

Szenario:

Sie sollen Unit-Tests für eine Spring Boot Anwendung schreiben, die Informationen über Studenten verwaltet.

Aufgabe 1 – Erstellen der Projektbasis

- Nutzen Sie als Projektbasis die zip-Datei aus dem ELO.
- Beachten Sie die zusätzlichen Dependencies für Unit-Tests und Mocks:

```
Spring Web
Spring Security
Spring Data JPA
H2

<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-engine</artifactId>
  <scope>test</scope>
</dependency>

<dependency>
  <groupId>org.mockito</groupId>
  <artifactId>mockito-core</artifactId>
  <scope>test</scope>
</dependency>
```

- Inspizieren Sie den Code rund um die Klasse Student:
 - **Student Entity:** Repräsentiert einen Studenten mit Feldern wie id, name, email.
 - **StudentRepository:** Ein Interface, das Methoden für die Interaktion mit den Studentendaten definiert (z.B. save, findById, findAll).
 - **StudentService:** Eine Service-Schicht, die die Geschäftslogik im Zusammenhang mit Studenten behandelt (z.B. Erstellen, Abrufen, Aktualisieren von Studenten).
 - **StudentController:** Ein REST-Controller, der Endpunkte für die Verwaltung von Studenten bereitstellt (z.B. GET /students, POST /students, PUT /students/{id}).

Aufgabe 2 - Erstellen Sie die Tests für Student im Paket „src/test/java“:

```
▼ src/test/java
  > de.othr.junit
  > de.othr.junit.controller
  > de.othr.junit.model
  > de.othr.junit.repository
  > de.othr.junit.service
```

Hinweise:

- Verwenden Sie die `@WebMvcTest` und `@AutoConfigureMockMvc` Annotationen für Controller-Tests.
- Verwenden Sie `@MockBean`, um das `StudentRepository` in Ihren Service-Tests zu mocken.
- Verwenden Sie `MockMvc`, um HTTP-Anfragen auszuführen und Antworten zu überprüfen.
- Verwenden Sie Assertions wie `assertEquals`, `assertNotNull` und `assertThrows`, um erwartete Ergebnisse zu überprüfen.

🕒 StudentRepository Tests:

- Testen `create`, `findById`, `findAll`, `findByNameContaining`

```
@DataJpaTest
public class StudentRepositoryTest {

    @Autowired
    private StudentRepository studentRepository;

    @Test
    public void test_givenStudent_whenRepositorySave_thenStudentObject() {
        //given
        Student student = new Student();
        student.setName("diana");
        //when
        Student studentSaved = studentRepository.save(student);
        //Then
        Assertions.assertNotNull(studentSaved);
        Assertions.assertEquals(student.getName(), "diana");
    }

    @Test
    public void test_Morestudents_findAllStudents_greaterThanOne () {
        //Given
        Student student1 = new Student();
        Student student2 = new Student();
        studentRepository.save(student1);
        studentRepository.save(student2);
        //when
        List <Student> listStudents = (List<Student>) studentRepository.findAll();
        //then
        assertTrue(listStudents.size()>1);
    }

    @Test
    public void test_studentID_findById_SameId () {
        //Given
        Student student2 ;
        Long id =( long) 2;
        //when
        student2 = (studentRepository.findById((long) id)).get();
        //then
        assertTrue(student2.getName().equals("Anja Fischer"));
    }

    @Test
    public void test_studentname_findByNameContaining_StudentNotNull () {
        //Given
        String name = "Fischer";
        //when
        List <Student> listStudents = studentRepository.findByNameContainingIgnoreCase(name);
        //then
        assertTrue(listStudents.size()>0);
    }
}
```

🕒 StudentService Tests:

- Testen der createStudent-Methode: Überprüfen Sie, ob ein neuer Student erfolgreich erstellt und im Repository gespeichert wird.
- Testen der getStudentById-Methode: Überprüfen Sie, ob der richtige Student abgerufen wird, wenn eine gültige ID angegeben wird.
- Testen der getStudentById-Methode für ungültige IDs: Überprüfen Sie, ob eine Ausnahme geworfen wird, wenn eine ungültige ID angegeben wird.

```
@ExtendWith(MockitoExtension.class)
public class StudentServiceTest {

    @Mock
    private StudentRepository studentRepository;
    @InjectMocks
    private StudentServiceImpl studentService;

    @Test
    public void StudentService_addStudent_EqualName() {

        //Given
        Student student = new Student();
        student.setName("Daniel");

        //When
        //(faking the repository action...)
        when(studentRepository.save(student)).thenReturn(student);
        Student studentSaved = studentService.saveStudent(student);

        //Test
        assertEquals(studentSaved.getName(), "Daniel");
    }

    @Test
    public void StudentService_findById() {

        //Given
        Student student = new Student();
        student.setName("Daniel");

        //When
        //(faking the repository action...)
        when(studentRepository.findById(1L)).thenReturn(Optional.of(student));
        Student studentSaved = studentService.getStudentById((long) 1);
        //Test
        assertNotNull(studentSaved);
    }

    @Test
    void testGetStudentById_NotFound() {
        Long studentId = 1L;
        when(studentRepository.findById(studentId)).thenReturn(Optional.empty());

        assertThrows(NotFoundException.class, () -> studentService.getStudentById(studentId));
    }
}
```

StudentController Tests:

- Testen Sie die createStudent-Methode: Überprüfen Sie, ob der Controller erfolgreich einen neuen Studenten erstellt und den erstellten Studenten zurückgibt.
-

```
@WebMvcTest(value = StudentRestController.class)
@AutoConfigureMockMvc
public class StudentRestControllerTest {

    @Autowired
    MockMvc mockMvc;

    @MockitoBean
    StudentService studentService;

    @MockitoBean
    CourseService courseService;

    @Autowired
    private ObjectMapper objectMapper = new ObjectMapper();

    Student student = null;

    @BeforeEach
    public void init(){

    }

    @Test
    public void StudentControllerTest_CreateStudent_ReturnStudentJSON() throws JsonProcessingException, Exception {

        // Given : Setup object or precondition
        Student student = new Student();
        student.setName("Daniel Klein");
        student.setId((long) 1);
        //When
        when(studentService.saveStudent(any()))
.thenReturn(student);

        ResultActions responseController =
mockMvc.perform(post("/api/students/")
.contentType(MediaType.APPLICATION_JSON)
.content(objectMapper.writeValueAsString(student)))
.andDo(print());

        //Then Test
        responseController

.andExpect(MockMvcResultMatchers.jsonPath("name", CoreMatchers.is("Daniel Klein")))
.andExpect(status().isCreated());

    }
}
```

Aufgabe 3 –Testen Sie die Komponenten

Die Anweisungen unten sind auf STS bezogen. In anderen IDEs wie VS Code (z.B. mit Extension „Test Runner for Java“) funktioniert es analog.

Tests ausführen:

- **Einzelne Testmethode ausführen:**
 - Klicken Sie mit der rechten Maustaste auf die gewünschte Testmethode.
 - Wählen Sie "Run As" -> "JUnit Test" aus.
- **Alle Testmethoden in einer Klasse ausführen:**
 - Klicken Sie mit der rechten Maustaste auf die Testklasse.
 - Wählen Sie "Run As" -> "JUnit Test" aus.
- **Alle Tests in einem Paket ausführen:**
 - Klicken Sie mit der rechten Maustaste auf das Testpaket.
 - Wählen Sie "Run As" -> "JUnit Test" aus.
- **Alle Tests im Projekt ausführen:**
 - Klicken Sie mit der rechten Maustaste auf das Projekt.
 - Wählen Sie "Run As" -> "JUnit Test" aus.

Testergebnisse überprüfen:

- Das JUnit-Fenster zeigt die Testergebnisse an.
- Grüne Balken zeigen erfolgreiche Tests an.
- Rote Balken zeigen fehlgeschlagene Tests an.
- Sie können auf einzelne Testmethoden klicken, um die Details des Testlaufs anzuzeigen, einschließlich der Ausgaben und Assertions.