TQS: Quality Assurance manual

*Gonçalo Abrantes[104152] João Morais [103730] Nuno Faria[112994] Pedro Rei [107463]*

v2025-05-19

Contents

[**TQS: Quality Assurance manual 1**](#_heading=h.jolrfj1blh85)

[**1**](#_heading=h.enk91qcde0en) **Project management 1**

[1.1](#_heading=h.qupk2r40pflm) Assigned roles 1

[1.2](#_heading=h.hxjodc4kkthn) Backlog grooming and progress monitoring 1

[**2**](#_heading=h.z97bj520hwq8) **Code quality management 2**

[2.1](#_heading=h.s4mirhfg4jtd) Team policy for the use of generative AI 2

[2.2](#_heading=h.iz1t2l86vvav) Guidelines for contributors 2

[2.3](#_heading=h.ett1xjaakhoa) Code quality metrics and dashboards 2

[**3**](#_heading=h.kr2m81lsaete) **Continuous delivery pipeline (CI/CD) 2**

[3.1](#_heading=h.uc07zmhn8exu) Development workflow 2

[3.2](#_heading=h.5abkxugz3zn2) CI/CD pipeline and tools 2

[3.3](#_heading=h.bcpiuheao7vv) System observability 3

[3.4](#_heading=h.jl8ii9kebv56) Artifacts repository [Optional] 3

[**4**](#_heading=h.oopdde8i7oig) **Software testing 3**

[4.1](#_heading=h.e3u19pl1yd10) Overall testing strategy 3

[4.2](#_heading=h.bge0xs95u1s3) Functional testing and ATDD 3

[4.3](#_heading=h.880zf1xk6g6g) Developer facing testes (unit, integration) 3

[4.4](#_heading=h.t3m61to9lnz4) Exploratory testing 3

[4.5](#_heading=h.epuzrlx2c4t0) Non-function and architecture attributes testing 3

# Project management

## Assigned roles

* Team Coordinator (Pedro Rei): Garantir a distribuição justa de tarefas, promover a colaboração da equipa, resolver problemas proativamente e assegurar a entrega pontual dos resultados do projeto.
* Product Owner (Nuno Faria): Representa os interesses das partes interessadas, compreende profundamente o produto e o domínio de aplicação, esclarece dúvidas sobre as funcionalidades esperadas e participa na aceitação dos incrementos da solução.
* QA Engineer (João Morais): Responsável, em articulação com outros papéis, por promover práticas de garantia de qualidade, aplicar instrumentos de medição e monitorar o cumprimento das práticas acordadas pela equipa.
* DevOps Master (Gonçalo Abrantes): Responsável pela infraestrutura de desenvolvimento e produção, garantindo o correto funcionamento do framework e liderando a preparação de máquinas ou contêineres de deployment, repositórios Git, infraestrutura em cloud, operações de base de dados, entre outros.
* Developer: Todos os membros contribuem para as tarefas de desenvolvimento.

## Backlog grooming and progress monitoring

O trabalho é gerido no JIRA, com tarefas organizadas como user stories e bugs. O backlog é revisto semanalmente para refinar tarefas e atribuir story points. O progresso é acompanhado através de burndown charts e quadros Scrum/Kanban.

Existe monitorização proativa da cobertura de requisitos com integração JIRA–Xray, permitindo associar testes a histórias e garantir cobertura antes da conclusão.

# Code quality management

## Team policy for the use of generative AI

O uso de assistentes de IA (como o ChatGPT e o GitHub Copilot) é permitido com responsabilidade. Pode ser usado para sugerir código, refatorar, gerar testes ou esclarecer dúvidas, mas nunca para submeter código gerado sem validação humana. Novos membros devem considerar a IA como uma ferramenta de apoio, não como substituto da análise crítica e revisão de código.

## Guidelines for contributors

**Coding style**

Adota-se o Google Java Style Guide como base. Algumas regras principais incluem:

* Nomes descritivos e camelCase para variáveis e métodos
* Classes e interfaces em PascalCase
* Cada classe pública num ficheiro separado

Para mais detalhes, consultar o [Google Java Style Guide](https://google.github.io/styleguide/javaguide.html).

**Code reviewing**

* Todo o código deve ser submetido via Pull Request.
* Revisões são obrigatórias por, pelo menos, um membro da equipa.
* Sugestões de IA podem ser usadas, mas devem ser explicadas ou comentadas no Pull Request.
* Revisores devem verificar: clareza, cobertura de testes, legibilidade e aderência ao estilo de programação.

## Code quality metrics and dashboards

* Análise estatística com SonarQube Cloud.
* Relatórios gerados automaticamente em cada push.
* Quality gates definidos:
  + Cobertura de testes: >80%
  + Nenhum code smell de gravidade alta
  + Nenhuma ocorrência de security hotspot breach

# Continuous delivery pipeline (CI/CD)

## Development workflow

**Coding workflow**

O projeto segue o modelo GitHub Flow, adequado para desenvolvimento contínuo e integração rápida. O processo é o seguinte:

1. O programador escolhe uma user story atribuída no JIRA.
2. Cria uma nova branch a partir da main, nomeada segundo a tarefa.
3. Implementa a funcionalidade localmente e escreve os testes correspondentes.
4. Faz commit e push da branch para o GitHub.
5. Abre um Pull Request para revisão do código.

Todos os Pull Requests são obrigatoriamente revistos por outro membro da equipa. A revisão foca-se em clareza, estilo, cobertura de testes e qualidade geral (verificada pelo SonarQube). O Pull Request deve passar na pipeline CI para ser aceite.

**Definition of done**

Uma tarefa é considerada concluída quando:

* A funcionalidade está implementada e a funcionar localmente.
* Todos os testes (unitários, de integração e funcionais) passam.
* A cobertura de código mantém-se acima de 80% e sem security hotspot breach.
* O código foi revisto e aprovado.
* A pipeline CI no GitHub Actions terminou com sucesso.
* O Pull Request foi integrado na branch main.

## CI/CD pipeline and tools

A integração contínua é realizada com GitHub Actions, configurado para executar automaticamente a pipeline de testes e análise de qualidade a cada push ou pull request na branch main.

**Configuração da CI**

* O ficheiro maven.yml define as seguintes etapas:
  + Compilação e testes unitários com Maven
  + Testes de integração em Spring Boot
  + Cobertura de testes gerada por JaCoCo
  + Análise estática com SonarQube Cloud
  + Envio automático dos resultados de testes para o JIRA, através da API do Xray, com associação a um Test Plan

**Entrega Contínua**

Após a aprovação e merge do Pull Request, o código é integrado na branch main. A aplicação é construída como uma imagem Docker, utilizando um Dockerfile definido no projeto.

Esta automação garante entregas frequentes, testadas, documentadas e integradas com o sistema de gestão de requisitos.

## System observability

Para garantir o acompanhamento proativo do funcionamento do sistema, foram implementadas as seguintes práticas:

* Logging estruturado com Spring Boot, registando eventos como requisições HTTP, erros, e execuções de tarefas agendadas.
* Em caso de falha no acesso à API externa, é registado um erro e ativado um alerta via logs.
* Os testes automatizados na CI garantem que cada build executa corretamente os principais fluxos da aplicação.
* Os dados de testes, cobertura e qualidade (via JaCoCo e SonarQube) são analisados continuamente para prevenir regressões.

# Software testing

## Overall testing strategy

A estratégia de testes adotada combina várias abordagens para garantir a qualidade do sistema:

* Aplicação parcial de TDD em componentes de lógica de negócio.
* Uso de BDD com Cucumber para testes funcionais escritos do ponto de vista do utilizador.
* Testes de API e integração com o Spring Boot Test.
* Mocking com Mockito para isolar dependências.

Todos os testes são executados automaticamente no GitHub Actions, e os resultados são:

* Avaliados na pipeline de CI para garantir que falhas impedem merge;
* Recolhidos e enviados para o Xray no JIRA, permitindo rastreabilidade entre requisitos e testes;
* Validados com cobertura gerada por JaCoCo, integrada ao SonarQube.

## Functional testing and ATDD

Os testes funcionais são escritos do ponto de vista do utilizador. A política do projeto define que:

* Cada funcionalidade visível para o utilizador deve ter pelo menos um teste funcional associado.
* Sempre que uma user story afeta o comportamento externo da aplicação, o programador deve escrever vários cenários de teste que validem os critérios de aceitação definidos no JIRA.
* Os testes seguem a abordagem ATDD, sendo preferencialmente escritos antes da implementação da funcionalidade.
* Utiliza-se Cucumber para descrever os testes em linguagem natural e o Selenium WebDriver para automatizar a execução dos cenários em ambiente real de navegador.
* Estes testes são integrados na pipeline CI e os resultados são enviados para o Xray, permitindo rastrear a cobertura por requisito.

## Developer facing tests (unit, integration)

O projeto define uma política clara para garantir a qualidade do código através de testes orientados ao programados, focados na lógica interna e na interação entre componentes.

**Testes Unitários**

* Devem ser implementados sempre que se introduz uma nova lógica de negócio ou se altera código existente.
* O objetivo é validar comportamentos individuais de métodos e classes de forma isolada.
* Utiliza-se JUnit 5 em conjunto com Mockito para simular dependências e testar unidades de forma independente.
* Os testes unitário mais relevantes abrangem:
  + Validação de regras de negócio (como critérios de reserva e cancelamento)
  + Cálculo de preços, tempos de carregamento e geração de notificações
  + Conversações entres objetos de domínio e DTOs

**Testes de Integração**

* São obrigatórios para verificar a interação entre componentes da aplicação.
* Utilizam Spring Boot Test, permitindo carregar o contexto da aplicação e testar com base de dados em memória.
* Devem ser escritos sempre que uma funcionalidade envolve múltiplas camadas ou integração com componentes externos.

**Testes de API**

* Considerados uma extensão dos testes de integração, validam o correto funcionamento dos endpoints REST.
* São implementos com Spring MockMvc, permitindo testar as respostas HTTP, códigos de status, validação de input/output e mensagens de erro.
* Garantem que a API está em conformidade com os contratos definidos e são integrados no processo de CI.

Todos os testes são executados automaticamente na pipeline do GitHub Actions, e os resultados são enviados para o Xray, garantindo cobertura e rastreabilidade por requisito.

## Exploratory testing

O projeto inclui sessões regulares de testes exploratórios manuais, com o objetivo de detetar falhas que não são facilmente identificadas pelos testes automatizados.

A estratégia adotada consiste em:

* Realização de testes sem guião pré-definido, explorando a aplicação como um utilizador final.
* Utilização do Selenium WebDriver como ferramenta de apoio para gravar e reproduzir interações manuais durante as sessões exploratórias.
* Sempre que um comportamento inesperado é detetado, o cenário pode ser transformado num teste automatizado com Selenium, facilitando a repetição e regressão futura.
* Os defeitos ou incoerências encontrados são registados no JIRA, com evidência e passos para reprodução.

## Non-function and architecture attributes testing

A equipa realiza testes não funcionais com foco no desempenho, acessibilidade e boas práticas da aplicação.

**Testes de Desempenho com k6**

* Utilizamos a ferramenta k6 para desenvolver testes de carga HTTP, simulando cenários progressivos com múltiplos utilizadores virtuais.
* Os testes validam os nossos Service Level Objectives, incluindo:
  + 95% das requisições têm duração inferior a 1.1 segundos.
  + Menos de 1% de requisições HTTP falhadas.
* O comportamento do sistema é analisado com base em métricas como tempo médio/máximo de resposta, taxa de erro, e volume de tráfego.
* Os resultados podem ser visualizados em tempo real via K6 Web Dashboard.

**Auditoria de Frontend com Lighthouse**

* Utilizamos o Lighthouse para auditar a aplicação web quanto a:
  + Desempenho
  + Acessibilidade
  + Boas práticas e SEO
* Esta análise é especialmente útil para identificar problemas de usabilidade e otimização de interface, desde os primeiros ciclos de desenvolvimento.