Condat Basket Club

MISSION :

Ce projet consiste à créer un site vitrine pour le Club de Basket de Condat. Il mettra en avant les valeurs du club, ses licenciés et ses produits. Avec ce site vitrine un Dashboard sera développé pour les gérants du club, il permettra la mise à jour du site facilement et rapidement.

Pour cela j'utiliserai le Framework NuxtJS

DURÉE DU PROJET :

- 7 semaines soit 245h

ÉTAPE DU PROJET :

Conception du site Vitrine :

- **Page accueil** :
  - Présente le club et ses partenaires
- **Page club** :
  - Présente les différentes catégorie des membres (masculin / féminin et U11, U13,...)
- **Page shop** :
  - Présente les différents produits vendu par le club utilisant le service Hooper-Store
- **Page contact** :
  - Permet de contacter le mail du Club et de situer le gymnase
- Et d'autres éléments tel que la barre de navigation, le pied de page

Conception du Dashboard :

- **Page d'Information** :
  - Une page d'accueil pour l'administrateur du site
- **Page pour la gestion de l'accueil** :
  - Ajout/Suppression/Modification des partenaires
- **Page pour la gestion du club** :
  - Ajout/Suppression/Modification d'un club
- **Page pour la gestion du shop** :
  - Ajout/Suppression/Modification d'un produit

- **Page pour la gestion des contacts** :
  - Modification du mail de contact
  - Modification de l'adresse du siège social
- Page pour la gestion du profil :
  - Modification pfp, nom, prenom, ...
  - logout

SEMAINE 1 :

Création du projet, mise en place du Framework Nuxtjs, qui utilise VueJS et TypeScript, grâce à la documentation de NuxtJS mais également de ce guide. J'ai choisis pnpm comme manager de paquets et j'ai décidé de ne pas utiliser tailwind ou NuxtUI mais de faire mon propre css pour rester sur des choses simples et personnalisées.

J'ai commencé à me familiariser avec le framework en créant le site vitrine que je vais appeler le "landing". Tout d'abord j'ai créé les dossiers dont j'avais besoins, les dossiers **pages**, **components** et **prisma**.

- Le dossier "**pages**" où se trouve les fichiers qui permettront de créer la vue.
- Le dossier "**components**" où se trouve les composants qui nous permettront d'être insérés dans les vues pour une efficacité et un code plus propre.
- Enfin le dossier "**prisma**" comporte un fichier "**schema.prisma**" qui permet de définir la structure de la base de données et ses relations. Grâce à Prisma, j'ai pu facilement définir des modèles pour ma base de données, comme les utilisateurs, les produits ou les équipes. Une fois le schéma défini, j'ai utilisé Prisma pour générer automatiquement les types **TypeScript** et les fonctions nécessaires pour interagir avec la base de données, ce qui simplifie grandement le développement et réduit les erreurs liées aux types.

Ensuite, j'ai donc commencé par modéliser la base de donnée :

IMAGE DE MODELISATION DE LA BDD A METTRE EN FIN DE STAGE !important

Il ma ensuite fallu créer une BDD, pour des raisons de faciliter j'ai utilisé **docker compose** et créer ma BDD à partir de l'image officielle de **MariaDB**. Pour encore plus de faciliter j'ai créer un script permettant de déployer une base de données MariaDB en conteneur avec une configuration prête à l'emploi.

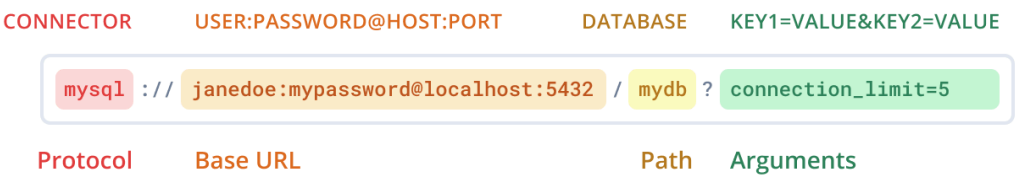
```
services:
   mariadb:
      image: mariadb:latest
      container_name: db-mariadb
      volumes:
```

```
      - mariadb_data:/var/lib/mysql
    ports:
      - "3307:3306"
    environment:
      MYSQL_ROOT_PASSWORD: password
      MYSQL_DATABASE: dbname
      MYSQL_USER: username
      MYSQL_PASSWORD: password

volumes:
  mariadb_data:
```

Pour ensuite lancer ce script en détacher :

```
docker-compose up -d
```

Une fois la BDD créer sur mon VPS Debian j'ai ouvert le port `3306` puis suivant le schéma ci dessous j'ai connecté ma BDD à mon application.



Pour enfin traduire la modélisation de ma bdd en language prisma :

```
generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "mysql"
  url      = env("DATABASE_URL")
}

model User {
  id        Int    @id @default(autoincrement())
  firstName String
  lastName  String
  phone     String
  email     String @unique
  password  String
  token     String @unique @default(cuid())
```

```
  grade     Grade  @default(USER)
}

model Partenaire {
  id    Int    @id @default(autoincrement())
  image String
  name  String
  url   String
}

model Product {
  id             Int              @id @default(autoincrement())
  name           String
  price          Float
  ProductVariant ProductVariant[]
}

model ProductVariant {
  id        Int          @id @default(autoincrement())
  Product   Product      @relation(fields: [productId], references: [id])
  productId Int
  color     ColorVariant
  image     String
  url       String
}

model Categorie {
  id             Int              @id @default(autoincrement())
  name           String
  prixLicence    Float
  CategorieAnnee CategorieAnnee[]
  CategorieImage CategorieImage[]
}

model CategorieAnnee {
  id          Int       @id @default(autoincrement())
  Categorie   Categorie @relation(fields: [categorieId], references: [id])
  categorieId Int
  annee       Int
}

model CategorieImage {
  id          Int       @id @default(autoincrement())
  Categorie   Categorie @relation(fields: [categorieId], references: [id])
  categorieId Int
  url         String
```

```
}

enum Grade {
  ADMIN
  MEMBER
  USER
}

enum ColorVariant {
  RED
  WHITE
  BLACK
}
```

Voici les commandes principale de prisma :

- Synchronise rapidement le schéma Prisma avec la base de données :

```
prisma db push
```

- Gère les migrations de manière versionnée, avec des fichiers traçant les modifications :

```
prisma migrate
```

- Ouvre une interface graphique pour gérer et explorer les données de ta base de données :

```
prisma studio
```

Une fois la BDD faites je me suis lancé dans le développement du landing, j'ai commencé par développé le CSS global en créant un thème dédié au landing. J'ai utilisé des variables CSS pour faciliter la gestion des couleurs, des polices et des styles, assurant ainsi une cohérence visuelle sur l'ensemble du site.

```
* {
  margin: 0;
  padding: 0;
  text-decoration: none;
  list-style: none;
  box-sizing: border-box;
  user-select: none;
  scroll-behavior: smooth;
  outline: none;
```

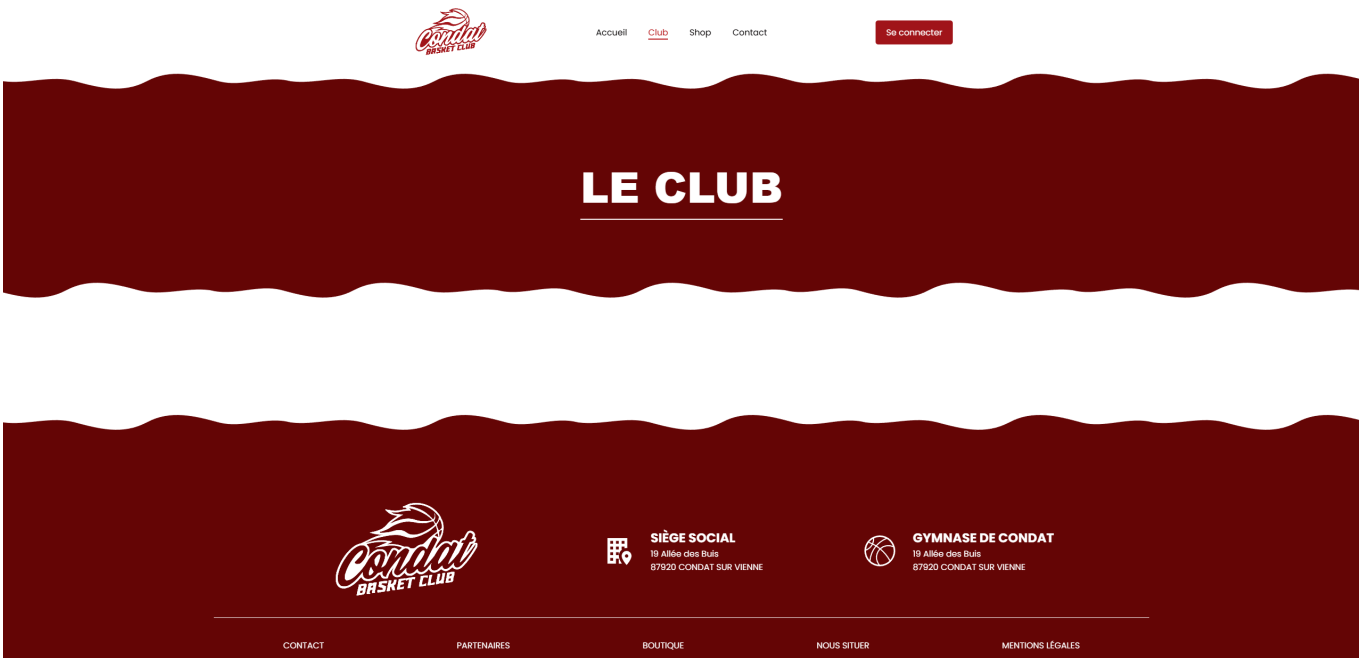
```
  font-family: 'Poppins', sans-serif;
  scrollbar-width: thin;
  scrollbar-color: rgba(255, 255, 255, 0.2) var(--background);
}

body {
  background-color: var(--deepWhite);
}

:root {
  --background: 102, 7, 8;
  --deepRed2: 164, 22, 26;
  --deepRed1: 186, 24, 27;
  --red: 229, 56, 59;
  --black: 11, 9, 10;
  --lightBlack: 22, 26, 29;
  --deepGray: 212, 212, 212;
  --lightGray: 235, 235, 235;
  --deepWhite: 245, 243, 244;
  --white: 255, 255, 255;
  --txt-xs: 0.75rem;
  --txt-sm: 0.875rem;
  --txt-base: 1rem;
  --txt-lg: 1.2rem;
  --txt-xl: 1.325rem;
  --txt-2xl: 1.875rem;
  --txt-3xl: 2.5rem;
  --txt-4xl: 3rem;
  --txt-5xl: 4rem;
  --txt-6xl: 5rem;
}

*::-webkit-scrollbar {
  width: 12px;
}

*::-webkit-scrollbar-track {
  background: var(--background);
  border-radius: 10px;
}

*::-webkit-scrollbar-thumb {
  background-color: rgba(255, 255, 255, 0.2);
  border-radius: 10px;
  border: 3px solid var(--background);
}
```

```css
*::-webkit-scrollbar-thumb:hover {
  background-color: rgba(255, 255, 255, 0.3);
}
```

Une fois cela terminé, j'ai créé deux dossiers dans le dossier "**components**", un dossier "**icons**" où se trouvera toutes mes icones sous format SVG grâce à icones.js. Puis ensuite le dossier "**landing**" où se trouvera touts les composant de mon landing :

```
components/
├── icons/                    # Icônes
│   ├── accueil/              # Icônes liées à l'accueil
│   ├── dashboard/            # Icônes pour le Dashboard
│   ├── footer/               # Icônes spécifiques au footer
│   ├── logoCondatBasketClub.vue  # Logo principal du club
│
├── landing/                  # Composants du site vitrine
│   ├── Footer.vue            # Footer pour le site vitrine
│   ├── NavBar.vue            # Barre de navigation du site vitrine
│   ├── Title.vue             # Titre dynamique des pages
```

J'ai créé trois composants : la barre de navigation, le pied de page et un composant nommé "**Title.vue**" qui permettra d'afficher un header différent suivant la page où nous sommes



J'ai ensuite commencé le développement des pages "accueil.vue", "contact.vue" et "login.vue" :

Voici le contenue de la page d'accueil :

Fondé en [année de création], Condat Basket Club accueille des joueurs de tous âges et tous niveaux. Des équipes juniors aux équipes séniors, en passant par les entraînements loisirs, chacun trouve sa place sur le terrain.

Ne manquez pas nos prochains matchs et événements spéciaux ! Venez encourager nos équipes et partager des moments inoubliables autour de votre passion pour le basket.

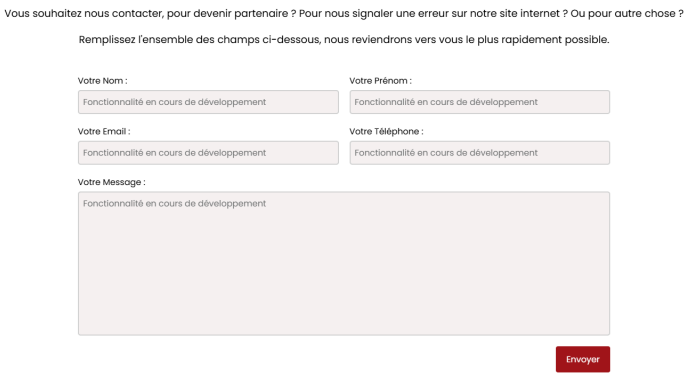
Rejoignez notre communauté dès aujourd'hui ! Inscrivez-vous pour découvrir nos équipes, participer aux entraînements, ou simplement venir nous encourager lors des matchs. Le basket n'est pas juste un sport, c'est une passion qui se vit en équipe.

Un club pour tous, une équipe pour chacun.

NOS PARTENAIRES :

Crédit Agricole · EMS · Garage Auto AD · Intermarché · La Pizza Maestria · Lobo Immo · Lorient · MMDD Peinture

Celui de la page de contact :



Et enfin la page de login :

note : *c'est uniquement du développement statique et non dynamique*

SEMAINE 2 :

J'ai commencé la deuxième semaine avec la création du backend, j'ai créer un dossier "**api**" et "**middleware**" dans le dossier "**server**".
Le dossier “api” est utilisé pour créer des API endpoints côté serveur. Ces endpoints permettent de gérer les requêtes HTTP (GET, POST, PUT, DELETE) et de communiquer avec ma base de donnée via Prisma.

Le dossier “middleware” est utilisé pour stocker les middlewares de mon application. Un middleware est une fonction qui s'exécute avant d'afficher une page et permet d'intercepter les requêtes pour protéger des routes (authentification), rediriger l'utilisateur (si non connecté, rediriger vers /login) et gérer les permissions (accès admin).

J'ai commencé a créer un fichier 01.prisma.ts :

```
import prisma from "~/prisma";

export default defineEventHandler(async (event) => {
    event.context.db = prisma
})
```

Grâce à cette approche, **toutes les requêtes API peuvent accéder à Prisma** sans avoir besoin de l'importer individuellement dans chaque route.

- Il évite d'importer Prisma dans chaque route API séparément.
- Permet de réutiliser une instance unique de Prisma (évite la recréation d'instances multiples).
- Si la configuration de Prisma change, elle ne devra être mise à jour qu’à un seul endroit.
- Toutes les routes API peuvent maintenant accéder à la base de données via `event.context.db`

Ensuite j'ai créer un fichier 02.auth.ts :

```
import { minimatch } from "minimatch";

const authRoute = ["**/dashboard/**/*", "**/dashboard"];

export default defineEventHandler(async (event) => {
    if (!authRoute.some((route) => minimatch(getRequestURL(event).pathname,
route))) return;
    const token = getCookie(event, "token");
    if (!token) return sendRedirect(event, "/login");

    const user = await event.context.db.user.findUnique({
        where: {
            token: token,
        },
    });
    if (!user) {
        deleteCookie(event, "token");
        return sendRedirect(event, "/login");
    }
```

```
    event.context.user = user;
});
```

Le fichier `02.auth.ts` est un **middleware d'authentification**. Son objectif est d'empêcher **les utilisateurs non authentifiés** d'accéder aux pages du Dashboard et de les **rediriger vers la page de connexion** (`/login`).

Une fois cela terminé, j'ai créer deux requêtes, une pour récupérer tout les partenaires de la BDD et une pour récupérer tout les produits et ses variantes.

Requêtes `partner.get.ts` :

```
export default defineEventHandler(async (event) => {
    return await event.context.db.partenaire.findMany({
        orderBy: {
            name: "asc",
        }
    });
});
```

Récupère toute les partenaires de la table `partenaire` et les ranges par ordre alphabétique

Requêtes `shop.get.ts` :

```
export default defineEventHandler(async (event) => {
    return await event.context.db.product.findMany({
        include: {
            ProductVariant: true,
        },
    });
});
```

Récupère toute les articles de la table `product` et ses variantes de la table `ProductVariant`

J'ai donc ensuite affiché dynamiquement mes partenaire comme ceci :

```
<script setup lang="ts">
    const { data: partners } = await useFetch("/api/partner", { server: true
})
</script>
```

Cette ligne utilise `useFetch` (une fonction de NuxtJS 3) pour **récupérer dynamiquement** la liste des partenaires depuis l'API
Résultat :

- `partners` contient un **tableau d'objets** avec les données de chaque partenaire (nom, logo,...)

```
<h1>NOS PARTENAIRES :</h1>
<div class="Icons">
    <a v-for="partner in partners" :key="partner.id" :href="partner.url"
target="_blank">
        <img :src="partner.image" alt="logo partenaire">
        <h3>{{ partner.name }}</h3>
    </a>
</div>
```

Boucle `v-for="partner in partners"` :

- Génère un `<a>` **pour chaque partenaire**.
- `:href="partner.url"` > Redirige vers le site du partenaire.
- `<img :src="partner.image">` > Affiche le **logo** du partenaire.
- `<h3>{{ partner.name }}</h3>` > Affiche **le nom** du partenaire.
  L'API retourne ceci :

```
[
  { "id": 1, "name": "EMS", "image": "https://i.postimg.cc/SKrGtLST/ems.png",
"url": "https://www.ems87.com/" },
  { "id": 2, "name": "La Pizza Maestria", "image":
"https://i.postimg.cc/8zL4LQqV/La-Pizza-Maestria.png", "url":
"https://www.pizzamaestria.fr/" }
 // TOUTE LA TABLE ...
]
```

La boucle génère :

```
<a href="https://www.ems87.com/" target="_blank">
    <img src="https://i.postimg.cc/SKrGtLST/ems.png" alt="logo partenaire">
    <h3>EMS</h3>
</a>
<a href="https://www.pizzamaestria.fr/" target="_blank">
    <img src="https://i.postimg.cc/8zL4LQqV/La-Pizza-Maestria.png" alt="logo
partenaire">
```

```
    <h3>La Pizza Maestria</h3>
</a>
<!-- TOUTE LA TABLE ... -->
```

Ce qui donne :

Pour l'affichage de mes produits cela m'a value plus de travail, je voulais afficher tout mes produits ainsi que leur variantes respectives, j'ai donc procédé ainsi :

```
const { data: articles } = await useFetch("/api/shop", { server: true })
```

- `useFetch` : Récupère la liste des articles depuis l'API `/api/shop`.
- **Option** `{ server: true }` : La requête est exécutée côté serveur avant de rendre la page.

```
const OverArticleID = ref<number | null>(null)
const CustomVariant = ref<Array<{ articleID: number, variantID: number }>>([])
```

- `OverArticleID` : Stocke l'ID de l'article survolé (pour afficher les variantes de couleur).
- `CustomVariant` : Tableau contenant **l'ID de l'article** et **l'ID de la variante sélectionnée**.

```
function preload() {
  if (!articles.value) return alert("No articles found")
  for (const article of articles.value) {
    CustomVariant.value.push({ articleID: article.id, variantID:
article.ProductVariant[0].id })
  }
}
```

- Cette fonction **précharge les variantes par défaut** pour chaque article.
- `ProductVariant[0]` : On sélectionne la première variante disponible.

```
const cacheValue = ref<{ articleID: number, variantID: number } | null>(null)

function setFastVariant(articleID: number, variantID: number) {
  const index = CustomVariant.value.findIndex((variant) => variant.articleID
=== articleID)
  cacheValue.value = { articleID: articleID, variantID:
```

```
CustomVariant.value[index].variantID }
  CustomVariant.value[index].variantID = variantID
}
```

- **Sauvegarde l'ancienne valeur** dans `cacheValue`.
- **Change temporairement** la variante affichée (utilisée lors du survol).

```
function setVariant(articleID: number, variantID: number) {
  cacheValue.value = null
  const index = CustomVariant.value.findIndex((variant) => variant.articleID
=== articleID)
  CustomVariant.value[index].variantID = variantID
}
```

- Efface la valeur en cache.
- Définit la nouvelle variante sélectionnée.

```
function restoreCache() {
  if (!cacheValue.value) return
  const index = CustomVariant.value.findIndex((variant) => variant.articleID
=== cacheValue.value?.articleID)
  CustomVariant.value[index].variantID = cacheValue.value.variantID
  cacheValue.value = null
}
```

- Si l'utilisateur **ne clique pas sur une variante**, on **remet l'ancienne** au lieu de la nouvelle temporaire.

```
function setOverArticle(articleID: number) {
  if (articles.value?.find((article) => article.id ===
articleID)?.ProductVariant.length === 1) return
  OverArticleID.value = articleID
}
```

- **Vérifie** si l'article a plusieurs variantes.
- **Si oui**, affiche les variantes **au survol**.

```
<div class="box" @mouseover="setOverArticle(article.id)"
@mouseleave="OverArticleID = null" v-for="article in articles"
:key="article.id">
```

```
...
</div>
```

- **Boucle** `v-for` : Parcourt la liste des articles.
- **Événements** `@mouseover` **et** `@mouseleave` :
  - Active l'affichage des variantes au survol.
  - Cache l'affichage des variantes quand la souris part.

```
<a :href="article.ProductVariant.find((variant) => variant.id ===
CustomVariant.find((variant) => variant.articleID ===
article.id)?.variantID)?.url" target="_blank">
   <img :src="article.ProductVariant.find((variant) => variant.id ===
CustomVariant.find((variant) => variant.articleID ===
article.id)?.variantID)?.image" alt=""/>
</a>
```

- **Trouve l’image et le lien** correspondant à la variante sélectionnée pour chaque article.

```
<div class="info">
    <div class="description" v-show="OverArticleID !== article.id">
        <h3>{{ article.name }}</h3>
        <h4>{{ article.ProductVariant.length }} couleur(s)</h4>
        <h1>{{ article.price }}€</h1>
    </div>
```

- **Affiche le nom, le prix et le nombre de variantes.**
- **Cache ces informations** lorsqu’on survole l’article (`v-show="OverArticleID !== article.id"`).

```
<div class="colorVariants" v-show="OverArticleID === article.id">
   <div>
      <img :src="variant.image" alt="" v-for="variant in
article.ProductVariant" :key="variant.id"
         @mouseover="setFastVariant(article.id, variant.id)"
         @mouseleave="restoreCache()"
         @click="setVariant(article.id, variant.id)" />
   </div>
</div>
```

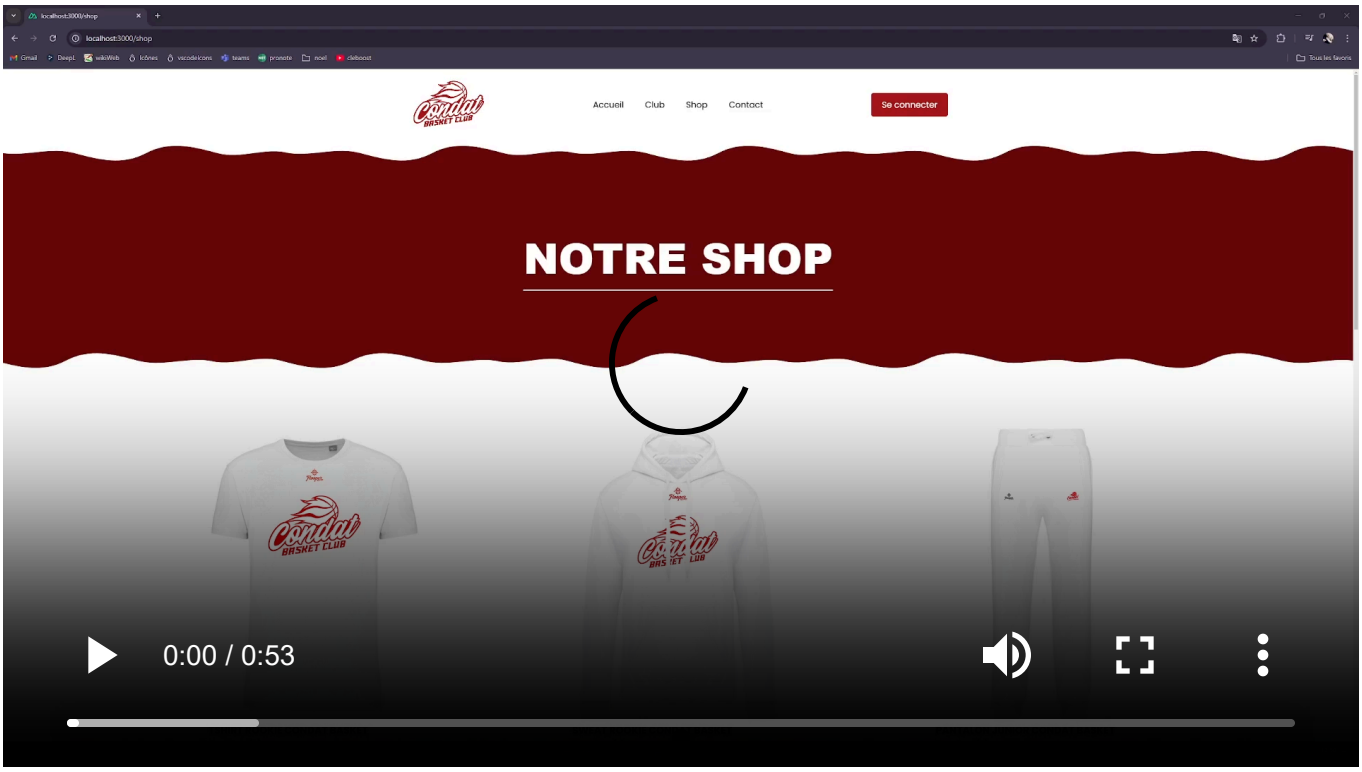
- **Affiche les images des variantes** au survol de l’article.
- **Événements :**

- `@mouseover` → Change temporairement l'image.
- `@mouseleave` → Restaure l'image précédente.
- `@click` → Sélectionne définitivement la variante.

Ce code permet d'afficher dynamiquement les articles du shop, avec une gestion avancée des variantes de produits :

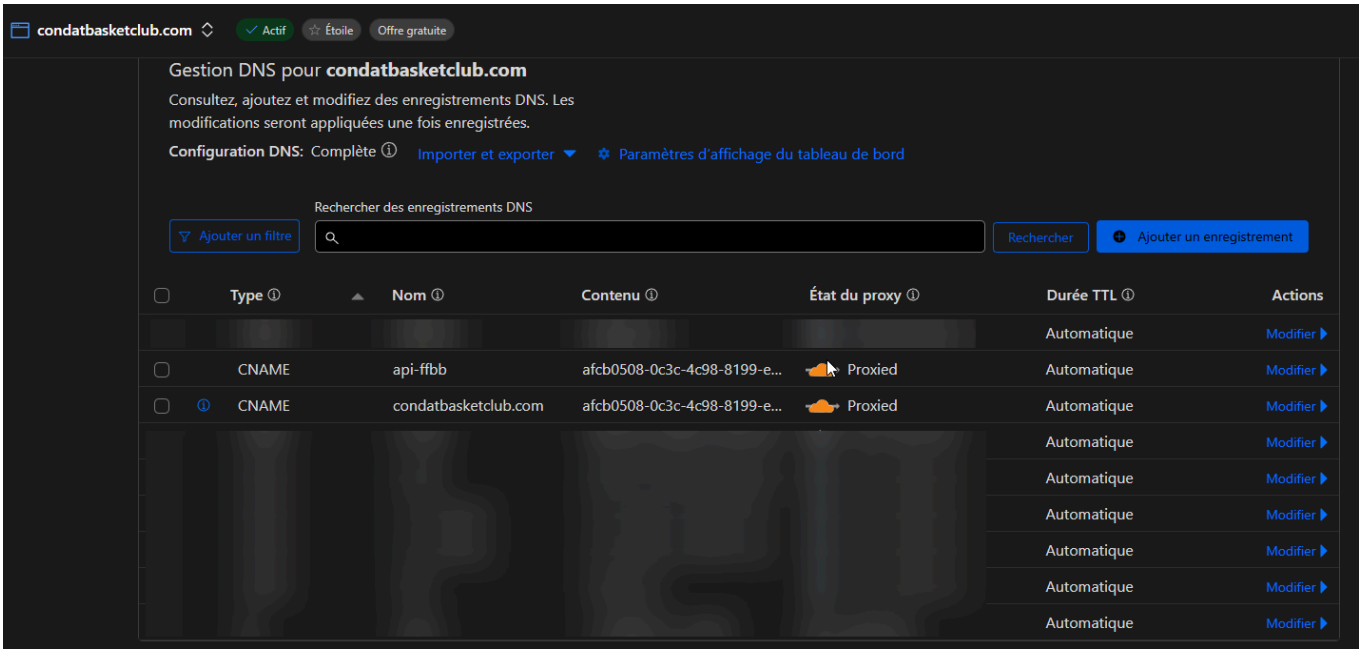
- **Chargement dynamique des données depuis l'API**
- **Affichage des variantes de couleur au survol**
- Sélection définitive d'une variante au click
- **Système de cache pour éviter les erreurs d'affichage**

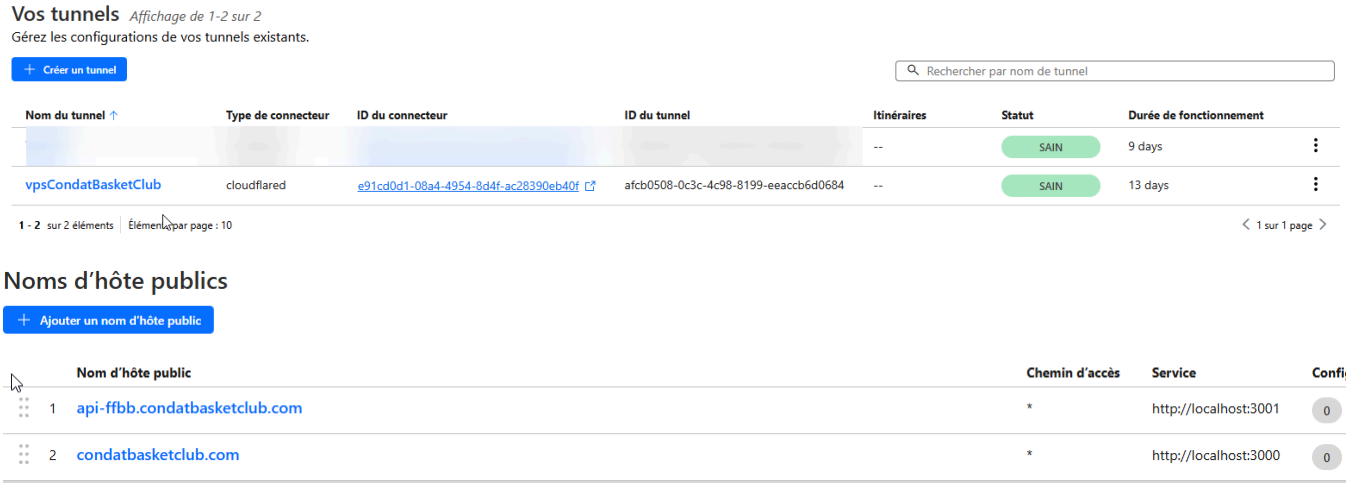
Ce qui donne :



Cloudflare

Nous utilisons un VPS pour héberger le site web. Pour que le site reste sécurisé, nous utilisons une solution de protection nommée Cloudflare. Cloudflare gère les enregistrements DNS de notre nom de domaine, ce qui nous permet de ne pas dévoiler l'IP du VPS et de faire du "proxy". Le proxy permet de faire passer tout le trafic par Cloudflare qui bloque les attaques DDoS ou autres avant de nous renvoyer seulement les requêtes légitimes à notre VPS. Pour cela, nous avons configuré un tunnel Argo qui permet de faire le lien entre notre VPS et Cloudflare. Il nous permet aussi de bloquer l'accès direct à l'IP car tous les ports sont fermés (excepté le 22 pour le SSH).





Semaine 3 :

J'ai commencé la troisième semaine avec la création du dossier "**layout**", c'est un dossier permettant de définir des structures globales réutilisables pour différentes pages du site. Il me permet de séparer mon **Dashboard** de mon **Landing**.

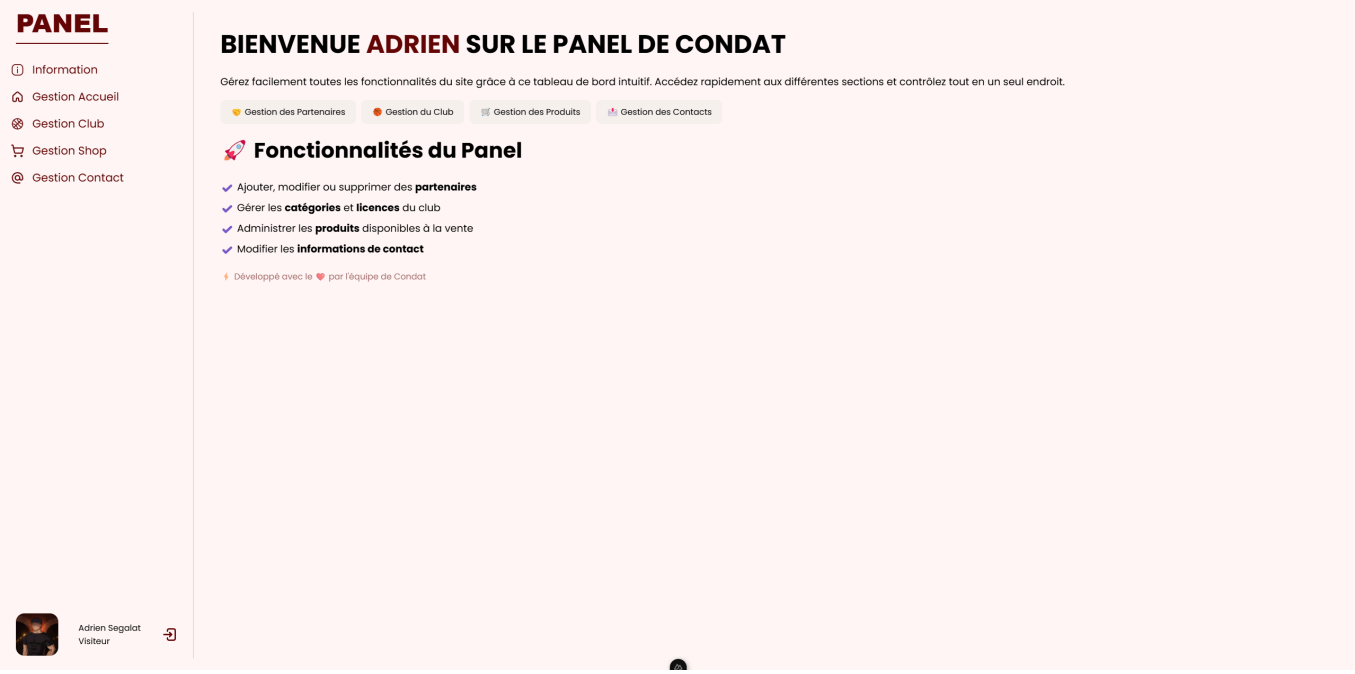
Une fois cela fait, le dossier "**pages**" est structuré ainsi :

```
pages/          # Pages du site vitrine
├── index.vue          # Page d'accueil du site
├── club.vue           # Page dédiée au club
├── shop.vue           # Page de la boutique
├── contact.vue        # Page de contact
├── login.vue          # Page de connexion
│
├── dashboard/             # Pages du panneau d'administration
│   ├── index.vue          # Page d'accueil du dashboard
```

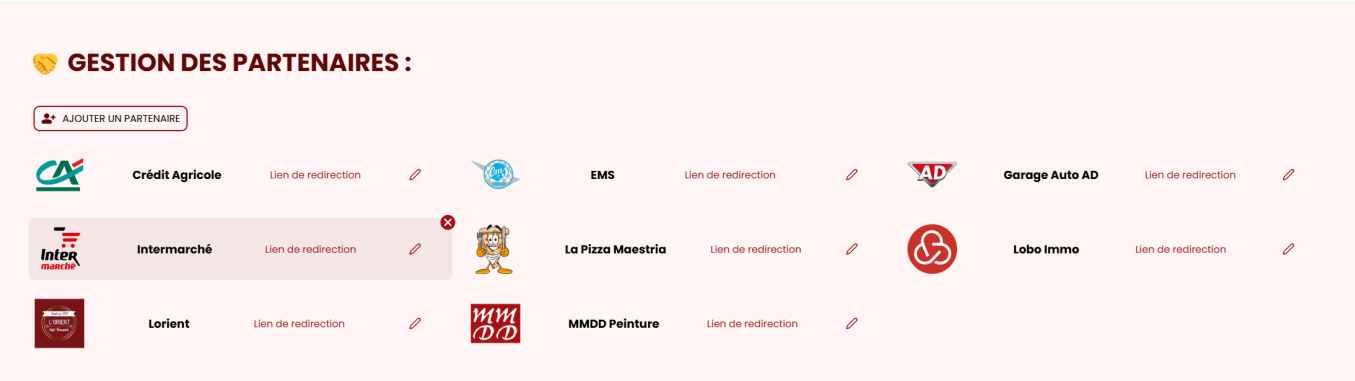
```
|   ├── club.vue          # Gestion du club
|   ├── shop.vue          # Gestion de la boutique
|   ├── contact.vue       # Gestion des contacts
|   ├── accueil.vue       # Gestion des partenaires
```

J'ai donc pu commencé le design du Dashboard :

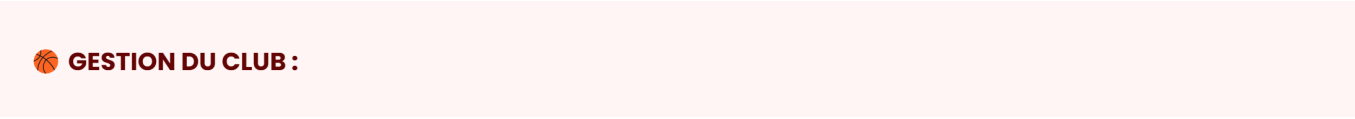
Voici la NavBar ainsi que la page index.vue :



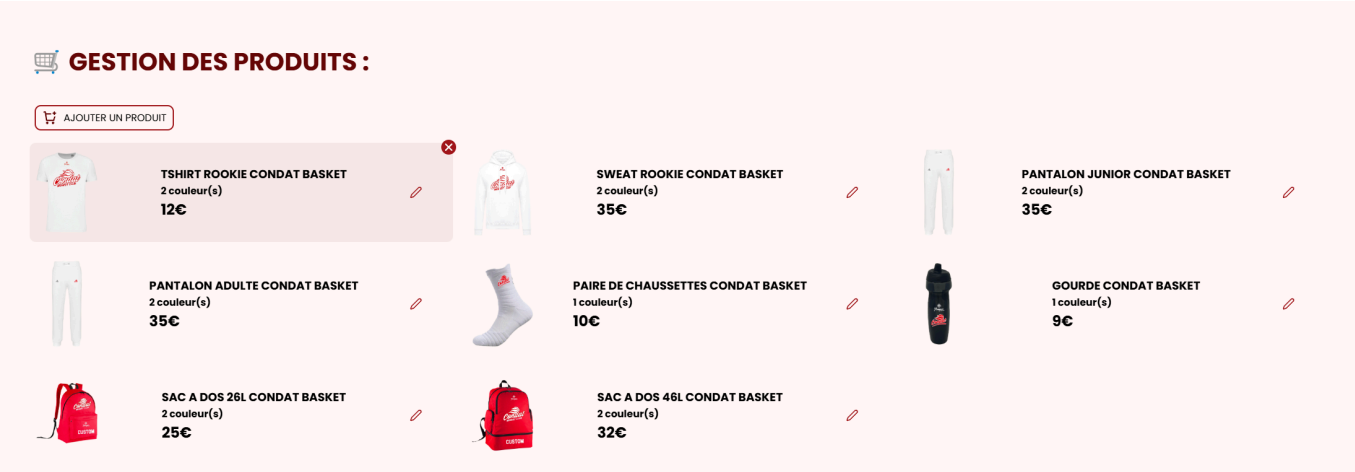
Voici la gestion de la page "**index.vue**" du landing :



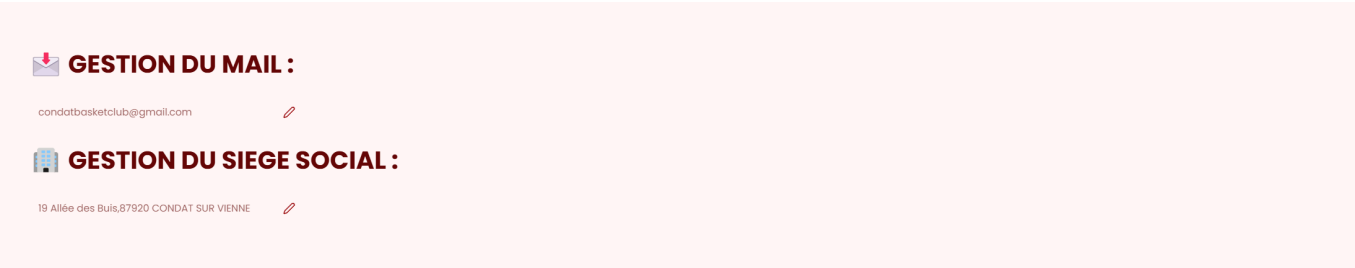
Voici la gestion de la page "**club.vue**" du landing : *(je n'ai pas encore décidé de quoi faire exactement dans cette section, c'est pour cela qu'elle est vide)*



Voici la gestion de la page "**shop.vue**" du landing :



Voici la gestion de la page "**contact.vue**" et du component "**Footer.vue**" du landing :



note : *c'est uniquement du développement statique et non dynamique*

J'ai ensuite commencé le backend du système de **login** et **logout** :

Pour le système de login le but est de vérifier l'identité de l'utilisateur et lui retourner un **token d'accès**.

Partie 1 : API d'Authentification (`login.post.ts`) :

```
export default defineEventHandler(async (event) => {
    const { email, password } = await readBody(event);
```

- `readBody(event)` : Récupère les données envoyées par le frontend.
- L'**email** et le **mot de passe** sont extraits du corps de la requête.

```
const regexMail = /^[a-zA-Z0-9._-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,6}$/;
if (!email || email === "" || !regexMail.test(email)) {
    throw createError({
        status: 400,
        statusMessage: "Bad Request",
        message: "Email is required",
        data: { email: email || "" },
    });
}
```

```
if (!password || password === "") {
    throw createError({
        status: 400,
        statusMessage: "Bad Request",
        message: "Password is required",
        data: { password: "" },
    });
}
```

- **Validation de l'email** avec une expression régulière (`regexMail`).
- **Si l'email ou le mot de passe est vide**, une erreur `400` est renvoyée.

```
const hashPassword = createHash("sha256").update(password).digest("hex");
const user = await event.context.db.user.findUnique({
    where: {
        email: email,
        password: hashPassword,
    },
    select: {
        token: true,
    },
});
```

- **Le mot de passe est converti en hash SHA-256** pour éviter de stocker des mots de passe en clair.
- **Recherche d'un utilisateur correspondant (email + mot de passe haché) en base de données**.

```
if (!user) {
    throw createError({
        status: 401,
        statusMessage: "Unauthorized",
        message: "Invalid email or password",
        data: { email: email, password: password },
    });
}
```

- **Si aucun utilisateur n'est trouvé, une erreur 401 est renvoyée** (identifiants invalides).

```
return {
    status: 200,
    statusMessage: "Login Success",
```

```
    data: { token: user.token },
};
```

- **Si tout est bon, l'API retourne un token** qui sera utilisé pour l'authentification dans les requêtes suivantes.

Partie 2 : Frontend (Page de Connexion)

```
const email = ref<String>('')
const password = ref<String>('')
```

- `email` et `password` **sont des variables réactives** utilisées dans les champs de saisie.

```
function login() {
  const notification = push.promise("Identification ...")
```

- **Affiche une notification** indiquant que la connexion est en cours.

```
if (email.value === "" || password.value === "")
   return notification.error(password.value === "" ? "Mot de passe requis" :
"Nom d'utilisateur requis")
```

- Vérifie si l'**email ou le mot de passe est vide**, et affiche une **erreur** si c'est le cas.

```
return $fetch('/api/login', {
  method: 'POST',
  body: JSON.stringify({ email: email.value, password: password.value })
})
```

- **Envoie une requête POST à** `/api/login` avec l'email et le mot de passe.

```
.then((reponse: any) => {
  if (!reponse?.data?.token) return notification.error("Server error")
  document.cookie = `token=${reponse.data.token}`
  document.location.href = "/dashboard"
})
.catch(error => {
  notification.error("Nom d'utilisateur ou mot de passe invalide")
})
```

- **Si le serveur répond avec un token** :

    - **Stocke le token dans un cookie**.
    - **Redirige l'utilisateur vers le tableau de bord (** `/dashboard` **)**.
- **Si une erreur survient** :
    - Affiche un message `"Nom d'utilisateur ou mot de passe invalide"`.

```
<template>
  <form>
    <input type="text" v-model="email" placeholder="Votre nom d'utilisateur"
/>
    <input type="password" v-model="password" placeholder="Votre mot de passe"
/>
    <p>Nouvel utilisateur ? <router-link to="/">Inscrivez-vous</router-link>
</p>
    <button type="submit" @click="login" onclick="return false">Se
connecter</button>
  </form>
</template>
```

- `v-model="email"` **et** `v-model="password"` > Lient les inputs aux variables réactives.
- **Bouton de connexion** :
    - `@click="login"` → Exécute la fonction `login()`.
    - `onclick="return false"` → Empêche le rechargement de la page.

Pour le système de logout le but est de supprimer le **token d'accès** sauvegardé dans les **cookies**.

```
function logout() {
  document.cookie = "token=; expires=Thu, 01 Jan 1970 00:00:00 UTC; path=/;";
  return useRouter().push("/");
}
```

- **Efface le cookie contenant le token de session** en lui attribuant une **date d'expiration passée** (`01 Jan 1970`).
- **Le chemin (** `path=/` **)** garantit que le cookie est supprimé pour l'ensemble du site.
- **Utilise** `useRouter()` **de Vue Router** pour rediriger immédiatement l'utilisateur vers la **page d'accueil (** `/` **)** après la déconnexion.

```
<IconsDashboardLogout @click="logout()" id="logout" />
```

- **Ajoute un bouton de déconnexion** avec l'icône `IconsDashboardLogout`.
- **Déclenche** `logout()` **au clic** (`@click="logout()"`), ce qui :

- **Supprime le token**
- **Redirige l’utilisateur vers la page d’accueil**

Pour finir la semaine j'ai créer un système **d'auto deploy** à l'aide de

Pour finir la semaine j'ai créer un système pour automatiser le déploiement du site, un workflow **GitHub Actions** a été mis en place. Ce workflow permet de **déployer automatiquement** le projet sur un serveur distant via **SSH** et **PM2**, garantissant ainsi une mise à jour rapide et efficace après chaque modification du code.

Le fichier de configuration du workflow est nommé `deploy.yml` et se trouve dans `.github/workflows/`

```yaml
name: Basket Landing Deploy
on:
   workflow_dispatch:

jobs:
   deploy:
      runs-on: ubuntu-latest
      concurrency:
         group: ${{ github.ref }}
         cancel-in-progress: true

      steps:
         - name: Checkout deployment branch
           uses: actions/checkout@v4
           with:
              ref: main

         - name: Set up Node
           uses: actions/setup-node@v4
           with:
              node-version: 22.x

         - name: Install Pnpm
           run: npm install -g pnpm

         - name: Install Dependencies
           run: pnpm i
           shell: bash

         - name: Generate DB
           run: npx prisma generate
           shell: bash
```

```
      - name: Build
        env:
          GH_TOKEN: ${{ secrets.GITHUB_TOKEN }}
        run: npm run build
        shell: bash

      - name: PM2 Config
        run: |
          cp ecosystem.config.cjs .output/
        shell: bash

      - name: Deploy to Server
        env:
          SSH_KEY: ${{ secrets.SSH_PRIVATE_KEY }}
          SSH_IP: ${{ secrets.REMOTE_IP }}
          SSH_PORT: 22
          SSH_USER: ${{ secrets.REMOTE_USER }}
        run: |
          echo "$SSH_KEY" > ssh_key
          chmod 600 ssh_key
          ssh -o StrictHostKeyChecking=no -i ssh_key -p $SSH_PORT
$SSH_USER@$SSH_IP "rm -rf /root/landing/*"
          scp -i ssh_key -P $SSH_PORT -r .output/*
$SSH_USER@$SSH_IP:/root/landing
          ssh -o StrictHostKeyChecking=no -i ssh_key -p $SSH_PORT
$SSH_USER@$SSH_IP "cd /root/landing && pm2 restart all && pm2 save"
          rm ssh_key
        shell: bash
```

```
on:
  workflow_dispatch:
```

- Le workflow se déclenche **manuellement** via l'interface GitHub Actions grâce à `workflow_dispatch`.
- Il est donc **exécuté à la demande** et non à chaque push.

```
jobs:
  deploy:
    runs-on: ubuntu-latest
```

- Le job de déploiement s'exécute sur **une machine virtuelle Ubuntu**.

```yaml
concurrency:
   group: ${{ github.ref }}
   cancel-in-progress: true
```

- **Empêche les conflits** en annulant un déploiement en cours si un autre est lancé simultanément.

```yaml
- name: Checkout deployment branch
  uses: actions/checkout@v4
  with:
     ref: main
```

- Récupère le code source depuis la branche **main**.

```yaml
- name: Set up Node
  uses: actions/setup-node@v4
  with:
     node-version: 22.x
```

- Configure **Node.js v22** pour l'exécution du projet.

```yaml
- name: Install Pnpm
  run: npm install -g pnpm
```

- Installe **pnpm**, le gestionnaire de paquets utilisé.

```yaml
- name: Install Dependencies
  run: pnpm i
  shell: bash
```

- Installe toutes les dépendances du projet.

```yaml
- name: Generate DB
  run: npx prisma generate
  shell: bash
```

- Génère les **types Prisma** et configure la base de données.

```yaml
- name: Build
  env:
```

```
      GH_TOKEN: ${{ secrets.GITHUB_TOKEN }}
  run: npm run build
  shell: bash
```

- Compile le projet **NuxtJS** en production.

```
- name: PM2 Config
  run: |
     cp ecosystem.config.cjs .output/
  shell: bash
```

- Copie la configuration **PM2** pour gérer le processus du serveur.

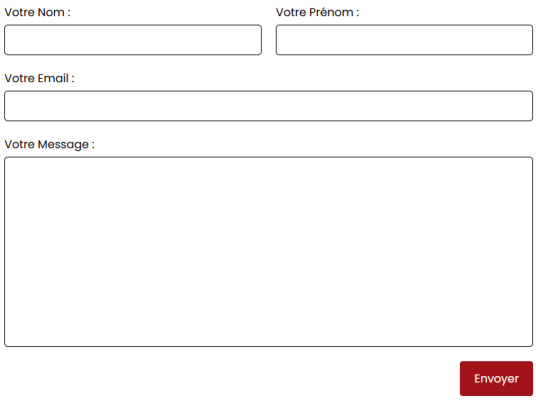
```
- name: Deploy to Server
  env:
     SSH_KEY: ${{ secrets.SSH_PRIVATE_KEY }}
     SSH_IP: ${{ secrets.REMOTE_IP }}
     SSH_PORT: 22
     SSH_USER: ${{ secrets.REMOTE_USER }}
  run: |
     echo "$SSH_KEY" > ssh_key
     chmod 600 ssh_key
     ssh -o StrictHostKeyChecking=no -i ssh_key -p $SSH_PORT $SSH_USER@$SSH_IP
"rm -rf /root/landing/*"
     scp -i ssh_key -P $SSH_PORT -r .output/* $SSH_USER@$SSH_IP:/root/landing
     ssh -o StrictHostKeyChecking=no -i ssh_key -p $SSH_PORT $SSH_USER@$SSH_IP
"cd /root/landing && pm2 restart all && pm2 save"
     rm ssh_key
  shell: bash
```

- **Connexion SSH sécurisée** grâce aux **clés privées GitHub Secrets**.
- **Suppression des anciens fichiers** avant le déploiement pour éviter tout conflit.
- **Transfert des nouveaux fichiers** via `scp`.
- **Redémarrage du serveur** avec **PM2**, garantissant que la dernière version est bien en ligne.

Semaine 4 :

📬 Formulaire de contact

Ce code met en place un **formulaire de contact** permettant à un utilisateur d'envoyer un message au club. Le message est ensuite envoyé par mail via l'API SendGrid côté serveur.




```
const nom = ref<string>('')        // Nom de l'utilisateur
const prenom = ref<string>('')     // Prénom
const email = ref<string>('')      // Adresse email
const message = ref<string>('')    // Message écrit
```

Les champs sont liés avec `v-model` aux inputs du formulaire, ce qui permet une synchronisation automatique entre les données et les inputs HTML.

```
function sendMsg() {
  const notification = push.promise('Envoie ...')
  if (nom.value === '' || prenom.value === '' || email.value === '' ||
message.value === '')
    return notification.error('Tous les champs sont requis')
```

Avant d'envoyer, on vérifie que **tous les champs sont remplis**. Si ce n'est pas le cas, une **notification d'erreur** est affichée.

```
return $fetch('/api/sendMsg', {
  method: 'POST',
  body: JSON.stringify({ nom, prenom, email, message }),
})
```

Si tous les champs sont valides, une **requête POST** est envoyée à l'API `/api/sendMsg` pour transférer les données. Des notifications s'affichent selon le succès ou l'échec.

```
<button type="submit" class="btnSubmit" @click="sendMsg" onclick="return
false">Envoyer</button>
```

Utilisation de `@click="sendMsg"` avec `onclick="return false"` pour empêcher le rechargement de la page.

```
export default defineEventHandler(async (event) => {
  const { nom, prenom, email, message } = await readBody(event)
```

Le corps de la requête est lu et destructuré pour obtenir les données envoyées.

```
if (!nom || nom === '') {
  throw createError({ status: 400, message: 'LastName is required' })
}
```

Si un champ est manquant, l'API renvoie une erreur 400 (Bad Request) avec un message clair.

```
const emailContent = `
  <div>
    <h1>📩 Nouveau Message</h1>
    <h3>🔎 Expéditeur : ${prenom} ${nom}</h3>
    <h3>📧 Email : ${email}</h3>
    <p>${message}</p>
  </div>
`
```

Un message HTML est généré dynamiquement à partir des données envoyées par l'utilisateur.

```
await event.context.sg.send({
  from: 'new-email@condatbasketclub.com',
  to: (await event.context.db.contact.findFirst())?.mail,
  subject: 'Vous avez un nouveau message...',
  html: emailContent,
})
```

L'email est ensuite envoyé à l'adresse récupérée dans la base de données via `event.context.db.contact.findFirst()`.
Si une erreur se produit avec SendGrid, elle est affichée dans la console (`console.error`).

Ensuite j'ai développé le backend de la localisation du gymnase :

J'ai simplement utilisé l'API de Google Maps en générant une API key :

```
const { onLoaded } = useScriptGoogleMaps({
  apiKey: 'AIzaSyB6ax4qq770wQAJJ1PKYxdPGNg2sflzy_w',
})

const map = ref()

onMounted(() => {
  onLoaded(async (instance) => {
    const maps = (await instance.maps) as any as typeof google.maps
    new maps.Map(map.value, {
      center: { lat: 45.791125, lng: 1.23236 },
      zoom: 18,
    })
  })
})
```

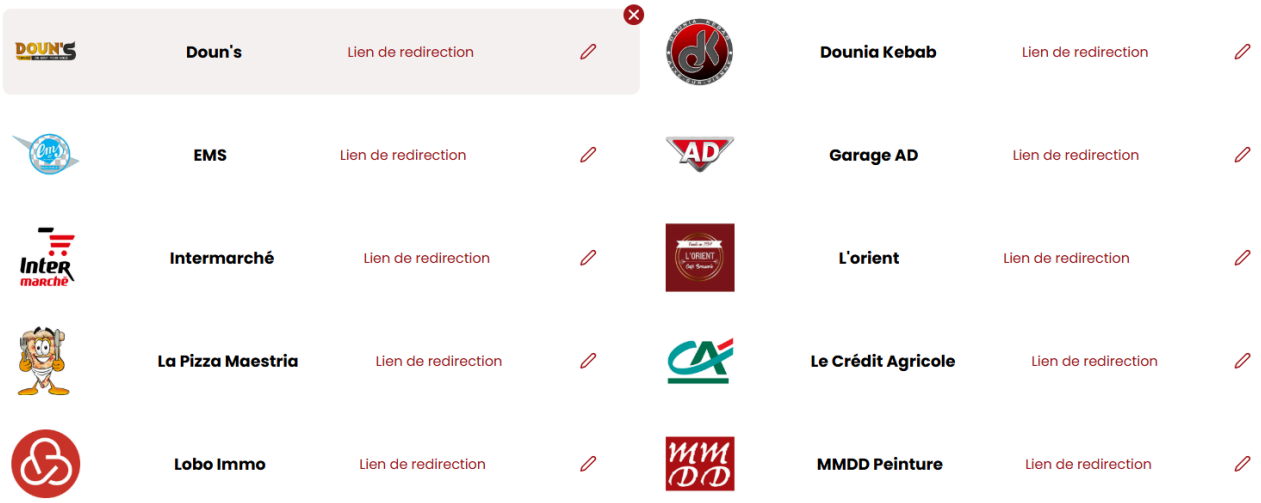
La latitude et la longitude sont celle du Gymnase

```
<template>
  <div ref="map" class="map"></div>
</template>
```

Une fois cela fait j'ai débuté le front du panel, plus précisément la gestion des partenaire et de la boutique :

Dans la section `<template>`, on retrouve :

- Un bouton pour ajouter un partenaire.
- Une grille affichant les partenaires.
- Des icônes pour modifier ou supprimer.
- Le composant modal `DashboardModalsModalPartners` s'affiche lors de l'ajout ou de l'édition.




```
const { data: partners, refresh } = await useFetch('/api/partner', { server:
true })
```

On récupère la liste des partenaires via l'API `/api/partner`. Le composable `useFetch` permet de faire une requête dès le rendu serveur. `refresh()` permet de recharger les données après une modification ou suppression.

```
async function deletePartner(id: number) {
  const notification = push.promise('Suppression')
  return $fetch(`/api/dashboard/partner/${id}`, {
    method: 'DELETE',
  })
```

Cette fonction envoie une requête `DELETE` à l'API pour supprimer un partenaire selon son `id`. Elle affiche une notification selon le résultat de l'opération.

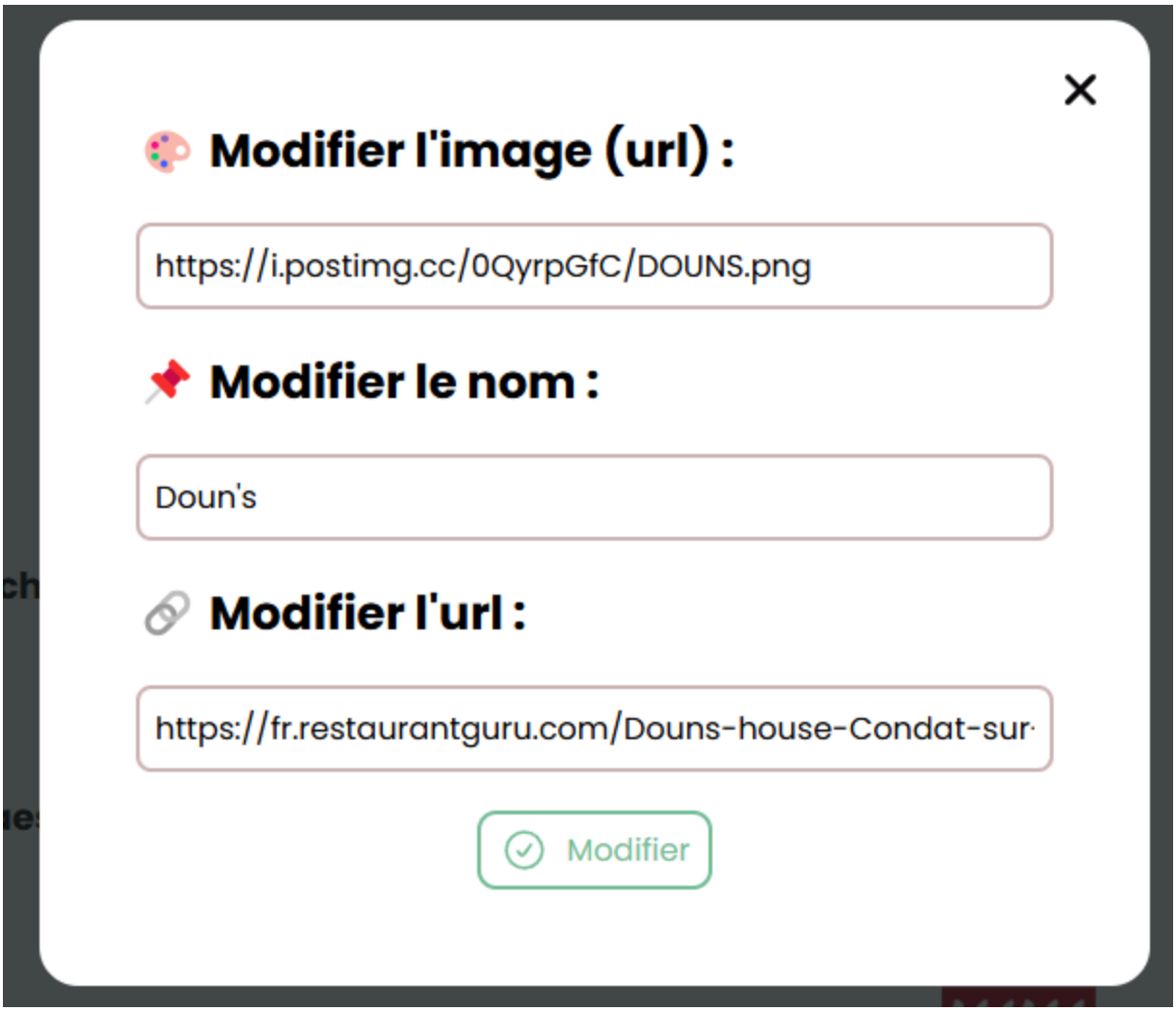
```
const showModal = ref<boolean>(false)
const ePartner = ref<Partner | false>(false)
```

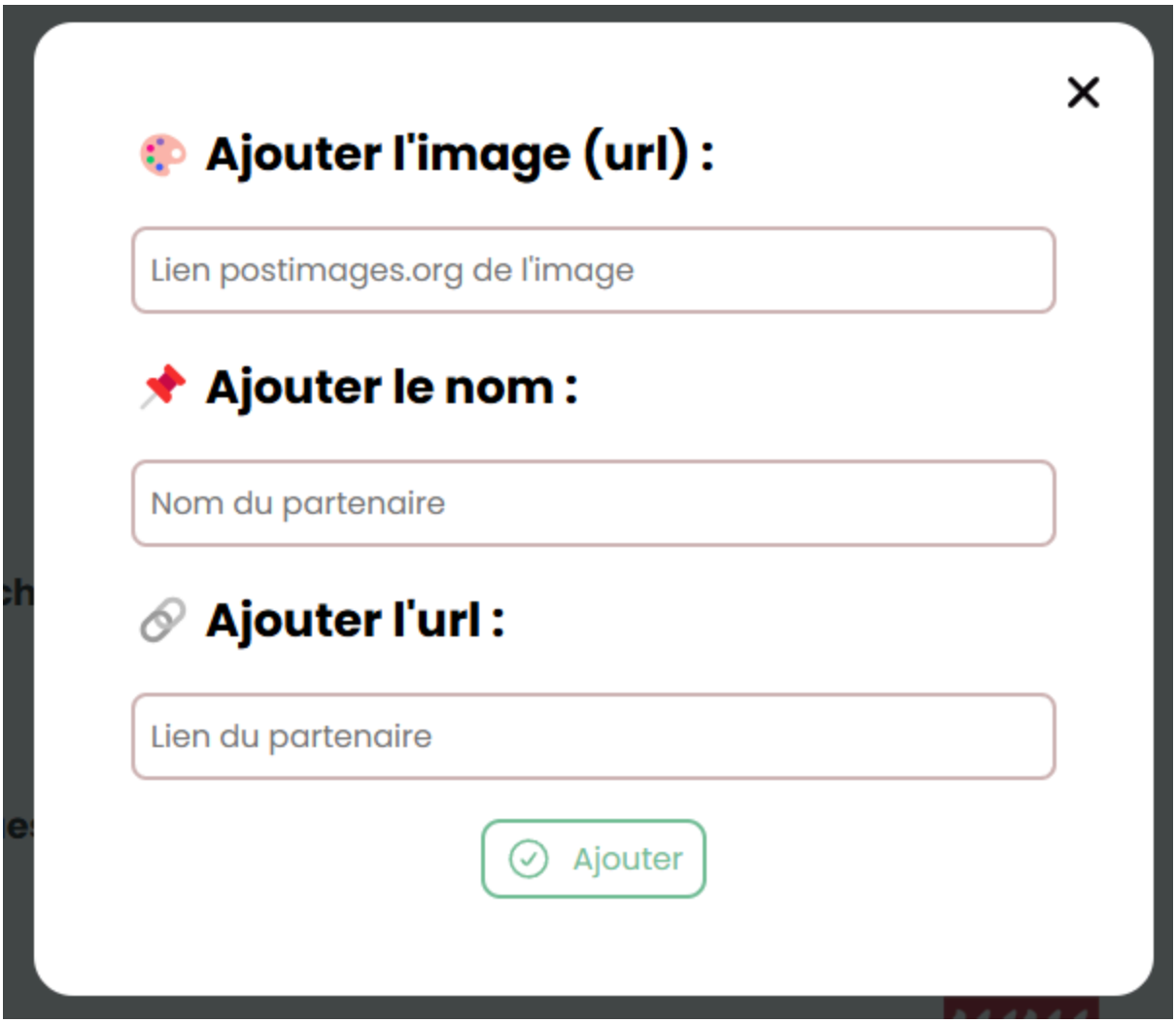
`showModal` contrôle l'affichage du modal. `ePartner` contient les données du partenaire en cours d'édition.

```
function editPartner(id: number) {
  ePartner.value = partners.value?.find((partner: Partner) => partner.id ===
id) || false
  showModal.value = true
}
```

Cette fonction ouvre le modal et pré-remplit les champs avec les infos du partenaire sélectionné.

Composant Modal (Ajout/Modification) :


Modifier l'image (url) :
https://i.postimg.cc/0QyrpGfC/DOUNS.png
Modifier le nom :
Doun's
Modifier l'url :
https://fr.restaurantguru.com/Douns-house-Condat-sur
Modifier




```
const props = defineProps<{ edit: Partner | false }>()
const emit = defineEmits(['close'])
const image = ref<string>(props.edit ? props.edit.image : '')
```

Ce composant prend en prop `edit` (vrai si on édite, faux si on ajoute). Les champs sont remplis automatiquement si `edit` est vrai.

```
await $fetch(`/api/dashboard/partner${props.edit ? `/${props.edit.id}` : ''}`,
{
  method: props.edit ? 'PUT' : 'POST',
})
```

Envoie une requête `POST` (ajout) ou `PUT` (modification) à l'API. Affiche une notification selon le succès ou l'erreur.

Semaine 5 :

Durant la semaine 5 j'ai créer la section "club", on peut y trouver les photos du club, les entrainements des différentes catégories, les membres du bureau et les documents à télécharger.

Voici les 4 vue correspondante du Landing :

- Les photos :

- Les entraînements :

  “Pasted image 20250407173234.png” ne peut être trouvé.

- Les membres du bureau

  “Pasted image 20250407173253.png” ne peut être trouvé.

- Les différents documents :

  “Pasted image 20250407173321.png” ne peut être trouvé.

Et voici leur arborescence dans le projet :

```
pages/          # Pages du site vitrine
├── ...
├── club/                   # Section club
│   ├── photos/
│   │   ├── [tID].vue       # Page des photos d'un catégorie
│   │   ├── index.vue       # Page des photos
│   ├── entrainement.vue    # Page des horaires des entrainements
│   ├── bureau.vue          # Page des membres du club
│   ├── licence.vue         # Page des documents
├── ...
```

Une fois le front end finit, j'ai débuter le backend. J'ai donc créer 3 routes dans le dossier api à la racine, une pour les photos qui est enfaite un dossier, une pour les entrainements et une pour les membres du bureau.

Voici ma requête pour les entrainements :

```
export default defineEventHandler(async (event) => {
  return await event.context.db.entrainement
    .findMany({
      orderBy: {
        name: 'asc',
      },
      include: {
        Horaires: true,
        _count: {
          select: {
            Photo: true,
          },
        },
      },
    })
    .catch((e: any) => {
      console.log(e)
      throw createError({
        statusCode: 500,
        message: 'DatabaseError',
        statusMessage:
          'An error occurred while retrieving the training. Please try again
later. If the problem persists, please contact the administrator.',
      })
    })
})
```

Je les récupères tous `findMany` puis les range par ordre alphabétique, je fais une jointure pour récupérer également les horaires et le nombre de photos par catégorie.

Ensuite je récupère mes entraînements dans ma vue :

```
const { data: trainings, error } = await useFetch('/api/training', {
  server: true,
})
```

Pour ensuite afficher les catégories avec leur entrainement(s) correspondant :

```
<div class="horaires-card" v-if="!error" v-for="training in trainings"
:key="training.id">
        <h2>{{ training.name }}</h2>
        <div class="line"></div>
        <div>
          <p class="horaires-item" v-for="horraire in training.Horaires"
```

```
:key="training.id">
            <span>{{ horraire.jour }}</span>
            <span>{{ horraire.heure }}</span>
          </p>
        </div>
      </div>
```

J'effectue a quelque précisions près la même chose pour les photos et les membres du bureau.

Une fois le landing en place je me lance dans la conception de la partie club du dashboard, pour cela je commence à créer les routes dans l'api :

```
dashboard/
├── club/
│   ├── photos/
│   │   ├── index.post.ts   # Ajouter une photo
│   │   ├── index.get.ts    # Obtenir les photos
│   │   ├── trains.get.ts   # Obtenir les entrainements
│   │   ├── [id].delete.ts  # Supprimer une photos
│   │   ├── [id].put.ts     # Modifier une photos
│   ├── training/
│   │   ├── index.post.ts   # Ajouter un horraire d'entrainement
│   │   ├── [id].put.ts     # Modifier un horraire d'entrainement
│   ├── bureau/
│   │   ├── index.post.ts   # Ajouter un membre
│   │   ├── [id].delete.ts  # Supprimer un membre
│   │   ├── [id].put.ts     # Modifier un membre
```

Je ne veux pas supprimer un entrainement car les catégorie sont les même, c'est universelle mais je souhaite seulement modifier leur horaires et pouvoir en ajouter.

Les requêtes sont du même style que celles d'avant, prenons l'exemple de bureau :

`index.post.ts` :

```
export default defineEventHandler(async (event) => {
  const { nom, prenom, role } = await readBody(event)
  if (!nom || nom === '') {
    throw createError({
      status: 400,
      statusMessage: 'Bad Request',
      message: 'Name is required',
      data: { nom: nom || '' },
    })
  }
```

```
  if (!prenom || prenom === '') {
    throw createError({
      status: 400,
      statusMessage: 'Bad Request',
      message: 'Prenom is required',
      data: { prenom: prenom || '' },
    })
  }
  if (!role || role === '') {
    throw createError({
      status: 400,
      statusMessage: 'Bad Request',
      message: 'Role is required',
      data: { role: role || '' },
    })
  }

  await event.context.db.bureau.create({
    data: {
      nom: nom,
      prenom: prenom,
      role: role,
    },
  })

  return {
    status: 204,
    statusMessage: 'Created',
  }
})
```

`[id].delete.ts` :

```
export default defineEventHandler(async (event) => {
  const param = getRouterParam(event, 'id')
  if (!param) {
    throw createError({
      status: 400,
      statusMessage: 'Bad Request',
      message: 'Id is required',
    })
  }

  const rex = new RegExp('^[0-9]+$')
  if (!rex.test(param)) {
```

```
    throw createError({
      status: 400,
      statusMessage: 'Bad Request',
      message: 'Id must be a number',
    })
  }
  const id = parseInt(param)
  if ((await event.context.db.bureau.count({ where: { id: id } })) === 0) {
    throw createError({
      status: 404,
      statusMessage: 'Not Found',
      message: 'Member not found',
    })
  }

  await event.context.db.bureau.delete({
    where: {
      id: id,
    },
  })

  return {
    status: 204,
    statusMessage: 'No Content',
  }
})
```

`[id].put.ts` :

```
export default defineEventHandler(async (event) => {
  const param = getRouterParam(event, 'id')
  if (!param) {
    throw createError({
      status: 400,
      statusMessage: 'Bad Request',
      message: 'Id is required',
    })
  }

  const rex = new RegExp('^[0-9]+$')
  if (!rex.test(param)) {
    throw createError({
      status: 400,
      statusMessage: 'Bad Request',
      message: 'Id must be a number',
```

```
  })
}
const id = parseInt(param)
if ((await event.context.db.bureau.count({ where: { id: id } })) === 0) {
  throw createError({
    status: 404,
    statusMessage: 'Not Found',
    message: 'Member not found',
  })
}

const { nom, prenom, role } = await readBody(event)
if (!nom || nom === '') {
  throw createError({
    status: 400,
    statusMessage: 'Bad Request',
    message: 'Name is required',
    data: { nom: nom || '' },
  })
}
if (!prenom || prenom === '') {
  throw createError({
    status: 400,
    statusMessage: 'Bad Request',
    message: 'Prenom is required',
    data: { prenom: prenom || '' },
  })
}
if (!role || role === '') {
  throw createError({
    status: 400,
    statusMessage: 'Bad Request',
    message: 'Role is required',
    data: { role: role || '' },
  })
}

await event.context.db.bureau.update({
  where: { id: id },
  data: { nom: nom, prenom: prenom, role: role },
})

return {
  status: 204,
  statusMessage: 'No Content',
```

```
    }
})
```

- Une fois cela fais je commence le dashboard :

“Pasted image 20250407180730.png” ne peut être trouvé.

Puis je créer des components modal pour chaque besoins (photos, entrainement et bureau).

- Pour Photos :

“Pasted image 20250407180857.png” ne peut être trouvé.

- Pour Entrainement :

“Pasted image 20250407180927.png” ne peut être trouvé.

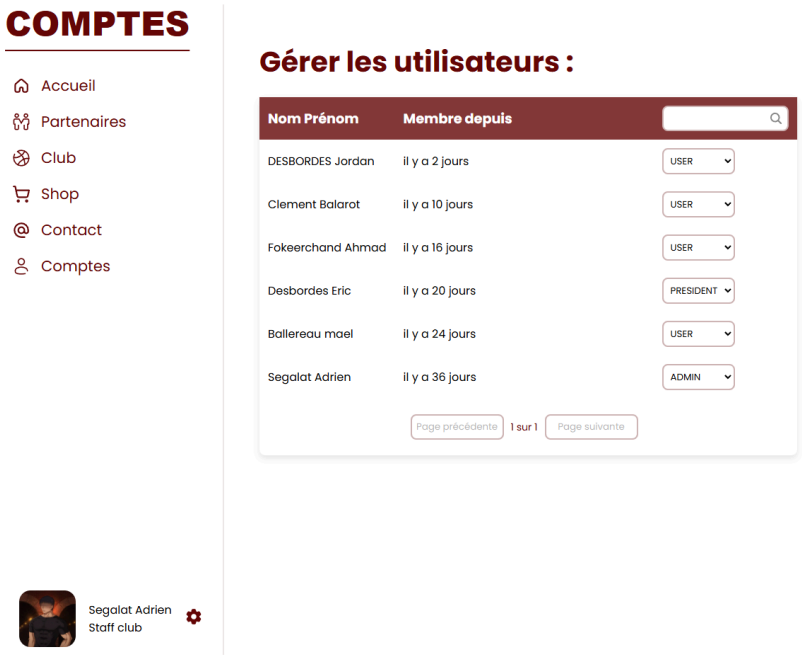
- Pour les membres :

“Pasted image 20250407180948.png” ne peut être trouvé.

Une fois les modal fais je m'appuis sur le shop pour faire fonctionner tout cela !

Semaine 6 :

L'avant dernière semaine j'ai commencé la partie gestion utilisateurs :



Voici la requête GET me permettant de récupérer les utilisateurs par date de création de compte.

```
export default defineEventHandler(async (event) => {
  const { name } = getQuery(event)

  if (name && typeof name == 'string') {
    return await event.context.db.user.findMany({
      select: {
        createdAt: true,
        grade: true,
        id: true,
        firstName: true,
        email: true,
        lastName: true,
      },
      where: {
        OR: [
          {
            firstName: {
              contains: name,
              search: 'insensitive',
            },
          },
          {
            lastName: {
              contains: name,
              search: 'insensitive',
            },
          },
        ],
      },
    })
  }
  return await event.context.db.user.findMany({
    orderBy: {
      createdAt: 'desc',
    },
    select: {
      createdAt: true,
      grade: true,
      id: true,
      firstName: true,
      email: true,
      lastName: true,
    },
    take: 20,
```

```
  })
})
```

Voici la fonction me permettant d'organiser la gestion des rôles :

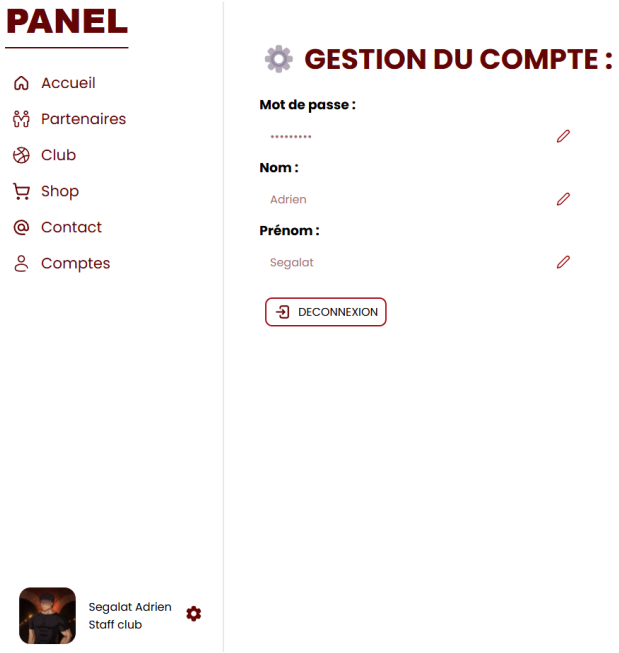
```
import { Grade } from '@prisma/client'

export default defineEventHandler(async (event) => {
  if (event.context.user.grade !== Grade.ADMIN && event.context.user.grade !==
Grade.PRESIDENT)
    throw createError({ status: 403 })

  const { grade } = await readBody(event)
  const userIDString = getRouterParam(event, 'id') as string | undefined
  const userID = parseInt(userIDString || '', 10)
  if (!userID) throw createError({ status: 400, message: 'id is required' })
  if (!grade) throw createError({ status: 400, message: 'grade is required' })
  if (grade !== 'USER' && grade !== 'MEMBER' && grade !== 'ADMIN' && grade !==
'PRESIDENT')
    throw createError({ status: 400, message: 'Invalid grade' })
  if (grade === 'MEMBER' && event.context.user.grade !== Grade.PRESIDENT &&
event.context.user.grade !== Grade.ADMIN)
    throw createError({
      status: 403,
      message: "Vous n'avez pas les droits pour changer un grade en MEMBER",
    })
  if (grade === 'ADMIN' && event.context.user.grade !== Grade.PRESIDENT)
    throw createError({
      status: 403,
      message: "Vous n'avez pas les droits pour changer un grade en ADMIN",
    })
  if (grade === 'PRESIDENT' && event.context.user.grade !== Grade.PRESIDENT)
    throw createError({
      status: 403,
      message: "Vous n'avez pas les droits pour changer un grade en
PRESIDENT",
    })
  if (userID === event.context.user.id)
    throw createError({ status: 403, message: 'Vous ne pouvez pas changer
votre propre grade' })
  if ((await event.context.db.user.count({ where: { id: userID } })) === 0)
    throw createError({ status: 404, message: 'User not found' })
  await event.context.db.user.update({
    where: { id: userID },
    data: { grade },
```

```
  })
  return { status: 204 }
})
```

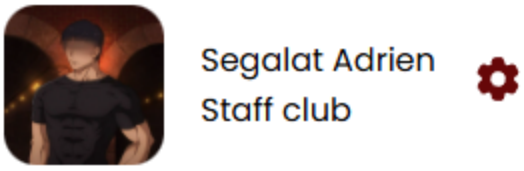
```
.../                    # backend
├── [id]/               # dossier ID
│   ├── grade.patch.ts  # gestion des permission pour changer le rôle
├── index.get.ts        # requete GET des utilisateurs
```

J'ai également finis la partie gestion profile :
Il est possible de :

- changer son mot de passe
- changer son Nom
- changer son Prénom



Pour récupérer le profil :



Voici la requête me permettant de récupérer seulement les informations d'on j'ai besoins dans la table utilisateur, je récupère donc : le prénom, le nom, l'email et le rôle de l'utilisateur puis je

traduit le rôle récupérer dans la bdd (ADMIN, PRESIDENT, ...) en nom de rôle compréhensible :

- Visiteur (USER)
- Membre du club (MEMBER)
- Staff du club (ADMIN)
- Président (PRESIDENT)

```
const tradRole: { [key: string]: string } = {
  USER: 'Visiteur',
  MEMBER: 'Membre du club',
  ADMIN: 'Staff club',
  PRESIDENT: 'Président',
}

export default defineEventHandler(async (event) => {
  const user = event.context.user
  return {
    firstName: user.firstName,
    lastName: user.lastName,
    email: user.email,
    role: tradRole[user.grade],
  }
})
```

J'ai créer fichier TypeScript pour récupérer le mot de passe, le prénom et le nom. Voici un exemple pour récupérer le mot de passe :

```
import { createHash } from 'crypto'

export default defineEventHandler(async (event) => {
    const {password} = await readBody(event)
    if (!password) {
        throw createError({
            status: 400,
            statusMessage: 'Bad Request',
            message: 'password is required',
        })
    }

    const hashPassword = createHash('sha256').update(password).digest('hex')

    await event.context.db.user.update({
        where: {
            id: event.context.user.id,
```

```
        },
        data: {
            password: hashPassword,
        },
    })
    return { message: 'Password updated successfully' }
})
```

J'importe la fonctione `createHash()` de la librairie `crypto` pour ensuite pouvoir hash le nouveau mot de passe.

Je fais de même pour le nom et prénom, voici donc l'arborescence :

```
.../          # backend
├── account/             # Dossier account du dashboard
│   ├── firstname.put.ts # Requete pour changer le prenom
│   ├── lastname.put.ts  # Requete pour changer le nom
│   ├── password.put.ts  # Requete pour changer le mot de passe
│   ├── me.get.ts        # Requete pour obtenir les informations de
l'utilisateur
```

Semaine 7 :

API FFBB

J'ai voulu intégrer sur le site les résultats des match en direct de la saison. Il sont publié sur le site officiel de la fédération française de basket. Le problème est que ce site ne propose aucun moyens d'obtenir les résultat de manière automatique via une api ou autre. J'ai donc eu l'idée de faire un robot, externe au site qui viendrais automatiquement une fois par semaine ouvrir un navigateur web, naviguer sur le site puis télécharger les pages html. Apres cela, un script pourrait traiter l'html pour extraire les données et les retranscrire en json pour les utiliser sur le site.

Pour commencer j'ai fais un serveur express qui gère les demandes du site auprès du script pour récupéré les données

```
import express from "express";
import { API, Classement, Match, Pool } from "./types";
import * as cheerio from "cheerio";
import puppeteer, { Browser } from "puppeteer";
import { championats } from "./data";
import cors from "cors";
import schedule from "node-schedule";
```

```
const app = express();
app.use(cors());
let cache: API | null = null;

app.get("/", (_req, res) => {
    if (!cache) {
        res.json({ error: "No data available" });
        return;
    }
    res.json(cache);
    return;
});

app.listen(3001, () => {
    console.log("Server is running on port 3001");
});
```

Ce script renvoie la valeur de `cache` a chaque demande sur l'enpoint de l'api `https://api-ffbb.condatbasketclub.com` . Le cache permet de renvoyé les données qui sont récupéré indépendamment par une fonction toute les semaines.

Pour obtenir les pages, j'utilise le module `puppeteer` qui permet d'ouvrir chrome en mode automatisé et d'effectué des actions sur le navigateur (dans mon cas aller sur la page ffbb et télécharger l'html)

> J'avais au début pensé a utiliser le module `axios` qui est plus léger pour simplement télécharger la page sans utiliser un navigateur sauf que `axios` ne téléchargeait pas les `iframe` de la page ce qui rendait impossible le traitement des données.

Pour le traitement du json, j'utilise `cheerio` qui est un module qui permet de parse l'html en simulant un dom de navigateur permettant de faire les même action qu'en javascript comme

```
parsedHtml.getClass("coucou")
```

*(me renvoyant toutes les balise parse avec la class coucou)

Cela m'a permet d'obtenir tout les `iframe` de la page ou était rangé les données pour pouvoir traiter les données de tableaux. Pour cela, j'ai simplement map les `array` renvoyé par les `getID` de `cheerio` ce qui me donnait les row. Il ne restait qu'a mettre les données en forme dans un json et le sauvegarder dans le cache pour toute demande éventuelle sur l'endpoint.

```
const page = await browser.newPage(); // Crée une page sur le navigateur
await page.goto(`https://resultats.ffbb.com/championnat/${url}`); // Va sur
```

```
l'url avec la page'
const content = await page.content(); // Télécharge le contenu HTML de la page

const $ = cheerio.load(content); // Parse l'html avec cheerio
return $(".liste tbody tr")
        .filter((_i, el) => {
            return $(el).css("display") !== "none"; // Supprime les élément
non visiblent
        })
        .slice(1) // Supprime la première ligne (titre tableau)
        .map((_i, el) => {
            return {
                date: $(el).find("td:nth-child(2)").text().trim(),
                heure: $(el).find("td:nth-child(3)").text().trim(),
                domicile: {
                    name: $(el).find("td:nth-child(4)").text().trim(),
                    score: parseInt($(el).find("td:nth-
child(6)").text().trim().split(" - ")[0]),
                },
                visiteur: {
                    name: $(el).find("td:nth-child(5)").text().trim(),
                    score: parseInt($(el).find("td:nth-
child(6)").text().trim().split(" - ")[1]),
                },
            };
        })
        .get();
```

Cette fonction par exemple renvoie un json avec la liste des match, l'heure, les équipes et les scores. Tout cela est obtenu dans un `iframe` qui a comme contenue une liste de class `.list`. Le `.filter` et `.slice` permettent de prétraiter le tableau car certains élément html non visible n'étaient pas voulu pour le traitement. Le `.slice` permet d'enlever la première ligne qui est l'entête du tableau. Le `.map` prend les données dans le tableau et les remet juste en forme dans le json.

Pour le reste des données, il suffisait de reprendre la même fonction et de l'adapter avec les id, les class et le contenue du `.map` pour renvoyer les données sous le bon format et les mettre dans le cache

Voici le rendu :

```
match/      # Dossier landing match
├── [u].vue # toutes les pages des catégories
```

Dernier match :

DOMICILE	SCORE	VISITEUR	DATE	HEURE
CONDAT BASKET CLUB - 1	73 - 72	IE - CTC BEAUNE RILHAC BONNAC BASKET - US BEAUNE B.B.	29/03/2025	17:00
Aubusson felletin Basket - 1	70 - 51	ASPTT LIMOGES - 3	29/03/2025	16:00
IE - CTC BEAUNE RILHAC BONNAC BASKET - B.B. RILHAC RANCON	59 - 57	ST LOUIS GONZAGUE - 2	29/03/2025	18:00
UNION SPORTIVE NANTIAT	40 - 82	AS SAINT JUNIEN - 1	29/03/2025	17:00

Classement :

	EQUIPE	Pts	J	V	D	N
1	CONDAT BASKET CLUB - 1	24	13	11	2	0
2	ST LOUIS GONZAGUE - 2	22	13	9	4	0
3	IE - CTC BEAUNE RILHAC BONNAC BASKET - US BEAUNE B.B.	22	13	9	4	0
4	Aubusson felletin Basket - 1	20	12	8	4	0
5	ASPTT LIMOGES - 3	20	13	7	6	0
6	IE - CTC BEAUNE RILHAC BONNAC BASKET - B.B. RILHAC RANCON	18	13	5	8	0
7	UNION SPORTIVE NANTIAT	14	13	1	12	0
8	AS SAINT JUNIEN - 1	13	12	1	11	0

```
const route = useRoute()
const { u } = route.params as { u: string }
```

`useRoute()` permet d'accéder à la route actuelle.
On extrait le paramètre `u` de l'URL, qui représente une catégorie (par exemple : `u13`, `seniorf`, etc.).

```
const keyID: Record<string, number> = {
  u11: 1,
  u13: 2,
  u18: 3,
  u21: 4,
  seniorm: 5,
  seniorf: 6,
}
```

On crée une **correspondance (mapping)** entre le paramètre de l'URL et un ID numérique utilisé dans les données de l'API.

```
const actualID = keyID[u]
```

On récupère l'ID associé à la catégorie sélectionnée.

```
const cachedFFBB = useState('ffbbData', () => ({
  timestamp: 0,
```

```
  data: null as ApiFfbb | null,
}))
```

`useState` permet de stocker les données dans un **état persistant partagé**, même entre les pages (dans Nuxt).

- On initialise un cache avec :
  - `timestamp` : pour savoir quand les données ont été récupérées,
  - `data` : les données de l'API au format `ApiFfbb`.

```
  const { data, error } = await useFetch<ApiFfbb>('https://api-ffbb.condatbasketclub.com/', {
    server: true,
  })
```

On fait une requête **serveur uniquement** vers l'API FFBB.

```
  if (error.value) {
    console.error(error.value)
  }
  cachedFFBB.value = {
    timestamp: now,
    data: data.value,
  }
```

On log l'erreur si besoin, puis on met à jour le cache avec les nouvelles données et le timestamp.

```
const actualData = computed(() => {
  return cachedFFBB.value.data?.find((item) => item.id === actualID)
})
```

On cherche dans les données de l'API l'entrée dont l'`id` correspond à l'`actualID`.

```
const selectedPoolName = ref<string>(actualData.value?.pool[0]?.name ?? '')
```

On sélectionne par défaut le **nom du premier pool** trouvé dans les données

```
const selectedPool = computed(() => {
  return actualData.value?.pool.find((pool) => pool.name ===
```

```
  selectedPoolName.value)
  })
```

Ensuite, on récupère **l’objet** `pool` complet correspondant au nom