TQS: Quality Assurance manual

Ana Alexandra Antunes [876543], Bruno Bernardes [876543]

v2025-12-16

Contents

[TQS: Quality Assurance manual 1](#_Toc197441196)

[1 Project management 1](#_Toc197441197)

[1.1 Assigned roles 1](#_Toc197441198)

[1.2 Backlog grooming and progress monitoring 1](#_Toc197441199)

[2 Code quality management 2](#_Toc197441200)

[2.1 Team policy for the use of generative AI 2](#_Toc197441201)

[2.2 Guidelines for contributors 2](#_Toc197441202)

[2.3 Code quality metrics and dashboards 2](#_Toc197441203)

[3 Continuous delivery pipeline (CI/CD) 2](#_Toc197441204)

[3.1 Development workflow 2](#_Toc197441205)

[3.2 CI/CD pipeline and tools 2](#_Toc197441206)

[3.3 System observability 3](#_Toc197441207)

[3.4 Artifacts repository [Optional] 3](#_Toc197441208)

[4 Software testing 3](#_Toc197441209)

[4.1 Overall testing strategy 3](#_Toc197441210)

[4.2 Functional testing and ATDD 3](#_Toc197441211)

[4.3 Developer facing testes (unit, integration) 3](#_Toc197441212)

[4.4 Exploratory testing 3](#_Toc197441213)

[4.5 Non-function and architecture attributes testing 3](#_Toc197441214)

[This report should be written with the following objectives in mind:

* Provide **clear guidelines** for project members needing to learn what the QA practices are. Use concise and informative content, allowing other software engineers to understand and comply with the practices.
* Provide context and **rationale** to the practices defined and **offer evidence of the tools/setups** used by the team. **Hyperlink team resources** in the text so it is possible to directly access the resources cited in context.

Tips on the expected content (marked in colored text) along with the document are meant to be removed.

You may use English or Portuguese; do not mix.]

# Project management

## Assigned roles

A equipa do projeto MineForNow encontra-se organizada de acordo com funções específicas atribuídas a cada membro, visando assegurar a cobertura de todas as áreas críticas do desenvolvimento e gestão do projeto.

A função de **Product Owner** é desempenhada por **Ana Pereira**, a quem cabe a responsabilidade de definir e priorizar o *backlog* do produto. Adicionalmente, atua como representante dos *stakeholders*, validando os requisitos implementados e garantindo que a solução final corresponde às necessidades efetivas dos utilizadores.

No que concerne à infraestrutura e operações, **Daniel Nunes** assume o papel de **DevOps Master**. As suas responsabilidades principais englobam a configuração e manutenção do *pipeline* de CI/CD, a gestão da infraestrutura, a automação dos processos de *build* e *deploy*, bem como a monitorização contínua dos sistemas.

A garantia de qualidade é assegurada por **Diego Aguilar**, na qualidade de **QA Engineer**. Este elemento define e implementa a estratégia de testes e assegura a integridade do código através de testes automatizados. É também responsável pela gestão da integração com ferramentas de análise de qualidade, tais como SonarCloud e Xray, e pela monitorização das respetivas métricas.

Por fim, a gestão operacional recai sobre **Vitalie Bologa**, o **Team Coordinator**. O seu foco centra-se em facilitar a comunicação entre os membros da equipa, coordenar as atividades de desenvolvimento, organizar as reuniões de equipa (como *stand-ups* e retrospetivas) e remover quaisquer impedimentos que possam comprometer o progresso dos trabalhos.

Para consulta detalhada sobre a dinâmica da equipa e o acompanhamento das tarefas, podem ser consultados o *README* do projeto e o quadro de gestão no Jira (*Jira Board - Team*).

## Backlog grooming and progress monitoring

A organização do trabalho e a gestão do ciclo de vida do desenvolvimento de *software* foram estruturadas com base em práticas ágeis (**Scrum**), utilizando a plataforma **Jira** como ferramenta central para a gestão do *backlog* e acompanhamento do progresso.

O *backlog* do projeto encontra-se organizado em *User Stories*, seguindo o formato padrão ("Como [papel], eu quero [ação], para [benefício]"), o que assegura o foco no valor para o utilizador. Para garantir a rastreabilidade total desde o requisito até à verificação, cada *User Story* é identificada com uma etiqueta única (no formato SCRUM-XX). O processo de gestão inclui a priorização das tarefas pelo *Product Owner*, baseada no valor de negócio e dependências técnicas, bem como sessões regulares de refinamento para estimativa e detalhe antes do início de cada *sprint*.

No que respeita à monitorização do progresso, a equipa privilegia indicadores de qualidade em detrimento de métricas puramente quantitativas como *story points*. O controlo é efetuado através do acompanhamento diário do estado das tarefas e, fundamentalmente, através de métricas automáticas de qualidade. Foram estabelecidos *Quality Gates* rigorosos via **SonarCloud**, que exigem uma cobertura de testes superior a 80% (monitorizada também pelo **JaCoCo**), ausência de vulnerabilidades e a não duplicação de código.

Um elemento diferenciador na abordagem da equipa é a implementação de uma **monitorização proativa da cobertura ao nível dos requisitos**, realizada através da integração com o **Xray (Test Management for Jira)**. Este processo estabelece uma ligação direta entre o código de teste e a especificação funcional.

A implementação técnica desta rastreabilidade inicia-se com a anotação dos testes, conforme exemplificado abaixo, permitindo associar inequivocamente um teste a um requisito específico:

* @Requirement("SCRUM-12") // Vehicle Details Feature
* @Requirement("SCRUM-32") // Authentication Feature
* @Requirement("SCRUM-46") // User Profile Feature
* ...

Esta estratégia é suportada por uma configuração automatizada no *pipeline* de CI/CD. Através do plugin Maven para Xray e da integração com **GitHub Actions**, os resultados dos testes (Unitários, Integração e BDD/Cucumber) são exportados automaticamente para o Jira após cada execução.

O resultado desta integração é a geração automática de uma Matriz de Rastreabilidade (*Traceability Matrix*) e de Planos de Teste no Xray. Isto oferece à equipa uma visibilidade em tempo real sobre quais os requisitos que possuem testes associados e o respetivo estado de aprovação, facilitando a identificação de lacunas na cobertura e suportando a tomada de decisão sobre a prontidão para o lançamento de novas versões (*releases*).

Para aprofundamento destes processos, podem ser consultados o *Backlog* no Jira, as configurações do *pipeline* de CI/CD do *backend* e o ficheiro pom.xml onde consta a configuração do Xray.

# Code quality management

## Team policy for the use of generative AI

## Política de Utilização de Inteligência Artificial Generativa

A equipa do projeto MineForNow reconhece o valor estratégico das ferramentas de Inteligência Artificial generativa (como *GitHub Copilot* ou *ChatGPT*) enquanto assistentes de desenvolvimento, mas estabelece diretrizes rigorosas para assegurar o seu uso responsável e a integridade do *software*.

**Posicionamento da Equipa**

A política adotada distingue o uso destas ferramentas em dois contextos principais, mantendo sempre a premissa de que a IA serve de apoio e não de substituto:

* **No Código de Produção:** O uso é **permitido e encorajado** para acelerar o desenvolvimento, sugerir padrões, criar estruturas iniciais e gerar código *boilerplate*. No entanto, impõe-se a obrigatoriedade de revisão: todo o código gerado **deve ser testado e, fundamentalmente, compreendido pelo programador** antes de ser integrado. É estritamente proibida a aceitação de código cujo funcionamento não seja dominado pelo autor.
* **No Código de Testes:** A equipa incentiva o uso de IA para a **geração de casos** de teste, dados de teste e identificação de *edge cases* **que possam ter sido ignorados**. Contudo, estes testes **exigem uma validação humana** para garantir que cobrem comportamentos reais e relevantes, rejeitando-se a geração automática sem verificação de eficácia.

**Diretrizes Práticas para os Membros**

Para garantir a qualidade e a segurança do projeto, foram definidos princípios de atuação que regem a interação com estas ferramentas:

1. **Compreensão e Pensamento Crítico:** O princípio fundamental é "compreender antes de integrar". Os membros devem utilizar as sugestões da IA como um ponto de partida ou rascunho, adaptando o código ao contexto específico do projeto. É vetada a prática de "copiar/colar" cegamente; o desenvolvedor deve analisar criticamente todas as sugestões e nunca substituir o seu julgamento de engenharia pela resposta da ferramenta.
2. **Qualidade, Segurança e Arquitetura:** O código gerado por IA está sujeito aos mesmos padrões de qualidade que o código escrito manualmente. Isto implica a execução rigorosa de testes (com cobertura superior a 80%), análise estática via SonarCloud e validação de vulnerabilidades de segurança. É crucial que as sugestões da IA não violem a arquitetura do sistema nem comprometam decisões de *design* previamente estabelecidas.
3. **Aprendizagem e Ética:** A IA deve ser encarada também como uma ferramenta educacional para descobrir novos padrões e técnicas. Caso sejam utilizados blocos substanciais de código, a equipa deve documentar a origem e respeitar as licenças aplicáveis.

**Fluxo de Trabalho**

O processo de desenvolvimento assistido por IA deve seguir um fluxo estruturado para mitigar riscos: inicia-se com a definição clara do requisito, passando pela consulta opcional à IA para sugestões. Segue-se a etapa crítica de revisão e adaptação do código, implementação de testes e análise estática local. O processo encerra-se com a revisão de código pela equipa (*Code Review*) e a validação automatizada no pipeline de CI/CD.

Em suma, a política da equipa estabelece que a responsabilidade final pelo código, e pelos seus eventuais falhas ou vulnerabilidades, recai sempre sobre o desenvolvedor, independentemente do grau de assistência tecnológica recebido.

## Guidelines for contributors

Coding style

Para assegurar a consistência, legibilidade e facilidade de manutenção do projeto MineForNow, a equipa estabeleceu um conjunto rigoroso de convenções de código adaptadas a cada tecnologia da *stack*.

No que respeita ao desenvolvimento *Backend*, a equipa segue diretrizes alinhadas com o *Google Java Style Guide* e as melhores práticas do Spring Boot. As convenções de nomenclatura impõem o uso de **PascalCase** para classes (ex: UserService) e **camelCase** para métodos e variáveis (ex: findById). As constantes devem ser sempre declaradas em **UPPER\_SNAKE\_CASE**.

A organização do código reflete uma arquitetura em camadas bem definida, onde os pacotes são estruturados em letras minúsculas (tqs.backend.\*) e segregados por responsabilidade: controller para *endpoints* REST, service para lógica de negócio, repository para acesso a dados, model para entidades de domínio e dto para transferência de dados.

Para reduzir a verbosidade do código (*boilerplate*), utiliza-se extensivamente a biblioteca **Lombok**. A preferência recai sobre o uso de @Builder para a construção de objetos complexos e anotações como @Getter, @Setter e @NoArgsConstructor

**Desenvolvimento Frontend (React/JavaScript)**

No *Frontend*, as práticas baseiam-se no *Airbnb JavaScript Style Guide* e na documentação oficial do React. A nomenclatura segue **PascalCase** para componentes e ficheiros (ex: HeroSearch.jsx), enquanto funções e variáveis adotam **camelCase**.

A estrutura do projeto separa claramente os componentes reutilizáveis (components/ui), as páginas (pages), os serviços de comunicação com a API (services) e a gestão de estado global (contexts). Em termos de padrões de desenho, a equipa privilegia componentes funcionais com *Hooks*, extraindo lógica complexa para *custom hooks* sempre que necessário. A qualidade do código é assegurada automaticamente através de configurações de **ESLint** que integram regras recomendadas para React e *Hook.*

Independentemente da tecnologia, a equipa adota práticas comuns para garantir a qualidade do repositório. A documentação no código foca-se no "porquê" das decisões de *design* não triviais, evitando comentários redundantes que apenas descrevem o óbvio.

A gestão de alterações segue o padrão de **Conventional Commits**, exigindo mensagens claras e estruturadas no formato tipo(scope): descrição. Os tipos de *commit* são padronizados (ex: feat, fix, docs, test, refactor), facilitando a geração automática de *changelogs* e a compreensão do histórico do projeto. Por exemplo, uma nova funcionalidade de pesquisa seria registada como feat(vehicle): add search by location. Por fim, a gestão de dependências é proativa, mantendo as bibliotecas atualizadas e justificando o uso de versões específicas quando estritamente necessário.

Code reviewing

A equipa implementa um processo rigoroso de revisão de código (*Code Review*) com o objetivo triplo de garantir a qualidade do software, promover a partilha de conhecimento entre os membros e assegurar a conformidade com os padrões arquiteturais estabelecidos.

A revisão de código é mandatória para todos os *Pull Requests* (PR) destinados aos ramos principais (main ou develop), bem como para quaisquer alterações em componentes críticos (como autenticação e segurança), configurações de infraestrutura ou refatorizações significativas. Embora opcional, encoraja-se também a revisão em *feature branches* e a prática de *pair programming*.

O processo inicia-se antes da submissão formal. O autor deve garantir que o código compila sem erros, que a cobertura de testes cumpre a meta definida (>80%) e que não existem alertas críticos na análise estática local. Ao criar o PR, é exigida uma descrição detalhada que inclua a motivação da alteração, o link para a *issue* correspondente no Jira (ex: Closes SCRUM-XX) e, no caso de alterações de interface, evidências visuais como capturas de ecrã.

Durante a revisão, é atribuído pelo menos um revisor (*Reviewer*) responsável por validar a lógica, a segurança, a performance e a aderência às convenções da equipa. O *feedback* deve ser construtivo e objetivo. Cabe ao autor endereçar todos os comentários, justificando decisões ou implementando as correções solicitadas.

A aprovação final e a consequente integração (*Merge*) dependem do cumprimento cumulativo de vários critérios: a aprovação explicita de pelo menos um revisor, a resolução de todos os conflitos e, crucialmente, o sucesso de todos os *checks* automatizados do pipeline de CI/CD. Isto inclui a passagem nos testes unitários e de integração, a aprovação no *Quality Gate* do **SonarCloud** e o envio bem-sucedido dos resultados de teste para o **Xray**. A equipa integra opcionalmente ferramentas de IA, como o *GitHub Copilot*, no fluxo de revisão. Estas ferramentas são utilizadas para auxiliar na deteção de *bugs*, sugerir melhorias de performance e verificar convenções. No entanto, o protocolo é claro: a IA não substitui a revisão humana. O contexto de negócio e as decisões arquiteturais requerem sempre validação manual.

## Code quality metrics and dashboards

A garantia de qualidade no projeto MineForNow baseia-se num sistema robusto de análise estática e monitorização contínua, totalmente integrado no pipeline de CI/CD. Este ecossistema automatiza a validação técnica, assegurando que apenas código conforme aos padrões definidos é integrado na base principal.

Infraestrutura de Análise Estática

A monitorização central é realizada pelo SonarCloud, que executa uma análise profunda a cada alteração submetida, avaliando pilares como fiabilidade, manutenibilidade e segurança (incluindo a deteção de vulnerabilidades e hotspots). Complementarmente, utiliza-se o JaCoCo para medir a cobertura de testes através de uma estratégia de "Merged Coverage". Esta configuração consolida a execução de testes unitários e de integração num único relatório e aplica exclusões a código boilerplate (como DTOs e configurações), garantindo que as métricas refletem a qualidade real da lógica de negócio.

Quality Gates e Critérios de Aceitação

Para assegurar a integridade do software, foram definidos Quality Gates rigorosos que bloqueiam automaticamente a integração de código em caso de incumprimento:

* Cobertura de Código: O sistema impõe um limiar mínimo de 80% de cobertura de linhas. Se a cobertura combinada cair abaixo deste valor, o processo de build é abortado.
* Qualidade do Novo Código: O SonarCloud aplica o perfil "Sonar way", focado no código novo ou alterado. Exige-se classificação máxima ("A") em todas as métricas e uma taxa de duplicação inferior a 3%, prevenindo o aumento da dívida técnica.
* Execução de Testes: Vigora uma política de tolerância zero, onde qualquer falha em testes unitários ou de integração impede a aceitação do código.

Visualização e Monitorização

A visibilidade destas métricas é assegurada pelo dashboard global do SonarCloud e por relatórios HTML locais do JaCoCo, que permitem aos desenvolvedores identificar linhas específicas não cobertas por testes. O controlo é contínuo: a validação dos Quality Gates é pré-requisito técnico para a fusão de qualquer Pull Request. Cabe ao QA Engineer e ao DevOps Master a revisão periódica destas tendências para ajustar limiares e orientar a refatorização do sistema.

# Continuous delivery pipeline (CI/CD)

## Development workflow

Coding workflow

A equipa adota uma metodologia de trabalho baseada numa adaptação do **GitHub Flow**, estabelecendo o ramo **develop** como o eixo central de integração contínua. O ciclo operacional inicia-se no Jira, onde a seleção de uma tarefa desencadeia a criação de um ramo de trabalho a partir de develop, que deve obedecer rigorosamente à nomenclatura **feat/SCRUM-XX-descricao**. Esta prática assegura, desde a origem, uma rastreabilidade completa e auditável que liga a *User Story* ao código, aos *commits* (padronizados via *Conventional Commits*) e aos testes (anotados com @Requirement), permitindo a correlação automática de resultados no Xray.

O processo de implementação rege-se por princípios de *Test-Driven Development* (TDD). Antes de qualquer submissão, é mandatória a validação local — utilizando mvn clean verify para o *backend* ou npm test para o *frontend* — e a garantia de uma cobertura de código superior a 80%. A integração das alterações na base de código ocorre exclusivamente via *Pull Request* para o ramo develop, exigindo cumulativamente a aprovação de pelo menos um revisor e o sucesso de todos os testes automáticos do *pipeline* de CI/CD.

A estratégia de ramificação foi desenhada para proteger a estabilidade do projeto: enquanto os ramos main e develop são protegidos e sujeitos à execução completa do *pipeline* (incluindo SonarCloud e Xray), os ramos temporários de funcionalidade (feat/\*) que permitem o desenvolvimento isolado com validação local, garantindo que apenas código estável atinge os ambientes de integração e produção.

Definition of done

Para que uma *User Story* seja considerada concluída ("Done"), deve cumprir rigorosamente um conjunto de requisitos de qualidade, técnicos e processuais. Ao nível da implementação, o código deve satisfazer todos os critérios de aceitação definidos e respeitar as convenções de estilo da equipa, sendo integrado no ramo **develop** apenas após a aprovação em revisão de código por pelo menos um par e a resolução de quaisquer conflitos.

A garantia de qualidade é validada através da execução bem-sucedida de testes unitários e de integração, exigindo-se uma cobertura de código mínima de 80% verificada pelo JaCoCo. Adicionalmente, o código deve superar todos os *Quality Gates* automatizados no *pipeline* de CI/CD, o que implica um *build* com sucesso e a aprovação na análise do SonarCloud, garantindo a ausência de *bugs*, vulnerabilidades ou *code smells* críticos.

Por fim, o processo assegura a total rastreabilidade e validação funcional. Isto materializa-se através da anotação técnica dos testes (para sincronização com o Xray), da atualização da documentação e do estado da tarefa no Jira com o respetivo *Pull Request*. O ciclo encerra-se com a validação da funcionalidade em ambiente de desenvolvimento.

## CI/CD pipeline and tools

A estratégia de Integração Contínua (CI) do projeto é orquestrada através do **GitHub Actions**, utilizando fluxos de trabalho distintos definidos nos ficheiros de configuração do repositório (.github/workflows/). Estes *pipelines* são acionados automaticamente mediante operações de *push* ou criação de *Pull Requests* direcionados aos ramos principais (main e develop), garantindo a validação constante do código antes da sua integração.

**Fluxos de Trabalho: Backend e Frontend**

O *pipeline* do *backend* caracteriza-se pela sua abrangência. O processo inicia-se com a preparação do ambiente (Java 21) e o carregamento eficiente de dependências via *cache*. De seguida, executa-se o ciclo completo de construção e verificação (mvn -B clean verify), englobando testes unitários e de integração. Posteriormente, o sistema gera relatórios de cobertura e *badges* visuais através do JaCoCo, executa a análise estática no SonarCloud e valida o cumprimento dos *Quality Gates*. O ciclo encerra-se com a exportação automática dos resultados dos testes para o Xray, assegurando a sincronização com o plano de testes no Jira.

Paralelamente, o fluxo do *frontend* foca-se na integridade da construção (*build*). Configurado sobre Node.js 20, o processo assegura a instalação limpa e determinística das dependências (npm ci) e a compilação bem-sucedida do projeto, estando prevista a inclusão futura de etapas de *linting* e testes automatizados.

**Segurança e Artefactos**

A integração segura com serviços externos é gerida através de variáveis encriptadas (Secrets), como o token de autenticação do SonarCloud e as credenciais de acesso à API do Xray. Como resultado de cada execução bem-sucedida, o sistema produz e arquiva diversos artefactos essenciais para auditoria e *deployment*, incluindo relatórios detalhados de cobertura (formatos HTML e XML), ficheiros padronizados de resultados de testes (JUnit XML), *badges* de estado e o executável final da aplicação (.jar).

TODO: CD

## System observability

What was prepared to ensure [proactive monitoring of the system operational conditions](https://sre.google/sre-book/monitoring-distributed-systems/)? Which events/alarms are triggered? Which data is collected for assessment?...

TODO

## Artifacts repository [Optional]

A gestão de dependências e do ciclo de vida dos artefactos no projeto é assegurada principalmente através do ecossistema **Maven**. Todas as bibliotecas externas necessárias — incluindo componentes fundamentais como Spring Boot (versão 3.3.6), JUnit e JaCoCo — são obtidas diretamente do **Maven Central Repository**. Para otimizar os tempos de execução no ambiente de Integração Contínua, a equipa utiliza estratégias de *cache* geridas pelo GitHub Actions, que armazenam as dependências descarregadas entre execuções do *build*.

**Artefactos Produzidos e Gestão Local**

O processo de compilação e empacotamento (mvn package) resulta na geração de diversos artefactos essenciais. O principal entregável é o executável da aplicação (*JAR*), nomeado backend-0.0.1-SNAPSHOT.jar. Paralelamente, são gerados artefactos de qualidade crítica para a validação do projeto, nomeadamente os relatórios de teste em formato XML (JUnit) e os dados de cobertura de código do JaCoCo, disponibilizados no diretório target/. Ao nível do desenvolvimento local, a gestão é facilitada pelo repositório local do Maven (~/.m2/repository/), que armazena as dependências para acelerar a compilação.

# Continuous testing

## Overall testing strategy

A equipa MineForNow implementa uma estratégia de testes em múltiplas camadas, integrando diversas metodologias num ecossistema unificado pelo pipeline de CI/CD, com o objetivo de assegurar a qualidade contínua do software.

Abordagem Metodológica

A filosofia de testes alicerça-se em três pilares metodológicos complementares. Primeiramente, aplica-se o Test-Driven Development (TDD) na implementação da lógica de negócio crítica (serviços e repositórios). Esta abordagem garante que os testes precedem o código, promovendo um design mais limpo e uma testabilidade intrínseca.

Para os testes de aceitação, utiliza-se o Behavior-Driven Development (BDD). Os cenários são descritos em linguagem natural (Português) através de ficheiros Gherkin e processados pelo Cucumber. Esta escolha facilita o alinhamento com os stakeholders e assegura a rastreabilidade direta com os requisitos, uma vez que os testes são etiquetados com as respetivas User Stories (@SCRUM-XX).

Estruturalmente, a distribuição dos testes obedece ao princípio da Pirâmide de Testes. A base é constituída por uma vasta cobertura de testes unitários (rápidos e isolados), suportada por uma camada intermédia de testes de integração e culminando, no topo, com testes End-to-End (E2E) mais seletivos.

Ecossistema Tecnológico

O ecossistema tecnológico de suporte à qualidade é liderado pelo JUnit 5, servindo como base para a automação no backend. A validação abrange desde o isolamento unitário, garantido pelo Mockito, até à verificação da camada Web e APIs REST através de MockMvc e Spring Boot Test. A gestão da cobertura de código é assegurada pelo JaCoCo, enquanto a rastreabilidade dos resultados é integrada no Jira via Xray. Adicionalmente, ferramentas como Selenium e Playwright estão configuradas para suportar futuros testes de interface, existindo planos para adotar Jest e React Testing Library no frontend.

Integração no Pipeline de CI/CD e Métricas

A integridade do sistema é validada automaticamente no pipeline de Integração Contínua (GitHub Actions), que executa a bateria completa de testes a cada alteração no código. O processo impõe critérios de aceitação rigorosos, bloqueando a integração de código caso ocorram falhas ou se a cobertura for inferior a 80%. Esta automação garante um ciclo de feedback rápido e assegura que os resultados são exportados em tempo real para as ferramentas de gestão de qualidade e análise estática.

## Acceptance testing and ATDD

A equipa MineForNow implementa uma política de testes de aceitação baseada em **Behavior-Driven Development (BDD)**, utilizando a *framework* **Cucumber**. Esta abordagem privilegia testes de "caixa fechada" desenhados sob a perspetiva do utilizador final, garantindo que o software cumpre os requisitos comportamentais definidos.

**Política e Âmbito de Aplicação**

A criação de testes de aceitação é mandatória para todas as *User Stories* que envolvam interação direta com o utilizador, bem como para funcionalidades críticas de negócio — tais como fluxos de autenticação, sistemas de reservas e motores de pesquisa — e quaisquer alterações que impactem fluxos existentes. Em contrapartida, esta prática é facultativa para tarefas estritamente técnicas, como refatorizações de código, otimizações de performance ou configurações de infraestrutura que não alterem o comportamento funcional do sistema.

**Implementação e Linguagem**

A especificação dos testes é realizada através de ficheiros de funcionalidade (*Feature Files*) escritos em linguagem **Gherkin**. A equipa optou deliberadamente pela utilização do Português na descrição dos cenários (ex: "Dado que...", "Quando...", "Então...") para eliminar barreiras linguísticas, facilitando a colaboração direta com o *Product Owner* e o alinhamento com os *stakeholders* locais. Estes cenários em linguagem natural funcionam como "documentação executável", sendo traduzidos tecnicamente para código Java através de *Step Definitions*, que executam as chamadas à API e validam as respostas do sistema.

**Execução, Relatórios e Rastreabilidade**

A execução destes testes encontra-se totalmente integrada no ciclo de vida de construção do projeto através do **Maven**. Podem ser executados isoladamente via perfil específico (-P cucumber) ou como parte integrante da fase de verify, juntamente com os testes unitários e de integração.

Os resultados da execução geram relatórios automáticos em formatos HTML e JSON no diretório target/. Mais importante ainda, estes resultados são sincronizados automaticamente com o **Xray**, estabelecendo uma rastreabilidade direta e auditável entre o requisito de negócio (identificado pela *tag* @SCRUM-XX) e a sua validação técnica. Esta estratégia assegura que a evolução do código se mantém fiel às especificações originais.

## Developer facing tests (unit, integration)

A estratégia de validação técnica do projeto divide-se em níveis de granularidade distintos, assegurando que tanto a lógica isolada como a integração de componentes são verificadas pelos desenvolvedores antes da submissão do código.

Testes Unitários (Unit Tests)

A base da pirâmide de testes é constituída pelos Testes Unitários, cuja implementação é obrigatória para toda a lógica de negócio. Esta política abrange serviços, utilitários, componentes de segurança, DTOs com validação e repositórios com consultas personalizadas. Apenas classes de configuração simples e objetos de domínio sem lógica (POJOs) estão isentos desta obrigatoriedade.

A abordagem técnica segue um modelo de "caixa branca", fundamentado no conhecimento da implementação interna. Utilizando o JUnit 5 em conjunto com o Mockito, a equipa privilegia o isolamento total das dependências. Isto garante que os testes são extremamente rápidos e determinísticos, não dependendo de recursos externos como redes ou bases de dados.

A estrutura dos testes obedece ao padrão Given-When-Then e adapta-se à camada testada:

* Camada de Serviço: Utiliza-se a injeção de mocks (@InjectMocks) para simular o comportamento de repositórios e componentes externos, validando a lógica de negócio pura.
* Camada de Persistência: Recorre-se à anotação @DataJpaTest e ao TestEntityManager para validar o mapeamento objeto-relacional e queries específicas.
* Camada Web (Controllers): Utiliza-se @WebMvcTest combinado com MockMvc para testar a serialização JSON e os códigos de resposta HTTP de forma isolada, sem carregar o contexto completo do servidor.

Testes de Integração (Integration Tests)

Complementarmente, os Testes de Integração são mandatários para validar endpoints críticos, fluxos de segurança (autenticação/autorização) e processos que atravessam múltiplas camadas arquiteturais (do controlador ao repositório). Ao contrário dos testes unitários, estes seguem uma abordagem de "caixa cinza", minimizando o uso de mocks para testar a interação real entre componentes.

A infraestrutura destes testes baseia-se no Spring Boot Test, que carrega o contexto completo da aplicação (@SpringBootTest) num servidor web embarcado com porta aleatória. A persistência é assegurada por uma base de dados real em memória (H2), que é reiniciada entre execuções para garantir a independência dos cenários. Estas classes são identificadas pelo sufixo IT e localizadas numa diretoria específica de integração.

Testes de API

A validação da API REST não constitui uma camada isolada, mas encontra-se integrada nos testes de integração através da utilização do TestRestTemplate. A equipa optou por esta ferramenta nativa do ecossistema Spring em detrimento de bibliotecas externas como o REST-Assured, visando maior consistência e facilidade de configuração.

A matriz de cobertura destes testes abrange todos os endpoints REST, validando exaustivamente os métodos HTTP (GET, POST, PUT, DELETE), os códigos de estado para cenários de sucesso e erro (200, 400, 401, 404), a estrutura dos corpos de resposta e o processamento de cabeçalhos de autenticação. A execução destes testes ocorre na fase verify do pipeline de CI/CD, sendo os resultados consolidados nos relatórios de cobertura unificada e automaticamente ligados aos requisitos no Xray.

## Exploratory testing

[strategy for non-scripted tests, if any]

TODO

## Non-function and architecture attributes testing

[Project policy for writing performance tests and associated resources.]  
TODO