PlugPoint - QA Manual

Diogo Oliveira [113 664],

Bruno Charão [111 590],

Hugo Sousa [112 733],

Guilherme Mesquita [112 957]

1. Project Management

1.1 Team and Roles

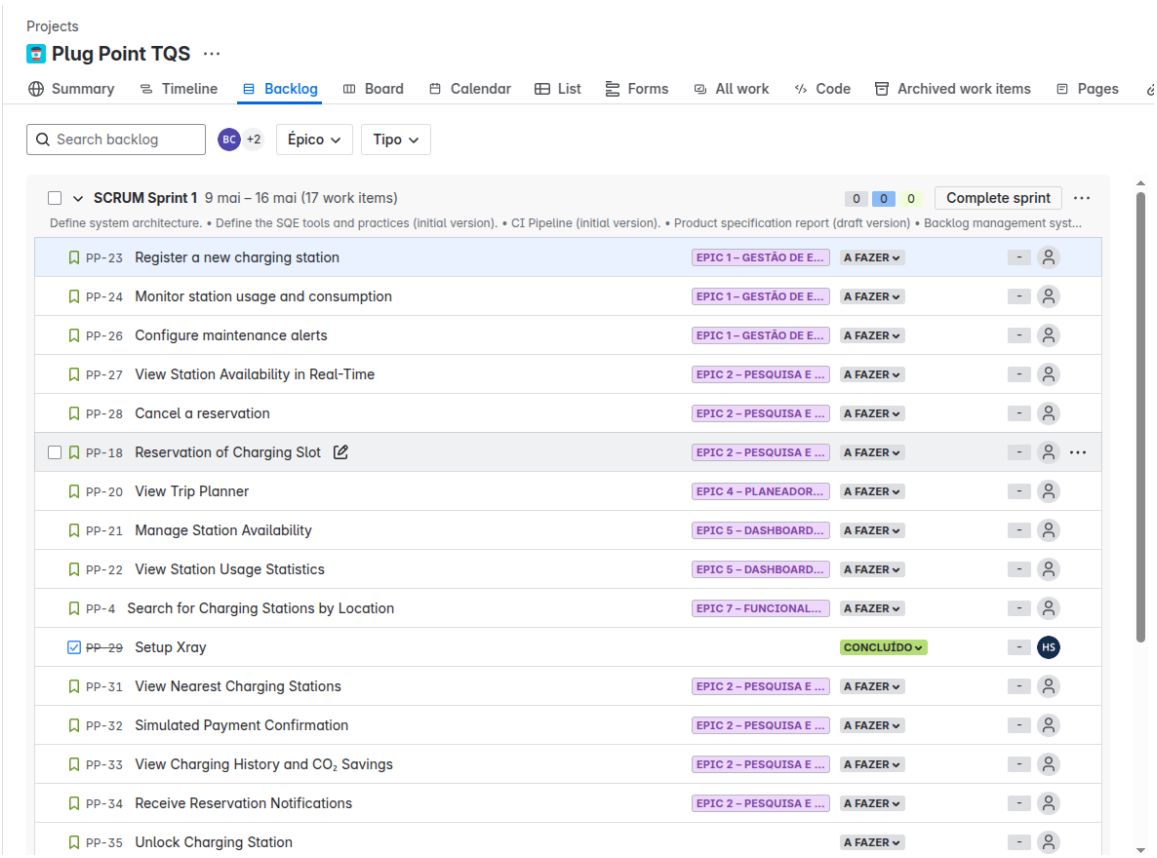
Membro	Papel	Responsabilidades
Diogo Oliveira	Team Coordinator	Coordenação geral do projeto, gestão de tarefas semanais, organização de reuniões, acompanhamento do progresso e mediação de conflitos.
Bruno Charão	QA Engineer	Definição e aplicação da estratégia de qualidade e automação de testes.
Hugo Sousa	DevOps Master	Criação e manutenção da infraestrutura de desenvolvimento e produção, configuração de Docker, CI/CD e deploy com GitHub Actions.
Guilherme Mesquita	Product Owner	Representa os interesses dos utilizadores. Define requisitos, mantém o backlog em Jira e valida funcionalidades entregues.

1.2 Agile Backlog Management and Work Assignment

Usamos Jira como ferramenta central para gestão do backlog, com integração ao GitHub. Todas as user stories incluem critérios de aceitação escritos em Gherkin.

O fluxo de trabalho segue as seguintes etapas:

1. Criação e priorização de épicos e user stories.
2. Criação de branches com nomeação padronizada (`feature/<id>-descricao`) a partir de Jira.
3. Desenvolvimento com abordagem TDD sempre que possível.
4. Testes automatizados com JUnit e/ou Cucumber.
5. Pull Requests obrigatórias com revisão por pelo menos um membro.
6. CI executado via GitHub Actions.
7. Merge em `develop` e deploy para staging com Docker Compose.



2. Code Quality Management

SonarCloud é usado para análise contínua da qualidade do código. Ele está integrado com GitHub Actions e aplica um Quality Gate com:

- Cobertura mínima de 80% (via JaCoCo)
- Nenhum bug ou vulnerability em aberto
- Limite de code smells críticos

Usamos os plugins JaCoCo para cobertura.

3. Continuous Delivery Pipeline (CI/CD)

3.1 Development Workflow

Trabalhamos com um modelo GitFlow simplificado:

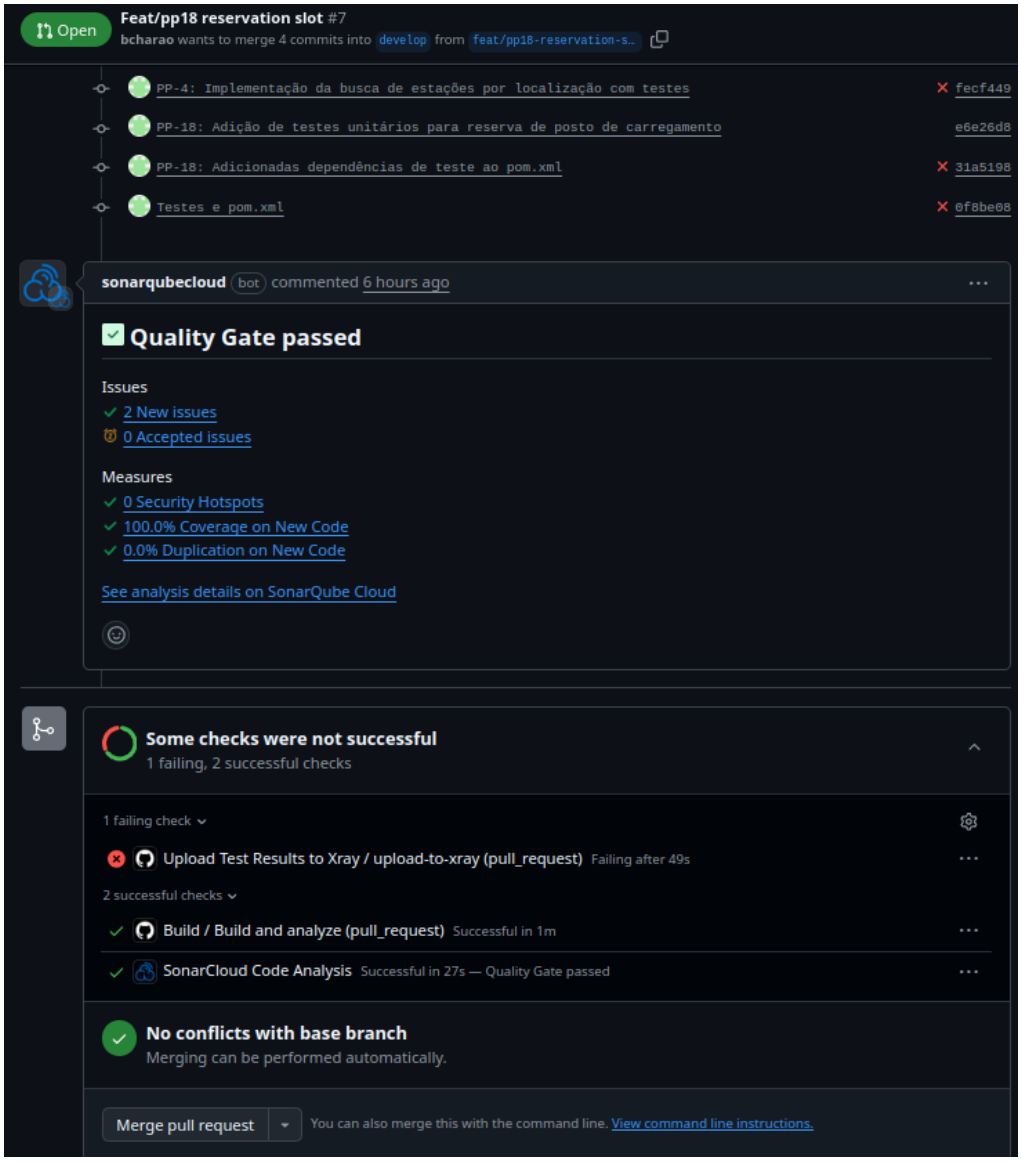
- `main`: produção
- `develop`: desenvolvimento
- `feature/<id>`: novas funcionalidades

Pull Requests exigem revisão, CI verde, e cumprimento de critérios de aceitação. O deploy para staging é feito via Docker Compose.

3.2 CI/CD Pipeline and Tools

CI é implementado com GitHub Actions, com pipeline que executa:
- Build com Maven (`mvn clean verify`)
- Testes unitários e de integração
- Análise de qualidade com SonarCloud

Ferramentas: GitHub Actions, Docker, Maven, SonarCloud.



3.3 System Observability

Planeamos monitorar os serviços com Prometheus e Grafana.
- Dashboards para tempo de resposta, falhas e consumo de recursos.

4. Software Testing

4.1 Overall Strategy for Testing

Adotamos abordagem TDD sempre que possível. Cada módulo é coberto por testes unitários e de integração. Estratégia futura inclui testes BDD automatizados e E2E com Cypress.

Camadas testadas:
- Serviços (lógica de negócio)
- Controllers (API REST)
- Integração com base de dados (Testcontainers)

4.2 Functional Testing / Acceptance

Critérios de aceitação estão descritos em formato Gherkin.

Usaremos Cucumber + JUnit para automação de testes BDD, cobrindo casos como:
- Reserva de slot com horário válido
- Notificação antes do carregamento

4.3 Unit Tests

Frameworks: JUnit 5, Mockito, AssertJ.

Testamos:
- Serviços: lógica e validações
- Controllers: resposta HTTP
- Repositórios: comportamento com dados reais (via Testcontainers)

4.4 System and Integration Testing

Usamos Testcontainers para integração com PostgreSQL.

Testes validam:
- APIs REST com MockMvc
- Fluxos entre controller, serviço e repositório

4.5 Performance Testing

Planeamos testes com K6 a partir da iteração 4.

Endpoints visados:
- /stations (pesquisa)
- /bookings (reserva)

Métricas alvo: latência <500ms (P95), 0% erro sob 50 req/s.

4.6 End-to-End Testing

Automação E2E será feita com Selenium.

Testes simulam:
- Pesquisa de estações
- Reserva de slot

Testes rodam em CI e produzem relatórios em HTML.

4.7 API Testing (Blackbox)

Testes de API são realizados com Postman + Newman.

Coleções testam:
- Status HTTP
- JSON válido
- Fluxo completo de reserva

Executados via GitHub Actions em push para develop.

5. Traceability and Coverage

5.1 Requisitos e Rastreabilidade

Integraremos Jira com Xray para garantir rastreabilidade entre user stories e testes.
Cada critério de aceitação será ligado a testes JUnit, Cucumber ou Cypress,
permitindo rastrear requisitos não cobertos.

6. Architecture Implications on Quality

A arquitetura modular da aplicação, composta por múltiplos containers (frontend, backend e base de dados PostgreSQL), exige uma abordagem de qualidade distribuída. A integração com serviços externos (pagamentos e geolocalização) justifica a adoção de testes de integração robustos e testes end-to-end para cobrir o comportamento real do utilizador.

Camadas bem definidas no backend (REST Controller, Service, Repository, API Clients) permitem:

- Testes unitários isolados por camada;
- Mocks de dependências com Mockito;
- Testes de API blackbox com Postman + Newman;
- Testes de integração com base de dados real via Testcontainers.

Além disso, o uso de Docker garante:

- Ambientes consistentes para testes automatizados;
- Execução de CI/CD com containers independentes;
- Facilidade na simulação de ambientes para testes de carga (K6) e falhas externas.