TQS: Quality Assurance manual

Gonçalo Abrantes[104152] João Morais [103730] Pedro Rei [107463]
v2025-06-06

Contents

1 Project management

1.1 Assigned roles

- Team Coordinator e Product Owner(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.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 (Gonçalo Abrantes): 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 (João Morais): 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.

1.2 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.

2 Code quality management

2.1 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.

2.2 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.

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.

2.3 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

3 Continuous delivery pipeline (CI/CD)

3.1 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.

3.2 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.

3.3 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.

4 Software testing

4.1 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.

4.2 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.

4.3 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.

4.4 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.

4.5 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 realizar testes de carga HTTP e validar os objetivos de desempenho dos nossos serviços. Os testes simulam o fluxo de reserva de veículos, envolvendo múltiplos utilizadores virtuais (VUs) em cenários progressivos.

O script de teste reproduz uma jornada típica do utilizador:

- Carregamento de dados (usuários e estações);
- Consulta de carros disponíveis para o usuário;
- Verificação de reservas ativas;
- Criação e finalização de uma reserva de veículo.

O teste implementa lógica de retry e pausas realistas entre ações, aproximando-se de um comportamento de usuário real.

Configuração do Cenário (`options`)

Utilizamos o `ramping-vus` para escalar gradualmente os usuários virtuais:

- Subida para 50 VUs em 30 segundos;
- Escalonamento até 100 VUs por 1 minuto;
- Redução para 0 VUs em 30 segundos.

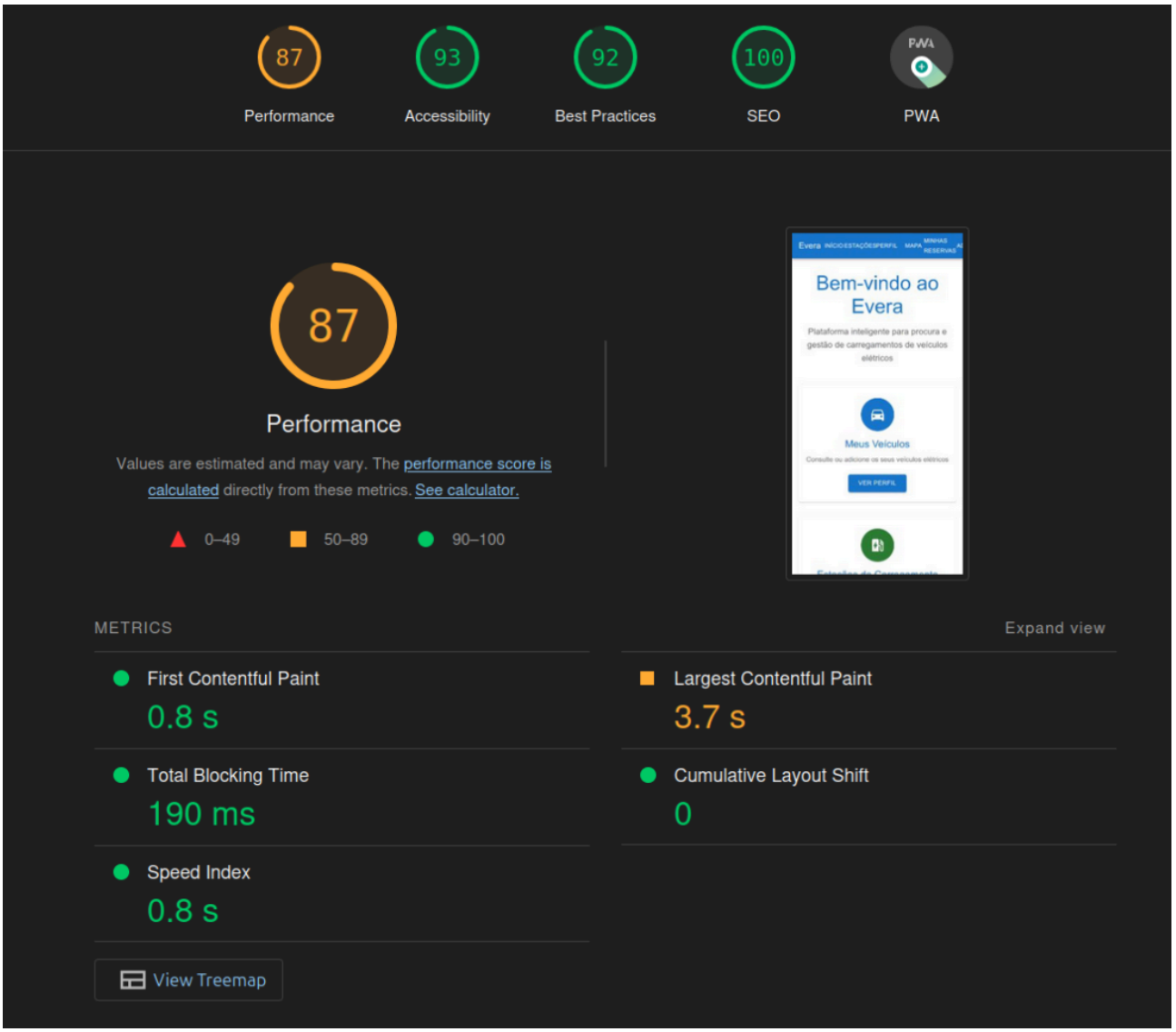
Métricas Monitoradas

O comportamento do sistema é analisado com base em:

- Tempo médio e máximo de resposta;
- Taxa de erro por endpoint;
- Volume total de tráfego gerado;
- Sucesso na criação e finalização de reservas.

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.


87
Performance
93
Accessibility
92
Best Practices
100
SEO
PWA
87
Performance
Values are estimated and may vary. The performance score is calculated directly from these metrics. See calculator.
0–49
50–89
90–100
Bem-vindo ao Evera
Meus Veículos
METRICS
Expand view
First Contentful Paint
0.8 s
Largest Contentful Paint
3.7 s
Total Blocking Time
190 ms
Cumulative Layout Shift
0
Speed Index
0.8 s
View Treemap