



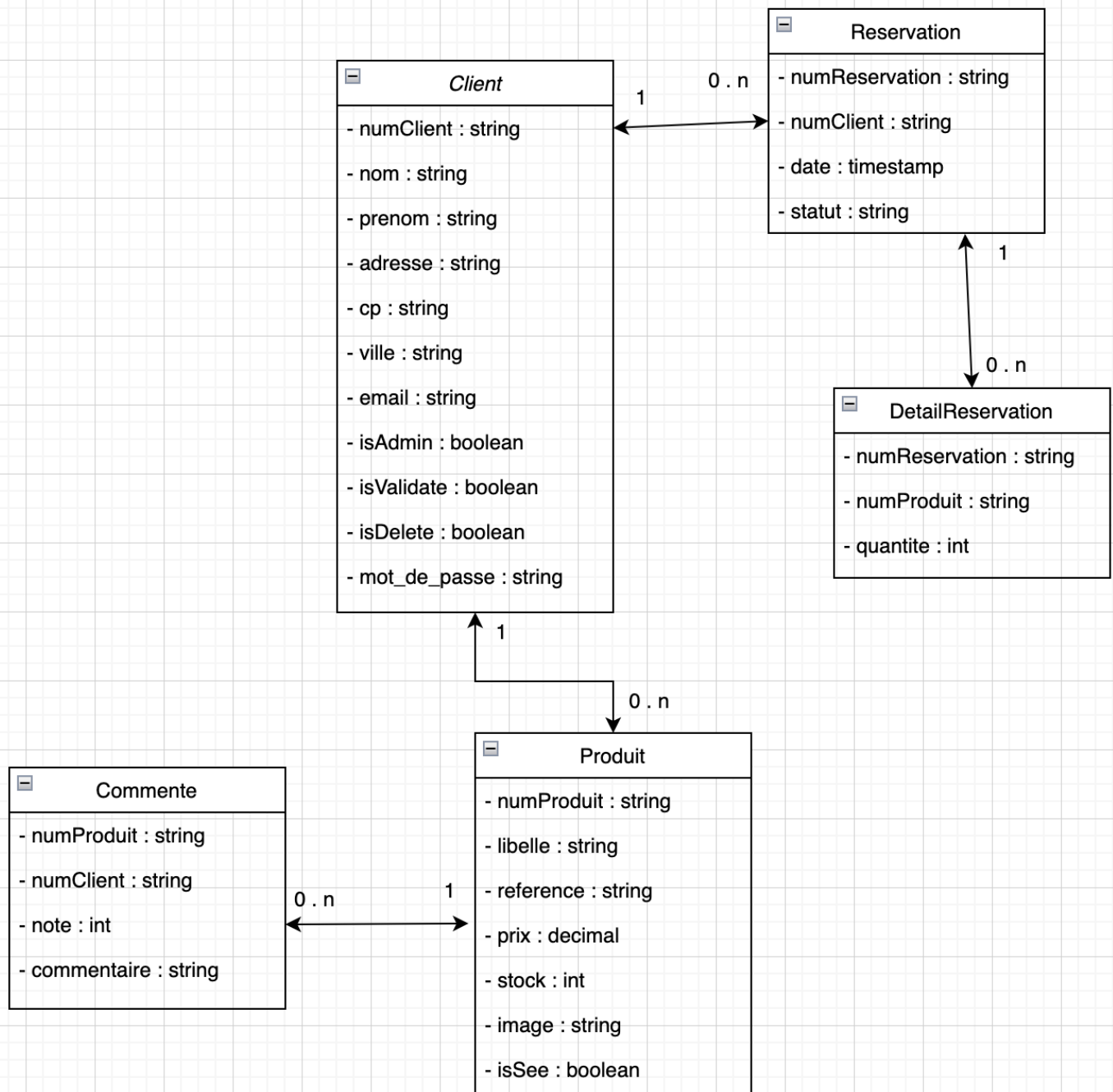
M2L

by Shop_def

Sommaire

1. Schéma de la Base de Données
2. Documentation de l'API
3. Documentation du code en React

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. 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 JOIN detailreservation dr ON r.numReservation = dr.numReservation JOIN produits p ON dr.numProduit = p.id WHERE r.numClient = ? GROUP BY r.numReservation, r.date, r.statut', [clientId]);
    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
1  import React from 'react';
2  import { Route, Routes } from 'react-router-dom';
3  import './App.css';
4  import { ContextAuth } from './Components/Context/context';
5  import Nav from './Components/navbar/Nav';
6  import Admin from './Components/Pages/Admin/Admin';
7  import Detail from './Components/Pages/DetailsProd/DetailsProd';
8  import Footer from './Components/Pages/Footer/Footer';
9  import HistPage from './Components/Pages/HistPage/HistPage';
10 import Home from './Components/Pages/Home/Home';
11 import Presentation from './Components/Pages/Home/Presentation';
12 import Log from './Components/Pages/Login/Log';
13 import Register from './Components/Pages/Register/Register';
14 import ListArticles from './Components/Pages/ShopPage/ListArticles';
15 import ShopPage from './Components/Pages/ShopPage/ShopPage';
16
17
18 function App() {
19   return (
20     <ContextAuth>
21       <Nav></Nav>
22       <Routes>
23         <Route path="/" element={<Home />} />
24         <Route path="/Presentation" element={<Presentation />} />
25         <Route path="/ListArticles" element={<ListArticles />} />
26         <Route path="/ShopPage" element={<ShopPage />} />
27         <Route path="/Log" element={<Log />} />
28         <Route path="/Register" element={<Register />} />
29         <Route path="/DetailsProd/:id" element={<Detail/>} />
30         <Route path="/Admin" element={<Admin />} />
31         <Route path="/Historique" element={<HistPage />} />
32       </Routes>
33     </ContextAuth>
34   );
35 }
36
37 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 :

```

src > Components > Pages > ShopPage > ShopPage.jsx > ShopPage > renderCartItems > cartItems.map() callback
1  import Cookies from "js-cookie";
2  import React, { useEffect, useState } from "react";
3  import { useNavigate } from "react-router-dom";
4  import { ClassicBtn } from "../../Form/ClassicBtn/ClassicBtn";
5  import { Line } from "../../Other/Line/Line";
6  import "./style/ShopPage.css";
7
8  export default function ShopPage() {
9    const navigate = useNavigate();
10   const [cartItems, setCartItems] = useState([]);
11
12   const getUserId = () => {
13     const userId = localStorage.getItem("uid");
14     return userId;
15   };
16
17   const getCartItems = () => {
18     const cartItems = JSON.parse(localStorage.getItem(getUserId())) || [];
19     setCartItems(cartItems);
20     return cartItems;
21   };
22
23   const checkToken = async () => {
24     const token = Cookies.get("token");
25     if (token === undefined) {
26       navigate("/Log");
27     }
28   };
29
30   const removeItem = (itemId) => {
31     const userId = getUserId();
32     const updatedCart = cartItems.filter((item) => item.id !== itemId);
33     localStorage.setItem(userId, JSON.stringify(updatedCart));
34     setCartItems(updatedCart);
35   };
36
37   //fonction qui change la quantité dans le localStorage
38   const updateItemQuant = (itemId, newQuantity) => {
39     const updatedCart = cartItems.map((item) => {
40       if (item.id === itemId) {
41         return { ...item, quantity: newQuantity };
42       }
43       return item;
44     });
45     setCartItems(updatedCart);
46     localStorage.setItem(getUserId(), JSON.stringify(updatedCart));

```

- 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 :

```

src > Components > Pages > ShopPage > ShopPage.jsx > ShopPage > renderCartItems > cartItems.map() callback
96  const FaireReservation = async () => {
97    const donnees = { numClient: getUserId(), Panier: cartItems };
98
99    try {
100     const response = await fetch(
101       "http://localhost:8000/api/user/reservation",
102       {
103         method: "POST",
104         headers: { "Content-Type": "application/json" },
105         body: JSON.stringify(donnees),
106       }
107     );
108     console.log(response.data);
109   } catch (error) {
110     console.error("Erreur lors de l'envoi de la réservation", error);
111   }
112   localStorage.removeItem(getUserId());
113   window.location.reload();
114 };
115
116 useEffect(() => {
117   checkToken();
118   getCartItems();
119 }, []);
120
121 return (
122   <div>
123     <div className="DispShop">
124       <h3>Panier</h3>
125     </div>
126     <div className="DispInf">
127       <h5>Produits</h5>
128       <h5>Total</h5>
129     </div>
130     <Line width="90%" />
131     <div>{renderCartItems()}</div>
132     <div className="DispValider">
133       <ClassicBtn
134         fontSize="1.5rem"
135         height="2.5rem"
136         width="8rem"
137         name="Valider"
138         onClickBtn=FaireReservation
139       />
140     </div>
141   </div>

```

- 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 :

```
src > Components > Pages > ShopPage > ShopPage.jsx > ShopPage > renderCartItems > cartItems.map() callback
48 |
49 | const renderCartItems = () => {
50 |   return cartItems.map((item) => {
51 |     <ul key={item.id}>
52 |       <li>
53 |         <div className={"DispCI"}>
54 |           <div className={"DispItem"}>
55 |             <h4 className={"np"}>{item.name}</h4>
56 |             <div className={"optionsDiv"}>
57 |               <p>Quantité :</p>
58 |               <select
59 |                 className={"selectQuantite"}
60 |                 name="quantite"
61 |                 id="nbrDeSelects"
62 |                 value={item.quantity}>
63 |                 // mise en place du onChange pour changer automatiquement le localStorage
64 |                 onChange={(e) => updateItemQuant(item.id, e.target.value)}>
65 |                 <option value={item.quantity}>{item.quantity}</option>
66 |                 { /* Création d'un nombre d'options en fonction du item.stocks dans le localStorage */ }
67 |                 {() => {
68 |                   const options = [];
69 |                   for (let i = 1; i <= item.stocks; i++) {
70 |                     options.push(
71 |                       <option key={i} value={i}>
72 |                         {i}
73 |                       </option>
74 |                     );
75 |                   }
76 |                   return options;
77 |                 }()}
78 |               </select>
79 |             </div>
80 |             { /* Calcul du prix en fonction de la quantité malgré le changement dans le localStorage */ }
81 |             <p className={"InfoP"}>
82 |               Prix: {parseFloat(item.price * item.quantity).toFixed(2)} €
83 |             </p>
84 |           </div>
85 |           <div className={"DispBtnDlt"}>
86 |             <button className={"BtnDlt"} onClick={() => removeItem(item.id)}>
87 |               X
88 |             </button>
89 |           </div>
90 |         </div>
91 |       </li>
92 |     </ul>
93 |   );
94 | }
```

- 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`

```

src > Components > Pages > ShopPage > ListArticles.jsx > ListArticles
5
6 const ListArticles = () => {
7   const [produits, setProduits] = useState([])
8
9   const recup = async () => {
10     await fetch("http://localhost:8000/api/product/produits/", {
11       method: "GET"
12     }).then(reponse => reponse.json())
13     .then(data => {
14       setProduits(data)
15     })
16     .catch(error => console.log(error))
17   }
18
19   useEffect(() => {
20     recup()
21   }, [])
22
23   return (
24     <div>
25       <div className="DispTitle">
26         <h3>Produits</h3>
27       </div>
28
29       <div id="grid">
30         {produits.map(produit => (
31           <Link to={`../DetailsProd/${produit.numProduit}`} className="nav-Link">
32             <div id="carte" key={produit.numProduit}>
33               <div id="image">
34                 <img src={process.env.PUBLIC_URL+produit.image} alt={produit.libelle}/>
35               </div>
36               <div id="titre">
37                 <h2>{produit.libelle}</h2>
38               </div>
39               <div id="note">
40                 <h3><Rating ratingValue="star" value="" /></h3>
41               </div>
42               <div id="prix">
43                 <h2>{produit.prix} €</h2>
44               </div>
45             </div>
46             </Link>
47           ))
48         }
49       </div>
50     </div>

```

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 :

```

src > Components > Pages > Register > Register.jsx > ...
1  import React, { useState } from "react";
2  import { useNavigate } from "react-router-dom";
3  import { ClassicBtn } from "../../Form/ClassicBtn/ClassicBtn";
4  import "../style/Register.css";
5
6  export default function Register() {
7    const navigate = useNavigate(); // Hook pour la navigation
8    const [firstName, setFirstName] = useState(""); // État pour le prénom
9    const [lastName, setLastName] = useState(""); // État pour le nom
10   const [adresse, setAdresse] = useState(""); // État pour l'adresse
11   const [email, setEmail] = useState(""); // État pour l'email
12   const [password, setPassword] = useState(""); // État pour le mot de passe
13   const [cp, setCp] = useState(""); // État pour le code postal
14   const [ville, setVille] = useState(""); // État pour la ville
15
16   // État pour suivre si chaque champ du formulaire est rempli ou non
17   const [isValid, setIsValid] = useState(false);
18
19   // Fonction pour gérer le changement de valeur des champs du formulaire
20   const handleInputChange = (e) => {
21     // Mise à jour de l'état correspondant au champ modifié
22     if (e.target.name === "firstName") {
23       setFirstName(e.target.value);
24     } else if (e.target.name === "lastName") {
25       setLastName(e.target.value);
26     } else if (e.target.name === "adresse") {
27       setAdresse(e.target.value);
28     } else if (e.target.name === "email") {
29       setEmail(e.target.value);
30     } else if (e.target.name === "password") {
31       setPassword(e.target.value);
32     } else if (e.target.name === "cp") {
33       setCp(e.target.value);
34     } else if (e.target.name === "ville") {
35       setVille(e.target.value);
36     }
37   }
38
39   // Vérification si tous les champs du formulaire sont remplis
40   if (firstName !== "" && lastName !== "" && adresse !== "" && email !== "" && password !== "" && cp !== "" && ville !== "") {
41     setIsValid(true); // Met à jour l'état pour indiquer que le formulaire est valide
42   } else {
43     setIsValid(false); // Met à jour l'état pour indiquer que le formulaire n'est pas valide
44   }
45
46   // 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 :

```

src > Components > Pages > Register > Register.jsx > ...
49
50   try {
51     // Appel à l'API pour l'inscription
52     const response = await fetch("http://localhost:8080/api/user/register", {
53       method: "POST",
54       headers: { "Content-Type": "application/json" },
55       body: JSON.stringify({
56         nom: lastName,
57         prenom: firstName,
58         adresse: adresse,
59         cp: cp,
60         ville: ville,
61         email: email,
62         mot_de_passe: password,
63       }),
64     });
65
66     // Récupération de la réponse de l'API au format JSON
67     const data = await response.json();
68
69     // Vérification si la requête a réussi (status 200-299)
70     if (response.ok) {
71       // Affichage du message retourné par l'API dans une alerte
72       alert(data.message);
73       // Réinitialisation des champs du formulaire après une inscription réussie
74       setFirstName("");
75       setLastName("");
76       setAdresse("");
77       setEmail("");
78       setPassword("");
79       setCp("");
80       setVille("");
81       // Redirection vers la page de connexion après une inscription réussie
82       navigate("/Log");
83     } else {
84       // Affichage de l'erreur retournée par l'API dans une alerte
85       alert(data.error);
86     }
87   } catch (error) {
88     // Affichage de l'erreur dans la console en cas d'erreur lors de la requête
89     console.error("Erreur lors de l'inscription :", error);
90   }
91
92   return (
93     <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 > ...
92
93   return (
94     <div>
95       <div className={"DispTitle"}>
96         <h3>Inscription</h3>
97       </div>
98       <div className={"subtitle"}>
99         <h6>Bonjour !</h6>
100        <h6>Veuillez vous inscrire</h6>
101        <form className="RegForm" onSubmit={handleRegister}>
102          { /* Champs du formulaire */ }
103          <label htmlFor="firstName"></label>
104          <input type="text" id="firstName" name="firstName" placeholder="Prénom" value={firstName} onChange={handleInputChange} required />
105
106          <label htmlFor="lastName"></label>
107          <input type="text" id="lastName" name="lastName" placeholder="Nom" value={lastName} onChange={handleInputChange} required />
108
109          <label htmlFor="email"></label>
110          <input type="email" id="email" name="email" placeholder="Email" value={email} onChange={handleInputChange} required />
111
112          <label htmlFor="adresse"></label>
113          <input type="text" id="adresse" name="adresse" placeholder="Adresse" value={adresse} onChange={handleInputChange} required />
114
115          <label htmlFor="cp"></label>
116          <input type="text" id="cp" name="cp" placeholder="Code Postal" value={cp} onChange={handleInputChange} maxLength={5} required />
117
118          <label htmlFor="ville"></label>
119          <input type="text" id="ville" name="ville" placeholder="Ville" value={ville} onChange={handleInputChange} required />
120
121          <label htmlFor="password"></label>
122          <input type="password" id="password" name="password" placeholder="Mot de passe" value={password} onChange={handleInputChange} required />
123
124          { /* Bouton pour soumettre le formulaire */ }
125          <ClassicBtn
126            name={"S'inscrire"}
127            fontSize={"1.2rem"}
128            height={"3rem"}
129            width={"12rem"}
130            type="submit" // Définit le type du bouton comme soumission du formulaire
131            disabled={!isFormValid} // Désactive le bouton si le formulaire n'est pas valide
132          />
133        </form>
134      </div>
135    </div>
136  );
137 }
```

- 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.