etape mini projet 1 AAW :

1ere etape :

1. \*\*Création d'un nouveau répertoire\*\* :

- Ouvrez Visual Studio Code.

- Dans la barre de menus, sélectionnez "Fichier" > "Ouvrir un dossier...".

- Choisissez l'emplacement où vous souhaitez créer votre projet.

- Cliquez sur "Sélectionner un dossier" pour ouvrir cet emplacement dans Visual Studio Code.

2. \*\*Ouverture du terminal intégré à Visual Studio Code\*\* :

- Une fois que vous avez ouvert votre dossier de projet dans Visual Studio Code, vous pouvez ouvrir le terminal intégré en sélectionnant "Terminal" dans la barre de menus, puis "Nouveau terminal".

3. \*\*Initialisation d'un nouveau projet Node.js avec npm\*\* :

- Dans le terminal intégré à Visual Studio Code, assurez-vous d'être dans le répertoire de votre projet.

- Ensuite, exécutez la commande `npm init` comme décrit précédemment pour initialiser un nouveau projet Node.js.

Une fois que vous avez initialisé votre projet Node.js, voici ce que vous pouvez faire ensuite :

2 eme etape:

1. \*\*Installation des dépendances\*\* :

- Maintenant que vous avez initialisé votre projet, vous devrez installer les dépendances nécessaires, à savoir Express.js et EJS. Vous pouvez le faire en exécutant la commande suivante dans le terminal intégré à Visual Studio Code :

```

npm install express ejs

```

Cela installera Express.js et EJS dans votre projet, ce qui vous permettra de les utiliser pour développer votre application web.

2. \*\*Création de la structure du projet\*\* :

- Créez les répertoires `models`, `views`, `controllers`, et `public` dans votre projet, comme décrit précédemment.

- À l'intérieur du répertoire `public`, créez un sous-répertoire `css` pour stocker vos fichiers de style CSS.

- Dans le répertoire `views`, créez des fichiers `.ejs` pour chaque vue de votre application, telles que `productList.ejs`, `productDetails.ejs`, `productForm.ejs`, etc.

""

Création du sous-répertoire `css` dans le répertoire `public` :

1. Ouvrez votre explorateur de fichiers et naviguez jusqu'au répertoire racine de votre projet.

2. Trouvez le répertoire nommé `public`.

3. Créez un nouveau dossier à l'intérieur de `public` en faisant un clic droit dans le répertoire et en sélectionnant "Nouveau dossier". Nommez ce dossier `css`.

Création des fichiers `.ejs` dans le répertoire `views` :

1. Ouvrez votre explorateur de fichiers et naviguez jusqu'au répertoire racine de votre projet.

2. Trouvez le répertoire nommé `views`.

3. Créez des nouveaux fichiers à l'intérieur de `views` en faisant un clic droit dans le répertoire et en sélectionnant "Nouveau fichier". Nommez ces fichiers `productList.ejs`, `productDetails.ejs` et `productForm.ejs`.

4. Ouvrez ces fichiers nouvellement créés avec votre éditeur de texte et commencez à écrire le code HTML et EJS pour vos différentes vues.

""

3. \*\*Configuration d'Express.js\*\* :

3.1 Créer un nouveau fichier app.js :

Dans votre éditeur de texte ou votre IDE, ouvrez le répertoire racine de votre projet.

Créez un nouveau fichier et nommez-le app.js.

3.2 Dans le fichier `app.js` que vous avez créé précédemment, configurez Express.js pour utiliser EJS comme moteur de template et pour servir les fichiers statiques depuis le répertoire `public`.

""

les étapes pour configurer Express.js dans fichier `app.js` :

1. \*\*Importez Express.js et EJS en utilisant `require`\*\* :

- Ajoutez les lignes suivantes au début de votre fichier `app.js` pour importer les modules Express.js et EJS :

```javascript

const express = require('express');

const app = express();

const ejs = require('ejs');

```

2. \*\*Configurez Express.js pour utiliser EJS comme moteur de template\*\* :

- Utilisez la méthode `app.set()` avec les paramètres `'view engine'` et `'ejs'` pour configurer Express.js pour utiliser EJS comme moteur de template. Ajoutez cette ligne quelque part dans votre fichier `app.js`, par exemple juste après les importations :

```javascript

app.set('view engine', 'ejs');

```

3. \*\*Configurez Express.js pour servir les fichiers statiques depuis le répertoire `public`\*\* :

- Utilisez la méthode `app.use()` avec la fonction `express.static()` pour servir les fichiers statiques (comme les fichiers CSS, JavaScript, images, etc.) depuis le répertoire `public`. Ajoutez cette ligne également dans votre fichier `app.js`, par exemple juste après la configuration du moteur de template :

```javascript

app.use(express.static('public'));

```

Une fois que vous avez ajouté ces lignes à votre fichier `app.js`, Express.js sera configuré pour utiliser EJS comme moteur de template et pour servir les fichiers statiques depuis le répertoire `public`. je peux maintenant continuer à développer votre application en ajoutant des routes, des contrôleurs, des modèles et des vues.

""

4. \*\*Création des modèles\*\* :

- Dans le répertoire `models`, créez un fichier `Product.js`.

- Dans ce fichier, définissez un modèle pour représenter les produits. Par exemple, vous pouvez utiliser une classe JavaScript pour cela:  
// Définition de la classe Product

class Product {

constructor(name, price, description, category, quantity, imageURL) {

this.name = name;

this.price = price;

this.description = description;

this.category = category;

this.quantity = quantity;

this.imageURL = imageURL;

}

// Méthode pour afficher les détails du produit

displayDetails() {

return `Nom: ${this.name}\nPrix: ${this.price} €\nDescription: ${this.description}\nCatégorie: ${this.category}\nQuantité: ${this.quantity}\nImage: ${this.imageURL}`;

}

}

module.exports = Product;

Vous pouvez utiliser cette classe Product de la même manière que l'objet const précédent. Par exemple, vous pouvez créer une nouvelle instance de produit en utilisant new Product() et accéder à ses propriétés et méthodes.

5. \*\*Création des contrôleurs\*\* :

5.1 Dans le répertoire `controllers`, créez un fichier `productController.js`.

5.2 Définition des fonctions de contrôleurs

exemple de définition des fonctions :  
// Importation du modèle de produit

const Product = require('../models/Product');

// Liste des produits (simulée pour cet exemple)

let products = [

new Product('Ordinateur portable', 1200, 'Un ordinateur portable puissant avec un processeur rapide.', 'Électronique', 10, 'laptop.jpg'),

new Product('T-shirt noir', 25, 'Un t-shirt noir de haute qualité.', 'Vêtements', 50, 'tshirt.jpg')

];

// Fonction de contrôleur pour afficher la liste des produits

const productList = (req, res) => {

res.render('productList', { products: products });

};

// Fonction de contrôleur pour afficher les détails d'un produit

const productDetails = (req, res) => {

const productId = req.params.id;

const product = products.find(product => product.id === productId);

if (!product) {

return res.status(404).send('Produit non trouvé');

}

res.render('productDetails', { product: product });

};

// Fonction de contrôleur pour ajouter un nouveau produit

const addProduct = (req, res) => {

const { name, price, description, category, quantity, imageURL } = req.body;

const newProduct = new Product(name, price, description, category, quantity, imageURL);

products.push(newProduct);

res.redirect('/products');

};

// Fonction de contrôleur pour supprimer un produit

const deleteProduct = (req, res) => {

const productId = req.params.id;

products = products.filter(product => product.id !== productId);

res.redirect('/products');

};

// Exportation des fonctions de contrôleur

module.exports = {

productList,

productDetails,

addProduct,

deleteProduct

};

-Dans cet exemple :

Nous importons le modèle Product depuis le fichier correspondant.

Nous simulons une liste de produits en utilisant un tableau products.

La fonction productList rend la vue productList en passant la liste des produits comme données.

La fonction productDetails récupère l'ID du produit à afficher à partir de la requête, trouve le produit correspondant dans la liste, puis rend la vue productDetails avec les détails du produit.

La fonction addProduct récupère les données du nouveau produit à partir du corps de la requête, crée un nouvel objet Product, l'ajoute à la liste des produits, puis redirige l'utilisateur vers la liste des produits.

La fonction deleteProduct récupère l'ID du produit à supprimer à partir de la requête, filtre la liste des produits pour supprimer le produit correspondant, puis redirige l'utilisateur vers la liste des produits.

6. \*\*Définition des routes\*\* :

- Dans `app.js`, importez le contrôleur de produit et configurez les routes Express pour chaque action.

-Par exemple, von peut definir une route pour afficher la liste des produits, une autre pour afficher les détails d'un produit, une pour ajouter un nouveau produit, etc.

-j'utilise les méthodes d'Express.js telles que app.get(), app.post(), app.put(), app.delete() pour définir ces routes.

-Connecter les routes aux contrôleurs :

Dans ce code, nous avons importé le contrôleur de produit que nous avons créé précédemment (productController) et connecté chaque route à la fonction de contrôleur correspondante.

Par exemple, la route '/products' avec la méthode GET est connectée à la fonction productList du contrôleur de produit. Cela signifie que lorsque quelqu'un accède à cette route via un navigateur, la fonction productList sera appelée pour gérer la demande et afficher la liste des produits.

De même, les autres routes sont connectées aux fonctions de contrôleurs appropriées pour gérer l'affichage des détails d'un produit, l'ajout d'un nouveau produit et la suppression d'un produit.

-

7. \*\*Création des vues\*\* :

- Dans les fichiers `.ejs` que vous avez créés précédemment dans le répertoire `views`, utilisez la syntaxe EJS pour générer dynamiquement le HTML en fonction des données fournies par le contrôleur.

8. \*\*Validation des données côté serveur\*\* :

- Ajoutez des mécanismes de validation des données dans votre application Express.js pour garantir que seules les données valides sont acceptées lors de l'ajout ou de la modification de produits.