**Création d’une application ecommerce en Angular**

# Introduction

Des petits exemples et ciblés me permettent de démontrer des fonctionnalités d’Angular spécifiques, mais ils peuvent manquer de contexte d’application. Pour aider à surmonter ce problème, je vais créer une application de commerce électronique simple mais réaliste qui reprend l’exemple que nous avons vu dans la partie Java JEE et Spring Boot.

Mon application, appelée ChezBio, suivra l'approche classique adoptée par les magasins en ligne du monde entier. Je vais créer un catalogue de produits en ligne que les clients peuvent parcourir par catégorie et par page, un panier d'achat où les utilisateurs peuvent ajouter et supprimer des produits, et une caisse où les clients peuvent entrer leurs détails d'expédition et passer leurs commandes.

Mon objectif dans ce projet est de vous donner une idée de ce que développer une véritable application en Angular, en créant un exemple aussi réaliste que possible. Je veux me concentrer sur Angular, bien sûr, et j'ai donc simplifié l'intégration avec des systèmes externes, tels que les données de magasin, et omis entièrement d'autres, tels que le traitement des paiements.

# Préparation du projet

Pour créer le projet chezBio, ouvrez une invite de commande, accédez à un emplacement pratique et exécutez la commande suivante :

dahamada@PT011374 MINGW64 ~/ecommerce-angular

$ ng new ChezBio --routing false --style css --skip-git --skip-tests

Le package angular-cli créera un nouveau projet pour le développement d’Angular, avec des fichiers de configuration, du contenu d'espace réservé et des outils de développement.

Pour executer dans le port 4500 :

dahamada@PT011374 MINGW64 ~/ecommerce-angular/ChezBio

$ ng s -o --port 4500

Des packages supplémentaires sont requis pour le projet ChezBio, en plus des packages Angular de base et des outils de construction configurés par la commande **ng new** . Exécutez les commandes suivantes pour accéder au dossier ChezBio et ajouter les packages requis :

Installer bootstrap version 4.4.1

dahamada@PT011374 MINGW64 ~/ecommerce-angular/ChezBio

$ npm install [bootstrap@4.4.1](mailto:bootstrap@4.4.1)

Installer font awesome version 5.12.1

dahamada@PT011374 MINGW64 ~/ecommerce-angular/ChezBio

$ npm install @fortawesome/fontawesome-free@5.12.1

Il est important d'utiliser les numéros de version indiqués dans la liste. Vous pouvez voir des avertissements concernant les dépendances homologues non satisfaites lorsque vous ajoutez les packages, mais vous pouvez les ignorer. Certains des packages sont installés à l'aide de l' argument --save-dev , qui indique qu'ils sont utilisés pendant le développement et ne feront pas partie de l'application ChezBio

## Ajout des feuilles de style CSS à l'application

Une fois les packages installés, ajoutez les instructions présentées ci-dessous au fichier angular.json pour incorporer les fichiers CSS du framework CSS Bootstrap et les packages Font Awesome dans l'application. J'utiliserai les styles CSS Bootstrap pour tout le contenu HTML de l'application ChezBio, et j'utiliserai les icônes du package Font Awesome pour présenter un résumé d'un panier à l'utilisateur.

 "styles": [

              "src/styles.css",

              "node\_modules/bootstrap/dist/css/bootstrap.min.css",

              "node\_modules/@fortawesome/fontawesome-free/css/all.min.css"

            ],

## Mise à jour du composant racine

app.component.ts

import { Component } from '@angular/core';

@Component({

  selector: 'app-root',

  // templateUrl: './app.component.html',

  template: `<div class="bg-success p-2 text-center text-white">

    Bienvenue ChezBio

  </div>`,

  styleUrls: ['./app.component.css'],

})

export class AppComponent {

  title = 'ChezBioV1';

}

Le décorateur **@Component** indique à Angular que la classe AppComponent est un composant et que ses propriétés configurent la manière dont le composant est appliqué

La propriété **selector** indique à Angular comment applique le composant dans le document HTML,   et la propriété **template** définit le contenu HTML que   le composant affichera.

Les composants peuvent définir des modèles en ligne, comme celui-ci, ou ils utilisent des fichiers HTML externes, ce qui peut faciliter la gestion de contenu complexe

## Mise à jour du module racine

App.module.ts

import { NgModule } from '@angular/core';

import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';

import { MagasinModule } from './components/magasin.model';

@NgModule({

  declarations: [AppComponent],

  imports: [BrowserModule],

  providers: [],

  bootstrap: [AppComponent],

})

export class AppModule {}

Il existe deux types de modules d’Angular: **les modules de fonctionnalités** et **le module racine**. Les **modules de fonctionnalités** sont utilisés pour regrouper les fonctionnalités d'application associées afin de faciliter la gestion de l'application**.**

Le **module racine** est utilisé pour décrire l'application à Angular. La description inclut les modules de fonctionnalités requis pour exécuter l'application, les fonctionnalités personnalisées à charger et le nom du composant racine. Le nom conventionnel du fichier de composant racine est **app.module.ts**

Semblable au composant racine, il n'y a pas de code dans la classe du module racine. C'est parce que le module racine n'existe vraiment que pour fournir des informations via le décorateur **@NgModule** . La propriété **imports** indique à Angular qu'il doit charger le module de fonctionnalités **BrowserModule** , qui contient les fonctionnalités d’Angular de base requises pour une application Web.

La propriété **declarations** indique à Angular qu'il doit charger le composant racine, la propriété **providers** indique à Angular les objets partagés utilisés par l'application et la propriété **bootstrap** indique à Angular que le composant racine est la classe **AppComponent** . J'ajouterai des informations aux propriétés de ce décorateur au fur et à mesure que j'ajouterai des fonctionnalités à l'application Chezbio, mais cette configuration de base est suffisante pour démarrer l'application.

On peut tester voir l’affichage sur le navigateur

## Création des classes de modèle

Chaque modèle de données a besoin de classes qui décrivent les types de données qui seront contenues dans le modèle de données. Pour l'application Chezbio, cela signifie des classes qui décrivent les produits vendus dans le magasin et les commandes reçues des clients.

dahamada@PT011374 MINGW64 ~/ecommerce-angular/ChezBioV1

$ ng g class models/produit --type=model --skipTests=true

Produit.model.ts

export class Produit {

  constructor(

    public id?: number,

    public nom?: string,

    public prix?: number,

    public description?: string,

    public urlPhoto?: string,

    public derniere\_maj?: Date,

    public categorie?: string

  ) {}

}

La classe **Produit** définit un constructeur qui accepte les propriétés id , nom , categorie , description et prix , qui correspondent à la structure des données utilisées pour remplir le service Web RESTful

Les points **d'interrogation (les caractères ? )** qui suivent les noms des paramètres indiquent qu'il s'agit de paramètres facultatifs qui peuvent être omis lors de la création de nouveaux objets à l'aide de la classe Produit , ce qui peut être utile lors de l'écriture d'applications dans lesquelles les propriétés des objets de modèle seront renseignées à l'aide de formulaires HTML.

## Création de la source de données fictives

dahamada@PT011374 MINGW64 ~/ecommerce-angular/ChezBioV1

$ ng g class models/StaticDataSource --skipTests=true

static.datasource.ts

import { Injectable } from '@angular/core';

import { Observable, from } from 'rxjs';

import { Produit } from './produit.model';

@Injectable()

export class StaticDataSource {

  date: Date = new Date();

  private listeproduits: Produit[] = [

    new Produit(

      1,

      'GRUYERE',

      12,

      'plaquette',

      '/assets/images/produits/GRUYERE.png',

      this.date,

      'FROMAGES'

    ),

    new Produit(2, 'PIZZA', 16, 'unite', '/assets/images/produits/PIZZA.png', this.date, 'PAINS'),

    new Produit(3, 'RADIS', 17, 'kg', '/assets/images/produits/RADIS.png', this.date, 'LEGUMES'),

    new Produit(4, 'POULET', 10, 'kg', '/assets/images/produits/POULET.png', this.date, 'VIANDES'),

  ];

  getProduits(): Observable<Produit[]> {

    return from([this.listeproduits]);

  }

}

La classe **StaticDataSource** définit une méthode appelée getProduits , qui renvoie les données factices. Le résultat de l'appel de la méthode getProduits est un **Observable<Produit[]>** , qui est un **Observable** qui produit des tableaux d' objets Product .

La classe Observable est fournie par le package **Reactive Extensions**, qui est utilisé par Angular pour gérer les changements d'état dans les applications. Je décris la classe Observable dans un autre chapitre , mais pour ce chapitre, il suffit de savoir qu'un objet Observable représente une tâche asynchrone qui produira un résultat à un moment donné dans le futur. Angular expose son utilisation d' objets Observable pour certaines fonctionnalités, y compris les requêtes HTTP, et c'est pourquoi la méthode getProduits renvoie un Observable<Produit[]> plutôt que de simplement renvoyer les données de manière synchrone.

En effet dans la réalité, le service **StaticDataSource** ne va pas chercher les produits dans un tableau local mais à partir d’un serveur externe. On ne peut donc pas contrôler combien de temps va prendre l’accès à ce serveur. Et on ne doit pas geler l’application en attendant que le serveur réponde. Par conséquent, il n’est pas acceptable que le service fasse des appels synchrones dans la méthode **getProduits()** .

Le décorateur **@Injectable** a été appliqué à la classe StaticDataSource . Ce décorateur est utilisé pour indiquer à Angular que cette classe sera utilisée en tant que service, ce qui permet à d'autres classes d'accéder à ses fonctionnalités via une fonctionnalité appelée injection de dépendance.

Remarque : la programmation réactive se base sur le concept d'[observateur](https://fr.wikipedia.org/wiki/Observateur_(patron_de_conception)). Le principe est tout simplement que l'on définit des **observables** et des **observateurs**. Les observables vont émettre des événements qui seront interceptés par les observateurs.

**Observable** est l'objet de base de la programmation réactive. C'est lui qui va nous permettre de créer des observables.

La méthode **subscribe** va nous permettre de  surveiller de l’observable en créant nos observateurs. La méthode **subscribe** prend en paramètre l'observateur, qui est une simple fonction qui recevra les valeurs émises par l'observable

## Création du référentiel modèle

La source de données est chargée de fournir à l'application les données dont elle a besoin, mais l'accès à ces données est généralement assuré par un référentiel(repository), qui est responsable de la distribution de ces données aux blocs de construction d'application individuels afin que les détails de la façon dont les données ont été obtenues soient tenus cachés.

Un repository est une abstraction nous permettant de manipuler les objets du domaine métier.

Il fournit un ensemble d’opérations élémentaires pour la manipulation des objets

produit.repository.

Produit.respository

import { Injectable } from '@angular/core';

import { Produit } from './produit.model';

import { StaticDataSource } from './static.datasource';

@Injectable()

export class ProduitRepository {

  private produits: Produit[] = [];

  private categories: string[] = [];

  constructor(private dataSource: StaticDataSource) {

    dataSource.getProduits().subscribe((data) => {

      this.produits = data;

      this.categories = data

        .map((p) => p.categorie)

        .filter((c, index, array) => array.indexOf(c) == index)

        .sort();

    });

  }

  getProduits(categorie: string = null): Produit[] {

    return this.produits.filter(

      (p) => categorie == null || categorie == p.categorie

    );

  }

  getProduit(id: number): Produit {

    return this.produits.find((p) => p.id == id);

  }

  getCategories(): string[] {

    return this.categories;

  }

}

Lorsque Angular doit créer une nouvelle instance du référentiel, il inspecte la classe et voit qu'il a besoin d'un objet **StaticDataSource** pour appeler le constructeur **ProductRepository** et créer un nouvel objet.

Le constructeur du référentiel appelle la méthode getProduits de la source de données , puis utilise la méthode subscribe sur l' objet Observable renvoyé pour recevoir les données du produit.

La méthode **subscribe()** reçoit une fonction (ici, une *arrow function*) à qui sont transmis les résultats retournés par **getProduits()**.

  La méthode map transforme les éléments d'un tableau et renvoie un nouveau tableau avec les valeurs transformées.Exemple :

  let numbers = [1, 2, 3, 4]

  let squares = numbers.map(num => num \* num);

> squares

= [ 1, 4, 9, 16 ]

Lorsque la methode filter termine l'itération, il renvoie le tableau des éléments sélectionnés

 : les éléments pour lesquels le rappel a renvoyé une valeur vraie.

 Dans notre exemple ci-dessous, la méthode filter sélectionne

 tous les éléments avec une valeur supérieure à 4.

> let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1, 2]

> numbers.filter(num => num > 4)

= [ 5, 6, 7, 8, 9, 10 ]

La méthode indexof retourne le premier index de l'élément dans le tableau

ou -1 si la valeur n'est pas trouvée.

var tableau = [2, 9, 9];

tableau.indexOf(2);     // 0

tableau.indexOf(7);     // -1

La méthode sort() trie les éléments d'un tableau, dans ce même tableau, et renvoie le tableau

const months = ['March', 'Jan', 'Feb', 'Dec'];

months.sort();

console.log(months);

Sortie attendue : Array ["Dec", "Feb", "Jan", "March"]

## Création du module de fonctionnalités

Je vais définir un module de fonctionnalité Angular qui permettra d'utiliser facilement la fonctionnalité du modèle de données ailleurs dans l'application. J'ai ajouté un fichier appelé **model.module.ts**

Le décorateur **@NgModule** est utilisé pour créer des modules de fonctionnalités, et ses propriétés indiquent à Angular comment le module doit être utilisé. Il n'y a qu'une seule propriété dans ce module, **providers** , et elle indique à Angular quelles classes doivent être utilisées comme services pour la fonctionnalité **d'injection de dépendances**

model.module.ts

import { NgModule } from '@angular/core';

import { ProduitRepository } from './produit.repository';

import { StaticDataSource } from './static.datasource';

@NgModule({

  providers: [ProduitRepository, StaticDataSource],

})

export class ModelModule {}

## Création du composant de magasin et de la template

dahamada@PT011374 MINGW64 ~/ecommerce-angular/ChezBioV1

$ ng g c composants/magasin --skipTests=true

magasin.component.ts

import { Component, OnInit } from '@angular/core';

import { Produit } from 'src/app/models/produit.model';

import { ProduitRepository } from 'src/app/models/produit.repository';

@Component({

  selector: 'app-magasin',

  templateUrl: './magasin.component.html',

  styleUrls: ['./magasin.component.css'],

})

export class MagasinComponent {

  constructor(private repository: ProduitRepository) {}

  get produits(): Produit[] {

    return this.repository

      .getProduits();

  }

  get categories(): string[] {

    return this.repository.getCategories();

  }

La classe **MagasinComponent** fournit la logique qui prendra en charge le contenu du modèle. Le constructeur reçoit un objet ProduitRepository en tant qu'argument, fourni via la fonctionnalité d'injection de dépendances

 Le composant définit les propriétés des produits et des catégories qui seront utilisées pour générer du contenu HTML dans la template, à l'aide des données obtenues à partir du référentiel.

magasin.component.html

<div class="container-fluid">

  <div class="row">

 <div class="col bg-dark text-white">

      <nav class="navbar navbar-expand-lg bg-dark navbar-dark">

        <a class="navbar-brand" >

          <img src="../assets/images/logo.png"/>

        </a>

    </nav>

    </div>

  </div>

<div class="row">

    <div class="col-3 bg-info p-2">{{ categories.length }} Categories</div>

      <div class="col-9 bg-success p-2">

      {{produits.length}} Products

    </div>

</div>

</div>

## Création du module de fonctionnalité de magasin

magasin.model.ts ( module)

import { NgModule } from '@angular/core';

import { FormsModule } from '@angular/forms';

import { BrowserModule } from '@angular/platform-browser';

import { ModelModule } from '../models/model.module';

import { CompterDirective } from './compter.directive';

import { MagasinComponent } from './magasin.component';

@NgModule({

  imports: [ModelModule, BrowserModule, FormsModule],

  declarations: [MagasinComponent],

  exports: [MagasinComponent],

})

export class MagasinModule {}

Le décorateur @NgModule configure le module, en utilisant la propriété imports pour indiquer à Angular que le module de magasin dépend du module de modèle ainsi que BrowserModule et FormsModule , qui contiennent les fonctionnalités angulaires standard pour les applications Web et pour travailler avec des éléments de formulaire HTML. Le décorateur utilise la propriété declarations pour indiquer à Angular la classe MagasinComponent , et la propriété exports indique à Angular que la classe peut également être utilisée dans d'autres parties de l'application, ce qui est important car elle sera utilisée par le module racine.

App.component.ts

import { Component } from '@angular/core';

//Le décorateur @Component indique à Angular que la classe

// AppComponent est un composant et que ses propriétés

//configurent la manière dont le composant est appliqué

@Component({

  /\*La propriété selector indique à Angular comment

  appliquer le composant dans le document HTML,

  et la propriété template définit le contenu HTML que

  le composant affichera. Les composants peuvent définir des modèles en ligne,

   comme celui-ci, ou ils utilisent des fichiers HTML externes,

  ce qui peut faciliter la gestion de contenu complexe

  \*/

  selector: 'app-root',

  // templateUrl: './app.component.html',

  /\*

  template: `<div class="bg-success p-2 text-center text-white">

    Bienvenue ChezBio

  </div>`,

  \*/

  template: '<app-magasin></app-magasin>',

  styleUrls: ['./app.component.css'],

})

export class AppComponent {

  title = 'ChezBioV1';

}

## Mise à jour du composant racine et du module racine

L'application du modèle de base et de la fonctionnalité de magasin nécessite la mise à jour du module racine de l'application pour importer les deux modules de fonctionnalités et nécessite également la mise à jour du modèle du module racine pour ajouter l'élément HTML auquel le composant du module de magasin sera appliqué

Ajout d'un élément dans le fichier **app.component.ts** dans le dossier src/app

L' élément : '<app-magasin></app-magasin>' remplace le contenu précédent dans le modèle du composant racine et correspond à la valeur de la propriété selector du décorateur @Component du

Importation de modules de fonctionnalités dans le fichier app.module.ts du dossier src/app

App.module.ts

import { NgModule } from '@angular/core';

import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';

import { MagasinModule } from './components/magasin.model';

@NgModule({

  declarations: [AppComponent],

  imports: [BrowserModule, MagasinModule],

  providers: [],

  bootstrap: [AppComponent],

})

export class AppModule {}

Importation de modules de fonctionnalités dans le fichier app.module.ts du dossier src/app

Lorsque vous enregistrez les modifications apportées au module racine, Angular disposera de tous les détails dont il a besoin pour charger l'application et afficher le contenu du module de magasin

Affichage

## Ajouter des fonctionnalités de magasin pour l’affichage des détails du produit

La plupart des éléments contrôlent la mise en page et l'apparence du contenu. Le changement le plus important est l'ajout d'une expression de liaison de données d’Angular ( interporlation de chaine).

magasin.component.html

<div class="container-fluid">

  <div class="row">

    <div class="col bg-dark text-white">

     <nav class="navbar navbar-expand-lg bg-dark navbar-dark">

        <a class="navbar-brand" >

          <img src="../assets/images/logo.png"/>

        </a>

    </nav>

    </div>

  </div>

<div class="row">

    <div class="col-3 bg-info p-2">{{ categories.length }} Categories</div>

  <div class="col-9 p-2">

      {{ produits.length }} Produits

      <div \*ngFor="let produit of produits" class="card-text m-1 p-1 bg-light">

        <h4>

          {{ produit.nom }}

          <span class="badge badge-pill badge-primary float-right">

            {{ produit.prix | currency: "USD":"symbol":"2.2-2" }}

          </span>

        </h4>

<div>

          <img class="produit-image" [src]="produit.urlPhoto">

      </div>

        <div class="card-text bg-white p-1">{{ produit.description }}</div>

      </div>

</div>

</div>

...

<div \*ngFor="let product of products" class="card m-1 p-1 bg-light">

...

Ceci est un exemple de directive, qui transforme l'élément HTML auquel elle s'applique. Cette directive spécifique s'appelle ngFor , et elle transforme l' élément div en le dupliquant pour chaque objet renvoyé par la propriété produits du composant . Angular comprend une gamme de directives intégrées qui effectuent les tâches les plus couramment requises

Comme il duplique l' élément div , l'objet actuel est affecté à une variable appelée produit , ce qui lui permet d'être référencé dans d'autres liaisons de données, telles que celle-ci, qui insère la valeur de la propriété de description du nom du produit actuel comme contenu de l' élément div :

...

<div class="card-text p-1 bg-white"> {{product.description}} </div>

...

Toutes les données du modèle de données d'une application ne peuvent pas être affichées directement à l'utilisateur. Angular inclut une fonctionnalité appelée pipes , qui sont des classes utilisées pour transformer ou préparer une valeur de données pour son utilisation dans une liaison de données. Il existe plusieurs pipes intégrés inclus avec Angular, y compris le pipe de devise , qui formate les valeurs numériques en devises, comme ceci :

...

{{ produit.prix | devise :"USD":"symbol":"2.2-2" }}

...

La syntaxe d'application des pipes peut être un peu délicate, mais l'expression dans cette liaison indique à Angular de formater la propriété de prix du produit actuel à l'aide du pipe de devise , avec les conventions monétaires des États-Unis

## Ajouter une sélection de catégorie

L'ajout de la prise en charge du filtrage de la liste des produits par catégorie nécessite de préparer le composant de magasin afin qu'il garde une trace de la catégorie que l'utilisateur souhaite afficher et nécessite de modifier la façon dont les données sont récupérées pour utiliser cette catégorie comme illustré ci-dessous :

magasin.component.ts

import { Component, OnInit } from '@angular/core';

import { Produit } from 'src/app/models/produit.model';

import { ProduitRepository } from 'src/app/models/produit.repository';

@Component({

  selector: 'app-magasin',

  templateUrl: './magasin.component.html',

  styleUrls: ['./magasin.component.css'],

})

export class MagasinComponent {

 public categorieSelectionnee = null;

  constructor(private repository: ProduitRepository) {}

  get produits(): Produit[] {

      return this.repository

      .getProduits(this.categorieSelectionnee);

  }

  get categories(): string[] {

    return this.repository.getCategories();

  }

  changerCategorie(nouvelleCategorie?: string) {

    this.categorieSelectionnee = nouvelleCategorie;

  }

Les changements sont simples car ils s'appuient sur les fondations qui ont mis tant de temps à créer au début du chapitre. La propriété categorieSelectionnee est affectée au choix de catégorie de l'utilisateur (où null signifie toutes les catégories) et est utilisée dans la méthode updateData comme argument de la méthode getProduits , déléguant le filtrage à la source de données. La méthode changerCategorie rassemble ces deux membres dans une méthode qui peut être invoquée lorsque l'utilisateur effectue une sélection de catégorie.

Ajouter le bouton de changement de catégorie :

magasin.component.html

<div class="container-fluid">

  <div class="row">

    <div class="col bg-dark text-white">

     <nav class="navbar navbar-expand-lg bg-dark navbar-dark">

        <a class="navbar-brand" >

          <img src="../assets/images/logo.png"/>

        </a>

    </nav>

    </div>

  </div>

<div class="row">

    <div class="col-3 p-2">

      {{ categories.length }} Categories

  <button

      class="btn btn-block btn-outline-primary"

      (click)="changerCategorie()"

    >

      Accueil

    </button>

    <button

      \*ngFor="let cat of categories"

      class="btn btn-outline-primary btn-block"

      [class.active]="cat == categorieSelectionnee"

      (click)="changerCategorie(cat)"

    >

      {{ cat }}

    </button>

    </div>

  <div class="col-9 p-2">

      {{ produits.length }} Products

      <div \*ngFor="let produit of produits" class="card-text m-1 p-1 bg-light">

        <h4>

          {{ produit.nom }}

          <span class="badge badge-pill badge-primary float-right">

            {{ produit.prix | currency: "USD":"symbol":"2.2-2" }}

          </span>

        </h4>

        <div class="card-text bg-white p-1">{{ produit.description }}</div>

      </div>

</div>

</div>

## Ajout de la pagination du produit

Il y a deux nouvelles fonctionnalités dans cette liste. Le premier est la possibilité d'obtenir une page de produits, et le second est de modifier la taille des pages, permettant de modifier le nombre de produits que chaque page contient.

magasin.component.ts

import { Component, OnInit } from '@angular/core';

import { Produit } from 'src/app/models/produit.model';

import { ProduitRepository } from 'src/app/models/produit.repository';

@Component({

  selector: 'app-magasin',

  templateUrl: './magasin.component.html',

  styleUrls: ['./magasin.component.css'],

})

export class MagasinComponent {

……..

 public produitsParPage = 4;

  public pageSectionnee = 1;

constructor(private repository: ProduitRepository) {}

  get produits(): Produit[] {

    let pageIndex = (this.pageSectionnee - 1) \* this.produitsParPage;

    return this.repository

      .getProduits(this.categorieSelectionnee)

      .slice(pageIndex, pageIndex + this.produitsParPage);

  }

  changerCategorie(nouvelleCategorie?: string) {

    this.categorieSelectionnee = nouvelleCategorie;

  }

  changerPage(nouvellePage: number) {

    this.pageSectionnee = nouvellePage;

  }

  changerPageTaille(nouvelleTaille: number) {

    this.produitsParPage = Number(nouvelleTaille);

    this.changerPage(1);

  }

  get pageNumbers(): number[] {

    return Array(

      Math.ceil(

        this.repository.getProduits(this.categorieSelectionnee).length /

          this.produitsParPage

      )

    )

      .fill(0)

      .map((x, i) => i + 1);

  }

 La méthode slice() renvoie un objet tableau, contenant une copie superficielle (shallow copy)

    d'une portion du tableau d'origine, la portion est définie par un indice de début et un indice de fin. Exemple

      const animals = ['ant', 'bison', 'camel', 'duck', 'elephant'];

       console.log(animals.slice(2));

   Sortie attendues : Array ["camel", "duck", "elephant"]

   console.log(animals.slice(2, 4));

   Sortie attendue : Array ["camel", "duck"]

Il y a une bizarrerie que le composant doit contourner. Il existe une limitation dans la directive ngFor intégrée fournie par Angular, qui peut générer du contenu uniquement pour les objets d'un tableau ou d'une collection, plutôt que d'utiliser un compteur. Puisque j'ai besoin de générer des boutons de navigation de page numérotés, cela signifie que je dois créer un tableau contenant les nombres dont j'ai besoin, comme ceci

 get pageNumbers(): number[] {

    return Array(

      Math.ceil(

        this.repository.getProduits(this.categorieSelectionnee).length /

          this.produitsParPage

      )

    )

      .fill(0)

      .map((x, i) => i + 1);

  }

Cette instruction crée un nouveau tableau, le remplit avec la valeur 0 , puis utilise la méthode map pour générer un nouveau tableau avec la séquence de nombres. Cela fonctionne assez bien pour implémenter la fonctionnalité de pagination, mais cela semble gênant, et je démontre une meilleure approche dans la section suivante.

Template :

magasin.component.html

<div class="container-fluid">

  <div class="row">

    <div class="col bg-dark text-white">

     <nav class="navbar navbar-expand-lg bg-dark navbar-dark">

        <a class="navbar-brand" >

          <img src="../assets/images/logo.png"/>

        </a>

    </nav>

    </div>

  </div>

…..

    <!-- Pagination

       ($any($event.target).value)-->

      <div class="form-inline float-left mr-1">

        <select

          class="form-control"

          [value]="produitsParPage"

          (change)="changerPageTaille($any($event.target).value)"

        >

          <option value="3">3 par Page</option>

          <option value="4">4 par Page</option>

          <option value="6">6 par Page</option>

          <option value="8">8 par Page</option>

        </select>

      </div>

      <div class="btn-group float-right">

        <button

          \*ngFor="let page of pageNumbers"

          (click)="changerPage(page)"

          class="btn btn-outline-primary"

          [class.active]="page == pageSectionnee"

        >

          {{ page }}

        </button>

      </div>

</div>

</div>

Les nouveaux éléments ajoutent un élément de sélection qui permet de modifier la taille de la page et un ensemble de boutons qui naviguent dans les pages de produits. Les nouveaux éléments ont des liaisons de données pour les relier aux propriétés et aux méthodes fournies par le composant. Le résultat est un ensemble de produits plus facile à gérer

## Création d'une directive personnalisée

Dans cette section, je vais créer une directive personnalisée afin de ne pas avoir à générer un tableau rempli de nombres pour créer les boutons de navigation de page.

Pour créer une directive personnalisée, nous devons remplacer @Component décorateur par le décorateur @Directive

ViewContainerRef représente un conteneur dans lequel une ou plusieurs vues peuvent être attachées à un composant.

 TemplateRef représente un template incorporé qui peut être utilisé pour instancier des vues incorporées.

La méthode ngOnChanges() utilise SimpleChanges omme argument qui donne les valeurs nouvelles et précédentes des valeurs d'entrée après les modifications

L'utilisation de  $implicit dans l'objet de contexte définira sa valeur par défaut.

 J'ai utilisé $implicit pour transmettre la valeur à template,en créant dynamiquement les numeros de pages

compter.directive.ts

import {

  Directive,

  ViewContainerRef,

  TemplateRef,

  Input,

  Attribute,

  SimpleChanges,

} from '@angular/core';

@Directive({

  selector: '[counterOf]',

})

export class CompterDirective {

  constructor(

    private container: ViewContainerRef,

    private template: TemplateRef<Object>

  ) {}

  @Input('counterOf')

  counter: number;

  ngOnChanges(changes: SimpleChanges) {

    this.container.clear();

    for (let i = 0; i < this.counter; i++) {

      this.container.createEmbeddedView(

        this.template,

        new CompterDirectiveContext(i + 1)

      );

    }

  }

}

class CompterDirectiveContext {

  constructor(public $implicit: any) {}

}

Il s'agit d'un exemple de directive structurelle, qui est décrit en détail dans un autre chapitre . Cette directive est appliquée aux éléments via une propriété compter et s'appuie sur les fonctionnalités spéciales fournies par Angular pour créer du contenu de manière répétée, tout comme la directive ngFor intégrée . Dans ce cas, plutôt que de produire chaque objet d'une collection, la directive personnalisée produit une série de nombres qui peuvent être utilisés pour créer les boutons de navigation de page

Remarque : cette directive supprime tout le contenu qu'elle a créé et recommence lorsque le nombre de pages change. Cela peut être un processus coûteux dans des directives plus complexe

Pour utiliser la directive, elle doit être ajoutée à la propriété declarations de son module de fonctionnalité.

Enregistrer dans le module :

magasin.model.ts ( module)

import { NgModule } from '@angular/core';

import { FormsModule } from '@angular/forms';

import { BrowserModule } from '@angular/platform-browser';

import { ModelModule } from '../models/model.module';

import { CompterDirective } from './compter.directive';

import { MagasinComponent } from './magasin.component';

@NgModule({

  imports: [ModelModule, BrowserModule, FormsModule],

  declarations: [MagasinComponent, CompterDirective],

  exports: [MagasinComponent],

})

export class MagasinModule {}

Ajouter la directive :

magasin.component.ts

import { Component, OnInit } from '@angular/core';

import { Produit } from 'src/app/models/produit.model';

import { ProduitRepository } from 'src/app/models/produit.repository';

@Component({

  selector: 'app-magasin',

  templateUrl: './magasin.component.html',

  styleUrls: ['./magasin.component.css'],

})

export class MagasinComponent {

……..

  changerCategorie(nouvelleCategorie?: string) {

    this.categorieSelectionnee = nouvelleCategorie;

  }

  changerPage(nouvellePage: number) {

    this.pageSectionnee = nouvellePage;

  }

  changerPageTaille(nouvelleTaille: number) {

    this.produitsParPage = Number(nouvelleTaille);

    this.changerPage(1);

  }

 get compterPage(): number {

    return Math.ceil(

      this.repository.getProduits(this.categorieSelectionnee).length /

        this.produitsParPage

    );

  }

Maintenant que la directive a été enregistrée, elle peut être utilisée dans la template du composant de magasin pour remplacer la directive ngFor .

magasin.component.html

<div class="container-fluid">

  <div class="row">

    <div class="col bg-dark text-white">

      <a class="navbar-brand">MAGASIN BIO</a>

    </div>

  </div>

…..

    <!-- Pagination

       ($any($event.target).value)-->

      <div class="form-inline float-left mr-1">

        <select

          class="form-control"

          [value]="produitsParPage"

          (change)="changerPageTaille($any($event.target).value)"

        >

          <option value="3">3 par Page</option>

          <option value="4">4 par Page</option>

          <option value="6">6 par Page</option>

          <option value="8">8 par Page</option>

        </select>

      </div>

        <div class="btn-group float-right">

        <button

          \*counter="let page of compterPage"

          (click)="changerPage(page)"

          class="btn btn-outline-primary"

          [class.active]="page == pageSectionnee"

        >

          {{ page }}

        </button>

      </div>

</div>

</div>

La nouvelle liaison de données s'appuie sur une propriété appelée compterPage pour configurer la directive personnalisée.

## Résumé :

Dans cette partie 1, j'ai commencé le projet Chezbio. La première partie du chapitre a été consacrée à la création de la base du projet, y compris la création des blocs de construction racine de l'application et le démarrage du travail sur les modules de fonctionnalités. Une fois la fondation en place, j'ai pu rapidement ajouter des fonctionnalités pour afficher les données du modèle factice à l'utilisateur, ajouter une pagination et filtrer les produits par catégorie. J'ai terminé le chapitre en créant une directive personnalisée pour montrer comment les fonctionnalités intégrées fournies par Angular peuvent être complétées par du code personnalisé. Dans le chapitre suivant, je continue à construire l'application Chezbio.