



MINISTÈRE CHARGÉ
DE L'EMPLOI

DOSSIER

PROFESSIONNEL (DP)



Nom de naissance

➤ NGO

Nom d'usage

➤

Prénom

➤ Quoc Nam

Adresse

➤ 71 impasse Biskra 83200 Toulon

Titre professionnel visé

Développeur web et web mobile

MODALITÉ D'ACCÈS :

- Parcours de formation
- Validation des Acquis de l'Expérience (VAE)

DOSSIER PROFESSIONNEL (DP)

Présentation du dossier

Le dossier professionnel (DP) constitue un élément du système de validation du titre professionnel.

Ce titre est délivré par le Ministère chargé de l'emploi.

Le DP appartient au candidat. Il le conserve, l'actualise durant son parcours et le présente **obligatoirement à chaque session d'examen.**

Pour rédiger le DP, le candidat peut être aidé par un formateur ou par un accompagnateur VAE.

Il est consulté par le jury au moment de la session d'examen.

Pour prendre sa décision, le jury dispose :

- 1.** des résultats de la mise en situation professionnelle complétés, éventuellement, du questionnaire professionnel ou de l'entretien professionnel ou de l'entretien technique ou du questionnement à partir de productions.
- 2.** du **Dossier Professionnel** (DP) dans lequel le candidat a consigné les preuves de sa pratique professionnelle.
- 3.** des résultats des évaluations passées en cours de formation lorsque le candidat évalué est issu d'un parcours de formation
- 4.** de l'entretien final (dans le cadre de la session titre).

[Arrêté du 22 décembre 2015, relatif aux conditions de délivrance des titres professionnels du ministère chargé de l'Emploi]

Ce dossier comporte :

- pour chaque activité-type du titre visé, un à trois exemples de pratique professionnelle ;
- un tableau à renseigner si le candidat souhaite porter à la connaissance du jury la détention d'un titre, d'un diplôme, d'un certificat de qualification professionnelle (CQP) ou des attestations de formation ;
- une déclaration sur l'honneur à compléter et à signer ;
- des documents illustrant la pratique professionnelle du candidat (facultatif)
- des annexes, si nécessaire.

DOSSIER PROFESSIONNEL (DP)

Pour compléter ce dossier, le candidat dispose d'un site web en accès libre sur le site.



<http://travail-emploi.gouv.fr/titres-professionnels>

Sommaire

Activité-type 1 - Développer la partie front-end d'une application web ou web mobile sécurisée	p .	5
• CP 1 - Installer et configurer son environnement de travail en fonction du web ou web mobile	p.	5
• CP 2 - Maquetter des interfaces utilisateur web ou web mobile	p.	13
• CP 3 - Réaliser des interfaces statiques web ou web mobile	p.	18
• CP 4 - Développer la partie dynamique des interfaces utilisateur web ou web mobile	p.	25
Activité-type 2 - Développer la partie back-end d'une application web ou web mobile sécurisée	p .	33
• CP 5 - Mettre en place une base de donnée relationnelle	p.	33
• CP 6 - Développer les composants d'accès aux données SQL et NoSQL	p.	43
• CP 7 - Développer des composants métier côté serveur	p.	54
• CP 8 - Documenter le déploiement d'une application dynamique	p.	65
Titres, diplômes, CQP, attestations de formation (facultatif)	p .	72
Déclaration sur l'honneur	p .	73
Documents illustrant la pratique professionnelle (facultatif)	p .	74
Annexes (Si le RC le prévoit)	p .	75

DOSSIER PROFESSIONNEL (DP)

EXEMPLES DE PRATIQUE PROFESSIONNELLE

Activité-type 1

Développer la partie front-end d'une application web ou web mobile sécurisée

CP 1 ▶ Installer et configurer son environnement de travail en fonction du web ou web mobile

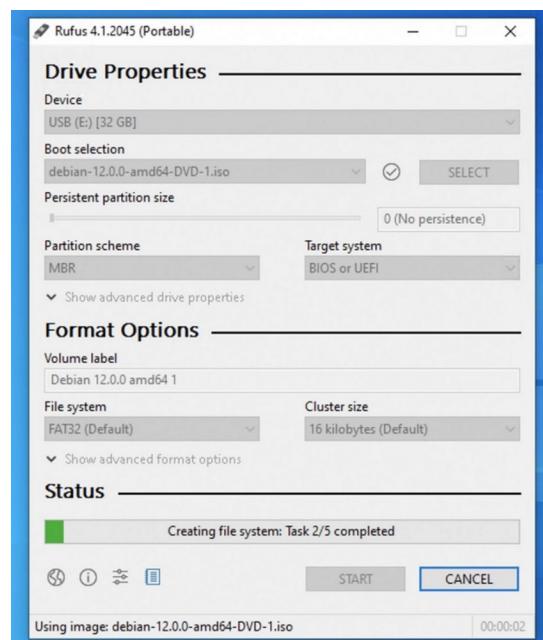
1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

Dans le cadre du développement d'une application web, j'ai installé et configuré un environnement de travail complet sur un poste sous Linux Debian. L'objectif était de disposer d'un environnement stable, sécurisé et adapté au développement web moderne, en particulier pour apprendre à travailler en environnement open source, maîtriser la ligne de commande, et suivre les bonnes pratiques en matière de gestion de versions, modularité, et conteneurisation.

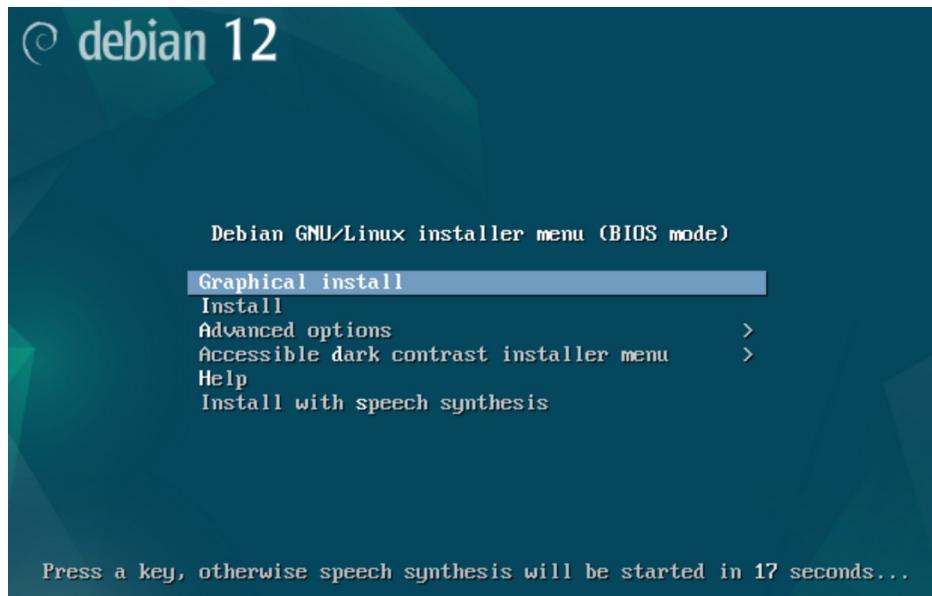
Installation de Debian

J'ai choisi Debian comme système d'exploitation principal pour sa stabilité, sa légèreté et son orientation vers le libre. Il s'agit d'une distribution largement utilisée dans les environnements serveurs, ce qui la rend idéale pour développer des applications web robustes et compatibles avec des infrastructures de production. De plus, Debian offre un très bon support communautaire et une documentation riche, ce qui est précieux pour la montée en compétence.

J'ai téléchargé l'image ISO de Debian depuis le site officiel debian.org, puis créé une clé USB bootable avec Rufus (sous Windows 11).



Après avoir sauvegardé mes données importantes, j'ai redémarré mon PC et accédé au BIOS/UEFI (en appuyant sur F2) pour modifier l'ordre de démarrage et lancer l'installation depuis la clé. J'ai choisi l'installation graphique pour faciliter la configuration initiale.



Pendant l'installation, j'ai défini la langue, le pays, le type de clavier et configuré la connexion Wi-Fi. J'ai défini un mot de passe root, puis créé mon utilisateur principal. Le disque a été partitionné manuellement pour mieux contrôler la structure du système. Une fois le système installé, j'ai accepté l'installation du chargeur de démarrage (GRUB) pour permettre un double démarrage avec Windows. Mon PC a redémarré directement sur Debian. J'ai immédiatement mis à jour le système avec `sudo apt update && sudo apt upgrade` afin de disposer des dernières versions des paquets et j'ai installé les pilotes audio manquants pour assurer le bon fonctionnement matériel.

Utilisation et configuration du terminal Bash

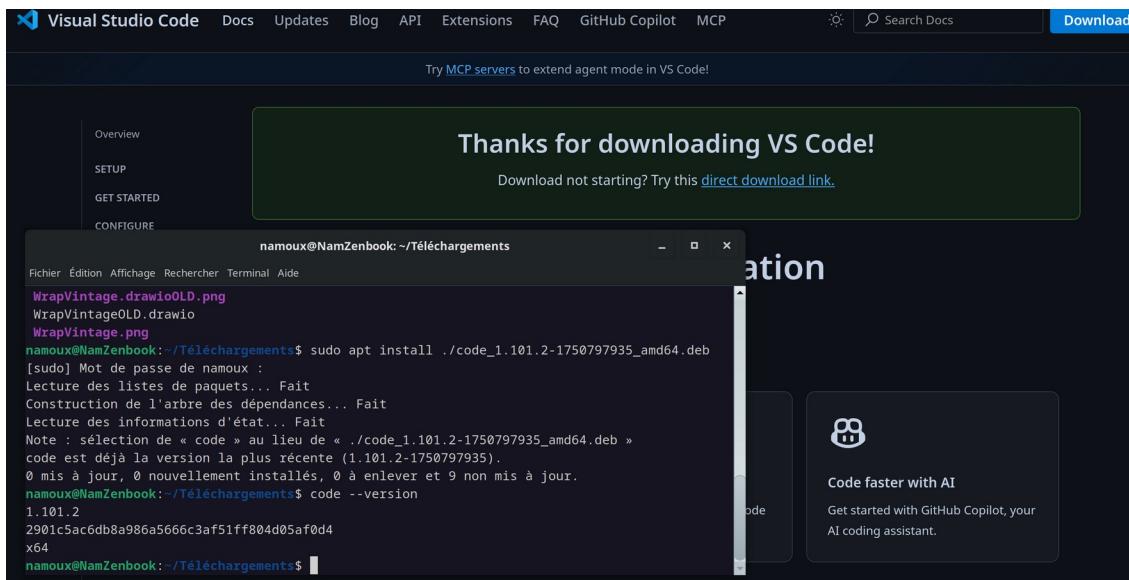
Le terminal Bash est un composant essentiel sous Debian pour piloter l'ensemble du système. Je l'ai utilisé tout au long de la mise en place de l'environnement : installation de paquets, exécution de commandes système, gestion des fichiers, configuration de Git, lancement de serveurs ou de conteneurs Docker, etc.

L'utilisation du terminal permet d'avoir un contrôle fin et rapide sur l'environnement, ce qui est indispensable en développement back-end, notamment sur les systèmes Linux.

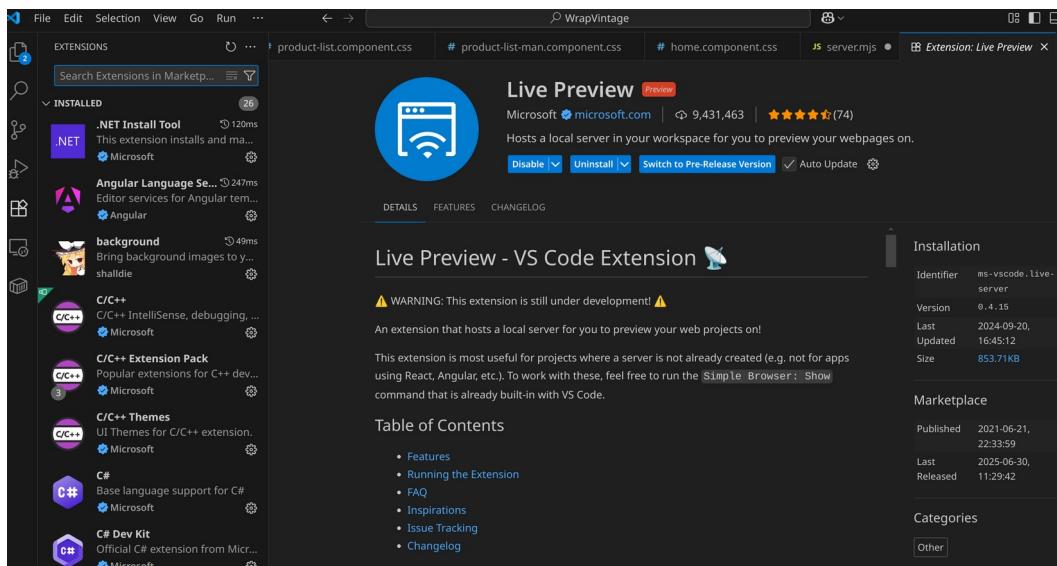
Installation de Visual Studio Code

J'ai installé Visual Studio Code car c'est un éditeur léger mais puissant, particulièrement bien adapté au développement JavaScript et Node.js. Il dispose de nombreuses extensions utiles, comme le formatage automatique du code, la coloration syntaxique, l'intégration Git ou encore le terminal intégré, ce qui améliore nettement la productivité.

J'ai téléchargé le fichier .deb sur le site officiel, puis je l'ai installé via la commande sudo apt install ./code_1.101.2-1750797935_amd64.deb.



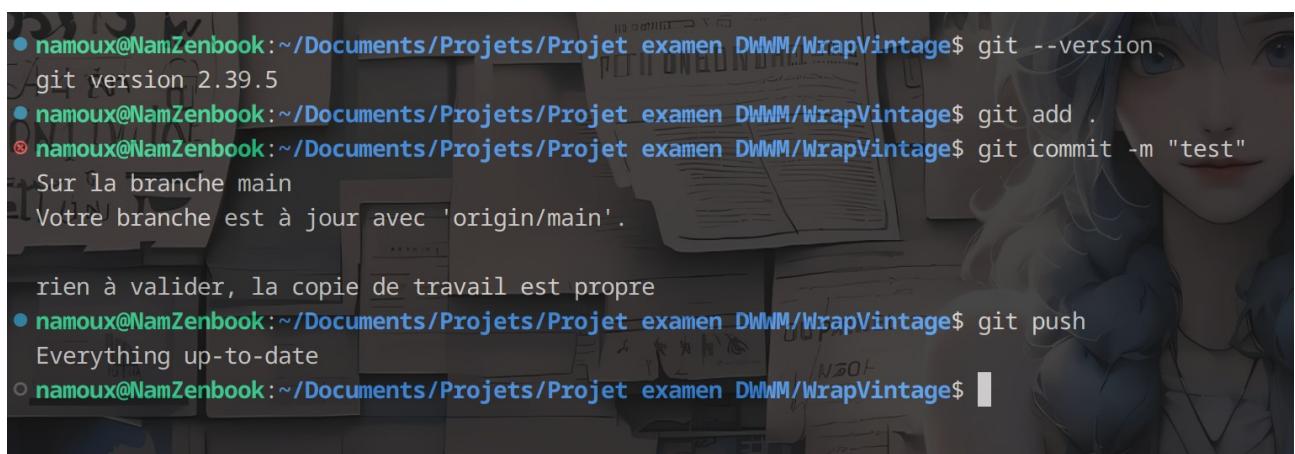
Les extensions nécessaires, comme livePreview, Angular language ou Node.js Snippets, ont été installées directement depuis le Marketplace intégré à VS Code



Mise en place de Git et GitHub

Git est un outil essentiel pour le développement collaboratif. Il permet de versionner son code, de collaborer à plusieurs, de créer des branches pour tester des fonctionnalités sans risque, et de documenter l'évolution du projet. J'ai choisi GitHub comme plateforme distante pour sa compatibilité avec Git, son interface simple et sa popularité dans l'écosystème open source.

Je me suis inscrit sur GitHub, puis j'ai créé un dépôt distant. J'ai installé Git avec sudo apt install git, vérifié la version, puis configuré mon nom d'utilisateur et mon e-mail. J'ai cloné le dépôt avec git clone, ouvert le projet avec code ., puis utilisé les commandes classiques (git pull, git add, git commit, git push) pour synchroniser mon code. En cas de besoin, je créais des branches avec git checkout -b pour développer de nouvelles fonctionnalités ou corriger des bugs sans impacter le code principal.



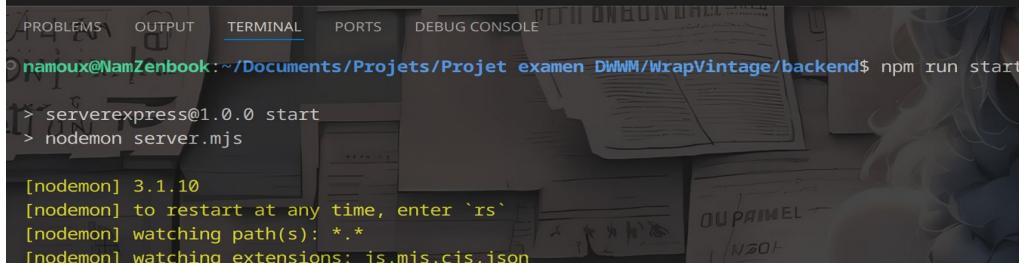
```
● namoux@NamZenbook:~/Documents/Projets/Projet examen DWMM/WrapVintage$ git --version
git version 2.39.5
● namoux@NamZenbook:~/Documents/Projets/Projet examen DWMM/WrapVintage$ git add .
○ namoux@NamZenbook:~/Documents/Projets/Projet examen DWMM/WrapVintage$ git commit -m "test"
Sur la branche main
Votre branche est à jour avec 'origin/main'.
rien à valider, la copie de travail est propre
● namoux@NamZenbook:~/Documents/Projets/Projet examen DWMM/WrapVintage$ git push
Everything up-to-date
○ namoux@NamZenbook:~/Documents/Projets/Projet examen DWMM/WrapVintage$
```

Installation de Node.js et initialisation du projet

J'ai installé Node.js avec npm car il s'agit de la plateforme la plus utilisée pour développer des serveurs web modernes avec JavaScript côté serveur. C'est aussi l'environnement natif d'Express.js, que j'allais utiliser pour le back-end de l'application. Npm, le gestionnaire de paquets associé, permet de gérer facilement les dépendances du projet.

J'ai installé Node.js et npm via la commande sudo apt install nodejs npm. J'ai ensuite initialisé mon projet avec npm init -y, ce qui a généré un fichier package.json. J'y ai installé Express avec npm

install express et configuré un script de démarrage. Pour faciliter le développement, j'ai utilisé nodemon app.js, qui relance automatiquement le serveur après chaque modification de code.



```
backend > {} package.json > ...
1  {
2    "dependencies": {
3      "bcrypt": "^6.0.0",
4      "cors": "^2.8.5",
5      "dotenv": "^16.5.0",
6      "express": "^5.1.0",
7      "jsonwebtoken": "^9.0.2",
8      "mariadb": "^3.4.2",
9      "mysql2": "^3.14.1",
10     "sequelize": "^6.37.7"
11   },
12   "name": "serverexpress",
13   "version": "1.0.0",
14   "description": "Full Shop en express",
15   "main": "server.mjs",
16   "type": "module",
17   >Debug
18   "scripts": {
19     "test": "echo \\\"Error: no test specified\\\" && exit 1",
20     "start": "nodemon server.mjs"
21   },
22   "keywords": [],
23   "author": "",
24   "license": "ISC"
25 }
```

PROBLEMS OUTPUT TERMINAL PORTS DEBUG CONSOLE

```
namoux@NamZenbook:~/Documents/Projets/Projet examen DMM/WrapVintage/backend$ npm run start

> serverexpress@1.0.0 start
> nodemon server.mjs

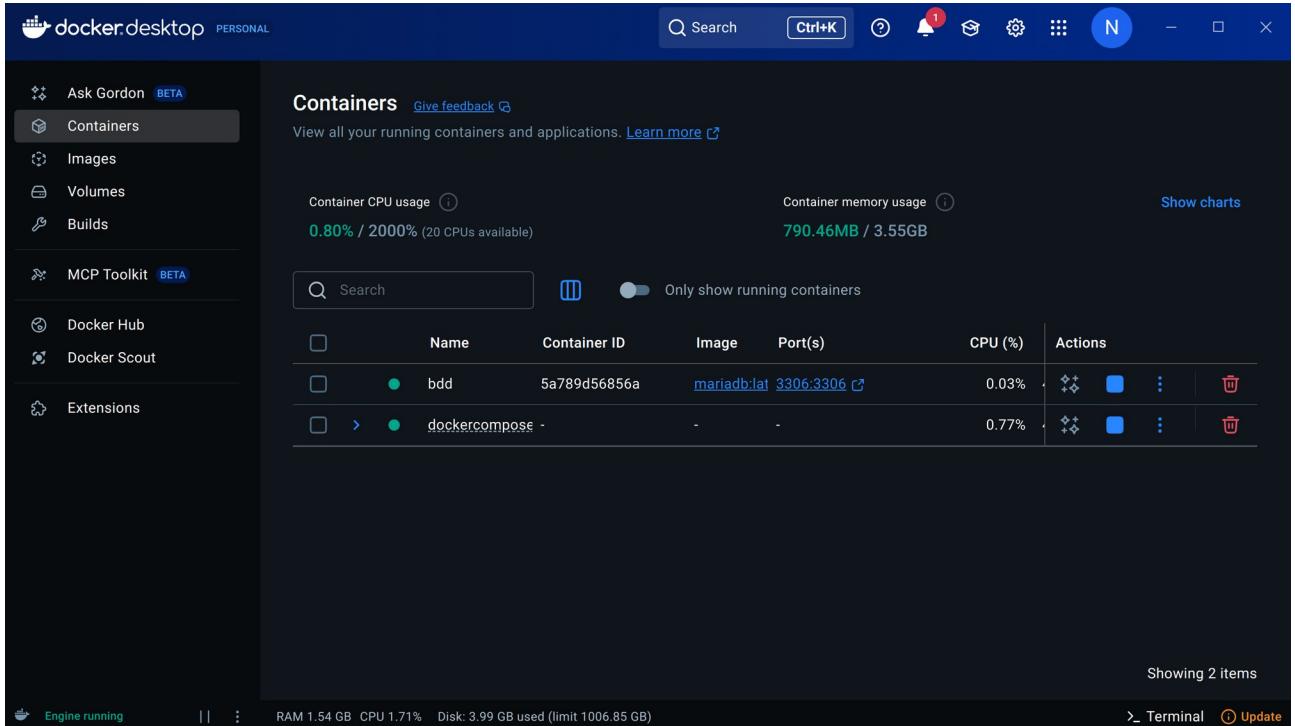
[nodemon] 3.1.10
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
```

Installation de Docker

J'ai installé Docker pour apprendre à créer des environnements isolés et reproductibles. Docker permet d'exécuter des applications (comme une base de données ou un serveur web) dans des conteneurs, ce qui évite les problèmes de compatibilité entre environnements. Il est largement utilisé en production pour son efficacité et sa portabilité.

J'ai installé Docker avec sudo apt install docker.io, démarré le service avec sudo systemctl start docker, et activé son lancement automatique avec sudo systemctl enable docker. Pour ne plus avoir à utiliser sudo à chaque commande, j'ai ajouté mon utilisateur au groupe docker avec sudo usermod -aG docker \$USER. J'ai validé l'installation avec docker run hello-world.

J'ai également téléchargé Docker Desktop depuis le site officiel pour bénéficier d'une interface graphique. Cela m'a permis de visualiser mes conteneurs, images et volumes, et de gérer plus facilement mes projets Docker, notamment lors de la mise en place d'une base de données locale (comme MariaDB) ou du serveur Express.



Installation de navigateurs pour les tests

Pour m'assurer que l'application web fonctionne correctement quel que soit le navigateur utilisé par les utilisateurs finaux, j'ai installé à la fois Google Chrome et Mozilla Firefox. Cela permet de réaliser des tests multi-navigateurs afin de vérifier la compatibilité du code (HTML, CSS, JS) avec différents moteurs de rendu.

Ces deux navigateurs sont les plus répandus, chacun ayant ses propres particularités d'interprétation du code. Les tests croisés permettent donc d'anticiper et de corriger d'éventuels problèmes d'affichage ou de comportement.

Contraintes techniques rencontrées

Le travail s'est déroulé exclusivement sous Linux, sans recours à Windows ni à WSL. J'ai dû faire face à quelques problèmes de droits d'accès lors de l'installation de certains paquets. Il a aussi fallu m'assurer que tous les outils étaient compatibles avec l'environnement Debian, et que le navigateur utilisé supportait les fonctionnalités modernes pour les tests front-end.

Objectif et résultat

L'objectif principal était de mettre en place un environnement de développement sécurisé, modulaire, et complet pour coder dans de bonnes conditions. Tous les outils installés permettent désormais de développer efficacement, de tester l'application dans un environnement local cohérent avec les standards actuels, et de travailler en équipe avec des outils professionnels.

Livrables

J'ai produit une documentation d'installation détaillée, ainsi que des captures d'écran et notes de configuration retracant toutes les étapes mises en œuvre.

2. Précisez les moyens utilisés :

Matériel :

- PC portable sous Debian Linux
- Processeur i9, 16 Go RAM, SSD 1To

Logiciels :

- Debian 12
- Visual Studio Code
- Git
- Terminal Bash
- Docker + docker-compose
- Navigateur Google Chrome / Firefox

Méthodes :

- Configuration manuelle de l'environnement
- Documentation continue pour assurer la reproductibilité

3. Avec qui avez-vous travaillé ?

J'ai travaillé seul sur ce projet.

4. Contexte

Nom de l'entreprise, organisme ou association ► La plateforme

Chantier, atelier, service ►

Période d'exercice ► *Du : 09/09/2024 au : 11/10/2024*

5. Informations complémentaires (facultatif)

Activité-type 1

Développer la partie front-end d'une application web ou web mobile sécurisée

CP 2 ► Maquetter des interfaces utilisateur web ou web mobile

1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

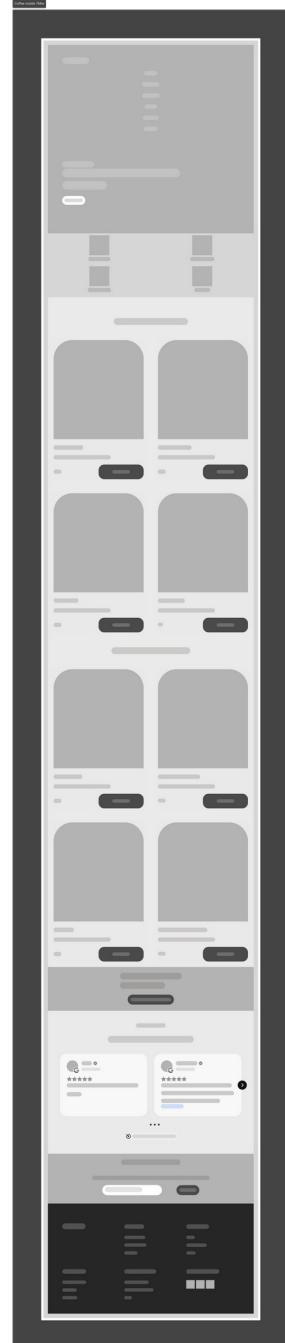
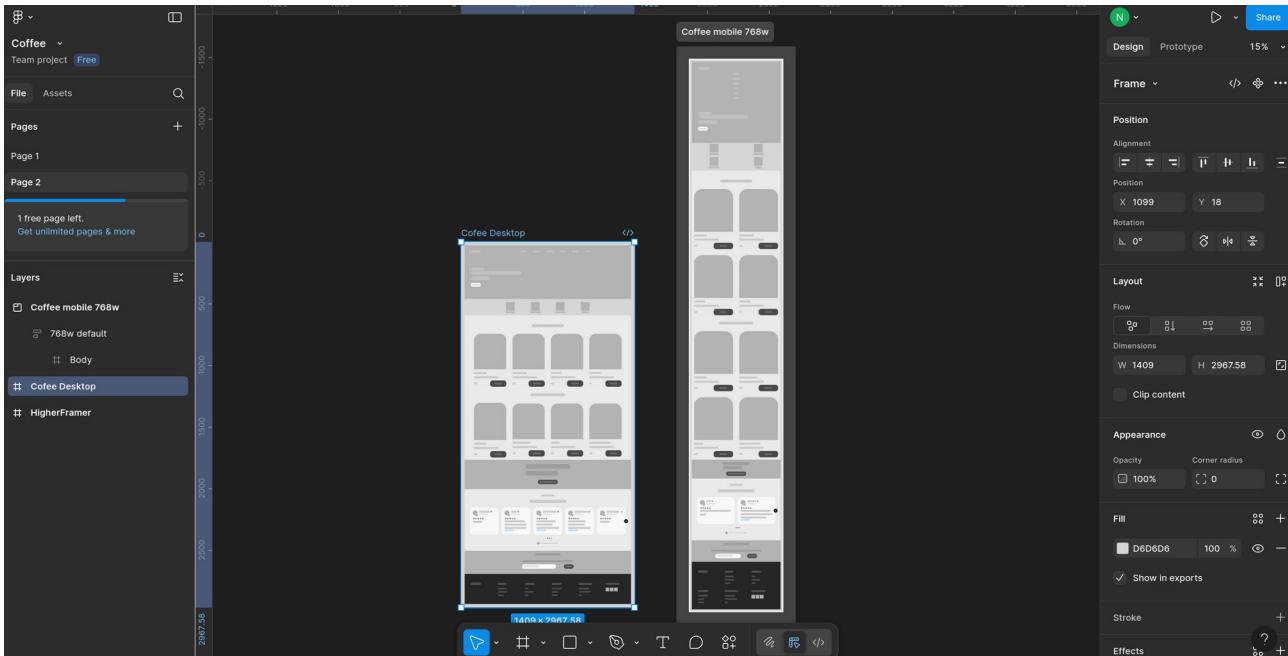
Dans le cadre de mon projet web, j'ai conçu des maquettes d'interfaces utilisateur avec Figma, en combinant une partie de design manuelle et une génération automatique à partir de structures HTML existantes via HTML.to.Design. Cette approche m'a permis de gagner un temps précieux tout en assurant la cohérence entre la structure HTML et le design visuel. L'objectif était de créer rapidement des interfaces claires, fonctionnelles et directement exploitables pour le développement.

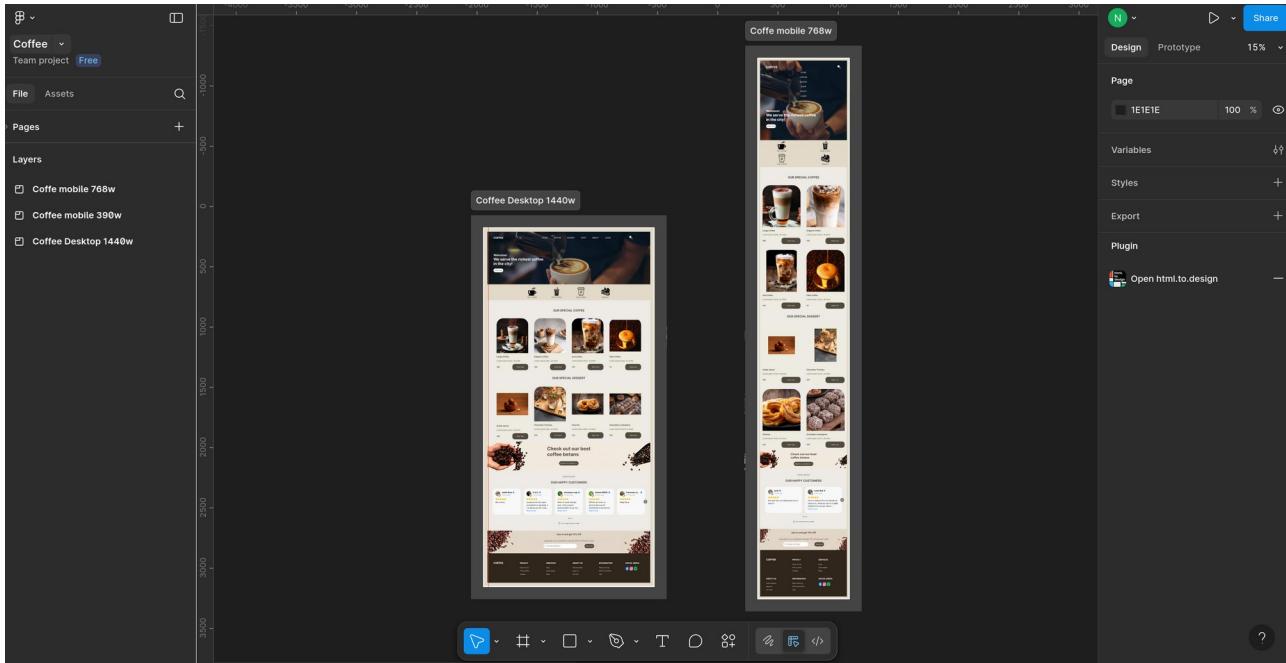
J'ai commencé par une analyse des besoins fonctionnels afin d'identifier les pages clés à maquetter, comme la page d'accueil, les formulaires ou encore le tableau de bord. À partir de là, j'ai utilisé HTML.to.Design pour convertir du code HTML en wireframes dynamiques, ce qui m'a permis d'avoir une base structurée sur laquelle m'appuyer. Ces wireframes ont ensuite été retravaillés et personnalisés dans Figma, en y intégrant l'identité visuelle du projet : typographie, couleurs, icônes, et autres éléments graphiques.

Afin d'assurer la cohérence visuelle du projet, j'ai mis en place un design system de base avec des composants réutilisables tels que des boutons, des cartes ou encore des champs de formulaire. Ce système m'a permis de concevoir des maquettes uniformes et facilement adaptables. J'ai également pris en compte le responsive design, en déclinant chaque maquette pour mobile, tablette et bureau, afin de garantir une bonne expérience utilisateur sur tous les supports.

Une fois les maquettes terminées, j'ai créé des prototypes interactifs dans Figma pour simuler les parcours utilisateurs, ce qui a facilité la validation des choix d'ergonomie avec les encadrants ou les utilisateurs tests. Après plusieurs retours, j'ai effectué des itérations ciblées pour améliorer la lisibilité, le contraste et la fluidité de navigation.

Enfin, pour illustrer concrètement mon travail, je présente ici deux versions d'un même écran : une wireframe générée automatiquement et une maquette finalisée sur le thème d'un site Coffee. Cette comparaison permet de visualiser la transformation d'une structure brute vers une interface finie, prête à être développée.





Objectif

Créer des maquettes efficaces, cohérentes et rapidement exploitables pour le développement web. Accélérer la production d'interfaces tout en conservant une cohérence graphique, et faciliter les échanges avec les parties prenantes grâce aux prototypes interactifs.

Contraintes techniques

Utiliser des outils adaptés à un workflow réaliste de développement (compatibilité, composants réutilisables, responsive). Rendre les maquettes directement exploitables pour les développeurs front-end tout en tenant compte des exigences techniques du projet.

Livrables

Maquettes modifiées et finalisées dans Figma avec structure responsive, prototypes de navigation simulant les parcours utilisateur, composants réutilisables intégrés dans un mini design system, et comparaison visuelle entre la wireframe brute et la version maquettée.

2. Précisez les moyens utilisés :

Outils utilisés :

- **Figma :**
 - Finalisation des maquettes et personnalisation graphique
 - Création de composants, prototypes, et déclinaisons responsive
- **HTML.to.Design :**
 - Génération initiale de maquette à partir de code HTML
- **LO FIER plugin**
 - Generation de wireframe à a partir d'une maquette

Méthodes :

- Travail hybride entre base générée et personnalisation manuelle
- Approche composant pour anticiper le développement front-end
- Grille de mise en page pour assurer l'alignement et la hiérarchie visuelle
- Responsive design intégré dès la phase de maquette

3. Avec qui avez-vous travaillé ?

J'ai travaillé seul sur ce projet.

4. Contexte

Nom de l'entreprise, organisme ou association ►

La plateforme

Chantier, atelier, service ►

Période d'exercice ► **Du : 12/10/2024 au : 22/11/2024**

5. Informations complémentaires (facultatif)

Voici le lien direct de la maquette :

[Lien Figma](#)

Activité-type 1 Développer la partie front-end d'une application web ou web mobile sécurisée

CP 3 ▶ Réaliser des interfaces statiques web ou web mobile

1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

Après la phase de maquettage validée dans Figma, j'ai développé les interfaces statiques de l'application web, c'est-à-dire l'intégration en HTML/CSS. Le travail consistait à transformer les maquettes en pages statiques fidèles, ergonomiques et prêtes à accueillir des fonctionnalités dynamiques par la suite. J'ai porté une attention particulière à la structure du code, à l'accessibilité, au responsive design et à la cohérence visuelle.

Activités réalisées

J'ai commencé par créer la structure HTML des pages à partir des maquettes Figma, section par section, en respectant la hiérarchie sémantique des balises telles que `<header>`, `<main>`, `<section>`, etc. Ensuite, j'ai intégré le style avec CSS en respectant précisément les espacements, les couleurs, les typographies et les dispositions définies dans Figma.

Pour les mises en page, j'ai utilisé Flexbox et Grid afin de créer des structures modernes et flexibles.

J'ai mis en place un design responsive en utilisant des media queries et des unités relatives comme les pourcentages, vw, vh, etc.

L'interface a été découpée en composants comme le header, le footer, les cards ou les formulaires afin de favoriser la réutilisabilité du code.

J'ai intégré des icônes, notamment via Font Awesome, ainsi que des images optimisées pour le web. J'ai ensuite testé l'interface sur différents navigateurs (Firefox, Chrome) et sur diverses tailles d'écran pour m'assurer de sa compatibilité et de sa réactivité.

Enfin, j'ai utilisé un serveur local de développement pour afficher les pages en temps réel et vérifier les ajustements au fur et à mesure.

Contraintes techniques

J'ai suivi rigoureusement les maquettes Figma afin d'obtenir un rendu le plus fidèle possible.

J'ai veillé à garantir une compatibilité multi-navigateurs tout en assurant un affichage responsive.

J'ai également maintenu une structure de code propre et claire, prête à accueillir ultérieurement la partie dynamique de l'application.

Objectif

Mon objectif était de convertir les maquettes Figma en pages HTML/CSS statiques et navigables. Je visais un rendu visuel complet, fidèle, ergonomique et responsive.

Il s'agissait également de poser les bases de l'interface pour y intégrer ensuite du JavaScript ou un framework front-end.

Livrables

Le projet comprend des fichiers HTML et CSS organisés de manière cohérente.

Les pages statiques sont fonctionnelles et compatibles aussi bien sur desktop que sur mobile.

Un dossier regroupe l'ensemble des assets utilisés, notamment les images et les icônes.

Enfin, un aperçu complet du rendu peut être visualisé dans le navigateur.

2. Précisez les moyens utilisés :

Langages :

- **HTML5**
- **CSS3** (avec Flexbox, Grid, media queries)

Outils et logiciels :

- **Visual Studio Code** (avec Live Preview pour le rafraîchissement auto)
- **Navigateur Chrome / Firefox Developer Edition**
- **Figma** (comme référence pour les maquettes)
- **Github** pour versionner le projet

Méthodes :

- Intégration "maquette → code" section par section
- Structuration en composants visuels pour anticiper la modularité
- Tests manuels sur différentes résolutions d'écran

3. Avec qui avez-vous travaillé ?

J'ai travaillé seul sur ce projet.

4. Contexte

Nom de l'entreprise, organisme ou association ► La plateforme

Chantier, atelier, service ►

Période d'exercice ► *Du : 25/11/2024 au : 27/12/2024*

5. Informations complémentaires (facultatif)

Structure du projet

Le projet est structuré de manière claire et organisée :

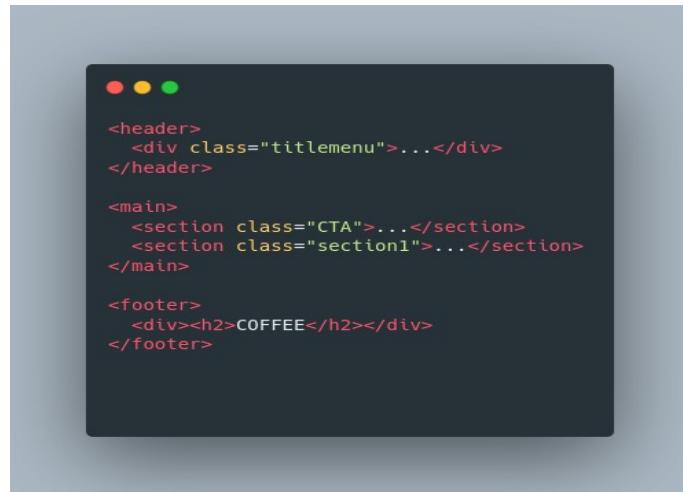
- index.html : contient la structure principale du site.
- style.css : gère l'ensemble des styles (mise en page, animations, couleurs, responsive, etc.).
- Le dossier public/ regroupe toutes les ressources visuelles (images, logos...).

Accessibilité et sémantique

Des balises HTML5 sémantiques ont été utilisées pour structurer le contenu du site de manière lisible : <header>, <main>, <section>, <footer>. Cela améliore l'accessibilité (notamment pour les lecteurs d'écran) et le référencement naturel (SEO).

Exemple de structure :

- Un <header> contenant le menu de navigation et le logo.
- Un <main> avec différentes sections (.CTA, .section1, .section2, etc.).
- Un <footer> simple contenant le nom du site.



```

<header>
  <div class="titlemenu">...</div>
</header>

<main>
  <section class="CTA">...</section>
  <section class="section1">...</section>
</main>

<footer>
  <div><h2>COFFEE</h2></div>
</footer>

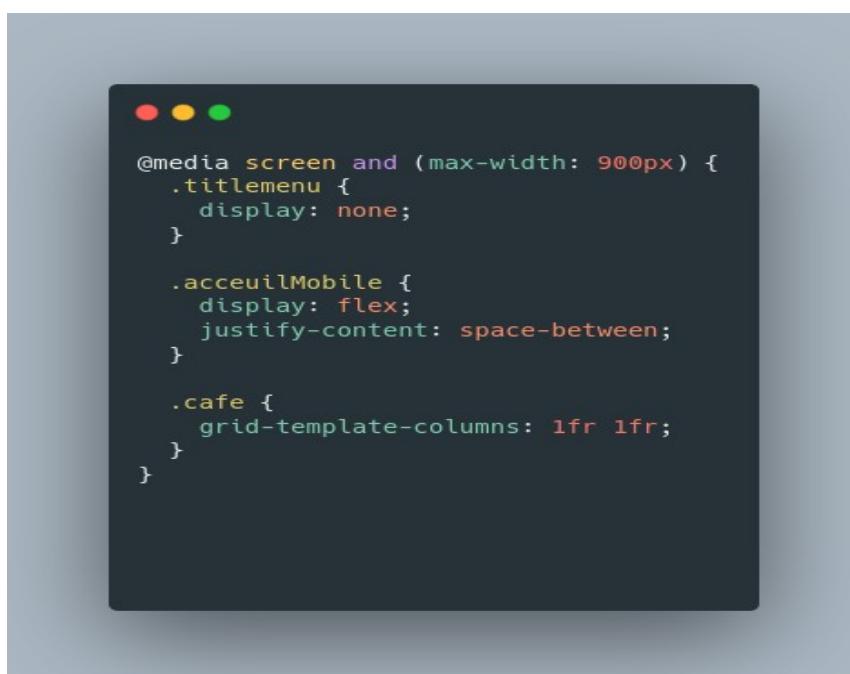
```

Responsive design (adaptation mobile)

Le site est entièrement responsive grâce à l'utilisation de media queries. Par exemple, à partir de 900px de largeur d'écran, certains éléments sont masqués ou réorganisés :

- .titlemenu est masqué sur mobile.
- .acceuilMobile est affiché avec un display: flex adapté.
- Le conteneur .cafe passe en deux colonnes sur mobile avec grid-template-columns: 1fr 1fr.

Cela permet une navigation fluide sur tous les types d'écrans (ordinateur, tablette, smartphone).



```

@media screen and (max-width: 900px) {
  .titlemenu {
    display: none;
  }

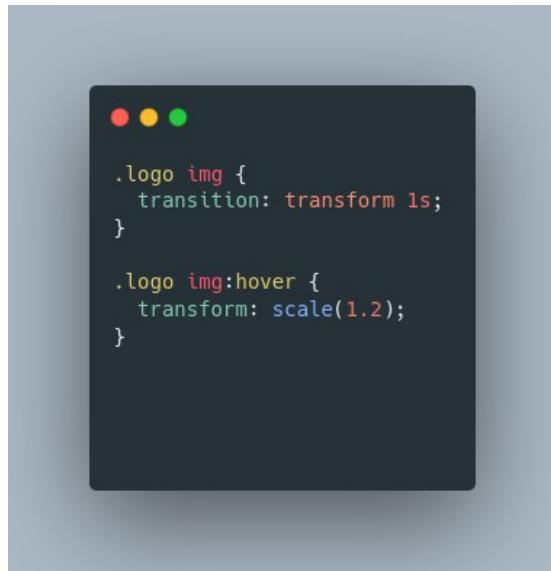
  .acceuilMobile {
    display: flex;
    justify-content: space-between;
  }

  .cafe {
    grid-template-columns: 1fr 1fr;
  }
}

```

Effets d'animation et interactions

Des animations CSS ont été utilisées pour dynamiser l'expérience utilisateur. Par exemple, le logo dans .logo img s'agrandit légèrement au survol grâce à transform: scale(1.2) et une transition: transform 1s. Cela rend l'interface plus vivante.

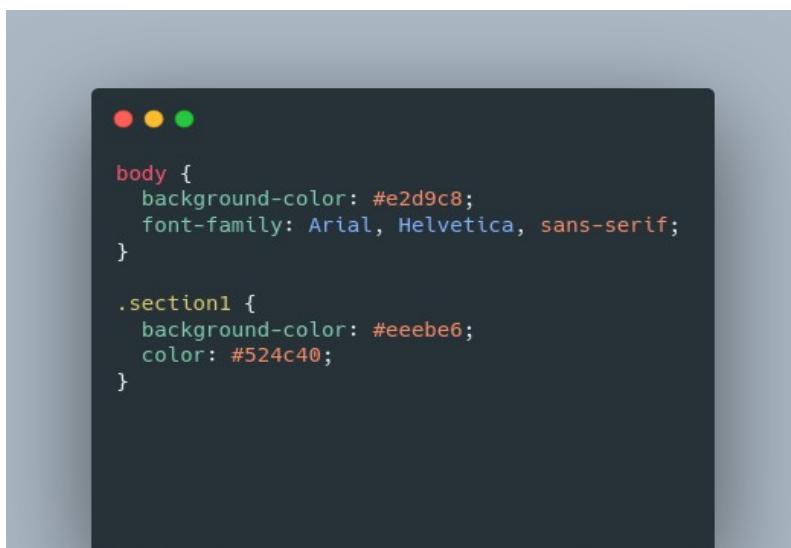


Hiérarchie visuelle et couleurs

J'ai utilisé une palette de couleurs sobres et chaudes qui rappellent l'univers du café :

- #e2d9c8 pour l'arrière-plan principal (beige clair)
- #eeebe6 pour les sections
- #524c40 pour le texte
- #e0b973 pour certains boutons et appels à l'action

La hiérarchie typographique est également soignée, avec des tailles de police bien différencierées, et une mise en page centrée sur la lisibilité.



Newsletter et formulaire

Une section en bas de page permet à l'utilisateur de s'inscrire à une newsletter. Elle contient un champ email avec un placeholder visuel (✉ email address) et un bouton Subscribe. Le formulaire est purement visuel et ne déclenche pas de traitement côté serveur.



Intégration de widget externe

Un widget Elfsight a été intégré dans la section .section3 pour simuler une zone d'avis ou de témoignages clients. Il est chargé dynamiquement via un script JS fourni par la plateforme.



Voici le lien github du projet

<https://github.com/Namoux/HTML-CSS/tree/main/Coffee>

Activité-type 1

Développer la partie front-end d'une application web ou web mobile sécurisée

CP 4

Développer la partie dynamique des interfaces utilisateur web ou web mobile

1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

Dans le cadre de mon projet front-end, j'ai développé une application de type To-Do list en JavaScript vanilla (sans framework), dans le but de mettre en œuvre l'interactivité entre l'utilisateur et l'interface.

Activités réalisées

J'ai commencé par la création d'une structure HTML simple, incluant un champ de saisie, un bouton d'ajout, et une liste pour afficher les tâches. Ensuite, j'ai utilisé JavaScript pour gérer toutes les fonctionnalités clés : l'ajout dynamique de tâches à la liste (via clic ou touche "Entrée"), le marquage d'une tâche comme complétée (toggle), la suppression de tâches, et l'affichage automatique d'un message si la liste devient vide.

J'ai mis en œuvre l'écoute d'événements avec addEventListener, permettant de réagir aux interactions utilisateur comme les clics ou la saisie clavier. Toute la logique repose sur la manipulation du DOM pour insérer, modifier ou supprimer des éléments dynamiquement, sans jamais recharger la page.

Pour assurer une persistance locale des données, j'ai intégré localStorage, ce qui permet de conserver la liste des tâches même après un recharge du navigateur. L'état de l'application est géré entièrement côté client, avec une liste des tâches stockée en mémoire et mise à jour en temps réel. L'interface respecte le design défini dans les maquettes Figma, garantissant une cohérence visuelle avec les attentes initiales.

Contraintes techniques

Application 100 % front-end, sans base de données ni serveur. J'ai utilisé uniquement JavaScript pur, sans bibliothèque ni framework. Le rendu est entièrement dynamique, sans rechargement de page. L'ensemble du projet reste volontairement simple et fonctionnel pour se concentrer sur la logique d'interactivité.

Objectifs

Créer une interface interactive permettant à l'utilisateur d'ajouter, modifier ou supprimer des données en direct. Démontrer ma maîtrise des bases du JavaScript appliquée à une interface utilisateur. Préparer le terrain pour une version future connectée intégrant un back-end ou une API.

Livrables

Une application To-Do list fonctionnelle développée en HTML, CSS et JavaScript. Un code entièrement commenté. Une démo locale accessible dans le navigateur. Une sauvegarde automatique des tâches via localStorage

2. Précisez les moyens utilisés :

Langages :

- HTML5
- CSS3
- JavaScript

Logiciels et outils :

- **Visual Studio Code**
- **Navigateur** pour les tests (Chrome, Firefox)
- **Github** pour le versioning
- Figma (pour le design de l'interface)

Méthodes :

- Manipulation du **DOM** en JavaScript
- Gestion des **événements** utilisateurs
- Mise à jour dynamique de l'interface sans rechargement
- Utilisation de localStorage pour la sauvegarde locale

3. Avec qui avez-vous travaillé ?

J'ai travaillé seul sur ce projet

4. Contexte

Nom de l'entreprise, organisme ou association ▶

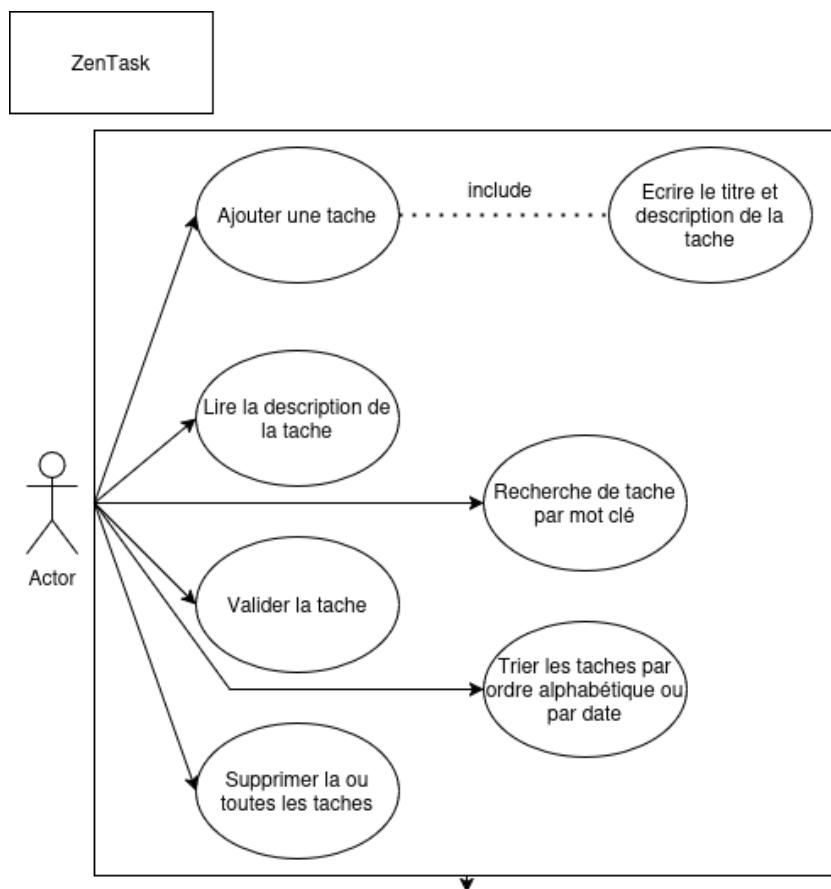
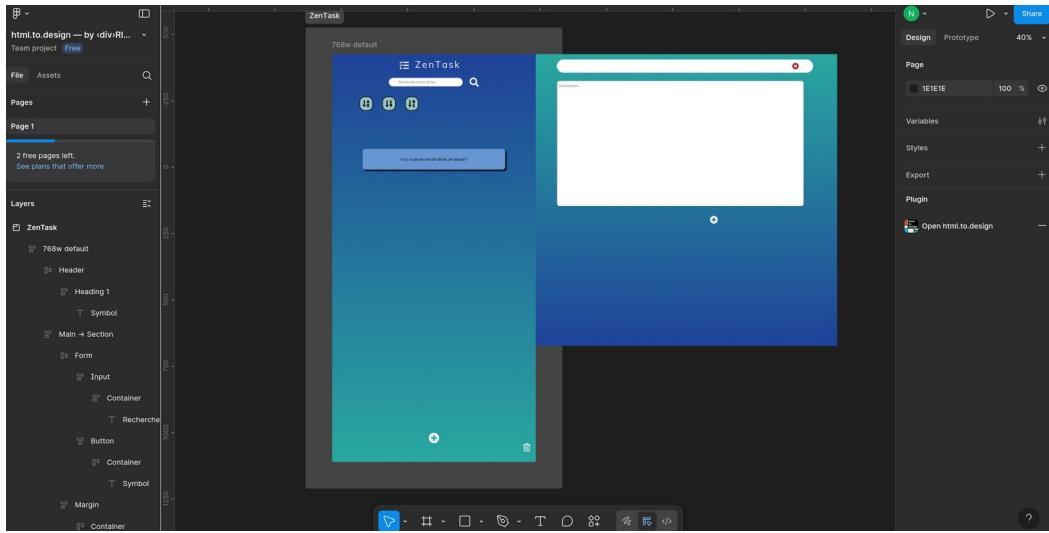
La plateforme

Chantier, atelier, service ▶

Période d'exercice ▶ *Du : 06/01/2025 au : 21/02/2025*

5. Informations complémentaires (facultatif)

Dans le cadre de ce projet, j'ai également réalisé la maquette de l'application ainsi qu'un use case détaillé pour mieux définir les fonctionnalités et l'expérience utilisateur. La maquette a été conçue à l'aide de Figma, ce qui m'a permis de visualiser l'interface avant de commencer le développement. Le use case a été essentiel pour identifier les interactions clés et les besoins des utilisateurs.



Voici quelques extraits de code des fonctions principales qui illustrent le fonctionnement de l'application :

1. Fonction pour ajouter une tâche :



```
export function onSubmit(event) {
    event.preventDefault();
    const formData = new FormData(event.target);
    const date = new Date();
    const formattedDate = date.toLocaleString();

    let task = {
        ID: CreateId(),
        title: formData.get("inputTitre"),
        description: formData.get("description"),
        date: formattedDate,
        check: false,
    };

    createTask(task.ID, task.title, task.description, task.date, task.check);
    cloneTask(task);

    // Réinitialiser les champs de saisie
    document.querySelector("#inputTitre").value = "";
    document.querySelector("#description").value = "";
}
```

La fonction onSubmit est déclenchée lorsque l'utilisateur valide le formulaire d'ajout de tâche. J'utilise event.preventDefault() afin d'éviter le rechargement de la page, ce qui permet de garder l'expérience utilisateur fluide et sans interruption. Je commence par récupérer les données du formulaire avec l'objet FormData, ce qui me permet d'accéder aux champs saisis, notamment le titre et la description de la tâche. Ensuite, je génère la date et l'heure de création au format local grâce à toLocaleString().

Je construis un objet task contenant toutes les informations nécessaires : un identifiant unique, le titre, la description, la date et un booléen check initialisé à false. Cet objet est ensuite enregistré dans le localStorage via la fonction createTask, ce qui permet de sauvegarder la tâche même après un rechargement de la page. J'appelle aussi OnClickaddTask() pour mettre à jour l'état de l'application, puis cloneTask pour afficher visuellement la tâche dans l'interface. J'utilise également sortbyIDdecreasing() pour trier les tâches dans l'ordre décroissant de leur création.

Enfin, je vide les champs du formulaire pour permettre l'ajout d'une nouvelle tâche, et je masque dynamiquement les messages indiquant qu'aucune tâche n'est présente si besoin. Grâce à cette fonction, j'assure une interactivité complète et une persistance locale efficace pour l'ajout de tâches.

2. Fonction pour afficher toutes les tâches :



```
export function printAllTasks() {
    const Alltasks = getAllTasks();
    RemoveTaskHTML();
    if (Alltasks != null) {
        Alltasks.forEach((task) => {
            cloneTask(task);
        });
    } else {
        AppearComponentNoTask();
    }
}
```

La fonction printAllTasks me permet d'afficher dynamiquement toutes les tâches actuellement enregistrées dans le localStorage. Elle commence par appeler getAllTasks() pour récupérer la liste complète des tâches, que je stocke dans une variable Alltasks. Avant d'afficher quoi que ce soit, je m'assure de nettoyer l'interface utilisateur en supprimant les anciennes tâches déjà affichées à l'écran, grâce à la fonction RemoveTaskHTML(). Cela évite tout doublon lors du rafraîchissement de l'affichage.

Ensuite, si des tâches sont présentes (et que Alltasks n'est pas null), je parcours ce tableau avec forEach et j'utilise la fonction cloneTask pour afficher chaque tâche à l'aide d'un template HTML. Si Alltasks est null ou si le tableau est vide (length === 0), j'appelle AppearComponentNoTask() pour afficher un message à l'utilisateur indiquant qu'aucune tâche n'est enregistrée.

Cette fonction est essentielle pour synchroniser visuellement le contenu du localStorage avec l'interface, notamment lors du chargement initial de l'application ou après la suppression de toutes les tâches.

3. Fonction pour supprimer une tâche :

A screenshot of a dark-themed code editor window. At the top, there are three circular icons: red, yellow, and green. Below them, the code is displayed in a monospaced font:

```
BTNdelete.addEventListener("click", () => {
  let PositionTab = 0;
  let Alltasks = getAllTasks();
  for (let i = 0; i < Alltasks.length; i++) {
    if (Alltasks[i].ID === task.ID) {
      PositionTab = i;
      break;
    }
  }
  Alltasks.splice(PositionTab, 1);
  localStorage.setItem("tasks", JSON.stringify(Alltasks));
  divTask.remove();
});
```

Ce bloc de code correspond à l'écouteur d'événement associé au clic sur le bouton de suppression d'une tâche. Lorsqu'un utilisateur clique sur le bouton `BTNdelete`, je commence par identifier la position de la tâche à supprimer dans le tableau `Alltasks`, que je récupère depuis le `localStorage`. Je compare chaque ID présent dans le tableau avec l'ID de la tâche cliquée afin d'en déterminer la position exacte.

Une fois la tâche localisée, j'ajoute la classe CSS "slide" à deux éléments HTML (`divTask` et `divDescription`) pour appliquer une transition visuelle avant la suppression — cela donne un effet fluide à la suppression, ce qui améliore l'expérience utilisateur.

Après un délai de 500 millisecondes (le temps de laisser jouer l'animation), j'utilise `splice` pour retirer la tâche du tableau à l'index trouvé. Ensuite, je supprime les éléments HTML correspondants du DOM avec `remove()`, et je mets à jour le `localStorage` en y réenregistrant le tableau modifié sous forme de JSON.

Enfin, je vérifie si le tableau des tâches est désormais vide. Si c'est le cas, j'appelle `AppearComponentNoTask()` pour afficher un message d'absence de tâche, et je supprime carrément la clé "tasks" du `localStorage` pour éviter de conserver un tableau vide inutilement. Cette gestion permet à l'application de rester propre, réactive et fidèle à l'état réel des données.

En résumé, à travers ces différentes fonctions, j'ai pleinement géré la manipulation du DOM pour rendre l'application dynamique et interactive. Chaque ajout, suppression ou affichage de tâche se fait sans rechargement de la page, grâce à l'écoute d'événements et à une mise à jour en temps réel de l'interface. En combinant JavaScript pur, localStorage et gestion fine des éléments HTML, j'ai conçu une To-Do list fluide, réactive, et fidèle à l'état des données. Ce projet démontre ma capacité à contrôler le DOM efficacement pour créer une expérience utilisateur intuitive.

Lien github

<https://github.com/Namoux/Todolist-JS>

Activité-type 2

Développer la partie back-end d'une application web ou web mobile sécurisée

CP 5 ▶

Mettre en place une base de donnée relationnelle

1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

Dans le cadre du développement d'un mini shop en ligne, j'ai mis en place une base de données relationnelle MariaDB afin de stocker efficacement les informations liées aux produits, utilisateurs et commandes.

Activités réalisées

J'ai d'abord conçu le schéma de la base de données en m'appuyant sur l'outil en ligne [drawdb.vercel.app](#), ce qui m'a permis de visualiser clairement les entités, les relations et les dépendances entre les tables. La base est structurée autour de plusieurs tables principales : product, user, category, cart, orders, ainsi que des tables de jointure comme productCategory, productCart et productOrder, permettant de gérer les relations de type plusieurs-à-plusieurs entre les entités.

Pour déployer la base dans un environnement propre et reproductible, j'ai installé et configuré MariaDB via Docker sur mon système Linux Debian. Cette approche m'a permis un déploiement rapide, isolé et portable de la base de données, tout en garantissant une bonne maîtrise des paramètres d'environnement.

Une fois le conteneur opérationnel, j'ai utilisé des outils classiques comme mysql en ligne de commande pour interagir avec la base. J'ai ensuite écrit et exécuté des requêtes SQL pour manipuler les données : création des tables, insertion de produits, gestion des utilisateurs, des paniers, des catégories, etc. Des tests fonctionnels ont été réalisés pour valider l'intégration avec le back-end en cours de développement.

Contraintes techniques

J'ai utilisé Docker pour isoler MariaDB dans un conteneur, évitant toute installation directe sur l'hôte local. La configuration du conteneur inclut la gestion des ports et des volumes pour garantir la persistance des données. Chaque champ de table a été défini avec un type de données adapté, ce qui facilite la robustesse et la scalabilité du système, en vue d'une intégration fluide avec un serveur back-end.

Objectifs

Déployer un environnement de base de données fiable, maintenable et adapté aux besoins du projet grâce à Docker. Structurer les données produits de manière relationnelle pour faciliter leur gestion. Faciliter la montée en charge et le développement local grâce à Docker. Préparer l'interconnexion avec des API REST côté back-end.

Livrables

Une commande Docker permettant de lancer l'instance MariaDB. Un script SQL complet pour la création des tables et des relations. Une base fonctionnelle et accessible depuis mon environnement de développement, prête à être connectée au serveur back-end.

2. Précisez les moyens utilisés :

Outils et technologies :

- **MariaDB** (base de données relationnelle)
- **Docker** (conteneurisation de MariaDB)
- **Linux Debian** comme système hôte
- **Terminal** (ligne de commande Docker, mysql)
- **Visual Studio Code** pour écrire les scripts SQL et le back-end

Méthodes :

- Installation et configuration de conteneurs Docker pour déployer MariaDB
- Gestion des volumes Docker pour la persistance des données
- Utilisation de requêtes SQL standards pour gérer la base
- Tests d'accès et manipulation via back-end

3. Avec qui avez-vous travaillé ?

J'ai travaillé seul sur ce projet

4. Contexte

Nom de l'entreprise, organisme ou association ►

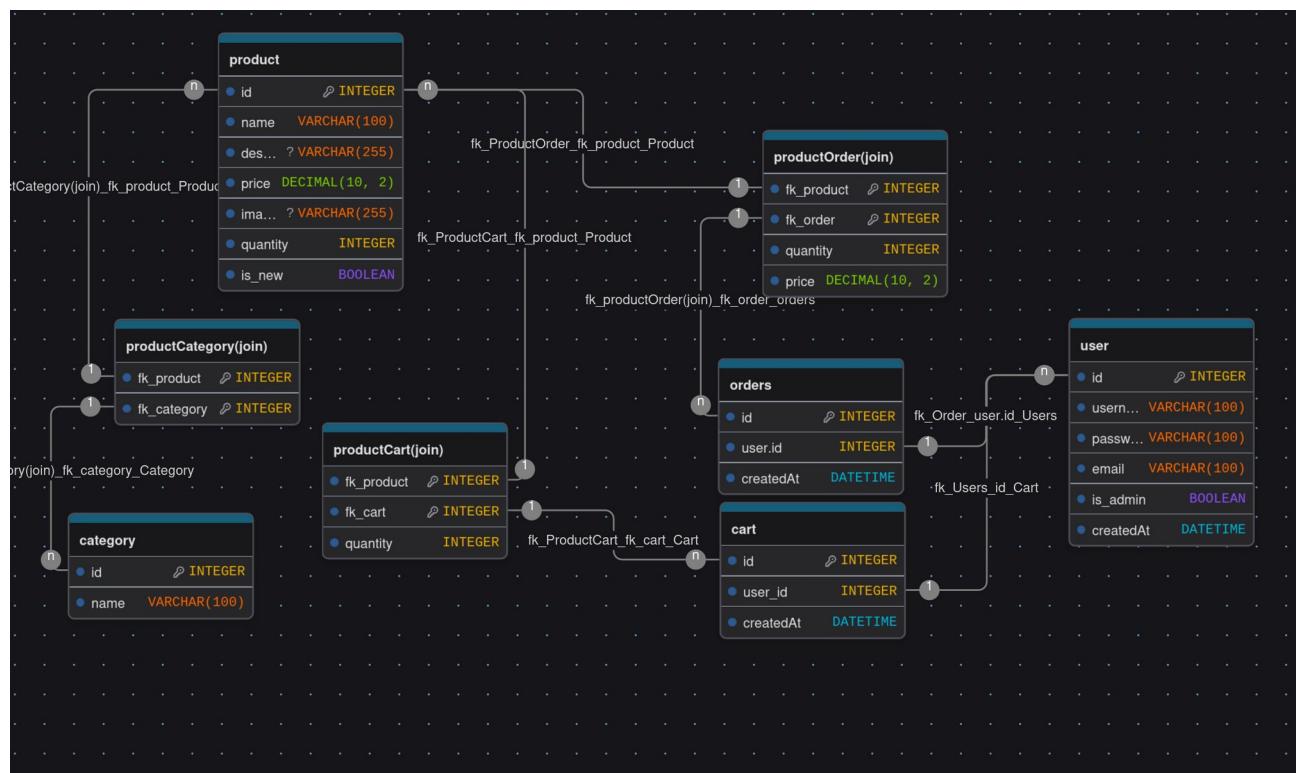
La plateforme

Chantier, atelier, service ►

Période d'exercice ► Du : 24/02/2025 au : 14/03/2025

5. Informations complémentaires (facultatif)

Diagramme de la base de donnée

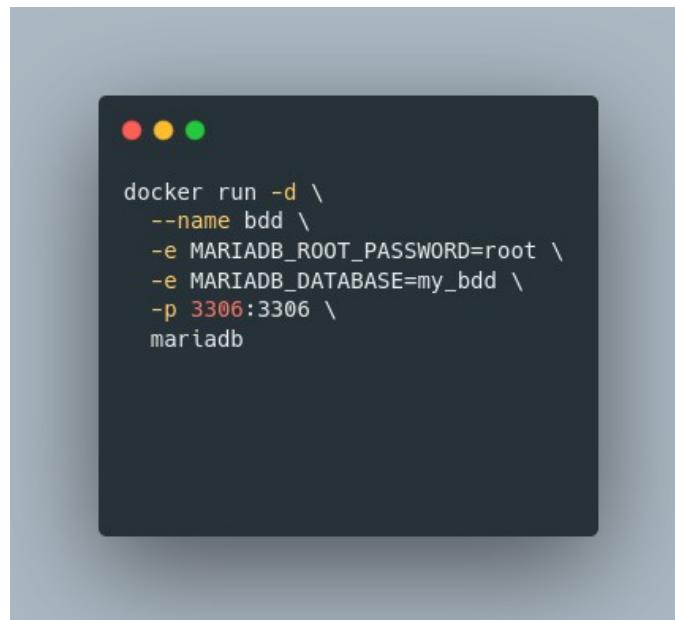


Déploiement de MariaDB avec Docker

Pour isoler et simplifier l’installation de la base de données, un conteneur **MariaDB** a été déployé avec **Docker**. Deux méthodes ont été explorées : en ligne de commande, et via **Docker Desktop** (interface graphique).

Méthode 1 – Docker CLI (terminal)

Création d’un conteneur MariaDB en ligne de commande avec persistance des données et configuration des variables d’environnement :



Méthode 2 – Docker Desktop (GUI)

1. Recherche de l’image mariadb dans la bibliothèque d’images.
2. Téléchargement via le bouton **Pull**.
3. Création d’un container depuis l’image :



L'image joue un rôle similaire à une ISO : elle permet de lancer plusieurs instances (conteneurs), chacune exécutant le service souhaité.

Vérification dans l'onglet *Containers* de Docker Desktop que le conteneur fonctionne (pastille verte + port 3306 ouvert).

The screenshot shows the Docker Desktop interface. The left sidebar has 'Containers' selected. The main area displays two containers:

Name	Container ID	Image	Port(s)	CPU (%)	Actions
bdd	5a789d56856a	mariadb:lat	3306:3306	0.03%	[Actions]
dockercompose	-	-	-	0.77%	[Actions]

At the bottom, it says 'Showing 2 items'.

Connexion à MariaDB via un client MySQL

Un client MariaDB (ligne de commande) est installé localement pour exécuter des requêtes SQL directement sur le conteneur :

```
sudo apt install mariadb-client
```

Connexion au serveur :

```
mariadb --host=127.0.0.1 --user=root --password=root
```

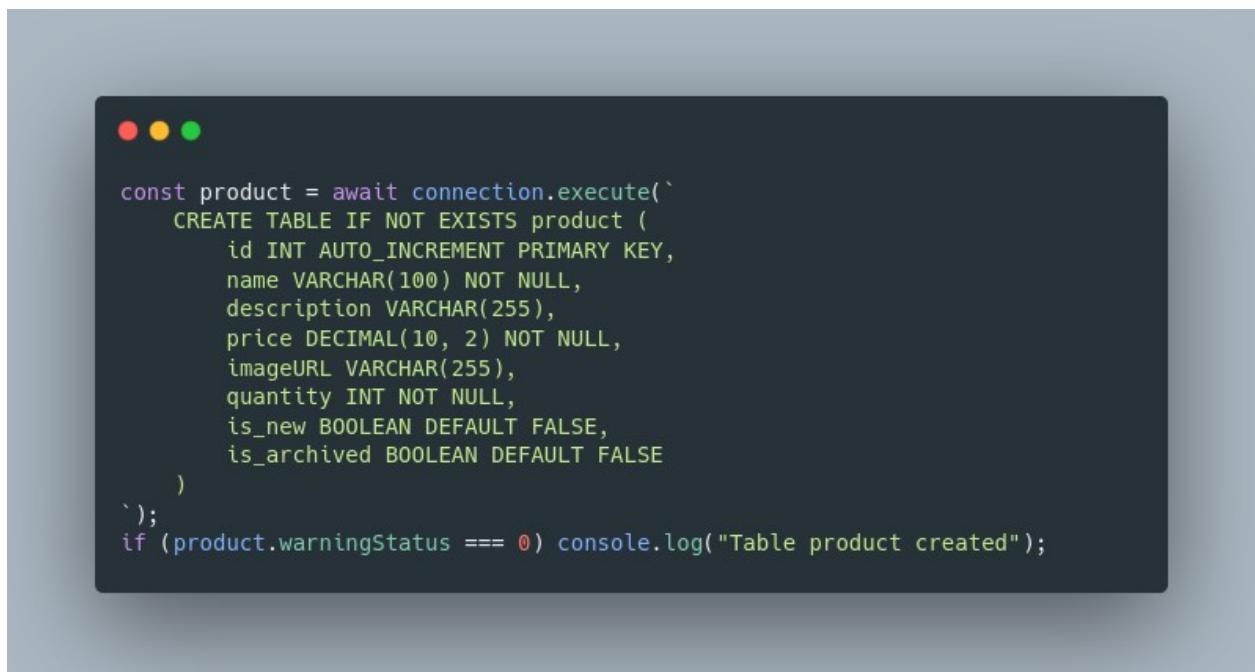
Création automatique des tables via le back-end

La création des tables ne se fait pas manuellement via un client SQL, mais de manière automatisée dès le lancement du projet. Le fichier principal server.js exécute automatiquement le module database.mjs, qui contient toutes les instructions SQL de création des tables avec la commande CREATE TABLE IF NOT EXISTS.

Cela signifie que :

- Lors du premier lancement, toutes les tables nécessaires à l'application sont créées dynamiquement dans MariaDB si elles n'existent pas encore.
- Si la base existe déjà, aucune duplication n'a lieu grâce à l'utilisation de IF NOT EXISTS.
- Ce fonctionnement permet de garder une base de données toujours prête à l'emploi, sans manipulation manuelle.

Ce processus simplifie le déploiement et le développement local : il suffit de démarrer l'application pour que la structure de base de données soit disponible et connectée.



```
const product = await connection.execute(`CREATE TABLE IF NOT EXISTS product (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    description VARCHAR(255),
    price DECIMAL(10, 2) NOT NULL,
    imageURL VARCHAR(255),
    quantity INT NOT NULL,
    is_new BOOLEAN DEFAULT FALSE,
    is_archived BOOLEAN DEFAULT FALSE
)
`);
if (product.warningStatus === 0) console.log("Table product created");
```

Une fois connecté, on peut manipuler les bases :

```

namoux@NamZenbook:~$ mariadb --host=127.0.0.1 --user=root --password=root
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 3
Server version: 11.7.2-MariaDB-ubuntu2404 mariadb.org binary distribution

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]> show databases;
+-----+
| Database      |
+-----+
| GestionMagasin |
| information_schema |
| my_back        |
| mysql          |
| performance_schema |
| shop           |
| shopTest       |
| sys            |
+-----+
8 rows in set (0,009 sec)

MariaDB [(none)]> use shop;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed

```

Structure de la base shop

La base shop est structurée autour de plusieurs entités et relations. Voici un aperçu des tables créées :

Tables principales :

- product : contient les informations sur les produits (nom, description, prix, stock, image, etc.).
- user : stocke les données des utilisateurs, avec un champ is_admin pour différencier les rôles.
- category : permet de regrouper les produits par catégories (ex. : smartphones, accessoires).
- cart : représente un panier utilisateur en cours de constitution.

- orders : table représentant une commande passée.

Tables de jointure :

- productCategory : table de relation plusieurs-à-plusieurs entre product et category, permettant qu'un produit appartienne à plusieurs catégories.
- productCart : table de jointure entre product et cart, gérant les produits présents dans un panier donné, avec leur quantité.
- productOrder : gère les détails des produits commandés dans une commande (orders), en incluant la quantité et le prix au moment de l'achat.

Chaque table de jointure utilise des clés étrangères pointant vers les entités principales, assurant l'intégrité référentielle, et certaines relations sont complétées par une suppression en cascade (ON DELETE CASCADE).

```
MariaDB [shop]> show tables;
+-----+
| Tables_in_shop |
+-----+
| cart           |
| category       |
| orders         |
| product        |
| productCart    |
| productCategory|
| productOrder   |
| user           |
+-----+
8 rows in set (0,002 sec)

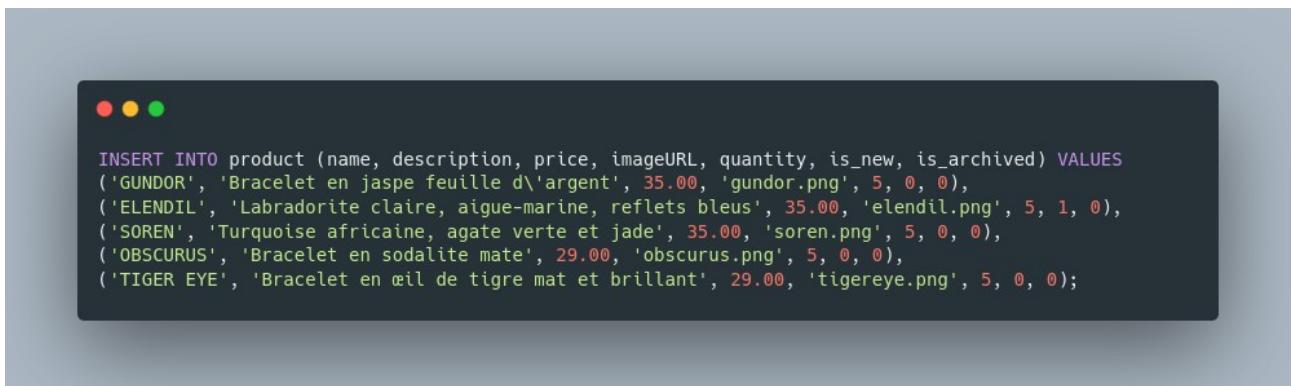
MariaDB [shop]> describe product;
+-----+-----+-----+-----+-----+-----+
| Field      | Type       | Null | Key | Default | Extra          |
+-----+-----+-----+-----+-----+-----+
| id         | int(11)    | NO  | PRI | NULL    | auto_increment |
| name       | varchar(100) | NO  |     | NULL    |                |
| description | varchar(255) | YES |     | NULL    |                |
| price      | decimal(10,2) | NO  |     | NULL    |                |
| imageURL   | varchar(255) | YES |     | NULL    |                |
| quantity    | int(11)    | NO  |     | NULL    |                |
| is_new      | tinyint(1)  | YES |     | 0       |                |
| is_archived | tinyint(1)  | YES |     | 0       |                |
+-----+-----+-----+-----+-----+-----+
8 rows in set (0,003 sec)

MariaDB [shop]> describe productCategory;
+-----+-----+-----+-----+-----+
| Field      | Type      | Null | Key | Default | Extra          |
+-----+-----+-----+-----+-----+
| fk_product | int(11)  | NO  | PRI | NULL    |                |
| fk_category | int(11)  | NO  | PRI | NULL    |                |
+-----+-----+-----+-----+-----+
```

Exemple de commande de requête SQL :

Insertion de données :

J'utilise la commande insert into.



```
● ● ●

INSERT INTO product (name, description, price, imageURL, quantity, is_new, is_archived) VALUES
('GUNDOR', 'Bracelet en jaspe feuille d\'argent', 35.00, 'gundor.png', 5, 0, 0),
('ELENDIL', 'Labradorite claire, aigue-marine, reflets bleus', 35.00, 'elendil.png', 5, 1, 0),
('SOREN', 'Turquoise africaine, agate verte et jade', 35.00, 'soren.png', 5, 0, 0),
('OBSCURUS', 'Bracelet en sodalite mate', 29.00, 'obscurus.png', 5, 0, 0),
('TIGER EYE', 'Bracelet en œil de tigre mat et brillant', 29.00, 'tigereye.png', 5, 0, 0);
```

Pour lire les données contenues dans la table product.

SELECT * FROM Produits;

Résultats :

id	name	description	price	imageURL	quantity	is_ne w	is_archive d
1	GUNDOR	Bracelet en jaspe feuille d'argent	35.00	gundor.png	5	0	0
2	ELENDIL	Labradorite claire, aigue-marine, reflets bleus	35.00	elendil.png	5	1	0
3	SOREN	Turquoise africaine, agate verte et jade	35.00	soren.png	5	0	0
4	OBSCURUS	Bracelet en sodalite mate	29.00	obscurus.png	5	0	0
5	TIGER EYE	Bracelet en œil de tigre mat et brillant	29.00	tigereye.png	5	0	0

Grâce à ce projet, j'ai bien compris les fondements d'une base de données relationnelle : de la conception du schéma logique à sa mise en œuvre concrète avec MariaDB, en passant par l'utilisation de Docker pour faciliter le déploiement dans un environnement isolé. J'ai appris à structurer les données de manière cohérente, à utiliser des types adaptés, à manipuler les enregistrements via des requêtes SQL, et à assurer la persistance des données.

L'expérience m'a permis de poser des bases solides pour une utilisation réelle en environnement connecté, et m'a donné les bons réflexes pour aller plus loin avec une API ou un back-end.

Activité-type 2

Développer la partie back-end d'une application web ou web mobile sécurisée

CP 6

Développer les composants d'accès aux données
SQL et NoSQL

1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

Pour le mini shop, j'ai développé un serveur back-end en Node.js en m'appuyant sur le framework Express.js. Ce serveur assure la communication entre la base de données MariaDB et le front-end, permettant une interaction fluide et structurée avec les données du magasin.

Activités réalisées

Dans un premier temps, j'ai mis en place un serveur Express.js capable de gérer les principales requêtes HTTP : GET, POST, PUT et DELETE. J'ai conçu des routes spécifiques pour interagir avec la table product, notamment pour récupérer la liste des produits, en ajouter un nouveau, modifier un produit existant ou le supprimer.

En complément, j'ai étendu cette logique aux autres tables du schéma relationnel que j'ai mis en place : la table user pour gérer les comptes utilisateurs avec leur rôle (administrateur ou non), la table category pour classer les produits, la table cart pour représenter les paniers, et la table orders pour gérer l'historique des commandes. J'ai aussi traité les relations complexes à travers les tables de jointure productCategory, productCart et productOrder, qui permettent de gérer les liens plusieurs-à-plusieurs entre les entités (produits/catégories, produits/paniers, produits/commandes).

Les données sont directement gérées dans MariaDB, et j'utilise le terminal pour effectuer des opérations manuelles via des commandes SQL en parallèle du développement back-end. Toutes les requêtes côté serveur sont asynchrones, gérées avec `async/await`, ce qui assure une bonne fluidité, une gestion correcte des délais de réponse, et évite les blocages du serveur.

Je traite les données envoyées par le client en appliquant une validation minimale avant leur insertion en base, afin de garantir l'intégrité et la cohérence. Pour tester les endpoints, j'ai utilisé Postman ainsi que le front-end connecté au back-end en cours de développement. La gestion des erreurs est rigoureuse : je retourne des réponses HTTP claires et adaptées à chaque cas (200 pour une réussite, 201 pour une création, 400 pour une erreur client, 404 si une ressource est introuvable, 500 pour une erreur serveur).

Contraintes techniques

j'ai structuré le serveur selon une architecture REST simple et efficace. J'ai veillé à maintenir une séparation nette entre la logique applicative du serveur et l'accès aux données en base. J'ai appliqué les bonnes pratiques Node.js, notamment la gestion asynchrone, la modularisation du code, et l'utilisation des middlewares Express pour gérer les différentes couches (authentification, parsing, logging, etc.).

Objectif

L'objectif principal de ce serveur est de fournir une API REST fonctionnelle pour le mini shop, capable d'assurer une communication fiable et évolutive entre le front-end et la base de données. Cela me permet aussi de prévoir l'ajout futur d'autres fonctionnalités, comme l'authentification utilisateur, la gestion des commandes ou encore le suivi des stocks.

Livrables

j'ai produit le code source complet du serveur Node.js utilisant Express, avec un ensemble de routes CRUD opérationnelles sur la ressource product ainsi que les autres entités liées (user, cart, orders, etc.). Tous les endpoints ont été testés et validés soit via Postman, soit depuis le front-end. J'ai également rédigé une documentation sommaire des principales routes de l'API, ce qui facilitera l'intégration future côté client ou pour d'autres développeurs.

2. Précisez les moyens utilisés :

Langages et frameworks :

- JavaScript (Node.js)
- Express.js pour la gestion du serveur HTTP et des routes
- Module MariaDB

Outils et logiciels :

- Visual Studio Code
- Postman pour tester les API

- Terminal Linux Debian
- Git pour le versioning

Méthodes

- Architecture RESTful
- Programmation asynchrone avec async/await
- Modularisation du code (routes, contrôleurs, services)
- Tests fonctionnels manuels avec Postman

3. Avec qui avez-vous travaillé ?

J'ai travaillé seul sur ce projet

4. Contexte

Nom de l'entreprise, organisme ou association ▶	La plateforme
Chantier, atelier, service ▶	
Période d'exercice ▶	<i>Du : 17/03/2025 au : 25/04/2025</i>

5. Informations complémentaires (facultatif)

Mise en place d'un serveur Node.js avec Express

Pour connecter le front-end au back-end et à la base MariaDB, un serveur Node.js a été développé en utilisant Express.js. Il expose une API REST permettant de gérer l'ensemble des ressources du

shop. Afin d'assurer une communication efficace avec MariaDB, j'ai structuré le projet autour de composants d'accès aux données séparés sous forme de modèles (models).

Lancement du serveur

Le serveur écoute sur le port 4004 et interagit avec la base de données shop, hébergée dans MariaDB et conteneurisée via Docker. L'application est entièrement connectée à cette base relationnelle, que j'ai conçue au préalable et qui regroupe plusieurs tables interconnectées : product, user, category, cart, orders, ainsi que les tables de jointure comme productCategory, productCart et productOrder.

Pour établir cette connexion entre le serveur Node.js et MariaDB, j'utilise le package mariadb. La connexion est initiée dès le lancement de l'application à l'aide d'un appel `createConnection()`, dans lequel je précise les informations de connexion : hôte, port, identifiants d'accès, et nom de la base de données.

A screenshot of a terminal window on a Mac OS X system. The window has the characteristic blue title bar with red, yellow, and green buttons. The main area of the terminal shows the following code in white text on a dark background:

```
const connection = await mariadb.createConnection({
  host: "localhost",
  port: 3306,
  user: "root",
  password: "root",
  database: "my_back"
});
```

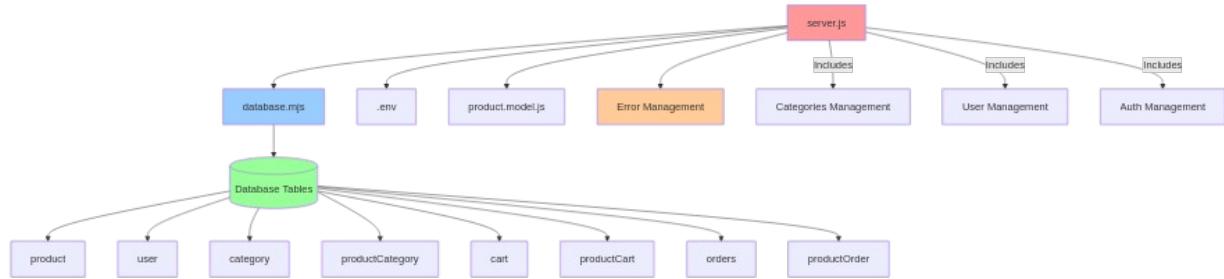
Cette connexion est ensuite partagée avec les différents modules d'accès aux données (*.model.js) pour exécuter les requêtes SQL. Elle reste persistante tout au long du cycle de vie du serveur, assurant une communication continue avec la base MariaDB.

Architecture générale des modèles

Au début de ma formation, je n'étais pas encore familier avec les principes de l'architecture MVC (Modèle-Vue-Contrôleur). Mes premiers projets étaient souvent complexes à maintenir : je produisais un code dit "spaghetti", mêlant logique métier, interface utilisateur et gestion des

événements dans un même bloc. Cette absence de structuration rendait la lecture, la maintenance et l'évolution de mes applications particulièrement difficiles.

Chaque nouvelle fonctionnalité ou correction de bug devenait un défi, car il fallait naviguer dans un code dense et peu lisible, sans réelle séparation des responsabilités. Cela nuisait à la qualité globale de mes projets.



La découverte de l'architecture MVC a marqué un tournant décisif. J'ai compris l'importance de séparer les différentes couches de l'application : le Modèle, chargé de la logique métier et de la gestion des données ; la Vue, dédiée à l'interface utilisateur ; et le Contrôleur, qui fait le lien entre les deux. Cette structuration m'a permis de produire un code plus clair, plus modulaire et donc bien plus facile à maintenir et à faire évoluer.

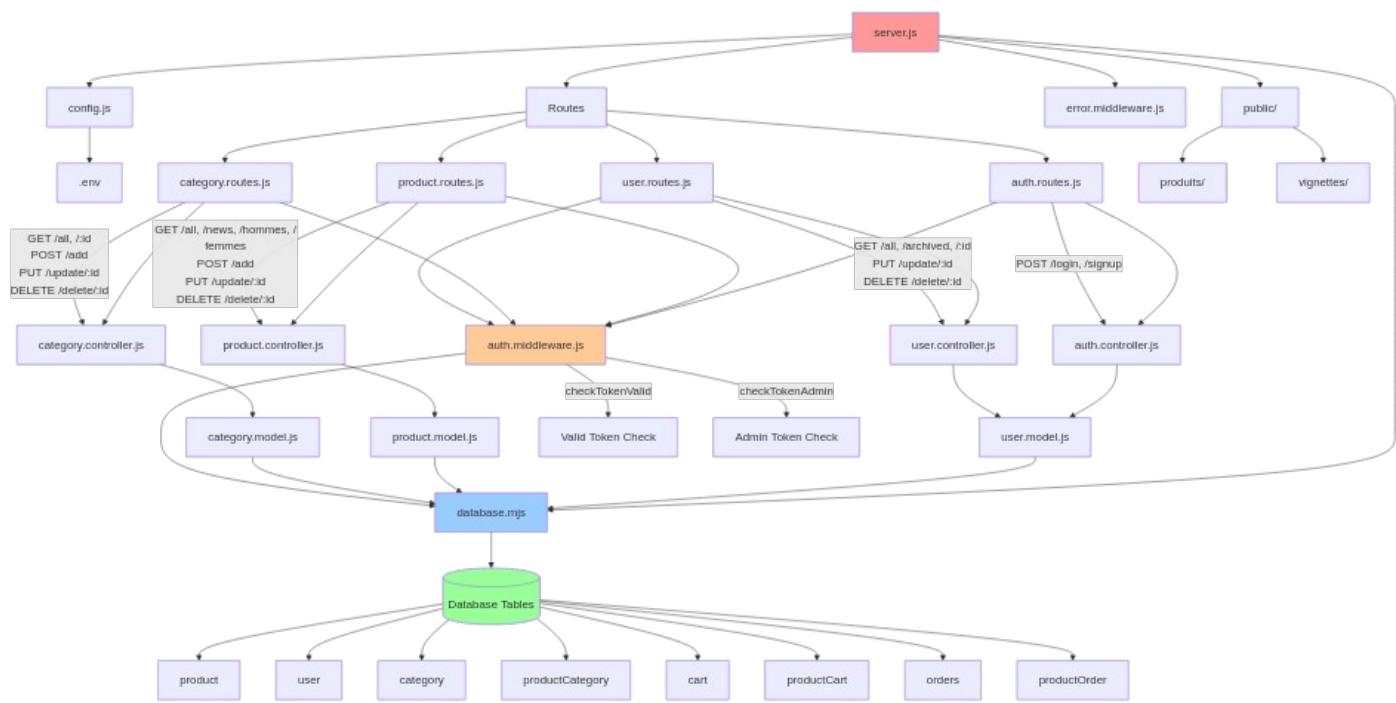
J'ai respecté une architecture modulaire et évolutive en séparant clairement la logique Express (côté serveur) de la couche d'accès aux données (models). Chaque ressource principale possède son propre fichier de modèle JavaScript, dans lequel je centralise les requêtes SQL :

- `product.model.js` : contient toutes les requêtes liées aux produits, notamment l'insertion, la suppression, la mise à jour et la lecture (avec ou sans filtre).
- `user.model.js` : gère l'accès aux données utilisateurs (création de comptes, récupération, suppression logique, etc.).
- `category.model.js` : permet de récupérer ou créer des catégories et de gérer les associations produit/catégorie via une table de jointure.
- `cart.model.js` : modèle associé à la gestion des paniers, création automatique au moment de l'inscription, ajout et suppression d'articles.
- `orders.model.js` : accès aux données de commandes passées par les utilisateurs, avec gestion de l'historique.

- productCategory.model.js, productCart.model.js, productOrder.model.js : modèles spécialisés pour gérer les relations entre produits et autres entités via des clés étrangères.

Chaque fichier modèle encapsule les requêtes SQL SELECT, INSERT, UPDATE et DELETE nécessaires à la manipulation des données, tout en utilisant des appels asynchrones avec async/await. Cette séparation entre les couches garantit une meilleure lisibilité du code, une maintenance facilitée et une évolution plus simple du projet.

Dans l'architecture de l'application, les fichiers *.controller.js font office d'intermédiaire entre les routes Express et les modèles. Par exemple, dans product.controller.js, j'importe les méthodes définies dans product.model.js afin de les appeler en réponse aux requêtes HTTP. Le contrôleur reçoit les données du client, effectue les vérifications nécessaires, puis appelle le modèle pour exécuter les requêtes SQL correspondantes (lecture, insertion, mise à jour ou suppression). Ce découplage permet de maintenir un code clair, testable et bien structuré.



Exemple de fonctionnement : modèle product.model.js

Dans product.model.js, j'ai implémenté plusieurs fonctions permettant d'interagir avec la table product. Par exemple :

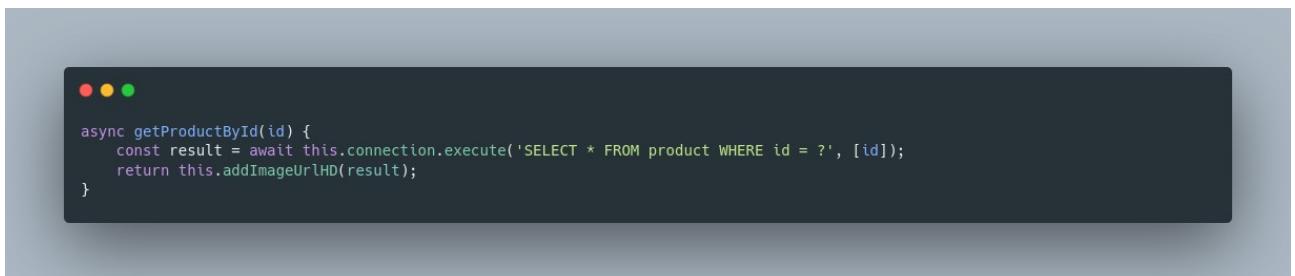
- une fonction pour lire tous les produits, avec une limite dynamique passée en paramètre,
- une autre pour récupérer un produit précis par son ID,
- une fonction pour insérer un ou plusieurs produits via une requête INSERT INTO,
- une fonction de mise à jour partielle qui utilise une boucle sur les champs fournis dans la requête,
- et une fonction de suppression avec vérification de l'existence du produit.

Lire tous les produits (avec limite) – GET /all-products/:limit



```
● ● ●
async getAllProductsActive(limit) {
    const result = await this.connection.execute('SELECT * FROM product WHERE is_archived = FALSE LIMIT ?', [limit]);
    // console.log("result : ", result);
    return this.addImageUrl(result);
}
```

Lire un produit par ID – GET /product/:id



```
● ● ●
async getProductById(id) {
    const result = await this.connection.execute('SELECT * FROM product WHERE id = ?', [id]);
    return this.addImageUrlHD(result);
}
```

Ajouter un ou plusieurs produits – POST /new-product

Lors de l'ajout d'un produit, je ne me contente pas d'insérer directement les données dans la table product. Avant toute insertion, je commence par effectuer plusieurs vérifications afin de garantir

l'intégrité des données. L'une des étapes essentielles consiste à vérifier que la ou les catégories associées au produit existent bien dans la table category. Si une catégorie renseignée n'existe pas, le processus est interrompu ou une erreur est renvoyée.

Ce contrôle préalable est indispensable, car le produit est lié à une ou plusieurs catégories via la table de jointure productCategory, qui impose des contraintes d'intégrité référentielle (FOREIGN KEY). Une tentative d'insertion sans validation entraînerait une erreur SQL.

Une fois les vérifications faites, le produit est inséré dans la table product, puis les associations correspondantes sont créées dans la table productCategory. Cette approche garantit une base cohérente et empêche la création d'enregistrements orphelins ou invalides.



```
● ● ●

async addProduct(newProducts) {
  const categories = await this.connection.execute('SELECT * FROM category');

  if (newProducts.length === undefined) {
    const categoryExists = categories.find(cat => cat.id === newProducts.categoryId);
    if (!categoryExists) {
      throw new Error("Categorie doesn't exist");
    } else {
      if (newProducts.is_new === undefined) {
        await this.connection.execute(
          'INSERT INTO product (name, description, price, imageURL, quantity) VALUES (?, ?, ?, ?, ?)',
          [newProducts.name, newProducts.description, newProducts.price, newProducts.imageURL, newProducts.quantity]
        );
      } else {
        await this.connection.execute(
          'INSERT INTO product (name, description, price, imageURL, quantity, is_new) VALUES (?, ?, ?, ?, ?, ?)',
          [newProducts.name, newProducts.description, newProducts.price, newProducts.imageURL, newProducts.quantity, newProducts.is_new]
        );
      }
    }

    const productCreated = await this.connection.execute('SELECT MAX(id) AS lastId FROM product');
    await this.connection.execute(
      'INSERT INTO productCategory (fk_product, fk_category) VALUES (?, ?)',
      [productCreated[0].lastId, newProducts.categoryId]
    );
  }

  return [newProducts];
} else {
  for (const newProduct of newProducts) {
    const categoryExists = categories.find(cat => cat.id === newProduct.categoryId);
    if (!categoryExists) {
      throw new Error("Categorie doesn't exist");
    }

    if (newProduct.is_new === undefined) {
      await this.connection.execute(
        'INSERT INTO product (name, description, price, imageURL, quantity) VALUES (?, ?, ?, ?, ?)',
        [newProduct.name, newProduct.description, newProduct.price, newProduct.imageURL, newProduct.quantity]
      );
    } else {
      await this.connection.execute(
        'INSERT INTO product (name, description, price, imageURL, quantity, is_new) VALUES (?, ?, ?, ?, ?, ?)',
        [newProduct.name, newProduct.description, newProduct.price, newProduct.imageURL, newProduct.quantity, newProduct.is_new]
      );
    }

    const [productCreated] = await this.connection.execute('SELECT MAX(id) AS lastId FROM product');
    await this.connection.execute(
      'INSERT INTO productCategory (fk_product, fk_category) VALUES (?, ?)',
      [productCreated.lastId, newProduct.categoryId]
    );
  }
}

return newProducts;
}
```

Modifier un produit – PUT /edit-product/:id

Pour modifier un produit, j'ai mis en place une méthode souple qui permet de mettre à jour uniquement les champs nécessaires, sans imposer l'envoi complet de l'objet produit. Lorsqu'une requête PUT est reçue, je commence par vérifier l'existence du produit en base via son identifiant. Si le produit est introuvable, une erreur est retournée.

Ensuite, je parcours dynamiquement les champs transmis dans le corps de la requête. Grâce à une boucle for, chaque champ est mis à jour un par un à l'aide de requêtes SQL de type UPDATE, uniquement s'il est présent dans la requête. Cela permet de modifier un ou plusieurs attributs (comme le nom, la description, le prix, la quantité, etc.) sans écraser les autres valeurs.

Avant chaque mise à jour, des vérifications de type et de validité sont effectuées pour garantir l'intégrité des données. Par exemple, si une nouvelle catégorie est associée, je m'assure qu'elle existe dans la table category avant de modifier la table de jointure productCategory.

Cette approche permet une mise à jour partielle, contrôlée et sécurisée des produits, tout en maintenant la cohérence du modèle relationnel et en limitant les risques d'erreurs.



```
async updateProduct(id, product, editProduct) {
    for await (const [key, value] of Object.entries(editProduct)) {
        if (key in product) {
            this.connection.execute(`UPDATE product SET ${key} = ? WHERE id = ?`, [value, id]);
            // some renvoi true si la valeur existe dans category.id
        } else if (key === "categoryId" && categories.some(cat => cat.id === Number(value))) {
            this.connection.execute('UPDATE productCategory SET fk_category = ? WHERE fk_product = ?', [value, id]);
        } else {
            // On lève une erreur pour que le controller la gère
            const error = new Error(`Wrong param: ${key}`);
            error.statusCode = 400;
            throw error;
        }
    };
}
```

Supprimer un produit – DELETE /product/:id

Pour la suppression d'un produit, j'ai volontairement choisi de ne pas effectuer un DELETE physique dans la base de données. En effet, bien que l'instruction DELETE FROM product WHERE id = ? soit courante dans les opérations CRUD, elle peut entraîner des problèmes d'intégrité lorsqu'il existe des relations entre la table product et d'autres tables comme productCart, productOrder ou productCategory.

Afin de préserver ces relations et d'éviter les erreurs ou pertes de données référentielles, j'ai opté pour une suppression logique : le champ is_archived du produit est mis à TRUE, ce qui me permet d'exclure les produits archivés des affichages classiques tout en conservant les données en base.

Cela facilite également la traçabilité, les restaurations éventuelles, et la gestion historique des commandes ou paniers associés.



Tests fonctionnels

Pour m'assurer du bon fonctionnement des modèles, j'ai réalisé des tests manuels via **Postman**, en interrogeant directement les routes connectées aux méthodes des modèles. Par exemple, j'ai testé l'ajout de produits (`POST /new-product`), la lecture avec limite (`GET /all-products/3`), la suppression d'un produit donné (`DELETE /product/2`) ou encore la création et récupération d'un panier utilisateur.

A screenshot of the Postman application. The left sidebar shows a collection named 'Nam's Workspace' containing various API endpoints such as 'Delete product by id', 'Delete category by id', 'Delete user by id', 'Get all users', 'Get all products', 'Get all products Copy', 'Get all products archive', 'Create new category with token', 'Edit product by id', 'Create new user', 'Connexion user', 'Edit category by id', 'Edit user by id', 'Get product by id', 'Create many products', 'Create one product', and 'New Request'. The main panel displays a specific request for 'Get all products' with a GET method, URL 'http://localhost:4004/all-products/100', and no body. Below the request, there is a response placeholder featuring a cartoon character holding a rocket.

Grâce à cette architecture modulaire et à la mise en place rigoureuse des modèles d'accès aux données, j'ai pu développer une API back-end solide, cohérente et évolutive pour mon mini shop. Chaque modèle (product, user, category, cart, orders...) gère de manière centralisée les interactions SQL avec MariaDB, tout en respectant les contraintes relationnelles définies dans la base.

Les opérations CRUD sont sécurisées par des vérifications en amont, une gestion des erreurs centralisée et une approche soignée, comme l'utilisation de la suppression logique (is_archived) pour préserver les relations entre entités. L'ajout, la modification ou la lecture des données sont traités avec précision, ce qui garantit la fiabilité de l'application, même à mesure que ses fonctionnalités s'élargissent.

Cette structure pose des bases solides pour l'intégration future d'un système complet incluant l'authentification, la gestion des commandes, le suivi des utilisateurs, ou encore le filtrage avancé des produits.

Voici le lien github du projet : <https://github.com/Namoux/minishop>

Activité-type 2

Développer la partie back-end d'une application web ou web mobile sécurisée

CP 7 ▶

Développer des composants métier côté serveur

1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

Dans le cadre du projet MiniShop, j'ai conçu et développé la logique métier côté serveur en adoptant une architecture modulaire basée sur le modèle MVC étendu, avec une séparation claire entre les modèles de données, la logique métier, les routes REST et les middlewares. L'objectif était de fournir une API RESTful robuste, sécurisée et évolutive, tout en respectant les bonnes pratiques Node.js.

Architecture MVC étendue (Models – Controllers – Routes – Middlewares)

Models – Couche de données

Développement de modèles métier dédiés, assurant l'interaction avec la base MariaDB et l'application des règles métier :

- ProductModel : gestion complète du cycle de vie produit (gestion CRUD des produits, filtres par genre/nouveauté, recherche)

```
export class ProductModel {  
    constructor(connection, baseUrl) {  
        this.connection = connection;  
        this.baseUrl = baseUrl;  
    }  
  
    async getAllProductsActive(limit) {  
        const result = await this.connection.execute('SELECT * FROM product WHERE is_archived = FALSE  
LIMIT ?',[limit]);  
        // console.log("result : ", result);  
        return this.addImageUrl(result);  
    }  
}
```

- UserModel : gestion des utilisateurs et authentification, hachage des mots de passe



```

● ● ●

export class UserModel {
    constructor(connection) {
        this.connection = connection;
    }

    async getAllUsersActive(limit) {
        const result = await this.connection.execute('SELECT * FROM user WHERE deleted_at IS NULL LIMIT ?',
        [limit]);
        // console.log("result : ", result);
        return result;
    }
}

```

- CategoryModel : gestion des catégories de produits



```

● ● ●

export class CategoryModel {
    constructor(connection) {
        this.connection = connection;
    }

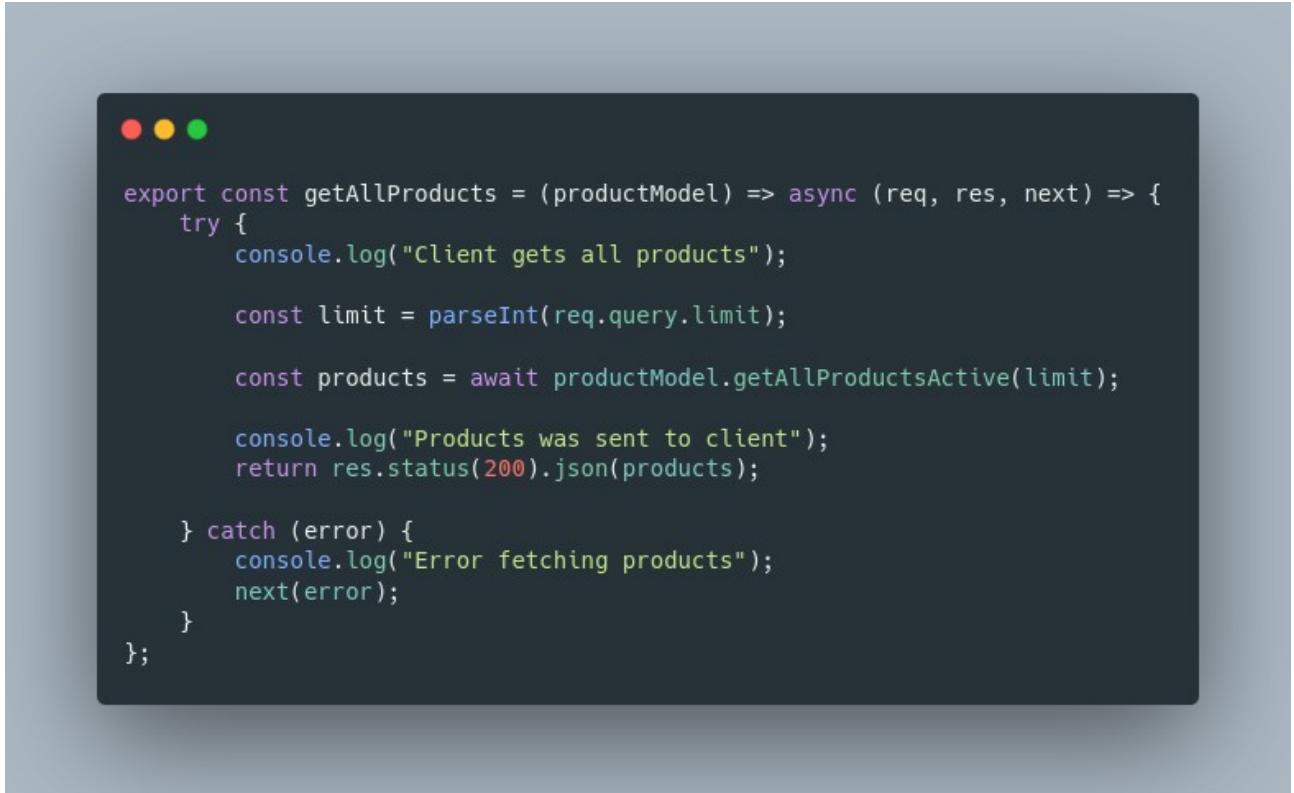
    async addCategory(newCategory) {
        if (newCategory.length === undefined) {
            // Un seul objet
            await this.connection.execute('INSERT INTO category (name) VALUES (?)',
            [newCategory.name]);
        } else {
            // Plusieurs objets
            for (const oneCategory of newCategory) {
                await this.connection.execute('INSERT INTO category (name) VALUES (?)',
                [oneCategory.name]);
            }
        }
        return newCategory;
    }
}

```

Controllers – Logique métier

Organisation de la logique métier dans des contrôleurs dédiés pour chaque ressource :

- productController.js : traitement des opérations sur les produits, génération d'URLs d'images



```
export const getAllProducts = (productModel) => async (req, res, next) => {
  try {
    console.log("Client gets all products");

    const limit = parseInt(req.query.limit);

    const products = await productModel.getAllProductsActive(limit);

    console.log("Products was sent to client");
    return res.status(200).json(products);

  } catch (error) {
    console.log("Error fetching products");
    next(error);
  }
};
```

- userController.js : gestion profil utilisateur

```
export const getAllUsersActive = (userModel) => async (req, res, next) => {
  try {
    console.log("Client gets all users");

    const limit = parseInt(req.query.limit);

    const users = await userModel.getAllUsersActive(limit);

    console.log("Users was sent to client");
    return res.status(200).json(users);

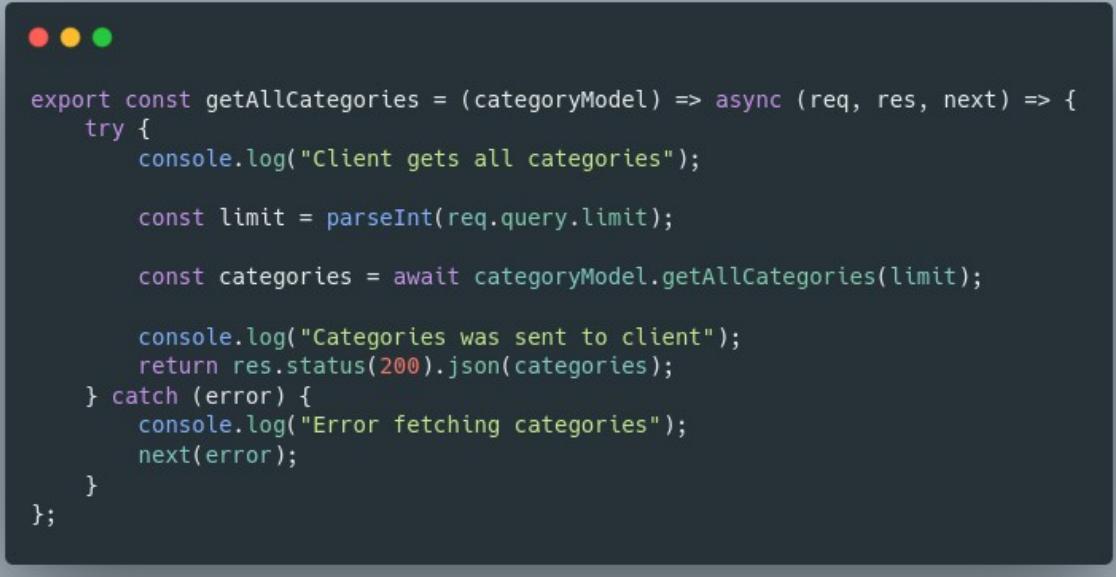
  } catch (error) {
    console.log("Error fetching active users",);
    next(error);
  }
};
```

- authController.js : connexion, inscription, validation des JWT

```
import jwt from "jsonwebtoken";
import bcrypt from "bcrypt";
import { SECRETKEY } from '../config.js';

/**
 * Authentifie un utilisateur et retourne un token JWT
 * @param {UserModel} userModel - Modèle utilisateur
 * @returns {Function} - Middleware Express
 */
export const loginUser = (userModel) => async (req, res, next) => {
```

- categoryController.js : opérations CRUD sur les catégories avec contrôles métier



```

export const getAllCategories = (categoryModel) => async (req, res, next) => {
  try {
    console.log("Client gets all categories");

    const limit = parseInt(req.query.limit);

    const categories = await categoryModel.getAllCategories(limit);

    console.log("Categories was sent to client");
    return res.status(200).json(categories);
  } catch (error) {
    console.log("Error fetching categories");
    next(error);
  }
};

```

Routes – Couche de présentation de l'API

Définition modulaire et claire des endpoints REST en respectant la sémantique HTTP :

- productRoutes.js, userRoutes.js, authRoutes.js, categoryRoutes.js
- Endpoints organisés par ressource : /products, /users, /auth, /categories
- Utilisation appropriée des méthodes : GET, POST, PUT, DELETE

```
// Routes
app.use("/products", productRoutes(connection, baseUrl));
app.use("/users", userRoutes(connection));
app.use("/categories", categoryRoutes(connection));
app.use("/auth", authRoutes(connection));
```

```
const productRoutes = (connection, baseUrl) => {
  const router = express.Router();
  const productModel = new ProductModel(connection, baseUrl);

  router.get('/all', getAllProducts(productModel));
  router.get('/news', getAllNewProducts(productModel));
  router.get('/archived', checkTokenAdmin(connection), getAllArchivedProducts(productModel));
  router.get('/hommes', getProductHomme(productModel));
  router.get('/femmes', getProductFemme(productModel));
  router.get('/search/:query', getSearchProduct(productModel));
  router.get('/:id', getProductById(productModel));
  router.post('/add', checkTokenAdmin(connection), addProduct(productModel));
  router.put('/update/:id', checkTokenAdmin(connection), updateProduct(productModel));
  router.delete('/delete/:id', checkTokenAdmin(connection), deleteProduct(productModel));

  return router;
};
```

Utilisation de middlewares personnalisés pour renforcer la sécurité, la fiabilité et la clarté du code :

- auth.middleware.js :
 - checkTokenValid : vérifie l'authenticité des tokens JWT
 - checkTokenAdmin : restreint l'accès aux routes sensibles aux administrateurs
- error.middleware.js :

- middleware global de gestion des erreurs serveur, avec messages adaptés et propagation contrôlée

```
export const errorHandler = (err, req, res, next) => {
  console.error("⚠️ Error:", err.message);
  return res.status(500).json({ error: err.message || 'Internal server error' });
};
```

```
backend/
├── package.json          # Dépendances et scripts
└── public/
    ├── produits/          # Fichiers statiques (images produits)
    └── vignettes/          # Images haute résolution
src/
├── server.js              # Point d'entrée principal
├── config.js               # Configuration (env variables)
├── database.mjs            # Configuration BDD + création tables
└── controllers/           # Logique métier
    ├── auth.controller.js
    ├── category.controller.js
    ├── product.controller.js
    └── user.controller.js
    └── middlewares/          # Middlewares personnalisés
        ├── auth.middleware.js
        └── error.middleware.js
    └── models/                # Modèles de données
        ├── category.model.js
        ├── product.model.js
        └── user.model.js
    └── routes/                 # Définition des routes API
        ├── auth.routes.js
        ├── category.routes.js
        ├── product.routes.js
        └── user.routes.js
```

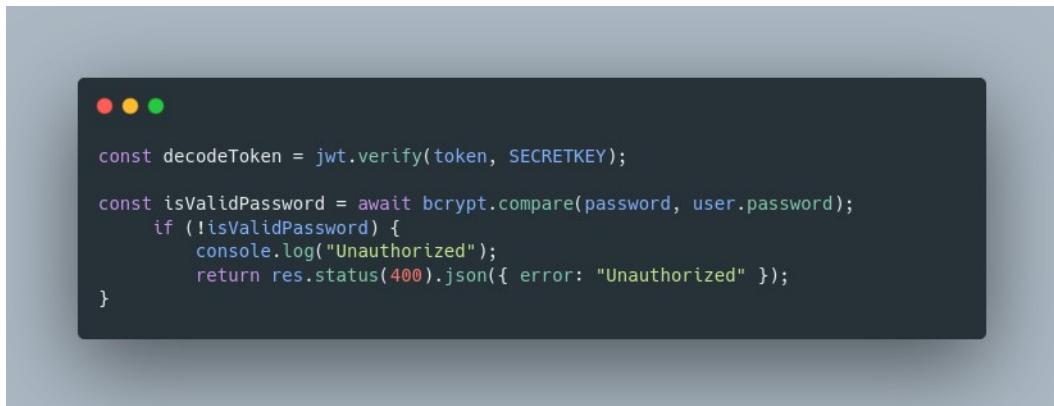
Fonctionnalités métier notables

- **Gestion intelligente des produits :**
 - Soft delete (archivage logique)

- Récupération des nouveautés
- Filtrage par catégorie (Homme/Femme)
- Recherche partielle par nom
- Pagination dynamique pour performance
- **Sécurité et robustesse :**
 - Authentification via JWT
 - Hachage sécurisé des mots de passe avec bcrypt
 - Protection des routes sensibles selon les rôles utilisateur
- **Validation métier intégrée :**
 - Contrôle de la validité des catégories avant création de produit
 - Génération automatique des URLs d'images produit (miniature et HD)

Contexte technique

- **Backend :** Node.js + Express.js avec architecture modulaire
- **Base de données :** MariaDB avec requêtes SQL optimisées
- **Authentification :** JWT + bcrypt



```

● ● ●

const decodeToken = jwt.verify(token, SECRETKEY);

const isValidPassword = await bcrypt.compare(password, user.password);
if (!isValidPassword) {
    console.log("Unauthorized");
    return res.status(400).json({ error: "Unauthorized" });
}

```

- **Sécurité :** Contrôle d'accès basé sur rôles, middlewares dédiés
- **Assets statiques :** gestion dynamique des images via chemins générés



```
addImageUrlHD(products) {
    return products.map(product => ({
        ...product,
        imageURL: `${this.baseUrl}/produits/${product.imageUrl}`
    }));
}
```

- **Gestion des erreurs** avec try/catch



```
try {
    // logique de traitement
} catch (error) {
    res.status(400).json({ msg: "Wrong request" });
}
```

Contraintes techniques

- Respect des bonnes pratiques de développement Node.js (modularisation, code asynchrone).
- Garantie de la cohérence des données et robustesse du serveur.
- Maintenabilité du code en structurant clairement la logique métier.

Résultats obtenus

- API RESTful complète, cohérente et sécurisée
- Gestion fluide du cycle de vie des produits
- Système d'authentification robuste avec rôles (admin ou utilisateurs)
- Architecture claire, scalable et facilement maintenable
- Séparation stricte des responsabilités (models / controllers / routes / middlewares)

Livrables

- Modules Node.js bien structurés et documentés
- Code commenté et conforme aux conventions
- Documentation sommaire des règles métier et des endpoints REST

2. Précisez les moyens utilisés :

Langages et frameworks :

- **JavaScript (Node.js)**
- **Express.js** pour la gestion du serveur et des routes

Outils et logiciels :

- **Visual Studio Code**
- **Postman** pour tests
- **Git** pour versionning

Méthodes :

- Programmation modulaire (séparation routes / contrôleurs / modèles)
- Gestion des erreurs avec `try/catch`
- Validation basique des données en entrée
- Réponses HTTP standardisées

3. Avec qui avez-vous travaillé ?

Jai travaillé seul sur ce projet

4. Contexte

Nom de l'entreprise, organisme ou association ► La plateforme

Chantier, atelier, service ►

Période d'exercice ► *Du : 28/04/2025 au : 20/06/2025*

5. Informations complémentaires (facultatif)

Voici le lien github du projet : <https://github.com/Namoux/miniShop>

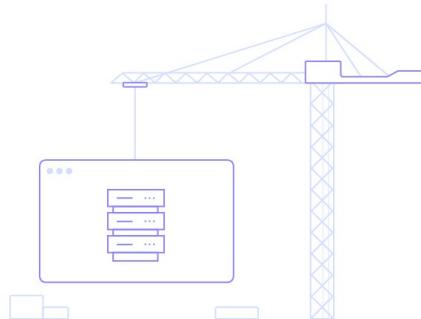
Activité-type 2

Développer la partie back-end d'une application web ou web mobile sécurisée

- CP 8** ▶ Documenter le déploiement d'une application dynamique

1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

Pour le déploiement, j'ai commencé par acquérir un VPS chez Hostinger, ce qui m'a permis d'avoir un serveur dédié pour héberger à la fois le backend et le frontend de mon application.



Configuration de votre VPS

Le processus prendra environ 10 minutes. Vous pouvez [quitter](#) cette page.
Vous recevez un email lorsque votre VPS sera prêt.

4 % : Début de la configuration

J'ai utilisé Docker et Docker Compose pour simplifier la gestion des services et garantir une configuration reproductible. Dans le fichier docker-compose.yml, j'ai défini plusieurs services :

- mariadb pour la base de données, avec les variables d'environnement définies dans un fichier .env.
- backend, un service Node.js exposant le port 4004, uniquement accessible à l'intérieur du réseau Docker.
- frontend, application Angular compilée avec ng build --configuration production, ce qui génère le dossier dist/. Celui-ci est ensuite servi par Nginx, qui joue deux rôles :
 - Servir efficacement les fichiers statiques (cache headers, compression).
 - Gérer le fallback vers index.html pour le routage côté Single Page Application.
 Sa configuration est simplifiée : il ne gère plus l'API, puisque Caddy intercepte déjà les requêtes /api.
- caddy, le serveur web principal, qui s'occupe de gérer les requêtes HTTP/HTTPS et de faire le reverse proxy vers le frontend et le backend selon les chemins.

```

git docker-compose.yml > ...
1  version: '3.9'
   ▷Run All Services
2  services:
   ▷Run Service
3  mariadb:
4    image: mariadb:11
5    container_name: mariadb
6    restart: unless-stopped
7    env_file:
8      - ./.env
9    environment:
10       MYSQL_ROOT_PASSWORD: ${DB_ROOT_PASSWORD}
11       MYSQL_DATABASE: ${DB_NAME}
12       MYSQL_USER: ${DB_USER}
13       MYSQL_PASSWORD: ${DB_PASSWORD}
14    volumes:
15      - mariadb_data:/var/lib/mysql
16    ports:
17      - "${DB_PORT}:${DB_PORT}"
18
   ▷Run Service
19  backend:
20    build: ./backend
21    container_name: backend
22    restart: unless-stopped
23    env_file:
24      - ./.env

```

```

# Étape 1 : Build Angular
FROM node:18 AS build

WORKDIR /app

# Copie package.json et package-lock.json
COPY package*.json ./

# Installe les dépendances
RUN npm install

# Copie le reste du code
COPY . .

# Build Angular en production
RUN npm run build -- --configuration production
# Le -- transmet --configuration production à ng build

# Étape 2 : Serveur Nginx pour l'Angular build
FROM nginx:stable-alpine

```

```

frontend > ⚙ nginx.conf
1  server {
2      listen 80;
3      server_name _;
4
5      root /usr/share/nginx/html;
6      index index.html;
7
8      location / {
9          try_files $uri /index.html;
10     }
11
12 }
13

```

Le fichier Caddyfile m'a permis de configurer Caddy pour :

- Servir le frontend sur toutes les requêtes sauf celles commençant par /api/.
- Rediriger les requêtes /api/* vers le backend.
- Gérer automatiquement les certificats SSL grâce à Let's Encrypt, en sécurisant ainsi toutes les communications via HTTPS.

```
≡ Caddyfile
1
2 # Adresse du site
3 wrapvintage.duckdns.org {
4     encode gzip
5     log
6
7     # Redirige toutes les requêtes /api/* vers le backend
8     handle_path /api/* {
9         |   reverse_proxy backend:4004
10    }
11
12     # Tout le reste → frontend
13     handle {
14         |   reverse_proxy frontend:80
15    }
16 }
17
```

Pour que mon serveur soit accessible depuis Internet avec un domaine dynamique, j'ai utilisé DuckDNS. Cela m'a permis d'associer un nom de domaine (wrapvintage.duckdns.org) à l'adresse IP publique de mon VPS. Grâce à DuckDNS, même si l'IP de mon VPS change, le domaine reste toujours valide.

The screenshot shows the Duck DNS web interface. At the top, there's a yellow rubber duck icon. To its right, the text "Duck DNS" is displayed. Below this, account information is shown: "account" (quoc-nam.ngo@laplateforme.io), "type" (free), and "token" (a long blue-red token). It also shows "token generated" (1 day ago) and "created date" (22 Aug 2025, 17:38:26). The main section is titled "domains 1/5". It lists one domain: "wrapvintage" with "current ip" (blueacted), "ipv6" (blueacted), and "changed" (1 day ago). Buttons for "update ip" and "update ipv6" are present, along with a "delete domain" button.

Enfin, après avoir démarré tous les services avec docker-compose up -d, j'ai pu vérifier que :

- L'API Node.js répond correctement aux requêtes via HTTPS (/api/*).
- Le frontend Angular est accessible via le domaine principal.
- La communication est sécurisée avec des certificats SSL valides.

Cette configuration me permet d'avoir un déploiement complet, sécurisé et maintenable, tout en séparant clairement le frontend, le backend et la base de données.

Contraintes techniques

Le déploiement en ligne a représenté un véritable défi : il fallait configurer un VPS chez Hostinger, gérer plusieurs services via Docker Compose, assurer la communication sécurisée en HTTPS avec Caddy et Let's Encrypt, et mettre en place un domaine dynamique avec DuckDNS. La configuration devait être reproductible, stable et suffisamment robuste pour garantir la disponibilité du site.

Objectifs

L'objectif principal était de rendre l'application accessible en ligne depuis n'importe où, et non plus uniquement en local. Le but était donc de disposer d'un site déployé, sécurisé et accessible par domaine, de manière à pouvoir le présenter, le tester et l'utiliser dans des conditions réelles.

Livrables

Le livrable final est un site web fonctionnel et accessible en ligne via le domaine wrapvintage.duckdns.org, déployé sur un VPS avec Docker et sécurisé en HTTPS.

2. Précisez les moyens utilisés :

Outils et technologies :

- **Node.js / Express.js**
- **Docker/Docker compose**
- **Serveur VPS**
- **Caddy**
- **NGINX**
- **DuckDNS**

Méthodes :

- Documentation progressive du projet au fil de son développement
- Tests réguliers en local pour valider le bon fonctionnement
- Séparation de la configuration dans des fichiers distincts (env, docker, etc.)

3. Avec qui avez-vous travaillé ?

J'ai travaillé seul sur ce projet

4. Contexte

Nom de l'entreprise, organisme ou association ► La plateforme

Chantier, atelier, service ►

Période d'exercice ► *Du : 23/06/2025 au : 01/08/2025*

5. Informations complémentaires (facultatif)

Voici le lien du site : <https://wrapvintage.duckdns.org/>

DOSSIER PROFESSIONNEL (DP)

Titres, diplômes, CQP, attestations de formation

(facultatif)

Intitulé	Autorité ou organisme	Date
Cliquez ici.	Cliquez ici pour taper du texte.	Cliquez ici pour sélectionner une date.

DOSSIER PROFESSIONNEL (DP)

Déclaration sur l'honneur

Je soussigné(e) Quoc-Nam NGO

déclare sur l'honneur que les renseignements fournis dans ce dossier sont exacts et que je suis l'auteur(e) des réalisations jointes.

Fait à Toulon le 25/08/2025

pour faire valoir ce que de droit.

Signature : NGO Quoc-Nam

DOSSIER PROFESSIONNEL (DP)

Documents illustrant la pratique professionnelle

(*facultatif*)

Intitulé

Cliquez ici pour taper du texte.

DOSSIER PROFESSIONNEL (DP)

ANNEXES

(Si le RC le prévoit)