

## Master Intelligence Artificielle et Analyse des Données

### Systèmes Distribués

---

### ***COMPTE RENDU DU TP4***

---

Réalisé par :

**HALIMA DAOUDI**

Année universitaire : 2023 – 2024

# INTRODUCTION

Dans ce travail pratique, je vais développer une application web sophistiquée en utilisant Angular pour le frontend et Spring pour le backend. Cette application est destinée à gérer les paiements des étudiants, permettant une administration détaillée et sécurisée des transactions financières dans un cadre éducatif.

**Première Partie** : Je commencerai par la mise en place d'une application Angular qui interagira avec un backend simulé. Cela me permettra de pratiquer la gestion et la manipulation de données dynamiques, essentielles pour les interactions entre le frontend et le backend.

**Deuxième Partie** : Je poursuivrai avec la création des composants nécessaires pour l'application en utilisant Spring. Cela comprendra la définition des entités JPA pour les étudiants et les paiements, la mise en place de l'architecture RESTful pour les services web, et l'implémentation de la sécurité avec Spring Security et JSON Web Tokens.

**Troisième Partie** : Enfin, je développerai une interface utilisateur avec Angular, intégrant Angular Material pour un design élégant et moderne. L'application permettra aux utilisateurs de gérer les paiements des étudiants, d'afficher les détails des transactions, et de mettre à jour les statuts des paiements de manière sécurisée.

Ce projet me permettra de renforcer mes compétences en développement full-stack tout en fournissant une solution efficace pour les besoins administratifs et financiers dans le secteur éducatif.

Voici les étapes détaillées pour mettre en place l'environnement de développement nécessaire à la réalisation de notre projet Angular avec un backend simulé par JSON Server.

### Étape 1 : Préparation de l'environnement Node.js

Je commencerai par télécharger et installer Node.js depuis le site officiel nodejs.org. Une fois l'installation terminée, je vérifierai que Node.js est correctement installé en exécutant **npm --version** dans le terminal. Ensuite, j'installerai JSON Server, un simulateur de serveur API RESTful, en utilisant la commande **npm install json-server**.

```
C:\Users\HP>npm --version  
9.6.6  
C:\Users\HP>
```

### Étape 2 : Installation d'Angular CLI

Pour installer Angular CLI, qui est essentiel pour créer et gérer des projets Angular, je saisirai la commande **npm install -g @angular/cli** dans mon terminal. Cette étape garantit que j'ai les outils nécessaires pour débuter la création de l'application.

```
C:\Users\HP>npm install -g @angular/cli  
changed 237 packages in 29s  
44 packages are looking for funding  
run `npm fund` for details
```

### Étape 3 : Création du projet Angular

Ensuite, je me positionnerai dans le répertoire où je souhaite créer le projet Angular. Là, je lancerai la commande **ng new TP4\_HALIMA.DAOUDI** pour initialiser mon projet Angular avec toutes les configurations par défaut nécessaires.

```
C:\Users\HP>cd C:\Users\HP\IdeaProjects  
C:\Users\HP\IdeaProjects>ng new TP4_HALIMA.DAOUDI  
? Which stylesheet format would you like to use? CSS [ https://developer.mozilla.org/docs/Web/CSS ]
```

### Étape 4 : Lancement du projet Angular

Après avoir créé le projet, je l'ouvrirai avec un IDE de développement, tel qu'IntelliJ IDEA. Dans le terminal intégré de l'IDE, je démarrerai le serveur de développement Angular en exécutant **ng serve**. Cette commande compile

l'application et lance un serveur web permettant de visualiser l'application dans un navigateur à l'adresse **http://localhost:4200**.

## Étape 5 : Intégration de Bootstrap

Pour styliser l'application, j'installerai Bootstrap et ses icônes en exécutant **npm i bootstrap bootstrap-icons** dans le terminal de mon projet. Pour que ces styles soient pris en compte, je les ajouterai dans les tableaux styles et scripts du fichier angular.json, permettant ainsi à Angular de les charger automatiquement lors du démarrage de l'application.

```
PS C:\Users\HP\IdeaProjects\TP4_HALIMA DAOUDI> npm i bootstrap bootstrap-icons

added 3 packages, and audited 927 packages in 13s

121 packages are looking for funding
```

```
  "styles": [
    "src/styles.css",
    "node_modules/bootstrap/dist/css/bootstrap.min.css"
  ],
  "scripts": [
    "node_modules/bootstrap/dist/js/bootstrap.bundle.js"
  ],
  "server": "src/main.server.ts",
  "prerender": true,
  "ssr": {
    "entry": "server.ts"
  }
}
```

Dans votre fichier styles.css vous ajouter la ligne suivante :

```
/* You can add global styles to this file, and also import other
@import "bootstrap-icons/font/bootstrap-icons.css";
```

## Étape 6 : Création de la barre de navigation

Je vais concevoir le style HTML de notre application en commençant par la barre de navigation dans le fichier **app.component.html**. Initialement, j'utiliserai un simple code HTML avec des éléments `<ul>` et `<li>` pour créer une barre de navigation comprenant trois boutons : Home, Products, et New Product. Cette barre de navigation sera intégrée aux routes de notre application Angular, permettant une navigation fluide entre les différentes sections.

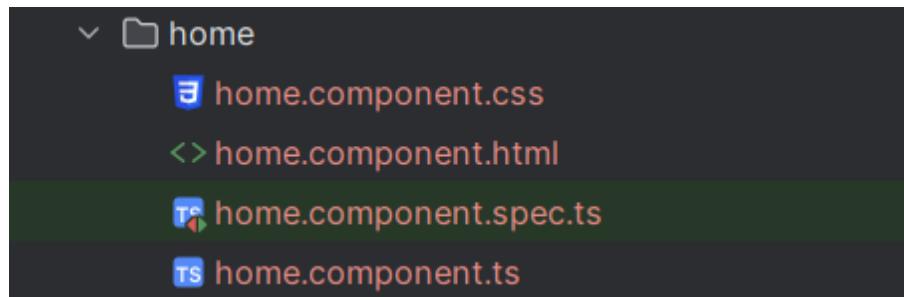
Home 

Products 

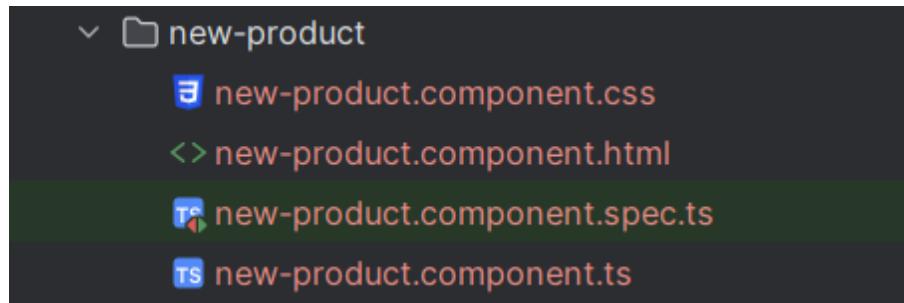
New product 

Après avoir établi la structure de base de la barre de navigation dans le fichier **app.component.html**, je passerai à la création de composants spécifiques pour chaque bouton. Pour ce faire, j'utiliserai les commandes Angular CLI suivantes pour générer un composant dédié à chaque section de l'application :

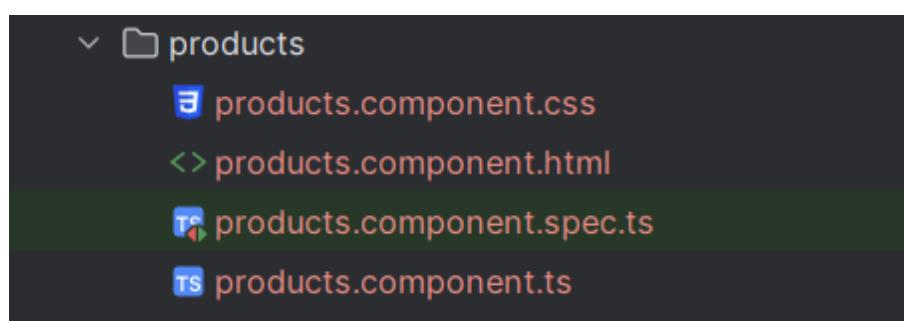
- **ng g c home** pour la page d'accueil.



- **ng g c products** pour la page des produits.

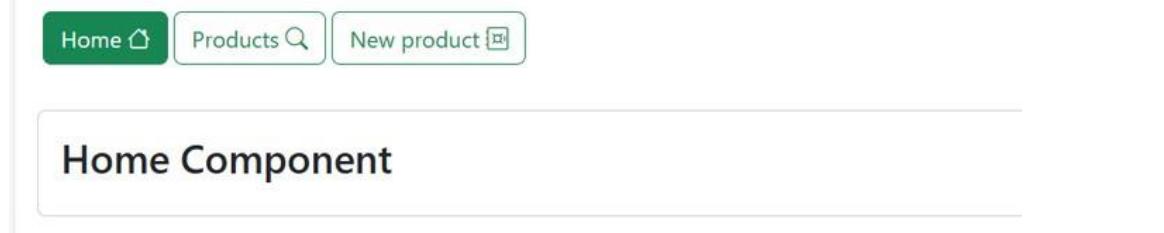


- **ng g c new-product** pour la page de création d'un nouveau produit.



Ces commandes permettront de structurer l'application de manière modulaire, avec un composant distinct pour chaque partie fonctionnelle, facilitant ainsi la maintenance et l'évolutivité de l'application.

Je vais ensuite configurer les routes de navigation dans le fichier **app.routes.ts**. Ces routes seront référencées dans les attributs **routerLink** du fichier **app.component.html** pour permettre une navigation efficace entre les différents composants de l'application.



```
import { Routes } from '@angular/router';
import { HomeComponent } from './home/home.component';
import { ProductsComponent } from './products/products.component';
import { NewProductComponent } from './new-product/new-product.component';

1+ usages new *
export const routes: Routes = [
  {path : "home", component : HomeComponent},
  {path : "products", component : ProductsComponent},
  {path : "newProduct", component : NewProductComponent},
]
```

Dans le fichier **app.component.ts**, je vais organiser la navigation en créant un tableau d'actions, qui inclura les titres, les routes, et les icônes pour chaque section de l'application. Le composant **AppComponent**, qui est le composant racine, utilisera ce tableau pour gérer la barre de navigation et faciliter le passage entre les différentes vues. Je définirai également une méthode **setCurrentAction()** pour actualiser l'élément actif de la navigation lorsque l'utilisateur sélectionne une option.

```
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet, RouterLink, NgForOf],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'
})
export class AppComponent {
  actions : Array<any> = [
    {title : "Home", "route":"/home", icon : "house"},
    {title : "Products", "route":"/products", icon : "search"},
    {title : "New product", "route":"/newProduct", icon : "safe"}
  ]
  currentAction : any;

1+ usages new *
  setCurrentAction(action: any) : void {
    this.currentAction = action;
  }
}
```

Dans app.component.html, je vais utiliser une barre de navigation <nav> où les boutons sont créés dynamiquement avec \*ngFor et liés à leurs routes respectives via routerLink. La classe de chaque bouton change selon l'action sélectionnée. La balise <router-outlet> permet d'afficher les vues associées à chaque route.

```
<nav class="p-3">
  <ul class="nav nav-pills">
    <li *ngFor="let action of actions">
      <button
        (click)="setCurrentAction(action)"
        routerLink="{{action.route}}"
        [class] = "action == currentAction? 'btn btn-success ms-1' : 'btn btn-outline-success ms-1'">
        >
          {{action.title}}
          <i class="bi bi-{{action.icon}}"></i>
      </button>

    </li>
  </ul>
</nav>
<router-outlet>

</router-outlet>
```

## Étape 7 : Configuration de la base de données locale

Pour créer la base de données pour mon projet, je vais d'abord créer un dossier nommé data au sein de mon projet. Ensuite, dans ce dossier, je créerai un fichier appelé db.json. Dans ce fichier, je spécifierai les données concernant les produits. Ce fichier db.json me servira de base de données simulée pour gérer les informations des produits de manière locale.



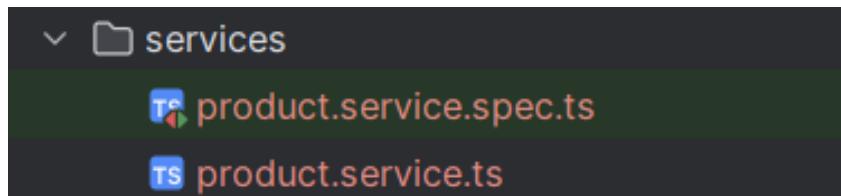
```
products": [
  {
    "id": "1",
    "name": "Computer",
    "price": 5000,
    "checked": true
  },
```

Pour démarrer le serveur JSON, j'utiliserai la commande **json-server -w data/db.json -p 8089**. Cette commande lance le serveur en mode écoute (-w) sur le fichier db.json situé dans le dossier data, et spécifie que le serveur doit écouter sur le port 8089.

```
1 | [ 
2 |   { 
3 |     "id": "1",
4 |     "name": "Computer",
5 |     "price": 5000,
6 |     "checked": true
7 |   },
```

## Étape 8 : Création d'un service

Je vais créer un service pour la gestion des produits en utilisant la commande Angular CLI : **ng g s services/products**. Ce service sera responsable de centraliser et de gérer les interactions avec la base de données des produits, facilitant ainsi les opérations de récupération, ajout, modification, et suppression des données produit.



Pour afficher la liste des produits en cliquant sur le bouton "**Products**", j'utiliserai le module **HttpClient** dans product.service.ts. Le service ProductService, injectable dans toute l'application grâce à l'annotation `@Injectable({ providedIn: 'root' })`, gère les opérations CRUD relatives aux produits. Il se sert de HttpClient pour envoyer des requêtes HTTP au backend. Le service propose plusieurs méthodes : **getProducts()** pour récupérer les produits, **checkProducts()** pour vérifier leur état, et **deleteProducts()** pour les supprimer. Ces méthodes retournent des observables qui permettent une gestion asynchrone des données et des erreurs.

```
✓ export class ProductService {  
  
  no usages  
  constructor(private http : HttpClient) { }  
  
  1+ usages  
  ✓ public getProducts( ) : Observable<Array<Product>>{  
    return this.http.get<Array<Product>>( url: "http://localhost:8089/products");  
  }  
}
```

```

}
1+ usages
public checkProducts(product : Product):Observable<Product>{
  return this.http.patch<Product>( url: `http://localhost:8089/products/${product.id}` ,
    body: { checked: !product.checked });
}

}
1+ usages
public deleteProducts(product : Product) : Observable<any> {
  return this.http.delete<any>( url: `http://localhost:8089/products/${product.id}` );
}

}

```

Dans **products.component.ts**, ProductsComponent s'occupe de l'affichage et de la gestion des produits en utilisant ProductService. Au démarrage avec ngOnInit(), il charge les produits via getProducts(). Il inclut aussi des méthodes pour vérifier (handleCheckProduct()) et supprimer (handleDelete()) des produits, rafraîchissant la liste après chaque opération. Le composant utilise les fonctionnalités d'Angular pour les requêtes HTTP et le rendu asynchrone des données.

```

1+ usages
@Component({
  selector: 'app-products',
  standalone: true,
  imports: [
    HttpClientModule,
    NgForOf,
    AsyncPipe
  ],
  templateUrl: './products.component.html',
  styleUrls: ['./products.component.css']
})
export class ProductsComponent implements OnInit{

  products : Array<Product>=[];
  no usages
  constructor(private productService:ProductService ) {
  }

  ngOnInit():void {
    this.getProducts();
  }
}

```

```

ngOnInit():void {
  this.getProducts();

}
```

```
getProducts() :void {  
  
    this.productService.getProducts()  
        .subscribe( observerOrNext: {  
            next : data :Product[] => {  
                this.products=data  
            },  
            error : err => {  
                console.log(err);  
            }  
        })  
}
```

```
handleCheckProduct(product: Product) :void {  
  
    this.productService.checkProducts(product)  
        .subscribe( observerOrNext: {  
            next: updatedProduct :Product  => {  
                this.getProducts();  
            },  
            error: err => {  
                console.log(err);  
            }  
        })  
}
```

```
handleDelete(product: Product) :void {  
    if (confirm("Etes vous sure? "))  
        this.productService.deleteProducts(product).subscribe( observerOrNext: {  
            next:value => {  
                //this.getProducts();  
                this.products = this.products.filter(p :Product =>p.id!=product.id);  
            }  
        })  
}
```

Dans **products.component.html**, j'utilise un tableau HTML pour lister les produits. Chaque ligne du tableau contient trois colonnes pour le "Nom", "Prix" et l'état "Checked" de chaque produit. La liste des produits est générée avec la directive \*ngFor. Les deux premières colonnes affichent le nom et le prix de chaque produit, tandis que la troisième colonne contient deux boutons : un pour vérifier/décocher le produit et un autre pour le supprimer. Les icônes de ces boutons changent dynamiquement selon l'état de vérification du produit, utilisant des classes conditionnelles.

```

<div class="p-3">
  <div class="card">
    <div class="card-body">
      <table class="table">
        <thead>
          <tr>
            <th>Name</th> <th>Price</th> <th>Checked</th>
          </tr>
        </thead>
        <tbody>
          <tr *ngFor="let product of products">
            <td>{{product.name}}</td>
            <td>{{product.price}}</td>
            <td>
              <button (click)="handleCheckProduct(product)" class="btn btn-outline-success">
                <i [class] = "product.checked?'bi bi-check': 'bi bi-circle'"></i>
              </button>
            </td>
            <td>
              <button (click)="handleDelete(product)" class="btn btn-outline-danger">
                <i class="bi bi-trash"></i>
              </button>
            </td>
          </tr>
        </tbody>
      </table>
    </div>

```

Home

Products

New product

Name	Price	Checked
Computer	5000	<input checked="" type="checkbox"/>
Printer	7000	<input type="checkbox"/>

## Ajouter un produit :

Dans **new-product.component.ts**, je définis **NewProductComponent** pour créer de nouveaux produits. Il importe les modules nécessaires comme Component, OnInit, FormBuilder, FormGroup, et Validators. Annoté avec @Component, il configure le sélecteur, le template HTML, et le style. Le constructeur injecte FormBuilder et ProductService. La méthode ngOnInit initialise le formulaire de produit avec des champs pour le nom, le prix, et l'état de vérification. La méthode saveProduct est appelée à la soumission du formulaire, crée un objet Product, et utilise ProductService pour sauvegarder le produit, gérant les réponses selon le succès ou l'erreur de la requête.

```

@Component({
  selector: 'app-new-product',
  standalone: true,
  imports: [
    ReactiveFormsModule,
    JsonPipe
  ],
  templateUrl: './new-product.component.html',
  styleUrls: ['./new-product.component.css']
})

export class NewProductComponent implements OnInit{
  public productForm!:FormGroup;

  no usages
  constructor(private fb:FormBuilder, private productService:ProductService) {
  }

  no usages
  ngOnInit(): void {
    this.productForm=this.fb.group( controls: {
      name: this.fb.control( formState: '', validatorOrOpts: [Validators.required]),
      price : this.fb.control( formState: 0),
      checked : this.fb.control( formState: false),
    });
  }

  1+ usages
  saveProduct() :void {
    let product:Product=this.productForm.value;
    this.productService.saveProduct(product).subscribe( observerOrNext: {
      next : data :Product  =>{
        alert(JSON.stringify(data));
      }, error : err => {
        console.log(err);
      }
    });
  }
}

```

## Le fichier **product.service.ts**

Dans ce fichier, j'utilise la méthode **saveProduct()** du service **ProductService** pour envoyer une requête POST au serveur backend à l'URL "http://localhost:8089/products". Cette requête enregistre le produit fourni en tant que paramètre, en l'envoyant au format JSON.

```

1+ usages
saveProduct(product: Product):Observable<Product> {
  return this.http.post<Product>( url: `http://localhost:8089/products` ,
  | product);
}

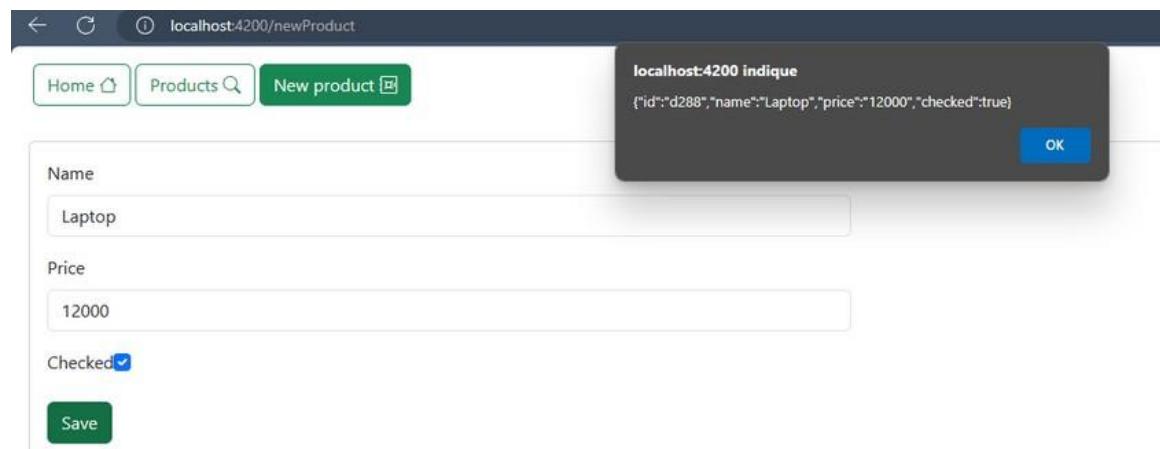
```

## Le fichier **new-product.component.html**:

Ce fichier contient le template HTML pour le composant **NewProductComponent**, définissant la structure visuelle du formulaire de création de produit. Les éléments du formulaire sont liés aux propriétés définies dans le composant via **formControlName**. La soumission du formulaire est gérée par (ngSubmit), qui appelle la méthode saveProduct() du composant. Des validations côté client, telles que required pour le nom et min pour le prix, sont appliquées en utilisant les validateurs fournis par Angular.

```
<div class="p-3">
  <div class="card">
    <div class="card-body">
      <div class="row">
        <div class="col-md-6">
          <form [formGroup]="productForm" (ngSubmit)="saveProduct()" >
            <div class="mb-3">
              <label class="form-label">Name</label>
              <input class="form-control" formControlName="name">
            </div>
            <div class="mb-3">
              <label class="form-label">Price</label>
              <input class="form-control" formControlName="price">
            </div>
            <div class="mb-3">
              <label class="form-label">Checked</label>
              <input type="checkbox" class="form-check-input" formControlName="checked">
            </div>
            <button [disabled]="productForm.invalid" class="btn btn-success">Save</button>
          </form>
        </div>
      </div>
    </div>
  </div>
```

L'affichage :



Name	Price	Checked	
Computer	5000	<input checked="" type="checkbox"/>	
Printer	7000	<input checked="" type="checkbox"/>	
Smart phone	3000	<input type="checkbox"/>	
Laptop	12000	<input checked="" type="checkbox"/>	

## Chercher un produit :

Dans **product.service.ts**, j'utilise la méthode `searchProducts` pour effectuer une recherche de produits basée sur un mot-clé. Cette méthode envoie une requête HTTP GET avec le paramètre `name_like` à l'API, filtrant les produits dont le nom correspond au mot-clé spécifié, offrant ainsi une fonctionnalité de recherche flexible.

```
public searchProducts(keyword:string):Observable<Array<Product>>{
  return this.http.get<Array<Product>>(`https://localhost:8089/products?name_like=${keyword}`)
}
```

Dans **product.component.ts**, je définis une méthode qui utilise `productService` pour rechercher des produits en fonction du mot-clé spécifié dans la variable `keyword`. Lors de la réception de la réponse, j'utilise la méthode `subscribe` pour écouter les résultats. Les nouveaux produits sont alors assignés à la variable `this.products`, ce qui met à jour l'affichage des produits dans le composant.

```
searchProducts(): void {
  this.productService.searchProducts(this.keyword).subscribe(observerOrNext: {
    next : value : Product[] => {
      this.products=value;
    }
  })
}
```

Dans **product.component.html**, j'ai créé une barre de recherche avec un champ de texte et un bouton de recherche. Le champ de texte est lié à la variable `keyword` dans mon composant TypeScript, permettant une saisie en temps réel. Lorsque je clique sur le bouton de recherche, cela déclenche la fonction `searchProducts()` dans mon composant TypeScript pour effectuer la recherche des produits correspondant au mot-clé saisi.

```

<div class="card-body">
  {{keyword}}
  <input type="text" [(ngModel)]="keyword" >
  <button (click)="searchProducts()" class="btn btn-outline-success">
    <i class="bi bi-search"></i>
  </button>
</div>

```

L'affichage :

The screenshot shows a web application interface. At the top, there is a navigation bar with three items: "Home", "Products", and "New product". Below the navigation bar, there is a search bar with the placeholder "Smart phone" and a magnifying glass icon. Underneath the search bar is a table with three columns: "Name", "Price", and "Checked". The table contains two rows of data:

Name	Price	Checked
Smart phone	3000	<input checked="" type="checkbox"/>
Smart phone	4500	<input type="checkbox"/>

## La pagination :

Pour implémenter la pagination, j'ai mis à jour les méthodes `getProducts` dans `product.service.ts` et `product.component.ts`.

Dans le fichier **product.service.ts**, j'utilise la méthode `getProducts()` pour récupérer une page de produits depuis l'API. Cette méthode prend en paramètres le numéro de page (`page`) et la taille de la page (`size`). En envoyant une requête HTTP GET, elle spécifie les paramètres `_page` et `_limit` dans l'URL pour indiquer quelle page et combien de produits doivent être récupérés. L'option `{observe: 'response'}` est utilisée pour obtenir la réponse complète, y compris les en-têtes. Cette méthode retourne un Observable qui émet la réponse HTTP contenant les produits de la page demandée.

```

public getProducts(page :number=1, size:number=4) : Observable<HttpResponse<Object...> {
  // Effectue une requête HTTP GET vers l'API en spécifiant le numéro de page et la taille de la page
  // dans l'URL pour récupérer les produits paginés.
  return this.http.get(url: 'http://localhost:8089/products?_page=${page}&_limit=${size}', options: {observe: 'response'});
}

```

Dans **product.component.ts**, j'appelle la méthode pour récupérer une page de produits à afficher. J'utilise `productService` pour invoquer `getProducts()` avec `this.currentPage` et `this.pageSize`. Lorsque je reçois la réponse, j'utilise `subscribe` pour traiter les données. J'extrais la liste des produits du corps de la réponse (`resp.body as Product[]`) et le nombre total de produits de l'en-tête `x-total-count`. Je calcule ensuite le nombre total de pages pour gérer la pagination. Cette méthode met à jour les propriétés `products`, `totalPages`, et `currentPage` en conséquence.

```

getProducts() : void {
  this.productService.getProducts(this.currentPage, this.pageSize)
    .subscribe( observerOrNext: {
      next: (resp : HttpResponse<Object>) : void => {
        // Extrait la liste des produits du corps de la réponse HTTP.
        this.products = resp.body as Product[];

        // Extrait le nombre total de produits de l'en-tête de réponse.
        let totalProducts: number = parseInt(resp.headers.get('x-total-count')!)];

        // Calcule le nombre total de pages en fonction de la taille de la page.
        this.totalPages = Math.floor( totalProducts / this.pageSize);

        // Si le nombre total de produits n'est pas divisible par la taille de la page,
        // ajoute une page supplémentaire pour les produits restants.
        if (totalProducts % this.pageSize != 0) {
          this.totalPages = this.totalPages + 1;
        }
      },
      error: err => {
        console.log(err);
      }
    });
}

```

Dans le fichier **product.component.html**, je génère une liste de boutons de pagination. L'élément `<ul>` s'affiche uniquement si `totalPages` est supérieur à zéro. J'utilise `*ngFor` pour créer chaque bouton de pagination, chaque `<li>` représentant une page. Les boutons affichent le numéro de page et changent de style selon la page actuelle, grâce à `[ngClass]`, qui applique '`btn-success`' pour la page active et '`btn-outline-success`' pour les autres.

```

</table>
<ul class="nav nav-pills" *ngIf="totalPages && totalPages > 0">
  <li *ngFor="let page of [].constructor(this.totalPages); let i=index">
    <button (click)="handleGoToPage( page: i+1)">
      [ngClass]="currentPage==(i+1)?'btn-success'
      :'btn-outline-success'"
      class="btn m-1">
        {{i+1}}
      </button>
    </li>
  </ul>

```

L'affichage :

The screenshot shows a web application running at `localhost:4200/products`. At the top, there is a navigation bar with links for 'Home' (highlighted), 'Products' (active), and 'New product'. Below the navigation, there is a search bar with a placeholder and a magnifying glass icon. The main content area displays a table with three rows of data:

Name	Price	Checked
Computer	5000	<input checked="" type="checkbox"/>
Printer	7000	<input checked="" type="checkbox"/>
Smart phone	3000	<input type="checkbox"/>

At the bottom of the table, there is a pagination control with three buttons labeled '1', '2', and '3', where '1' is highlighted.

J'ai renommé la méthode `getProduct()` en `searchProduct()` pour mieux refléter sa fonctionnalité de recherche de produits par mot-clé. Cette modification a été appliquée dans les fichiers `product.service.ts` et `product.component.ts`.

Dans **product.service.ts**, j'ai mis à jour la méthode `searchProducts()` pour accepter un paramètre supplémentaire `keyword`, représentant le mot-clé à rechercher. Cette méthode utilise cet argument pour inclure le mot-clé dans l'URL de la requête HTTP, permettant ainsi de rechercher des produits contenant ce mot-clé spécifique.

```
public searchProducts(keyword:string='',page :number=1, size:number=4) : Observable<HttpResponse<Object...> {
    // Effectue une requête HTTP GET vers l'API avec le mot-clé de recherche, le numéro de page et la taille de page spécifiés.
    return this.http.get( url: 'http://localhost:8089/products?name_like=${keyword}&page=${page}&limit=${size}' , options: {observe:'response'});
}
```

Dans **product.component.ts**, j'ai également mis à jour la méthode `searchProducts()` pour inclure le paramètre `keyword` lors de l'appel à la méthode `searchProducts()` du service `productService`. Cette modification permet de rechercher des produits avec pagination, en prenant en compte le mot-clé spécifié pour filtrer les résultats de la recherche.

```
1+ usages
searchProducts() : void {
    this.productService.searchProducts(this.keyword, this.currentPage, this.pageSize)
        .subscribe( observerOrNext: {
            next: (resp : HttpResponse<Object>) : void => {
                // Extrait la liste des produits du corps de la réponse HTTP.
                this.products = resp.body as Product[];

                // Extrait le nombre total de produits de l'en-tête de réponse.
                let totalProducts: number = parseInt(resp.headers.get('x-total-count')!);

                // Calcule le nombre total de pages en fonction de la taille de la page.
                this.totalPages = Math.floor( x totalProducts / this.pageSize);

                // Si le nombre total de produits n'est pas divisible par la taille de la page,
                // ajoute une page supplémentaire pour les produits restants.
                if (totalProducts % this.pageSize != 0) {
                    this.totalPages = this.totalPages + 1;
                }
            },
            error: err => {
                console.log(err);
            }
        });
}
```

L'affichage :

The image contains two identical-looking screenshots of a web application interface. At the top left is a search bar with the placeholder 'Printer' and a magnifying glass icon. Below the search bar is a table with three columns: 'Name', 'Price', and 'Checked'. The first row shows a printer named 'Printer' with a price of 7000 and a checked status (green checkmark). The second row shows a printer named 'Printer' with a price of 4000 and a partially checked status (grey checkmark). The third row shows a printer named 'Printer' with a price of 3600 and an unchecked status (empty circle). To the right of each row is a red trash can icon. At the bottom left of each screenshot are two small buttons labeled '1' and '2'.

Name	Price	Checked
Printer	7000	<input checked="" type="checkbox"/>
Printer	4000	<input checked="" type="checkbox"/>
Printer	3600	<input type="checkbox"/>

1 2

Name	Price	Checked
Printer	14000	<input type="checkbox"/>
Printer	4000	<input checked="" type="checkbox"/>

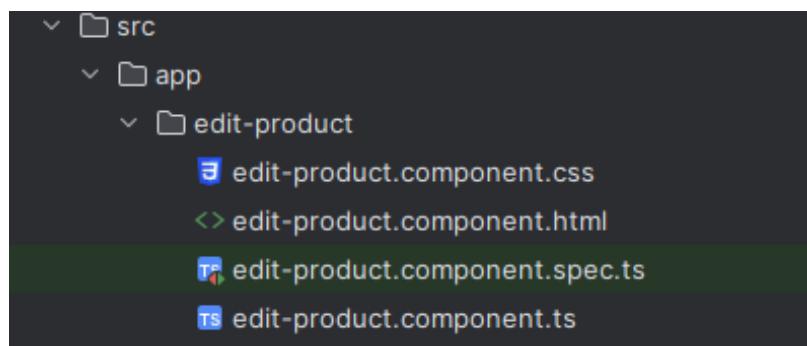
1 2

## La modification des produits :

Pour permettre la modification des produits, j'ai ajouté un bouton pour chaque produit dans `product.component.html`. Ce bouton déclenche l'édition du produit et est lié à la fonction `handleEdit()` définie dans **product.component.ts**.

```
<td>
  <button (click)="handleEdit(product)" class="btn btn-outline-success">
    <i class="bi bi-pencil"></i>
  </button>
</td>
```

J'ai créé un nouveau composant Angular appelé `edit-product` à l'aide de la commande **ng generate component edit-product**. Ce composant fournit une interface pour la modification des produits.



Dans **product.component.ts**, j'ai défini la fonction handleEdit(), qui est responsable de la navigation vers la page de modification (editProduct) avec l'identifiant du produit à éditer.

```
1+ usages
handleEdit(product: Product): void {
  this.router.navigateByUrl(`./editProduct/${product.id}`);
}
```

Dans le fichier **app.routes.ts**, j'ai ajouté la route editProduct/:id pour prendre en charge la modification des produits, associée au composant EditProductComponent.

```
import {EditProductComponent} from './edit-product/edit-product.component';

1+ usages
export const routes: Routes = [
  {path: "home", component: HomeComponent},
  {path: "products", component: ProductsComponent},
  {path: "newProduct", component: NewProductComponent},
  {path: "editProduct/:id", component: EditProductComponent},
];
```

Dans **product.service.ts**, j'ai défini la méthode getProductById() pour envoyer une requête HTTP GET au backend et récupérer les détails d'un produit spécifique en utilisant son identifiant. Elle renvoie un observable avec les détails du produit.

```
getProductById(productId: number): Observable<Product> {
  return this.http.get<Product>(`http://localhost:8089/products/${productId}`);
}
```

J'ai également ajouté la méthode updateProduct() pour mettre à jour les informations d'un produit existant sur le backend. Elle envoie une requête HTTP PUT avec les données du produit modifié et renvoie un observable contenant le produit mis à jour.

```
1+ usages
updateProduct(product: Product): Observable<Product> {
  return this.http.put<Product>(`http://localhost:8089/products/${product.id}`, product);
}
```

Dans **edit-product.component.ts**, je récupère l'identifiant du produit depuis l'URL avec ActivatedRoute dans ngOnInit(). J'appelle ensuite getProductById() de ProductService pour obtenir les détails du produit. En utilisant subscribe(), je remplis un formulaire réactif avec les données du produit, incluant des validations pour les champs name et price. Si une erreur survient, elle est affichée dans la console.

```
export class EditProductComponent implements OnInit{
  productId!: number;
  productFormGroup!: FormGroup;
  no usages
  constructor(private activatedRoute: ActivatedRoute,
    private productService: ProductService,
    private fb : FormBuilder) {
  }
  no usages
  ngOnInit(): void {
    this.productId= this.activatedRoute.snapshot.params['id'];
    this.productService.getProductById(this.productId).subscribe( observerOrNext: {
      next: (product : Product ) : void =>{
        this.productFormGroup= this.fb.group( controls: {
          id : this.fb.control(product.id),
          name : this.fb.control(product.name, [Validators.required]),
          price : this.fb.control(product.price, [Validators.min( min: 100)]),
          checked : this.fb.control(product.checked),
        });
      },
      error : err => {
        console.log(err);
      }
    });
  }
}
```

Dans **edit-product.component.ts**, j'utilise la méthode updateProduct() pour mettre à jour les informations d'un produit. Je récupère les valeurs actuelles du formulaire avec this.productFormGroup.value. Ensuite, j'appelle updateProduct(product) de ProductService, qui envoie une requête HTTP PUT au backend avec les nouvelles valeurs. J'utilise subscribe() pour m'abonner à l'observable renvoyé. En cas de succès, je affiche les données du produit mis à jour dans une alerte avec alert(JSON.stringify(data)).

```
updateProduct(): void {
  let product : Product = this.productFormGroup.value;
  this.productService.updateProduct(product).subscribe( observerOrNext: {
    next : data : Product =>{
      alert(JSON.stringify(data));
    }
  });
}
```

Dans le fichier **edit-product.component.html**, je crée le formulaire de modification d'un produit. Il utilise une carte Bootstrap avec un en-tête affichant l'ID du produit en cours d'édition. Le contenu est conditionnellement rendu pour éviter les erreurs d'affichage à l'initialisation. Le formulaire est lié à `productFormGroup` et comporte des champs pour le nom, le prix, et l'état de vérification du produit. Un bouton "Save" permet de soumettre les modifications, et est désactivé si le formulaire est invalide. Ce fichier offre une interface utilisateur claire pour la modification des détails d'un produit.

```
<div class="p-3">
  <div class="card">
    <div class="card-header">
      Product ID = {{productId}}
    </div>
    <div class="card-body" *ngIf="productFormGroup">
      <form [formGroup]="productFormGroup" (ngSubmit)="updateProduct()">
        <div class="mb-3">
          <label class="form-label">Name</label>
          <input class="form-control" formControlName="name">
        </div>
        <div class="mb-3">
          <label class="form-label">Price</label>
          <input class="form-control" formControlName="price">
        </div>
        <div class="mb-3">
          <label class="form-label">Checked</label>
          <input type="checkbox" class="form-check-input" formControlName="checked">
        </div>
        <button [disabled]="productFormGroup.invalid" class="btn btn-success">Save</button>
      </form>
    </div>
  </div>
</div>
```

L'affichage :

Product ID = 1

Name  
Computer

Price  
10000

Checked

Save

localhost:4200 indique  
{id: 1, name: 'Laptop', price: 14000, checked: true}

## Création de l'AppStateService :

Je vais créer le service AppStateService avec **ng g s services/app-state**. Ce service gérera l'état des produits via la propriété productsState, incluant des champs comme products, keyword, totalPages, pageSize, currentPage, totalProducts, status, et errorMessage.

La méthode setProductState(state: any) permettra de mettre à jour cet état en fusionnant l'état actuel avec le nouvel état, grâce à l'opérateur de propagation (...).

```
export class AppStateService {
  public productsState :any={
    products:[],
    keyword:"",
    totalPages:0,
    pageSize:3,
    currentPage : 1,
    totalProducts : 0,
    status : "ERROR",
    errorMessage : ""
  }
  no usages
  constructor() { }
  1+ usages
  public setProductState(state:any) :void {
    this.productsState={...this.productsState, ...state}
  }
}
```

## Création ProductsComponent :

Je vais créer le composant **Angular ProductsComponent** pour gérer l'affichage et les interactions des produits dans mon application. Ce composant utilisera divers modules et services pour effectuer des opérations telles que la recherche de produits, la gestion des actions comme la sélection, la suppression, la navigation et l'édition des produits.

```
export class ProductsComponent implements OnInit {
  no usages
  constructor(private productService:ProductService,
             private router: Router,
             public appState: AppStateService)
  {}

  no usages
  ngOnInit() :void {
    this.searchProducts();
  }

  1+ usages
  searchProducts() :void { ... }

  1+ usages
  handleCheckProduct(product: Product) :void { ... }
  1+ usages
  handleDelete(product: Product) : void {
    if (confirm("Etes-vous sûr de vouloir supprimer ce produit?"))
      this.productService.deleteProducts(product).subscribe( observerOrNext: {
        next:value => {
          this.searchProducts();
          this.appState.productsState.products = this.appState.productsState.products.filter((p:any) :boolean =>p.id!=product.id);
        }
      })
  }
}
```

```

    handleGotoPage(page: number) : void {
      this.appState.productsState.currentPage=page;
      this.searchProducts();
    }

    1+ usages
    handleEdit(product: Product) : void {
      this.router.navigateByUrl(url: '/editProduct/${product.id}');
    }
}

```

Dans le fichier **products.component.html**, je vais créer une interface utilisateur pour afficher et interagir avec une liste de produits dans mon application Angular. Cette interface permettra de rechercher des produits par mot-clé, d'afficher les produits sous forme de tableau, et d'inclure des fonctionnalités de sélection, suppression, édition, et pagination.

```

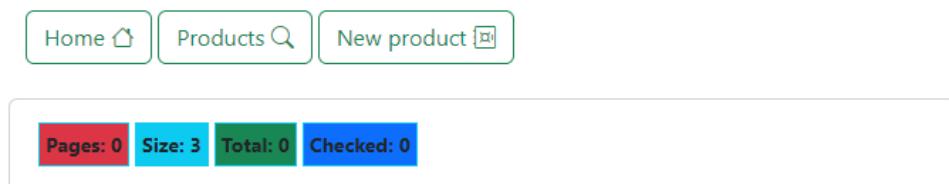
<div class="p-3">
  <div class="card">
    <div class="card-body">
      <div class="card-body">
        {{appState.productsState.keyword}}
        <input type="text" [(ngModel)]="appState.productsState.keyword" >
        <button (click)="searchProducts()" class="btn btn-outline-success ms-1">
          <i class="bi bi-search"></i>
        </button>
      </div>
      <table class="table">
        <thead>
          <tr>
            <th>Name</th> <th>Price</th> <th>Checked</th>
          </tr>
        </thead>
        <tbody>
          <tr *ngFor="let product of appState.productsState.products">
            <td>{{product.name}}</td>
            <td>{{product.price}}</td>
            <td>
              <button (click)="handleCheckProduct(product)"
                  class="btn btn-outline-success">
                <i [class] = "product.checked?'bi bi-check' : 'bi bi-circle'"></i>
              </button>
            </td>
            <td>
              <button (click)="handleDelete(product)" class="btn btn-outline-danger">
                <i class="bi bi-trash"></i>
              </button>
            </td>
            <td>
              <button (click)="handleEdit(product)" class="btn btn-outline-success">
                <i class="bi bi-pencil"></i>
              </button>
            </td>
          </tr>
        </tbody>
      </table>
      <ul class="nav nav-pills" *ngIf="appState.productsState.totalPages && appState.productsState.totalPages > 0">
        <li *ngFor="let page of [].constructor(this.appState.productsState.totalPages); let i=index">
          <button (click)="handleGotoPage(page: i+1)"
                  [ngClass] = "appState.productsState.currentPage==(i+1)?'btn-success'
                  :'btn-outline-success'"
                  class="btn m-1">
            {{i+1}}
          </button>
        </li>
      </ul>
    </div>
  </div>

```

## Création du DashboardComponent :

Je vais créer un composant Angular appelé DashboardComponent en utilisant la commande `ng g c dashboard`. Ce composant affichera des informations résumées sur les produits de mon application. Il utilisera le service AppStateService pour accéder à l'état des produits et afficher des détails tels que le nombre de pages, la taille des pages, le total de produits, et le nombre de produits sélectionnés.

```
<small>
  <nav class="p-1 m-1 text-white fw-bold">
    <div class="card">
      <div class="card-body">
        <ul class="nav nav-pills">
          <li>
            <div class="border border-info bg-danger p-1 ms-1">
              Pages: {{appState.productsState.totalPages}}
            </div>
          </li>
          <li>
            <div class="border border-info bg-info p-1 ms-1">
              Size: {{appState.productsState.pageSize}}
            </div>
          </li>
          <li>
            <div class="border border-info bg-success p-1 ms-1">
              Total: {{appState.productsState.totalProducts}}
            </div>
          </li>
          <li>
            <div class="border border-info bg-primary p-1 ms-1">
              Checked: {{totalCheckedProducts()}}
            </div>
          </li>
        </ul>
      </div>
    </div>
  </nav>
</small>
```



## Création du NavbarComponent :

Je vais créer un composant Angular appelé NavbarComponent en utilisant la commande `ng g c navbar`. Ce composant représentera la barre de navigation de l'application. Il contiendra des boutons pour accéder aux différentes sections de l'application et affichera un spinner de chargement lorsque nécessaire.

```
export class NavbarComponent {
  actions : Array<any> = [
    {title : "Home", "route":"/home", icon : "house"},
    {title : "Products", "route":"/products", icon : "search"},
    {title : "New product", "route":"/newProduct", icon : "safe"}
  ]
  currentAction : any;
  public isLoading :boolean=false;
  no usages
  constructor(public appState :AppStateService ,
              public loadingService : LoadingService) {
    /*this.loadingService.isLoading$.subscribe({
      next : (value)=>{
        this.isLoading=value;
      }
    })*/
  }
  1+ usages
  setCurrentAction(action: any) :void  {
    this.currentAction = action;
  }
}
```

```
<nav class="p-3">
  <ul class="nav nav-pills">
    <li *ngFor="let action of actions">
      <button
        (click)="setCurrentAction(action)"
        routerLink="{{action.route}}"
        [class] = "action == currentAction? 'btn btn-success ms-1' : 'btn btn-outline-success ms-1'"
      >
        {{action.title}}
        <i class="bi bi-{{action.icon}}"></i>
      </button>

    </li>
    <li *ngIf="loadingService.isLoading$ | async">
      <div class="spinner-border text-primary ms-2" role="status">
        </div>
    </li>
  </ul>
</nav>
```



Pages: NaN Size: 3 Total: NaN Checked: 2

## Création du AppErrorsComponent :

Je vais créer un composant Angular appelé AppErrorsComponent en utilisant la commande `ng g c app-errors`. Ce composant affichera les erreurs de l'application lorsque le statut est "ERROR". Il utilisera le service AppStateService pour récupérer et afficher les informations sur les erreurs.

```
export class AppErrorsComponent {  
    no usages  
    constructor(public appState : AppStateService) {  
    }  
}
```

```
<div class="p-3" *ngIf="appState.productsState.status=='ERROR'">  
    <div class="alert alert-danger">  
        {{appState.productsState.errorMessage.message}}  
    </div>  
</div>
```



Pages: 0 Size: 3 Total: 0 Checked: 0

Http failure response for http://localhost:8089/products?  
name\_like=&\_page=1&\_limit=3: 0 undefined

Name	Price	Checked
<input type="text"/>	<input type="button" value="Search"/>	

## Création de l'AppHttpInterceptor

Je vais créer un intercepteur HTTP appelé AppHttpInterceptor en utilisant la commande ng g interceptor app-http. Cet intercepteur gérera l'ajout d'un en-tête d'autorisation aux requêtes sortantes et contrôlera l'affichage du spinner de chargement pendant la durée des requêtes, en utilisant les services AppStateService et LoadingService.

```
@Injectable()
export class AppHttpInterceptor implements HttpInterceptor {

  no usages
  constructor(private appState : AppStateService,
             private loadingService:LoadingService) {}

  no usages
  intercept(request: HttpRequest<unknown>, next: HttpHandler): Observable<HttpEvent<unknown>>
  {
    /*this.appState.setProductState({
      status :"LOADING"
    })*/
    this.loadingService.showLoadingSpinner();
    let req = request.clone( update: {
      headers : request.headers.set("Authorization","Bearer JWT")
    });
    return next.handle(req).pipe(
      finalize( callback: ()=>{
        /*this.appState.setProductState({
          status : ""
        })*/
        this.loadingService.hideLoadingSpinner();
      })
    );
  }
}
```

## Création du LoadingService

Je vais créer un service Angular appelé LoadingService en utilisant la commande ng g s services/loading. Ce service utilisera un observable (isLoading\$) pour gérer l'état du spinner de chargement dans l'application. Il fournira des méthodes showLoadingSpinner() et hideLoadingSpinner() pour afficher et cacher le spinner en émettant des valeurs booléennes.

```

export class LoadingService
{
    public isLoading$ : Subject<boolean> = new Subject<boolean>();

    no usages
    constructor() { }

    1+ usages
    showLoadingSpinner():void {
        this.isLoading$.next( value: true);
    }
    1+ usages
    hideLoadingSpinner():void {
        this.isLoading$.next( value: false);
    }
}

```

## Création de la page login:

ng g c login

Dans le fichier **login.component.html**, je crée la partie HTML de la page de connexion avec les deux champs username et password.

```

<div class="p-3">
  <div class="card">
    <div class="card-header">Authentication</div>
    <div class="card-body">
      <form [formGroup]="formLogin" (ngSubmit)="handleLogin()">
        <div class="mb-3">
          <label class="form-label">Username</label>
          <input type="text" class="form-control" formControlName ="username">
        </div>
        <div class="mb-3">
          <label class="form-label">Password</label>
          <input type="password" class="form-control" formControlName ="password">
        </div>
        <button class="btn btn-success"> Login</button>
      </form>
    <div>
      <div>
        </div>
    </div>
  </div>
</div>

```

Dans le fichier **login.component.ts**, je modifie le fichier pour créer un formulaire de connexion comportant les champs username et password. J'ajoute également une méthode pour gérer la soumission du formulaire et afficher les valeurs entrées dans la console.

```
export class LoginComponent implements OnInit {
  formLogin!: FormGroup;
  no usages
  constructor(private fb :FormBuilder) {
  }
  no usages
  ngOnInit(): void {
    this.formLogin=this.fb.group( controls: {
      username : this.fb.control( formState: ""),
      password : this.fb.control( formState: "")
    })
  }

  1+ usages
  handlelogin(): void {
  console
    console.log(this.formLogin.value);
  }
}
```

Dans le fichier **app.routes.ts**, j'ajoute le chemin (path) pour la page de connexion, afin de rendre cette page accessible via le routage de l'application.

```
export const routes: Routes = [
  {path : "login", component : LoginComponent},
```

La page de connexion comprendra deux champs pour l'authentification : username et password, et sera accessible via le routage de l'application.



[Création de la template admin:](#)

```
ng g c admin-template
```

Dans **admin-template.component.html**, je configure le template de l'admin de manière à ce que la barre de navigation (navbar) et le tableau de bord (dashboard) ne s'affichent que lorsque je suis connecté.

```
<app-navbar></app-navbar>

<app-dashboard></app-dashboard>

<app-app-errors></app-app-errors>

<router-outlet></router-outlet>
```

Dans **app.component.html**, je place uniquement la balise router-outlet pour n'afficher que le formulaire de connexion initialement.

```
<router-outlet>

</router-outlet>
```

J'apporte des modifications aux routes pour rendre certaines pages accessibles uniquement après la connexion.

```
export const routes: Routes = [
  {path : "login", component : LoginComponent},
  {
    path : "admin", component : AdminTemplateComponent, children : [
      {path : "products", component : ProductsComponent},
      {path : "newProduct", component : NewProductComponent},
      {path : "editProduct/:id", component : EditProductComponent},
      {path : "home", component : HomeComponent},
    ],
  },
  {path : "", redirectTo : "login" ,pathMatch:'full'}
];
```

Je vais effectuer un test pour la connexion de l'administrateur et vérifier la redirection vers la page admin après une connexion réussie.

```
1+ usages
handlelogin() : void {
    console.log(this.formLogin.value);
    if(this.formLogin.value.username=="admin" && this.formLogin.value.password=="1234"){
        this.router.navigateByUrl( url: "/admin");
    }
}
```

Je vais apporter des modifications au fichier navbar.component.ts pour gérer la navigation en fonction de l'état de connexion de l'utilisateur.

```
export class NavbarComponent {
  actions : Array<any> = [
    {title : "Home", "route":"/admin/home", icon : "house"}, 
    {title : "Products", "route":"/admin/products", icon : "search"}, 
    {title : "New product", "route":"/admin/newProduct", icon : "safe"}]
```

Je vais créer des utilisateurs pour permettre la connexion via des tokens JWT (JSON Web Token). Ces utilisateurs pourront se connecter et obtenir un token JWT pour authentifier leurs requêtes.

```
{
  "users": [
    {"id": "user1",
      "password": "1234",
      "roles": ["USER"],
      "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJ1c2VyMSIsImhlhdCI6MTUxNjIzOTAyMiwiZ
      {"id": "user2",
        "password": "1234",
        "roles": ["USER"],
        "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJ1c2VyMiIsImhlhdCI6MTUxNjIzOTAyMiwiZ
      {"id": "admin",
        "password": "1234",
        "roles": ["USER", "ADMIN"],
        "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJhZG1pbisImhlhdCI6MTUxNjIzOTAyMiwiZ
    ],
    ]
```

```
1  [
2  {
3    "id": "user1",
4    "password": "1234",
5    "roles": ["USER"]
6    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJ1c2VyMSIsImhlhdCI6MTUxNjIzOTAyMiwiZ
7    "willJoinIn90vWitZmQ1LCjxyX000Tmf2S16111PVVNTRkk1LCj1bwFpbC161m12EBnbwfpbC5jb201LCjyb2xlcyl6y3VU0VS119.RhY-9c5ryub2HOqu6SPPxqafuxlZSCnMk128qy17w"
8  ],
9  {
10   "id": "user2",
11   "password": "1234",
12   "roles": [
13     "USER"
14   ]
15   "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJ1c2VyMSIsImhlhdCI6MTUxNjIzOTAyMiwiZ
16   "willJoinIn90vWitZmQ1LCjxyX000Tmf2S16111PVVNTRkk1LCj1bwFpbC161m12EBnbwfpbC5jb201LCjyb2xlcyl6y3VU0VS119.RhY-9c5ryub2HOqu6SPPxqafuxlZSCnMk128qy17w"
17 }
```

## Configuration d'AuthService

Dans le fichier **auth.service.ts**, j'ai défini authState pour stocker l'état initial de l'authentification de l'utilisateur, incluant les rôles et le token, et ajouté la méthode setAuthState pour mettre à jour cet état de manière dynamique, facilitant la gestion des sessions utilisateur

```
public authState :any ={  
  isAuthenticated : false,  
  username : undefined,  
  roles : undefined,  
  token : undefined  
}
```

```
public setAuthState(state : any) :void{  
  this.authState={...this.authState, ...state};  
}
```

J'ai installé le package **jwt-decode** via npm pour décoder et extraire les informations des tokens JWT dans notre application, améliorant ainsi la gestion de l'authentification et la sécurité des données utilisateur.

```
PS C:\Users\HP\Downloads\master-iaad\master-iaad> npm i jwt-decode  
  
added 1 package, and audited 968 packages in 4s
```

Dans le fichier **auth.service.ts**, j'importe Injectable pour l'injection de dépendances, HttpClient pour effectuer les requêtes HTTP, firstValueFrom pour gérer les Observables, AppStateService pour gérer l'état global, et jwtDecode pour décoder les tokens JWT.

```
import { Injectable } from '@angular/core';  
import {HttpClient} from "@angular/common/http";  
import {firstValueFrom} from "rxjs";  
import {AppStateService} from "./app-state.service";  
import { jwtDecode } from "jwt-decode";
```

La méthode **login** dans **AuthService** utilise HttpClient pour récupérer les données utilisateur, vérifie le mot de passe, et met à jour l'état global via AppStateService si les identifiants sont valides, sinon rejette avec "Bad credentials".

```

export class AuthService {
  no usages
  constructor(private http: HttpClient, private appState: AppStateService) {
  }
  1+ usages
  async login(username: string, password: string) {
    let user: any = await firstValueFrom(this.http.get( url: "http://localhost:8089/users/* + username));
    if (password == atob(user.password)) {
      let decodedJwt: any = jwtDecode(user.token);
      this.appState.setAuthState({
        isAuthenticated: true,
        username: decodedJwt.sub,
        roles: decodedJwt.roles,
        token: user.token
      });
      return Promise.resolve( value: true);
    } else {
      return Promise.reject( reason: "Bad credentials");
    }
  }
}

```

## Mise à jour du LoginComponent

Dans cette mise à jour du **LoginComponent**, j'ai ajouté la variable errorMessage pour capturer et afficher les erreurs lors de la tentative de connexion de l'utilisateur.

```

export class LoginComponent implements OnInit {
  formLogin!: FormGroup;
  errorMessage: undefined = undefined;
  no usages
}

```

Dans la méthode **handleLogin**, j'appelle la fonction de connexion avec les identifiants fournis. Si l'authentification est réussie, l'utilisateur est redirigé vers la page d'administration. En cas d'échec, le message d'erreur correspondant est affiché.

```

handlelogin() {
  let username=this.formLogin.value.username;
  let password=this.formLogin.value.password;
  this.authService.login(username, password) Promise<Awaited<...>>
    .then(resp : boolean =>{
      this.router.navigateByUrl( url: "/admin");
    }) Promise<void>
    .catch(error=>{
      this.errorMessage=error;
    })
}

```

Dans le fichier **login.component.html**, j'ai ajouté une alerte qui affiche les messages d'erreur lorsqu'une tentative de connexion échoue.

```
<div class="p-3">
  <div class="card">
    <div class="card-header">Authentication</div>
    <div class="alert alert-danger" *ngIf="errorMessage">
      {{errorMessage}}
    </div>
    <div class="card-body">
      <form [FormGroup]="formLogin" (ngSubmit)="handleLogin()">
        <div class="mb-3">
          <label class="form-label">Username</label>
          <input type="text" class="form-control" formControlName="username">
        </div>
        <div class="mb-3">
          <label class="form-label">Password</label>
          <input type="password" class="form-control" formControlName="password">
        </div>
        <button class="btn btn-success"> Login</button>
      </form>
    </div>
  </div>
</div>
```

Le formulaire de connexion affiche l'alerte "Bad credentials" en cas d'erreur d'identification, au-dessus des champs de saisie.



## Affichage de l'Utilisateur dans la Barre de Navigation

Dans le fichier **navbar.component.html**, j'ai ajouté des conditions pour afficher le nom de l'utilisateur et un bouton de déconnexion si l'utilisateur est authentifié, sinon un bouton de connexion apparaît pour les utilisateurs non authentifiés.

```

<ul class="nav-pills">
  <ul class="nav nav-pills">
    <li *ngIf="appState.authState.isAuthenticated">
      {{appState.authState.username}}
      <button class="btn btn-success" (click)="logout()">Logout</button>
    </li>
    <li *ngIf="!appState.authState.isAuthenticated">
      <button class="btn btn-success" (click)="login()">Login</button>
    </li>
  </ul>
</ul>
</nav>

```

Dans le fichier **navbar.component.ts**, j'ai ajouté les méthodes logout et login. La méthode logout réinitialise l'état d'authentification et redirige l'utilisateur vers la page de connexion, tandis que la méthode login dirige simplement l'utilisateur vers la page de connexion.

```

logout() {
  this.appState.authState={};
  this.router.navigateByUrl(url: "/login");
}

1+ usages
login() {
  this.router.navigateByUrl(url: "/login");
}
}

```

La barre de navigation affiche le nom de l'utilisateur connecté, user1, avec une option pour se déconnecter, indiquant une gestion réussie de l'état de connexion de l'utilisateur.



## Protection routes

J'ai créé deux gardiens Angular, **AuthenticationGuard** et **AuthorizationGuard**, pour sécuriser les routes en vérifiant l'authentification et les autorisations des utilisateurs.

```

PS C:\Users\HP\Downloads\master-iaad\master-iaad> ng g g guards/authentication
? Which type of guard would you like to create? CanActivate
CREATE src/app/guards/authentication.guard.spec.ts (518 bytes)
CREATE src/app/guards/authentication.guard.ts (143 bytes)
PS C:\Users\HP\Downloads\master-iaad\master-iaad> ng g g guards/authorization
? Which type of guard would you like to create? CanActivate
CREATE src/app/guards/authorization.guard.spec.ts (514 bytes)
CREATE src/app/guards/authorization.guard.ts (142 bytes)

```

Dans la classe **AuthenticationGuard**, j'ai implémenté la méthode canActivate pour vérifier si l'utilisateur est authentifié. Si l'état d'authentification est confirmé, l'accès est autorisé. Dans le cas contraire, l'utilisateur est redirigé vers la page de connexion et l'accès à la route est refusé. Cette méthode assure que seuls les utilisateurs authentifiés peuvent accéder aux routes protégées.

```
export class AuthenticationGuard{
  no usages
  constructor(private appState : AppStateService, private router : Router) {
  }
  no usages
  canActivate(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree {
    if (this.appState.authState.isAuthenticated){
      return true;
    } else{
      this.router.navigateByUrl(url: "/login");
      return false;
    }
  }
}
```

Dans **AuthorizationGuard**, la méthode canActivate vérifie si les rôles de l'utilisateur correspondent aux rôles requis de la route. Si ce n'est pas le cas, l'utilisateur est redirigé vers une page "Non autorisé". Cela assure que seules les personnes autorisées accèdent à certaines routes.

```
export class AuthorizationGuard {
  no usages
  constructor(private appState : AppStateService, private router : Router) {
  }
  no usages
  canActivate(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree {
    if(this.appState.authState.roles.includes(route.data['requiredRoles'])){
      return true;
    } else {
      this.router.navigateByUrl(url: "/admin/notAuthorized")
      return false;
    }
  }
}
```

J'ai créé le composant **not-authorized** pour afficher une page d'erreur lorsque les utilisateurs tentent d'accéder à des routes pour lesquelles ils n'ont pas les autorisations nécessaires.

```
PS C:\Users\HP\Downloads\master-iaad\master-iaad> ng g c not-authorized
CREATE src/app/not-authorized/not-authorized.component.html (30 bytes)
CREATE src/app/not-authorized/not-authorized.component.spec.ts (669 bytes)
CREATE src/app/not-authorized/not-authorized.component.ts (277 bytes)
CREATE src/app/not-authorized/not-authorized.component.css (0 bytes)
```

Dans le fichier **not-authorized.component.html**, j'ai conçu une page qui affiche le message "You are not authorized" pour signaler un accès refusé.

```
<div class="p-3">
  <div class="card">
    <div class="card-body">
      <div class="alert alert-danger">
        You are not authorized
      </div>
    </div>
  </div>
</div>
```

Dans le fichier **app.routes.ts**, j'ai défini les routes sécurisées par **AuthenticationGuard** et **AuthorizationGuard**. La route `notAuthorized` redirige vers une page spécifique lorsqu'un accès non autorisé est détecté. **AuthenticationGuard** vérifie si l'utilisateur est connecté, tandis que **AuthorizationGuard** contrôle les permissions pour accéder aux routes administratives.

```
export const routes: Routes = [
  {path : "login", component : LoginComponent},
  {
    path : "admin", component : AdminTemplateComponent, canActivate:[AuthenticationGuard], children : [
      {path : "products", component : ProductsComponent},
      {path : "newProduct", component : NewProductComponent, canActivate:[AuthorizationGuard],
       data :{requiredRoles :'ADMIN'}
      },
      {path : "editProduct/:id", component : EditProductComponent, canActivate : [AuthorizationGuard],
       data :{requiredRoles :'ADMIN'}
      },
      {path : "home", component : HomeComponent},
      {path : "notAuthorized", component : NotAuthorizedComponent}
    ]
  },
]
```

Si un utilisateur sans droits d'admin essaie d'accéder aux sections d'ajout ou de modification de produits, le message "You are not authorized" s'affiche comme suit :



Dans le fichier **products.component.html**, j'ai utilisé des conditions `*ngIf` pour afficher les boutons d'édition, de suppression, et de vérification uniquement si je possède le rôle 'ADMIN'. Cela assure que seuls les administrateurs peuvent modifier, supprimer ou vérifier les produits.

```

<td *ngIf="appState.authState.roles.includes( value: 'ADMIN')">
  <button (click)="handleCheckProduct(product)" class="btn btn-outline-success">
    <i [class] = "product.checked?'bi bi-check' : 'bi bi-circle '"></i>
  </button>
</td>
<td *ngIf="appState.authState.roles.includes( value: 'ADMIN')">
  <button (click)="handleDelete(product)" class="btn btn-outline-danger">
    <i class="bi bi-trash"></i>
  </button>
</td>
<td *ngIf="appState.authState.roles.includes( value: 'ADMIN')">
  <button (click)="handleEdit(product)" class="btn btn-outline-success">
    <i class="bi bi-pencil"></i>
  </button>
</td>

```

Si un utilisateur qui ne possède pas le rôle '**ADMIN**' consulte le fichier products.component.html, il ne voit pas les options pour éditer, supprimer ou cocher les produits. Il peut seulement visualiser la liste des produits avec leurs noms et prix, ce qui lui permet de parcourir les informations sans pouvoir apporter des modifications.

The screenshot shows a web application interface for managing products. At the top, there is a navigation bar with links for 'Home', 'Products', and 'New product'. On the right side of the navigation bar, there is a user session indicator ('user1') and a 'Logout' button. Below the navigation bar, there is a search bar with fields for 'Page: 4', 'Size: 3', 'Total: 12', and 'Checked: 0'. The main content area displays a table of products with columns for 'Name', 'Price', and 'Checked'. The table contains three rows: 'Computer' (Price: 7000), 'Smart phone' (Price: 3000), and 'Laptop' (Price: 12000). At the bottom of the table, there is a page navigation bar with buttons for '1', '2', '3', and '4'.

Name	Price	Checked
Computer	7000	
Smart phone	3000	
Laptop	12000	

## Partie 2 :

### A. Développer et Tester la partie Backend avec Spring.

#### *1. Créer les entités JPA*

Payment :

J'ai créé l'entité Payment pour capturer les détails d'un paiement effectué par un étudiant. Cette entité comprend plusieurs attributs, notamment l'identifiant du paiement, le montant, la date de paiement, le type de paiement et l'état du paiement. Elle est essentielle pour suivre et gérer les transactions financières des étudiants.

```
@Entity
@NoArgsConstructor @AllArgsConstructor @Getter @Setter @ToString @Builder
public class Payment
{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private LocalDate date;
    private double amount;
    private PaymentType type;
    private PaymentStatus status;
    private String file;
    @ManyToOne
    private Student student;
}
```

PaymentType :

J'ai également défini l'énumération PaymentType, qui énumère les différents types de paiements possibles. Cette classification aide à distinguer les différents modes de paiement utilisés par les étudiants.

```
public enum PaymentType
{
    no usages
    CASH, CHECK, TRANSFER, DEPOSIT
}
```

PaymentStatus :

Pour représenter l'état d'un paiement, j'ai créé l'énumération PaymentStatus. Cette énumération comprend des états tels que CREATED, VALIDATED et REJECTED. Elle est cruciale pour suivre le statut des paiements et gérer les processus de validation et de vérification.

```
public enum PaymentStatus
{
    1 usage
    CREATED, VALIDATED, REJECTED
}
```

## Student :

J'ai également développé l'entité Student, qui représente les informations d'un étudiant. Cette entité inclut des détails personnels et des informations d'identification.

```
@Entity
@NoArgsConstructor @AllArgsConstructor @Getter @Setter @ToString @Builder
public class Student
{
    @Id
    private String id;
    private String firstName;
    private String lastName;
    @Column(unique = true)
    private String code;
    private String programId;
    private String photo;
}
```

## 2. Créer les interfaces JPARepository basées sur Spring Data

### PaymentRepository :

J'ai créé l'interface PaymentRepository en étendant JpaRepository<Payment, Long>. Elle inclut des méthodes pour récupérer les paiements par code étudiant, par statut, par type de paiement et par identifiant de programme étudiant. Ces méthodes facilitent l'accès aux données des paiements et permettent des requêtes spécifiques en quelques lignes de code.

```
public interface PaymentRepository extends JpaRepository<Payment, Long>
{
    1 usage
    List<Payment> findByStudentCode(String code);
    1 usage
    List<Payment> findByStatus(PaymentStatus status);
    1 usage
    List<Payment> findByType(PaymentType type);
    1 usage
    List<Payment> findByStudentProgramId(String programId);
}
```

### StudentRepository :

J'ai développé l'interface StudentRepository en étendant JpaRepository<Student, String>. Cette interface offre des méthodes pour trouver un étudiant par son code unique et pour lister les étudiants en fonction de leur identifiant de programme. Ces méthodes simplifient la gestion et la récupération des données des étudiants de manière efficace et structurée.

```

public interface StudentRepository extends JpaRepository<Student, String>
{
    1 usage
    Student findByCode(String code);
    1 usage
    List<Student> findByProgramId(String programId);
}

```

### *3. Générer des données aléatoires concernant quelques étudiants et pour chaque étudiant des paiements*

J'ai généré des données aléatoires pour tester l'application. D'abord, j'ai créé et sauvegardé plusieurs étudiants avec des codes uniques, des prénoms et des identifiants de programme dans studentRepository. Ensuite, pour chaque étudiant, j'ai généré plusieurs paiements aléatoires avec des montants, des types de paiement, des statuts "CREATED" et des dates actuelles, puis j'ai sauvegardé ces paiements dans paymentRepository. Cela m'a permis de vérifier le bon fonctionnement de l'application avec des données variées.

```

@Bean
CommandLineRunner commandLineRunner(StudentRepository studentRepository, PaymentRepository paymentRepository)
{
    return args -> {
        studentRepository.save(Student.builder().id(UUID.randomUUID().toString())
            .code("112233").firstName("Mohamed").programId("SDIA").build());
        studentRepository.save(Student.builder().id(UUID.randomUUID().toString())
            .code("112244").firstName("Imane").programId("GLSID").build());
        studentRepository.save(Student.builder().id(UUID.randomUUID().toString())
            .code("112255").firstName("Alae").programId("IAAD").build());
        studentRepository.save(Student.builder().id(UUID.randomUUID().toString())
            .code("112266").firstName("Najat").programId("GLSID").build());

        PaymentType[] paymentTypes = PaymentType.values();
        Random random=new Random();
        studentRepository.findAll().forEach(st->{
            for (int i = 0; i <10 ; i++)
            {
                int index = random.nextInt(paymentTypes.length);
                Payment payment = Payment.builder()
                    .amount(1000+(int)(Math.random()*2000))
                    .type(paymentTypes[index])
                    .status(PaymentStatus.CREATED)
                    .date(LocalDate.now())
                    .student(st)
                    .build();
                paymentRepository.save(payment);
            }
        });
    };
}

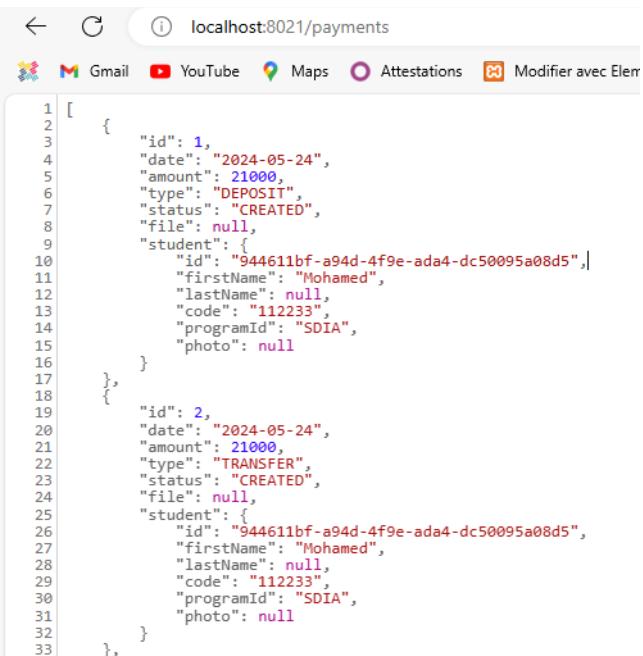
```

#### 4. Créer une Web service RESTful (ResController) qui permet d'exposer les fonctionnalités suivantes :

##### 4.1. Consulter tous les paiements

J'ai créé une méthode pour récupérer et retourner tous les paiements enregistrés.

```
@GetMapping(path="/payments")
public List<Payment> allPayments()
{
    return paymentRepository.findAll();
}
```



A screenshot of a web browser window titled "localhost:8021/payments". The address bar shows the URL. Below the title bar, there are several browser tabs. The main content area displays a JSON array of two payment objects. Each payment object contains fields such as id, date, amount, type, status, file, student, and photo. The student object within each payment contains an id, first name, last name, code, program id, and photo.

```
[{"id": 1, "date": "2024-05-24", "amount": 21000, "type": "DEPOSIT", "status": "CREATED", "file": null, "student": {"id": "944611bf-a94d-4f9e-ada4-dc50095a08d5", "firstName": "Mohamed", "lastName": null, "code": "112233", "programId": "SDIA", "photo": null}}, {"id": 2, "date": "2024-05-24", "amount": 21000, "type": "TRANSFER", "status": "CREATED", "file": null, "student": {"id": "944611bf-a94d-4f9e-ada4-dc50095a08d5", "firstName": "Mohamed", "lastName": null, "code": "112233", "programId": "SDIA", "photo": null}}]
```

##### 4.2. Consulter un paiement sachant son id

J'ai ajouté une méthode pour obtenir un paiement spécifique en utilisant son identifiant unique.

```
@GetMapping(path="/payments/{id}")
public Payment getPaymentById(@PathVariable Long id)
{
    return paymentRepository.findById(id).get();
}
```

```

1 [
2   {
3     "id": 7,
4     "date": "2024-05-24",
5     "amount": 21000,
6     "type": "CHECK",
7     "status": "CREATED",
8     "file": null,
9     "student": {
10       "id": "944611bf-a94d-4f9e-ada4-dc50095a08d5",
11       "firstName": "Mohamed",
12       "lastName": null,
13       "code": "112233",
14       "programId": "SDIA",
15       "photo": null
16     }
17   ]

```

#### 4.3. Consulter les paiements d'un type donné

J'ai mis en place une méthode pour récupérer les paiements filtrés par type.

```

@GetMapping("/payments/byType")
public List<Payment> paymentsByType(@RequestParam PaymentType type)
{
    return paymentRepository.findByType(type);
}

```

```

1 [
2   {
3     "id": 2,
4     "date": "2024-05-24",
5     "amount": 21000,
6     "type": "TRANSFER",
7     "status": "CREATED",
8     "file": null,
9     "student": {
10       "id": "944611bf-a94d-4f9e-ada4-dc50095a08d5",
11       "firstName": "Mohamed",
12       "lastName": null,
13       "code": "112233",
14       "programId": "SDIA",
15       "photo": null
16     }
17   },
18   {
19     "id": 21,
20     "date": "2024-05-24",
21     "amount": 21000,
22     "type": "TRANSFER",
23     "status": "CREATED",
24     "file": null,
25     "student": {
26       "id": "67a0f631-ce76-4d60-b742-1fe4b492db4e",
27       "firstName": "Alae",
28       "lastName": null,
29       "code": "112255",
30       "programId": "IAAD",
31       "photo": null
32     }
33   }

```

#### 4.4. Consulter les paiements d'un status donné

J'ai défini une méthode pour obtenir les paiements en fonction de leur statut

```

@GetMapping("/payments/byStatus")
public List<Payment> paymentsByStatus(@RequestParam PaymentStatus status)
{
    return paymentRepository.findByStatus(status);
}

```

```

1 [ 
2   {
3     "id": 1,
4     "date": "2024-05-24",
5     "amount": 21000,
6     "type": "DEPOSIT",
7     "status": "CREATED",
8     "file": null,
9     "student": {
10       "id": "944611bf-a94d-4f9e-ada4-dc50095a08d5",
11       "firstName": "Mohamed",
12       "lastName": null,
13       "code": "112233",
14       "programId": "SDIA",
15       "photo": null
16     }
17   },
18   {
19     "id": 2,
20     "date": "2024-05-24",
21     "amount": 21000,
22     "type": "TRANSFER",
23     "status": "CREATED",
24     "file": null,
25     "student": {
26       "id": "944611bf-a94d-4f9e-ada4-dc50095a08d5",
27       "firstName": "Mohamed",
28       "lastName": null,
29       "code": "112233",
30       "programId": "SDIA",
31       "photo": null
32     }
33 }

```

#### 4.5. Consulter les paiements d'un étudiant donné

J'ai créé une méthode pour récupérer les paiements effectués par un étudiant spécifique en utilisant son code.

```

@GetMapping("/students/{code}/payments")
public List<Payment> paymentsByStudent(@PathVariable String code)
{
    return paymentRepository.findByStudentCode(code);
}

```

```

1 [ 
2   {
3     "id": 31,
4     "date": "2024-05-24",
5     "amount": 21000,
6     "type": "CHECK",
7     "status": "CREATED",
8     "file": null,
9     "student": {
10       "id": "c60dd45f-a832-4d96-a378-98d93d000210",
11       "firstName": "Najat",
12       "lastName": null,
13       "code": "112266",
14       "programId": "GLSID",
15       "photo": null
16     }
17   },
18   {
19     "id": 32,
20     "date": "2024-05-24"
21 }

```

#### 4.6. Consulter les paiements d'une filière donnée

J'ai ajouté une méthode pour obtenir les paiements des étudiants inscrits dans une filière spécifique.

```

@GetMapping("/payments/byProgramId")
public List<Payment> getPaymentsByProgramId(@RequestParam String programId) {
    return paymentRepository.findByStudentProgramId(programId);
}

```

```

1 [ 
2   { 
3     "id": 21,
4     "date": "2024-05-24",
5     "amount": 21000,
6     "type": "TRANSFER",
7     "status": "CREATED",
8     "file": null,
9     "student": {
10       "id": "67a0f631-ce76-4d60-b742-1fe4b492db4e",
11       "firstName": "Alae",
12       "lastName": null,
13       "code": "112255",
14       "programId": "IAAD",
15       "photo": null
16     }
17   },
18   { 
19     "id": 22,
20     "date": "2024-05-24",
21     "amount": 21000
22   }
23 ]

```

#### 4.7. Consulter tous les étudiants

J'ai mis en place une méthode pour retourner la liste de tous les étudiants inscrits.

```

@GetMapping(path = "/students")
public List<Student> allStudents()
{
    return studentRepository.findAll();
}

```

```

1 [ 
2   { 
3     "id": "944611bf-a94d-4f9e-ada4-dc50095a08d5",
4     "firstName": "Mohamed",
5     "lastName": null,
6     "code": "112233",
7     "programId": "SDIA",
8     "photo": null
9   },
10  { 
11    "id": "9ad3f047-1191-4329-8f4c-c6696f4f1144",
12    "firstName": "Imane",
13  }
14 ]

```

#### 4.8. Consulter les étudiants d'une filière donnée

J'ai défini une méthode pour récupérer les étudiants d'une filière spécifique.

```

@GetMapping(path = "/studentsByProgramId")
public List<Student> getStudentsByProgramId(@RequestParam String programId)
{
    return studentRepository.findByProgramId(programId);
}

```

```

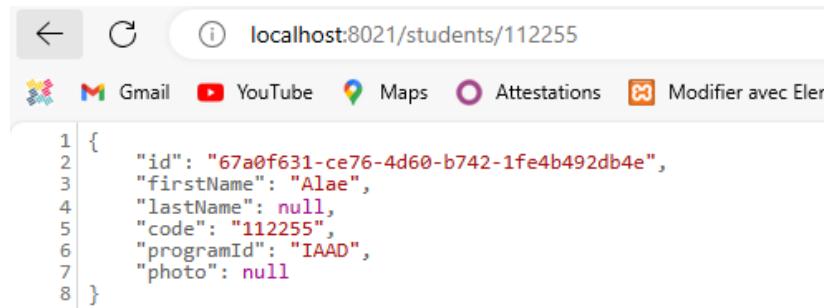
1 [ 
2   { 
3     "id": "67a0f631-ce76-4d60-b742-1fe4b492db4e",
4     "firstName": "Alae",
5     "lastName": null,
6     "code": "112255",
7     "programId": "IAAD",
8     "photo": null
8 ]

```

#### 4.9. Consulter un étudiant sachant son code

J'ai créé une méthode pour obtenir les détails d'un étudiant spécifique en utilisant son code unique.

```
@GetMapping(path = "/students/{code}")
public Student getStudentByCode(@PathVariable String code)
{
    return studentRepository.findByCode(code);
}
```



#### 4.10. Effectuer un nouveau paiement avec les données et le reçu de paiement au format pdf

J'ai créé une méthode pour enregistrer un nouveau paiement. Cette méthode prend un fichier PDF, les détails du paiement, enregistre le fichier sur le serveur, et sauvegarde les informations de paiement dans la base de données.

```
@PostMapping(path = "/payments", consumes = MediaType.MULTIPART_FORM_DATA_VALUE)
public Payment savePayment(@RequestParam MultipartFile file, LocalDate date, double amount,
                           PaymentType type, String studentCode) throws IOException {
    Path folderPath = Paths.get(System.getProperty("user.home"), "master-iaad", "payments");
    if(!Files.exists(folderPath)){
        Files.createDirectories(folderPath);
    }
    String fileName = UUID.randomUUID().toString();
    Path filePath = Paths.get(System.getProperty("user.home"), "master-iaad", "payments", fileName + ".pdf");
    Files.copy(file.getInputStream(), filePath);
    Student student = studentRepository.findByCode(studentCode);
    Payment payment = Payment.builder()
        .date(date)
        .type(type)
        .student(student)
        .amount(amount)
        .file(filePath.toUri().toString())
        .status(PaymentStatus.CREATED)
        .build();
    return paymentRepository.save(payment);
}
```

#### 4.11. Mettre à jour le status d'un paiement

J'ai ajouté une méthode pour modifier le statut d'un paiement existant. Cette méthode récupère le paiement par son ID, met à jour son statut, et sauvegarde les modifications.

```

@PutMapping(path="/payments/{id}/updateStatus")
public Payment updatePaymentStatus(@RequestParam PaymentStatus status, @PathVariable Long id) {
    Payment payment=paymentRepository.findById(id).get();
    payment.setStatus(status);
    return paymentRepository.save(payment);
}

```

#### 4.12. Consulter le reçu d'un payement (fichier pdf)

J'ai mis en place une méthode pour récupérer le fichier PDF du reçu de paiement. Cette méthode localise le paiement par son ID et retourne le contenu du fichier PDF associé.

```

@GetMapping(path="/paymentFile/{paymentId}", produces = MediaType.APPLICATION_PDF_VALUE)
public byte[] getPaymentFile(@PathVariable Long paymentId) throws IOException {
    Payment payment = paymentRepository.findById(paymentId).get();
    return Files.readAllBytes(Path.of(URI.create(payment.getFile())));
}

```

### 5. Tester le backend en utilisant un client REST (Postman) et avec SWAGGER UI

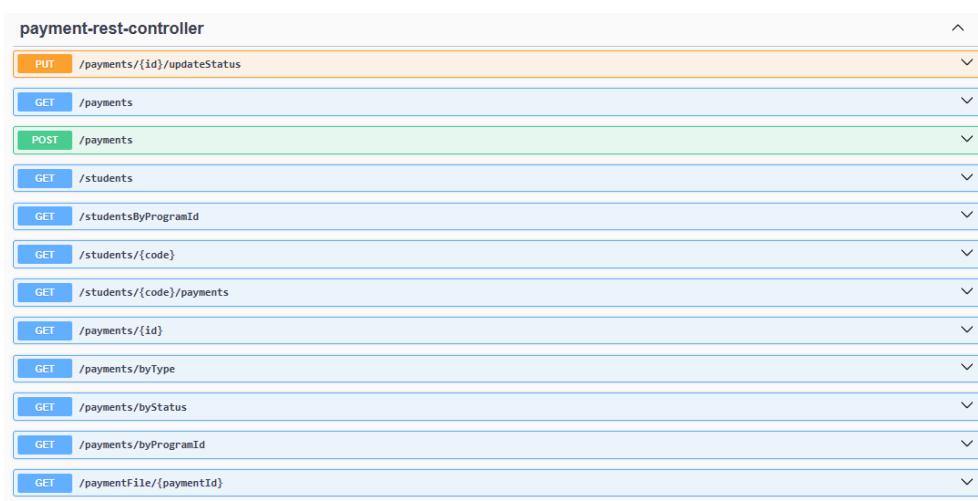
Pour intégrer Swagger UI et générer automatiquement la documentation de l'API, j'ai ajouté la dépendance suivante dans le fichier pom.xml :

```

<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>2.5.0</version>
</dependency>

```

J'accède à <http://localhost:8021/swagger-ui/index.html> pour visualiser la documentation interactive de l'API, explorer les différents endpoints disponibles et tester les requêtes directement via l'interface utilisateur de Swagger.



Je teste la mise à jour du statut d'un paiement avec l'ID 1 en exécutant :

PUT /payments/{id}/updateStatus

Parameters

Name	Description
<b>status</b> * required	VALIDATED
string (query)	
<b>id</b> * required	1
integer(\$int64) (path)	

Execute Clear

200 Response body

```
{
  "id": 1,
  "date": "2024-05-18",
  "amount": 21000,
  "type": "CASH",
  "status": "VALIDATED",
  "file": null,
  "student": {
    "id": "85695dff-d261-493e-9323-ccb4870d3f84",
    "firstname": "Mohamed",
    "lastname": null,
    "code": "112233",
    "programId": "SDIA",
    "photo": null
  }
}
```

Download

Je teste la création d'un nouveau paiement, en incluant les données requises et en attachant le reçu de paiement au format PDF.

POST /payments

Parameters

Name	Description
<b>amount</b> * required	35000
number(\$double) (query)	
<b>type</b> * required	CASH
string (query)	
<b>date</b> * required	2024-05-18
string(\$date) (query)	
<b>studentCode</b> * required	112266
string (query)	

Request body

multipart/form-data

file \* required

string(\$binary) Choisir un fichier TP4\_SD.pdf

Code Details

200 Response body

```
{
  "id": 41,
  "date": "2024-05-18",
  "amount": 35000,
  "type": "CASH",
  "status": "CREATED",
  "file": "C:/Users/HP/Iaad/payments/cdff8223-5364-4962-bc26-97959a543ar0.pdf",
  "student": {
    "id": "384ec5ff-d1d9-4ac8-a8f0-8bd6b1b6178d",
    "firstname": "Nata",
    "lastname": null,
    "code": "112266",
    "programId": "GLSID",
    "photo": null
  }
}
```

Download

Et pour récupérer le fichier associé à un paiement spécifique :

The screenshot shows a REST API tool interface. At the top, it says "GET /payments/{id}/file". Below that is a "Parameters" section with a table. A row in the table has "id" highlighted in red with the note "\* required" and "Integer(\$int64)" as the type, with the value "41" entered. There are "Execute" and "Clear" buttons below the table. Under "Responses", there is a "Curl" section containing a command to make a GET request to "http://localhost:8021/payments/41/file" with headers "-X GET" and "-H accept: application/pdf". Below that is a "Request URL" field with the same URL.

Et J'accède à <http://localhost:8021/payments/41/file> pour voir le file

## 6. Faire un refactoring du code en utilisant la couche service, les DTOs et les Mappers

Pour organiser la logique métier de l'application, j'ai créé la classe **PaymentService**. Dans cette classe, j'ai injecté les dépendances PaymentRepository et StudentRepository afin de pouvoir interagir avec les données des paiements et des étudiants.

```
@Service @Transactional
public class PaymentService {
    5 usages
    private PaymentRepository paymentRepository;
    2 usages
    private StudentRepository studentRepository;
    public PaymentService(PaymentRepository paymentRepository, StudentRepository studentRepository) {
        this.paymentRepository = paymentRepository;
        this.studentRepository = studentRepository;
    }
}
```

J'ai ajouté une méthode savePayment pour enregistrer un nouveau paiement. Cette méthode gère le téléchargement du fichier PDF, l'enregistrement des informations du paiement, et l'association du paiement à l'étudiant correspondant. Elle crée également le répertoire de stockage si nécessaire et génère un nom de fichier unique.

```
public Payment savePayment(MultipartFile file, double amount, PaymentType type,
                           LocalDate date, String studentCode) throws IOException {
    Path folderPath = Paths.get(System.getProperty("user.home"), ...more: "iaad", "payments");
    if(!Files.exists(folderPath)){
        Files.createDirectories(folderPath);
    }
    String fileName = UUID.randomUUID().toString();
    Path filePath = Paths.get(System.getProperty("user.home"), ...more: "iaad", "payments", fileName+".pdf");
    Files.copy(file.getInputStream(), filePath);
    Student student = studentRepository.findByCode(studentCode);
```

```

    Payment payment=Payment.builder()
        .type(type)
        .status(PaymentStatus.CREATED)
        .date(date)
        .student(student)
        .amount(amount)
        .file(filePath.toUri().toString())
        .build();
    return paymentRepository.save(payment);
}

```

Ensute, j'ai implémenté getPaymentFile pour récupérer et lire le fichier PDF d'un paiement existant, ce qui permet de consulter le reçu de paiement.

```

public byte[] getPaymentFile(Long id) throws IOException {
    Payment payment = paymentRepository.findById(id).get();
    return Files.readAllBytes(Path.of(URI.create(payment.getFile())));
}

```

Enfin, j'ai défini la méthode updatePaymentStatus pour mettre à jour le statut d'un paiement. Cette méthode récupère le paiement par son ID, modifie son statut, et sauvegarde les changements dans la base de données.

```

public Payment updatePaymentStatus(PaymentStatus status, Long paymentId){
    Payment payment = paymentRepository.findById(paymentId).get();
    payment.setStatus(status);
    return paymentRepository.save(payment);
}

```

En **PaymentRestController**, J'ai utilisé le service PaymentService pour centraliser la logique métier. Pour mettre à jour le statut d'un paiement, j'appelle updatePaymentStatus du service. Pour enregistrer un nouveau paiement avec un fichier PDF, j'utilise savePayment du service. Enfin, pour récupérer le fichier PDF d'un paiement, j'utilise getPaymentFile du service. Cette approche permet de maintenir un code propre et bien structuré.

```

@PutMapping(path="/payments/{paymentId}/updateStatus")
public Payment updatePaymentStatus(@RequestParam PaymentStatus status, @PathVariable Long paymentId){
    return paymentService.updatePaymentStatus(status,paymentId);
}

@PostMapping(path="/payments", consumes = MediaType.MULTIPART_FORM_DATA_VALUE)
public Payment savePayment(@RequestParam MultipartFile file, double amount, PaymentType type,
                           LocalDate date, String studentCode) throws IOException {
    return paymentService.savePayment(file,amount,type,date,studentCode);
}

@GetMapping(path="/payments/{id}/file",produces = MediaType.APPLICATION_PDF_VALUE)
public byte[] getPaymentFile(@PathVariable Long id) throws IOException {
    return paymentService.getPaymentFile(id);
}

```

J'ai créé la classe PaymentDTO pour transférer les données de paiement entre les couches de l'application. Elle inclut l'identifiant, la date, le montant, le type et le statut du paiement. Grâce à Lombok, j'ai généré automatiquement les constructeurs, getters, setters et la méthode toString, simplifiant ainsi la gestion des données de paiement.

```
@NoArgsConstructor @AllArgsConstructor @Getter @Setter @ToString @Builder
public class PaymentDTO {
    private Long id;
    private LocalDate date;
    private double amount;
    private PaymentType type;
    private PaymentStatus status;
}
```

## B: Développer et Tester la partie Frontend avec angular :

### *1. Créer un projet Angular*

Pour démarrer le développement frontend avec Angular, j'ai exécuté la commande suivante pour créer le projet :

- ng new frontend-ang-app --directory= ./ --no-standalone

Ensuite, j'ai installé Angular Material avec la commande :

- ng add @angular/material

### *2. Intégrer Angular Material*

Pour intégrer Angular Material, j'ai généré un composant admin-template en utilisant la commande :

- ng g c admin-template

### *3. Créer une page template avec une Toolbar et un Side Menu*

J'ai créé une page template incluant une barre de navigation (Toolbar) et un menu latéral (Side Menu) dans le fichier correspondant au composant admin-template.

#### **Admin-template.component.html**

Pour créer une interface utilisateur interactive, j'utilise Angular Material dans le fichier admin-template.component.html. J'ai configuré une barre d'outils avec la balise <mat-toolbar> en utilisant une couleur primaire. À l'intérieur de cette barre d'outils, j'ai intégré plusieurs éléments interactifs.

Un bouton avec une icône de menu déclenche la fonction `toggle()` pour ouvrir ou fermer un tiroir latéral lorsqu'il est cliqué. Ensuite, j'ai ajouté trois boutons qui redirigent les utilisateurs vers les différentes sections de l'application : `/home`, `/profile`, et `/logout`. Enfin, un bouton avec l'attribut `matMenuTrigger` ouvre un menu déroulant. Ce menu propose deux options : "Load Students" et "Load Payments", chacune redirigeant vers une route spécifique de l'application lorsqu'elle est sélectionnée.

```
<mat-toolbar color="primary">
  <button mat-icon-button (click)="myDrawer.toggle()">
    <mat-icon>menu</mat-icon>
  </button>
  <span style="flex: auto"></span>
  <button mat-button routerLink="/home">Home</button>
  <button mat-button routerLink="/profile">Profile</button>
  <button mat-button [matMenuTriggerFor]="importMenu">

    <mat-icon iconPositionEnd>keyboard_arrow_down</mat-icon>
    Import
  </button>
  <mat-menu #importMenu>
    <button mat-menu-item routerLink="/loadStudents">Load Students</button>
    <button mat-menu-item routerLink="/loadPayments">Load Payments</button>
  </mat-menu>
  <button mat-raised-button color="accent" routerLink="/logout">Logout</button>
</mat-toolbar>
```

J'ai mis en place un conteneur de tiroirs en utilisant les composants appropriés pour créer une interface utilisateur structurée. Dans le tiroir, j'ai inclus une liste de boutons qui permettent de naviguer vers différentes sections de l'application, telles que le tableau de bord, la gestion des étudiants et des paiements. Le contenu principal de l'application est affiché dynamiquement en fonction de la route sélectionnée, ce qui permet de charger les différents composants de manière fluide et efficace.

```
<mat-drawer-container>
  <mat-drawer #myDrawer position="start" mode="side" opened="true">
    <mat-list>
      <mat-list-item>
        <button mat-button routerLink="/dashboard">
          <mat-icon>dashboard</mat-icon>
          dashboard
        </button>
      </mat-list-item>
      <mat-list-item>
        <button mat-button routerLink="/students">
          <mat-icon>dashboard</mat-icon>
          Students
        </button>
      </mat-list-item>
```

```
<mat-list-item>
  <button mat-button routerLink="/payments">
    <mat-icon>dashboard</mat-icon>
    Payments
  </button>
</mat-list-item>
</mat-list>
</mat-drawer>
<mat-drawer-content>
  <div style="height: 600px">
    <router-outlet></router-outlet>
  </div>
</mat-drawer-content>
</mat-drawer-container>
```

#### 4. Créer les différents composants de l'application

J'ai généré les différents composants nécessaires pour l'application en utilisant les commandes Angular CLI :

J'ai créé le composant de connexion (login) pour gérer l'authentification.

- ng g c login

J'ai ajouté le composant students pour la gestion des étudiants.

- ng g c students

J'ai généré le composant dashboard pour afficher le tableau de bord.

- ng g c dashboard

J'ai créé le composant payments pour la gestion des paiements.

- ng g c payments

J'ai ajouté le composant home pour la page d'accueil.

- ng g c home

J'ai généré le composant profile pour la gestion des profils utilisateurs.

- ng g c profile

J'ai créé les composants load-students et load-payments pour charger les informations des étudiants et des paiements respectivement.

- ng g c load-students
- ng g c load-payments

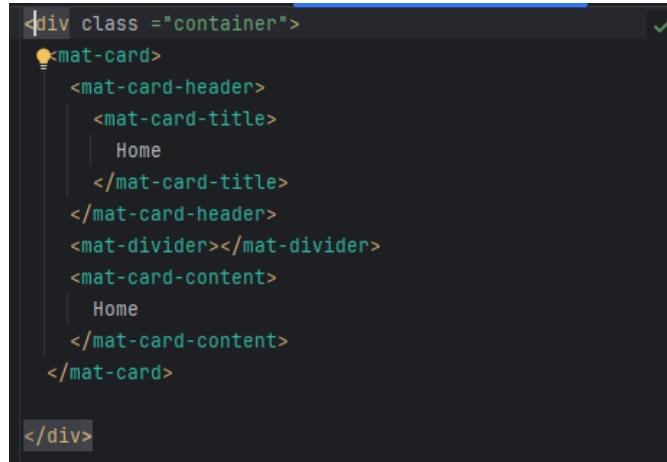
Ces composants constituent les différentes parties de l'application, permettant une organisation claire et une navigation efficace.

J'ai configuré les routes de l'application pour lier les chemins URL aux composants spécifiques. Par exemple, accéder à "/home" affiche le HomeComponent, "/profile" affiche le ProfileComponent, Cela permet de charger les bons composants en fonction de l'URL demandée par l'utilisateur, assurant une navigation dynamique et cohérente.

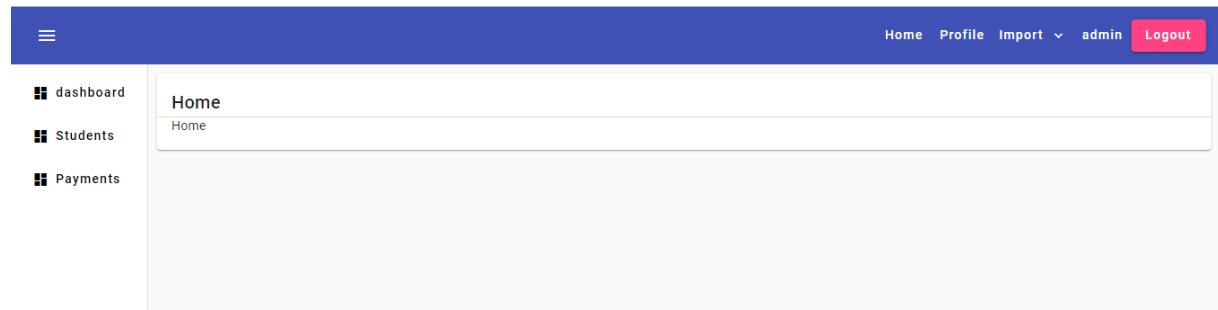
Et Après, J'utilise ces routes dans le composant admin-template pour gérer efficacement la navigation de l'application

```
const routes: Routes = [
  {path : "home", component : HomeComponent},
  {path : "profile", component : ProfileComponent},
  {path : "logout", component : LogoutComponent},
  {path : "students", component : StudentsComponent},
  {path : "payments", component : PaymentsComponent},
  {path : "loadStudents", component : LoadStudentsComponent},
  {path : "loadPayments", component : LoadPaymentsComponent},
  {path : "dashboard", component : DashboardComponent}
];
```

J'ai mis à jour **home.component.html** en utilisant un mat-card pour structurer la page d'accueil. Le titre "Home" est affiché dans le mat-card-header et le contenu principal dans le mat-card-content. Cette mise à jour améliore l'apparence et l'organisation de la page d'accueil.



```
<div class ="container">
  <mat-card>
    <mat-card-header>
      <mat-card-title>
        Home
      </mat-card-title>
    </mat-card-header>
    <mat-divider></mat-divider>
    <mat-card-content>
      Home
    </mat-card-content>
  </mat-card>
</div>
```



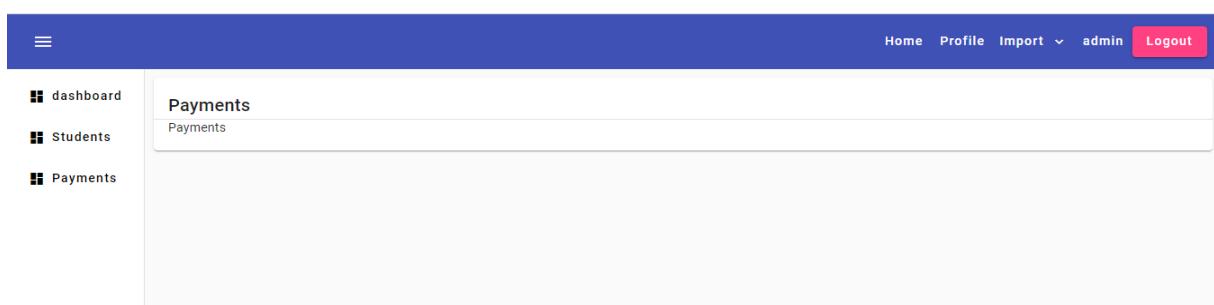
J'ai créé une classe container dans le fichier CSS global de l'application. Cette classe ajoute un padding de 10 pixels pour structurer l'espacement du contenu. J'ai également défini des styles globaux pour html et body, en spécifiant une hauteur de 100% et des marges à zéro, tout en définissant la police de caractères à "Roboto" et "Helvetica Neue". Cette configuration améliore la mise en page et la présentation de l'application.

```
html, body { height: 100%; }
body { margin: 0; font-family: Roboto, "Helvetica Neue",
       .container{
         padding: 10px;
       }
```

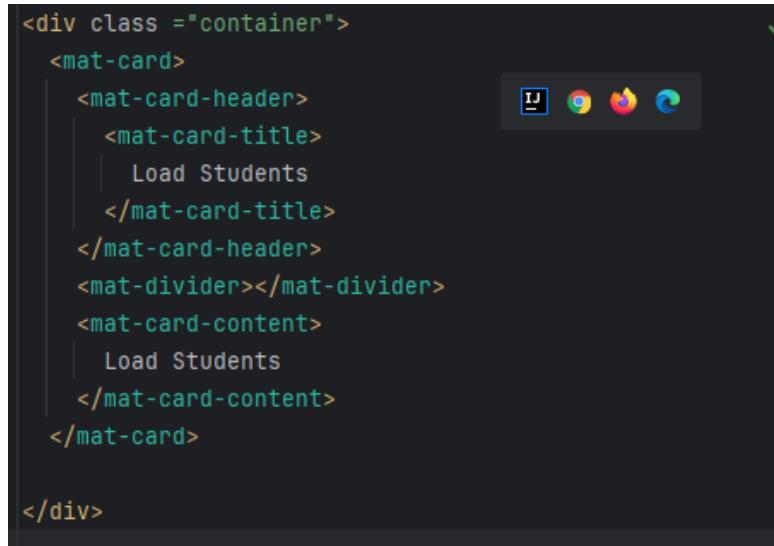
J'ai structuré **payments.component.html** en utilisant mat-card pour afficher les paiements avec un titre, un diviseur, et le contenu principal pour une meilleure organisation et lisibilité.

```
<div class ="container">
  <mat-card>
    <mat-card-header>
      <mat-card-title>
        Payments
      </mat-card-title>
    </mat-card-header>
    <mat-divider></mat-divider>
    <mat-card-content>
      Payments
    </mat-card-content>
  </mat-card>

</div>
```



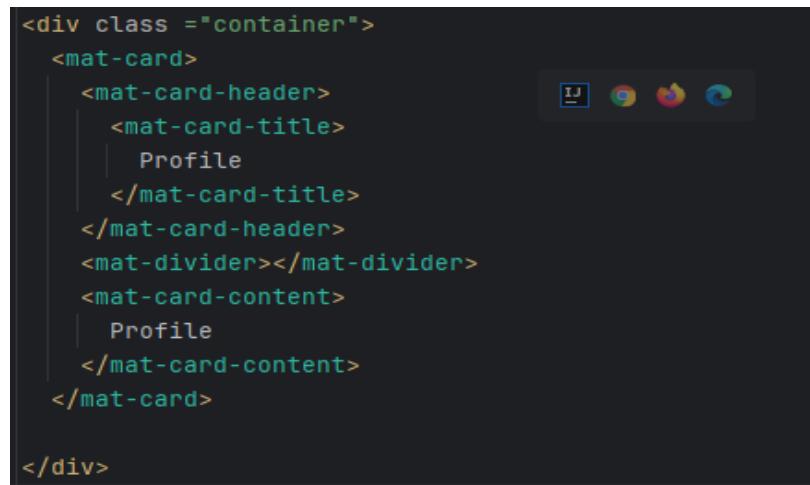
J'ai structuré **load-students.component.html** en utilisant mat-card pour afficher les étudiants avec un titre, un diviseur, et le contenu principal pour une meilleure organisation et lisibilité.



```
<div class ="container">
  <mat-card>
    <mat-card-header>
      <mat-card-title>
        Load Students
      </mat-card-title>
    </mat-card-header>
    <mat-divider></mat-divider>
    <mat-card-content>
      Load Students
    </mat-card-content>
  </mat-card>

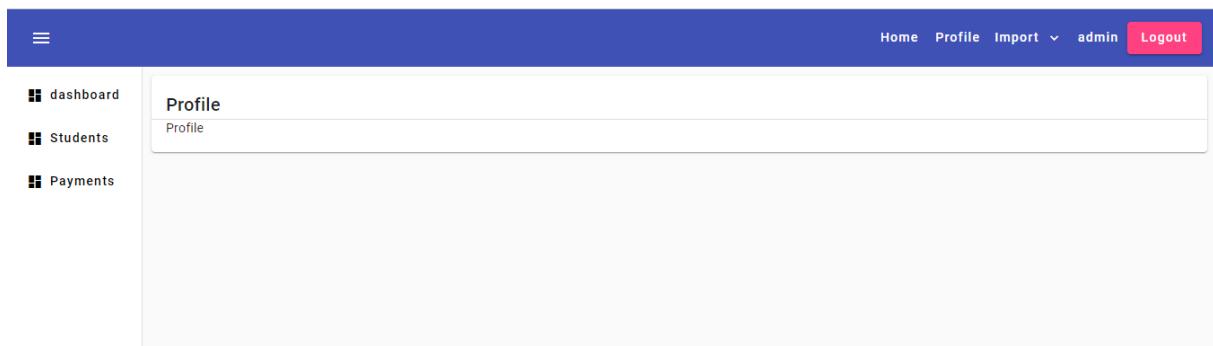
</div>
```

J'ai structuré **profile.component.html** en utilisant mat-card pour afficher le profil utilisateur avec un titre, un diviseur, et le contenu principal pour une meilleure organisation et lisibilité.



```
<div class ="container">
  <mat-card>
    <mat-card-header>
      <mat-card-title>
        Profile
      </mat-card-title>
    </mat-card-header>
    <mat-divider></mat-divider>
    <mat-card-content>
      Profile
    </mat-card-content>
  </mat-card>

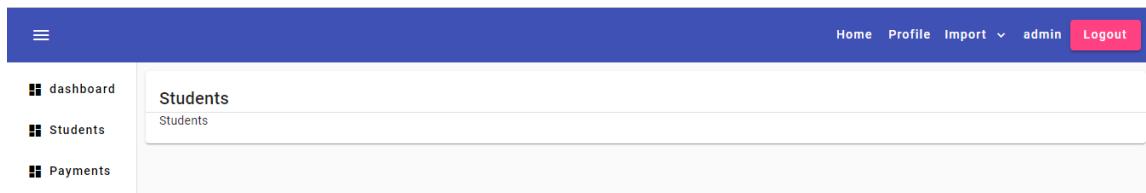
</div>
```



J'ai structuré **students.component.html** en utilisant mat-card pour afficher les informations des étudiants avec un titre, un diviseur, et le contenu principal pour une meilleure organisation et lisibilité.

```
<div class="container">
  <mat-card>
    <mat-card-header>
      <mat-card-title>
        Students
      </mat-card-title>
    </mat-card-header>
    <mat-divider></mat-divider>
    <mat-card-content>
      Students
    </mat-card-content>
  </mat-card>

</div>
```

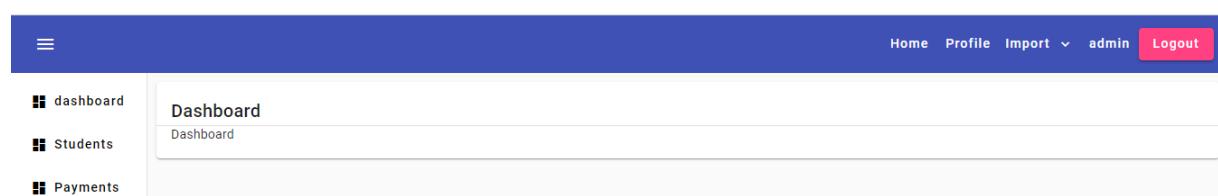


J'ai mis à jour **app.component.html** pour inclure une balise `<router-outlet>`. Cette balise permet de charger dynamiquement les composants correspondant aux différentes routes de l'application, assurant ainsi une navigation fluide et cohérente entre les pages.

```
<router-outlet></router-outlet>
```

J'ai structuré **dashboard.component.html** en utilisant mat-card pour afficher les éléments du tableau de bord avec un titre, un diviseur, et le contenu principal pour une meilleure organisation et lisibilité.

```
<div class="container">
  <mat-card>
    <mat-card-header>
      <mat-card-title>
        Dashboard
      </mat-card-title>
    </mat-card-header>
    <mat-divider></mat-divider>
    <mat-card-content>
      Dashboard
    </mat-card-content>
  </mat-card>
```



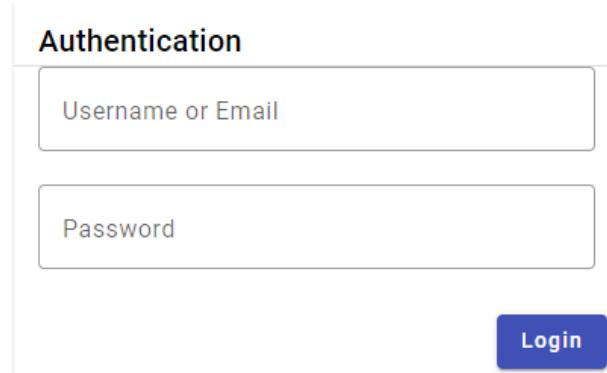
Ensuite, j'ai modifié les routes de l'application pour inclure les différents composants et structurer la navigation.

```
const routes: Routes = [
  {path: "", component: LogoutComponent},
  {path: "logout", component: LogoutComponent},
  {path: "admin", component: AdminTemplateComponent, children: [
    {path: "home", component: HomeComponent},
    {path: "profile", component: ProfileComponent},
    {path: "students", component: StudentsComponent},
    {path: "payments", component: PaymentsComponent},
    {path: "loadStudents", component: LoadStudentsComponent},
    {path: "loadPayments", component: LoadPaymentsComponent},
    {path: "dashboard", component: DashboardComponent}
  ]},
];
```

J'ai conçu la page d'authentification avec Angular Material, incluant un formulaire de connexion pour le nom d'utilisateur/email et le mot de passe, encapsulé dans un mat-card pour une présentation propre et structurée.

```
<div class="container">
{{loginForm.value | json}}
<mat-card class="login-form" [formGroup]="loginForm">
  <mat-card-header>
    <mat-card-title>
      Authentication
    </mat-card-title>
  </mat-card-header>
  <mat-divider></mat-divider>
  <mat-card-content>
    <mat-form-field appearance="outline">
      <mat-label>Username or Email</mat-label>
      <input matInput formControlName="username">
    </mat-form-field>
    <mat-form-field appearance="outline">
      <mat-label>Password</mat-label>
      <input matInput type="password" formControlName="password">
    </mat-form-field>
  </mat-card-content>
  <mat-card-actions align="end">
    <button (click)="login()" mat-raised-button color="primary">Login</button>
  </mat-card-actions>
</mat-card>
</div>
```

Voici l'affichage final de la page d'authentification.



J'ai configuré le composant **LoginComponent** pour récupérer les données du formulaire de connexion. J'ai utilisé FormBuilder pour initialiser le formulaire avec les champs de nom d'utilisateur et de mot de passe. La méthode login() récupère les valeurs du formulaire, appelle le service d'authentification, et si l'authentification est réussie, redirige l'utilisateur vers la route /admin.

```
export class LoginComponent implements OnInit{
  public loginForm!: FormGroup;
  no usages
  constructor(private fb : FormBuilder, private authService : AuthService, private router : Router) {
  }
  no usages
  ngOnInit(): void {
    this.loginForm= this.fb.group( controls: {
      username : this.fb.control( formState: ''),
      password : this.fb.control( formState: '')
    });
  }
  1+ usages
  login(): void {
    let username = this.loginForm.value.username;
    let password = this.loginForm.value.password;
    let auth :boolean = this.authService.login(username, password);
    if(auth==true){
      this.router.navigateByUrl( url: "/admin")
    }
  }
}
```

Pour mettre en place un système d'authentification basique, j'ai créé un service avec la commande :

- ng g s services/auth

J'ai créé **AuthService** pour gérer l'authentification de deux utilisateurs : admin et user1. Le service maintient l'état d'authentification et les rôles des utilisateurs. La méthode login() vérifie les identifiants de l'utilisateur, et si corrects, met à jour l'état d'authentification et redirige vers /admin. La méthode logout() réinitialise l'état et redirige vers /logout.

```
1+ usages
@Injectable({
  providedIn: 'root'
})
export class AuthService {

  public isAuthenticated : boolean = false;
  public username : any;
  public roles : string[] = [];

  public users: any={
    admin :{password : '1234', roles : ['STUDENT', 'ADMIN']},
    user1 :{password : '1234', roles : ['STUDENT']},
  }

  no usages
  constructor(private router : Router) { }
```

```
constructor(private router : Router) { }

1+ usages
public login(username: string, password : string): boolean{
  if(this.users[username] && this.users[username]['password']==password){
    this.isAuthenticated = true;
    this.username = username;
    this.roles = this.users[username]['roles'];
    return true
  } else { return false;}
}

1+ usages
logout() : void {
  this.isAuthenticated = false;
  this.username = undefined;
  this.roles=[];
  this.router.navigateByUrl(url: "/logout")
}
```

J'ai intégré AuthService dans le composant AdminTemplateComponent pour gérer les opérations de déconnexion. Le constructeur injecte le service d'authentification, et la méthode logout() appelle authService.logout() pour déconnecter l'utilisateur et réinitialiser l'état d'authentification.

```
import { Component } from '@angular/core';
import {AuthService} from "../services/auth.service";

1+ usages
@Component({
  selector: 'app-admin-template',
  templateUrl: './admin-template.component.html',
  styleUrls: ['./admin-template.component.css'
})
export class AdminTemplateComponent {
  no usages
  constructor(public authService : AuthService) {

  }

  1+ usages
  logout() : void {
    this.authService.logout()
  }
}
```

J'ai implémenté la fonction logout dans auth.service.ts pour gérer la déconnexion des utilisateurs.

```
1+ usages
Logout(): void {
    this.isAuthenticated = false;
    this.username = undefined;
    this.roles=[];
    this.router.navigateByUrl(url: "/logout")
}
```

**Connexion de l'administrateur** : Lorsque l'administrateur se connecte, l'interface affiche les privilèges et les options spécifiques à l'admin.



**Connexion de l'utilisateur user1** : Lorsque l'utilisateur user1 se connecte, l'interface montre les options et les privilèges spécifiques à ce rôle.



Pour sécuriser la connexion, j'ai créé un guard en utilisant la commande :

- ng g guard auth

J'ai créé AuthGuard pour protéger les routes. Le guard utilise AuthService pour vérifier si l'utilisateur est authentifié. Si l'utilisateur est authentifié, il permet l'accès à la route, sinon l'accès est refusé. Cela assure que seules les personnes connectées peuvent accéder aux routes protégées.

```
> import ...
no usages
@Injectable()
export class AuthGuard {

    no usages
    constructor(private authService : AuthService, private router : Router) {
    }
    no usages
    canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): MaybeAsync<GuardResult> {
        return this.authService.isAuthenticated;
    }
}
```

Ensute, j'ai déclaré le service dans app.module.ts.

```
providers: [
  provideClientHydration(),
  provideAnimationsAsync() , AuthGuard
],
```

Puis, j'ai protégé la route /admin dans app-routing.module.ts en ajoutant le guard.

```
{path : "admin", component : AdminTemplateComponent,
  canActivate : [AuthGuard],
  children: [
    {path: "home", component: HomeComponent},
    {path: "profile", component: ProfileComponent},
    {path: "students", component: StudentsComponent},
```

Maintenant, lorsque nous accédons à l'URL : <http://localhost:4200/admin>, la protection est active et la page n'affiche rien si l'utilisateur n'est pas authentifié.

Le fichier auth.guard.ts contient un gardien d'authentification (AuthGuard) qui protège les routes. Il utilise AuthService pour vérifier l'authentification de l'utilisateur et Router pour la navigation. La méthode canActivate autorise l'accès à la route si l'utilisateur est authentifié, sinon, il est redirigé vers /logout. Cela garantit que seules les personnes authentifiées peuvent accéder aux routes protégées.

```
@Injectable()
export class AuthGuard {

  no usages
  constructor(private authService : AuthService, private router : Router) {
  }
  no usages
  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): MaybeAsync<GuardResult> {
    if(this.authService.isAuthenticated){
      return true;
    }
    else{
      this.router.navigateByUrl( url: '/logout')
      return false;
    }
  }
}
```

Pour protéger les routes selon les rôles, j'ai ajouté un autre gardien : authorization.guard.ts. et J'ai déclaré AuthorizationGuard dans les providers du fichier app.module.ts.

```
providers: [
  provideClientHydration(),
  provideAnimationsAsync() , AuthGuard , AuthorizationGuard
],
```

Ensute, j'ai protégé les routes appropriées dans le fichier app-routing.module.ts en utilisant AuthorizationGuard.

```
const routes: Routes = [
  {path : "", component : LogoutComponent},
  {path : "logout", component : LogoutComponent},
  {path : "admin", component : AdminTemplateComponent,
  canActivate : [AuthGuard],
  children: [
    {path: "home", component: HomeComponent},
    {path: "profile", component: ProfileComponent},
    {path: "students", component: StudentsComponent},
    {path: "payments", component: PaymentsComponent},
    {
      path: "loadStudents", component: LoadStudentsComponent,
      canActivate : [AuthorizationGuard] , data: {roles : ['ADMIN']}
    },
    {path: "loadPayments", component: LoadPaymentsComponent},
    {path: "dashboard", component: DashboardComponent}
  ],
];
]
```

J'ai créé le fichier authorization.guard.ts pour protéger les routes en fonction des rôles des utilisateurs. J'utilise AuthService pour vérifier l'authentification et obtenir les rôles des utilisateurs, et Router pour gérer la navigation. La méthode canActivate redirige vers /logout si l'utilisateur n'est pas authentifié. Si l'utilisateur est authentifié, je compare ses rôles avec ceux requis pour la route. Si un rôle correspond, j'autorise l'accès ; sinon, je le refuse. Ce gardien s'assure que seuls les utilisateurs authentifiés et autorisés accèdent aux routes protégées.

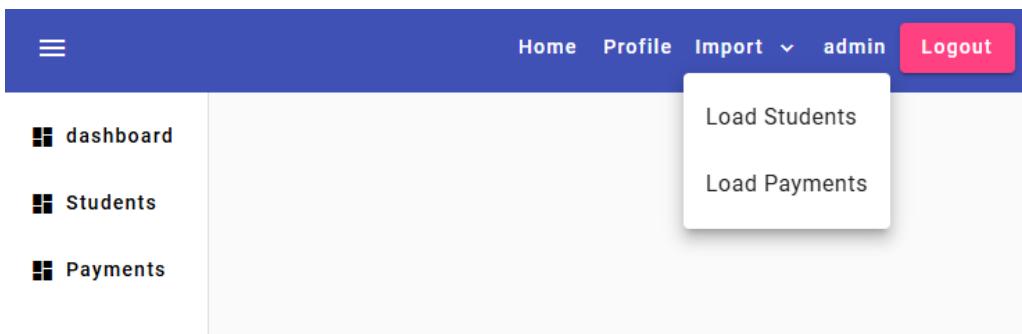
```
@Injectable()
export class AuthorizationGuard {

  no usages
  constructor(private authService : AuthService, private router : Router) {
  }
  no usages
  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): MaybeAsync<GuardResult> {
    if(this.authService.isAuthenticated){
      let requiredRoles = route.data['roles'];
      let userRoles : string[] = this.authService.roles;
      for (let role : string  of userRoles){
        if(requiredRoles.includes(role)){
          return true;
        }
      }
      return false;
    }
    else{
      this.router.navigateByUrl(url: '/logout')
      return false;
    }
  }
}
```

Le menu doit être masqué pour l'utilisateur user1 et visible pour l'administrateur. Pour cela, j'ajoute une vérification des rôles dans admin-template.component.html.

```
<button *ngIf="authService.roles.includes('ADMIN')" mat-button [matMenuTriggerFor]="importMenu">  
    <mat-icon iconPositionEnd>keyboard_arrow_down</mat-icon>  
    Import  
</button>
```

**Authentification avec l'utilisateur admin :** Lorsque l'administrateur se connecte, le menu est affiché.



**Authentification avec l'utilisateur user1 :** Lorsque user1 se connecte, le menu est masqué.



Dans le fichier payment.component.ts, j'ai créé le composant PaymentsComponent pour afficher les données de paiement provenant du backend en utilisant Angular Material. Lors de l'initialisation (ngOnInit), j'envoie une requête GET à "http://localhost:8021/payments" pour récupérer les paiements. Si la requête réussit, je stocke les données dans payments et les assigne à dataSource en tant que MatTableDataSource. Ensuite, je configure la pagination et le tri avec MatPaginator et MatSort. En cas d'erreur, je la logue dans la console. Le tableau affiche les colonnes définies par displayedColumns, offrant une présentation structurée et interactive des paiements.

```

export class PaymentsComponent implements OnInit{
  public payments : any;
  public dataSource : any;
  public displayedColumns : string[] = ['id','date','amount','type','status', 'firstName'];

  @ViewChild(MatPaginator) paginator!: MatPaginator ;
  @ViewChild(MatSort) sort! : MatSort;
  no usages
  constructor(private http: HttpClient) {
  }
  no usages
  ngOnInit(): void {
    this.http.get(<url> "http://localhost:8021/payments")
      .subscribe( observerOnNext: {
        next : data : Object => {
          this.payments = data;
          this.dataSource = new MatTableDataSource(this.payments);
          this.dataSource.paginator = this.paginator;
          this.dataSource.sort = this.sort;
        },
        error : err => {
          console.log(err);
        }
      })
  }
}

```

Dans le fichier payment.component.html, j'ai utilisé Angular Material pour afficher les données de paiement dans un tableau interactif. Le tableau inclut des colonnes pour l'ID, la date, le montant, le type, le statut et le prénom de l'étudiant. J'ai ajouté des fonctionnalités de tri avec mat-sort-header et une pagination configurable avec mat-paginator. Cette configuration permet une visualisation claire, structurée et interactive des paiements récupérés du backend.

```

<div class ="container">
  <mat-card>
    <mat-card-header>
      <mat-card-title>
        Payments
      </mat-card-title>
    </mat-card-header>
    <mat-divider></mat-divider>
    <mat-card-content>
      <table matSort mat-table [dataSource]="dataSource" class="mat-elevation-z1">
        <!-- ID Column -->
        <ng-container matColumnDef="id">
          <th mat-header-cell *matHeaderCellDef mat-sort-header > ID </th>
          <td mat-cell *matCellDef="let element"> {{element.id}} </td>
        </ng-container>
        <!-- Date Column -->
        <ng-container matColumnDef="date">
          <th mat-header-cell *matHeaderCellDef mat-sort-header> Date </th>
          <td mat-cell *matCellDef="let element"> {{element.date}} </td>
        </ng-container>
        <!-- Amount Column -->
        <ng-container matColumnDef="amount">
          <th mat-header-cell *matHeaderCellDef mat-sort-header> Amount </th>
          <td mat-cell *matCellDef="let element"> {{element.amount}} </td>
        </ng-container>
        <!-- Type Column -->
        <ng-container matColumnDef="type">
          <th mat-header-cell *matHeaderCellDef mat-sort-header> Type </th>
          <td mat-cell *matCellDef="let element"> {{element.type}} </td>
        </ng-container>
      </table>
    </mat-card-content>
  </mat-card>
</div>

```

```

<!-- Status Column -->
<ng-container matColumnDef="status">
  <th mat-header-cell *matHeaderCellDef mat-sort-header> Status </th>
  <td mat-cell *matCellDef="let element"> {{element.status}} </td>
</ng-container>

<!-- Student Column -->
<ng-container matColumnDef="firstName">
  <th mat-header-cell *matHeaderCellDef mat-sort-header> Student </th>
  <td mat-cell *matCellDef="let element"> {{element.student.firstName}} </td>
</ng-container>

<tr mat-header-row *matHeaderRowDef="displayedColumns" ></tr>
<tr mat-row *matRowDef="let row; columns: displayedColumns;"></tr>
</table>

<mat-paginator [pageSizeOptions]=[5, 10, 20, 30, 100]>
  showFirstLastButtons
  aria-label="Select page of periodic elements"
</mat-paginator>

</mat-card-content>
</mat-card>

</div>

```

L'affichage :

Payments					
ID	Date	Amount	Type	Status	Student
1	2024-05-24	9086	CHECK	CREATED	Mohamed
2	2024-05-24	20427	CASH	CREATED	Mohamed
3	2024-05-24	10593	CASH	CREATED	Mohamed
4	2024-05-24	5912	TRANSFER	CREATED	Mohamed
5	2024-05-24	1639	TRANSFER	CREATED	Mohamed

Items per page: 5 | 1 - 5 of 40 | < < > >|

# CONCLUSION

En conclusion, ce TP m'a offert l'opportunité de concevoir et de déployer une application web complète en utilisant Angular pour le frontend et Spring pour le backend. À travers ce projet, j'ai créé une application permettant la gestion des paiements des étudiants, incluant l'administration détaillée et sécurisée des transactions financières dans un cadre éducatif.

La mise en place d'une application Angular interagissant avec un backend simulé m'a permis de mieux comprendre la gestion et la manipulation des données dynamiques, essentielles pour les interactions entre le frontend et le backend. Ensuite, la création des composants backend avec Spring, y compris la définition des entités JPA, la mise en place de l'architecture RESTful, m'a donné une expérience pratique de bout en bout, allant du développement à la sécurisation des transactions.

Enfin, le développement de l'interface utilisateur avec Angular, intégrant Angular Material pour un design élégant et moderne, m'a permis de renforcer mes compétences en conception d'interfaces utilisateur intuitives et fonctionnelles.

Ce projet m'a donc permis de renforcer mes compétences en développement full-stack, en utilisant des technologies modernes pour créer des applications web interopérables et robustes, répondant aux besoins administratifs et financiers dans le secteur éducatif.