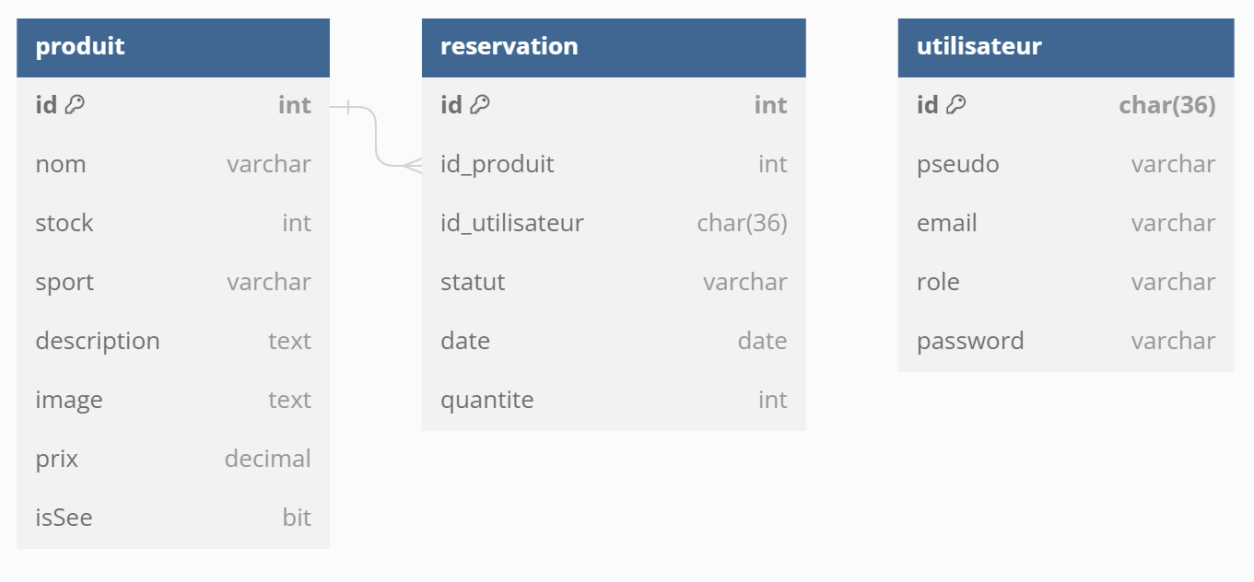
DOCUMENTATION TECHNIQUE

Index

Schéma base de données



Nous avons 3 tables dans la base de données qui s'occupent chacune de leur propre rôle respectivement suivant leur nom .

La documentation API

Cette API est développée en utilisant Node.js et Express, avec une base de données MariaDB pour stocker les informations sur les produits, les réservations, et les utilisateurs. Les fonctionnalités principales incluent la gestion des produits, des réservations, des utilisateurs, et des opérations de connexion. Elle fait donc le simplement le lien entre notre base de données et le site web .

Package utilisés

```
require('dotenv').config()
const express = require('express')
const app = express()
const mariadb = require('mariadb');
const bcrypt = require("bcrypt");
const crypto = require ("crypto");
const multer = require("multer");
const path = require("path");
process.env.public_url;
console.log(crypto.randomUUID());
let cors = require('cors')

app.use(express.json())
app.use(cors())
app.get('/favicon.ico', (req, res) => {
  res.status(204);
});
```

Package	Utilité (global)	Dans le code
Express	Framework d'application web pour Node.js. Son approche modulaire permet d'ajouter des fonctionnalités à la demande, ce qui en fait un choix populaire pour construire des applications robustes et évolutives.	Express configure le serveur web, gère les routes via les méthodes HTTP (GET, POST, PUT) pour définir les points d'entrée de l'API. Il permet l'ouverture du server sur un port dédier
Dotenv	Charge les variables d'environnement à partir d'un fichier.env.	Sert à se connecter à la BDD sans avoir les log' en dur dans le code

Cors	Middleware permettant le partage des ressources entre origines (CORS) dans les applications Express.	Active CORS pour l'application Express, lui permettant d'accepter des requêtes depuis différentes origines.
MariabDB	Un module Node.js pour MariaDB, fournissant une API asynchrone pour interagir avec les bases de données MariaDB.	Établit un pool de connexions vers la base de données MariaDB avec les informations fournies dans les variables d'environnement, permettant à l'application d'exécuter des requêtes SQL.
Bcrypt	Une bibliothèque pour hacher les mots de passe, offrant une manière sécurisée de stocker les mots de passe des utilisateurs.	Hache et vérifie les mots de passe pour l'authentification des utilisateurs.
Multer	Un middleware pour gérer les données de formulaire multipart, couramment utilisé pour les téléchargements de fichiers.	Configure le stockage des images téléchargées et configure multer pour gérer les téléchargements de fichiers.
Path	Un module de base fournissant des utilitaires pour travailler avec les routes/chemins de fichiers et de répertoires.	Extrait l'extension de fichier du nom original d'un fichier téléchargé.
Jsonwebtoken	Crée et vérifie des jetons Web JSON (JWT) pour une communication sécurisée entre les parties.	Génère un JWT pour l'authentification des utilisateurs lors de la connexion.

Authentification :

```
const pool = mariadb.createPool({
    host : process.env.DB_HOST,
    database : process.env.DB_DTB,
    user : process.env.DB_USER,
    password : process.env.DB_PWD,
    port:process.env.DB_PORT,
    connectionLimit: 100
});
```

Ici on utilise un fichier .env qui comprend tous les logs nécessaire a la connexion a la base de données

Endpoints :

Les endpoints sont endroit de l'API qui définissent où les requêtes HTTP peuvent être faites. Chaque endpoint est associé à une opération spécifique, permettant d'effectuer des actions sur les données du serveur. Ici les endpoint utiliser sont basiques et ne servent qu'a touché à la base de données.

```
app.get('/produit', async (req, res) => {  ···
  });
app.get('/reservations', async (req, res) => {  ···
  });

  app.put('/reservations/:id/valider', async(req,res)=>{
    const id = parseInt(req.params.id)
    let conn;
    try{
        console.log("lancement de la connexion")
        conn = await pool.getConnection();
        console.log("lancement de la requete")
        const rows=await conn.query("UPDATE reservation JOIN produit ON produit.id = reservation.id_produit SET reservation.statut ='validé',produit.stock
        res.status(200).json(rows.affectedRows)
    }
    catch(err){
        console.log(err);
    }
  })
  app.put('/reservations/:id/refuser', async(req,res)=>{···
  })
```

```
app.post('/ajouter',upload.single('image'),async(req,res)=>{
  let conn;
  var nom = req.body.nom;
  var stock = req.body.stock;
  var sport = req.body.sport;
  var description = req.body.description;
  var image = '/images/' + req.file.filename; // Chemin de l'image dans votre application
  var prix= req.body.prix;
  try{
      console.log("lancement de la connexion")
      conn = await pool.getConnection();
      console.log("lancement de la requette")
      const rows = await conn.query(`INSERT INTO produit (nom, stock, sport, description, image, prix,isSee) VALUES (?, ?, ?, ?, ?, ?,1)`, [nom, stock, sp
      res.status(200).json(rows.affectedRows)
  }
  catch(err){
      console.log(err);
  }
})
```

- **GET /produit**

Récupère tous les produits visibles (isSee = 1).

- **GET /reservations**

Récupère toutes les réservations avec des informations étendues sur les produits et les utilisateurs.

- **PUT /reservations/:id/valider**

Valide une réservation en mettant à jour le statut et en ajustant le stock du produit.

- **PUT /reservations/:id/refuser**

Refuse une réservation en mettant à jour le statut.

- **POST /ajouter**

Ajoute un nouveau produit avec image.

- **PUT /modifier/:id**

Modifie un produit existant avec une nouvelle image.

- **GET /sport/:sport**

Récupère tous les produits d'un certain sport visibles (isSee = 1).

- **GET /commande/:id_utilisateur**

Récupère toutes les réservations d'un utilisateur avec des informations étendues.

- **POST /sport**

Ajoute un nouvel utilisateur.

- **POST /connexion**

Connecte un utilisateur et génère un jeton JWT.

- **POST /reservation**

Effectue une réservation pour un ou plusieurs produits.

- **GET /:nom**

Récupère un produit spécifique par son nom.

- **PUT /supprimer/:id**

Marque un produit comme non visible (isSee = 0).

La documentation React :

Package utilisés

react-router-dom :	Description : React Router DOM est une bibliothèque pour la gestion déclarative des routes dans les applications React. Elle permet la navigation et le rendu de différents composants en fonction de l'URL.	Rôle dans les composants React : Le code utilise le composant Link de react-router-dom pour créer des liens de navigation. Le hook useNavigate est utilisé pour la navigation programmatique dans le composant Ajouter.
useState :	Description : Le hook useState fait partie de l'API des Hooks de React, permettant aux composants fonctionnels de gérer leur état. Il renvoie une variable d'état et une fonction pour mettre à jour cette variable.	Rôle dans les composants React : Les composants Accueil et Ajouter utilisent tous deux useState pour gérer des variables d'état locales telles que nom, stock, sport, etc., afin de gérer les saisies de formulaire et les changements d'état.
useEffect	Le hook useEffect est utilisé pour les effets secondaires dans les composants fonctionnels. Il effectue des actions après le rendu, telles que la récupération de données, les abonnements ou la modification manuelle du DOM.	Dans le composant Accueil, useEffect est importé mais n'est pas actuellement utilisé. Il pourrait être destiné à une utilisation future ou pourrait être supprimé s'il est inutile.
Link :	Link est un composant fourni par react-router-dom pour la navigation déclarative entre différentes parties de l'application sans provoquer de rechargement complet de la page.	Dans le composant Accueil, Link est utilisé pour envelopper les boutons, créant des liens cliquables qui naviguent vers des pages spécifiques de sports lorsqu'ils sont cliqués.
useParams :	useParams est un hook de react-router-dom permettant d'accéder aux paramètres de l'itinéraire actuel.	Bien que useParams soit importé dans le code, il n'est actuellement pas utilisé. Il est généralement utilisé dans les composants rendus par Route

		pour accéder aux paramètres dynamiques de l'itinéraire.
useNavigate	useNavigate est un hook fourni par react-router-dom permettant la navigation programmatique dans les composants fonctionnels.	Dans le composant Ajouter, useNavigate est utilisé pour naviguer de manière programmatique vers l'itinéraire '/Produit' après la soumission réussie du formulaire.
FormData	L'objet FormData est utilisé pour représenter un ensemble de paires clé/valeur représentant les champs de formulaire et leurs valeurs.	Dans le composant Ajouter, FormData est utilisé pour construire un ensemble de paires clé/valeur représentant les données du formulaire. Il est ensuite envoyé dans une requête POST au serveur pour ajouter un nouveau produit.

Hook = élément qui permet de récupérer/ garder une information

Fonctions

Fonction fetchCommande (Commande.js)

La fonction fetchCommande utilise useEffect pour récupérer les commandes d'un utilisateur. Elle effectue une requête au serveur pour obtenir les données des commandes et les met à jour dans l'état.

```
const [Commande,setCommande]= useState(null);
const id_utilisateur = localStorage.getItem("userId");
useEffect(() => {
    const fetchCommande = async () => {
        try {
            const response = await fetch(`http://localhost:3000/commande/${id_utilisateur}`);
            if (response.ok) {
                const data = await response.json();
                setCommande(data);
            } else {
                throw new Error('Erreur lors de la récupération de la liste des réservations');
            }
        } catch (error) {
            console.error(error);
        }
    };

    fetchCommande();
}, []);
```

Fonction handleSubmit (Connexion.js)

La fonction handleSubmit gère la soumission du formulaire de connexion. Elle fait une requête au serveur pour authentifier l'utilisateur et gère les réponses. Elle utilise également localStorage pour stocker le token et l'ID utilisateur.

```
const handleSubmit = async (e) => {-
  e.preventDefault();

  try {
    const response = await fetch('http://localhost:3000/connexion', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
      },
      body: JSON.stringify({
        login: identifiant,
        mdp: motDePasse,
      }),
    });

    if (response.ok) {
      const data = await response.json();
      const token = data.token;
      const userId = data.userId;
      const role = data.role;
      const pseudo = data.pseudo;

      if (token) {
        localStorage.setItem('token', token);
        localStorage.setItem('userId',userId);
        localStorage.setItem('role',role);
        localStorage.setItem('pseudo',pseudo);
        alert('Connexion réussie !');
        navigate('/');
        window.location.reload();
      } else {
        alert('Token manquant dans la réponse');
      }
    } else {
      alert('Identifiants incorrects');
    }
  } catch (error) {
    console.error('Erreur lors de la requête :', error);
    alert('Une erreur s\'est produite lors de la communication avec le serveur');
  }
};
```

Gestion de l'Authentification (Header.js)

Le composant Header gère l'état pour l'authentification et le rôle de l'utilisateur. Il utilise localStorage pour stocker les informations de l'utilisateur.

```
function Header() {
  const [loggedIn, setLoggedIn] = useState(false);
  const [isSubmenuOpen, setIsSubmenuOpen] = useState(false);
  const [isUserMenuOpen, setIsUserMenuOpen] = useState(false);
  const [role,setRole] = useState('');
  const navigate =useNavigate();


  const handleLogout = () => {
    localStorage.removeItem('token');
    localStorage.removeItem('userId');
    localStorage.removeItem('role');
    setLoggedIn(false);
    navigate('/');
    window.location.reload();
  }

  useEffect(() => {
    // Vérifiez la présence du token dans le localStorage au chargement de la page
    const token = localStorage.getItem('token');
    const storedRole = localStorage.getItem('role');
    if (token) {
        setLoggedIn(true);
    }
    if(storedRole){
      setRole(storedRole);
    }
}, []);
```

Utilisation de FormData (Modifier.js)

La fonction handleSubmit utilise FormData pour envoyer des données au serveur pour la modification d'un produit.

```
  const handleSubmit = async (e) => {
      e.preventDefault();

      const formData = new FormData();
formData.append('nom', nom);
formData.append('stock', stock);
formData.append('sport', sport);
formData.append('description', description);
formData.append('prix',prix);

if (image) {
  formData.append('image', image); // Si une nouvelle image est sélectionnée, l'ajouter
}

  try {
      const response = await fetch(`http://localhost:3000/modifier/${id}`, {
        method: 'PUT',
        body: formData,
      });

      if (response.status === 200) {
        console.log('Produit mis à jour avec succès');
        alert('Produit mis à jour avec succès');
        navigate('/Produit');
      } else {
        console.error('Erreur lors de la mise à jour du produit.');
      }
    } catch (error) {
      console.error('Erreur lors de la mise à jour du produit :', error);
    }
  }
```

Fonction recup (Produit.js)

La fonction recup gère la suppression d'un produit.

```
const recup = async (id) => {
  const confirmation = window.confirm('Voulez-vous supprimer ce produit?');
  if (confirmation){
  try {
    const response = await fetch(`http://localhost:3000/supprimer/${id}`, { method: "PUT" });
    if (response.status === 200) {
      recup();
    } else {
      console.error("Erreur lors de la suppression du produit.");
    }
  } catch (error) {
    console.error("Erreur lors de la suppression du produit :", error);
  }
}
}
```

Fonctions validerReservation et refuserReservation (Reservation.js)

La fonction validerReservation est responsable de l'interaction avec le serveur pour valider une réservation spécifique. Elle est déclenchée lorsque l'utilisateur clique sur le bouton "Valider" associé à une réservation.

Outils Utilisés :

fetch API :

La fonction utilise l'API fetch pour effectuer une requête HTTP de type PUT vers le serveur.

La méthode PUT est utilisée pour indiquer au serveur de valider la réservation.

Confirmation de Fenêtre (window.confirm) :

Avant d'envoyer la requête de validation, la fonction affiche une fenêtre de confirmation (window.confirm) pour s'assurer que l'utilisateur souhaite vraiment valider la réservation.

Mise à Jour Réactive du Statut :

Une fois la validation réussie du côté serveur (statut 200), la fonction met à jour réactivement les réservations dans le composant en utilisant setReservations.

Elle crée une nouvelle liste mise à jour en modifiant le statut de la réservation spécifique à 'validé' et utilise setReservations pour mettre à jour le composant.

Fonction refuserReservation :

Description :

La fonction refuserReservation est similaire à validerReservation, mais elle est utilisée pour refuser une réservation plutôt que de la valider.

```
const validerReservation = async (id) => {
  const confirmation = window.confirm('Voulez-vous valider ce produit?');
  if (confirmation) {
    try {
      const response = await fetch(`http://localhost:3000/reservations/${id}/valider`, { method: 'PUT' });
      if (response.status === 200) {
        // Mettre à jour les réservations après validation
        const updatedReservations = reservations.map((reservation) =>
          reservation.id === id ? { ...reservation, statut: 'validé' } : reservation
        );
        setReservations(updatedReservations);
      } else {
        console.error('Erreur lors de la validation du produit.');
      }
    } catch (error) {
      console.error('Erreur lors de la validation du produit :', error);
    }
  }
};
const refuserReservation = async (id) => {
  const confirmation = window.confirm('Voulez-vous refuser ce produit?');
  if (confirmation) {
    try {
      const response = await fetch(`http://localhost:3000/reservations/${id}/refuser`, { method: 'PUT' });
      if (response.status === 200) {
        // Mettre à jour les réservations après validation
        const updatedReservations = reservations.map((reservation) =>
          reservation.id === id ? { ...reservation, statut: 'refusé' } : reservation
        );
        setReservations(updatedReservations);
      } else {
        console.error('Erreur lors du refus du produit.');
      }
    } catch (error) {
      console.error('Erreur lors du refus du produit :', error);
    }
  }
};
```

Outils Utilisés :

fetch API :

Comme dans la fonction validerReservation, l'API fetch est utilisée pour effectuer une requête HTTP de type PUT vers le serveur.

La méthode PUT est utilisée pour indiquer au serveur de refuser la réservation.

Confirmation de Fenêtre (window.confirm) :

Avant d'envoyer la requête de refus, la fonction affiche une fenêtre de confirmation (window.confirm) pour s'assurer que l'utilisateur souhaite vraiment refuser la réservation.

Mise à Jour Réactive du Statut :

Après le refus réussi du côté serveur (statut 200), la fonction met à jour réactivement les réservations dans le composant en utilisant setReservations.

Elle crée une nouvelle liste mise à jour en modifiant le statut de la réservation spécifique à 'refusé' et utilise setReservations pour mettre à jour le composant.

Ces fonctions utilisent une approche réactive pour mettre à jour l'interface utilisateur après avoir interagi avec le serveur.

La fenêtre de confirmation offre une couche de sécurité supplémentaire pour éviter des actions indésirables.

Ces outils et approches garantissent une interaction fluide et sécurisée avec le serveur pour valider ou refuser les réservations.