Universidade de Aveiro

Tecnologias de Programação Web

Rest Review

Relatório Final

Autores :
89234 João Marques
88931 Vasco Ramos

Professor :
Hélder Zagalo

17 de Dezembro de 2019

Conteúdo

Lista de Códigos

Lista de Tabelas

1 Introdução

No seguimento do plano curricular da disciplina de Tecnologias de Programação Web, da Licenciatura em Engenharia Informática, da Universidade de Aveiro, este relatório é o resultado da execução do segundo trabalho prático e tem como principal objetivo documentar as partes da nossa solução que achamos essenciais.

O Rest Review é um sistema que permite aos seus utilizadores pesquisar por restaurantes, filtrá-los e ordená-los conforme preferirem. Permite também a criação de *reviews* que têm em conta uma série de parâmetros de forma a permitir uma avaliação mais completa destes.

O sistema em análise tem 2 tipos de utilizadores diferentes: ***Clients*** e ***Owners***.

Os ***clients*** podem, como referido acima, pesquisar por restuarante, filtrar, escrever *reviews* e também adicionar os restaurantes que queiram à sua lista de favoritos.

Por outro lado, os ***owners*** podem registar os seus próprios restaurantes, editá-los e eliminá-los, bem como a pesquisa, filtragem e ordenação dos restaurantes presentes na plataforma.

Considerando a escalabilidade da solução desenvolvida, optámos por desenvolver um sistema composto por 3 módulos separados, que podem ser executados em ambientes distintos:

- Interface Gráfica;
- REST API;
- Base de Dados.

2 Angular

Angular é uma plataforma open-source de aplicações web com front-end baseado em TypeScript, liderada pela Google.

No segundo projeto, Angular foi a tecnologia utilizada para desenvolver a interface e, para tal, foram implementadas as seguintes funcionalidades providenciadas pelo Angular:

- **Components**: Usámos components para todas as views e toda a lógica a estas associada;
- **Data Binding**: Data binding foi usado, principalmente, para mostrar dados persistentes;
- **Directives**: Usámos directives para munir os elementos do DOM com comportamento adicional;
- **Service**: Todos os dados vindos do servidor foram obtidos através de services;
- **Dependency Injection**: Injeção do service em diversos componentes;
- **Routing**: Toda a navegação dentro da plataforma é feita através de routing;
- **Bootstrap**: Bootstrap é utilizado como ferramenta de estilo para a interface;
- **Observables**: Usados, principalmente, para lidar com programação assíncrona.

2.1 Alterações face ao primeiro projeto

2.1.1 Two-way Binding e Event Binding

Enquanto que, em Django, a parte lógica associada a uma view apenas é atualizada recorrendo a métodos adicionais, em Angular, através do two-way binding, conseguimos facilmente atualizar uma variável de acordo com a view.

Devido à simplicidade providenciada por esta funcionalidade, Data Binding foi fortemente utilizado em casos como, por exemplo, os formulários. Os Django Forms utilizados no primeiro projeto acabaram por ser substituídos com recurso à utilização do ngSubmit (event binding) e do two-way binding, como foi referido anteriormente.

```
<form (ngSubmit)="registerRestaurant()" enctype="multipart/form-data">
      <span class="login100-form-title pb-5">New Restaurant</span>

      <div class="follow-img">
        <img alt="#" class="img-fluid" id="photo"
            src="{{imageSrc}}" style="border-radius:
            0">
      </div>
      <div class="mt-3 mb-5 text-center">
        <button (click)="triggerInput()" class="btn btn-primary btn-lg"
            type="button">Adicionar foto</button>
        <div style="visibility: hidden;">
          <input (change)="handleInputChange($event)" accept="image/*"
              id="photo_input" name="photo" type="file">
        </div>
      </div>

      <label for="id_name">Name:</label>
      <div class="wrap-input100">
        <input [(ngModel)]="rest.name" class="input100" id="id_name"
            name="name" placeholder="Insert Name" required type="text">
        <span class="focus-input100"></span>
      </div>
      <div *ngIf="name.invalid && (name.dirty || name.touched)"
          class="alert alert-danger">
        Name is required.
      </div>
```

Código 1: Formulário para adicionar um novo restaurante (parcial)

2.1.2 NgIf e NgFor

No que toca a diretivas, o angular apresenta vários tipos, sendo que as mais utilizadas no nosso projeto foram as diretivas estruturais, mais especificamente, o **NgIf** e o **NgFor**.

Estas permitem facilmente substituir as template tags provenientes do Django e renderizar de uma forma naturalmente fluída o template adequado, bem como o seu conteúdo.

```
<div *ngFor="let rest of rests"[ngClass]="rests.length != 1 ? 'col-md-12
    col-sm-12 col-lg-6 col-xl-6 featured-responsive': 'col-12
    featured-responsive'" style="margin: auto">
        <div class="featured-place-wrap">
          <a routerLink="/restaurant/{{rest.id}}">
            <img alt="#" class="img-fluid"
                src="{{rest.photo}}" style="height:
                35em">
            <span *ngIf="rest.avg_score < 5.5"
                class="featured-rating-orange">{{ rest.avg_score }}</span>
            <span *ngIf="rest.avg_score >= 5.5"
                class="featured-rating-green">{{ rest.avg_score }}</span>
          </a>
          <div class="featured-title-box">
            <div class="container row">
              <a routerLink="/restaurant/{{rest.id}}">
                <h6>{{ rest.name }}</h6></a>
              <div *ngIf="rest.is_favorite != null">
                <a (click)="rest.is_favorite ? removeFromFavorites(rest,
                    $event) : addToFavorites(rest, $event)"
                   [ngStyle]="{'color': rest.is_favorite ? 'red' : ''}"
                       class="ti-heart" href="javascript:void(0)"></a>
              </div>
            </div>
            <a routerLink="/restaurant/{{rest.id}}">
              <p>Restaurant | {{ rest.n_of_reviews }} Reviews</p></a>
          </div>
        </div>
</div>
```

Código 2: Listagem de todos os restaurantes na plataforma (parcial)

2.1.3 Service e Dependency Injection

Por fim, como já foi referido, todas as chamadas à REST API são feitas a partir de *services* criados na aplicação Angular. Para que isto aconteça, é utilizada **Dependency Injection** nos componentes que precisam de dados provenientes do servidor, injetando o service nesses mesmos componentes. Através deste processo, conseguimos tornar os componentes mais sustentáveis, reusáveis e testáveis através da remoção de dependências *hard coded*.

```
import {ClientService} from '../../services/client.service';

(...)

export class ClientFavoritesComponent implements OnInit {
  rests: Restaurant[];

  constructor(private clientService: ClientService, private router:
      Router) {
  }

  ngOnInit() {
    this.getFavorites();
  }

  getFavorites(): void {
    this.clientService.getFavorites()
      .subscribe(
        rests => {
          this.rests = rests.message;
          if (rests.token !== 'null') {
            localStorage.setItem('token', rests.token);
          }
        }, (error: HttpResponse<any>) => {
          localStorage.removeItem('token');
          localStorage.removeItem('username');
          this.router.navigateByUrl('login');
        });
    }
```

Código 3: Injeção de um serviço e uso posterior do mesmo

3 Django REST Framework

A Django Rest Framework (DRF) é uma biblioteca do Framework Django utilizada para a implementação de REST APIs.

Esta permite-nos a utilização de políticas de autenticação e autorização, bem como serialização de dados provenientes de uma base de dados.

3.1 Alterações face ao primeiro projeto

3.1.1 Autenticação

O maior desafio da implementação do sistema, utilizando uma REST API, prende-se com o facto da autenticação de utilizadores ter de ser efetuada ao nível da API, em vez de na interface, através da qual os utilizadores acedem ao sistema.

Para tal, recorremos a *AuthTokens*.

Desta forma, sempre que o utilizador envia os seus dados de login, este recebe um Token de Autenticação que deve acompanhar todos os pedidos à API que forem efetuados posteriormente. Após um dado período (o tempo de expiração do token) este é atualizado e a utilização do token anterior já não é permitida (o acesso é barrado).

O Token fica guardado (e atualizado) do lado do cliente e é eliminado aquando do logout do mesmo, sendo, então, criado um novo token aquando de uma nova sessão.

```python
def expires_in(token):
    time_elapsed = timezone.now() - token.created
    left_time = timedelta(seconds=settings.TOKEN_EXPIRED_AFTER_SECONDS) -
        time_elapsed
    return left_time


# token checker if token expired or not
def is_token_expired(token):
    return expires_in(token) < timedelta(seconds=0)


# If token is expired new token will be established:
# If token is expired then it will be removed
# and new one with different key will be created
def token_expire_handler(token):
    is_expired = is_token_expired(token)
    if is_expired:
        token.delete()
        token = Token.objects.create(user=token.user)
    return is_expired, token


class ExpiringTokenAuthentication(TokenAuthentication):
    """
    If token is expired then it will be removed
    and new one with different key will be created,
    this new one will be given to the user only the 1st time
    """
    def authenticate_credentials(self, key):
        try:
            token = Token.objects.get(key=key)
        except Token.DoesNotExist:
            raise AuthenticationFailed("Token invalido!")
        if not token.user.is_active:
            raise AuthenticationFailed("Credenciais invalidas!")
        is_expired, token = token_expire_handler(token)
        return token.user, token
```

Código 4: Classe que trata a autenticação de um utilizador na plataforma

```python
@api_view(["POST"])
@permission_classes((AllowAny,))
def login(request):
    """
    Functions that logs in the requester user. If there is a existing
        token, it returns that one.
    Otherwise, creates a new token for valid users.
    :param request: Who has made the request.
    :return: Response 200 with user_type, data and token, if everything
        goes smoothly.
    Or Response 404 for not found error.
    """
    login_serializer = UserLoginSerializer(data=request.data)
    if not login_serializer.is_valid():
        return Response(login_serializer.errors,
            status=HTTP_400_BAD_REQUEST)

    user = authenticate(
        username=login_serializer.data['username'],
        password=login_serializer.data['password']
    )
    if not user:
        message = "Credenciais para o login invalidas!"
        return Response({'detail': message}, status=HTTP_404_NOT_FOUND)

    # TOKEN STUFF
    token, _ = Token.objects.get_or_create(user=user)

    # token_expire_handler will check, if the token is expired it will
        generate new one
    is_expired, token = token_expire_handler(token)
    user_serialized = UserSerializer(user)

    return Response({"user_type": get_user_type(user.username), "data":
        user_serialized.data, "token": token.key}, status=HTTP_200_OK)
```

Código 5: Processo de login de um utilizador

3.1.2 URLs e Views

Na criação da REST API foram disponibilizados novos métodos, de forma a permitir o envio de todos os dados necessários para a visualização de informação, bem como para a introdução e atualização da mesma.

Desta forma, disponibilizámos métodos GET, POST, PUT e DELETE.

Tendo em conta a separação lógica e a melhor organização do código, criámos um novo ficheiro python (*queries.py*) cuja única função é a interação com a base de dados e a serialização dos dados necessários.

O ficheiro *views.py*, por sua vez, invoca métodos do *queries.py*, avaliando sempre se o utilizador tem autorização para efetuar o pedido em causa.

3.1.3 Novos módulos de Python

Para o bom funcionamento da REST API foi necessário instalar os seguintes módulos:

- *djangorestframework*
- *django-cors-headers*
- *mysqlclient*

3.1.4 Acesso a Bases de Dados Remotas

A REST API desenvolvida está apta para interagir com bases de dados remotas. Uma vez que a base de dados remota utilizada era uma base de dados MySQL, foi necessário instalar o módulo mysqlclient para que fosse possível interagir com ela.

Isto permite-nos facilmente criar uma aplicação de 3 camadas, o que traz claras vantagens para a escalabilidade da mesma.

Seguem-se as configurações necessárias para permitir o acesso à base de dados remota. Estas devem ser feitas no ficheiro *settings.py*:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'jmarques$yelp',
        'USER': "jmarques",
        'PASSWORD': "tpwbdpassword",
        'HOST': "jmarques.mysql.pythonanywhere-services.com",
        'PORT': "3306",
    }
}
```

Código 6: Configurações para acesso à base de dados remota

4 Funcionalidades

- **/reload_database** (REST API)**:** Dar reset à base de dados com informação default.
- **/register:** Página com opção de registo com Client ou Owner.
- **/register-client:** Registar conta como Client.
- **/register-owner:** Registar conta como Owner.
- **/login:** Efetuar login na plataforma.
- **/:** Apresenta a página principal com os 2 melhores restaurants e possibilidade de pesquisa.
- **/list-restaurants:** Listagem de todos os restaurantes com a possibilidade de pesquisar por nome ou cidade e ordenar por uma série de parâmetros relacionados com a qualidade dos restaurantes.
- **/owner/<name>:** Aceder ao perfil público de um dado Owner (informações e restaurantes desse owner).
- **/restaurant/<number>:** Aceder à página de detalhe de um dado restaurante.

As restantes funcionalidades precisam sempre de um login prévio:

- **/new-restaurant:** Registo de um novo restaurant no sistema (apenas Owners têm acesso a esta funcionalidade).
- **/client:** O Client acede ao seu próprio perfil (com opção de editar campos ou eliminar conta).
- **/owner:** O Owner acede ao seu próprio perfil (com opção de editar campos ou eliminar conta).
- **/my-restaurants:** O Owner acede a uma listagem de todos os seus restaurantes (apenas Owners têm acesso a esta funcionalidade).
- **/my-favorites:** O Client acede a uma listagem de todos os restaurantes na sua lista de favoritos (apenas Clients têm acesso a esta funcionalidade).
- **/list-restaurants:** Listagem de todos os restaurantes com a possibilidade de pesquisar por nome ou cidade e ordenar por uma série de parâmtetros relacionados com a qualidade dos restaurantes.

- **/edit-restaurant/<number>:** O Owner acede a um restaurante seu (com opção de editar campos ou eliminar esse restaurante).

Funcionalidades não disponíveis por URL:

- Um Client pode adiconar restaurantes à sua lista de favoritos nas páginas *list-restaurants* e *home*.
- O display da localização de cada restuarante é implementado com a utilização da API do Google Maps.
- Em cada funcionalidade de edição (Client, Owner e Restaurant) é, também, possível eliminar essa entidade, que irá comportar uma eliminação cascade em todas as entidades devidas.

5 Acessos

5.1 Clients

Username	Password
client1@ua.pt	client1
client2@ua.pt	client2
client3@ua.pt	client3

Tabela 1: Acessos default de clientes

5.2 Owners

Username	Password
owner1@ua.pt	owner1
owner2@ua.pt	owner2
owner3@ua.pt	owner3

Tabela 2: Acessos default de owners

6 Notas Finais

1. Git:
    - Front-end: https://gitlab.com/tpw-group/project-2-frontend
    - Back-end: https://gitlab.com/tpw-group/tpw-project-2
2. REST Api: https://jmarques.pythonanywhere.com
3. Interface: https://rest-review.herokuapp.com

Em retrospectiva, consideramos que todos os objetivos relativos a este trabalho foram atingidos.

Em suma, concluímos que, com este trabalho, os conhecimentos de Django e Angular foram melhorados consolidados e que a capacidade de pesquisa e autonomia foram, também elas, bem desenvolvidas, na tentativa de obter soluções face aos problemas encontrados.