IPBeja

INSTITUTO POLITÉCNICO DE BEJA

Escola Superior de Tecnologia e Gestão

Licenciatura em Engenharia Informática

Relatório de Estágio

Desenvolvimento em Backend na Optiply

Gonçalo Candeias Amaro

Beja, Portugal, 28 de Junho de 2022

INSTITUTO POLITÉCNICO DE BEJA

Escola Superior de Tecnologia e Gestão

Licenciatura em Engenharia Informática

Relatório de Estágio

Desenvolvimento em Backend na Optiply

Gonçalo Candeias Amaro

Orientado por :

Fábio Belga
Gonçalo Fontes, IPBeja

Relatório de estágio, realizado na Optiply, apresentado na
Escola Superior de Tecnologia e Gestão do Instituto Politécnico de Beja

2022

Resumo

Relatório de Estágio

Desenvolvimento em Backend na Optiply

Este relatório consiste na representação e documentação do decorrer do Estágio Profissional, realizado como parte integrante e conclusiva da Licenciatura em Engenharia informática pela Escola Superior de Tecnologia e Gestão do Instituto Politécnico de Beja.

O Estágio Profissional desenvolveu-se na Optiply, em Évora, no ano letivo de 2021/2022, tendo como objetivo favorecer a integração e consolidação, no contexto da pratica, os conhecimentos teóricos adquiridos durante o decorrer da Licenciatura.

O objetivo primordial do estágio seria o de integração no mundo do trabalho. A ideia de se estagiar na Optiply veio no sentido de propiciar ao estudante um primeiro contacto com a área do desenvolvimento em Backend, e a possibilidade de se desenvolver pessoalmente num ambiente de trabalho que seja compatível com o que se pretende fazer.

As atividades foram desenvolvidas tendo sempre em conta os objetivos inicialmente delineados e que se propôs atingir para a função do estagiário, que foram: treino inicial via cursos do Udemy, o desenvolvimento de um projeto, planificação e implementação do mesmo.

Este referido projeto foi um projeto de desenvolvimento de Backend, que foi desenvolvido em Java, utilizando o framework Micronaut, ligado a uma base de dados Postgres, e que foi desenvolvido num ambiente de desenvolvimento local usando containers Docker.

A aprendizagem durante o estágio foi efetiva e percetível, na medida em que se desenvolveram diversas atividades que proporcionaram a aquisição e o desenvolvimento de diferentes competências técnicas e organizacionais.

Palavras-chave: *Estágio, Profissional, Postgres, Backend, Desenvolvimento, Docker, Java, Micronaut.*

Abstract

Relatório de Estágio

Desenvolvimento em Backend na Optiply

This report consists in the representation and documentation of the coursework of the Professional Internship, carried out as a part of the Bachelor Degree in Computer Science at the School of Technology and Management of the Institute of Technology of Beja.

The Internship was carried out at Optiply, in Évora, in the year of 2020/2021, with the aim of improving the integration and consolidation of the acquired knowledge, in the context of practice, the theoretical knowledge acquired throughout the Degree.

The primordial objective of the internship was to improve the integration in the world of work. The idea of being interned at Optiply came from the idea of improving the student's first contact with the area of development in Backend, and the possibility of developing personally in an environment of work that is compatible with what is intended to do.

The activities were developed taking into account the objectives initially outlined and that were: initial training via Udemy courses, the development of a project, planning and implementation of it.

This project was a development of a Backend, which was developed in Java, using the framework Micronaut, linked to a Postgres database, and was developed in a local development environment using Docker containers.

The learning was effective and perceptible, in the measure in which the activities were developed that provided the acquisition and the development of different technical and organizational competences.

Keywords: *Internship, Professional, Postgres, Backend, Development, Docker, Java, Micronaut.*

Índice

Índice de Figuras

Abreviaturas, Siglas e Acrónimos

API	Application Programming Interface
CLI	Command Line Interface
CRUD	Create Read Update and Delete
DAO	Data Access Object
HTTP	Hypertext Transfer Protocol
IPBeja	Instituto Politécnico de Beja
JDBC	Java Database Connectivity
JDK	Java Development Kit
jOOQ	jOOQ Object Oriented Querying *(Acrónimo Recursivo)*
JSON	JavaScript Object Notation
POJO	Plain Old Java Object
REST	Representational State Transfer
SDK	Software Development Kit
SQL	Structured Query Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator

Capítulo 1

Introdução

Este presente relatório tem como objetivo apresentar o decorrer do estágio profissional e as consequências ou resultados do mesmo, o qual ocorreu no período de 02/03/2022 a 02/06/2022, na cidade de Évora, Portugal. O estágio foi hospedado pela Optiply, que é uma empresa de gestão inteligente de *stocks* dos produtos e serviços de lojas online, cujo orientador (Fábio Belga) é o *Team Leader/Tech Lead*, que gere todo o processo de desenvolvimento e gestão do projeto.

O meu papel como estagiário foi um de treino para desenvolvimento em backend com um pequeno projeto, um *microservice* que realiza a gestão de especificações de lojas online. Este projeto envolveu variadas tecnologias e paradigmas de trabalho e de programação, os quais passam por diversas etapas de desenvolvimento, testes e documentação, mas no que toca à gestão e organização de projeto foi de escolha livre, ou seja, eu geria o meu tempo e o projeto à minha vontade sem vigilância ou controlo. O qual, admitindo a verdade, não geri o meu tempo de qualquer forma, apenas os objetivos de projeto em sí, num estilo primitivo de Kanban.

Com a leitura deste relatório, pretendo que gradualmente se expanda e detalhe o referido no paragrafo anterior, e que seja possível compreender o que foi aprendido e o que foi desenvolvido.

Na realização deste estágio, foram obtidos diversos conhecimentos que serão fundamentais para minha carreira profissional. Espero que este relatório possa contribuir para uma melhor avaliação do estágio e auxiliar na tomada de decisões futuras.

Capítulo 2

A Empresa

Figura 2.1: Logo da empresa

A empresa que me hospedou num estágio foi a Optiply, durante três meses, para uma posição de aprendizagem de desenvolvimento de backend.

2.1 Caracterização

A Optiply é uma empresa de software cujo produto/serviço é a gestão inteligente de *stocks* dos produtos e serviços de lojas online. Esta empresa foi fundada em 2015 em Amesterdão, Países Baixos e que depressa (em 2017), expandiu a sua equipa de desenvolvido de software para Évora, Portugal.

Em Évora, esta empresa detém por volta de 20 empregados, dos quais três quartos são engenheiros de software, distribuídos em frontend, backend e integrações.

2.2 Produto

Como referido brevemente na secção anterior esta é uma empresa de SaaS, ou seja um software vendido como um serviço, o uso deste software está associado a uma subscrição que oferece os serviços promovidos. Mais especificamente são serviços de inteligência artificial que fazem sugestões automáticas de compras e promoções para o cliente, facilitando a gestão dos stocks do seu armazém.

2.3 Organização e Comunicação

Apesar da empresa deter instalações e equipamentos, esta suporta trabalho remoto, o qual me pareceu ser a escolha da maioria dos empregados que estão em desenvolvimento de software, eu incluso. O qual para a Comunicação da empresa, inter e entre empregados era essencialmente via Slack, com canais de comunicação gerais, divididos em equipas, e comunicação privada. Evitava-se ao máximo o uso de video chamadas, e ao invés de isso, usava-se mensagem de texto, à exceção do *Tech Lead/Team Leader* que usava mensagem de texto e video chamada para coordenar as equipas locais e estrangeiras.

Para a gestão dos projetos, esta é uma empresa *Agile*, o qual obviamente usa a suite da Atlassian, nomeadamente o BitBucket para hospedar os repositórios de software, bibliotecas e frameworks, como também, o Jira.

2.4 *Tech Stack*

O serviço oferecido pela empresa foi desenvolvido por um conjunto de tecnologias, dos quais do tempo em que estive ativo consegui identificar que se usa a framework de Javascript: Angular para a construção do front-end o qual pode ser acedido num browser ou empacotado numa aplicação Electron, a qual comunica com um servidor, possivelmente Linux e *Debian-based*. Este servidor detêm variados containers Docker, os quais são a infraestrutura vital do backend do serviço, hospedando as bases de dados PostgreSQL e MongoDB, e os *microservices*, implementados nas frameworks de Java: Micronaut (as mais recentes) e Spring.

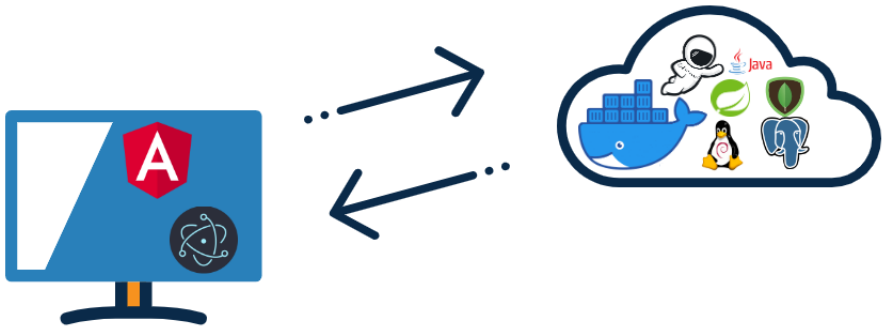

Figura 2.2: Diagrama exemplar do serviço

Capítulo 3

Onboarding

Aqui descrevo o meu breve processo de Onboarding. O qual passou pelo primeiro dia onde foi a parte administrativa, e posteriormente a parte formativa onde durante duas semanas estive a ser preparado para o projeto.

3.1 O que é Onboarding?

Em linhas gerais, onboarding trata-se de um processo para integrar o novo membro à equipa, cultura e forma de operação da empresa, com o objetivo de assegurar a adaptação e a retenção deste profissional.

É o processo de integração de novos empregados numa empresa, para que eles possam obter os conhecimentos, as habilidades e os comportamentos necessários a fim de efetivamente se tornarem parte da equipa.

Envolve várias etapas, que podem ser conjuntas ou separadas, como orientação, supervisão, acompanhamento e treinamento, por exemplo.

3.2 Fase Administrativa

Chamo de fase administrativa ao primeiro dia de estágio, este dia passou pela introdução informal do local de estágio, as regras de trabalho, as normas de segurança e o que é necessário para o bom desenvolvimento do projeto.

Foi também neste dia que me foi atribuído a conta empresarial (backend da Google), com email, senha e perfil de acesso, como também o que esta conta oferece (tal como drive, sheets, etc).

Essa conta base serve também para login na conta da Atlassian, onde tenho acesso a todas as ferramentas de estágio, como o Jira, Confluence, Bitbucket, etc.

Finalmente foi feita uma reunião, comigo e com todos os outros estagiários (três alunos da Universidade de Évora, sendo que um também é um estagiário de Backend), para nos passar à fase formativa, onde estarei a desenvolver competências base para o projeto.

3.3 Fase Formativa

Esta fase teve uma duração base de duas semanas. Aqui foi-nos doado três cursos do Udemy, gradualmente e squencialmente dependendo do progresso do estagiário.

Estes cursos tinham como objetivo ajudar ao estagiário a desenvolver competências para o projeto que será desenvolvido, o qual reflete as competências que o estagiário deve possuir para poder trabalhar no backend da empresa que está a trabalhar.

O primeiro curso foi *"Build Reactive MicroServices using Spring WebFlux/SpringBoot"*, que têm como objetivo ensinar a fazer backends de serviços, ou micro serviços em Spring, mas com uma particularidade: usar Spring WebFlux que é a implementação da Spring do Project Reactor. Isto permite fazer uma API reativa, ou seja, uma API que é capaz de receber requisições e retornar respostas de forma assíncrona e com a menor latência possível.

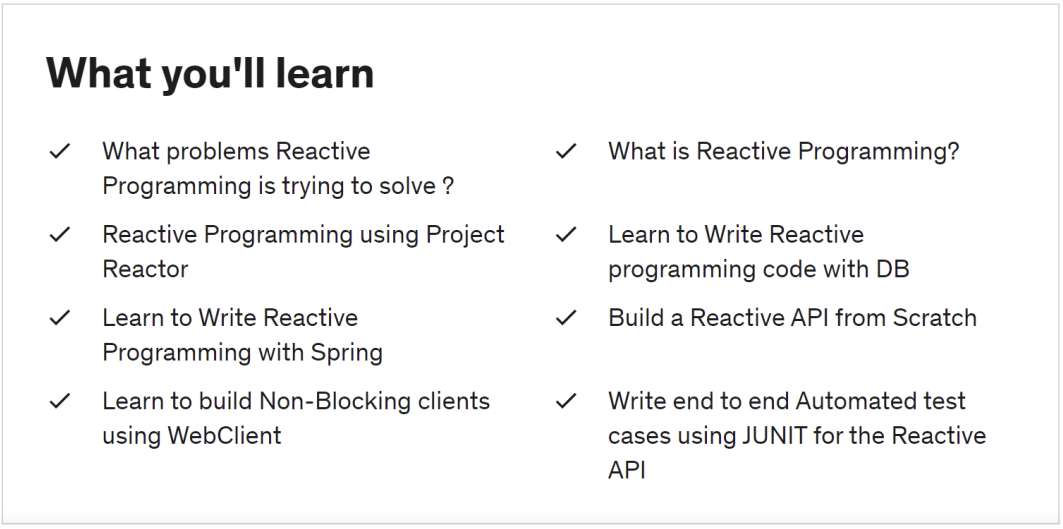

Figura 3.1: Conteúdo do curso *"Build Reactive MicroServices using Spring WebFlux/SpringBoot"* do Udemy

Seguidamente foi-me destacado o *"Microservices with gRPC [Java + Spring Boot + Protobuf]"*, que é um curso com o objetivo de ensinar ao estagiário a desenvolver *microservices* com gRPC em Java usando Protocol Buffers, ou seja, um serviço que usa a implementação de RPC da Google e usa o Protocol Buffers para o transporte de dados. Isto têm a vantagem de ser um serviço de baixo custo, e alta velocidade de comunicação.

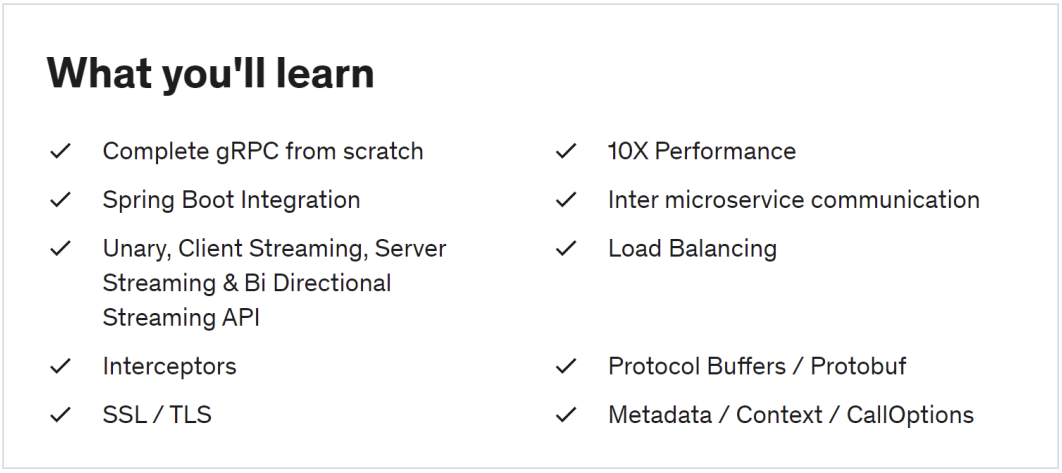

Figura 3.2: Conteúdo do curso *"Microservices with gRPC [Java + Spring Boot + Protobuf]"* do Udemy

O ultimo curso foi o *"Learn Micronaut - cloud native microservices with Java"*, que nos mostra como fazer *microservices* em Micronaut, um framework de Java, como o Spring mas com o objetivo de ser mais leve, modular e escalável. Este curso também passa pela integração do Apache Kafka, um *message broker* que permite a comunicação entre *microservices* e como exportar o projeto para um *native binary* e como usar o GraalVM, que é uma JVM de nova geração, mais leve e mais rápida que também suporta outras linguagens de programação.

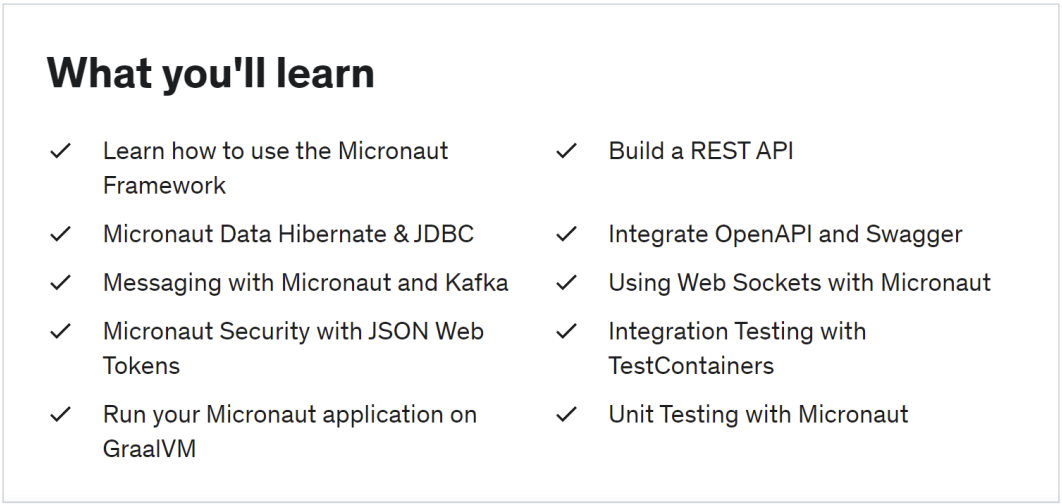

Figura 3.3: Conteúdo do curso *"Learn Micronaut - cloud native microservices with Java"* do Udemy

Capítulo 4

Desenvolvimento do projeto

Este capítulo descreve o projeto atribuído no estágio e o seu desenvolvimento. Sendo que este capítulo será o mais longo, mas consequentemente o mais importante e o mais complexo.

4.1 Introdução

Como anteriormente referido foi me destacada a tarefa de Implementação de um projeto no estágio. Este projeto consiste em um software que permite a gestão das especificações de Webshops.

Estas Webshops são, como o nome indica, as lojas online as quais são clientes da Optiply. Estas lojas online são responsáveis por fornecer os produtos que os clientes compram e a Optiply é responsável por fornecer a gestão inteligente dos produtos em stock.

O trabalho foi recebido num *.pdf*, numa reunião de video-conferencia, com o coordenador do estágio (Fábio Belga), o seu subordinado (André Figueira)que ficou encarregado de orientar os estagiários de Backend, e nós (eu, Gonçalo Amaro e o estagiário da universade de Évora, José Azevedo), após a nossa fase formativa do *Onboarding* descrita no capítulo anterior.

Assim, as primeiras secções deste capítulo servem como uma apresentação do equivalente à minha introdução ao projeto.

4.2 Objetivos

O objetivo descrito deste projeto é desenvolver um microserviço que permita a gestão das especificações de Webshops, já o verdadeiro objetivo deste projeto é fornecer treino ao estagiário nas tecnologias da *Tech Stack* da empresa, ou pelo menos num dos projetos da mesma.

Essa *Tech Stack* referida é a seguinte:

- Micronaut: Framework de desenvolvimento de microserviços.
- Java: Linguagem de programação.
- Gradle: Sistema de gestão de dependências e tarefas.
- jOOQ: Framework de código-fonte para acesso a bases de dados.
- Flyway: Framework de migração de bases de dados.
- PostgreSQL: Sistema de bases de dados.
- Junit5 (Spock também é aceitável): Framework de testes.
- Mockito: Framework de auxiliar a testes via simulação.

Voltando ao objetivo escrito do projeto (desenvolver um microserviço que permita a gestão das especificações de Webshops), o objetivo é desenvolver uma RESTful API que permita gerir as especificações de Webshops.

Para isso temos de saber que cada Webshop têm um conjunto de especificações, as quais são:

- *URL:* URL da loja online, têm validação e requer protocolo na URL;
- *Handle:* identificador único da loja online;
- *Interest Rate:* taxa de juros que a loja online paga, 20% é o valor por defeito;
- *Service Level Categories:* categorias de níveis de serviço que a loja têm, são três categorias (A,B e C) e as suas somas requerem ser iguais a 100%;
- *Contact Email List:* lista de emails de contacto da loja online, têm validação;
- **Extra:** *Settings:* configurações da loja online:
  - *Enable Multi Supplier:* permite múltiplos fornecedores;
  - *Enable Run Jobs:* permite execução de tarefas;
  - *Currency:* moeda da loja online em ISO-4217;

Sendo que as ultimas especificações (as *Settings*) são Extras, ou seja, não são obrigatórias, mas foram implementadas.

Essa API tem um determinado conjunto de tarefas a cumprir as quais são:

- Obter uma única Webshop;
- Obter várias Webshops:
  - Deve ser capaz de ordenar e filtrar por qualquer campo da tabela;
  - Só é necessário ordenar por um único campo. Os resultados devem ser consistentes com cada pedido. (Se ordenar por Taxa de Juros, como pode-se garantir que os mesmos resultados sejam obtidos em todos os pedidos?)
  - Só é necessário filtrar por um único campo. Os filtros suportados são:
    - ":" significa *Igual*. Exemplo: handle:optiply
    - "%" significa *ILIKE* (semelhante, *case-insensitive*).
      Exemplo: handle%optiply
    - **Extra:** ">" significa *Maior Que*. Exemplo: interestRate>20
    - **Extra:** "<" significa *Menor Que*. Exemplo: interestRate<20
- Apagar uma única Webshop.
- Criar uma única Webshop.
- Atualizar qualquer campo da Webshop.
- **Extra:** Filtrar por múltiplos campos.
- **Extra:** Criar múltiplas Webshops.
- **Extra:** Obter as configurações da Webshop.
- **Extra:** Atualizar as configurações da Webshop.

Tendo sempre em conta que os resultados devem ser idempotentes e no seu estado mais recente e que os pedidos HTTP retornam:

- Criar deve retornar 201.
- Obter e Atualizar devem retornar 200.
- Apagar deve retornar 204.
- Qualquer pedido deve retornar 404 se a loja não existir.
- Qualquer outro erro interno deve retornar 500 (Erro Interno).

Esta lista (tradução do que está no *.pdf* recebido, que está também no Apêndice I), é bastante extensa, mas é bastante simples para entender o que é.

No entanto é estupidamente obscura a segunda intenção da lista, esta era a lista implícita de *endpoints* da API.

O qual inicialmente não vendo uma lista de *endpoints* explicita nem uma mera referência na reunião, a primeira iteração do trabalho usei os *endpoints* que eu achava mais convenientes para o trabalho. Escusado será dizer, que tive de os refazer após a primeira receção de *feedback*.

4.3 Implementação

4.3.1 Pré-Requisitos

Para começar a implementar a API, precisamos de um conjunto de ferramentas. Essas ferramentas passam por um JDK (um SDK de Java), otimamente algo aberto e conforme os standards de OpenJDK, o qual usei Amazon Corretto, visto à sua licença aberta e gratuita, multiplataforma e vem com suporte de longo prazo que incluirá melhorias de desempenho e correções de segurança.

Para gestão de pacotes e tarefas, precisamos de um gestor de pacotes, o qual usei o Gradle, e como um dos meus computadores de trabalho usa em vez de Linux, a instalação do Gradle sem um gestor de pacotes e alteração do path, dá-nos jeito usar um IDE que trate desses assuntos, o qual foi-me recomendado (e usado): IntelliJ IDEA da JetBrains.

Para hospedar a base de dados e o projeto, numa pequena rede de containers interna, foi instalado o Docker no *desktop* Windows (e usado o Podman no portátil Linux, pelo simplesmente facto de já o ter instalado previamente).

No entanto ainda nos falta algo bastante importante. Nomeadamente, algo quer faça o Bootstrap do projeto em Micronaut, para isso temos variadas opções:

- Ir ao o Micronaut Launch Website
- Usar o Micronaut CLI em que temos aqui a documentação
- Fazer curl à API do Micronaut Launch https://launch.micronaut.io/create/default

4.3.2 Inicio do Projeto

No meu caso em especifico foi-me fornecido um repositório privado no BitBucket, o qual apenas me foi necessário fazer uma *fork*. O estado desse repositório e da *fork* pode ser visto no neste *commit* (num repositório meu do GitHub, onde no projeto o adicionei como segunda origem, para backup).

Esta diretoria de projeto nos atribuída, pessoalmente achei que era maior e mais complicada que o necessário, talvez esta seja única e o que varia são os projetos que a usam. Com isso em conta eu decidi, fazer uma redução ao projeto, para que ficasse mais simples de trabalhar e não houvessem pacotes ou funcionalidades que não fossem necessárias. Isto pode ser observado neste *commit* (o qual descrição reflete o meu estado mental sobre determinada observação).

Após a redução, o projeto ficou com apenas dois subprojetos para o Gradle gerir, um que contem a aplicação em sí e o outro que trata dos repositórios/classes de transações à base de dados. O numero de pacotes externos e funcionalidades foi reduzido para o mínimo necessário, esses incluíram: Flyway, Jackson, jOOQ, JUnit, Logback, Lombok, Mockito, Postgres, R2DBC e Reactor.

Detalhes sobre as tecnologias

Flyway

O Flyway é um framework de migrações de bases de dados, que é usado para gerênciar as migrações de bases de dados de projetos Java. Funciona de maneira semelhante às migrações nativas do ASP.NET Core.

Este pacote adiciona essas capacidades a tarefas do Gradle, como o *flywayMigrate* e *flywayInfo*. A migrações são feitas através de um ficheiro de migrações, que é um ficheiro de SQL, dentro da diretoria de migrações (*PROJECT_ROOT/src/main/resources/db/-migrations*), com a versão em que a migração deve ser executada e dois *underscores*.

Jackson

O Jackson é um framework de serialização de objetos, que é usado para serializar objetos em JSON.

A serialização é um processo de transformação de um objeto em um JSON, e a desserialização é o processo de transformação de um JSON em um objeto.

Isto é feito principalmente através de um objeto *ObjectMapper*, que é um objeto que implementa a interface *com.fasterxml.jackson.databind.ObjectMapper*.

jOOQ

O jOOQ é um framework de código-fonte de código-aberto, que é usado para gerir a base de dados. Este funciona de maneira semelhante às operações do Entity Framework Core para ASP.NET Core, sendo que este abstrai as operações de SQL em wrappers programáticos.

JUnit

O JUnit é um framework de testes, que é usado para gerênciar os testes de unidades. Usado muito na disciplina de Programação Orientada a Objetos.

Logback

O Logback é um framework de logging, que é usado para gerir os logs de um projeto, com o foco em abstrair o uso de logs ao mais simples possível. É o sucessor do Log4j, que foi alvo de uma vulnerabilidade recentemente.

No Windows devemos alterar uma configuração: a desativação do JANSI, que não funciona com alguns Locales, em especial os que o Windows usa.

Lombok

O Lombok é um framework de código-fonte de código-aberto, que é usado para gerir a criação de classes de objetos através de anotações. Com estas anotações, abstraímos o código, evitamos repetição e automatizamos muito o processo desenvolvimento.

Por exemplo a anotação *@Getter* faz com que o Java crie automaticamente os getters. Ou, a anotação *@Data* faz com que o Java crie automaticamente os getters, setters, equals, hashCode, toString e clone.

Mockito

O Mockito é um framework auxiliar de testes, que é usado para gerir os mocks de objetos. Os mocks são objetos que são usados para simular o comportamento de objetos realmente existentes.

Com os mocks, podemos testar objetos que ainda não existem, como um objeto de um repositório de dados, ou um objeto de um serviço. Como também isolamos o comportamento dos objetos, evitamos que os objetos sejam alterados durante o teste ou para testar apenas o comportamento do que comunica com o mesmo.

PostgreSQL

O PostgreSQL é um driver JDBC, que é usado para conectar a bases de dados PostgreSQL. Um driver JDBC é um driver que permite ao Java a comunicação com bases de dados.

PostgreSQL é o tipo de base de dados usado no projeto.

R2DBC

O R2DBC é um driver de conexão à base de dados, mas contrariamente ao anterior este permite fazer transações reativas como as do Project Reactor ou do RxJava.

Reactor

O Reactor é um framework de eventos, que é usado para gerir eventos e criar aplicações reativas. Uma aplicação reativa é uma aplicação que é executada em um fluxo de eventos.

4.3.3 Paradigma de Programação

Programação reativa é o acto de programar para trabalhar com fluxos de dados assíncronos. Isto é importante devido o crescimento da Internet e a demanda enorme de dados em tempo real. Esta programação precisa de ser dinâmica, ou seja; diferente das formas tradicionais de desenvolvimento.

Nas formas tradicionais de programar/desenvolver, de modo **muito** genérico, cria-se variadas tarefas e elas comunicam-se em tempos pré-determinados, com respostas pré-determinadas, são "rígidas", seguem regras diretas.

Isto funciona e continua a ser utilizada até hoje, entretanto esta "lógica" não é compatível com as necessidades de alguns serviços atuais e os seus inúmeros clientes e dados. Na programação reativa isto ocorre de uma forma semelhante, mas mais inteligente, interligada em paralelo, sem seguir uma ordem cronológica e linear.

Os pilares da programação reativa são:

- **Elástico:** Reage à demanda/carga: aplicações podem fazer uso de múltiplos núcleos e múltiplos servidores;
- **Resiliente:** Reage às falhas; aplicações reagem e se recuperam de falhas de software, hardware e de conectividade;
- **Message Driven:** Reage aos eventos (event driven): em vez de compor aplicações por múltiplas threads síncronas, sistemas são compostos de gerenciadores de eventos assíncronos e não bloqueantes;
- **Responsivo:** Reage aos usuários: aplicações que oferecem interações ricas e "tempo real" com usuários.

4.3.4 Estrutura do Projeto

A estrutura do projeto, como dito anteriormente, foi uma modificação do herdado da estrutura inicial vinda do repositório oferecido para *forking*. Este projeto consitis de um projeto Gradle com três subprojetos: um pacote com classes entendidas de Monos, um pacote para o core do projeto, e um pacote para os repositórios que tratam das transações com a base de dados.

Sendo que as classes eram apenas *MonoVoid*, *MonoFalse* e *MonoTrue* que simplesmente implementavam a interface *Mono* e retornavam um valor booleano (ou nenhum), decidi cortá-las visto que não trazem quais quer nova funcionalidade ao projeto e não me custa escrever *Mono<Boolean>* e retornar um valor booleano.

Com isto, a estrutura do projeto foi alterada para um projeto Gradle com dois subprojetos.

4.3.5 Metodologia de desenvolvimento

Foi-me notificado que o projeto seria desenvolvido de forma livre, sem qualquer metodologia de desenvolvimento. No entanto, sendo eu um alguém novo na area e a trabalhar remotamente, decidi que é extremamente importante arranjar um ambiente de desenvolvimento que me permita trabalhar, ponto. Por isto, decidi referir às estratégias de gestão de projeto ensinadas na disciplina de Engenharia de Software, como o *Scrum* e o *Kanban*.

Sabendo que a empresa onde estou usa *Scrum*, ponderei usar o mesmo e as ferramentas disponíveis no Atlassian como *Jira* e *Confluence*; mas acabei por decidi usar uma estratégia menos rígida, o *Kanban*.

Kanban

Para o meu projeto, a estratégia de desenvolvimento foi o *Kanban*, ou pelo menos uma forma primitiva do mesmo. Expandindo, foi feito um quadro de tarefas divididas em cinco colunas: *Tarefas*, *Por aprender*, *Por implementar* *Por testar* e *Terminadas*.

A metodologia de trabalho começava por ir identificando tarefas, se muito complexas dividi-las em pequenas tarefas e julgando a minha capacidade de as fazer. Colocando na coluna respetiva e depois trabalhando de acordo com o estado do quadro.

Este quadro infelizmente já não está acessível, pois a conta empresarial já foi fechada.

Nota sobre a escolha

Na minha opinião subjetiva, a escolha do *Kanban* sobre *Scrum*, foi uma boa decisão, visto que o *Scrum* é um padrão mais rígido e linear, havendo a (extra) necessidade em criar *user-stories* e em definir *sprints*. Já o *Kanban* é um padrão mais flexível, tanto por ter menos etapas para organizar como por ter um quadro de tarefas menos *standard* e mais flexível.

4.3.6 Testes

Para testar o software, foi recomendado uma mistura de *JUnit* e *Mockito* ou usar *Spock*. Estes testes foram feitos dentro do subprojeto principal, e foram divididos em duas partes:

- *Testes de unidade*
- *Testes de integração*

Houve também uma secção chamada *shared*, onde havia um conjunto de classes que orientava o pacote do *TestContainers* e o seu *container* de teste *PostgreSQL*.

Noto que uma dos requerimentos do projeto era que um pacote do Gradle, chamado JaCoCo e que verifica a percentagem de código testado, fosse incluído no projeto e o resultado mínimo obtido fosse de 80%. O que foi feito e entregue, com 82% na entrega final e que houve uma altura que foi entregue com uns 100% de coverage (na secção seguinte 4.3.6 saberão porquê), no então noto também que este relatório refere-se principalmente ao estado da entrega final.

Testes de unidade

Os testes de unidade são testes que testam uma unidade do software, isto quer dizer que testam algo em isolamento do resto do software. Este tipo de testes permitem verificar o correto funcionamento daquela especifica classe ou função, assim permitir identificar (ou excluir da procura) *bugs* ou erros no software.

Foram feitos testes de unidade para as classes dos modelos (os objetos com que comunicamos) e para as classes de serviços.

Testes de integração

Os testes de integração são testes que testam o funcionamento do software como um todo. Este tipo de testes permitem verificar o correto funcionamento do software ou permitir identificar se existe algum *bug* no software, sabendo também se esse erro está na integração das unidades se em conjunto com testes de unidade sem testes falhados.

Foi feita uma serie de testes de integração para a classe do *controller*, que é responsável por receber os *requests* à API e assim testando o funcionamento da API num todo.

4.3.7 *Feedback*

O *feedback* sobre o desenvolvido seria feito sob pedido via Slack ao responsável sobre os estagiários de backend, o André Figueira.

De forma concreta eu obtive dois sets de *feedbacks*:

- *Feedback* da primeira entrega
- *Feedback* da segunda entrega (final)

Feedback da primeira entrega

Este foi o *Feedback* mais volumoso, que combinou comentário sobre *Clean Code* e os conceitos *SOLID*, comentário sobre os verbos dos métodos *HTTP* e comentário sobre ler as inferências do *.pdf* do projeto.

Começando de trás para a frente, o primeiro comentário foi sobre os *endpoints*. Ou seja, as ações que o software pode realizar, descritas no documento, inferem os endpoints que o software deve ter e não algo que apenas os satisfaça, lembrar o que foi dito no final da secção 4.2.

Seguinte, os caminhos da URI, para os simplificar o mais possível, não precisam de conter os verbos das ações que fazem, visto que sendo ações CRUD, descritas facilmente com os métodos HTTP (GET, PUT, POST, DELETE), podem e devem ser cortados ao máximo.

Exemplo: `HTTP DELETE -> http://localhost/remove/{id}` passar para `HTTP DELETE -> http://localhost/{id}`.

Por último, e não menos importante, os conceitos *SOLID* devem ser seguidos ao máximo independentemente do que achamos que vai ser o rumo do software. Isto porquê? Eu achei que visto que não iriam haver mais do que uma interface de comunicação com o software, sendo apenas necessário um *controller* que serve a comunicação REST HTTP, não seria necessário uma outra classe com o *business logic*, ou seja uma classe de serviço onde *controllers* iriam buscar. Isto estava errado mesmo que a lógica sobre esta decisão não estivesse muito errada. O software deve separar o *business logic* do *controller* o mais possível e o software deve ser extensível.

Feedback da segunda entrega (final)

O *feedback* desta entrega foi muito mais simples, visto que todos os pontos anteriores foram corrigidos, o *feedback* foi simplesmente que o software estava satisfatório com o que foi pedido se bem que a documentação poderia ter sido um pouco mais extensa.

Capítulo 5

Projeto desenvolvido

Este capitulo descreve em detalhe o funcionamento do projeto desenvolvido durante o decorrer do estágio.

5.1 Funcionamento

5.1.1 Descrição geral

O *Webshop Service Specification* é uma RESTful API reactiva, que consiste em gerir as especificações de Webshops, ou seja um microserviço reactivo. Este microserviço recebe pedidos HTTP e retorna uma resposta JSON com o resultado, nomeadamente uma Webshop ou uma caraterística da mesma.

Sendo esta API reactiva, ela emprega o uso de *threading* (divisão de tarefas em subprocessos) para poder executar variados pedidos em simultâneo, o mais depressa possível. No entanto esses processos concorrentes e assíncronos requerem um outro nível de cuidado e atenção no que toca à integridade e idempotência dos dados requeridos.

As relações empregues por uma aplicação reactiva são os padrões de *Publisher/Subscriber*, onde um pedido, ou uma transação, é uma mensagem enviada pela fonte desses dados, chamada de um *Publisher* e a sua receção, ou seja onde os dados são consumidos, é encarregado pelo(s) *Subscriber(s)*. Se um desses pedidos for uma mensagem com varias subscrições ao longo do tempo, devemos alterar o *scheduling*, que gere as filas de processos e acessos.

Com isto podemos dizer que os *endpoints* desta API são *Subscribers* e o serviço transacional que comunica diretamente com os dados da base de dados é o nosso *Publisher*. Este tipo de acessos reactivos à base dados requer um outro tipo de mecanismo de processo de transações SQL, o qual deve ser também reactivo de modo a que a base de dados seja vista e funcione com um *Publisher* e seja configurável a sua propagação.

Pegando na excelente descrição anterior, o funcionamento deste microserviço pode ser reduzido ao seguinte *flowchart*:

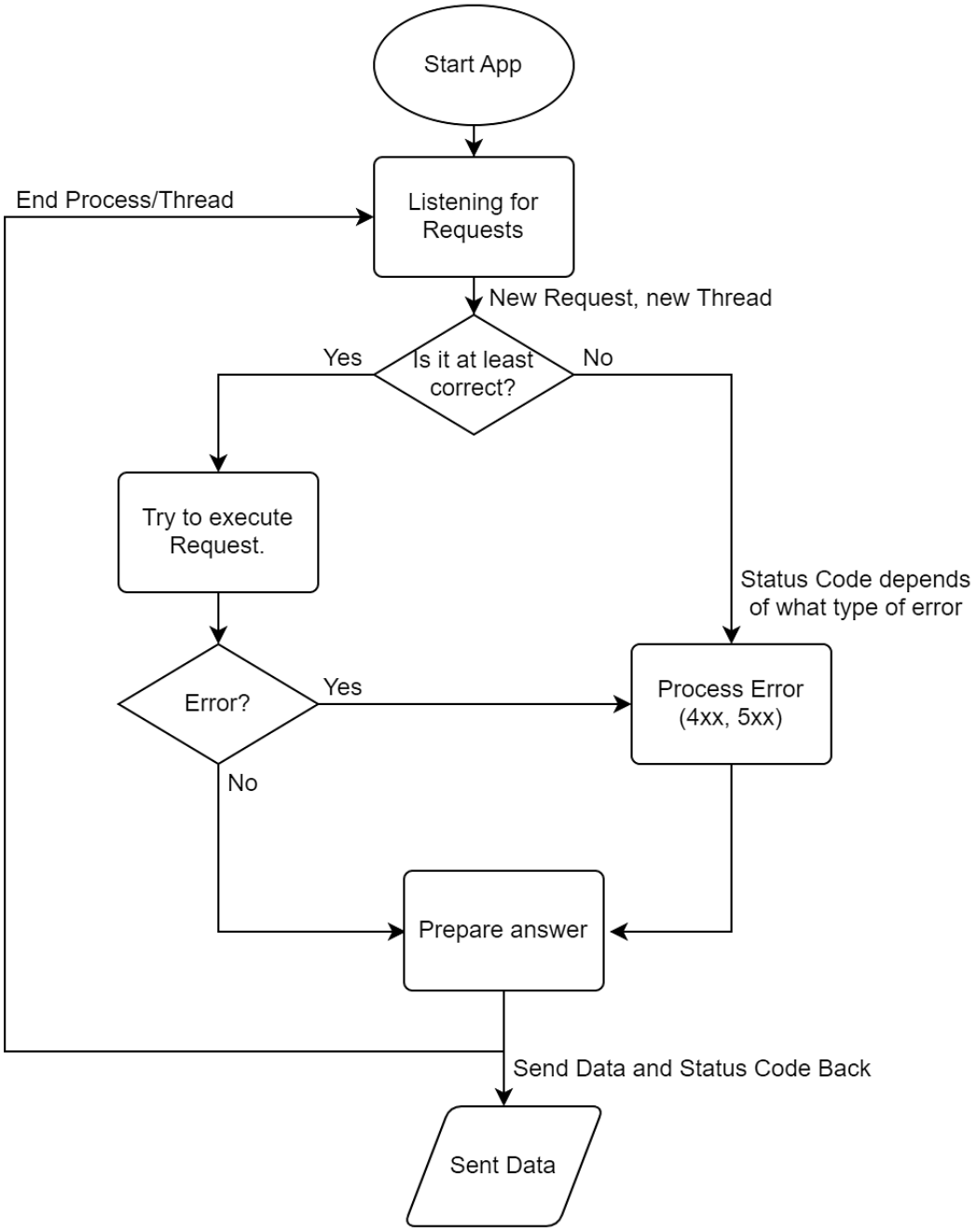

Figura 5.1: *Flowchart* do funcionamento do microserviço, criado no GitMind

Cada um destes processos descritos no *flowchart* (os quadrados), é um objeto ou classe. Os principais vão ser seguidamente descritos com maior detalhe, nas secções seguintes.

5.2 Organização

Este projeto (como anteriormente mencionado), é um projeto Gradle que contem dois subprojetos, os quais gerem tarefas diferentes mas co-dependentes:

- O pacote dos *Endpoints*: um pacote para o core do projeto, este contem a estrutura MC do projeto, com os respectivos modelos, controller e o serviço que comunica com os repositórios do pacote seguinte;
- O pacote da *Infrastructure*: pacote para os repositórios que tratam das transações com a base de dados e as classes geradas do jOOQ que os repositórios utilizam.

Dentro da *root directory* do projeto contemos variadas subdirectorias e ficheiros, dos quais podemos apontar:

- `build/ ->` diretoria onde o Gradle gera os binários, os *jars*, artefactos, etc. . . da tarefa de compilação;
- `endpoints/ ->` *source directory* do subprojeto *Endpoints*;
- `gradle/ ->` diretoria onde existem os *wrappers* do Gradle;
- `infrastructure/ ->` *source directory* do subprojeto *Infrastructure*;
- `javadoc/ ->` diretoria onde se gera o JavaDoc, ou seja um documento/*website* com a documentação (derivada dos *block comments*);
- `build.gradle ->` ficheiro de configuração **principal** do Gradle, onde definimos os pacotes a ir buscar e programamamos as tarefas de (pré e pós) compilação e de testes;
- `gradle.properties ->` ficheiro de configuração **opcional** do Gradle onde se definem *compiler flags*, argumentos para a JVM e outras configurações mais profundas e especificas;
- `lombok.config ->` ficheiro de configuração **opcional** do Lombok, onde aqui defino para adicionar a anotação `@Generated` as suas classes geradas para *fugir* ao JaCoCo;
- `micronaut-cli.yml ->` ficheiro de preferências da criação de um projeto Micronaut via a sua ferramenta CLI;
- `postgres-compose.yml ->` ficheiro de **docker-compose** para compor e lançar containers com pré configurações, neste caso um container de PostgreSQL;
- `settings.gradle ->` ficheiro de configuração do Gradle onde se definem os subprojetos do projeto Gradle, é executado a cada *build task*.

5.2.1 *Endpoints package*

Este é o subprojeto *Endpoints* onde contêm toda a estrutura base da API, deste o ponto inicial da aplicação, às configurações, modelos, controladores e serviços. Tendo em conta que `com/optiply/endpoint/` fica como reticências, temos que:

- `src/main/`
  - `java/`
    - `.../config/`
      - `DataSourceConfig.java`
    - `.../controllers/`
      - `shared/interfaces/IBaseController.java`
      - `shared/BaseController.java`
      - `JSONController.java`
    - `.../models/`
      - `EmailListModel.java`
      - `HandleModel.java`
      - `InterestRateModel.java`
      - `ServiceLevelsModel.java`
      - `SettingsModel.java`
      - `UrlModel.java`
      - `WebshopFullModel.java`
      - `WebshopModel.java`
      - `WebshopSettingsModel.java`
    - `.../services/`
      - `RepositoryService.java`
    - `.../EndpointApplication.java`
  - `resources/`
    - `db/migration/`
      - `V1__create_initial_schema.sql`
    - `application.yaml`
    - `bootstrap.yaml`
    - `logback.xml`

- `src/test/`
  - `java/`
    - `integration/`
      - `.../controllers/JSONControllerIntegrationTests.java`
    - `shared/`
      - `.../container/TestContainer.java`
      - `.../environment/TestEnvironment.java`
    - `unit/`
      - `.../models/WebshopFullModelsUnitTests.java`
      - `.../services/RepositoryServiceUnitTests.java`
  - `resources/`
    - `application-test.yaml`

Sendo que é clara a funcionalidade de cada classe pelo nome e pelo local onde se encontra. Mesmo sendo esse o caso, seguimos para uma explicação breve do que cada classe ou ficheiro faz, excluindo as classes de modelos, pois são obviamente modelos dos objetos transacionais (os corpos em JSON do *request* HTTP) ou de suas partes.

`DataSourceConfig.java`

Esta classe executa o carregamento e pós-configuração das configurações do ficheiro de configuração `application.yaml`, criando o contexto DSL (uma interface de comunicação do jOOQ com a base de dados via JDBC), e consequentemente cria a *ConnectionFactory*, que permite usar R2DBC e fazer queries transacionais de forma reactiva.

`JSONController.java`

Esta classe é um controlador de *requests* HTTP com *payloads* em JSON. Esta, extende a classe de controlador base `BaseController.java` (abstrata), que por si é uma implementação da interface `IBaseController.java`, que contem as funções der *parsing* dos parâmetros de procura e sorteamento do *endpoint* de pesquisa de Webshops.

`RepositoryService.java`

Esta classe é responsável por deter toda a *business logic* necessária e acessível pelos *controllers* e que os isola de contacto direto com os repositórios de dados. É aqui que se executam as tarefas que queremos executadas e recebemos os resultados quando usamos os *endpoints* do *controller*.

`EndpointApplication.java`

Classe principal/base de onde executa a aplicação.

`application.yaml`

Ficheiro de configurações da aplicação (referido anteriormente `DataSourceConfig.java`). É aqui onde temos configurações da framework, como o uso do FlyWay, do jOOQ, configurações da fonte de dados (base de dados) e dos meios como lhe comunica (JDBC e R2DBC).

`application-test.yaml` *Testes*

Semelhante ao anterior, em funcionalidade e não só em nome, são as configurações especificas a ser usadas quando executamos testes. Assim podendo escolher meios de segregar ambientes e containers de teste, ou ajustar recursos de sistema. Neste caso, foi para usar containers de teste novos por cada novo *set* de testes, via TestContainers.

`TestContainer.java` *Testes*

Classe que configura o lançamento de uma nova instância de um container PostgreSQL (sobre TestContainers) para ser usado como container de testes.

`TestEnvironment.java` *Testes*

Classe que abstrai o funcionamento da classe anterior quando esta for instanciada qualquer classe de testes que a extenda. Sendo que todas as classes de testes extendem esta classe, ou seja, estão sobre o mesmo ambiente de testes.

`JSONControllerIntegrationTests.java` *Testes*

Para fazer os testes de integração, testes que testam todo um funcionamento ou percurso não isolado de um processo do projeto, apenas precisamos de fazer testes ao *controller*. Fazendo com que este esteja a receber pedidos HTTP e a escutar as respostas que ele dá, verificando se os resultados obtidos são os esperados.

`WebshopFullModelsUnitTests.java` *Testes*

Qualquer processo que ocorra nesta aplicação, é um processo de *messaging*, em que o conteúdo das mensagens é dados de um modelo ou o modelo em si, se queremos ter a certeza que estas mensagens ocorrem de forma esperada temos de testar os modelos que elas comunicam. Esta classe faz testes unitários a cada modelo incluso neste projeto.

`RepositoryServiceUnitTests.java` ***Testes***

O serviço neste projeto é o *middleware* que ofusca o *business logic*, como também serve como a comunicação entre os pontos de entrada e o repositório dados que contem o que é pedido, aqui estão definidos os fluxos reactivos das tarefas a fazer. É preciso fazer testes unitários a este serviço, usando o Mockito para imitar comportamentos de classes e objetos que este comunique para isolar o *scope* dos testes.

`V1__create_initial_schema.sql`

Ficheiro `.sql` com o esquema inicial da base de dados, onde o Gradle executa uma tarefa com o pacote FlyWay para fazer a migração. Esta base de dados contêm duas tabelas, uma com Webshops e as suas caraterísticas e outra para os emails de contacto das Webshops, relação de um-para-muitos.

5.2.2 *Infrastructure package*

O subprojeto do *Infrastructure* é onde toda a lógica de comunicação com a base de dados está instalada, desde as classes autogeradas do jOOQ aos repositórios, que são as classes que visam o isolamento e abstração das transações SQL com os que requerem que sejam executadas, nomeadamente o serviço no subprojeto anterior.

- `src/main/java/.../data`
  - `repositories/`
    - `interfaces/`
      - `IWebshopemailsRepository.java`
      - `IWebshopRepository.java`
    - `WebshopemailsRepository.java`
    - `WebshopRepository.java`
  - `support/sql/`
    - `QueryResult.java`
  - `package-info.java`
  - `resources/`
    - `jooq_schema.sql`

Aqui contem artefactos peculiares, dos quais interfaces para cada Repositório, uma classe de Enums, um ficheiro SQL e um *package-info*. Estes vãos ser seguidamente explicados, tal como no capitulo anterior.

`IWebshopRepository.java`

Interface para implementação do repositório que faz as operações CRUD da tabela de Webshops. Esta interface permite ao Mockito interpretar o comportamento a imitar da classe `WebshopRepository`.

`IWebshopemailsRepository.java`

Semelhante à anterior mas para a tabela de emails das Webshops.

`WebshopRepository.java`

Classe de operações CRUD reactivas, esta é a classe usada pelo serviço quando necessita de fazer operações na tabela de Webshops.

`WebshopemailsRepository.java`

Semelhante à anterior mas para operações na tabela de emails das Webshops.

`QueryResult.java`

Enum de resultados das queries SQL, permitindo fazer operações de equivalência a *result codes* e *status*.

`jooq_schema.sql`

Ficheiro com o esquema da base de dados em SQl para que o jOOQ possa criar todas as classes e sistemas referentes à base de dados que é usada. Desde POJOs para usar como objetos transacionais como DAOs e *Specific Result Sets*.

`package-info.java`

Classe que é interpretada pela framework e que têm apenas como objetivo obrigar a que os parâmetros seja *Nullable by default*. Esta configuração foi herdada pelo estado inicial do projeto.

5.3 Código

Nesta seguinte secção vai ser vista em maior detalhe algumas classes e o seu funcionamento especifico, como também explicada algumas decisões po detrás das escolhas feitas sobre o que foi (e como foi) construído.

5.3.1 *JSON Controller*

Esta classe de controlador estende o controlador base que por sí é uma implementação do da seguinte interface:

```
/**
 * Interface for a base controller.
 */
public interface IBaseController {

        /**
         * Parse params and return a condition.
         *
         * @param params the params
         * @return the condition
         */
        Condition parseParamsWebshop(String... params);

        /**
         * Sort parser for the webshops.
         *
         * @param sort  the sort
         * @param order the order
         * @return the sort field
         */
        SortField<?> sortParserWebshop(String sort, String order);
}
```

Sendo que estes dois métodos são apenas precisos para a função de pesquisa, porém esta sendo um ponto crucial e também o mais importante do micro serviço, estará sempre como requerimento de qualquer possível implementação de um controlador, quer seja ele de *requests* HTTP, gRPC ou quais quer outros.

O `parseParamsWebshop` faz a gestão de toas as condições exposats na URL de pesquisa, tais como *"nome igual a"*, *"nome semelhante a"* ou *"taxa de juro maior/menor que"*. Já a `sortParserWebshop` indica qual a ordem de chegada dos resultados, sendo que podem

de forma ascendente ou descendente (numérica ou alfa-numericamente), ser ordenados por quais quer das suas colunas.

As implementações desses métodos estão nos anexos um e dois. A classe implementação também Injeta dois objetos que qualquer classe extensa de esta base precisa: o acesso ao serviço e o `ObjectMapper` do Jackson.

```
 /**
 * The repository service.
 */
@Inject
public RepositoryService repositoryService;
/**
 * Jackson object mapper.
 */
@Inject
public ObjectMapper objectMapper;
```

Em suma, este controlador funciona da seguinte forma:

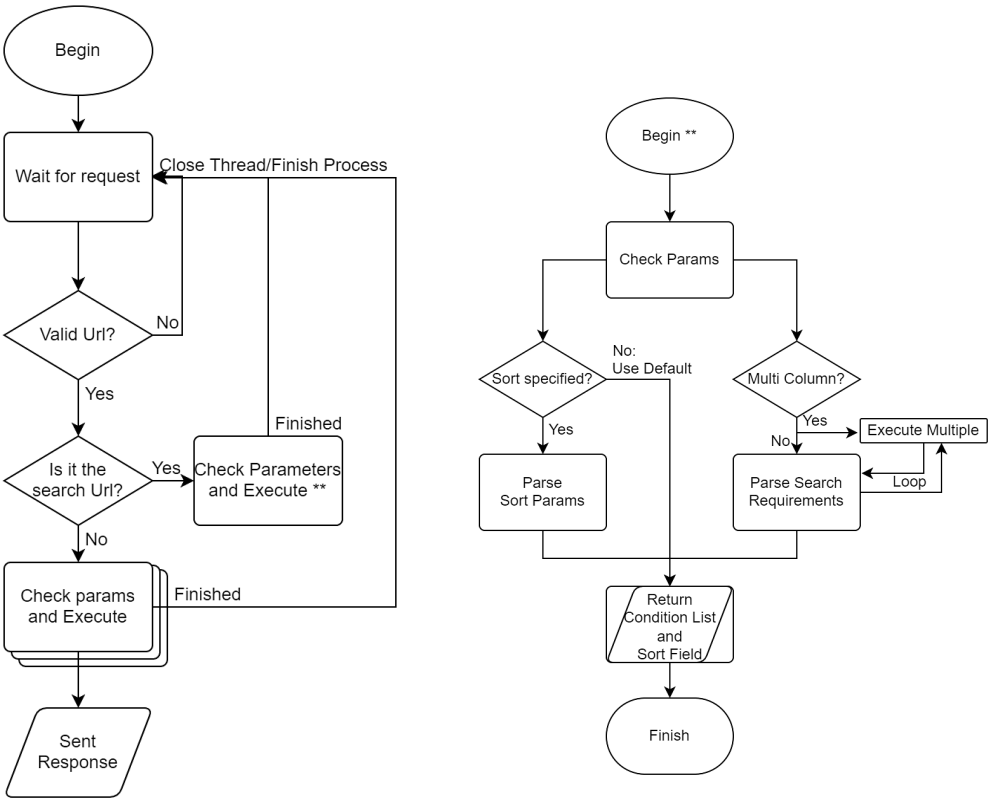

Tabela 5.1: *Flowchart* do funcionamento geral do controlador, criado no GitMind

5.3.2 *Webshop Service*

Esta classe têm como objetivo abstrair a lógica de negocio do controlador, assim sendo, esta implementa um método para cada endpoint que possa haver no controlador, em que cada um destes métodos é um fluxo de dados (`Mono` ou `Flux`), que direcione os dados obtidos num fluxo seguro e previsível desde os repositórios de dados à resposta possível que é enviada. Exemplo:

```
/**
 * Finds webshops via a conditionally defined
 * query sorted by a defined sort field.
 *
 * @param condition the condition
 * @param sortField the sort field
 * @return the webshops
 */
public Mono<MutableHttpResponse<List<WebshopModel>>>
     getWebshops(Condition condition, SortField<?> sortField) {

 return webshopRepository.findVarious(condition, sortField)
             .flatMapSequential(this::getWebshopPriv)
     .collectList().flatMap(webshops -> {
       if (webshops.isEmpty()) {
         return Mono.empty();
       }
       return Mono.just(HttpResponse.ok(webshops));
     }).switchIfEmpty(Mono.just(HttpResponse.notFound()))
     .onErrorReturn(HttpResponse.serverError());
}
```

Aqui, como podemos ver o método é apenas um conjunto de ações no retorno, não existe um momento de espera e como não existe mudança de *schedulers*, o que vem por defeito é o `Schedulers.parallel()`. Ou seja, esta função funciona em paralelo, sem qualquer espera de dados de outros fluxos, o que vier, vem. Porém:

O que esta faz é a execução da função de pesquisa, aciona a execução de um Fluxo `Flux` de múltiplos dados, com as condições e ordem no repositório, ao receber junta os dados sequencialmente para não perder a ordem: existe uma espera para verificação da ordem (dentro da função usada), e dependendo dos dados obtidos, a resposta ao cliente dada é diferente. Uma Lista vazia é trocada por um fluxo `Mono` vazio (sem Lista), para identificar e trocar como uma resposta `404 Not found`.

Mais exemplares encontram-se no anexo três.

5.3.3 Modelos

Modelos *Webshop(...)Model*

Restantes modelos

5.3.4 Repositórios

Estas classes lidam com as conceções e transações com a base de dados, as quais abstraem emfluxos do Project Reactor (Mono e Flux). Estas classes precisam de lidar de forma segura e com o menor conflito possivel com a base de dados, logo qualquer repositório que seja feito para as ações numa (e para só uma e exclusiva) tabela precisa de ser uma classe saída dos clássicos *design patters*: um *Singleton* ou um *Monostate*.

Embora pessoalmente prefira usar o *Monostate* pois este não quebra os conceitos *Open/Closed Principle* e o *Dependency Inversion Principle* do SOLID, é esperado e está preparado para ser usado *Singletons* não só na framework, como também no ambiente de trabalho onde estive inserido. Como também que ao usar um *Singleton* facilita para o propósito exclusivo desta classe.

Este requerimento é facilmente subscrito com uma pequena anotação do Jackarta (incluída dentro da framework do Micronaut como dependência). A anotação *@Singleton* automaticamente, abstrai o processo de fechar (ao "público") o construtor e a criação do método *getter* da instância.

Foram precisas usar duas classes, ou seja, dois repositórios. Isto porque a base de dados apenas tem duas tabelas, inicialmente ponderado três tabelas, mas para facilitar o desenvolvimento decidi colocar as *settings* das Webshops na tabela principal. Podemos ver no seguinte gráfico a estrutura da base de dados:

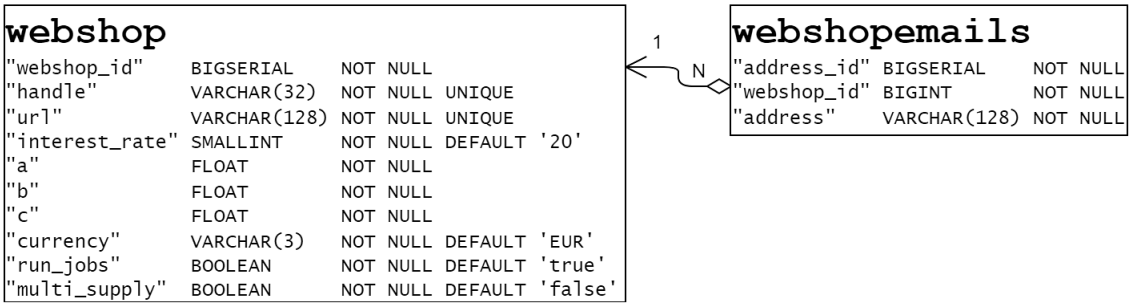

Figura 5.2: Gráfico da estrutura da BD, criado no GitMind

Seguidamente vamos a uma introdução de cada uma das classes:

- `WebshopRepository`
- `WebshopemailsRepository`

WebshopRepository

WebshopemailsRepository

Anexos

Anexo I

Implementação do `parseParamsWebshop`

```
@Override
public Condition parseParamsWebshop(String... params) {

if (params == null || params.length == 0) {
return null;
}

List<Condition> filterList = new ArrayList<>();

for (String param : params) {

String operation = "";
final Matcher m = Pattern.compile("[:%<>]").matcher(param);
if (m.find())
switch (m.group().charAt(0)) {
case ':' -> operation = ":";
case '%' -> operation = "%";
case '<' -> operation = "<";
case '>' -> operation = ">";
}
```

```
if (operation.isEmpty()) return null;

String[] split = param.split(operation);
switch (split[0]) {
case "handle" -> {
if (operation.equals(":")) {
filterList.add(Tables.WEBSHOP.HANDLE.equalIgnoreCase(split[1]));
} else if (operation.equals("%")) {
filterList.add(Tables.WEBSHOP.HANDLE.likeIgnoreCase(split[1]));
}
}
case "interestRate" -> {
if (operation.equals(">")) {
filterList.add(Tables.WEBSHOP.INTEREST_RATE.greaterThan(Short.parseShort(split[1])));
} else if (operation.equals("<")) {
filterList.add(Tables.WEBSHOP.INTEREST_RATE.lessThan(Short.parseShort(split[1])));
}
}
// To add more later
}
}

Condition condition = filterList.get(0);
filterList.remove(0);
for (Condition c : filterList) {
condition = condition.and(c);
}

return condition;
}
```

Anexo II

Implementação do `sortParserWebshop`

```
@Override
public SortField<?> sortParserWebshop(String sort, String order) {

if (sort == null || sort.isEmpty()) {
sort = "handle";
}

if (order == null || order.isEmpty()) {
order = "asc";
}

sort = sort.toLowerCase();
order = order.toLowerCase();

switch (sort) {
case "handle" -> {
if (order.equals("asc")) {
return Tables.WEBSHOP.HANDLE.asc();
} else if (order.equals("desc")) {
return Tables.WEBSHOP.HANDLE.desc();
}
return Tables.WEBSHOP.HANDLE.asc();
}
```

```
case "url" -> {
if (order.equals("asc")) {
return Tables.WEBSHOP.URL.asc();
} else if (order.equals("desc")) {
return Tables.WEBSHOP.URL.desc();
}
return Tables.WEBSHOP.URL.asc();
}
case "interestrate" -> {
if (order.equals("asc")) {
return Tables.WEBSHOP.INTEREST_RATE.asc();
} else if (order.equals("desc")) {
return Tables.WEBSHOP.INTEREST_RATE.desc();
}
return Tables.WEBSHOP.INTEREST_RATE.asc();
}
// To add more later
}

return Tables.WEBSHOP.HANDLE.asc();
}
```

Anexo III

Implementação do `RepositoryService`

```
/**
 * Repository service (middleware to remove business logic from controller).
 */
public class RepositoryService {


/**
 * The Webshop repository.
 */
@Inject
public WebshopRepository webshopRepository;
/**
 * The Webshopemails repository.
 */
@Inject
public WebshopemailsRepository webshopemailsRepository;

/**
 * Finds webshops via a conditionally defined
   * query sorted by a defined sort field.
 *
 * @param condition the condition
 * @param sortField the sort field
 * @return the webshops
 */
```

```
public Mono<MutableHttpResponse<List<WebshopModel>>>
    getWebshops(Condition condition, SortField<?> sortField) {

return webshopRepository.findVarious(condition, sortField)
      .flatMapSequential(this::getWebshopPriv)
.collectList().flatMap(webshops -> {
if (webshops.isEmpty()) {
return Mono.empty();
}
return Mono.just(HttpResponse.ok(webshops));
}).switchIfEmpty(Mono.just(HttpResponse.notFound()))
.onErrorReturn(HttpResponse.serverError());


}

/**
 * Helper method for getWebshops(), works exaclty
   * like the getWebshop() method, but without the
 * HttpResponse wrapping.
 *
 * @param handle the handle
 * @return the webshop
 */
private Mono<WebshopModel> getWebshopPriv(String handle) {

return webshopRepository.find(handle)
      .flatMap(webshop -> Mono.just(new WebshopModel(webshop)))
.flatMap(webshopModel -> webshopemailsRepository.findEmails(handle)
.flatMap(webshopemails -> {
webshopModel.setEmails(webshopemails);
return Mono.just(webshopModel);
}).flatMap(Mono::just));
}

/**
 * Gets webshop by handle.
 *
 * @param handle the handle
```

```
 * @return the webshop
 */
public Mono<MutableHttpResponse<WebshopModel>> getWebshop(String handle) {

return webshopRepository
      .find(handle).flatMap(webshop -> Mono.just(new WebshopModel(webshop)))
.flatMap(webshopModel -> webshopemailsRepository.findEmails(handle)
.flatMap(webshopemails -> {
webshopModel.setEmails(webshopemails);
return Mono.just(webshopModel);
}).flatMap(webshopModelWithEmails ->
Mono.just(HttpResponse.ok(webshopModelWithEmails))))
.switchIfEmpty(Mono.just(HttpResponse.notFound()))
.onErrorReturn(HttpResponse.serverError());
}

/**
 * Gets webshop settings by handle.
 *
 * @param handle the handle
 * @return the webshop settings
 */
public Mono<MutableHttpResponse<WebshopSettingsModel>>
    getWebshopSettings(String handle) {

return webshopRepository.find(handle)
.flatMap(webshop ->
Mono.just(new WebshopSettingsModel(webshop)))
.flatMap(webshopSettingsModel ->
Mono.just(HttpResponse.ok(webshopSettingsModel)))
.switchIfEmpty(Mono.just(HttpResponse.notFound()))
.onErrorReturn(HttpResponse.serverError());

}

/**
 * Creates a single webshop via a webshop full model (webshop model + settings).
 *
 * @param webshopModel the webshop model
```

```
 * @return the mono
 */
public Mono<MutableHttpResponse<String>>
    createWebshop(WebshopFullModel webshopModel) {

if (!webshopModel.isValid()) {
return Mono.just(HttpResponse.badRequest());
}

return webshopRepository.create(
webshopModel.getHandle(), webshopModel.getUrl(),
webshopModel.getServiceLevelA(), webshopModel.getServiceLevelB(),
            webshopModel.getServiceLevelC(), webshopModel.getInterestRate(),
            webshopModel.getCurrency(), webshopModel.getRunJobs(),
            webshopModel.getMultiSupplier()
).flatMap(webshopResponse -> {
if (webshopResponse) {
return webshopemailsRepository
              .createVarious(webshopModel.getHandle(), webshopModel.getEmails())
.flatMap(emailsResponse -> {
if (emailsResponse) {
return Mono.just(HttpResponse.ok("Webshop created."));
}
return Mono.empty();
});
}
return Mono.empty();
}).switchIfEmpty(Mono.just(HttpResponse.notFound()))
.onErrorReturn(HttpResponse.serverError());
}

/**
 * Creates various webshops via a list of
   * webshop full models (webshop model + settings).
 *
 * @param webshopModels the webshop models
 * @return the mono
 */
public Mono<MutableHttpResponse<String>>
```

```
    createWebshops(List<WebshopFullModel> webshopModels) {

for (WebshopFullModel webshopFullModel : webshopModels) {
if (!webshopFullModel.isValid()) {
return Mono.just(HttpResponse.badRequest());
}
}

return Mono.just(webshopModels)
      .publishOn(Schedulers.boundedElastic()).flatMap(webshops -> {
if (webshops.isEmpty()) {
return Mono.empty();
}
for (WebshopFullModel webshopModel : webshops) {
webshopRepository.create(
webshopModel.getHandle(), webshopModel.getUrl(),
webshopModel.getServiceLevelA(), webshopModel.getServiceLevelB(),
                    webshopModel.getServiceLevelC(), webshopModel.getInterestRate(),
                    webshopModel.getCurrency(),webshopModel.getRunJobs(),
                    webshopModel.getMultiSupplier()
).subscribeOn(Schedulers.boundedElastic())
.then(webshopemailsRepository
                .createVarious(webshopModel.getHandle(), webshopModel.getEmails()))
.subscribe();
}
return Mono.just(HttpResponse.ok("Webshops created."));
}).switchIfEmpty(Mono.just(HttpResponse.badRequest()))
.onErrorReturn(HttpResponse.serverError());

}

/**
 * Deletes a single webshop by handle.
 *
 * @param handle the handle
 * @return the mono
 */
public Mono<MutableHttpResponse<Object>> deleteWebshop(String handle) {
```

```
return webshopRepository.deleteWebshop(handle).flatMap(response -> {
if (response) {
return Mono.just(HttpResponse.noContent());
}
return Mono.empty();
}).switchIfEmpty(Mono.just(HttpResponse.badRequest()))
.onErrorReturn(HttpResponse.serverError());
}


/**
 * Fully updates a single webshop by getting the webshop full model.
 *
 * @param handle       the handle
 * @param webshopModel the webshop model
 * @return the mono
 */
public Mono<MutableHttpResponse<String>>
    updateWebshop(String handle, WebshopFullModel webshopModel) {

return webshopRepository.updateWebshop(handle, webshopModel.getHandle(),
webshopModel.getUrl(), webshopModel.getServiceLevelA(),
webshopModel.getServiceLevelB(), webshopModel.getServiceLevelC(),
webshopModel.getInterestRate(), webshopModel.getCurrency(),
webshopModel.getRunJobs(), webshopModel.getMultiSupplier()
).flatMap(response -> {
if (response) {
return webshopemailsRepository.deleteAll(webshopModel.getHandle())
.flatMap(deleteResponse ->
webshopemailsRepository.createVarious(webshopModel.getHandle(),
                      webshopModel.getEmails())
.flatMap(emailsResponse -> {
if (emailsResponse) {
return Mono.just(HttpResponse.ok("Webshop updated."));
}
return Mono.just(HttpResponse
                            .ok("Webshop updated without emails."));
}));
}
```

```
return Mono.empty();
}).switchIfEmpty(Mono.just(HttpResponse.notFound()))
.onErrorReturn(HttpResponse.serverError());
}

/**
 * Updates a single webshop's handle.
 *
 * @param handle    the handle
 * @param newHandle the new handle
 * @return the mono
 */
public Mono<MutableHttpResponse<String>>
    updateWebshopHandle(String handle, String newHandle) {

return webshopRepository.updateWebshopHandle(handle, newHandle)
.flatMap(response -> {
if (response) {
return Mono.just(HttpResponse.ok("Webshop updated."));
}
return Mono.empty();
}).switchIfEmpty(Mono.just(HttpResponse.notFound()))
.onErrorReturn(HttpResponse.serverError());
}

/**
 * Updates a single webshop's url.
 *
 * @param handle the handle
 * @param url    the url
 * @return the mono
 */
public Mono<MutableHttpResponse<String>>
    updateWebshopUrl(String handle, String url) {

return webshopRepository.updateWebshopUrl(handle, url)
.flatMap(response -> {
if (response) {
return Mono.just(HttpResponse.ok("Webshop updated."));
```

```
}
return Mono.empty();
}).switchIfEmpty(Mono.just(HttpResponse.notFound()))
.onErrorReturn(HttpResponse.serverError());
}


/**
 * Updates a single webshop's interest rate.
 *
 * @param handle       the handle
 * @param interestRate the interest rate
 * @return the mono
 */
public Mono<MutableHttpResponse<String>>
    updateWebshopInterestRate(String handle, Short interestRate) {

return webshopRepository.updateWebshopInterestRate(handle, interestRate)
.flatMap(response -> {
if (response) {
return Mono.just(HttpResponse.ok("Webshop updated."));
}
return Mono.empty();
}).switchIfEmpty(Mono.just(HttpResponse.notFound()))
.onErrorReturn(HttpResponse.serverError());
}


/**
 * Updates a single webshop's settings.
 *
 * @param handle        the handle
 * @param settingsModel the webshop settings model
 * @return the mono
 */
public Mono<MutableHttpResponse<String>>
    updateWebshopSettings(String handle, SettingsModel settingsModel) {

return webshopRepository.updateWebshopSettings(handle,
```

```
settingsModel.getCurrency(), settingsModel.getRunJobs(),
settingsModel.getMultiSupplier())
.flatMap(response -> {
if (response) {
return Mono.just(HttpResponse.ok("Webshop updated."));
}
return Mono.empty();
}).switchIfEmpty(Mono.just(HttpResponse.notFound()))
.onErrorReturn(HttpResponse.serverError());
}


/**
 * Updates a single webshop's service levels.
 *
 * @param handle             the handle
 * @param serviceLevelsModel the webshop service levels model
 * @return the mono
 */
public Mono<MutableHttpResponse<String>> updateWebshopServiceLevels
(String handle, ServiceLevelsModel serviceLevelsModel) {

return webshopRepository.updateWebshopServiceLevels(
handle, serviceLevelsModel.getServiceLevelA(),
serviceLevelsModel.getServiceLevelB(),
            serviceLevelsModel.getServiceLevelC())
.flatMap(response -> {
if (response) {
return Mono.just(HttpResponse.ok("Webshop updated."));
}
return Mono.empty();
}).switchIfEmpty(Mono.just(HttpResponse.notFound()))
.onErrorReturn(HttpResponse.serverError());
}


/**
 * Updates a single webshop's emails.
 *
```

```
 * @param handle      the handle
 * @param emailsModel the emails model
 * @return the mono
 */
public Mono<MutableHttpResponse<String>>
    updateWebshopEmails(String handle, EmailListModel emailsModel) {

return webshopemailsRepository.updateWebshopEmails(handle,
emailsModel.getEmails())
.flatMap(response -> {
if (response) {
return Mono.just(HttpResponse.ok("Webshop emails updated."));
}
return Mono.empty();
}).switchIfEmpty(Mono.just(HttpResponse.notFound()))
.onErrorReturn(HttpResponse.serverError());
}
}
```