TQS: Relatório de Controlo de Qualidade

#### Pedro Figueiredo [97487], Fábio Martins [98119], Renato Dias [98380], José Trigo [98597]

v2022-06-15

1. [1. Bookmarks do Projeto 1](#_Toc794996074)
2. [2. Gestão do Projeto 2](#_Toc678379813)
   1. [2.1. Equipa e Papéis 2](#_Toc1029455854)
   2. [2.2. Gestão do Backlog e Atribuição de Trabalho 3](#_Toc1408377937)
3. [3. Gestão e Qualidade do Código 5](#_Toc2118784319)
   1. [3.1. Guia de Contribuição (Coding style) 5](#_Toc1245381723)
   2. [3.2. Métricas de Qualidade de Código 5](#_Toc8641762)
4. [4. Pipeline de Entrega Continua (CI/CD) 7](#_Toc1642474629)
   1. [4.1. Development workflow 7](#_Toc827103597)
      1. [4.2. Ferramentas e Pipelines de CI/CD 7](#_Toc2010478366)
   2. [5.1. Estratégia de Teste 1](#_Toc2073032460)2
      1. [5.2. Testes Funcionais/Aceitação 1](#_Toc357068230)2
      2. [5.3. Testes Unitários 1](#_Toc685246538)3

# Bookmarks do Projeto

Sistematização dos links para os recursos desenvolvidos no projecto:

* + Acesso ao projecto de código, bem como Atlassian Jira e GitHub Actions:

a. Repositório de Grupo

* + Ambiente de produção:

1. Aplicação Web
2. REST API
   * Ambiente SQA :
3. Análise estática
   * Coordenação da equipa:

a. Discord

# Gestão do Projeto

## Equipa e Papéis

Para facilitar a divisão de responsabilidades dentro da equipa, decidimos atribuir cargos específicos a cada elemento:

|  |  |  |
| --- | --- | --- |
| **Cargo** | **Descrição** | **Membro/s da equipa** |
| Product Owner | O Product Owner representa os interesses dos stakeholders. Tem um conhecimento detalhado sobre o produto e o domínio onde se insere e por isso os restantes elementos da equipa irão falar com ele para clarificar dúvidas sobre futuras funcionalidades do produto. Tem um papel ativo em aceitar os novos incrementos desenvolvidos para a solução final. | Pedro Figueiredo |
| Quality Engineer | Responsável, em articulação com outras funções, por promover as práticas de garantia de qualidade e colocar em prática instrumentos para medir a qualidade da implementação. | Renato Dias |
| DevOps Master | O DevOps master é responsável pela infraestrutura de desenvolvimento e produção, aplicando as configurações necessárias à mesma. Gere a configuração e a preparação das máquinas de deployment, repositório git, infraestrutura da cloud, operações da base de dados, entre outras responsabilidades. | José Trigo |
| Team Manager | O Team Manager assegura que existe uma divisão justa de tarefas e que o plano de desenvolvimento é seguido por todos | Fábio Martins |

|  |  |  |
| --- | --- | --- |
|  | todos os elementos da equipa. Promove um bom ambiente dentro da equipa e assegura que os requisitos do projeto são entregues dentro dos prazos estabelecidos. |  |
| Developer | O developer desenvolve a aplicação/projeto consoante o plano definido pelo team manager. | Pedro Figueiredo  Fábio Martins  Renato Dias  José Trigo |

*Tabela 1: Distribuição de Papéis de Trabalho dentro da Equipa*

## Gestão do Backlog e Atribuição de Trabalho

Tendo em conta que o desenvolvimento é orientado a *user stories*, existem diversas fases na criação e gestão do *backlog*:

* + 1. **Adicionar as User Stories:** O *Team Manager*, tendo em conta as orientações do cliente, cria diversas *user stories* e adiciona-as ao *backlog* do projeto. Inicialmente, estas não são atribuídas a nenhum elemento da equipa, nem têm uma *timeline* definida;
    2. **Priorizar as User Stories:** O *Project Manager* estabelece a prioridade de cada *story*, através de um sistema de pontos;
    3. **Estimar o tempo de desenvolvimento de cada User Story:** Numa reunião entre todos os elementos da equipa, estima-se o tempo que demorará a desenvolver cada *story* e complementa-se esta com informação extra (por exemplo: critérios de aceitação/qualidade);
    4. **Desenvolver cada User Story:** O desenvolvimento das *stories* está pronto para ser iniciado. Assim, e tendo em conta os prazos apertados de desenvolvimento, cada *story* é atribuída a um *developer*. Após isto, os *developers* começam a desenvolver as *user stories* daquela interação, passando estas para o estado “*In Progress*”;
    5. **Entregar as User Stories:** Após o desenvolvimento de uma *story*, acompanhado dos respetivos testes, o *developer* entrega o seu trabalho. Isto vem acompanhado de um *merge request*, uma vez que cada *feature* será desenvolvida num *branch* diferente;
    6. **Testar as User Stories:** Após o código entregue por um developer ser validado pelo ambiente de CI/CD, este é *deployed* num ambiente de teste. Após isto a *story* passará para o estado “*In Review*”;
    7. **Aceitar ou Recusar cada User Story:** O *Project Manager*, em conjunto com o cliente e com outros *developers*, *testers* e *designers*, verifica se os critérios de aceitação definidos inicialmente se verificam e, caso todas as entidades envolvidas concordem, a *user story* é aprovada, saindo do *backlog* para o painel “*Closed*”. Caso contrário, esta *story* terá de ser refeita.

Neste projeto, a ferramenta de gestão de *backlog* que será utilizada será o Atlassian Jira que, levando a sua utilização ao máximo e tentando centralizar todo o desenvolvimento e processos numa só ferramenta, vai coexistir também como *Issue Tracking System*.

Uma imagem com texto

Descrição gerada automaticamente

*Figura 1: Jira - Gestão de Backlog + Issue Tracking System*

# Gestão e Qualidade do Código

## Guia de Contribuição (Coding style)

Com o intuito de aplicar as melhores convenções possíveis referentes a escrita de código, iremos utilizar dois guias de estilo de código: um para a **linguagem Java** e outro para a **biblioteca ReactJS**. Em ambos os guias, estão definidas algumas regras e convenções da respetiva linguagem, bem como apresentados bons e maus exemplos práticos de utilização.

Relativamente à linguagem Java, iremos utilizar o seguinte guia de estilo de código, desenvolvido pela equipa do Android:<https://source.android.com/setup/contribute/code-style>.

Quanto à biblioteca ReactJS, o guia de estilo de código escolhido foi o seguinte:

<https://github.com/airbnb/javascript/tree/master/react>.

## Métricas de Qualidade de Código

A análise estática de código é um método para fazer o *debugging* do código, antes do programa ser corrido. Durante esta análise, compara-se o código escrito com regras gerais de escrita de código.

Para isto, recorremos ao SonarQube, mais especificamente ao SonarCloud e SonarLint - uma plataforma *open source* para efetuar reviews automáticas e análise estática de código. Esta ferramenta, não só detecta código com *bugs*, como também *bad smells* e vulnerabilidades de segurança. No caso do SonarLint, este permite a deteção de *bugs* e o seguimento de boas práticas em tempo de escrita de código com a sua extensão para o IDE *VS Code*, que foi comum a todos os elementos da equipa.

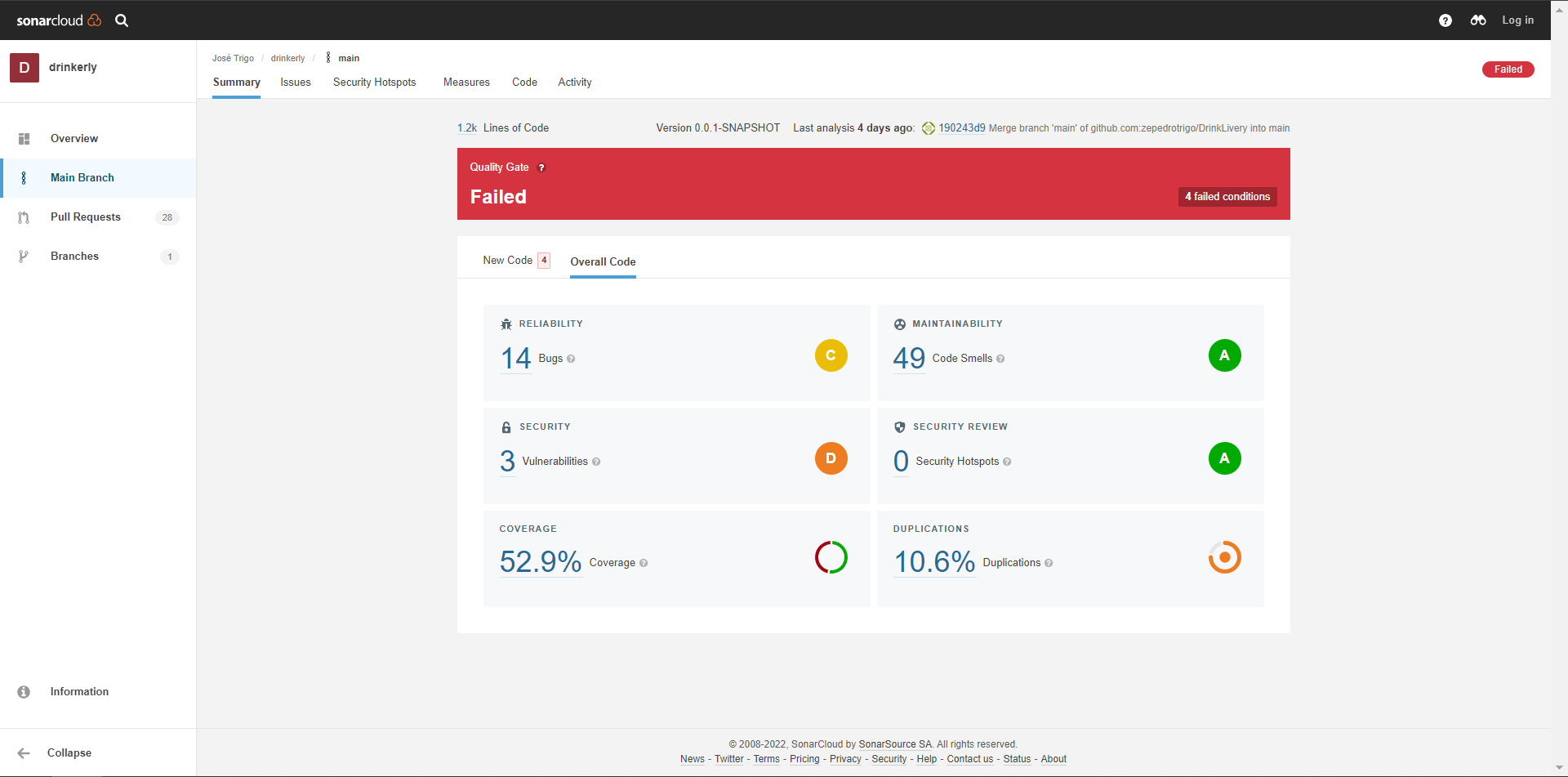
Para além disto, o SonarQube permite uma análise de código contínua, pelo que, para além de nos mostrar o estado da aplicação que estamos a desenvolver, também nos mostra quais os problemas introduzidos por cada incremento produzido.

Podemos, também, definir *quality gates* para o nosso projeto. Estas não são mais que meras métricas para a integração de novo código no projeto em desenvolvimento. Se uma nova submissão de código não passar numa *quality gate*, este código terá de ser refeito até que consiga atingir os padrões definidos. Ainda neste tópico, podemos definir *quality gates* relativos a:

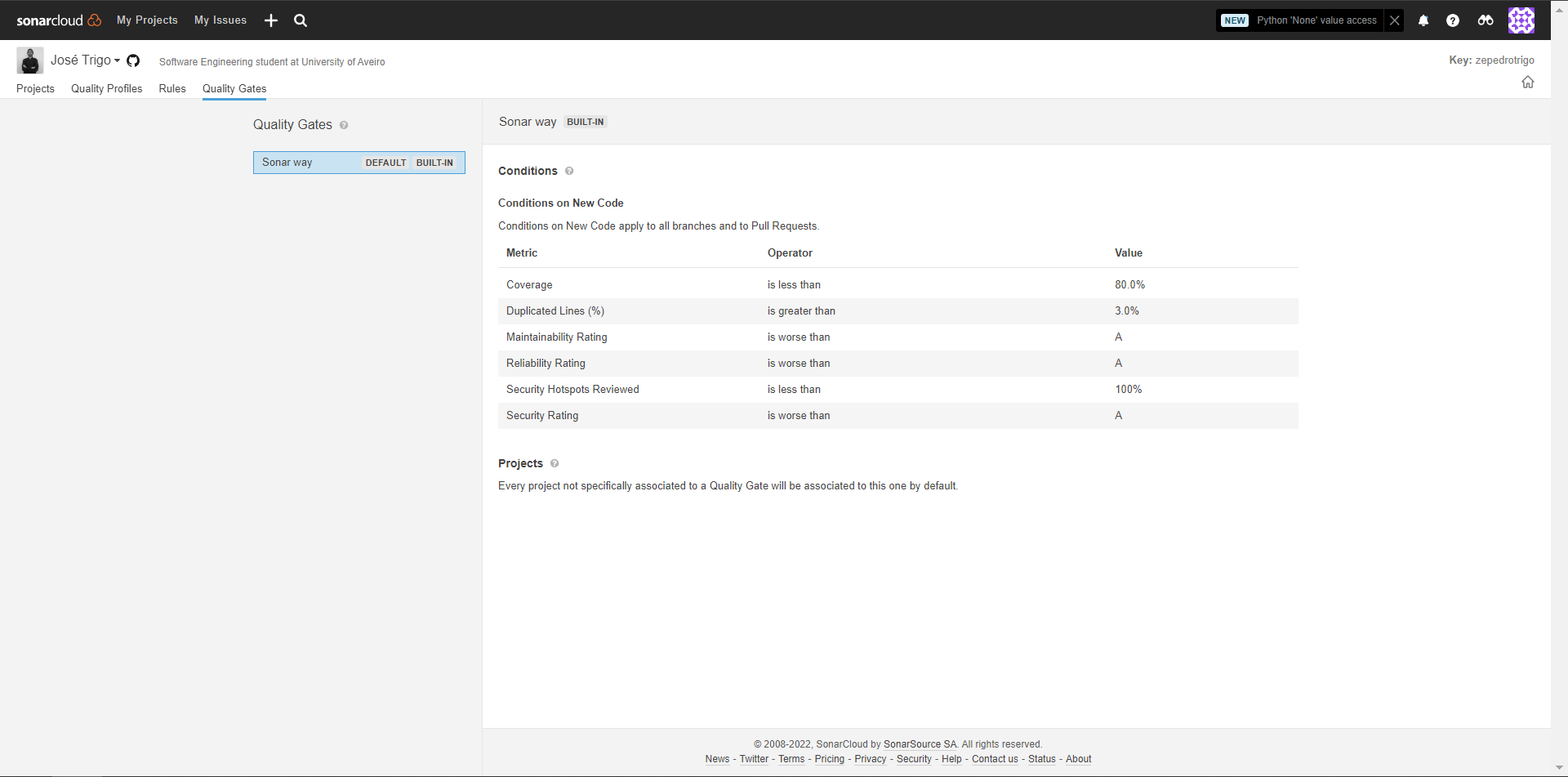
* + - Duplicação de código
    - Manutenção necessário para o código escrito
    - Fiabilidade
    - Segurança

A plataforma escolhida apresenta, também, um grande vantagem no que toca a integração contínua. Uma vez que o SonarQube é facilmente integrado com o GitHub CI/CD, é bastante simples automatizar esta análise estática de código, de forma a que a mesma seja executada de forma contínua. Por fim, tendo em

conta que as *quality gates default* do SonarQube nos pareceram adequadas, decidimos manter estas ao invés de redefini-las para uma configuração própria. No final estas acabaram por não ser alcançadas por falta de tempo.



*Figura 3: Resultados da Análise do SonarCloud*



*Figura 4: Quality Gates Utilizados na Análise do SonarCloud*

# Pipeline de Entrega Continua (CI/CD)

## Development workflow

### SCM Workflow

Relativamente ao *SCM Workflow*, iremos usar ***Git Feature Branch Workflow*** do *Bitbucket*. Este transmite que:

* + - * Sempre que um *developer* queira desenvolver uma nova *feature/user story*, este deverá fazer o seu desenvolvimento numa nova *branch*, especificamente criada para o efeito. Esta deve ser denominada de forma a que permita rapidamente identificar qual o *ambient/issue* a ser tratada, utilizando o padrão ***ambient/<feature\_name>***.
      * Na nova *branch* criada, o *developer* edita e dá *commits* das suas implementações.
      * Quando um *developer* acabar de desenvolver as *features* associadas à *branch* que criou, este terá de criar um *pull request* para que a sua *branch* seja unida com a *main branch*. Desta forma, os outros membros da equipa irão receber uma notificação referente a esta situação.
      * Os outros *developers* da equipa dão *feedback* sobre o código a ser inserido na *master branch*. Após isto, este código poderá ter de ser reformulado. Assim que o código for aprovado pelos *reviewers*, a *branch* onde está a nova *feature* será *merged* com a *master branch*.
      * Por fim, define-se também (como acrescento ao *flow* em que nos baseámos) que todos os *pull requests* têm de ser aprovados por, pelo menos, um *reviewer* que não seja a próprio que submeteu o *pull request*.

Devido ao ambiente de colaboração presencial, estas práticas não são totalmente refletidas no repositório.

### Revisão de Código

De forma a melhorar a qualidade geral do código produzido, é necessário que este seja revisto por diversos *developers*, de forma a que se encontrarem erros e potenciais situações de riscos. Para que este processo decorra eficazmente, definimos um conjunto de princípios a seguir:

* + - * Caso alguma porção de código não seja percetível, o *developer* que o escreveu deverá reformular/explicar esta secção;
      * Não se pode assumir que o código submetido funciona. É necessário fazer *build* do projeto e correr todos os testes associados ao mesmo.
      * É necessário rever, também, o código “temporário”, uma vez que este se poderá tornar em código para produção;
      * Deve ser realizada uma *review*, quer aos testes, quer aos *build files* associados a código que está a ser revisto;
      * As *reviews* de código devem ter em atenção se o *code style* está de acordo com o definido no início do projeto;
      * A arquitetura de uma solução poderá, também esta, ser revista;
      * Os comentários de uma *code review* devem ser críticas construtivas;
      * Ao fazer uma *code review*, as sugestões devem ser feitas de acordo com a seguinte prioridade:
        + Melhorias a nível funcional;
        + Alterações para manter o código *clean* e fácil de manter;
        + Por fim, sugestões para otimizar o código.
      * Acompanhar o estado de uma *code review* é tão importante como fazer a *code review*, pelo que cada *developer* deverá fazer o *follow up* das *reviews* que fez.

Relativamente ao processo de *code review* implementado neste projeto, este tem como suporte as ferramentas disponibilizadas pelo GitHub Actions. Sempre que é feito um *pull request*, inicia-se um processo de *code review* do código submetido.

Por fim, para esclarecimento futuro, uma *user story* é considerada como terminada após completar todo este processo de: Pull Request -> Build e Execução de Testes Automáticos -> Code Review e Validação de Código.

## Ferramentas e Pipelines de CI/CD

Um dos objetivos deste projeto, é manter ambientes de integração e *delivery* contínuos e, para isso, usámos as ferramentas do GitHub Actions.

Uma imagem com texto, captura de ecrã, monitor, preto

Descrição gerada automaticamente

*Figura 5: GitHub Actions utilizados*

Como podemos ver na figura 5, existem duas ações a serem feitas após um merge de um pull request, que na verdade são repetições dos testes executados no momento em que o mesmo é criado:

* **build\_services:** Executa os testes desenvolvidos na parte do serviço de Management, assim como o build do mesmo.
* **build\_app:** Executa os testes desenvolvidos para o backend da aplicação, assim como o build da mesma. Esta falhou ao longo do desenvolvimento do projeto devido à necessidade de termos o *frontend* a correr para os testes ao mesmo serem executados.

Não estão listados na figura acima os testes estáticos ao código, uma vez que estes foram feitos no SonarCloud, não sendo esta uma ação sobre o repositório em si. Para além disso, houve uma tentativa de implementação de uma pipeline para fazer o deploy do projeto em um container Docker, no entanto, devido a problemas técnicos, este não foi implementado. Apesar de termos tudo o que estava estipulado como necessário para conseguir executar essa ação, a mesma falhava na tentativa de login e, devido à falta de documentação acerca do erro, esta foi descartada.

### Compose e CI

Este repositório é utilizado unicamente para o objetivo de executar o *deploy* para a nossa VM por SSH. Para isso existem dois ficheiros importantes: docker-compose.yml e sonarcloud-yml.

Na figura 12, podemos ver o docker compose utilizado para o *deploy*. Neste ficheiro temos 3 serviços:

* Base de dados MySQL, usada para persistência.
* Web Application relativa ao utilizador.
* Web Application relativa ao Management.
* Web Application relativa aos estafetas.

Uma imagem com texto

Descrição gerada automaticamenteUma imagem com texto

Descrição gerada automaticamente

*Figura 12: Docker Compose utilizado*

Uma imagem com texto

Descrição gerada automaticamente

Uma imagem com texto

Descrição gerada automaticamente

*Figura 13: Pipeline de CI*

# Testes de Software

## Estratégia de Teste

Esta secção é destinada a documentar a estratégia de testes usada no projeto. Desde o início do desenvolvimento, que o objetivo foi sempre aplicar TDD (*Test-Driven Development*), de forma a que se crie apenas o essencial para implementar os testes e, a partir daí, desenvolver as funcionalidades.

Tecnologicamente, usámos JUnit 5 e Mockito (quando necessário) para os testes da REST API, Cucumber e Selenium para os testes funcionais na Web App.

## Testes Funcionais/Aceitação

Este tipo de testes são, geralmente, realizados em cooperação com o cliente (interface), uma vez que são os testes que irão validar se a aplicação está de acordo com o que o cliente pretende. Para este tipo de teste, usamos casos de teste que cobrem os cenários típicos sob os quais esperamos que o software seja usado. Estes testes devem ser conduzidos num ambiente de "produção", e num hardware que seja igual ou próximo do que cliente usará, o que não foi possível fazer devido a não termos um mecanismo de CD, pelo qual o produto utilizado foi o de desenvolvimento.

Uma imagem com texto

Descrição gerada automaticamente

*Figura 15: Cenário para o Login e Checkout*

Uma imagem com texto

Descrição gerada automaticamente

*Figura 16: Teste Funcional ao Comparador de Casas*

Como se vê nas figuras 15 e 16, utilizando *Cucumber* e *Selenium* facilmente conseguimos implementar um sistema de Casos de Utilização e de Cenários de forma a testar o comportamento funcional da nossa aplicação.

## Testes Unitários

De forma a garantir que todas as funcionalidades, por mais minimalista que sejam, não perturbem o bom funcionamento da aplicação e funcionam como pretendido, são expostas a testes unitários. Cada teste

unitário executa o código fonte de uma e apenas uma funcionalidade, daí ser unitário. Tem como metodologia *white-box testing*, ou seja, tem como objetivo testar a estrutura interna de uma aplicação.

Uma imagem com texto

Descrição gerada automaticamente

*Figura 17: DrinkLivery - Testes Unitários – Criar um Produto (Vinho)*