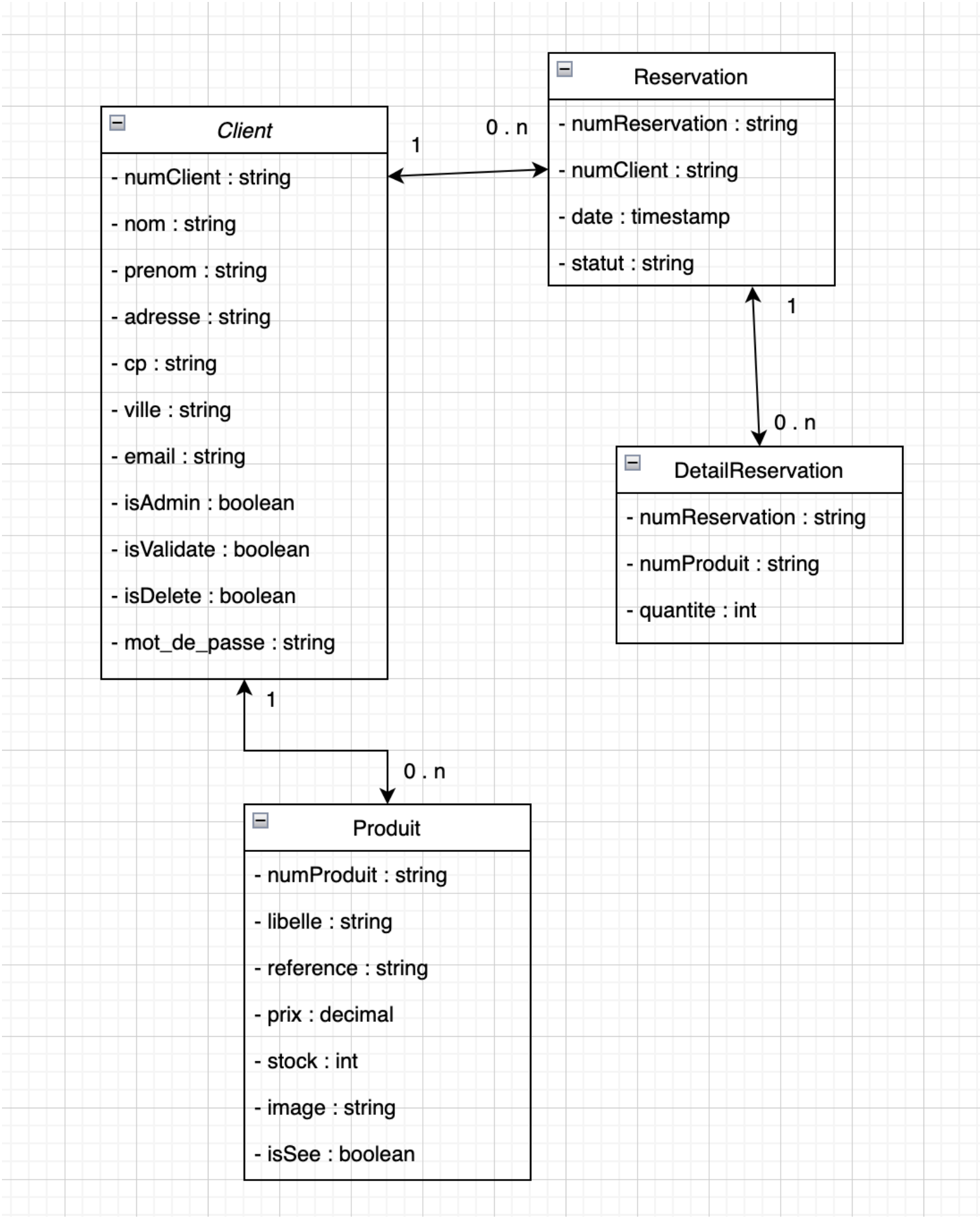
M2L AP_4

by Shop_def

Sommaire

Schéma de la Base de Données



Documentation de l'API

Connexion à la base de données

Avant toute chose il faut se connecter à la base de données créée précédemment et pour cela on fait appel à la fonction suivante mysql.

back > database > JS database.js > ...

```
1   const mysql = require('mysql');
2   require('dotenv').config();
3
4   const pool = mysql.createConnection({
5       host: process.env.DB_HOST,
6       database: process.env.DB_DTB,
7       user: process.env.DB_USER,
8       password: process.env.DB_PWD,
9       port: process.env.DB_PORT
10  })
11
12
13
14  module.exports =pool;
```

Une fois que nous avons accès à la base de données nous pouvons faire des requêtes et donc interagir avec celle-ci, grace à cela nous pouvons continuer notre projet.

Configuration Requise

- Node.js

- Express.js

- MySQL

Installation et Configuration

1. Installer Node.js depuis [le site officiel](https://nodejs.org/).

2. Cloner ou télécharger le projet depuis le référentiel.

3. Exécuter `npm install` pour installer toutes les dépendances.

4. Configurer la base de données MySQL en créant une base de données nommée `database` et en important le schéma fourni.

5. Configurer les variables d'environnement dans un fichier `.env` avec les clés suivantes :

   - `API_KEY`: Clé secrète pour signer les jetons JWT.

Structure du Projet

- `database`: Contient les scripts SQL pour initialiser la base de données.

- `routes`: Contient les routes API pour les fonctionnalités de l'application.

- `controllers`: Contient les fonctions de contrôleur qui implémentent la logique métier.

- `middleware`: Contient les fonctions de middleware, comme l'authentification JWT.

- `models`: Contient les modèles pour représenter les données de la base de données.

- `config`: Contient la configuration de la base de données et d'autres configurations de l'application.

- `app.js`: Fichier principal de l'application.

Fonctionnalités

1. Connexion (`Login`)

```
exports.Login = async (req, res) => {
    const { email, mot_de_passe } = req.body;
    try {
        await pool.query('SELECT * FROM Client WHERE email = ? AND isValidate = true', [email],
            async function (error, results) {
                if (error) throw error;
                const user = results;
                if (user.length > 0) {
                    const match = await bcrypt.compare(mot_de_passe, user[0].mot_de_passe);
                    if (match) {
                        const token = jwt.sign({ email: user[0].email }, process.env.API_KEY, { expiresIn: '1h' });
                        res.status(200).json({
                            message: 'Connexion réussie', data: {
                                numClient: user[0].numClient,
                                nom: user[0].nom,
                                prenom: user[0].prenom,
                                email: user[0].email,
                                isAdmin: user[0].isAdmin,
                                token: token  // Envoyer le token au client
                            }
                        });
                    } else {
                        res.status(401).json({ error: 'Mot de passe incorrect' });
                    }

                } else {
                    res.status(404).json({ error: 'Utilisateur non trouvé' });
                }
            });
    } catch (error) {
        res.status(500).json({ error: 'Erreur lors de la récupération des thèmes' });
    }
};
```

- **Endpoint** : `/login`

- **Description** : Permet à un utilisateur de se connecter.

- **Méthode HTTP** : `POST`

Paramètres Requis :

- `email`: Adresse email de l'utilisateur.

- `mot_de_passe`: Mot de passe de l'utilisateur.

- Réponse Succès (HTTP 200): Retourne un jeton JWT valide avec les informations de l'utilisateur.

- Réponse Erreur (HTTP 401/404/500): En cas d'erreur lors de la connexion, retourne un message d'erreur approprié.

2. Récupération d'un Produit par ID (`getProduit`)

```
const pool = require('../database/database');

//Endpoint pour lister un article grâce à son id
exports.getProduit =  async (req, res) => {
    const articleId = req.params.id;
    try {
        await pool.query('SELECT * FROM Produit WHERE numProduit = ?', [articleId],
            function (error, results) {
                if (error) throw error;
                res.status(200).json(results)
            });
    } catch (error) {
        console.error(error);
        res.status(500).json({ error: 'Erreur lors de la récupération des thèmes' });
    }
};
```

- **Endpoint** : `/produit/:id`

- **Description** : Permet de récupérer un produit spécifique en fonction de son identifiant.

- **Méthode HTTP :** `GET`

- **Paramètres Requis :**

  - `id`: Identifiant du produit.

- Réponse Succès (HTTP 200): Retourne les détails du produit correspondant à l'identifiant spécifié.

- Réponse Erreur (HTTP 500): En cas d'erreur lors de la récupération du produit, retourne un message d'erreur.

3. Récupération de Tous les Produits (`getAllProduits`)

```
// Endpoint pour récupérer tous les produits
exports.getAllProduits = async (req, res) => {
    try {
        await pool.query('SELECT * FROM Produit WHERE isSee = true',
        function (error, results) {
            if (error) throw error;
            res.status(200).json(results)
        });
    } catch (error) {
        console.error(error);
        res.status(500).json({ error: 'Erreur lors de la récupération des produits' });
    }
};
```

- **Endpoint** : `/produits`

- **Description** : Permet de récupérer tous les produits disponibles.

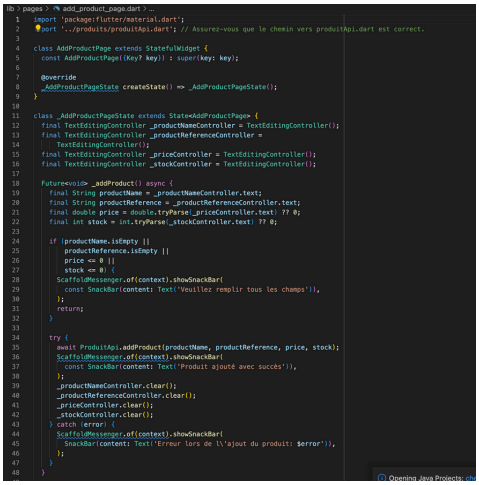
- **Méthode HTTP :** `GET`

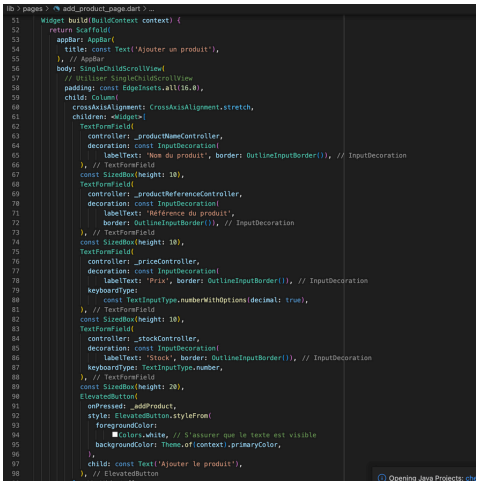
- Réponse Succès (HTTP 200): Retourne la liste de tous les produits visibles.

- Réponse Erreur (HTTP 500): En cas d'erreur lors de la récupération des produits, retourne un message d'erreur.

Documentation du Code en Dart

Ce code Dart est une implémentation d'une page de modification de produit dans une application Flutter. Voici une analyse détaillée du code :





1. Importations:

- Le code commence par importer les packages nécessaires, notamment `material.dart` de Flutter pour les composants d'interface utilisateur et un fichier `produitApi.dart` pour

accéder aux fonctions d'API liées aux produits. Assurez-vous que le chemin du fichier `produitApi.dart` est correctement spécifié en fonction de la structure de votre projet.

2. Classe `EditProductPage`:

- Cette classe hérite de `StatefulWidget`, ce qui signifie que la page est susceptible de subir des modifications d'état au fil du temps.

- Elle définit un état interne de type `_EditProductPageState`.

3. Classe `_EditProductPageState`:

- Cette classe gère l'état de la page de modification de produit.

- Elle stocke les produits récupérés, l'identifiant du produit sélectionné, l'état de chargement, la visibilité des champs, les contrôleurs de texte pour les différents champs de saisie et le chemin de l'image actuellement sélectionnée.

- Elle initialise l'état de la page et récupère les produits lors de son initialisation.

- Elle dispose des contrôleurs de texte lorsqu'elle est détruite pour éviter les fuites de mémoire.

- Elle définit une méthode privée `_fetchProducts` pour récupérer la liste des produits à partir de l'API.

- Elle définit une méthode privée `_fetchAndDisplayProductDetails` pour récupérer les détails d'un produit spécifique à partir de son identifiant.

- Elle utilise les données récupérées pour remplir les champs de saisie lorsqu'un produit est sélectionné.

4. Méthode `build`:

- Cette méthode construit l'interface utilisateur de la page de modification de produit.

- Elle affiche un indicateur de chargement si les produits sont en cours de récupération.

- Elle affiche un menu déroulant des produits disponibles une fois qu'ils sont récupérés.

- Elle affiche les champs de saisie pour modifier les détails du produit sélectionné.

- Elle affiche un bouton pour modifier le produit, qui déclenche une requête API pour mettre à jour le produit.

5. Widgets Flutter utilisés:

- `Scaffold`: Fournit une structure de base pour la page, y compris la barre d'applications.

- `AppBar`: Affiche le titre de la page.

- `Center`, `SingleChildScrollView`, `Column`, `DropdownButtonFormField`, `TextField`, `ElevatedButton`: Widgets pour la mise en page et la saisie utilisateur.

- `CircularProgressIndicator`, `SnackBar`: Widgets pour les indicateurs de chargement et les messages d'erreur.

Ce code permet aux utilisateurs de sélectionner un produit existant, de modifier ses détails et de sauvegarder les modifications via l'API. Il offre une interface utilisateur simple et réactive pour la modification de produits dans une application Flutter.