



BO - HUB

B-INN-000

Workshop Backend

Création d'une API pour un site de e-commerce





Workshop Backend

language: NodeJS, Typescript
build tool: npm start

Sujet de Workshop créé par Ludovic Sutter.

MISE EN PLACE

INSTALLATION DES DÉPENDANCES

Allez dans le dossier `front` puis entrer la commande `npm install` dans votre terminal.

Une fois cela effectué, vous pouvez vous rendre laisser un terminal ouvert dans le dossier `front` pour pouvoir le lancer plus tard et aller avec un autre terminal dans le dossier `back`.

Une fois dans le dossier `back`, vous pouvez entrer la commande `npm install` dans votre terminal comme pour le `front`.

MISE EN PLACE DE LA BASE DE DONNÉE



Si vous avez mariaDB: Si vous avez déjà mariaDB installé sur votre PC, les étapes suivantes ne sont pas obligatoires.

Créez simple une nouvelle base de données et changer la valeur `DATABASE_URL` dans le `.env` en suivant le format `mysql://user:password@localhost:3306/mydb`

Renommez le fichier `.env.sample` en `.env` ensuite entrez dans le terminal la commande `docker-compose up --build` et attendez qu'il n'y ait plus de nouveaux messages qui s'affichent.

Une fois cela fait, vous pouvez aller dans un autre terminal et entrer la commande `npm run db:migrate` et faire entrer à toutes les questions.

Parfait, maintenant votre projet est initialisé, pour le lancer vous pouvez faire la commande `npm start` et de même pour le `front`.



DOCUMENTATION

LIENS UTILES

Fastify: lien vers la documentation [ici](#)

Prisma: lien vers la documentation [ici](#)

HttpStatus: lien vers l'explication des codes de réponse HTTP [ici](#)

HttpRequests: lien vers l'explication des méthodes de requête HTTP [ici](#)

CODE CHEAT SHEET

```
// exemple de route avec fastify
instance.get("/", async (request: FastifyRequest, res: FastifyReply) => {
  res.status(HttpStatus.OK).send("Hello World!");
});
```

Explications:

- get représente la méthode de requête (ici get car on veut récupérer des données)
- "/" représente la route (url) sur laquelle on veut récupérer ces données
- FastifyRequest et FastifyReply sont les types de la requête et la réponse par défaut, on verra plus tard comment changer le type de la requête.
- la méthode status permet de définir le status de la réponse (ici OK car tout s'est bien passé)
- pour finir la méthode send définit ce que la route va renvoyer

EXERCICES

EXERCICES 0: HELLO WORLD !

Dans le fichier `src/routes/base.routes.ts`, créez une route qui renvoie la string **Hello World** !

Pour tester si votre code fonctionne, vous pouvez faire la commande `npm start` puis aller dans votre navigateur à l'url afficher dans votre terminal. Si le message **Hello World** ! s'affiche vous avez réussi !

EXERCICE 1: REGISTER



A partir de cet exercice, vous allez interagir avec la base de données. Pour **visualiser** ce qui se passe dans votre base de données en temps réel vous pouvez entrer la commande `npm run db:show` dans votre terminal.

Dans le fichier `src/routes/user.routes.ts`, il vous faut une route qui a pour chemin `/users/` qui enregistre un nouvel utilisateur dans la base de données, crypte son mot de passe et crée un token contenant l'adresse email et l'id de l'utilisateur.



Architecture: En règle général, il faut éviter de mettre beaucoup de logique et d'action sur la base de données directement dans la route.

Services: Pour éviter cela, on crée des services (fonction qui encapsule la logique) dans le dossier `src/services/`

Il vous faut donc pour faire fonctionner cette route modifier la fonction `create` dans le fichier `src/services/user.services.ts` pour implémenter les fonctionnalités citées ci-dessus.

Si vous avez bien fait la chose, quand vous lancez le front vous allez pouvoir vous enregistrer et un message vert apparaîtra en haut à droite de votre écran.



EXERCICE 2: LOGIN

Cette partie va être assez similaire à la précédente.

Vous allez devoir créer une route qui a pour chemin `/users/login`, cette fois ci vous allez devoir créer le type de la requête vous-même en vous inspirant de l'exercice 1.

Vous allez recevoir dans le body un **email** en string ainsi **password** en string également.

Il vous faudra ensuite créer un service qui gère la connexion d'un utilisateur déjà existant et renvoyait un token comme pour l'exercice 1.

Pour vérifier votre code est fonctionnel, vous pouvez essayer de vous connecter à un compte déjà créé sur le front.

EXERCICE 3: ME

Pour que le front puisse récupérer des informations de l'utilisateur, il faut une route qui renvoie les informations d'un utilisateur connecté.

Pour récupérer les informations de notre utilisateur connecter nous allons utiliser un middleware.



Ici le terme **middleware** symbolise une fonction qui est exécutée avant de traiter la requête.

Dans notre cas, le **middleware** a pour but de récupérer le token et de la déchiffrer pour obtenir l'id et l'email de l'utilisateur et de sécuriser la route.

Une fois les informations de l'utilisateur récupéré, il va vous falloir que vous fassiez un service qui récupère les informations de cet utilisateur et les renvoie ensuite dans la route `/users/me`.



La totalité des routes suivantes utiliseront le middleware d'authentification pour assurer la sécurité de l'application.



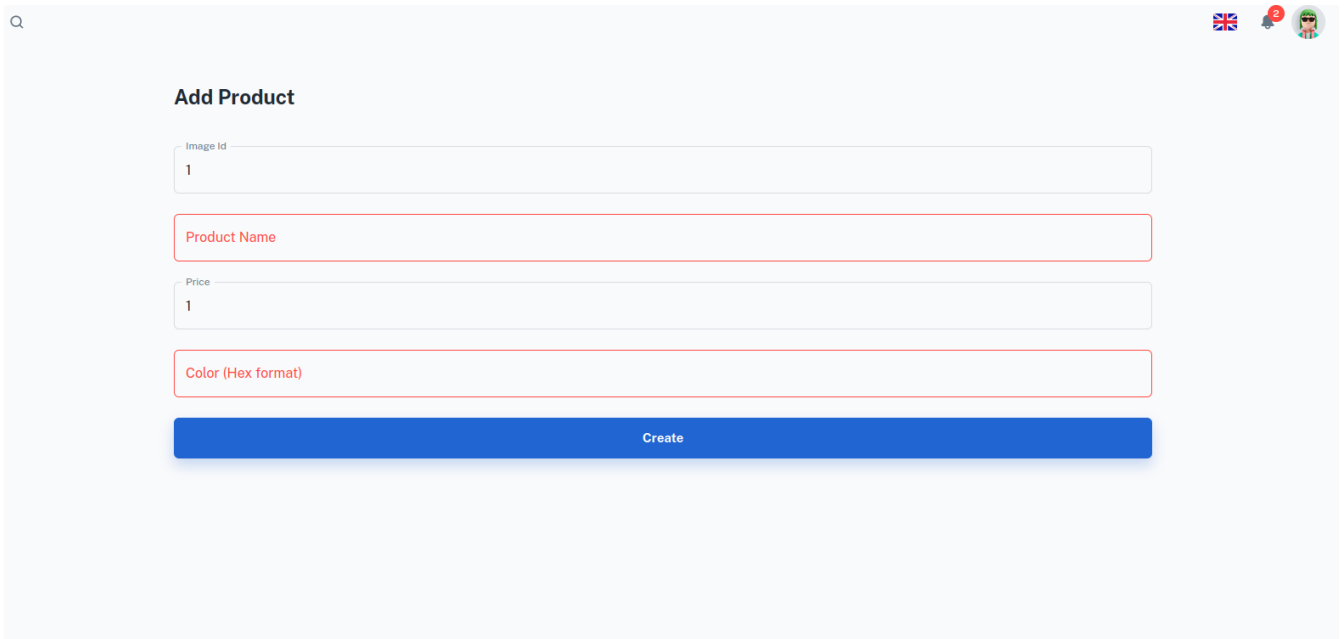
EXERCICE 4: CREATE PRODUCT

Pour faire fonctionner la page de création de produit, vous allez devoir faire une route POST `/products` dans le fichier `src/routes/product.routes.ts`.

La route recevra dans le body de la requête :

- un **name** en string
- un **imageId** en number
- un **price** en number
- une **color** en string

Il faut ensuite créer un service dans `src/services/product.services.ts` qui créera le produit et renverra ensuite le produit qui vient d'être créé.

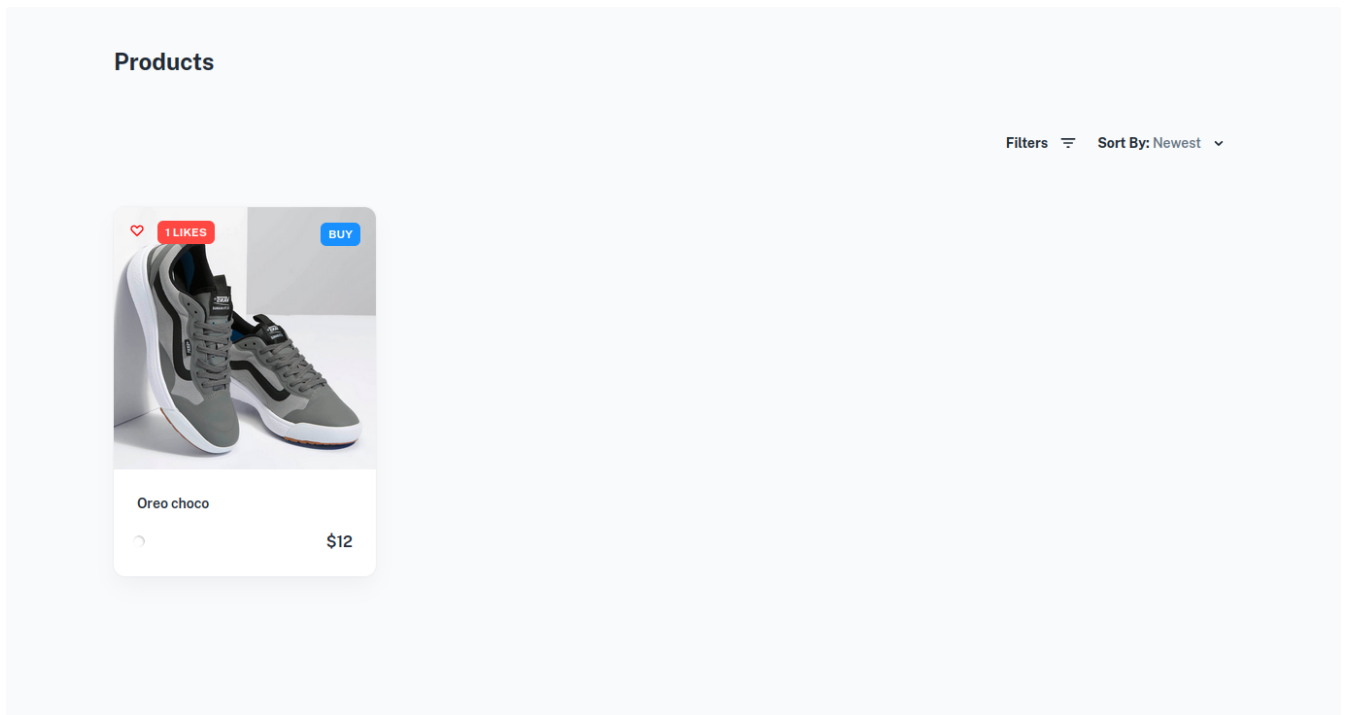


The screenshot shows a web application interface for adding a product. At the top right, there are icons for a search bar, a flag (UK), a notification bell, and a user profile. The main heading is "Add Product". Below it, there are four input fields: "Image Id" with the value "1", "Product Name" (empty), "Price" with the value "1", and "Color (Hex format)" (empty). Each input field has a red border indicating it is required. At the bottom, there is a blue button labeled "Create".

EXERCICE 5: GET PRODUCTS

Pour faire fonctionner la page **products** du front, vous allez devoir faire une route GET `/products` dans le fichier `src/routes/product.routes.ts`.

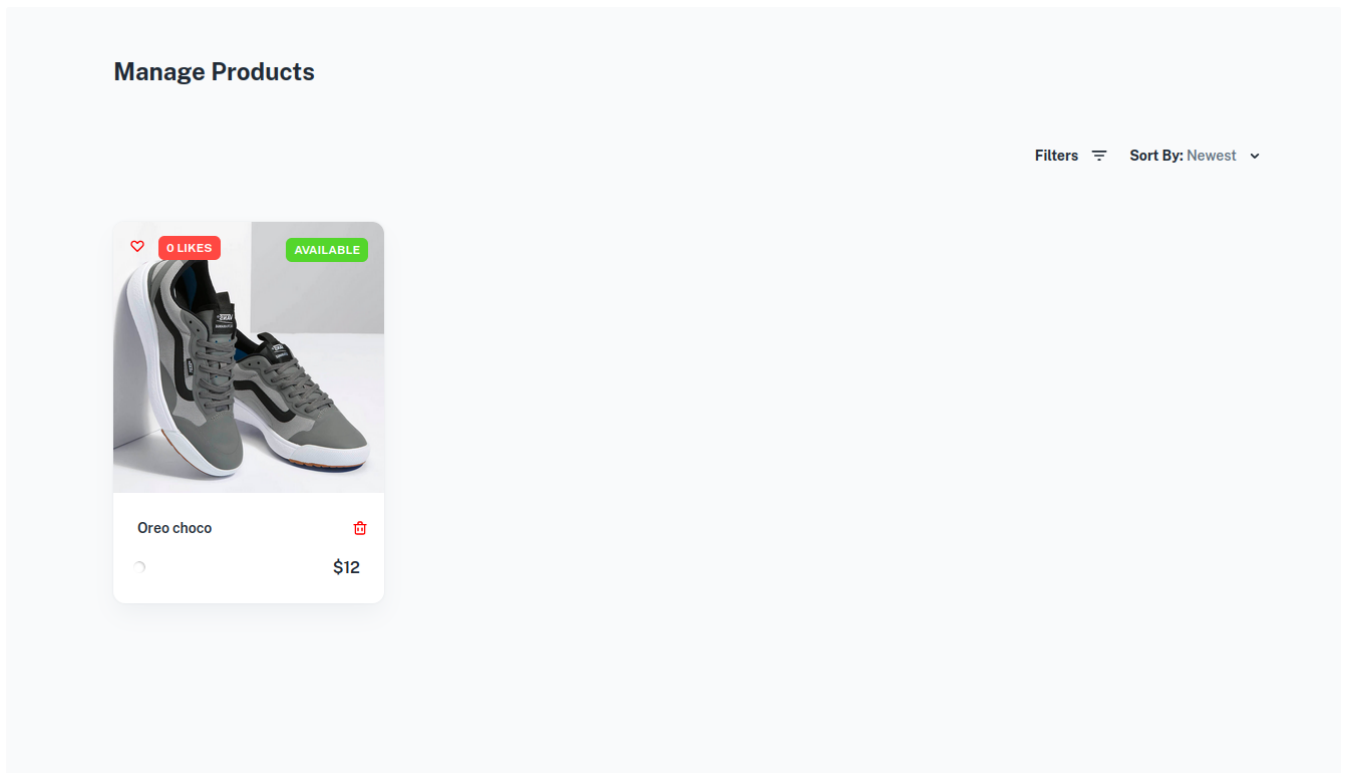
Il faut ensuite créer un service dans `src/services/product.services.ts` qui récupèrera tous les produits et les renverra ensuite dans la route `/products`.



EXERCICE 6: GET USER PRODUCTS

Pour faire fonctionner la page **manage products** du front, vous allez devoir faire une route GET `/users/products` dans le fichier `src/routes/user.routes.ts`

Il faut ensuite créer un service dans `src/services/product.services.ts` qui récupèrera tous les produits de l'utilisateur connecté et les renverra ensuite dans la route `/users/products`.



EXERCICE 7: DELETE PRODUCT

Pour faire fonctionner la suppression sur la page **manage products** du front, vous allez devoir faire une route DELETE `/products/:id` dans le fichier `src/routes/product.routes.ts`.

Vous allez donc recevoir dans les params un **id** en string.

Il faut ensuite créé un service dans `src/services/product.services.ts` qui supprimera le produit avec l'id reçu et renverra ensuite le produit qui vient d'être supprimé dans la route `/products/:id`.



Attention: Vous devez vérifier que le produit appartient bien à l'utilisateur connecté avant de le supprimer.

EXERCICE 8: LIKE PRODUCT

Pour faire fonctionner le like sur la page **products** du front, vous allez devoir faire une route PUT `/products/:id/like` dans le fichier `src/routes/product.routes.ts`.

Vous allez donc recevoir dans les params un **id** en string.

Il faut ensuite créé un service dans `src/services/product.services.ts` qui va liker le produit avec l'id reçu et renverra ensuite le produit qui vient d'être liké dans la route `/products/:id/like`.



Attention: Vous devez utiliser la table **likedProduct** car il est important de savoir quels produits un utilisateur a déjà liké.

EXERCICE 9: DISLIKE PRODUCT

Pour faire fonctionner le dislike sur la page **products** du front, vous allez devoir faire une route PUT `/products/:id/dislike` dans le fichier `src/routes/product.routes.ts`.

Vous allez donc recevoir dans les params un **id** en string.

Il faut ensuite créé un service dans `src/services/product.services.ts` qui va disliker le produit avec l'id reçu et renverra ensuite le produit qui vient d'être disliké dans la route `/products/:id/dislike`.



EXERCICE 10: GET LIKED PRODUCTS

Pour faire fonctionner la sauvegarde des likes sur page **products** du front, vous allez devoir faire une route GET `/users/products/liked` dans le fichier `src/routes/user.routes.ts`.

Il faut ensuite créé un service dans `src/services/product.services.ts` qui va récupérer tous les produits likés par l'utilisateur connecté et les renverra ensuite dans la route `/users/products/liked`.

EXERCICE 11: GET SALES

Pour faire fonctionner la page **dashboard** du front, vous allez devoir faire une route GET `/users/sales` dans le fichier `src/routes/user.routes.ts`.

Il faut ensuite créé un service dans `src/services/sale.services.ts` qui va récupérer tous les produits vendus par l'utilisateur connecté et les renverra ensuite dans la route `/users/sales`.

EXERCICE 12: CREATE SALE

Pour faire fonctionner la page **dashboard** du front, vous allez devoir faire une route POST `/sales` dans le fichier `src/routes/sale.routes.ts`.

La route recevra dans le body de la requête :

- un **userId** en number
- un **name** en string
- un **amount** en number

Il faut ensuite créé un service dans `src/services/sale.services.ts` qui va créer la vente et renverra ensuite la vente qui vient d'être créée dans la route `/sales`.

EXERCICE 13: BUY PRODUCT

Pour faire fonctionner l'achat de produit sur la page **products** du front, vous allez devoir faire une route PUT `/products/:id/buy` dans le fichier `src/routes/product.routes.ts`.

Vous allez donc recevoir dans les params un **id** en string.

Il faut ensuite créer un service dans `src/services/product.services.ts` qui va acheter le produit avec l'id reçu et renverra ensuite le produit qui vient d'être acheté dans la route `/products/:id/buy`.



Attention: Vous ne devez pas supprimer le produit, mais simplement mettre à jour son statut en **SOLD**.

EXERCICE FINAL: OAUTH

Pour faire fonctionner l'authentification avec Google sur la page **login** du front, vous allez devoir faire une route GET `/auth/google/link` ainsi que post `/auth/google/register` dans le fichier `src/routes/auth.routes.ts`.

La route `/auth/google/register` recevra dans le body de la requête un code en string.

Il faut ensuite créer un service dans `src/services/auth.services.ts` qui vont permettre de récupérer les informations de l'utilisateur Google et de créer un utilisateur dans la base de données ou de la connecter.

CONCLUSION

Vous avez maintenant terminé le projet **E-commerce**.

Vous avez pu voir comment créer une API REST avec Node.js, Fastify et Prisma.

Bien joué !

