

Documentation Technique M2L

1. Description générale

2. La base de données

3. L'API REST

4. Architecture

1. Description générale de l'application

Nom du site : Maison des Liges

Présentation : Le site web de la Maison des Liges est un site de commerce électronique qui permet aux utilisateurs de parcourir, rechercher et acheter une variété de produits en ligne en lien avec les pratiques sportives des ligues. Avec une interface conviviale et intuitive, il offre une expérience de magasinage pratique et sécurisée pour les clients.

Fonctionnalités principales :

- **Catalogue de produits :** La Maison des Liges propose un large éventail de produits, allant des vêtements aux accessoires de sports, et bien plus encore.
- **Création de compte utilisateur :** Les utilisateurs peuvent créer un compte pour pouvoir interagir avec le panier et passer des commandes
- **Panier d'achat :** Les utilisateurs peuvent ajouter des produits à leur panier d'achat et passer ensuite à la caisse pour finaliser leur commande. Le panier d'achat conserve les produits sélectionnés même lorsque l'utilisateur quitte le site et revient plus tard.
- **Dashboard Admin - Gestion des Produits :** le dashboard offre aux administrateurs un ensemble d'outils puissants pour gérer efficacement les produits disponibles sur la plateforme.

Les administrateurs peuvent facilement ajouter de nouveaux produits à la plateforme en remplissant un formulaire détaillé. Ils peuvent spécifier des informations telles que le nom du produit, la description, les images, les prix, les quantités disponibles, etc.

Les administrateurs ont la possibilité de modifier les détails des produits existants à tout moment. Ils peuvent mettre à jour des informations telles que le prix, la description, les images, la disponibilité, etc.

En cas de besoin, les administrateurs peuvent supprimer des produits de la plateforme. Cette fonctionnalité est utile pour retirer les produits discontinués ou obsolètes de la liste des produits disponibles.

2. La base de données

La base de données est composée de trois tables :

Table : compte

Description : cette table stocke les informations des utilisateurs enregistrés sur la plateforme en afin les authentifier pour leurs futures interactions

```
CREATE TABLE IF NOT EXISTS `compte` (  
  `uuid` varchar(32) NOT NULL,  
  `prenom` varchar(50) DEFAULT NULL,  
  `nom` varchar(50) DEFAULT NULL,  
  `email` varchar(50) DEFAULT NULL,  
  `mdp` varchar(100) DEFAULT NULL,  
  `estadmin` int(11) DEFAULT NULL,  
  PRIMARY KEY (`uuid`)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;
```

Table : produit

Description : Cette table stocke les informations sur les produits disponibles à l'achat afin d'afficher les produits sur le site, gérer les stocks et effectuer des transactions d'achat.

```
CREATE TABLE IF NOT EXISTS `produit` (  
  `uuid` varchar(32) NOT NULL,  
  `nom` varchar(50) DEFAULT NULL,  
  `description` varchar(50) DEFAULT NULL,  
  `prix` int(11) DEFAULT NULL,  
  `quantite` int(11) DEFAULT NULL,  
  `visibilite` int(11) DEFAULT NULL,  
  `image` varchar(100) DEFAULT NULL,  
  PRIMARY KEY (`uuid`)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;
```

Table : panier

Description : cette table stocke les détails des paniers d'achat des utilisateurs et est liée aux deux tables précédentes.

```
CREATE TABLE IF NOT EXISTS `panier` (  
  `id` int(11) NOT NULL,  
  `uuid_compte` varchar(32) DEFAULT NULL,  
  `uuid_produit` varchar(32) DEFAULT NULL,  
  `quantite` int(11) DEFAULT NULL,  
  PRIMARY KEY (`id`),  
  KEY `FK_panier_produit` (`uuid_produit`),  
  KEY `FK_panier_compte` (`uuid_compte`),  
  CONSTRAINT `FK_panier_compte` FOREIGN KEY (`uuid_compte`) REFERENCES `compte` (`uuid`) ON DELETE CASCADE ON UPDATE CASCADE,  
  CONSTRAINT `FK_panier_produit` FOREIGN KEY (`uuid_produit`) REFERENCES `produit` (`uuid`) ON DELETE CASCADE ON UPDATE CASCADE  
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;
```

3. L'API Rest

L'application communique avec la base de données via une API créée avec NodeJS et Express.

```
const express = require('express');
const cors = require('cors');
const app = express();
const mariadb = require("mariadb");
const bcrypt = require("bcrypt");
const crypto = require("crypto");
const jwt = require('jsonwebtoken');
const multer = require('multer');
```

Nos informations importantes comme la clé des tokens ou les informations de connexion à la base de données sont cachées dans un fichier .env

```
const secretKey = process.env.JWT_SECRET_KEY;

const pool = mariadb.createPool({
  host: process.env.DB_HOST,
  database : process.env.DB_DTB,
  user: process.env.DB_USER,
  password: process.env.DB_PWD,
  connectionLimit: 1000
});
```

On utilise un middleware servant à authentifier les personnes connectés grâce à un jeton de connexion temporaire avec JsonWebToken.

```
function verifierJWT(req, res, next) {
  const token = req.headers.authorization.substring(7);
  if (!token) {
    return res.status(401).json({ message: 'Pas de token, accès non autorisé' });
  }

  try {
    const decoded = jwt.verify(token, secretKey);
    req.user = decoded;
    next();
  } catch (error) {
    return res.status(403).json({ message: 'Token invalide' });
  }
}
```

La gestion des comptes (inscription, test des doublons de mail, connexion et récupération des autorisations du compte pour savoir s'il est administrateur)

```
> app.post('/inscription', async(req,res)=>{ ...
});

> app.post('/inscription/mail', async(req,res)=>{ ...
});

> app.post('/connexion', async(req,res)=>{ ...
});

> app.get('/autorisation/:token', verifierJWT, async(req,res)=>{ ...
});
```

On utilise bcrypt pour comparer le mot de passe fourni au mot de passe hashé en base de données.

```
app.post('/connexion', async(req,res)=>{
  let conn;
  try {
    const {email,mdp} = req.body
    conn = await pool.getConnection();

    const rows = await conn.query(
      "SELECT * FROM compte WHERE email=?",
      [email]
    );

    if (rows.length === 1) {
      const match = await bcrypt.compare(mdp, rows[0]["mdp"]);
```

La gestion des produits (récupération des infos des produits, suppression, ajout au panier et récupération d'un produit spécifique)

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

> app.get('/produit/:uuid', verifierJWT, async(req,res)=>{ ...
});

> app.delete('/produit/:uuid', verifierJWT, async(req,res)=>{ ...
});

> app.post('/produit/panier', verifierJWT, async(req,res)=>{ ...
});
```

La gestion du panier (récupérer les infos du panier, le supprimer, supprimer un produit spécifique, valider le panier et décrémenter les stocks)

```
app.get('/panier/:token', verifierJWT, async(req,res)=>{ ...
});

app.delete('/panier/compte/:token', verifierJWT, async(req,res)=>{ ...
});

app.delete('/panier/compte/produit/:idPanier', async(req,res)=>{ ...
});

app.put('/panier/validation/:token', verifierJWT, async(req,res)=>{ ...
});

app.get('/historique/:token', verifierJWT, async(req,res)=>{ ...
});
```

La gestion du dashboard administrateur (récupérer les infos des produits , récupérer les infos des utilisateurs, ajouter un produit au site, modifier un produit)

```
app.get('/dashboard/produits',verifierJWT, async(req,res)=>{ ...
});

app.get('/dashboard/utilisateurs',verifierJWT, async(req,res)=>{ ...
});

app.delete('/john', async (req, res) => { ...
});

app.post('/dashboard/ajouterproduit', verifierJWT, upload.single("image"), async (req, res) => { ...
});

app.put('/dashboard/modifierproduit/:uuid', verifierJWT, async(req,res)=>{ ...
});
```

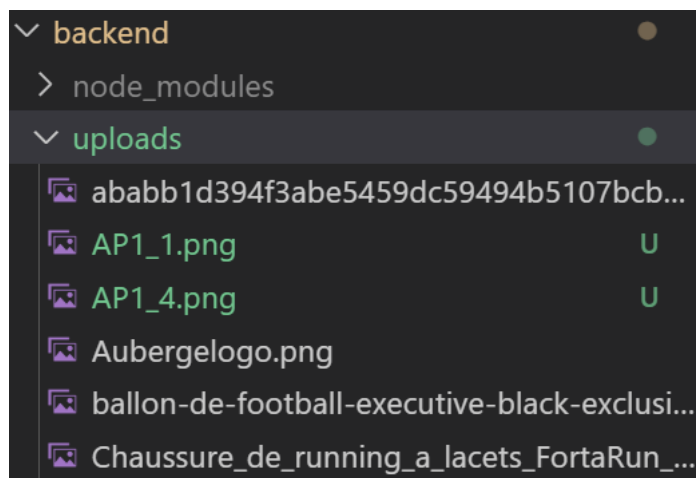
La gestion des images se fait avec multer, on les stocke dans un dossier uploads.

```
app.use('/uploads', express.static('uploads'));

// ----- Setup de Multer -----

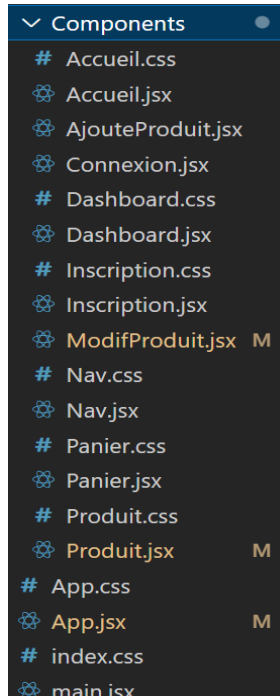
const storage = multer.diskStorage({
  destination: function (req, file, cb) {
    cb(null, 'uploads/');
  },
  filename: function (req, file, cb) {
    cb(null, file.originalname);
  }
});

const upload = multer({ storage: storage });
```



4. Architecture

Voici les composants de notre application en frontend :



Voici nos routes dans l'application

```
<Nav estConnecte={estConnecte} estAdmin={estAdmin} updateNavState={updateNavState} setEstAdmin={setEstAdmin}></Nav>
<Routes>
  <Route path="/" element={<Accueil></Accueil>}></Route>
  <Route path="/connexion" element={<Connexion updateNavState={updateNavState}></Connexion>}></Route>
  <Route path="/inscription" element={<Inscription></Inscription>}></Route>
  <Route path="/panier" element={<Panier></Panier>}></Route>
  <Route path="/produit" element={<Produit estConnecte={estConnecte}></Produit>}></Route>
  <Route path="/dashboard" element={<Dashboard></Dashboard>}></Route>
  <Route path="/ajouterproduit" element={<AjouteProduit></AjouteProduit>}></Route>
  <Route path="/modifierproduit/:uuid" element={<ModifProduit></ModifProduit>}></Route>
</Routes>
```

estConnecte est un hook d'état permettant de savoir si l'utilisateur est connecté à partir d'un token valide qu'on met dans la barre de navigation pour accéder aux services des utilisateurs connectés. De même estAdmin est un hook d'état pour savoir si l'utilisateur est administrateur.