INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

ÁREA DEPARTAMENTAMENTAL DE ENGENHARIA DE ELECTRÓNICA E TELECOMUNICAÇÕES E DE COMPUTADORES

Licenciatura em Engenharia Informática e Multimédia

Projeto Final

Dynamic Blocks – D-Blocks

Setembro, 2020

Orientador:
Professor Doutor Carlos Gonçalves

Alunos:
Heral Mukesh, 38834
João Ventura, 38950

Resumo

O projeto *Dynamic-Blocks*, cujo acrónimo é D-Blocks, tem como conceito a rápida construção de aplicações, explorando novas tecnologias de contentores dinâmicos e desenvolvimento por módulos. Este conceito permitirá a criação rápida e individual de aplicações reutilizando módulos já existentes ou acrescentando novos com o menor esforço e tempo possível.

Este projeto teve como objetivo realizar uma base demonstrativa de plataformas de venda modular e que terá como caso de estudo uma plataforma *web* de venda de fotografias *online*. Este conceito foi desenvolvido utilizando a tecnologia *Docker* permitindo assim criar uma estrutura modular, organizada e singular.

Assim, com este conceito e caso de estudo, no futuro será muito mais rápido e eficiente construir uma loja *online* ou plataforma de venda dado que essa base já está construída.

Palavras-chave:

Docker; Dynamic-Blocks; Python; Flask; Base de dados; Contentor; Imagem; Módulo; API, Servidor Web, Servidor Aplicacional;Plataforma;

Abstract

Dynamic Blocks whose acronym is D-Blocks, it's a project which the idea is to create a quick and good quality applications, exploring the containers technology known as *Docker*, to develop modules for applications.

The concept of this project is to create a platform that will be the base for different kinds of applications with separated modules, that together create a complete application. For this demonstration it will be created an online sales platform.

Specifically, for the case study it will be created an online sales platform to sell photographs and which his name is *Capture*. In the future with this case study, an online sales platform can be created much faster reusing/improving some of the modules and adding some new modules.

Keywords:

Docker; Dynamic-Blocks; Python; Flask; Database; Container; Module; Image; API, Web server, Applicational server; Platform;

Índice de Conteúdos

Índice de Figuras

Índice de Tabelas

1. Introdução

Por curiosidade e interesse, surgiu a ideia de poder ter uma infraestrutura que permitisse criar aplicações digitais em pouco tempo e com pouco esforço, e ainda que ao longo do tempo fosse cada vez mais rápido produzir estas aplicações.

De acordo com a Figura 1-1 o conceito *Dynamic Blocks* (cujo acrónimo é D-Blocks), tem por base três principais componentes para a criação de aplicações.

Figura 1-1 – Diagrama de componentes do projeto D-Blocks

A componente visual será responsável por toda a interface gráfica visível pelo cliente. A componente de controlo e validação aplicacional irá validar as ações do utilizador e controlar o fluxo de direcionamento entre as diferentes componentes visuais da aplicação. A componente de armazenamento de dados será responsável por guardar todos os dados necessários para o funcionamento da aplicação.

Para facilitar futuras aplicações, do mesmo género, é necessária uma base de construção, que no projeto será uma plataforma de vendas e como caso de estudo em específico uma plataforma de venda de fotografias.

Assim, a próxima aplicação do género loja *online* de produtos ou plataforma de vendas, reaproveitará componentes ou módulos já existentes e criará ou modificará apenas o que é necessário acrescentar para aquela aplicação, reduzindo o tempo e esforço na sua produção.

1.1. Descrição do conceito D-Blocks

A independência de secções de desenvolvimento é algo que facilita o tempo de entrega e a capacidade monetária de concorrer no desenvolvimento de produtos para clientes.

Para concretizar o conceito do projeto foi necessário estudar o denominador comum de certas aplicações separadas por géneros. O mercado alvo foi também uma peça fundamental no desenvolvimento do projeto pois existem muitos clientes sem conhecimentos/predisposição para desenvolverem eles próprios um *software* para o seu negócio por mais fácil que seja.

O D-Blocks permite que uma aplicação, tendo a base construída, seja rápida e fácil de entregar ao cliente e por isso reduzir custos e tempo para competir no mercado atual.

Com um pedido por parte do cliente para o desenvolvimento de uma aplicação, o trabalho a realizar será tanto menor quanto os módulos já realizados para outras aplicações, o que faz com que haja um reaproveitamento. Pequenas alterações, ou novos módulos serão acrescentados de acordo com o negócio específico.

O diagrama Figura 1-2 é a fórmula geral do conceito do projeto e da dinâmica de construção de aplicações futuras.

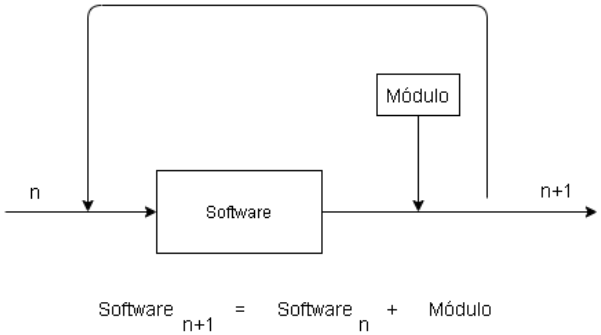

Figura 1-2 – Diagrama e Fórmula do conceito do projeto D-Blocks

Em que o *software* seguinte é o reaproveitamento dos *softwares* anteriores do mesmo género (alterando alguns detalhes de acordo com o negócio) e acrescentando módulos novos caso necessário. Se não houver módulos novos a acrescentar então o resultado do *software* novo será o total reaproveitamento dos *softwares* realizados anteriormente.

1.2. Exemplo base para o conceito D-Blocks

Entre *softwares* de gestão, plataformas de vendas, plataformas informativas ou apenas módulos específicos, o exemplo base para o conceito D-Blocks incide sobre uma construção base de uma plataforma web de vendas.

1.3. Objetivos

Pretende-se com este projeto perceber como funcionam tecnologias modulares para criar vários módulos de software que permitam uma maior rapidez e uma maior eficácia na entrega do produto final. Outro dos objetivos é a compreensão e o estudo sobre novas tecnologias no mercado atual para este tipo de projeto em específico.

A flexibilidade e o conhecimento de vários tipos de ferramentas de construção de software, permitem desenvolver capacidades críticas na escolha ideal para a realização de projetos futuros.

1.4. Organização do relatório

Além deste capítulo, este relatório está organizado de acordo com os capítulos, Trabalho relacionado e tecnologias utilizadas (capítulo onde serão descritas as tecnologias usadas no desenvolvimento do projeto e o que já existe de semelhante no mercado), Modelo proposto (capítulo onde serão apresentados os Casos de utilização, os Requisitos e a

Arquitetura), Implementação (capítulo onde serão apresentados aspetos relacionados com a implementação da estrutura modular (*Docker)*, da API (controlo aplicacional) e da base de dados), Caso de estudo (neste capítulo será apresentado a visão de negócio para a venda dos diferentes *softwares* requisitados pelo cliente e também um caso de estudo de uma plataforma de vendas), Conclusões e Trabalho Futuro (onde serão apresentadas conclusões do projeto e melhorias para o futuro) e Bibliografia (onde estão disponibilizados as fontes de informação que ajudaram no desenvolvimento do projeto).

O dossier do projeto encontra-se disponível no seguinte *link* https://github.com/J999Ventura/dblocks.

2. Trabalho relacionado e tecnologias utilizadas

O mercado atual e as necessidades dos clientes levam a transportar os seus negócios para um mundo mais digital e não tanto presencial. Cada vez mais os clientes requisitam, junto de empresas ou particulares, softwares para expandir o seu negócio de local para nacional ou internacional, mas muitos acabam por nem o fazer pois a componente financeira ou a falta de conhecimentos técnicos assim não o permite.

Uma plataforma relacionada com o conceito do projeto *Dynamic Blocks* (cujo acrónimo é D-Blocks) é o *Shopify*.

2.1. Trabalho relacionado

Um dos exemplos existentes no mercado é o *Shopify*, uma plataforma de criação de plataformas de venda *online* que permite a qualquer pessoa a criar, sem ter grandes conhecimentos a nível informático.

O *Shopify* delega o trabalho para o cliente onde estes podem personalizar as suas plataformas, no entanto requer uma mensalidade (despesas de alojamento e suporte) e está limitado a plataformas de vendas online e não para diversos tipos de produtos e dispositivos.

O D-Blocks comparativamente ao *Shopify* não delega o trabalho para o cliente, nem cobra mensalidades, e onde o pagamento é feito uma única vez e o orçamento inclui despesas temporárias de alojamento (ex. limite de 3 anos) suportados pelo cliente. Com o trabalho delegado para o D-Blocks, o cliente não tem de se preocupar com aspetos técnicos, nem ter o mínimo de capacidades na área das tecnologias de informação. Assim, permite ao cliente focar-se apenas nos requisitos de negócio poupando tempo para outras questões.

O pedido do cliente com o D-Blocks pode ser totalmente customizado à medida e sem necessidade de ter conhecimentos médios ou avançados o que é mais complicado no *Shopify* visto que o trabalho é delegado para o cliente.

2.2. Tecnologias utilizadas

A tecnologia base que permite o desenvolvimento do conceito D-Blocks são os contentores dinâmicos, onde é possível separar os diferentes blocos do *software* de forma independente e facilitar implementações e o *deploy* em qualquer máquina. Para o exemplo base deste conceito (plataforma de vendas*)* são usadas as descritas nos próximos subcapítulos.

2.2.1.Postgres-SQL

Um sistema de base de dados *open source* baseado em *Structured Query Language* (SQL) que corre em todas as plataformas e oferece linguagens de definição e manipulação de dados simples e intuitivas. Este sistema é o sistema usado na implementação da demonstração do conceito.

2.2.2.Python

É uma linguagem interpretada e de alto nível, e uma das mais populares da atualidade. Esta linguagem foi escolhida pois o conhecimento retido sobre ela durante o curso foi somente numa vertente de processamento de imagem, classificação de dados e outro tipo de funcionamento básico da linguagem, sendo então uma possibilidade aprofundar num contexto diferente (*web* e *API*) do que costumava ser usada.

2.2.3.Docker

De forma a aplicar o conceito do projeto assente na base da modularização, é necessária uma tecnologia que o permita. O *Docker* é uma aplicação que fornece a capacidade de abstração e automação de uma virtualização de sistema operativo. Permite criar contentores, que correm uma instância de um sistema operativo com determinados serviços, evitando a sobrecarga de manter máquinas virtuais e dando a possibilidade de correr estas aplicações em qualquer máquina que a hospede. Tendo um contentor para a componente visual e outro para a componente de controlo aplicacional (API) é mais flexível a mudança não só neste projeto como também em futuros projetos, alterando assim apenas no respetivo contentor. Um exemplo com a base de demonstração do projeto (plataforma de vendas) é o facto de se puder modificar apenas o contentor da componente visual (alterando cores, esquemas da interface) sem se alterar no contentor da API, e o inverso também é válido. O acréscimo ou remoção de funcionalidades pode ou não aplicar alterações em ambos. Para modularizar mais ainda as aplicações, pode-se colocar em diferentes contentores alguns módulos da API, como por exemplo, o módulo de autenticação. O sistema de autenticação passando para um contentor torna-o independente e mais facilmente utilizável em projetos no futuro. Toda a funcionalidade que seja possível de funcionar de forma independente num projeto, pode ser colocada num contentor, interligando-os posteriormente. O *Docker* facilita também o processo de *deploy*, em que não será necessário instalar no servidor do cliente toda e qualquer tecnologia utilizada no projeto, bastando por isso apenas instalar o *Docker*, fazer *build* e *run*. No caso de o cliente optar por uma solução na *cloud*, existe a possibilidade de um *deploy* continuo em que qualquer alteração após a aplicação estar em produção é facilmente substituível carregando o novo contentor que já contém as novas alterações.

2.2.4.Flask

É uma micro *framework* (não requer ferramentas ou bibliotecas) desenvolvida em *python* e simples para desenvolvimento de aplicações web e por isso foi utilizada na demonstração do conceito (plataforma de *web* de vendas). Tanto a API como a componente de visualização utilizam o *Flask*.

2.2.5.Flask-SqlAlchemy

É uma extensão do *Flask* que que facilita a comunicação de programas desenvolvidos em *python* com base de dados. Na maioria dos casos é utilizada como um ORM (*Object Relation Mapper*) que traduz as classes de *python* em tabelas relacionais na base de dados e converte automaticamente funções em instruções SQL.

2.2.6.Flask-Migrate

É uma extensão que lida com migrações de base de dados *SQL Alchemy* para aplicações *Flask* usando *Alembic*. Assim, é possível fazer um versionamento da base de dados em que é importante não só perceber as alterações como retroceder caso algo corra mal.

2.2.7.Flask-Mail

É uma extensão que permite enviar emails pela aplicação configurando o servidor SMTP.

2.2.8. Flask-Buzz

Extensão que permite um tratamento de exceções onde é possível converter mensagens de erro em objetos JSON.

2.2.9. WTForms

Biblioteca que permite a criação de *Forms* HTML com validações, de uma forma muito simples em *python*.

2.2.10. Gunicorn

O servidor por defeito do *Flask* apenas serve para desenvolvimento onde o número de pedidos é pequeno. Para um servidor de produção é necessário utilizar o *Gunicorn* que é um servidor HTTP WSGI *Python* para Unix onde este permite múltiplas instâncias e múltiplos pedidos.

2.2.11. HTML

Linguagem de marcas constituída por conjunto de *tags* bem definidas que formam a estrutura de uma página *web* que todos os *browsers* sabem interpretar e apresentar ao utilizador as informações dentro dessas *tags*. Será permitida a apresentação de toda a informação necessária para o cliente final.

2.2.12. Javascript

Linguagem de *scripting* para páginas *web* que proporciona dinamismo às páginas através de cliques ou inserção de texto do utilizador da aplicação. Com esta linguagem, é possível fazer uma validação primária de dados antes de enviar ao servidor e evitar comunicações desnecessárias entre cliente e servidor derivado a erros de inserção de dados ou incorreta utilização da aplicação.

2.2.13. CSS

Linguagem que informa o browser como deve ser apresentado o conteúdo definindo os estilos dos elementos *HTML*. Com esta linguagem é possível colocar mais beleza e organização nas páginas web construídas.

2.2.14. Bootstrap

Conjunto de *templates* e regras CSS e *Javascript*, que facilitam a construção do layout de páginas *web*.

2.2.15. GIT

Controlador de versões que permite gerir e organizar trabalho entre uma ou mais pessoas, guardando registos de várias etapas do trabalho e permitindo a sobreposição organizada de informação, inserida por vários trabalhadores ao mesmo tempo.

3. Modelo proposto

Este capítulo descreve a análise do projeto, passando pelos temas: **Casos de utilização (secção 3.1), Requisitos (secção 3.2)** e **Arquitetura (secção 3.3).**

3.1. Casos de utilização

Na implementação do conceito, os casos de utilização são generalizados para qualquer plataforma de venda *online*. O cliente terá de fazer um registo e posteriormente um login para comprar ou vender algum produto, e fazer *logout* caso queira terminar a sua sessão. Para visualizar qualquer produto na plataforma não necessita de estar autenticado.

Os casos de utilização serão semelhantes em qualquer projeto futuro da mesma categoria (plataforma de venda).

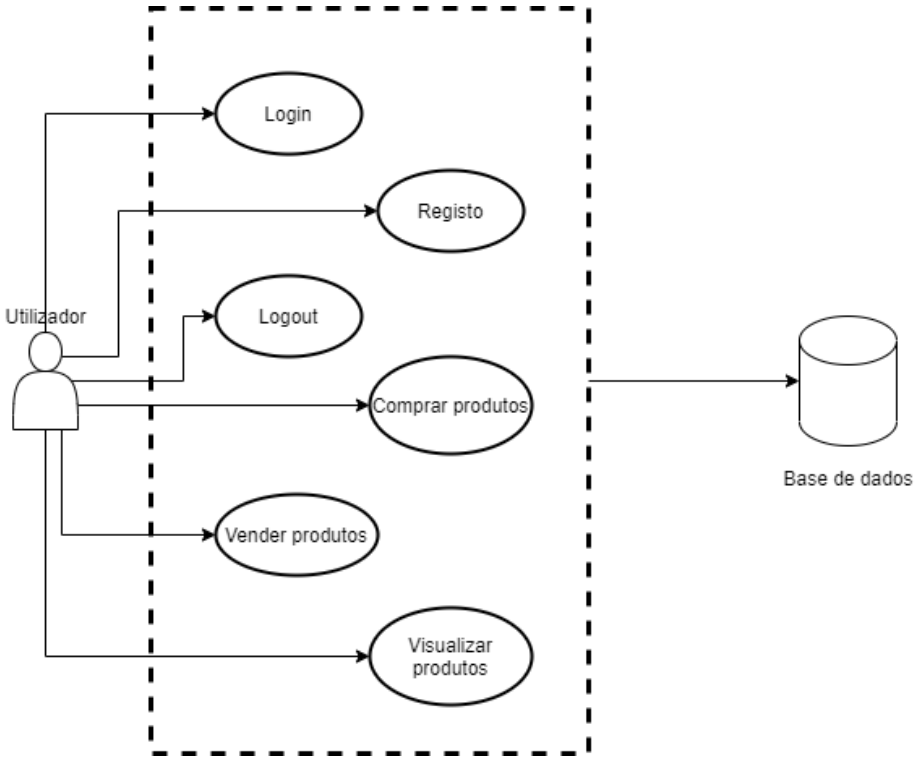

Figura 3-1 – Casos de utilização de uma plataforma de vendas

Na Figura 3-1 é possível observar várias interações que um utilizador pode. A possibilidade da autenticação, registo ou término de sessão são inerentes a qualquer aplicação, já a venda, compra e visualização de produtos é transversal a todas as plataformas de vendas. A grande vantagem será a capacidade de reutilizar estas ações em outros produtos futuros, podendo fornecer a vários clientes, que pretendam adquirir uma aplicação que necessite destas funções, várias soluções diferentes, reutilizando as mesmas e modificando apenas o *layout* apresentado.

Só é possível este reaproveitamento devido á modularização, fazendo com que sempre que exista um destes pedidos, este é realizado do servidor *web* para o servidor aplicacional através da API, que por sua vez processa, armazena e visualiza dados numa base de dados, para poder dar resposta aos pedidos efetuados. Esta base de dados utilizada é uma base de dados com alguns componentes já genéricos, como por

exemplo, uma tabela de *Users* (Utilizadores), *Products* (Produtos), *Purchases* (Compras) e *Roles* (Permissões).

3.2. Requisitos

Neste capítulo serão apresentados os requisitos funcionais e não funcionais que constituíram o desenvolvimento realizado durante o projeto para construir o conceito *Dynamic-Blocks* (cujo acrónimo é D-Blocks). As tabelas de requisitos são constituídas por três colunas:

- Referência – Indica a prioridade dada á execução do requisito.
- Função – Indica qual a função do requisito
- Categoria – representa a capacidade de poder ver visualmente o resultado do requisito e pode ter dois valores, "Evidente" ou "Invisível".

Um requisito funcional representa o que o *software* faz, em termos de tarefas e serviços. Uma função é descrita como um conjunto de entradas, o seu comportamento e as saídas. Os requisitos funcionais podem ser cálculos, detalhes técnicos, manipulação de dados e de processamento e outras funcionalidades específicas que definem o que um sistema, idealmente, será capaz de realizar. Por outro lado, os requisitos não funcionais são os requisitos relacionados ao uso da aplicação em termos de desempenho, usabilidade, confiabilidade, segurança, disponibilidade, manutenção e tecnologias envolvidas. Estes requisitos dizem respeito a como as funcionalidades serão entregues ao utilizador do *software*.

3.2.1. Requisitos funcionais da base de dados

Foram criados os requisitos base/comuns necessários para a construção de uma base de dados transversal a qualquer plataforma de vendas.

Tabela 1 – Requisitos funcionais da base de dados

Referência	Função	Categoria
R2.1	Definição da Base de Dados (Modelo Entidade-Associação)	Invisível
R2.2	Criação de Schemas	Invisível
R2.3	Criação de Tabelas	Invisível
R2.4	Criação de possíveis funções, procedimentos ou triggers	Invisível
R2.5	Inserção de dados nas tabelas	Invisível

3.2.2. Requisitos funcionais do servidor aplicacional

Foram criados os requisitos base/comuns necessários para a construção de uma API transversal a qualquer plataforma de vendas.

Tabela 2 – Requisitos funcionais servidor aplicacional

Referência	Função	Categoria
R3.1	Definição da API	Invisível
R3.2	Conexão com o módulo da Base de Dados	Invisível
R3.3	Construção da API	Invisível
R3.4	Registo de utilizador	Invisível
R3.5	Autenticação	Invisível
R3.6	Gestão de permissões (admin, user comum)	Invisível
R3.7	Gestão de vendas e pagamentos	Invisível
R3.8	Testes de pedidos à API	Invisível

3.2.3. Requisitos não funcionais da estrutura modular

Para se que seja possível a concretização do conceito, os requisitos da estrutura modular são a base do projeto (Tabela 3). Por observação pode-se verificar que todos os requisitos são de Categoria "Invisível", pois por ser de motivos estruturais o cliente não tem qualquer visão sobre este.

Tabela 3 – Requisitos funcionais estrutura modular (Docker)

Referência	Função	Categoria
R1.1	Contentor para servidor aplicacional em Linux	Invisível
R1.2	Contentor do servidor web	Invisível
R1.3	Configuração do contentor do servidor aplicacional (exposição de portos, python)	Invisível
R1.4	Configuração do contentor do servidor web (exposição de portos, python)	Invisível
R1.5	Colocar os contentores numa infraestrutura (servidor)	Invisível
R1.6	Habilitar visibilidade para o exterior DNS público	Invisível

3.3. Arquitetura

Nesta secção é explicada a arquitetura geral do projeto bem como o detalhe de cada bloco.

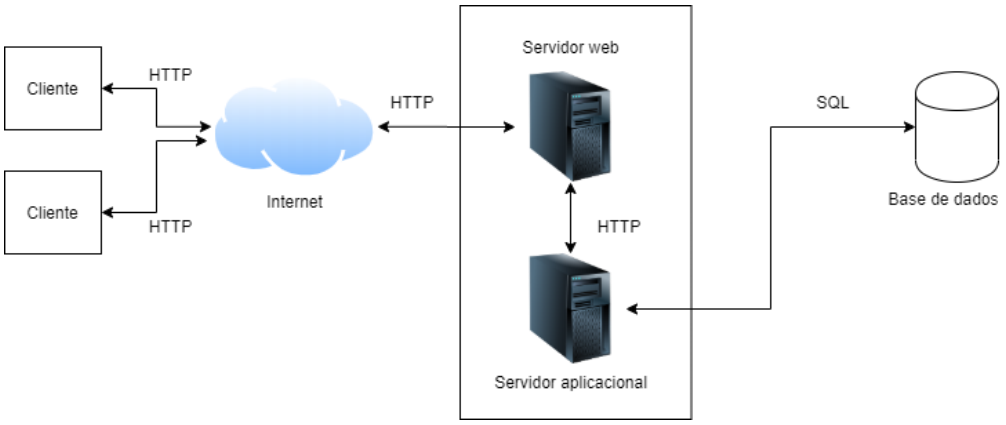

Figura 3-2 – Arquitetura cliente-servidor

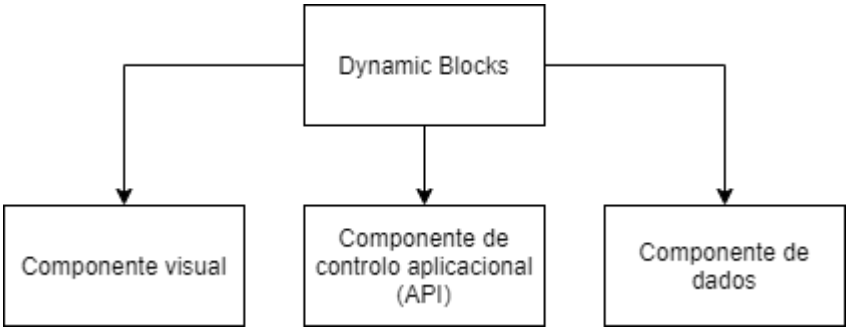

Figura 3-3 – Arquitetura geral do projeto Dyanamic Blocks

Na Figura 3-2 está representada a arquitetura cliente servidor onde cada pedido feito pelo cliente, é enviado pela internet através do protocolo HTTP e é processado pelo servidor que retorna uma resposta à ação do cliente.

A comunicação do cliente para o servidor é feita através de browser, usando páginas HTML e quando o servidor recebe um pedido, a parte responsável de tratamento desse pedido gera uma resposta. Neste caso em concreto o servidor da plataforma web, irá comunicar com o servidor da API que posteriormente irá comunicar com o servidor de base de dados fazendo assim com que o servidor web receba os pedidos do cliente e envie informação para o servidor aplicacional (API) para este processar a informação (podendo ou não comunicar com o servidor de base de dados) e retornar elementos para o servidor web.

A Figura 3-3 representa a separação por blocos do desenvolvimento do projeto e que está dividido em três partes:

- Componente visual que é a representação visual de uma aplicação do conceito D-Blocks.
- Componente de controlo aplicacional API (*Application Program Interface*) que representa todo o fluxo de negócio com regras bem definidas.
- Componente de dados que trata do armazenamento de todos os dados necessários para o bom funcionamento do projeto.

Este tipo de arquitetura permite que ao longo do tempo sejam acrescentados blocos singulares que formam uma só aplicação, conforme a Descrição do conceito D-Blocks e a Figura 1-2.

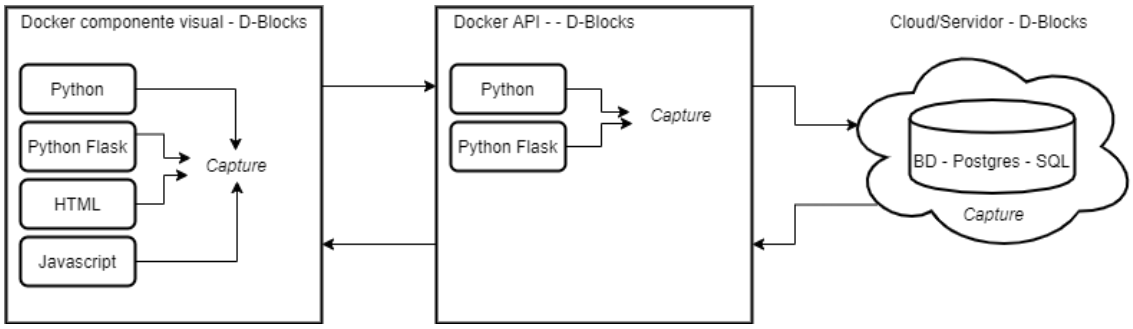

Figura 3-4 - Arquitetura interna do módulo servidor web e aplicacional

Como descrito na Figura 3-4, tanto o módulo da componente visual como da API, são independentes e estão dentro de um contentor Docker.

O módulo visual, composto pelo servidor web, foi desenvolvido em Python, HTML e JavaScript, com auxílio da framework Flask. O módulo responsável pela API, foi realizado em Python com auxílio da framework Flask que facilita na criação de uma API através de rotas e ainda a interagir com uma base de dados Postgres SQL, para armazenar informação necessária da aplicação. A interligação entre estes componentes é feita da seguinte forma:

- Componente visual faz pedidos à API
- API faz acessos à base de dados
- API processa, trata e valida os dados
- API responde à componente visual
- Componente visual mostra ao utilizador os resultados

Novas funcionalidades:

Esta forma de construção, permite adicionar ou modificar um componente e é possível fazê-lo sem alterar funcionalidades já existentes.

Um exemplo pode ser a autenticação em que se for necessário passar a utilizar um serviço de autenticação por exemplo da *Google*, nada se terá que modificar neste módulo *web*, pois o pedido é sempre feito á API do servidor aplicacional. E assim poderá funcionar para o módulo do pagamento e outros.

Novas aplicações:

Este exemplo de plataforma de vendas *online* de fotografias pode ser modificado para outro tipo de plataforma de vendas de qualquer produto, como por exemplo uma de roupa em que apenas é necessário fazer modificações ao *layout*, algumas regras de negócio, e o sistema de utilizadores, podendo reaproveitar todos os outros componentes.

O ideal seria ter um *template* para vários tipos de negócio e que fosse possível reaproveitar cada *template* e que ajudaria a poupar tempo que por sua vez se traduz em dinheiro para uma empresa.

4. Implementação

Nesta secção vai ser descrita toda a implementação para uma plataforma de vendas, de uma forma genérica. Como já referido o *Docker* é a tecnologia que nos permite realizar todo este conceito. No entanto neste capítulo irá perceber como é que ele foi utilizado nas diferentes áreas de implementação do conceito.

4.1. Implementação e utilização do Docker

O *Docker* é base de quase todo o conceito *Dynamic-Blocks*, como já referido anteriormente, esta tecnologia permite a criação e isolamento de componentes especificas de um sistema. A vantagem para a sua utilização neste projeto é o facto de ter componentes que se comportam como peças de um *puzzle*, e que se complementam quando juntas.

Foram utilizadas duas imagens *Docker*, uma para o sistema responsável pelo modelo de negócio e API (Servidor Aplicacional), exposta para o exterior para ser consumida, e outra com o modelo visual do sistema (Servidor Web), que é a página Web, ou uma possível aplicação Desktop ou Mobile.

A utilização desta tecnologia *D*ocker baseia-se na sua característica de ser um contentor, isto é, uma imagem de um sistema operativo que permite correr um serviço isolado em qualquer máquina que suporte a tecnologia *Docker* (basta fazer a instalação do *software Docker*), e esta irá correr em qualquer máquina, sem necessitar de configurações externas ou adicionais, como apontar para endereços de IP diferentes, passwords, portos, entre outros. Esta vantagem permite mover estes contentores para qualquer máquina, alojamento, ou até fornecer um determinado módulo separado a um cliente.

Outra utilização bastante importante é a possibilidade de criar um ambiente virtual em que o módulo funcional é instalado e inicializado nessa imagem.

Se não fosse utilizado o *Docker* todas as configurações e instalações necessárias para os módulos funcionarem (*Python*, respetivas bibliotecas e o código da aplicação) teriam de ser instalados manualmente e individualmente no servidor de alojamento, aumentando o tempo de desenvolvimento de cada módulo e podendo causar incompatibilidades/erros não calculados (e.g., “Mas no meu computador funciona/funcionou).

É esta tecnologia (Docker) que contém o código de cada servidor aplicacional, web ou outro possível modulo a adicionar.

Para que o conceito seja concretizado, cada modulo ou cada instância Docker, terá de conter o código que forma a aplicação funcionar. Para isto ser possível foi configurado um ficheiro chamado *Dockerfile*, que tem todos os passos todos da configuração do ambiente virtual a ser criado. (Figura 4-1)

```
FROM python:3.7

ENV FLASK_ENV=production
ENV SECRET_KEY=b'\xcb\xd8\x15\xd9\xf1\x94R\xbd\x8b\xdfsv\xe7(\x9b\x08'
ENV MAIL_USERNAME=betamain10@gmail.com
ENV MAIL_PASSWORD=alphamain_10
ENV PRAETORIAN_CONFIRMATION_SENDER=betamain10@gmail.com
ENV PRAETORIAN_RESET_SENDER=betamain10@gmail.com
ENV PRAETORIAN_RESET_URI=https://captureapi-2uo2zekoka-ew.a.run.app/api/v1/auth/completeresetpwd
ENV SQLALCHEMY_DATABASE_URI=postgres+pg8000://postgres:captureadmin@/capturedb?unix_sock=/cloudsql/polished-bounty-276011:europe-west1:capturedb/.s.PGSQL.5432
ENV MAIL_SERVER=smtp.googlemail.com
ENV MAIL_PORT=587
ENV MAIL_USE_TLS=True
ENV PRAETORIAN_CONFIRMATION_SUBJECT=UserRegistration
ENV PRAETORIAN_CONFIRMATION_URI=https://captureapi-2uo2zekoka-ew.a.run.app/api/v1/auth/verify
ENV JWT_ACCESS_LIFESPAN=24hours
ENV JWT_REFRESH_LIFESPAN=2days

COPY . /usr/src/app
WORKDIR /usr/src/app/src

RUN pip install --upgrade pip && \
    pip install -r requirements.txt


CMD gunicorn --worker-class gevent --workers 4 --bind 0.0.0.0:8080 wsgi:app --max-requests 10000 --timeout 5 --keep-alive 5 --log-level info --reload
#CMD ["flask", "run", "--host=0.0.0.0", "--port=5000"]
```

Figura 4-1 - Dockerfile do servidor aplicacional

Analisando o ficheiro (*Dockerfile*) tem-se:

- "FROM python:3.7" – Traduz a "criação" de uma máquina virtual Ubuntu com o Python, versão 3.7 instalada.
- "ENV ..." – Criação de variáveis de ambiente. Estas servirão para utilizar qualquer tipo de configuração necessária na máquina. Por exemplo, IP's de Bases de Dados, Emails, entre outros.
- "COPY . /usr/src/app" – Copia a diretoria onde o ficheiro se encontra para uma diretoria (/usr/src/app) na máquina a ser criada. Neste caso a pasta "app" é onde se encontra todo o código realizado para um determinado modulo.
- "WORKDIR /usr/src/app/src" – Situa o local onde "nos" encontramos a trabalhar no ambiente a ser criado. Pode-se traduzir num comando "cd /usr/src/app/src".
- "RUN ..." – Corre qualquer comando que esteja á sua frente durante a construção do ambiente. Neste caso está a instala a ferramenta *"pip"*.
- "CMD ..." – Serve para quando se liga o ambiente, correr um determinado comando, quer o ambiente já esteja a construir ou após este ter acabado de se construir. Um exemplo seria ter um comando que correr sempre que se liga o computador. Neste caso está-se a correr o código que liga o servidor aplicacional disponibilizando a API para o exterior, pronta a ser utilizada.

De modo a que o conceito funcione corretamente tem de se ter uma instância de *Docker* para cada servidor, neste caso duas (servidor aplicacional e outro Web).

Para aplicar o conhecimento de *Docker*, foi necessário aprender alguns comandos que ajudam no seu entendimento e no seu funcionamento. Os comandos são:

- docker build {diretoria do dockerfile} – Constrói uma imagem a partir de um ficheiro *Dockerfile*. É assim que se constrói as primeiras imagens de cada módulo. Este comando quando é corrido a primeira vez constrói o ambiente virtual do zero, no entanto quando corrido pela segunda vez, não constrói de novo o ambiente apenas atualiza os ficheiros diferentes, isto caso não exista nenhuma imagem deste módulo.
- docker stop – Desliga um ou mais contentor que estejam ligados
- docker start – Liga um ou mais contentores estejam desligados
- docker run – Para correr uma imagem no caso do projeto utilizou-se "docker run -d --name website -p 5001:5000 docker_website", para dizer qual o porto que se quer que seja exposto para o exterior do contentor.

Foi ainda utilizado um outro ficheiro (docker-compose.yaml, Figura 4-2), que permite a criação de várias máquinas *Docker* ao mesmo tempo e por ordem.

```
version: '3'

services:
    product-services:
        build: ./server
        volumes:
            - ./server:/usr/src/app
        ports:
            - "5000:5000"
    website:
        build: ./website
        volumes:
            - ./website:/usr/src/app
        ports:
            - "5001:5000"
        depends_on:
            - product-services
```

Figura 4-2 - Ficheiro docker-compose

O ficheiro da Figura 4-2 contém as diretorias para onde os ficheiros de projeto serão guardados dentro do ambiente virtual a ser criado, e especifica os portos a serem expostos para o exterior da máquina *Docker*, neste caso o "*website*", estará a expor a porta 5000 interna, para a máquina local no porto 5001. Assim, para aceder ao website seria utilizar o url **http://127.0.0.1:5001/**.

Para executar este ficheiro basta correr o comando "docker-compose up --build", e todos os contentores serão instanciados corretamente apenas com um comando.

Um contentor é uma instância de uma Imagem. Uma imagem é como um ficheiro com código que é utilizado para correr dentro do contentor. É construído essencialmente pelas instruções utlizadas durante a construção do ambiente virtual.

Neste conceito a utilização do *Docker*, traduz-se como já dito anteriormente em duas instâncias *Docker*, uma para o servidor aplicacional e outra para o servidor *web*, que têm configurados no *Dockerfile* os portos e endereços de ambos, para se poderem comunicar. Para que possa tudo começar a funcionar, basta na linha de comandos executar o comando, *"docker build"* e esperar uns minutos, no fim tudo ficará pronto a ser utilizado.

A base de dados não foi possível incorporar no *Docker*, pois este é demasiado instável para guardar dados em produção. Caso se destrua por algum motivo o contentor que representa a base de dados, todos os dados serão perdidos. Por este motivo utilizamos a *Cloud-SQL* da *Google Cloud Platform*.

Caso se pretenda reutilizar um módulo para o tornar numa outra aplicação, basta pegar na pasta que contém o *dockerfile* e o código da aplicação e colocar num local pretendido, e após isso modificar o código da aplicação a gosto do cliente. Assim que terminadas as modificações, é apenas necessário abrir a linha de comandos e fazer *"docker bluid"*. Ao fim de poucos minutos a aplicação já a correr e pronta a ser utilizada.

Possíveis modificações futuras seriam, utilizar a vantagem modelar que o *Docker* fornece e criar um sistema de autenticação, ou de pagamento em módulos separados, onde teriam uma API exposta para que fosse possível interagir com eles, e assim ter duas componentes individuais, que poderiam ser utilizadas, caso seja necessário criar um servidor aplicacional para um negocio diferente ou para um outra situação.

4.2. Implementação da API

A API é o que permite a um modulo ser comunicado pelo exterior, e que define as possíveis interações com este modulo através de *endpoints*. A implementação da API requer à semelhança dos restantes blocos, uma análise prévia dos requisitos, definição da estrutura (*endpoints*) e a implementação das regras de negócio para o funcionamento do projeto. Este bloco é extremamente importante porque comunica com outros possíveis blocos, como por exemplo, o bloco da componente visual, enviando o conteúdo para ser apresentado, e o bloco da componente de dados para obter, atualizar, eliminar e inserir conteúdo, ou até outros blocos externos que representem algum comportamento necessário (e.g. Módulo de autenticação ou pagamentos).

Neste caso em particular utilizou-se um contentor *Docker* para armazenar o servidor Aplicacional que expõe uma API, permitindo a quem necessitar, executar pedidos á mesma.

4.3. Implementação da base de dados

De acordo com os Casos de utilização é necessário criar um modelo genérico de gestão dos dados. A Figura 4-3 demonstra as tabelas da base de dados necessárias para iniciar uma plataforma de vendas de produtos.

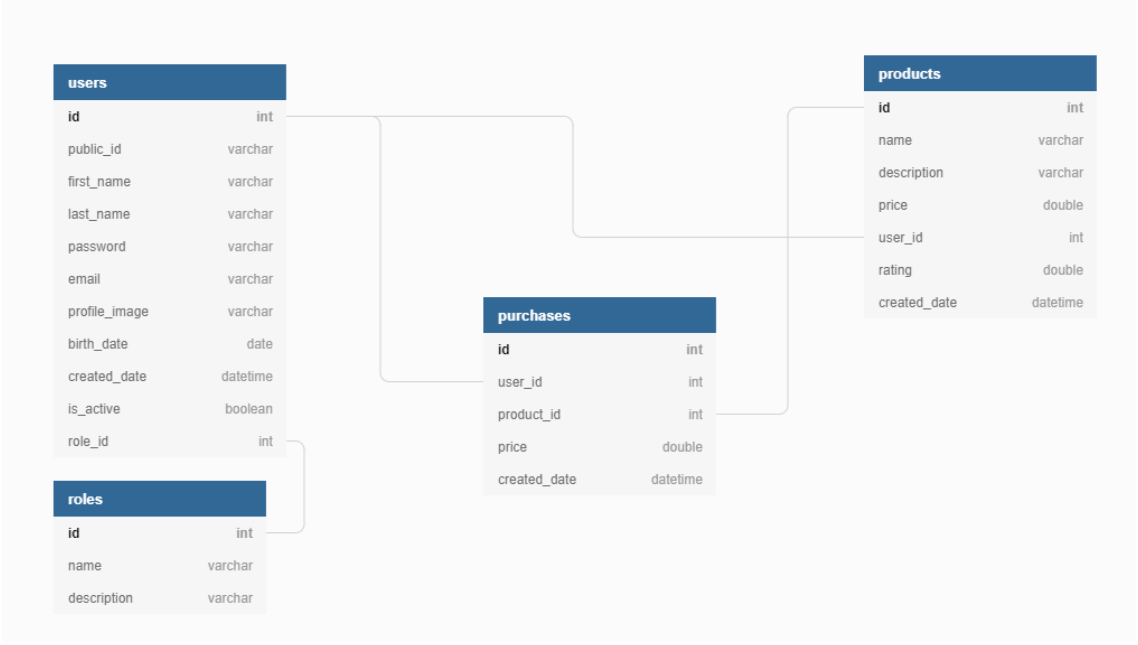

Figura 4-3 - Modelo entidade associação genérico para uma plataforma de vendas

No tópico de Anexos na secção B é possível verificar o detalhe dos campos pertencentes a estas tabelas (restrições e associações).

5. Caso de estudo

O caso de estudo incide na construção de uma plataforma de vendas que servirá de base para vendas de qualquer produto no entanto para exemplo será uma plataforma de venda de fotografias de acordo com o conceito Dynamic Blocks (cujo acrónimo é D-Blocks) e também com os Casos de utilização (secção 3.1).

Esta plataforma de vendas, denominada *Capture*, será uma plataforma de venda de fotografias que surgiu da importância e gosto pela fotografia, como arte, lazer e material de utilização profissional em diversas áreas, como revistas, livros, documentários, filmes, páginas web, anúncios, entre outros. Cada vez mais a imagem é algo com grande relevo na nossa vida, podemo-la encontrar desde os computadores, às televisões, smartphones ou anúncios de rua. A imagem é a grande linguagem simplificada, "Uma imagem vale mais que mil palavras" – Confúcio (filósofo chinês).

Hoje em dia com a proteção de dados e os protocolos de *copyright*, torna-se complicado de encontrar uma fotografia que se adapte ao que se procura, e nem sempre é fácil encontrar um profissional da área ou alguém que, por lazer, nos possa ajudar a descobrir a imagem desejada.

Dados estes fatores, a venda de fotografias despertou a curiosidade de construir uma loja online para a visualização e venda das mesmas. Por outro lado, o facto da fotografia ser em formato digital ajuda na gestão de compra, venda e meio de entrega do produto, não havendo um stock físico ou um intermediário de entrega. Como exemplo para um segundo *software* de loja online poder-se-á reaproveitar muitos componentes desta base e acrescentar alguns módulos caso necessário (por exemplo módulo de gestão de stock).

Assim que esta aplicação base estiver terminada poder-se-á reaproveitar os seus módulos para uma futura plataforma de vendas ou loja *online* de outro tipo de negócio. Neste exemplo base o que irá diferenciar uma plataforma de vendas de uma loja online será apenas o proprietário do produto vendido. Na plataforma de venda de fotografias o proprietário será o utilizador registado, numa loja online seria uma empresa proprietária dos produtos.

Como referido na Introdução deste relatório, o conceito é dividido em três componentes distintas e os seguintes tópicos definem o enquadramento do caso de estudo nesses componentes.

5.1. Modelo de negócio

A divulgação empresarial do projeto passará no futuro por uma plataforma onde o cliente pode requisitar um *software* definindo o tipo, loja *online*/plataforma de venda, *software* de gestão de produtos, *software* de gestão de recursos humanos, entre outros. Numa fase inicial esta divulgação será pelas redes sociais ou por conhecimento de pessoas que estejam à procura de criar ou melhorar o software para o seu negócio, e o pedido feito através de *e-mail* ou contacto telefónico. Num futuro existirá uma plataforma de realização de pedidos em que é possível o cliente fazer o pedido, verificar o estado do pedido que fez e algumas outras funcionalidades.

O orçamento ao pedido do cliente, para corresponder às suas expectativas e combater a concorrência dependerá do esforço aplicado para a realização do *software* solicitado. Quanto mais reaproveitamento houver de *software* anterior, mais baixo será o orçamento fornecido, menor será o tempo de realização e haverá mais hipóteses de convencer o cliente a contratar o serviço.

O cliente pode solicitar apenas um módulo a acrescentar a um *software* que tenha (ex.: módulo de autenticação).

O *Dynamic Blocks* (cujo acrónimo é *D-Bloks*) oferece pedidos de suporte, com um número limitado e dependendo da complexidade do pedido. Caso seja um acréscimo de uma nova funcionalidade terá uma cobrança diferente de uma mudança por exemplo apenas na visualização ou numa regra de negócio.

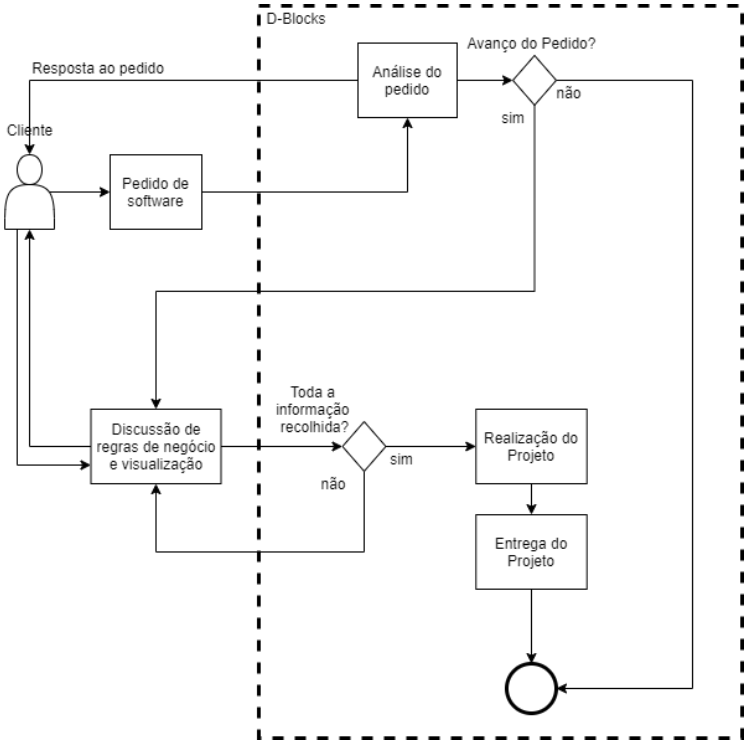

Figura 5-1 – Processo do modelo de negócio Dynamic Blocks

Conforme a explicação anterior e a Figura 5-1 após um pedido ser aprovado (já com orçamentos definidos e aceites pelo cliente), discutidas as regras de negócio e todos os detalhes gráficos, a equipa D-Blocks inicia o desenvolvimento e posteriormente entrega o projeto finalizado e pronto a funcionar ao cliente. Como 12 e aplicação deste processo, foi desenvolvida uma base para uma plataforma *web* de vendas e especificamente para venda de fotografias. Esta plataforma é denominada de *Capture*.

5.2. Componente visual

A componente visual é diferente de projeto para projeto e na implementação desta componente o peso temporal foi menor porque apenas serve como exemplo para uma plataforma web de vendas. Na Figura 5-2 é possível verificar as etapas de desenvolvimento.

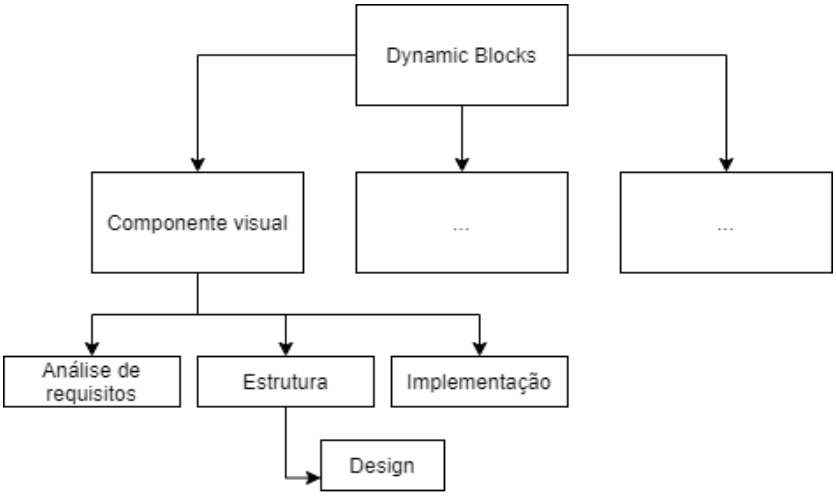

Figura 5-2 – Etapas na construção da componente de visualização

Primeiramente é necessário fazer o levantamento e análise de requisitos para definir os diferentes fluxos de transição na plataforma (conforme explicado no Modelo de negócio e na Figura 5-1). Após esta etapa tem de se definir a estrutura gráfica a ser apresentada e passar à fase de implementação abstraindo qualquer comunicação com outras componentes e esta abstração permite que no futuro esta componente de visualização possa ser ou reaproveitada para outro projeto alterando pequenas informações gráficas ou paleta de sem ter impacto nas restantes componentes (Componente de controlo aplicacional (API) e Componente de dados). Após a fase de implementação é possível ver uma estrutura visual montada e que precisará apenas de alguns ajustes para a entrega do resultado desta componente. Seguem-se algumas demonstrações da estrutura visual que vai de encontro aos Casos de utilização definidos para este conceito.

Na Tabela 4 encontram-se os requisitos funcionais desta componente, onde na sua maior parte, são de categoria "Evidente", isto é, são visíveis para o utilizador.

5.2.1. Requisitos funcionais da aplicação web

Tabela 4 – Requisitos funcionais servidor web

Referência	Função	Categoria
R4.1	Registar do utilizador	Evidente
R4.2	Autenticar do utilizador	Evidente
R4.3	Listar fotografias disponíveis para venda (novas, mais vendidas e destaque)	Evidente
R4.4	Visualizar fotografias por categoria ou pesquisa	Evidente
R4.5	Visualizar carrinho de compra	Evidente
R4.6	Garantir persistência e integridade dos dados	Invisível
R4.7	Visualizar página e ações de administrador	Evidente
R4.8	Download e Upload e Eliminação de fotografias	Evidente
R4.9	Comprar uma fotografia	Evidente

a) Página inicial

A página inicial é a página onde os utilizadores poderão ver um resumo das fotografias mais recentes, das mais votadas, ou por exemplo uma foto histórica.

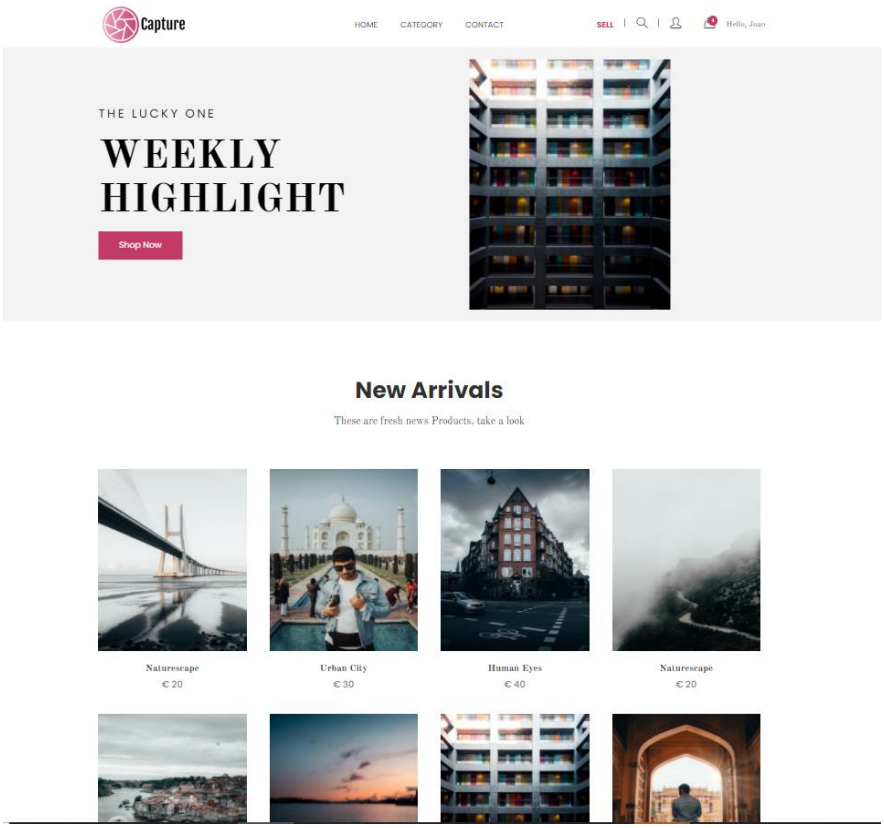

Figura 5-3 - Página inicial

b) Registo

O formulário de registo requer os campos necessários para obter a informação essencial de um utilizador da aplicação. Qualquer utilizador que queira comprar, ou vender fotografias terá de se registar e por sua vez fazer o login, para realizar essas ações.

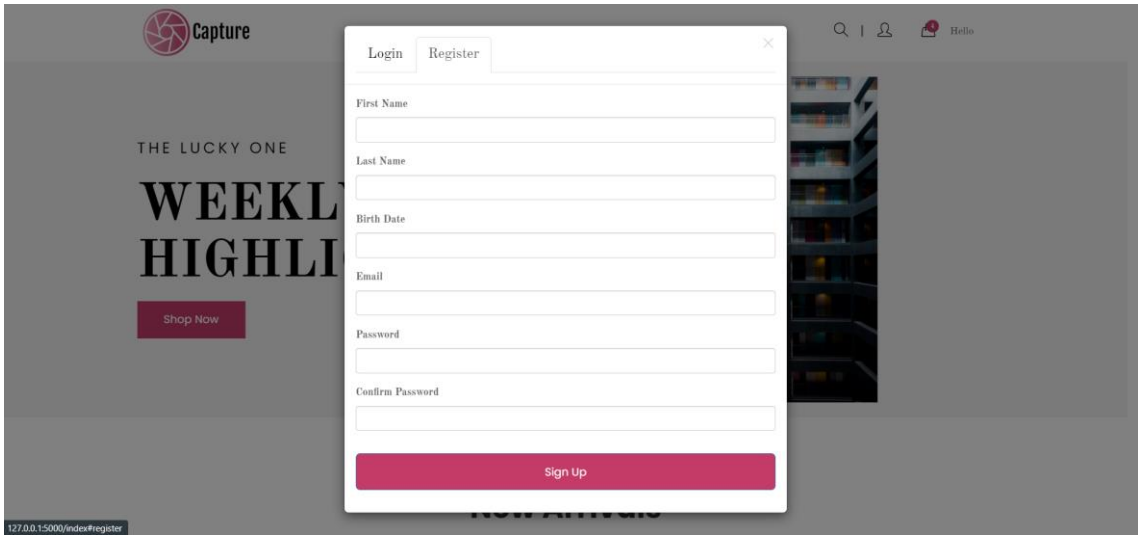

Figura 5-4 – Formulário de Registo

c) Login

O formulário de *Login* requer a inserção de um *email* (previamente registado) e de uma *password* com uma combinação já existente no sistema para que seja feita a autenticação com sucesso. Existe uma validação no formato do *email* importante para evitar sobrecargas no sistema na procura de utilizadores.

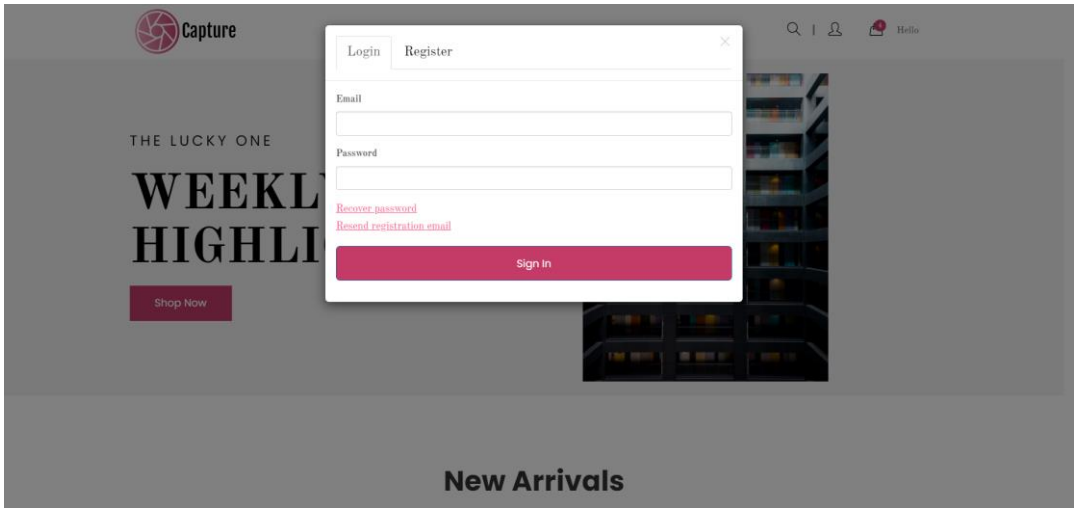

Figura 5-5 - Formulário de login

d) Visualização de vários produtos

A visualização de vários produtos em simultâneo é possível através da escolha de uma categoria e assim são apresentadas apenas as dessa categoria em específico.

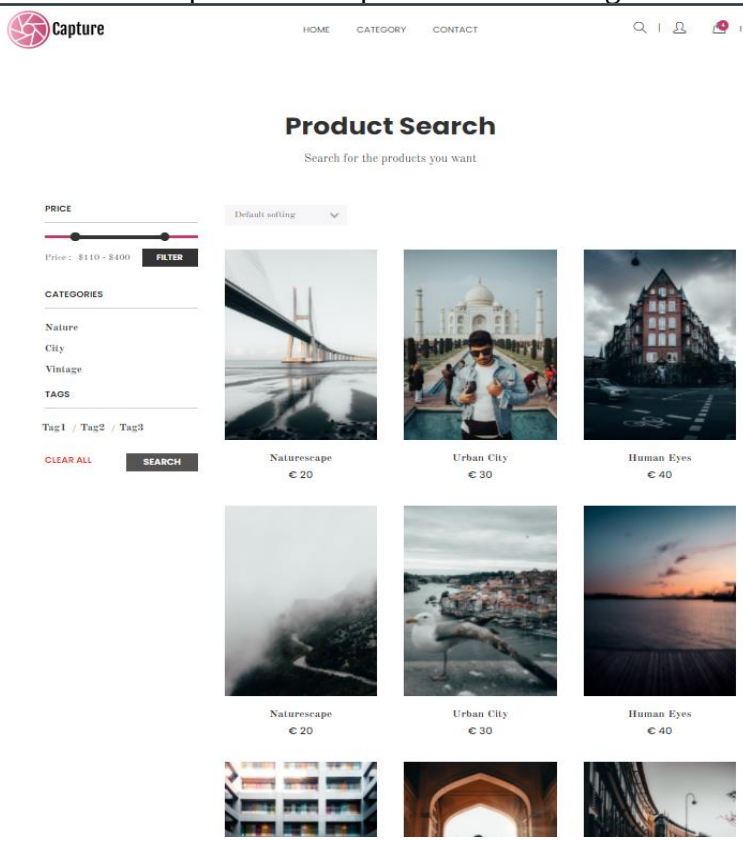

Figura 5-6 - Visualização de vários produtos e restringir por categoria

e) Visualização do detalhe de um produto

Ao carregar numa fotografia o utilizador é redirecionado para a página dessa fotografia, onde encontrará toda a descrição da mesma, desde o fotógrafo ao preço. Encontrará ainda um botão para fazer a adicionar a fotografia ao carrinho de compras.

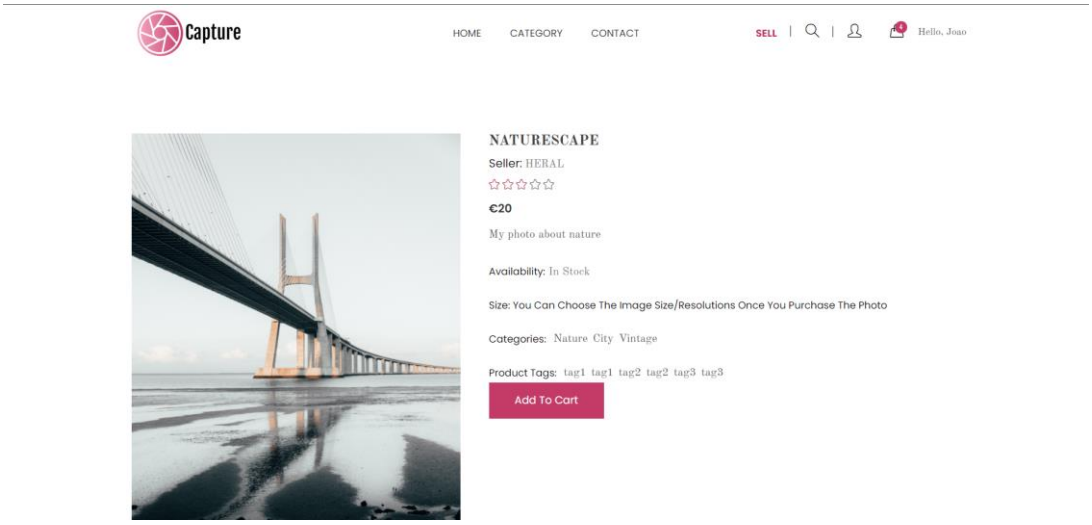

Figura 5-7 - Visualização de um produto ao detalhe

f) Venda de produto

Para vender uma fotografia o utilizador tem uma página disponível que lhe permite fazer *upload* da imagem, colocar uma descrição e o preço que pretende pela fotografia. Após esta ação a imagem ficará disponível no seu perfil numa secção de fotos para venda. Só é possível fazer esta operação caso o utilizador esteja autenticado (*Login)*.

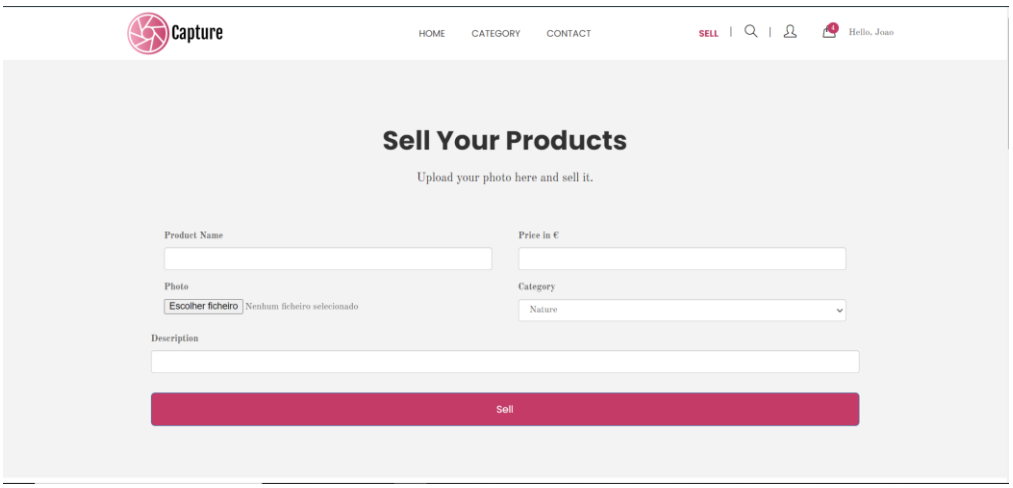

Figura 5-8 - Venda de produto

g) Carrinho de compras

O utilizador pode selecionar qualquer fotografia para guardar no carrinho de compras e posteriormente fazer uma compra conjunta.

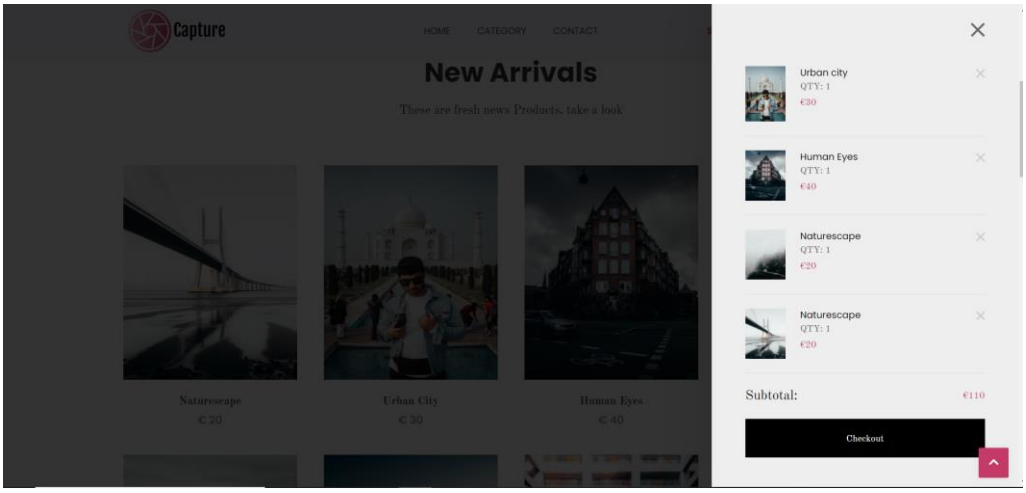

Figura 5-9 - Carrinho de compras

h) Compra de produto

Assim que a compra for realizada, este receberá a foto em *download* e ficará com ela numa secção do perfil para que possa aceder futuramente.

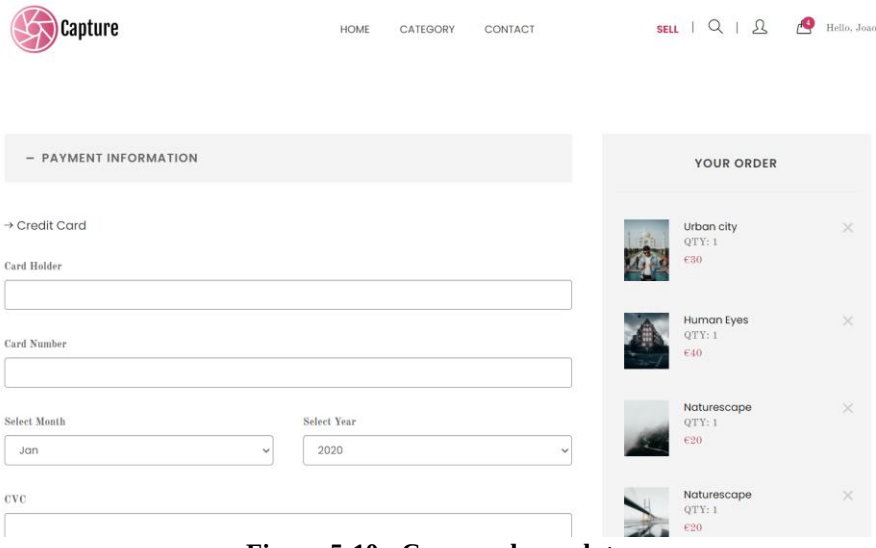

Figura 5-10 - Compra de produto

i) Perfil de utilizador

Cada utilizador tem um perfil, composto pelas informações do mesmo e ainda com uma secção onde tem as suas fotografias compradas e as disponíveis para venda.

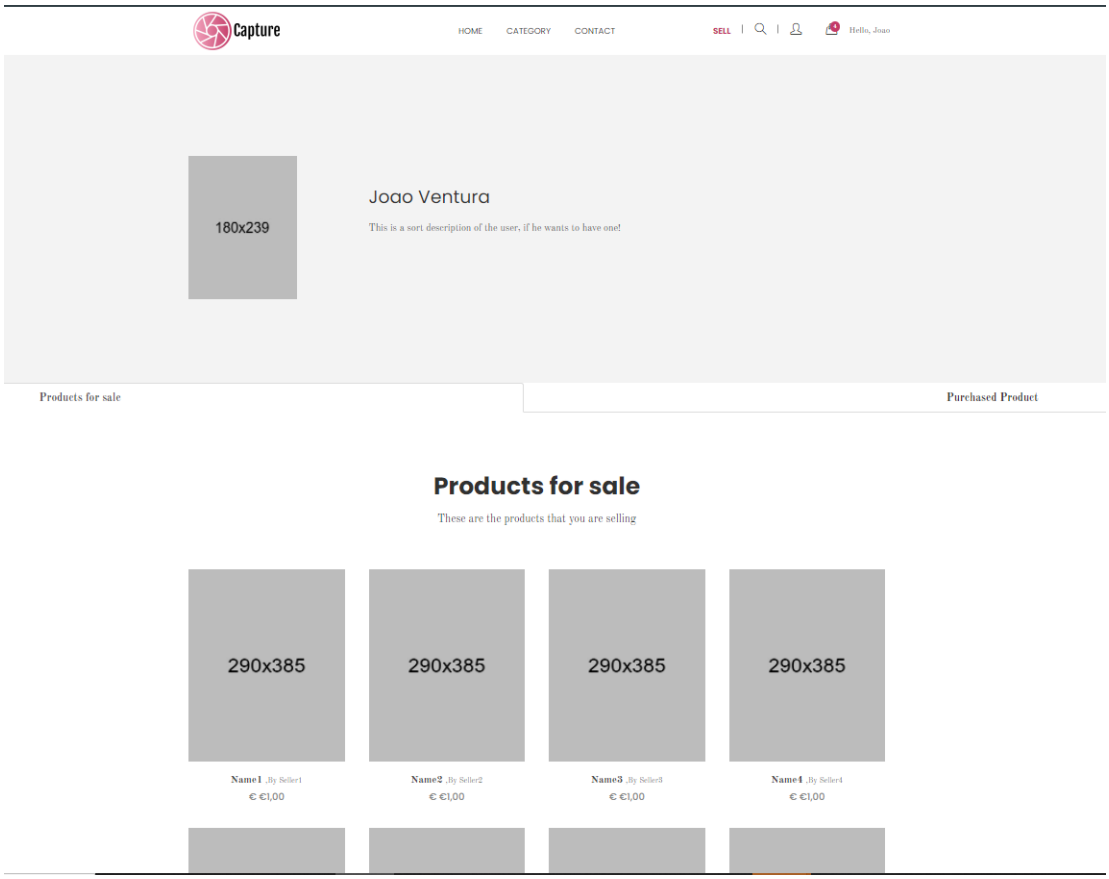

Figura 5-11 - Perfil de utilizador

5.3. Componente de controlo aplicacional (API)

Esta componente destina-se ao controlo do fluxo da componente visual e da aplicação e validação de regras de negócio. Conforme descrito no tópico de Implementação na secção 4, o conceito de *Application Program Interface* (API) permitiu construir para o caso de estudo *endpoints* (rotas) bem definidos e consistente com os Casos de utilização de modo a que no futuro em outros projetos semelhantes a este caso de estudo seja possível um reaproveitamento total ou quase total minimizando assim o tempo de desenvolvimento desta componente. A componente de controlo aplicacional é extremamente importante pois tanto comunica com a componente visual, na receção e envio de informação, como com a componente de dados para obter, eliminar, inserir ou atualizar informação.

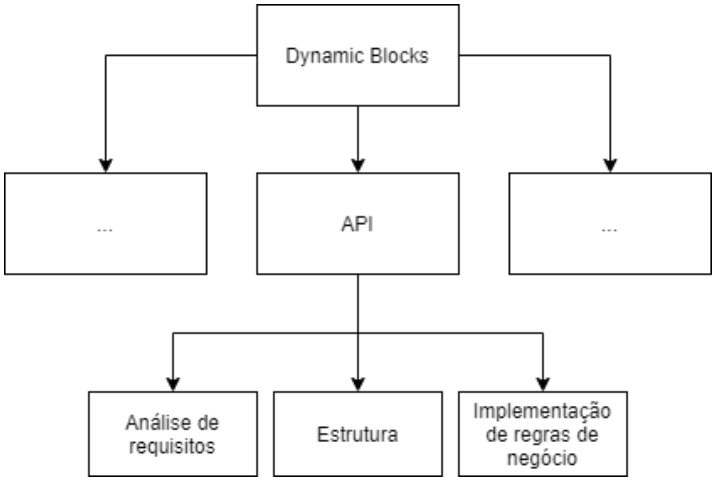

Figura 5-12 – Etapas na construção da componente de controlo aplicacional (API)

Para a construção desta componente é necessário analisar os requisitos, montar uma estrutura de *endpoints*, implementar as regras de negócio e todas as ações funcionais para a plataforma.

Os *endpoints* realizados nesta componente encontram-se descritos no tópico de Anexos.

5.4. Componente de dados

No desenvolvimento da componente de dados é necessário subdividir em diferentes etapas de acordo com a Figura 5-13.

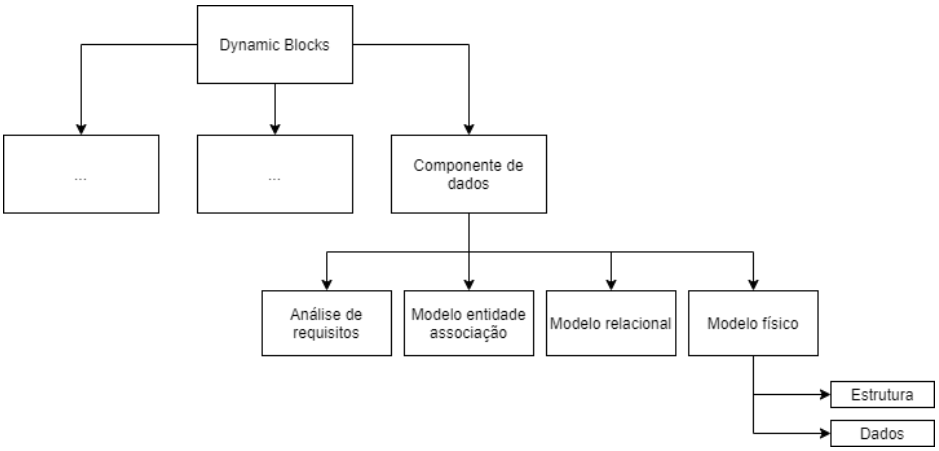

Figura 5-13 – Etapas na construção da componente de dados

A Figura 5-13 descreve as etapas de desenvolvimento do bloco da componente de dados começando com a análise de requisitos onde é feito o levantamento das regras de negócio e com isso montar uma estrutura de armazenamento de informação relacional e normalizada. Com base no tópico de Implementação na secção 4.3, a componente de dados do caso de estudo irá reaproveitar todas as tabelas gerais de uma plataforma de vendas e acrescentar algumas especificas do negócio. O diagrama de dados e os requisitos das tabelas encontram-se no tópico de Anexos na secção B.

5.5. Obstáculos

O processo de desenvolvimento deste conceito gerou muitas dúvidas na forma de configurar o *Docker* de modo a que este construísse um ambiente pronto a ser utilizado tanto para o bloco da plataforma web como para o bloco da API.

Foi ainda superado algum desconforto relativamente aos comandos *Docker*, pois para que este funcione é necessário ter algum domínio sobre os mesmos na linha de comandos da máquina, o que pode não ser muito intuitivo num primeiro contacto.

Outro problema encontrado foi o facto de em alguns sistemas operativos, mais particularmente o Windows, a criação de ambientes virtuais pedir para ativar a *flag* "Hyper-V", no entanto as novas versões do Docker já não têm este problema.

Algo que impactou uma estrutura inicial foi o facto de o Docker ser um ambiente instável para o armazenamento de dados. Numa primeira arquitetura o objetivo era ter três contentores *Docker*, um para o servidor Web, outro para o servidor aplicacional e um outro para a base de dados, permitindo assim ter ainda uma base de dados que se criaria automaticamente (a sua estrutura) num ambiente controlado. O problema foi que após algumas pesquisas e leituras percebeu-se que o *Docker*, se torna instável para armazenamento de dados em produção, pois caso o contentor por algum motivo se destrua/apague, todos os dados são perdidos. Optou-se então por colocar a base de dados num servidor fornecido pela Google (*Cloud SQL*).

5.6. Vantagens

O *Docker*, permitiu o conceito D-Blocks seguir em frente pois fornece uma agilidade no processo de *deploy* onde é mais fácil transportar uma aplicação de um servidor para o outro criando apenas uma instância de *Docker*. Inicialmente existiu uma alternativa que era a utilização de uma ferramenta chamada *Vagrant*, que permite também criar ambientes virtuais configurados automaticamente, mas que utiliza todo um *kernel* novo como por exemplo um ambiente virtual criado pelo Virtual box, fazendo com que o tempo de construção do ambiente seja mais lento, ao passo que o Docker reaproveita o *kernel* da maquina onde está alojado, diminuindo o tempo de espera na criação de um ambiente.

6. Conclusões e trabalho futuro

Pretendeu-se o desenvolvimento de um conceito (*Dynamic-Blocks*) que tem como base a modularização das componentes que constituem uma aplicação, e com vista no futuro ser possível reaproveitar partes desta na construção em outras aplicações. Para concretizar este conceito procurou-se explorar uma ferramenta de virtualização de ambientes chamada *Docker*. Foi então explorada e trabalhada de forma a puder criar uma separação entre módulos, sendo assim possível ter duas componentes que caracterizam uma aplicação, o modulo servidor aplicacional, que disponibiliza uma API para interação com a mecânica do negocio e outro modulo (servidor Web) que contem a parte visual do sistema, que permite ao utilizador interagir com a aplicação web, e onde estas interações se podem traduzir em pedidos á API do servidor aplicacional.

A realização deste conceito foi sem dúvida um grande desafio pois houve sempre um pensamento sobre a possibilidade de criar um mecanismo que tornasse a criação de aplicações mais simples, uma implementação menos trabalhosa, algo que realmente demonstrasse os avanços tecnológicos para desenvolvimento de aplicações. Perceber que o *Docker* apesar de ser uma ferramenta de virtualização de ambientes não tem as mesmas finalidades que um *VirtualBox*, *VmWare* ou mesmo *Vagrant*. Outro assunto interessante foi a utilização da linguagem de programação *Python* para a construção do servidor Web e servidor aplicacional que em nada foi parecido com o utilizado em projetos académicos de unidades curriculares.

É gratificante observar uma pesquisa a ser concluída e compreendida, e ainda ver um conceito que que já há algum tempo estava para ser desenvolvido, a ficar "materializado".

Algumas melhorias futuras poderiam partir do desenvolvimento, de novos módulos da aplicação web e API, mais modulares, e diversificados, neste caso com uma maior visão para a construção modelar do módulo em si, pois para a demonstração deste trabalho o foco foi perceber e descobrir como utilizar o *Docker*. Permitir agora modularizar cada componente permitirá criar no futuro a ideia de reutilizar módulos para qualquer tipo de aplicação, desde aplicação *web*, *mobile* e *desktop*, pois já existirá um conjunto de módulos que juntos formam uma aplicação.

Bibliografia

https://flask.palletsprojects.com/en/1.1.x/
https://pt.wikipedia.org/wiki/Flask_(framework_web)
https://flask-migrate.readthedocs.io/en/latest/
https://flask.palletsprojects.com/en/1.1.x/patterns/sqlalchemy/
https://flask-script.readthedocs.io/en/latest/
https://pythonhosted.org/Flask-Mail/
https://flask.palletsprojects.com/en/1.1.x/patterns/wtforms/
https://flask-buzz.readthedocs.io/en/latest/
https://gunicorn.org/
https://docs.gunicorn.org/en/stable/
https://getbootstrap.com/
https://github.com/
https://pt.wikipedia.org/wiki/Docker_(software)
https://www.docker.com/

Anexos

A. Documentação API postman

https://documenter.getpostman.com/view/10812175/TVK8bfXK

B. Modelo entidade associação - *Capture*

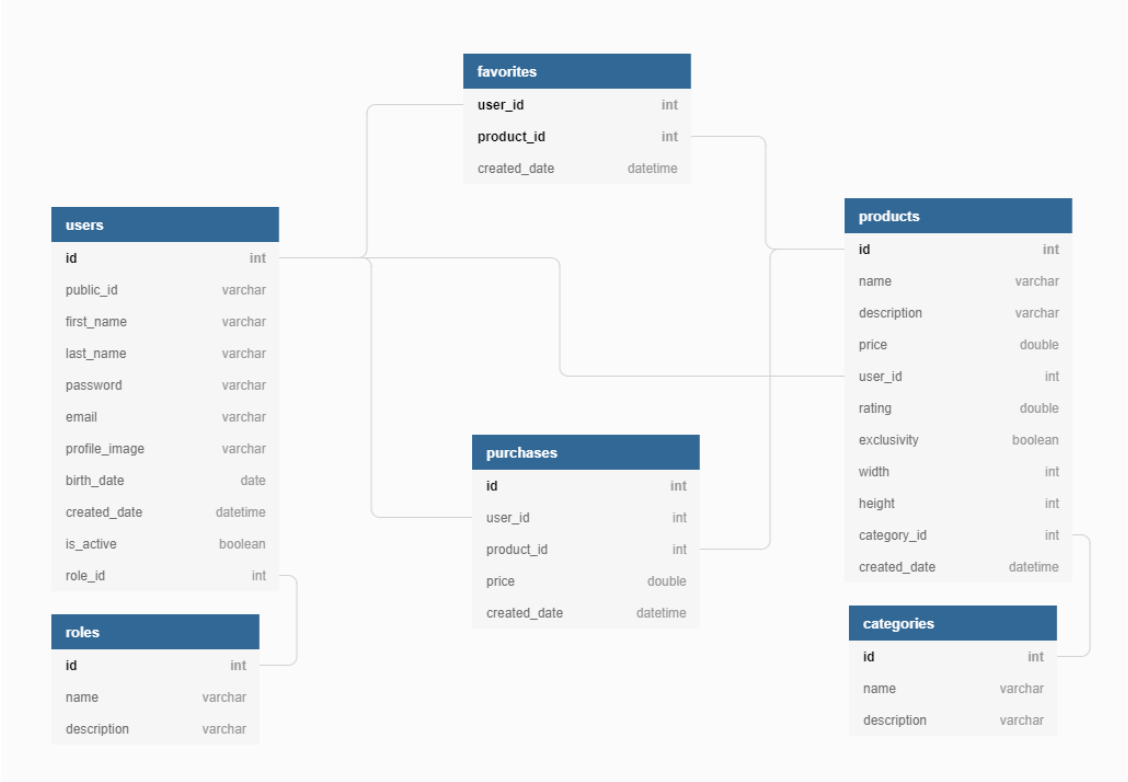

Figura 0-1 - Modelo entidade associação da componente de dados para a plataforma de vendas de fotografias *Capture*

C. Estrutura das tabelas da base de dados - Capture

C1. Roles

Tabela 5 - Definições da tabela Roles

Colunas	Tipo	Primary Key	Foreign Key	Null	Unique	Default
id	int	X				
name	varchar				X	
description	varchar			X		

C2. Users

Tabela 6 - Definições da tabela Users

Colunas	Tipo	Primary Key	Foreign Key	Null	Unique	Default
id	int	X				
public_id	varchar				X	
first_name	varchar					
last_name	varchar					
password	varchar					
email	varchar				X	
profile_image	varchar					Default.jpg
birth_date	date					
created_date	datetime					Sysdate
Is_active	boolean					
role_id	int		X (roles)			

C3. Categories

Tabela 7 - Definições da tabela Categories

Colunas	Tipo	Primary Key	Foreign Key	Null	Unique	Default
id	int	X				
name	varchar				X	
description	varchar			X		

C4. Products

Tabela 8 - Definições da tabela Products

Colunas	Tipo	Primary Key	Foreign Key	Null	Unique	Default
id	int	X				
name	varchar					
description	varchar			X		
price	numeric					
user_id	int		X (users)			
category_id	int		X (categories)			
exclusivity	boolean					
rating	numeric					0.0
width	int					
height	int					
created_date	date					Sysdate

C5. Purchases

Tabela 9 - Definições da tabela Purchases

Colunas	Tipo	Primary Key	Foreign Key	Null	Unique	Default
id	int	X				
user_id	varchar	X				
product_id	varchar	X				
price						
created_date						Sysdate

A tabela de "*Purchases*" tem uma chave composta de três elementos pois um utilizador pode comprar um produto hoje, e fazer a mesma ação amanhã, o que vai criar dois registos com *id's* diferentes. (e.g. id - 1, *user_id* - 1, *product_id* – 1; id - 2, *user_id* - 1, *product_id* – 1). Caso a chave fosse apenas o *id* de utilizador e o *id* de produto então o utilizador não poderia comprar o mesmo produto duas vezes.

C6. Favorites

Tabela 10 - Definições da tabela Favorites

Colunas	Tipo	Primary Key	Foreign Key	Null	Unique	Default
user_id	varchar	X				
product_id	varchar	X				
created_date						Sysdate

Um utilizador apenas pode colocar uma vez o produto como favorito, se o produto já estiver nos favoritos não faz sentido voltar a colocar e com uma chave composta entre *user_id* e *product_id* consegue-se restringir pois os valores não se podem repetir em simultâneo.