher, dass bei korrekter Eingabe von gültigen Anmeldeinformationen ein entsprechendes User-Objekt erstellt wird.

Klonen Sie dazu das Repository https://github.com/PatrickATdIT/atdit-y2025-b.

Hilfestellungen

1. Stellen Sie fest, wo die Abhängigkeit in Ihrem Code eingeführt wird. Es muss natürlich der Moment sein, in dem Sie eine Verbindung zum Server herstellen.
2. Wählen Sie eine geeignete Ersetzungsmethode, um zu verhindern, dass eine Verbindung zum Server erstellt wird. Es empfiehlt sich das Auslagern der Verbindungserstellung [java. net. URL#openConnection( )] in eine externe Klasse mit entsprechender Schnittstelle. Vorschlag: Schnittstelle ServiceConnector mit der Methode connect, die eine URL entgegennimmt und eine HttpURLConnection zurückliefert.
3. Wählen Sie eine Methode für die Dependency Injection, die es ermöglicht in der Produktivumgebung einen anderen Connector zu verwenden, als in der Testumgebung. Die Musterlösung wird die Constructor-Injection für den MockConnector nutzen; im Produktivcode wird ein ProductiveConnector standardmäßig erzeugt, wenn kein anderer Connector injiziert wird.

4. Der Connector wird eine unechte HttpURLConnection erstellen müssen. Es bietet sich an, diese als Subklasse von HttpURLConnection zu erstellen. Prüfen Sie in der LoginService-Klasse, welche Methoden dieser HttpURLConnection benutzt und folglich implementiert werden müssen. Neben der abstrakten Methoden müssen Sie zusätzlich implementieren:
    a. getInputStream()
    b. getErrorStream()
    c. getResponseCode()
    d. getResponseMessage()
5. Wählen Sie eine Implementierung, die möglichst einfach ist und Festwerte zurückliefert. Sie wollen genau einen Fall mocken, nämlich dass korrekte Daten übergeben werden, woraufhin der Service mit einem Response Code 200 antworten würde, eine Nachricht im HTTP-Header „OK“ und mit der Benutzerolle in der Payload.
6. Injizieren Sie eine Mockinstanz in die zu testende Methode im Unit Test.

Lösung

1. Erstellen Sie eine Schnittstelle Connector im Middleware-Paket mit einer Methode, die eine URL entgegennimmt und eine HttpURLConnection zurückliefert.

```
package atdit.y2025.client.middleware;

import java.io.IOException;
import java.net.HttpURLConnection;
import java.net.URL;

public interface Connector {
  HttpURLConnection connect( URL url) throws IOException;
}
```

2. Erstellen Sie eine Klasse (ProductiveConnector) für die Produktivnutzung, die das Interface implementiert. Implementieren Sie die Klasse entsprechend wie den zu ersetzenden Code, das heißt, weißen Sie die eingehende URL an eine Verbindung zu öffnen.

```
package atdit.y2025.client.middleware;

import java.io.IOException;
import java.net.HttpURLConnection;
import java.net.URL;

class ProductiveConnector implements Connector {
 @Override
 public HttpURLConnection connect( URL url ) throws IOException {
   return (HttpURLConnection) url.openConnection( );
  }
}
```

3. Erstellen Sie ein Attribut für den Connector (Schnittstelle!) in der LoginService-Klasse. Erstellen Sie außerdem einen Defaultconstructor, der das Attribut mit einer neuen Instanz des ProductiveConnectors belegt.

```
public class LoginService {

 private static final Logger logger = LoggerFactory.getLogger( LoginService.class );
 public static final String LOGIN_SERVICE_PATH = "http://localhost:4711/login";
 public static final String REQUEST_METHOD_GET = "GET";
 public static final int CONNECTION_TIMEOUT_VALUE = 10_000;

 private final Connector connector;

 public LoginService( ) {
  connector = new ProductiveConnector();
 }
```

4. Ersetzen Sie den ursprünglichen Code, der die Verbindung öffnet, durch den Aufruf des Connectors.

```
 private HttpURLConnection makeServiceConnection( String user, String password ) throws ServiceException {
  String queryString = "?user=" + password +
                "&password=" + user;

  try {
   var serviceURL = URI.create( LOGIN_SERVICE_PATH + queryString ).toURL( );
   logger.trace( "serviceURL: {}", serviceURL );
   var connection = (HttpURLConnection) serviceURL.openConnection( );
   var connection = connector.connect( serviceURL );
   connection.setRequestMethod( REQUEST_METHOD_GET );
   connection.setConnectTimeout( CONNECTION_TIMEOUT_VALUE );
```

5. Erstellen Sie einen neuen Unit-Test in der Klasse LoginServiceTest, der zunächst als Integrationstest agiert, um zu beweisen, dass die gemachten Änderungen nicht zerstört haben. Lassen Sie den Test laufen und bestätigen Sie, dass er bestanden wird.

```
 @Test
 void ifCorrectCredentials__createUserObject( ) {
  var server = new Server( new LoginRequestHandler( ) );
  server.start( ); //this makes it an integration test, but not a unit test!!!

  User user = null;

  try {
   var cut = new LoginService( );
   user = cut.login( "admin", "admin" );
  } catch( ServiceException | LoginFailedException e ) {
   Assertions.fail("No exception expected");
  }

  Assertions.assertNotNull( user );
 }
```

6. Nun geht es an das Erstellen der Mocks. Erstellen Sie einen SuccessfulLoginConnector im selben Paket, wo auch die Testklasse liegt. Dieser soll die neue Schnittstelle Connector implementieren. Ignorieren Sie zunächst die Methodenimplementierung.

```
package atdit.y2025.client.middleware;

import java.io.IOException;
import java.net.HttpURLConnection;
import java.net.URL;

public class SuccessfulLoginConnector implements Connector{
 @Override
 public HttpURLConnection connect( URL url ) throws IOException {
  return null;
 }
}
```

7. Es wird noch eine Connection benötigt, die von der Methode connect( ) der gerade erstellen Klasse erstellt wird. Erstellen Sie die Klasse SuccessfulLoginConnection, die von HttpURLConnection erbt. Implementieren Sie die abstrakten Methoden (diese müssen nichts machen, da sie für den Mock irrelevant sind) und den Constructor entsprechend des Superconstructors.

```
package atdit.y2025.client.middleware;

import java.io.IOException;
import java.net.HttpURLConnection;
import java.net.URL;

public class SuccessfulLoginConnection extends HttpURLConnection{
 protected SuccessfulLoginConnection( URL u ) {
  super( u );
 }

 @Override
 public void disconnect( ) {
  //nothing to do
 }

 @Override
 public boolean usingProxy( ) {
  return false; //irrelevant
 }

 @Override
 public void connect( ) throws IOException {
  //nothing to do
 }
}
```

8. Implementieren Sie die Methoden, die von dem Unit-Test gerufen werden. Das sind folgende:
    a. getInputStream(): Bei erfolgreichem Login wird diese Methode gerufen. Der Stream enthält die Rolle des Users. Wir geben einen ByteArrayInputStream auf dem Text „MockRole“ zurück.
    b. getErrorStream(): Die Methode sollte bei erfolgreichem Login nicht gerufen werden, daher nutzen wir die Assertions.fail()-Methode, um den Unit-Test fehlschlagen zu lassen, falls die Methode dennoch gerufen wird. Die Implementierung ist nicht nötig, macht es aber später leichter, den Fehler zu finden, falls durch einen Bug in der LoginService-Klasse doch einmal der falsche Stream abgeholt wird.
    c. getResponseCode(): Ein erfolgreicher Login sollte in einer Antwort 200 resultieren.
    d. getResponseMessage(): Eine Antwort 200 wird üblicherweise von einem „OK“ begleitet.

```
@Override
public void connect( ) throws IOException {
  //nothing to do
}

@Override
public InputStream getInputStream( ) throws IOException {
  return new ByteArrayInputStream( "MockRole".getBytes( ) );
}

@Override
public InputStream getErrorStream( ) {
  Assertions.fail( "getErrorStream was called but should not have be called" );
  return null;
}

@Override
public int getResponseCode( ) throws IOException {
  return 200;
}

@Override
public String getResponseMessage( ) throws IOException {
  return "OK";
 }
}
```

9. Lassen Sie den SuccessfulLoginConnector beim Aufruf seiner connect-Methode eine neue Instanz der SuccessfulLoginConnection zurückgeben.

```
public class SuccessfulLoginConnector implements Connector{
 @Override
 public HttpURLConnection connect( URL url ) throws IOException {
  return null;
   return new SuccessfulLoginConnection( url );
 }
}
```

10. Erstellen Sie einen Constructor für den LoginService, mit dem Sie eine andere Instanz als Connector injizieren können.

```
private final Connector connector;

public LoginService( ) {
  connector = new ProductiveConnector();
}

public LoginService( Connector connector ) {
  this.connector = connector;
}
```

11. Passen Sie den bereits geschrieben Test so an, dass er dem LoginService einen SuccessfulLoginConnector unterjubelt. Stellen Sie außerdem sicher, dass der Server nicht mehr aktiviert wird.

```
@Test
void ifCorrectCredentials__createUserObject( ) {
  var server = new Server( new LoginRequestHandler( ) );
  server.start( ); //this makes it an integration test, but not a unit test!!!

  User user = null;

  try {
    var cut = new LoginService( );
    var cut = new LoginService( new SuccessfulLoginConnector() );
    user = cut.login( "admin", "admin" );
  } catch( ServiceException | LoginFailedException e ) {
    Assertions.fail("No exception expected");
  }

  Assertions.assertNotNull( user );
}
```

12. Lassen Sie den Test laufen und interpretieren Sie das Ergebnis.
    Der Test wird bestanden.
13. Die Musterlösung finden Sie in Repository https://github.com/PatrickATdIT/atdit-y2025-c

Zusatzaufgabe

Schreiben Sie einen Unit Test, der sicher stellt, dass der richtige Service gerufen wird und die HTTP-GET-Parameter User und Password korrekt an den Service übergeben werden.