Tecnologias e Programação Web

Professor: Hélder Zagalo

Aplicação Web: TechSekai

Frontend: Angular 9

Backend: Django Rest Framework (DRF)

Equipa:

Fábio Carmelino, 93406

Maria Rocha, 93320

Pedro Souto, 93106

DETI

Universidade de Aveiro

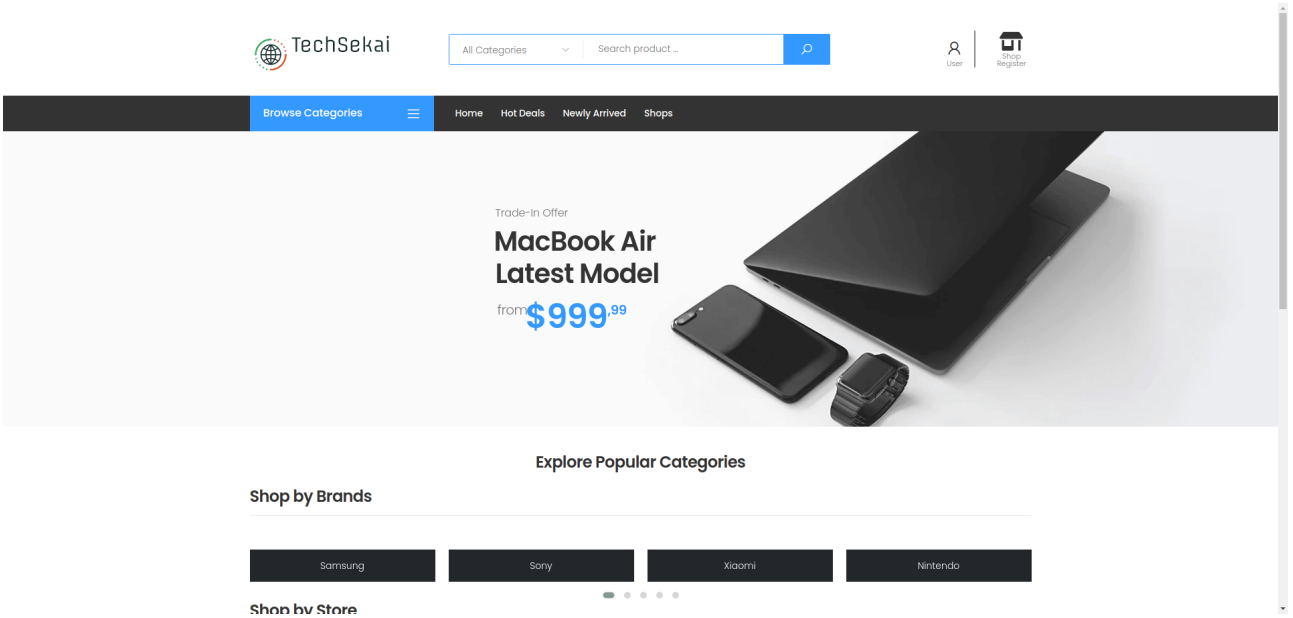
01-07-2021

Índice

1. Introdução

No âmbito da disciplina de Tecnologias e Programação Web, como segundo projeto, foi solicitado o desenvolvimento de uma aplicação web usando as frameworks DRF para o Back-end e Angular para o Front-end. Deste modo, este relatório visa descrever todas as funcionalidades implementadas e estratégias adotadas durante o desenvolvimento desta aplicação.

O nome da aplicação é TechSekai (Tech da palavra inglesa "technology" e Sekai da palavra japonesa "mundo" resultando em "Mundo da Tecnologia"), e tal como o nome sugere, a **sua temática é tecnologia**. Usando como base a aplicação desenvolvida durante o primeiro projeto da disciplina, TechSekai, é uma **plataforma que permite várias lojas venderem os seus produtos** tecnológicos. As lojas podem criar uma conta e, ou adicionar novos produtos que ainda não existam, ou adicionar produtos iguais aos das outras lojas, mas definindo o seu próprio preço. Deste modo, **semelhante a websites como 'KuantoKusta', 'Kimovil' e 'Aliexpress'**, os clientes podem facilmente procurar pelos melhores preços ou pela loja em que mais confiam.



2. Funcionalidades da aplicação

De forma a melhor compreender as funcionalidades implementadas na globalidade da aplicação, passamos a enumerá-las.

Utilizador Normal (Sem conta):

- Visualização das marcas e lojas na página principal;
- Visualização dos produtos mais vendidos na aba “Hot Deals”;
- Visualização dos produtos criados mais recentemente através da aba “Newly Arrived”;
- Visualização das lojas na aba “Shops”;
- Visualização dos detalhes de uma loja clicando na mesma (aba “Shops”);
- Pesquisa de produtos por nome ou nome e categoria através da barra de pesquisa (uma pesquisa vazia resultará na visualização de todos os produtos);
- Visualização dos produtos de uma determinada categoria através da aba “Browse Categories”;
- Visualização dos detalhes de um produto clicando no mesmo (informações técnicas e lojas onde é vendido);

Utilizador Normal (Com conta):

- Todas as funcionalidades descritas anteriormente;
- Visualização do carrinho através do ícone “Cart” (possui detalhes como produtos, preços, quantidades e preço total);
- Visualização dos produtos favoritos através do ícone “Wishlist”;
- Visualização e alteração das definições da conta (endereço na aba “Address” e detalhes do utilizador na aba “Account Details”) através do ícone de utilizador e opção “My Account”;
- Visualização das compras feitas na aba “Orders” da secção “My Account” descrita anteriormente;
- Possibilidade de adicionar um produto, de entre múltiplas lojas, ao carrinho através da sua página de detalhes (botão “Add to cart”);
- Possibilidade de remover um produto do carrinho (botão “x” no carrinho);
- Possibilidade de adicionar ou remover um produto aos favoritos através da sua página de detalhes (botão “Add to wishlist” se não estiver nos favoritos, botão ”Remove from wishlist” se já se encontrar nos favoritos);
- Visualização do número de produtos no carrinho e favoritos através dos respetivos “Badges” na barra de navegação;
- Possibilidade de fazer uma compra através do carrinho (botão “Confirm purchase”), se o utilizador possuir um endereço definido e houver stock suficiente, caso contrário serão mostrados os devidos avisos;

Utilizador com conta de Loja:

- Listar todos os produtos criados pela própria loja;
- Listar ,criar, editar e apagar os items da loja;
- Criar, editar[1] e apagar[1] um produto da própria loja; [1]quando outras não dependem dele
- Pesquisar por nome e categoria os seus produtos;

3. Domain Model:

Entidades :

- ❖ User [django_user, gender, age, phone_number, avatar, address(FK)]
- ❖ Address [country, city, zip_code, street, door, floor]
- ❖ WishList [user(FK), prods(FK)]
- ❖ Cart [user(FK), total_price]
- ❖ Cart_Item [item(FK), qty, cart(FK)]
- ❖ Order [user(FK), item(FK), quantity, total_price, order_state, payment_meth]
- ❖ Category [name, totDevices, image]
- ❖ Brand [name]
- ❖ Shop [name, owner(FK), phone_number,address(FK),website,opening_hours,image]
- ❖ Item [Product(FK), Shop(FK), price, stock]
- ❖ Product [reference_number,name,details,warehouse,qty_sold,image,category(FK),brand(FK), lowest_price, creator(FK)]

Breve Descrição:

- A nossa BD prevê 2 tipos de utilizadores. 'Shop' para lojas e 'User' para clientes respetivamente.
- As entidades de 'Categoria' e 'Brand' são usadas para categorizar produtos.
- Ambos o 'Shop' e o 'User' podem ter um Address.
- Um 'User' tem uma 'WishList' e uma 'Cart' com 'Cart_Items'.
- Sempre que um utilizador faz uma compra, os 'Cart_Items' são eliminados e é gerada uma 'Order' para cada um deles.
- Os 'Shops' quando criam um novo produto que ainda não existe, criam também um 'Item' que terá o seu preço e stock. Mas o produto em si, pode ser reutilizado pelas outras lojas, ou seja, essas outras lojas podem também criar um item para esse produto e definir o seu preço e stock.
- Uma Loja pode editar o seu Item, mas não pode editar um 'Product' mesmo que seja seu, caso outras lojas já tenham 'Items' com correspondência para esse 'Product'.

4. Restful Services (Django Rest Framework)

A grande vantagem de usar uma Rest API para o consumo de serviços é o aumento da independência entre os módulos de uma aplicação. Assim sendo, a parte do Front-end tem total liberdade para ser desenvolvida usando qualquer outra framework, dando possibilidade para uma arquitetura 2-tier, BD e a API num módulo e Front-end em outro, ou 3-tier, BD/API/Front-end separados, entre outras. A nossa aplicação tem arquitetura 2-tier.

A DRF, é uma framework do Django que usa a linguagem python para a criação de APIs. Esta permite o desenvolvimento de rest endpoints em forma de 'views' que são mapeadas por 'urls' e fornece várias possibilidades de métodos de autenticação como por exemplo, 'Session Authentication' e 'Token Authentication'. Além disso, tem ainda Serializers que suportam ambos ORM e non-ORM datasources.

4.1. Endpoints

Todos os acessos à API têm de passar por algum endpoint. Estes endpoints são 'urls' que os consumidores dos serviços podem chamar e que conduzem um pedido para uma view. Estas views irão então posteriormente processar cada um dos pedidos recebidos e enviar uma resposta de volta.

Endpoints da nossa API:

```
########## REST ENDPOINTS ##############

# Products
path('api/products/', views.list_prods),
path('api/products/create', views.create_product),
path('api/products/edit/<int:pid>', views.update_product),
path('api/products/<int:pid>', views.see_product),
path('api/products/delete/<int:pid>', views.delete_prod),
path('api/products/hotdeals',                   views.get_prods_hotdeals),
path('api/products/newarrivals',                views.get_prods_newarrivals),
path('api/products/search', views.search),
path('api/products/search2/<str:filter>/<str:value>', views.search2),

# User
path('api/account/signup',                  views.sign_up),
path('api/account/login',                   obtain_auth_token),
path('api/account/role',                    views.get_user_role),
path('api/account/info',                    views.get_user_info),
path('api/account/info/update',             views.update_user_info),
path('api/account/orders',                  views.get_user_orders),
path('api/account/address/add',             views.user_address_add),
path('api/account/address/update',          views.user_address_update),
path('api/account/address/rem',             views.user_address_rem),
path('api/account/cart',                    views.get_cart),
path('api/account/wishlist',                views.get_wishlist),
path('api/account/wishlist/add',            views.wishlist_add),
path('api/account/wishlist/rem/<int:prod_id>', views.wishlist_remove),
path('api/account/order', views.order_product),
path('api/account/cart/add/<int:item_id>', views.add_to_Cart),
path('api/account/cart/rem/<int:item_id>', views.rem_from_Cart),
```

```
# Items
path('api/items/', views.get_list_items),
path('api/items/create', views.create_item),
path('api/items/<int:id>', views.see_item),
path('api/items/edit/<int:id>', views.update_item),
path('api/items/delete/<int:id>', views.item_delete),

# Brands and Categories
path('api/brands/', views.list_brands),
path('api/categories/', views.list_categories),

# Shops
path('api/shops/', views.get_shops_list),
path('api/shops/<int:sid>', views.get_shop),
path('api/shops/create', views.create_shop),
path('api/shops/delete', views.shop_delete),
path('api/shops/edit', views.edit_shop),

path('api/home', views.home_content),
path('api/shop/products/<int:prod_id>', views.product_shops),
path('api/shop/products/wished/<int:prod_id>', views.isWished),

# Cart
path('api/cart/enoughQty', views.enoughQty),
path('api/cart/sum', views.getSumCart),
path('api/products/stock', views.prod_stock),
```

4.2. Views

As Views, tal como foi referido anteriormente, possuem toda a lógica para processar os pedidos. Para esse objetivo, em métodos do tipo PUT ou POST, elas tiram proveito de serializers para converter e validar o conteúdo recebido. Além do mais, é possível restringir o acesso a estas views com anotações como é o caso da anotação de autenticação obrigatória - `@permission_classes([IsAuthenticated])`.

Exemplo da view que atualiza o address do utilizador através de um 'PUT':

```python
@api_view(['PUT'])
@permission_classes([IsAuthenticated])
def user_address_update(request):
    user = User.objects.get(django_user=request.user)

    if user.address is None:
        return Response("This User does not have an address yet. ", status=status.HTTP_404_NOT_FOUND)

    serializer = AddressSerializer(user.address, data=request.data)
    if serializer.is_valid():
        serializer.save()
        return Response("Address updated successfully.", status=status.HTTP_201_CREATED)

    return Response(serializer.errors, status=status.HTTP_201_CREATED)
```

4.3. Serializers

Na ilustração da seção acima podemos ver que está a ser usado um *AddressSerializer*. Os Serializers, semelhante aos forms do Django, podem não só usar uma classe de um *model* permitindo escolher quantos atributos dessa classe queremos que o Serializer trate, como também ser usado para definir um conjunto de atributos à nossa escolha sem estar associado a qualquer *model*.

Estes Serializers são particularmente úteis no primeiro caso, pois conseguem validar se os parâmetros recebidos no **PUT** ou **POST** respeitam a modelação do *model* de uma entidade e ainda fazer a conversão do formato json para a classe em questão, sendo assim fácil guardar um novo objeto ou atualizar um existente na base de dados.

Em caso de haver 'nested classes', pode-se ainda definir funções para atacar essas situações, como é visível no exemplo abaixo com o *ItemSerializer*:

```python
class ItemSerializer(serializers.ModelSerializer):
    product = ProductSerializer(read_only=True)
    shop = ShopSerializer(read_only=True)

    class Meta:
        model = Item
        fields = '__all__'
        read_only_fields = ['id']

    def create(self, prod, shop):
        item = Item.objects.create(price=self.validated_data['price'], stock=self.validated_data['stock'], product=prod, shop=shop)
        return item

    def update(self, instance):
        instance.price = self.validated_data['price']
        instance.stock = self.validated_data['stock']
        instance.save()
```

4.4 Autenticação

Para a autenticação foram utilizados Tokens que é um dos métodos disponibilizados pelo DRF. Sempre que um utilizador é criado, um Token é gerado, e nos momentos de login o utilizador consegue obter o seu token ao introduzir corretamente as suas credenciais de username e password que são validadas por uma view do Django chamada 'obtain_auth_token'.

A tecnologia usada no frontend deverá então ser capaz de reter o token obtido no login, pois este será necessário e deverá ser incluído nos headers para ser possível o acesso a endpoints com autenticação obrigatória.

Token Authentication Endpoint:

```
path('api/account/login', obtain_auth_token)
```

5. Angular (Front-end)

O Angular é uma framework utilizada para desenvolver o front-end de aplicações web. Esta é baseada na linguagem Typescript e é orientada a componentes. As interfaces produzidas em Angular são compostas por vários componentes, e um programador que esteja a trabalhar com esta tecnologia, dentro do possível, deve tentar criar componentes de maneira a que eles possam ser reutilizados em diferentes páginas, pois essa é uma das maiores vantagens do Angular em oposição a outros modelos como o MVC em que cada página está associada a um único ficheiro html.

5.1 Services

Para a comunicação com a REST API foram criados vários 'services' em Angular. Estes *services* permitem criar várias funções para comunicar com os endpoints da API. Deste modo, qualquer componente que precise de alguma informação vinda do Back-end pode adquiri-la através destes *services*.

Exemplo de um *service* para comunicar com endpoints relativos à entidade Item:

```
export class ItemsService {
  private baseURL= REST_API_BASE_URL + "/items/"

  constructor(private http:HttpClient) { }

  getItems():Observable<Item[]> {
    const url = this.baseURL
    return this.http.get<Item[]>(url);
  }

  getItem(id: number):Observable<Item> {
    const url = this.baseURL + id
    return this.http.get<Item>(url);
  }

  createItem(item:Item):Observable<any>{
    const url = this.baseURL + "create";
    item['shopId'] = localStorage.getItem( key: 'shopId')
    return this.http.post(url,item,httpOptions);
  }

  updateItem(item: Item):Observable<any>{
    const url = this.baseURL + "edit/"+item.id;
    console.log(item)
    item['shopId'] = localStorage.getItem( key: 'shopId')
    return this.http.put(url,item,httpOptions);
  }

  deleteItem(id: number):Observable<any>{
    const url = this.baseURL + "delete/"+id;
    return this.http.delete(url,httpOptions);
  }

  itemPerShop(id:number):Observable<Item[]>{
    const url = REST_API_BASE_URL + "/shop/products/"+id;
    return this.http.get<Item[]>(url);
  }
```

5.2 Authentication

Para a autenticação ser possível em Angular, foi criado um serviço chamado 'AuthService' e um Interceptor chamado *AuthHeaderInterceptor*.

O 'AuthService' possui diversas funções relativas à autenticação, nomeadamente funções como a de 'signIn()', 'signUp()', 'IsAutheticated()' e 'getToken()' cujos nomes são auto-explicativos. É ainda importante referir que para guardar o Token de autenticação a função de signIn() tira proveito do 'localStorage' que esta framework disponibiliza.

O *AuthHeaderInterceptor* é um serviço que implementa a interface *HttpInterceptor* e portanto será capaz de interceptar todos os 'http requests'. Deste modo, enquanto o Token estiver no localStorage, este interceptor irá colocá-lo nos headers mantendo assim o utilizador autenticado.

Interceptor:

```
import {Injectable, Injector} from '@angular/core';
import {
  HttpRequest,
  HttpHandler,
  HttpEvent,
  HttpInterceptor
} from '@angular/common/http';
import { Observable } from 'rxjs';
import {AuthService} from "../../_services/auth.service";

@Injectable()
export class AuthHeaderInterceptor implements HttpInterceptor{

  constructor(private injector: Injector) {}

  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    const authService = this.injector.get(AuthService);
    if (authService.isAuthenticated()){
      const tokenizedReq = req.clone( update: {
        setHeaders: {Authorization: `Token ${authService.getToken()}`}
      });
      return next.handle(tokenizedReq);
    }
    return next.handle(req);
  }
```

6. Executar os projetos

6.1 Localmente

Para ser executado o sistema necessita de correr o serviço *REST* (DRF) e a aplicação web (Angular).

Antes de correr a aplicação web é necessário instalar as suas dependências correndo o comando **npm install** na raiz do projeto.

Relativamente ao serviço *REST* (Django) é preciso instalar os requisitos que se encontram no ficheiro **requirements.txt**.

Tendo as dependências e requisitos instalados basta executar ambos os projetos em simultâneo.

6.2 Deploy

O deploy foi feito com recurso às plataformas *Python Anywhere* para o serviço *REST* e *Heroku* para a aplicação web.

- Serviço *REST*: http://pedrogsouto.pythonanywhere.com/
- Aplicação Web: http://itadakimasu-angular.herokuapp.com/

Nota: O site em angular deverá ser acedido com http ou não comunicará com o serviço *REST*.

6.3 Contas

Projeto	**Tipo de conta**	**Utilizador**	**Palavra Passe**
Django *Rest*	Admin	Admin	ADMIN
Aplicação Web	Cliente	UserDemo	userdemo
Aplicação Web	Loja	Worten	worten2021

7. Conclusão

Este projeto serviu, fundamentalmente, para explorar as vantagens de uma arquitetura *client-side*, orientada a componentes, e perceber a forma como a troca de dados com o ambiente *server-side deve ocorrer*, mantendo uniforme a interoperabilidade entre os sistemas. Esta uniformidade deve ser assegurada mesmo quando ambas as lógicas são desenvolvidas em diferentes linguagens de programação, diferentes ambientes e apresentam diferentes características, como pudemos comprovar.

Relativamente a trabalho futuro, se tivéssemos novo prazo para melhorar a solução desenvolvida teríamos como principais objetivos: permitir o *upload* de uma imagem como capa de um produto aquando da criação ou edição do mesmo, adicionar mais formulários de pesquisa opções de filtragem combinadas de produtos, permitir autenticação com conta do Google ou Facebook e acrescentar paginação às páginas de listagem de resultados.

Em suma, consideramos que os objetivos do projeto foram concluídos com sucesso, tendo contribuído para um maior aprofundamento na área de tecnologias web, mais especificamente no desenvolvimento com Angular e Django *Rest Framework*.