Gestion des features

## // lib/validations/features.ts

Ce script définit deux schémas de validation Zod utilisés pour **valider les données d'une "feature" (fonctionnalité)** dans une application (probablement liée à un système de gestion de projets ou de tâches).

**📄 Emplacement :**

lib/validations/features.ts

**🔍 Fonction principale :**

Ce script :

1. **Valide les données lors de la création d’une fonctionnalité (createFeatureSchema)**.
2. **Valide partiellement les données lors de la mise à jour d’une fonctionnalité (updateFeatureSchema)**.
3. **Génère les types TypeScript (CreateFeatureInput, UpdateFeatureInput) à partir des schémas.**

**🧩 Détail de chaque champ du schéma createFeatureSchema :**

| **Champ** | **Type** | **Validation** |
| --- | --- | --- |
| name | string | Obligatoire, minimum 1 caractère, maximum 100 caractères |
| description | string | Optionnel |
| status | enum | Enum (Status.TODO, Status.DONE, etc.) avec valeur par défaut TODO |
| priority | number | Nombre entier entre 1 et 5, valeur par défaut = 1 |
| startDate | date | Optionnelle |
| endDate | date | Optionnelle |
| projectId | string | Doit être un UUID valide (identifiant du projet) |
| parentFeatureId | string | Optionnel, UUID valide (identifiant de la feature parente, si hiérarchie) |

**🔁 Schéma de mise à jour : updateFeatureSchema**

Il est dérivé de createFeatureSchema via .partial(), ce qui signifie :

Tous les champs deviennent optionnels pour permettre les mises à jour partielles.

**🧠 Types dérivés :**

Les types TypeScript sont automatiquement générés à partir des schémas :

* CreateFeatureInput : Type d’entrée pour la création.
* UpdateFeatureInput : Type d’entrée pour la mise à jour.

**📦 Utilisation typique dans ton app :**

* Dans une **API route** (POST /api/features) : tu utiliseras createFeatureSchema.parse(req.body) pour valider les données reçues.
* Dans une **route PATCH/PUT** : tu utiliseras updateFeatureSchema.parse(req.body).

**🚀 Résumé :**

Ce fichier permet de **valider** et **typer** les données liées à la création et mise à jour de "features" (fonctionnalités) pour garantir leur cohérence, éviter les erreurs, et faciliter l’autocomplétion/contrôle de types dans ton code.

## lib/actions/features-actions.ts

Ce fichier lib/actions/features-actions.ts contient des **fonctions serveur ('use server')** pour gérer les opérations **CRUD côté serveur** sur les "features" (fonctionnalités) d’un projet via **Prisma** et **Zod**.

**✅ Objectif du fichier :**

Créer, mettre à jour, et supprimer des **features** dans une base de données **PostgreSQL/SQLite** via Prisma ORM, avec validation via Zod.

**🧩 Détail des fonctions**

**🔧 createFeature(data: unknown)**

Crée une nouvelle **feature** (fonctionnalité).

**Étapes :**

1. Valide les données avec createFeatureSchema.
2. Calcule la prochaine position (ordrer) via getNextOrder(projectId).
3. Crée la feature avec Prisma.
4. Recharge la page /projects/[projectId]/features via revalidatePath.
5. Retourne un objet { success: true, feature } ou une erreur.

**Exemple d’utilisation :**

ts

CopierModifier

await createFeature({

name: 'Ajouter l’authentification',

projectId: 'uuid-du-projet'

})

**✏️ updateFeature(id: string, data: unknown)**

Met à jour une **feature** existante.

**Étapes :**

1. Valide les données avec updateFeatureSchema.
2. Met à jour la feature par son id avec Prisma.
3. Recharge la page liée au projet.
4. Retourne un objet { success: true, feature } ou une erreur.

**🗑️ deleteFeature(id: string)**

Supprime une **feature** de la base.

**Étapes :**

1. Supprime la feature via Prisma.
2. Recharge la page des features du projet (revalidatePath).
3. Retourne { success: true } ou une erreur.

**🔢 getNextOrder(projectId: string): Promise<number>**

Calcule la **prochaine position** (ordrer) pour trier les features dans un projet.

**Exemple :**

* Si la dernière feature a un ordrer = 300, la nouvelle aura ordrer = 400.
* Cela permet de garder de l’espace pour des insertions futures entre deux features.

**🔐 Spécificité "use server"**

* use server rend ces fonctions **exécutables uniquement côté serveur** dans Next.js 13+ (avec App Router).
* Elles sont **applicables dans les actions de formulaire ou les appels server actions**.

**🚀 Résumé**

| **Fonction** | **Action** | **Validations** | **Accès DB** | **Revalidation** |
| --- | --- | --- | --- | --- |
| createFeature | Créer une feature | ✅ | ✅ | ✅ |
| updateFeature | Mettre à jour une feature | ✅ | ✅ | ✅ |
| deleteFeature | Supprimer une feature | ❌ | ✅ | ✅ |
| getNextOrder | Déterminer l’ordre à assigner | ❌ |  |  |

## components/features/feature-form.tsx

Ce fichier **components/features/feature-form.tsx** définit un **formulaire React client** ('use client') pour **créer ou modifier une "feature"** (fonctionnalité) dans ton application de gestion de projet.

**🎯 Objectif du composant FeatureForm :**

Permettre à un utilisateur :

* de **créer une nouvelle fonctionnalité**
* ou de **modifier une fonctionnalité existante**

Avec :

* **validation** via Zod
* **gestion de l’état** avec Zustand (useFeaturesStore)
* **appel aux fonctions serveurs** (createFeature, updateFeature)

**🧩 Détail des parties importantes**

**✅ Props attendues**

ts

CopierModifier

interface FeatureFormProps {

projectId: string

feature?: Features

onSuccess?: () => void

}

* projectId : ID du projet auquel rattacher la feature.
* feature : si fourni, le formulaire est en mode **édition**.
* onSuccess : callback exécuté après succès.

**⚙️ Initialisation du formulaire avec react-hook-form**

ts

CopierModifier

const form = useForm<CreateFeatureInput>({

resolver: zodResolver(createFeatureSchema),

defaultValues: { ... }

})

* Utilise le schéma Zod pour valider les entrées.
* Pré-remplit le formulaire si feature est fourni.

**📤 Soumission du formulaire**

ts

CopierModifier

const onSubmit = async (data: CreateFeatureInput) => {

...

}

* Si feature est définie → appel updateFeature.
* Sinon → appel createFeature.
* Met à jour le store Zustand avec la feature reçue (addFeature ou updateFeature).
* Appelle onSuccess() si défini.

**🧱 Structure du formulaire (UI)**

**Champs inclus :**

1. **Nom de la feature** (name)
2. **Description** (description)
3. **Statut** (status) — menu déroulant avec les valeurs de l'enum Status
4. **Priorité** (priority) — champ numérique entre 1 et 5
5. **Bouton de soumission** :
   * Texte = "Créer" ou "Modifier" selon le mode

Chaque champ est géré par FormField de ton système de design (probablement basé sur shadcn/ui ou similaire), avec :

* Label
* Champ de saisie
* Message d'erreur (FormMessage)

**🔁 Gestion d’état local**

ts

CopierModifier

const [isLoading, setIsLoading] = useState(false)

Affiche "Enregistrement..." sur le bouton pendant la soumission.

**🧠 Utilisation de Zustand**

ts

CopierModifier

const { addFeature, updateFeature } = useFeaturesStore()

* addFeature(feature) : ajoute une feature dans le store après création.
* updateFeature(id, feature) : met à jour une feature dans le store après modification.

**📌 En résumé :**

| **Élément** | **Fonction** |
| --- | --- |
| useForm | Gère les champs et validation avec Zod |
| createFeature | Crée une feature côté serveur |
| updateFeature | Met à jour une feature côté serveur |
| useFeaturesStore | Met à jour l’état global côté client |
| revalidatePath | (dans actions) recharge les données à jour côté serveur |
| onSuccess | Callback exécuté après succès |

## components/features/features-list.tsx

Le fichier components/features/features-list.tsx définit un composant React client nommé FeaturesList qui sert à afficher la liste des fonctionnalités ("features") d’un projet donné, avec la possibilité d’en créer une nouvelle.

🧩 Fonction du composant FeaturesList

C’est un composant UI de présentation + gestion d’état local qui :

Reçoit une liste initiale de fonctionnalités (initialFeatures)

La stocke dans un store Zustand (useFeaturesStore)

Filtre ces fonctionnalités par projectId

Les affiche dans une grille de cartes (<FeatureCard />)

Affiche un bouton pour en créer une nouvelle via le callback onCreateClick

🔧 Props attendues

ts

Copier

Modifier

interface FeaturesListProps {

initialFeatures: Features[] // Liste des features au chargement

projectId: string // ID du projet courant

onCreateClick: () => void // Action déclenchée quand l'utilisateur veut créer une feature

}

⚙️ Fonctionnement interne

1. Chargement initial dans le store Zustand

ts

Copier

Modifier

useEffect(() => {

setFeatures(initialFeatures)

}, [initialFeatures, setFeatures])

Dès que le composant est monté (ou que initialFeatures change), il initialise le store Zustand avec les fonctionnalités fournies.

2. Filtrage des features du projet courant

ts

Copier

Modifier

const projectFeatures = features.filter(feature => feature.projectId === projectId)

Seules les fonctionnalités du projet en cours sont affichées.

3. Rendu UI

En-tête

tsx

Copier

Modifier

<h2 className="text-2xl font-bold">Features</h2>

<Button onClick={onCreateClick}>Nouvelle Feature</Button>

Affiche le titre et un bouton pour créer une feature.

Grille des fonctionnalités

tsx

Copier

Modifier

<div className="grid gap-4 md:grid-cols-2 lg:grid-cols-3">

{projectFeatures.map((feature) => (

<FeatureCard key={feature.id} feature={feature} />

))}

</div>

Affiche les fonctionnalités dans des cartes (<FeatureCard />), dans une grille responsive.

Message vide

tsx

Copier

Modifier

{projectFeatures.length === 0 && (

<div className="text-center py-8">

<p>Aucune feature trouvée</p>

<Button onClick={onCreateClick}>Créer votre première feature</Button>

</div>

)}

S’il n’y a pas encore de fonctionnalité, affiche un message + bouton d’invitation.

📌 En résumé

Élément Rôle

initialFeatures Données passées initialement (SSR ou client)

useFeaturesStore Store Zustand pour centraliser les features

onCreateClick Callback déclenché à la création d’une feature

FeatureCard Composant UI pour afficher une feature

projectFeatures Liste filtrée par projet courant

## components/features/feature-card.tsx

Le fichier components/features/feature-card.tsx définit un composant React client nommé FeatureCard.

Ce composant sert à afficher une fonctionnalité ("feature") sous forme de carte dans l’interface utilisateur, avec des actions pour modifier ou supprimer cette feature.

🧩 Résumé du rôle du composant

👉 FeatureCard est une carte interactive qui affiche les détails d’une fonctionnalité (nom, statut, description, priorité, date de début) et permet de :

Modifier la feature (via un bouton "Modifier")

Supprimer la feature (avec confirmation et appel API)

📥 Props attendue

ts

Copier

Modifier

interface FeatureCardProps {

feature: Features // Une instance du modèle Feature (Prisma)

}

⚙️ Fonctionnement détaillé

🔹 1. Import des dépendances

Types : Features, Status (enum Prisma)

UI : Card, Badge, Button, Icons (Lucide)

Store Zustand : useFeaturesStore pour modifier l’état global des features

Action API : deleteFeature() supprime la feature côté serveur (probablement une action server ou API route)

🔹 2. Couleurs associées au statut

ts

Copier

Modifier

const statusColors = {

[Status.TODO]: 'bg-gray-500',

[Status.IN\_PROGRESS]: 'bg-blue-500',

...

}

Chaque statut (TODO, DONE, etc.) a une couleur définie pour le badge visuel.

🔹 3. Gestion de la suppression

ts

Copier

Modifier

const handleDelete = async () => {

if (confirm('Êtes-vous sûr de vouloir supprimer cette feature ?')) {

const result = await deleteFeature(feature.id)

if (result.success) {

deleteFeatureStore(feature.id)

}

}

}

Demande une confirmation à l'utilisateur

Appelle l’action côté serveur deleteFeature(feature.id)

Si suppression réussie, met à jour le store Zustand pour refléter la suppression localement

🔹 4. Gestion de l’édition

ts

Copier

Modifier

const handleEdit = () => {

setSelectedFeature(feature)

}

Déclenche la sélection de la feature dans le store Zustand (probablement pour l’éditer dans un formulaire ailleurs)

🔹 5. Rendu de la carte

Affiche :

Nom de la feature (feature.name)

Statut via un badge coloré

Description (tronquée à 2 lignes avec line-clamp-2)

Priorité (feature.priority) dans un badge

Date de début si présente (feature.startDate)

Deux boutons : Modifier et Supprimer

🖼️ Exemple visuel

less

Copier

Modifier

+--------------------------------------+

| Feature Name [TODO] 🟠 |

|--------------------------------------|

| Lorem ipsum dolor sit amet... |

| |

| Priorité: [Haute] 📅 28/05/2025 |

| |

| [Modifier] [Supprimer] |

+--------------------------------------+

🔄 Interaction avec le store Zustand

Le store useFeaturesStore() est utilisé pour :

Supprimer une feature du store après suppression côté serveur

Définir la feature sélectionnée pour l'édition

✅ En résumé

Élément Rôle

FeatureCard Affiche une carte avec les infos d’une feature

handleEdit Prépare l’édition (ex : ouvrir un modal)

handleDelete Supprime la feature côté API + localement

useFeaturesStore Gère l’état global des features

## app/projects/[projectId]/features/page.tsx

Le fichier app/projects/[projectId]/features/page.tsx est une **page serveur** de votre application Next.js 15 (avec App Router) qui affiche les fonctionnalités (features) d’un projet donné.

**📄 But de ce fichier**

Cette page a pour objectif de :

* Récupérer **les features liées à un projet** donné (via projectId)
* Les **charger côté serveur avec Prisma**
* Les transmettre au composant client FeaturesPageClient pour affichage

**🔍 Analyse ligne par ligne**

**✅ Imports**

ts

CopierModifier

import { prisma } from '@/lib/prisma'

import { FeaturesPageClient } from './features-page-client'

* prisma : Client Prisma pour interroger la base de données
* FeaturesPageClient : Composant React **client** chargé de l’affichage des features

**✅ Props de la page**

ts

CopierModifier

interface FeaturesPageProps {

params: {

projectId: string

}

}

* params.projectId est extrait de l’URL dynamique [projectId]

**✅ Fonction principale**

ts

CopierModifier

export default async function FeaturesPage({ params }: FeaturesPageProps) {

* Fonction serveur async appelée automatiquement par Next.js quand l’URL est visitée (ex: /projects/abc123/features)

**✅ Récupération des données avec Prisma**

ts

CopierModifier

const features = await prisma.features.findMany({

where: { projectId: params.projectId },

include: {

creator: true,

parentFeature: true,

childFeatures: true

},

orderBy: { ordrer: 'asc' }

})

Cette requête :

* Cherche toutes les **features** liées à un **projet** donné (projectId)
* **Inclut** les relations suivantes :
  + creator : la personne ayant créé la feature
  + parentFeature : la feature parente (si hiérarchie)
  + childFeatures : les sous-features
* Trie les résultats par un champ ordrer (⚠️ probablement une faute de frappe pour order ou ordre)

**🔧 Correction possible :**

ts

CopierModifier

orderBy: { order: 'asc' } // si le champ correct est "order"

**✅ Rendu de la page**

ts

CopierModifier

return <FeaturesPageClient features={features} projectId={params.projectId} />

* Passe les données récupérées au composant **client FeaturesPageClient**
* Ce composant s’occupera d’afficher l’UI avec gestion des interactions (édition, suppression, etc.)

**🧠 Résumé**

| **Élément** | **Rôle** |
| --- | --- |
| FeaturesPage | Page serveur pour un projet spécifique |
| projectId | Identifiant du projet (provenant de l'URL) |
| Prisma findMany | Récupère les features associées au projet |
| FeaturesPageClient | Affiche les features dans une interface interactive côté client |

## features-page-client.tsx

Le fichier features-page-client.tsx est un **composant client React** dans votre application Next.js. Il gère **l'affichage interactif des fonctionnalités (features)** pour un projet donné, ainsi que **la création et la modification** de ces features via une **fenêtre modale (dialog)**.

**🧠 Résumé fonctionnel**

Ce composant :

1. Affiche la liste des features d’un projet (FeaturesList)
2. Ouvre une modale pour **créer** ou **modifier** une feature (FeatureForm)
3. Utilise **Zustand** pour gérer l’état global des features (useFeaturesStore)

**🔍 Analyse détaillée**

**✅ Props**

ts

CopierModifier

interface FeaturesPageClientProps {

features: Features[]

projectId: string

}

* features : liste initiale des features du projet (fournie par le serveur)
* projectId : identifiant du projet (pour associer les features au bon projet)

**✅ État local**

ts

CopierModifier

const [isDialogOpen, setIsDialogOpen] = useState(false)

* Gère l'ouverture/fermeture de la modale (dialog) pour le formulaire

ts

CopierModifier

const { selectedFeature, setSelectedFeature } = useFeaturesStore()

* **selectedFeature** : feature actuellement sélectionnée (pour édition)
* **setSelectedFeature** : setter Zustand pour choisir une feature ou réinitialiser

**✅ Gestion des actions utilisateur**

**▶️ Créer une nouvelle feature**

ts

CopierModifier

const handleCreateClick = () => {

setSelectedFeature(null)

setIsDialogOpen(true)

}

* Réinitialise la sélection (null)
* Ouvre la modale pour créer une **nouvelle** feature

**✅ Succès du formulaire**

ts

CopierModifier

const handleSuccess = () => {

setIsDialogOpen(false)

setSelectedFeature(null)

}

* Ferme la modale
* Réinitialise la feature sélectionnée

**✅ Rendu**

tsx

CopierModifier

<FeaturesList

initialFeatures={features}

projectId={projectId}

onCreateClick={handleCreateClick}

/>

* Affiche la liste des features du projet
* Passe le callback pour créer une nouvelle feature

**✅ Modale pour formulaire de feature**

tsx

CopierModifier

<Dialog open={isDialogOpen} onOpenChange={setIsDialogOpen}>

<DialogContent>

<DialogHeader>

<DialogTitle>

{selectedFeature ? 'Modifier la feature' : 'Nouvelle feature'}

</DialogTitle>

</DialogHeader>

<FeatureForm

projectId={projectId}

feature={selectedFeature || undefined}

onSuccess={handleSuccess}

/>

</DialogContent>

</Dialog>

* Si une selectedFeature est définie : formulaire de modification
* Sinon : formulaire de **création**
* Le composant FeatureForm s'occupe de la soumission (création/mise à jour dans la DB)

**📌 En résumé**

| **Élément** | **Rôle** |
| --- | --- |
| FeaturesList | Affiche toutes les features liées au projet |
| FeatureForm | Formulaire pour créer ou modifier une feature |
| Dialog | Modale affichant le formulaire |
| useFeaturesStore | État global pour stocker la feature sélectionnée |
| useState | Contrôle l'ouverture de la modale |