TQS: Quality Assurance manual

André Oliveira [107637], Duarte Cruz [107359], Tomás Matos [108624], Diogo Almeida [108902]
v2024-06-03

Índice

1 Project management

1.1 Team and roles

Dentro da nossa equipa, designámos funções específicas para garantir um funcionamento harmonioso e um progresso eficiente em direção aos nossos objetivos de projeto.

André Oliveira assume o papel fundamental de *Team Leader*. Ele supervisiona o progresso de toda a equipa, coordenando tarefas, gerindo recursos e assegurando o alinhamento com os objetivos do projeto. O André serve como o ponto central de comunicação entre a equipa e os *stakeholders*, resolvendo habilmente quaisquer conflitos que possam surgir.

Duarte Cruz, o nosso especialista em *DevOps*, é essencial no desenvolvimento, implantação e manutenção da infraestrutura e das ferramentas do nosso projeto. O seu foco na automatização de processos, assegurando a integração e entrega contínuas, e na otimização do desempenho do sistema é crucial para simplificar o nosso fluxo de trabalho e aumentar a fiabilidade.

Tomás Matos ocupa a posição crucial de *Product Owner*, atuando como a voz dos nossos clientes. Ele define e prioriza meticulosamente o nosso *backlog* de produto, recolhendo requisitos e garantindo que as nossas entregas correspondam e superem as expectativas dos clientes. O Tomás trabalha de perto com a equipa, guiando-os para que cumpram a visão e os objetivos do projeto.

Por último, Diogo Almeida, o nosso Arquiteto, é responsável por desenhar a estrutura e arquitetura geral do projeto. Ele formula estratégias técnicas, seleciona as tecnologias apropriadas e assegura que o nosso sistema cumpre rigorosos padrões de desempenho, escalabilidade e segurança. O Diogo colabora estreitamente com a equipa de desenvolvimento, fornecendo orientação arquitetónica e resolvendo quaisquer desafios técnicos que surjam ao longo do ciclo de vida do projeto.

1.2 Agile backlog management and work assignment

Adotámos metodologias ágeis para gerir o nosso *backlog* e atribuir trabalho, com uma forte ênfase em histórias de utilizador e testes baseados no *Behavior Driven Development* (BDD). A nossa abordagem centra-se em dividir funcionalidades em histórias de utilizador que proporcionam valor tangível aos nossos utilizadores finais.

Para a gestão do *backlog*, confiamos no *Jira*, uma ferramenta poderosa que nos permite organizar meticulosamente o nosso *backlog*. Com o *Jira*, criamos e priorizamos histórias de utilizador com base em fatores como *feedback* dos clientes, objetivos empresariais e dependências técnicas. Esta priorização garante que estamos sempre a trabalhar nas funcionalidades mais impactantes primeiro, maximizando a entrega de valor.

As nossas equipas participam em sessões de planeamento colaborativo onde decidimos coletivamente quais as histórias de utilizador a abordar no próximo *sprint*. Durante estas sessões, dividimos funcionalidades maiores em tarefas menores e mais geríveis. Esta decomposição de tarefas facilita a atribuição de trabalho aos membros da equipa com base na sua especialização e disponibilidade.

Um aspeto fundamental do nosso processo de atribuição de trabalho é a integração dos testes BDD. Antes de qualquer trabalho começar, definimos critérios de aceitação para cada história de utilizador em colaboração com os *stakeholders*. Estes critérios servem de base para escrever cenários de teste BDD, garantindo que a funcionalidade implementada está alinhada com os resultados desejados.

Ao aderir a estas práticas ágeis, pretendemos manter um *backlog* bem organizado, atribuir trabalho de forma eficiente e entregar *software* de alta qualidade que satisfaça as necessidades e expectativas dos nossos utilizadores.

2 Code quality management

2.1 Guidelines for contributors (coding style)

Estabelecemos um conjunto de diretrizes conhecido como *"GateMate Coding Style"* para manter a consistência e clareza na nossa base de código. Estas diretrizes são essenciais para garantir que todos os colaboradores seguem uma abordagem unificada para a codificação, tornando a base de código mais legível, fácil de manter e colaborativa.

O *"GateMate Coding Style"* abrange vários aspetos chave. Primeiramente, define convenções de nomenclatura para variáveis, funções, classes e outros identificadores, enfatizando o uso de nomes descritivos e significativos para melhorar a legibilidade.

Comentários claros e informativos são outro componente crucial do nosso estilo de codificação. Enfatizamos a importância de documentar a funcionalidade do código, a lógica por trás dele e possíveis armadilhas, para ajudar os futuros mantenedores a compreenderem a base de código.

Além disso, o nosso estilo de codificação fornece diretrizes para organizar ficheiros de código, módulos e diretórios de maneira lógica e intuitiva. Esta abordagem estruturada simplifica a navegação pelo código e reduz a sobrecarga cognitiva para os desenvolvedores.

A incorporação de melhores práticas para tratamento de erros, tratamento de exceções e otimização de código também faz parte do nosso estilo de codificação. Ao seguir as melhores práticas estabelecidas, garantimos que o nosso código é robusto, eficiente e escalável.

A documentação abrangente é requerida para interfaces públicas, *APIs* e módulos de código, incluindo exemplos de uso, descrições de parâmetros, valores de retorno e considerações ou advertências relevantes.

Finalmente, o nosso estilo de codificação inclui diretrizes para o uso do controlo de versões, tais como convenções para mensagens de *commit*, estratégias de ramificação e processos de revisão de código. Aderir a estas diretrizes promove a colaboração, a qualidade do código e a rastreabilidade do projeto.

No geral, o *"GateMate Coding Style"* serve como uma estrutura fundamental para uma colaboração eficaz e garante que o nosso *software* mantém os mais altos padrões de qualidade e profissionalismo ao longo do seu ciclo de desenvolvimento.

2.2 Code quality metrics and dashboards

A integração do *SonarCloud* no nosso fluxo de trabalho de desenvolvimento elevou significativamente a nossa abordagem para garantir a qualidade do código e minimizar a dívida técnica. Este foi incorporado de forma harmoniosa no nosso *pipeline* de *Continuous Integration* (CI), tornando-o uma parte integral do nosso processo de desenvolvimento. Antes que qualquer *pull request* possa ser fundido na *branch* principal, ele passa por uma rigorosa análise automatizada de código.

A nossa adesão aos *quality gates* padrão do *SonarCloud* é inabalável. Estabelecemos uma política rigorosa que dita que todas as alterações de código devem passar por esses *quality gates* antes de serem consideradas aptas para fusão na *branch* principal. Isto assegura que apenas o código de mais alta qualidade, livre de problemas e violações maiores, chege ao nosso ambiente de produção.

Quando o *SonarCloud* sinaliza problemas ou violações durante o processo de análise, temos um fluxo de trabalho padronizado para abordá-los. Os desenvolvedores são responsáveis por resolver quaisquer problemas destacados nos seus respetivos *pull requests* antes de serem elegíveis para fusão. Este compromisso em resolver problemas de forma atempada garante que a qualidade do código permaneça uma prioridade ao longo do ciclo de vida do desenvolvimento.

Durante as revisões de código, os relatórios e *dashboards* do *SonarCloud* servem como pontos de referência valiosos. Os revisores confiam nas descobertas do *SonarCloud* para identificar potenciais áreas de melhoria ou para discutir quaisquer questões de qualidade do código com o autor. Esta abordagem colaborativa promove uma cultura de melhoria contínua e responsabilidade coletiva pela qualidade do código dentro da nossa equipa.

A análise regular de tendências históricas nas métricas de qualidade de código, utilizando os *dashboards* do *SonarCloud*, permite-nos obter *insights* sobre as nossas práticas de desenvolvimento ao longo do tempo. Acompanhar estas tendências possibilita-nos identificar áreas para melhoria, medir o impacto das nossas iniciativas de qualidade de código e tomar decisões informadas para melhorar ainda mais os nossos processos de desenvolvimento.

Estamos comprometidos em fornecer formação e suporte aos desenvolvedores para ajudá-los a compreender as descobertas do *SonarCloud* e abordar eficazmente as questões de qualidade do código. Esta abordagem proativa capacita os desenvolvedores a escreverem código mais limpo e mais fácil de manter, além de incutir um senso de responsabilidade pela qualidade do código na equipa.

Ao integrar o *SonarCloud* no nosso processo de desenvolvimento desta forma, conseguimos manter um elevado padrão de qualidade de código, reduzir a dívida técnica e entregar *software* mais fiável e fácil de manter para os nossos utilizadores.

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

Adotámos um fluxo de trabalho *Git*, semelhante ao modelo *GitHub Flow*, para orquestrar o nosso processo de desenvolvimento de forma fluida. Aqui está uma visão geral de como o nosso fluxo de trabalho opera e o seu alinhamento com as histórias de utilizador:

O nosso fluxo de trabalho *Git* começa com os desenvolvedores a criar *branches* de funcionalidades derivadas da *branch* principal. Estas *branches* são dedicadas a histórias de utilizador ou funcionalidades específicas identificadas no nosso *backlog*. Os desenvolvedores trabalham diligentemente nestas *branches* para implementar a funcionalidade descrita na história de utilizador correspondente.

Uma vez que um desenvolvedor conclui o trabalho numa funcionalidade ou história de utilizador, ele inicia um *Pull Request* (PR) para fundir as suas alterações de volta na *branch* principal. Este *pull request* torna-se o ponto focal para a revisão de código e colaboração entre os membros da equipa.

Durante a fase de revisão de código, os membros da equipa avaliam minuciosamente as alterações de código introduzidas no *pull request*. Eles examinam o código para garantir a conformidade com os padrões de codificação, correção e qualidade geral. São fornecidos *feedbacks* e sugestões de melhoria, promovendo uma cultura de colaboração e melhoria contínua.

Além disso, criámos regras de *branch* para garantir a qualidade e a integridade do nosso processo de desenvolvimento. Estas regras incluem a necessidade obrigatória de haver uma revisão no *pull request* por outra pessoa e a impossibilidade de fazer fusão da *branch* diretamente para a *branch* principal sem passar por um *pull request*.

O nosso *pipeline* de CI é acionado automaticamente ao abrir um PR. Este pipeline executa uma série de testes, incluindo testes unitários, testes de integração e análise estática de código usando o *SonarCloud*. As alterações de código devem passar por todas as verificações no *pipeline* de CI antes de serem consideradas prontas para fusão na *branch* principal.

Uma vez que as alterações de código tenham passado por uma revisão de código bem-sucedida e todas as verificações no *pipeline* de CI, elas são fundidass na *branch* principal. Esta integração marca a conclusão da história de utilizador, tornando a funcionalidade implementada parte da base de código do projeto.

Cada *branch* de funcionalidade corresponde diretamente a uma história de utilizador ou funcionalidade específica identificada no nosso *backlog*. Os desenvolvedores trabalham dentro destas *branches* para implementar a funcionalidade descrita nas histórias de utilizador, garantindo um mapeamento claro entre as alterações de código e os requisitos do utilizador.

Em termos de práticas de revisão de código, a nossa equipa enfatiza a minúcia e a colaboração. Os revisores fornecem *feedbacks* construtivos e sugestões de melhoria durante o processo de revisão de código. Utilizamos ferramentas como o *SonarCloud* para análise estática automatizada de código, o que ajuda a identificar potenciais problemas de qualidade do código logo no início do processo de revisão.

A nossa *Definition of Done* (DoD) para uma história de utilizador abrange vários critérios. Estes incluem a implementação bem-sucedida da funcionalidade, revisão completa do código, aprovação nos testes automatizados, integração sem problemas na *branch* principal e quaisquer atualizações de documentação necessárias.

Ao aderir a este fluxo de trabalho e práticas de revisão de código, mantemos um processo de desenvolvimento eficiente e colaborativo, focado na entrega de software de alta qualidade que atende às necessidades dos nossos utilizadores.

3.2 CI/CD pipeline and tools

Adotamos práticas robustas de *Continuous Integration* (CI) e *Continuous Deployment* (CD) para garantir a entrega consistente e de alta qualidade de incrementos de *software*. Utilizamos *GitHub Actions* como a nossa principal plataforma para CI/CD, integrando várias ferramentas e práticas para automatizar e otimizar o nosso processo de desenvolvimento.

***Continuous Integration* (CI)**

- **Criação de *Branches* e *Pull Requests*:** Desenvolvedores criam *branches* específicas para cada funcionalidade ou história de utilizador e submetem *Pull Requests* (PRs) para integrar essas mudanças na *branch* principal. Cada PR é sujeito a uma revisão obrigatória de outro membro da equipa, garantindo que as alterações são analisadas criticamente antes da integração.
- **Automação com *GitHub Actions*:** Cada vez que um PR é aberto, atualizado ou integrado na *branch* principal, o *GitHub Actions* é acionado para executar uma série de *pipelines* de CI.
- **Execução de Testes Automatizados:** O *pipeline* de CI executa uma bateria de testes automatizados, incluindo testes unitários (usando *frameworks* como *JUnit*), testes de integração (com ferramentas como *REST-Assured* ou *Spring MVC Test*), e análises estáticas de código (utilizando *SonarCloud*). Estas verificações automáticas garantem que o código se mantém estável e funcional após cada alteração.
- **Relatórios de Qualidade:** Após a execução dos testes, relatórios detalhados são gerados e disponibilizados para os desenvolvedores. Estes relatórios incluem o estado de cada teste, qualquer falha ocorrida e *insights* fornecidos pelo *SonarCloud* sobre a qualidade do código.

***Continuous Deployment* (CD)**

- **Construção e Distribuição de Artefactos:** Com cada integração bem-sucedida na *branch* principal, o *pipeline* de CD é acionado para construir e distribuir os artefactos de *software*. Utilizamos o *GitHub Actions* para orquestrar a construção de containers *Docker*, que encapsulam todas as dependências e configurações necessárias para a aplicação.

Configuração das Ferramentas

- ***GitHub Actions***: Configuramos diversos *workflows* no *GitHub Actions* que automatizam os *pipelines* de CI/CD. Estes *workflows* incluem etapas para *checkout* do código, configuração do ambiente de *build*, execução de testes, análise de qualidade do código e construção dos containers *Docker*.
- ***Docker*:** Containers *Docker* são configurados com base em *Dockerfiles* que definem todas as dependências e configurações necessárias para a aplicação. Utilizamos *Docker Compose* para orquestrar containers em ambientes de desenvolvimento.
- ***SonarCloud*:** Integrado no *pipeline* de CI, *SonarCloud* analisa o código em busca de problemas de qualidade e segurança, gerando relatórios que são revistos durante as revisões de código.
- **Ferramentas de Teste:** *JUnit*, *REST-Assured*, e *Spring MVC Test* são utilizados para implementar e executar testes unitários e de integração, assegurando a funcionalidade correta dos componentes individuais e da aplicação como um todo.

Ao integrar essas práticas e ferramentas, conseguimos manter um fluxo contínuo de desenvolvimento e entrega, assegurando que nossas aplicações são testadas, validadas e desdobradas com máxima eficiência e qualidade.

4 Software testing

4.1 Overall strategy for testing

Desenvolvemos uma estratégia de testes robusta que abrange várias metodologias e ferramentas para garantir uma cobertura de testes abrangente e manter a qualidade do código. No centro da nossa abordagem está o *Test Driven Development* (TDD), onde os desenvolvedores escrevem testes para novas funcionalidades ou alterações antes de implementar o código correspondente. Este método proativo assegura que o código é rigorosamente testado desde o início, promovendo melhores práticas de *design* e minimizando a ocorrência de *bugs*.

Além do TDD, adotámos os princípios do *Behavior Driven Development* (BDD), aproveitando ferramentas como o *Cucumber* para descrever e testar o comportamento do nosso *software* do ponto de vista do utilizador final. O *Cucumber* permite-nos escrever especificações executáveis em linguagem simples, promovendo a colaboração com os *stakeholders* e garantindo o alinhamento com os requisitos de negócio.

O nosso arsenal de testes é composto por uma mistura de diferentes ferramentas adaptadas para responder às várias necessidades de teste dentro do nosso projeto. Por exemplo, utilizamos o *REST-Assured* para testes de *API*, o *Selenium* para testes automatizados de *browser*, e o *JUnit* para testes unitários de código *Java*. Este conjunto diversificado de ferramentas permite-nos testar eficazmente diferentes camadas da nossa aplicação, garantindo que cada componente se comporta conforme esperado.

Além disso, a nossa estratégia de testes está estreitamente integrada no nosso processo de *Continuous Integration* (CI). Os testes automatizados são executados automaticamente após alterações de código, aproveitando plataformas de CI como o *GitHub Actions*. Esta integração perfeita fornece *feedback* rápido aos desenvolvedores e protege contra regressões, assegurando que as alterações de código mantêm o nível de qualidade esperado.

No nosso processo de CI, os resultados dos testes desempenham um papel crucial. Estabelecemos políticas e práticas para monitorizar e considerar meticulosamente os resultados dos testes antes de fundir alterações de código na *branch* principal. Os *pull requests* devem passar por uma bateria de testes automatizados, incluindo testes unitários, testes de integração e testes de aceitação, antes de serem aprovados para fusão. Qualquer falha ou regressão nos testes é prontamente investigada e resolvida pelos desenvolvedores.

No geral, a nossa estratégia de testes enfatiza testes proativos, colaboração e automação para entregar *software* de alta qualidade que atende às necessidades dos nossos utilizadores. Ao incorporar testes no nosso processo de desenvolvimento e *pipeline* de CI, garantimos que as alterações de código passam por uma validação rigorosa, mitigando o risco de defeitos e melhorando a confiabilidade geral do nosso *software*.

4.2 Functional testing/acceptance

A nossa abordagem aos testes funcionais gira em torno das histórias de utilizador, garantindo que cada cenário de teste se alinha estreitamente com as interações e fluxos de trabalho pretendidos dentro da aplicação. Estabelecemos uma política estruturada que prioriza a validação da funcionalidade do ponto de vista do utilizador, aproveitando vários recursos e práticas para alcançar este objetivo.

No cerne da nossa estratégia de testes funcionais está a definição de cenários de teste claros e abrangentes, derivados diretamente das histórias de utilizador. Estes cenários servem como base para os nossos testes, encapsulando os vários caminhos e comportamentos que os utilizadores devem seguir ao interagir com a aplicação. Ao focarmo-nos nas histórias de utilizador, garantimos que os nossos testes estão estreitamente alinhados com os requisitos e expectativas definidos pelos *stakeholders*.

Para implementar esses testes de forma eficaz, contamos com uma *framework* de automação que nos permite automatizar casos de teste repetitivos e executá-los de forma consistente. Esta *framework* agiliza o processo de teste, permitindo-nos validar a funcionalidade da aplicação de forma eficiente e fiável em diferentes ambientes.

A gestão de dados de teste é outro aspeto crucial da nossa estratégia de testes funcionais. Curamos cuidadosamente conjuntos de dados de teste que representam perfis de utilizador diversos, parâmetros de entrada e casos extremos, garantindo que os nossos testes refletem com precisão cenários do mundo real.

Esta abordagem aumenta a minúcia e a eficácia dos nossos esforços de teste, descobrindo potenciais problemas que os utilizadores podem encontrar em cenários de uso prático.

A integração no nosso pipeline de *Continuous Integration* (CI) assegura que os testes funcionais são executados automaticamente após alterações de código. Esta integração permite-nos detetar regressões cedo no processo de desenvolvimento, evitando a introdução de *bugs* e garantindo que novas funcionalidades ou alterações não comprometem a funcionalidade da aplicação.

Após cada ciclo de execução, geramos relatórios de teste detalhados que fornecem *insights* sobre o status de cada caso de teste e quaisquer falhas encontradas. Estes relatórios facilitam a colaboração entre os membros da equipa, permitindo que desenvolvedores, testadores e *stakeholders* identifiquem e resolvam problemas de forma rápida.

Incorporar testes funcionais na nossa estratégia de testes de regressão garante que a funcionalidade existente permanece intacta à medida que novas funcionalidades são introduzidas ou alterações são feitas. Ao priorizarmos as histórias de utilizador nos nossos esforços de testes funcionais, mantemos a qualidade e a fiabilidade do nosso *software*, entregando, em última análise, uma experiência de utilizador excecional.

4.3 Unit tests

Temos uma política clara e estruturada para a escrita de testes unitários, focada na validação de componentes individuais do nosso *software* do ponto de vista do desenvolvedor. Esses testes são projetados para garantir que cada unidade de código se comporte conforme esperado e atenda aos requisitos especificados, contribuindo para a qualidade e a manutenibilidade geral do nosso código.

A nossa abordagem aos testes unitários abrange vários elementos-chave. Em primeiro lugar, definimos um nível alvo de cobertura de testes para o nosso código, fornecendo uma diretriz para os desenvolvedores garantirem que os componentes críticos e casos extremos estejam adequadamente cobertos por testes.

Também defendemos práticas de *Test Driven Development* (TDD), encorajando os desenvolvedores a escreverem testes unitários antes de implementar o código correspondente. Esta abordagem promove um melhor *design*, aumenta a manutenibilidade do código e garante testes rigorosos desde o início do desenvolvimento.

Para isolar unidades de código para testes, empregamos técnicas como *mocking* para simular o comportamento das dependências. Isso permite-nos testar componentes de forma isolada, sem depender de dependências externas ou infraestrutura, facilitando esforços de teste mais rápidos e focados.

Utilizamos *frameworks* e ferramentas de teste padrão da indústria, como o *JUnit*, para facilitar a escrita e a execução de testes unitários. Estas *frameworks* fornecem funcionalidades essenciais para definir casos de teste, executar testes e relatar resultados de testes.

Os testes unitários são integrados no nosso *pipeline* de *Continuous Integration* (CI), onde são executados automaticamente após alterações de código. Esta integração garante que quaisquer mudanças introduzidas pelos desenvolvedores não quebrem a funcionalidade existente e que o código permaneça estável e fácil de manter ao longo do tempo.

Durante as revisões de código, os testes unitários desempenham um papel crucial, com os desenvolvedores revisando os testes uns dos outros para garantir uma cobertura adequada da funcionalidade pretendida e casos extremos. Esta abordagem colaborativa promove a partilha de conhecimento e garante a robustez e eficácia dos nossos testes.

Os testes unitários fornecem suporte valioso durante o processo de refatoração, atuando como uma rede de segurança para identificar e resolver quaisquer regressões introduzidas pelas mudanças de código. Ao aderir a estas práticas, garantimos que os nossos esforços de testes unitários são sistemáticos, eficazes e contribuem significativamente para a qualidade e manutenibilidade geral do nosso código.

4.4 System and integration testing

No nosso projeto, temos uma estratégia clara para testes de sistema e integração, com um foco especial nos testes de *API*. Os desenvolvedores criam e mantêm testes de integração juntamente com suas alterações de código, garantindo uma validação completa da funcionalidade das *APIs*. Utilizamos *frameworks* de automação como o *REST-Assured* ou o *Spring MVC Test* para automatizar a execução dos testes, assegurando consistência e eficiência.

Os dados de teste são geridos cuidadosamente para refletir cenários do mundo real. Esta gestão cuidadosa garante que os testes abordem uma ampla gama de casos de uso, aumentando a eficácia dos testes de integração.

Os testes de integração são integrados de forma contínua no nosso *pipeline* de *Continuous Integration* (CI), sendo executados automaticamente após alterações no código. Esta integração contínua permite a deteção precoce de regressões e problemas de integração, garantindo que o sistema permaneça estável e funcional ao longo do tempo.

Relatórios detalhados fornecem *insights* sobre o estado dos testes e falhas, permitindo a resolução rápida de problemas. Estes relatórios são essenciais para a colaboração entre os desenvolvedores e para a manutenção de altos padrões de qualidade.

Esta abordagem garante a fiabilidade e o desempenho das nossas *APIs*, proporcionando uma experiência fluida e consistente para os utilizadores. Ao aderir a estas práticas, asseguramos que nossas *APIs* são robustas, eficientes e prontas para enfrentar os desafios do uso real.