Application de gestion de la M2L – Documentation technique

Sommaire :

I) Langages utilisés :

Dart :

- Créer les différentes pages de l'application ainsi que leurs routes
- Communiquer avec l'API REST afin de récupérer et d'ajouter des informations dans la base de données du site

MySQL :

- Créer la base de données de l'application

NodeJs :

- Créer l'API REST et permettre la liaison entre le front et le back

II) Déploiement du Backend :

La partie backend, réalisée en NodeJS, du projet est déployée sur une machine Ubuntu. En suivant les étapes suivantes :

1. Mettre à jour l'environnement Ubuntu.

sudo apt update && sudo apt upgrade

2. Installer Git, NodeJS, npm et mariadb-server sur la machine.

sudo apt install git nodejs npm mariadb-server

3. Configurer mariadb-server.

sudo mysql_secure_installation

4. Récupérer le backend sur GitHub ou GitLab et le mettre dans le dossier /opt.

Cd /opt/

Sudo git clone nomDuRepository

5. Exporter votre base de données en .sql dans la machine Ubuntu.

6. Déployer la base de données sur le server MySQL d'Ubuntu et créer un utilisateur qui aura accès à la base de données.

- *sudo mysql -u root -p*
- *CREATE DATABASE votreDatabase;*
- *use nom BDD;*
- *sudo mysql -u root -p votreDatabase < chemin/votreDatabase.sql*
- *GRANT ALL PRIVILEGES ON votreDatabase.* TO votreUser@localhost identified by "votreMotDePasse";*
- *FLUSH PRIVILEGES;*

7. Créer le fichier .env dans le dossier du projet et lui donner la configuration nécessaire.

sudo nano /opt/votreServeurNode/.env

8. Essayer de lancer votre serveur et d'y accéder via votre ordinateur.

Pour voir l'adresse IP de la machine : {ip a}. Il faut d'abord s'être mis en connexion NAT sur la configuration de la VM et avoir effectué une redirection de port vers celui sur lequel est configuré votre serveur.

Pour lancer votre serveur : npm start (à l'emplacement où se trouve votre server.js)

9. Mettre en place le fichier permettant de lancer le serveur au démarrage. Créer un fichier serveurnode.service dans le dossier etc/systemd/system/

sudo nano /etc/systemd/system/serveurnode.service

10. Dans ce fichier, modifier la configuration suivante pour qu'elle soit conforme pour votre machine. Il faut modifier le nom de l'user ainsi que les chemins vers le back de l'application.

```
[Unit]
Description=My Node.js App
After=network.target

[Service]
User=name
WorkingDirectory=/path/to/your/app
ExecStart=/usr/bin/node /path/to/your/app/server.js
Restart=always
RestartSec=10

[Install]
WantedBy=multi-user.target
```

11. Si le fichier est bien mis en place, actualisez les unités systemsd puis activez le service que vous venez de créer.

- *sudo systemctl daemon-reload*
- *sudo systemctl enable serveurnode.service*
- *sudo systemctl start serveur.service*

12. Démarrez le service puis redémarrer la machine. Si tout est bien réalisé, votre api se lancera au démarrage. S'il y a des erreurs, revoir les étapes précédentes.

III) Fonctionnement de l'application :

1. La connexion entre le frontend et l'API se fait via le fichier produit.dart.

On définit d'abord url de l'application en respectant bien le port sur lequel se trouve notre Back.

```
5   class Produit {
6     static String baseUrl = "http://localhost:8000"; //localhost:8000 pour mon api
```

On utilise ensuite les routes de l'API comme suit :

```
8     static Future<List> getAllProduit() async {
9       try {
10        var res = await http.get(Uri.parse("$baseUrl/produit")); // /routes de l'api
11        if (res.statusCode == 200) {
12          return jsonDecode(res.body);
13        } else {
14          return Future.error("erreur serveur");
15        }
16      } catch (err) {
17        return Future.error(err);
18      }
19    }
```

On prend ici l'exemple de la route qui récupère l'entièreté des produits de la base de données. Dans ce cas, c'est donc une méthode Get qui est utilisée. En cas de succès, celle-ci va afficher les données demandées, en cas d'erreur elle renverra « erreur serveur ».

La combinaison de « async » et « await » permet d'attendre de recevoir toutes les informations avant de poursuivre et de les renvoyer.

Les autres routes sont construites de la même façon mais peuvent utiliser une méthode différente ou une uri différente, comme avec l'id des produits pour les récupérer indépendamment.

2. On définit le chemin des différentes pages de l'application dans le fichier main.dart.

```dart
void main() => runApp(const MaterialApp(home: Home()));

class Home extends StatelessWidget {
  const Home({super.key});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'M2L catalog management',
      theme: ThemeData(
        primarySwatch: Colors.amber,
      ), // ThemeData
      routes: {
        '/': (context) => const Connexion(),
        '/produit': (context) => const AffichageProduit(),
        '/ajoutProduit': (context) => const Ajout(),
      },
    ); // MaterialApp
  }
}
```

3. Connexion à l'application :

La connexion est gérée par le fichier connexion.dart. Il est chargé d'envoyer les identifiant entrés par l'utilisateur et de les comparer à ceux présent dans la base de données

Pour cela, il faut initialiser des contrôleurs pour les champs de l'identifiant et du mot de passe.

```
12  class _ConnexionState extends State<Connexion> {
13    final GlobalKey<FormState> login = GlobalKey<FormState>();
14    final userController = TextEditingController();
46            Padding(
47              padding: const EdgeInsets.symmetric(horizontal: 15),
48              child: TextFormField(
49                controller: userController,
50                decoration: const InputDecoration(
51                    border: OutlineInputBorder(),
52                    labelText: 'Email',
53                    hintText: 'Entrez votre email'), // InputDecoration
54                validator: (value) {
55                  if (value == null || value.isEmpty) {
56                    return "Veuillez rentrer un email";
57                  }
58                  return null;
59                },
60              ), // TextFormField
61            ), // Padding
```

On appel ensuite ces contrôleurs pour qu'ils récupèrent les valeurs misent dans le formulaire de connexion.

4. Affichage des données :

Pour afficher les différents produits et leurs informations, on les récupère simplement en appelant la route getAllProduit du fichier produit.dart, puis on appel chaque donnée des produits via leurs noms dans la bases de données.

```
17    @override
18    void initState() {
19      super.initState();
20      _produit = Produit.getAllProduit();
21    }
```

```
38                  children: [
39                    Image.asset(snapshot.data![index]['img'],
40                      height: 250,
41                      width: 250,
42                    ), // Image.asset
43                    Text(snapshot.data![index]['nom'], // utilisation des champs de la BDD
44                        style: const TextStyle(fontSize: 20)), // Text
45                    Text(snapshot.data![index]['prix'].toString(), // utilisation des champs de la BDD
46                        style: const TextStyle(fontSize: 20)), // Text
47                    Text(snapshot.data![index]['description'], // utilisation des champs de la BDD
48                        style: const TextStyle(fontSize: 20)), // Text
49                    Text(snapshot.data![index]['stock'].toString(), // utilisation des champs de la BDD
50                        style: const TextStyle(fontSize: 20)), // Text
51                    Row(
52                    mainAxisAlignment: MainAxisAlignment.end,
```

5. Suppression de produit :

La suppression est gérée dans le même fichier que l'affichage, c'est-à-dire affichageProduit.dart. Chaque bouton « supprimé » est associé au produit à côté duquel il se trouve. Cliquer dessus ouvre un panneau qui permet de valider ou non. En validant, cela va appeler la route Delete qui va supprimer le produit selon l'id de l'article concerné.

```
child: ElevatedButton(
    child: const Text(
        'Oui'), // Text
    onPressed: () {
      Produit.Delete(
          context,
          int.parse(snapshot
              .data![index]
                  ["id"]
              .toString()));
    }), // ElevatedButton
```

6. Ajouter un produit :

Pour ajouter un produit, on initialise tout d'abord des contrôleurs pour chaque donnée à entrer.

```
11  class _AjoutState extends State<Ajout> {
12    final GlobalKey<FormState> login = GlobalKey<FormState>();
13    final nomController = TextEditingController();
14    final imageController = TextEditingController();
15    final prixController = TextEditingController();
16    final stockController = TextEditingController();
17    final descriptionController = TextEditingController();
```

Ensuite, pour chaque champ du formulaire, on attribut un contrôleur à une valeur afin qu'il puisse la récupérer et ajouter le produit dans la base de données avec toute les informations voulues.

```
52                  Padding(
53                    padding: const EdgeInsets.symmetric(horizontal: 15),
54                    child: TextFormField(
55                      controller: nomController,
56                      decoration: const InputDecoration(
57                          border: OutlineInputBorder(),
58                          labelText: 'Nom',
59                          hintText: 'Entrez un nom'), // InputDecoration
60                      validator: (value) {
61                        if (value == null || value.isEmpty) {
62                          return "Veuillez rentrer un nom";
63                        }
64                        return null;
65                      },
66                    ), // TextFormField
67                  ), // Padding
```

Une fois qu'on valide l'ajout, il faut faire attention à ce que les valeurs respectent bien le type prévu dans la Base de données. Comme elles sont toute des chaînes de caractères au moment de les rentrées dans le formulaire, il faut convertir celles qui doivent être des entiers par exemple.

```
145                     onPressed: () {
146                       if (login.currentState!.validate()) {
147                         Produit.ajout(context, nomController.text,
148                             imageController.text, double.parse(prixController.text),
149                             int.parse(stockController.text), descriptionController.text);
150                       }
151                     },
```

7. Modification de produit :

Le fonctionnement de la modification des produits est très similaire à l'ajout. Les seules différences sont :

- La route utilisée (ici on prend celle qui utilise la requête Update)
- On préremplit le formulaire avec les données déjà présentes

```
22    @override
23    void initState() {
24      super.initState();
25      // On récupère les informations de la question avec son id
26      // passé en paramètre
27      _produit = Produit.getProduit(widget.id);
28      _produit.then((value) => {
29            // On pré-remplit le formulaire avec les données récupérer de l'API
30            nomController.text = value[0]['nom'],
31            imageController.text = value[0]['img'],
32            prixController.text = value[0]['prix'],
33            stockController.text = value[0]['stock'],
34            descriptionController.text = value[0]['description'],
35          });
36    }
```