

Tecnologias e Programação Web

Professor:
Hélder Zagalo

Aplicação Web com DRF e Angular

Hugo Paiva, 93195
Daniel Gomes, 93015
Pedro Bastos, 93150



DETI
Universidade de Aveiro
20-01-2020

Índice

1	Introdução	2
2	Features da aplicação	3
3	API (Django REST Framework)	4
3.1	Autenticação	4
3.2	Autorização	4
3.3	Views	5
3.4	Serializers	6
3.5	Documentação da API	7
4	Angular	8
4.1	Apresentação de dados	8
4.2	Serviços	11
4.2.1	Obtenção de dados	11
4.2.2	Sistema de avisos	12
5	Execução do sistema	13
5.1	Localmente	13
5.2	Deploy	13
5.3	Utilizadores	13
6	Conclusão	14

1 Introdução

No âmbito da disciplina de Tecnologias e Programação Web, este relatório visa clarificar os aspetos mais importantes do segundo projeto, sendo este uma adaptação do primeiro projeto. Assim, o *Django* passa a servir como *Rest API* e, para *front-end*, foi utilizado *Angular*. Assim como no projeto anterior, a App Store tem como objetivo permitir aos utilizadores a pesquisa, visualização e compra de aplicações, bem como a possibilidade de adição de reviews e favoritos.

Estão presentes na aplicação dois tipos de utilizadores:

- **Os clientes**, que serão os utilizadores gerais da aplicação e, como já foi referido anteriormente, têm um conjunto de ações permitidas.
- **Os administradores**, que podem fazer todo o tipo de operações de adição, edição e remoção de aplicações, *developers*, categorias, etc. Podem ainda adicionar balanço aos clientes.

Assim, a aplicação é composta por duas módulos principais:

- Interface gráfica (*Angular*)
- REST API (*Django REST framework*)

2 *Features* da aplicação

Importante mencionar uma visão geral das *features* da aplicação, com o objetivo de dar um *overview* do que a mesma é capaz de fazer. As funcionalidades comuns mais importantes são:

- Visualizar a página inicial com os produtos mais vendidos e mais recentes;
- Navegar pela Shop, podendo aplicar os filtros, para ver a lista de produtos disponíveis;
- Registo de um novo utilizador, que posteriormente necessita de realizar o Login.

No lado do cliente, após o login, são possíveis as seguintes ações:

- Além da navegação pela Shop já anteriormente referida, é possível entrar dentro da página de cada produto, coisa que sem o Login não seria permitida;
- Dentro de cada produto, é possível ver as suas informações e comprar, bem como ver as informações do *Developer*, adicionar/remover dos favoritos e adicionar/editar/remover reviews;
- Na página dos detalhes do cliente (clicando no seu *username*) é possível alterar as suas informações e password, ver as suas compras, favoritos e reviews feitas;

Já no lado do administrador, tem como extra as funcionalidades:

- Ver os detalhes de todas as compras efetuadas na aplicação;
- Ver os detalhes de todos os *users* e adicionar balanço às respetivas contas;
- Ver uma lista de todos os produtos, com opção de edição dos mesmos. É permitido também adicionar um produto;
- Ver uma lista de *developers*, editá-los e adicionar novos;
- Ver uma lista de categorias, editá-las e adicionar novas;

3 API (Django REST Framework)

A DRF é uma *framework* do Django para a criação de APIs REST. Assim, toda a interação de dados do *front-end* com o *back-end* é feita através da API, criando uma abstração entre as duas partes e tornando a aplicação mais segura.

3.1 Autenticação

Para a autenticação, foi decidido utilizar *tokens* de autenticação, próprios do *Rest Framework* do Django, que permite obter e verificar um *token* no *login* do utilizador.

```
# Login
path('ws/token-auth/', authviews.obtain_auth_token, name='api-token-auth'),
```

Figure 1: URL da autenticação

A partir do momento do *login*, o *token* é guardado no *front-end* e todos os pedidos à API que necessitem de autorização são acompanhados deste mesmo *token*. Assim, é garantida a autenticidade do utilizador ao fazer os pedidos.

3.2 Autorização

Mesmo depois do cliente efetuar o login, este não tem acesso a todas as informações. Para isso, criou-se uma função que verifica se o pedido é permitido. Por exemplo, um cliente só deverá ter acesso às suas compras, e não às compras dos restantes clientes. Assim, em cada pedido à API, é verificado se o cliente que fez o pedido é o correto para a informação que é pedida.

```
def check_client_permission(request, entity):
    """
    Function used to verify whether a authenticated client can perform a request or not
    For example, a client should only be able to view his purchases, or it's personal information
    @:parameter request : the request received
    @:parameter entity: the object of the class that the client wants to access
    """
    request_username = request.user.username
    if isinstance(entity, Purchase):
        return request_username == entity.client.user.username
    elif isinstance(entity, Client):
        return request_username == entity.user.username
    elif isinstance(entity, User):
        return request_username == entity.username
    elif isinstance(entity, Reviews):
        return request_username == entity.author.user.username
    return None
```

Figure 2: Função de verificação das permissões

3.3 Views

As *views* têm um papel importante na *API*, visto que é nelas que estão presentes todos os pedidos disponíveis. Estão disponíveis os métodos *GET*, *PUT*, *POST* e *DELETE*. Cada *endpoint*, nos *URLs*, chama uma *view* que define o método, as autorizações e só depois efetua o pedido, caso sejam validadas todas as condições.

```
@api_view(['PUT'])
def update_client(request, id):
    """
    The main Goal of this endpoint it to edit client's favorite applications.
    To Edit personal data like email or name it is used the User endpoint
    @:parameter id : the id of the client
    """
    user = check_request_user(request)
    try:
        client = Client.objects.get(id=id)
        if user == 'Client':
            if not check_client_permission(request, client):
                return Response({'error_message': "You're not allowed to do this Request!"},
                                status=status.HTTP_403_FORBIDDEN)

        serializer = ClientSerializer(client, request.data)

        # Only Admins can Edit the Clients balance,
        # therefore if a client tries to edit it's own balance, this request will not be Allowed
        if serializer.is_valid():
            if (user == 'Client'
                and 'balance' in serializer.validated_data
                and serializer.validated_data['balance'] != client.balance):
                return Response({'error_message': "You're not allowed to edit your own Balance!."},
                                status=status.HTTP_403_FORBIDDEN)

            serializer.save()
            return Response(serializer.data)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
    except Client.DoesNotExist:
        return Response({'error_message': "User not found!"},
                        status=status.HTTP_404_NOT_FOUND)
```

Figure 3: Exemplo de uma *view* para o *update* (*PUT*) das informações do cliente

3.4 Serializers

Toda a informação é processada pelos *serializers*, utilizando o módulo '*serializers.ModelSerializer*' importado do *Django Rest Framework*. Estes permitem fazer verificações, tratar da informação e definir os campos a retornar e em que formato. Por exemplo, para o administrador criar um produto, basta fazer a seguinte chamada na *view*:

```
serializer = ProductSerializer(data=request.data)
if serializer.is_valid():
    serializer.save()
    return Response(serializer.data, status=status.HTTP_201_CREATED)
return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

Figure 4: Chamada do *serializer* do *administrador*

O *Serializer* vai se certificar que os campos introduzidos estão válidos e, em caso afirmativo, cria o produto, retornando a própria instância do produto:

```
class ProductSerializer(serializers.ModelSerializer):
    created_at = serializers.DateTimeField(read_only=True)
    update_at = serializers.DateTimeField(read_only=True)
    stars = serializers.SerializerMethodField(read_only=True)
    n_of_purchases = serializers.SerializerMethodField(read_only=True)

    class Meta:
        model = Product
        fields = ('id', 'name', 'icon', 'description',
                  'category', 'developer', 'created_at',
                  'update_at', 'price', 'stars', 'n_of_purchases')

    def get_stars(self, obj):
        stars = Reviews.objects.filter(product=obj).aggregate(rating__avg=Ceil(Avg('rating')))[ 'rating__avg' ]
        if stars is None:
            stars = 0
        return int(stars)

    def get_n_of_purchases(self, obj):
        n_purchases = Purchase.objects.filter(product=obj).count()
        return n_purchases

    def to_representation(self, instance):
        data = super().to_representation(instance)
        data['developer'] = DeveloperSerializer(Developer.objects.get(pk=data['developer'])).data
        data['category'] = [CategorySerializer(catg).data for catg in instance.category.all()]
        return data
```

Figure 5: *Serializer* do produto

3.5 Documentação da API

Foi decidido, também, incluir um ponto importante em todas as *REST APIs* que é a sua documentação. Para tal utilizou-se o módulo ***drf-yasg*** do *Python*. Este permite facilmente a criação de um *endpoint* específico, e auto-gerado, contendo a listagem de todos os métodos permitidos para cada *endpoint* na nossa *REST API* e também informação relativa à autenticação e autorização necessária para aceder a estes.

Filter by tag		
categories		
GET	/categories	categories_list
category		
GET	/category/{id}	category_read
categorycre		
POST	/categorycre	categorycre_create
categorydel		
DELETE	/categorydel/{id}	categorydel_delete
categoryupd		
PUT	/categoryupd/{id}	categoryupd_update
client		
GET	/client/	client_list
GET	/client/{id}	client_read

Figure 6: Excerto do conteúdo apresentado na página da documentação da API

4 Angular

Todo o *front-end* da nossa aplicação foi feito utilizando a Tecnologia *Angular*. Esta *framework* apresenta um Arquitetura *Component-Based* e utiliza a linguagem *TypeScript*, aproveitando-se as seguintes funcionalidades:

- **Components:** Utilizou-se *Components* para todas as *views* e lógica proveniente, tentando a divisão ao máximo destes em "*Sub-Components*", de forma passar informação entre componentes dinamicamente;
- **Data Binding:** Com vista a mostrar os dados persistentes;
- **Directives:** Com o intuito de manipular os elementos do DOM, com o comportamento adicional que se pretendia;
- **Services:** Usou-se serviços para obter todos os dados vindo da *REST API*;
- **Routing:** Para a navegação dentro da interface;
- **Observables:** De forma a conseguir trabalhar com programação assíncrona;

4.1 Apresentação de dados

Para popular principalmente os *forms* com dados, utilizou-se o *Two-way Binding*, permitindo através da diretiva *ngModel* a mudança de dados automática entre o modelo de dados e a *view*, combinando a utilização de *Property-Binding* com *Event-Binding*.

```
<form [formGroup]="updateForm" *ngIf="selectedDev" (ngSubmit)="editDeveloper()">
  <div class="modal-body">

    <div class="form-group">
      <label>Name</label>
      <input #name type="text" [(ngModel)]="selectedDev.name" formControlName="name" class="form-control" [ngClass]="{ 'is-invalid': f.name.errors }"/>
      <div *ngIf="f.name.errors" class="invalid-feedback">
        <div *ngIf="f.name.errors.required">Name is required</div>
      </div>
    </div>

    <div class="form-group">
      <label>Address</label>
      <input #address type="text" [(ngModel)]="selectedDev.address" formControlName="address" class="form-control" [ngClass]="{ 'is-invalid': f.address.errors }"/>
      <div *ngIf="f.address.errors" class="invalid-feedback">
        <div *ngIf="f.address.errors.required">Address is required</div>
      </div>
    </div>

    <div class="form-group">
      <label>Email</label>
      <input #email type="email" [(ngModel)]="selectedDev.email" formControlName="email" class="form-control" [ngClass]="{ 'is-invalid': f.email.errors }"/>
      <div *ngIf="f.email.errors" class="invalid-feedback">
        <div *ngIf="f.email.errors.required">Email is required</div>
      </div>
    </div>

  </div>
  <div class="modal-footer">
    <button type="submit" [disabled]="Loading || f.name.errors || f.address.errors || f.email.errors" class="btn btn-outline-dark" (click)="modal.close('Save click'); ">Save Developer
  </div>
</form>
```

Figure 7: Exemplo da utilização de *Two-way Binding*

Ao introduzir dados nas páginas gerais utilizou-se principalmente a *Interpolation*, permitindo que uma variável do *Angular* apareça com as suas informações na página.

```
</div>
<div class="col-md-10">
  <h3 style="..." class="float-left" ><strong> {{ developer.name }}</strong></h3>
  <div class="clearfix"></div>
  <div class="col-md-2" style="...">
    <strong>Email </strong>
  </div>
  <div class="col-md-10" style="...">
    <span>{{ developer.email }}</span>
  </div>
  <div class="clearfix"></div>
  <div class="col-md-2" style="...">
    <strong>Address </strong>
  </div>
  <div class="col-md-10" style="...">
    <span>{{ developer.address }}</span>
  </div>
</div>
</div>
```

Figure 8: Exemplo da utilização de *Interpolation*

Como forma de aproveitar os dados fornecidos pelos *WebServices* da *DRF*, tentou-se procurar uma solução para conseguir passar informação entre *Parent* e *Child Components*. A solução encontrada foi a utilização dos Decoradores *@Input()* e *@Output()* aliados a *Property* e *Event Bindings*, o que facilmente permite o fluxo de dados, tal como é representado na imagem a seguir:

```
</div>
<div class="container text-right mt-3" style="padding-bottom: 20px">

  <a *ngIf="has_review; else noreview" routerLink="review" href="" type="button" class="site-btn">Edit Review</a>
  <app-delete-review *ngIf="has_review" [clientReview]="has_review" (deleteReviewEvent)="updateReviewList($event)"></app-delete-review>
</div>
```

Figure 9: Fluxo de Informação entre Parent e Child Components

4.2 Serviços

Como já foi referido, utilizou-se Serviços para a obtenção dos dados que são disponibilizados pela *REST API*, através de chamadas à mesma. De notar que, para a obtenção deste dados, todos os pedidos *HTTP* são interceptados por um *interceptor*, que tem o papel de introduzir o *token* de autenticação nos *headers* destes pedidos. Após isto acontecer o pedido segue naturalmente para a *API*.

4.2.1 Obtenção de dados

Na imagem seguinte, é possível ver um exemplo de como se implementou os *Services* do *Angular*. Como é possível observar, a utilização destes é combinado com o uso de *Observables*, permitindo a um componente dizer que quer uma informação sobre um objeto e só recolhe os dados quando estes chegarem, através de uma arquitetura *Publish-Subscribe*. Além disto, é também utilizada *Dependency Injection* de forma a que, quando este serviço for utilizado por *components* que necessitem de dados da *API*, o serviço seja injetado nesses *components*. Assim, é permitida a reutilização de *components* e impedir o uso de dependências *hard-coded*.

```
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs';
import { Product } from '../../models/product';
import { environment } from '../../environments/environment';
import { HttpClient } from '@angular/common/http';
import { Developer } from '../../models/developer';

@Injectable({
  providedIn: 'root'
})
export class DeveloperService {

  constructor(private http: HttpClient) { }

  createDeveloper(dev: Developer): Observable<Developer>{
    const url = environment.baseUrl + 'developercre';
    return this.http.post<Developer>(url, dev);
  }

  updateDeveloper(dev: {}, id: number ): Observable<Developer>{
    const url = environment.baseUrl + 'developerupd/' + id;
    return this.http.put<Developer>( url, dev, environment.httpOptions);
  }

  getDeveloper(id: number): Observable<Developer>{
    const url = environment.baseUrl + 'developer/' + id;
    return this.http.get<Developer>(url);
  }

  getDevelopers(): Observable<Developer[]>{
    const url = environment.baseUrl + 'developers';
    return this.http.get<Developer[]>(url);
  }

  getDevelopersP(page: number): Observable<Developer[]>{
    const url = environment.baseUrl + 'developers?page=';
    return this.http.get<Developer[]>(url);
  }
}
```

Figure 10: Implementação de um Serviço

4.2.2 Sistema de avisos

É importante salientar neste documento um serviço específico utilizado, o *Shared Service* (assim denominado por nós), que é um serviço que acaba por ser utilizado em muitos dos componentes. Neste serviço são implementados vários métodos que permitem a subscrição de eventos e o envio de eventos para os componentes que o subscreveram. Assim, foi permitido mostrar mensagens de Sucesso e Erro, através da utilização do componente *Alert-Component* como subscritor de eventos de alertas, permitindo a reutilização de código para *Feedback* ao utilizador final.

```
sendUserEvent(): void {
  this.subject.next();
}

getUserEvent(): Observable<any> {
  return this.subject.asObservable();
}

// enable subscribing to alerts observable
onAlert(id: string = this.defaultId): Observable<Alert> {
  return this.subjectAlert.asObservable().pipe(filter( predicate: x => x.id === id));
}

// convenience methods
success(message: string, options?: any): void {
  this.alert(new Alert( {init: {...options, type: AlertType.Success, message}}));
}

error(message: string, options?: any): void {
  this.alert(new Alert( {init: {...options, type: AlertType.Error, message}}));
}
```

Figure 11: Implementação do *Shared Service*

Este serviço foi também utilizado para evitar que seja possível o utilizador estar a visualizar uma página à qual não tem acesso, por exemplo, caso o *admin* esteja na página de administrador e este faça *logout* através da *navbar*, o componente desta envia um evento para avisar os componentes da página de administrador que o utilizador em questão já não tem permissão de visualização, existindo um redireccionamento.

```
logout(): void {
  this.authService.logout();
  this.client = undefined;
  this.sharedService.sendUserEvent();
}
```

Figure 12: Função de *logout* do componente da *navbar* que envia um evento para os componentes verificarem a sessão

5 Execução do sistema

5.1 Localmente

Para executar o serviço localmente é necessário executar tanto a aplicação *web* como o serviço *REST*.

Começando pela aplicação *web* em *Angular*, é necessário instalar as dependências deste projeto, executando ***npm install*** na raiz do mesmo.

No caso do serviço *REST* em *Django*, é necessário instalar os ***requirements*** que se encontram no ficheiro ***requirements.txt*** na raiz deste projeto.

Feito isto, basta executar ambos os projetos através da linha de comandos ou de um *IDE*.

5.2 Deploy

Como especificado pelo professor, o *deploy* foi realizado através das plataformas *Heroku* e *Python Anywhere* para a aplicação *web* em *Angular* e para o serviço *REST* em *Django REST Framework*, respetivamente.

O acesso ao sistema pode ser feito através dos seguintes endereços, contudo é importante referir que a ligação aos *websites* terá de ser feita **apenas** por HTTP:

- **Aplicação Web** - <http://app-store-frontend.herokuapp.com>
- **REST API** - <http://hugofpaiva.pythonanywhere.com>
- **Documentação da REST API** - <http://hugofpaiva.pythonanywhere.com/swagger>

5.3 Utilizadores

Em ambos os cenários de execução podem ser usados estes utilizadores:

Administrador

- **username:** admin
- **password:** admin

Clientes

- **username:** user1,user2,...,user10
- **password:** admin111

6 Conclusão

Para terminar, pensa-se que, de acordo com as metas estabelecidas pelo docente, o trabalho foi bem sucedido. Foram aprofundados conceitos relativos às tecnologias *Angular* e *Django Rest Framework* que sem dúvida será uma mais valia para o futuro profissional dos membros do grupo.

Alem disso, em termos de trabalho, foi criado um bom ambiente de equipa e uma boa gestão de *backlog* que permitiu alcançar os objetivos de uma forma mais rápida e estável.