TQS: Quality Assurance manual

Diogo Gaitas [73259], Giovanni Santos [115637], Rafael Semedo [115665]

v2025-06-09

Contents

1 Project management

1.1 Assigned roles

Role	**Student**
Team Coordinator	Rafael Semedo
Product owner	Diogo Gaitas

QA Engineer	Giovanni Santos
Developer	Diogo Gaitas, Giovanni Santos, Rafael Semedo

1.2 Backlog grooming and progress monitoring

No nosso projeto, organizamos o trabalho no JIRA estruturando Epics para cada grande área funcional (Discovery, Reservation, Charging, Payments, Reporting e Analytics) e criar User Stories detalhadas sobre esses Epics, com critérios de aceitação claros e estimativas em story points. Utilizamos um board Scrum com sprints semanais, cujas colunas padrão (Backlog, To Do, In Progress, In Review e Done) refletem o fluxo de trabalho desde o planejamento até a entrega. Para manter tudo facilmente filtrável e rastreável, aplicamos labels que identificam o tipo de tarefa (por exemplo, epic/discovery, story/charging, bugfix). Cada entrega na main é registrada como uma Release no JIRA, vinculada às User Stories e Epics concluídos naquele ciclo.

O progresso da equipe é monitorado diariamente e ao longo de cada sprint por meio de métricas ágeis tradicionais. As User Stories são estimadas em planning poker usando uma escala de story points (1, 2, 3, 5, 8), levando em conta esforço, riscos e dependências. Acompanhamos o desempenho por meio do gráfico de burndown, que compara o trabalho planejado versus o trabalho restante, e do cumulative flow diagram, que revela gargalos ao mostrar quantas tarefas estão em cada coluna do board.

Para garantir que todo requisito tenha cobertura de testes, integramos o JIRA com o Xray. Cada User Story possui um conjunto de casos de teste associados, documentando tanto os fluxos felizes quanto os cenários alternativos. À medida que os testes unitários, de integração e end-to-end são executados, seus resultados são automaticamente vinculados às histórias correspondentes, fornecendo uma visão em tempo real da cobertura de testes por requisito. Criamos dashboards no JIRA que exibem o percentual de User Stories cobertas por testes automatizados, bem como métricas de qualidade do código vindas do SonarQube. Além disso, estabelecemos gates que impeçam o fechamento de uma história caso ela não tenha ao menos um caso de teste associado ou esteja abaixo do threshold mínimo de 80% de cobertura, garantindo assim um monitoramento proativo da qualidade desde as fases iniciais do desenvolvimento.

2 Code quality management

2.1 Team policy for the use of generative AI

A IA (ex: **Copilot**, **ChatGPT**, etc.) pode ser usada como apoio, mas não substitui compreensão e revisão humanas.

Do’s:

- Usar IA para:
    - Sugerir esqueleto de classes, métodos, testes unitários ou mocks
    - Explicar trechos de código
    - Gerar templates de documentação
- Validar com leitura crítica e testes antes de aceitar o código

Don’ts:

- Copiar código gerado sem entender o que faz
- Usar IA para escrever código sensível sem revisão humana
- Ignorar os testes sugeridos ou gerados automaticamente

2.2 Guidelines for contributors

Coding style

Adotamos um estilo de código consistente com as práticas recomendadas para projetos Spring Boot em Java. Não exigimos uma definição exaustiva, mas seguimos os princípios principais abaixo:

Pontos-chave do estilo adotado:

- Padrão de nomenclatura:
    - Classes: CamelCase (ex: ChargingStationController)
    - Variáveis e métodos: camelCase (ex: findNearbyStations)
- Organização de pacotes: controller, service, repository, model, config
- Separação clara de camadas e responsabilidades (SRP – Single Responsibility Principle)
- Evitar métodos longos e lógicas duplicadas
- Uso obrigatório de @Override quando necessário

Referência externa:

Android Open Source Project Java Style Guide (AOS)

Code reviewing

Todas as alterações devem passar por **Pull Request (PR)** no GitHub. Isso é obrigatório para manter a qualidade e consistência do código.

Regras para revisão de código:

- Um PR só pode ser *merge* após:
    - Aprovação por pelo menos **2 colega**
    - Todos os **checks de CI estarem verdes**
    - Passagem no **Quality Gate** (SonarQube)
- Commits devem ser pequenos, focados e com mensagens claras

- Reviewers devem observar:
    - Clareza e simplicidade do código
    - Cobertura de testes
    - Estilo e estrutura
    - Nomes significativos
    - Performance e legibilidade

Uso de AI (Copilot, ChatGPT) em PRs:

- Pode-se usar AI para sugerir código ou revisão inicial
- Todo código gerado por AI deve ser **entendido e testado** antes de ser aceito
- A equipe incentiva usar AI para:
    - Sugerir nomes melhores
    - Gerar testes de unidade e integração
    - Explicar código legado

2.3 Code quality metrics and dashboards

Ferramentas e práticas adotadas

Durante o desenvolvimento do projeto, definimos uma pipeline de qualidade contínua que roda automaticamente a cada push ou pull request. As ferramentas utilizadas incluem:

- **SonarQube**: para análise estática de código (identificação de bugs, code smells, duplicações e vulnerabilidades).
- **JaCoCo**: para análise da cobertura de testes automatizados (unitários e de integração).
- **GitHub Actions**: como orquestrador da pipeline de CI/CD, garantindo que todas as validações ocorram antes do merge.
- **Build Breaker via Quality Gate**: bloqueia merges quando as métricas mínimas de qualidade não são atingidas.

Quality Gates definidos

Para manter o padrão de qualidade do código, estabelecemos regras claras que devem ser satisfeitas em cada PR:

1. **Cobertura de testes (JaCoCo)**: mínimo de 80% de cobertura .
2. **Bugs críticos (SonarQube)**: nenhum bug classificado como “blocker” ou “critical” pode existir.
3. **Code Smells**: aceitamos pequenos code smells em desenvolvimento inicial, mas incentivamos que o número seja sempre reduzido e justificado em revisões.
4. **Duplicação de código**: o código duplicado deve ser mantido abaixo de 5%.
5. **Vulnerabilidades**: nenhuma vulnerabilidade de segurança com severidade alta é permitida.

Esses critérios estão configurados no painel do SonarQube como parte de um **Quality Gate**, aplicado automaticamente a cada build. Caso qualquer métrica viole os critérios definidos, o merge no GitHub é bloqueado até que as correções sejam feitas.

Dashboards e relatórios disponíveis

- **Painel do SonarQube**: apresenta métricas em tempo real, com visualização por arquivos, classes e pacotes. Utilizado para acompanhamento contínuo da saúde do código.
- **Relatório do JaCoCo (HTML)**: mostra cobertura de testes detalhada, destacando linhas cobertas e não cobertas.
- **GitHub Actions**: fornece logs e status de execução dos testes, builds e integração com o SonarQube.
- **Integração com PRs**: todo Pull Request exibe automaticamente os resultados do CI e da análise de qualidade, permitindo decisões informadas antes do merge.

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

Coding workflow

Nosso time segue um fluxo baseado no **GitFlow**, adaptado para projetos Agile com iteração semanal, integrado com **Jira** e **Xray** para mapear os user stories. O processo é:

1. **Seleção de uma User Story**:
    - O desenvolvedor escolhe uma user story do *current sprint backlog* (no Jira).
    - Cada story tem pontos de esforço, critérios de aceitação e é rastreada por um ID (ex: *CMR-13*).
2. **Criação de uma branch**:
    - O nome da branch segue o padrão: *feature/CMR-13-feature-name*
    - Associada à história/task no Jira.
3. **Implementação e testes locais**:
    - Desenvolvimento de funcionalidades + testes unitários.
    - Testes unitários e de integração criados conforme critérios de aceitação.
4. **Pull Request (PR) e Code Review**:
    - Todo código passa por revisão de pelo menos um colega.
    - O PR é verificado com base em:
        - Cobertura de testes (JaCoCo)
        - Análise estática (SonarQube),
        - Status de CI Build.
5. **Merge para as branch *main* ou *dev*:**
    - Apenas com aprovação + todos os checks passando.

Definition of done

Uma user story é considerada *done* quando:

- Código implementado na branch *feature/**
- Cobertura de testes mínima: **80%** para lógica de negócio
- Aceitação dos critérios funcionais (em Jira)
- Código revisado e PR aprovado
- Análise estática (SonarQube) sem falhas críticas
- Build e testes passaram na pipeline CI
- Cenários de teste definidos no Xray concluídos
- Deploy automático no ambiente de staging

3.2 CI/CD pipeline and tools

Continuous Integration (CI)

A pipeline de **CI** é automatizada com **GitHub Actions**, executada na abertura de cada pull request e push:

Etapas:

- Build do projeto com Maven
- Execução de testes com JUnit
- Verificação de cobertura com JaCoCo
- Análise de qualidade com SonarQube
- Geração de artefatos .jar
- Publicação de status no PR

O merge é bloqueado caso falhe algum estágio

Continuous Delivery (CD)

O CD é integrado com containers aumentando a velocidade e capacitando a automatização segura e consistente.

- Conteinerização com Docker
- Deploy automático para ambiente staging ao merge na *main*
- Release tags disparam build de produção

3.3 System observability

Logs estruturados:

- Logback padronizado e configurado com logs separados por serviço, com nível DEBUG/INFO/ERROR.

Alertas:

- CI falhando envia alerta via GitHub Actions (e-mail)

4 Software testing

4.1 Overall testing strategy

Nossa estratégia de testes combina várias abordagens, integradas de forma contínua à pipeline de CI:

- Aplicarmos quando for possível **TDD (Test-Driven Development)** em funcionalidades centrais, especialmente nas camadas de serviço e lógica de negócio.
- Utilizamos **JUnit 5** para testes unitários e de integração.
- Utilizamos **MockMVC + TestContainers** para testes de APIs REST, focando em cobertura de endpoints.
- Nossa estratégia de testes foi aprimorada com a adoção de testes baseados em comportamento (BDD) utilizando o **Cucumber**, permitindo validar os principais fluxos funcionais do sistema, como criação e cancelamento de reservas, processamento de pagamentos, busca de estações com filtros e marcação de manutenção, garantindo maior confiabilidade e alinhamento com os requisitos do negócio.
    - Os testes são organizados por Feature e escritos com clareza usando o formato Gherkin (Dado/Quando/Então), garantindo alinhamento entre a perspetiva do utilizador e a lógica de negócio.

Todos os testes são executados automaticamente via **GitHub Actions** em cada commit ou pull request. A cobertura de testes é monitorada com **JaCoCo**, e os resultados são considerados no **Quality Gate** do **SonarQube**, impedindo o merge de código que não passe pelos testes definidos.

4.2 Functional testing and ATDD

Além dos testes unitários e de integração, adotamos uma estratégia robusta de **ATDD (Acceptance Test Driven Development)** com Cucumber + Spring Boot Test. Cada User Story funcional classificada como crítica para o MVP possui cenários automatizados escritos em Gherkin, validados com steps em Java e integração com o contexto da aplicação via *@SpringBootTest.*

Principais features cobertas:

- Reserva de slots com múltiplos cenários (válido, inválido, indisponível)
- Cancelamento de reserva com liberação do slot
- Pagamento (Pay-per-use) vinculado à reserva
- Busca de estações por tipo e disponibilidade
- Marcação de estação como "em manutenção" por admin

Políticas adotadas:

- Todo requisito com impacto funcional precisa ter pelo menos um cenário de aceitação automatizado.
- Esses testes são versionados junto ao código e integrados ao Jira/Xray como "Test Cases".
- São exigidos antes do merge de histórias que envolvam interações de alto nível.

4.3 Developer facing tests (unit, integration)

Unit Tests

- Escrevemos testes unitários para todos os métodos com lógica de negócio.
- Usamos **JUnit 5** com **Mockito** para simulações.
- Um PR não é aceito se não houver testes unitários para novos métodos de serviço ou classes utilitárias.

Exemplos relevantes:

- Verificação de disponibilidade de vagas (hasAvailableSlots)
- Cálculo de distância geográfica (Haversine)
- Conversão de DTOs

Integration Tests

- Escrevemos testes de integração para **verificar a comunicação entre camadas (Controller ↔ Service ↔ Repository)**.
- Utilizamos o **Spring Boot Test com banco em memória (H2)**.

Os testes de integração são identificados por @SpringBootTest e segregados dos unitários para evitar execução desnecessária em todos os builds.

4.4 Exploratory testing

Adotamos uma abordagem complementar de **testes exploratórios** durante as fases finais de cada sprint. Essas sessões seguem o seguinte modelo:

- Um membro da equipe explora livremente o sistema com base em cenários não formalizados (ex: uso incorreto de parâmetros, simulação de falhas de rede).

- Defeitos encontrados são documentados no Jira e, se relevantes, transformados em cenários automatizados.

- Usamos o Postman para simular chamadas com entradas incomuns, repetidas ou inválidas.

Esses testes ajudaram a detectar falhas não cobertas por testes sistemáticos, como casos de null, formatos inválidos e headers ausentes.

4.5 Non-function and architecture attributes testing

Até à data da entrega do MVP não foram ainda implementados testes de performance automatizados. Optamos por priorizar cobertura funcional (unit, integration, BDD) e garantir qualidade de código antes de investir em ensaios de carga. A aplicação ainda está em fase de validação funcional; cargas de produção reais só serão simuladas após estabilização dos fluxos principais.

A política seguinte garante que os ensaios de performance sejam introduzidos de forma incremental, alinhados com os objetivos de escalabilidade.

Política para introduzir testes de performance

Para garantir a performance e confiabilidade do sistema, adotamos o k6 (open-source) para execução de testes de carga HTTP, integrado ao GitHub Actions, com foco nos endpoints críticos: POST /reservations, PUT /charging-sessions/{id} e GET /stations/search. Nossos SLOs incluem latência P95 abaixo de 300ms, taxa de erro inferior a 1% sob 100 RPS e throughput sustentado mínimo de 200 RPS. Implementamos testes de carga leves (smoke) a cada merge na branch de desenvolvimento e ensaios de stress semanais em jobs dedicados, configurando a pipeline para falhar automaticamente caso qualquer SLO seja violado. Além disso, as métricas do k6 são exportadas para o Grafana Cloud, com alertas de falha enviados via Slack/e-mail para monitoramento proativo.

Até essa implementação, eventuais problemas de performance são avaliados **manual/iterativamente** através de:

- **Spring Boot Actuator** (métricas HTTP e uso de memória)
- **VisualVM** / **JProfiler** para análise ad-hoc em ambiente local.