AP3 - CDC

Cahier des charge ayant pour objectif l'étude et l'accompagnement dans la réalisation de l'Atelier Professionnel n°3 – M2L.

BERNIER FLORENT | BTS SIO SLAM

Table of Contents

Expression des besoins

Pour ce 3e Atelier Professionnel, nous allons devoir réaliser un site e-commerce à l'aide de React et NodeJS.

Ce projet a pour objectif de développer nos compétences à travailler sur un langage différent et de s'organiser.

Le site devra permettre aux utilisateurs de se connecter à leur espace. De plus, un catalogue sera accessible sur le site proposant aux utilisateurs inscris de pouvoir acheter un ou plusieurs objets disponibles

L'administrateur pourra ajouter de nouveaux objets dans le catalogue, de les modifier ou bien les supprimer du catalogue dans une section privée.

L'utilisateur aura la possibilité de modifier ses informations sur son profil, d'ajouter des objets dans son panier ou de visualiser ses historiques de commandes.

Les rôles obligatoires

- **Administrateur (2)** : Il aura la possibilité d'ajouter, modifier ou supprimer un utilisateur, un staff, un produit, et a l'accès à la base de donnée.
- **Staff (1)** : Il aura la possibilité d'ajouter, modifier ou supprimer un produit, mais n'a pas la possibilité de contrôler les utilisateurs ou staff. De plus, il aura accès à des statistiques sur le dashboard.
- **Utilisateur (0)** : Il peut acheter un produit, modifier son profil, et voir son historique de commande.

Les différents diagrammes

Les diagrammes de cas d'utilisations

L'objectif de faire un diagramme d'utilisation, c'est de montrer les différentes façons dont un utilisateur peut interagir avec le système, c'est-à-dire, ce qu'il va pouvoir faire lorsqu'il naviguera sur le site internet : acheter un objet, modifier son profil, voir ses historiques de commandes, ajouter un utilisateur, … etc.

Diagramme de cas d'utilisation - Utilisateur

On peut le voir, nous avons deux types d'utilisateurs :

- Les utilisateurs non inscris.
- Les utilisateurs inscris.

Les utilisateurs non inscris sont les nouveaux utilisateurs ou des utilisateurs non connectés. Ils ont la possibilité d'accéder au catalogue d'objet, mais n'ont pas la possibilité d'en ajouter dans son panier (ni même d'acheter un objet).

Les utilisateurs inscris, à la différence des utilisateurs non inscris, ont plus d'accès. Ils ont la possibilité, tout comme un utilisateur lambda, d'accéder au catalogue, avec la possibilité en plus d'ajouter un ou plusieurs objets dans son panier. De plus, il peut accéder à son compte pour modifier ses informations ou bien voir son historique de commande.

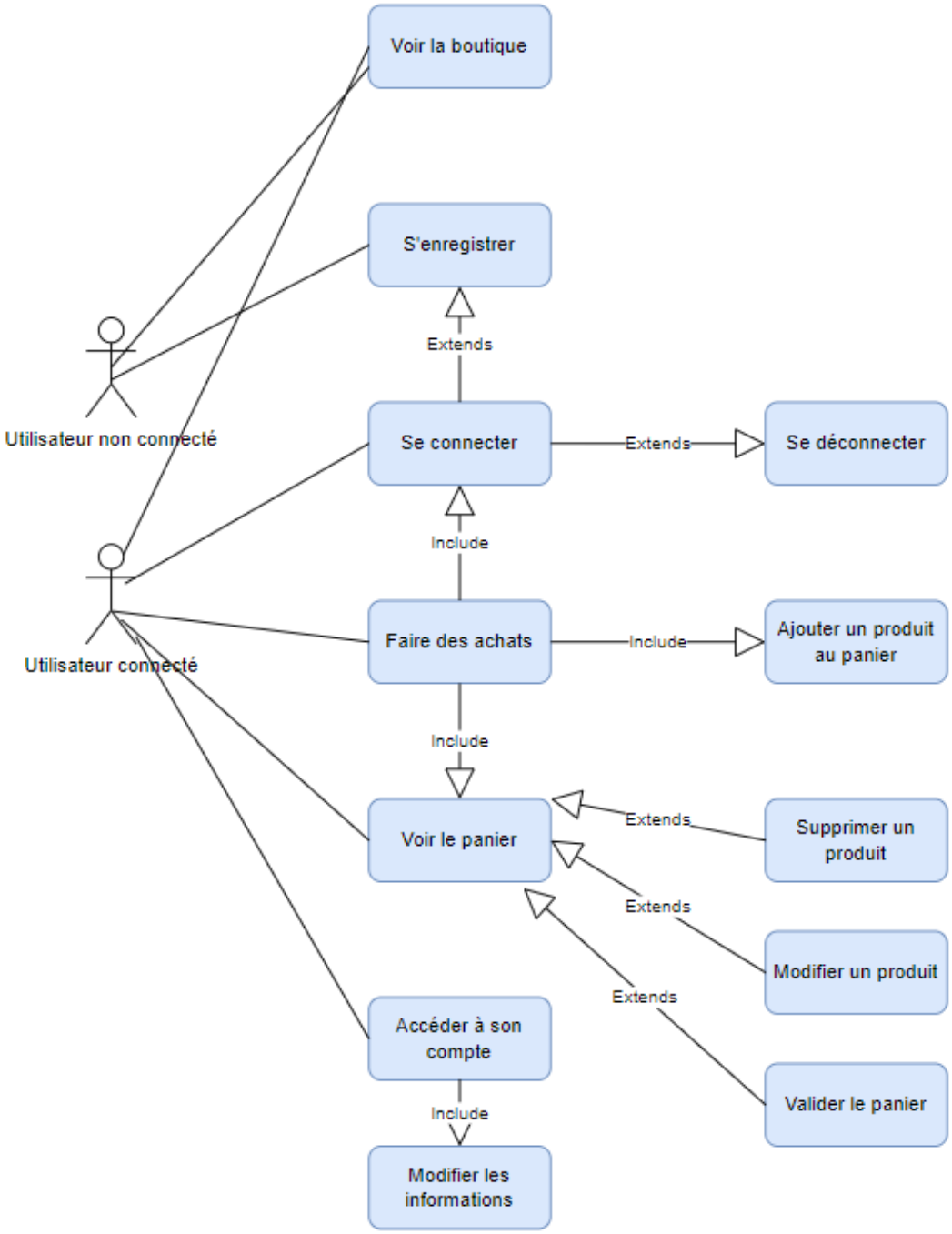


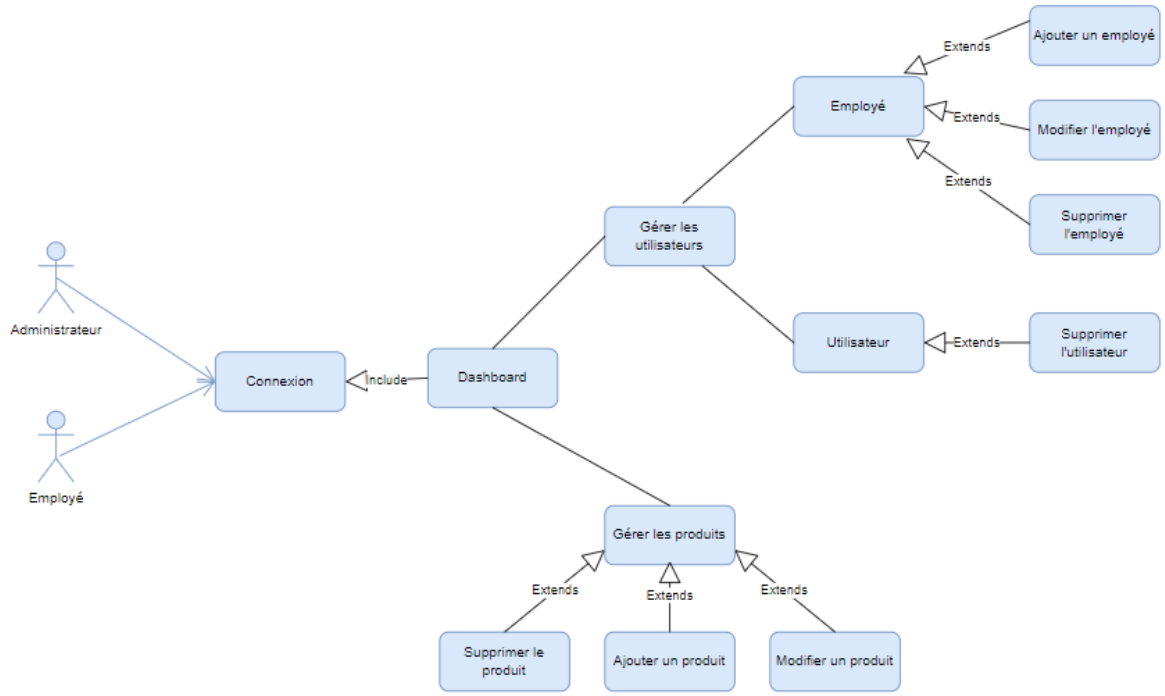
Diagramme de cas d'utilisation – Administrateur

Nous avons un acteur, disposant de différentes permissions :

Après connexion, il a la possibilité d'accéder à un dashboard (page permettant à l'administration d'obtenir plusieurs informations sur l'ensemble du site) dans lequel il pourra faire différentes actions :

- Il peut gérer les commandes (accepter ou refuser une commande et voir l'historique des commandes).
- Il peut gérer les utilisateurs (ajouter, modifier, et supprimer un utilisateur et un staff).

- Il peut gérer les items (ajouter, modifier et supprimer un item, ainsi que voir le stock des items).

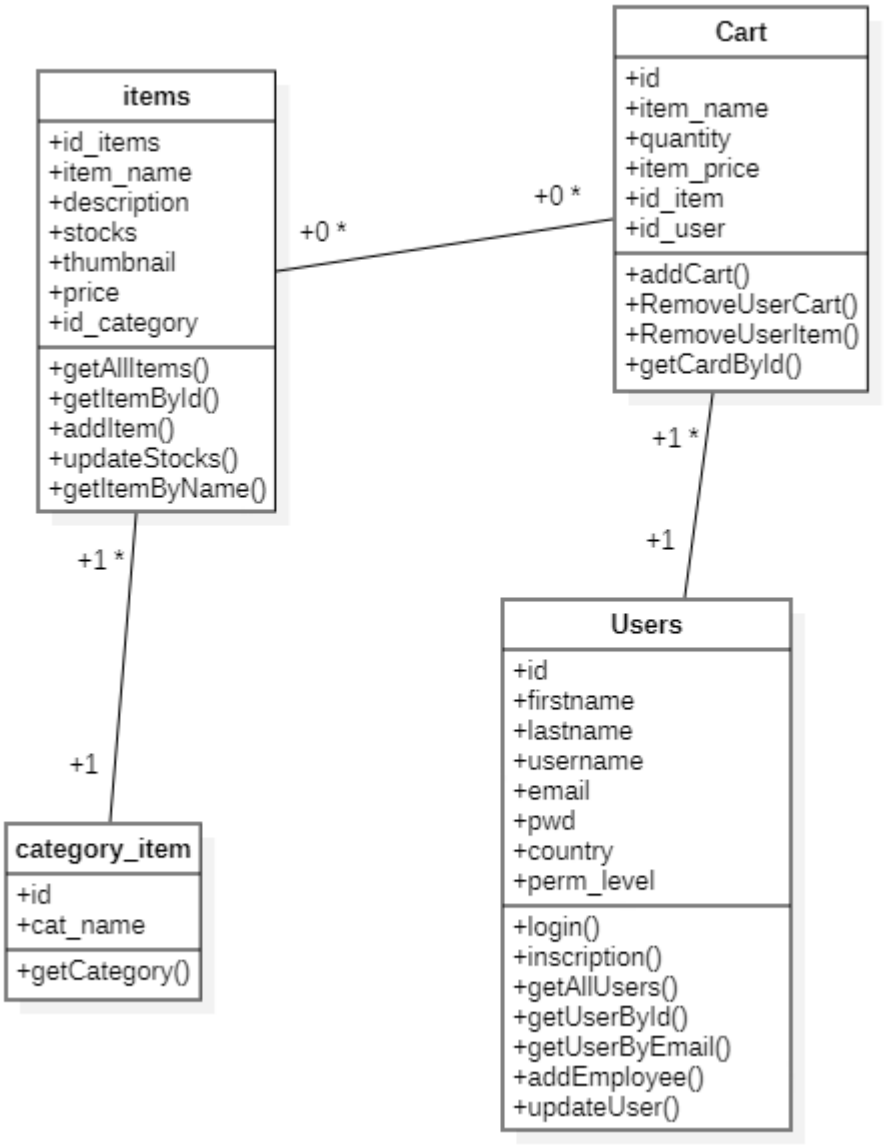


Le diagramme de classe

L'objectif d'un diagramme de classe est de présenter les classes et les interfaces des systèmes ainsi que leurs relations, c'est-à-dire de montrer à quoi ressemblera la base de donnée.

Il y a différentes tables disponibles dans le diagramme de classe :

- category_item
- Items
- Cart
- Users

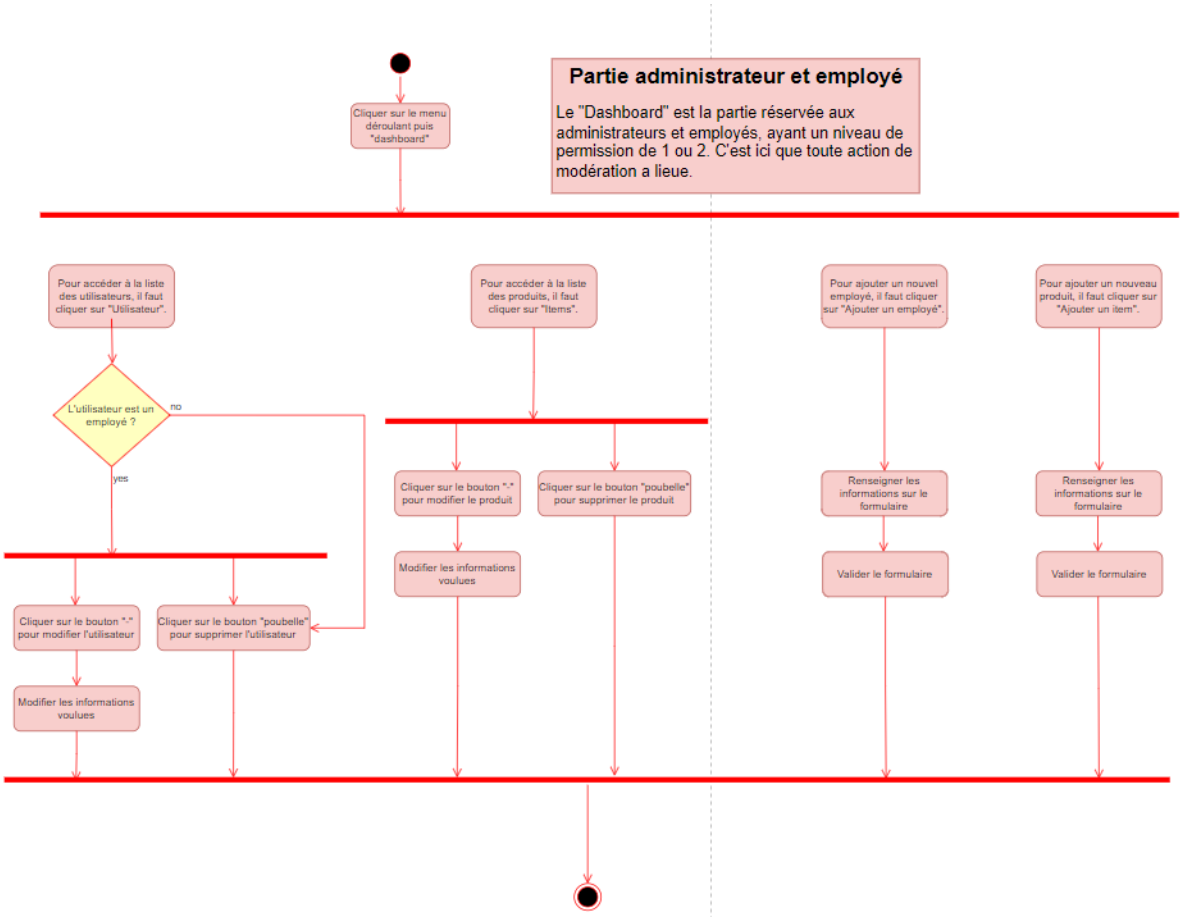


Les Diagramme d'activités

Le digramme d'activité est utilisé pour montrer les étapes d'un processus sur le site internet, c'est-à-dire de fournir une vue pour chaque séquence du code.

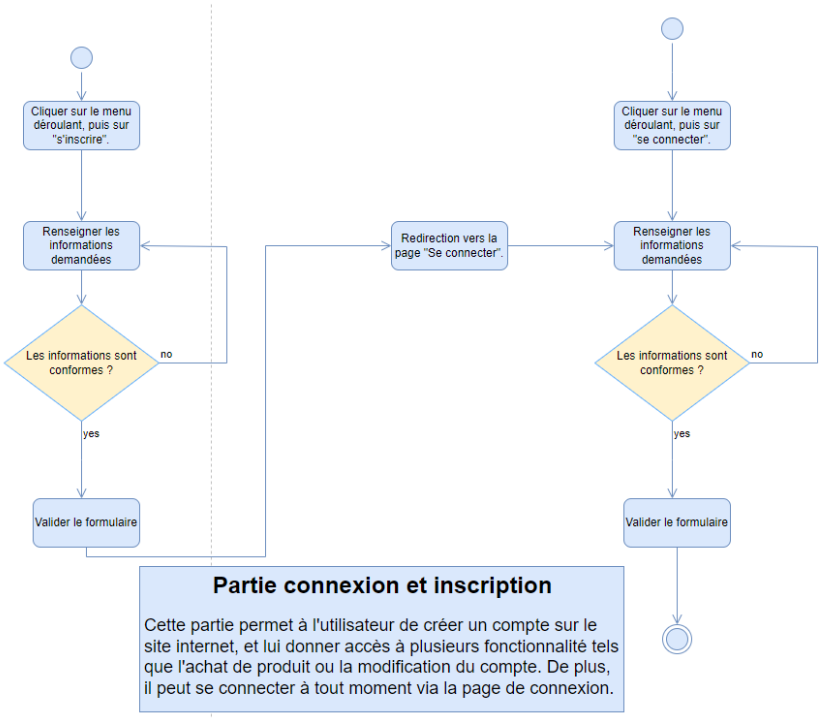
Partie Administrateur

- A gauche, on voit les étapes pour la gestion des items.
- Au milieu, c'est les étapes pour la gestion des utilisateurs.
- A droite, c'est les étapes pour la gestion des commandes.



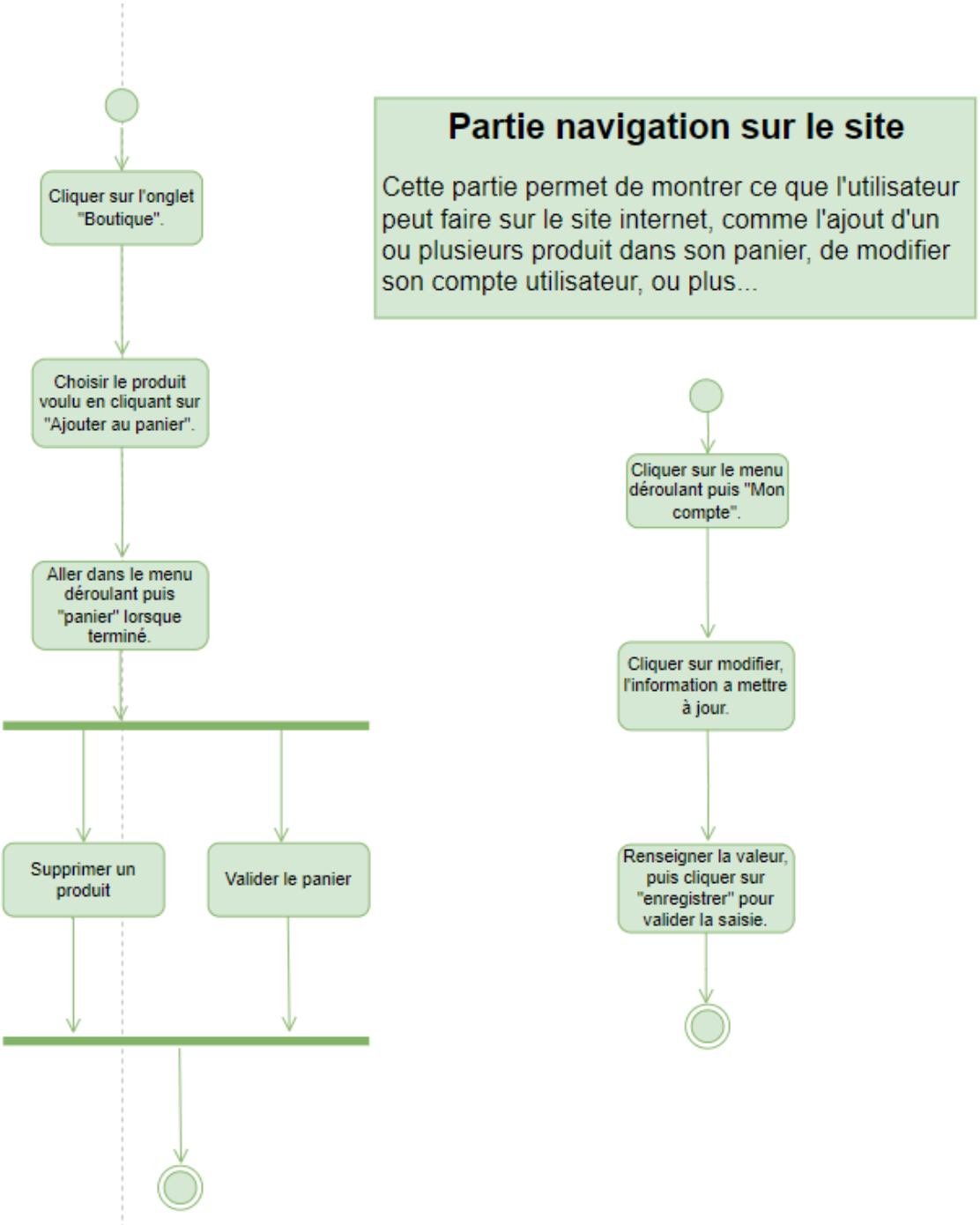
Partie Connexion

- A gauche, on voit les étapes pour l'inscription et la connexion d'un utilisateur.
- A droite, c'est les étapes lors d'une déconnexion de l'utilisateur.



Partie Utilisateur

- A gauche, c'est les étapes pour l'ajout d'un objet dans le panier.
- Au milieu, les étapes pour la validation de la commande, ou la suppression/modification d'un item.
- A droite, c'est les étapes pour la mise à jour du compte utilisateur.



Détails de la base de donnée

Table utilisateurs

La table « users » est utilisé pour stocké les informations de l'utilisateur :

- L'ID de l'utilisateur, mis en « AUTO_INCREMENT » qui augmentera à chaque nouvel utilisateur ajouté.
- Le prénom et nom de famille, qui sont le prénom et le nom de l'utilisateur, mis par défaut sur « NULL », n'étant pas obligatoire pour l'utilisateur.
- Le pseudonyme et l'email, étant tout deux utilisés pour la connexion à son compte utilisateur, et son unique évitant la création de plusieurs comptes sur la même adresse mail, et l'utilisation de plusieurs fois le même nom d'utilisateur.
- Le mot de passe est le texte caché, permettant à l'utilisateur de se connecter à son compte, devant être haché pour éviter l'usurpation d'identité.
- Le pays, permettant à l'utilisateur d'indiquer son pays résident, qui sera majoritairement « France » dans le cas du projet.
- Le niveau de permission, qui par défaut est défini à 0. Ce paramètre indique le pouvoir que le compte utilisateur possède :
    - Niveau 0 : Utilisateur
    - Niveau 1 : Staff
    - Niveau 2 : Administrateur

#	Nom	Type de données	Taille/Ensem...	Non signé	NULL autorisé	ZEROFILL	Par défaut
1	id	INT	10	☐	☐	☐	AUTO_INCREME...
2	firstname	VARCHAR	40	☐	☑	☐	NULL
3	lastname	VARCHAR	40	☐	☑	☐	NULL
4	username	VARCHAR	40	☐	☐	☐	Pas de défaut
5	email	VARCHAR	40	☐	☐	☐	Pas de défaut
6	pwd	VARCHAR	250	☐	☐	☐	Pas de défaut
7	country	VARCHAR	40	☐	☑	☐	NULL
8	perm_level	INT	11	☐	☐	☐	'0'

Table category_item

La table « category_item » permet de stocker les différentes catégories d'item :

- L'ID, mis en « AUTO_INCREMENT », qui définit l'id de la catégorie.
- Le nom de la catégorie (par exemple : ballon, vêtement, équipements, … etc.).

#	Nom	Type de données	Taille/Ensem...	Non signé	NULL autorisé	ZEROFILL	Par défaut
1	id	INT	10	☐	☐	☐	AUTO_INCREME...
2	name	VARCHAR	40	☐	☐	☐	''

Table items

La table « items » permet de stocker les informations d'un article :

- L'ID, mis en « AUTO_INCREMENT », qui définit l'id de l'article.
- Le nom de l'article, qui sera affiché sur le site.
- La description de l'article, affichée sur le site donnant des informations sur l'article.
- Les stocks, permettant d'afficher le nombre d'article restant.
- L'image de l'article, affichant à quoi ressemble l'article.
- Le prix de l'article, indiquant à l'utilisateur le prix de l'article.

#	Nom	Type de données	Taille/Ensem...	Non signé	NULL autorisé	ZEROFILL	Par défaut
1	**id_items**	**INT**	**10**	☐	☐	☐	**AUTO_INCREME...**
2	item_name	VARCHAR	40	☐	☐	☐	Pas de défaut
3	description	LONGTEXT		☐	☐	☐	Pas de défaut
4	stocks	INT	10	☐	☐	☐	Pas de défaut
5	thumbnail	LONGTEXT		☐	☑	☐	NULL
6	price	FLOAT		☐	☐	☐	Pas de défaut
7	id_category	INT	11	☐	☑	☐	NULL

Table panier

La table « cart » permet d'afficher le panier d'un utilisateur :

- L'ID.
- Le nom du produit.
- La quantité d'article ajoutés.
- Le prix total des articles dans le panier.
- L'id_item, faisant référence à l'article ajouté dans le panier.
- L'id_user, étant l'identifiant de l'utilisateur ayant ajouté le produit.

#	Nom	Type de données	Taille/Ensem...	Non signé	NULL autorisé	ZEROFILL	Par défaut
1	**id**	**INT**	**11**	☐	☐	☐	**AUTO_INCREME...**
2	item_name	VARCHAR	50	☐	☐	☐	Pas de défaut
3	quantity	INT	11	☐	☐	☐	Pas de défaut
4	item_price	FLOAT		☐	☐	☐	Pas de défaut
5	id_item	INT	11	☐	☐	☐	Pas de défaut
6	id_user	INT	11	☐	☐	☐	Pas de défaut

Ressources et code

Connexion à la BDD

Pour des raisons de sécurité, les constantes de connexion sont dans un dossier .env.

On récupère les constantes de connexion du fichier .env pour les insérer dans la constante « pool »

ConnexionDB.js

```
require('dotenv').config();
const mariadb = require('mariadb');

// Connexion à la BDD
const pool = mariadb.createPool({
    host: process.env.DB_HOST,
    database: process.env.DB_BDD,
    user: process.env.DB_USER,
    password: process.env.DB_PASSWORD,
    connectionLimit: 100,
    connectTimeout: 30000,
});
console.log(pool)
module.exports = {pool: pool};
```

Page server.js

Le serveur est centralisé sur une seule page : server.js. C'est sur cette page qu'on lance le serveur et où toutes les routes sont rassemblés afin d'être utilisé.

server.js

```
const express = require('express');
const app = express();
const cors = require('cors');
const userRoute = require('./routes/userRoute');
const path = require('path');
const boutiqueRoute = require('./routes/boutiqueRoute');
const articleRoute = require('./routes/articleRoute');

const whiteList = ['http://192.168.1.28:8001', 'http://localhost:3000'];
const corsOptions = {
   origin: whiteList,
   credentials: true,
   optionsSuccessStatus: 200,
};
app.use(cors(corsOptions));
app.use('/upload', express.static(path.join(__dirname, 'upload')));

app.use('/users', userRoute);
app.use('/boutique',boutiqueRoute);
app.use('/article', articleRoute);

const port = 8005;
module.exports = app.listen(port, () => {
    console.log(`Le serveur a été démarré sur le port ${port}`);
});
```

Les routes & controllers

Les routes – userRoute.js

La route « userRoute.js » est le fichier où toutes les fonctions du controller utilisateur sont rassemblés. On y retrouve la connexion ou l'inscription d'un utilisateur, la récupération de tous les utilisateurs ou d'un utilisateur en particulier, ou encore le suppression et la modification d'un utilisateur.

Ces fonctions sont accessibles grâce à l'url suivante : « url/users/.. ».

userRoute.js

```
const express = require('express');
const router = express.Router();
const userDAO = require('../controllers/userDAO')
const middlewareAuth = require('../middleware/authorization')
const middlewareAdmin = require('../middleware/isAdmin');

router.use(express.json());

router.get('/', middlewareAuth.authenticator, middlewareAdmin.isAdmin, userDAO.getAllUsers);
router.get('/id/:id', middlewareAuth.authenticator, middlewareAdmin.isAdmin, userDAO.getUserById);
router.get('/email/:email', middlewareAuth.authenticator, middlewareAdmin.isAdmin,
userDAO.getUserByEmail);
router.post('/inscription', userDAO.inscription);
router.post('/login', userDAO.login);
router.post('/addemployee', middlewareAuth.authenticator, middlewareAdmin.isAdmin,
userDAO.addEmployee);
router.put('/updateUser/:id', middlewareAuth.authenticator, userDAO.updateUser);
router.delete('/deleteUser/:id', middlewareAuth.authenticator, middlewareAdmin.isAdmin,
userDAO.deleteUser);

module.exports = router;
```

Les routes – boutiqueRoute.js

Même principe que la route utilisateur, ce fichier rassemble toutes les fonctions du controller produit, où on a l'ajout, la modification, la suppression, ou la récupération des produits.

On utilise « multer » pour l'ajout d'une image dans la base de donnée.

Ces fonctions sont accessibles grâce à l'url suivante : « url/boutique/.. ».

boutiqueRoute.js

```
const express = require('express');
const path = require('path');
const multer = require('multer');
const router = express.Router();
const itemDAO = require('../controllers/itemDAO');
const middlewareAuth = require('../middleware/authorization')
const middlewareAdmin = require('../middleware/isAdmin');

const storage = multer.diskStorage({
    destination: (req, file, cb) => {
        cb(null, 'upload');
    },
    filename: (req, file, cb) => {
        console.log(file)
        cb(null, 'img-' + Date.now() + path.extname(file.originalname));
    }
});

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

router.use(express.json());

router.get('/', itemDAO.getAllItems);
router.get('/id/:id',itemDAO.getItemById);
router.post('/additem', middlewareAuth.authenticator, middlewareAdmin.isAdmin,
upload.single('thumbnail'), itemDAO.addItem);
router.get('/category', middlewareAuth.authenticator, itemDAO.getCategory);
router.put('/updatestocks/:id', middlewareAuth.authenticator, middlewareAdmin.isAdmin,
itemDAO.updateStocks);
router.get('/s', middlewareAuth.authenticator,itemDAO.getItemByName);
router.delete('/deleteitem/:id', middlewareAuth.authenticator, middlewareAdmin.isAdmin,
itemDAO.deleteItem);
router.put('/updateitem/:id', middlewareAuth.authenticator, middlewareAdmin.isAdmin,
itemDAO.updateItem);

module.exports = router;
```

Les routes – articleRoute.js

Le fichier « articleRoute.js » est la route où on peut insérer des produits dans le panier, supprimer un produit, ou valider le panier.

Ces fonctions sont accessibles grâce à l'url suivante : « url/article/.. ».

articleRoute.js

```
const express = require('express');
const router = express.Router();
const articleDAO = require('../controllers/articleDAO');
const middlewareAuth = require('../middleware/authorization')
const middlewareAdmin = require('../middleware/isAdmin');

router.use(express.json());

router.post('/add', middlewareAuth.authenticator, articleDAO.addCart);
router.get('/getCart/:id', articleDAO.getCartById);
router.delete('/deleteUserCart/:id', middlewareAuth.authenticator, articleDAO.removeUserCart);
router.delete('/deleteUserItem/:id', middlewareAuth.authenticator, articleDAO.removeUserItem);

module.exports = router;
```

Le controller – userDAO.js

Les controllers sont des fichiers dans lesquels se trouvent toutes les requêtes qu'on souhaite effectuer. Par exemple la récupération d'un utilisateur, par un identifiant ou par son adresse mail, ou encore la connexion, comme ci-dessous.

C'est ici que toutes les requêtes sont effectuées, comme ici pour vérifier si le compte existe, et ensuite connecter l'utilisateur.

userDAO.js

```
const connexion = require('../db/connexionDB');
const bcrypt = require('bcrypt');
const jwt = require('jsonwebtoken');

exports.login = async (req, res) => {
    try{
        const {email, password} = req.body;
        conn = await connexion.pool.getConnection()
        const result = await conn.query(`SELECT * FROM users WHERE email = ?`, [email]);
        conn.release()
        if (result.length === 0) {
            return res.status(401).json({error: 'Cet utilisateur n\'existe pas.'})
        }
        const user = result[0];
        const passwordMatch = await bcrypt.compare(password, user.pwd);
        if (!passwordMatch) {
            return res.status(401).json({error: 'Mot de passe incorrect.'})
        }
        const playload = {
            userId: user.id,
            username: user.username,
            firstname: user.firstname,
            lastname: user.lastname,
            email: user.email,
            permLevel: user.perm_level,
            isConnected: true,
            password: user.pwd,
        }
        console.log(playload)
        const token = jwt.sign(playload, process.env.API_KEY, {expiresIn: '1h'});
        res.json({token, user: playload})
    } catch(error){
        console.error(error)
        res.status(500).json({error: 'Erreur lors de la connexion.'})
    }
}
```

Le controller – itemDAO.js

itemDAO.js

```
const connexion = require('../db/connexionDB');

exports.getAllItems = async (req, res) => {
    try {
        conn = await connexion.pool.getConnection();
        const items = await conn.query('SELECT * FROM items INNER JOIN category_item ON
items.id_category = category_item.id');
        res.status(200).json(items)
    }
    catch (err) {
        console.log(err);
        res.status(500).send("Erreur lors de l'affichage des items.");
    }
}
```

Le controller – articleDAO.js

articleDAO.js

```
const connexion = require('../db/connexionDB');

exports.addCart = async (req, res) => {
    try {
        const { itemId, item_name, quantity, basePrice, userId } = req.body;
        conn = await connexion.pool.getConnection();
        const query = `
            INSERT INTO cart(item_name, quantity, item_price, id_item, id_user)
            VALUES (?, ?, ?, ?, ?)
        `;
        await conn.query(query, [item_name, quantity, basePrice, itemId, userId]);

        res.status(200).json({ message: 'L\'article a été ajouté au panier avec succès.' });
    } catch (error) {
        console.error('Erreur lors de l\'ajout de l\'article au panier : ', error);
        res.status(500).json({ error: 'Une erreur est survenue lors de l\'ajout de l\'article au
panier.' });
    }
};
```

Les middlewares

Vérification de l'administrateur

Le middleware « isAdmin » est mis sur les routes. Il permet de limiter l'utilisateur des fonctions/routes par les personnes n'étant pas considéré comme administrateur ou employé.

A l'aide du token de l'utilisateur, on récupère sont niveau de permission et son adresse mail pour vérifier qu'un utilisateur ressort. Si c'est le cas, l'accès lui est autorisé, sinon un message d'erreur apparaît.

isAdmin.js

```
const connexion = require('../db/connexionDB');
const jwt = require('jsonwebtoken');

exports.isAdmin = async (req, res, next) => {
    const token = req.query.token || req.headers.authorization;
    if (!token) {
        return res.status(401).json({ error: 'Unauthorized. Token not provided'});
    }
    try {
        const tokenValue = token.replace('Bearer ', '');

        const decodedToken = jwt.verify(tokenValue, process.env.API_KEY);

        const email = decodedToken.email;

        conn = await connexion.pool.getConnection();
        const result = await conn.query('SELECT perm_level FROM users WHERE email = ?', [email]);
        conn.release();
        console.log(result[0].perm_level)
        if (result.length === 1 && (result[0].perm_level === 1 || result[0].perm_level === 2)) {
            next();
        } else {
            return res.status(403).json({ error: 'Unauthorized. You are not an admin'});
        }
    } catch (error) {
        console.error(error);
        res.status(500).json({error: 'Internal server error.'})
    }
};
```

Vérification de connexion.

authorization.js

```
const jwt = require('jsonwebtoken');

exports.authenticator = (req, res, next) => {
    const token = req.query.token ? req.query.token : req.headers.authorization;

    if (token && process.env.API_KEY) {
        jwt.verify(token, process.env.API_KEY, (err, decoded) => {
            if (err) {
                res.status(401).json({ error: 'Unauthorized' });
            } else {
                req.user = decoded;
                next();
            }
        });
    } else {
        res.status(401).json({ error: 'Unauthorized' });
    }
};
```