Instituto Politécnico de Lisboa

Instituto Superior de Engenharia de Lisboa

Engenharia Informática e de Computadores

Trabalho no âmbito da unidade curricular de Projecto e Seminário

Plataforma de Exposição de trabalhos artísticos Cri-Art

Autores:
Cristian Clefos, nº 37686

Josué Patatas, nº 42199

Bernardo Rodrigues, nº 44784

Orientador:

Prof. Luís Falcão

Lisboa, 18 de setembro de 2021

Esta página foi intencionalmente deixada em branco

Agradecimentos

Pela contínua ajuda, apoio e auxílio, queremos prestar os nossos melhores agradecimentos ao professor da unidade curricular de Projeto e Seminário Fernando Sousa, ao Professor Luís Falcão, nosso orientador, ao Professor Paulo Pereira, nosso arguente, ao nossos colegas e docentes da Licenciatura em Engenharia Informática e de Computadores, ao Instituto Superior de Engenharia de Lisboa e ao Instituto Politécnico de Lisboa e a todos os que contribuíram para a realização deste projeto.

Com os melhores cumprimentos,

Bernardo Rodrigues, Cristian Clefos, Josué Patatas

Grupo 42

Esta página foi intencionalmente deixada em branco

Lista de siglas utilizadas

CI/CD - *Continuous Integration / Continuous Delivery*

HTTP - *Hypertext Transfer Protocol*

ES - *Elasticsearch*

WS - *WebSocket*

API - *Application Programming Interface*

JVM - *Java Virtual Machine*

BD - *Base de dados*

REST - *Representational State Transfer*

URI - *Uniform Resource Identifier*

Esta página foi intencionalmente deixada em branco

Índice

Esta página foi intencionalmente deixada em branco

Introdução

Identificação do problema

Nos últimos anos temos vindo a observar constantes melhorias no que diz respeito à atualização das ferramentas de trabalho com que os profissionais lidam. A maior parte das vezes estas ferramentas facilitam em parte o trabalho que é necessário realizar por parte do profissional, quer seja uma ferramenta de contabilidade (SAP, Primavera, etc.) ou uma ferramenta de design (AutoCAD, Blender, etc.). No entanto, dado o surgimento de ferramentas profissionais, muitas vezes fornecidas pelas empresas para as quais os profissionais trabalham, pouco foi o crescimento do de formas de expor este trabalho. Note-se que os profissionais que trabalham na área da arte utilizam, ou não, tecnologias para criarem as suas obras, no entanto ainda se centram nas redes sociais (Instagram, Facebook, etc.) para poderem expor o seu trabalho. Esta solução, apesar de funcionar para certos artistas, torna-se complicada de suportar para aqueles que têm menos visibilidade, o que acaba por ser incomportável dado que quanto mais visibilidade se tem, mais visibilidade se consegue. Apesar de ser um meio de comunicação social, nos dias de hoje, as redes sociais centram-se mais no foco pessoal de cada indivíduo e não tanto no foco profissional.

Sendo assim, devia haver uma plataforma dedicada a estes artistas menos "visíveis" que, apesar de poderem tomar proveito da tecnologia para a criação dos seus trabalhos artísticos, não o fazem para a exposição das mesmas. Haverá, sim, ferramentas que permitem criar uma maior proximidade entre artistas e potenciais clientes (*i.e.* Fiverr), mas plataformas destas generalizadas criam um problema de sobre-população e oferta desmedida. Sendo assim propusemo-nos a desenvolver esta plataforma.

Idealização de uma solução

Havendo o problema supracitado, propomo-nos a criar uma plataforma que não só aproxima os profissionais do mundo da arte do seu público alvo, mas também que permita a este público alvo uma ligação mais forte com cada artista que quiser.

A solução deverá permitir que o público possa ver trabalhos de artistas das suas áreas de interesse e permita uma maior facilidade na comunicação com estes artistas. Dado que a arte não é interpretada da mesma maneira por todos, deverá ser possível partilhar opiniões publicamente acerca de uma perspectiva de uma obra de arte seja por parte de quem a criou como de quem a vê. Deverá também facilitar o acesso a estes trabalhos específicos bem como outros trabalhos com base no tipo de arte que se pretende.

Um artista que crie uma obra nova, deverá poder partilhá-la com o público e este deverá poder aceder aos trabalhos que um determinado artista criou. O público deverá também ter uma forma de ver os novos trabalhos que um determinado grupo de artistas partilha, sem ter de ir individualmente procurá-los.

Sendo uma plataforma que permite uma maior proximidade entre os artistas e o público, deverá ser possível para o público contactar o artista, para pedir uma peça, ou requisitar um trabalho. Sendo assim, deverá haver uma forma de manter uma conversa pessoal entre dois utilizadores, desde que um destes seja um artista.

Requisitos do sistema

Tendo a solução idealizada em mente, há então que concretizar os requisitos da plataforma. Notamos então que é necessário que um artista possa expor um trabalho e ter um sítio específico onde todos podem ver os trabalhos expostos e toda a informação que o artista escolhe apresentar. Então deverá haver um perfil individual para cada artista para que possam ter estas informações. Num acesso a esta página por parte de um utilizador comum, deverá haver a opção de ver individualmente cada trabalho bem como deixar comentários e/ou gosto.

Deverá ser possível também um utilizador seguir um artista para estar a par das novas exposições do artista. Um utilizador terá então uma lista de artistas que segue para poder fazer a gestão dessa lista e terá uma página onde poderá ver as atualizações dos artistas sem ter de as procurar.

Haverá uma distinção entre utilizadores comuns (clientes) e utilizadores que pretendam expor trabalhos (artistas), na medida em que um artista terá um perfil e um local onde expor trabalhos e um cliente não o terá pois não é o objetivo deste utilizador a exposição mas sim a visualização. No entanto, as funcionalidades de um cliente não lhe deverão ser exclusivas dado que um artista não deve ter de criar uma conta secundária caso queira utilizar a aplicação da perspectiva de um cliente.

A procura de trabalhos e artistas deverá ser o mais facilitada possível para que a plataforma seja *user friendly* (de uso amigável). Por isso um utilizador deverá poder pesquisar por artistas bem como por trabalhos específicos. No caso de se querer ver uma área de trabalho específica, deverá também ser possível pesquisar por área de trabalho. Esta pesquisa, bem como a visualização de perfil e trabalhos, deverá estar aberta a todas as pessoas.

Para um artista, é importante que nada no seu perfil seja definitivo, por isso cada artista deverá poder alterar os detalhes que decidir (Nome de utilizador, descrição do perfil bem como dos trabalhos, áreas de trabalho, etc.)

Já para um utilizador comum, será importante que as decisões que toma no que concerne artistas (gostar de um trabalho, comentar um trabalho, seguir um artista) também sejam sempre reversíveis. Sendo assim, deverá haver uma liberdade sobre o que os utilizadores decidem fazer.

Construção técnica dos requisitos

Tendo em conta os requisitos do sistema, seguimos então para a concretização mais técnica desses requisitos.

A plataforma irá consistir de um *website*. Este website deverá ter uma componente pública, como a visualização de perfis de artistas ou qualquer tipo de pesquisa, bem como uma componente privada como as ações de comentar ou de gostar de uma publicação.

Haverá uma página principal que servirá para a pesquisa de artista, trabalhos ou por áreas de trabalho, bem como para a apresentação de um *Feed* onde constarão as publicações mais recentes dos artistas que um utilizador segue. Este *Feed* apenas estará disponível para utilizadores que estejam devidamente autenticados.

Haverá páginas de *login* e *sign up* para que um utilizador se possa registar e possa ligar-se à sua conta e haverá uma página dedicada a cada perfil. Haverá também uma página de pesquisa, alcançada através de um botão de pesquisa.

Tendo em conta que o *Feed* é privado, será lá que se encontra o centro das suas mensagens.

Planeamento da implementação

Para a criação desta plataforma, será necessário implementar uma metodologia de trabalho e desenhar um plano para que haja uma forma de manter a organização e impedir atrasos desmedidos. Sendo assim, para a metodologia de implementação, decidiu-se implementar uma plataforma base completa (*Database, Backend* e *Frontend*) e iterativamente ir implementando as funcionalidades de forma a que não fosse comprometido o funcionamento geral da aplicação. Sendo assim desenhou-se um planeamento geral onde se encontra o tempo destacado para o desenvolvimento de determinada funcionalidade. (Anexo 1)

Para as tecnologias em que se escolheu adotar, após a discussão entre os membros do grupo e o orientador, decidiu-se utilizar *Spring Framework* utilizando a linguagem *Kotlin* para o implementar o servidor de recursos, *Elasticsearch* como base de dados (BD) não relacional e, para a aplicação cliente, utilizar em *Typescript*, tirando partido da biblioteca *React*.

Ainda mais, decidiu-se implementar também uma solução *cloud*. Para isso escolheu-se a plataforma *heroku* para hospedar as aplicações tanto servidora como cliente dada a vasta documentação e o livre acesso, ainda que com recursos limitados. O modelo da aplicação fica então como na figura abaixo:

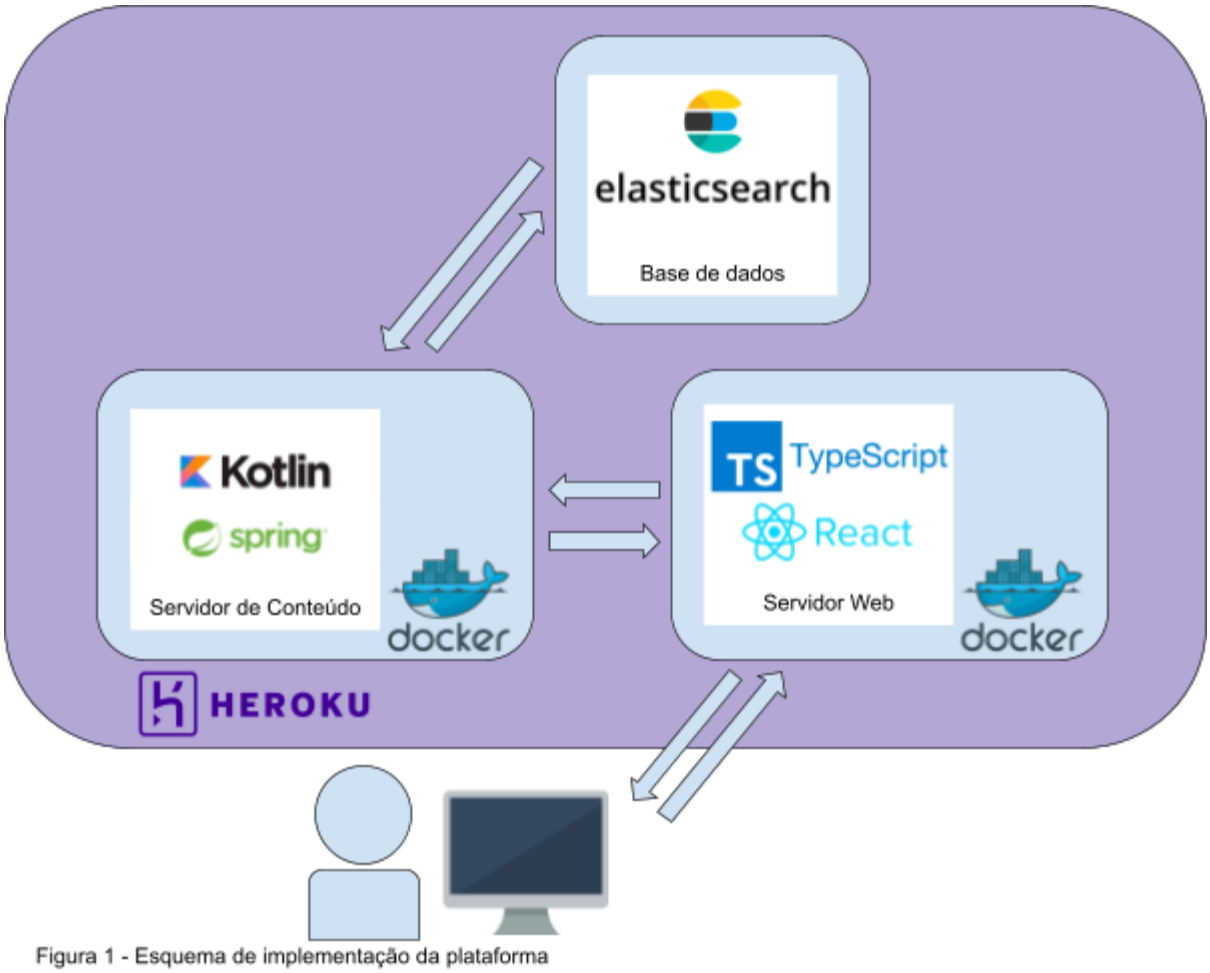

Figura 1 - Esquema de implementação da plataforma

Dado que o projeto não seria apenas alojado localmente mas sim também no *heroku*, decidiu-se utilizar um mecanismo de *Continuous Integration / Continuous Delivery (CI/CD)* para otimizar todo o processo de *deployment* da aplicação para a *cloud*. Este mecanismo foi feito através de *Github Actions.*

A decisão de utilização de *Spring framework* veio da necessidade de mercado de programadores que sejam experientes com esta ferramenta. Dado que a ferramenta poderá servir para uma componente definida (servidor de recursos) seria uma mais valia a utilização da mesma. Visto que o número de programadores em *Kotlin* tem vindo a aumentar, decidiu-se utilizar esta linguagem de programação com o *Spring framework* no âmbito do *backend.*

Já no âmbito da aplicação cliente, decidiu-se utilizar *React* por ser uma das ferramentas mais comuns de desenho de aplicações *frontend*, pela sua simplicidade e pela quantidade de suporte e apoio disponível *online*. Para trabalhar com esta biblioteca decidiu-se utilizar a linguagem de programação *Typescript* por ser baseada em *Javascript* mas ter uma natureza mais declarativa e ser fortemente tipificada.

Por fim, para a base de dados, estudou-se, inicialmente, a possibilidade de utilizar uma base de dados relacional, tendo em mente o *PostgreSQL*, mas após uma reunião com o Professor Luís Falcão, tornou-se clara a transição para uma base de dados *NoSQL* neste caso o *Elasticsearch (ES),* por conhecermos a tecnologia por ter sido utilizada na unidade curricular de Programação na Internet (PI).

Desenvolvimento

Familiarização com as tecnologias

Dado que as tecnologias escolhidas não eram conhecidas por todos os elementos do grupo, decidiu-se tomar um tempo de familiarização das tecnologias. Através das aulas públicas de "Desenvolvimento de Aplicações *Web*" do Professor Paulo Pereira, dado que as tecnologias utilizadas neste projeto são as mesmas que foram lecionadas na unidade curricular, foram um apoio forte a esta aprendizagem, maioritariamente no que toca à utilização da *framework Spring*. A tecnologia usada no *frontend*: *React Framework* era desconhecida para todos os elementos do grupo.

Conteinerização e Orquestração

O desenvolvimento moderno de *software* favorece o uso de *containers Docker* em vez de máquinas virtuais remotas para garantir que o *software* tenha o mesmo comportamento na máquina de alojamento remoto que na máquina do desenvolvedor.

Decidimos separar o *frontend* e o *backend* em 2 *containers,* pela regra de *separation of concern*: quando houver alterações no *frontend* não é necessário recompilar o *backend* e vice-versa.

Contêiner frontend

O *contêiner frontend* isola um servidor *Nginx* que tem 2 responsabilidades:

- Servir conteúdo estático que constitui o *frontend* compilado.
- Encaminhar os pedidos que tem a *path* que começa com *"/api"* para o servidor *backend* e vice-versa, basicamente fazendo papel de *reverse-proxy*. Esta solução permite minimizar os erros *CORS*.

Container backend

O *container backend* isola um servidor *Tomcat* com a aplicação *Spring Boot*.

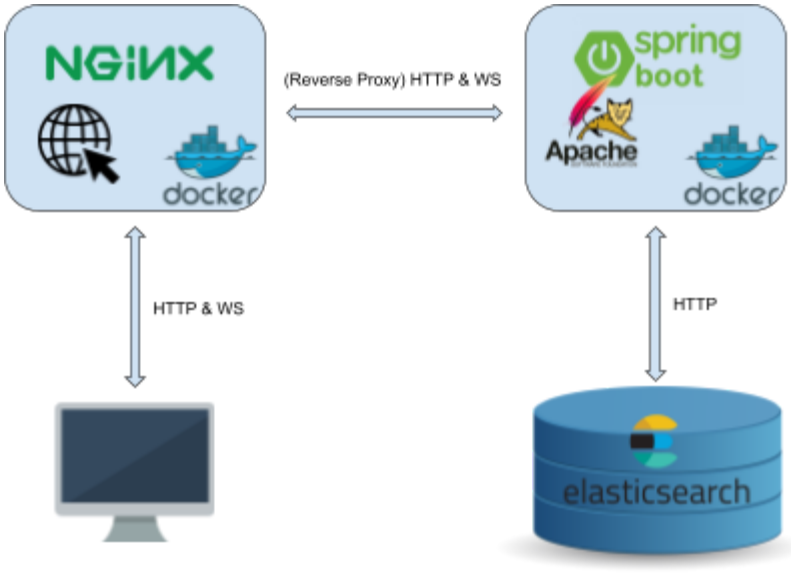

Figura 2 - Orquestração dos vários componentes da aplicação

Compatibilidade com Heroku [7]

Para serem compatíveis com a plataforma *Heroku* os *contêineres* devem ter a opção de aceitar a variável de ambiente *PORT* e escutarem o mesmo quando forem lançados.

Alojamento e Armazenamento na *Cloud*

O alojamento da aplicação é feito na plataforma *cloud Heroku* usando *Dynos,* contentores *Linux* virtualizados, do tipo *free*, que são gratuitos. Cada *Dyno* gratuito suporta um *container Docker*, logo, para o alojamento da nossa aplicação, dado que o *frontend* e o *backend* são *conteinerizados* à parte, precisamos de 2 *Dynos* para as respectivas partes. O 3º e último *Dyno* é necessário para a base de dados *Elasticsearch*, cujo armazenamento persistente também é fornecido pelo *Heroku* recorrendo a um *add-on* disponível na plataforma.

Desafios do *Continuous Integration, Continuous Deployment* (*CICD* [6])

Os desafios de entrega contínua surgem a partir da necessidade de automatizar a compilação e a instalação da aplicação no seu alojamento remoto cada vez que são submetidas alterações no código no repositório *Github* remoto. Isto permite não só poupar tempo e garantir entrega contínua, mas também elimina o erro humano no processo da submissão das novas versões.

Soluções para os problemas de *CICD*

A solução *CICD* adotada para esse projeto foi através de *Github Actions* [5]. As *Github Actions* são *pipelines* executadas automaticamente cada vez que há alterações no código.

Para este projeto foi necessário criar duas *pipelines*: uma para o *frontend* e outra para o *backend*. Isso permite que, por exemplo, no caso de haver alterações só no *frontend* a *pipeline* do *backend* não será executada.

Ambas as *pipelines* podem ser divididas em duas partes:

- Compilar o código e criar a imagem *Docker.*
- Enviar a imagem *Docker* para a plataforma *Heroku.*

Desenvolvimento de código

Numa primeira fase de implementação, começou-se por criar o projeto da componente servidora. Construiu-se o projeto de forma a que pudesse guardar e devolver dados através da utilização do protocolo de *Hypertext Transfer Protocol* (*HTTP*).

Para este efeito desenhou-se o esquema de funcionamento da *Application Programming Interface* (*API*). Utilizou-se a ferramenta *Swagger* para uma mais fácil documentação da *API,* definindo os vários pedidos *HTTP* que se poderão fazer à *API* (Anexo 2). Este documento pode ser alterado consoante as necessidades da aplicação.

Arquitetura Três Camadas

A arquitetura que adoptamos é uma das mais comuns, sendo composta pelas seguintes camadas:

- Camada dos *Controllers*. Camada responsável por atender pedidos HTTP e resolvê-los com base na camada dos Serviços.
- Camada dos Serviços. Camada responsável por aplicar as regras de negócio. Consome a camada de Dados.
- Camada de Dados. Camada que serve para aceder aos dados.

Construção base dos *controllers* e dos *handlers*

Sendo uma *API* que usa como meio de comunicação o protocolo *HTTP*, deverá ter mecanismos responsáveis pelo tratamento destes pedidos. Para isso, a *framework Spring* permite a anotação de classes de *@RestController* para dotarem os componentes dessas classes de propriedades *REST* e poderem mapear pedidos *HTTP* para o seu respectivo *handler.* A partir da versão 4.0 da *framework Spring,* esta anotação foi introduzida facilitando o processo de criação de *handlers*, eliminando a necessidade de os anotar individualmente com *@ResponseBody*. Desta forma, todos os *handlers* presentes nesta classe, mapeiam o objeto de retorno numa resposta do tipo *HttpResponse*. (1)

Dada a construção das classes *Controller*, segue-se a implementação dos *handlers*, responsáveis pelo tratamento de pedidos *HTTP* para um caminho específico. É necessário criar um *handler* para cada caminho diferente da *API*, para que pedidos diferentes possam ser tratados de forma diferente.

Para a construção dos *Controller*s que incluíssem os *handlers* de tratamento de pedidos discriminaram-se os pedidos com base no recurso ou tipo de recurso a que pretende aceder, alterar, remover ou criar. Sendo assim, criaram-se *controllers* responsáveis pelo tratamento de pedidos afectos a artistas, e a trabalhos de artistas.

Para que a implementação do sistema de comunicação com a BD e a implementação destes *controllers* não estivessem no mesmo sítio, criaram-se classes de serviço responsáveis por atender aos pedidos dos *handlers* e devolver-lhes o conteúdo apropriado, sendo esta camada de serviços a responsável pelas regras de negócio, sem que o *handler* tenha de estar responsável por tudo isso, criando uma vasta desorganização no código e dificultando o processo de *debug* e a própria leitura de código. Estas classes não estão responsáveis pelo acesso aos recursos mas sim pelo tratamento dos dados recebidos nos pedidos aos *handlers* e que são necessários para concretizar o acesso aos recursos.

Artist Controller

A classe *ArtistController* será responsável pelo tratamento de pedidos afetos a um artista. O caminho genérico deste *Controller*, definido usando a anotação *@RequestMapping*, é *[caminho da API]/artist*. \

Nesta classe encontramos métodos (*handler methods)* que permitem:

- Obter as informações de um artista específico, recebendo apenas como parâmetro o *id* deste artista nas variáveis do caminho (*Path variables)* com o nome *aid.* O pedido para este *handler* será, então, GET [caminho da *API]/artist/{aid}*
- Criar um artista. Para este pedido é necessária a informação associada ao perfil que será disponibilizado, sendo esta constituída por: nome do artista, descrição do perfil do artista, email, o identificador do artista e uma lista de *tags* (áreas de trabalho em que o artista

trabalha). Para se realizar este pedido é necessário fazer um pedido POST [caminho da *API*]/artist e deverá ter como corpo do pedido as informações supracitadas.

- Seguir um artista, para poder ver as novas publicações do mesmo sem ter de navegar até à página deste artista. Para se realizar este pedido, é necessário fazer um pedido *HTTP* de PUT *[caminho da API]/artist/{aid}/follow*, não precisando de corpo da mensagem, já que os atributos de utilizador são adicionados pelo filtro de autenticação. Este método foi feito aquando da implementação do componente de cliente.

Artist Tag Controller

Nesta classe encontram-se os *handlers* que concernem à manipulação da lista de tags de um artista. O caminho genérico para este *Controller* é [caminho da *API*]*/artist/{aid}/tags.* Através dos métodos desta classe, será possível:

- Adicionar *tags* representativas de áreas de trabalho ao perfil de um artista. Para realizar esta função, será necessário introduzir na variável do caminho *aid* o identificador do artista bem como enviar no corpo do pedido o nome da *tag* a ser adicionada. Uma mensagem de pedido de POST [caminho da *API*]/artist/{aid}/tags com a *tag* desejada no corpo da mensagem irá executar este método.
- Remover *tags* representativas de áreas de trabalho ao perfil de um artista. Esta função tem um funcionamento semelhante à descrita no ponto anterior, mas o método *HTTP* realizado deverá ser DELETE e não POST.

Work Controller

A esta classe estão associados todos os métodos de *handling* que interajam com trabalhos de artistas. O caminho genérico utilizado pelas funções de tratamento de pedidos *HTTP* é */artist/{aid}/worksofart*. Neste *Controller* encontram-se os métodos que permitem:

- Obter a lista de trabalhos de um utilizador específico. Este método serve para quando se abrir o perfil de um artista se possa ver todos os trabalhos que este publicou. Para isso, será necessário passar na variável do caminho genérico *aid* o identificador do artista, quando se faz um pedido *HTTP GET [caminho da API]/artist/{aid}/worksofart*.
- Adicionar um trabalho. Esta função estará disponível apenas para o próprio artista e permitir-lhe-á adicionar um trabalho novo ao conjunto dos trabalhos que apresenta. Sendo assim este *handler* necessita então de receber o identificador do artista, passado no caminho do pedido, o conteúdo do trabalho, como MultipartFile, o nome do trabalho e a descrição desse mesmo trabalho. É necessário, portanto, realizar um pedido de *POST [caminho da API]/artist/{aid}/worksofart*, com o conteúdo acima descrito no corpo do pedido.
- Adicionar um comentário ao trabalho. Uma das formas de interação entre os utilizadores e os artistas é que os primeiros podem deixar comentários nas publicações dos artistas. Para tal, é necessário que este pedido receba o identificador do artista bem como o identificador do trabalho e o comentário. Ambos os identificadores serão enviados através do caminho do pedido, já o comentário é enviado no corpo do mesmo. Um pedido para este *handler* seguirá a forma *POST [caminho da API]/artist/{aid}/worksofart/{wid}* onde *aid* e *wid* são os identificadores de artista e trabalho, respetivamente. Este método foi feito aquando da implementação do componente de cliente.
- Dar ou retirar um *upvote*. Da mesma forma que o ponto anterior, outra forma de interagir com os artistas será dar um *upvote* a um trabalho deste, sendo um *upvote* uma forma de representar que se gosta do trabalho em questão. Neste caso, o mesmo método é

responsável por dar ou retirar um *upvote*, deixando se o identificador do utilizador não constar na lista de *ups* do trabalho ou retirando se constar. Para realizar esta funcionalidade é necessário, então, fazer um pedido de *PUT [caminho da API]/artist/{aid}/worksofart/{wid}*. Este método foi feito aquando da implementação do componente de cliente.

Construção do modelo de comunicação com a BD

Para se fazer a comunicação entre aplicação servidora e base de dados, visto que as tecnologias usadas são *Spring framework* e *ES* respectivamente, utilizou-se a biblioteca *Spring Data Elasticsearch* que permite a uma comunicação mais facilitada entre estes dois componentes da aplicação. A forma de implementação desta biblioteca baseia-se na definição das estruturas que irão ficar guardadas na base de dados e na implementação de interfaces de comunicação com a BD utilizando essas estruturas.

Foram, então, definidas as estruturas a guardar na BD. Dado que a aplicação estava ainda num processo de construção, foram criadas as estruturas de dados representantes de um artista e de um trabalho, com os atributos necessários para a caracterização dos mesmos. Um artista é identificado por um atributo *artist_id* único e é caracterizado por ter um nome de utilizador, uma descrição do próprio artista e uma lista de áreas de trabalho, denominada de *tags*. Um trabalho de um artista será também identificado por um atributo *id* único e caracterizado por ter um nome, uma descrição, um dono, atributo que remete para o identificador do artista que adicionou esse trabalho, uma lista de comentários, uma lista de "*upvotes"*, constituída pelos identificadores dos utilizadores que gostaram do trabalho, e uma data de publicação.

Nesta fase inicial os pedidos *HTTP* realizados foram todos feitos a partir da ferramenta *Postman*, dado que permite um fácil manuseamento tanto dos pedidos à *API* como dos pedidos diretamente à BD.

Para implementar a comunicação com a BD foi necessário implementar uma interface, anotada com *@Repository* por cada índice presente na BD. Neste momento esses índices eram apenas *artists* onde estão guardados os dados dos artistas e *works* onde estão guardados os trabalhos, mas posteriormente foram acrescentados os índices *user*, *token* e *message.*

Implementação de um sistema de autenticação

Com o objetivo de dotar a aplicação de validação de utilizadores foi então criado um sistema de registo e autenticação para que os utilizadores possam criar contas de utilização, escolhendo se se querem registar como artistas, podendo publicar os seus trabalhos e ter um perfil próprio, ou como clientes, podendo interagir com os artistas mas sem o objetivo de expor trabalhos próprios.

Para este efeito, implementou-se um sistema de autenticação baseado em *tokens* de sessão. Um utilizador, ao inserir as credenciais de autenticação na plataforma, é-lhe atribuído um *token* de sessão. Este *token* será utilizado para manter a sessão do utilizador bem como para este poder ser identificado. No processo de registo na plataforma, é necessário que o utilizador confirme a sua tentativa. Para isso necessitará de aceder ao *email* introduzido aquando da criação da conta, abrir a mensagem enviada pelo *email* *criartserviceacc@gmail.com* e aceder ao *link* presente. Isto irá realizar o pedido de verificação da conta que dirá que o utilizador está pronto a utilizar a plataforma.

Palavras-passe

As palavras-passe dos utilizadores são guardadas de forma encriptada usando o algoritmo *SHA-256*, possibilitando assim verificar se coincide com as palavras-passe submetidas nas tentativas de autenticação sem ser necessário decifrar a palavra-passe do utilizador da base de dados.

Descrição do sistema de autenticação

Este sistema baseia-se num filtro de autenticação, implementado ao nível da aplicação servidora. Neste filtro, aos pedidos que contenham um parâmetro *token* válido, adiciona um atributo *user* contendo as informações do utilizador responsável pelo pedido, caso contrário bloqueia o pedido e retorna uma resposta *HTTP 401 Unauthorized*.

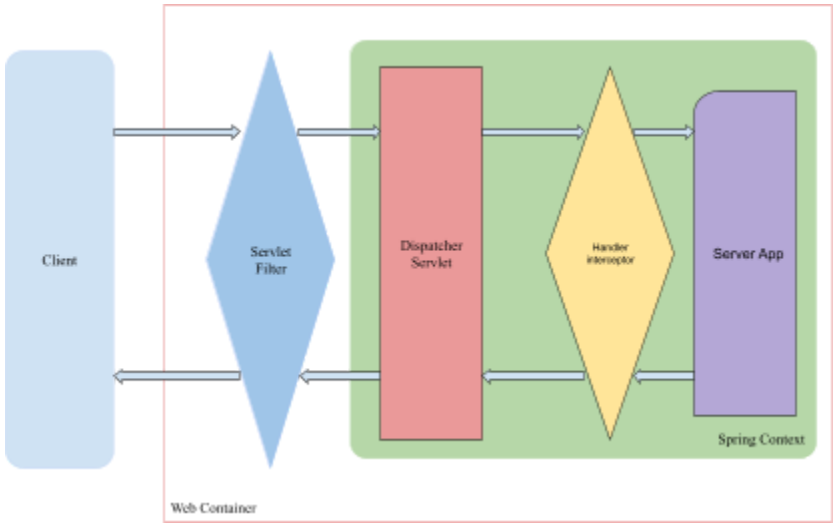

Figura 3 - Esquema de funcionamento de filtros e interceptores na *framework Spring*

Para se conseguir isto, adicionaram-se dois novos índices à BD. O índice *user*, no qual se guarda a informação dos utilizadores autenticados, e o índice *token*, onde constam os *tokens* de sessão. No processo de *login* um utilizador apresenta as suas credenciais e, caso sejam concordantes com as credenciais guardadas, é-lhe atribuído um *token* de sessão. Esse *token* será o identificador do utilizador durante a sua utilização da *API*. Este *login* é feito através de um pedido *HTTP POST* para o caminho *[caminho da API]/auth/login*, necessitando de enviar o *email* e a *password* no corpo do pedido.

Para que um utilizador possa ter as suas credenciais guardadas, necessita de passar pelo processo de *sign up*. Para tal, é preciso fazer um pedido de *POST [caminho da API]/auth/signup* cujo corpo deverá conter o *username* pretendido, o *email* de registo e a *password* desejada.

Implementação de um sistema de *Logging*

Havendo a possibilidade de fazer pedidos à aplicação servidora, e com o sistema de autenticação implementado, surgiu a necessidade de fazer *log* das mensagens de pedido e de resposta que chegassem e saíssem da *API*. Sendo assim, implementou-se um *handler interceptor* para imprimir na consola as mensagens de pedido e de resposta afetas à aplicação. Dado que o conteúdo do corpo da mensagem não pode ser lido mais do que uma vez [(2)], estas mensagens de *log* imprimem o método *HTTP* feito e o *URI* para o qual este foi feito.

Mensagens assíncronas

A aplicação suporta envio e recepção assíncrona de mensagens entre utilizadores devido à utilização de *websockets*. Os *websockets*, apesar de serem uma tecnologia do mesmo nível que *HTTP*, seguem um paradigma diferente: em vez de seguir a abordagem "pedido-resposta", mantém-se um canal de comunicação *full-duplex* aberto, possibilitando o envio de mensagens que não foram explicitamente pedidas pelo cliente, sem recorrer à *polling* de pedidos *HTTP*.

Organização do código

Concluída a implementação do sistema de autenticação foi necessária uma reorganização do código para possibilitar o acesso mais facilitado aos vários ficheiros e para possibilitar uma leitura e identificação de problemas mais acessível. Criaram-se então pastas cujo conteúdo integra-se em *Controllers*, serviços, acesso a dados, configurações, modelos de dados e filtros e interceptores. Dentro da pasta que diz respeito aos *Controllers*, foram ainda criadas pastas que contêm os *controllers* e os *data transfer objects* (*dto)* relacionados.

Front-end: Início da construção

A componente de cliente da aplicação, como mencionado anteriormente, foi implementada tomando partido da biblioteca *React*. Para este efeito criou-se um projeto *React* através do *Node Package Manager* que disponibiliza a base estrutural sobre a qual se pode construir o projeto.

Criação da estrutura e organização

Os componentes criados para a aplicação de *front-end* foram implementados utilizando componentes funcionais. Para tal, tomou-se partido de *hooks*, funções que dotam componentes funcionais de propriedades de componentes-classe. Com o foco de organizar a estrutura e implementar uma base de trabalho, organiza-se o código nas seguintes estruturas:

- Classe *Api:* Esta classe é a responsável pela comunicação com a aplicação servidora, implementando um método que realiza os pedidos *HTTP* e retorna o conteúdo enviado na resposta.
- Componente *App*: Esta classe é criada automaticamente na criação do projeto *react*. Esta classe apenas devolve um componente *AppRouter* que determina a estrutura e a navegação entre os vários componentes da aplicação
- Componente *Header:* Este componente acompanha o utilizador durante toda a navegação na aplicação, podendo sofrer alterações consoante a utilização.

Por fim, o código foi ordenado em pastas, tendo cada pasta apenas conteúdo inter- relacionado.

Definição do *Design*

Dado que esta plataforma é direcionada ao público, é necessária a definição de uma linguagem visual, para que haja consistência visual durante a experiência de utilização da plataforma. Para isso é necessário que sejam utilizados elementos visuais estilizados, com animações ou efeitos e que as cores e combinações de cores das páginas sejam consistentes em toda a plataforma.

Com este objectivo em mente, decidiu-se utilizar uma biblioteca gráfica e criar um logótipo. Escolheu-se como cor identificativa da plataforma a cor-de-laranja pois, segundo psicologia das cores, é uma cor que transmite criatividade e otimismo e é uma cor muito associada às artes e a cor azul, pois é uma cor que transmite confiança e segurança interior [(3)] é uma cor que, visualmente, dá um contraste agradável com o cor-de-laranja.

Bootstrap

Dado que se decidiu utilizar uma biblioteca gráfica, optou-se pela utilização de *Bootstrap*. Esta biblioteca disponibiliza uma *stylesheet* e apresenta, na própria documentação, vários elementos que podem ser copiados diretamente para o código da página, sendo apresentados logo com o estilo definido e apresentado na página. É também uma biblioteca *open-source*, o que permite a sua utilização livre de custos.

Logótipo

Para o logótipo, decidiu-se criar uma imagem que remetesse tanto para o website em si, como para a plataforma. Decidiu-se dar o nome de *Cri-Art* pois é uma junção das palavras "Criar" e "Art" (arte em inglês). Deu-se as cores definidas anteriormente para a plataforma ao logótipo. Para a criação deste logótipo foi utilizada a ferramenta online *Vectr* [(4)].

Figura 4 - Logótipo da aplicação

Construção da *Home Page*

Esta página será a página inicial de acesso a um utilizador. Caso o utilizador não esteja autenticado, é apresentada uma barra de pesquisa para que esse utilizador possa ver conteúdo. Já no caso de o utilizador estar autenticado, é aqui que são apresentadas as publicações, feitas na última semana, dos artistas que o utilizador segue.

Home Controller

Anteriormente à construção deste componente, foi implementado, na aplicação servidora, os *handlers* que servem conteúdo a esta página. A esta classe *Controller* está afeto todo o pedido que seja necessário fazer quando se está na página inicial da plataforma. Também nesta classe, encontram-se todas as operações explicitamente públicas. Através dos métodos presentes nesta classe, será possível:

- Requisitar a página principal, quando o utilizador está autenticado, para obter as publicações feitas, na última semana, pelos artistas que um utilizador segue. Para isto é necessário ser realizado um pedido *HTTP* GET [caminho da *API*]/feed. Neste pedido não é necessário acrescentar mais nada, dado que as informações necessárias são as informações do utilizador, que são acrescentadas ao pedido pelo filtro de autenticação.
- Pesquisar artistas por nome. Para esta funcionalidade, será necessário enviar o nome pelo qual se procura. Esta informação será enviada nos parâmetros do pedido através da variável *nameToSearchBy.* Um pedido para este *endpoint* será então feito com um pedido *HTTP GET* [caminho da *API]/public/home/search.*
- À semelhança do ponto anterior, também é possível realizar pesquisas com base nas *tags* que um artista possui. Para que isto seja possível, é necessário enviar, nos parâmetros do pedido, uma variável *tagToSearchBy* com o nome da *tag* pela qual se pretende pesquisar. Para que este método seja realizado é necessário fazer-se um pedido *HTTP GET* [caminho da *API]/public/home/tags.*
- Obter integralmente a lista de *tags* disponibilizadas pela *API*. Esta função será utilizada para que as alterações às áreas de trabalho disponíveis sejam feitas apenas na componente servidora da aplicação, sendo que o cliente obtém esta lista através deste pedido. Para isso é necessário realizar um pedido *HTTP GET [caminho da API]/public/tags.*

- Obter o *username* de um utilizador. Esta função será utilizada para obter o nome de um utilizador que tenha interagido com um trabalho ou artista. Para obter esta informação é necessário enviar uma mensagem de *GET [caminho da API]/public/user-name* com o parâmetro do pedido *userId* preenchido com o identificador de um utilizador.

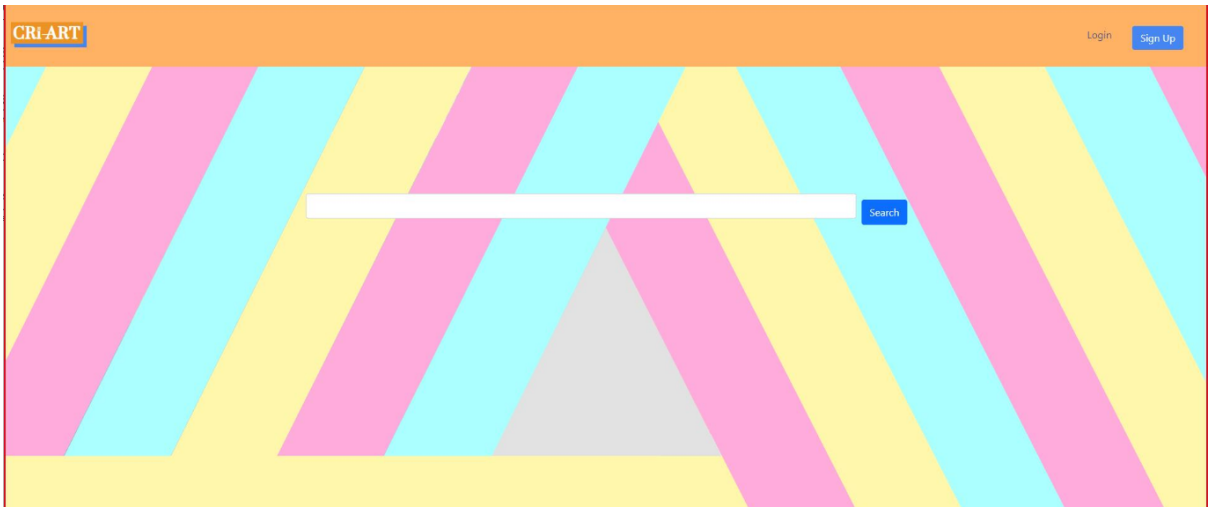

Figura 5 - *Home page* da aplicação para utilizadores não autenticados

Feed

O componente *Feed* apresenta o conteúdo ao utilizador, caso este esteja autenticado. Nesta página é apresentado um cartão por cada publicação presente com o conteúdo da aplicação e com um botão que remete o utilizador para o perfil do artista que efetivou a publicação. Isto é conseguido através de um pedido *HTTP* GET *[caminho da API]/feed* ao servidor de recursos. Este devolve uma lista de trabalhos que é apresentada no formato mostrado ao lado.

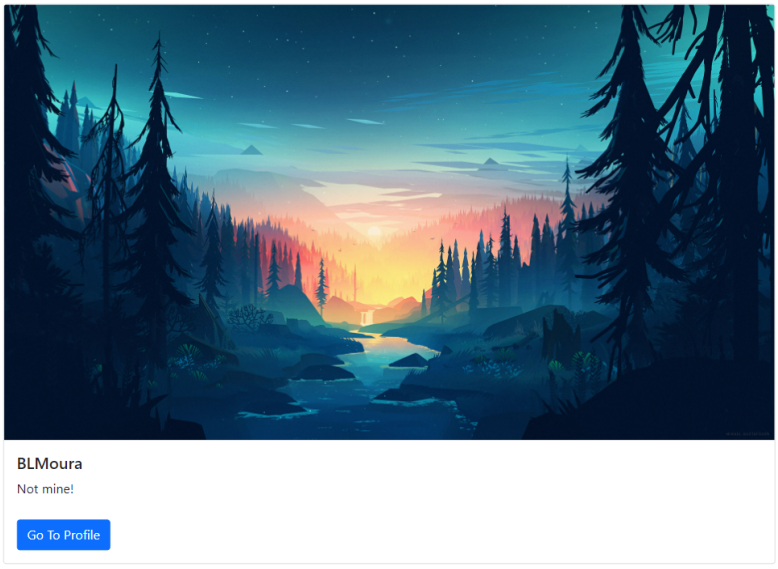

Figura 6 - Trabalho de um artista apresentado no *feed*

Barra de navegação

Esta barra acompanha todas as páginas de navegação dentro da aplicação. Nela estão contidos o botão de navegação que remete para a página inicial, a barra e botões de pesquisa (excepto quando se está na página principal), os botões de *login* e *sign up*, quando o utilizador não está autenticado, e, caso o utilizador esteja autenticado, o botão de *logout* e um botão *my profile* que remete o utilizador para o seu perfil, caso seja um artista.

Página de Pesquisa

Como mencionado anteriormente, é possível realizar pesquisas tanto por nome como por *tag*. Esta pesquisa é feita através de uma caixa de texto, onde se coloca o termo pelo qual se pretende pesquisar, ou através de um botão que apresenta as *tags* todas disponíveis.

Ao efetivar a pesquisa será feito um pedido *HTTP GET* à aplicação servidora, para o caminho *[caminho da API]/public/home/tags,* caso a pesquisa seja feita por uma *tag* específica, ou para o caminho [caminho da *API]/public/home/search*, caso seja por nome, e o utilizador será redirecionado para uma página onde são mostrados os resultados desta pesquisa. Nesta página aparece uma lista de cartões com o nome e a descrição de cada artista que corresponda ao parâmetro de pesquisa. Em cada um desses cartões é apresentado um botão que levará o utilizador ao perfil do artista em questão.

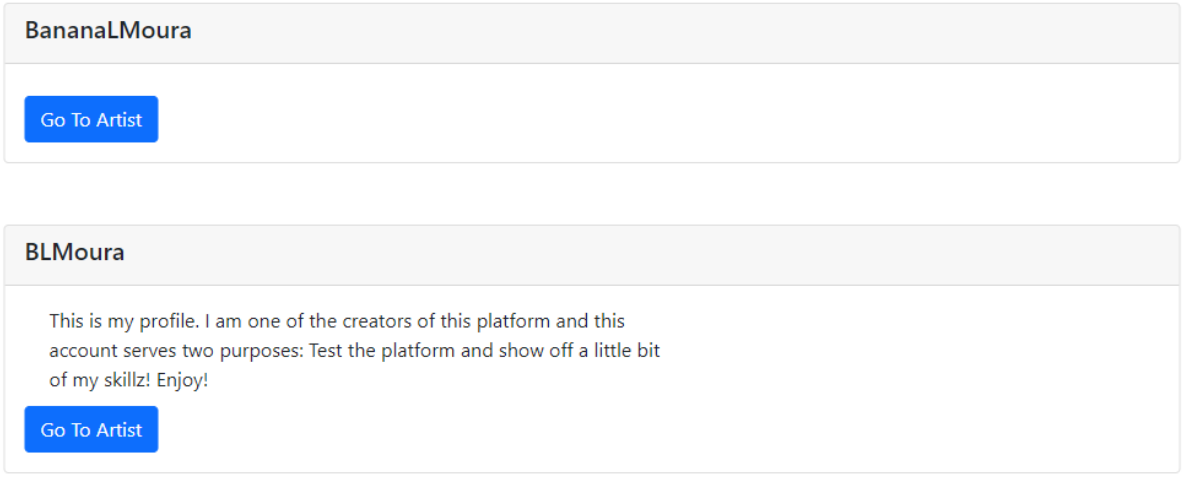

Figura 7 - Pesquisa por nome: Moura

Figura 8 - Pesquisa por *tag*: *Storytelling*

Construção das páginas de *login*, *sign up* e confirmação de conta

Para que um utilizador possa usar a aplicação integralmente, foram definidos os componentes que permitem o registo e autenticação na plataforma. Sendo assim, foram implementados os componentes *Login, Signup e AccountConfirmation* que servem para um utilizador se autenticar, registar-se na aplicação e confirmar esse registo respetivamente.

Login - Autenticação de utilizadores

A página de login disponibiliza campos de texto que permitem ao utilizador introduzir as suas credenciais, *email* e *password*. Estas são necessárias para a autenticação do utilizador como a figura abaixo demonstra. Ao carregar no botão de *login,* este componente irá realizar um pedido de *POST* à *API* para o caminho *[caminho da API]/auth/login* enviando as credenciais introduzidas no corpo do pedido. Na resposta, caso seja bem sucedida, o utilizador será redirecionado para a página principal, página esta que agora apresentará o *feed.*

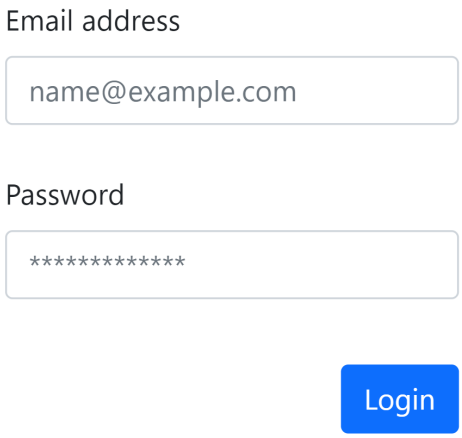

Figura 9 - Página de *login*

Signup

Para que o utilizador possa efetuar o *login*, terá primeiro de registar a sua conta. Para tal, a aplicação disponibiliza um componente que permite a realização do registo. Nesta página serão apresentadas caixas de texto onde o utilizador irá colocar o nome que deseja para ser utilizado dentro da aplicação, o *email*, a *password* e um campo para escolher se se quer registar como cliente ou como artista.

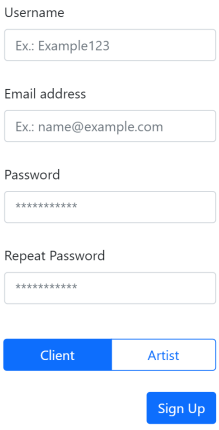

Figura 10 - Página do signup

Após a conclusão do registo, a aplicação cliente irá fazer um pedido de *POST [caminho da API]/auth/signup* onde incluirá no corpo as informações introduzidas pelo cliente. A aplicação, posteriormente, mostrará uma mensagem pedindo ao utilizador para verificar o registo.

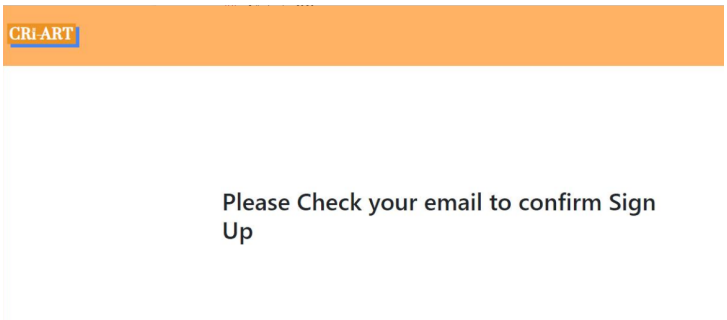

Figura 11 - Mensagem apresentada na página de confirmação do *email*

Confirmação de conta

Ao carregar no *link* enviado para o *email,* o utilizador irá deparar-se com a mensagem apresentada em seguida. Passados cinco segundos será redirecionado para a página de *login.*

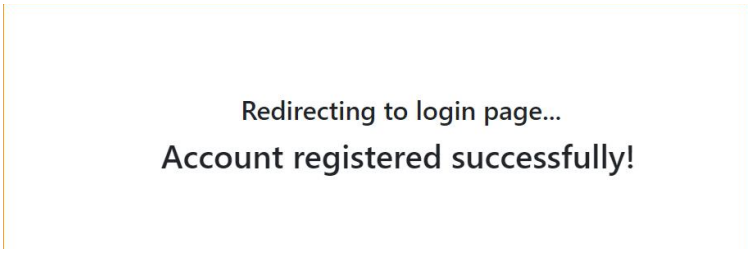

Figura 12 - Mensagem apresentada na página de sucesso de confirmação do email

No caso de se tratar de um artista, aquando do primeiro *login*, será apresentada a página de criação do seu perfil de artista. Após o preenchimento do campo de texto com a descrição do perfil e da seleção das *tags,* poderá criar o seu perfil.

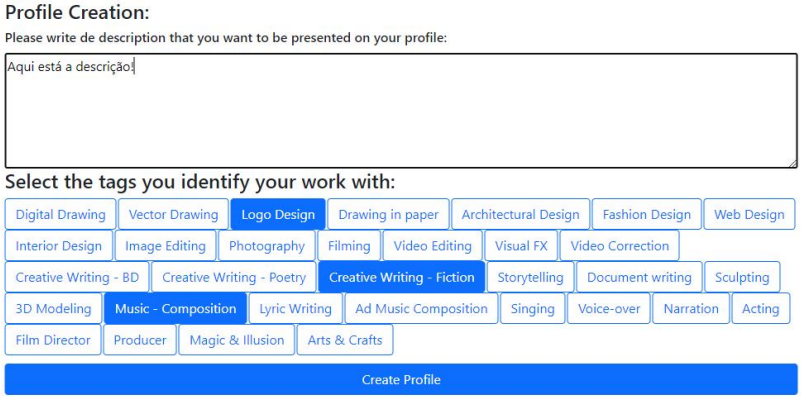

Figura 13 - Página de criação do perfil do utilizador

Perfil de um artista

Para suportar as funcionalidades disponibilizadas no perfil do artista, fizeram-se as seguintes adições de *handlers* à aplicação servidora, nas respetivas classes *controller* :

- *Artist Controller:*
  - Seguir um artista, para poder ver as novas publicações do mesmo sem ter de navegar até à página deste artista. Para se realizar este pedido, é necessário fazer um pedido *HTTP* de PUT *[caminho da API]/artist/{aid}/follow*, não precisando de corpo da mensagem, já que os atributos de utilizador são adicionados pelo filtro de autenticação.
- *Work Controller:*
  - Adicionar um comentário ao trabalho. Uma das formas de interação entre os utilizadores e os artistas é que os primeiros podem deixar comentários nas publicações dos artistas. Para tal, é necessário que este pedido receba o identificador do artista bem como o identificador do trabalho e o comentário. Ambos os identificadores serão enviados através do caminho do pedido, já o comentário é enviado no corpo do mesmo. Um pedido para este *handler* seguirá a forma *POST [caminho da API]/artist/{aid}/worksofart/{wid}* onde *aid* e *wid* são os identificadores de artista e trabalho, respetivamente. Este método foi feito aquando da implementação do componente de cliente.
  - Dar ou retirar um *upvote*. Da mesma forma que o ponto anterior, outra forma de interagir com os artistas será dar um *upvote* a um trabalho deste, sendo um *upvote* uma forma de representar que se gosta do trabalho em questão. Neste caso, o mesmo método é responsável por dar ou retirar um *upvote*, deixando se o identificador do utilizador não constar na lista de *ups* do trabalho ou retirando se constar. Para realizar esta funcionalidade é necessário, então, fazer um pedido de *PUT [caminho da API]/artist/{aid}/worksofart/{wid}*. Este método foi feito aquando da implementação do componente de cliente.

Esta página será apresentada de cada vez que um utilizador pretenda aceder à página de um artista. Nesta página será apresentado o conteúdo referente a um artista e os respectivos trabalhos. É apresentado o nome, a descrição e as *tags* representantes das áreas de trabalho do artista, bem como o botão *follow* que permite ao utilizador seguir o artista. Para apresentar os detalhes afectos ao artista, é feito um pedido *GET* à *API* para o *endpoint [caminho da API]/artist/{aid}*.

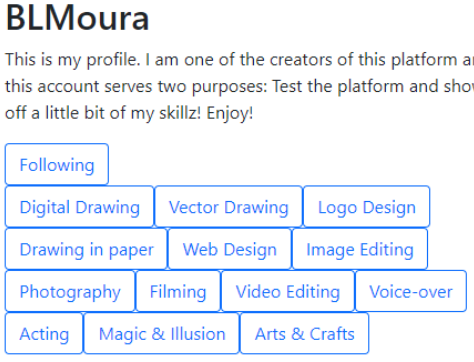

figura 14 - Página de perfil do artista

Esta página irá também apresentar o conteúdo dos trabalhos deste artista. Para tal, implementaram-se os componentes seguintes:

- *WorkManagement* que, dependendo se o utilizador é o dono do perfil, mostra a opção de adicionar mais trabalhos;
- *WorkList* que apresenta uma lista de *WorkPost*s

WorkPost

Este componente define uma publicação de um trabalho do artista. Neste cartão está presente o conteúdo do trabalho. Está também presente um botão que permite a um utilizador deixar um *upvote* no trabalho. Este *upvote* é realizado através de um pedido PUT [caminho da API]/artist/{aid}/worksofart/{wid}.

Para além do conteúdo mostrado, é possível carregar na publicação e irá abrir um *modal*, definido no *Bootstrap* [8] que, para além do conteúdo apresentado, irá apresentar a lista de comentários feitos a esse respectivo trabalho.

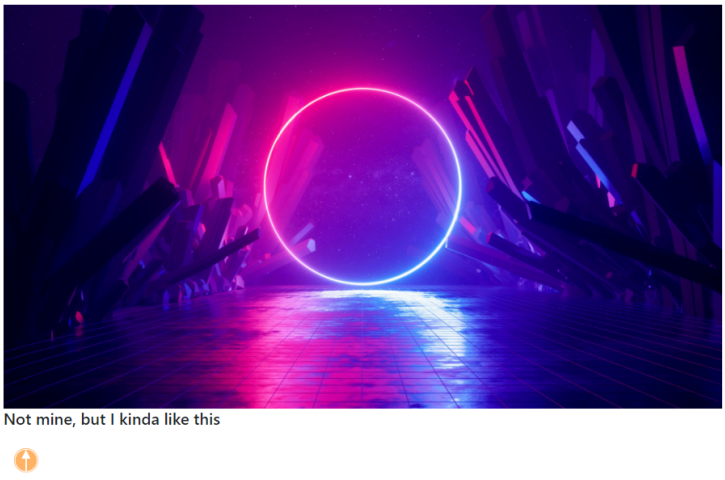

Figura 15 - Cartão com o trabalho de um artista

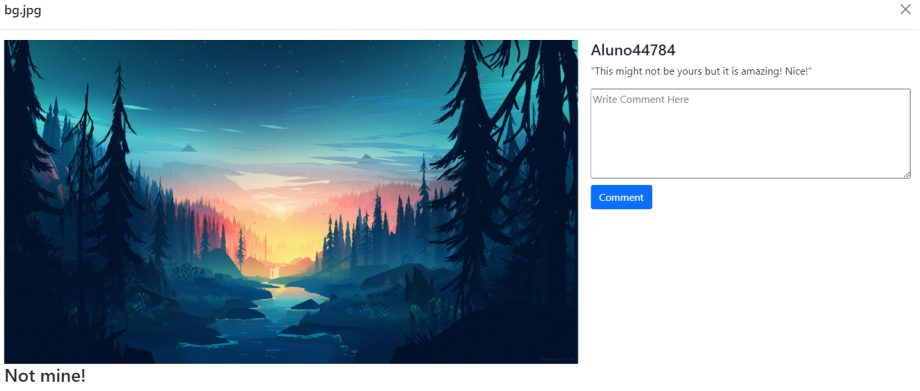

Figura 16 - *Modal* com os comentários ao trabalho do artista

Conclusão

A título de conclusão realça-se que, para o desenvolvimento futuro, a abordagem monolítica não deverá ser a escolha, mas sim uma arquitetura em micro-serviços, dado que é uma abordagem mais moderna, seguindo a regra *separation of concern*. Serve como exemplo a componente de autenticação e autorização bem como o serviço de mensagens/notificações. A autenticação e autorização poderiam ser implementados num micro-serviço separado e o serviço de mensagens e notificações poderia ser implementado utilizando um mecanismo *Pub/Sub.*

No que concerne à aplicação cliente, nota-se que poderia haver uma maior preocupação com a apresentação do conteúdo, dando alguma prioridade à componente de *web design*, e também uma maior preocupação com a interatividade da plataforma, tentando tornar a aplicação o mais *user-friendly* possível.

Tendo em conta, agora, o projeto no seu todo, faz-se notar que o esqueleto necessário para o funcionamento do que era a proposta inicial se encontra concluído. Faz-se também notar que se revelou uma falta de organização de trabalho bem como falta na definição das prioridades do projeto.

Por fim, a título pessoal, este projeto permitiu aos integrantes aprender a importância da organização e trabalho em equipa, bem como lhes permitiu aprender por si novas tecnologias e metodologias. Foi também importante para desenvolver o espírito crítico e a vontade de trabalhar.

Apesar de o projeto não corresponder exatamente às expectativas iniciais, foi uma ferramenta de aprendizagem muito importante para os integrantes.

Bibliografia

(1) https://www.baeldung.com/spring-controller-vs-restcontroller#spring-mvc-rest-controller

(2) https://www.baeldung.com/spring-http-logging#custom-request-logging, The main issue with the reading request is that, as soon as the input stream is read for the first time, it's marked as consumed and cannot be read again.

(3) https://graf1x.com/color-psychology-emotion-meaning-poster, pontos "*Orange Color Meaning"* e *"Blue Color Meaning"*

(4) *https://vectr.com/*

(5) *Github Actions*: https://docs.github.com/pt/actions/learn-github-actions/understanding-github-actions

(6) *CI/CD*: https://www.redhat.com/pt-br/topics/devops/what-is-ci-cd

(7) Imagens *Docker* para *Heroku*:

https://devcenter.heroku.com/articles/container-registry-and-runtime#dockerfile-commands-and-runtime

(8) Bootstrap *https://getbootstrap.com/docs/5.1/components/modal/*