Tecnologias e Programação Web, Projeto 02

- The aim of this second project was to addapt the one developed previously, which only used Django.
- Thus, the all-new version of **Mercadinho dos Cliques** is an Angular-App, using a Django-Rest API as the backbone for the majority of interactions. Similarly to before, it emulates an online grocery-store, providing all the functionalities needed to manage an online store, and buying items.
- All the functionalities you expect to find are covered and implemented.

Features

There are two types of users in our system: an *administrator* and a *regular user*. A *regular user* cannot do everything an *administrator* can, and vice-versa. However, there are a couple of key-features that are available to both user-groups.

A brief breakdown of each can be found in the following table

TYPE	CAN DO	CAN'T DO
ADMIN	View all products in store; View a specific product details; Search for a product/brand; View the stores' stock; Edit a product; Add a new product	Purchase a(several) products; Place an order; Review its past orders; Manage its personal account
CLIENT	View all products in store; View a specific product details; Search for a product/brand; Purchase a(several) products; Place an order; Review its past orders; Manage its personal account	View the stores' stock; Edit a product; Add a new product
NOT LOGGED IN	Create an Account; View all products in store; View a specific product details; Search for a product/brand	Purchase a(several) products; Place an order; Review its past orders; Manage its personal account; View the stores' stock; Edit a product; Add a new product

Technological Stack

1. API (DRF)

As previously indicated, Django was chosen as the technology to create a straightforward REST API.
This abstraction-layer mediates all interactions between the frontend and the backend.

1) Authentication

When an user logs into the app with a valid account, a *JSON Web-Token* is encoded and issued to the user.
Following that, the token is returned to the frontend and temporarily saved as a *cookie*, with an expiration time.
From moment the *cookie* is saved in the frontend, all requests to the API that require authorization are accompanied by it.

As a result, the user's legitimacy is ensured.

```python
@permission_classes([AllowAny])
class LoginView(APIView):

    def post(self, request):
        password = request.data['password']
        username = request.data['username']

        user = User.objects.filter(username=username).first()

        if user is None:
            raise AuthenticationFailed('User not found!')

        if not user.password == password:
            raise AuthenticationFailed('Incorrect password!')

        payload = {
            'id': user.id,
            'exp': datetime.datetime.utcnow() + datetime.timedelta(minutes=60),
            'iat': datetime.datetime.utcnow()
        }

        token = jwt.encode(payload, 'secret', algorithm='HS256')

        response = Response()

        response.set_cookie(key='jwt', value=token, httponly=True)

        response.data = {
            'jwt': token,
            # 'user_type': user_type
        }

        return response
```

***2)* Authorization**

There is still a distinction between user kinds even after logging in.
As a result, another function was written to validate the *user type*.

This ensures that a person can only access what is granted to him by his permissions.

```python
@api_view(['GET'])
def get_user_type(request):
    username = request.GET.get('username', '')
    user_type = None
    if username:
        if Customer.objects.filter(user__username=username).exists():
            user_type = "customer"
        if Manager.objects.filter(user__username=username).exists():
            user_type = "manager"

    return Response({'user_type': user_type})
```

***3)* General Views**

Views are an essential component of the API since they provide access to all requests. A view is called for each endpoint defined in the URLs, which describes the method, the authorizations, and the request is sent if all of the criteria are satisfied. The image below shows an example of of a view that has been defined.

```python
# web service to get products of a specific brand
@api_view(['GET'])
def get_brandproducts(request):
    id = int(request.GET['id'])
    try:
        products = Product.objects.filter(brand__id=id)
    except Product.DoesNotExist:
        return Response(status=status.HTTP_404_NOT_FOUND)
    serializer = ProductSerializer(products, many=True)
    return Response(serializer.data)
```

***4)* Serializers**

In order to transfer information between the backend and the frontend, all the information must be processed by *serializers*.
Serializers alow you to process information, and specify which fields should be

returned and in what format.

```python
class ProductSerializer(serializers.ModelSerializer):
    You, a month ago | 1 author (You)
    class Meta:
        model = Product
        fields = ('id', 'name', 'description', 'category', 'brand', 'price', 'quantity', 'image')

    def to_representation(self, instance):
        self.fields['brand'] =  BrandSerializer(read_only=True)
        self.fields['category'] =  CategorySerializer(read_only=True, many=True)
        return super(ProductSerializer, self).to_representation(instance)

    def create(self, validated_data):
        category_data = validated_data.pop('category')
        product = Product.objects.create(**validated_data)
        for category_data in category_data:
            cat = Category.objects.get(id=category_data.id)
            product.category.add(cat)
        return product
```

---

2. Angular

The technology Angular was used to create the whole frontend of this application. This framework employs a Component-Based-Architecture and the Typescript language, utilizing all of the features that they have to offer.

1) **Data Presentation**

To primarily populate forms with data, *Two-way Binding* was utilized, allowing automatic data update between the data model and the view via the *ngModel directive*. This binding type combines *Property-Binding* and *Event-Binding*.

```html
<mat-form [formGroup]="editForm">
    <div class="row">
        <input hidden [(ngModel)]="data.product.id" name="id" id="id" type="number" class="fadeIn second" placeholder="Id do Produto" formControlName="id">
        <br>
        <input [(ngModel)]="data.product.name" name="name" id="name" type="text" class="fadeIn second" placeholder="Nome do Produto" formControlName="name">
        <mat-hint class="form-hint-input">Nome do Produto</mat-hint>
        <br><br>
        <input [(ngModel)]="data.product.description" name="description" id="description" type="text" class="fadeIn second" placeholder="Descrição" formControlName="description">
        <mat-hint class="form-hint-input">Descrição</mat-hint>
        <br><br>
        <select multiple [compareWith]="compator" [(ngModel)]="data.product.category" name="category" id="category" type="text" class="fadeIn third" placeholder="Categoria" formControlName="category">
            <option *ngFor="let item of categories" [ngValue]="item.id" >
                {{ item.name }}
            </option>      You, 2 days ago • edit products done; minor refactoring
        </select>
        <mat-hint class="form-hint-input">Categoria</mat-hint>
        <br><br>
        <select [compareWith]="compator" [(ngModel)]="data.product.brand" name="brand" id="brand" type="text" class="fadeIn third" placeholder="Marca" formControlName="brand">
            <option *ngFor="let item of brands" [ngValue]="item.id" >
                {{ item.name }}
            </option>
        </select>
        <mat-hint class="form-hint-input">Marca</mat-hint>
        <br><br>
        <input [(ngModel)]="data.product.price" matInput name="price" id="price" type="number" step=".01"  class="fadeIn third" placeholder="Preço" formControlName="price">
        <mat-hint class="form-hint-input">Preço</mat-hint>
        <br><br>
        <input [(ngModel)]="data.product.quantity" matInput name="quantity" id="quantity" type="number" class="fadeIn third" placeholder="Quantidade" formControlName="quantity">
        <mat-hint class="form-hint-input">Quantidade</mat-hint>
        <br><br>
        <input [(ngModel)]="data.product.image" type="url" name="image" id="image" class="fadeIn third" placeholder="URL Imagem" formControlName="image">
        <mat-hint class="form-hint-input">URL Imagem Representativa</mat-hint>
        <br><br>
    </div>
</mat-form>
```

Accordingly, in order to input data on the pages developed, *Interpolation* was used - as it allows for an Angular variable to be displayed with it's informations.

```
<div class="container">
    <h1 id="products_title">Produtos da Marca {{ brand.name }}</h1>
    <div class="row">
        <ng-template let-product ngFor [ngForOf]="products">
            <div class="col-md-4">
                <mat-card class="product-card">
                    <img mat-card-image src="{{product.image}}" alt="Product Image" routerLink="/product/{{product.id}}">
                    <mat-card-content routerLink="/product/{{product.id}}">
                        <mat-card-title>{{ product.name }}</mat-card-title>
                        <mat-card-subtitle *ngFor="let category of product.category ">{{ product.brand.name }} - {{ category.name }}
                            <br> {{ product.price | currency:"EUR" }}
                        </mat-card-subtitle>
                    </mat-card-content>
                    <mat-card-actions>
                        <button mat-button (click)="addToShoppingCart(product)">
                            <mat-icon color="accent">add_shopping_cart</mat-icon>
                        </button>
                    </mat-card-actions>
                </mat-card>
            </div>
        </ng-template>

    </div>
```

The decorators *@Input()* and *@Inject()* were used to transmit information between *Parent and Child Components*, allowing for a straightforward data-flow.

2) **Services**

Services were a critical component to create since they enable for the retrieval of data made available via the Rest API.
Their usage is paired with *Observables*, using a Publish-Subscribe architecture.
Dependency Injection is also utilized - when this service is accessed by components that require API data, it is injected into them.

This eliminates the need for hard-coded dependencies.

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

  constructor(private http: HttpClient) { }

  getBrands(): Observable<any[]> {
    const url = environment.API_BASE_URL + 'brands';
    return this.http.get<any[]>(url);
  }

  getBrandId(id: number): Observable<Brand> {
    const url = environment.API_BASE_URL + 'brand?id=' + id;
    return this.http.get<Brand>(url);
  }

  getBrandProducts(id: number): Observable<Product[]> {
    const url = environment.API_BASE_URL + 'productsofbrand?id=' + id;
    return this.http.get<Product[]>(url);
  }

}
```

---

3. Database

In order to fully implement a 3-Tier Architecture, the database that drives this application was deployed on a *Raspberry Pi*, and is accessed directly on Django.

The steps required to access the database and make it available to the Django-App are listed below.

1. Access the Raspberry Pi remotely

   ```
   $ ssh ubuntu@161.230.191.221
   ```

2. When the prompt to insert a password appears, please type **raspberry**, and you'll be *inside* the Raspbery Pi.

   ```
   $ sudo docker exec -it tpw bash
   $ mysql -u tpw -p
   ```

3. When the prompt to insert the database's password appears, please type **mercadinhocliques**

   ```
   $ use tpw;
   ```

4. Finally, the database is now ready to be used!

Accessing the App

The app has been deployed in Heroku, and may be accessed via the following link Mercadinho dos Cliques.
The backend of the app was deployed on a Raspbery Pi, and is accessible on the DNS - 94.60.22.100. The following link Products endpoint is an example of an endpoint.

You can register yourself in order to gain access to some functionalities (as a client), but we also have two accounts listed bellow, created and used in development :)

TYPE	USERNAME	PASSWORD
ADMIN	gestor	gestor
CLIENT	cliente	cliente

DEVELOPERS NOTE

We also want to apologize ourselves of the small bugs found during the presentation of the project.

We weren't expecting it to happen, and we went digging in order to try and find it, in order to fix it.

Sadly, the issue only seems to be present in the deployed version, and using the application locally brought up zero issues regarding that.

Authors

Lucas Sousa (nmec 93019)
Francisca Barros (nmec 93102)
G03, TPW-UA, 2020/2021