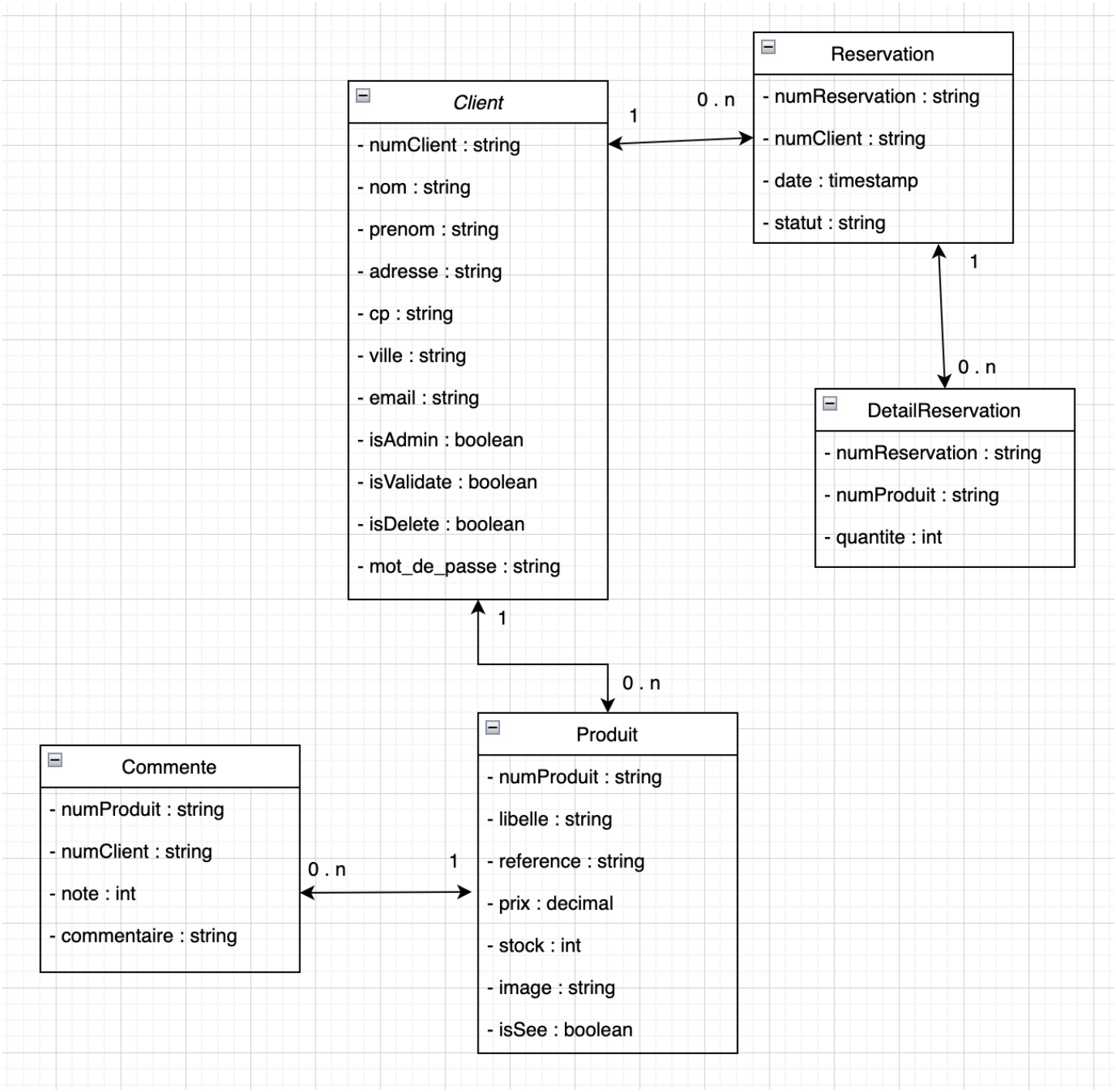
M2L

by Shop_def

Sommaire

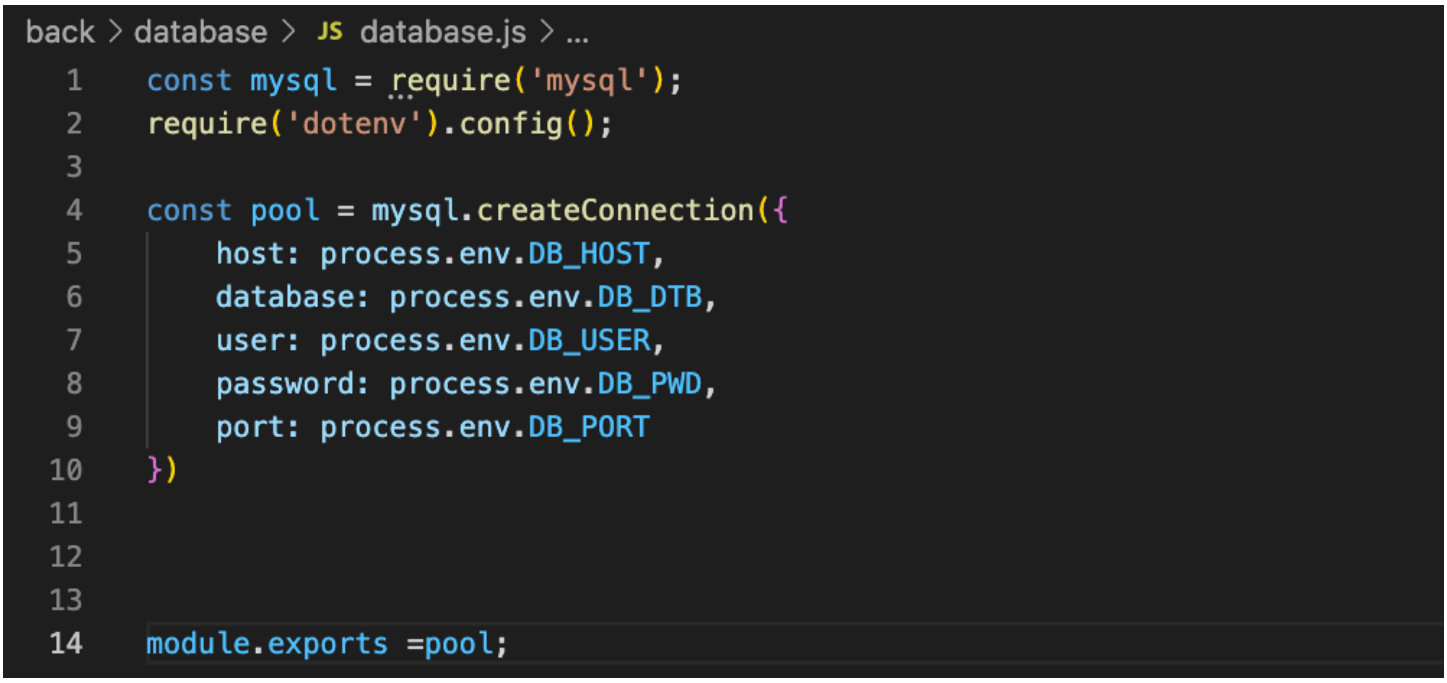
Schéma de la Base de Données



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.



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. Inscription (`Register`)

```
//Endpoint permettant de s'inscrire
exports.Register = async (req, res) => {
    const { nom, prenom, adresse, cp, ville, email, mot_de_passe } = req.body;

    try {
        const hashedPassword = await bcrypt.hash(mot_de_passe, saltRound);
        // Générez un UUID pour le nouvel utilisateur
        const numClient = crypto.randomUUID();

        // Insérez le nouvel utilisateur dans la base de données
        await pool.query('INSERT INTO Client (numClient, nom, prenom, adresse, cp, ville, email, mot_de_passe, isValidate, isAdmin, isDelete) VALUES (?, ?, ?, ?, ?, ?, ?, ?, false,false,false)',
            [numClient, nom, prenom, adresse, cp, ville, email, hashedPassword]);

            const message = 'Compte créé avec succès. Veuillez attendre la validation de votre compte par les administrateurs.';
            res.status(201).json({ message: message });
    } catch (error) {
        console.error('Erreur lors de la création du compte', error);
        res.status(500).json({ error: 'Erreur lors de la création du compte' });
    }
};
```

- **Endpoint**: `/register`

- **Description**: Permet à un utilisateur de s'inscrire en créant un nouveau compte.

- **Méthode HTTP**: `POST`

Paramètres Requis :

- `nom`: Nom de l'utilisateur.

- `prenom`: Prénom de l'utilisateur.

- `adresse`: Adresse de l'utilisateur.

- `cp`: Code postal de l'utilisateur.

- `ville`: Ville de l'utilisateur.

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

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

- Réponse Succès (HTTP 201): Retourne un message de succès.

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

2. 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 :

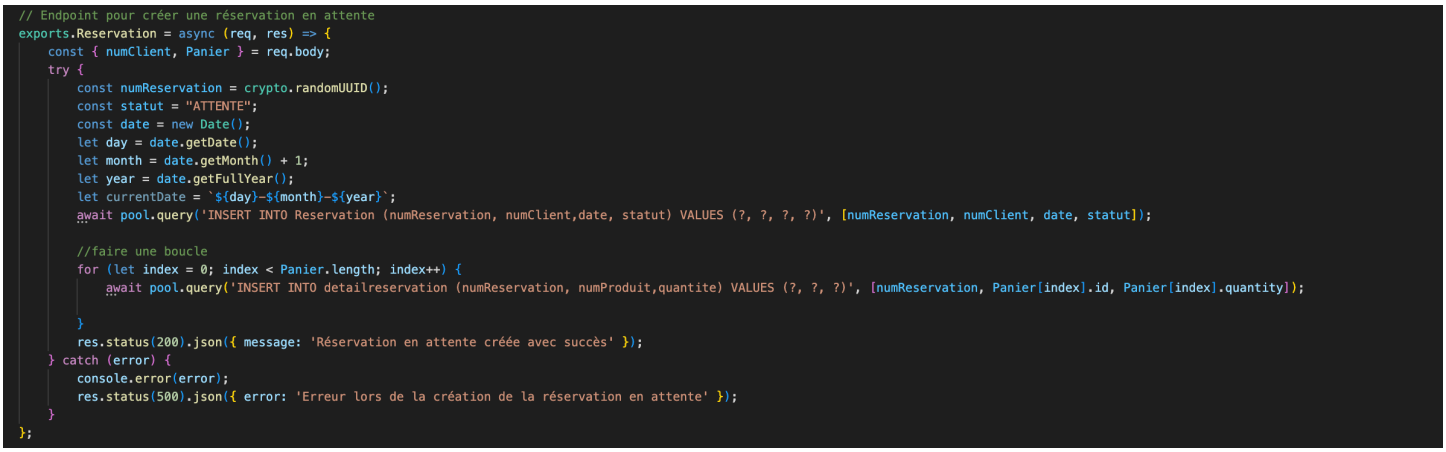
- `email`: Adresse email de l'utilisateur.

- `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.

3. Création de Réservation (`Reservation`)



```
// Endpoint pour créer une réservation en attente
exports.Reservation = async (req, res) => {
    const { numClient, Panier } = req.body;
    try {
        const numReservation = crypto.randomUUID();
        const statut = "ATTENTE";
        const date = new Date();
        let day = date.getDate();
        let month = date.getMonth() + 1;
        let year = date.getFullYear();
        let currentDate = `${day}-${month}-${year}`;
        await pool.query('INSERT INTO Reservation (numReservation, numClient,date, statut) VALUES (?, ?, ?, ?)', [numReservation, numClient, date, statut]);

        //faire une boucle
        for (let index = 0; index < Panier.length; index++) {
            await pool.query('INSERT INTO detailreservation (numReservation, numProduit,quantite) VALUES (?, ?, ?)', [numReservation, Panier[index].id, Panier[index].quantity]);

        }
        res.status(200).json({ message: 'Réservation en attente créée avec succès' });
    } catch (error) {
        console.error(error);
        res.status(500).json({ error: 'Erreur lors de la création de la réservation en attente' });
    }
};
```

- **Endpoint**: `/reservation`

- **Description**: Permet à un utilisateur de créer une réservation en attente.

- **Méthode HTTP:** `POST`

Paramètres Requis :

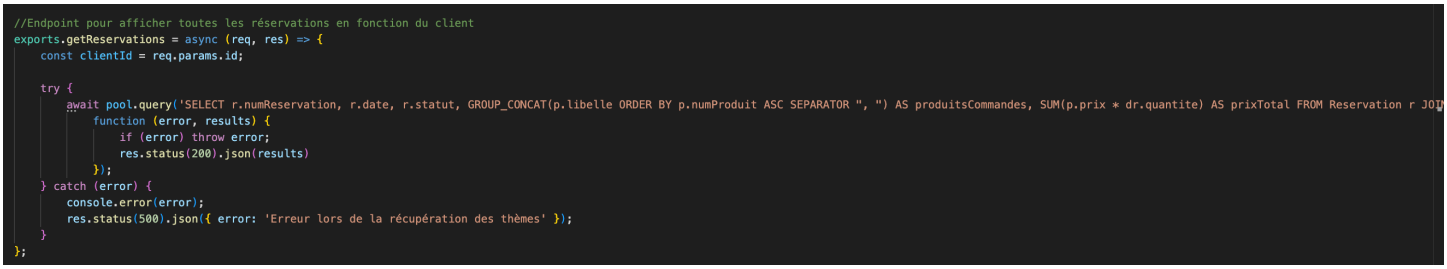
- `numClient`: Numéro d'identification du client.

- `Panier`: Liste des produits ajoutés au panier avec leur quantité.

- Réponse Succès (HTTP 200): Retourne un message de succès après la création de la réservation.

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

4. Affichage des Réservations (`getReservations`)



```
//Endpoint pour afficher toutes les réservations en fonction du client
exports.getReservations = async (req, res) => {
    const clientId = req.params.id;

    try {
        await pool.query('SELECT r.numReservation, r.date, r.statut, GROUP_CONCAT(p.libelle ORDER BY p.numProduit ASC SEPARATOR ", ") AS produitsCommandes, SUM(p.prix * dr.quantite) AS prixTotal FROM Reservation r JOI
            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** : `/reservations/:id`

- **Description** : Permet à un utilisateur d'afficher toutes les réservations en fonction de son identifiant client.

- **Méthode HTTP** : `GET`

Paramètres Requis :

- `id`: Identifiant du client.

- Réponse Succès (HTTP 200): Retourne la liste des réservations du client avec les détails.

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

Sécurité

- Les mots de passe des utilisateurs sont hashés avant d'être stockés en base de données en utilisant l'algorithme bcrypt.

- Les jetons JWT sont signés avec une clé secrète pour empêcher la manipulation des jetons.

Dépendances Principales

- `express`: Framework web pour Node.js.

- `bcrypt`: Bibliothèque pour le hashage sécurisé des mots de passe.

- `jsonwebtoken`: Bibliothèque pour la création et la vérification des jetons JWT.

- `mysql`: Pilote MySQL pour Node.js.

- `crypto`: Module Node.js pour la génération de nombres aléatoires sécurisés.

5. 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.

6. 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 React

Ce code représente le composant principal de l'application React, `App`, qui définit les routes et les composants associés pour la navigation dans l'application.

Le composant `App` est exporté par défaut pour être utilisé comme point d'entrée principal de l'application React.

Composant `App`

Importations

- `React`: Module principal de React pour la création de composants.

- `Route` et `Routes` depuis `react-router-dom`: Composants de routage pour la navigation dans l'application.

- Styles CSS importés depuis `'./App.css'`.

- Composants importés depuis différents fichiers pour la barre de navigation (`Nav`), la page d'administration (`Admin`), la page de détails d'un produit (`Detail`), le pied de page (`Footer`), la page d'historique (`HistPage`), la page d'accueil (`Home`), la présentation (`Presentation`), la page de connexion (`Log`), la page d'inscription (`Register`), et la liste des articles (`ListArticles`).

- `ContextAuth` depuis `'./Components/Context/context'`: Contexte d'authentification pour gérer l'état de l'authentification de l'utilisateur.

La fonction `App` est un composant fonctionnel qui retourne l'arborescence de composants de l'application enveloppée dans le contexte d'authentification.

src > App.jsx > App

```jsx
import React from 'react';
import { Route, Routes } from "react-router-dom";
import './App.css';
import { ContextAuth } from './Components/Context/context';
import Nav from './Components/navbar/Nav';
import Admin from './Components/Pages/Admin/Admin';
import Detail from './Components/Pages/DetailsProd/DetailsProd';
import Footer from './Components/Pages/Footer/Footer';
import HistPage from './Components/Pages/HistPage/HistPage';
import Home from './Components/Pages/Home/Home';
import Presentation from './Components/Pages/Home/Presentation';
import Log from './Components/Pages/Login/Log';
import Register from './Components/Pages/Register/Register';
import ListArticles from './Components/Pages/ShopPage/ListArticles';
import ShopPage from './Components/Pages/ShopPage/ShopPage';


function App() {
  return (

    <ContextAuth>
      <Nav></Nav>
      <Routes>
        <Route path='/' element={<Home />} />
        <Route path='/Presentation' element={<Presentation />} />
        <Route path='/ListArticles' element={<ListArticles />} />
        <Route path='/ShopPage' element={<ShopPage />} />
        <Route path='/Log' element={<Log />} />
        <Route path='/Register' element={<Register />} />
        <Route path='/DetailsProd/:id' element={<Detail/>} />
        <Route path='/Admin' element={<Admin />} />
        <Route path='/Historique' element={<HistPage />} />
      </Routes>
      <Footer/>
    </ContextAuth>

  );
}

export default App;
```

Structure de l'Application

1. Barre de Navigation (`Nav`) : Affiche la barre de navigation en haut de la page.

2. Routes (`Routes`) : Définit les différentes routes de l'application avec les composants associés.

- `/`: Page d'accueil (`Home`).

- `/Presentation`: Page de présentation (`Presentation`).

- `/ListArticles`: Page listant les articles (`ListArticles`).

- `/ShopPage`: Page de shopping (`ShopPage`).

- `/Log`: Page de connexion (`Log`).

- `/Register`: Page d'inscription (`Register`).

- `/DetailsProd/:id`: Page de détails d'un produit avec un identifiant spécifique (`Detail`).

- `/Admin`: Page d'administration (`Admin`).

- `/Historique`: Page d'historique (`HistPage`).

3. Pied de Page (`Footer`) : Affiche le pied de page en bas de la page.

Contexte d'Authentification

- L'ensemble des routes et des composants est enveloppé dans le contexte d'authentification `ContextAuth`, permettant de partager l'état de l'authentification avec tous les composants de l'application.

Maintenant nous pouvons décrire les fonctionnalités et le fonctionnement technique de deux composants React: `ShopPage` et `ListArticles`.

Composant `ShopPage`

Description

Ce composant représente la page de shopping où les utilisateurs peuvent visualiser et gérer leur panier d'achats avant de passer une réservation.

Dépendances

- `js-cookie`: Bibliothèque pour la manipulation des cookies dans JavaScript.

- `react-router-dom`: Bibliothèque de routage pour les applications React.

Fonctionnalités

1. Récupération du Panier :

- La fonction `getUserId` récupère l'identifiant de l'utilisateur à partir du stockage local.

- La fonction `getCartItems` récupère les articles du panier à partir du stockage local en utilisant l'identifiant de l'utilisateur.

2. Vérification du Token :

- La fonction `checkToken` vérifie la présence d'un token d'authentification dans les cookies. Si aucun token n'est trouvé, l'utilisateur est redirigé vers la page de connexion.

3. Suppression d'Article :

- La fonction `removeItem` permet de supprimer un article du panier en fonction de son identifiant.

4. Mise à Jour de la Quantité d'Article :

- La fonction `updateItemQuant` permet de mettre à jour la quantité d'un article dans le panier.

5. Affichage des Articles du Panier :

- La fonction `renderCartItems` affiche les articles actuellement présents dans le panier, avec leurs détails et options de quantité.

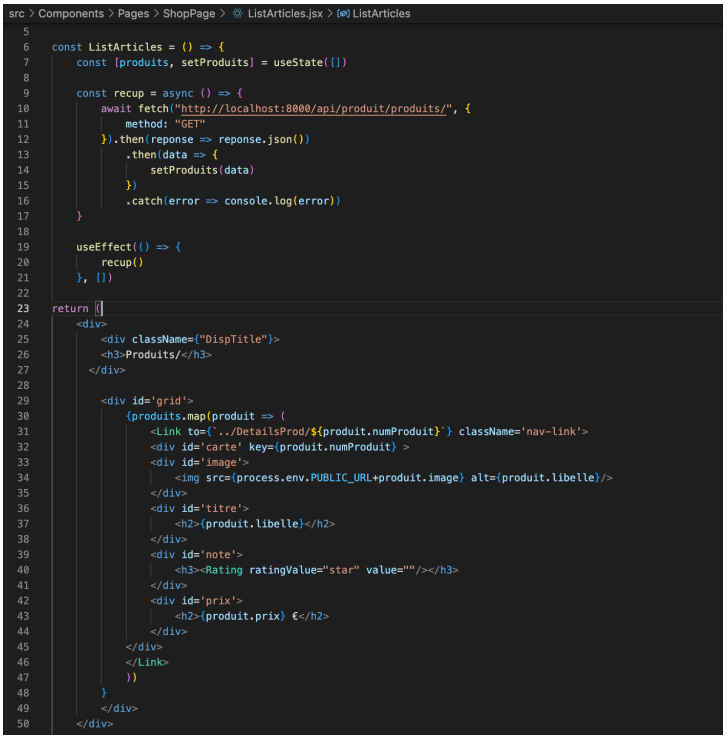
6. Passage de Réservation :

- La fonction `FaireReservation` envoie la réservation au serveur avec les données du panier.

7. Effet Secondaire :

- Les fonctions `checkToken` et `getCartItems` sont exécutées au montage du composant pour assurer la cohérence de l'état initial.

Composant `ListArticles`



Fonctionnalités

1. Récupération des Produits :

   - Le composant récupère la liste des produits disponibles à partir de l'API REST du serveur.

2. Affichage des Produits :

   - Les produits sont affichés sous forme de cartes, avec leur image, titre, note, et prix.

   - Chaque produit est un lien cliquable redirigeant vers sa page de détails.

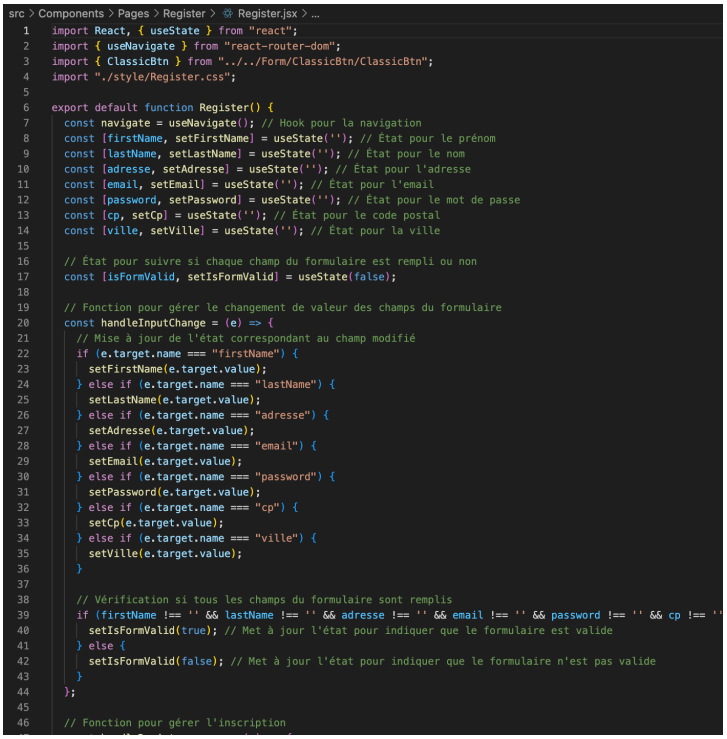
Ce document fournit une vue d'ensemble des deux composants React, ainsi que de leurs fonctionnalités et dépendances associées. Les composants `ShopPage` et `ListArticles` permettent aux utilisateurs d'interagir avec l'application de manière intuitive et efficace pour parcourir les produits, gérer leur panier, et passer des réservations.

Composant `Register`

Ce composant fournit une interface utilisateur permettant aux nouveaux utilisateurs de s'inscrire à l'application.

Fonctionnalités

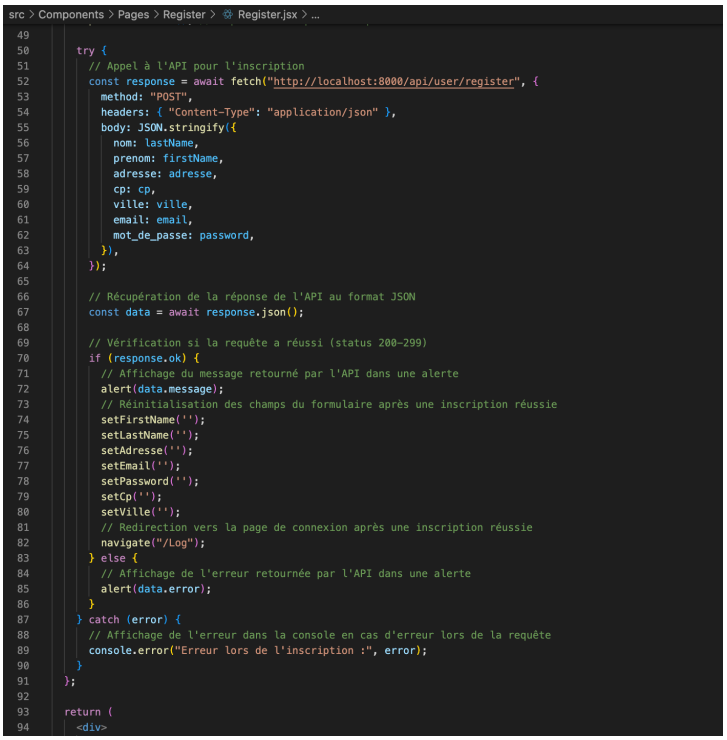
1. **Initialisation des États** :



```jsx
src > Components > Pages > Register > Register.jsx > ...
import React, { useState } from "react";
import { useNavigate } from "react-router-dom";
import { ClassicBtn } from "../../Form/ClassicBtn/ClassicBtn";
import "./style/Register.css";

export default function Register() {
  const navigate = useNavigate(); // Hook pour la navigation
  const [firstName, setFirstName] = useState(''); // État pour le prénom
  const [lastName, setLastName] = useState(''); // État pour le nom
  const [adresse, setAdresse] = useState(''); // État pour l'adresse
  const [email, setEmail] = useState(''); // État pour l'email
  const [password, setPassword] = useState(''); // État pour le mot de passe
  const [cp, setCp] = useState(''); // État pour le code postal
  const [ville, setVille] = useState(''); // État pour la ville

  // État pour suivre si chaque champ du formulaire est rempli ou non
  const [isFormValid, setIsFormValid] = useState(false);

  // Fonction pour gérer le changement de valeur des champs du formulaire
  const handleInputChange = (e) => {
    // Mise à jour de l'état correspondant au champ modifié
    if (e.target.name === "firstName") {
      setFirstName(e.target.value);
    } else if (e.target.name === "lastName") {
      setLastName(e.target.value);
    } else if (e.target.name === "adresse") {
      setAdresse(e.target.value);
    } else if (e.target.name === "email") {
      setEmail(e.target.value);
    } else if (e.target.name === "password") {
      setPassword(e.target.value);
    } else if (e.target.name === "cp") {
      setCp(e.target.value);
    } else if (e.target.name === "ville") {
      setVille(e.target.value);
    }

    // Vérification si tous les champs du formulaire sont remplis
    if (firstName !== '' && lastName !== '' && adresse !== '' && email !== '' && password !== '' && cp !== ''
      setIsFormValid(true); // Met à jour l'état pour indiquer que le formulaire est valide
    } else {
      setIsFormValid(false); // Met à jour l'état pour indiquer que le formulaire n'est pas valide
    }
  };

  // Fonction pour gérer l'inscription
```

- Utilisation du hook `useState` pour gérer les états des champs du formulaire ainsi que l'état pour suivre si le formulaire est valide ou non.

2. **Gestion des Changements des Champs :**



```jsx
src > Components > Pages > Register > Register.jsx > ...

    try {
      // Appel à l'API pour l'inscription
      const response = await fetch("http://localhost:8000/api/user/register", {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({
          nom: lastName,
          prenom: firstName,
          adresse: adresse,
          cp: cp,
          ville: ville,
          email: email,
          mot_de_passe: password,
        }),
      });

      // Récupération de la réponse de l'API au format JSON
      const data = await response.json();

      // Vérification si la requête a réussi (status 200-299)
      if (response.ok) {
        // Affichage du message retourné par l'API dans une alerte
        alert(data.message);
        // Réinitialisation des champs du formulaire après une inscription réussie
        setFirstName('');
        setLastName('');
        setAdresse('');
        setEmail('');
        setPassword('');
        setCp('');
        setVille('');
        // Redirection vers la page de connexion après une inscription réussie
        navigate("/Log");
      } else {
        // Affichage de l'erreur retournée par l'API dans une alerte
        alert(data.error);
      }
    } catch (error) {
      // Affichage de l'erreur dans la console en cas d'erreur lors de la requête
      console.error("Erreur lors de l'inscription :", error);
    }
  };

  return (
    <div>
```

- La fonction `handleInputChange` est appelée à chaque modification d'un champ du

formulaire pour mettre à jour les états correspondants.

- Les valeurs des champs du formulaire sont stockées dans les états pour le prénom, le nom, l'adresse, l'email, le mot de passe, le code postal et la ville.

- Vérification si tous les champs du formulaire sont remplis pour activer ou désactiver le bouton de soumission du formulaire.

3. **Soumission du Formulaire d'Inscription :**

src > Components > Pages > Register > Register.jsx > ...

```jsx
  return (
    <div>
      <div className={"DispTitle"}>
        <h3>Inscription/</h3>
      </div>
      <div className={"subtitle"}>
        <h6>Bonjour !</h6>
        <h6>Veuillez vous inscrire</h6>
        <form className="RegForm" onSubmit={handleRegister}>
          {/* Champs du formulaire */}
          <label htmlFor="firstName"></label>
          <input type="text" id="firstName" name="firstName" placeholder="Prénom" value={firstName} onChange={handleInputChange} required />

          <label htmlFor="lastName"></label>
          <input type="text" id="lastName" name="lastName" placeholder="Nom" value={lastName} onChange={handleInputChange} required />

          <label htmlFor="email"></label>
          <input type="email" id="email" name="email" placeholder="Email" value={email} onChange={handleInputChange} required />

          <label htmlFor="adresse"></label>
          <input type="text" id="adresse" name="adresse" placeholder="Adresse" value={adresse} onChange={handleInputChange} required />

          <label htmlFor="cp"></label>
          <input type="text" id="cp" name="cp" placeholder="Code Postal" value={cp} onChange={handleInputChange} maxLength={5} required />

          <label htmlFor="ville"></label>
          <input type="text" id="ville" name="ville" placeholder="Ville" value={ville} onChange={handleInputChange} required />

          <label htmlFor="password"></label>
          <input type="password" id="password" name="password" placeholder="Mot de passe" value={password} onChange={handleInputChange} required />

          {/* Bouton pour soumettre le formulaire */}
          <ClassicBtn
            name={"S'inscrire"}
            fontSize={"1.2rem"}
            height={"3rem"}
            width={"12rem"}
            type="submit" // Définit le type du bouton comme soumission du formulaire
            disabled={!isFormValid} // Désactive le bouton si le formulaire n'est pas valide
          />
        </form>
      </div>
    </div>
  );
}
```

- La fonction `handleRegister` est appelée lors de la soumission du formulaire.

- Empêche le comportement par défaut du formulaire.

- Effectue une requête POST à l'API pour l'inscription avec les données du formulaire.

- Affiche les messages retournés par l'API en cas de succès ou d'erreur.

- Réinitialise les champs du formulaire après une inscription réussie.

- Redirige l'utilisateur vers la page de connexion après une inscription réussie.

Structure du Composant

- La page d'inscription est structurée avec un titre, des sous-titres et un formulaire.

- Les champs du formulaire sont contrôlés par les états et mis à jour via la fonction `handleInputChange`.

- Le bouton de soumission du formulaire est désactivé s'il n'est pas valide.

Composants Externes Utilisés

- `ClassicBtn` (depuis `../../Form/ClassicBtn/ClassicBtn`): Composant bouton utilisé pour soumettre le formulaire.

Gestion de la Navigation

- Utilisation du hook `useNavigate` pour la navigation vers une autre page après une inscription réussie.