Créer des applications web Angular

Pr . ELAACHAK Lotfi
2020-2021

Sommaire

Commencer avec Angular

Histoire :

Aux débuts du développement web, seul le HTML permet de créer des sites. Les sites web en 1990 ressemblent à des documents texte, car il n'existe pas vraiment d'autres possibilités. Ensuite, en 1998, le CSS arrive pour aider à créer des mises en page plus esthétiques. Finalement, à partir du début des années 2000, le JavaScript commence son règne sur le développement web, permettant des interactions entre l'utilisateur et la page.

À partir de 2005, le système AJAX (Asynchronous Javascript And XML) permet des interactions entre l'utilisateur et des backend HTTP : il est enfin possible d'échanger des informations et de générer du contenu à partir de ces interactions.

En 2010, la première version d'AngularJS est lancée. Elle permet de créer plus facilement des Single Page Applications, des applications web qui imitent les applications natives : pas de rafraîchissement du navigateur, temps de chargement réduits, une UI beaucoup moins "internet" etc. Cette version permet déjà de faire énormément de choses, mais souffre d'une syntaxe plutôt complexe ainsi que des limitations du JavaScript. Voilà pourquoi Google choisit de complètement réécrire le framework pour sa version 2. Aujourd'hui, nous en sommes à Angular 11.x (maintenant appelé simplement "Angular") ; la version 3 ayant été sautée pour des raisons sémantiques tout simplement.

Pourquoi Angular ?

Angular est géré par Google — il y a donc peu de chances qu'il disparaisse, et l'équipe de développement du framework est excellente.

- Le TypeScript — ce langage permet un développement beaucoup plus stable, rapide et facile.
- Le framework Ionic — le framework permettant le développement d'applications mobiles multi-plateformes à partir d'une seule base de code — utilise Angular.

Les autres frameworks ont leurs avantages également, mais Angular est un choix très pertinent pour le développement frontend.

ES2015 :

ECMAScript est le nom officiel de JavaScript

La version standard actuelle est ES6 ou ES2015

ES6 ajoute une tonne de fonctionnalités à JavaScript (ES5), comme:

les classes
les constantes
les arrow functions
les générateurs…

Qu'est-ce que le TypeScript ? Pourquoi l'utiliser ?

Le TypeScript est un sur-ensemble (un "superset") de JavaScript qui est justement transcompilé (transcompilation : "traduction" d'un langage de programmation vers un autre - différent de la compilation, qui transforme généralement le code vers un format exécutable) en JavaScript pour être compréhensible par les navigateurs. Il ajoute des fonctionnalités extrêmement utiles, comme, entre autres :

- le typage strict, qui permet de s'assurer qu'une variable ou une valeur passée vers ou retournée par une fonction soit du type prévu ;
- les fonctions dites lambda ou arrow, permettant un code plus lisible et donc plus simple à maintenir ;
- les classes et interfaces, permettant de coder de manière beaucoup plus modulaire et robuste.

L'architecture Angular :

Angular est une plate-forme et un cadre pour la création d'applications clientes d'une seule page « SPA » à l'aide de HTML et de TypeScript. Angular est écrit en TypeScript. Il implémente les fonctionnalités de base et facultatives sous la forme d'un ensemble de bibliothèques TypeScript que peut importer dans des applications.

L'architecture d'une application Angular repose sur certains concepts fondamentaux. Les blocs de construction de base du framework Angular sont des composants angulaires organisés en NgModules. Les NgModules rassemblent le code associé en ensembles fonctionnels; une application angulaire est définie par un ensemble de NgModules. Une application a toujours au

moins un module racine qui permet le démarrage, et a généralement beaucoup plus de modules de fonctionnalités.

- Les composants définissent des vues, qui sont des ensembles d'éléments d'écran parmi lesquels Angular peut choisir et modifier en fonction de la logique et des données de programme.
- Les composants utilisent des services, qui fournissent des fonctionnalités spécifiques non directement liées aux vues. Les fournisseurs de services peuvent être injectés dans les composants en tant que dépendances, ce qui rend votre code modulaire, réutilisable et efficace.

Les modules, composants et services sont des classes qui utilisent des décorateurs. Ces décorateurs marquent leur type et portent des métadonnées qui envisagent à Angular comment les utiliser.

- Les métadonnées d'une classe de composant l'associent à un modèle qui définit une vue. Un modèle combine du HTML ordinaire avec des directives Angular et un balisage de liaison qui permettent à Angular de modifier le HTML avant de le rendre pour l'affichage.
- Les métadonnées d'une classe de service fournissent les informations dont Angular a besoin pour les rendre disponibles aux composants via l'injection de dépendances (DI).

Note Decorator :https://www.tutorialspoint.com/design_pattern/decorator_pattern.htm

Modules :

Un NgModule déclare un contexte de compilation pour un ensemble de composants qui est dédié à un domaine d'application, un flux de travail ou un ensemble de fonctionnalités étroitement liées. Un NgModule peut associer ses composants à du code associé, comme des services, pour former des unités fonctionnelles.

Chaque application Angular possède un module racine, appelé conventionnellement AppModule, qui fournit le mécanisme d'amorçage qui lance l'application. Une application contient généralement de nombreux modules fonctionnels.

Composantes :

Chaque application Angular a au moins un composant, le composant racine qui connecte une hiérarchie de composants avec le modèle d'objet de document de page (DOM). Chaque composant définit une classe qui contient des données d'application et une logique, et est associée à un modèle HTML qui définit une vue à afficher dans un environnement cible.

Le décorateur @Component () identifie la classe immédiatement en dessous comme composant et fournit le modèle et les métadonnées spécifiques au composant associé.

Templates, directives, et data binding :

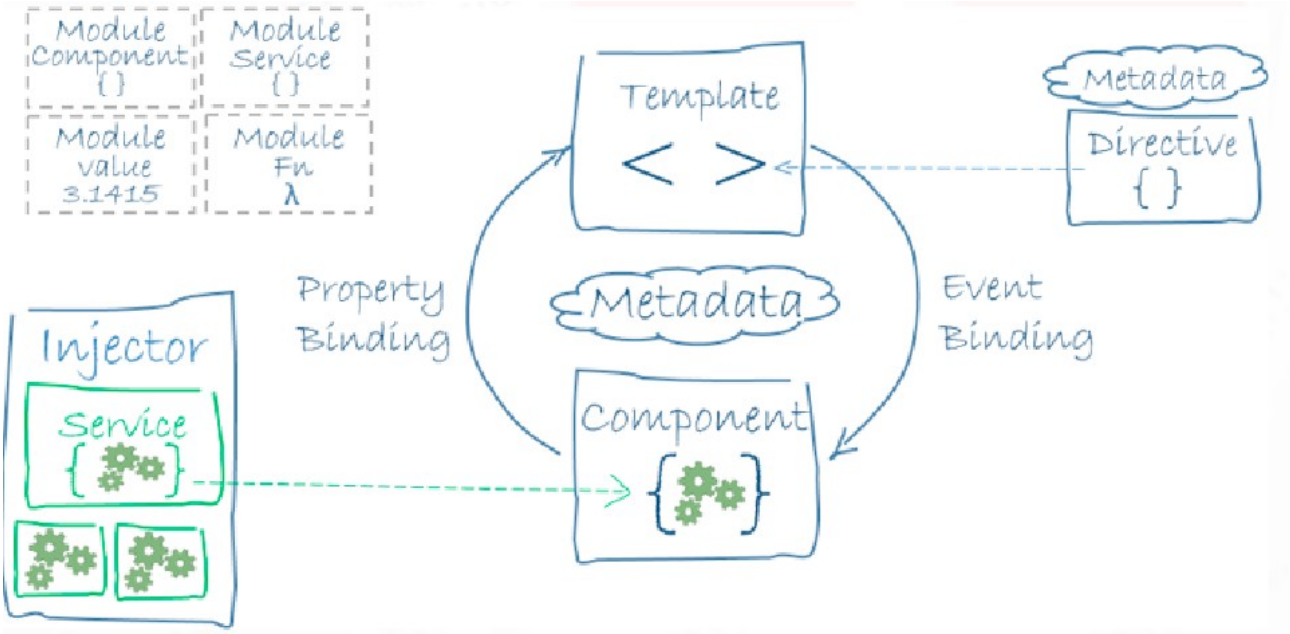
Une template combine du HTML avec un balisage angulaire qui peut modifier les éléments HTML avant qu'ils ne soient affichés. Les directives de modèle fournissent la logique du programme et le data binding connecte les données d'application et le DOM. Il existe deux types de data binding:

- Le event data binding permet à l'application de répondre aux entrées de l'utilisateur dans l'environnement cible en mettant à jour les données de l' application.

- Le property databinding permet d'interpoler les valeurs calculées à partir des données de l’ application dans le HTML.

Services et injection de dépendances :
Pour les données ou la logique qui ne sont pas associées à une vue spécifique et qu’on souhaite partager entre les composants, on crée une classe de service. Une définition de classe de service est immédiatement précédée du décorateur @Injectable (). Le décorateur fournit les métadonnées qui permettent à d'autres fournisseurs d'être injectés en tant que dépendances dans la classe.

L'injection de dépendances (DI) permet de garder les classes de composants simples et efficaces. Ils ne récupèrent pas les données du serveur, ne valident pas l'entrée utilisateur ou ne se connectent pas directement à la console; ils délèguent ces tâches aux services.



Préparez l'environnement de développement

Qu’est-ce que le CLI ?

Le CLI, ou “Command Line Interface” (un outil permettant d'exécuter des commandes depuis la console), d’Angular est l’outil qui vous permet d’exécuter des scripts pour la création, la structuration et la production d’une application Angular.

Installez les outils :

Vous devez installer les outils suivants si vous ne les avez pas déjà sur votre machine :

NODE.JS
Téléchargez et installez la dernière version LTS de Node.js ici :
https://nodejs.org/en/download/

NPM

NPM est un package manager qui permet l'installation d'énormément d'outils et de libraries dont vous aurez besoin pour tout type de développement. Pour l'installer, ouvrez une ligne de commande et tapez la commande suivante :

```
npm install -g npm@latest
```

ANGULAR/CLI
Vous allez maintenant installer le CLI d'Angular de manière globale sur votre machine avec la commande suivante (avec sudo si besoin) :

```
npm install -g @angular/cli
```

À partir de là, la commande `ng` est disponible depuis la ligne de commandes depuis n'importe quel dossier de l'ordinateur.

Pour créer un nouveau projet Angular, naviguez vers le dossier souhaité depuis une ligne de commande et saisissez la commande suivante :

```
ng new mon-premier-projet

ou ng new mon-projet-angular --style=scss --skip-tests=true
```

Ensuite, naviguez dans le dossier du projet et lancez le serveur de développement :

```
cd mon-premier-projet

ng serve --open
```

Si tout se passe bien, vous verrez les informations du serveur qui se lance à l'adresse localhost:4200

Développer une application Angular :

Structurer avec les components :

On peut exécuter une commande pour installer Bootstrap dans le projet. Depuis le dossier `mon-projet-angular`, avec une ligne de commande :

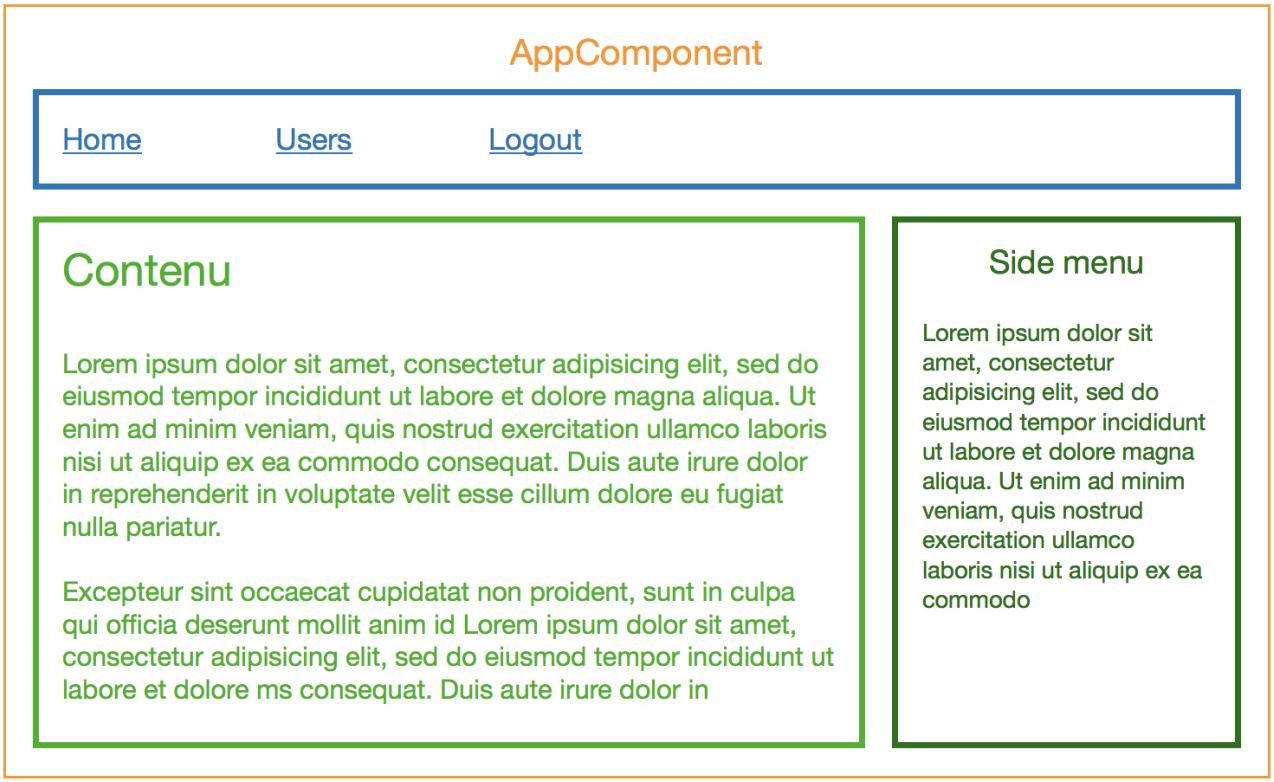
```
npm install bootstrap@3.3.7 --save
```

Cette commande téléchargera Bootstrap et l'intégrera dans le `package.json` du projet.

```
"styles": [
   "../node_modules/bootstrap/dist/css/bootstrap.css",
   "styles.scss"
]
```

La structure des components d'une application Angular :

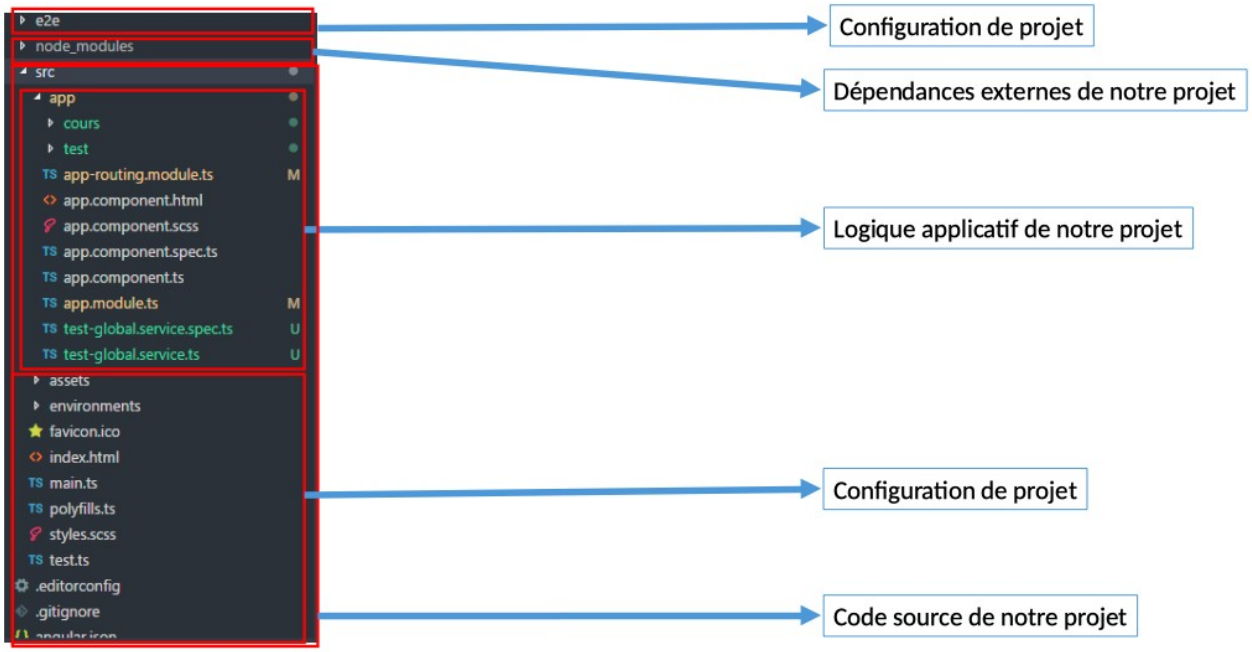
Les components sont les composantes de base d'une application Angular : une application est une arborescence de plusieurs components.



`Le AppComponent` est le component principal : tous les autres components de l' application seront emboîtés ou "nested" dans celui-ci.

Pour cette structure, on peut imaginer un component pour la barre de menu, un autre pour la partie contenu et un dernier pour le menu à droite. Il n'y a pas de règle d'or : je vous montrerai au cours des chapitres suivants comment réfléchir à votre structure afin de trouver la séparation des components la plus pertinente.

Découvrez la structure du code :



Le CLI crée énormément de fichiers au moment de la création d'une nouvelle application. Le dossier `e2e` est généré pour les tests end-to-end.

le dossier `node_modules` contient toutes les dépendances pour votre application : les fichiers source Angular et TypeScript.

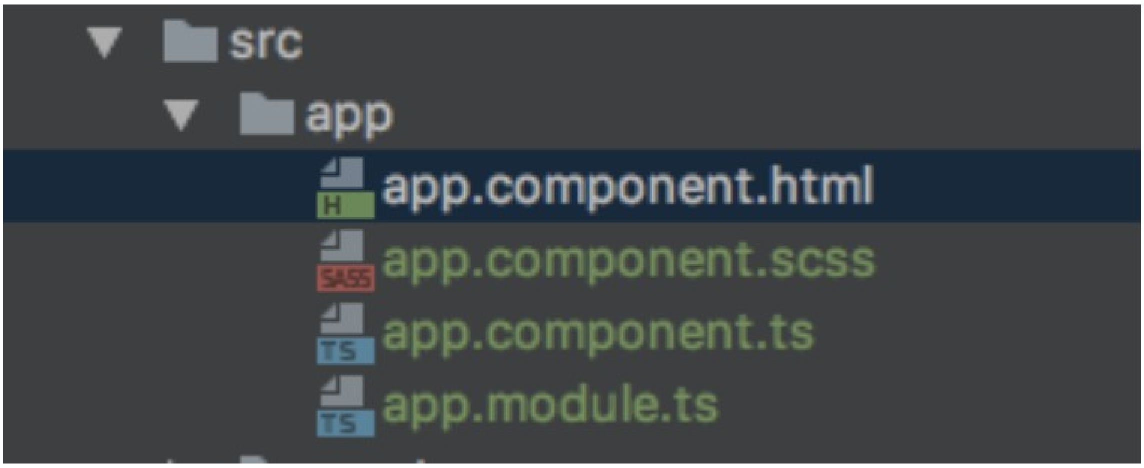
Le dossier `node_modules` contient toutes les dépendances pour votre application : les fichiers source Angular et TypeScript.

Pour commencer à comprendre la structure d'une application Angular, on ouvre `index.html` dans l' éditeur :

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>MonProjetAngular</title>
  <base href="/">

  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

Le ficheier index.html contient la balise vide `<app-root>` : il s'agit d'une balise Angular. Pour en savoir plus, ouvrez le dossier `app` :



Ce dossier contient le module principal de l'application et les trois fichiers du component principal `AppComponent` : son template en HTML, sa feuille de styles en SCSS, et son fichier TypeScript, qui contiendra sa logique.

le fichier `app.component.html` :

```
<!--The content below is only a placeholder and can be replaced.-->
<div style="text-align:center">
  <h1>
    Welcome to {{ title }}!
  </h1>
  <img width="300" alt="Angular Logo"
src="
C9zdmciIHZpZXdCb3g9IjAgMCAyNTAgMjUwIj4KICAgIDxwYXRoIGZpbGw9IiNERDAwMz
EiIGQ9Ik0xMjUgMzBMMzEuOSA2My4ybDE0LjIgMTIzLjFMMTI1IDIzMGw3OC45LTQzLjc
gMTQuMi0xMjMuMXoiIC8+CiAgICA8cGF0aCBmaWxsPSIjQzMwMDJGIiBkPSJNMTI1IDM
wdjIyLjItLjFWMjMwbDc4LjktNDMuNyAxNC4yLTEyMy4xTDEyNSAzMHoiIC8+CiAgICA8c
GF0aCAgZmlsbD0iI0ZGRkZGRiIgZD0iTTEyNSA1Mi4xTDY2LjggMTgyLjZoMjEuN2wxMS4
3LTI5LjJoNDkuNGwxMS43IDI5LjJIMTgzTDEyNSA1Mi4xem0xNyA4My4zaC0zNGwxNy00M
C45IDE3IDQwLjl6IiAvPgogIDwvc3ZnPg==">
</div>
<h2>Here are some links to help you start: </h2>
<ul>
  <li>
    <h2><a target="_blank" rel="noopener" href="https://angular.io/tutorial">Tour of
Heroes</a></h2>
  </li>
  <li>
    <h2><a target="_blank" rel="noopener"
href="https://github.com/angular/angular-cli/wiki">CLI Documentation</a></h2>
  </li>
  <li>
    <h2><a target="_blank" rel="noopener" href="https://blog.angular.io/">Angular blog</a></h2>
  </li>
</ul>
```

Ce fichier contient le code HTML correspondant à ce que vous voyez dans votre navigateur.

Le fichier `app.component.ts` :

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent {
  title = 'app';
}
```

à l'intérieur du décorateur `@Component()` , on trouve un objet qui contient les éléments suivants :

- `selector` : il s'agit du nom qu'on utilisera comme balise HTML pour afficher ce component, comme vous l'avez vu avec `<app-root>` . Ce nom doit être unique et ne doit pas être un nom réservé HTML de type `<div>` , `<body>` etc. On utilisera donc très souvent un préfixe comme `app` , par exemple ;
- `templateUrl` : le chemin vers le code HTML à injecter ;
- `styleUrls` : un array contenant un ou plusieurs chemins vers les feuilles de styles qui concernent ce component ;

Quand Angular rencontre la balise `<app-root>` dans le document HTML, il sait qu'il doit en remplacer le contenu par celui du template `app.component.html` , en appliquant les styles `app.component.scss` , le tout géré par la logique du fichier `app.component.t.`

Créez un component

Pour créer un nouveau component à l'aide du CLI d'Angular. Depuis le dossier principal de votre projet, il faut taper la commande suivante :

```
ng generate component mon-premier
```

```
installing component
  create src/app/mon-premier/mon-premier.component.scss
  create src/app/mon-premier/mon-premier.component.html
  create src/app/mon-premier/mon-premier.component.spec.ts
  create src/app/mon-premier/mon-premier.component.ts
  update src/app/app.module.ts
```

Le CLI nous prévient également qu'il a mis à jour le fichier `app.module.ts` :

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


import { AppComponent } from './app.component';
import { MonPremierComponent } from './mon-premier/mon-premier.component';


@NgModule({
  declarations: [
    AppComponent,
    MonPremierComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Le fichier `mon-premier.component.ts` :

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

@Component({
  selector: 'app-mon-premier',
  templateUrl: './mon-premier.component.html',
  styleUrls: ['./mon-premier.component.scss']
})
export class MonPremierComponent implements OnInit {

  constructor() { }

  ngOnInit() {
  }

}
```

`app-mon-premier` . On peut donc utiliser ce sélecteur dans notre code pour y insérer ce component.

Aller Dans `app.component.html` et modifier-le comme suit :

```
<div style="text-align:center">
  <h1>
    Welcome to {{ title }}!
  </h1>
</div>
```

```
<app-mon-premier></app-mon-premier>
```

Dans `mon-premier.component.html` :

```
<p>
  mon-premier works!
</p>
```

Gérez des données dynamiques

L'intérêt d'utiliser Angular est de pouvoir gérer le DOM (Document Object Model : les éléments HTML affichés par le navigateur) de manière dynamique, et pour cela, il faut utiliser la liaison de données, ou "databinding".

Le databinding, c'est la communication entre le code TypeScript et le template HTML qui est montré à l'utilisateur. Cette communication est divisée en deux directions :

- les informations venant du code qui doivent être affichées dans le navigateur. Les deux principales méthodes pour cela sont le "string interpolation" et le "property binding" ;

- les informations venant du template qui doivent être gérées par le code : l'utilisateur a rempli un formulaire ou cliqué sur un bouton, et il faut réagir et gérer ces événements. On parlera de "event binding".

String interpolation :

L'interpolation est la manière la plus basique d'émettre des données issues du code TypeScript.

On va travailler sur un exemple :

```
ng generate component etudiant
```

Puis aller au niveau du fichier : etudiant.component.html

```
<li class="list-group-item">
  <h4>Etudiant : {{ etudiantName }}</h4>
</li>
```

Editer le fichier app.component.html :

```
<div class="container">
  <div class="row">
    <div class="col-xs-12">
      <h2>Mes Etduiants</h2>
      <ul class="list-group">
        <app-etudiant></app-etudiant>
      </ul>
```

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

Editer le fihcier etudiant.component.ts :

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

@Component({
  selector: 'app-etudiant',
  templateUrl: './etudiant.component.html',
  styleUrls: ['./etudiant.component.scss']
})
export class EtudiantComponent implements OnInit {

  etudiantName: string = 'Mohammed Ben';

  constructor() { }

  ngOnInit() {
  }

}
```

On peut ajouter autres variables :

```
etudiantName: string = ' Mohammed Ben';
etudiantSexe: string = 'Masculin';
```

```
<li class="list-group-item">
  <h4>Etudiant : {{ etudiantName }} -- Sexe : {{ etudiantSexe }}</h4>
</li>
```

On peut utiliser toute expression TypeScript valable pour l'interpolation. Pour démontrer cela, ajouter une méthode au fichier Etudiant`Component` :

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

@Component({
  selector: 'app-etudiant',
  templateUrl: './etudiant.component.html',
  styleUrls: ['./etudiant.component.scss']
})
export class EtudiantComponent implements OnInit {

  etudiantName: string = 'Yassine moha';
  etudiantSexe: string = 'Masculin';
```

```
  constructor() { }

  ngOnInit() {
  }

  getSexe() {
    return this.etudiantSexe ;
  }

}
```

```
<li class="list-group-item">
  <h4>Etudiant : {{ etudiantName }} -- SEXE : {{ getSexe() }}</h4>
</li>
```

Property binding :

La liaison par propriété ou "property binding" est une autre façon de créer de la communication dynamique entre le TypeScript et la template : plutôt qu'afficher simplement le contenu d'une variable, On peut modifier dynamiquement les propriétés d'un élément du DOM en fonction de données dans le TypeScript.

Dans le app.component.ts on ajoute IsAuth varaible de type boolean :

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent {
  isAuth = false;

  constructor() {
    setTimeout(
      () => {
        this.isAuth = true;
      }, 4000
    );
  }


}
```

Dans le fichier app.component.html :

```
<div class="container">
  <div class="row">
    <div class="col-xs-12">
      <h2>Mes Etudiants</h2>
```

```
    <ul class="list-group">
      <app-etudiant></app-etudiant>
    </ul>
    <button class="btn btn-success"  [disabled]="!isAuth" >Tout Activer</button>
   </div>
  </div>
</div>
```

Event binding

Pour ajouter une fonctionnalité liée à l'événement "click" (déclenché quand l'utilisateur clique dessus).

Ajoutez la liaison suivante à votre bouton dans le template HTML :

```
<div class="container">
  <div class="row">
    <div class="col-xs-12">
      <h2>Mes etudiants</h2>
      <ul class="list-group">
        <app-etudiant></app-etudiant>
      </ul>
      <button class="btn btn-success"
            [disabled]="!isAuth"
            (click)="onAllumer()">Tout allumer</button>
    </div>
  </div>
</div>
```

Dans le fichier app.component.ts:

```
onAllumer() {
   console.log('On allume tout !');
}
```

Two-way binding :

La liaison à double sens (ou two-way binding) utilise la liaison par propriété et la liaison par événement en même temps. on l'utilise, par exemple, pour les formulaires, afin de pouvoir déclarer et de récupérer le contenu des champs, entre autres.

Pour pouvoir utiliser le two-way binding, il faut importer `FormsModule` depuis @angular/forms dans l'application.

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

```
import { AppComponent } from './app.component';
import { MonPremierComponent } from './mon-premier/mon-premier.component';
import { EtudiantComponent } from './etudiant/etudiant.component';
import { FormsModule } from '@angular/forms';


@NgModule({
  declarations: [
    AppComponent,
    MonPremierComponent,
    EtudiantComponent
  ],
  imports: [
    BrowserModule,
    FormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Le two-way binding emploie le mélange des syntaxes de property binding et d'event binding : des crochets et des parenthèses `[()]` . Pour une première démonstration, ajoutez un `<input>` dans votre template `etudiant.component.html` et liez-le à la variable etudiant`Name` en utilisant la directive `ngModel`

```
<li class="list-group-item">
  <h4>Etudiant : {{ etudiantName }} -- Statut : {{ getSexe() }}</h4>
  <input type="text" class="form-control" [(ngModel)]="etudiantName">
</li>
```

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

@Component({
  selector: 'app-etudiant',
  templateUrl: './etudiant.component.html',
  styleUrls: ['./etudiant.component.scss']
})
export class EtudiantComponent implements OnInit {

  @Input() etudiantlName: string;

  etudiantSexe: string = 'Masculin';

  constructor() { }

  ngOnInit() {
  }
```

```
  getSexe() {
    return this.etudianSexe;
  }

}
```

Propriétés personnalisées :

Il est possible de créer des propriétés personnalisées dans un component afin de pouvoir lui transmettre des données depuis l'extérieur.

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

@Component({
  selector: 'app-etudiant',
  templateUrl: './etudiant .component.html',
  styleUrls: ['./etudiant.component.scss']
})
export class EtudiantComponent implements OnInit {

  @Input() etudiantName: string;

  etudiantSexe: string = 'éteint';

  constructor() { }

  ngOnInit() {
  }

  getStatus() {
    return this.etudiantSexe;
  }

}
```

```
<div class="container">
  <div class="row">
    <div class="col-xs-12">
      <h2>Mes etudiants</h2>
      <ul class="list-group">
        <app-etudiant etudiantName="Mohammed"></app-etudiant>
        <app- etudiant etudiantName="Yassine"></app-etudiant>
        <app- etudiant etudiantName="Amina"></app-etudiant>
      </ul>
      <button class="btn btn-success"
              [disabled]="!isAuth"
              (click)="onAllumer()">Tout allumer</button>
```

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

On peut plus dynamique de pouvoir passer des variables depuis `AppComponent` pour nommer les étudiants.

```
export class AppComponent {
 isAuth = false;

 etudiant One = 'Machine à laver';
 etudiant Two = 'Frigo';
 etudiant Three = 'Ordinateur';

 constructor() {
```

Il faut créer d'abord vos trois variables dans `AppComponent` :

```
export class AppComponent {
 isAuth = false;

 etudiantOne = ‘Yassine’';
etudiantTwo = 'Lotfi';
etudiantThree = 'Nadia';

 constructor() {
```

Utiliser les crochets `[]` pour lier le contenu de ces variables à la propriété du component dans app.component.html :

```
<ul class="list-group">
   <app- etudiant [etudiantName]="yassine"></app-etudiant>
   <app- etudiant [etudiantName]="mohammed"></app-etudiant>
   <app- etudiant [etudiantName]="amina"></app-etudiant>
</ul>
```

On peut également créer une propriété pour régler l'état de l’étudiant :

```
export class EtudiantComponent implements OnInit {

 @Input() etudaintName: string;
 @Input() etudiantSexe: string;

 constructor() {
```

```
<ul class="list-group">
   <app- etudiant [etudiantName]="yassine" [etudiantSexe]="'Masculin'"></app-etudiant>
```

```
   <app- etudiant [etudiantName]="mohammed" [etudiantSexe]="'Masculin'"></app-etudiant>
   <app- etudiant [etudiantName]="amina" [etudiantSexe]="'Feminin'"></app-etudiant>
</ul>
```

Structurez le document avec des Directives :

Les directives sont des instructions intégrées dans le DOM que on utilisere presque systématiquement quand on créere des applications Angular. Quand Angular lit la template et rencontre une directive qu'il reconnait, il suit les instructions correspondantes.

Il existe deux types principaux de directive : les directives structurelles et les directives par attribut.

Les directives structurelles :

Ce sont des directives qui, comme leur nom l'indique, modifient la structure du document. Dans ce chapitre, on va en découvrir deux (il en existe d'autres) : `*ngIf` , pour afficher des données de façon conditionnelle, et `*ngFor` , pour itérer des données dans un array.

*ngIf :

Un component auquel on ajoute la directive `*ngIf="condition"` ne s'affichera que si la condition est "truthy" (elle retourne la valeur `true` où la variable mentionnée est définie et non-nulle), comme un statement `if` classique.

```
<li class="list-group-item">
  <div style="width:20px;height:20px;background-color:red;"
      *ngIf="etudiantSexe === 'Masculin'"></div>
  <h4>Etudiant : {{ etudiantName }} -- Sexe : {{ getSexe() }}</h4>
  <input type="text" class="form-control" [(ngModel)]="etudiantName">
</li>
```

*ngFor :

Lorsque l'on ajoute la directive `*ngFor="let obj of myArray"` à un component, Angular itérera l'array `myArray` et affichera un component par objet `obj` .

```
export class AppComponent {
  isAuth = false;

  etudiants = [
    {
      name: 'lotfi',
      sexe: 'Masculin'
    },
    {
```

```
    name: 'yassine',
    sexe: 'Masculin'
  },
  {
    name: 'sara',
    sexe: 'Feminin'
  }
];

constructor() {
```

```
<div class="container">
 <div class="row">
  <div class="col-xs-12">
   <h2>Mes Etudiants</h2>
   <ul class="list-group">
    <app-etudiant  *ngFor="let etudiant of etudiants"
                [etudiantName]="etudiant.name"
                [etudiantSexe]="etudiant.sexe"></app-etudiant>
   </ul>
   <button class="btn btn-success"
         [disabled]="!isAuth"
         (click)="onAllumer()">Tout allumer</button>
  </div>
 </div>
</div>
```

Les directives par attribut :

À la différence des directives structurelles, les directives par attribut modifient le comportement d'un objet déjà existant.

ngStyle

Cette directive permet d'appliquer des styles à un objet du DOM de manière dynamique.

```
<h4 [ngStyle]="{color: getColor()}">Etudiant : {{ etudiantName }} -- Sexe : {{ getSexe()
}}</h4>
```

Dans EtudiantComponent on va maintenant créer :

```
getColor() {
  if(this.etudiantSexe === 'Masculin') {
    return 'green';
  } else if(this.etudiantSexe === 'Feminin') {
    return 'red';
  }
}
```

ngClass

Au-delà de modifier des styles directement, il peut être très utile d'ajouter des classes CSS à un élément de manière dynamique.

```
<li [ngClass]="{'list-group-item': true,
              'list-group-item-success': etudiant Status === 'free',
              'list-group-item-danger': etudiant Status === 'busy'}">
  <div style="width:20px;height:20px;background-color:red;"
      *ngIf="etudiantSexe === 'Masculin'"></div>
  <h4 [ngStyle]="{color: getColor()}">Etudiant : {{ etudiantName }} -- Statut :
{{ getSexe() }}</h4>
  <input type="text" class="form-control" [(ngModel)]="etudiantName">
</li>
```

Modifiez les données en temps réel avec les Pipes :

Les pipes (/pʌɪp/) prennent des données en input, les transforment, et puis affichent les données modifiées dans le DOM. Il y a des pipes fournis avec Angular, et on peut créer des pipes personnalisés.

Utilisez et paramétrez les Pipes :

Un pipe que l'on utilise très souvent est DatePipe , qui analyse des objets JS de type Date et qui les affiche d'une manière plus lisible que leur encodage de base.

```
export class AppComponent {
  isAuth = false;
  lastUpdate = new Date();
```

```
<h2>Mes etudiants</h2>
<p>Mis à jour : {{ lastUpdate }}</p>
```

L'objet Date a bien été créé, mais sous sa forme actuelle, il n'est pas très utile. L'avantage d'un pipe est de pouvoir modifier l'affichage de cet objet sans en modifier la nature. Ajoutons le DatePipe dans le template grâce au caractère | :

```
<p>Mis à jour : {{ lastUpdate | date }}</p>
<p>Mis à jour : {{ lastUpdate | date: 'short' }}</p>
<p>Mis à jour : {{ lastUpdate | date: 'yMMMMEEEEd' }}</p>
```

Utilisez une chaîne de Pipes :

On peut avoir besoin de plusieurs pipes pour un seul élément du DOM. Imaginez, par exemple, que vous souhaitiez afficher la date de l'exemple précédent en majuscules.

```
<p>Mis à jour : {{ lastUpdate | date: 'yMMMMEEEEd' | uppercase }}</p>
```

Async :

Le pipe `async` est un cas particulier mais extrêmement utile dans les applications Web, car il permet de gérer des données asynchrones, par exemple des données que l'application doit récupérer sur un serveur.

```
lastUpdate = new Promise((resolve, reject) => {
   const date = new Date();
   setTimeout(
     () => {
       resolve(date);
     }, 2000
   );
 });
```

Si on enregistre le fichier, l'application va créer une erreur :

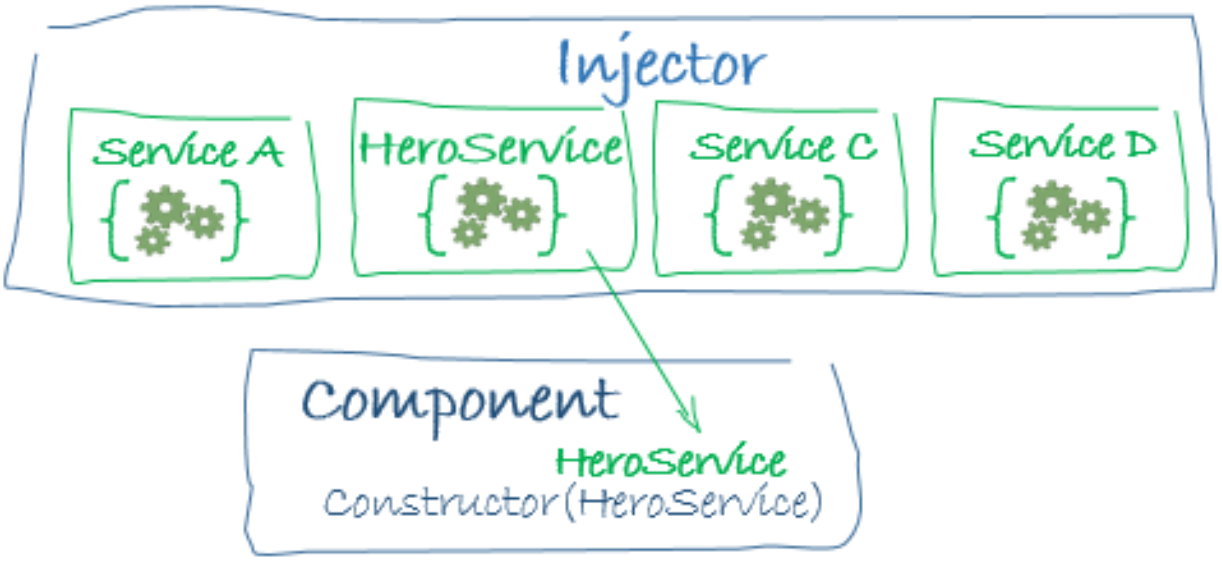
```
Error: InvalidPipeArgument: '[object Promise]' for pipe
```

`'DatePipe'`.

Il nous faut donc ajouter AsyncPipe en début de chaîne pour dire à Angular d'attendre l'arrivée des données avant d'exécuter les autres pipes :

```
<p>Mis à jour : {{ lastUpdate | async | date: 'yMMMMEEEEd' | uppercase }}</p>
```

Enrichir l'application Angular



Améliorez la structure du code avec les Services :

Un service permet de centraliser des parties de code et des données qui sont utilisées par plusieurs parties de l 'application ou de manière globale par l'application entière. Les services permettent donc :

- de ne pas avoir le même code doublé ou triplé à différents niveaux de l'application - ça facilite donc la maintenance, la lisibilité et la stabilité du code ;
- de ne pas copier inutilement des données - si tout est centralisé, chaque partie de l'application aura accès aux mêmes informations, évitant beaucoup d'erreurs potentielles.

Dans le cas de l'application que vous avez créée lors des derniers chapitres, on pourrait imaginer un service EtsudiantService qui contiendrait les données des étudiants.

Injection et instances :

Pour être utilisé dans l'application, un service doit être injecté, et le niveau choisi pour l'injection est très important. Il y a trois niveaux possibles pour cette injection :

- dans `AppModule` : ainsi, la même instance du service sera utilisée par tous les components de l'application et par les autres services ;
- dans `AppComponent` : comme ci-dessus, tous les components auront accès à la même instance du service mais non les autres services ;
- dans un autre component : le component lui-même et tous ses enfants (c'est-à-dire tous les components qu'il englobe) auront accès à la même instance du service, mais le reste de l'application n'y aura pas accès.

Pour les exemples de ce cours, on injecte systématiquement les services dans `AppModule` pour rendre disponible une seule instance par service à toutes les autres parties de l'application.

Créer maintenant un sous-dossier `services` dans `app`, et créez-y un nouveau fichier appelé etudiant`.service.ts` :

```
ng generate service  Etudiant
```

```
export class EtudiantService {

}
```

On injecte ce service dans `AppModule` en l'ajoutant à l'array `providers` (n'oubliez pas d'ajouter l'import correspondant en haut du fichier :

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

```
import { AppComponent } from './app.component';
import { MonPremierComponent } from './mon-premier/mon-premier.component';
import { EtudiantComponent } from './etudiant/etudiant .component';
import { FormsModule } from '@angular/forms';
import { EtudiantService } from './services/etudiant.service';


@NgModule({
  declarations: [
    AppComponent,
    MonPremierComponent,
    Etudiant Component
  ],
  imports: [
    BrowserModule,
    FormsModule
  ],
  providers: [
    EtudiantService
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Angular crée maintenant une instance du service `EtudiantService` pour l'application entière. Pour l'intégrer dans un component, on le déclare comme argument dans son constructeur. Intégrez-le dans `AppComponent` (sans oublier d'ajouter l'import en haut) :

```
constructor(private etudiantService: EtudiantService) {
    setTimeout(
      () => {
        this.isAuth = true;
      }, 4000
    );
  }
```

Utilisez les services :

Le premier élément qu'il serait logique de déporter dans le service serait l'array etudiants. Copiez-le depuis `AppComponent` , collez-le dans Etudiant`Service` et, de nouveau dans `AppComponent` , déclarez etudiants simplement comme un array de type `any` :

```
export class EtudiantService {
  etudiants = [
    {
      name: ' Mohammed‘,
      sexe: 'Masculin'
    },
```

```
    {
      name: 'Yassine',
      status: 'Masculin'
    },
    {
      name: 'Sara',
      status: 'Feminin'
    }
  ];
}
```

```
export class AppComponent {

  isAuth = false;

  etudiants: any[];
```

Il faut maintenant que `AppComponent` puisse récupérer les informations stockées dans `EtudiantService`. Pour cela, vous allez implémenter la méthode `ngOnInit()`.

```
import { Component, OnInit } from '@angular/core';
import { EtudiantService } from './services/etudiant .service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent implements OnInit {

constructor(private etudiantService: EtudiantService) {
    setTimeout(
      () => {
        this.isAuth = true;
      }, 4000
    );
  }

ngOnInit() {

    this.etudiants = this.etudiantService.etudiants;
}
}
```

L ‘application devrait fonctionner à nouveau, avec la liste des étudiants qui s'affiche comme avant. Il n'y a aucune différence visuelle, mais votre code est maintenant plus modulaire,

Gérez la navigation avec le Routing

L'un des énormes avantages d'utiliser Angular est de pouvoir créer des "single page application" (SPA). Sur le Web, ces applications sont rapides et lisses : il n'y a qu'un seul chargement de page au début, et même si les données mettent parfois du temps à arriver, la sensation pour l'utilisateur est celle d'une application native.

Au lieu de charger une nouvelle page à chaque clic ou à chaque changement d'URL, on remplace le contenu ou une partie du contenu de la page : on modifie les components qui y sont affichés, ou le contenu de ces components. On accomplit tout cela avec le "routing", où l'application lit le contenu de l'URL pour afficher le ou les components requis.

On peut continuer sur l'exemple précédent en ajoutant un mécanisme authentification.

```
ng g c auth
```

On va également devoir modifier un peu l'organisation actuelle afin d'intégrer plus facilement le routing : vous allez créer un component qui contiendra toute la view actuelle et qui s'appellera Etudiant`ViewComponent` :

```
ng g c etudiant-view
```

Ensuite, coupez tout le contenu de la colonne dans `app.component.html`, enregistrez-le dans etudiant`-view.component.html`, et remplacez-le par la nouvelle balise `<app-etudiant-view>` :

```
<div class="container">
  <div class="row">
    <div class="col-xs-12">
      <app-etudiant-view></app-etudiant-view>
    </div>
  </div>
</div>
```

Il faudra également déménager la logique de cette view pour que tout re-marche : injectez Etudiant`Service`, créez l'array etudiants, intégrez la logique `ngOnInit.`

```
import { Component, OnInit } from '@angular/core';
import { EtudiantService } from '../services/etudiant.service';

@Component({
  selector: 'app-etudiant-view',
  templateUrl: './etudiant-view.component.html',
```

```
  styleUrls: ['./aetudiant-view.component.scss']
})
export class EtudiantViewComponent implements OnInit {

  etudiants: any[];

  lastUpdate = new Promise((resolve, reject) => {
    const date = new Date();
    setTimeout(
      () => {
        resolve(date);
      }, 2000
    );
  });

  constructor(private etudiantService: EtudiantService) { }

  ngOnInit() {
    this.etudiants = this.etudiantService.etudiants ;
  }



}
```

Ajouter la barre de navigation suivante à `AppComponent` :

```
<nav class="navbar navbar-default">
  <div class="container-fluid">
    <div class="navbar-collapse">
      <ul class="nav navbar-nav">
        <li><a href="#">Authentification</a></li>
        <li class="active"><a href="#">Etudiants</a></li>
      </ul>
    </div>
  </div>
</nav>
```

Créez des routes

Puisque le routing d'une application est fondamentale pour son fonctionnement, on déclare les routes dans `app.module.ts` .

On crée une constante de type `Routes` (qu'on importe depuis `@angular/router`) qui est un array d'objets JS qui prennent une certaine forme :

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

import { AppComponent } from './app.component';
import { MonPremierComponent } from './mon-premier/mon-premier.component';
import { EtudiantComponent } from './etudiant/etudiant.component';
```

```
import { FormsModule } from '@angular/forms';
import { EtudiantService } from './services/etudiant.service';
import { AuthComponent } from './auth/auth.component';
import { EtudiantViewComponent } from './etudiant-view/etudiant-view.component';
import { Routes } from '@angular/router';

const appRoutes: Routes = [
  { path: 'etudiants', component: EtudiantViewComponent },
  { path: 'auth', component: AuthComponent },
  { path: '', component: EtudiantViewComponent }
];
```

Le path correspond au string qui viendra après le `/` dans l'URL : sur votre serveur local, le premier path ici correspond donc à `localhost:4200/etudiants`.

Les routes sont maintenant créées, mais il faut les enregistrer dans l'application. Pour cela, on va importer `RouterModule` depuis `@angular/router` et on va l'ajouter à l'array imports de `AppModule`, tout en lui appelant la méthode `forRoot()` en lui passant l'array de routes que vous venez de créer :

```
imports: [
    BrowserModule,
    FormsModule,
    RouterModule.forRoot(appRoutes)
],
```

Maintenant que les routes sont enregistrées, il ne reste plus qu'à dire à Angular où on souhaite afficher les components dans le template lorsque l'utilisateur navigue vers la route en question. On utilise la balise `<router-outlet>` :

```
<div class="container">
  <div class="row">
    <div class="col-xs-12">
      <router-outlet></router-outlet>
    </div>
  </div>
</div>
```

Naviguez avec les routerLink :

Afin que l'utilisateur puisse naviguer à l'intérieur de l'application, il est nécessaire de créer des liens ou des boutons qui naviguent vers les routes qu' on a creé.

Du coup, on retire l'attribut `href` et on le remplace par l'attribut `routerLink` :

```
<nav class="navbar navbar-default">
  <div class="container-fluid">
    <div class="navbar-collapse">
      <ul class="nav navbar-nav">
```

```
      <li><a routerLink="auth">Authentification</a></li>
      <li class="active"><a routerLink="etudiants">Etudiants</a></li>
    </ul>
   </div>
  </div>
</nav>
<div class="container">
  <div class="row">
   <div class="col-xs-12">
    <router-outlet></router-outlet>
   </div>
  </div>
</div>
```

Naviguez avec le Router :

Il peut y avoir des cas où on aura besoin d'exécuter du code avant une navigation. Par exemple, on peut avoir besoin d'authentifier un utilisateur et, si l'authentification fonctionne, de naviguer vers la page que l'utilisateur souhaite voir.

Tout d'abord, créer un nouveau fichier `auth.service.ts` dans le dossier services pour gérer l'authentification (n'oubliez pas de l'ajouter également dans l'array `providers` dans `AppModule`) :

```
export class AuthService {

  isAuth = false;

  signIn() {
    return new Promise(
      (resolve, reject) => {
        setTimeout(
          () => {
            this.isAuth = true;
            resolve(true);
          }, 2000
        );
      }
    );
  }

  signOut() {
    this.isAuth = false;
  }
}
```

La variable `isAuth` donne l'état d'authentification de l'utilisateur. La méthode `signOut()` "déconnecte" l'utilisateur, et la méthode `signIn()` authentifie automatiquement l'utilisateur au bout de 2 secondes, simulant le délai de communication avec un serveur.

Dans le component `AuthComponent` :

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

@Component({
  selector: 'app-auth',
  templateUrl: './auth.component.html',
  styleUrls: ['./auth.component.scss']
})
export class AuthComponent implements OnInit {

  authStatus: boolean;

  constructor(private authService: AuthService) { }

  ngOnInit() {
    this.authStatus = this.authService.isAuth;
  }

  onSignIn() {
    this.authService.signIn().then(
      () => {
        console.log('Sign in successful!');
        this.authStatus = this.authService.isAuth;
      }
    );
  }

  onSignOut() {
    this.authService.signOut();
    this.authStatus = this.authService.isAuth;
  }

}
```

Puisque la méthode `signIn()` du service retourne une Promise, on peut employer une fonction callback asynchrone avec `.then()` pour exécuter du code une fois la Promise résolue.

```
<h2>Authentification</h2>
<button class="btn btn-success" *ngIf="!authStatus" (click)="onSignIn()">Se connecter</button>
<button class="btn btn-danger" *ngIf="authStatus" (click)="onSignOut()">Se
déconnecter</button>
```

Le comportement recherché serait qu'une fois l'utilisateur authentifié, l'application navigue automatiquement vers la view des etudiants. Pour cela, il faut injecter le `Router` (importé depuis `@angular/router`) pour accéder à la méthode `navigate()` :

```
constructor(private authService: AuthService, private router: Router) { }

onSignIn() {
   this.authService.signIn().then(
     () => {
       console.log('Sign in successful!');
       this.authStatus = this.authService.isAuth;
       this.router.navigate(['etudiants']);
     }
   );
}
```

La fonction navigate prend comme argument un array d'éléments, qui, dans ce cas, n'a qu'un seul membre : le `path` souhaité.

Paramètres de routes :

On peut passer des paramètres a travers des routes ou bien des liens.

Tout d'abord, on va créer la route dans `AppModule` :

```
const appRoutes: Routes = [
  { path: 'etudiants', component: EtudiantViewComponent },
  { path: 'etudiants/:id', component: SingleEtudiantComponent },
  { path: 'auth', component: AuthComponent },
  { path: '', component: EtudiantViewComponent }
];
```

Pour SingleEtudiantComponent :

```
import { Component, OnInit } from '@angular/core';
import { EtudiantService } from '../services/etudiant.service';

@Component({
  selector: 'app-single-etudiant',
  templateUrl: './single-etudiant.component.html',
  styleUrls: ['./single-etudiant .component.scss']
})
export class SingleEtuidant Component implements OnInit {

  name: string = 'etudiant';
  sexe: string = 'Masculin';

  constructor(private etudiantService: EtudiantService) { }

  ngOnInit() {
  }

}
```

Code Html :

```
<h2>{{ name }}</h2>
<p>Statut : {{ sexe }}</p>
<a routerLink="/etudiants">Retour à la liste</a>
```

Pour l'instant, si vous naviguez vers `/etudiants/nom`, peu importe le nom que vous choisissez, vous avez accès à `SingleEttudiantComponent`. Maintenant, vous allez y injecter `ActivatedRoute`, importé depuis `@angular/router`, afin de récupérer le fragment `id` de l'URL :

```
constructor(private etudiantService: EtudiantService,
          private route: ActivatedRoute) { }
```

Puis, dans `ngOnInit()`, vous allez utiliser l'objet `snapshot` qui contient les paramètres de l'URL et, pour l'instant, attribuer le paramètre `id` à la variable `name` :

```
ngOnInit() {
   this.name = this.route.snapshot.params['id'];
}
```

Pour atteindre l'objectif souhaité, commencez par ajouter, dans Etudiant`Service`, un identifiant unique pour chaque étudiant et une méthode qui rendra l' étudiant correspondant à un identifiant :

```
export class EtudiantService {
  etudiants = [
    {  id: 1,
      name: ' Mohammed‘,
      sexe: 'Masculin'
    },
    {  id: 2,
      name: 'Yassine',
      status: 'Masculin'
    },
    {
       id: 3,
      name: 'Sara',
      status: 'Feminin'
    }
  ];
}


//


getEtdiantById(id: number) {
   const etudiant = this.etudiants.find(
     (s) => {
       return s.id === id;
```

```
    }
  );
  return etudiant;
}
```

Maintenant, dans `SingleEtudiantComponent` , vous allez récupérer l'identifiant de l'URL et l'utiliser pour récupérer l'etudiant correspondant :

```
ngOnInit() {
  const id = this.route.snapshot.params['id'];
  this.name = this.etudiantService.getEtudiantById(+id).name;
  this.sexe = this.etudiantService.getAEtudiantById(+id).sexe;
}
```

Pour finaliser cette fonctionnalité, intégrez l'identifiant unique dans Etudiant`Component` et dans Etudiant`ViewComponent` , puis créez un `routerLink` pour chaque etudiant qui permet d'en regarder le détail :

```
@Input() etudiantName: string;
@Input() etudiantSexe: string;
@Input() id: number;
```

```
<ul class="list-group">
  <app-etudiant  *ngFor="let etudiant of etudiants; let i = index"
            [etudiant Name]="etudiant .name"
            [etudiant Sexe]="etudiant .sexe"
            [id]="etudiant .id"></app-etudiant>
</ul>
```

```
<h4 [ngStyle]="{color: getColor()}">Etudiant : {{ etudiantName }} -- Statut : {{ getSexe()
}}</h4>
<a [routerLink]="[id]">Détail</a>
```

Redirection :

Il peut y avoir des cas de figure où l'on souhaiterait rediriger un utilisateur, par exemple pour afficher une page 404 lorsqu'il entre une URL qui n'existe pas.
Pour l'application des étudiants, commencez par créer un component 404 très simple, appelé `four-oh-four.component.ts` :

```
<h2>Erreur 404</h2>
<p>La page que vous cherchez n'existe pas !</p>
```

```
const appRoutes: Routes = [
  { path: 'etudiants', component: EtudiantViewComponent },
  { path: 'etudiants/:id', component: SingleEtudiantComponent },
```

```
  { path: 'auth', component: AuthComponent },
  { path: '', component: EtudiantViewComponent },
  { path: 'not-found', component: FourOhFourComponent },
  { path: '**', redirectTo: 'not-found' }
];
```

Interagissez avec un serveur avec HttpClient

Dans une application Angular, vous aurez très souvent besoin de faire des appels à un backend ou à un autre serveur.

Angular met à disposition un service appelé `HttpClient` qui permet de créer et d'exécuter des appels HTTP (fait par AJAX - Asynchronous JavaScript and XML) et de réagir aux informations retournées par le serveur.

Envoyez vers le backend :

Pour avoir accès au service `HttpClient`, il faut tout d'abord ajouter `HttpClientModule`, importé depuis `@angular/common/http`, à votre `AppModule`

```
imports: [
    BrowserModule,
    FormsModule,
    ReactiveFormsModule,
    HttpClientModule,
    RouterModule.forRoot(appRoutes)
],
```

Pour utiliser `HttpClient`, dans un premier temps, pour la gestion des données de la liste d'étudiants . Vous allez donc l'injecter dans `EtudiantService`, en y ayant auparavant ajouté le décorateur `@Injectable()` (importé depuis `@angular/core`)

```
import { Subject } from 'rxjs/Subject';
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Injectable()
export class EtudiantService {

 ….



  constructor(private httpClient: HttpClient) { }
```

Créer une méthode qui va enregistrer l'array étudiants dans la base de données au endpoint `/etudiants` par la méthode POST :

```
saveEtudiantsToServer() {
   this.httpClient
     .post('https://link', this.etudiants)
     .subscribe(
       () => {
         console.log('Enregistrement terminé !');
       },
       (error) => {
         console.log('Erreur ! : ' + error);
       }
     );
}
```

Créer maintenant un bouton dans Etudiant`ViewComponent` qui déclenche cette sauvegarde.

```
…...
<button class="btn btn-primary"
       (click)="onSave()">Enregistrer les etudiants</button>
```

```
onSave() {
   this.etudiantService.saveEtudiantsToServer();
}
```

Recevez depuis le backend :

Afin de demander la liste des étudiants (maintenant stocké au endpoint `/etudiants`), vous allez créer une nouvelle méthode qui emploie la méthode `get()` dans Etudiant`Service` :

```
getEtudiantsFromServer() {
   this.httpClient
     .get<any[]>('https://link/etudiant.json')
     .subscribe(
       (response) => {
         this.etudiants = response;

       },
       (error) => {
         console.log('Erreur ! : ' + error);
       }
     );
}
```

```
…..
<button class="btn btn-primary"
       (click)="onFetch()">Récupérer les etudiants</button>
```

```
onFetch() {
   this.etudiantService.getEtudiantsFromServer();
}
```

References :
https://openclassrooms.com/fr/courses/4668271-developpez-des-applications-web-avec-angular