TQS: Quality Assurance manual

1 Project management

1.1 Team and roles

Role	Name	Email	Nmec
Team Manager	Airton Moreira	Agm@ua.pt	100480
Product Owner	João Diogo Craveiro Ferreira	jd.ferreira@ua.pt	95316
QA Engineer	João Diogo Craveiro Ferreira	jd.ferreira@ua.pt	95316
DevOps Master	Airton Moreira	Agm@ua.pt	100480

1.2 Agile backlog management and work assignment

No âmbito do projeto *ElectraNet*, adotamos uma metodologia ágil baseada em sprints semanais, com a gestão do backlog centralizada no Jira. No início de cada sprint, a equipa define colaborativamente as *user stories* a desenvolver, sendo posteriormente distribuídas pelo Gestor de Equipa de forma equilibrada.

Cada *user story* é desdobrada em subtarefas específicas de desenvolvimento e testes, assegurando uma implementação estruturada e orientada para a qualidade. A integração com Xray permite-nos monitorizar automaticamente os resultados dos testes, mantendo a rastreabilidade e visibilidade no Jira.

// TO DO --- Put jira user stories coverage here (pics)

//

2. Code quality management

2.1 Guidelines for contributors (coding style)

Todas as contribuições para o repositório de código do projeto *ElectraNet* devem seguir as seguintes diretrizes:

- Os nomes das classes devem utilizar PascalCase (por exemplo: HelloWorld.java);
- Os nomes dos *packages* devem ser declarados no início do ficheiro, em minúsculas e sem números;
- Devem ser utilizados *imports* específicos, evitando o uso de *;
- A declaração de variáveis deve seguir a convenção camelCase (por exemplo: someVariable);
- O código deve seguir uma formatação consistente, recorrendo à ferramenta Prettier;
- O âmbito das variáveis deve ser o mais restrito possível;
- Utilização do Lombok para geração automática de construtores, *getters*, *setters* e métodos toString;
- O código deve ser devidamente comentado, de forma a garantir clareza e facilidade de manutenção;
- Todas as contribuições devem incluir testes automatizados utilizando o framework JUnit;
- A análise contínua da qualidade do código é realizada com recurso ao SonarQube integrado no IntelliJ IDEA, promovendo a deteção precoce de *code smells* e potenciais vulnerabilidades.

2.2 Code quality metrics and dashboards

A análise estática do código no projeto *ElectraNet* é realizada através do SonarQube, utilizando a *Quality Gate* padrão "Sonar Way", a qual se revelou adequada para os nossos objetivos de qualidade sem necessidade de personalização adicional.

Esta *Quality Gate* impõe os seguintes critérios, que devem ser cumpridos para validação do código:

- Ausência de novos *bugs*;
- Ausência de novas vulnerabilidades;
- Dívida técnica limitada no novo código;
- Todos os *security hotspots* devem ser revistos;
- Cobertura de testes igual ou superior a 80%;
- Percentagem de linhas duplicadas inferior ou igual a 3%.

A análise de qualidade é automaticamente desencadeada sempre que é efetuado um *pull request* ou um *push* para a *main branch*, sendo os resultados publicados no painel do SonarQube integrado no IntelliJ IDEA. Este processo garante visibilidade contínua sobre a saúde do código e facilita a deteção precoce de problemas.

3. Continuous delivery pipeline (CI/CD)

3.1 Development workflow

No projeto *ElectraNet*, adotamos o fluxo de trabalho **GitHub Flow** para o desenvolvimento. Para cada tarefa registada na plataforma Jira, é criada uma *branch* dedicada no repositório Git. Após a conclusão da implementação, é submetido um *Pull Request* para a *main branch* do respetivo submódulo.

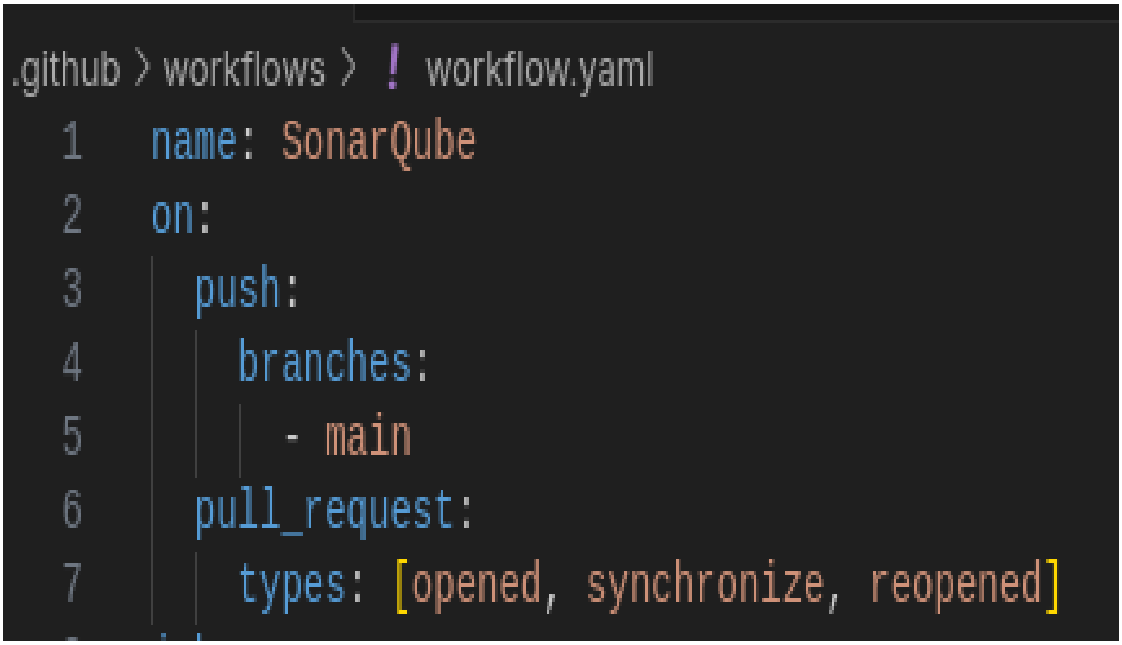
Este *Pull Request* deve ser revisto por pelo menos um outro membro da equipa, garantindo a conformidade com os padrões de qualidade definidos e a aprovação na análise automática de código efetuada pelo SonarQube.

3.2 CI/CD pipeline and tools

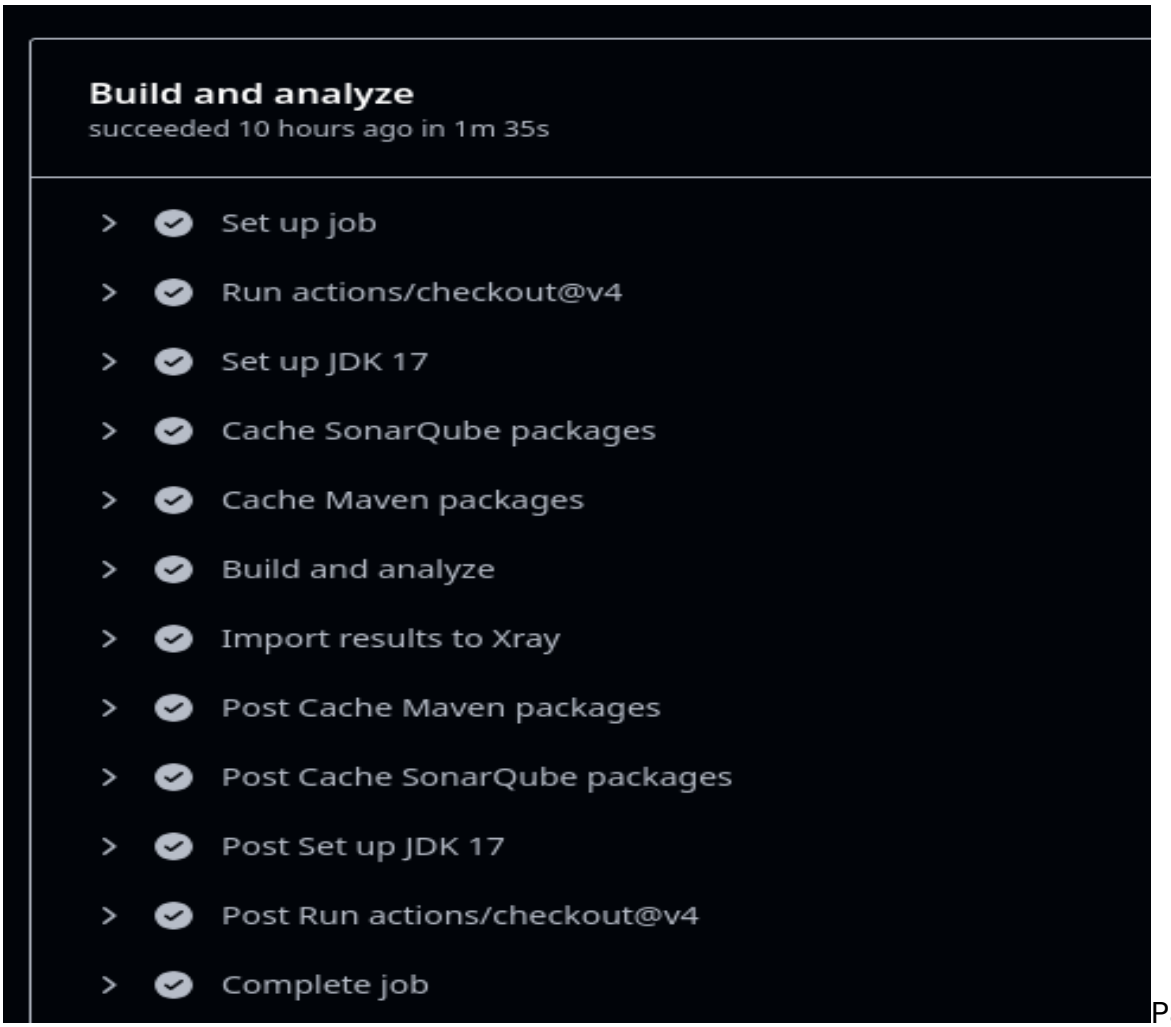
A pipeline de Integração Contínua (CI) implementada no projeto *ElectraNet* visa detetar precocemente erros e garantir a estabilidade contínua da aplicação ao longo do seu desenvolvimento.

Recorremos ao **GitHub Actions** para automatizar este processo. A cada *commit* ou *pull request* realizado no repositório principal, o código é automaticamente testado e validado, assegurando que novas alterações não comprometem a integridade do sistema.

A arquitetura do projeto assenta na **containerização** dos seus módulos, o que permite o desenvolvimento e validação de componentes de forma isolada. Esta abordagem favorece a escalabilidade e prepara o projeto para uma futura implementação de uma pipeline de Entrega Contínua (CD), permitindo testar e atualizar serviços específicos sem afetar o sistema global.




```
  SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
run: mvn -B verify org.sonarsource.scanner.maven:sonar-maven-plugin:sonar -Dsonar.projectKey=APG2000_TqsTeam2025
```



Para além da execução automática de testes em cada *pull request* ou *push*, o projeto *ElectraNet* integra uma **GitHub Action** que permite o envio dos resultados dos testes automatizados para o **Jira**, a nossa plataforma de gestão de tarefas. Esta integração reforça a rastreabilidade entre o código desenvolvido e os requisitos definidos, facilitando o acompanhamento da qualidade e do progresso das *user stories*.

```yaml
- name: Import results to Xray
  uses: mikepenz/xray-action@v3
  with:
    username: ${{ secrets.XRAY_CLIENT_ID }}
    password: ${{ secrets.XRAY_CLIENT_SECRET }}
    testFormat: "junit"
    testPaths: "server/eletranet/backend/target/surefire-reports/TEST-*.xml"
    testExecKey: "SCRUM-48"
    projectKey: "SCRUM"
```

Também foi desenvolvida uma GitHub Action para permitir implementações automáticas.
Esta action combina imagens Docker e runners self-hosted do GitHub para fazer o deployment automático do projeto na máquina virtual indicada.
Sempre que ocorre um push para o repositório principal, esta action é ativada seguindo o seguinte processo:
Cada imagem Docker é construída e depois enviada para o Docker Hub.

```yaml
name: Publish Docker images
on:
  push:
    branches: ['main']

permissions:
  contents: read
  deployments: write
  statuses: write


jobs:
  build-backend-image:
    name: Push the Backend Docker image to Docker Hub
    runs-on: ubuntu-latest
    steps:
      - name: Check out the repo
        uses: actions/checkout@v4
        with:
          submodules: recursive

      - name: Log in to Docker Hub
        uses: docker/login-action@f4ef78c080cd8ba55a85445d5b36e214a81df20a
        with:
          username: ${{ secrets.DOCKER_USERNAME }}
          password: ${{ secrets.DOCKER_PASSWORD }}

      - name: Extract metadata (tags, labels) for Docker
        id: meta
        uses: docker/metadata-action@v4
        with:
          images: apg2000/eletranet-backend

      - name: Build and push Docker image
        uses: docker/build-push-action@v5
        with:
          context: ./server/eletranet/backend
          file: ./server/eletranet/backend/Dockerfile
          push: true
          tags: apg2000/eletranet-backend:latest
          labels: ${{ steps.meta.outputs.labels }}

  build-user-frontend-image:
    name: Push the User-Frontend Docker image to Docker Hub
    runs-on: ubuntu-latest
    steps:
      - name: Check out the repo
        uses: actions/checkout@v4
        with:
          submodules: recursive

      - name: Log in to Docker Hub
        uses: docker/login-action@f4ef78c080cd8ba55a85445d5b36e214a81df20a
        with:
          username: ${{ secrets.DOCKER_USERNAME }}
          password: ${{ secrets.DOCKER_PASSWORD }}

      - name: Extract metadata (tags, labels) for Docker
        id: meta
        uses: docker/metadata-action@v4
        with:
          images: apg2000/eletranet-frontend

      - name: Build and push Docker image
        uses: docker/build-push-action@v5
        with:
          context: ./client/EletranetFrontend
          file: ./client/EletranetFrontend/Dockerfile
          push: true
          tags: apg2000/eletranet-frontend:latest
          labels: ${{ steps.meta.outputs.labels }}
          build-args: |
            VITE_BACKEND_API=https://deti-tqs-17.ua.pt:8080
```

Depois de todas as imagens terem sido enviadas com sucesso, inicia-se o workflow de deployment.
Um runner self-hosted do GitHub é executado na máquina, à procura de tarefas para

completar.
Um guia sobre como funcionam os runners self-hosted pode ser consultado aqui.
A action final de deployment do GitHub é a seguinte.

```yaml
deploy:
  runs-on: self-hosted
  name: "Deploy to VM"
  needs: [build-backend-image, build-user-frontend-image]

  steps:
    - name: "Checkout"
      uses: actions/checkout@v4
    - name: Create deployment (Jira)
      id: create_deployment
      uses: chrnorm/deployment-action@releases/v1
      with:
        token: ${{ secrets.GITHUB_TOKEN }}
        environment: production
        ref: ${{ github.ref }}
        description: "Deploy da versão para produção"


    - name: "Deploy"
      run: |
        echo "CREATE DATABASE tqs_backend;" > init.sql
        sudo docker compose -f docker-compose.prod.yml pull
        sudo docker compose -f docker-compose.prod.yml up -d --remove-orphans
        sudo docker container prune -f
        sudo docker image prune -af
        sudo docker builder prune -af

    - name: Update deployment status (Jira)
      uses: chrnorm/deployment-status@releases/v1
      with:
        token: ${{ secrets.GITHUB_TOKEN }}
        deployment_id: ${{ steps.create_deployment.outputs.deployment_id }}
        state: success
```

3.3 System observability

Foi implementado logging na aplicação para monitorizar possíveis erros.
O driver de logging utilizado é o Logback, que regista os logs tanto no terminal .

3.4 Artifacts repository

Para o componente backend, é utilizado apenas um projeto Maven, com segmentação das funcionalidades para os diferentes papéis — condutorVE, operador de Estacao. Dado isto, não há necessidade de gerir artefactos Maven, visto que é usado apenas um projeto.

As imagens Docker utilizadas para o deployment podem ser encontradas nos seguintes links:

Frontend - https://hub.docker.com/repository/docker/apg2000/eletranet-

frontend/general
Backend - https://hub.docker.com/repository/docker/apg2000/eletranet-backend/general

4 Software testing

4.1 Overall strategy for testing

No projeto ElectraNet, a estratégia de desenvolvimento variou conforme o serviço em causa. No backend, os testes unitários vão ser desenvolvidos em paralelo com as funcionalidades, utilizando ferramentas específicas para testar controladores, serviços e repositórios de forma eficiente.

No frontend, os testes vão ser criados após a implementação das funcionalidades principais, permitindo uma rápida identificação e correção de erros nos fluxos de utilizador sem necessidade de reescrita dos testes. Ferramentas como Cucumber BDD e REST-Assured vão ser usadas para simular estes fluxos de forma rigorosa.

O pipeline de Integração Contínua incorpora métricas de qualidade de código que bloqueiam merges na branch principal caso os critérios do Quality Gate não sejam cumpridos, garantindo a qualidade e estabilidade das alterações antes da produção.

Esta abordagem assegura que o ElectraNet mantém elevados padrões de qualidade, minimizando riscos e facilitando a manutenção e evolução da plataforma.

4.2 Functional testing/acceptance

4.3 Unit tests

Paralelamente ao desenvolvimento das funcionalidades, são também desenvolvidos testes unitários para garantir que as funcionalidades específicas adicionadas por estes componentes funcionam corretamente.

Estes testes são desenvolvidos utilizando o framework JUnit 5, com o apoio do framework Mockito para criação de mocks.

vao ser desenvolvidos testes unitários para as várias camadas da aplicação Spring Boot (controladores, serviços e repositórios),

4.4 System and integration testing

Para o serviço backend, os testes de integração permitem validar tanto os componentes individuais como as camadas finais de comunicação entre eles, garantindo que a aplicação completa consegue implementar corretamente as funcionalidades previstas.
Para este propósito, utilizámos uma base de dados PostgreSQL em memória, providenciada pelo Testcontainers no Docker.
Esta base de dados foi escolhida por dois motivos: permitir o reset da base entre testes, evitando assim a "contaminação" com dados de testes anteriores, e possibilitar o uso de dados pré-carregados que podem ser partilhados entre as instâncias de teste.
Isto assegura que o mesmo estado da base de dados é obtido no início de cada teste de integração.

4.5 Performance testing