

TQS: Relatório de Controlo de Qualidade

Francisco Fontinha [76490], João Vasconcelos [88808], Tiago Mendes [88886], Vasco Ramos [88931]

v2020-05-28

1. Bookmarks do Projeto	2
2. Gestão do Projeto	2
2.1. Equipa e Papéis	2
2.2. Gestão do Backlog e Atribuição de Trabalho	3
3. Gestão e Qualidade do Código	4
3.1. Guia de Contribuição (Coding style)	4
3.2. Métricas de Qualidade de Código	4
4. Pipeline de Entrega Contínua (CI/CD)	6
4.1. Development workflow	6
4.1.1. SCM Workflow	6
4.1.2. Revisão de Código	7
4.2. Ferramentas e Pipelines de CI/CD	8
4.2.1. REST API	11
4.2.2. Web Application	12
4.2.3. Mobile Application	13
4.2.4. Compose (Deployment)	14
4.3. Repositório de Artifacts e Containers	16
5. Testes de Software	17
5.1. Estratégia de Teste	17
5.2. Testes Funcionais/Aceitação	17
5.3. Testes Unitários	17
5.4. Testes de Sistema e Integração	18
5.5. Testes de Desempenho	19
6. Monitorização do Ambiente de Produção	21

1. Bookmarks do Projeto

Sistematização dos links para os recursos desenvolvidos no projecto:

- Acesso ao(s) projecto(s) de código, bem como GitLab Agile e GitLab CI/CD:
 - a. Repositório de Grupo ([GitLab](#), mediante acesso)
- Ambiente de produção:
 - a. Aplicação Web ([Spring Boot](#))
 - b. REST API ([SpringBoot](#))
 - c. Documentação da API ([Swagger](#))
- Ambiente SQA :
 - a. Análise estática ([SonarQube](#))
 - b. Monitorização ([Nagios XI](#))
- Coordenação da equipa:
 - a. Slack ([aqui](#) , mediante acesso)

2. Gestão do Projeto

2.1. Equipa e Papéis

Para facilitar a divisão de responsabilidades dentro da equipa, decidimos atribuir cargos específicos a cada elemento:

Cargo	Descrição	Membro/s da equipa
Product Owner	O product owner representa os interesses dos stakeholders. Tem um conhecimento detalhado sobre o produto e o domínio onde se insere e por isso os restantes elementos da equipa irão falar com ele para clarificar dúvidas sobre futuras funcionalidades do produto. Tem um papel ativo em aceitar os novos incrementos desenvolvidos para a solução final.	João Vasconcelos
Quality Engineer	Responsável, em articulação com outras funções, por promover as práticas de garantia de qualidade e colocar em prática instrumentos para medir a qualidade da implementação.	Francisco Fontinha
DevOps Master	O devops master é responsável pela infraestrutura de desenvolvimento e produção, aplicando as configurações necessárias à mesma. Gere a configuração e a preparação das máquinas de deployment, repositório git, infraestrutura da cloud, operações da base de dados, entre outras responsabilidades.	Tiago Mendes Vasco Ramos
Team Manager	O team manager assegura que existe uma divisão justa de tarefas e que o plano de desenvolvimento é seguido por	Vasco Ramos

	todos os elementos da equipa. Promove um bom ambiente dentro da equipa e assegura que os requisitos do projeto são entregues dentro dos prazos estabelecidos.	
Developer	O developer desenvolve a aplicação/projeto consoante o plano definido pelo team manager.	Francisco Fontinha João Vasconcelos Tiago Mendes Vasco Ramos

Tabela 1: Distribuição de Papéis de Trabalho dentro da Equipa

2.2. Gestão do Backlog e Atribuição de Trabalho

Tendo em conta que o desenvolvimento é orientado a *user stories*, existem diversas fases na criação e gestão do *backlog*:

1. **Adicionar as User Stories:** O *Project Manager*, tendo em conta as orientações do cliente, cria diversas *user stories* e adiciona-as ao *backlog* do projeto. Inicialmente, estas não são atribuídas a nenhum elemento da equipa, nem têm uma *timeline* definida;
2. **Priorizar as User Stories:** O *Project Manager* estabelece a prioridade de cada *story*, através de um sistema de pontos;
3. **Estimar o tempo de desenvolvimento de cada User Story:** Numa reunião entre todos os elementos da equipa, estima-se o tempo que demorará a desenvolver cada *story* e complementa-se esta com informação extra (por exemplo: critérios de aceitação/qualidade);
4. **Desenvolver cada User Story:** O desenvolvimento das *stories* está pronto para ser iniciado. Assim, e tendo em conta os prazos apertados de desenvolvimento, cada *story* é atribuída a um *developer*. Após isto, os *developers* começam a desenvolver as *user stories* daquela interação, passando estas para o estado “*In Progress*”;
5. **Entregar as User Stories:** Após o desenvolvimento de uma *story*, acompanhado dos respetivos testes, o *developer* entrega o seu trabalho. Isto vem acompanhado de um *merge request*, uma vez que cada *feature* será desenvolvida num *branch* diferente;
6. **Testar as User Stories:** Após o código entregue por um *developer* ser validado pelo ambiente de CI/CD, este é *deployed* num ambiente de teste. Após isto a *story* passará para o estado “*In Review*”;
7. **Aceitar ou Recusar cada User Story:** O *Project Manager*, em conjunto com o cliente e com outros *developers*, *testers* e *designers*, verifica se os critérios de aceitação definidos inicialmente se verificam e, caso todas as entidades envolvidas concordem, a *user story* é aprovada, saindo do *backlog* para o painel “*Closed*”. Caso contrário, esta *story* terá de ser refeita.

Neste projeto, a ferramenta de gestão de *backlog* que será utilizada será o GitLab que, levando a sua utilização ao máximo e tentando centralizar todo o desenvolvimento e processos numa só ferramenta, vai coexistir também como *Issue Tracking System*.

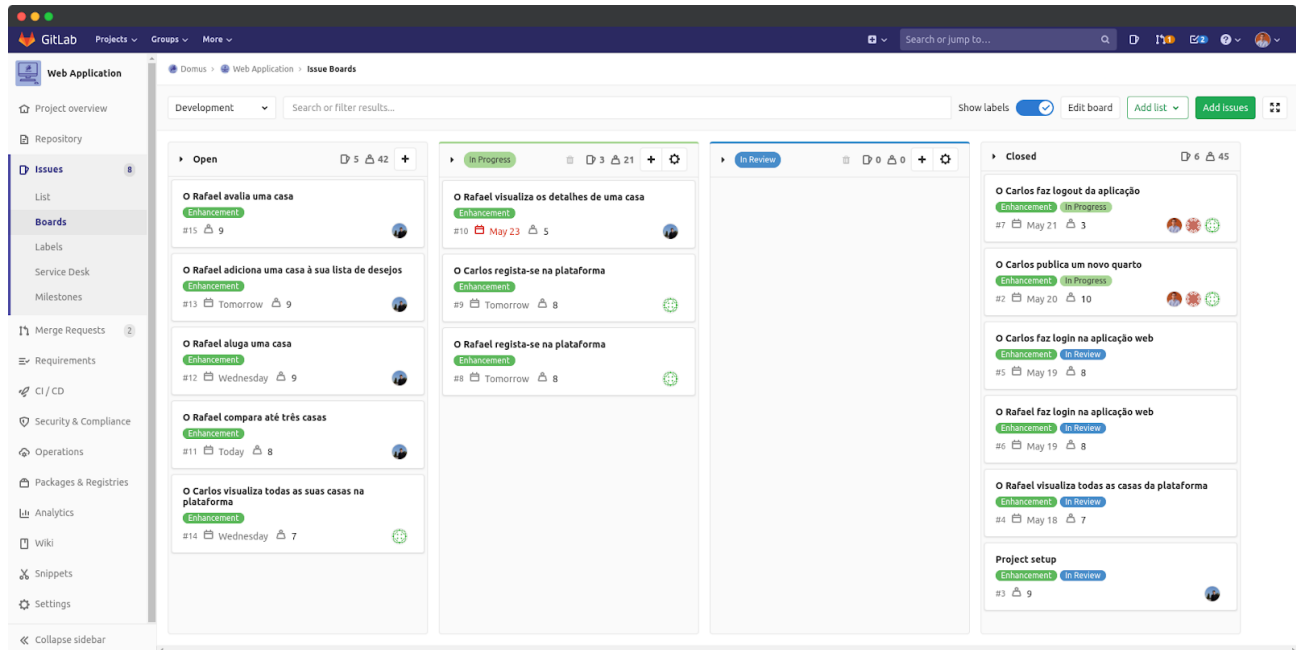


Figura 1: GitLab Agile - Gestão de Backlog + Issue Tracking System

3. Gestão e Qualidade do Código

3.1. Guia de Contribuição (Coding style)

Com o intuito de aplicar as melhores convenções possíveis referentes a escrita de código, iremos utilizar dois guias de estilo de código: um para a **linguagem Java** e outro para a **biblioteca ReactJS**. Em ambos os guias, estão definidas algumas regras e convenções da respetiva linguagem, bem como apresentados bons e maus exemplos práticos de utilização.

Relativamente à linguagem Java, iremos utilizar o seguinte guia de estilo de código, desenvolvido pela equipa do Android: <https://source.android.com/setup/contribute/code-style>.

Quanto à biblioteca ReactJS, o guia de estilo de código escolhido foi o seguinte: <https://github.com/airbnb/javascript/tree/master/react>.

3.2. Métricas de Qualidade de Código

A análise estática de código é um método para fazer o *debugging* do código, antes do programa ser corrido. Durante esta análise, compara-se o código escrito com regras gerais de escrita de código.

Para isto, recorreremos ao SonarQube - uma plataforma *open source* para efetuar reviews automáticas e análise estática de código. Esta ferramenta, não só detecta código com *bugs*, como também *bad smells* e vulnerabilidades de segurança.

Para além disto, o SonarQube permite uma análise de código contínua, pelo que, para além de nos mostrar o estado da aplicação que estamos a desenvolver, também nos mostra quais os problemas introduzidos por cada incremento produzido.

Podemos, também, definir *quality gates* para o nosso projeto. Estas não são mais que meras métricas para a integração de novo código no projeto em desenvolvimento. Se uma nova submissão de código não passar numa *quality gate*, este código terá de ser refeito até que consiga atingir os padrões definidos. Ainda neste tópico, podemos definir *quality gates* relativos a:

- Duplicação de código
- Manutenção necessário para o código escrito
- Fiabilidade
- Segurança

A plataforma escolhida apresenta, também, um grande vantagem no que toca a integração contínua. Uma vez que o SonarQube é facilmente integrado com o GitLab CI/CD, é bastante simples automatizar esta análise estática de código, de forma a que a mesma seja executada de forma contínua. Por fim, tendo em conta que as *quality gates default* do SonarQube nos pareceram adequadas, decidimos manter estas ao invés de redefini-las para uma configuração própria.

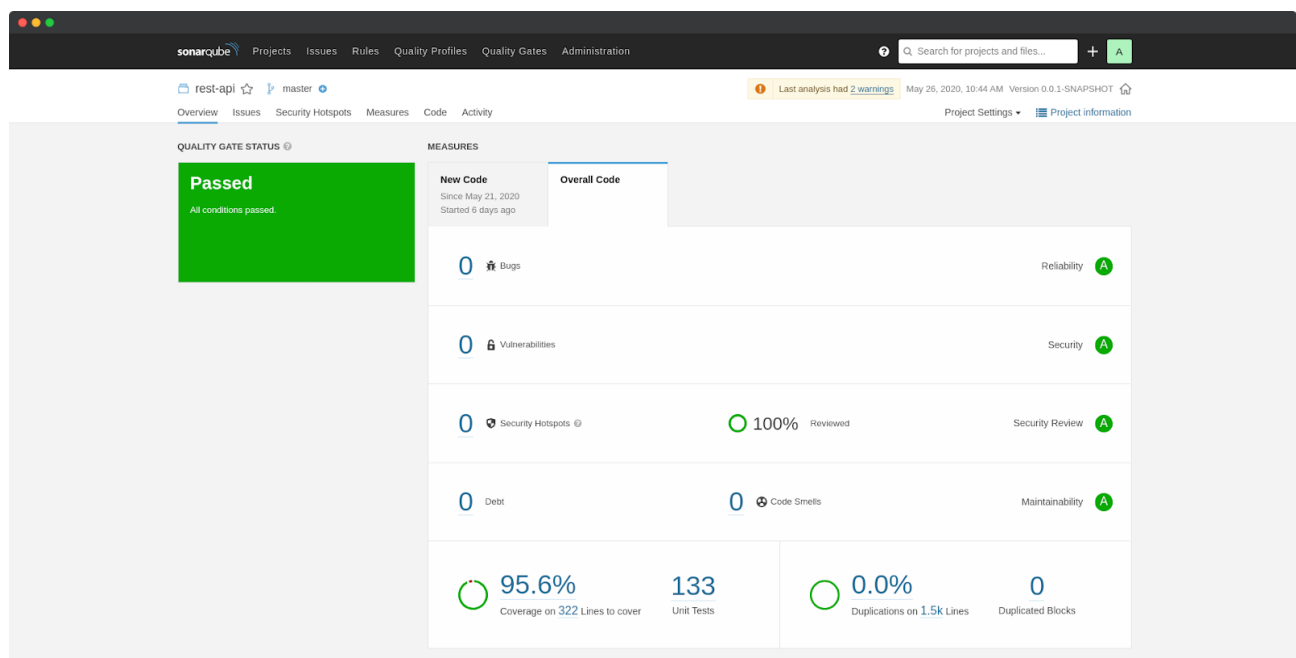


Figura 2: Resultados da Análise do SonarQube

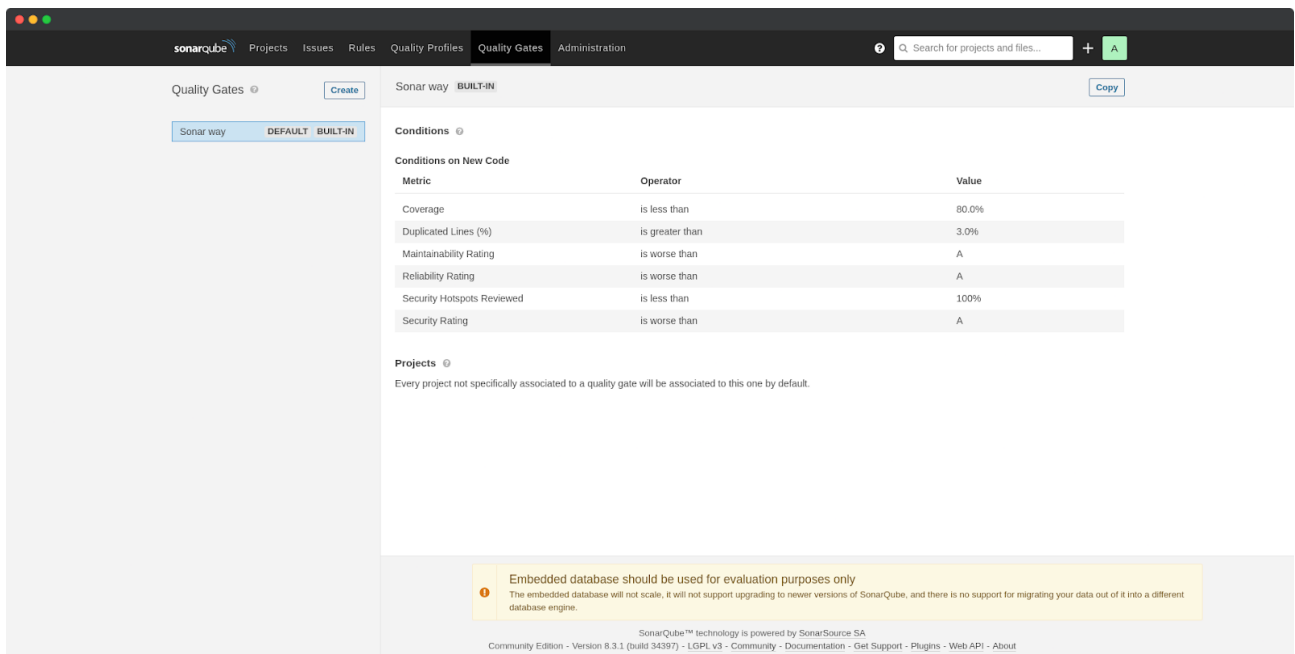


Figura 3: Quality Gates Utilizados na Análise do SonarQube

4. Pipeline de Entrega Contínua (CI/CD)

4.1. Development workflow

4.1.1. SCM Workflow

Relativamente ao *SCM Workflow*, iremos usar **Git Feature Branch Workflow** do *Bitbucket*. Este transmite que:

- Sempre que um *developer* queira desenvolver uma nova *feature/user story*, este deverá fazer o seu desenvolvimento numa nova *branch*, especificamente criada para o efeito. Esta deve ser denominada de forma a que permita rapidamente identificar qual a *feature/issue* a ser tratada, utilizando o padrão **feature/<feature_name>**.
- Além disso, definimos que sempre que seja necessário corrigir algum *bug* devemos criar uma nova *branch* a partir do *master*, com o seguinte formato: **hotfix/<fix_name>**.
- Na nova *branch* criada, o *developer* edita e dá *commits* das suas implementações. Para além disso, este pode também dar *push* da sua *branch* para o repositório central, onde esta irá ser armazenado (*backup*).
- Quando um *developer* acabar de desenvolver as *features* associadas à *branch* que criou, este terá de criar um *merge request* para que a sua *branch* seja unida com a *master branch*. Desta forma, os outros membros da equipa irão receber uma notificação referente a esta situação.

- Os outros *developers* da equipa dão *feedback* sobre o código a ser inserido na *master branch*. Após isto, este código poderá ter de ser reformulado. Assim que o código for aprovado pelos *reviewers*, a *branch* onde está a nova *feature* será *merged* com a *master branch*.
- Por fim, define-se também (como acréscimo ao *flow* em que nos baseámos) que todos os *merge requests* têm de ser aprovados por, pelo menos, um *reviewer* que não seja a próprio que submeteu o *merge request*.

Para mais detalhes, pode ser encontrada uma maior descrição do funcionamento deste *workflow* [aqui](#).

4.1.2. Revisão de Código

De forma a melhorar a qualidade geral do código produzido, é necessário que este seja revisto por diversos *developers*, de forma a que se encontrem erros e potenciais situações de riscos. Para que este processo decorra eficazmente, definimos um conjunto de princípios a seguir:

- Uma *code review* tem como objetivo uma análise minuciosa do código submetido. Tentar entender apenas algumas partes do código poderá, a longo prazo, ter consequências severas, pelo que é normal que uma *code review* demore uma elevada quantidade de tempo;
- Caso alguma porção de código não seja perceptível, o *developer* que o escreveu deverá reformular/explicar esta secção, sendo que os comentários não devem ser esquecidos;
- Não se pode assumir que o código submetido funciona. É necessário fazer *build* do projeto e correr todos os testes associados ao mesmo. O *reviewer* poderá até criar novos testes;
- Caso o código não tenha comentários explicativos, este deve ser corretamente documentado pelo *developer* que o escreveu;
- É necessário rever, também, o código “temporário”, uma vez que este se poderá tornar em código para produção;
- Deve ser realizada uma *review*, quer aos testes, quer aos *build files* associados a código que está a ser revisto;
- As *reviews* de código devem ter em atenção se o *code style* está de acordo com o definido no início do projeto;
- A arquitetura de uma solução poderá, também esta, ser revista;
- Os comentários de uma *code review* devem ser críticas construtivas;
- Ao fazer uma *code review*, as sugestões devem ser feitas de acordo com a seguinte prioridade:
 - Melhorias a nível funcional;
 - Alterações para manter o código *clean* e fácil de manter;

- Por fim, sugestões para otimizar o código.
- Acompanhar o estado de uma *code review* é tão importante como fazer a *code review*, pelo que cada *developer* deverá fazer o *follow up* das *reviews* que fez.

Relativamente ao processo de *code review* implementado neste projeto, este tem como suporte as ferramentas disponibilizadas pelo GitLab. Sempre que é feito um *merge request*, inicia-se um processo de *code review* do código submetido.

Por fim, para esclarecimento futuro, uma *user story* é considerada como terminada após completar todo este processo de: Merge Request -> Build e Execução de Testes Automáticos -> Code Review e Validação de Código -> Deployment. Ou seja, uma *user story* é concluída quando esta é *deployed* com sucesso, respeitando o *flow* previamente especificado.

4.2. Ferramentas e Pipelines de CI/CD

Um dos objetivos deste projeto, é manter ambientes de integração e *delivery* contínuos e, para isso, usámos as ferramentas do GitLab. Para melhor se perceber o *flow* de CI/CD deste projeto, apresenta-se na imagem abaixo a estruturação do nosso projeto, em diferentes repositórios.

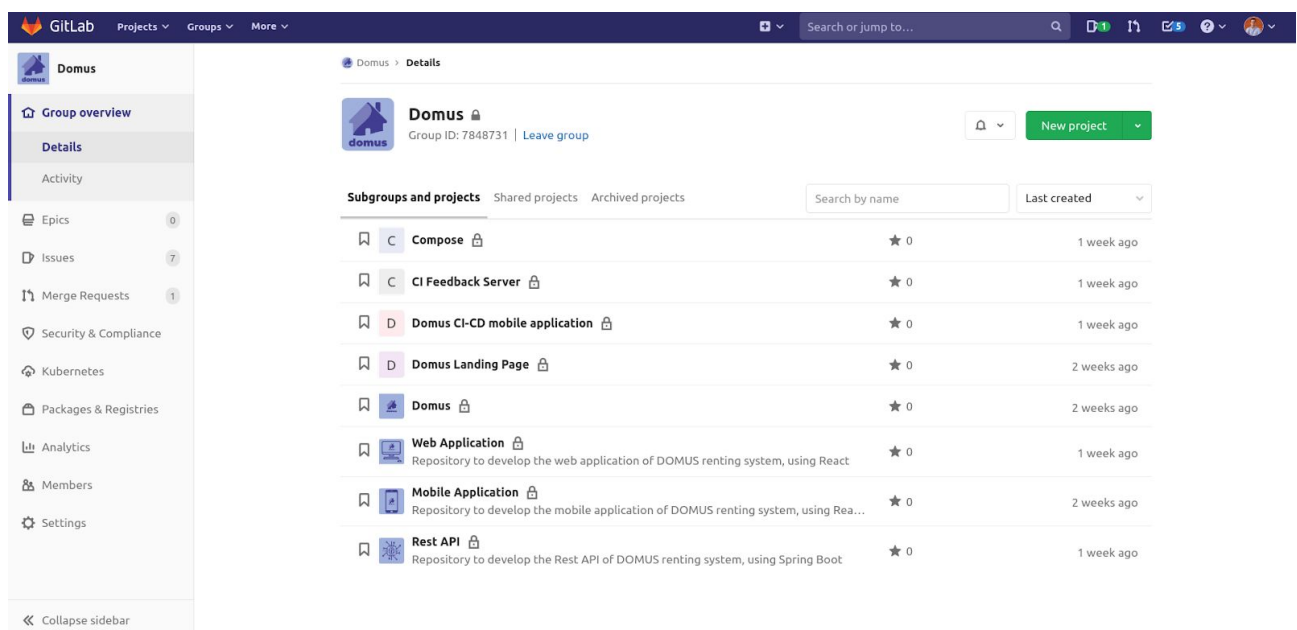


Figura 4: Repositórios do Grupo Domus no GitLab

Como podemos ver na figura 4, existe uma série de repositórios associados ao grupo utilizado para o desenvolvimento deste projeto:

- **Domus:** contém informação básica sobre o projeto e os relatórios associados a este.
- **Domus Landing Page:** contém todo o código associado à nossa *landing page* (que pode ser encontrada [aqui](#)).
- **CI Feedback Server:** contém todo o código necessário para o nosso micro-serviço (em Flask) responsável por servir a informação relativamente às *builds* atuais dos nossos projetos de desenvolvimento, bem como, as notificações de falha nas mesmas.

- **Domus CI-CD mobile application:** contém todo o código para a aplicação *mobile* (em React Native) que consome do servidor do ponto anterior e mostra os estados das *builds* na aplicação.
- **Rest API:** contém todo o código relativo à REST API que serve todo o produto.
- **Web Application:** contém todo o código do nosso *web marketplace*.
- **Mobile Application:** contém todo o código da aplicação *mobile* do nosso *marketplace*.
- **Compose:** contém o código necessário para executar a fase final da *pipeline* (o *deployment* no servidor).

Dos últimos quatro repositórios elencados, cada um tem a sua própria configuração de *pipelines*, que serão analisadas em detalhe nos próximos capítulos. Contudo, antes será dada uma vista geral dos mecanismos de *feedback* ativo que implementámos para o nosso processo de CI/CD.

Maioritariamente, usámos dois mecanismos, que foram: a integração do Slack com o GitLab e um mecanismo próprio de *feedback* através de uma aplicação *mobile* e sistema de notificações.

Tal como se pode ver na figura 5, através da integração do GitLab com o Slack, sempre que são criadas novos *issues*, *merge requests*, *commits* para a *branch master* ou falhas em *builds*, somos notificados no Slack sobre o que acabou de acontecer.

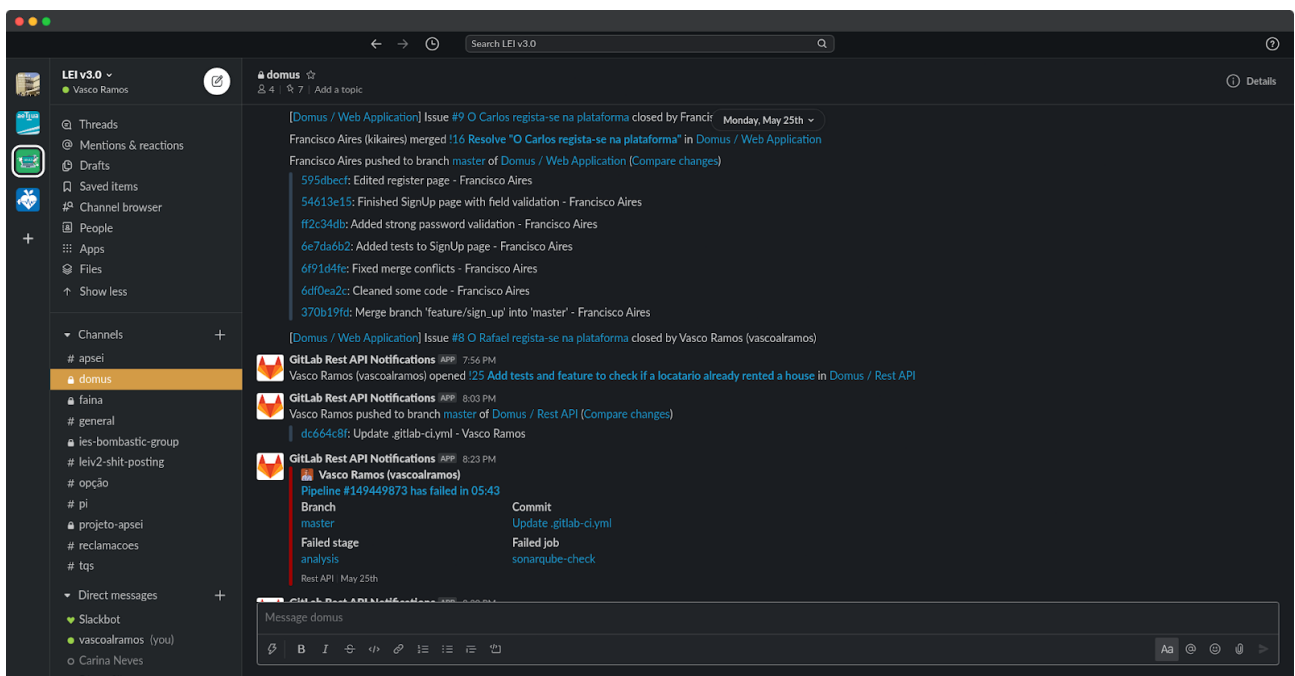


Figura 5: Integração do GitLab no Slack

Para além disto, de forma a mais facilmente obter algum *feedback* sobre que *pipelines* estão a correr e em que estado de execução estão, decidimos criar um micro-serviço que, consultando a api do GitLab, agregasse toda esta informação por nós e uma aplicação *mobile* que, de uma forma simples e visual, nos mostrasse esses estados. Para além disto, decidimos ter também um mecanismo de notificações através da

aplicação que nos avisa se uma *build* falhou. Na figura 6 vê-se um excerto do código do micro-serviço em flask e na figura 7 um *screenshot* da aplicação com uma das notificações referidas.

```
REST_API_ID, MOBILE_ID, WEB_ID, COMPOSE_ID = "18575706", "18577476", "18577488", "18784831"

def fetch_build_state(project_id):
    response = get(url=f"https://gitlab.com/api/v4/projects/{project_id}/pipelines?per_page=1&page=1",
headers=HEADER).json()
    if len(response) > 0:
        return response[0]["status"]
    else:
        return None

@app.route("/build-info")
def get_build_info():
    info = {"rest_api": "null", "mobile": "null", "web": "null", "deploy": "null"}

    # rest_api
    status = fetch_build_state(REST_API_ID)
    if status is not None:
        info["rest_api"] = status

    # web
    status = fetch_build_state(WEB_ID)
    if status is not None:
        info["web"] = status

    # mobile
    status = fetch_build_state(MOBILE_ID)
    if status is not None:
        info["mobile"] = status

    # deploy
    status = fetch_build_state(COMPOSE_ID)
    if status is not None:
        info["deploy"] = status

    return info
```

Figura 6: Micro-serviço CI/CD em Flask

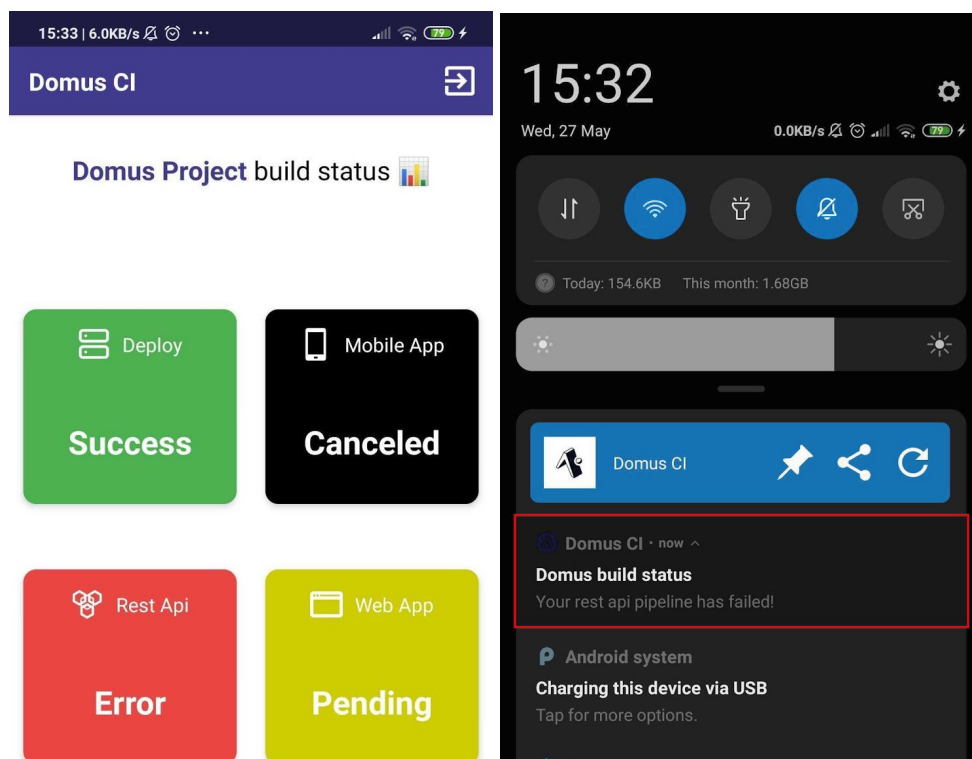


Figura 7: Mecanismo Próprio de Feedback para os processos de CI/CD

4.2.1. REST API

A nossa REST API foi desenvolvida em Spring Boot no ambiente de gestão de dependências Maven, como tal, toda a nossa pipeline se baseou em maven e docker.

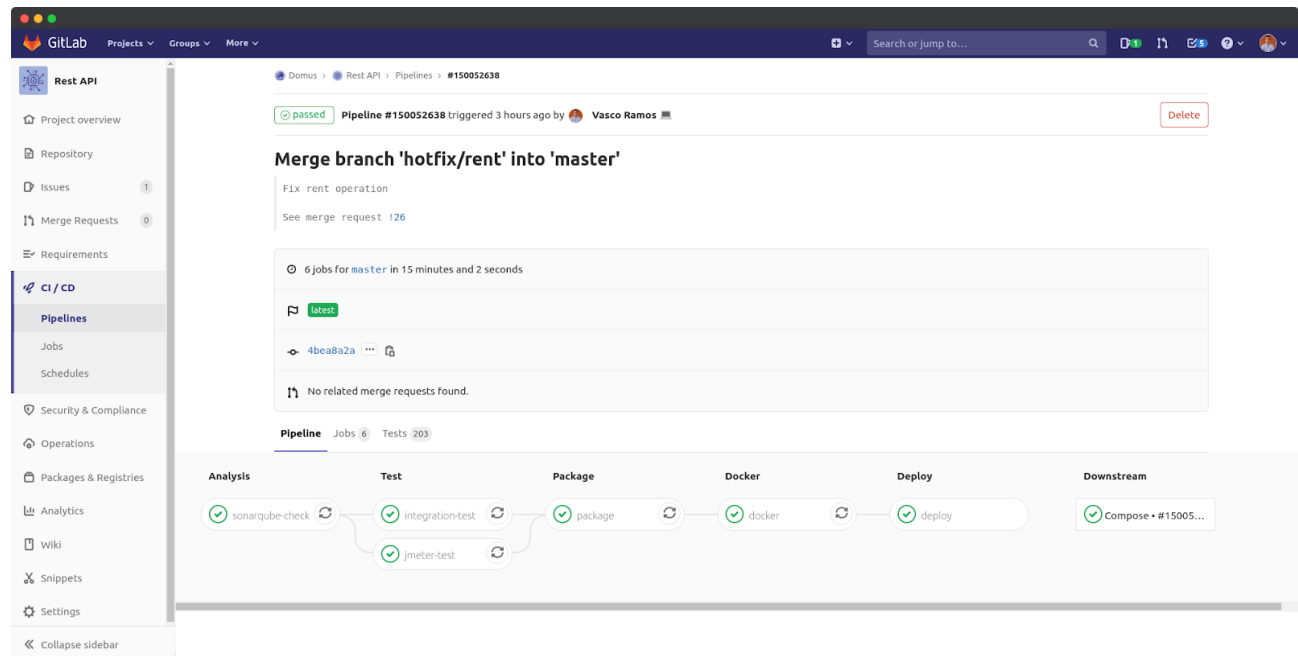


Figura 8: Diagrama CI da REST API

Como se pode ver na figura 8, a nossa pipeline para a REST API tem 5 *stages* diferentes, que vão desde análise estática, até *deployment*, passando pela fase de *testing*:

- **Analysis:** neste *stage* decorre um único *job* que corresponde a análise estática do código através do SonarQube.
- **Test:** neste *stage* decorrem 3 *jobs* diferentes:
 - *test* - este *job* é apenas executado em *branches* que não a *master* e corre apenas os testes unitários.
 - *integration-test* - este *job* é apenas executado em *merge requests* e na *branch master* e corre toda a suíte de testes (unitários e de integração).
 - *jmeter-test* - este *job* é sempre executado e é responsável por correr toda a suíte de testes de carga e desempenho levados a cabo pelo JMeter, tópico explorado mais em detalhe na [secção 5.5](#).
- **Package:** neste *stage* decorre um único *job* responsável por criar o maven *artifact* que será guardado no GitLab Registry e utilizado para acelerar processos de *build*.
- **Docker:** neste *stage* também decorre um único *job* com a finalidade de containerizar o nosso serviço, tirando partido do *artifact* anteriormente criado. Este container também é guardado no GitLab Registry e é utilizado no processo de deployment.

- **Deploy:** o último stage serve apenas de *trigger* para outra *pipeline*, pois, caso todos os jobs tenham sucesso até aqui, este desencadeia o processo de *deployment* desenvolvido no repositório Compose.

Por fim, para esclarecimento e melhor compreensão, é disponibilizado o ficheiro YAML com a configuração desta pipeline [aqui](#).

4.2.2. Web Application

A nossa *Web Application* foi desenvolvida em React.js, através de Node.js, logo, toda a nossa pipeline se baseou em node e docker.

Figura 9: Diagrama CI da Web Application


Como se pode ver na figura 9, a nossa pipeline para a *Web Application* tem 5 *stages* diferentes, que, tal como no repositório anterior, vão desde análise estática, até *deployment*, passando pela fase de *testing*:

- **Lint:** neste *stage* decorre um único *job* com a responsabilidade de analisar padrões de formatação e standards de código.
- **Analysis:** neste *stage* decorre um único *job* que corresponde a análise estática do código através do template de Code Quality do GitLab, baseado em Code Climate.
- **Test:** neste *stage* decorre um único *job* com o propósito de executar os testes funcionais (*end-to-end*), usando *Jest* e *Puppeteer* (tópico explorado em maior detalhe na [secção 5.2](#)).
- **Registry:** neste *stage* também decorre um único *job* com a finalidade de containerizar a nossa *web app*, criando uma imagem *docker* com esta. Este *container* também é guardado no GitLab Registry e é utilizado no processo de *deployment*.
- **Deploy:** este último *stage* é inteiramente igual ao *stage deploy* na REST API, pois serve, precisamente o mesmo propósito.

Por fim, para esclarecimento e melhor compreensão, é disponibilizado o ficheiro YAML com a configuração desta pipeline [aqui](#).

4.2.3. Mobile Application

A nossa *Mobile Application* foi desenvolvida em React Native, através de Node.js, logo, toda a nossa pipeline se baseou em node e docker.



```
image: node:alpine

stages:
  - lint
  - test
  - deploy

cache:
  key: ${CI_COMMIT_REF_SLUG}
  paths:
    - ~/.npm
    - .jest

before_script: ①
  - npm ci

lint: ②
  stage: lint
  script:
    - npm install eslint
    - npm install eslint-plugin-react@latest
    - eslint --fix --max-warnings 20 src/

test: ③
  stage: test
  script:
    - npx jest --ci

deploy: ④
  stage: deploy
  script:
    - npm install expo-cli
    - apk add --no-cache bash
    - echo fs.inotify.max_user_watches=524288 | tee -a /etc/sysctl.conf && sysctl -p
    - npx expo login -u $EXPO_USERNAME -p $EXPO_PASSWORD
    - npx expo publish --non-interactive

only:
  - master
```

Figura 10: Pipeline de CI/CD da Mobile Application

Como se pode ver na figura 10, a nossa pipeline para a *Mobile Application* tem 3 *stages* diferentes, que vão desde *lint*, até *deployment*, passando pela fase de *testing*:

- **Lint:** este *stage* é muito semelhante ao stage de *lint* mostrado na *Web Application*.
- **Test:** neste *stage* decorre um único *job* com o propósito de executar os testes à aplicação, usando *Jest* e *Puppeteer*.
- **Deploy:** neste último stage decorre apenas um *job* com o propósito de executar o *deploy* da aplicação *mobile*. Este *deploy* funciona através da ferramenta *publish* da *framework* de React

Native, Expo, que executa uma atualização “*over the air*” da aplicação nos clientes que já têm o apk instalado. Este mecanismo permite-nos não ter de andar sempre a gerar o apk, conforme vão surgindo os incrementos à aplicação.

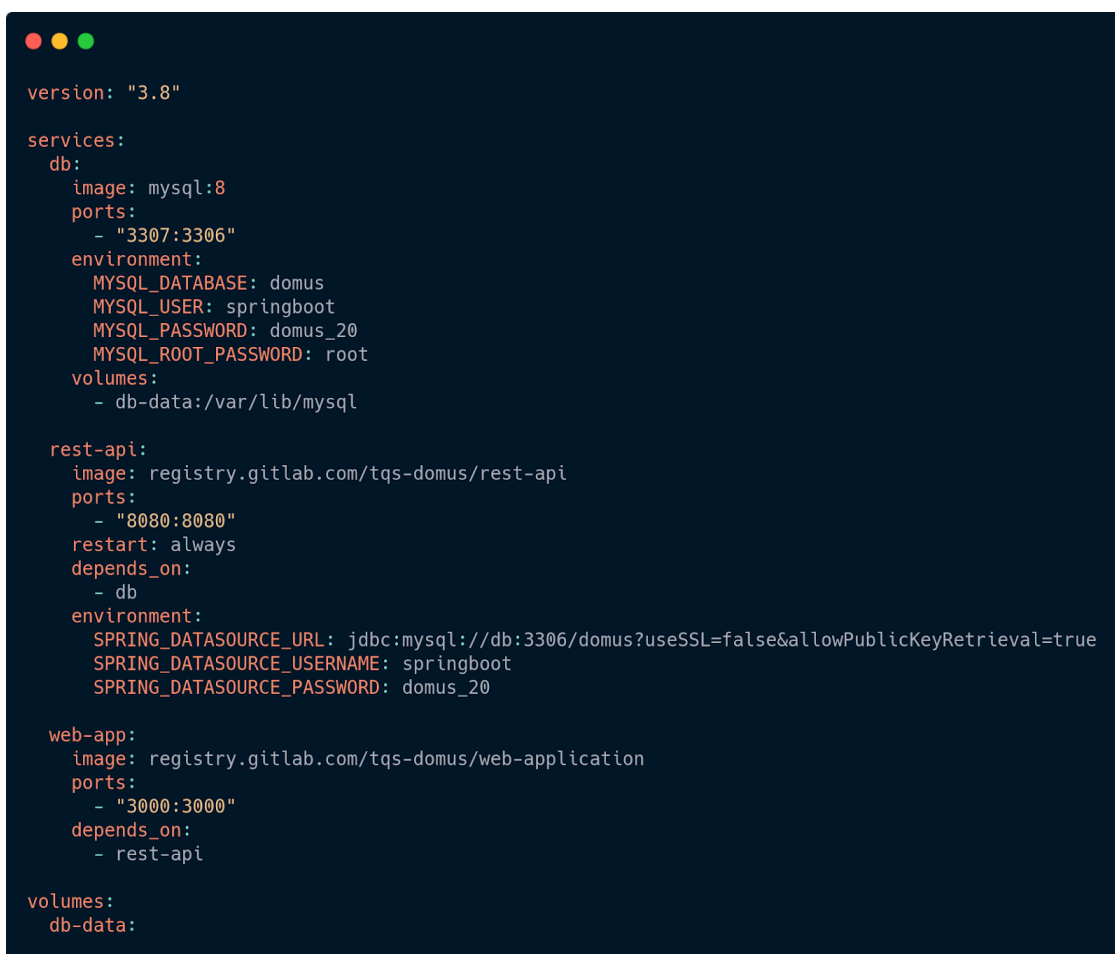
Antes de qualquer um dos stages correr, é executado o *before_script*, que tem a responsabilidade de instalar todos os *packages* necessários à aplicação. A numeração presente na figura 10 corresponde à ordem de execução dos vários jobs.

4.2.4. Compose (Deployment)

Este repositório é utilizado unicamente para o objetivo de executar o *deploy* para a nossa VM por SSH. Para isso existem dois ficheiros importantes: *docker-compose.yml* e *.gitlab-ci.yml*.

Na figura 11, podemos ver o docker compose utilizado para o *deploy*. Neste ficheiro temos 3 serviços:

- Base de dados MySQL, usada para persistência.
- REST API, usando a imagem que está guardada no GitLab Registry, o serviço da API é iniciado (este serviço depende da base de dados, pois comunica com ela).
- Web Application, tal como na REST API, usa a imagem do projeto que está guardado no GitLab Registry. À semelhança do serviço anterior, este depende da REST API, pois comunica com esta para obter os dados.



```
version: "3.8"

services:
  db:
    image: mysql:8
    ports:
      - "3307:3306"
    environment:
      MYSQL_DATABASE: domus
      MYSQL_USER: springboot
      MYSQL_PASSWORD: domus_20
      MYSQL_ROOT_PASSWORD: root
    volumes:
      - db-data:/var/lib/mysql

  rest-api:
    image: registry.gitlab.com/tqs-domus/rest-api
    ports:
      - "8080:8080"
    restart: always
    depends_on:
      - db
    environment:
      SPRING_DATASOURCE_URL: jdbc:mysql://db:3306/domus?useSSL=false&allowPublicKeyRetrieval=true
      SPRING_DATASOURCE_USERNAME: springboot
      SPRING_DATASOURCE_PASSWORD: domus_20

  web-app:
    image: registry.gitlab.com/tqs-domus/web-application
    ports:
      - "3000:3000"
    depends_on:
      - rest-api

volumes:
  db-data:
```

Figura 11: Docker Compose utilizado no Deployment

Antes de passarmos para a pipeline em si, é de salientar que a utilização do GitLab Registry foi bastante importante, pois, ajuda imenso a desacoplar os processos de desenvolvimento e *deploy*, sendo que basta ter as imagens *docker* lá guardadas e utilizá-las quando precisamos.

Relativamente à *pipeline*, como podemos ver na figura 12, temos uma primeira fase (*before_script*), que é executada antes do único *stage* que temos (o *deploy_staging*). Nesta primeira fase são executados os comandos necessários para garantir que a nossa ligação SSH à VM está a funcionar e que o GitLab consegue comunicar com esta.

No *stage* de *deploy*, começamos por copiar o docker compose mostrado acima para a VM, depois fazemos o *login* com as nossas credenciais no GitLab Registry para conseguirmos aceder às imagens que precisamos e, por fim, executamos o docker compose, de forma a que todos os serviços fiquem a correr no nosso *host*.

```
image: ubuntu

stages:
  - deploy

before_script:
  - 'which ssh-agent || ( apt-get update -y && apt-get install openssh-client git -y )'
  - eval $(ssh-agent -s)
  - echo "$SSH_PRIVATE_KEY" | tr -d '\r' | ssh-add -
  - mkdir -p ~/.ssh
  - chmod 700 ~/.ssh
  - ssh-keyscan gitlab.com >> ~/.ssh/known_hosts
  - chmod 644 ~/.ssh/known_hosts
  - '[[ -f /.dockerenv ]] && echo -e "Host *\n\tStrictHostKeyChecking no\n\n" > ~/.ssh/config'

deploy_staging:
  type: deploy
  stage: deploy
  script:
    - echo 'Deploying...'
    - scp docker-compose.yml tq$@HOST:/home/tqs/compose/
    - ssh tq$@HOST "docker login registry.gitlab.com -u $GITLAB_USER_NAME -p $GITLAB_USER_PASSWORD && exit"
    - ssh tq$@HOST "source .profile && cd /home/tqs/compose/ && docker-compose pull && docker-compose up -d &&
exit"
  only:
    - master
```

Figura 12: Pipeline de Deploy para a VM por SSH

4.3. Repositório de Artifacts e Containers

Para facilitar e agilizar a gestão de *artifacts*, bem como a execução das *pipelines* de integração e *deployment*, decidimos usar o GitLab Registry (para Maven *artifacts* e Docker *containers*). Estes permitem-nos uma rapidez maior na execução das *pipelines* de integração através dos mecanismos de cache do GitLab, bem como rapidamente executar o deployment, tirando partido dos *containers docker* respetivos aos vários serviços que são criados durante as *pipelines* de execução. O repositório-mãe encontra-se nos seguinte endereços:

- Package Registry: <https://gitlab.com/groups/tqs-domus/-/packages>
- Container Registry: https://gitlab.com/groups/tqs-domus/-/container_registries.

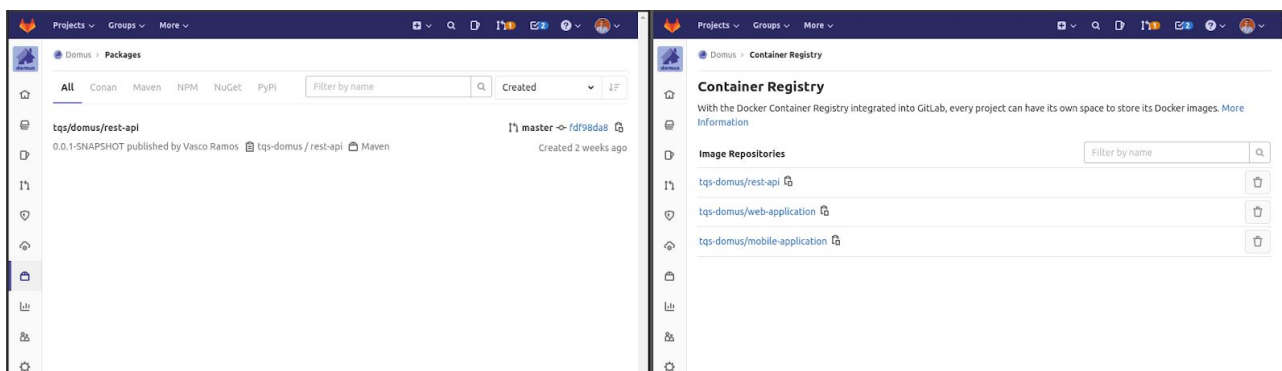


Figura 13: GitLab Registry - Maven Artifacts + Docker Containers

5. Testes de Software

5.1. Estratégia de Teste

Esta secção é destinada a documentar a estratégia de testes usada no projeto. Desde o início do desenvolvimento, que o objetivo foi sempre aplicar TDD (*Test-Driven Development*), de forma a que se crie apenas o essencial para implementar os testes e, a partir daí, desenvolver as funcionalidades.

Tecnologicamente, usámos JUnit 5 e Mockito (quando necessário) para os testes da REST API, Jest e Puppeteer para os testes funcionais na Web App e Jest para testar a Mobile App.

5.2. Testes Funcionais/Aceitação

Este tipo de testes são, geralmente, realizados em cooperação com o cliente (interface), uma vez que são os testes que irão validar se a aplicação está de acordo com o que o cliente pretende. Para este tipo de teste, usamos casos de teste que cobrem os cenários típicos sob os quais esperamos que o software seja usado. Estes testes devem ser conduzidos num ambiente de "produção", e num hardware que seja igual ou próximo do que cliente usará.

Este tipo de testes são, também conhecidos como *black-box testing*, pois não existe qualquer noção do sistema interno da aplicação, sendo apenas enviado os valores de entrada e analisado os valores de saída correspondentes.

[Project policy for writing functional tests (closed box, user perspective) and associated resources.]

5.3. Testes Unitários

De forma a garantir que todas as funcionalidades, por mais minimalista que sejam, não perturbem o bom funcionamento da aplicação e funcionam como pretendido, são expostas a testes unitários. Cada teste unitário executa o código fonte de uma e apenas uma funcionalidade, daí ser unitário. Tem como metodologia *white-box testing*, ou seja, tem como objetivo testar a estrutura interna de uma aplicação.

```

@Test
void testCreateHouseReview_missingParameters() {
    UserDTO userDTO = new UserDTO("v1@ua.pt", "Vasco", "Ramos", "pwd", "123", "M", null);
    User user = new ModelMapper().map(userDTO, User.class);
    Locatario locatario = new Locatario();
    locatario.setUser(user);

    when(locatarioRepository.findById(anyLong())).thenReturn(Optional.of(locatario));

    HouseDTO houseDTO = new HouseDTO("Av. da Misericórdia", "São João da Madeira", "3700-191", 2, 2, 2, 300,
        true, 230, "Casa T2", "Casa muito bonita", "WI-FI;Máquina de lavar",
        null, null);
    House house = new ModelMapper().map(houseDTO, House.class);

    when(repository.findById(anyLong())).thenReturn(Optional.of(house));

    HouseReviewDTO reviewDTO = new HouseReviewDTO(0L, 0L, "string", 0D);

    when(reviewRepository.save(any(HouseReview.class))).thenThrow(ConstraintViolationException.class);

    assertThrows(ErrorDetails.class, () -> {
        service.registerReview(reviewDTO);
    });
}

```

Figura 15: House Service - Testes Unitários - Operação de criar uma nova review a uma casa

5.4. Testes de Sistema e Integração

Apesar dos testes unitários, por vezes, agregarem alguns módulos para realizar um teste, nunca testam o que se denomina de lógica do negócio, isto é, os módulos nunca são agregados de forma a serem testados como um todo, e é aqui que surgem os testes de sistema e de integração, e por isso, estes testes são realizados após os testes unitários. Estes testes foram, também, implementados usando as ferramentas do Spring Boot e do JUnit 5.

```

@Test
void testCreateHouse_correctParameters() throws Exception {
    House house = new ModelMapper().map(houseDTO, House.class);
    house.setLocador(locador);
    String houseJsonString = mapper.writeValueAsString(house);

    servlet.perform(post("/houses")
        .content(houseJsonString).contentType(MediaType.APPLICATION_JSON)
        .accept(MediaType.APPLICATION_JSON)).andExpect(status().isOk())
        .andExpect(jsonPath("street", is(house.getStreet())))
        .andExpect(jsonPath("city", is(house.getCity())))
        .andExpect(jsonPath("postalCode", is(house.getPostalCode())))
        .andExpect(jsonPath("noRooms", is(house.getNoRooms())))
        .andExpect(jsonPath("noBathrooms", is(house.getNoBathrooms())))
        .andExpect(jsonPath("noGarages", is(house.getNoGarages())))
        .andExpect(jsonPath("habitableArea", is(house.getHabitableArea())))
        .andExpect(jsonPath("price", is(house.getPrice())))
        .andExpect(jsonPath("name", is(house.getName())))
        .andExpect(jsonPath("description", is(house.getDescription())))
        .andExpect(jsonPath("propertyFeatures", is(house.getPropertyFeatures())))
        .andExpect(jsonPath("photos", is(house.getPhotos())));
}

```

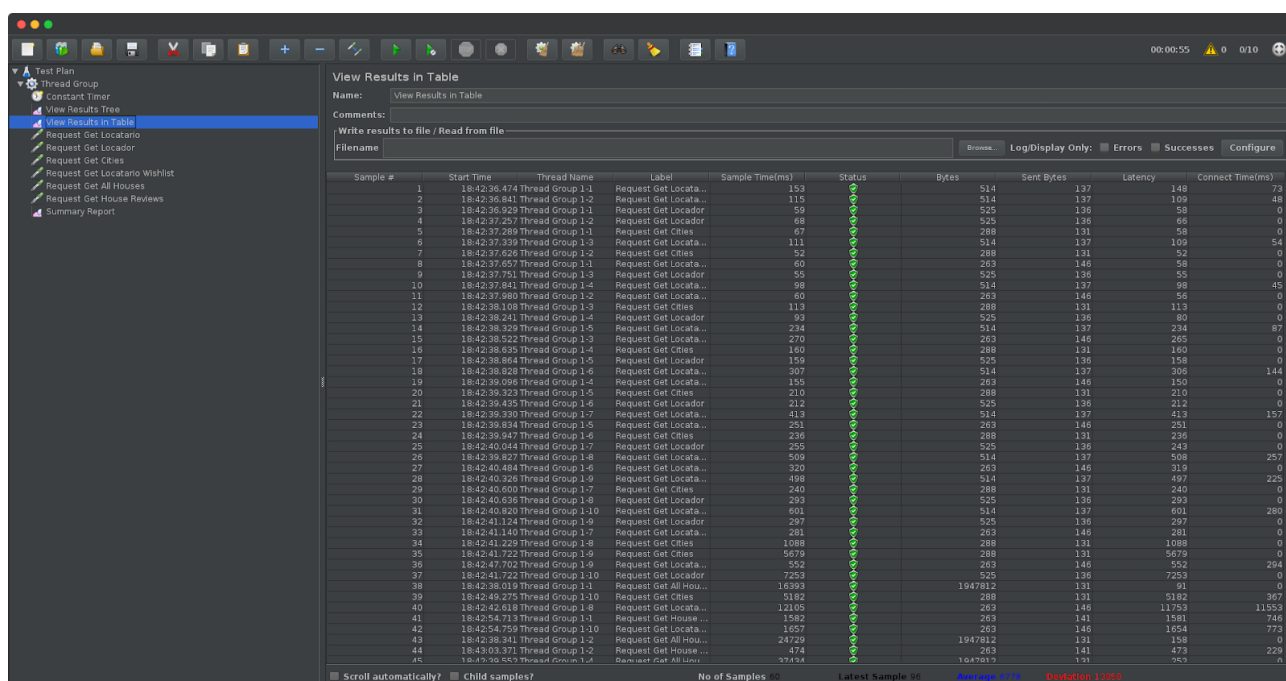
Figura 16: House Controller - Testes de Integração - Endpoint para registar uma nova casa

5.5. Testes de Desempenho

O tempo de resposta de qualquer aplicação deve ser o mais baixo quanto possível, de forma a que o utilizador final possa usufruir de uma experiência estável e fluída.

Posto isto, usámos o Apache JMeter 5.2.1 - uma ferramenta de testes de carga para analisar e medir o desempenho de serviços. Dado que tem como foco as aplicações web, é open source e está escrito em Java, é ideal para a nossa solução. Os testes implementados incidem sobre a REST API e executam um conjunto de operações, num grupo de *threads* constituído por 10 *threads*, que representam uma simulação de 10 utilizadores a aceder à REST API.

Como se pode ver na figura 17, foram feitos testes a 6 endpoints diferentes (alguns dos mais “críticos”) e cada teste foi executado por 10 *threads* (utilizadores).



The screenshot shows the JMeter GUI in Table View. The left sidebar lists the test plan structure, including 'Test Plan', 'Thread Group', 'Constant Timer', 'View Results Tree', and 'View Results in Table'. The main area displays a table of test results with columns: Sample #, Start Time, Thread Name, Label, Sample Time(ms), Status, Bytes, Sent Bytes, Latency, and Connect Time(ms). The table contains 45 rows of data, representing 10 threads per endpoint across 5 different endpoints. The endpoints are: Request Get Locatario, Request Get Locador, Request Get Cites, Request Get Locatario Wishlist, Request Get All Houses, and Request Get House Reviews. The status column shows green checkmarks for successful requests and red X's for failed requests. The bottom status bar indicates 'No of Samples: 45', 'Latest Sample: 45', 'Average: 1.000', and 'Deviation: 1.000'.

Sample #	Start Time	Thread Name	Label	Sample Time(ms)	Status	Bytes	Sent Bytes	Latency	Connect Time(ms)
1	18:42:36.474	Thread Group 1-1	Request Get Locatario	153	Success	514	137	148	73
2	18:42:36.841	Thread Group 1-2	Request Get Locatario	115	Success	514	137	109	48
3	18:42:36.929	Thread Group 1-1	Request Get Locador	59	Success	525	136	58	0
4	18:42:37.257	Thread Group 1-2	Request Get Locador	68	Success	525	136	66	0
5	18:42:37.289	Thread Group 1-1	Request Get Cites	67	Success	288	131	58	0
6	18:42:37.339	Thread Group 1-3	Request Get Locatario	111	Success	514	137	109	54
7	18:42:37.630	Thread Group 1-2	Request Get Cites	52	Success	288	131	52	0
8	18:42:37.657	Thread Group 1-1	Request Get Locatario	60	Success	263	146	58	0
9	18:42:37.751	Thread Group 1-3	Request Get Locador	55	Success	525	136	55	0
10	18:42:37.843	Thread Group 1-4	Request Get Locatario	98	Success	514	137	98	45
11	18:42:37.980	Thread Group 1-2	Request Get Locatario	60	Success	263	146	56	0
12	18:42:38.108	Thread Group 1-3	Request Get Cites	113	Success	288	131	113	0
13	18:42:38.243	Thread Group 1-4	Request Get Locador	93	Success	505	136	80	0
14	18:42:38.329	Thread Group 1-5	Request Get Locatario	234	Success	514	137	234	87
15	18:42:38.522	Thread Group 1-3	Request Get Locatario	270	Success	263	146	265	0
16	18:42:38.635	Thread Group 1-4	Request Get Cites	160	Success	288	131	160	0
17	18:42:38.864	Thread Group 1-5	Request Get Locador	159	Success	525	136	158	0
18	18:42:38.828	Thread Group 1-6	Request Get Locatario	207	Success	514	137	206	144
19	18:42:39.096	Thread Group 1-4	Request Get Locatario	155	Success	263	146	150	0
20	18:42:39.323	Thread Group 1-5	Request Get Cites	210	Success	288	131	210	0
21	18:42:39.435	Thread Group 1-6	Request Get Locador	212	Success	525	136	212	0
22	18:42:39.330	Thread Group 1-7	Request Get Locatario	413	Success	514	137	413	157
23	18:42:39.834	Thread Group 1-5	Request Get Locatario	251	Success	263	146	251	0
24	18:42:39.947	Thread Group 1-6	Request Get Cites	236	Success	288	131	236	0
25	18:42:40.044	Thread Group 1-7	Request Get Locador	255	Success	525	136	243	0
26	18:42:39.827	Thread Group 1-8	Request Get Locatario	509	Success	514	137	508	257
27	18:42:40.484	Thread Group 1-6	Request Get Locatario	320	Success	263	146	318	0
28	18:42:40.526	Thread Group 1-9	Request Get Locatario	498	Success	514	137	497	225
29	18:42:40.600	Thread Group 1-7	Request Get Cites	240	Success	288	131	240	0
30	18:42:40.636	Thread Group 1-8	Request Get Locador	293	Success	525	136	293	0
31	18:42:40.820	Thread Group 1-10	Request Get Locatario	601	Success	514	137	601	280
32	18:42:41.124	Thread Group 1-9	Request Get Locador	297	Success	525	136	297	0
33	18:42:41.140	Thread Group 1-7	Request Get Locatario	261	Success	263	146	261	0
34	18:42:41.223	Thread Group 1-8	Request Get Cites	1088	Success	288	131	1088	0
35	18:42:41.722	Thread Group 1-9	Request Get Cites	5679	Success	288	131	5679	0
36	18:42:47.702	Thread Group 1-9	Request Get Locatario	552	Success	263	146	552	294
37	18:42:41.722	Thread Group 1-10	Request Get Locador	7253	Success	505	136	7253	0
38	18:42:38.019	Thread Group 1-1	Request Get All Houses	16393	Success	1947812	131	91	0
39	18:42:49.275	Thread Group 1-10	Request Get Cites	5162	Success	288	131	5162	367
40	18:42:43.618	Thread Group 1-8	Request Get Locatario	12105	Success	263	146	11753	11553
41	18:42:54.713	Thread Group 1-1	Request Get House Reviews	1582	Success	263	141	1581	746
42	18:42:54.759	Thread Group 1-10	Request Get Locatario	1057	Success	263	146	1054	773
43	18:42:38.841	Thread Group 1-2	Request Get All Houses	24706	Success	1947812	131	188	0
44	18:43:03.371	Thread Group 1-2	Request Get House Reviews	474	Success	263	141	473	229
45	18:43:30.553	Thread Group 1-1	Request Get All Houses	57434	Success	1647815	131	765	0

Figura 17: JMeter GUI - Table View

Como podemos ver na figura 18, que foi obtida como um resultado da execução dos testes do JMeter na *pipeline* de CI, os resultados médios obtidos foram: ~36.23ms por pedido. Relativamente ao *throughput*, os testes dizem-nos que o servidor consegue servir cerca de 10 pedidos por segundo, contudo, isto deve-se ao facto do pedido para retornar a lista das casas registadas no sistema ser bastante pesado, pois, existe muita informação a ser transferida como, por exemplo, as fotografias das várias casas. No entanto, tendo em conta o âmbito geral do sistema, apesar de não serem ideais, os valores alcançados são bastante satisfatórios.

Statistics													
Requests		Executions			Response Times (ms)						Throughput	Network (KB/sec)	
Label	#Samples	KO	Error %	Average	Min	Max	90th pct	95th pct	99th pct	Transactions/s	Received	Sent	
Total	60	0	0.00%	36.23	6	299	106.70	137.65	299.00	10.35	3285.54	1.41	
Request Get All Houses	10	0	0.00%	112.00	86	139	138.90	139.00	139.00	2.50	4750.65	0.32	
Request Get Cities	10	0	0.00%	8.70	6	17	16.40	17.00	17.00	2.53	0.71	0.33	
Request Get House Reviews	10	0	0.00%	12.80	9	20	19.60	20.00	20.00	2.58	0.66	0.36	
Request Get Locador	10	0	0.00%	12.30	7	21	20.50	21.00	21.00	2.52	1.29	0.34	
Request Get Locatario	10	0	0.00%	49.60	8	299	275.20	299.00	299.00	2.41	1.21	0.33	
Request Get Locatario Wishlist	10	0	0.00%	22.00	9	63	60.60	63.00	63.00	2.54	0.65	0.37	

Figura 18: Tabela com Estatísticas representativas dos testes executados

6. Monitorização do Ambiente de Produção

Tendo em mente a necessidade de monitorizar e consultar o estado das máquinas usadas no ambiente de produção, utilizamos a ferramenta Nagios XI, uma extensão à interface já existente do Nagios Core, com a função de monitorizar sistemas, redes e infraestruturas.

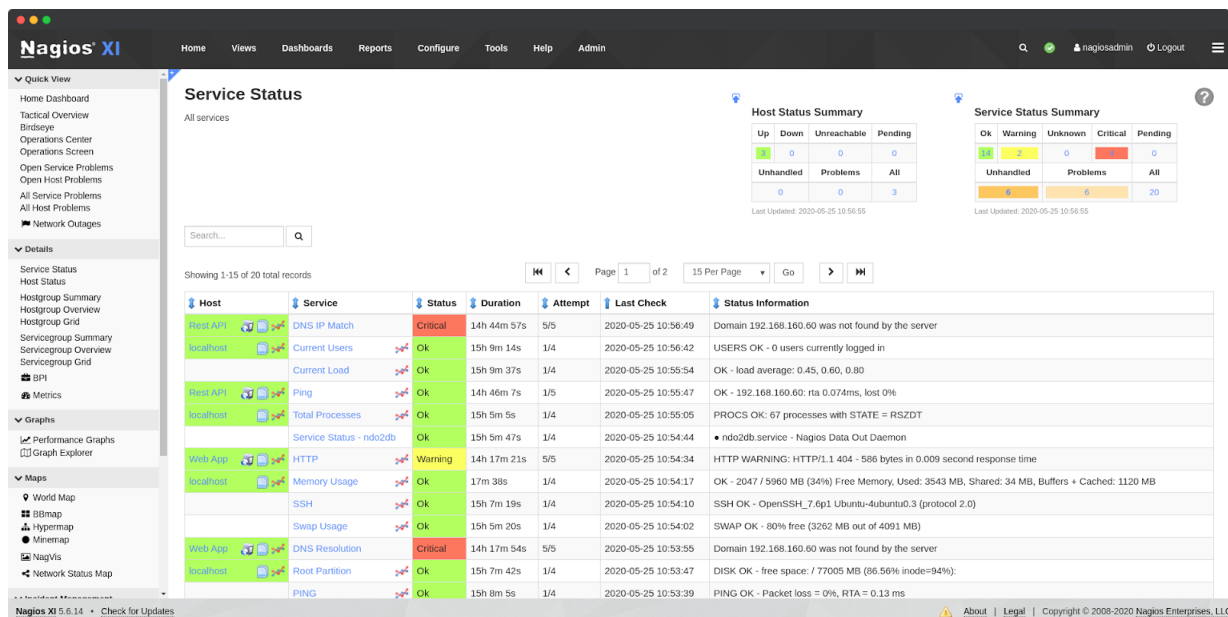


Figura 19: Dashboard do Nagios XI

Posto isto, na nossa dashboard do Nagios XI, podemos monitorizar os *hosts* configurados (*localhost* e *deployment* - REST API + Web App), e os serviços a eles associados como, por exemplo, o uso de memória, a carga atual e http, entre outros. Para além disto, é, também, possível visualizar os dados processados sobre o desempenho através de gráficos, como podemos observar na figura abaixo.

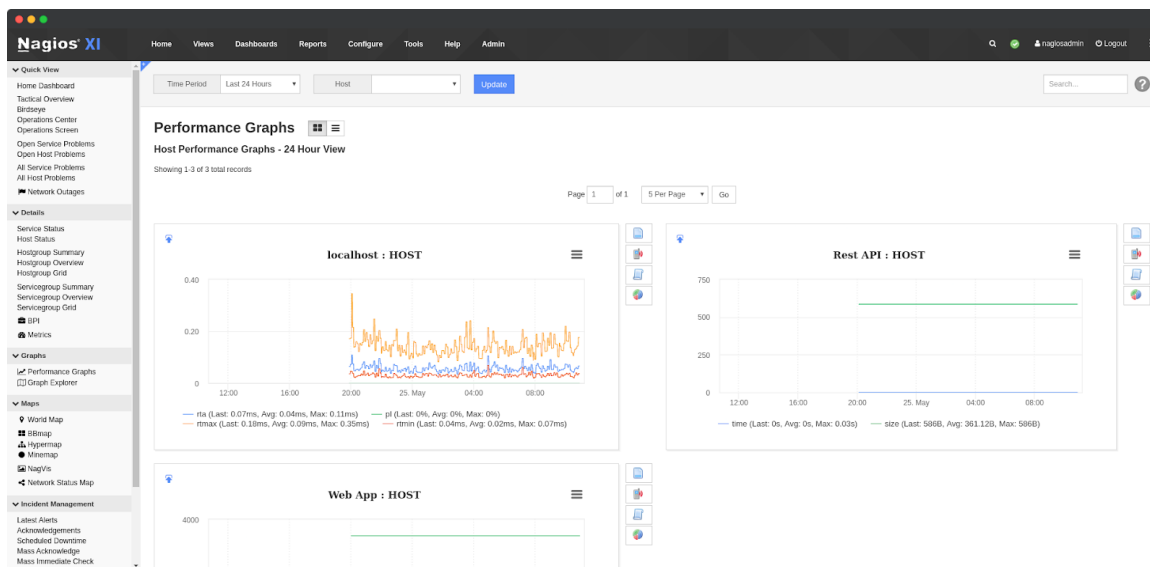


Figura 20: Gráficos de desempenho do Nagios XI