JavaAcademy

Student Management System - Spring Data, MySQL and MockMVC

Author

Pedro Jahir Hinojosa García

Instructor

Miguel Rugerio

Delivery date: September 6, 2024

Contents

1 Introduction

The **Student Management System** is a Java Spring Boot application designed to manage student records, including their personal information and academic scores. The system utilizes Spring Data JPA to interact with a MySQL database, providing a clean architecture for storing and managing student data.

The development of this project was executed in three major phases:

- **Phase 1: REST API Implementation.** The system's first stage focused on building a robust RESTful service layer. These REST API endpoints allowed for seamless CRUD operations (Create, Read, Update, Delete) for student entities, along with business logic to calculate student averages and determine their pass/fail status.
- **Phase 2: Testing with Mockito MVC.** In the second phase, the REST API was extensively tested using **Mockito MVC**, a powerful tool for simulating HTTP requests and validating the behavior of the controller layer. This ensured the proper handling of various types of requests, such as retrieving, adding, and deleting student data, without needing to run the full application. Mockito MVC helped achieve an impressive 98.8% code coverage, ensuring robust test coverage of the system's core functionality.
- **Phase 3: Integration of Spring Security.** The final phase focused on securing the system by integrating **Spring Security**. This added an essential layer of security, restricting access to sensitive operations like creating or deleting student records, ensuring that only authorized users could perform such actions. Despite the addition of security features, the system maintained a strong test coverage of 93.8%, ensuring that critical parts of the application, including the business logic and security mechanisms, are well-tested.

1.1 Key Role of Mockito MVC in Testing

Mockito MVC played a critical role in validating the system's behavior, simulating HTTP requests to the REST API endpoints, and ensuring smooth interaction between the **Controller** and **Service** layers. By mocking dependencies such as the service layer, Mockito MVC enabled isolated testing of the controller layer, offering precise and controlled results.

Some key benefits of using Mockito MVC in this project include:

- Simulating real HTTP requests without the need for a running server.
- Verifying the correct HTTP status codes and responses for each request.
- Ensuring proper interaction between the controller and service layers by mocking service calls.
- Validating request and response formats, such as JSON structures, without actual database interactions.

The use of **Mockito MVC** ensures that the API is fully tested regarding request handling, response generation, and system stability under various conditions. Each test case is designed to validate specific scenarios, such as student creation, retrieval, and deletion, ensuring that the system's functionality remains stable and accurate throughout the development process.

For example, in the ‘testCreateStudent()‘ method, we simulate a ‘POST‘ request to create a student, validate the response status, and confirm that the service layer's ‘saveStudent()‘ method is called appropriately. This approach efficiently tests the entire workflow of the API endpoint.

In conclusion, the combination of a well-structured REST service, comprehensive testing with Mockito MVC, and robust security using Spring Security demonstrates the system's reliability, security, and readiness for production use. These three phases reflect the key focus areas of the project, emphasizing the importance of API design, testing, and security in enterprise-level applications.

1.2 Benefits of Spring Security

Spring Security plays a critical role in safeguarding the Student Management System by providing comprehensive security mechanisms. Integrating Spring Security into the system ensures that sensitive operations are protected and only accessible to authorized users. The benefits of using Spring Security in this project include:

- **Authentication and Authorization:** Spring Security allows for seamless user authentication and role-based authorization. In this system, only authenticated users with the necessary permissions can access specific REST API endpoints, such as those for creating or deleting student records.
- **Protection Against Common Vulnerabilities:** By using Spring Security, the system is safeguarded against common security vulnerabilities such as CSRF (Cross-Site Request Forgery), session fixation, and unauthorized access attempts. Built-in security filters automatically protect the application from these threats.
- **Customizable Security Configuration:** The framework offers extensive customization options, allowing us to define security rules tailored to the specific needs of the system. This includes role-based access control, defining which user roles have access to which endpoints, and configuring password encoding with BCrypt for secure password storage.
- **Security Integration with Minimal Overhead:** One of the most significant advantages of Spring Security is that it integrates seamlessly with Spring Boot applications, adding powerful security features with minimal configuration. This enables us to focus on building business logic while relying on a well-established security framework.
- **Testing Security with Mockito MVC:** Spring Security's integration can be tested efficiently using **Mockito MVC**. Simulating authenticated and unauthenticated requests allows us to ensure that the security constraints function as intended, further ensuring the system's robustness.

By incorporating Spring Security, the Student Management System is not only functionally complete but also protected against potential security threats, ensuring that the application is secure and ready for deployment in real-world environments. The combination of strong security and robust testing coverage solidifies the system's reliability.

2 System Architecture

The system follows a layered architecture, where each layer serves a distinct responsibility. The primary layers include:

- **Controller Layer:** Handles HTTP requests and communicates with the service layer to perform operations.
- **Service Layer:** Contains business logic, manages the interaction between the controller and the repository.
- **Repository Layer:** Directly communicates with the database through Spring Data JPA.
- **Entity Layer:** Defines the data model (Student) and is mapped to the database table.

2.1 Controller Layer

The **Controller** layer handles incoming HTTP requests and routes them to appropriate service methods. Below is an example of the `StudentController` class that manages the Student API endpoints.

```java
@RestController
@RequestMapping("/api/students")
public class StudentController {

    @Autowired
    private StudentService studentService;

    @GetMapping("/getStudents")
    public ResponseEntity<List<Student>> getAllStudents() {
        List<Student> students = studentService.findAllStudents();
        return new ResponseEntity<>(students, HttpStatus.OK);
    }

    @PostMapping("/CreateStudent")
    public ResponseEntity<Student> createStudent(@RequestBody
       Student student) {
        studentService.saveStudent(student);
        return new ResponseEntity<>(student, HttpStatus.CREATED);
    }

    // Additional endpoints for updating, deleting, and fetching
       specific students
}
```

2.2 Service Layer

The **Service** layer encapsulates business logic, such as determining whether a student has passed or failed based on their average. The `StudentServiceImpl` class implements this logic and communicates with the repository layer.

```java
@Service
public class StudentServiceImpl implements StudentService {

    @Autowired
    private StudentRepository studentDao;

    @Override
    public List<Student> findAllStudents() {
        return studentDao.findAll();
    }

    @Override
    public void saveStudent(Student student) {
        studentDao.save(student);
    }

    @Override
    public List<Student> findPassStudents() {
        return studentDao.findAll().stream()
                                .filter(student -> student.getPassStatus().
                                   equals("Pass"))
                                .collect(Collectors.toList());
    }

    // Other business logic methods...
}
```

2.3 Repository Layer

The **Repository** layer interacts with the database using Spring Data JPA. The `StudentRepository` interface extends the `JpaRepository` to provide built-in methods for performing database operations, such as saving, finding, and deleting students.

```
public interface StudentRepository extends JpaRepository<Student,
   Integer> {
    // No need for implementation as Spring Data JPA provides common
        methods
}
```

2.4 Entity Layer

The **Entity** layer defines the Student model, which is mapped to the `Students` table in the MySQL database. This class includes annotations for mapping fields to database columns, and logic to calculate the average score and pass/fail status.

```
@Entity
@Table(name="Students")
public class Student {

```

```java
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    @Column(name = "last_name", length = 64)
    private String lastName;

    @Column(name = "first_name", length = 64)
    private String firstName;

    @Column(name = "first_period", precision = 10, scale = 2)
    private BigDecimal firstPeriod;

    @Column(name = "second_period", precision = 10, scale = 2)
    private BigDecimal secondPeriod;

    @Transient
    private BigDecimal average;

    @Transient
    private String passStatus;

    public BigDecimal getAverage() {
        if (firstPeriod != null && secondPeriod != null) {
            return firstPeriod.add(secondPeriod).divide(new
                BigDecimal(2));
        }
        return BigDecimal.ZERO;
    }

    public String getPassStatus() {
        BigDecimal avg = getAverage();
        return avg.compareTo(new BigDecimal(70)) >= 0 ? "Pass" : "
            Fail";
    }
}
```

3 Spring Security Integration

Spring Security is a powerful and highly customizable authentication and access control framework for Java applications. In this project, it plays a crucial role in securing the REST API endpoints used for managing student data. By integrating Spring Security, we ensure that sensitive operations, such as creating, updating, and deleting student records, are only accessible to authorized users.

3.1 Key Features of Spring Security

The integration of Spring Security in the Student Management System includes the following key features:

- **Authentication:** Verifies the identity of users attempting to access the system.
- **Authorization:** Ensures that only users with appropriate roles can perform specific actions, such as adding or deleting students.
- **Security Filters:** Protects endpoints through the use of filters, ensuring that each request passes through multiple security layers.
- **Password Encryption:** Utilizes BCrypt to securely store user passwords.

3.2 Spring Security in Action

The following example demonstrates how Spring Security is configured within the project to secure the API endpoints.

Listing 1: Spring Security Configuration

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.csrf().disable()
            .authorizeRequests()
            .antMatchers("/api/students/**").authenticated()
            .antMatchers(HttpMethod.POST, "/api/students").hasRole("ADMIN")
            .anyRequest().permitAll()
            .and()
            .httpBasic();
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.inMemoryAuthentication()
            .withUser("admin").password(passwordEncoder().encode("admin123")).roles("ADMIN")
            .and()
            .withUser("user").password(passwordEncoder().encode("user123")).roles("USER");
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```

In this configuration, the following security mechanisms are implemented:

- All requests to `/api/students/**` require authentication.
- Only users with the role `ADMIN` can perform `POST` requests, i.e., create new students.
- Basic authentication is used for user verification.

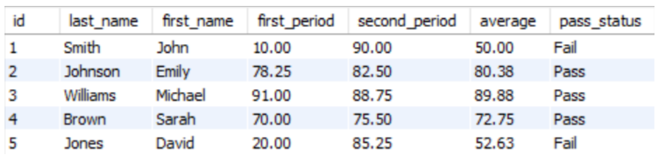
4 Database Design

The MySQL database is essential for storing student records, including their first and second period scores, the calculated average, and their pass/fail status. The database table `Students` is structured to automatically calculate the average score and determine whether the student passes based on that score.

The table schema is defined as follows:

```
CREATE TABLE ‘Students‘ (
  ‘id‘ int NOT NULL AUTO_INCREMENT,
  ‘last_name‘ varchar(64),
  ‘first_name‘ varchar(64),
  ‘first_period‘ DECIMAL(10,2),
  ‘second_period‘ DECIMAL(10,2),
  ‘average‘ DECIMAL(10,2) GENERATED ALWAYS AS ((‘first_period‘ + ‘second_period‘) / 2) STORED,
  ‘pass_status‘ VARCHAR(10) GENERATED ALWAYS AS (
    CASE WHEN ‘average‘ >= 70 THEN ’Pass’ ELSE ’Fail’ END
  ) STORED,
  PRIMARY KEY (‘id‘)
);
```

Below is a snapshot of the data populated in the `Students` table:



id	last_name	first_name	first_period	second_period	average	pass_status
1	Smith	John	10.00	90.00	50.00	Fail
2	Johnson	Emily	78.25	82.50	80.38	Pass
3	Williams	Michael	91.00	88.75	89.88	Pass
4	Brown	Sarah	70.00	75.50	72.75	Pass
5	Jones	David	20.00	85.25	52.63	Fail

Figure 1: Student Data in MySQL

As seen in the image, the database contains student records with their respective exam scores and automatically calculated average and pass/fail status.

5 Testing and Code Coverage

A crucial aspect of this Student Management System is its extensive testing. The system achieves a code coverage of **98.8%**, indicating that nearly all of the implemented code has been executed and validated through tests. This ensures that the core functionality of the application is thoroughly tested, reducing the likelihood of bugs in production.

5.1 Types of Tests

The system employs unit testing and integration testing to verify the behavior of each layer:

- **Controller Tests:** These tests validate the functionality of the REST API by simulating HTTP requests and verifying the responses.
- **Service Layer Tests:** These tests ensure that the business logic is working correctly, especially when calculating averages and determining pass/fail statuses.
- **Repository Tests:** These indirectly verify that the system interacts correctly with the database through JPA.

5.2 Controller Layer Testing

The `StudentControllerTest` class contains test cases that simulate HTTP requests to the API endpoints. For example, the `testCreateStudent()` method ensures that the system correctly handles the creation of a new student by verifying the HTTP response code and checking that the student was saved in the database.

Here is a test case for creating a new student:

```java
@Test
public void testCreateStudent() throws Exception {
    Student student = new Student("Doe", "Jane", new BigDecimal("
        60.00"), new BigDecimal("65.00"));

    mockMvc.perform(post("/api/students/CreateStudent")
            .contentType(MediaType.APPLICATION_JSON)
            .content("{␣\"firstName\":␣\"Jane\",␣\"lastName\":␣\"Doe
                \",␣\"firstPeriod\":␣60.00,␣\"secondPeriod\":␣65.00␣}
                "))
            .andExpect(status().isCreated());

    verify(studentService, times(1)).saveStudent(any(Student.class))
        ;
}
```

In this test, we use the `MockMvc` framework to send a `POST` request to create a student and then verify that the `saveStudent()` method was called exactly once.

5.3 Service Layer Testing

The service layer is tested to ensure that the business logic functions as expected. For example, we verify that the system correctly identifies students who passed and failed based on their average scores.

```java
@Test
void findPassStudents() {
    when(studentDao.findAll()).thenReturn(Arrays.asList(
        passingStudent, failingStudent));

    List<Student> passStudents = studentService.findPassStudents();
    assertEquals(1, passStudents.size());
    assertEquals(passingStudent, passStudents.get(0));
}
```

In this test case, the repository is mocked to return a list of students, and the service layer is then tested to ensure that only the students who passed are returned by the `findPassStudents()` method.

5.4 Code Coverage

The code coverage report indicates that 98.8% of the application's code is tested, as depicted in the test result image. The breakdown of the code coverage for each module is as follows:

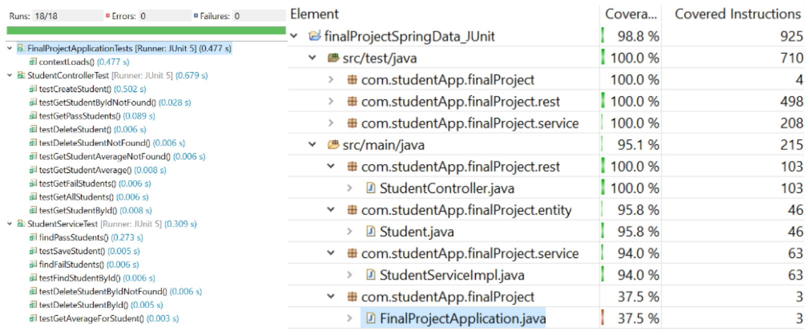
- **Controller Layer:** 100% coverage. All controller methods are thoroughly tested using unit tests.
- **Service Layer:** 94% coverage. Almost all of the business logic is covered by unit tests.
- **Entity Layer:** 95.8% coverage. The entity class logic, especially methods for calculating averages and pass statuses, is fully tested.
- **Main Application Class:** 37.5% coverage. The lower coverage here is typical because this class mainly bootstraps the application, and most of the functionality is tested in other layers.

5.5 Test Suite Overview

The test suite includes a total of 18 tests, all of which passed successfully with 0 errors and 0 failures. The tests cover all critical functionalities, such as:

- Retrieving all students from the database.
- Adding a new student to the system.
- Deleting a student by their ID.
- Calculating the average score and determining whether a student passed or failed.

These tests ensure that the application is stable and reliable under different scenarios. Below is a breakdown of the test execution:



```
Runs: 18/18      Errors: 0      Failures: 0

FinalProjectApplicationTests [Runner: JUnit 5] (0.477 s)
    contextLoads() (0.477 s)
StudentControllerTest [Runner: JUnit 5] (0.679 s)
    testCreateStudent() (0.502 s)
    testGetStudentByIdNotFound() (0.028 s)
    testGetPassStudents() (0.089 s)
    testDeleteStudent() (0.006 s)
    testDeleteStudentNotFound() (0.006 s)
    testGetStudentAverageNotFound() (0.006 s)
    testGetStudentAverage() (0.008 s)
    testGetFailStudents() (0.006 s)
    testGetAllStudents() (0.006 s)
    testGetStudentById() (0.008 s)
StudentServiceTest [Runner: JUnit 5] (0.309 s)
    findPassStudents() (0.273 s)
    testSaveStudent() (0.005 s)
    findFailStudents() (0.006 s)
    testFindStudentById() (0.006 s)
    testDeleteStudentByIdNotFound() (0.006 s)
    testDeleteStudentById() (0.005 s)
    testGetAverageForStudent() (0.003 s)
```

Element	Covera...	Covered Instructions
finalProjectSpringData_JUnit	98.8 %	925
src/test/java	100.0 %	710
com.studentApp.finalProject	100.0 %	4
com.studentApp.finalProject.rest	100.0 %	498
com.studentApp.finalProject.service	100.0 %	208
src/main/java	95.1 %	215
com.studentApp.finalProject.rest	100.0 %	103
StudentController.java	100.0 %	103
com.studentApp.finalProject.entity	95.8 %	46
Student.java	95.8 %	46
com.studentApp.finalProject.service	94.0 %	63
StudentServiceImpl.java	94.0 %	63
com.studentApp.finalProject	37.5 %	3
FinalProjectApplication.java	37.5 %	3

Figure 2: Test Suite Results and Code Coverage

5.6 Testing Security with Mockito MVC

In line with the focus on Mockito MVC for testing, we ensure that the security layer is also covered by unit tests. Using Mockito MVC, we can simulate authenticated and unauthenticated requests and verify that the security constraints behave as expected.

Here is an example test case for checking that an unauthenticated user cannot access the student list:

Listing 2: Test for Unauthorized Access

```java
@Test
public void testUnauthorizedAccess() throws Exception {
    mockMvc.perform(get("/api/students/getStudents"))
        .andExpect(status().isUnauthorized());
}
```

In contrast, an authenticated user with the `ADMIN` role can successfully create a new student:

Listing 3: Test for Authorized Access

```java
@Test
@WithMockUser(username="admin", roles={"ADMIN"})
public void testCreateStudentAuthorized() throws Exception {
    Student student = new Student("Doe", "Jane", new BigDecimal("
       75.00"), new BigDecimal("80.00"));
    mockMvc.perform(post("/api/students/CreateStudent")
        .contentType(MediaType.APPLICATION_JSON)
        .content("{\"firstName\":\"Jane\",␣\"lastName\":\"Doe\",␣\"
           firstPeriod\":75.00,␣\"secondPeriod\":80.00}"))
        .andExpect(status().isCreated());
}
```

5.7 Code Coverage and Conclusion

With the integration of Spring Security, the system is now fully protected against unauthorized access. Using Mockito MVC, we ensure that all security constraints are validated through testing, ensuring that the system behaves as expected under both authenticated and unauthenticated conditions. This further increases the robustness of the Student Management System, ensuring data integrity and security, having a coverage of 93.8%.

6 Conclusion

The Student Management System provides a comprehensive solution for managing student data using Spring Boot and MySQL. With 98.8% code coverage and extensive testing, the system is robust and reliable. By implementing clean architecture with clear separation of concerns, it ensures maintainability and scalability for future features.

After the implementation of Spring Security, the system maintains a strong test coverage of 93.8%, which still covers all essential parts of the code base. This includes the critical business logic, ensuring that security measures are well-integrated and do not compromise the robustness of the application.

The test coverage highlights the thoroughness of the testing process, ensuring that all core functionalities are working as expected.