

# TQS: Quality Assurance manual

**Rafael Kauati [105925], Rafael Vilaça [107476], Miguel Cruzeiro [107660], Diogo Silva [107647]**  
v2024-06-02

1.1 Team and roles	1
1.2 Agile backlog management and work assignment	1
2.1 Guidelines for contributors (coding style)	3
2.2 Code quality metrics and dashboards	4
3.1 Development workflow	6
3.2 CI/CD pipeline and tools	7
3.3 System observability	7
4.1 Overall strategy for testing	7
4.2 Functional testing/acceptance	8
4.3 Unit tests	9
4.3.1 Controller Tests	9
4.3.2 Service Tests	10
4.3.3 Repository tests	11
4.4 System and integration testing	13

## 1 Project management

### 1.1 Team and roles

Name	Email	NMEC	Role
Diogo Silva	diogo.manuel@ua.pt	107647	QA Engineer
Miguel Cruzeiro	miguelcruzeiro@ua.pt	107660	Product Owner
Rafael Vilaça	rafael.vilaca@ua.pt	107476	Team Leader
Rafael Kauati	rafaelkauati@ua.pt	105925	DevOps Master

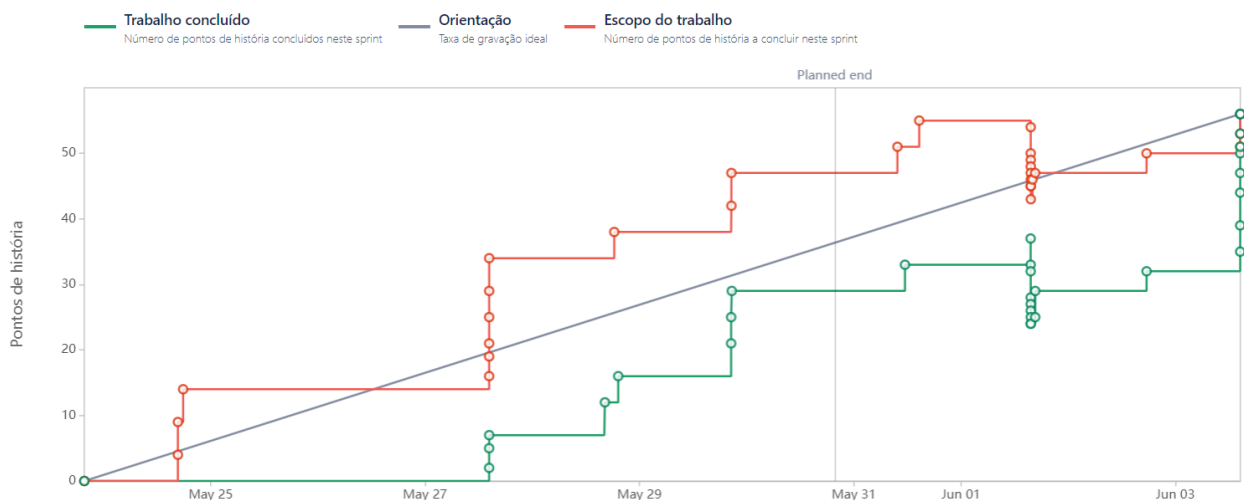
## 1.2 Agile backlog management and work assignment

In this project, we utilized the Scrum team management agile strategy. Each sprint lasted one week, during which we created numerous issues based on user stories for our project backlog. Each issue corresponded to a different feature in the final application and was assigned to one of the four team members.

Each issue was assigned different metrics, specifically story points ranging from 1 to 5 based on their importance and complexity. A story point value of 1 indicated a relatively quick implementation, while a value of 5 represented a user story that required a longer duration to complete.

To manage our backlog, we used Jira to efficiently track and prioritize tasks, ensuring timely completion within each sprint. For each new backlog item, team members created a new branch in the GitHub remote repository to develop, test, and integrate new software components without interfering with others' work. They then performed pull requests, including reviewers and tags to classify changes, merging into the "dev" branch to keep the codebase up to date.

Data - 23 de maio de 2024 - 30 de maio de 2024



## 2 Code quality management

### 2.1 Guidelines for contributors (coding style)

For someone to contribute to this project they must adhere to the coding style and conventions used in the project: This include:

- Following the style guide for the programming language used in this project.
- Writing clear, concise, and well-documented code.
- Using meaningful variable names

We utilize SonarCloud, a cloud-based code analysis platform, to identify potential bugs, vulnerabilities, code smells, and other issues that could impact the project's performance or security.

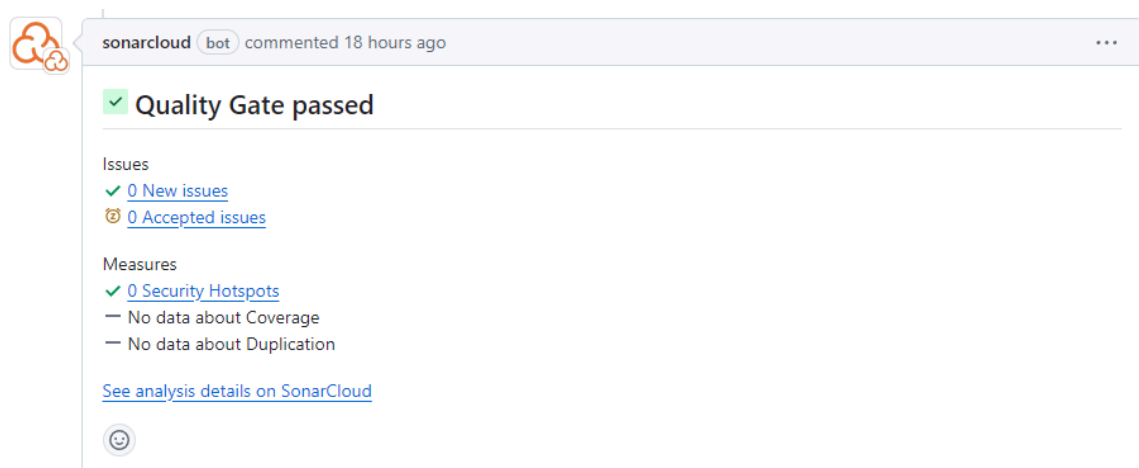
## 2.2 Code quality metrics and dashboards

Regarding the static code analysis, we used a **github-actions** workflow was defined that runs every time a commit is done to the main or dev branches or a pull request is open from

```
name: SonarCloud
on:
  push:
    branches:
      - dev
      - main
  pull_request:
    types: [opened, synchronize, reopened]
jobs:
  build:
    name: Build and analyze
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
        with:
          fetch-depth: 0
      - name: Set up JDK 17
        uses: actions/setup-java@v3
        with:
          java-version: 17
          distribution: 'zulu'
      - name: Cache SonarCloud packages
        uses: actions/cache@v3
        with:
          path: ~/.sonar/cache
          key: ${ runner.os }-sonar
          restore-keys: ${ runner.os }-sonar
      - name: Cache Maven packages
        uses: actions/cache@v3
        with:
          path: ~/.m2
          key: ${ runner.os }-m2-${ hashFiles('**/pom.xml') }
          restore-keys: ${ runner.os }-m2
      - name: Build and analyze
        env:
          SONAR_TOKEN: ${ secrets.SONAR_TOKEN }
        run: |
          cd backend/cinemax
          mvn -B -Dtest=!deti.tqs.cinemax.frontend.** verify org.sonarsource.scanner.maven:sonar-maven-plugin:sonar -Dsonar.projectKey=DiogoSilva1904_CineMax
```

This workflow includes all tests besides the functional tests.

If a pull request's workflow fails due to issues or insufficient coverage on the new code, its merge will be blocked until these issues are resolved and the workflow passes without any problems. Even if there are no issues, the pull request still requires review by at least one team member before it can be merged.



## Conditions

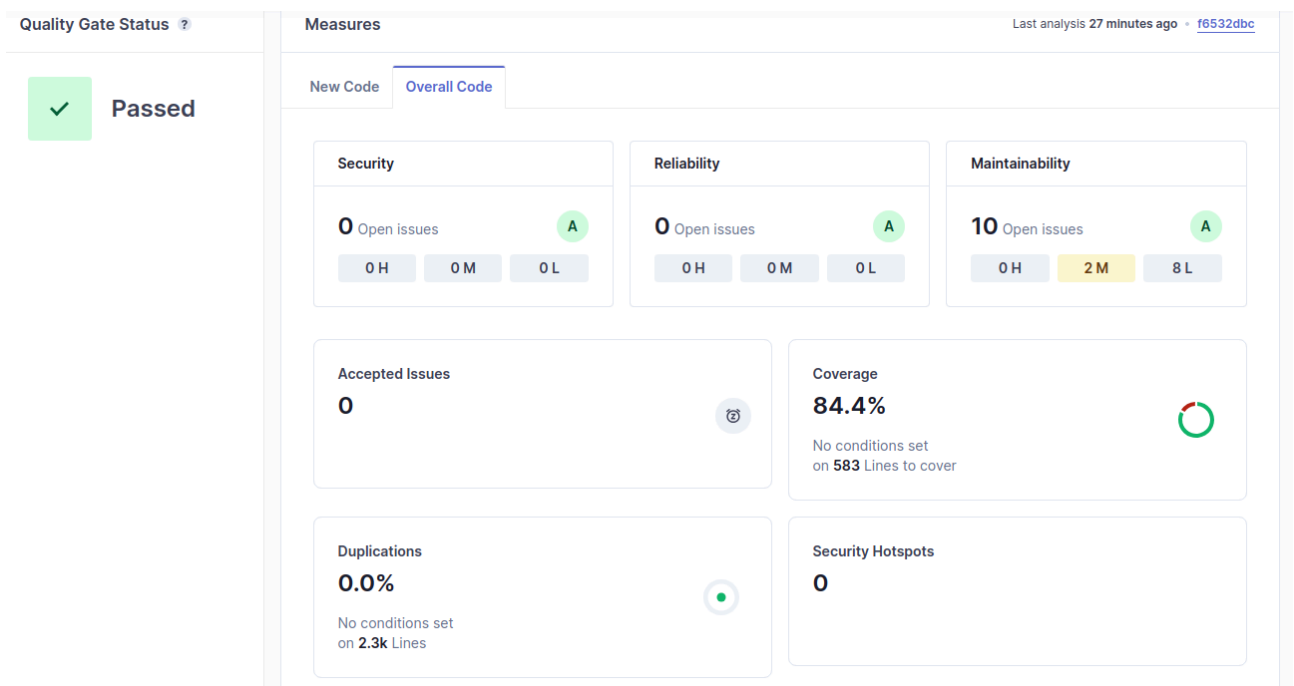
Your new code will be clean if: ?

No new bugs are introduced	Reliability rating is <b>A</b>
No new vulnerabilities are introduced	Security rating is <b>A</b>
New code has limited technical debt	Maintainability rating is <b>A</b>
All new security hotspots are reviewed	
New code is sufficiently covered by test	Coverage is greater than or equal to <b>80.0%</b> ?
New code has limited duplication	Duplicated Lines (%) is less than or equal to <b>3.0%</b> ?

[Which quality gates were defined? What was the rationale?]

Sonarcloud's quality gate was primarily based on the metric of code coverage, with a quality threshold of 80%, and the metric of line duplication, with a restriction of 3.0% line duplication.

The justification behind these metrics and values is based on the prediction that, with the quality referenced achieved, the project as a software project would be properly tested, i.e. functionally operating, and the DRY principle would also be followed, which would lead to a cleaner code to be analyzed for future developers.



## 3 Continuous delivery pipeline (CI/CD)

### 3.1 Development workflow

#### Workflow Adopted

For our project, we have adopted the **GitFlow workflow**. This workflow is well-suited for projects that require rigorous version control and multiple parallel developments. It allows for effective management of feature development, releases, and hotfixes.

- **Branches in GitFlow:**
  - **master:** This branch contains the production-ready code.
  - **dev:** The integration branch where features are merged before a release.
  - **SCRUM-#:** Branches derived from **dev** for working on new features.

#### Mapping to User Stories

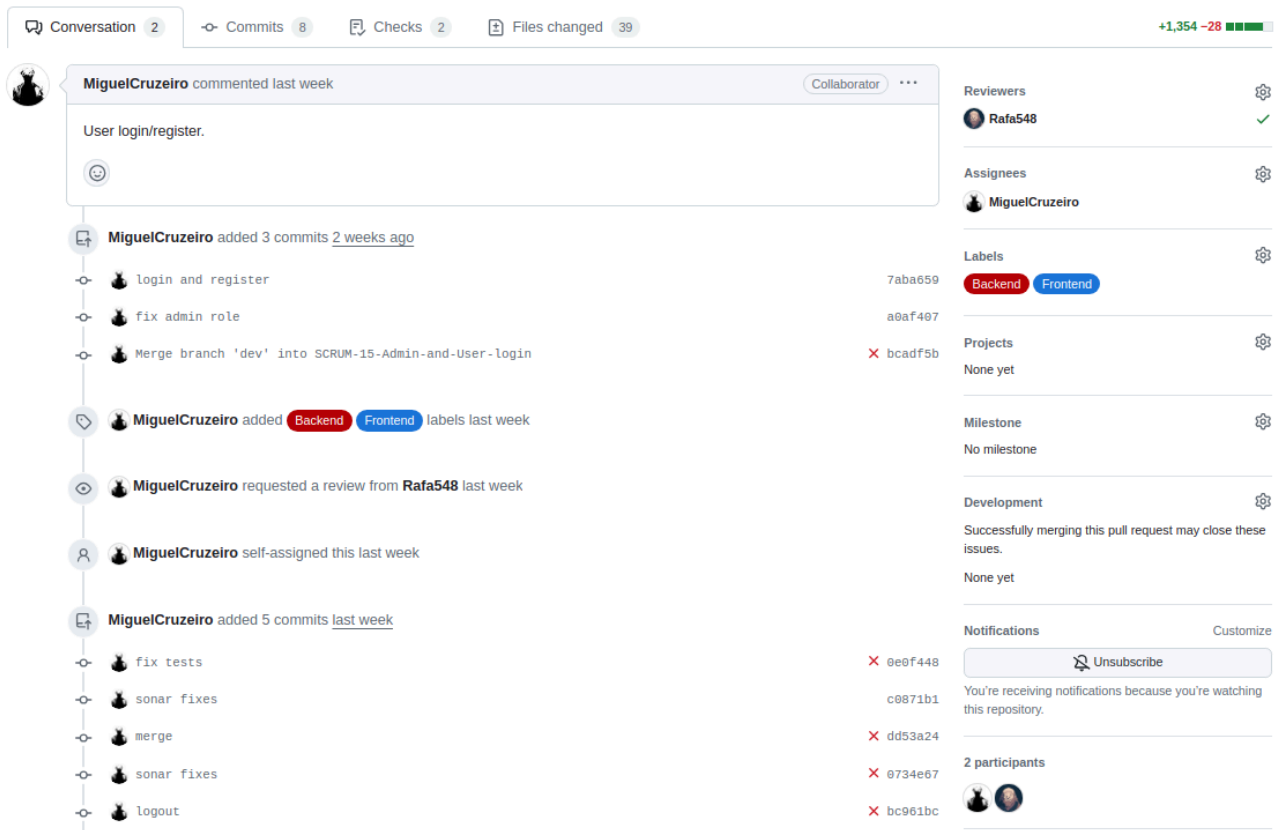
User stories guide our development process and are mapped to our GitFlow workflow as follows:

1. **User Story Creation:** When a user story is created and prioritized, a new feature branch is initiated.
  - **Naming Convention:** Feature branches follow the naming convention **SCRUM-`<story-id>-<description>`**, e.g., **SCRUM-1234-add-login**.
2. **Development:** The user story is implemented in the SCRUM branch. Developers commit changes with messages referencing the user story ID.
3. **Code Review and Integration:** Once the feature is complete, a pull request is created to merge the feature branch into **dev**.

We also established a Definition of Done (DoD), which outlines the specific criteria that must be met for a task to be considered complete.

Below is our **Definition of Done**:

1. **Functionality Implemented:** The code must implement the desired functionality as described in the task or user story.
2. **Unit Tests Written:** Unit tests covering the implemented functionality must be written and passed.
3. **Code Reviewed:** The code must undergo a review by at least one other team member to ensure quality, readability, and adherence to coding standards.
4. **Documentation Updated:** Any relevant documentation, including README files, API documentation, and comments within the code, must be updated to reflect changes introduced by the contribution. Integration
5. **Tests Passed:** If applicable, integration tests must be executed, and all relevant tests must pass.
6. **Quality Assurance (QA) Testing:** The contribution must undergo QA testing to verify its correctness, usability, and compatibility with other components of UA Smart Signage.
7. **Peer Approval:** A peer must approve the completion of the task based on the above criteria. By adhering to this Definition of Done, we ensure that all contributions meet the required standards of quality and readiness for deployment.



The screenshot shows a GitHub pull request (PR) for the repository 'User login/register.' by MiguelCruzeiro. The PR is in the 'dev' branch and targets the 'main' branch. It has 3 commits and 5 files changed. The PR is currently open and has 1 reviewer (Rafa548) and 1 assignee (MiguelCruzeiro). The PR is labeled with 'Backend' and 'Frontend' labels. The PR is currently in the 'Development' state. The PR is currently in the 'Development' state. The PR is currently in the 'Development' state.

### 3.2 CI/CD pipeline and tools

Github workflow was the primary CI tool used in the project, and a CI pipeline was implemented to automate the process of verifying the spring boot project build and evaluating code quality (with sonarcloud) for each push operation performed to the "dev" or "main" branch, which was also triggered in PRs targeting the branches.

### 3.3 System observability

Our system generates a stream of logs to standard output, providing insights into the operations being executed. Although we intended to direct these logs to a separate file for better organization and analysis, time constraints prevented us from doing it.

## 4 Software testing

### 4.1 Overall strategy for testing

In general, unit tests with entity mocking were built for each component of the backend service (service, controller, model, and repository), as well as IT tests to evaluate the functioning condition of the entire set of logical entities in a production-like environment.

Aside from the basic testing tools, such as Junit 5, it was used BDD cucumber tool to write behavior driven test for frontend functional

## 4.2 Functional testing/acceptance

Functional testing was conducted using Selenium and Cucumber to ensure the system performs as expected. Selenium automates browser actions to simulate user interactions, while Cucumber defines the test steps in plain language. The tests cover navigating to the website, selecting options, entering details, and confirming actions. This approach helps validate the end-to-end functionality, identifying issues early and testing a real world scenario. Below are illustrative code snippets demonstrating the implementation of these tests:

```
@buy_ticket
```

```
Feature: Buy ticket
```

```
Scenario: User creates an account, logs in, chooses a film and buys a ticket for a specific film
```

```
Given the user is on the log in page
```

```
When the user does not have an account, so clicks on the button to create an account
```

```
Then the user creates an account with username rafa5481, password 123456 and email rafa548@gmail.com
```

```
Then the user logs in with username rafa5481 and password 123456
```

```
When the user selects the first film
```

```
And chooses the first session
```

```
And selects the third seat
```

```
And clicks on reserve button
```

```
Then the user should see a success message
```

```
Then the user go to the tickets page
```

```
Then the user should see the ticket
```

```
@createmovieandsession
```

```
Feature: Create movie and Session
```

```
Scenario: User logs in, creates a film and adds a session
```

```
Given the user is on the log in page
```

```
Then the user logs in with username admin and password admin
```

```
When the user clicks on add movie button
```

```
And the user fills the form with title Drive, duration 128, studio Universal Studios  
and choose the first genre from the dropdown
```

```
Then the user should see the movie with title Drive in the list
```

```
Then the user goes to the sessions page
```

```
When the user clicks on add session button
```

```
And the user fills the form with date 2024-05-30, time 21:00, choose the first room from the dropdown  
and the movie with title Drive from the dropdown
```

```
Then the user logs out
```



## 4.3 Unit tests

To test each component independently, we implemented unit tests using JUnit, leveraging mocking to isolate dependencies. Our approach followed a top-down methodology: starting with controller tests using mock services, then service tests with mock repositories, and finally repository tests. Given our detailed knowledge of the system's structure, these unit tests were developed with an open-box perspective. The tests are integrated into the CI/CD pipeline, ensuring they are automatically executed as part of the SonarCloud workflow, which helps maintain code quality and catch issues early.

### 4.3.1 Controller Tests

Controller tests were done using JUnit and MockMvc to ensure that the endpoints behave as expected. By annotating the test class with “@WebMvcTest”, we focus on the web layer, and with “@AutoConfigureMockMvc(addFilters = false)”, we configure MockMvc to bypass security filters, allowing us to call endpoints without needing JWT tokens.

MockMvc is used to simulate HTTP requests and verify responses, while Mockito mock dependencies like services to isolate the controller's functionality. For example, a test might mock a service to return a specific result and then use MockMvc to send a GET request to an endpoint, verifying the response status and JSON content to ensure the controller processes requests correctly.

These tests follow a structured approach: setting up mock behavior, performing HTTP requests, and asserting the outcomes. Following this there will be some examples of some controller tests:

```
@Test
void testCreateFileEndpoint() throws Exception {
    CustomFile file = new CustomFile();
    file.setId(1L);
    file.setName("testFile");

    when(fileService.createFile(Mockito.any())).thenReturn(file);

    mvc.perform(post("/api/files").contentType(MediaType.APPLICATION_JSON)
        .content(objectMapper.writeValueAsString(file)))
        .andExpect(status().isCreated())
        .andExpect(jsonPath("$.name", is("testFile")));

    verify(fileService, Mockito.times(1)).createFile(Mockito.any());
}
```

```

@Test
void testGetMovieById() throws Exception{
    Movie movie1 = new Movie();
    movie1.setId(1L);
    movie1.setTitle("Oppenheimer");
    movie1.setCategory("Action");
    movie1.setGenre("Thriller");
    movie1.setStudio("Studio X");
    movie1.setDuration("120min");

    when(movieService.getMovieById(1L)).thenReturn(movie1);

    mvc.perform(get("/api/movies/1"))
        .andExpect(status().isOk())
        .andExpect(jsonPath("$.title", is("Oppenheimer")))
        .andExpect(jsonPath("$.category", is("Action")))
        .andExpect(jsonPath("$.genre", is("Thriller")))
        .andExpect(jsonPath("$.studio", is("Studio X")))
        .andExpect(jsonPath("$.duration", is("120min")));
}

```

### 4.3.2 Service Tests

Service tests are conducted using JUnit and Mockito to ensure that the service layer functions correctly and handles business logic appropriately. By annotating the test class with `@ExtendWith(MockitoExtension.class)`, we enable Mockito's annotations, such as `@Mock` and `@InjectMocks`, to create mock instances of dependencies and inject them into the service being tested. This isolation allows us to focus solely on the service's behavior without involving the actual implementations of its dependencies.

For example, consider the `FileServiceTest` class, which tests the `FileService` methods using mocked `FileRepository` instances. Each test follows a structured approach: setting up mock behavior, invoking service methods, and asserting the outcomes to verify the service's handling of data and interactions with the repository.

Below is an illustrative code snippet demonstrating these principles:

```

@Test
@Order(10)
void testDownloadFileById_FileExistsAndIsReadable() throws IOException {

    // Creating a temporary file
    Path tempFile = Files.createTempFile("test", ".png");
    byte[] content = "Hello, World!".getBytes();
    Files.write(tempFile, content);

    MockMultipartFile mockMultipartFile = new MockMultipartFile(
        "file", // parameter name
        tempFile.getFileName().toString(), // file name
        "image/png", // content type
        Files.readAllBytes(tempFile) // content as byte array
    );

    FilesClass filesClass = new FilesClass(null, mockMultipartFile);
    CustomFile savedFile = service.createFile(filesClass);

    // Paths to the saved file and the root directory
    Path filePath = Paths.get(savedFile.getPath());
    String rootPath = System.getProperty("user.dir");
    String fileDirectory = rootPath + "/uploads/";

    when(repository.findById(1L)).thenReturn(Optional.of(savedFile));

    ResponseEntity<Resource> response = service.downloadFileById(1L);

    assertEquals(HttpStatus.OK, response.getStatusCode());
    assertNotNull(response.getBody());
    assertEquals(filePath.getFileName().toString(), response.getBody().getFilename());

    HttpHeaders headers = response.getHeaders();
    assertTrue(headers.containsKey(HttpHeaders.CONTENT_DISPOSITION));
    assertEquals("attachment; filename=\"" + filePath.getFileName().toString() + "\"",
        headers.getFirst(HttpHeaders.CONTENT_DISPOSITION));

    File file = new File(fileDirectory + savedFile.getName());
    assertTrue(file.exists());

    // Clean up
    if (file.exists()) {
        file.delete();
    }
}

```

### 4.3.3 Repository tests

Repository tests were conducted to ensure that each entity's repository correctly filters and returns the entities.

To achieve this, we use an in-memory database configured by the annotation "@DataJpaTest" which sets up Spring Data JPA that allows us to create and persist the entities we want to test.

```

@Test
void testFindByUserUsername() {
    AppUser user = new AppUser();
    user.setUsername("user2");
    user.setPassword("password");
    user.setEmail("user2@example.com");
    user.setRole("USER");
    userRepository.save(user);

    Session session = new Session();
    session.setDate("2024-06-01");
    session.setTime("12:00");
    sessionRepository.save(session);

    Reservation reservation = new Reservation();
    reservation.setUser(user);
    reservation.setPrice(10);
    reservation.setUsed(false);
    reservation.setSession(session);
    reservationRepository.save(reservation);

    List<Reservation> reservations = reservationRepository.findByUserUsername("user2");
    assertThat(reservations).hasSize(1);
}

```

## 4.4 System and integration testing

The goal of integration tests is to check if the components of the system work together as expected. As the unitary tests, we used the open-box strategy to develop the integration tests.

For these tests we used the framework TestContainers, which is a powerful tool for running integration tests, allowing developers to spin up lightweight, throwaway instances of common databases or anything else that can run in a Docker container. Each integration test class has its own container.

```
@Testcontainers
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT, properties = {"spring.profiles.active=test"})
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
public class MovieIT {

    @Container
    public static GenericContainer container = new GenericContainer("mysql:latest")
        .withExposedPorts(3306)
        .withEnv("MYSQL_ROOT_PASSWORD", "rootpass")
        .withEnv("MYSQL_DATABASE", "cinemax")
        .withEnv("MYSQL_USER", "user")
        .withEnv("MYSQL_PASSWORD", "secret");

    @DynamicPropertySource
    static void configureProperties(DynamicPropertyRegistry registry) {
        String jdbcUrl = "jdbc:mysql://" + container.getHost() + ":" + container.getMappedPort(3306) + "/cinemax";
        registry.add("spring.datasource.url", () -> jdbcUrl);
        registry.add("spring.datasource.username", () -> "user");
        registry.add("spring.datasource.password", () -> "secret");
    }
}
```

To load the data to use in the integration tests we created a Datalnit class that runs in the test profile and implements the "CommandLineRunner" that allows it to run this class after the Spring Boot application context has been loaded. Inside this class, we create all the necessary data for the integration tests. All of the tests in the IT test classes use the @Order annotation to ensure that tests don't have any conflicts.

We create a Docker container for our tests using the Testcontainers library. Specifically, we use the "GenericContainer" class to create a container with a MySQL image.

Additionally, the @DynamicPropertySource annotation dynamically registers the necessary properties (such as database URL, username, and password) for the Spring context. This set up is equal for all the test classes.

To access the endpoints, we use JWT (JSON Web Token) authentication. A JWT token is generated upon login. Therefore, before each test, we log in with admin credentials to generate a token, which is then used to access the endpoints.

```

@LocalServerPort
private int port;

@Autowired
private TestRestTemplate restTemplate;

private static String jwtToken;
private static final TestRestTemplate restTemplate1 = new TestRestTemplate();

@BeforeAll
static void setup(@LocalServerPort int port1) {
    String requestBody = "{\"username\":\"" + "admin" + "\",\"password\":\"" + "admin" + "\"}";

    HttpHeaders headers = new HttpHeaders();
    headers.set("Content-Type", "application/json");

    HttpEntity<String> requestEntity = new HttpEntity<>(requestBody, headers);

    ResponseEntity<String> response = restTemplate1.exchange(
        "http://localhost:" + port1 + "/api/login",
        HttpMethod.POST,
        requestEntity,
        String.class
    );

    assertEquals(HttpStatus.OK, response.getStatusCode());

    String responseBody = response.getBody();

    System.out.println(responseBody);

    JsonElement jsonElement = JsonParser.parseString(responseBody);
    JsonObject jsonObject = jsonElement.getAsJsonObject();

    jwtToken = jsonObject.get("jwt").getAsString();
}

```

Following, there will be some examples of tests that were conducted:

```

@Test
@Order(5)
void whenCreateSessionWithOverlappingSession_ReturnBadRequest(){
    HttpHeaders headers = new HttpHeaders();
    headers.set("Content-Type", "application/json");
    headers.setBearerAuth(jwtToken);

    HttpEntity<?> entity1 = new HttpEntity<>(headers);

    ResponseEntity<Room> response = restTemplate.exchange("http://localhost:" + port + "/api/rooms/1", HttpMethod.GET, entity1, Room.class);
    assertEquals(HttpStatus.OK, response.getStatusCode());
    Room room = response.getBody();

    ResponseEntity<Movie> response2 = restTemplate.exchange("http://localhost:" + port + "/api/movies/1", HttpMethod.GET, entity1, Movie.class);
    Movie movie = response2.getBody();

    Session session = new Session();
    session.setDate("2024-05-23");
    session.setTime("20:00");
    session.setMovie(movie);
    session.setRoom(room);
    session.setBookedSeats(List.of("A2", "A1"));

    HttpEntity<Session> entity = new HttpEntity<>(session, headers);

    ResponseEntity<Session> response1 = restTemplate.exchange("http://localhost:" + port + "/api/sessions", HttpMethod.POST, entity, Session.class);

    assertEquals(HttpStatus.BAD_REQUEST, response1.getStatusCode());
}

```

whenCreateSessionWithOverlappingSession\_ReturnBadRequest->In this test we try to create a session in a room and in a date and hour that is already booked, so the result expected is a bad request.

```

@Test
@Order(8)
void whenCreateReservation_SeatAlreadyBooked() {
    HttpHeaders headers = new HttpHeaders();
    headers.set("Content-Type", "application/json");
    headers.setBearerAuth(jwtToken);

    HttpEntity<?> entity1 = new HttpEntity<>(headers);

    ResponseEntity<Session> response = restTemplate.exchange("http://localhost:" + port + "/api/sessions/1", HttpMethod.GET, entity1, Session.class);
    assertEquals(HttpStatus.OK, response.getStatusCode());
    Session session = response.getBody();

    ResponseEntity<AppUser> response2 = restTemplate.exchange("http://localhost:" + port + "/api/users/2", HttpMethod.GET, entity1, AppUser.class);
    assertEquals(HttpStatus.OK, response2.getStatusCode());
    AppUser user = response2.getBody();

    Reservation reservation = new Reservation();
    reservation.setPrice(10);
    reservation.setSeatNumbers(List.of("A1"));
    reservation.setSession(session);
    reservation.setUser(user);

    HttpEntity<Reservation> entity = new HttpEntity<>(reservation, headers);

    ResponseEntity<Reservation> response1 = restTemplate.exchange("http://localhost:" + port + "/api/reservations", HttpMethod.POST, entity, Reservation.class);

    assertEquals(HttpStatus.CONFLICT, response1.getStatusCode());
}

```

whenCreateReservation\_SeatAlreadyBooked->In this test we try to create a reservation but the seat we choose is already booked so the expected result is conflict.