Solução *Headless REST API* para *e-commerce*

Mestrado em Engenharia Informática – Computação Móvel

Micael José de Sousa Farinha

Leiria, Setembro de 2019

Solução *Headless REST API* para *e-commerce*

Mestrado em Engenharia Informática – Computação Móvel

Micael José de Sousa Farinha

Projeto realizada/o sob a orientação do Professor Marco António de Oliveira Monteiro

Leiria, Setembro de 2019

Originalidade e Direitos de Autor

O presente relatório de projeto é original, elaborado unicamente para este fim, tendo sido devidamente citados todos os autores cujos estudos e publicações contribuíram para o elaborar.

Reproduções parciais deste documento serão autorizadas na condição de que seja mencionado o Autor e feita referência ao ciclo de estudos no âmbito do qual o mesmo foi realizado, a saber, Curso de Mestrado em Engenharia Informática – Computação Móvel, no ano letivo 2018/2019, da Escola Superior de Tecnologia e Gestão do Instituto Politécnico de Leiria, Portugal, e, bem assim, à data das provas públicas que visaram a avaliação destes trabalhos.

Agradecimentos

De modo a conseguir concluir este projeto, existiram várias pessoas que contribuíram com várias peças que acabaram por encaixar neste “puzzle”. Sublinho esta ideia de puzzle, pois este nunca estaria completo caso existisse a falta de qualquer uma destas peças.

Ao orientador deste projeto, Professor Doutor Marco António de Oliveira Monteiro pela ajuda e disponibilidade mostrada durante o desenvolvimento deste projeto.

À minha família e amigos que sempre acreditaram em mim e me deram um grande apoio quando necessário.

À Beatriz pela paciência, companheirismo e incentivo em momentos importantes.

À empresa Blue-Infinity (agora conhecida como Isobar Switzerland) pelo desafio que me foi proposto, que me permitiu desenvolver conhecimentos vantajosos que serão uma mais valia na minha carreira profissional, assim como pela disponibilidade que todos os membros tiveram para me ajudar, sempre que requisitados.

A todos vocês, dedico este projeto.

Resumo

A empresa Blue-Infinity (agora conhecida como *Isobar Switzerland*), para quem foi desenvolvido este projeto, decidiu que existia a necessidade de se reestruturar e reformular um dos seus projetos mais antigos associados a uma plataforma web de comércio digital (uma loja online). Esta plataforma que assenta sobre uma *framework* de *PHP* (acrónimo de "*PHP: Hypertext Preprocessor"*) conta com vários anos de desenvolvimento. Devido à dimensão e complexidade do seu ecossistema, tornou-se cada vez mais difícil a adaptação às necessidades dos seus clientes.

Para efetuar a reformulação, o ecossistema foi dividido em duas partes. A parte associada à vista, que é apresentada ao cliente e a parte lógica, onde é efetuado todo o tratamento da informação. Este projeto irá focar-se apenas neste segundo aspeto, que será implementado através de uma *Headless REST* (*Representional State Transfer*) *API* (*Application Programming Interface*) com o foco para comércio digital. O objetivo final será obter uma API genérica com aplicabilidade ao comércio digital, completamente independente, que possa ser consumida pelo projeto responsável pela vista que é mostrada ao cliente.

Este relatório descreve todo este processo de desenvolvimento, assim como as variadas decisões que foram tomadas para a implementação do mesmo, que será utilizada com o propósito de substituir a solução atual e com possibilidade de servir como base a futuros projetos.

Palavras-chave: *Headless REST API, PHP, Laravel, E-commerce, CMS*

Abstract

Blue-Infinity (also known now as Isobar Switzerland), the company in charge and responsible for this project, decided that they needed to restructure and redesign one of their oldest solutions associated with a digital commerce platform (online store). This platform is based on a PHP framework (an acronym of "PHP: Hypertext Preprocessor") and has several years of development. Due to the dimension and complexity of its ecosystem, it has become harder to adapt the platform to the client needs.

In order to restructure the platform, the ecosystem was divided into two different parts: the visual aspect that is presented to the customer and the logic part, which manages and processes all the information. This project will merely focus on the second part which will be implemented as a Headless REST (Representional State Transfer) API (*Application Programming Interface*) focused on digital commerce. *The final objective will be to develop a generic and fully independent API which will be consumed by the project responsible for presenting the information to the customer.*

This report will describe the whole development process as well as explain the multiple decisions that were taken through this journey with the purpose of replacing the current solution and also possibly to be used for future developments of platforms with the same objective.

Keywords: *Headless REST API, PHP, Laravel, E-commerce, CMS*

Índice

Lista de Figuras

Lista de tabelas

Lista de siglas e acrónimos

API	*Application Programming Interface*
B2B	*Business to Business*
B2C	*Business to Consumer*
C2B	*Consumer to Business*
C2C	*Consumer to Consumer*
CMS	*Content Management System*
CRUD	*Create Read Update Delete*
ESTG	Escola Superior de Tecnologia e Gestão
GDPR	*General Data Protection Regulation*
HATEOAS	*Hypermedia As The Engine Of Application State*
HTML	*HyperText Markup Language*
HTTP	*HyperText Transfer Protocol*
IDE	*Integrated Development Environment*
IP	*Internet Protocol*
IPL	Instituto Politécnico de Leiria
IVA	Imposto sobre o Valor Acrescentado
JSON	*Javascript Object Notation*
MVC	*Model View Controller*
ORM	*Object-Relational Mapping*
PDO	*PHP Data Objects*
PHP	*PHP: Hypertext Preprocessor*
REST	*Representional State Transfer*
SEO	*Search Engine Optimization*
SLA	*Service Level Agreement*
SQL	*Structured Query Language*
UI	*User Interface*
URI	*Uniform Resource Identifier*
URL	*Uniform Resource Locator*

1. Introdução

Atualmente, existe uma simplificação do processo de compras *online* que nos permite, através de alguns simples cliques, ver, pagar e escolher o local de entrega do produto pretendido sem sairmos da nossa própria casa. Todo o processo associado a este tipo de compras tem sido estudado ao longo dos anos de maneira a fornecer uma *user experience* melhorada, que poderá ter impacto positivo nas vendas de qualquer loja que utilize uma plataforma *online* para este propósito [1]. Estes tipos de processos têm surgido com ajuda de aplicações, componentes e plataformas criadas especificamente para este propósito.

No âmbito deste trabalho foi desenvolvido uma *e-commerce Headless REST API*, com o propósito de gerir lógica associada a uma loja *online*, de uma forma completamente desacoplada de qualquer interface visual. Todos os requisitos associados ao trabalho foram definidos previamente pela entidade responsável pelo projeto, assim como as prioridades associadas a funcionalidades consideradas o *core* como por exemplo o "carrinho de compras".

Este documento descreve os vários processos associados ao desenvolvimento do projeto e as várias decisões que foram tomadas ao longo deste, como o desenho e arquitetura da solução final, o modelo de dados e todas as funcionalidades adicionadas.

1.1. Estrutura do documento

O presente documento apresenta a seguinte estrutura para os próximos capítulos:

- O capítulo 2 (Enquadramento) pretende efetuar uma apresentação das entidades responsáveis pela supervisão deste projeto, assim como responder, de uma maneira simplificada, a questões que envolvem o "porquê?", "como?" e "qual o objetivo?" do seu desenvolvimento;
- O capítulo 3 (Conceitos) disponibiliza toda a informação técnica necessária, de uma maneira aprofundada. Esta inclui uma descrição exaustiva dos vários tópicos associados às características que fazem parte do núcleo deste trabalho: conceito *Headless*; arquitetura *REST*; estrutura de um *CMS* (*Content Management System*) e *API.*;

- O capítulo 4 (Estado da arte) descreve as várias soluções similares disponíveis no mercado, onde foi efetuada uma pesquisa por diferentes tipos de *API*s (associadas ao tema em questão) através da verificação das várias funcionalidades de cada uma destas. Após a descrição de todas as anteriores, será efetuada uma comparação associada às suas funcionalidades existentes e a sua forma de utilização;
- O capítulo 5 (Metodologia) descreve as metodologias de trabalho que foram seguidas durante a etapa de desenvolvimento e preparação;
- O capítulo 5.2 (Arquitetura e tecnologias) descreve a arquitetura geral do projeto a alto e baixo nível assim como as várias tecnologias que foram utilizadas durante a implementação;
- O capítulo 7 (Implementação) descreve os vários tipos de funcionalidades adicionadas ao projeto, de uma perspetiva técnica. Este capitulo engloba secções que explicam o modelo de dados na sua totalidade (incluindo a descrição de cada um destes), as respostas ao "como?" e "porquê" das funcionalidades desenvolvidas;
- O capítulo 8 (Testes) descreve os testes que foram executados no projeto. Estes incluem a descrição dos testes unitários que pretendem garantir o bom funcionamento da aplicação (incluindo o detalhe de cada um destes), assim como os testes de carga que foram utilizados para apoiar uma das decisões tomadas, que será posteriormente descrita em detalhe;
- Por fim, o capítulo 9 (Conclusão) descreve as principais conclusões obtidas com a realização do projeto, as dificuldades encontradas durante os vários processos de pesquisa, desenvolvimento e finalização, assim como o possível trabalho futuro a ser realizado (melhorias e novas funcionalidades).

2. Enquadramento

Nesta secção irá ser efetuada uma contextualização do projeto desenvolvido em relação ao objetivo final pretendido, às restrições definidas pela empresa e a razão das várias decisões tomadas até à finalização da solução final.

2.1. Âmbito do projeto

Este projeto foi desenvolvido no âmbito do Mestrado em Engenharia Informática com especialidade em Computação Móvel, da Escola Superior de Tecnologia e Gestão (ESTG) do Instituto Politécnico de Leiria (IPL) em parceria com a entidade empresarial *Blue-Infinity* (atualmente conhecida como *ISOBAR SWITZERLAND*, no entanto, será referenciada neste documento como *Blue-Infinity* uma vez que esta mudança de nome ocorreu posteriormente ao inicio do desenvolvimento deste documento), empresa esta onde exerço atividade profissional.

2.2. Base do projeto

A empresa para qual este trabalho foi desenvolvido é proprietária de uma plataforma de comércio digital (considerada o projeto base da aplicação desenvolvida neste trabalho), a ser utilizada por um dos seus clientes e onde me encontro atualmente a trabalhar. Esta trata-se de uma aplicação *web* assente sobre uma versão feita "à medida" de um *CMS* (*Content Management System*) denominado "CS-cart". Trata-se de uma versão "à medida" pois os programadores do próprio *CMS, Simtech conglomerate* [2], efetuaram modificações em determinadas funcionalidades *legacy* do seu projeto de modo a adaptar-se ás necessidades propostas pela Blue-Infinity.

O que era um projeto inicialmente pequeno e com alguma afluência de clientes, rapidamente cresceu devido à popularidade e necessidades do mercado. Neste momento, o "CS-cart" é responsável por tratar das duas componentes principais da loja, *backend* e *frontend*. O contínuo desenvolvimento da aplicação durante vários anos, tornou-a mais lenta e complexa, sendo cada vez mais difícil adicionar novas funcionalidades à mesma, o que levou à necessidade do desenvolvimento de uma solução que superasse este tipo de problemas. A base de todo este projeto foi definida através da experiência profissional obtida durante o desenvolvimento efetuado neste *CMS*.

2.3. Restrições do projeto

As condições necessárias ao desenvolvimento do projeto foram simples e com alguma liberdade em tomar decisões que mudassem o rumo do projeto, sem retirar o seu objetivo final. As condições impostas foram o desenvolvimento de uma aplicação *web* com as seguintes características:

- Desenvolvimento utilizando a *framework* Laravel;
- Disponibilização das suas funcionalidades através de uma *API*;
- Utilização da arquitetura *REST*;
- Aplicação do conceito *headless*;
- Com aplicabilidade ao comércio digital.

2.4. Objetivo do projeto

O objetivo principal do projeto trata-se do desenvolvimento de uma *Headless REST API* que servirá como protótipo ou base de qualquer aplicação associada a *e-commerce*. Depois da adição de funcionalidades especificas, esta poderá vir a substituir uma das soluções proprietárias desenvolvida e atualmente em utilização por parte da empresa. Estas funcionalidades deverão ser implementadas de uma maneira genérica de modo a servir também para possíveis futuros projetos que possam utilizar esta solução como base.

Tal como referido na secção 2.2, a plataforma base tornou-se lenta ao longo do tempo. De modo a conseguir obter um maior desempenho por parte da aplicação, facilidade na manutenção e implementação de novas funcionalidades, a empresa pretende separar o *frontend* do *backend*, de modo a que cada uma destas componentes trate de uma secção específica do próprio projeto.

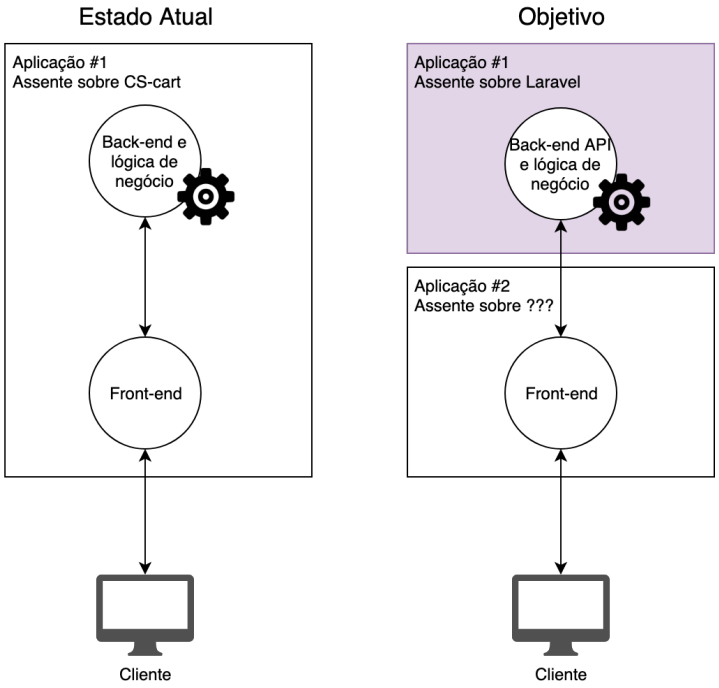


Ilustração 1 - Estado atual vs Objetivo de implementação

Tal como se consegue verificar através da Ilustração 1, o nosso objetivo de implementação pretende desacoplar o produto atual em dois sub-produtos, de modo a conseguirmos ter uma aplicação dedicada ao tratamento de informação e outra para apresentação de conteúdo, no entanto, este projeto pretende apenas focar na parte de *backend*, apresentada a roxo na Ilustração 1.

É importante referir que as várias decisões que foram tomadas ao longo do desenvolvimento do projeto, foram-no com base na experiência profissional obtida através do trabalho desenvolvido na plataforma de comércio digital cuja empresa é proprietária. O facto de me encontrar a trabalhar nesta plataforma aquando a realização deste trabalho, facilitou a capacidade de tomar decisões aquando da implementação das várias funcionalidades "genéricas" de uma plataforma de comércio digital.

2.5. Aplicabilidade do projeto

Este projeto foi desenvolvido com o propósito de servir como lógica base de qualquer projeto associado ao comércio digital e possivelmente substituir a solução atualmente existente apresentada através da Ilustração 1 (após adição de componentes necessários à lógica de negócio específica), no entanto, o desenvolvimento deste projeto foi efetuado de maneira a que as funcionalidades implementadas sejam genéricas.

O projeto não está preparado para um ambiente de produção, pois este não tem determinados componentes necessários a uma loja *online* (exemplo: métodos de pagamento e faturação) pois estas serão posteriormente adicionadas através de integrações *third party* (atualmente existentes no projeto base representado na Ilustração 1).

3. Conceitos

Esta secção tem como objetivo definir e explicar os quatro conceitos principais associados ao projeto – *E-commerce*, *CMS*, *REST* e *Headless*.

3.1. E-commerce

Podemos designar de "*E-commerce*", do inglês "*Electronic Commerce*" (comércio digital ou comércio eletrónico) o ato de compra ou venda de qualquer tipo de produto através do uso da internet [3]. A primeira transação online ocorreu a 11 de Agosto de 1994, quando foi usado um cartão de crédito e lógica de encriptação (base para todos os pagamentos de hoje em dia) pela primeira vez para efetuar uma compra *online*.

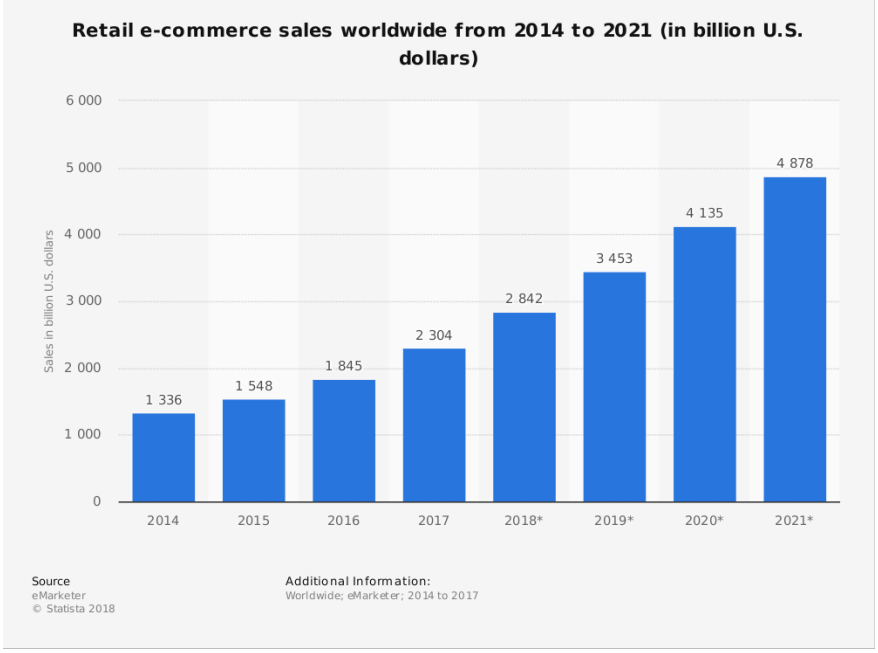


Ilustração 2 - Crescimento de vendas relacionadas com comércio digital nos Estados Unidos da América em biliões de euros de 2014 a 2021 (este como previsão) [4]

Como podemos verificar através da Ilustração 2, tem-se assistido a um rápido crescimento do *e-commerce* [5] nos últimos anos. Existem vários tipos de modelos de negócio ou classificações associadas ao *e-commerce* que têm objetivos um pouco diferentes, apesar de serem baseados no mesmo conceito. Segue-se uma enumeração de quatro destes modelos, considerados como os principais tipos devido à sua taxa de utilização [5]:

- ***B2C (Business to Consumer)*** – Modelo de negócio mais comum, onde existem transações efetuadas entre clientes e empresas, em que a empresa vende produtos

ao cliente final. Qualquer loja *online* que tenha como objetivo a venda direta aos clientes de produtos ou serviços, é inserida neste modelo;

- ***B2B (Business to Business)*** – Modelo de negócio onde as transações são efetuadas de empresa para empresa. Qualquer tipo de negócio que tenha como clientes outra(s) empresa(s) é inserido neste modelo;
- ***C2B (Consumer to Business)*** – Modelo de negócio onde o cliente vende ou contribui de alguma forma financeira para uma empresa. Um exemplo seria o caso de uma empresa criar um *crowdfunding* de modo a conseguir apoiar financeiramente um projeto ou ideia;
- ***C2C (Consumer to Consumer)*** – Modelo de negócio onde o próprio cliente vende a outro cliente. O uso de sites como o Ebay para compra e venda é um bom exemplo relativo ao *C2C*.

Devido ao crescimento do comércio digital, começaram a surgir plataformas e ferramentas de modo a facilitar o trabalho do lado da empresa a nível de programação e preparação da sua lógica para um determinado modelo de negócio.

Neste momento conseguimos facilmente abrir a nossa própria loja online sem grandes problemas, já que existem enumeras plataformas que permitem configurar e começar o nosso próprio negócio de vendas *online*.

3.2. Designação e estrutura de um *CMS*

Um *CMS* é uma *framework* ou aplicação para gestão de conteúdo. Esta gestão poderá incluir a modificação, remoção ou adição de informação que será posteriormente apresentada [6].

Um dos objetivos principais de um *CMS* consiste em dar o controlo total a uma empresa ou entidade, sobre o conteúdo apresentado no seu *website*, que por sua vez traz autonomia à mesma. Com uma interface intuitiva, colaboradores da empresa podem atualizar o conteúdo, sem necessitar de uma equipa especializada. Em *websites* tradicionais existe a necessidade da intervenção de uma equipa contratada para a sua manutenção e atualização devido à inexistência da possibilidade de modificar a plataforma sem existir a inserção ou modificação de código.

Um *CMS* é geralmente uma ferramenta indicada para projetos de média e larga escala já que esta é muito poderosa a nível de configurações e processamento de dados. Existem vários tipos de *CMS*, que têm como objetivo áreas de negócio especificas. Estas plataformas já incluem um arsenal de componentes completamente funcionais de modo a que se consiga chegar a um determinado objetivo rapidamente sem necessitar de se perceber a parte interna de funcionamento. Por exemplo Frameworks associadas ao comércio digital, trazem muitas das vezes sistemas de inserção de produtos para venda, integração de carrinho de compras métodos de pagamento e *SEO* (*Search Engine Optimization*) através de simples configurações [7].

No decurso deste trabalho irão ser referenciadas várias vezes os componentes principais associados a um *CMS* – "*frontend*" e "*backend*", pelo que existe a necessidade de explicar o conceito e objetivo de cada um destes.

3.2.1. Frontend

Entenda-se como *frontend* o componente genérico associado à parte gráfica de uma ferramenta, plataforma ou *website*. Por outras palavras, é a parte que o utilizador final vê [8]. É também responsável por proporcionar a interação entre o utilizador e a aplicação e por transmitir as ações efetuadas por este durante a sua utilização a outro componente importante de um *CMS*, o *backend*.

3.2.2. Backend

Entenda-se como *backend* o componente que trata todo o tipo de informação e lógica associada a uma ferramenta, plataforma ou *website*.

Este tratamento de informação consiste na execução de procedimentos ou blocos de código com o objetivo de obter um determinado resultado ou *output*. Este componente utiliza informação, geralmente guardada numa base de dados, de modo a ser tratada e enviada para o *frontend* para ser disponibilizada ao utilizador final. Toda a lógica e validação de informação deverá ser incluída neste componente, pois o *frontend*, sendo disponibilizado ao cliente, poderá ser manipulado.

3.2.3. Uso de módulos

Como os *CMS* se tratam de plataformas bastante dinâmicas, existiu a necessidade de implementar algo que prevenisse que estes se tornassem demasiado grandes devido a

conteúdo ou lógica que o utilizador final não necessitasse. Para prevenir este tipo de situações, estas plataformas normalmente incluem um sistema de módulos, por vezes também chamados de "*add-ons*" ou "*plug-ins*". Este sistema de *add-ons* permite que os *CMS* adicionem funcionalidades adicionais ou removam funcionalidades desnecessárias [9].

Como exemplo, um blog assente sobre um *CMS* não necessita de um módulo que integre métodos de pagamento, por isso não precisa que este esteja instalado. Poderemos considerar estes módulos como uma extensão à própria plataforma [10], que permite estender a sua lógica e modo de funcionamento através da integração de novos *add-nos*, tal como se pode observar na Ilustração 3.

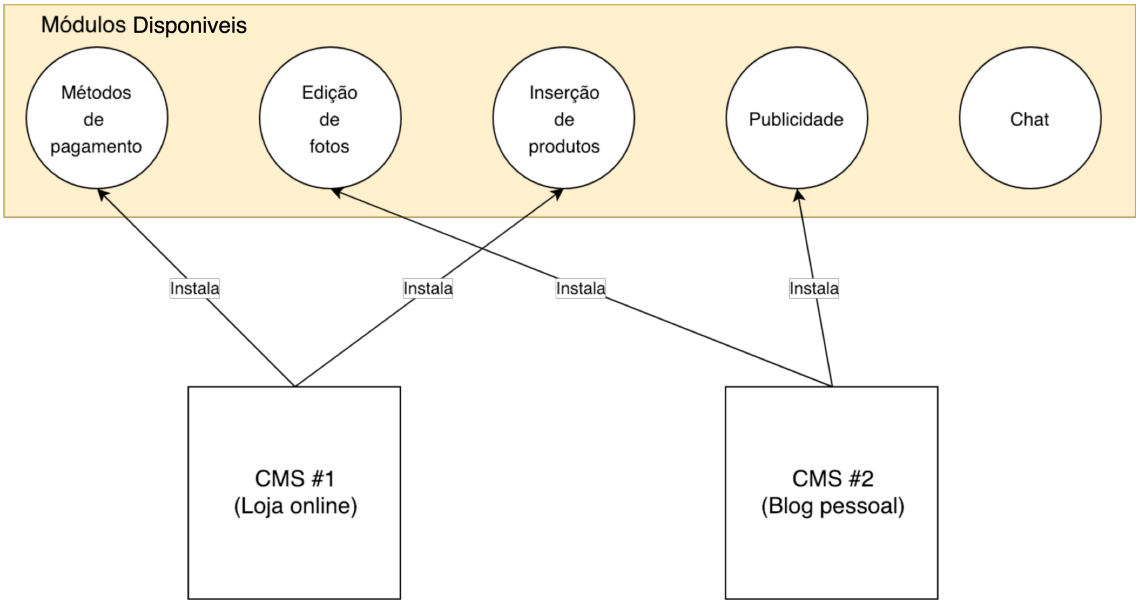


Ilustração 3 - Exemplo da diferente escolha de módulos entre dois *CMS* diferentes

Nas plataformas *CMS* mais utilizadas, é comum haver um *marketplace* que onde é possível encontrar *add-ons* disponibilizados pela própria equipa de programação do *CMS* ou até mesmo soluções desenvolvidas por outras equipas. Este tipo de *add-ons* disponibilizados são geralmente grátis, mas por vezes poderão ter algum tipo de custos, através de compra única ou até mesmo subscrição mensal.

3.2.4. Foco em *e-commerce*

Com a evolução do comércio digital, começaram a surgir *CMS* com funcionalidades e características focadas neste. Este tipo de foco está associado à integração e facilidade de executar tarefas com o intuito de venda ao cliente. Muitos destes *CMS* vêm com várias

integrações por omissão (que podem ser ou não instaladas), onde existe uma necessidade de configuração mínima por parte da entidade vendedora.

Através da experiência profissional obtida ao trabalhar com este tipo de plataformas, é possível identificar várias funcionalidades ou módulos associados aos *CMS* com foco ao *e-commerce* que incluem:

- **Visão geral de atividade (*dashboard*)** – Geralmente representado como uma página que disponibiliza informação sobre vários aspetos do negócio, incluindo por exemplo – número de vendas num determinado período de tempo, número de visitantes e de clientes registados;
- **Catálogo de produtos** – Página de *frontend* onde são disponibilizados os produtos aos clientes, de modo a que possam efetuar a sua compra;
- **Cálculo automático de taxas** – Lógica relativa ao calculo automático da taxa associada a produtos em relação a um determinado país. Por exemplo, em caso de configurarmos a nosso *website* para venda em Portugal, os produtos poderiam ser configurados para incluírem a taxa de IVA (Imposto sobre o Valor Acrescentado);
- **Cálculo automático de portes de envio** – Lógica associada ao calculo dos portes de envio. Esta geralmente é efetuada através do peso e dimensões relativos aos produtos da encomenda. Este calculo poderá, em certos casos, ser obtido através de uma entidade externa que disponibiliza preços associados aos vários tipos de envio;
- **Carrinho de compras** – Lógica que permite aos clientes finais adicionarem produtos ao seu carrinho de compras (com ou sem *login*). O objetivo deste módulo é fazer com que o cliente permaneça com os seus produtos no seu carrinho de compras caso estes não efetuem a(s) compra(s) no momento. Este deverá ser persistente no tempo e não apenas na sessão em que o utilizador se encontra a navegar na loja. O carrinho de compras é também responsável por agrupar os vários produtos numa compra única e guardar o método de pagamento, envio, fatura, entre outros tipos de informação importantes;
- **Integração com vários métodos de pagamento** – Através de configurações, conseguimos disponibilizar ao cliente vários métodos de pagamento diferentes. Os mais comuns incluem “VISA” ou “Paypal” por exemplo;

- **SEO (*Search Engine Optimization*)** – A "presença forte" de um *website* nos motores de pesquisa é uma peça fundamental para qualquer negócio associado ao comércio digital. Um estudo efetuado em 2010 revelou que o primeiro resultado de uma pesquisa no Google recebe 32.53% do tráfico em relação aos 17.6% da segunda posição, assim como os resultados na primeira página de uma pesquisa no Google recebem 91.5% do tráfego total em relação às outras páginas [11]. Tendo o objetivo de venda ao público (no caso do *B2C*), é fácil de concluir que existe um foco grande para que o *SEO* da plataforma seja efetivo.

Algumas das funcionalidades descritas previamente são consideradas como o *core* de aplicações associadas a *e-commerce* e representam também o *backbone* do próprio CMS [12].

3.3. Arquitetura *REST*

A definição de *REST*, do inglês "*Representational State Transfer*", representa um modelo de arquitetura associado a aplicações *web*, que disponibiliza determinadas regras de implementação a seguir [13]. O objetivo principal deste modelo é estabelecer padrões ou normas de modo a facilitar a comunicação com aplicações *web*.

A arquitetura tem algumas características relativas ao modelo estrutural que iremos explicar em algum detalhe – Separação cliente-servidor, sem estado, *cacheable*, sistema por camadas e interface uniformizada [14].

3.3.1. Separação cliente – servidor

O estilo arquitetural associado ao *REST* permite que a implementação da parte associada ao servidor possa ser desenvolvida de uma maneira completamente independente da parte associada ao cliente sem que haja qualquer conhecimento de como cada uma destas funciona "internamente". Isto permite que as componentes associadas ao lado do servidor não interfiram com o lado do cliente, assim como o oposto. As componentes conseguem ser mantidas de maneira separada e modular desde que ambas as partes entendam o formato associado às mensagens ou pedidos trocados.

3.3.2. Sem estado

Tal como descrito na secção 3.3.1, existe uma separação a nível de implementação entre o cliente e o servidor. O mesmo acontece a nível do estado de cada um destes

componentes, o que indica que o servidor não guarda nem tem "conhecimento" do estado em que se encontra o cliente, e vice-versa. Isto permite que os pedidos efetuados sejam sempre interpretados e compreendidos independentemente dos pedidos/respostas enviados previamente. Este tipo de "restrição" permite que aplicações *RESTful (*que seguem o estilo arquitetural *REST)* consigam alcançar uma maior *performance* e escalabilidade, onde os vários componentes podem sofrer alterações sem afetar o sistema como um todo [14].

3.3.3. Cacheable

Por vezes existem pedidos efetuados ao servidor que resultam sempre na mesma resposta, no entanto, o servidor tem de processar a lógica associada a esse pedido, sempre que este é efetuado. O objetivo da cache é prevenir que seja gerada a mesma resposta através da prevenção da execução de lógica "desnecessária" de informações que foram processadas previamente. A cache é geralmente guardada através de ficheiros com uma data de expiração associada. A cache é descartada quando a sua data de expiração é atingida.

Existem várias *frameworks* que fornecem este tipo de sistema *out-of-the-box*, no entanto, a melhor maneira para usufruir de um sistema de cache associado a uma API é através da implementação de um intermediário chamado "*gateway cache*" ou "*reverse proxy*". Este intermediário tem como objetivo efetuar o pedido à *API*, caso esta não esteja ainda em *cache*, de modo a guardá-la para uso futuro.

Apesar do uso da cache ser um sistema com muitas vantagens, existe também uma grande desvantagem associada – ao evitarmos execução de lógica para obter uma determinada resposta com o uso de informação em cache, arriscamos a que a informação devolvida pelo servidor não esteja atualizada até que a cache seja invalidada [15].

3.3.4. Sistema por camadas

A arquitetura *REST* permite que sejam implementadas diferentes partes da aplicação em diferentes servidores. Isto é geralmente transparente ao utilizador final, que se limita a utilizar a aplicação sem perceber a sua "engrenagem", ou seja, a maneira de funcionamento desta.

Através deste sistema por camadas, uma aplicação que segue esta arquitetura poderá por exemplo utilizar um servidor dedicado apenas à *API*, outro servidor para autenticar pedidos efetuados pelo cliente e outro servidor para guardar informação e/ou *cache*. Tal como dito anteriormente, isto acaba por ser transparente ao utilizador final, que acaba por

não ter a noção se os pedidos deste estão a chegar ao nó final ou a um dos intermediários [16].

3.3.5. Interface uniformizada

Uma das características que distingue o *REST* de outro tipo de arquiteturas é a importância que é dada à uniformização da *interface*. Este tipo de característica define princípios a seguir de modo a conseguirmos obter uma interface uniforme, nomeadamente [17]:

- Cada recurso é definido nos pedidos efetuados através da utilização de *URI*s (*Uniform Resource Identifier*);
- Os pedidos efetuados sobre um recurso disponibilizam informação suficiente que representa a ação que se permite efetuar, através dos múltiplos meta dados associados, como o verbo *HTTP* (*HyperText Transfer Protocol*) do pedido ou *Content-Type*;
- Utilização de *HATEOAS* (*Hypermedia As The Engine Of Application State*) que define que o estado da aplicação deverá vir bem explicito nos pedidos do cliente e respostas do servidor. Os clientes deverão representar o estado através do envio de informação no corpo do pedido, *headers*, parâmetros *query-string* e o recurso associado enquanto que os servidores deverão representar o estado através informação no corpo da resposta, códigos de resposta e *headers*.

3.3.5.1. Verbos HTTP

Quando se pretende efetuar uma determinada ação sobre uma entidade, o pedido associado a este deverá vir com um método *HTTP* (por vezes referido como "verbo") específico no pedido. A atribuição deste "verbo" ao pedido permite fazer com que o serviço *web* passe a ser definido por regras bastantes restritas, associados à caracteristica "Interface uniformizada" [18], descrita na secção 3.3.5. Este tipo de normas permite estabelecer um mapeamento entre o método *HTTP* e o objetivo que se pretende com o pedido efetuado, geralmente associados a operações *CRUD* (Create, Read, Update e Delete). Alguns dos metodos *HTTP* que serão abrangidos nesta secção são – "GET", "POST", "PUT", "PATCH" e "DELETE". Existem algumas propriedades associadas a estes que devem ser definidas previamente à sua descrição. Estas propriedades são:

- **Idempotência** – Um verbo *HTTP* é considerado idempotente caso este possa ser chamado múltiplas vezes acabando por produzir sempre o mesmo resultado do lado do servidor. Fazer um pedido tem o mesmo resultado que fazer múltiplos pedidos. Apesar deste tipo de operações produzir sempre o mesmo resultado do lado do servidor, a resposta do servidor pode não ser a mesma (pode-se usar a mudança de propriedades de uma determinada entidade como exemplo) [19];
- **Seguro** – Um método é considerado seguro caso este não exponha operações "inseguras", que envolvam a modificação de informação associada a recursos.

As propriedades de idempotência e pedido seguro são associadas ao modelo arquitetural REST quando seguido devidamente, no entanto, estas poderão ser quebradas através de uma implementação "fraca". De modo a permitir uma explicação detalhada de cada um dos verbos através de um exemplo, consideremos o seguinte cenário:

- Existe um serviço *web* criado através do modelo arquitetural *REST* disponibilizando os seus serviços através do *URL* (*Uniform Resource Locator*) "http://www.xpto.com";
- O serviço *web* disponibiliza a entidade "Utilizador" através do *URI* "utilizadores";
- O serviço *web* tem disponível dois registos associados à entidade "Utilizador" na base de dados com as seguintes propriedades:
    - Registo associado ao identificador "12" contém as seguintes propriedades:
        - nome - "Ana";
        - sexo - "Feminino".
    - Registo associado ao identificador "13" contém as seguintes propriedades:
        - nome - "João";
        - sexo - "Masculino".

GET

Método *HTTP* utilizado com o objetivo de se adquirir informação sobre um recurso através do fornecimento do identificador associado a um determinado registo (geralmente guardado num base de dados). Utilizando o exemplo do serviço *web* anterior, em caso de se

pretender adquirir informação sobre a entidade "utilizador" cujo identificador é "13", o *flow* associado ao pedido seria o seguinte:

Ilustração 4 - Flow de um pedido "GET" associado ao recurso "utilizadores" com identificador

Através da ilustração 4, conseguimos verificar que o servidor devolveu o código de resposta "200 OK", o que indica que o registo associado à entidade "Utilizador" com o identificador "13" foi encontrado e devolvido com sucesso. Caso o registo com este identificador não exista, o servidor deverá devolver o código de resposta "404 NOT FOUND".

Caso não seja fornecido o identificador de um determinado registo, este pedido passa a ter como objetivo, a "recolha de informação" de todos os registos associados ao recurso, geralmente acompanhados por alguma informação associada à paginação, filtros ou ordenação dos mesmos.

Ilustração 5 - Flow de um pedido "GET" associado ao recurso "utilizador" sem identificador

Através da Ilustração 5 podemos verificar que o servidor devolveu informação sobre os dois registos disponíveis na base de dados. Caso não existam registos associados a esta entidade, a resposta do servidor deverá ser um *array* vazio com o código de resposta "200 OK". Este método é considerado seguro e idempotente.

POST

Método *HTTP* utilizado com o objetivo de se criar um recurso usando as propriedades enviadas no pedido. Para este tipo de método não é necessário especificar o identificador deste, pois este ainda não existe.

Ilustração 6 - Flow de um pedido "POST" associado ao recurso "utilizadores"

Através da Ilustração 6, conseguimos verificar que foi requisitado a criação de um novo recurso com as propriedades "nome" com o valor "Adriana" e "sexo" com o valor

"Feminino", na qual o servidor devolveu o código de resposta "201 CREATED", associado à inserção bem-sucedida do recurso. Este método não é seguro nem idempotente.

PUT

Método *HTTP* utilizado com o objetivo de se efetuar modificações a um determinado recurso através do seu identificador. Este verbo é apenas ser utilizado caso se pretenda modificar um recurso na sua totalidade (exceto o seu identificador). Isto indica que todas as propriedades (exceto o identificador) deste deverão ser enviadas no pedido (mesmo que estas não sejam modificadas). Utilizando o exemplo anterior, se o objetivo for a atualização do utilizador com o identificador "13" de modo a que a propriedade "nome" seja modificado para "Marcelo", o *flow* será o seguinte:

Ilustração 7 - Flow de um pedido "PUT" associado ao recurso "utilizadores"

Através da Ilustração 7 podemos verificar que no corpo do pedido a informação sobre o atributo "sexo" também se encontra presente, apesar do objetivo ser apenas a mudança do atributo "nome". O utilizador cujo identificador é "13" terá agora o atributo nome modificado para "Marcelo" e o atributo "sexo" modificado para "Masculino" pois como podemos verifica, a resposta devolvida por parte do servidor foi "200 OK". Caso o registo com este identificador não exista, o servidor deverá devolver o código de resposta "404 NOT FOUND". Este método é considerado idempotente mas não seguro.

PATCH

Método *HTTP* utilizado com o objetivo de se efetuar modificações parciais a um determinado recurso através do seu identificador. Ao contrário do Verbo "PUT" descrito na secção anterior, este não necessita que todas as propriedades do recurso sejam enviadas no

pedido, apenas as propriedades enviadas sofrerão alterações. Utilizando o exemplo anterior, se o objetivo for a atualização do utilizador com o identificador "13" de modo a que o nome seja "Marcelo", o *flow* será o seguinte:

Ilustração 8 - Flow de um pedido "PATCH" associado ao recurso "utilizadores"

Através da Ilustração 8 podemos verificar que no corpo do pedido apenas se encontra a propriedade e o respetivo valor que se pretende alterar, neste caso a propriedade "nome". O utilizador cujo identificador é "13" terá agora o atributo nome modificado para "Marcelo", pois como se pode verificar, a resposta devolvida por parte do servidor foi "200 OK". Caso o registo com este identificador não exista, o servidor deverá devolver o código de resposta "404 NOT FOUND". Este método não é considerado idempotente nem seguro.

DELETE

Método *HTTP* utilizado com o objetivo de se eliminar um determinado recurso através do seu identificador. Através do exemplo anterior, caso se pretenda eliminar o recurso "Utilizador" com o identificador "13", o *flow* associado ao pedido seria o seguinte:

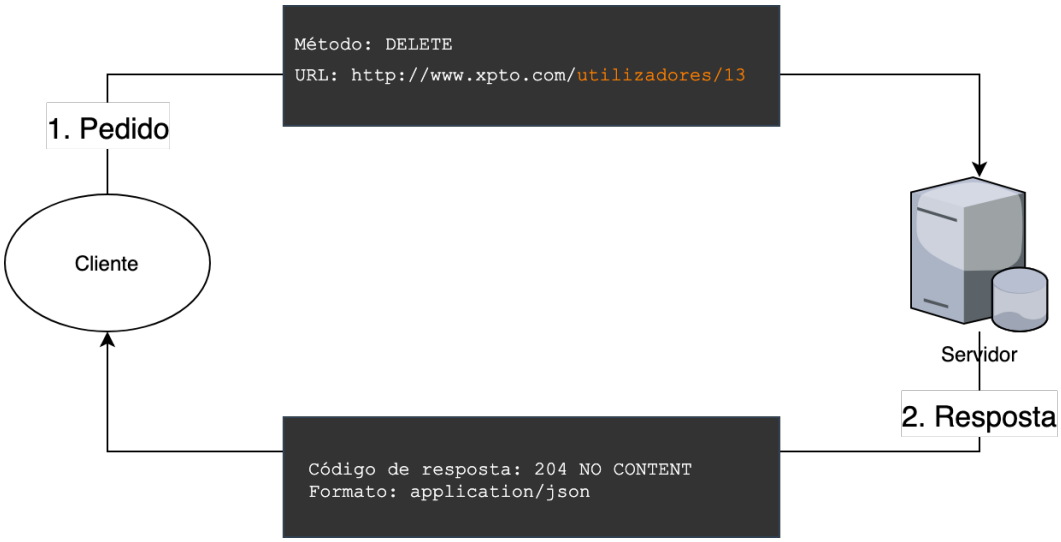


Ilustração 9 - Flow de um pedido "DELETE" associado ao recurso "utilizadores"

Através da Ilustração 9, é possível verificar a devolução do código de resposta "204 NO CONTENT" por parte do servidor que indica que este recurso com este identificador foi eliminado com sucesso. Caso o registo com este identificador não exista, o servidor deverá devolver o código de resposta "404 NOT FOUND".

Este método é considerado seguro. Existe alguma "controvérsia" em relação a este método pois existem referências que o consideram idempotente [20] e outras que não o consideram associado a esta caracteristica [21]. Esta situação ocorre, pois, após a remoção de um recurso, os pedidos posteriores associados a essa remoção passam a devolver sempre o código de resposta "404 NOT FOUND", quebrando o pressuposto da idempotencia.

3.3.5.2. Códigos de resposta

Os códigos de resposta associados a um pedido têm um simples objetivo - informar se um determinado pedido foi bem-sucedido, mal-sucedido, ou ocorreu algum tipo de situação mais específica [22].

Tal como demonstrado no capítulo anterior através de ilustrações, foram demonstrados alguns tipos de códigos de resposta que podem ser devolvidos por parte do servidor. Este tipo de códigos de resposta está divido por blocos:

- **Bloco 1XX** – Associado a pedidos de informação, exemplo: "100 Continue";
- **Bloco 2XX** – Associado a pedidos que foram bem-sucedidos, exemplo: "200 OK";
- **Bloco 3XX** – Associado a pedidos que acabaram por ser redirecionados, exemplo: "301 Moved Permanently";

- **Bloco 4XX** – Associado a pedidos onde ocorreu um erro por parte do cliente, exemplo: "404 Not Found";
- **Bloco 5XX** – Associado a pedidos onde foi despoletado um error por parte do servidor, exemplo: "502 Bad Gateway".

3.4.API

Uma *API*, acrónimo para *Application Programming Interface* é um conjunto de rotinas e padrões de programação que permitem a criação de aplicações com intuito de servir de intermediário que permite que duas aplicações comuniquem entre si [23]. A explicação associada a este termo é simples, existe um *endpoint* que fornece um serviço (através de tratamento ou disponibilização de informação), que é posteriormente consumido por outro(s) *endpoint*(*s*), como podemos verificar através da Ilustração 10.

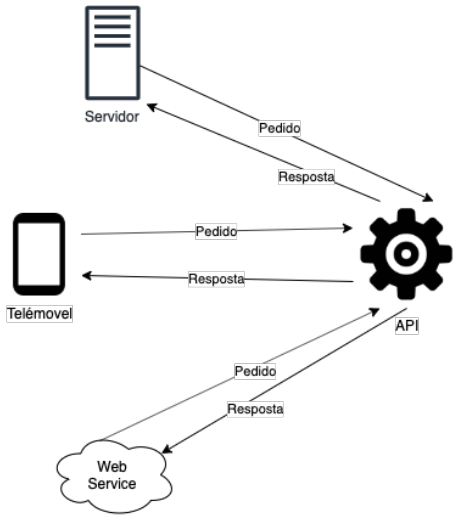


Ilustração 10 – Consumo de uma *API* por múltiplos *endpoints*

O principal objetivo de uma *API* é permitir abstrair a funcionalidade entre dois sistemas diferentes através do desacoplamento da infraestrutura que fornece o serviço da aplicação que a consome.

Em projetos de larga escala, é comum haver uma divisão de determinados componentes em diferentes *APIs*, tal como acontece neste projeto. A divisão destes componentes permite uma maior facilidade de desenvolvimento, pois esta separação permite que cada API seja responsável por tratar de um determinado tipo de informação ou funcionalidade. Podemos usar este projeto para explicar esta afirmação. Sendo que se

pretende separar o *backend* do *frontend*, a *API* criada para tratar da lógica do *backend*, será apenas responsável por isso.

A utilização de *APIs* num projeto pode acelerar o desenvolvimento, no entanto, pode também ser uma "fonte de problemas". Até agora apenas foi descrito a separação de componentes de um projeto, no entanto, é comum haver o consumo de *APIs* criadas e geridas por entidades externas (exemplo: GoogleMaps *API*), que acaba por fazer com que o projeto em si fique dependente (em muitos dos casos) do serviço consumido, geralmente estão associados a serviços disponibilizados na *cloud*. Qualquer serviço que seja "utilizado" através de uma camada de rede traz para a "equação" mais uma incógnita: a disponibilidade do serviço. Se uma aplicação for desenvolvida com dependência total na disponibilidade da *API* gerida pela entidade externa, a sua indisponibilidade poderá causar um impacto na lógica da aplicação devido a esta dependência. Estes tipos de situações podem ser "contornados" através de SLAs (*Service Level Agreement*), garantia de *uptime* do serviço em percentagem num determinado intervalo de tempo, com possibilidade de implicações legais caso sejam quebradas.

3.5. Conceito "*Headless*"

Para percebermos o conceito de *headless* temos que entender a razão pela qual tem essa designação. Este termo, que pode ser traduzido à letra por "sem cabeça", refere-se à falta de uma *Graphical User Interface* por parte de uma aplicação ou *software*.

O exemplo mais simples de entender é o caso dos sistemas operativos *headless*. Estes geralmente disponibilizam apenas uma linha de comandos, pelo que não disponibilizam qualquer tipo de *UI* (*User Interface*) complexa (exemplo: Ubuntu *Server*).

Um exemplo associado ao tema em questão seria o de um *CMS*. Um *CMS* contém um *backend* para manuseamento de conteúdo e um *frontend* para mostrar esse mesmo conteúdo. Um *Headless CMS* faria com que este passasse a não ter parte gráfica e restasse apenas a parte lógica do mesmo [24].

3.5.1. Caso exemplo

Para facilitar a explicação das vantagens e desvantagens associadas à aplicação deste conceito, iremos começar por um exemplo que poderia refletir um caso real de negócio, que irá ser usado como o nosso "caso exemplo".

Imaginemos que uma empresa é proprietária de quatro *websites* diferentes associados ao comércio eletrónico digital com as seguintes características:

- Todos os *websites* estão assentes sobre a mesma *framework* de *CMS*;
- Três dos quatro *websites* têm uma lógica de negócio e tratamento de informação de maneira similar;
- Cada um destes *websites* contém um *frontend* bastante distinto.

Através das características referidas previamente, conseguimos extrair diferenças através da aplicação da metodologia tradicional de uso de um *CMS* e através do desacoplamento do mesmo (usando o conceito *headless*).

3.5.1.1. Aplicação da metodologia tradicional de um CMS ao caso exemplo

Consideremos "metodologia tradicional" a maneira habitual de implementação de um *website* onde um *backend* serve apenas um *frontend*.

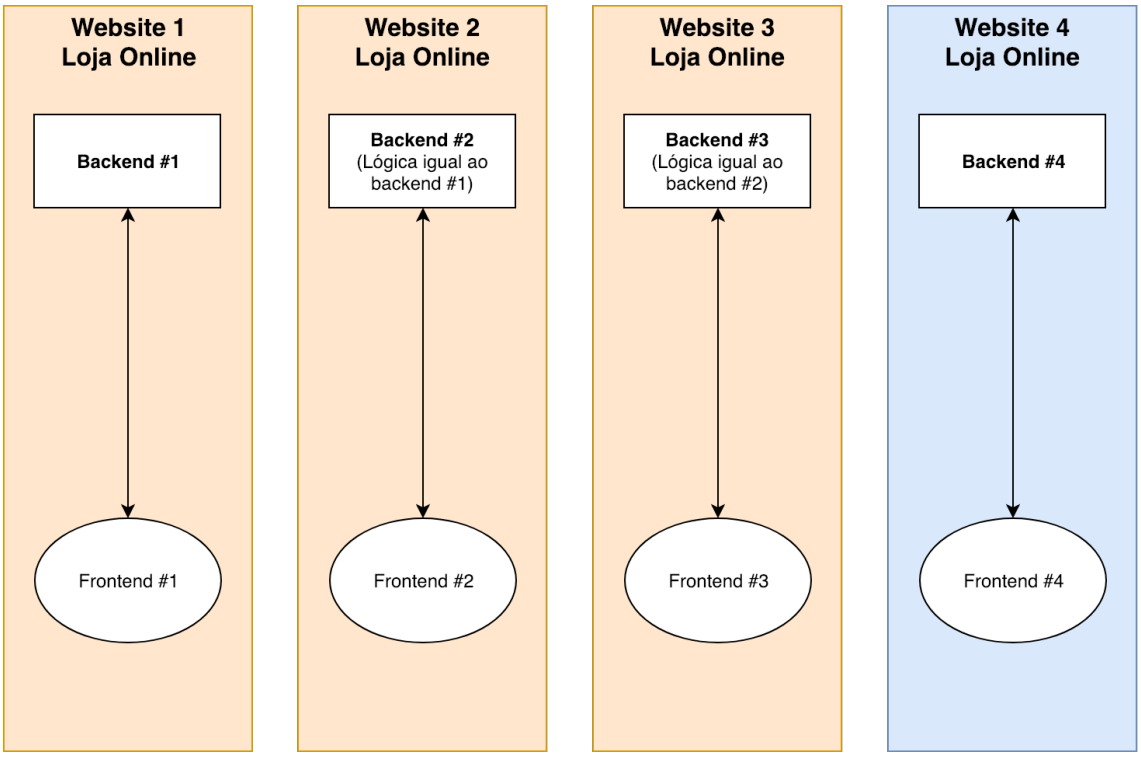


Ilustração 11 – Aplicação da metodologia tradicional de um *CMS* ao caso exemplo a 4 *Websites*

Através da Ilustração 11, consegue-se ter uma perspetiva visível deste caso de uso. Através da análise e descrição de características, conseguimos extrair algumas situações problemáticas. Estas situações serão descritas de maneira a que se perceba o seu contexto e

que se consiga engenhar uma possível solução. Enumerando os problemas que podemos extrair, conseguimos verificar que existe:

- **Redundância de código** – três *backends* com a mesma lógica de negócio ou que diferem vagamente entre eles, irão acabar por ter uma grande redundância de código, pelo que a maior parte da lógica poderia ser partilhada;
- **Custos acrescidos** – três *endpoints*, apesar de praticamente iguais, precisam de uma equipa para os gerir. Em caso de projetos em larga escala, é necessária uma equipa dedicada para cada um destes, o que poderá trazer custos acrescidos
- **Tempo acrescido de desenvolvimento** – Apesar de existirem ferramentas que permitem a rápida migração de código entre repositórios ou projetos, há que entender que uma migração de código poderá também herdar *bugs* ou falhas de segurança que não foram previamente encontradas. Estes tipos de situações trazem ainda mais complexidade ao ecossistema, que acaba por trazer perdas de tempo para encontrar a sua resolução.

3.5.1.2. Aplicação do conceito Headless ao caso exemplo

Após a explicação do conceito *Headless*, é possível agora demonstrar o resultado da sua aplicação ao caso exemplo. Como 3 dos 4 *websites* contam com uma lógica de *backend* equivalente, através da aplicação do conceito *headless* é possível obter o desacoplamento da nossa estrutura relativa aos *websites*.

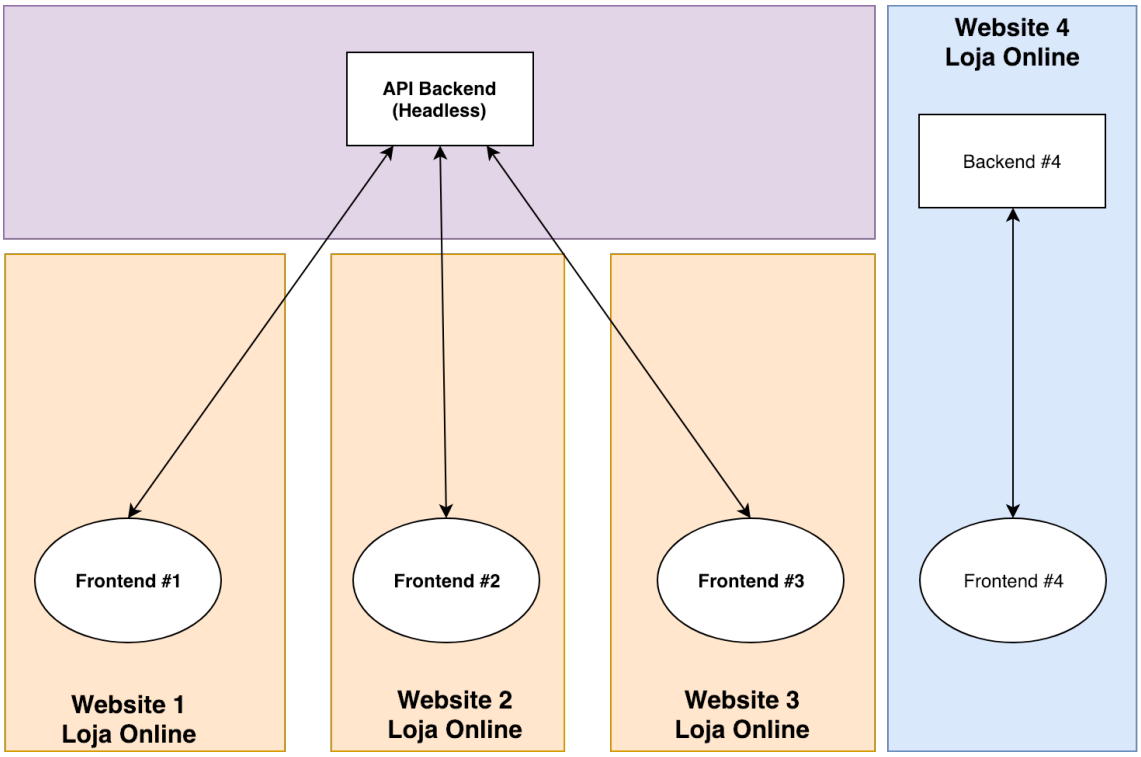


Ilustração 12 – Aplicação da arquitetura *headless* ao caso exemplo

Através da Ilustração 12 conseguimos verificar a reutilização de lógica através da utilização de apenas um nó de *backend*, que irá partilhar a lógica comum entre as várias lojas.

3.5.2. Vantagens

Nesta secção, irão ser abordadas as vantagens associadas ao conceito *Headless* aplicado a um *CMS* comparativamente ao seu módulo tradicional.

3.5.2.1. Flexibilidade e escalabilidade

Ao facto de falarmos de uma arquitetura que envolve o desacoplamento total de duas componentes que envolvem um *CMS*, estas podem trabalhar quase independentemente uma da outra. Cada uma destas pode ser modificada sem interferir com a outro, trazendo flexibilidade e escalabilidade ao ecossistema [25].

3.5.2.2. Redução de custos operacionais e tecnológicos

A arquitetura tradicional de um *CMS* indica que o *backend* e o *frontend* trabalham em conjunto como uma plataforma só. A relação associada a este tipo de metodologia trata-se de um para um, o que por outras palavras se pode traduzir que para cada *CMS* existe uma relação de um (*backend*) – um (*frontend*). Através do desacoplamento destes dois

componentes conseguimos fazer com que o nosso *website* consiga sofrer alterações numa destas componentes de maneira independente e com mais facilidade.

Para além da vantagem referida previamente, existe a redução de custos operacionais através da utilização do sistema de manuseamento de conteúdo, pelo que muitas das vezes deixa de ser necessária uma equipa especializada para tratar do conteúdo apresentado no *website* [26].

3.5.2.3. Estrutura bem definida

O consumo de uma *API* é uma realidade completamente diferente para o *frontend* relativamente ao método tradicional, em que a própria estrutura *HTML* (*HyperText Markup Language*) é enviada na própria resposta do servidor ou pedido a uma determinada página *web*.

Ao consumirmos uma *API* através do *frontend*, temos uma variedade maior de possibilidades a nível de estrutura e design da própria página *web*. Sabendo que o nosso pedido à *API* irá devolver uma determinada resposta (geralmente documentada) podemos rapidamente moldar-nos a esta resposta e apresentar facilmente o conteúdo pretendido [26].

3.5.3. Desvantagens

Nesta secção, irão ser abordadas as desvantagens associadas ao conceito *headless* comparativamente à metodologia tradicional de um *CMS*.

3.5.3.1. Ponto de falha comum

É relativamente fácil entender o principal problema e desvantagem associado à aplicação do conceito *Headless* ao nosso caso exemplo, representado pela Ilustração 13. Como os *CMS* deixaram de ter o seu *backend* dedicado e passaram a consumir uma *API* externa para tratamento de dados, estes passam a depender diretamente desse *endpoint*. Em caso de falha por parte da *API*, o *website* poderá ficar estagnado por completo, pois todo o processamento de lógica necessária não é efetuado.

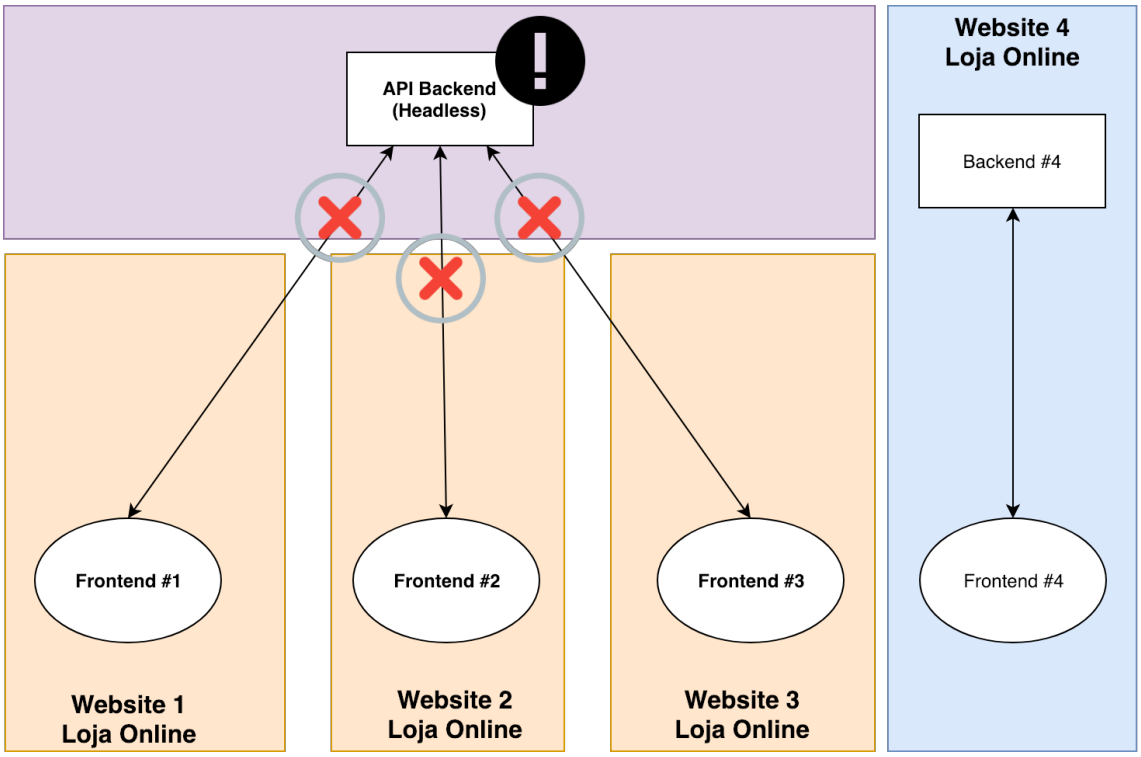


Ilustração 13 - Exemplo em caso de falha da *API*

3.5.3.2. Bugs e falhas de segurança

O consumo de lógica entre várias aplicações ao mesmo *endpoint* poderá ser bastante problemático. No caso de haver *bugs* ou falhas de segurança do *endpoint*, todos os consumidores irão partilhar este tipo de situações não desejadas.

Imaginando que o serviço que é consumido contém uma falha de segurança que abre a porta a ataques através de *SQL* (*Structured Query Language*) *injection*, existe uma grande probabilidade dos *endpoints* que consumam este serviço estejam suscetíveis a sofrer ataques deste tipo.

4. Estado da arte

A experiência profissional obtida através da utilização de várias plataformas e/ou ferramentas associadas ao *e-commerce* permite-me afirmar que é praticamente impossível generalizar uma solução que englobe todas as necessidades (uma aplicação *all-in-one*) de uma loja. A diferenciação das especificações e pormenores associados à lógica de negócio de cada loja é por vezes, demasiado grande e especifica. Este tipo de pormenores envolve os vários processos internos efetuados por parte da aplicação, como por exemplo: criação de encomendas; envio de *e-mails* associados a campanhas promocionais; entre outros.

Este capítulo tem como principal objetivo a pesquisa e investigação de aplicações ou plataformas já existentes relativas ao tema de "*Headless REST API* associado a *e-commerce*". Será efetuada uma descrição de cada uma destas, de modo a tentar extrair as suas possíveis vantagens e desvantagens. De modo a fazer parte desta lista, as soluções têm que ser complacentes com as seguintes características:

- Fornecem uma *API* com a possibilidade de trabalhar completamente desacoplada de um *frontend* (*headless*);
- Utilizam *REST* como modelo arquitetural;
- Têm aplicabilidade ao comércio digital.

Foram escolhidas três situações de aplicações distintas, com algumas características interessantes associadas à sua maneira de utilização. É possível verificar que as soluções são apresentadas em dois tipos de opções:

- **"Como serviço"** – Toda a arquitetura e funcionalidade é "servida" através de uma entidade externa onde o cliente final é responsável pelo consumo destas funcionalidades. A entidade responsável por consumir o serviço não tem acesso ao código fonte e é "obrigada" a consumir o serviço tal como este é disponibilizado, sem possibilidade de efetuar alterações ao seu *core*;
- **"Como plataforma"** - Toda a arquitetura e funcionalidade é disponibilizada através do código fonte do projeto. A entidade que obtém este código fonte pode efetuar as alterações necessárias ao *core* da solução de modo a adaptá-la às suas necessidades.

Existem alguns fatores que foram decididos como muito importantes na consideração das soluções descritas nos próximos tópicos. Estes fatores são:

- **Documentação –** Qualquer *API* depende de documentação concisa e bem escrita, de modo a que os programadores consigam integrar ou ajustar facilmente as funcionalidades à logica de negócio;

- **Exemplos (oficiais) de código a utilizar a API –** Os exemplos com integrações já efetuadas por parte da entidade associada à solução, são importantes para os programadores utilizarem-nos como ponto de referência ou até mesmo ponto de partida.

4.1. GetCandy

Solução de código aberto (Licença Apache-2.0) muito completa, construída através da utilização da *framework* Laravel (desenvolvido em *PHP*), com o objetivo de criar lojas *online*, fornecendo um sistema de administração que permite controlo total sobre as funcionalidades de cada uma das suas lojas [27]. Disponibiliza uma documentação bastante extensa com os passos necessários de instalação e de referências à utilização da *API*.

Em caso de necessidade, esta disponibiliza também de uma parte visual para associar à administração dos vários componentes, através da introdução de novos ficheiros disponíveis num dos repositórios criados pela mesma equipa, denominado de "Candy Hub" [28].

Sendo esta solução *open-source*, esta acaba por partilhar vantagens e desvantagens associadas às mesmas de um repositório com estas características:

- **Vantagens:**
    - Solução grátis e utilizável em ambiente comercial;
    - Fornece a possibilidade de implementação de novas funcionalidades, permitindo a adaptação aos requisitos necessários do utilizador da solução.
- **Desvantagens:**
    - Não sendo um serviço pago, não existe suporte dedicado ao cliente (como geralmente acontece com serviços pagos);
    - Falhas de segurança e introdução de *bugs* poderão ser adicionadas por qualquer pessoa que adicione novo código ao repositório.

4.2.CommerceLayer

Solução que fornece uma *API* responsável por tratar de toda a lógica associada ao comércio digital de uma maneira mais "simplificada". Esta simplificação ocorre devido ao facto de toda a lógica se encontrar previamente desenvolvida, sendo disponibilizada como "serviço". A entidade que pretende utilizar esta solução necessita apenas de consumir este serviço de modo a integrá-lo no seu sistema.

A aplicação em si disponibiliza uma secção de manuseamento de conteúdo fornecido pela *API* onde poderemos adicionar categorias, compras efetuadas, métodos de pagamento, manutenção de *stock*, entre outros no seu próprio *website*. Todo este conteúdo é associado a uma loja, de modo a ser consumida posteriormente. Existe também a possibilidade de integração de aplicações "fora da caixa" ao próprio *website*, de modo a poder utilizar várias funcionalidades da mesma (exemplo - funcionalidades do "Twitter" ou "Dropbox"). Através da integração destas aplicações, podemos depois automatizar determinados processos, como exemplo, caso adicionássemos um serviço de *e-mail*, poderíamos configurar para que fosse enviado um *e-mail* sempre que exista uma nova compra efetuada [29].

- **Vantagens:**
  - Oferece um plano flexivel para comerciantes de qualquer dimensão. Os preços de utilização do serviço começam nos 50€ mensais e escala dependendo de vários fatores como o número de encomendas, de produtos, entre outros;
  - O serviço fornece uma *API* que poderá ser rapidamente integrada num cenário de uma loja *online*.
- **Desvantagens:**
  - A flexibilidade de implementação de novas funcionalidades torna-se um pouco restrita, devido à solução ser disponibilizada como serviço. Não existe contacto direto com o código associado a esta;
  - Muita da informação é guardada do lado do CommerceLayer, o que poderá trazer problemas a nível de segurança e regulação dos dados associados aos clientes.

4.3. Moltin

Solução poderosa semelhante ao "CommerceLayer". Oferece um nível de customização maior [30], pois existe a possibilidade de adicionar novas propriedades a determinados componentes associados à plataforma [31] (como por exemplo os produtos), onde é possível definir o nome e o tipo de dados associado à propriedade (como por exemplo Boolean), que posteriormente poderá ser utilizado programaticamente, pois este passa a ser devolvido pela *API*.

Oferece planos a partir dos 1995$ por mês, um valor muito superior ao disponibilizado pelo "CommerceLayer". É possível contactar os serviços internos para obter um preço associado aos requisitos da plataforma, no entanto, o nível de customização dos requisitos pretendidos não consegue ser efetuado no momento, como disponibilizado no "CommerceLayer".

- **Vantagens:**
  - Oferece um nível superior de customização associado aos vários *endpoints*;
  - O *website* oferece múltiplos exemplos (em várias linguagens) para se começar um projeto.
- **Desvantagens:**
  - O preço alto associado ao plano inicial poderá obliterar a possibilidade de uma empresa mais pequena usar o serviço.

4.4. Comparação de soluções

Através das várias soluções encontradas, conseguimos extrair as várias caracteristicas associadas a cada uma das plataformas analisadas. nomeadamente:

- **Licença** – Licença associada ao código. Aplicável a soluções que disponibilizem código fonte;
- **Preço** – Preço aplicável para o uso da plataforma;
- **Integrações *out-of-the-box*** – Possibilidade de ativar integrações previamente feitas;
- **Disponibilização do código fonte** – O cliente final recebe o código fonte associado à solução;

- **Suporte ao cliente** – A entidade responsável pela solução presta suporte dedicado à entidade que se encontra a usar os seus serviços;
- **Exemplos oficiais com a *API*** – A entidade responsável pela plataforma fornece exemplos, a nível de programação, de aplicações que consomem o serviço;
- **Documentação da *API*** – A entidade responsável pela plataforma fornece uma documentação oficial sobre os vários *endpoints* da *API*;
- **Disponibiliza biblioteca de *endpoints*** – Existe uma biblioteca de *endpoints* que poderá ser importada em aplicações que permitem efetuar pedidos (Exemplo: Postman);
- **Disponibiliza interface para gestão** – A entidade responsável disponibiliza uma interface de gestão dos vários componentes (produtos, categorias, promoções, entre outros) da plataforma (o consumo da *API* não dependerá desta componente).

Tabela 1 - Comparação de vários componentes associados às soluções encontradas

Solução / **Componente**	**GetCandy**	**CommerceLayer**	**Moltin**
Licença	Apache-2.0	–	–
Preço	Grátis	A partir de 50€ / mês	A partir de 1995$ / mês
Integrações *out-of-the-box*?	Não	Sim	Sim
Disponibilização do código fonte	Sim	Não	Não
Suporte ao cliente	Não	Sim (*)	Sim (**)
Exemplos oficiais com a *API*	Não	Sim	Sim
Documentação da *API*	Sim	Sim	Sim
Biblioteca de *endpoints*	Não	Sim	Sim

Disponibiliza *interface* de gestão	Sim (***)	Sim	Sim

(*) apenas para planos "*Enterprise*"

(**) considerado *feature* adicional

(***) é disponibilizado num repositório à parte, pela equipa que criou a solução.

5. Metodologia

A gestão e planeamento de um projeto são dois aspetos que permitem uma melhor organização de uma equipa relativamente ao plano de desenvolvimento. Neste capitulo irá ser feito uma descrição associada ao Scrum, *framework* de trabalho utilizada durante o desenvolvimento deste trabalho. Será também descrito o *software* utilizado para gestão de todas as tarefas que foram utilizadas como *guideline*.

5.1. Processo de desenvolvimento

A entidade responsável pelo projeto não especificou uma metodologia concreta para o desenvolvimento do deste, pelo que decidi utilizar a heurística e valores associados à *framework Scrum*. Esta decisão foi tomada como a mais adequada devido à minha experiência profissional e devido à Blue-Infinity utilizar este tipo de forma de desenvolvimento em alguns dos seus projetos.

O *Scrum* implementa metodologias de trabalho de maneira a resolver problemas complexos através da construção do produto final de maneira iterativa e incremental [32]. Isto indica que o resultado é obtido através de várias entregas do produto, que permitem obter uma visão mais objetiva do que é pretendido pelo cliente final. É comum um projeto ser entregue ao cliente quando completo, no entanto, poderão existir especificações mal interpretadas por qualquer uma das partes (cliente e entidade responsável pelo desenvolvimento), que acabam por necessitar de novas modificações. Tendo em conta que o *Scrum* permite entregas incrementais, as modificações ao projeto ou alterações das especificações podem ocorrer de uma de uma maneira continua, com o objetivo de responder à "visão do cliente".

Os principais eventos associados a esta *framework* são [32]:

- ***Sprint* -** Intervalo de tempo associado à entrega de incrementos do produto. Com uma duração geralmente de 2 a 4 semanas. Este trabalho contou com sprints de 4 semanas;
- **Reuniões (*Scrums*) diárias -** Reuniões diárias de uma duração máxima de quinze minutos utilizada para sincronizar as atividades efetuadas e criar o plano de desenvolvimento das 24 horas seguintes. Estas reuniões devem ocorrer no mesmo local e à mesma hora de modo a facilitar este processo;

- **Planeamento de *sprints* –** Planeamento do trabalho a ser efetuado no *sprint* seguinte através da escolha de tarefas do *backlog* e do trabalho que poderá ser entregue na próxima iteração associada ao *sprint* [33];
- ***Sprint review* –** Evento que ocorre a cada fim de *sprint* de modo a verificar o incremento efetuado com o objetivo de adaptar o *backlog* se necessário [34].

O *Scrum* implementa várias funções entre os vários membros da equipa, sendo estas importantes para chegar a bom termo com o projeto. Conseguimos encontrar as seguintes funções numa equipa de Scrum [32]:

- ***Product owner*** - Cliente ou entidade responsável por gerir o *backlog* do produto. O *backlog* é constituído pelas funcionalidades que o cliente pretende ver no produto final. É representado neste projeto por um dos "responsáveis" pelo projeto já desenvolvido, descrito na secção 2.4;
- **Equipa de desenvolvimento** - Equipa responsável por desenvolver os incrementos do produto associados a um *sprint*;
- **Scrum *Master*** - Responsável por promover a teoria, práticas de desenvolvimento, regras e valores do Scrum.

De modo a percebermos algumas das razões pela qual esta forma de desenvolvimento foi escolhida, vão agora ser referenciadas algumas das vantagens e desvantagens associadas à utilização de Scrum [35]:

- **Vantagens**:
    - As reuniões diárias, permitem que a equipa se encontre atualizada em relação aos desenvolvimentos efetuados previamente e ao trabalho futuro;
    - As entregas incrementais permitem facilmente mudar o rumo de desenvolvimento, caso necessário.
- **Desvantagens**:
    - Por vezes, torna-se complicado o Scrum *Master* planear, estruturar e organizar um projeto que não esteja bem definido;
    - A constante mudança de especificações e rumo de desenvolvimento, pode levar por vezes a inconsistências e incertezas em relação ao produto final.

Existe a necessidade de referir que nem todas as regras associadas à *framework* Scrum foram seguidas "à risca", pois não houve reuniões diárias e *sprint reviews*, uma vez que a equipa de desenvolvimento e Scrum *master* foram representados pela mesma pessoa.

5.2. Contexto do Scrum no projeto

A utilização dos valores e heurísticas associadas ao Scrum não foram efetuados na sua totalidade. As entregas dos vários incrementos associados ao produto foram efetuadas com sucesso, no entanto, as meetings diárias e algumas datas associadas a *sprint reviews* não foram cumpridas. Este tipo de situação ocorreu devido muitas das vezes devido a indisponibilidade do *Product owner*.

Tal como dito previamente, as entregas dos incrementos associados ao produto foram efetuadas a cada sprint, o que acabou por facilitar a revisão por parte do cliente em relação ao produto em desenvolvimento. Estas revisões foram cruciais para uma das decisões que foram tomadas, e levaram o projeto num rumo algo diferente, descrito ao pormenor na secção 6.6.

5.3. Gestão de tarefas

A gestão de tarefas que foram realizadas ao longo do projeto, foram geridas através de um *software* disponibilizado pela "Atlassian" chamado "Jira" [36]. Este permite o registo e criação de tarefas (que no caso do "Jira" se designam como "*issues*") para que possam atribuídas aos vários elementos da equipa. Algumas das principais funcionalidades existentes e utilizadas neste *software* são:

- **Mudança de estado do *issue*** – Cada tarefa (ou *issue*) tem uma *tag* associada que define o estado atual da tarefa, como por exemplo: o estado "*In progress*" define que a tarefa está atualmente em desenvolvimento ou o estado "*done*" define que a tarefa foi finalizada [37];
- **Adição de comentários** – É possível adicionar comentários onde podemos mencionar outras pessoas que fazem parte do projeto, que acabam por receber um *e-mail* com a finalização de chamar a atenção. Esta funcionalidade foi utilizada com o propósito de descrever o que foi feito;
- **Acompanhamento de horas de desenvolvimento** – De modo a conseguir acompanhar as horas utilizadas para o desenvolvimento do projeto, é possível atribuir

as horas de estimativa, utilizadas e restantes. Esta funcionalidade foi utilizada para se obter uma perspetiva do trabalho necessário até concluir a tarefa;

- **Repartição de uma tarefa em subtarefas** – Por vezes as funcionalidades são demasiado grandes para serem resolvidas em apenas uma tarefa. Para resolver este tipo de situações, estas podem ser divididas em várias tarefas pequenas. Este tipo de situação aconteceu em a várias tarefas que foram divididas, como por exemplo a implementação de carrinho de compras.

6. Arquitetura e tecnologias

O requisito deste projeto consistia no desenvolvimento de uma aplicação associada a *e-commerce* através de uma *API* com arquitetura *REST* sem qualquer interface visual utilizando a *framework* Laravel, requisitado pelo cliente. A implementação de todo o projeto foi efetuada através da utilização da linguagem de *scripting PHP* em conjunto com o armazenamento de dados efetuado através da tecnologia *MySQL* com o propósito de criar uma aplicação *web* para ser disponibilizada como serviço.

Neste capítulo será descrita a arquitetura através da explicação dos seus componentes e camadas de abstração. Começamos por explicar a arquitetura conceptual, que envolve uma generalização de um pedido *HTTP* a uma *API*, seguido pela arquitetura do Laravel e finalizando com a arquitetura final obtida no projeto.

6.1. Tecnologias base

Tal como referido na introdução deste capítulo, a aplicação foi desenvolvida através do uso de *PHP* e *MySQL*, consideradas tecnologias "principais" associadas a este projeto.

6.1.1. PHP

PHP (acrónimo recursivo de "*PHP: Hypertext Preprocessor*") é uma linguagem de *scripting open-source* interpretada do lado do servidor. Esta foi criada especialmente para servir o propósito de desenvolvimento de aplicações *web*. Como referido previamente, esta é interpretada apenas do lado do servidor, o que quer dizer que nunca é o cliente (*browser* por exemplo) que faz qualquer tipo de processamento de informação [38].

Um ficheiro *PHP* tem como extensão ".php", e requer a abertura ("<?php") e fecho ("?>") de determinadas *tags* que dirão ao interpretador que o que está dentro destas, precisa de ser executado. É geralmente utilizada juntamente com código *HTML*, de modo a fornecer conteúdo dinâmico.

6.1.2. MySQL

Trata-se do mais popular sistema *open-source* de gestão de bases de dados relacional *SQL* desenvolvido e distribuído pela Oracle [39]. Este tipo de gestão permite-nos interagir com os conteúdos de uma base de dados, assim como a sua própria estrutura, incluindo tabelas, vistas, colunas, entre outros.

A escolha deste sistema foi efetuada com base no projeto antigo, que usa este mesmo sistema para o armazenamento de dados, assim como pela "boa relação" que existe entre o Laravel e MySQL. Também foi tido em conta o facto de este sistema ser resiliente e de resposta rápida quando submetido ao tratamento de informação substancial.

6.2. Arquitetura conceptual

A arquitetura conceptual é simples e consiste numa aplicação que disponibilize uma *API*. A comunicação com a *API* é efetuada através de pedidos *HTTP* com o objetivo de consumir a informação, através da devolução de uma resposta em formato *JSON* (*Javascript Object Notation*). Podemos exemplificar dois casos de uso para a utilização desta, que demonstram dois objetivos diferentes:

- Pedidos efetuados por componentes visuais – Uma plataforma *Headless* não fornece componentes visuais, no entanto, poderá ser consumida por aplicações dedicadas a tal. Neste caso a informação será devolvida já previamente tratada para consumo dos componentes visuais das referidas aplicações. Este é o caso principal associado à criação deste projeto;
- Pedidos efetuados por componentes associados a lógica – O consumo da *API* criada pode também ser utilizada por outro componente associado a lógica com o objetivo de obter informação para ser posteriormente utilizada.

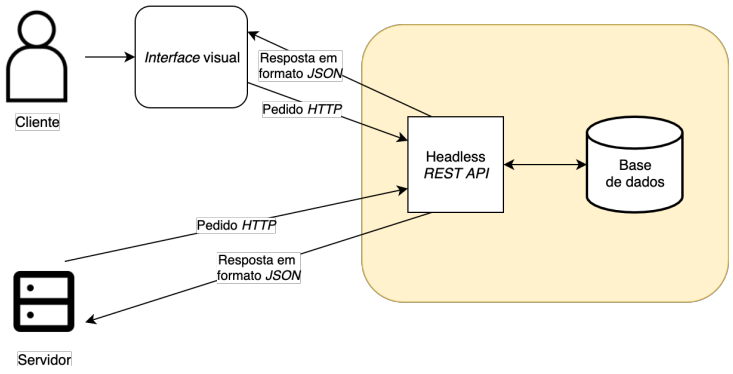


Ilustração 14 - Arquitetura de alto nível da aplicação

Através da Ilustração 14 conseguimos verificar os dois casos de uso explicados previamente. A arquitetura da aplicação torna-se muito mais complexa quando se pretende descrever processos e fluxos internos associados à *framework* utilizada.

6.3. Arquitetura do Laravel

O Laravel foi construído sobre o padrão arquitetural *MVC* (*Model View Controller*). Embora neste projeto o MVC não seja utilizado na totalidade, é importante descrevê-lo para melhor entender o funcionamento da *framework* Laravel.

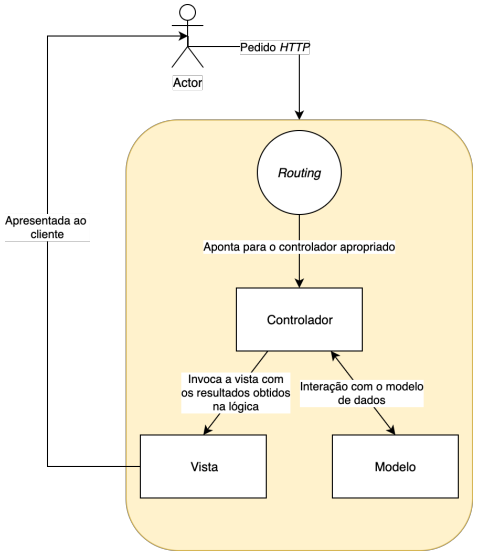


Ilustração 15 - Arquitetura MVC associada ao Laravel

Aplicações que sigam esta arquitetura têm características bem definidas, onde existem três componentes principais responsáveis pela "engrenagem" e estrutura desta, representadas pela Ilustração 15 [40]:

- **Modelo** – Funciona como uma "ponte" entre a vista e o controlador, e é responsável por guardar e gerir a informação geral utilizada por toda a aplicação (não confundir com base de dados, o modelo gere e guarda a informação através da interação com a própria base de dados);
- **Vista** – Componente responsável por gerar e mostrar a *UI* (*User Interface*) ao utilizador final, utilizando a informação previamente processada pelos outros componentes;
- **Controlador** – Estabelece uma ligação entre o modelo e a vista. Responsável por receber o pedido, processar os dados provenientes deste e devolver uma resposta associada ao pedido (inclusive a própria vista).

.

6.4. Arquitetura do projeto

De modo a conseguirmos ter uma visão geral da arquitetura da aplicação, vamos descrever os seus componentes e camadas de abstração.

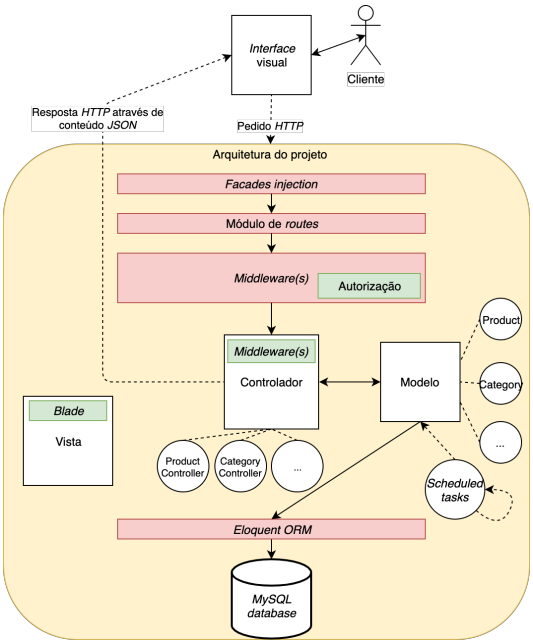


Ilustração 16 - Arquitetura de baixo nível da aplicação

A Ilustração 16 demonstra a arquitetura, a baixo nível associada ao projeto, em que serão explicados os vários componentes e camadas que a compõe. Vamos agora explicar, por subtópicos, os componentes e camadas de abstração quando é efetuado um pedido *HTTP* à *API*.

6.4.1.1. Facades injection

Apesar de haver uma descrição bastante técnica de *facades*, irá ser explicado o conceito geral a "alto nível". A utilização de *facades* permite disponibilizar em várias partes da aplicação, utilitários associados a funcionalidades do Laravel (por exemplo: *cache*, sessão, autenticação, entre outros). Estes utilitários contêm pedaços de código injetados durante o *bootstrapping* permitindo à *framework* "saber" que utilitários deverá disponibilizar pela aplicação. Existem vários componentes do Laravel (como por exemplo o *Eloquent*) que dependem diretamente de *facades*, o que explica a sua injeção no inicio do fluxo aplicacional.

6.4.1.2. Routes

As *routes* associadas ao Laravel permitem mapear o método e controlador para onde aponta um determinado *URL*/*URI*. Este tipo de mapeamento permite responder à pergunta

efetuada por parte da própria *framework* "Que lógica devo executar para este pedido?". Isto permite a criação de aplicações *RESTful* com variados parâmetros e comportamentos, como por exemplo [41]:

- **Definir grupos de *routes*** – A definição de grupos de *routes* permite partilhar atributos entre *routes* que estejam dentro do grupo (como por exemplo prefixo, *middlewares*, parâmetros, entre outros). Esta funcionalidade deverá ser utilizada de modo a prevenir repetição de código e a melhorar a estrutura geral da aplicação;
- ***Data binding*** – O Laravel permite, através de *routes*, conectar um identificador a uma entidade específica. Um exemplo de aplicação poderá ser a definição da *route* "users/{ID do utilizador}", em que a *framework* irá automaticamente conectar o identificador à entidade "User", neste caso.

6.4.1.3. Middlewares

Mecanismo de filtragem de pedidos *HTTP* ao entrar na nossa aplicação. São geralmente associados a lógica que é processada com o objetivo de efetuar verificações antes de executar algum extrato de código, com a possibilidade de serem injetados em controladores ou até nas próprias *routes*. De uma maneira geral, um *middleware* permite a execução de lógica previamente a um determinado pedido ser executado, e apenas após a execução dessa lógica é que o pedido será processado (ou não). No contexto deste projeto, foram adicionados *middlewares* de baixo e alto nível, explicados em detalhe na secção 7.4.4.

6.4.1.4. Autenticação e autorização

O processo de autenticação é uma das funcionalidades comuns presentes em qualquer aplicação *web*. A lógica associada a este processo é geralmente complicada de implementar devido às possíveis falhas de segurança que esta pode trazer. O Laravel traz um sistema de autenticação *out-of-the-box*, que pode ser adicionado à nossa aplicação através de um simples comando.

6.4.1.5. Eloquent ORM (Object-Relational Mapping)

Camada de abstração que permite uma interação simples e intuitiva com a base de dados. Os modelos do projeto são associados automaticamente a uma tabela especifica, onde podemos depois efetuar *queries* à base de dados através da utilização de funções específicas do próprio modelo. O Eloquent fornece uma camada de abstração dos procedimentos à base dados, que permitem rapidez e segurança no desenvolvimento, pois esta inclui automaticamente os seguintes processos:

- Conexão à base de dados através das configurações nas variáveis de ambiente;
- Relacionamento com a tabela associada ao modelo;
- Gera as *queries* através do método chamado durante a utilização do modelo;
- Uso de interface *PDO* (*PHP Data Objects*) para prevenir injeções de *SQL* não desejado;
- Execução de *queries* e tratamento do seu resultado.

Apesar de existirem muitas especificidades dos vários sistemas de gestão de base de dados, a única configuração que o Laravel necessita será, através do ficheiro de configuração, especificar que sistema de gestão de base de dados está atualmente a ser utilizado.

Como previamente descrito, foi utilizado *MySQL* como tecnologia de armazenamento na base de dados e sendo esta completamente suportada por parte do Laravel, não houve necessidade de efetuar qualquer tipo de configurações ou alterações que envolvessem a mudança da camada de abstração.

```
$product = new Product();
$product->name = $request->name;
$product->description = $request->description;
$product->stock = $request->stock;
$product->price = $request->price;
$product->discount = $request->discount;
$product->store_id = $request->store_id;
$product->created_by = Authentication::user()->id;
$product->save();
```

Ilustração 17 - Exemplo da utilização de *ORM* na criação de um produto

Através da Ilustração 17 conseguimos verificar o exemplo da criação de uma entidade Produto na base de dados. Esta é feita através da instanciação da entidade respetiva, seguida da atribuição dos valores associados às propriedades desta entidade, concluindo o processo através da função "save" que acaba por armazenar esta informação na base de dados. Como podemos verificar, não existiu qualquer necessidade de se iniciar a ligação à base de dados, criar a *query*, ou qualquer outro passo associado ao processo tradicional de inserção de dados através de SQL.

6.4.1.6. Scheduled tasks

As *scheduled tasks* (agendamento de tarefas) são uma funcionalidade que é utilizada quando se pretende que um determinado processo seja executado num determinado intervalo de tempo em segundo plano sem necessidade de intervenção manual. Esta *framework* disponibiliza este sistema *out-of-the-box*, permitindo definir tarefas. Com uma simples linha

de código, podemos definir por exemplo, que todos os registos de uma tabela sejam removidos diariamente às 02:00 por exemplo. Todo o processo é depois efetuado de forma automática.

No contexto do projeto, as tarefas agendadas foram utilizadas para executar lógica "remover logs", detalhado na secção 7.4.5. Como podemos verificar através da Ilustração 16, não é necessário qualquer tipo de interação direta com as tarefas para estas serem executadas pois existe um componente dedicado a verificar que tarefas deverão ser despoletadas. As tarefas são pedaços de lógica que têm acesso aos modelos e controladores.

É de notar que as tarefas agendadas cumprem rigorosamente o horário a ser executadas. Caso ocorra um erro que interrompa o fluxo aplicacional, a execução da tarefa pode não ocorrer.

6.5. *MVC* no contexto do projeto

Através da Ilustração 16 conseguimos verificar a utilização do padrão arquitetural *MVC* através dos componentes Modelo, Vista e Controlador. No entanto, como este projeto implementa uma *Headless API*, não existe qualquer tipo de interação por parte da vista. Esta situação levantou algumas questões em relação à arquitetura final do projeto, uma vez que não fazia sentido deixar componentes inutilizados pois estes continuam a ser. Esta situação acaba por causar impacto negativo no desempenho da *API*, uma das suas características mais importantes. Deste modo, acabou por ser efetuada uma pesquisa por possíveis soluções associadas a Laravel (sendo este um requisito), que fossem direcionadas para o desenvolvimento de *APIs*. A solução encontrada foi a *framework* Lumen, uma versão *lightweight* do Laravel "dedicada" à produção de *APIs* com foco especial no desempenho da aplicação.

Devido ao estado avançado do projeto quando foram levantadas estas questões e após uma breve discussão com o cliente (Blue-infinity), foi acordado que o projeto seria migrado para Lumen, com a espectativa de se obter uma arquitetura ideal que encaixasse no âmbito do tema em questão.

6.6.Migração para Lumen

O Lumen é uma *micro-framework*, uma versão mais "leve" do Laravel especialmente criada para serviços *web* ou *APIs*, onde são apenas incluídas as ferramentas e componentes

estritamente necessários para o desenvolvimento deste tipo de aplicações. Algumas das camadas de abstração não vêm ativas por defeito, mas são incluídas com o Lumen e podem ser ativadas durante o seu *bootstrapping*. De uma maneira geral, esta é mais simples de configurar, oferece um maior desempenho e aguenta com uma maior carga de pedidos que o próprio Laravel [42].

No contexto da arquitetura demonstrada através da Ilustração 16, o Lumen não traz de raiz o *templating engine* "Blade" responsável pelo componente visual (vistas), tem um *bootstrapping* muito mais simplificado, o módulo associado à autenticação não é suportado nativamente e o Eloquent *ORM* vem desativado por defeito.

De modo a reaproveitar todo o trabalho desenvolvido até à migração, foi necessário efetuar mudanças em várias partes do projeto, no entanto, é necessário descrever previamente a arquitetura obtida pós-migração.

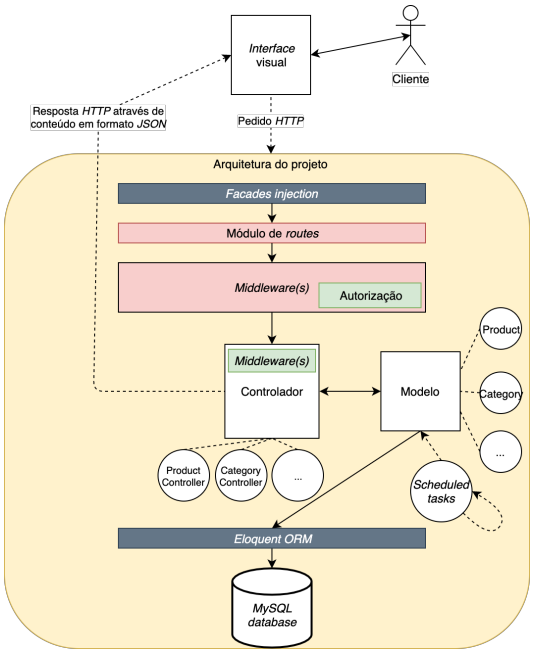


Ilustração 18 - Arquitetura do final da aplicação após migração para Lumen

A Ilustração 18 demonstra a arquitetura da aplicação pós-migração. Como podemos verificar, a componente associada à vista (juntamente com o *templating engine* "Blade") já não faz parte da arquitetura. As camadas de abstração representadas por um retângulo cinzento, indicam que estas não fazem parte do Lumen, mas foram ativadas ou modificadas para fazerem novamente parte do fluxo aplicacional. As razões associadas serão devidamente argumentadas nos tópicos seguintes.

6.6.1. Ativação do Eloquent ORM

Devido ao estado avançado do projeto pré-migração que envolvia já um modelo de dados bem definido, foi decidido que o *ORM* (utilizado por toda a aplicação) seria ativado no Lumen para evitar a "tradução" completa para *raw queries* (SQL). Apesar do impacto negativo no desempenho geral da aplicação, o cliente aceitou a ativação do ORM devido à relação utilidade/desempenho justificada em estudos efetuados que comparam o Eloquent *ORM* a *raw queries* (SQL) [43].

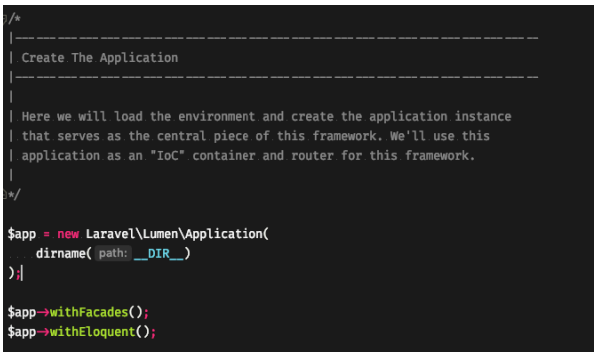


Ilustração 19 - Ativação do Eloquent *ORM* durante o *boostrapping* da aplicação

A ativação do *Eloquent ORM* (Ilustração 19) é efetuada após o *boostrapping*, no entanto, esta funcionalidade exige a injeção das *facades*, devido à sua dependência com outros componentes associados ao Laravel. Esta situação justifica a razão para qual a presença da "*Facades injection*" ser mantida na arquitetura final na aplicação.

6.6.2. Modificações ao módulo de autorização

O módulo de autenticação associado ao Laravel não faz parte do Lumen, no entanto, foi criada a lógica necessária que fornecesse um sistema de autorização *stateless* a pedidos à *API* através de *tokens*, embutido diretamente num *middleware*. Simplificando a explicação associada a este processo, os pedidos efetuados que necessitem de autenticação, deverão enviar o parâmetro obrigatório "api_token" para ser efetuado o *handshake* com o servidor. Esta funcionalidade será explicada em detalhe na secção 7.4.4.1 durante a descrição das funcionalidades implementadas.

7. Implementação

Este capítulo irá descrever o modelo de dados em detalhe, a relação entre as várias entidades e as funcionalidades adicionadas à solução. Apesar da descrição das tecnologias utilizadas ter ocorrido na secção 6.4, não foram enquadradas todas as que foram utilizadas no projeto, que acabarão por ser explicadas neste capítulo.

7.1. Modelo de dados

Esta secção tem como objetivo descrever o modelo de dados associado ao projeto, assim como as várias modificações que foram feitas e a sua razão. Cada tabela será pormenorizada de modo a perceber o papel de cada um na aplicação. O modelo de dados final conta com o total de vinte entidades "genéricas" com utilidades associadas a qualquer plataforma inserida no âmbito de *e-commerce*. Estas entidades são:

- **Store** - Representa uma loja na aplicação. Tem como função principal definir a loja a que o cliente está a aceder (devido à funcionalidade *multi-store*, como descrito na secção 7.4.2), através do *URL* (*Uniform Resource Locator*) de acesso. Internamente, a loja definida irá influenciar os recursos disponibilizados para essa determinada loja (categorias, produtos, entre outros);
- **User** - Representa um utilizador na aplicação. Tem como função principal permitir definir uma conta para cada utilizador, onde a informação destes pode ser utilizada para efetuar *login* e aceder às diferentes áreas da aplicação;
- **PasswordReset** - Representa um pedido de mudança de *password* por parte de utilizador. Este tem como função criar uma chave que irá permitir que o utilizador mude a sua password com "segurança". É geralmente utilizado através do envio de um *link* para o *e-mail* do utilizador com esta chave;
- **Category** - Representa uma categoria na aplicação. Tem como função principal permitir associar vários produtos a este, de modo a que se consiga extrair produtos de uma determinada categoria, muitas das vezes utilizado para filtragens;

- **Product** - Representa um produto na aplicação. Tem como função principal a sua disponibilização aos clientes, de modo a que estes possam efetuar várias ações (adicionar ao carrinho, comprar, entre outros);
- **Review** - Representa uma opinião associada a um produto. Tem como função principal ser utilizado para que os clientes verifiquem opiniões de outros clientes;
- **Promotion** - Representa uma promoção na aplicação. Tem como função descontar um valor especifico a um produto ou encomenda;
- **Usergroup** - Representa um grupo, que poderá ser associado a diferentes utilizadores. Este poderá ter vários privilégios;
- **Privilege** - Representa um privilégio na aplicação, definidos programaticamente. A função destes é verificar se um utilizador tem a permissão de efetuar uma determinada ação;
- **Image** - Representa uma imagem na aplicação. Esta foi desenvolvida de maneira genérica e a sua função é associar uma imagem a uma determinada entidade sem qualquer tipo de restrições;
- **Order** - Representa uma encomenda efetuada. Cada encomenda pode ter vários produtos e descontos;
- **PromotionCode** - Representa os códigos associados às promoções;
- **Basket** - Representa o carrinho de compras associados a uma sessão de um utilizador;
- **Log** - Representa uma entrada de informação adicionada pelo sistema;
- **PromotionProduct** - Representa a ligação entre promoções e produtos. Os clientes poderão apenas aplicar promoções que estejam ativas nos produtos. Define os produtos que estão disponíveis à aplicação de uma determinada promoção;
- **OrderProduct** - Representa a ligação entre encomendas e produtos. Define os produtos associados a uma determinada encomenda. Esta entidade inclui uma propriedade que tem como objetivo guardar o preço do produto no momento exato em que a compra foi efetuada, de modo a prevenir que este seja perdido quando este é alterado;
- **UsergroupPrivilege** - Representa a ligação entre usergroups e privilégios. Define os privilégios associados a um usergroup;

- **ProductCategory** - Representa a ligação entre produtos e categorias. Define os produtos associados a uma determinada categoria;
- **UserUsergroup** - Representa a ligação entre utilizadores e usergroups. Define os usergroups associados a um determinado utilizador;
- **BasketProduct** - Representa a ligação entre o carrinho de compras e produtos. Define os produtos que estão inseridos num determinado carrinho de compras.

Após a apresentação de cada entidade juntamente com o seu significado no projeto, irão agora ser apresentadas as propriedades de cada uma com a seguinte legenda:

- As propriedades a sublinhado representam uma chave primária;
- As propriedades a tracejado representam uma chave estrangeira.
- As propriedades a vermelho representam uma chave única.

Tabela 2 – Entidades e propriedades da solução final

Entidade	**Propriedades**
Store	id, name, created_at, updated_at
User	id, name, email, password, api_token, type, store_id, remember_token, s_city, s_address, b_city, b_address, created_at, updated_at
PasswordReset	email, token, created_at
Category	id, name, created_by, store_id, description, created_at, updated_at
Product	id, name, description, price, stock, discount, status, created_by, store_id, created_at, updated_at
Review	id, product_id, created_by, review, star, created_at, updated_at
Promotion	id, name, description, discount, discount_type, created_by, store_id, available_from, available_until, created_at, updated_at
PromotionProduct	promotion_id, product_id
Usergroup	id, name, created_by, store_id, description, created_at, updated_at
UserUsergroup	user_id, usergroup_id

Privilege	id, name, description, created_by, created_at, updated_at
UsergroupPrivilege	usergroup_id, privilege_id
ProductCategory	product_id, category_id
Image	id, entity, entity_id, description, path, created_at, updated_at
Order	id, user_id, raw_value, discounted_value, value, store_id, deleted_at, created_at, updated_at
OrderProduct	order_id, product_id, price
PromotionCode	id, code, created_by, promotion_id, burned, created_at, updated_at
Basket	id, user_id, promotion_code_id, active, s_city, s_address, b_city, b_address, created_at, updated_at
BasketProduct	basket_id, product_id, quantity
Log	id, severity, action, details, created_at

7.2. Descrição das propriedades por entidade

Após a análise da Tabela 2, podemos verificar que existem vários modelos com propriedades comuns. Estas representam na generalidade um comportamento equivalente, independentemente do modelo em que estão inseridas. Para prevenir a repetição da explicação de cada uma destas, estas propriedades serão descritas previamente ao detalhe das propriedades de cada modelo, de modo a serem omitidas.

Tabela 3 - Propriedades comuns entre alguns dos modelos.

Propriedade	**Detalhes**
id	Identificador principal dos vários modelos. Valor atribuído automaticamente através de incrementação quando ocorre a inserção deste na base de dados.
name	Nome associado ao modelo, geralmente utilizado como identificador "secundário".
description	Descrição do modelo, é utilizado para adicionar alguma informação extra que não pode ser adicionada ao nome.
created_at	Data de inserção do modelo.
updated_at	Data da última modificação ao modelo.
deleted_at	Data de eliminação do modelo. Usado para propósitos de *soft delete*.

store_id	Identificador do modelo Store associado a este modelo. Utilizado para propósitos de *multi-store*.
product_id	Identificador do modelo Product.
created_by	Identificador do modelo User responsável pela inserção. Utilizado para propósitos de *logging* e auditoria.

De seguida, será explicado o objetivo das propriedades associadas às entidades, é de notar que as propriedades presentes na Tabela 3 não serão apresentadas novamente. Caso um determinado modelo não seja descrito, indica que contém apenas propriedades "comuns". A descrição das propriedades das entidades é a seguinte:

- **Entidade Store**:
  - storefront_url – *URL* que define o acesso a esta loja.
- **Entidade User**:
  - email – email associado ao cliente;
  - api_token – Credencial utilizada para efetuar pedidos à *API;*
  - password – Credencial utilizada para efetuar *login*. Devido a esta se tratar de dados sensíveis, o seu valor é definido através da utilização de uma função de *hashing* "bcrypt", nativa da *framework* Laravel;
  - type – Tipo de utilizador. Define a função deste na aplicação – cliente, administrador ou super-administrador, permitindo o acesso a diferentes áreas;
  - s_city – Cidade associada ao envio da encomenda;
  - s_address – Morada associada ao envio da encomenda;
  - b_city – Cidade associada à morada fiscal do utilizador;
  - b_address – Morada fiscal do utilizador;
  - remember_token – Chave associada ao utilizador que permite que este fique com a sessão ativa infinitamente.
- **Entidade PasswordReset**:
  - email – Email do utilizador que pretende efetuar o reset da sua password;
  - token – Chave aleatória gerada durante o pedido de reset da password, geralmente utilizada quando se envia um email com um link de password reset.

- **Entidade Product**
    - price – Preço atual do produto;
    - stock – Unidades disponíveis para compra. Não podem ser adicionados produtos ao carrinho caso o seu valor seja "0";
    - status – Estado atual do produto. Este pode estar ativo (valor "1") ou inativo (valor "0"). Os produtos inativos não são apresentados aos clientes;
    - discount – Desconto direto associado ao produto.
- **Entidade Review**:
    - review – Descrição da opinião do utilizador;
    - star – Número de estrelas associadas à opinião do utilizador.
- **Entidade Promotion**:
    - discount – Valor do desconto;
    - discount_type – Tipo de desconto, se é valor directo ou percentagem.
    - available_from – Data em que esta promoção é começa a ser utilizável;
    - available_until – Data que dita o fim da promoção. Esta deixa de ser utilizável após esta data.
- **Entidade Image**:
    - entity – Entidade associada a esta imagem, exemplo - "Product", "Category";
    - entity_id – ID associado à entidade definida através da propriedade "entity" nesta entidade;
    - path – Caminho no sistema associado à imagem.
- **Entidade Order**:
    - raw_value – Valor total da encomenda, sem qualquer tipo de descontos;
    - discounted_value – Valor total descontado da encomenda;
    - value – Valor final da encomenda após os descontos ao valor.
- **Entidade PromotionCode**:
    - code – Código a utilizar;
    - burned – Representa o estado do código. Códigos usados não poderão ser novamente utilizados. Valor por omissão: "0".

- **Entidade Basket**:
    - promotion_code_id – ID de um código de promoção previamente inserido;
    - s_city – Cidade associada ao envio da encomenda, automaticamente preenchida com a "s_city" associada ao utilizador quando um registo desta entidade é criado;
    - s_address – Morada associada ao envio da encomenda, automaticamente preenchida com o "s_address" do utilizador quando um registo desta entidade é criado;
    - b_city – Cidade associada à morada fiscal associada à encomenda. Preenchida automaticamente com a cidade fiscal da entidade User;
    - b_address – Morada fiscal associada à encomenda. Preenchida automaticamente com a morada fiscal da entidade User;
    - active – Representa se um determinado carrinho de compras se encontra ativo ou foi descartado.
- **Entidade Log**:
    - severity - Severidade associada ao *log*. O valor está associado aos tipos disponíveis apresentados na Tabela 5.;
    - action - Informação sobre a ação que ocorreu, de modo a facilitar a pesquisa, exemplo: "encomenda criada";
    - details - Detalhes sobre o *log*, inclui por exemplo as modificações que foram efetuadas a uma entidade, ou o fluxo aplicacional no caso de erros.

7.3. Tecnologias e ferramentas utilizadas

Nesta secção serão descritas as várias ferramentas e tecnologias utilizadas que não ficaram enquadradas na secção da arquitetura, no entanto, fizeram parte do projeto.

7.3.1. Nginx

O Nginx é um projeto de código aberto que serve como servidor *web* escalável, de alta *performance* e disponibilidade, com capacidades de servir como servidor de *proxy* e acelerador *web*. Oferece uma arquitetura muito diferente do Apache (outro servidor *web* com grande taxa de utilização), com um resultado muito positivo, onde o Nginx por vezes consegue servir um maior número de pedidos em comparação com o Apache [44].

No contexto deste projeto, o Nginx foi utilizado como servidor *web*. A decisão de escolha associada à utilização desta ferramenta foi baseada no facto de esta vir por omissão com este servidor *web* na máquina virtual disponibilizada no *website* oficial do Laravel [45].

7.3.2. Laravel *framework*

Trata-se de uma das mais populares *frameworks* de *PHP*, *open-source*, que segue os padrões de arquitetura *MVC*, criada por Taylor Otwell em 2011, a sua taxa de utilização tem sido cada vez maior devido à sua *performance*, facilidade de utilização. A sua capacidade de escalabilidade é também um fator que ajudou ao seu sucesso, pois o Laravel foi desenvolvido para trabalhar de maneira modular, ou seja, por módulos. Estes módulos podem ser desenvolvidos "à parte", que podem depois ser integrados com a aplicação, em que a interação com os outros componentes continua a ser efetuada com sucesso [46].

Existe a necessidade explicar algumas das funcionalidades core associadas o Laravel, pois muitas destas influenciaram, de maneira geral, as várias decisões que foram tomadas ao longo do desenvolvimento deste projeto.

7.3.2.1. Caching

A *cache* tem um papel fundamental na melhoria de *performance* de qualquer sistema. O objetivo desta é o armazenamento de informação pré-processada num local de armazenamento temporário, geralmente num ficheiro de texto. Esta poderá depois ser rapidamente extraída através do ficheiro, de forma a prevenir o processamento da mesma (pois esta já foi processada). O Laravel traz esta funcionalidade de maneira transparente, metendo em *cache* vários elementos da *framework*, sem qualquer tipo de configuração.

7.3.2.2. Sistema de migrações e seeders

Atualizar conteúdo da base de dados manualmente, é geralmente considerado uma má prática, especialmente em projetos de média/larga escala. De maneira a automatizar e a tornar este processo menos propício a erros, foram criadas as migrações, que são totalmente suportadas pelo Laravel. Estas atuam como um sistema de controlo de versões para a base de dados [47], onde mudanças granulares a esta são adicionadas em scripts diferentes, que são executados dependendo da versão da última migração processada. O objetivo destas migrações é a possibilidade de adicionar mudanças à base de dados de maneira contínua, sem a necessidade de criar ficheiros específicos da base de dados atual que estamos a usar, e executar estas modificações manualmente.

Também relativo às migrações, existe outro componente denominado de *seeders*, que fornece a possibilidade de preencher a base de dados com informação através da execução de um comando. As classes associadas aos *seeders* têm alguma lógica básica que permite definir que estes tipos de dados têm de ser inseridos em cada coluna específica associada ao modelo. Os dados poderão ser *hardcoded* ou gerados *on-the-fly* através da utilização de um utilitário associado à plataforma denominado de "*Faker*", que permite gerar vários tipos de informação ficticia (nomes, países, *e-mails*, entre outros) que podem ser atribuídos aos registos da base de dados. O objetivo desta funcionalidade é permitir que a base de dados seja preenchida com informação de forma automatizada para evitar o trabalho da sua inserção manual.

No contexto deste projeto, as migrações foram utilizadas para criar todas as tabelas necessárias (a sua descrição pormenorizada será feita na secção 7.1) ao funcionamento do projeto, enquanto os *seeders* foram utilizados para criar todo o tipo de informação necessária à base de dados, nomeadamente:

- ***Root store*** – Loja principal e obrigatória da aplicação, não existe possibilidade da sua remoção;
- **Administrador total** – Conta de utilizador associada ao administrador que tem acesso ao contexto total da aplicação;
- ***Usergroups*** – Usergroups predefinidos associados aos controladores e ações existentes;
- **Privilégios** – Privilégios predefinidos associados aos usergroups e ações dos controladores.

7.3.3. IntelliJ IDEA

Desenvolvido pela equipa JetBrains, o IntelliJ IDEA é um *IDE* que fornece suporte a várias linguagens de programação e proporciona várias funcionalidades como [48]:

- Análise de código;
- Compreensão e conexão entre símbolos (classes, propriedades, métodos, ficheiros, entre outros) do projeto;
- Navegação rápida entre os vários símbolos do projeto;
- Análise de erros com possível exemplo ou sugestão de resolução;
- *Refactoring* de todo o tipo de símbolos.

Para além destas funcionalidades principais, muitas das *frameworks* mais conhecidas podem ser utilizadas usando este *IDE*, pois este reconhece e adiciona os símbolos específicos dessa, de forma a serem utilizados facilmente.

Este *IDE* fornece a possibilidade de integração com *plugins*, muitas das vezes criados pela própria comunidade. O objetivo destes é integrar funcionalidades ainda não existentes na própria ferramenta de desenvolvimento. Para este projeto, os *plugins* usados foram:

- **Laravel** – Adiciona funcionalidades e símbolos específicos do Laravel de um modo mais extensa (o suporte básico já vem com o próprio *IDE*);
- **EnvFile** – Adiciona suporte aos ficheiros que definem variáveis de ambiente.

De notar que esta ferramenta exige uma licença paga dependendo a versão do *software*. A versão utilizada durante a implementação da *API* foi "2019.1.2 (*Ultimate edition*)" através da utilização de uma licença de estudante.

7.3.4. Postman

Ferramenta utilizada para testar pedidos *HTTP*. É geralmente utilizada para testar *RESTful APIs* pois simplifica o processo de configuração e execução do pedido.

De modo a configurar o pedido *HTTP*, existem várias "variáveis" que podem ser adicionadas e/ou alteradas das quais podemos referir:

- *URL* do *endpoint* que será afetado pelo pedido;
- *Headers*;
- Parametros do *URL* e do corpo do pedido;
- Tipo de autorização;
- *Cookies*;
- Entre outros.

Após efetuarmos a configuração, é possível extrair em várias linguagens de programação ou comando usado na consola, de forma de efetuar o pedido manualmente.

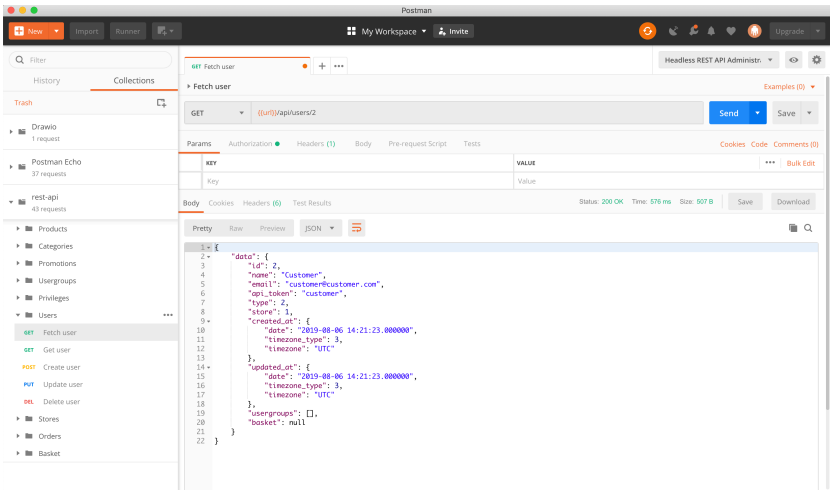


Ilustração 20 - Exemplo da *interface* do Postman

Como se pode verificar na Ilustração 20, posteriormente ao pedido ser efetuado, a resposta irá ser disponibilizada e na qual podemos verificar algumas estatísticas como o *status* da resposta, o tempo demorado e o tamanho desta.

7.3.5. Git

Git é um sistema de controlo de versões que é utilizado para facilitar o processo de verificação de modificações efetuadas a um ou mais ficheiros de um projeto. Este tipo de verificação inclui informação sobre quais as alterações que foram feitas, quem as efetuou e a razão associada às mesmas. Este tipo de sistema é útil em projetos de qualquer dimensão que exija algum tipo de coordenação entre as pessoas que trabalhem no mesmo [49], através de um repositório, que é uma espécie de plataforma que guarda várias versões do nosso projeto sempre que existem modificações, com possibilidade de utilizar novamente estas versões.

Apesar do projeto ter sido desenvolvido por apenas uma pessoa, este sistema foi utilizado para guardar as várias versões do projeto, assim como o desenvolvimento de várias funcionalidades em paralelo.

7.3.5.1. Sourcetree

De modo a facilitar a utilização dos comandos Git, existem várias aplicações que fornecem uma *interface* visual que executa estes comandos de maneira transparente ao utilizador final, tornando a sua utilização muito mais intuitiva. A aplicação utilizada para este propósito foi a “Sourcetree”, desenvolvida pela empresa “Atlassian”.

7.4. Funcionalidades implementadas

Nesta secção, serão descritas as várias funcionalidades que foram implementadas ao longo do projeto. Muitas destas foram desenvolvidas com o intuito de manter ou melhorar funcionalidades existentes no projeto base.

7.4.1. Carrinho de compras

Através da palavra "loja", é possível extrair automaticamente duas entidades que fazem parte desta, cada uma com um comportamento específico. Estas duas entidades são:

- **O vendedor** – O comportamento esperado associado a esta entidade será a possibilidade da venda de algo;
- **O cliente/comprador** – O comportamento esperado associado a esta entidade será a possibilidade de efetuar a compra de algo disponibilizado pelo vendedor.

Como este projeto se baseia numa loja *online*, foi dado como prioridade a possibilidade de um vendedor adicionar produtos que serão disponibilizados ao cliente e do cliente adicionar esses produtos a uma encomenda. Para isto foi adicionada a funcionalidade de carrinho de compras que guarda a informação de produtos e promoções adicionados, de modo a serem utilizados, para finalização da sua encomenda.

A funcionalidade do carrinho de compras foi desenvolvida de modo dar suporte a utilizadores autenticados, através da associação do carrinho de compras ao identificador do cliente, e utilizadores não autenticados (convidados) através da associação do carrinho de compras à sessão definida ao utilizador através de *cookies*. Em ambos os casos, toda a informação é guardada em base de dados.

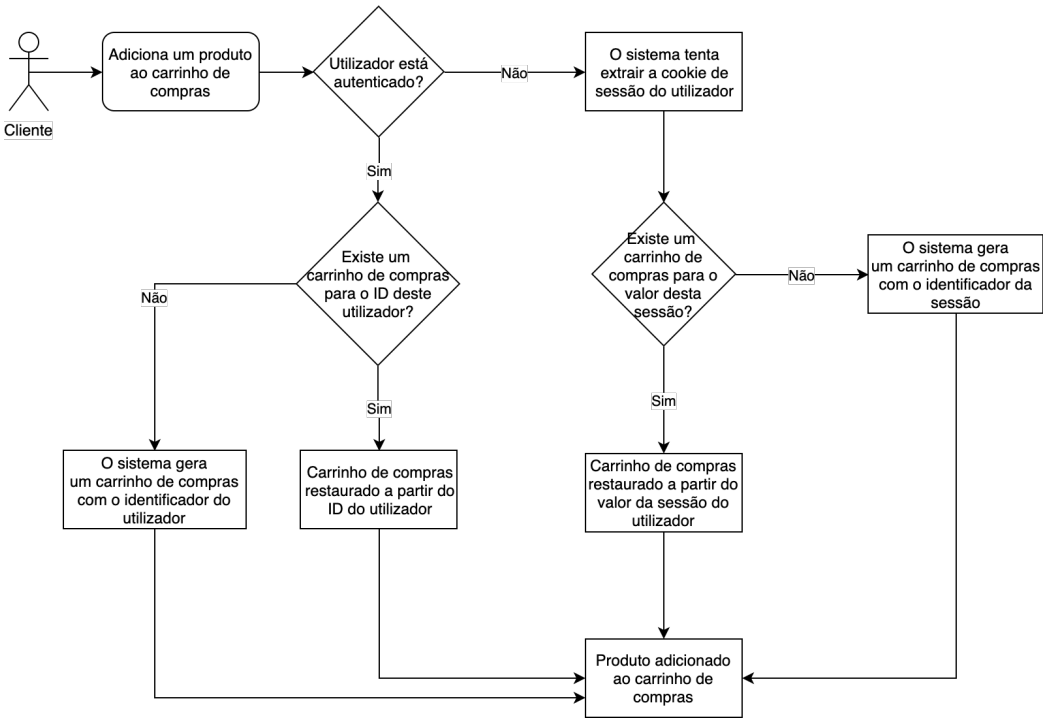


Ilustração 21 - *Flow* associado à adição de um produto ao carrinho de compras por parte de um utilizador

Como podemos verificar a partir da Ilustração 21, o *flow* associado ao carrinho de compras é um pouco complexo, com várias ramificações. A possibilidade de adicionar um produto ao carrinho sem que o utilizador necessite de autenticação aumenta a profundidade e complexidade deste sistema, que permite a criação de carrinhos "anónimos". Isto permite que o *frontend*, ou qualquer sistema que consuma a *API*, adicione por exemplo *checkout* anónimo, pois toda a informação necessária está disponível no carrinho de compras associado à sessão ou *cookie* do cliente.

Tal como dito previamente, para além dos produtos que são guardados na informação do carrinho, as promoções aplicadas são também guardadas e acabam por ser aplicadas automaticamente durante o restauro do carrinho de compras.

7.4.2. Implementação de *multi-store*

Um dos principais objetivos durante a implementação deste projeto foi, tal como referido anteriormente, tentar manter ou melhorar algumas das *core features* já existentes na plataforma base. Uma destas *features* principais chama-se "multi-store" e é responsável por permitir que a mesma plataforma contenha várias lojas diferentes, com recursos diferentes ou até mesmo partilhados. Como exemplo, através da mesma plataforma, podemos ter duas lojas *online* diferentes, para clientes completamente distintos. A diferenciação entre estas é

feita através do *URL* de acesso, que irá definir internamente o identificador da loja a qual o cliente está a aceder e para qual a lógica será processada. Este conceito não foi descrito previamente, onde foram mencionadas as *features* de um *CMS*, pois nem todos dão suporte a tal.

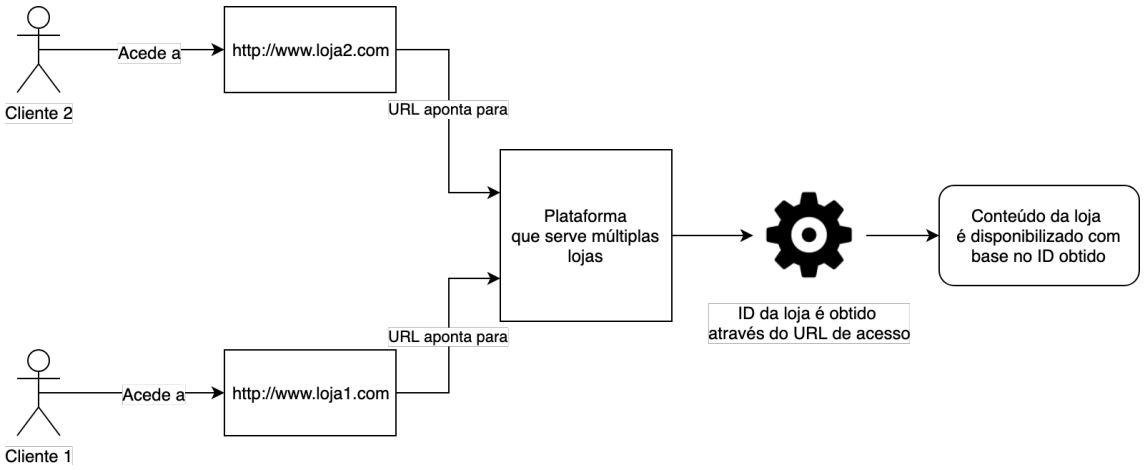


Ilustração 22 - Exemplo do funcionamento de uma plataforma *multi-store*

De modo a percebermos a implementação deste conceito, podemos verificar a Ilustração 22 e o nosso modelo de dados, o facto de alguns destes modelos conterem uma coluna chamada "store_id". Esta coluna irá definir a loja específica para qual este modelo ou recurso aponta. Este tipo de especificidade pode incluir produtos, categorias, promoções, entre outras.

Para além da especificidade de recursos, podemos também ter partilha dos mesmos entre todas as lojas. Este tipo de situação acontece quando a coluna "store_id" de um determinado modelo é definida pelo valor "0". A loja cujo identificador é "0" está reservado e é atribuído caso exista partilha deste modelo entre todas as lojas. Este tipo de situação pode ser explicado através do seguinte exemplo – Imaginando que temos quatro lojas na mesma plataforma, em que uma das categorias é igual entre todas estas lojas, esta categoria pode ser criada apenas uma vez caso seja partilhada entre todas as lojas, o que irá poupar tempo de criação e configuração. Por sua vez, está implícito que a sua remoção irá afetar todas as lojas.

7.4.3. Área de cliente e área administrativa

Tal como existia previamente no projeto base, foram criadas duas áreas distintas com propósitos diferentes, a área administrativa e a área de cliente. De modo a explicar esta implementação, é necessário ter conhecimento dos tipos de utilizador existentes.

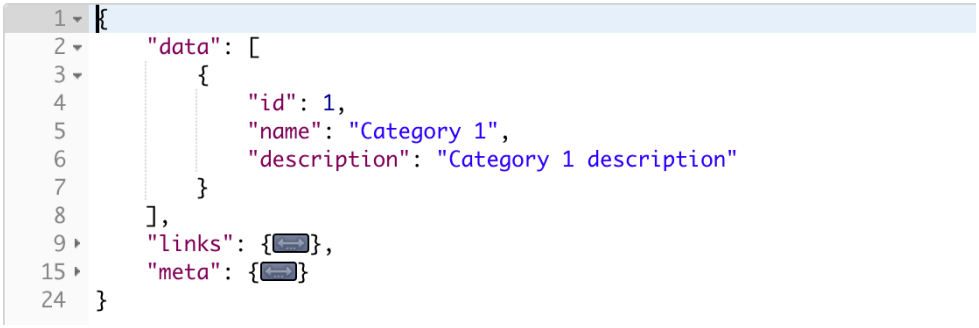
Tabela 4 - Tipos de utilizadores existentes e detalhes de acesso ás diferentes áreas.

Tipo	Nome	Detalhe
0	Administrador total	Tipo de utilizador com acesso a toda a aplicação ignorando privilégios e qualquer tipo de restrições. É comum haver apenas um administrador total na aplicação.
1	Administrador	Tipo de utilizador com acesso à área administrativa (com restrição através das suas permissões associadas) e área de cliente.
2	Cliente	Tipo de utilizador com acesso unicamente à área de cliente. Clientes que acedam à aplicação sem efetuar *login* (convidados), são inseridos neste tipo.

A área de administrativa é restrita a utilizadores do tipo "1" (Administrador) ou tipo "0" (Administrador total) e contém os pedidos necessários à administração geral da aplicação, incluindo a adição de produtos, categorias, lojas, entre outros. A área de cliente contém todo o tipo de acesso a recursos que são utilizados pelo cliente sem qualquer tipo de restrição, exemplo: acesso aos produtos no carrinho de compras, acesso ás categorias, entre outros.

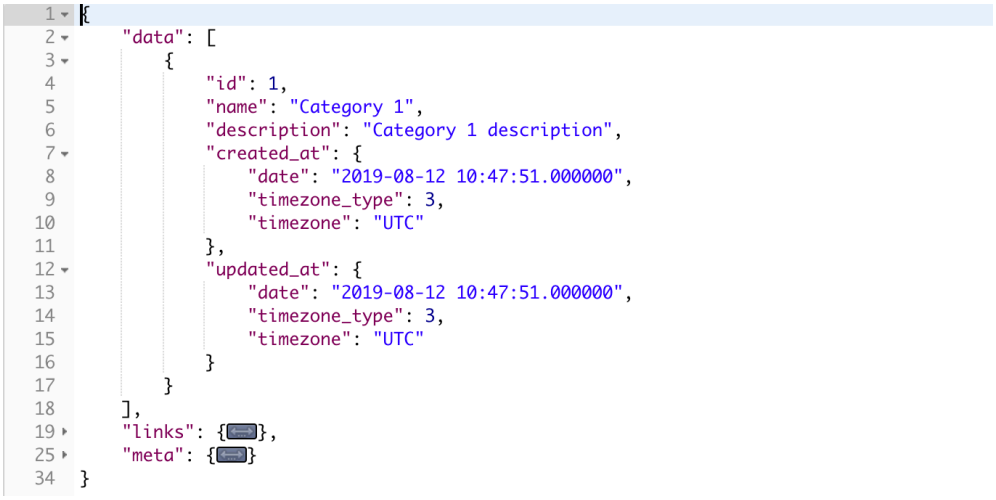
A diferenciação entre as duas áreas é feita através do *URI* de acesso à *API*. Todos os pedidos para à área administrativa deverão ser efetuados utilizando o prefixo "api/admin" no *URI*. Exemplo do *URI* de um pedido na área administrativa para aceder a todos os produtos: "api/admin/products". Todos os pedidos para a área de cliente são efetuados apenas com o prefixo "api". Exemplo do *URI* de um pedido na área de cliente para aceder a todas as categorias: "api/categories".

Existem vários *URIs* que são comuns entre as diferentes áreas (apesar de serem acedidos com prefixos diferentes), ou seja, apontam para o mesmo controlador que trata da sua lógica. No entanto, a resposta é enviada com algumas diferenças, como podemos verificar no caso das categorias ("api/categories" e "api/admin/categories"):



```
{
    "data": [
        {
            "id": 1,
            "name": "Category 1",
            "description": "Category 1 description"
        }
    ],
    "links": {},
    "meta": {}
}
```


Ilustração 23 - Exemplo de uma resposta ao pedido para adquirir as categorias a partir da área de cliente



```
{
    "data": [
        {
            "id": 1,
            "name": "Category 1",
            "description": "Category 1 description",
            "created_at": {
                "date": "2019-08-12 10:47:51.000000",
                "timezone_type": 3,
                "timezone": "UTC"
            },
            "updated_at": {
                "date": "2019-08-12 10:47:51.000000",
                "timezone_type": 3,
                "timezone": "UTC"
            }
        }
    ],
    "links": {},
    "meta": {}
}
```


Ilustração 24 - Exemplo de uma resposta ao pedido para adquirir as categorias a partir da área administrativa

Se compararmos a Ilustração 23 com a Ilustração 24, podemos verificar algumas diferenças entre as respostas, apesar destas serem devolvidas pelo mesmo controlador. É habitual verificarem-se controladores comuns entre as diferentes áreas onde a área administrativa irá geralmente devolver informação mais detalhada sobre o recurso a ser acedido.

7.4.4. Proteção de recursos através de *middlewares*

Um *middleware* é um mecanismo de filtragem de pedidos *HTTP* ao entrar na nossa aplicação. Estes geralmente contêm lógica que é usada para fazer verificações antes de executar algum extrato de código. São muitas das vezes injetados em controladores ou até nas próprias *routes* do Laravel. Para prevenir a repetição de código, podemos simplesmente criar o *middleware*, registá-lo na nossa aplicação, de modo a que este possa ser utilizado em várias partes de uma maneira simples e intuitiva.

Estes *middlewares* foram considerados como barreiras neste projeto pois estes são executados antes de uma determinada ação, onde podemos incluir verificações de determinadas especificações de modo a tomar uma decisão (exemplo: redirecionar o utilizador por falta de permissões).

7.4.4.1. Middleware "API Barrier"

De modo a proteger alguns dos pedidos, foi implementada uma "barreira" que permite verificar a autenticidade dos mesmos. Esta verificação é efetuada através de um *handshake* utilizando um *API token* (criado para ser utilizado especificamente para este propósito). Por sua vez, o servidor verifica se o utilizador existe e é válido com base na informação de autenticação, de modo a autenticá-lo apenas para efetuar este pedido dependendo da área de acesso pretendida. Este é considerado um *middleware* de "alto nível", pois é executado previamente à inicialização de qualquer lógica por parte do controlador. Através da arquitetura da aplicação representada pela Ilustração 18, este é representado através de um retângulo vermelho, e tal como podemos verificar, este é executado previamente a qualquer tipo de lógica da própria aplicação.

Este tipo de barreira traz as seguintes vantagens ao ecossistema:

- Permite uma maior segurança pois os pedidos que não contenham uma informação de autenticação válida são imediatamente rejeitados;
- Através da autenticação dos utilizadores, consegue-se extrair informação de tentativas falhadas de autenticação;
- Permite que após a autenticação, os pedidos sejam filtrados com possível rejeição, com base na ação que se pretende efetuar num determinado recurso e as características do utilizador autenticado.

Com base na informação descrita previamente, conseguimos verificar através da Ilustração 25 o seu funcionamento:

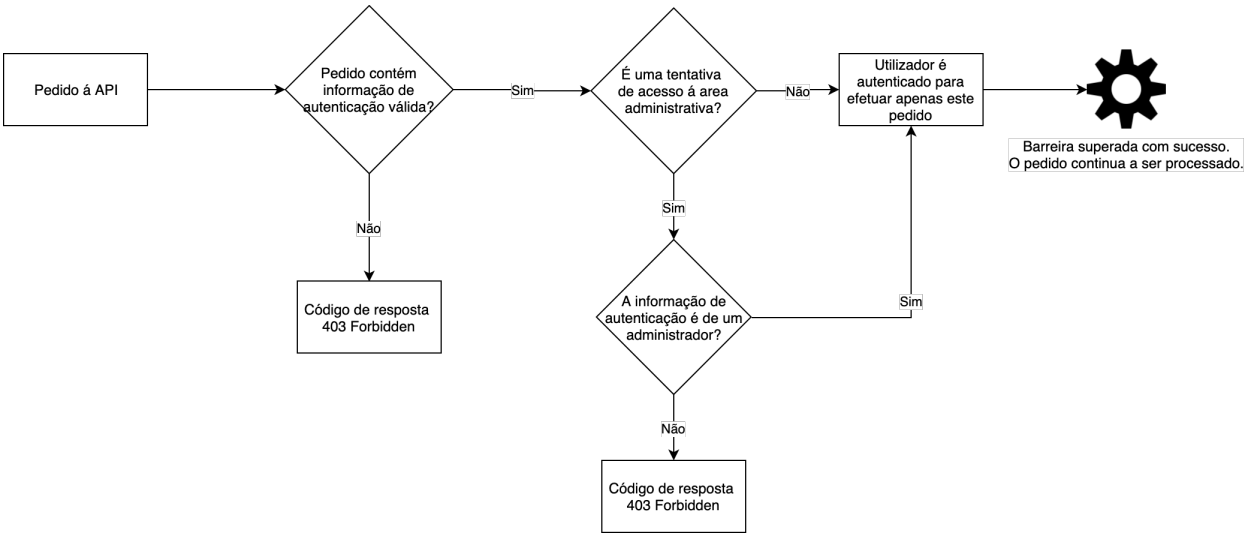


Ilustração 25 – Diagrama demonstrativo do funcionamento do *middleware* "*API barrier*"

De modo a injetar este *middleware*, podemos pegar no exemplo da classe CategoryController, onde queremos que todos os pedidos que sejam efetuados a qualquer dos seus métodos, obriguem o pedido a passar pela verificação de autenticidade.

```php
$router->group(['prefix' => 'api'], function() use ($router) {
    /*
    |--------------------------------------------------------------------------
    | ADMIN Area
    |--------------------------------------------------------------------------
    */
    $router->group(['prefix' => 'admin', 'middleware' => \App\Http\Middleware\ApiBarrier::class], function() use ($router) {

        /**
         * Products controller
         */
        $router->get('/products', 'ProductController@index');
        $router->get('/products/{id}', 'ProductController@show');
        $router->post('/products', 'ProductController@store');
        $router->put('/products/{id}', 'ProductController@update');
        $router->patch('/products/{id}', 'ProductController@update');
        $router->delete('/products/{id}', 'ProductController@destroy');
```

Ilustração 26 - Exemplo da injeção do *middleware* "API barrier" a todas as *routes* de "/api/admin" (Área administrativa)

Como podemos verificar através da Ilustração 26, a injeção é feita diretamente nas *routes*, através do parâmetro "middleware" durante a definição do grupo de *routes*. Neste caso em específico podemos verificar que estas estão a "proteger" as *routes* associadas à área administrativa, pois esta requere obrigatoriamente dados de autenticação corretos.

7.4.4.2. Middleware "Privilege barrier"

Este *middleware* foi implementado para servir como "barreira" de proteção associado a privilégios. O objetivo principal desta classe é verificar se um utilizador associado à área de administração tem permissões de acesso a um determinado método ou de efetuar uma

ação específica sobre um recurso. Trabalha a um nível mais baixo que a *API barrier*, pois esta não afeta um *scope* tão grande do fluxo aplicacional, representada como um retângulo verde dentro do controlador na Ilustração 18. É de notar que esta classe foi desenvolvida de maneira a ser utilizada de uma maneira bastante dinâmica, onde podemos referir um método especifico que será afetado. Isto é bastante útil para controladores que afetam várias ações sobre os recursos.

Caso a passagem por esta barreira não seja bem-sucedida, o servidor irá devolver um código de resposta 403, juntamente com a mensagem de erro "*You do not have permission to access this method or perform this action on this resource*".

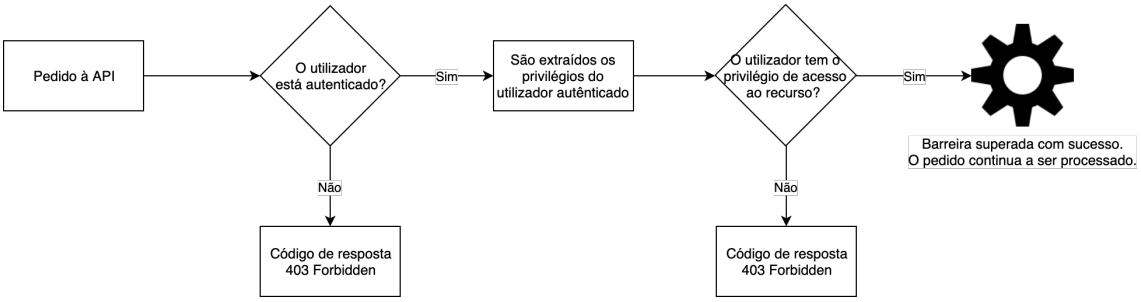


Ilustração 27 – Diagrama demonstrativo de como funciona o *middleware* privilege barrier

De modo a injetar este *middleware*, podemos pegar no exemplo da classe UserController, onde queremos que todos os pedidos que sejam efetuados a qualquer dos seus métodos, obriguem o pedido a passar pela verificação se o utilizador que está a efetuar o pedido tem os privilégios necessários para executar um determinado método.

```php
class UserController extends Controller
{
    public function __construct()
    {
        if (Area::A()) {
            $this->middleware( middleware: PrivilegeBarrier::class . ':view_users', ['only' => ['index', 'show']]);
            $this->middleware( middleware: PrivilegeBarrier::class . ':create_users', ['only' => ['store']]);
            $this->middleware( middleware: PrivilegeBarrier::class . ':update_users', ['only' => ['update']]);
            $this->middleware( middleware: PrivilegeBarrier::class . ':delete_users', ['only' => ['destroy']]);
        }
    }

    public function index(){ ... }

    public function show($id){ ... }

    public function update(Request $request, $id){ ... }

    public function store(Request $request){ ... }

    public function destroy($id){ ... }
}
```

Ilustração 28 - Exemplo da injeção do *middleware privilege barrier* no controlador UserController para a Área administrativa

Para a sua utilização, é apenas necessário que este seja injetado no controlador especifico, em conjunto com o nome do privilégio e o método que será afetado. Através da Ilustração 28 podemos verificar a maneira de utilização deste *middleware*, que é injetado diretamente no construtor do controlador, com o objetivo de executar a sua lógica previamente ao método específico do próprio controlador. Para garantir que cada um destes métodos está protegido devidamente pelo seu privilégio responsável, podemos verificar que, por exemplo, o utilizador responsável pela chamada do método "*update*", necessita do privilégio "update_users", no caso da área administrativa, onde este é injetado para servir apenas este "*update*", definido através um dos parâmetros associados à função "middleware".

Isto pode verificar-se através da Ilustração 28, que demonstra a injeção da barreira de privilégios no controlador associado aos utilizadores da área administrativa.

7.4.5. Advanced logging

Um dos aspetos mais importantes das lojas *online*, é a possibilidade de se conseguir perceber como ocorreu uma determinada ação. De modo a que isto aconteça, existe informação que é guardada durante o fluxo aplicacional, muitas das vezes na base de dados, ou em ficheiros de texto.

De modo a conseguirmos obter algum tipo de filtragem após serem criados, foram implementados diferentes tipos de severidade que podem ser posteriormente acedidos.

Tabela 5 - Tipos de severidades existentes para *logs*

Tipo de severidade	Detalhes
INFO	Atribuído quando se pretende armazenar algum tipo de informação, onde não existe qualquer tipo de severidade.
WARNING	Atribuído quando algum tipo de situação inoportuna ocorreu, no entanto, o fluxo de código continuou. Usado geralmente para chamar a atenção quanto à situação ocorrida. Exemplo: Não foi possível encontrar a rua utilizada pelo cliente ao efetuar uma encomenda durante o seu processamento. Prioridade baixa de resolução.
ERROR	Atribuído quando ocorreu um erro na aplicação em que o fluxo de código foi abruptamente terminado devido a um erro do sistema. Exemplo: ocorreu um erro ao adicionar uma encomenda na base de dados. Prioridade média/alta de resolução.
CRITICAL	Tipo reservado a situações criticas que metam em risco a infraestrutura ou o próprio negócio. Exemplo: os clientes não conseguem efetuar compras na loja devido a um erro do próprio sistema. Prioridade máxima de resolução.

A partir destes *logs*, pode-se extrair informação valiosa que serve para descobrir um problema que ocorreu no sistema ou até mesmo para auditoria. Existe, no entanto, um problema que surge com a implementação deste sistema. A quantidade de informação guardada ao longo do tempo acaba por causar alguma latência nos acessos à base de dados devido à quantidade de informação.

De modo a prevenir este tipo de situação e automatizar a limpeza do sistema, foi implementada uma classe responsável pela limpeza e extração desta informação de forma a minimizar a quantidade de informação na tabela de *logs*. O objetivo da classe "ClearLogsCommand" encontrada em "app/Console/Commands" é compactar a informação

num ficheiro de texto e eliminar toda a informação da tabela. Esta informação é guardada através de um ficheiro ".tar.gz" guardado em "storage/logs". O nome deste ficheiro é o resultado da concatenação da data do log mais antigo, seguido de um hífen e a data do *log* mais recente. Como exemplo, caso este tipo de limpeza ocorresse em *logs* que tivessem um intervalo de tempo de "1 de Maio de 2019" e "10 de Maio de 2019", o resultado do nome do ficheiro seria "2019_05_01-2019_05_10.tar.gz".

De modo a automatizar todo este processo, foram implementadas duas maneiras de possível execução – limpeza automática e limpeza manual.

7.4.5.1. Tarefa agendada de limpeza automática

A limpeza automática de *logs* foi definida de modo ser executada automaticamente através de uma *scheduled task*. Esta corre no primeiro dia de cada mês, de modo a que a informação fique compactada num intervalo de tempo consistente.

7.4.5.2. Limpeza manual de logs

Em caso de necessidade, podemos também efetuar a limpeza manual dos *logs* através da utilização do comando "php artisan logs:clear", como podemos verificar na Ilustração 29.

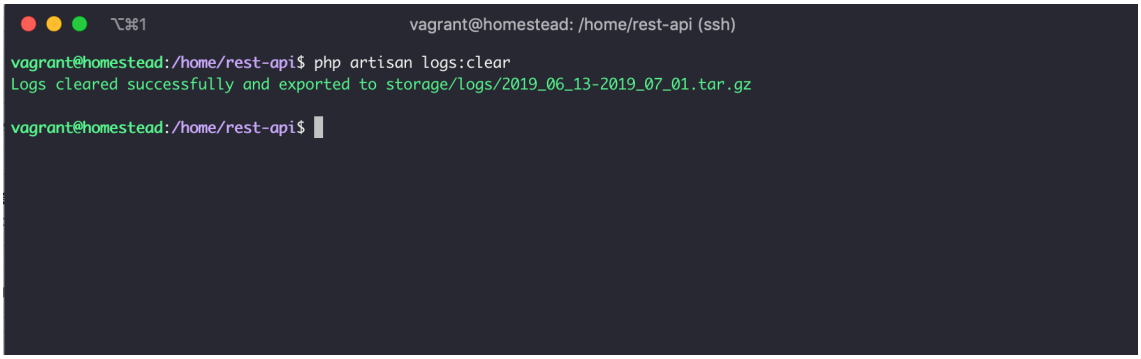


Ilustração 29 - Exemplo da execução do comando "php artisan logs:clear"

Este comando deve ser usado apenas em casos especiais, para prevenir uma grande quantidade de ficheiros com nomes semelhantes de serem criados e para prevenir que exista uma discrepância de dados entre os vários ficheiros comprimidos.

7.4.6. Sistema de promoções

Qualquer plataforma de comércio digital tem várias formas de cativar clientes. Uma dessas é o uso de promoções, de modo a que os clientes consigam obter um desconto ao efetuar uma ou mais compras.

Foi implementado um sistema de promoções na aplicação, com base no projeto anterior. Este sistema inclui dois tipos diferentes distintos de promoções que poderão ser configurados:

- **Catálogo** – Promoção aplicada a nível de produtos no catálogo, ou seja, se o cliente adicionar esse produto ao carrinho, a promoção é imediatamente aplicada;
- **Carrinho de compras** – Promoção aplicada através de um código que aplica um desconto a um determinado produto, estes códigos são gerados separadamente da promoção e são "queimados", prevenindo que estes sejam novamente utilizados.

Durante a adição de uma promoção, é necessário configurar um valor que será descontado, e o tipo de desconto associado. Existem dois tipos de desconto disponíveis:

- **Valor fixo** – O valor do desconto configurado é diretamente subtraído ao preço do produto;
- **Valor percentual** – O valor do desconto é calculado através da subtração da percentagem configurada na promoção ao valor atual do produto.

7.4.7. Soft delete

Através da experiência pessoal obtida no projeto base, existe por vezes uma dificuldade em efetuar o *debug* de certos cenários (em que os *logs* não disponibilizam informação suficiente) que envolvem modelos inexistentes, removidos posteriormente. Com a finalidade de prevenir este tipo de situações, foi adicionada a implementação de *soft delete*, funcionalidade nativa do Laravel, em alguns dos modelos do projeto.

Esta funcionalidade permite que um modelo não seja removido completamente da base de dados, com a possibilidade da fácil restauração do mesmo. O conceito técnico desta *feature* é bastante simples – é adicionada de uma coluna denominada de "deleted_at" em determinados modelos preenchível com um *timestamp*, que indica que o modelo foi removido na data definida pelo valor preenchido nesta coluna. A sua restauração ocorre de maneira simples, através da remoção do valor da coluna "deleted_at" [50]. Algumas das vantagens e desvantagens da utilização de *soft delete* são:

- Vantagens:
    - Facilidade em restaurar informação com a simples remoção de um valor de uma coluna. Os dados não são completamente removidos da base de dados;

  - Um *update* é uma operação mais rápida de se efetuar que a operação *delete*.
- Desvantagens:
  - Não remover definitivamente informação pode tornar as operações associadas à base de dados mais lenta;
  - A utilização de remoção de dados relativos ao modelo através de chaves secundárias através de *cascading* passa a não ser possível. Em casos extremos existe a necessidade de fazer este tipo de operação através de um *trigger* no *update* da coluna "deleted_at". Isto poderá tornar o ecossistema mais complexo.

Existe também a necessidade de ter conhecimento que existem certas politicas de regulação associadas à informação que é guardada sobre um utilizador, como por exemplo o *GDPR* (*General Data Protection Regulation*). Neste tipo de situações, a informação associada ao utilizador (aquando a sua eliminação) deverá ser anonimizada, pois a utilização do *soft delete* vai permitir que estes dados permaneçam na base de dados. A plataforma desenvolvida não oferece suporte a este tipo de anonimização de informação.

7.5. Funcionalidades "genéricas" em falta

Foi acordado com o cliente que algumas funcionalidades consideradas necessárias a qualquer loja *online* (exemplo: métodos de pagamento, sistema de logística ou cálculo de taxas) seriam posteriormente integradas por parte do próprio cliente após a entrega do projeto. Isto deve-se ao facto de o projeto base ter integrações *third party* responsáveis por tratar deste tipo de componentes. O sistema de pagamentos associado ao projeto base é fornecido por através de um *payment gateway* que permite que os clientes da loja paguem as suas encomendas através de um sistema responsável por disponibilizar vários métodos de pagamento dependendo do país associado ao cliente, chamado "Paynext". O sistema de logística é fornecido através de um sistema denominado "Arvato", que fornece informação associados à logística e taxas associadas aos produtos dependendo do peso, dimensões e sitio de entrega associada à encomenda.

8. Testes

Neste capítulo irão ser apresentadas as várias etapas que foram executadas de modo a testar as funcionalidades tendo em conta dois aspetos importantes associados a uma *API*, funcionalidade e velocidade de resposta. A funcionalidade foi garantida através da implementação de testes unitários, enquanto a parte de velocidade de resposta foi testada através de testes de carga, executados a partir da ferramenta Postman.

8.1. Testes unitários

Os testes unitários do projeto são executados através de um *package* denominado de "PHPUnit", que se trata de uma *framework* específica para execução de testes orientados à programação. A sua execução devolve o *output*/resultado de todos os testes efetuados. O *output* demonstra o sucesso ou falha associada a cada uma das classes do projeto (cada uma inclui vários testes) desenvolvidas para este propósito.

Ilustração 30 - Exemplo de um *output* associado ao resultado dos testes

Como podemos verificar a partir da Ilustração 30, existiram trinta e oito testes (representado por pontos na imagem) executados, com noventa e quatro afirmações verificadas na sua totalidade. Este é um caso exemplo de que todo este processo foi bem-sucedido, no entanto, existem 6 possíveis resultados (cada um destes com o seu diferente símbolo) associados à execução de um teste [51].

Tabela 6 - Tipos de resultados possíveis após a execução de um teste

Símbolo	Descrição
.	Teste sucedido.

F	Teste falhado, houve pelo menos uma das verificações que não foi bem-sucedida.
E	Ocorreu um erro ao executar o teste, o fluxo de código parou abruptamente.
R	Teste marcado manualmente como arriscado. Este tipo de *output* ocorre quando não existem verificações num teste e se pretende que o PHPUnit insira uma "etiqueta" em testes considerados "inúteis" (caso não existam verificações).
S	Teste marcado manualmente para ser ignorado. A sua lógica não foi executada.
I	Teste marcado manualmente como incompleto ou não implementado. Geralmente utilizado quando um teste não foi terminado.

Apesar de todos estes tipos de *output* apresentados na Tabela 6, será necessário apenas focar os símbolos ".", "F" e "E" pois nenhum dos testes foi "marcado" para devolver qualquer outro tipo de *output* durante o desenvolvimento destes.

8.1.1. Implementação e detalhes

A implementação da maior parte dos testes unitários foi feita previamente ao próprio código ser desenvolvido, de modo a garantir que os desenvolvimentos seguiam uma direção correta. Após a implementação de qualquer tipo de funcionalidade, os testes unitários foram executados novamente de modo a verificar a possível entrada de *bugs* com o novo código.

De modo a garantir o bom funcionamento de cada modelo, foram feitos os testes "básicos" associados ao *CRUD* (*Create Read Update Delete*), de muitos destes, assim como funcionalidades específicas de cada um.

8.1.1.1. Classes de testes

Os testes unitários foram adicionados através de classes. Cada classe está associada a um componente ou funcionalidade específica da aplicação e poderá conter múltiplos testes, definidos através de funções que envolvem várias asserções. As asserções são utilizadas para determinar se o resultado de uma expressão ou condição é o esperado [52]. No contexto deste projeto, as asserções são efetuadas através de variantes da função "assert", como por exemplo "assertEquals", que recebe como parâmetros o valor esperado e o valor atual utilizados para efetuar uma comparação rigorosa [53]. Caso a asserção falhe, é apresentada uma mensagem que identifica a falha.

Todas as classes de testes adicionadas ao projeto partilham as seguintes características:

- Utilizam o *Trait* "DatabaseTransaction", funcionalidade existente no Laravel que permite que a sua utilização efetue o *rollback* de alterações à base de dados após a execução de cada teste. Isto indica que todas as alterações efetuadas à base de dados durante a execução de cada teste, são descartadas;
- Utilizam a função "setUp", responsável por preparar o ambiente de testes antes da sua execução de qualquer teste. É normalmente utilizado para adicionar os recursos necessários à base de dados;

```php
class ProductsTest extends \TestCase
{
    use DatabaseTransactions;
    private $root_admin = null;

    protected function setUp():void
    {
        parent::setUp();
        factory(Store::class, 1)->create();

        $user = factory(User::class, 1)->create(['type' => User::TYPE_ROOT_ADMIN, 'api_token' => 'root']);
        $this->root_admin = $user[0];

        factory(Product::class, 4)->create(['status' => 1]);
        factory(Product::class, 2)->create(['status' => 0]);
    }

    public function testDisabledProduct()
    {
        $disabled_product = Product::where(['status' => 0])->first();

        $this->json( method: 'GET',  uri: 'api/products' . $disabled_product->id);
        $this->assertEquals( expected: Response::HTTP_NOT_FOUND, $this->response->status(),  message: "The disabled product was possibly found for clients");

        $this->json( method: 'GET',  uri: 'api/products' . $disabled_product->id, [
            'user'      => $this->root_admin->email,
            'api_token' => $this->root_admin->api_token,
        ]);

        $disabled_product_returned = json_decode($this->response->getContent())->data;

        $this->assertEquals( expected: Response::HTTP_OK, $this->response->status(),  message: "The disabled product was not found for the root administrator");
        $this->assertEquals($disabled_product_returned->status,  actual: Product::STATUS_DISABLED,  message: "The returned product is not disabled");
    }
```

Ilustração 31 - Exemplo do teste "testDisabledProduct" associado à classe "ProductsTest"

Através da Ilustração 31 conseguimos verificar que a classe "ProductsTest" contém pelo menos um teste, denominado "testDisabledProduct". O seu objetivo é aferir a funcionalidade de uma das propriedades associadas aos produtos, que determinam se estes estão ativos. Tal como referido na secção 7.1, produtos que não estejam ativos não deverão ser mostrados ao cliente, mas deverão estar visíveis para a área administrativa. O teste começa por encontrar um produto inativo através de *ORM* (adicionado previamente através da função "setUp"), de seguida é efetuado um pedido *GET* como cliente (sem autenticação) ao *endpoint* "api/products" juntamente com o identificador do produto inativo, aferindo a devolução de um código *HTTP* "404". Caso a aplicação não devolva "404", a asserção irá

falhar e o teste irá mostrar um erro com a indicação "The disabled product was possibly found for clients". Por sua vez, o mesmo teste irá ser efetuado para a área administrativa no sentido de garantir que o produto está disponível através do código *HTTP* "200".

As classes associadas a testes existentes no contexto deste projeto são as seguintes:

- **AccessControlTest** - Classe associada a testes de acesso à *API* por tipo de utilizador. Os testes baseiam-se em verificar o código de resposta do servidor através da tentativa de acesso a recursos ou áreas onde um determinado tipo de utilizador tem acesso permitido/restrito. Todos os testes associados a esta classe são efetuados sem que os utilizadores tenham qualquer tipo de privilégios;
- **PrivilegeControlTest** - Classe associada a testes de privilégios. Os testes baseiam-se no acesso a diferentes recursos (que necessitem de determinados privilégios para serem acedidos) através da utilização de utilizadores de diferentes tipos, em diferentes usergroups (com diferentes privilégios);
- **ProductsTest** - Classe associada a testes que envolvam produtos e a devolução destes por parte do servidor;
- **PromotionsTest** - Classe associada a testes que envolvem promoções. Estes baseiam-se maioritariamente em garantir que após a utilização de uma promoção, os valores descontados de uma encomenda são corretos. A criação e utilização de códigos associados a promoções é também verificada nesta classe de testes;
- **StoresTest** - Classe associada a testes que envolvem lojas. A verificação de recursos partilhados entre lojas é também verificada;
- **BasketsTest** - Classe associada a testes que envolvem carrinho de compras do cliente. Baseia-se maioritariamente em verificar se os produtos são adicionados corretamente ao carrinho, assim como garantir que estes são "restaurados" com os valores previamente inseridos;
- **CategoriesTest** - Classe associada a testes que envolvem categorias. Baseia-se na adição e remoção de produtos às categorias, de modo a garantir que estas disponibilizam corretamente os seus produtos associados;
- **OrdersTest** – Classe associada a testes que envolvem encomendas. Estes baseiam-se principalmente em criar múltiplas encomendas, com e sem promoções, com diferentes produtos, garantindo que o preço é calculado corretamente;

- **UsersTest** – Classe associada a testes que envolvem utilizadores de diferentes tipos, garantindo a população de campos necessários durante a sua criação (exemplo – apenas os administradores deverão ter um api_token associado para efetuarem a autenticação);
- **UsergroupsTest** – Classe associada a testes que envolvem usergroups. Estes baseiam-se em garantir que os seus privilégios são associados e devolvidos corretamente;
- **ReviewsTest** – Classe associada a testes que envolvem opiniões de clientes associados a um produto.

8.2. Testes de carga

Nesta secção irão ser descritos os vários passos e resultados obtidos através dos testes de carga efetuados. Estes testes foram possíveis através da utilização de uma das funcionalidades do Postman chamada "*Collection Runner*". Apesar de existirem ferramentas *open-source* desenvolvidas para este propósito (The Grinder, Gatling, JMeter ou Locust por exemplo [54]), foi decidido que não houve necessidade de "aprender" a utilizar qualquer uma destas pois existe apenas a necessidade de extrair, para este caso, as velocidades de resposta por parte da *API* (já que tenho experiência a trabalhar com o Postman). Esta funcionalidade básica permite escolher determinados pedidos a efetuar (previamente adicionados ao próprio Postman) num determinado intervalo de tempo, com a possibilidade de escolher o número de iterações de execução por cada pedido. Após completar todas as iterações, é disponibilizada uma analise associada dados relevantes (tempo do pedido, código de resposta, entre outros). Durante o desenvolvimento do projeto, houve a necessidade de executar vários pedidos de modo a efetuar os vários testes das funcionalidades dos modelos e controladores. De modo a facilitar este tipo de operações foi criada uma "biblioteca" de pedidos através da ferramenta Postman que disponibiliza todos os *endpoints* e métodos *HTTP* associados a cada controlador disponibilizado. Este ficheiro, com informação em formato JSON, poderá ser consultado na raiz do projeto através do nome "postman-collection.json".

Tal como descrito na secção 6.6, todo este projeto acabou por ser migrado para a *micro-framework* Lumen devido ás suas capacidades relativamente à melhoria de desempenho. Como temos a disponibilidade de usar o projeto pré e pós migração para Lumen, foram definidos os testes de carga para serem executados em ambas as aplicações

para comparar a melhoria associada à migração, de modo a possivelmente apoiar a decisão de mudança para Lumen através da verificação de melhorias de desempenho.

8.2.1. Condições para os testes de carga

De modo a garantir que estes testes permitam obter resultados concisos, estes foram executados em ambientes e condições similares. Durante a realização dos testes, ambos os projetos estavam disponibilizados na mesma máquina virtual, com recursos alocados de maneira equivalente e todos os pedidos aos diferentes *URIs* foram efetuados com iguais condições (conteúdo do *body* associado ao pedido e autenticação quando necessária), utilizando a mesma ferramenta.

8.2.2. Execução e comparação

Através da execução dos testes na aplicação assente sobre Laravel nativo, foram efetuadas cem iterações por vários *URIs*, os resultados obtidos foram os seguintes:

Tabela 7 - Resultados após a execução dos testes de carga usando a aplicação assente sobre Laravel nativo

HTTP VERB	URI	Tempo de resposta mais lento	Tempo de resposta mais rápido	Tempo médio de resposta
GET	api/products/1	535	278	370.31
GET	api/products	655	311	414.46
POST	api/admin/products	895	283	386.26
PUT	/api/admin/products/1	879	263	366.96

Depois da execução destes testes, foram efetuados os mesmos utilizando as condições previamente descritas na aplicação sobre a *micro-framework* Lumen, obtendo os seguintes resultados:

Tabela 8 - Resultados após a execução dos testes de carga usando a aplicação assente sobre Lumen

HTTP VERB	URI	Tempo de resposta mais lento	Tempo de resposta mais rápido	Tempo médio de resposta
GET	api/products/1	248	137	201.92
GET	api/products	363	192	300.86

POST	api/admin/products	375	162	243.12
PUT	/api/admin/products/1	427	159	234.42

Como podemos verificar através da comparação da Tabela 7 com a Tabela 8, os testes associados ao Lumen tiveram uma melhoria média de 37.3% (baseado apenas nos testes efetuados) no desempenho, que pode ser explicada devido ao facto de o Lumen ser muito mais "leve" em relação ao Laravel nativo. Através da análise da percentagem associada à melhoria de tempo de resposta, podemos verificar que a migração para Lumen acabou por ser uma decisão acertada, no entanto, há que ter em mente que estes valores serão completamente diferentes num caso real de uma aplicação em produção, onde existem bases de dados com grandes quantidades de informação que acabam por afetar o desempenho geral.

9. Conclusão

O objetivo principal da entidade responsável por este projeto seria o desenvolvimento de uma Headless *REST API* através da utilização da *framework* Laravel de modo a encaixar num cenário de *e-commerce*, que servisse como candidato à substituição da solução atual de uma das plataformas em utilização, como descrito na secção 2.4. Este objetivo foi superado com sucesso, pois o resultado final obtido foi uma Headless *API* completamente funcional, utilizando a arquitetura *REST* com a finalidade de servir uma loja *online*, com o "extra" do desenvolvimento mais genérico, que consiga ser utilizado também como base de qualquer loja *online*.

O facto de não existirem condições predefinidas associadas às funcionalidades necessárias que este trabalho devia abordar (as únicas condições impostas foi o tópico de *e-commerce*), todas estas foram adicionadas através da experiência profissional obtida no projeto em que este se baseia, que permitiu definir prioridades e diferentes objetivos. O objetivo principal de implementar um sistema de carrinho de compras foi finalizado com sucesso. Todas as outras funcionalidades foram implementadas de modo a complementar este comportamento através da adição de "utilidades" que podem ser encontradas em muitas das lojas *online* - promoções, áreas de administração, auditoria, entre outros.

Os dois tipos de testes implementados (testes unitários e testes de carga) serviram para garantir o bom funcionamento geral da aplicação. Os testes unitários desenvolvidos previamente às várias funcionalidades implementadas tiveram um papel importante na descoberta de vários *bugs* que foram aparecendo após cada iteração de entrega. Isto prova a relevância da implementação destes previamente à adição de novas funcionalidades em qualquer ambiente de desenvolvimento. Por fim, os testes de carga concluíram que a migração para uma versão da *framework* "mais leve" trouxe vantagens na *performance* da aplicação, que faz parte de uma das grandes necessidades de qualquer *API*.

A realização deste trabalho permitiu o aumento de competências associadas a uma área interessante, com inúmeros paradigmas, que continua a mostrar-se relevante. O aprofundamento da linguagem de programação *PHP* e da arquitetura de uma loja *online* são definitivamente conhecimentos que se irão mostrar relevantes em ambiente profissional.

9.1. Utilização do conceito *Headless*

Através da implementação deste projeto e na minha experiência pessoal e profissional no desenvolvimento *web* foi possível concluir que o conceito *Headless* fornece uma maior flexibilidade rapidez no seu desenvolvimento a qualquer serviço. A flexibilidade deve-se ao facto de ser fácil modificar qualquer parte da aplicação sem que estas modificações afetem diretamente componentes ou serviços que estejam a consumir a nossa *API*. Apesar de ter sido a primeira vez que utilizo este conceito para efetuar uma implementação, consigo dizer que esta seja uma forma melhor de implementação em relação ao método tradicional de qualquer *CMS*.

9.2. Dificuldades encontradas

Durante várias etapas de desenvolvimento, houve por vezes um sentimento de insegurança em relação à possível integração com uma componente visual devido ao desenvolvimento efetuado "às cegas". A minha experiência profissional habituou-me a desenvolver aplicações com a "visão geral" destas, que me permite engenhar a solução indicada. O desenvolvimento de uma solução genérica associada a *e-commerce* utilizando um conceito *Headless* tirou-me da minha "zona de conforto", tornando este um projeto complexo e interessante.

9.3. Potenciais melhorias e funcionalidades futuras

Todas as funcionalidades associadas ao comércio digital foram escolhidas como prioridade em relação a outro tipo de situações. No entanto, existem certas funcionalidades consideradas importantes em qualquer aplicação *web* que acabaram por não ser implementadas devido ao tipo de prioridades previamente estabelecidas.

9.3.1. Integração com *GUI*

O projeto foi desenvolvido abstratamente em relação à *interface* visual. Isto indica que muitos dos potenciais problemas associados à integração entre a lógica e parte visual ainda não foram identificados. De modo a garantir que esta integração seja funcional, será necessário criar um protótipo de uma GUI que consome a nossa *API* com o objetivo de certificar que a mesma está adaptada às necessidades das aplicações.

9.3.2. Melhorias de desempenho

Como previamente referido na secção 6.6, existiu uma migração da implementação do projeto para Lumen, no entanto, houve a necessidade de se ativar alguns dos componentes inativos por omissão, que pode afetar um pouco o desempenho desta. Apesar de não ser uma situação de nível critico, há que ter noção que o propósito principal da criação do Lumen foi baseado em fornecer muitas das funcionalidades do Laravel priorizando o desempenho da aplicação, através da simplificação e desativação de alguns dos seus componentes. No contexto deste trabalho houve necessidade de ativar a abstração da base de dados, uma melhoria poderia ser efetuada através da "tradução" para *raw queries*, permitindo assim desativar a sua injeção no projeto, fornecendo um maior desempenho por parte da *API.*

10. **Bibliografia**

[1] K. Flaherty e A. Kaley, "The New Ecommerce User Experience: Changes in Users' Expectations," 10 06 2018. [Online]. Available: https://www.nngroup.com/articles/ecommerce-expectations/. [Acedido em 01 03 2019].

[2] "One of the best ecommerce providers - THIS IS OUR TEAM," [Online]. Available: https://www.cs-cart.com/about.html. [Acedido em 09 08 2019].

[3] "Ecommerce," [Online]. Available: https://www.shopify.com/encyclopedia/what-is-ecommerce. [Acedido em 2019 01 17].

[4] L. Centenaro, "Optimizely," 26 06 2015. [Online]. Available: https://blog.optimizely.com/2015/06/26/ecommerce-evolution-part-1/. [Acedido em 11 01 2019].

[5] "What is Ecommerce?," 12 04 2018. [Online]. Available: https://ecommerceguide.com/guides/what-is-ecommerce/. [Acedido em 13 01 2019].

[6] M. Silva, "CMS: Para que serve um Sistema de Gestão de Conteúdos?," 13 04 2017. [Online]. Available: http://ecossistemadigital.pt/sistema-de-gestao-de-conteudos/. [Acedido em 13 12 2018].

[7] L. Machuca, "What Is a CMS? All You Need to Know About Content Management Systems," 2017 07 24. [Online]. Available: https://fitsmallbusiness.com/what-is-a-content-management-system-cms/. [Acedido em 2018 12 17].

[8] C. Chapman, "Front End, Back End, Full Stack—What Does it All Mean?," 2018 11 05. [Online]. Available: https://skillcrush.com/2017/02/27/front-end-back-end-full-stack/. [Acedido em 16 01 2019].

[9] B. Powell, "All About CMS Plug-Ins," 26 11 2018. [Online]. Available: https://www.lifewire.com/what-is-a-cms-plugin-756561. [Acedido em 22 07 2019].

[10] M. Rouse, "content management system (CMS)," [Online]. Available: https://searchcontentmanagement.techtarget.com/definition/content-management-system-CMS. [Acedido em 2019 01 10].

[11] J. Lee, "No. 1 Position in Google Gets 33% of Search Traffic [Study]," [Online]. Available: https://searchenginewatch.com/sew/study/2276184/no-1-position-in-google-gets-33-of-search-traffic-study. [Acedido em 12 01 2019].

[12] Enginess, “ Guide Content management systems,” [Online]. Available: https://www.nibusinessinfo.co.uk/content/important-cms-features-and-functions. [Acedido em 10 03 2019].

[13] S. Mane, “Understanding REST (Representational State Transfer),” 08 06 2017. [Online]. Available: https://medium.com/@sagar.mane006/understanding-rest-representational-state-transfer-85256b9424aa. [Acedido em 2019 01 21].

[14] “What is REST?,” [Online]. Available: https://www.codecademy.com/articles/what-is-rest. [Acedido em 01 03 2019].

[15] “Caching your REST API,” [Online]. Available: http://restcookbook.com/Basics/caching/. [Acedido em 01 03 2019].

[16] “REST Architectural Constraints,” [Online]. Available: https://restfulapi.net/rest-architectural-constraints. [Acedido em 19 02 2019].

[17] G. Levin, “RESTful API Basic Guidelines,” 06 09 2016. [Online]. Available: https://blog.restcase.com/restful-api-basic-guidelines/.

[18] “Using HTTP Methods for RESTful Services,” [Online]. Available: https://www.restapitutorial.com/lessons/httpmethods.html. [Acedido em 07 08 2019].

[19] “What Is Idempotence?,” [Online]. Available: https://www.restapitutorial.com/lessons/idempotency.html. [Acedido em 08 08 2019].

[20] “What are idempotent and/or safe methods?,” [Online]. Available: http://restcookbook.com/HTTP%20Methods/idempotency/. [Acedido em 08 08 2019].

[21] “Using HTTP Methods for RESTful Services,” [Online]. Available: https://www.restapitutorial.com/lessons/httpmethods.html. [Acedido em 08 08 2019].

[22] T. Griffith, “HTTP Explained: The HTTP Status & Response Code Guide,” [Online]. Available: https://www.clickminded.com/http-status-codes/. [Acedido em 28 01 2019].

[23] S. Pearlman, “What are APIs and how do APIs work?,” 7 09 2016. [Online]. Available: https://blogs.mulesoft.com/biz/tech-ramblings-biz/what-are-apis-how-do-apis-work/. [Acedido em 29 08 2019].

[24] D. Angerer, “Headless CMS explained in 5 minutes,” 12 2017. [Online]. Available: https://www.storyblok.com/tp/headless-cms-explained. [Acedido em 11 2019].

[25] C. Arsenault, "What Is a Headless CMS and Should You Be Using One?," 03 2018. [Online]. Available: https://www.keycdn.com/blog/headless-cms. [Acedido em 15 12 2018].

[26] F. L. Nadeau, "Why would a decoupled CMS or Headless CMS benefit me?," 04 03 2017. [Online]. Available: https://www.quora.com/Why-would-a-decoupled-CMS-or-Headless-CMS-benefit-me. [Acedido em 02 01 2019].

[27] "OPEN SOURCE E-Commerce API & Admin Hub," 08 08 2019. [Online]. Available: https://getcandy.io.

[28] "Learn Git and GitHub without any code!," 08 08 2019. [Online]. Available: https://github.com/getcandy/candy-hub.

[29] "Add Enterprise Ecommerce to your Static Sites.," [Online]. Available: https://commercelayer.io/.

[30] "The only way to deliver commerce. Your way.," [Online]. Available: https://www.moltin.com/. [Acedido em 08 08 2019].

[31] "Flows is the custom commerce workflow," 08 08 2019. [Online]. Available: https://www.moltin.com/product/flows.

[32] "What is Scrum?," [Online]. Available: https://www.scrum.org/resources/what-is-scrum. [Acedido em 08 08 2019].

[33] "What is sprint planning?," 09 08 2019. [Online]. Available: https://www.scrum.org/resources/what-is-sprint-planning.

[34] "What is a sprint review?," 09 08 2019. [Online]. Available: https://www.scrum.org/resources/what-is-a-sprint-review.

[35] D. K. Uhlig, "Advantages and Disadvantages of the Scrum Project Management Methodology," 08 08 2019. [Online]. Available: https://smallbusiness.chron.com/advantages-disadvantages-scrum-project-management-methodology-36099.html.

[36] "Jira | Issue & Project Tracking Software | Atlassian," [Online]. Available: https://www.atlassian.com/software/jira. [Acedido em 07 06 2019].

[37] "Issue statuses, priorities, and resolutions," [Online]. Available: https://confluence.atlassian.com/adminjiracloud/issue-statuses-priorities-and-resolutions-973500867.html. [Acedido em 07 06 2019].

[38] "What is PHP?," [Online]. Available: https://www.guru99.com/what-is-php-first-php-program.html. [Acedido em 02 05 2019].

[39] "1.3.1 What is MySQL?," [Online]. Available: https://dev.mysql.com/doc/refman/8.0/en/what-is-mysql.html. [Acedido em 19 06 2019].

[40] C. Hopkins, "The MVC Pattern and PHP, Part 1," 04 04 2013. [Online]. Available: https://www.sitepoint.com/the-mvc-pattern-and-php-1/. [Acedido em 07 08 2019].

[41] "Routing," [Online]. Available: https://laravel.com/docs/5.6/routing. [Acedido em 09 08 2019].

[42] "Laravel vs Lumen," [Online]. Available: https://www.educba.com/laravel-vs-lumen/. [Acedido em 07 06 2019].

[43] I. Jound, "Comparison of performance between Raw SQL and Eloquent ORM in Laravel," SE-371 79 Karlskrona Sweden.

[44] "What is NGINX? How different is it from Apache (for example)?," [Online]. Available: https://www.nginx.com/faq/what-is-nginx-how-different-is-it-from-e-g-apache/. [Acedido em 08 08 2019].

[45] "Laravel Homestead," [Online]. Available: https://laravel.com/docs/5.8/homestead. [Acedido em 08 08 2019].

[46] "What is Laravel and Why You Should Learn it?," 2018 09 5. [Online]. Available: https://www.larashout.com/what-is-laravel-and-why-you-should-learn-it. [Acedido em 06 06 2019].

[47] "Database: Migrations," [Online]. Available: https://laravel.com/docs/5.6/migrations. [Acedido em 07 06 2019].

[48] "MAKING DEVELOPMENT AN ENJOYABLE EXPERIENCE," [Online]. Available: https://www.jetbrains.com/idea/features. [Acedido em 07 06 2019].

[49] K. Cooper, "Understanding Git (part 1)—Explain it Like I'm Five," 17 10 2017. [Online]. Available: https://hackernoon.com/understanding-git-fcffd87c15a3. [Acedido em 2019 09 19].

[50] "Soft Delete vs Hard Delete," 28 06 2015. [Online]. Available: http://abstraction.blog/2015/06/28/soft-vs-hard-delete. [Acedido em 10 08 2019].

[51] "Chapter 3. The Command-Line Test Runner," [Online]. Available: https://phpunit.de/manual/6.5/en/textui.html. [Acedido em 01 08 2019].

[52] “Test Assertions - How do they work?,” 2010 09 19. [Online]. Available: https://www.thoughtworks.com/insights/blog/test-assertions-how-do-they-work. [Acedido em 23 07 2019].

[53] “Assertions,” [Online]. Available: https://phpunit.readthedocs.io/en/7.4/assertions.html#assertequals. [Acedido em 08 08 2019].

[54] D. Tikhanski, “Open Source Load Testing Tools: Which One Should You Use?,” 29 08 2018. [Online]. Available: https://www.blazemeter.com/blog/open-source-load-testing-tools-which-one-should-you-use/. [Acedido em 16 08 2019].

[55] L. Gupta. [Online]. Available: https://howtodoinjava.com/dropwizard/dropwizard-basic-auth-security-example/. [Acedido em 12 06 2019].

[56] J. Paul, “How Does HTTP Basic Authentication Work in Spring Security?,” [Online]. Available: https://dzone.com/articles/how-does-http-basic-authentication-work-in-spring. [Acedido em 07 06 2019].

[57] W. Abirached, “Business Rules Must Be Enforced by the API,” [Online]. Available: https://designingforscale.com/business-rules-must-be-enforced-by-the-api/. [Acedido em 01 07 2019].

Anexos

Anexo A - Modelo de dados da solução final