

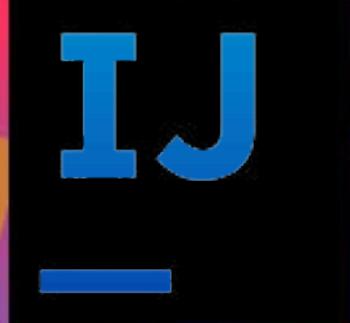
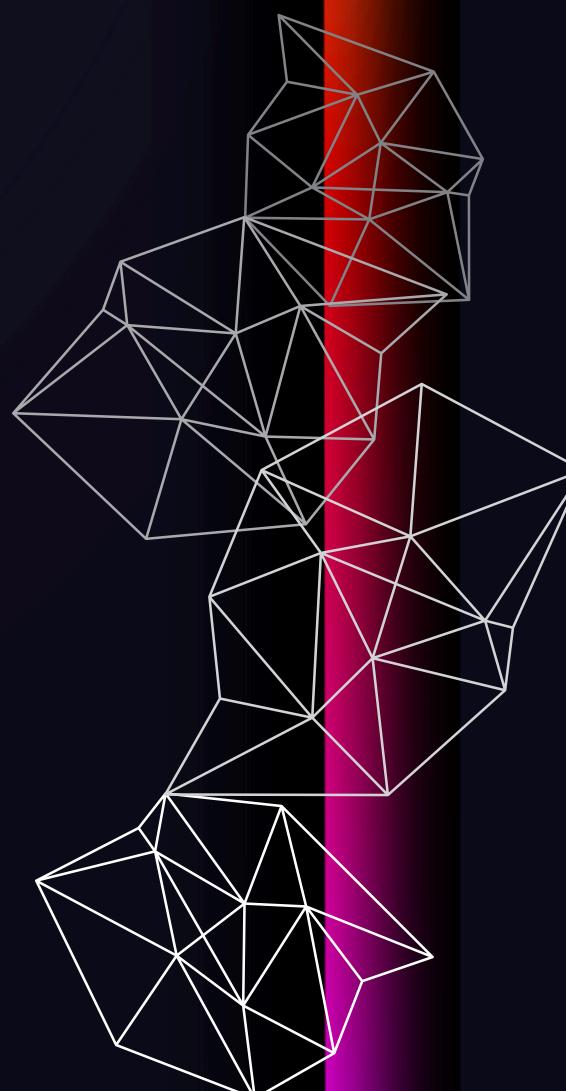
RAPPORT TP4

SPRING DATA JPA

HIBERNATE

Réalisé par :
Bouchama Hajar

Demandé par :
Mohamed Youssfi



Introduction

Dans le cadre de notre activité pratique, nous avons entrepris le développement d'une application web complète basée sur Angular et Spring pour la gestion des paiements des étudiants. Cette initiative combine la puissance de deux technologies de pointe pour offrir une solution robuste et efficace pour les besoins de gestion financière dans le domaine de l'éducation.

La première partie de notre projet a consisté à créer une application web Angular permettant la gestion des produits, avec un backend utilisant Json-server pour la gestion des données. Cette étape nous a permis de mettre en place les bases de notre système et de comprendre les principes fondamentaux de la création d'une interface utilisateur dynamique et réactive grâce à Angular.

Ensuite, nous avons abordé la seconde partie de notre projet, qui était de développer la partie backend avec Spring. Notre objectif était de créer un environnement sécurisé et performant pour la gestion des paiements des étudiants. Cette partie impliquait la création d'entités JPA, la mise en place d'interfaces JPRepository basées sur Spring Data, la génération de données aléatoires pour simuler des scénarios réels, et la création de services RESTful pour exposer les fonctionnalités essentielles de notre système.

En parallèle, nous avons développé la partie frontend de notre application en utilisant Angular avec Angular Material pour le design. Cette phase nous a permis de concevoir une interface utilisateur intuitive et conviviale, tout en mettant en place un système d'authentification robuste pour garantir la sécurité des données et des transactions.

Enfin, nous avons intégré la sécurité dans notre système en utilisant Spring Security et Json Web Token pour sécuriser à la fois le backend et le frontend. Cette étape était cruciale pour assurer la confidentialité et l'intégrité des données des étudiants et des transactions financières.

Dans ce rapport, nous détaillerons chaque étape de

Enoncé !

Première Partie (First Demo) :

- <https://www.youtube.com/watch?v=ZdcqRYIILgE>

Deuxième Partie :

Créer une application web Angular qui permet de gérer des produits avec un backend basé sur Json-server

- Part 1- <https://www.youtube.com/watch?v=Bq-vewCZk-o>
- Part 2-<https://www.youtube.com/watch?v=h0zPn2d4fGI&authuser=0>
- Part 3 - <https://www.youtube.com/watch?v=ejQg0Ugm9s8>
- Part 4- <https://www.youtube.com/watch?v=ZWQtLaRM49o&authuser=0>
-

Troisième Partie : Spring Angular

Objectif :

Développer une application qui permet de gérer le paiement des étudiants. Chaque étudiant peut effectuer plusieurs paiements

- Chaque étudiant est défini par son : id, firstName, lastName, email, sa filière, sa photo,
- Chaque Paiement est défini par son id, son code, sa date, son type (CASH, CHECK, TRANSFER), son status (CREATED, VALIDATED, REJECTED), file (fichier pdf représentant le reçu de paiement)

A: Développer et Tester la partie Backend avec Spring. :
<https://www.youtube.com/watch?v=oTuAXfD2M1g>

1. Créer les entités JPA
2. Créer les interfaces JPRepository basées sur Spring Data
- 3 . Générer des données aléatoires concernant quelques étudiants et pour chaque étudiants des payements
4. Créer une Web service RESTful (ResController) qui permet d'exposer les fonctionnalités suivantes :
 - Consulter tous les payements
 - Consulter un payement sachant son id
 - Consulter les payement d'un type donné
 - Consulter les payements d'un status donné
 - Consulter les payements d'un étudiant donné
 - Consulter les payements d'une filière donnée
 - Consulter tous les étudiants
 - Consulter les étudiants d'une filière donnée
 - Consulter un étudiant sachant son code
- Effectuer un nouveau payement avec les données et le reçu de payement au format pdf

- Mettre à jour le status d'un payement
- Consulter le reçu d'un payement (fichier pdf)

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

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

B : Développer la partie frontend en utilisant Angular avec Angular Material pour la partie design : <https://www.youtube.com/watch?v=QqmoMDGr3Ww>

1. Créer un projet angular
2. Intégrer Angular Material
3. Créeer une page template contenant un Toolbar avec une barre de menu et un Side Menu
4. Créeer les différents component de l'application

5. Créeer un système d'authentification simple qui oblige l'utilisateur à s'authentifier avant d'accéder à l'application. Dans un premier temps, les utilisateurs et les rôles qui ont le droit d'accéder à l'application sont stockés de manière statique dans le service d'authentification. Protéger les routes par Un Guard d'authentification et un Guard pour les autorisations

6. Créeer les components fonctionnels de l'application :

- Afficher les payements avec une Pagination frontend
- Afficher et chercher les étudiants
- Afficher le Dashboard d'un étudiant (Infos et Payements)
- Ajouter un nouveau payement en uploadant le reçu de payement
- Consulter le détail d'un payement
- Mettre à jour le status d'un payement

C : Sécurité avec Spring Security et Json Web Token

- Sécuriser le backend
- Sécuriser le front end

Code et explications :

première et deuxième Partie

1. telechargement et Installeation

télécharger et installer nodejs en utilisant la commande suivante :

https://nodejs.org/en

Verifier l'installation avec cette commande dans votre terminal : **npm --version**

installer le serveur de json avec cette ligne de commande :

npm install json-server

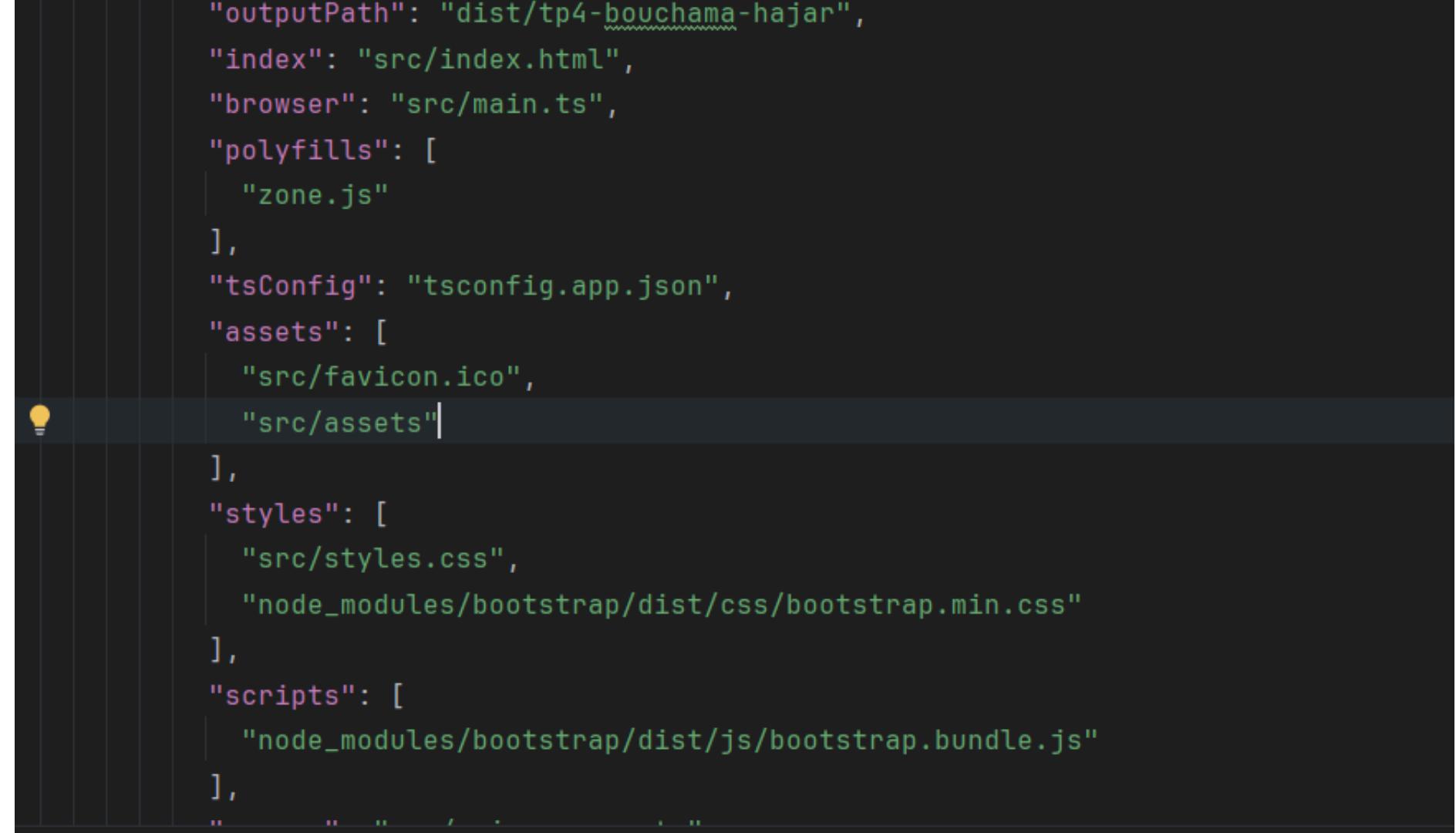
Dans votre cmd vous tapez la commande suivant pour installer Angular : **npm install -g @angular/cli**

Vous ouvrez le cmd de votre dossier ou vous souhaitez créer le projet. Et vous tapez cette commande pour créer le projet angular : **ng new nom-de-projet**

Vous ouvrez votre projet dans intelliJ ide et dans le terminal vous executer la commande : **ng serve pour démarrer votre projet.**

Il faut installer bootstrap dans votre terminal vous executer cette ligne de commade :
npm i bootstrap bootstrap-icons

Ensute dans votre fichier angular.json vous allez ver styles :[] et scripts : [] et vous ajouter les liens indiquer dans le scrin ci-dessous



```
"outputPath": "dist/tp4-bouchama-hajar",
"index": "src/index.html",
"browser": "src/main.ts",
"polyfills": [
  "zone.js"
],
"tsConfig": "tsconfig.app.json",
"assets": [
  "src/favicon.ico",
  "src/assets"
],
"styles": [
  "src/styles.css",
  "node_modules/bootstrap/dist/css/bootstrap.min.css"
],
"scripts": [
  "node_modules/bootstrap/dist/js/bootstrap.bundle.js"
],
```

styles.css on ajoute la ligne suivante :

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

dans cette partie nous créons **le style html** de notre application dans le fichier **app.component.html** , commenant par la bar de navigation qui fonctionne avec des routes dans notre application Angular :

```
    ✓  home
        home.component.css
        home.component.html
        home.component.spec.ts
        home.component.ts
```

ng g c home

```
    ✓  new-product
        new-product.component.css
        new-product.component.html
        new-product.component.spec.ts
        new-product.component.ts
```

ng g c products

```
    ✓  products
        products.component.css
        products.component.html
        products.component.spec.ts
        products.component.ts
```

ng g c new-product

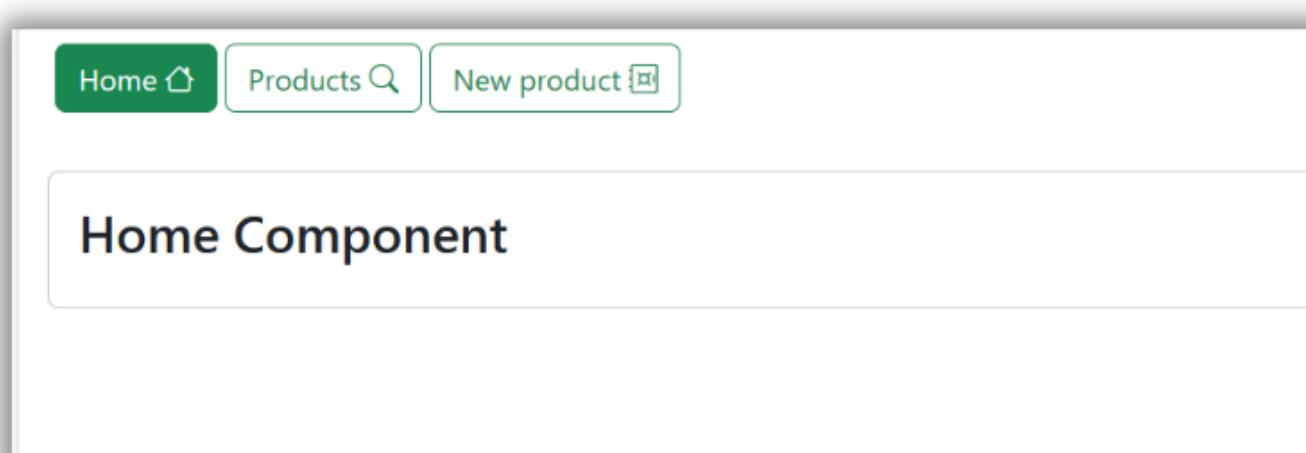
Ces commandes permettent de créer un composant (component) pour chaque bouton dans Angular. Chaque commande crée un composant avec le nom spécifié (home, products, ou new-product). Les composants sont créés dans le répertoire src/app de votre projet Angular.

```
✓ 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},
]
```

app.routes.ts

ici on va créer les routes suivants : qui seront utiliser dans l'argument **routerLink=""**, l'affichage sera comme suite :



app.component.ts

On va créer le composant **AppComponent** ci-dessous est le composant racine de l'application Angular, responsable de la mise en page générale et de la navigation entre différentes vues. Ce composant gère une liste d'actions de navigation représentées par un tableau d'objets contenant des titres, des routes et des icônes.

```
@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;
```

Il définit également une méthode `setCurrentAction()` pour mettre à jour l'action actuelle lorsqu'un bouton est cliqué.

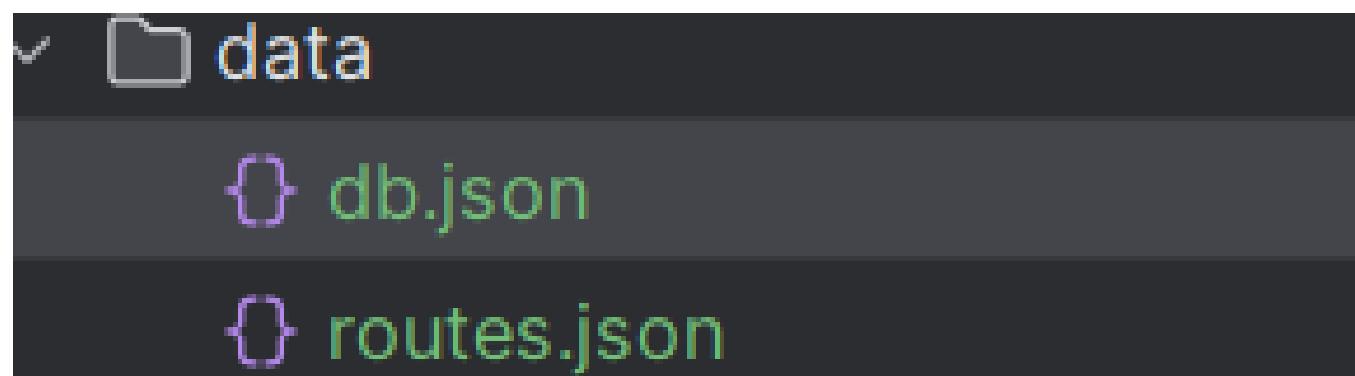
```
1+ usages new *
setCurrentAction(action: any) : void {
  this.currentAction = action;
}
```

Le fichier de vue `app.component.html` associé à ce composant contient une barre de navigation `<nav>` avec des boutons générés dynamiquement à l'aide de `*ngFor`. Chaque bouton est lié à une route spécifique grâce à `routerLink`. La classe des boutons est déterminée dynamiquement en fonction de l'action actuelle. Enfin, la balise `<router-outlet>` est utilisée pour afficher les vues des différentes routes de l'application.

Ce composant utilise la directive `*ngFor` pour itérer sur une liste d'actions de navigation et générer un bouton pour chaque action. La directive `routerLink` est utilisée pour lier chaque bouton à une route spécifique. La classe des boutons est déterminée dynamiquement en fonction de l'action actuelle, ce qui permet de mettre en évidence l'action actuellement sélectionnée. Enfin, la balise `<router-outlet>` est utilisée pour afficher les vues des différentes routes de l'application, en fonction de l'action sélectionnée par l'utilisateur.

```
<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>
```

Pour créer ma base de données je dois créer un dossier dans mon projet appelé data dans lequel je dois créer un fichier appelé db.json où je vais spécifier mes données de produits ;

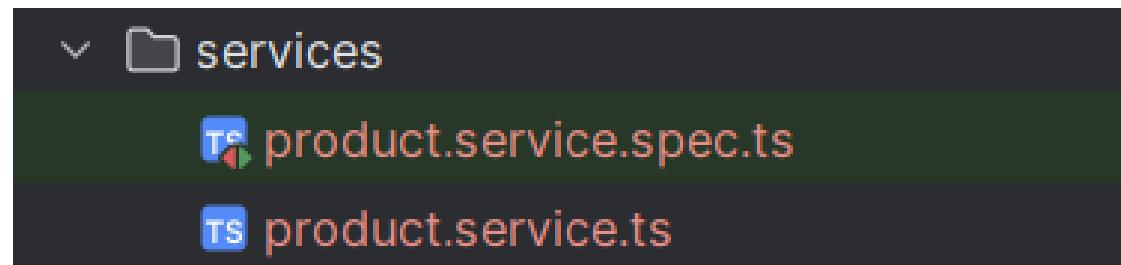


```
/*
  "products": [
    {
      "id": "1",
      "name": "Computer",
      "price": 5000,
      "checked": true
    },
    {
      "id": "2",
      "name": "Printer",
      "price": 7000,
      "checked": false
    }
]
```

```
1 [   {
2   "id": "1",
3   "name": "Computer",
4   "price": 5000,
5   "checked": true
6   },
7   {
8   "id": "2",
9   "name": "Printer",
10  "price": 7000,
11  "checked": false
12  }
13 ]
14 ]
```

Affichage des produits

Nous allons créer un service en utilisant la commande : **ng g s services/products**.



```
import {HttpClient} from "@angular/common/http";
import {Observable} from "rxjs";
import {Product} from "../model/product.model";

1+ usages
@ Injectable({
  providedIn: 'root'
})
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}` );

```

Le ProductService est un service Angular injectable qui gère les opérations CRUD (Create, Read, Update, Delete) liées aux produits. Il utilise HttpClient pour effectuer des requêtes HTTP vers un serveur backend. Ce service offre les méthodes suivantes :`getProducts()` : pour récupérer la liste des produits

- `checkProducts()` : pour vérifier ou décocher un produit
- `deleteProducts()` : pour supprimer un produit

Chaque méthode retourne un observable qui peut être souscrit dans d'autres composants. Cela permet d'obtenir les données ou de gérer les erreurs de manière asynchrone. Le service est annoté avec `@Injectable({ providedIn: 'root' })`, ce qui le rend disponible pour l'injection de dépendances dans toute l'application, sans nécessiter d'importations supplémentaires.

Products.component.ts :

Le composant **ProductsComponent** est un composant Angular qui gère l'affichage et la manipulation des produits. Il utilise le service ProductService pour interagir avec les données des produits. Dans sa méthode `ngOnInit()`, il appelle `getProducts()` pour récupérer la liste des produits au chargement du composant. La méthode `getProducts()` utilise le service ProductService pour récupérer les produits via une requête HTTP, puis met à jour la liste des produits. Les méthodes `handleCheckProduct()` et `handleDelete()` sont des gestionnaires d'événements qui interagissent avec le service ProductService pour vérifier ou supprimer un produit, respectivement. Après chaque opération réussie, elles mettent à jour la liste des produits en appelant à nouveau `getProducts()`. Ce composant utilise également des fonctionnalités d'Angular telles que HttpClientModule, NgForOf et AsyncPipe pour gérer les requêtes HTTP et le rendu asynchrone des données.

```

import { Component, OnInit } from '@angular/core';
import {AsyncPipe, NgForOf} from "@angular/common";
import {HttpClient, HttpClientModule} from "@angular/common/http";
import {ProductService} from "../services/product.service";
import {Product} from "../model/product.model";
import {Observable} from "rxjs";

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();
}

}
1+ usages
getProducts() : void {

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

//this.products$=this.productService.getProducts();
}

```

```

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

```

```

1+ usages
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);
            }
        })
}

```

products.component.htm

Ce balisage HTML décrit la présentation d'une liste de produits sous forme de tableau. Chaque produit est représenté dans une ligne de tableau comprenant trois colonnes : "Nom", "Prix" et "Coché". Les produits sont extraits d'une liste nommée "products" en utilisant la directive *ngFor. Chaque produit affiche son nom et son prix dans les deux premières colonnes. Un bouton est utilisé pour cocher ou décocher le produit, tandis qu'un autre bouton permet de le supprimer de la liste. Les icônes des boutons varient dynamiquement en fonction de l'état de vérification du produit, en utilisant des classes conditionnelles basées sur la propriété "checked" de chaque produit.

```

<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>

```

Name	Price	Checked
Computer	5000	
Printer	7000	

Ajouter un produit :

product.service.ts :

Dans ce fichier, nous avons la méthode **saveProduct()** du service `ProductService`. Cette méthode utilise le protocole HTTP pour envoyer une requête POST au serveur backend, spécifiquement à l'URL **http://localhost:8089/products**, pour enregistrer le produit fourni en tant que paramètre. Le produit est envoyé au format JSON.

```

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

```

new-product.component.ts

Le fichier **new-product.component.ts** définit le composant Angular `NewProductComponent`, chargé de la création de nouveaux produits. Il importe les modules et services Angular nécessaires comme `Component`, `OnInit`, `FormBuilder`, `FormGroup`, `Validators`, etc. Ce composant est annoté avec `@Component` pour définir ses métadonnées telles que le sélecteur, le modèle de fichier HTML associé et les styles. Son constructeur injecte les services `'FormBuilder'` et `'ProductService'`. La méthode `'ngOnInit'`, héritée de l'interface `'OnInit'`, initialise le formulaire de produit avec des champs tels que le nom, le prix et un indicateur de vérification. Enfin, la méthode `saveProduct` est déclenchée lors de la soumission du formulaire. Elle récupère les valeurs du formulaire, crée un objet `'Product'`, puis utilise `'ProductService'` pour enregistrer le produit, en traitant les réponses en fonction du succès ou de l'erreur de la requête.

```

import {ProductService} from "../services/product.service";
import {Product} from "../model/product.model";
import {JsonPipe} from "@angular/common";

1+ usages
@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),
    });
  }

  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);
      }
    });
  }
}

```

Product.service.ts :

"saveProduct()" du service ProductService enregistre un produit sur le serveur backend via une requête POST JSON.

```

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

```

new-product.component.html

ce fichier définit le formulaire de création de produit avec des éléments HTML classiques et des classes Bootstrap. Les champs du formulaire, liés aux propriétés du composant, déclenchent la méthode 'saveProduct()' lors de la soumission. Des validations côté client garantissent la saisie correcte des informations.

```

<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>

```

localhost:4200 indique
{"id": "d288", "name": "Laptop", "price": "12000", "checked": true}

OK

Chercher un produit :

product.component.ts:

La méthode searchProducts() du composant product.component.ts utilise le service productService pour rechercher des produits en fonction du mot-clé fourni. Elle s'abonne à la réponse du service et met à jour la variable this.products avec les résultats de la recherche, ce qui déclenche une mise à jour de l'affichage des produits.

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

product.service.ts:

La méthode searchProducts() du fichier product.service.ts effectue une recherche de produits basée sur un mot-clé en envoyant une requête HTTP GET à l'API produits. Le paramètre name_like inclus dans l'URL permet à l'API de filtrer les produits dont le nom correspond au mot-clé recherché, offrant ainsi une recherche flexible.

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

product.component.html :

Une barre de recherche de produits est implémentée dans product.component.html à l'aide d'un champ de saisie texte et d'un bouton de recherche. Le champ de saisie est lié à une variable keyword dans le composant TypeScript, permettant une saisie en temps réel. Cliquer sur le bouton de recherche déclenche la fonction searchProducts() du composant TypeScript pour filtrer les produits en fonction du 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>
```

The screenshot shows the product component's user interface. At the top, there is a navigation bar with three buttons: "Home" (with a house icon), "Products" (with a magnifying glass icon), and "New product" (with a plus icon). Below the navigation bar is a search bar with a placeholder "Search" and a magnifying glass icon. Underneath the search bar is a table with three columns: "Name", "Price", and "Checked". The table contains three rows of data: Computer (Price: 5000, Checked: checked), Printer (Price: 7000, Checked: checked), and Smart phone (Price: 3000, Checked: unchecked). To the right of each row are two small red trash can icons.

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

La pagination :

The screenshot shows the product component's user interface with pagination. It features a search bar at the top and a table below it. The table has the same structure as the previous screenshot: "Name", "Price", and "Checked" columns. The data rows are the same: Computer (Price: 5000, Checked: checked), Printer (Price: 7000, Checked: checked), and Smart phone (Price: 3000, Checked: unchecked). To the right of each row are two small red trash can icons. At the bottom of the table, there is a pagination control with three numbered buttons: 1, 2, and 3. The button for page 1 is highlighted in green, while the others are grey.

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

product.service.ts:

La méthode `getProductsByPage()` récupère une page de produits depuis l'API. Elle utilise une requête HTTP GET en spécifiant les paramètres `_page` et `_limit` dans l'URL pour indiquer la page souhaitée et le nombre de produits à afficher. L'option `observe:'response'` permet de récupérer 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.

```
1+ usages
public getProducts(page :number=1, size:number=4) : Observable<HttpResponse<Object...> {
    return this.http.get(url: 'http://localhost:8089/products?_page=${page}&_limit=${size}', options: {observe:'response'});
}
```

La méthode `searchProducts()` du fichier `product.service.ts` a été enrichie d'un paramètre `keyword` pour spécifier le mot-clé à rechercher. Ce paramètre est utilisé pour inclure le mot-clé dans l'URL de la requête HTTP, permettant ainsi de filtrer les produits en fonction de la recherche effectuée.

```
1+ usages
public searchProducts(keyword:string="",page :number=1, size:number=4) : Observable<HttpResponse<Object...> {
    return this.http.get(url: 'http://localhost:8089/products?name_like=${keyword}&_page=${page}&_limit=${size}', options: {observe:'response'});
}
```

product.component.ts :

Cette méthode est appelée dans le composant `product.component.ts` pour récupérer une page de produits à afficher. Elle utilise le service `productService` pour appeler la méthode `getProducts` avec les numéros de page et la taille de page actuels (`this.currentPage` et `this.pageSize`). Lorsque la réponse est reçue avec succès, la méthode `subscribe` est utilisée pour traiter la réponse. La liste des produits est extraite du corps de la réponse (`resp.body as Product[]`), et le nombre total de produits est extrait de l'en-tête de réponse `x-total-count`. En utilisant ces informations, le nombre total de pages est calculé en fonction de la taille de la page, ce qui permet de déterminer la pagination. Cette méthode met à jour les propriétés `products` (liste des produits), `totalPages` (nombre total de pages) et `currentPage` en conséquence.

```
getProducts(): void {
    this.productService.getProducts(this.currentPage, this.pageSize)
        .subscribe(observerOrNext: {
            next: (resp: HttpResponse<Object>) => {
                this.products = resp.body as Product[];
                let totalProducts: number = parseInt(resp.headers.get('x-total-count')!)!;
                this.totalPages = Math.floor(totalProducts / this.pageSize);
                if (totalProducts % this.pageSize != 0) {
                    this.totalPages = this.totalPages + 1;
                }
            },
            error: err => {
                console.log(err);
            }
        });
}
```

La méthode `searchProducts()` du fichier `product.component.ts` a été modifiée pour transmettre le paramètre `keyword` à la méthode `searchProducts()` du service `productService`. Cela permet d'effectuer une recherche de produits paginée et filtrée par le mot-clé spécifié.

```
1+ usages
searchProducts(): void {
    this.productService.searchProducts(this.keyword, this.currentPage, this.pageSize)
        .subscribe(observerOrNext: {
            next: (resp: HttpResponse<Object>) => {
                this.products = resp.body as Product[];
                let totalProducts: number = parseInt(resp.headers.get('x-total-count')!)!;
                this.totalPages = Math.floor(totalProducts / this.pageSize);
                if (totalProducts % this.pageSize != 0) {
                    this.totalPages = this.totalPages + 1;
                }
            },
            error: err => {
                console.log(err);
            }
        });
}
```

product.component.html:

Le code HTML et Angular génère une liste de boutons de pagination dynamiquement. La liste n'est affichée que si la variable totalPages est définie et supérieure à zéro. Chaque bouton correspond à une page et son style change en fonction de la page actuelle, indiquant clairement la page active à l'utilisateur.

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

Modifier un produit :

product.component.html

Le fichier product.component.html intègre désormais un bouton d'édition pour chaque produit. Ce bouton déclenche la fonction handleEdit() définie dans le fichier product.component.ts, permettant de lancer l'édition du produi

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

product.component.ts:

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

La fonction handleEdit() permet la navigation vers la page de modification (editProduct) avec l'identifiant du produit à éditer.

app.routes.ts :

Afin de gérer la nouvelle route d'édition des produits, la définition editProduct/:id a été ajoutée au fichier app.routes.ts. Cette route est liée au composant EditProductComponent nouvellement créé.

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

product.service.ts :

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

getProductById() effectue une requête HTTP GET vers le backend afin d'obtenir les détails d'un produit spécifique. Elle construit l'URL de l'endpoint correspondant en utilisant l'identifiant du produit fourni en paramètre. Une fois la réponse reçue, elle retourne un observable contenant les informations du produit, lesquelles peuvent ensuite être utilisées par le composant edit-product.

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

edit-product.component.ts !

Dans la méthode ngOnInit(), nous initialisons le composant lors de son chargement. Nous utilisons ActivatedRoute pour extraire l'identifiant du produit à partir de l'URL actuelle avec snapshot.params['id']. Ensuite, nous appelons la méthode getProductById() du service ProductService pour récupérer les détails du produit correspondant. Une fois la réponse reçue, nous utilisons subscribe() pour écouter les données émises par l'observable. Lorsqu'elles sont reçues, nous construisons un formulaire réactif avec FormBuilder, en liant chaque champ du formulaire aux propriétés du produit récupéré. Les champs name et price sont soumis à des validations : name est requis et price doit avoir une valeur minimale de 100.

```
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) => {
            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);
        }
    });
}
```

Cette méthode updateProduct() est responsable de la mise à jour des informations d'un produit dans l'application Angular. Tout d'abord, elle récupère les valeurs actuelles du formulaire de produit à l'aide de this.productFormGroup.value, où productFormGroup est un objet de type FormGroup qui représente le formulaire réactif associé à l'édition du produit.

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

edit-product.component.html :

Ce fichier HTML représente le formulaire de modification d'un produit dans l'application. Il est structuré avec une carte Bootstrap qui affiche l'ID du produit en cours d'édition dans l'en-tête. Le contenu de la carte est affiché de manière conditionnelle pour éviter les problèmes d'affichage lors de l'initialisation du formulaire. Le formulaire lui-même est associé au FormGroup productFormGroup et contient des champs pour le nom, le prix et l'état de vérification du produit. Un bouton "Save" est inclus pour soumettre les modifications, et il est désactivé de manière conditionnelle si le formulaire est invalide. En bref, ce fichier fournit une interface utilisateur claire et fonctionnelle pour modifier les 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>
```

Product ID = 3
Name
Smart phone
Price
30001
Checked <input checked="" type="checkbox"/>
Save

app-state.service :

```

no usages
@Injectable({
  providedIn: 'root'
})
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, ...
  }
}

```

Ce service gère l'état des produits via la propriété productsState, incluant divers champs comme products, keyword, totalPages, pageSize, currentPage, totalProducts, status, et errorMessage pour les messages d'erreur. Pour mettre à jour cet état, le service utilise la méthode setProductState(state), qui fusionne l'état actuel avec un nouvel objet state à l'aide de l'opérateur de propagation (...), permettant ainsi une gestion efficace de l'état des produits.

products.component.ts :

ProductsComponent prend en charge l'affichage et les interactions liées aux produits dans votre application. Il utilise divers modules et services pour effectuer des opérations telles que la recherche, la gestion des actions comme la sélection, la suppression, la navigation et l'édition des produits.

```

import { Component, OnInit } from '@angular/core';
import { ProductService } from '../product.service';
import { Router } from '@angular/router';
import { AppStateService } from '../app-state.service';

@Component({
  selector: 'app-products',
  templateUrl: './products.component.html',
  styleUrls: ['./products.component.css']
})
export class ProductsComponent implements OnInit {
  no usages
  constructor(private productService:ProductService,
    private router: Router,
    public appState: AppStateService)
  {}

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

  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}`);
  }
}

```

une interface utilisateur est créée pour visualiser et interagir avec une liste de produits dans une application Angular. Il inclut des fonctionnalités de recherche par mot-clé, d'affichage des produits sous forme de tableau avec des options de sélection, de suppression, d'édition et de 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>
          </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>
</div>
```

dashboard :

```
<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>
```

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

navbar :

Ce code définit un composant Angular appelé NavbarComponent, qui représente la barre de navigation de l'application. Il inclut des boutons pour accéder aux différentes sections de l'application et affiche un indicateur de chargement (spinner) 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>
```

app-errors :

Le composant Angular AppErrorsComponent affiche les erreurs de l'application lorsque le statut est "ERROR". Il utilise le service AppStateService pour obtenir les informations sur les erreurs à afficher.

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

```
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>
```

app-http.interceptor :

AppHttpInterceptor est un intercepteur HTTP qui ajoute un en-tête d'autorisation aux requêtes sortantes et contrôle l'affichage du spinner de chargement pendant la durée de la requête, grâce aux services AppStateService et 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();
        })
    );
}

```

loading.services :

le Angular LoadingService utilise un observable (isLoading\$) pour gérer l'état du spinner de chargement dans l'application. Il fournit des méthodes showLoadingSpinner() et hideLoadingSpinner() pour afficher et cacher le spinner en émettant des valeurs booléennes.

```

no usages
constructor() { }

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

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

```

login :

```


<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 ="us...
                </div>
                <div class="mb-3">
                    <label class="form-label">Password</label>
                    <input type="password" class="form-control" formControlName ...
                </div>
                <button class="btn btn-success"> Login</button>
            </form>
        DD
    </div>
</div>


```

login.component.ts

la modification dans le fichier login.component.ts consiste à créer un formulaire de connexion avec deux champs (nom d'utilisateur et mot de passe) et une méthode pour gérer la soumission du formulaire et afficher les valeurs saisies dans la console.

```

import { Component, OnInit } from '@angular/core';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';

@Component({
  selector: 'app-login',
  templateUrl: './login.component.html',
  styleUrls: ['./login.component.css']
})
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);
  }
}

```

```

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'}
];

```

Authentication

Username

Password

Login

admin-template

admin-template.component.html

Dans le fichier admin-template.component.html, le code est structuré de manière à ce que la barre de navigation et le tableau de bord 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>

```

```

<router-outlet>

</router-outlet>
```

```

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'}
];

```

Un test pour la connexion de l'admin et se dériger vers la page admin:

```

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

```

navbar.component.ts

```
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"}]
```

Créer des utilisateurs pour la connexion avec token jwt.

```
{
  "users": [
    {"id": "user1",
      "password": "1234",
      "roles": ["USER"],
      "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJ1c2VyMSIsImlhCI6MTUxNjI..."},

    {"id": "user2",
      "password": "1234",
      "roles": ["USER"],
      "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJ1c2VyMiIsImlhCI6MTUxNjI..."}
```

J'ai mis en place et configuré un service d'authentification en Angular, puis j'ai déplacé les fichiers concernés dans le package service pour une meilleure organisation. **ng g s auth**

auth.service.ts

J'ai défini authState pour stocker l'état initial de l'authentification de l'utilisateur, y compris les rôles et le token. J'ai également ajouté la méthode setAuthState pour mettre à jour cet état de manière dynamique, facilitant ainsi 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, ce qui améliore la gestion de l'authentification et la sécurité des données utilisateur.

```
C:\Users\h\TP Hajar Sys\TP4-bouchama-hajar>npm install jwt-decode
added 1 package, and audited 928 packages in 4s
121 packages are looking for funding
  run 'npm fund' for details
found 0 vulnerabilities

C:\Users\h\TP Hajar Sys\TP4-bouchama-hajar>
```

```

1+ usages
async login(username: string, password: string) {
  let user: any = await firstValueFrom(this.httpClient.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");
  }
}

```

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, elle renvoie une erreur "Bad credentials"

Login.component.ts

```

export class LoginComponent implements OnInit {
  formLogin!: FormGroup;
  errorMessage: undefined = undefined;

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

login.component.html

```


<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>


```

The screenshot shows a web application interface. At the top, there's a navigation bar with the text 'Authentication'. Below it, a pink header bar displays the error message 'Bad credentials'. The main content area contains a login form. It has two input fields: 'Username' with the value 'admin' and 'Password' with the value '****'. At the bottom of the form is a green button labeled 'Login'.

Affichage quel est l'Utilisateur dans navbar

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>
```

navbar.component.ts

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

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

Protégé les routes

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

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. Sinon, l'utilisateur est redirigé vers la page de connexion et l'accès à la route est refusé. Cette méthode garantit que seuls les utilisateurs authentifiés peuvent accéder aux routes protégées.

```
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;
  }
}
```

Pour afficher une page d'erreur lorsque les utilisateurs tentent d'accéder à des routes sans les autorisations nécessaires, j'ai créé le composant not-authorized.

ng g c not-authorized

not-authorized.component.html

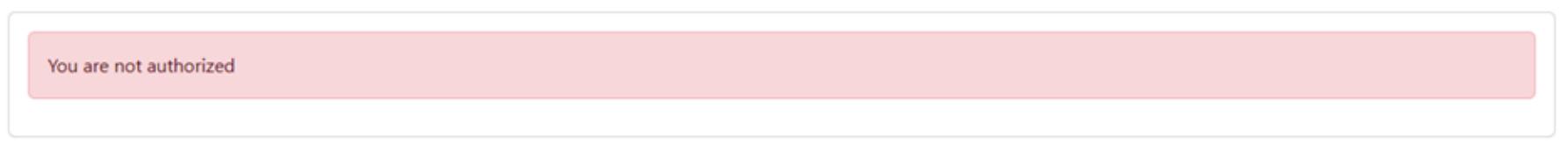
j'ai créé une page qui affiche le message **You are not authorized** pour signaler un accès refusé.

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

app.routes.ts

```
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}
    ]
  },
]
```

j'ai défini des routes sécurisées par AuthenticationGuard et AuthorizationGuard. La route notAuthorized redirige vers une page spécifique en cas d'accès non autorisé. AuthenticationGuard vérifie si l'utilisateur est connecté, tandis qu'AuthorizationGuard contrôle les permissions pour accéder aux routes administratives.



You are not authorized

products.component.html,

```
<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>
```

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 l'utilisateur a le rôle 'ADMIN'. Cela garantit que seuls les administrateurs peuvent modifier, supprimer ou vérifier les produits.

Troisième Partie

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

1. Créer les entités JPA

- **Payment**

```
package ma.bouchama.tp4partie3bouchamahajar.entities;

> import ...

25 usages
└@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;
  }
```

- **PaymentStatus**

```
package ma.bouchama.tp4partie3bouchamahajar.entities;

13 usages
public enum PaymentStatus
{
  2 usages
  CREATED, VALIDATED, REJECTED
}
```

- **PaymentType**

```
package ma.bouchama.tp4partie3bouchamahajar.entities;

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

- **Student**

```
package ma.bouchama.tp4partie3bouchamahajar.entities;

> import ...

16 usages
└@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**

```
package ma.bouchama.tp4partie3bouchamahajar.repository;

> import ...

8 usages
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**

```
package ma.bouchama.tp4partie3bouchamahajar.repository;

> import ...

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

3 . Générer des données aléatoires concernant quelques étudiants et pour chaque étudiants des payements

```
package ma.bouchama.tp4partie3bouchamahajar;

import ...

@SpringBootApplication
public class Tp4Partie3BouchamaHajarApplication {

    public static void main(String[] args) { SpringApplication.run(Tp4Partie3BouchamaHajarApplication.class, args);
        @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("Alaa").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();
                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()*20000))
                            .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 :

- Consulter tous les paiements

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

```
1 [   {
2     "id": 1,
3     "date": "2024-05-18",
4     "amount": 21000,
5     "type": "CHECK",
6     "status": "CREATED",
7     "file": null,
8     "student": {
9         "id": "8bc4b034-6aa5-4423-af3b-0ca506ebde10",
10        "firstName": "Mohamed",
11        "lastName": null,
12        "code": "112233",
13        "programId": "SDIA",
14        "photo": null
15    }
16 },
17 {
18     "id": 2,
19     "date": "2024-05-18",
20     "amount": 21000,
21     "type": "TRANSFER",
22     "status": "CREATED",
23     "file": null,
24     "student": {
25         "id": "8bc4b034-6aa5-4423-af3b-0ca506ebde10",
26         "firstName": "Mohamed".
27     }
28 }
```

- Consulter un paiement sachant son id

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

- Consulter les paiements d'un type donné

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

- Consulter les paiements d'un status donné

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

- Consulter les paiements d'un étudiant donné

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

- Consulter les paiements d'une filière donnée

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

- Consulter tous les étudiants

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

- Consulter les étudiants d'une filière donnée

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

- Consulter un étudiant sachant son code

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

- Effectuer un nouveau payement avec les données et le reçu de payement au format pdf

```
@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);
}
```

- Mettre à jour le status d'un payement

```
@PutMapping("/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);
}
```

- Consulter le reçu d'un payement (fichier pdf)

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

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

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

```
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-web</artifactId>
</dependency>

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

payment-rest-controller

- PUT** /payments/{id}/updateStatus
- GET** /payments
- POST** /payments
- GET** /students
- GET** /studentsByProgramId
- GET** /students/{code}
- GET** /students/{code}/payments
- GET** /payments/{id}
- GET** /payments/byType
- GET** /payments/byStatus
- GET** /payments/byProgramId
- GET** /paymentFile/{paymentId}

la mise à jour du statut d'un paiement avec l'ID 1

PUT /payments/{id}/updateStatus

Parameters

Name	Description
status * required	VALIDATED
string (query)	
id * 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": "384ec5ff-d1a9-4ac8-aef0-8bd6bb6178d",
    "firstname": "Najat",
    "lastname": null,
    "code": "112266",
    "programId": "GSIID",
    "photo": null
  }
}
```

Cancel Download

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
amount * required	35000
number(\$double) (query)	
type * required	CASH
string (query)	
date * required	2024-05-18
string(\$date) (query)	
studentCode * 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": "VALIDATED",
  "file": "file:///C:/Users/HP/Downloads/payments/cdff8223-5364-4962-bc26-97959a543af0.pdf",
  "student": {
    "id": "384ec5ff-d1a9-4ac8-aef0-8bd6bb6178d",
    "firstname": "Najat",
    "lastname": null,
    "code": "112266",
    "programId": "GSIID",
    "photo": null
  }
}
```

Cancel Reset Download

récupérer le fichier associé à un paiement

GET /payments/{id}/file

Parameters

Name	Description
id * required	41
integer(\$int64) (path)	

Execute Clear

Responses

Curl

```
curl -X 'GET' \
'http://localhost:8021/payments/41/file' \
-H 'Accept: application/pdf'
```

Request URL

```
http://localhost:8021/payments/41/file
```

Après cela on peut voir notre fichier pdf.

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

```
package ma.bouchama.tp4partie3bouchamahajar.services;

> import ...

3 usages
@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;
    }
    1 usage
}
```

```
1 usage
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);
}
```

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

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

PaymentRestController : paymentDTO :

```
@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);
}

import lombok.*;
import ma.bouchama.tp4partie3bouchamahajar.services.PaymentService;
import ma.bouchama.tp4partie3bouchamahajar.services.StudentService;
import ma.bouchama.tp4partie3bouchamahajar.services.PaymentRepository;
import ma.bouchama.tp4partie3bouchamahajar.services.StudentRepository;
import ma.bouchama.tp4partie3bouchamahajar.services.PaymentStatus;
import ma.bouchama.tp4partie3bouchamahajar.services.PaymentType;
import java.time.LocalDate;
no usages
@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 la partie frontend en utilisant Angular avec Angular Material pour la partie design

1. Créer un projet angular

Créer un dossier frontend-angular : dans ce dernier on crée un projet angular

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

- `ng add @angular/material`

2. Intégrer Angular Material

- `ng g c admin-template`

3. Créer une page template contenant un Toolbar avec une barre de menu et un Side Menu

Admin-template.component.html :

Ce code utilise Angular Material pour créer une barre d'outils avec plusieurs éléments interactifs. La balise `<mat-toolbar>` est utilisée avec la couleur primaire définie. À l'intérieur, il y a un bouton avec une icône de menu qui, lorsqu'il est cliqué, déclenche la fonction `toggle()` pour ouvrir ou fermer un tiroir latéral. Ensuite, il y a trois boutons qui redirigent vers différentes routes de l'application (`/home`, `/profile`, `/logout`). Le quatrième bouton est un bouton `matMenuTrigger` qui ouvre un menu déroulant contenant deux options (`Load Students` et `Load Payments`), chacune redirigeant vers une autre route 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>
```

Ce code crée un conteneur de tiroirs à l'aide des composants `<mat-drawer-container>`, `<mat-drawer>`, et `<mat-drawer-content>`. À l'intérieur du tiroir (`<mat-drawer>`), il y a une liste (`<mat-list>`) de boutons (`<mat-list-item>`) qui redirigent vers différentes parties de l'application lorsqu'ils sont cliqués (`/dashboard`, `/students`, `/payments`). Le contenu principal de l'application est affiché dans `<mat-drawer-content>` à travers `<router-outlet>`, ce qui permet de charger dynamiquement les composants correspondant aux différentes routes de l'application.

```
<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>
  </mat-drawer>
<router-outlet>
```

```

<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

```

jar\frontend-angular> ng g c payments
UCHAMA-HAJAR\frontend-angular> ng g c home
Hajar\frontend-angular> ng g c load-students
Hama-Hajar\frontend-angular> ng g c students
Na-hajar\frontend-angular> ng g c load-payments
jar\frontend-angular> ng g c dashboard

```

nous avons défini les routes qui spécifient les chemins et les composants correspondants pour la navigation dans une application Angular. Par exemple, lorsque le chemin "/home" est accédé, le composant HomeComponent est chargé. De même, "/profile" charge ProfileComponent et "/logout" charge LogoutComponent, etc. Cela permet à l'application de charger les composants appropriés en fonction de l'URL demandée par l'utilisateur.

```

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}
  ]},
];

```

hom.component.html :

```

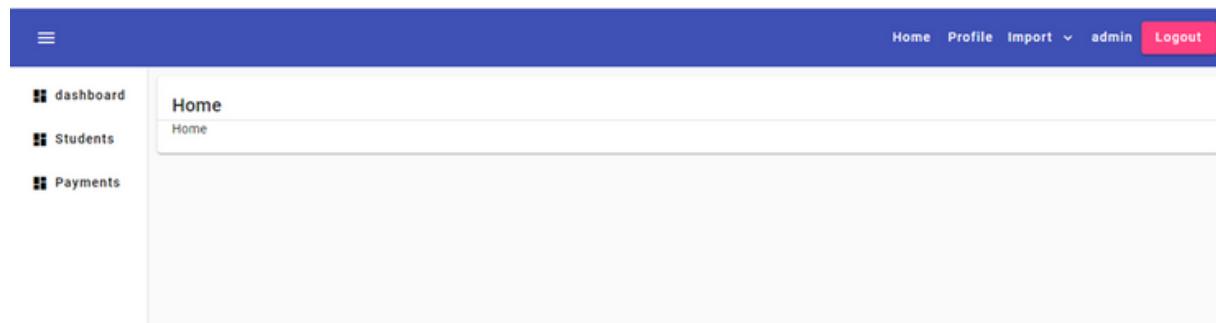
<mat-card>
  <mat-card-header>
    <mat-card-title>
      Home
    </mat-card-title>
  </mat-card-header>
  <mat-divider></mat-divider>
  <mat-card-content>
    Home
  </mat-card-content>
</mat-card>

```

container.css :

```
html, body { height: 100%; }
body { margin: 0; font-family: Roboto, "Helvetica Neue",
      sans-serif; }

.container{
  padding: 10px;
}
```



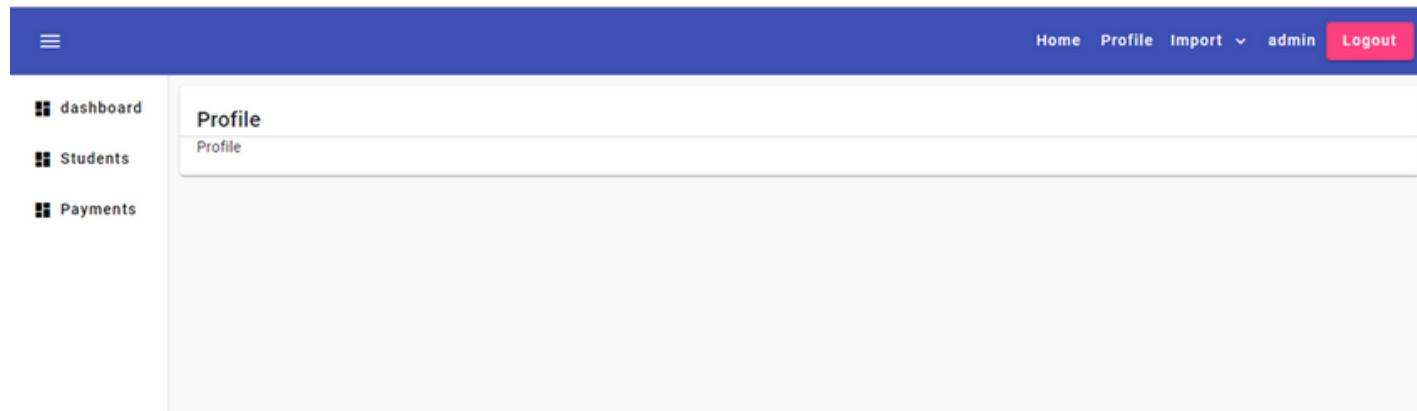
load-payments.component.html

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

load-students.component.html

```
<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>
```

profile.component.html



```
<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>
```

app.component.html :

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

Dashboard.component.html

```
<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>

</div>
```

Login :

```
<div class="container">
  <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>
```

```
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")
  }
}
```

Authentication

The form consists of two stacked input fields. The top field is labeled 'Username or Email' and the bottom field is labeled 'Password'. To the right of the bottom field is a blue rectangular button with the word 'Login' in white.

Et pour faire un système d'authentification basique on va créer service avec:
ng g s sevices/auth

auth.service.ts

```
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) { }

  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")
  }
}
```

Admin-template.ts

```
> import ...

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()
  }
}
```

créer un guard

```
ng g g guards/auth
```

Auth.guard.ts

```
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;
  }
}
```

AuthGuards :

```
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('/logout')
      return false;
    }
  }
}
```

SIGN IN

localhost:4200/admin/payments

Home Profile Import admin Logout

Payments

	ID	Date	Amount	Type	Status	Student
	11	2024-05-31	7680	CASH	CREATED	Imane
	12	2024-05-31	14234	CHECK	CREATED	Imane
	13	2024-05-31	2304	CASH	CREATED	Imane
	14	2024-05-31	8869	DEPOSIT	CREATED	Imane
	15	2024-05-31	2254	TRANSFER	CREATED	Imane

Items per page: 5 11 - 15 of 40 | < < > >|

dashboard Students Payments