deti universidade de aveiro departamento de eletrónica, telecomunicações e informática

TQS: Quality Assurance manual

Nuno Afonso Fontes Vieira [107283]
v2026-01-28

Contents

1 Gestão da qualidade de código

1.1 Políticas para o uso de gen AI

Durante o desenvolvimento deste projeto, a AI generativa foi usada como uma ferramenta de auxílio e que serviu, principalmente, para acelerar o processo de desenvolvimento e melhorar a produtividade, sendo que o tempo que tivemos para desenvolver foi curto.
A face do desenvolvimento onde esteve mais presente esta ajuda foi no desenvolvimento de testes e, principalmente, testes unitários. Como estes são os testes que devem existir em maior volume, a AI foi bastante útil para escrever vários destes, tanto com uma metodologia TDD (primeiro escrever os testes) como numa metodologia de escrever testes após ter implementado a lógica de negócio subjacente.

1.2 Guias para os contribuintes

Coding style

O projeto segue princípios de **Clean Code** e uma separação clara de responsabilidades, dividindo o código entre módulos para Controller (API e comunicação), Service (lógica de negócio), Repository (ORM) e Entity (entidades do modelo).

O código está escrito de forma compacta, sem que haja métodos demasiado extensos ou com demasiadas responsabilidades.

Code reviewing

Como o trabalho foi desenvolvido a solo, não existiu uma componente de code reviewing entre pares, mas usei também AI para ir fazendo debugging de problemas ou confirmar se o código estava bem escrito. No entanto, continuou a haver feature branching e pull requests a ser aprovados por outrem (simulado com uma segunda conta).

1.3 Métricas de qualidade de código e dashboards

As métricas de qualidade de código e "code smells" foram introduzidas através do SonarQube (Cloud). Para este efeito, fiz a integração com o repositório GitHub e defini um quality gate de 50% de cobertura de código para pull requests para a main.

2 Pipeline de Continuous Delivery (CI/CD)

2.1 Workflow de desenvolvimento

Coding workflow

O desenvolvimento do projeto segue um workflow GitFlow estruturado que visa garantir a qualidade do código, a rastreabilidade das funcionalidades e a integração contínua das alterações. O ponto de partida para o desenvolvimento é o backlog de user stories, gerido na ferramenta de gestão de projeto Jira, onde cada funcionalidade é descrita em termos de requisitos funcionais e critérios de aceitação. O workflow definido foi o seguinte:

a) Um developer que pretende trabalhar numa nova user story, deve primeiro criar uma nova feature branch para essa story a partir da branch dev;
b) De seguida, implementa a feature e quando tiver terminado, vai fazer um pull request para a branch dev;
c) Este pull request deve ser aprovado por um reviewer para ser integrado na branch;
d) Depois de serem implementadas um conjunto de features que tematicamente façam sentido para alcançar algum objetivo de produto num todo (por exemplo, um MVP), faz-se um pull request da dev para a main, que deve também ser aprovado por um reviewer.

Definition of done

Uma user story é considerada concluída ("done") quando verifica os cenários e critérios de aceitação na sua definição. Para além disso, e apesar de nem sempre ter sido respeitado, deve também ter um conjunto de testes que validem o seu funcionamento, quer sejam unitários, de integração ou de aceitação.

2.2 Pipeline CI/CD e ferramentas

O projeto integra um pipeline de CI/CD com o objetivo de automatizar a validação da qualidade do código e garantir que cada incremento do sistema é integrado de forma segura e consistente. O pipeline é executado automaticamente sempre que ocorrem alterações relevantes no repositório, nomeadamente em pushes diretos ou na abertura de pull requests para dev ou main.
Usei o GitHub actions para criar o seguinte workflow:

e) Build do projeto;
f) Execução de testes unitários;
g) Execução de testes de integração;
h) Execução de testes de aceitação;
i) Execução de testes de carga;
j) Geração de relatório de cobertura do JaCoCo e análise estática com SonarQube

2.3 Observabilidade do sistema

A aplicação inclui mecanismos de logging, utilizados para registar eventos relevantes durante a execução, como pedidos recebidos pela API, erros de validação e exceções lançadas pela lógica de negócio. Estes registos permitem analisar falhas ocorridas em tempo de execução e facilitam a identificação da sua causa, tanto em ambiente de desenvolvimento como durante a execução dos testes automatizados. Em particular, erros críticos são registados com informação suficiente para apoiar o diagnóstico, incluindo mensagens de erro e contexto de execução.

Para além do logging, a utilização de códigos de estado HTTP apropriados e mensagens de erro claras contribui para a observabilidade do sistema do ponto de vista do cliente da API. Isto permite que consumidores da API, bem como testes automatizados, consigam distinguir corretamente entre cenários de sucesso, erros de autenticação, falhas de validação e erros internos.

3 Continuous testing

3.1 Estratégia de testes

A estratégia de testes adotada no projeto segue a **pirâmide clássica de testes**, privilegiando uma maior quantidade de testes unitários, um número mais reduzido de testes de integração e um conjunto ainda mais limitado de testes de aceitação. Esta abordagem permite obter feedback rápido durante o desenvolvimento, ao mesmo tempo que assegura a validação do comportamento integrado do sistema e das funcionalidades do ponto de vista do utilizador final.

Durante o desenvolvimento, não foi consistentemente adotada uma prática de desenvolvimento orientado a testes mas sim uma combinação de várias delas: em certas features, escrevi primeiro testes que depois validei com lógica de negócio, enquanto que em outras escrevi código e depois testei. O BDD foi usado na medida em que todas as user stories são acompanhadas de cenários para a sua validação.

3.2 Testes de aceitação e ATDD

Os testes de aceitação têm como objetivo validar o sistema a partir da perspetiva do utilizador final, assegurando que as funcionalidades implementadas cumprem os requisitos funcionais definidos nas user stories.

Os testes de aceitação foram escritos em selenium e são testes black box, nos quais não existe interação alguma com o funcionamento interno da aplicação: apenas são definidos comportamentos de browser que simulam aqueles de um utilizador real e a interface é navegada para chegar a determinado estado.

Apesar de, mais uma vez, não representarem a 100% o comportamento de um utilizador humano a navegar a interface, aproximam-se bastante e são suficientes para validar fluxos de utilização que percorram várias páginas.

3.3 Developer facing tests (unitários, integração)

Os developer-facing tests têm como principal objetivo validar o comportamento interno do sistema durante o desenvolvimento, garantindo que cada componente funciona corretamente de forma isolada e que a integração entre componentes ocorre como esperado. Estes testes são escritos a partir da perspetiva do programador (white-box testing) e constituem a base da estratégia de garantia de qualidade do projeto.

Os testes unitários são utilizados para validar a lógica de negócio individual, focando-se principalmente nos serviços e repositórios. Estes testes são desenvolvidos com JUnit, Mockito e AssertJ, permitindo o isolamento das dependências através de mocks, de forma a testar apenas o comportamento da unidade em causa.

Os testes de integração complementam os testes unitários, validando a interação entre diferentes camadas da aplicação, nomeadamente controladores, serviços e repositórios.

3.4 Testes não funcionais e de arquitetura

Testes não funcionais servem para validar o sistema de um ponto de vista arquitetural, não olhando a features ou requisitos mas sim a qualidades de performance do sistema. Foram escritos pequenos testes de carga para validar a performance nos endpoints mais críticos do sistema.