TQS: Quality Assurance manual

João Dourado [108636], José Mendes [107188], Miguel Figueiredo [108287], Vítor Santos [107186]
v2024-06-02

1 Project management

1.1 Team and roles

- Product Owner – Miguel Figueiredo
- Team Leader - João Dourado
- DevOps Master - Vítor Santos
- QA Engineer – José Mendes
- Developer - Everyone

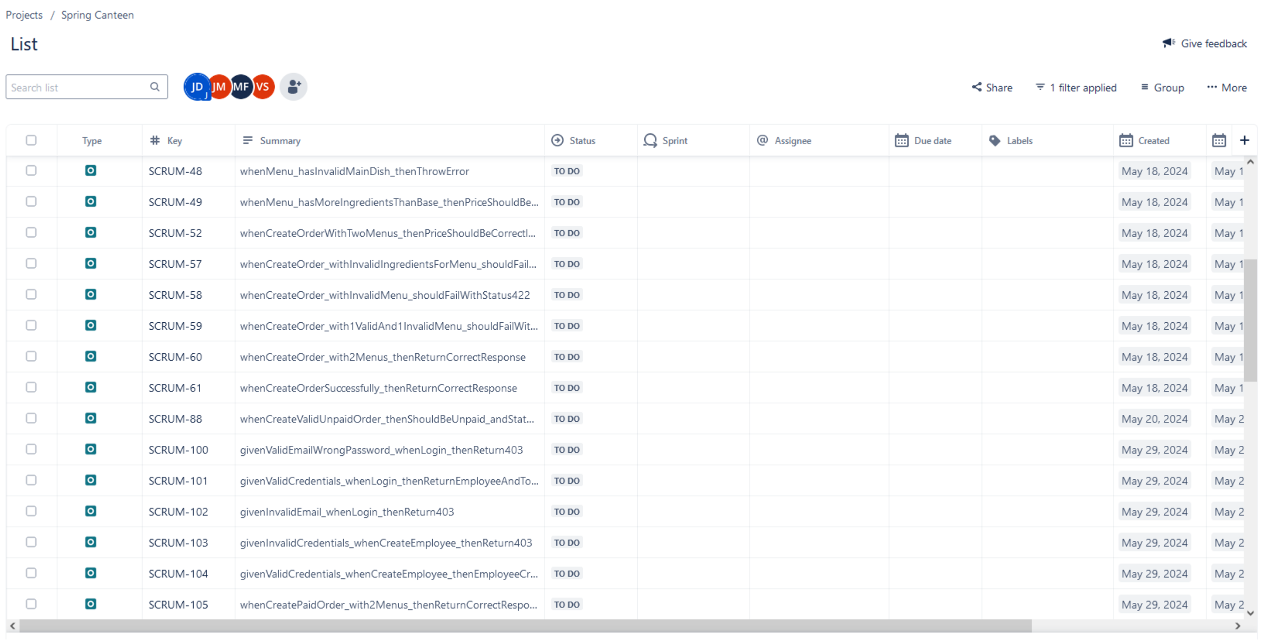
1.2 Agile backlog management and work assignment

In our project, we utilized Agile practices to manage the backlog and assign work efficiently, with a focus on user stories. JIRA was employed to create, prioritize, and track user stories, ensuring a streamlined workflow and clear task distribution among team members. The flow the team adopted was the following:

- At the start of each iteration, new defined user stories are written on the backlog. Each team member selects an issue and begins working on it;
- Upon completing a task, the member ensures the definition of done is satisfied, then notifies and assigns other members to review the pull request. The team reviews and either approves or disapproves the pull request, iterating until the code meets the acceptance criteria;
- Once the pull request is accepted, it is merged into the 'dev' branch. The issue is then marked as closed, and the member selects a new user story to work on.

All user stories were given a story point estimate. Our convention was a number between 1 through 5, with 1 being a relatively quick to implement story, and 5 being a user story with a higher duration of implementation.

Our backlog also has a log of all tests run, thanks to the use of X-ray for Jira. It provides a full log of tests run when pull requests to the **dev** or **main** branches are made, thanks to a workflow run that sends test reports to X-ray (more on this workflow later). Here's an example list of those tests:



Type	Key	Summary	Status	Sprint	Assignee	Due date	Labels	Created	
●	SCRUM-48	whenMenu_hasInvalidMainDish_thenThrowError	TO DO					May 18, 2024	May 1
●	SCRUM-49	whenMenu_hasMoreIngredientsThanBase_thenPriceShouldBe...	TO DO					May 18, 2024	May 1
●	SCRUM-52	whenCreateOrderWithTwoMenus_thenPriceShouldBeCorrectl...	TO DO					May 18, 2024	May 1
●	SCRUM-57	whenCreateOrder_withInvalidIngredientsForMenu_shouldFail...	TO DO					May 18, 2024	May 1
●	SCRUM-58	whenCreateOrder_withInvalidMenu_shouldFailWithStatus422	TO DO					May 18, 2024	May 1
●	SCRUM-59	whenCreateOrder_with1ValidAnd1InvalidMenu_shouldFailWit...	TO DO					May 18, 2024	May 1
●	SCRUM-60	whenCreateOrder_with2Menus_thenReturnCorrectResponse	TO DO					May 18, 2024	May 1
●	SCRUM-61	whenCreateOrderSuccessfully_thenReturnCorrectResponse	TO DO					May 18, 2024	May 1
●	SCRUM-88	whenCreateValidUnpaidOrder_thenShouldBeUnpaid_andStat...	TO DO					May 20, 2024	May 2
●	SCRUM-100	givenValidEmailWrongPassword_whenLogin_thenReturn403	TO DO					May 29, 2024	May 2
●	SCRUM-101	givenValidCredentials_whenLogin_thenReturnEmployeeAndTo...	TO DO					May 29, 2024	May 2
●	SCRUM-102	givenInvalidEmail_whenLogin_thenReturn403	TO DO					May 29, 2024	May 2
●	SCRUM-103	givenInvalidCredentials_whenCreateEmployee_thenReturn403	TO DO					May 29, 2024	May 2
●	SCRUM-104	givenValidCredentials_whenCreateEmployee_thenEmployeeCr...	TO DO					May 29, 2024	May 2
●	SCRUM-105	whenCreatePaidOrder_with2Menus_thenReturnCorrectRespo...	TO DO					May 29, 2024	May 2

Figure 1: Xray Test Execution List

Unfortunately, due to unknown reasons (we followed the given x-ray tutorial), we could not link tests to User Stories automatically, as X-ray was meant for.

2 Code quality management

2.1 Guidelines for contributors (coding style)

In order to promote consistency and code readability and maintainability, the whole team followed the definition of Google's coding standards for source code in the Java™ Programming Language – Google Java Style Guide. Furthermore, a github-actions workflow was set up using google-java-format so that the Java code committed into the shared repository would get reformatted to comply with the Google Java Style.

Moreover, SonarCloud was used to detect bad smells and other issues, being required that every developer had installed in his IDE SonarLint to detect them before the SonarQube Code Analysis.

When committing code to the remote repository, we followed the Conventional Commits specification, so that commits can be self-explanatory and specific (we only started following this convention halfway through implementation).

2.2 Code quality metrics and dashboards

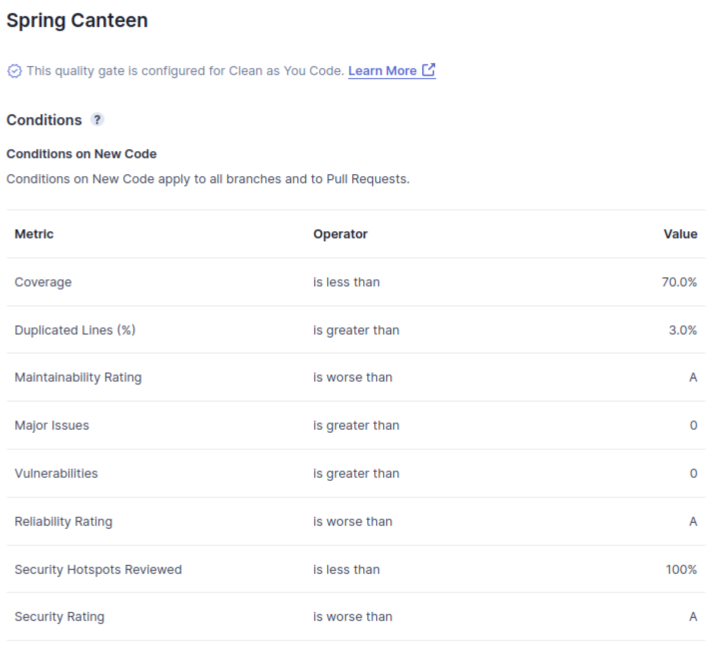
Regarding the static code analysis, a **github-actions** workflow was defined that runs every time a commit is done to the main and dev branches as well as every time a pull request is opened, synchronized or reopened.

```
name: SonarCloud

on:
  push:
    branches:
      - main
      - dev
    paths:
      - "backend/springcanteen/**"
  pull_request:
    types: [opened, synchronize, reopened]
```

Figure 2: Excerpt of Github-actions workflow for SonarCloud

This workflow encompasses all tests but the functional tests. Furthermore, in the case of pull requests, if at some point the workflow fails due to, for example, security hotspots on the static code analysis or not passing tests, then the merger of said pull request will be blocked until the matter is resolved. The same behavior is expected if the pull request is yet to be approved by at least one person.

The defined quality gate is based on the default quality gate provided by SonarCloud with some tweaks regarding coverage, major issues and vulnerabilities. This definition was defined before the start of development, and it is important to mention that the quality gate has not been altered since then. As we used Lombok to automatically generate methods, it is important to note that these methods weren't considered for coverage analysis as the following was added in the lombok.config file: lombok.addLombokGeneratedAnnotation = true.



Spring Canteen

This quality gate is configured for Clean as You Code. Learn More

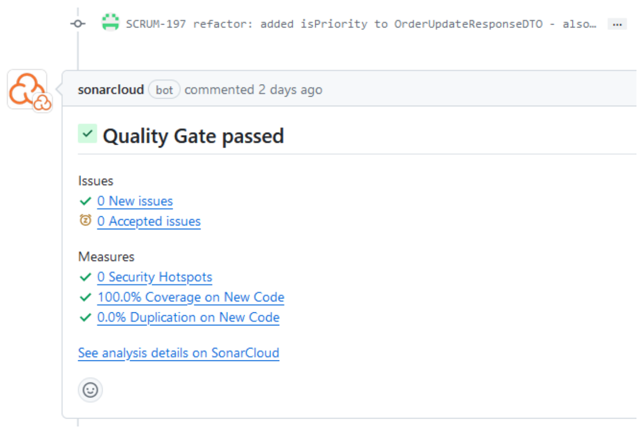
Conditions

Conditions on New Code

Conditions on New Code apply to all branches and to Pull Requests.

Metric	Operator	Value
Coverage	is less than	70.0%
Duplicated Lines (%)	is greater than	3.0%
Maintainability Rating	is worse than	A
Major Issues	is greater than	0
Vulnerabilities	is greater than	0
Reliability Rating	is worse than	A
Security Hotspots Reviewed	is less than	100%
Security Rating	is worse than	A

Figure 3: Custom Quality Gate for SpringCanteen



SCRUM-197 refactor: added isPriority to OrderUpdateResponseDTO - also…

sonarcloud bot commented 2 days ago

Quality Gate passed

Issues
✓ 0 New issues
0 Accepted issues

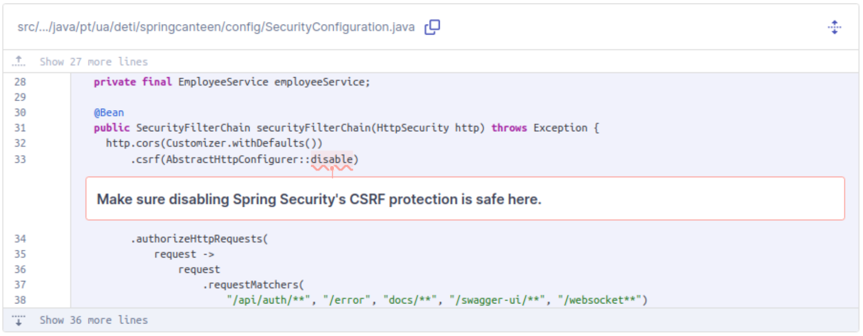
Measures
✓ 0 Security Hotspots
✓ 100.0% Coverage on New Code
✓ 0.0% Duplication on New Code

See analysis details on SonarCloud

Figure 4: Sonar Cloud Feedback on Pull Request

However, it's also important to note that some of the reported issues by SonarCloud were accepted following one of these aspects:

- they weren't major concerns regarding the objective of this project: achieve an MVP that respects the core features of Spring Canteen (e.g. disabling **csrf** in the securityFilterChain of our security configuration as seen in Fig. 5);



src/.../java/pt/ua/deti/springcanteen/config/SecurityConfiguration.java

Show 27 more lines

```
    private final EmployeeService employeeService;

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
      http.cors(Customizer.withDefaults())
          .csrf(AbstractHttpConfigurer::disable)
```

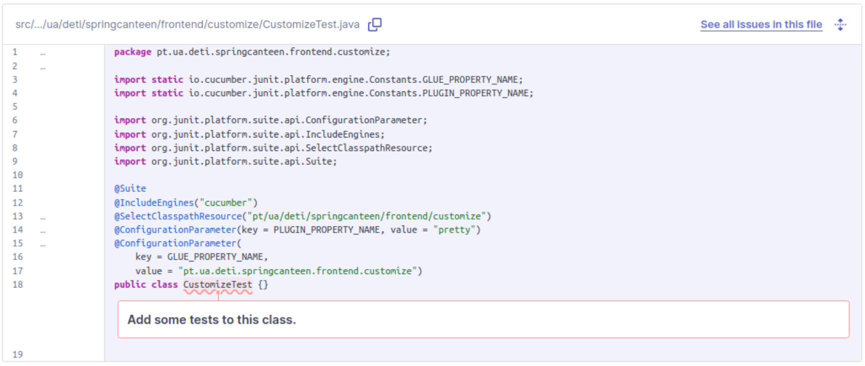
Make sure disabling Spring Security's CSRF protection is safe here.

```
          .authorizeHttpRequests(
              request ->
                  request
                      .requestMatchers(
                          "/api/auth/**", "/error", "docs/**", "/swagger-ui/**", "/websocket**")
```

Show 36 more lines

Figure 5: Flagged Security Hotspot by Sonar Cloud

- the issue flagged by Sonar Cloud does not actually qualify as an issue (e.g. lack of tests in cucumber test definition as seen in Fig. 6).



src/.../ua/deti/springcanteen/frontend/customize/CustomizeTest.java

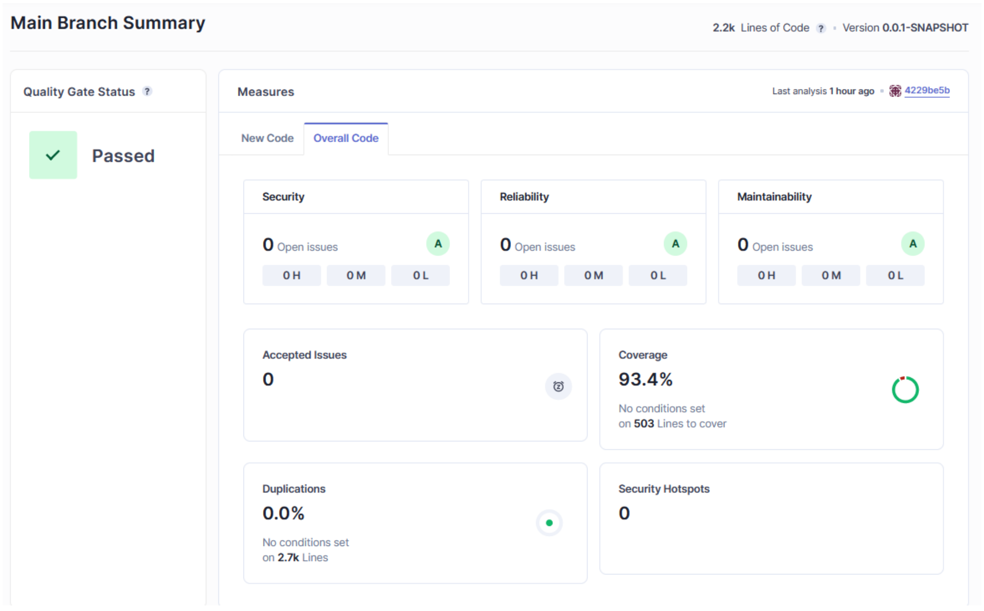
See all issues in this file

```
package pt.ua.deti.springcanteen.frontend.customize;

import static io.cucumber.junit.platform.engine.Constants.GLUE_PROPERTY_NAME;
import static io.cucumber.junit.platform.engine.Constants.PLUGIN_PROPERTY_NAME;

import org.junit.platform.suite.api.ConfigurationParameter;
import org.junit.platform.suite.api.IncludeEngines;
import org.junit.platform.suite.api.SelectClasspathResource;
import org.junit.platform.suite.api.Suite;

@Suite
@IncludeEngines("cucumber")
@SelectClasspathResource("pt/ua/deti/springcanteen/frontend/customize")
@ConfigurationParameter(key = PLUGIN_PROPERTY_NAME, value = "pretty")
@ConfigurationParameter(
    key = GLUE_PROPERTY_NAME,
    value = "pt.ua.deti.springcanteen.frontend.customize")
public class CustomizeTest {}
```

Add some tests to this class.

Figure 6: Issue Flagged by Sonar Cloud regarding lack of testing in class

Regarding the Sonar Cloud results they are very good as coverage is high and there are a total of 0 issues at the end of development.



Main Branch Summary

2.2k Lines of Code · Version 0.0.1-SNAPSHOT

Quality Gate Status: Passed

Measures — Last analysis 1 hour ago · 4229be5b

New Code | Overall Code

Security	Reliability	Maintainability
0 Open issues (A)	0 Open issues (A)	0 Open issues (A)
0 H · 0 M · 0 L	0 H · 0 M · 0 L	0 H · 0 M · 0 L

Accepted Issues: 0

Coverage: 93.4% — No conditions set on 503 Lines to cover

Duplications: 0.0% — No conditions set on 2.7k Lines

Security Hotspots: 0

Figure 7: Final Sonar Cloud Summary

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

The adopted development workflow was the gitflow workflow, which uses two branches to record the history of the project: **main** and **dev**. The first branch stores the official release history, and the latter serves as an integration branch for features. Furthermore, the developer shall create a new branch from **dev** every time a new feature is to be implemented. When it is in fact implemented, passes all tests, passes the quality gate and has been reviewed, the feature branch can then be merged into **dev**. In turn, the development branch will be merged into the main branch every time a new release is to be deployed. However, being the agile development processes driven by user stories, efforts have been made to break features into separate user stories that will in turn constitute a branch in the repository.

As for branch protection rules, we defined that the **main** and **dev** branches are protected branches and require a pull request with approval by review of at least 1 developer to accept new code.

In terms of the code review process, there are a few considerations to be aware of. As mentioned previously, upon each pull request its merger is blocked until one reviewer approves said pull request (also making it impossible for a person to approve his own pull request). We would have liked to mandate two approved reviews for each pull request to proceed, but that would extremely slow down the development process as the development team only had 4 members, most of the members with different schedules.

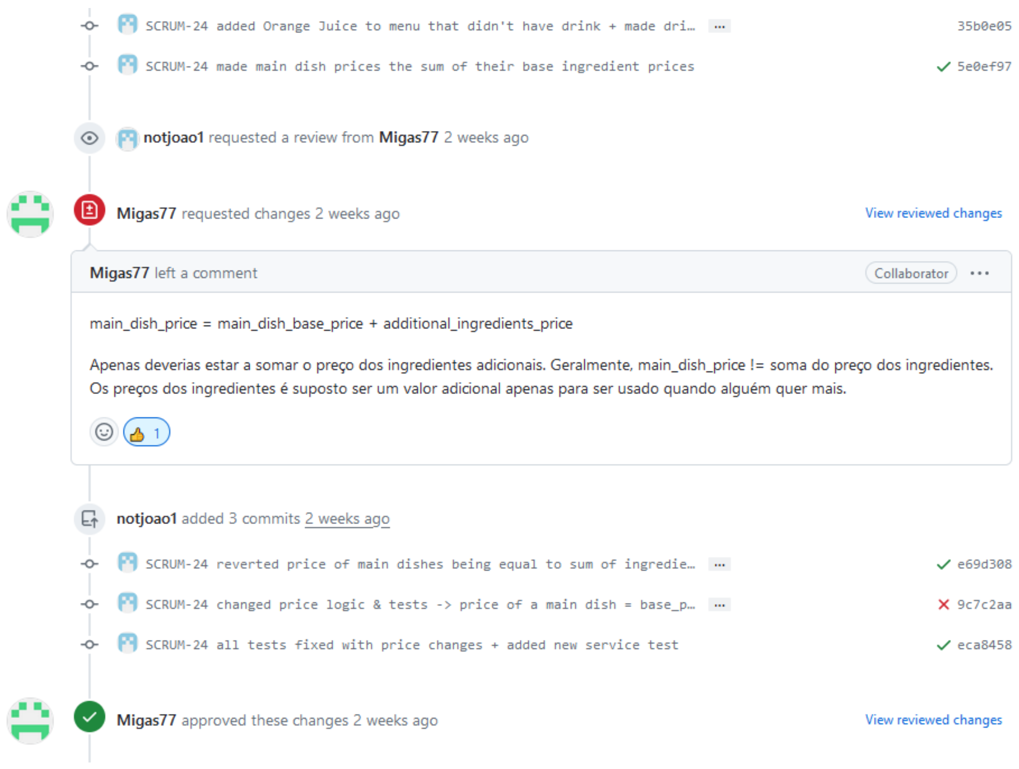
Regarding, the process itself of a reviewer analyzing the code it had in goal guaranteeing the following aspects:

- The code is explicit and easy to understand - “clean code”;
- The code demonstrates good code design practices;
- The code logic is aligned with the defined requirements;
- The suite of tests encompasses all relevant logical branches.

With the goal of delivering value with every sprint and minimizing rework, a DoD (Definition of Done) has been established. The following set of requirements constitutes a shared understanding among all team members to consider a product increment completed and ready for release:

- The code is written and with appropriate tests for the code developed, such as functional tests, controller tests, service tests, repository tests and integration tests;
- All tests are passing;
- The code’s quality is checked by a static analysis tool and is passing the quality gate;
- The user story implementation meets the acceptance criteria;
- The submitted code has been reviewed by at least 1 person.

The image below shows an example of code review being effective at catching an implementation mistake, that did not follow the defined criteria for a particular user story. One of the developers was misinterpreting the way prices are calculated whenever an order with customization is made, but, thanks to the review process, the mistake was caught and acted upon, resulting in code that met the defined criteria.



SCRUM-24 added Orange Juice to menu that didn't have drink + made dri… … 35b0e05

SCRUM-24 made main dish prices the sum of their base ingredient prices ✓ 5e0ef97

notjoao1 requested a review from Migas77 2 weeks ago

Migas77 requested changes 2 weeks ago View reviewed changes

Migas77 left a comment Collaborator

main_dish_price = main_dish_base_price + additional_ingredients_price

Apenas deverias estar a somar o preço dos ingredientes adicionais. Geralmente, main_dish_price != soma do preço dos ingredientes. Os preços dos ingredientes é suposto ser um valor adicional apenas para ser usado quando alguém quer mais.

👍 1

notjoao1 added 3 commits 2 weeks ago

SCRUM-24 reverted price of main dishes being equal to sum of ingredie… … ✓ e69d308

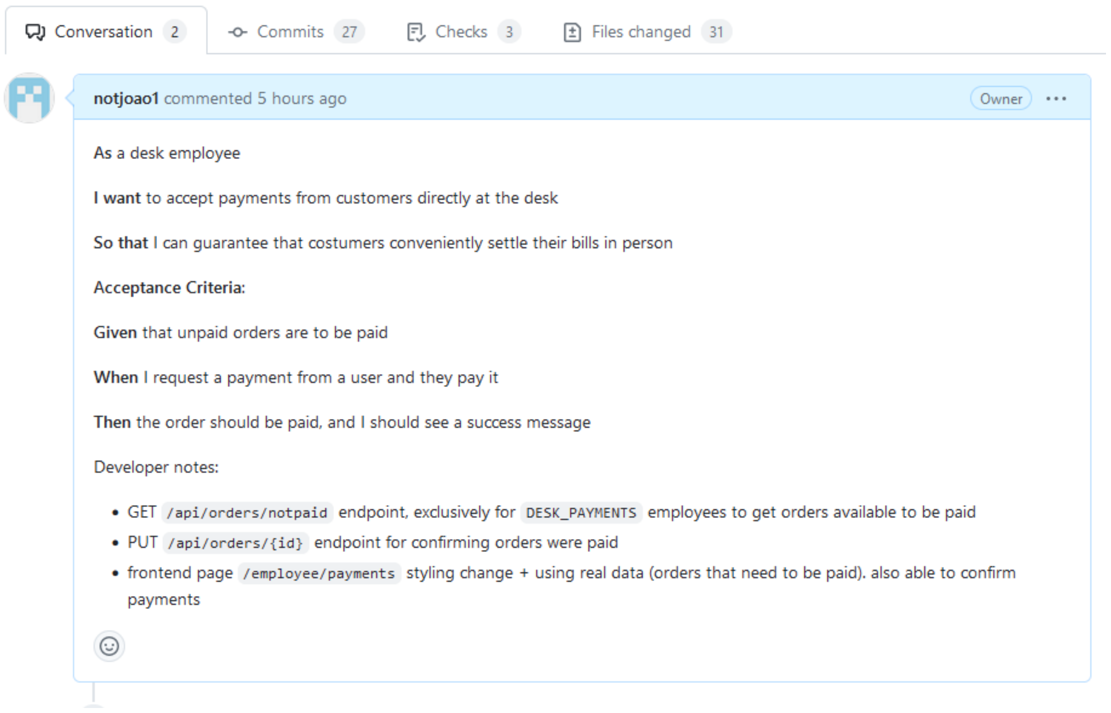
SCRUM-24 changed price logic & tests -> price of a main dish = base_p… … ✗ 9c7c2aa

SCRUM-24 all tests fixed with price changes + added new service test ✓ eca8458

Migas77 approved these changes 2 weeks ago View reviewed changes

Figure 8: Code Review Interaction

When making a pull requests, developers also followed a specific convention for pull request descriptions – first, write the User Story description along with its acceptance criteria, and additionally, include developer notes to explain what implementation outcomes came out of the given user story. The image below illustrates the explained convention.



notjoao1 commented 5 hours ago Owner

As a desk employee

I want to accept payments from customers directly at the desk

So that I can guarantee that costumers conveniently settle their bills in person

Acceptance Criteria:

Given that unpaid orders are to be paid

When I request a payment from a user and they pay it

Then the order should be paid, and I should see a success message

Developer notes:

- GET `/api/orders/notpaid` endpoint, exclusively for `DESK_PAYMENTS` employees to get orders available to be paid
- PUT `/api/orders/{id}` endpoint for confirming orders were paid
- frontend page `/employee/payments` styling change + using real data (orders that need to be paid). also able to confirm payments

Figure 9: Pull Request Convention

3.2 CI/CD pipeline and tools

The CI/CD pipeline was built by creating multiple workflows with github-actions. There are essentially 4 different workflows:

- **Formatting Workflow** – It runs every time code is pushed into branches main and dev. Utilizes google-java-format to format all code according to the Google Java Style;
- **Sonar Cloud and Xray Workflow** – It runs every time code is pushed into branches main and dev and every time a pull request is opened, synchronized and reopened. In this workflow, all tests but the functional tests are run and the SonarCloud analysis is triggered. This workflow is also responsible for posting the tests results to Xray;
- **Selenium Tests With Cucumber Workflow** - It runs every time code is pushed into branches main and dev and every time a pull request is opened, synchronized and reopened. In this workflow, functional tests with selenium and cucumber are run. This workflow is separated from the previous workflow as measures needed to be taken regarding the installation of firefox drivers and the startup of a docker compose of the SpringCanteen application;
- **Deployment Workflow –** every time code is pushed into the main branch, a workflow is triggered for deploying SpringCanteen, here's how it works. Using **Github Runners** (self-hosted at a DETI UA virtual machine:

1. First off, code must be pushed to the **main** branch, which typically (unless forced) happens only through pull request.

2. When code is checked into the **main** branch, a deployment workflow is triggered, which uses Github Actions to notify a self-hosted runner (running in the DETI virtual machine) listening to workflow requests through a background process.

3. When the self-hosted runner receives the deployment request, it rebuilds all containers (**keeping volumes intact – they are named containers and we don't delete them**) using the production configuration and runs them once again, exposing only port 80 to the outside world.

4. Now, SpringCanteen is successfully deployed and can be used by end users.

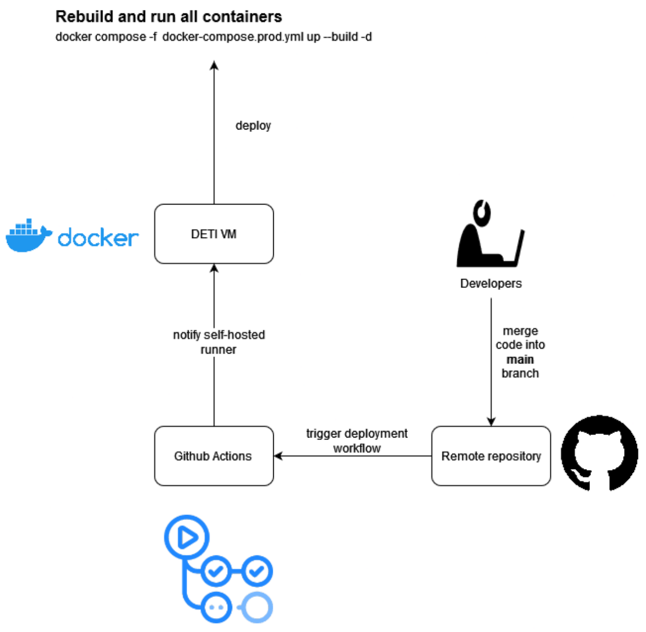
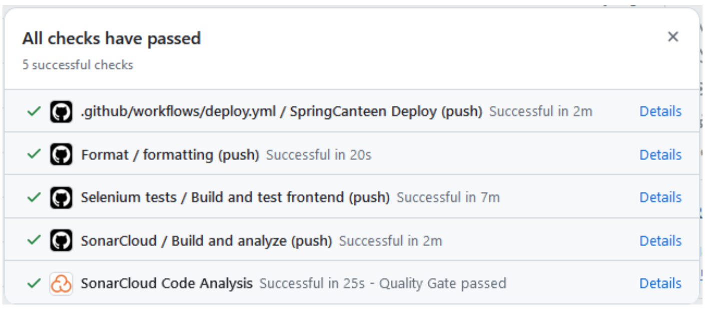

Rebuild and run all containers
docker compose -f docker-compose.prod.yml up --build -d
deploy
docker
DETI VM
Developers
notify self-hosted runner
merge code into **main** branch
Github Actions
trigger deployment workflow
Remote repository

Figure 10: Deployment Workflow

For handling database schema migrations, we used **Flyway** and reported schema changes with separate SQL files. That way, whenever we deploy Spring Canteen, all schema changes are automatically applied in the correct order, ensuring consistency, and reducing the risk of errors during deployment.



All checks have passed
5 successful checks

✓ .github/workflows/deploy.yml / SpringCanteen Deploy (push) Successful in 2m Details
✓ Format / formatting (push) Successful in 20s Details
✓ Selenium tests / Build and test frontend (push) Successful in 7m Details
✓ SonarCloud / Build and analyze (push) Successful in 2m Details
✓ SonarCloud Code Analysis Successful in 25s - Quality Gate passed Details

Figure 11: All Workflows Running Sucessfully upon a main push

3.3 System observability

Our system supplies a stream of logs (in standard output) that can be used to understand what happened whenever any operation happens. It's particularly helpful when it comes to understanding Websocket behavior, such as when authenticated connections are set, order queues are updated, and the state of queues are sent over to the user. Although these logs are streamed to standard output, we intended to have them stream to a file instead when deployed in production using a specific Logback configuration, so that we would not have to rely on Docker logs when our system is deployed, but due to time constraints, we didn't.

Furthermore, our backend containers (Spring Boot server and PostgreSQL database) have periodic health checks that detect when they stop behaving as expected, such as whenever our Spring Boot server stops answering HTTP requests, or our database is not able to receive connections.

4 Software testing

4.1 Overall strategy for testing

The development strategy followed a Test Driven Development (TDD – tests are written before the code that implements said feature) with a top-down approach: starting from the controller, then the service, then the repository (and ending with the integration tests). However, sometimes developers deviated from this approach when the task was too complex to write the tests before hand or when it was necessary to rapidly deliver part of a feature for the rest of the group to work properly on their feature (e.g. writing controller endpoint and associated services so that the frontend developer can work with real and properly formatted data).

Furthermore, a Behavior Driven Development (BDD) approach with cucumber and selenium was followed to perform functional tests interacting with the system from the user's perspective through an instance of a browser.

4.2 Functional testing/acceptance

As previously mentioned, there were written user-facing tests following a BDD approach with Cucumber and Selenium Webdriver. These tests are crafted to simulate interaction with the system from the user's perspective (a browser using a headless Firefox driver) and encompass the system's main user stories. All user stories were covered through these tests. Here are two example user stories, and their Gherkin definitions tested:

1. **Place order (**User-facing user story)

```
@order
Feature: Make an order on SpringCanteen
```

```gherkin
  @order_full_flow
  Scenario: User wants to make an order with a menu, and see his order number
    When I navigate to "http://localhost/order"
    And I select the menu number "3"
    And I select the Main Dish number "1"
    And I select the Drink number "1"
    And I click on "Confirm selection"
    And I click on "Customize and pay"
    And I fill in the NIF with "123456789"
    And I fill in the name on the card with "John Doe"
    And I fill in the card number with "1231231231231231"
    And I fill in the expiration date with "12/24"
    And I click on "Confirm order"
    Then I should see my order number as "7"
    And I should see total cost as "7.00"€
```

2. **Accept Payments** (Employee-facing user story)

```gherkin
@employee_desk_payments
Feature: Operations that a "Desk payments" employee can perform

  Background: Desk payments employee is Logged In
    Given I have a clean local storage
    And I navigate to the sign in page
    When I submit username and password
    And I click the sign in button
    Then I should be logged in

  @view_ready_to_pay_orders
  Scenario: Desk payments employee wants to see orders that are not yet paid
    When I navigate to the Desk payments page
    Then I should see the existing not yet paid orders

  @confirm_payment_succeeded
  Scenario: Desk payments employee wants to confirm an order was successfully paid
    When I navigate to the Desk payments page
    And I click the 'Request Payment' button for the first not yet paid order
    And I click the 'Confirm' button
    Then I should see the confirmation snackbar
    And the desk payments table should have one less order to be paid
```

These tests constitute a form of closed box testing as the system's internal logic is not known by the tester. The tester evaluates the system functionality through the user interface based on the input and output of his actions (simulating interaction with the browser with Selenium WebDriver). As previously mentioned, functional tests are integrated in the CI/CD pipeline to be automatically executed as outlined on the **Selenium Tests With Cucumber Workflow**.

4.3 Unit tests

Aiming to test each component independently unit tests were written with Junit while mocking the underlying components the first component depends on for its functioning with Mockito. RestAssured with mockMvc was also used to validate the RestAPI and to promote easier code readibility and maintainability. As previously mentioned, these tests were built following a top-down approach: starting with the controller with mock service tests, then the service with mock repository tests and ending with the repository tests. These kinds of tests have the objective of examining the design and internal structure and working of the software which classifies these has open box testing. Following TDD, based on the developer's initial understanding of the user story/feature the tests are written before the actual implementation. As previously mentioned, unit tests are integrated in the CI/CD pipeline to be automatically executed as outlined on the **Sonar Cloud and Xray Workflow**.

4.3.1 Controller With Mock Service Tests

Controller tests were done for the menu controller and the order controller. The first has just one endpoint responsible for retrieving all the existing menus. The latter has various endpoints with the following objectives: creating a new order, get all not paid orders (so that the desk payment employee can process the payment of orders that are not done at the kiosk) and another endpoint to actually pay those orders. Although these last two endpoints are secured so that only employees with the role DESK_PAYMENTS can access them, the authentication was ignored by mocking the jwtService and adding the following decorator:

```
@AutoConfigureMockMvc(addFilters = false)
```

Furthermore, the authentication controller (to retrieve the jwt tokens) and the OrderUpdatesController (controller that receives requests to change the status of the order – e.g. from idle to preparing) didn't get tested in this section.

Thus, from the implemented tests in the following image it is highlighted one unit test from the OrderControllerTest where the correct creation of an order with 2 menus is tested. The controller depends on the orderService which was mocked.

```
@Test
void whenCreateOrder_with2Menus_thenReturnCorrectResponse() {
  OrderMenu orderMenu1 = new OrderMenu(null, menu1, null);
  orderMenu1.setCalculatedPrice(11.0f);
  OrderMenu orderMenu2 = new OrderMenu(null, menu2, null);
  orderMenu2.setCalculatedPrice(5.0f);
  Optional<Order> returnOrder =
    Optional.of(new Order(1L, OrderStatus.IDLE, false, orderMenu1.getCalculatedPrice() +
orderMenu2.getCalculatedPrice(), false, "123456789", null, Set.of(orderMenu1, orderMenu2)));
  when(orderService.createOrder(any())).thenReturn(returnOrder);

  // order for 2 menus - 'lunch menu' and 'breakfast menu'
  //     lunch menu has main dish 'Potato chips with Beef and lettuce' (10€) with 0 potatoes and
  // drink 'Water' (1€)
  //     breakfast menu has main dish 'Pancakes' (3€) and drink 'Orange Juice' (2€)
```

```
String orderRequest =
    "{"
      + "   \"kioskId\": 1,"
      + "   \"isPaid\": false,"
      + "   \"isPriority\": false,"
      + "   \"nif\": \"123456789\","
      + "   \"orderMenus\": ["
      + "     {"
      + "       \"menuId\": 1,"
      + "       \"customization\": {"
      + "         \"customizedDrink\": {"
      + "           \"itemId\": 1"
      + "         },"
      + "         \"customizedMainDish\": {"
      + "           \"itemId\": 2,"
      + "           \"customizedIngredients\": ["
      + "             {"
      + "               \"ingredientId\": 3,"
      + "               \"quantity\": 0"
      + "             },"
      + "             {"
      + "               \"ingredientId\": 4,"
      + "               \"quantity\": 2"
      + "             },"
      + "             {"
      + "               \"ingredientId\": 55,"
      + "               \"quantity\": 4"
      + "             }"
      + "           ]"
      + "         }"
      + "       }"
      + "     },"
      + "     {"
      + "       \"menuId\": 2,"
      + "       \"customization\": {"
      + "         \"customizedDrink\": {"
      + "           \"itemId\": 3"
      + "         },"
      + "         \"customizedMainDish\": {"
      + "           \"itemId\": 3,"
      + "           \"customizedIngredients\": ["
      + "             {"
      + "               \"ingredientId\": 5,"
      + "               \"quantity\": 2"
      + "             },"
      + "             {"
      + "               \"ingredientId\": 6,"
      + "               \"quantity\": 2"
      + "             }"
      + "           ]"
      + "         }"
      + "       }"
      + "     }"
      + "   ]"
      + "}";

// expect 2 menus, first one with price (10€ + 1€), second one w ith price (3€ + 2€)
RestAssuredMockMvc
  .given()
```

```
            .mockMvc(mockMvc)
            .contentType(ContentType.JSON)
            .body(orderRequest)
        .when()
            .post("api/orders")
        .then()
            .statusCode(HttpStatus.SC_CREATED)
            .and()
            .body("orderMenus.size()", is(2))
            .and()
            .body("price", is(orderMenu1.getCalculatedPrice() + orderMenu2.getCalculatedPrice()))
            .and()
            .body("orderMenus.menu.name", containsInAnyOrder(menu1.getName(), menu2.getName()))
            .and()
            .body("orderMenus.menu.price", containsInAnyOrder(11.0f, 5.0f));
}
```

As we can see, in the Order Controller Tests the correct validation of the requests (through the corresponding DTO's) is one of the main aspects to be tested due to its complexity.

As previously mentioned, controller tests are integrated in the CI/CD pipeline to be automatically executed as outlined on the **Sonar Cloud and Xray Workflow**.

4.3.2 Service With Mock Repository Tests

Service tests were done for the following services:

- Authentication Service – service responsible for the processing of sign in and sign ups as well as the delivery of the respective token;
- Order Management Service – service responsible for managing an order through its different possible status: NOT_PAID, IDLE, PREPARING, READY and PICKED_UP. This service was implemented with a series of queues with maximum capacity for the status IDLE, PREPARING and READY. Furthermore, there are independent queues for priority queues totaling a maximum of 6 queues. Thus, is important to ensure the correct transferring of orders between queues and ensure that if an order can't be pushed into the new queue doesn't get removed from the queue that it is currently in;
- Order Service – service responsible for creating new orders and utilizing the Order Management Service to switch the order status. It will also enforce a validation of the old order status before calling the Order Management Service (i.e. if an employee is paying an order, then the only correct old order status is NOT_PAID, but if a request was made to change the order status after the order is paid then the correct old order status are IDLE, PREPARING, READY);
- Price Service – service responsible for calculating the price of a menu based on its customization (customization was passed as a json in the Order Controller Service previously). The calculated price shall equal to the sum of the base price of the drink and the base price of the main dish plus the price of additional ingredients for the main dish. The calculated price shall not be affected if the customer took ingredients from the main dish.
- Queue Notifier Service – service responsible for detecting if an employee subscribes to the websocket and, if that's the case, sending them in return all orders present in all queues. This response should only be sent if the role of the employee that subscribed is COOK or DESK_ORDERS and should return all orders in a different list depending in each queue it was in. These lists will contain DTO's of the orders to deliver only the necessary information;

As previously mentioned, in each service test the repositories, services or other components it depends on get mocked. In the following example a service test gets highlighted from OrderManagementServiceTest where we test transferring order between queues.

```
private static Stream<Arguments> providePriorityAndAllOrderStatusArgumentsForIdlePreparing() {
 return Stream.of(
    Arguments.of(true, OrderStatus.IDLE, OrderStatus.PREPARING),
    Arguments.of(false, OrderStatus.IDLE, OrderStatus.PREPARING),
    Arguments.of(true, OrderStatus.PREPARING, OrderStatus.READY),
    Arguments.of(false, OrderStatus.PREPARING, OrderStatus.READY));
}


@ParameterizedTest
@MethodSource("providePriorityAndAllOrderStatusArgumentsForIdlePreparing")
void
whenQueueNotFullAndManageOrderWithCertainStatus_thenOrderAdded_andNewStatusSet_andSentStatusUpd
ateThroughWS(
    boolean priority, OrderStatus oldOrderStatus, OrderStatus newOrderStatus) {
 boolean result;
 Queue<OrderEntry> oldOrdersQueue = getQueueFromPriorityAndStatus(priority, oldOrderStatus);
 Queue<OrderEntry> nextOrdersQueue = getQueueFromPriorityAndStatus(priority, newOrderStatus);
 order1.setOrderStatus(oldOrderStatus);
 order1.setPriority(priority);
 updatedOrder1.setOrderStatus(newOrderStatus);
 updatedOrder1.setPriority(priority);

 when(oldOrdersQueue.contains(any(OrderEntry.class))).thenReturn(true);
 when(nextOrdersQueue.offer(any(OrderEntry.class))).thenReturn(true);
 when(oldOrdersQueue.remove(any(OrderEntry.class))).thenReturn(true);
 when(orderRepository.save(any())).thenReturn(updatedOrder1);

 // act
 result = orderManagementService.manageOrder(order1);

 // should -> change order status and save it to DB; add to next orders queue according to
 // priority;
 //          remove from old orders queue;
 //          send message through websockets notifying status update
 OrderEntry oldOrderEntry = new OrderEntry(order1.getId(), oldOrderStatus);
 assertTrue(result);
 verify(oldOrdersQueue, times(1)).contains(oldOrderEntry);
 verify(nextOrdersQueue, times(1)).offer(argThat(
  orderEntry -> orderEntry.getId().equals(order1.getId()) && orderEntry.getOrderStatus() == newOrderStatus
 ));
 verify(oldOrdersQueue, times(1)).remove(oldOrderEntry);
 for (Queue<OrderEntry> unwantedQueue :
    getUnwantedQueues(List.of(oldOrdersQueue, nextOrdersQueue))) {
  verify(unwantedQueue, times(0)).contains(any(OrderEntry.class));
  verify(unwantedQueue, times(0)).offer(any(OrderEntry.class));
  verify(unwantedQueue, times(0)).remove(any(OrderEntry.class));
 }
 verify(orderRepository, times(1)).save(argThat(
  order -> order.getId().equals(order1.getId()) && order.getOrderStatus() == newOrderStatus
 ));
 verify(orderNotifierService, times(1)).sendOrderStatusUpdates(1L, newOrderStatus, priority);
}
```

It’s important to note the utilization of **@ParameterizedTest** and **@MethodSource** to test the transference of orders between IdleQueue and PreparingQueue or PreparingQueue and ReadyQueue for different order priorities as the inner logic of the function responsible for managing the orders between these queues is similar.

As previously mentioned, service tests are integrated in the CI/CD pipeline to be automatically executed as outlined on the **Sonar Cloud and Xray Workflow**.

4.3.3 Repository Tests

Regarding data repositories, all queries used were generated by Spring Data JPA based on the method names.

```
@Repository
public interface EmployeeRepository extends JpaRepository<Employee, Long> {
 Optional<Employee> findByEmail(String email);
}

@Repository
public interface OrderRepository extends JpaRepository<Order, Long> {
 List<Order> findByIsPaid(boolean isPaid);
}
```

Therefore, no extensive testing is needed apart from crafting a test to ensure it's configured properly. For this test, **@TestContainers** were used together with **Flyway** in order to create a temporary database container configured with preexisting data. There is also no need to mock any component as the repository doesn't depend on one.

```
@DataJpaTest
@Testcontainers
@AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE)
class OrderRepositoryTest {
 @Container
 public static PostgreSQLContainer container = new PostgreSQLContainer<>("postgres:latest")
  .withUsername("testname")
  .withPassword("testpassword")
  .withDatabaseName("sc_test");
 @DynamicPropertySource
 static void properties(DynamicPropertyRegistry registry) {
  registry.add("spring.datasource.url", container::getJdbcUrl);
  registry.add("spring.datasource.password", container::getPassword);
  registry.add("spring.datasource.username", container::getUsername);
 }
 @Autowired
 private OrderRepository orderRepository;
 private static final Logger logger = LoggerFactory.getLogger(IMenuService.class);

 // Just to verify if the repository is working (already implemented by Spring Data JPA)
 // no need to test findByIsPaid(false) as is all implemented by Spring Data JPA
 @Test
 void whenFindByNotPaid_thenReturnNotPaidOrders(){
  logger.info("Testing OrderRepository - findByIsPaid(true)");
  assertThat(orderRepository.findByIsPaid(true))
   .isNotNull()
   .hasSize(6);
 }
}
```

As previously mentioned, repository tests are integrated in the CI/CD pipeline to be automatically executed as outlined on the **Sonar Cloud and Xray Workflow**.

4.4 System and integration testing

With the goal of ensuring that all the system's components work together properly integration tests were implemented. We followed an open box/white box philosophy by testing our system knowing its inside logic. For this reason, we used spies (@SpyBean) in some of the integration tests, as with these beans, we can make assertions about the behavior of our modules, to guarantee that they do exactly what we expected.

By using **@SpringBootTest** the whole Application Context gets loaded guaranteeing a testing environment close to production. Furthermore, **@TestContainers** were used together with **Flyway** in order to create a temporary database container configured with preexisting data. RestAssured was also used to communicate with the RestAPI and to promote easier code readibility and maintainability.

Integration tests were crucial regarding testing the Authentication Controller, the receival of websocket messages, the detection of subscribe events on said websocket as well as all the other components that have been tested in all other testing layers that have been mentioned in this report. As previously mentioned, integration tests are integrated in the CI/CD pipeline to be automatically executed as outlined on the **Sonar Cloud and Xray Workflow**.

In the following examples we showcase integration tests for the:

- Authentication Controller – Validates the response of a successful sign up and verifies if it contains all intended data (including token and refresh token);

```
@Test
void givenValidCredentials_whenCreateEmployee_thenEmployeeCreatedAndTokenReturned() {
 String signUpRequest =
   "{"
     + "   \"username\": \"hello_world_test\","
     + "   \"email\": \"test@gmail.com\","
     + "   \"password\": \"person\","
     + "   \"role\": \"COOK\"}";

 given()
   .contentType(ContentType.JSON)
           .body(signUpRequest)
 .when()
   .post("api/auth/signup")
 .then()
   .statusCode(HttpStatus.SC_CREATED)
   .and()
   .body("username", is("hello_world_test"))
   .and()
   .body("email", is("test@gmail.com"))
   .and()
   .body("userRole", is("COOK"))
   .and()
   .body("$", not(hasKey("password")))
   .and()
   .body("$", hasKey("refreshToken"))
   .and()
   .body("$", hasKey("token"));
}
```

- Order Controller – This test tests the payment of an order for every employee type (COOK, DESK_ORDERS and DESK_PAYMENTS). The order should only be paid for if the role of the employee making the request is DESK_PAYMENTS. It also verifies the inner logic of the respective process;

```
private Stream<Arguments> provideAllArguments() {
 return employeeOrderAndUpdateRequestAndHeader.stream();
}

@ParameterizedTest
@MethodSource("provideAllArguments")
void whenPayNotPaidOrder_WithStatusNotPaid_thenReturn204_OnlyForDeskPayments_elseReturn403(
 Order testOrder, OrderUpdateRequestDTO orderUpdateRequest, StompHeaders userHandshakeHeaders,
EmployeeRole employeeRole
) {
 // setup
 Order newOrder = new Order(

OrderStatus.NOT_PAID,testOrder.isPaid(),testOrder.isPriority(),testOrder.getNif(),testOrder.getKioskTerminal()
 );
 orderRepository.save(newOrder);
 logger.info("newOrder: {}; orderUpdateRequest: {}; employeeRole: {}; userHandshakeHeaders: {};",
  newOrder.getId(), orderUpdateRequest, employeeRole, userHandshakeHeaders);

 int statusCode = RestAssured
  .given()
   .contentType(ContentType.JSON)
   .header("Authorization", userHandshakeHeaders.get("Authorization").get(0))
  .when()
   .put(String.format("api/orders/%d", newOrder.getId()))
  .then()
   .statusCode(employeeRole == EmployeeRole.DESK_PAYMENTS ? HttpStatus.SC_NO_CONTENT :
HttpStatus.SC_FORBIDDEN)
   .extract().statusCode();

 verify(orderServiceSpy, times(0)).changePaidOrderToNextOrderStatus(orderUpdateRequest.getOrderId());
 verify(orderNotifierServiceSpy, times(0)).sendOrderStatusUpdates(anyLong(), any(), anyBoolean());
 if (statusCode == HttpStatus.SC_FORBIDDEN) {
  verify(orderServiceSpy, times(0)).changeNotPaidOrderToNextOrderStatus(newOrder.getId());
  verify(orderManagementServiceSpy, times(0)).manageOrder(any());
  verify(orderManagementServiceSpy, times(0)).manageNotPaidOrder(any());
  verify(orderNotifierServiceSpy, times(0)).sendNewOrder(any());
 } else {
  verify(orderServiceSpy, times(1)).changeNotPaidOrderToNextOrderStatus(newOrder.getId());
  verify(orderManagementServiceSpy, times(1)).manageOrder(
   argThat((Order order) -> order.getId().equals(newOrder.getId()))
  );
  verify(orderManagementServiceSpy, times(1)).manageNotPaidOrder(
   argThat((Order order) -> order.getId().equals(newOrder.getId()))
  );
  verify(orderNotifierServiceSpy, times(1)).sendNewOrder(
   argThat((Order order) -> order.getId().equals(newOrder.getId()) && order.getOrderStatus() ==
OrderStatus.IDLE)
  );
 }
```

- Order Updates Controller – This test verifies if the request to change the status of an order from ready to picked up, which is sent from a websocket connection, passes for all the employees with different roles. Emphasize the use of awaitility to wait until the message is received and the response is sent. Ordering was used in these tests as in this file we tested the whole flow of status of an order from IDLE to PICKED_UP.

```java
private Stream<Arguments> provideAllTokenHeaders() {
 return employeeOrderAndUpdateRequestAndToken.stream();
}

@ParameterizedTest
@MethodSource("provideAllTokenHeaders")
@org.junit.jupiter.api.Order(3)
void whenReceiveUpdateOrder_FromREADY_PICKED_UP_thenRemoveFromQueue(
   Order testOrder, OrderUpdateRequestDTO orderUpdateRequest, StompHeaders userHandshakeHeaders
) throws ExecutionException, InterruptedException, TimeoutException {
 // setup
 logger.info("testOrder: {}; orderUpdateRequest: {}; userHandshakeHeaders: {}",
   testOrder, orderUpdateRequest, userHandshakeHeaders);
 stompSession =
    connectAsyncWithHeaders(websocketURL, webSocketStompClient, userHandshakeHeaders);

 // setup (have to change the status of the object because the reference isn't the same)
 testOrder.setOrderStatus(OrderStatus.READY);

 // act
 stompSession.send("/app/order_updates", orderUpdateRequest);

 // wait until message received and update message is sent
 Awaitility.await()
    .atMost(2, TimeUnit.SECONDS)
    .untilAsserted(
       () -> {
        verify(orderUpdatesControllerSpy, times(1)).receiveOrderUpdates(any());
        verify(orderServiceSpy, times(1)).changePaidOrderToNextOrderStatus(testOrder.getId());
        verify(orderManagementServiceSpy, times(1))
           .manageOrder(argThat((Order order) -> order.getId().equals(testOrder.getId())));
        verify(orderManagementServiceSpy, times(1))
          .manageReadyOrder(argThat((Order order) -> order.getId().equals(testOrder.getId())));
        verify(orderNotifierServiceSpy, times(1))
           .sendOrderStatusUpdates(
              testOrder.getId(), OrderStatus.PICKED_UP, testOrder.isPriority());
       });

 // assert
 verify(orderNotifierServiceSpy, times(0)).sendNewOrder(any());
 verify(orderServiceSpy, times(0)).changeNotPaidOrderToNextOrderStatus(any());
}
```