Coviran's Online Store and Delivery Platform

Quality Assurance Manual

Universidade de Aveiro

31-05-2022

Written by

Alexandre Serras - 97505

Gonçalo Leal - 98008

João Reis - 98474

Pedro Duarte - 97673

Table of Contents

Abstract

The present report was written to help new members coming to the project to learn the QA practices defined by the team. We will provide vital information about the roles on the project, tools for backlog and version management, coding style and testing, git branching and CI/CD pipelines.

This project has two different applications: an online store for Coviran and a delivery management platform. Both the applications are connected, since all the online orders made through Coviran's online store are delivered through the developed delivery platform.

In order to allow Coviran's online store to request deliveries for new orders both APIs have to communicate, so both have to follow a predefined standard.

1. Project Management

1.1. Team and Roles

- Team Leader: Pedro Duarte

The Team Leader is the person responsible for monitoring, organizing and directing the team. This is the member with more expertise and should be able to perform all the technical skills required in the project.

- Product Owner: João Reis

The product owner is the link between the team and the client. He writes the user stories in a clear way, orders them in the backlog by priority and ensures the team understands items in the backlog.

- DevOps Master: Gonçalo Leal

The DevOps is responsible for the application and infrastructure planning, automatic testing and deployment. He is the one who implements the CI/CD pipelines, automates these flows and deals with incidents.

- QA Engineer: Alexandre Serras

The QA Engineer develops the testing procedures based on the requirements. Since we are using Test Driven Development, the team follows the recommendations of the QA Engineer when writing the business logic and backEnd code.

1.2. Agile backlog management and work assignment

Our group is using two different tools for backlog management and work assignment: ClickUp and GitHub Actions. We chose these tools because the group was already used to working with them.

For the backlog management we are using ClickUp. Our Team Leader creates the tasks based on the user stories and distributes them among the team members. He also attributes the points to those tasks based on their difficulty. The team uses this tool to update tasks status and to track the time spent in each task.

For code management and task completion we use GitHub. When we complete a task the member assigned to it changes its state to “Validation” and opens a pull request if that pull request is accepted the task is considered completed.

To communicate with each other we use a discord server where we have channels dedicated to both ClickUp and GitHub. There we receive messages when someone changes the status of a task and everything that occurs in our GitHub organization: pushes, pull requests and tests results.

2. Code Quality Management

2.1. Guidelines for contributors

2.1.1. Development Guidelines

To have a better understanding of what is going on on the code we decided to create our own exceptions. This way, when an exception is thrown we know what caused it.

Exception Name	Meaning
BadLocationException	Invalid latitude or longitude
BadPhoneNumberException	Invalid phone number
UserAlreadyAssignedException	Email already registered

Another rule we are following is related to test's title:
void testWhen<Action><ClassName>ThenReturn<ReturnObject>.
Where we should only change what is class related, everything else remains the same.

2.1.2. Git Guidelines

Our team has agreed in following some rules related to Git:

- branches' names have to follow the "feature/<description>" pattern
- developers should not open pull requests to main, instead they should open them to the branch dev
- the default branch is dev
- all pull request should have a revisor

3. Continuous Delivery Pipeline (CI/CD)

3.1. Development Workflow

3.1.1. Github Flow

Every developer must create a new branch when developing a new feature, giving it a name that describes its purpose. After finishing the task the developer opens a pull request and asks one of his teammates to review it. The choice of the reviewer should take into account the reviewer's knowledge about the task.

3.1.2. Definition of Done

We consider that a certain task is completed when all the tests and code for that task have been written, the tests are successful, the code builds without errors, the SonarQube check is successful and the code is reviewed and approved.

3.2. CI/CD pipeline and tools

Our CI/CD pipeline is managed and hosted on Github using Github actions and runners configured in our server.

3.2.1. Continuous Integration

To assure the continuous integration of the code and its quality we have tests being run on every push, everytime there is a pull request to dev or to main we run SonarQube and a linter. On every push to dev we deploy the app on a test environment where we run Selenium tests configured using Cucumber.

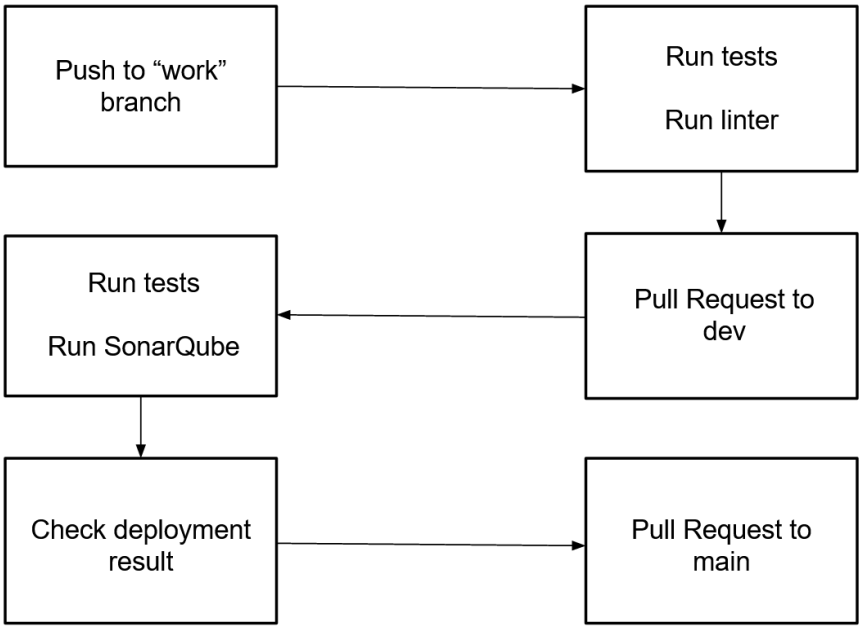

Fig 1: Continuous Integration - GitHub Pipeline

3.2.2. Continuous Deployment

All the pieces of our project are dockerized which makes the deployment process much easier. Whenever there is a push to the dev or main branches, a Github Agent hosted in our server pulls the code and deploys it through a docker-compose file.
Our organization has 5 different code repositories, 3 with angular web apps and 2 with Spring Boot APIs. Every repository has its own agent configured in the server.

Since we have two different deployment flows, one for testing and one for production and both are handled by the same agent, the code is copied to a /test_env or /deploy_env folder when pulled by the agent. This allows us to consult the code that is running on the server whenever there is a problem.

Because several containers are running on the same machine each environment has its own docker network. This creates a more realistic environment considering that in a real delivery system the stores would not have access to the network of the delivery system and vice versa.

To keep track of the state of our server we have included monitoring mechanisms on our continuous delivery pipeline. This monitoring is done using a Prometheus Node Exporter which stores metrics on a Prometheus Server which is a Time Series Database. The metrics collected by the Node Exporter are displayed in a Grafana dashboard. The display is available at http://deti-tqs-12.ua.pt:3000/d/rYdddlPWk/node-exporter-full?orgId=1&refresh=1m.

Fig 2: Grafana Dashboard

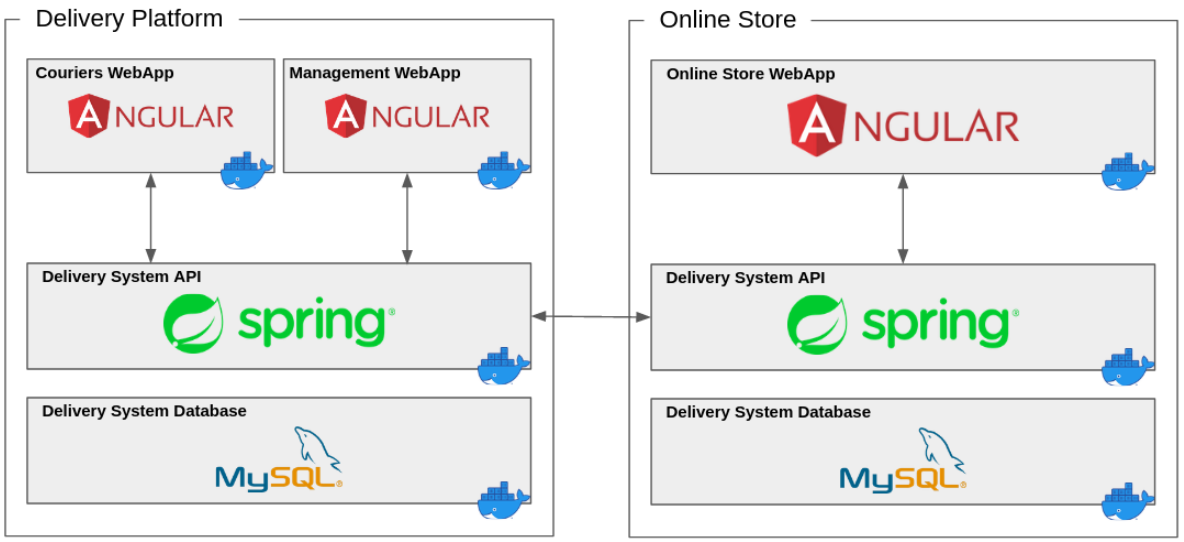

Fig 3: Deployment Diagram

4. Software Testing

4.1. Overall strategy for testing

We try to always develop our code following TDD methodology, although it may seem difficult sometimes.

To assure that we are following rigorously what was defined in the user stories we convert them into Cucumber features and we use them to create tests.

To test our repositories we follow the Test Container methodology and Rest Assurance. For Selenium we use Page Object Pattern.

4.2. Functional Testing/Acceptance

Our functional tests are made using Selenium and the actions performed are chosen based on the resulting features of Cucumber. To prevent errors related to browsers or OS we run Selenium tests inside TestContainers.

To assure that our RestApi runs as expected we will test its functionalities with a database with a lot of data, very few data and no data at all. We expect that the API behaves as expected in each scenario.

4.3. Unit Tests

Unit tests are those where we are going to test isolated parts of the system and, according to TDD methodology, so we only do repositories when the models have been tested and well implemented, and so on.

These tests will be implemented on models, repositories, services and controllers. In the case of models, as they are entities that do not depend on repositories, services or controllers, they are the first ones to be tested if class' functionalities are running as expected.

Moving to repositories we use the @DataJpaTest annotation and we run the database in MySQL TestContainers to keep the test environment

similar to the production one. All the functionalities implemented in the repositories were tested along with the default ones like save, findAll, findById, etc.
To test services we used Mockito to mock repositories so we can test only the services isolated from the remaining code. We test possible error scenarios to have a bigger test range.

Unit tests for controllers use mocked services where these mocks are made using RestAssuredMockMvc, allowing us to test the controllers isolated from everything else. Errors and unusual scenarios are also tested.

4.4. System and Integration Testing

Systems integration tests test the whole system's life cycle. A request to a specific URL is made and we track the system's behavior for that request. The request should be mapped to a Controller and then everything under it is tracked.

Every endpoint has a TemplateIT to assure that everything works as expected and there are no surprises while the application is running. For that, we will also test errors and rare situations to cover the most of the situations that may occur in production.