Framework Angular

Adil Anwar
anwar@emi.ac.ma

Programme

- **Partie 1 : Généralités sur le Framework Angular**
  - Principe des single pages application
  - Découvrir TypeScript
  - **Labs** : Préparation de l'environnement de travail
    - Installation du package typescript
    - Installation du package Angular-CLI
    - Développer une application avec TypeScript

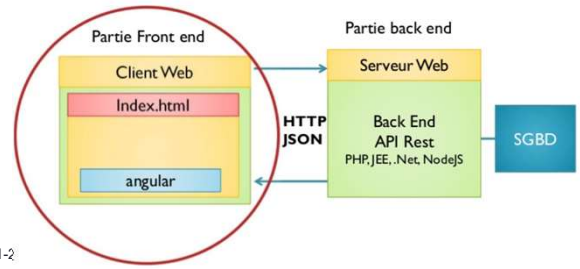
Programme

- **Partie 2 : Les bases du Framework Angular**
  - Les composants
  - La syntaxe du templates
  - Data Binding (liaison de données)
  - Les services
  - Les pipes
  - Les directives
  - Les modules
  - Routage
  - Echanges de données via HTTP
  - Communication avec les utilisateurs : Forms
  - Déploiement d'une application Angular dans un serveur de production
  - **Lab** : mettre en place une application web avec Angular

Partie 1 : : Généralités sur le Framework Angular

Présentation du Framework Angular

- Angular est un framework de développment web qui permet de pour créer des applications utilisant HTML et JavaScript / TypeScript.
- Il permet de créer des applications en composant des **templates** HTML avec un balisage personnalisé (propre à Angular)
- Avec Angular on créer des classes de **composants** pour gérer ces templates à l'aide de la **liaison de données**, en ajoutant une logique applicative dans les **services** et en mettant en boîte des composants et des services dans des **modules**.



Présentation du Framework Angular

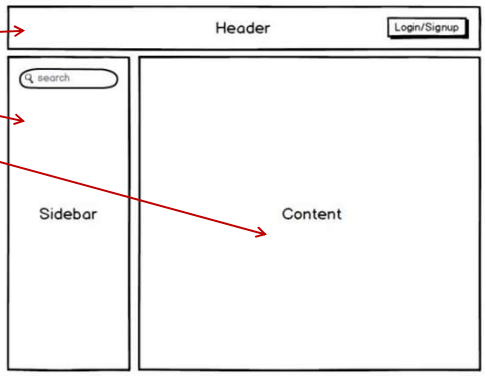
- Angular est un framework JavaScript qui permet de créer des SPAs réactives.
- Single Page Application
  - Application dont la navigation se fait sans rechargement de la page
  - Html minimaliste composé uniquement de blocs de récupération de code JavaScript (sous forme de bundle(s))
  - Le DOM (HTML) est entièrement manipulé par le JavaScript
  - Consultation du serveur uniquement pour manipuler des ressources (CRUD).
  - Applis Web avec une expérience proche de celle d'applications mobiles

Philosophie d'Angular

- Angular est un framework orienté composant. Tu vas écrire de petits composants, et assemblés, ils vont constituer une application complète.
- Un composant est un groupe d'éléments HTML, dans un template, dédiés à une tâche particulière.
- Pour cela, tu auras probablement besoin d'un peu de logique métier derrière ce template, pour peupler les données, et réagir aux événements par exemple.
- Web components c'est quoi ?
  - Un standard a été défini autour de ces composants : le standard Web Component ("composant web"). Même s'il n'est pas encore complètement supporté dans les navigateurs, on peut déjà construire des petits composants isolés, réutilisables dans différentes applications.
  - Cette orientation composant est largement partagée par de nombreux frameworks front-end : c'est le cas depuis le début de ReactJS. EmberJS et AngularJS ont leur propre façon de faire quelque chose de similaire ; et les petits nouveaux Aurelia ou Vue.js parient aussi sur la construction de petits composants.

Web components : Exemple

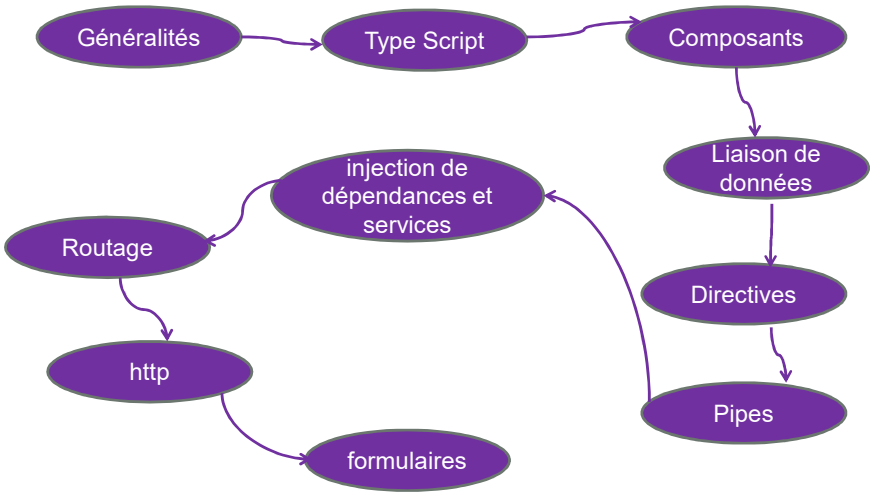
- Avant de commencer le développement d'une application Angular, il faut:
  - Découper l'application en plusieurs composants
  - Décrire la responsabilité de chaque composant
  - Spécifier les inputs/outputs de chaque composant
- Içi nous avons créer trois composants
  - Header Component
  - SidebarComponent
  - ContentComponent



AngularJS, Angular 2, Angular 7

- AngularJS est la première version qui a fait la popularité du framework.
- Angular 2 est une réécriture complète du Framework AngularJS. Sa base représente le socle futur du framework
- Angular 4 est un update d'Angular 2. La version 3 a été sautée pour des "problématiques" de versioning,
- Angular sort une nouvelle version majeure tous les 6 mois :
  - Angular 2 est sorti en septembre 2016,
  - Angular 4 en mars 2017,
  - Angular 5 en novembre 2017,
  - Angular 6 en mai 2018,
  - Angular 7 en octobre 2018,
  - Angular 8 en mai 2019.
- Dernière version, 8.2.9 (2 octobre 2019)
- les mises à jour sont désormais complètement automatisées. Pour exemple, cela prend 5 minutes de passer d'Angular 6 à 8 avec la commande *ng update*.

Angular : Feuille de Route



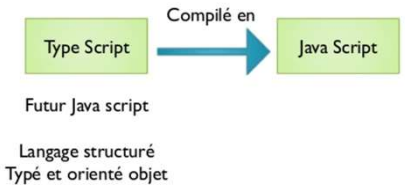
Découvrir TypeScript

Développement web avec JavaScript

- JavaScript est un excellent langage pour développer des applications de petites tailles => inadapté pour les applications complexes,
  - un code JavaScript complexe
  - nombreuses lacunes dans le langage (typage dynamique, etc.).
- Évolution du langage, il est devenu le moyen idéal pour écrire :
  - des applications multiplates-formes (bureau, mobile, web, etc.).
  - Des applications indépendantes aux navigateurs
- Mais, lors de sa création, JavaScript n'était pas destiné à cela, il était principalement conçu pour des utilisations simples dans de petites applications.
- **Constatations** :
  - besoin de suivre la croissance spectaculaire de l'utilisation de JavaScript,
  - besoin de simplification et d'amélioration du langage
  - besoin de détecter le plus tôt les erreurs au cours du développement, plutôt qu'au moment de l'exécution
  - évoluer le langage afin de créer facilement et efficacement des applications de grande taille.

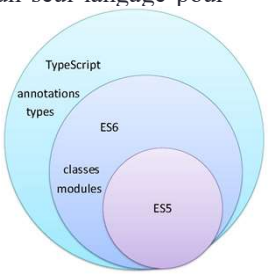
Découvrir TypeScript

- TypeScript est un langage de programmation open source développé et maintenu par Microsoft. C'est un **sur-ensemble** typé de JavaScript qui se compile en JavaScript. Par conséquent, un programme JavaScript est également un programme TypeScript valide
- TypeScript ajoute le **typage statique** ou strict au langage JavaScript, en plus de la programmation orientée objet basée sur les classes.
- Les navigateurs ne comprenant que JavaScript, lorsque vous écrivez votre application en TypeScript, vous avez besoin d'un compilateur (**Transpileur**) qui puisse compiler votre code et le convertir en JavaScript.
- Transpiler, tout comme la compilation, transforme du code dans un langage vers un autre langage du même niveau : ex Exemple transformer la version de JavascriptES6 vers la version ES5



Propriétés du langage TypeScript

- TypeScript peut être utilisé pour écrire à la fois le code côté client (exécuté sur un navigateur) et le code côté serveur (exécuté sur le serveur d'applications) à l'aide de la bibliothèque Node.js.
- Cela signifie qu'avec TypeScript, vous n'apprenez qu'un seul langage pour devenir un développeur Web à pile complète (fullstack).
- TypeScript implémenté les fonctionnalités de ES6
  - Variables fortement typées.
  - Paramètres de fonction fortement typés.
  - Des classes.
  - Interfaces
  - Héritage.
  - Modules permettant de diviser les éléments en éléments constitutifs permettant à plusieurs personnes de développer une application.



TypeScript et Angular

- A partir de la version 2.0 d'Angular, le framework entend tirer parti des nouveautés de EcmaScript 6 et même plus,
- avec TypeScript, qui rassemble à ES6 et un système de typage et d'annotations (décorateurs).
- Les développeurs pourront choisir d'écrire en TypeScript (préconisé) ou en ES6 (très rare) avec une phase de transpilation vers ES5 ou directement en ES5 (extrêmement rare).

Installation de TypeScript

- Nous avons installé le package Typescript :

```
npm install -g typescript
```

- l'option : **-g** indique que l'installation est globale à tous les projets, nous n'aurons plus à le refaire
- Nous pouvons maintenant créer un fichier TypeScript: test.ts
- Dans la console :

- Indique au transpileur de transformer les fichiers **.ts** en **.js** à chaque modification de ceux-ci.

```
tsc --watch
```

- Démo TypeScript
  - Se rendre sur le site typescriptlang.org
  - Cliquer sur Try it out
  - Jouer avec PlayGround
-

Les types de TypeScript

- Le fait de différencier le type d'une variable
- Dans un langage dit typé, on ne peut enregistrer qu'un type de valeur dans une variable.

```
let variable : Type ;
const unNombre : number = 12;
```

- **boolean**:
  - Valeurs : true, false
  - let isDone : boolean = false;
- **number**:
  - Comme en JS, tous les nombres sont des flottants.
  - Accepte aussi les valeurs : Binaires, Octales, Hexadecimale

```
let decimal: number = 6;
let hex: number = 0xf00d;
let binary: number = 0b1010;
let octal: number = 0o744;
```


Les types de TypeScript

- **string**: Chaîne de caractères

```
let prenom: string = 'Adrien';
let nom: string = "Vossough";
```

- Nous pouvons taper une chaîne sur plusieurs lignes avec le templating.

```
let texte: string = `mon prénom est ${ prenom }
et mon nom est ${ nom }`;
```

- Équivalent à :

```
let texte2: string = "mon prénom est " + prenom + "\n"
+ "et mon nom est " + nom;
```

Les types de TypeScript

- **Array**: Tableau typé
- Le tableau suivant ne peut contenir que des nombres :

```
let tab0: number[] = [ -5, 2, 8 ];

tab0.push(12.56); // OK

tab0.push("Salut !"); // Erreur
// Argument of type 'string' is not assignable to parameter of type 'number'.
```

- Il existe une seconde écriture utilisant des tableaux génériques :

```
let tab1: Array<number> = [ -8, 15.9, 0 ];
```


Les types de TypeScript

- **Tuple**: Tableau avec différents types

```
// déclaration
let user: [string, number];
// initialisation correcte
user = ["Adrien", 35]; // ok
// initialisation incorrecte
user = [35, "Adrien"]; // Erreur
```

- Utilisation

```
console.log( user[0] );
// affiche : Adrien

console.log( user[1] );
// affiche : 35
```

Les types de TypeScript

- **Enum**: Ensemble de nombres constants ayant un nom.

```
enum Fruit {Banane, Kiwi, Orange};
let k: Fruit = Fruit.Kiwi;

console.log(k);
// Affiche : 1
```

- Par défaut, la première valeur vaut 0.
- Il est possible de définir des valeurs manuellement :

```
enum Couleur {Rouge, Vert=-0.2, Jaune=58};
let n: Couleur = Couleur.Jaune;

console.log(n);
// Affiche : 58
```

- Il est possible de récupérer la chaîne de caractère ainsi :

```
let s: string = Couleur[-0.2];

console.log(s);
// Affiche : Vert
```


Les types de TypeScript

- **any**: Accepte n'importe quel type de variable.

```
let v: any = "Salut";
v = 1;
v = true;

let liste: any[] = ["Adrien", false, 123];
liste[0] = 35;
```

- **void**: Accepte que les valeurs null et undefined
  - let n : void = null;
- Généralement utilisé pour indiquer qu'une fonction ne renvoie pas de valeurs.

```
function direCoucou(): void {
    console.log("Coucou !");
}
```

- Il existe d'autres types mais très peu utilisés directement comme null et undefined, car très limités.

Les types de TypeScript

- Cast : Changer une valeur dans un **typeA** vers une valeur de **typeB**

```
let uneValeur: any = "Peut-être une chaîne ?";

let longueur: number = (<string>uneValeur).length;
```

- Le type any pouvant être de n'importe quel type, nous indiquons que uneValeur est de type String.
- Dans ce cas, TypeScript nous fait confiance et autorise l'utilisation des méthodes propre aux objets String.
- Seconde syntaxe :

```
let uneValeur: any = "Peut-être une chaîne ?";

let longueur: number = (uneValeur as string).length;
```


Les fonctions : Typage

- Type des paramètres et des valeurs de retour.

Type du 1er argument — Type du 2ème argument — Type de la valeur retournée

```
function addition(x: number, y: number): number {
    return x + y;
}

let addition2 = function(x: number, y: number): number {
    return x + y;
};
```

Utilisation :

```
let resultat0 = addition("b", 2);  // erreur de type
let resultat1 = addition(2);       // erreur, il manque un paramètre
let resultat2 = addition(2, 3, 8); // erreur, paramètre en trop

let resultat2 = addition(55, 20);  // ok
let resultat2 = addition2(10, 40); // ok
```

Fonctions : paramètre optionnel

- Paramètre pouvant être omis lors de l'appel de la fonction
- En javascript, tu ne passes pas les paramètres à l'appel de la fonction, leur valeur sera undefined. Mais en TypeScript, si tu déclares une fonction avec des paramètres typés, le compilateur déclare une erreur si les oublies :

```
function addition(x: number, y?: number): number {
    if(y) {       // nous testons si y est défini
        return x + y;
    }
     return x;
}
```

```
let resultat1 = addition(2);        // ok
let resultat2 = addition(55, 20);   // ok
```


Fonctions : Paramètre par défaut

- Si le paramètre est omis, il recevra une valeur par défaut.
- Le type d'un paramètre ayant une valeur par défaut peut être omis
- La valeur par défaut définit le type du paramètre

```
function addition(x: number, y = 12): number {
    return x + y;
}

function addition2(x = 8, y: number): number {
    return x + y;
}
```

```
let resultat1 = addition(2);        // ok
let resultat2 = addition(55, 20);   // ok

console.log( resultat1, resultat2);

addition2(5, 2);         // ok
addition2(3);            // erreur, paramètre manquant
addition2(undefined, 5); // ok
```

```
14 75
```

Rappel de la POO : Classes

- Classe : Schéma à suivre pour la création d'une classe

```
class Bonjour {
    message: string;

    constructor(message: string) {
        this.message = message;
    }
    affiche() {
        console.log( "Bonjour " + this.message );
    }
}

let bjr = new Bonjour("Adrien");

bjr.affiche()
```


Rappel de la POO : l'héritage

- Héritage : Permet de créer une classe "parent" qui donnera toutes ses caractéristiques à ses enfants
- L'héritage multiple n'existe pas en TypeScript

Classe mère : Animale

```
class Animale {
    nom: string;
    constructor(leNom: string) {
        this.nom = leNom;
    }
    deplacer(distance: number = 0) {
        console.log(`${this.nom} bouge de ${distance}m.`);
    }
}
```

Classe enfant : Chien

```
class Chien extends Animale {
    constructor(name: string) {
        super(name);
    }
}
```

Chien de par son parent :
- A un nom
- Peut se déplacer

Utilisation :

```
let monChien = new
Chien("Milou");
monChien.deplacer(5);
```

Rappel de la POO : L'héritage

- Si nous considérons que l'héritage est reprendre l'ADN du parent, nous pouvons considérer que l'enfant est une évolution.
- Nous pouvons donc lui ajouter de nouveaux attributs et fonctions que le parent n'a pas.

Classe enfant : Chien

```
class Chien extends Animale {
    couleurPoil: string;

    constructor(name: string) {
        super(name);
    }

    manger() {
        console.log("Je mange");
    }
```

Utilisation :

```
let monChien = new Chien("milou");
monChien.deplacer(5);
monChien.manger();
```


Rappel de la POO : L'héritage

- super : Indique que nous utilisons une fonction crée dans le parent

Classe enfant : Chien

```
class Chien extends Animale {
    constructor(name: string) {
        super(name);
    }
}
```

- Dans ce cas, super(), indique que nous appelons la fonction **constructor** du parent.
-

Les interfaces

- TypeScript permet l'utilisation des interfaces pour améliorer le typage dynamique des objets.
  - Permet d'utiliser dans une fonction des objets de types différents mais qui ont un comportement commun
  - Se base sur une série d'attributs et de méthodes attendus.
  - L'objet est considéré valide quel que soit sa classe s'il respecte les attributs et les méthodes attendus.

Sans interface:

```
function test(obj) {
    console.log( obj.x + 3 ); // additionne l'attribute x avec 3
}

let o1 = {};
let o2 = { x: "coucou" };
let o3 = { x: 10 };

test(o1);  // affiche: NaN car undefined + 3
test(o2);  // affiche: coucou3
test(o3);  // affiche: 13
```


Les interfaces

Avec interface: 1ère écriture

```
function test( obj: {x: number} ) {
    console.log( obj.x + 3 ); // additionne l'attribute x avec 3
}
```

Avec interface: 2ème écriture

```
interface xNumber {
    x: number;
}

function test( obj: xNumber) {
    console.log( obj.x + 3 ); // additionne l'attribute x avec 3
}
```

```
let o1 = {};
let o2 = { x: "coucou" };
let o3 = { x: 10 };

test(o1);  // erreur car pas d'attribute x
test(o2);  // erreur car x n'est pas de type number
test(o3);  // affiche: 13
```

Nous avons bien filtré les objets, seuls ceux ayant un comportement attendu sont acceptés

Les modules

- Un module est une façon de découper du code pour le réutiliser facilement
- S'apparente à un import de package en Java ou un include en PHP

module1.js

```
export function test(message) {
    console.log(message);
}
export function truc() {
    console.log("Je ne fais rien");
}
```

module2.js

```
import {test, truc } from './module1.js';

test("coucou");
truc();
```

module3.js

```
import {*} from './module1.js';

test("coucou");
truc();
```


Les décorateurs

- C'est une façon de faire de la méta-programmation. Une sorte d'annotation, ne servant pas réellement au langage lui-même mais plutôt aux frameworks et au librairies.
- Les décorateurs attachent dynamiquement des responsabilités supplémentaires à un objet. Il fournit une alternative souple à l'héritage, pour étendre des fonctionnalités..
- En Angular, on utilisera les annotations fournies par le framework. Leur rôle est assez simple: ils ajoutent des métadonnées à nos classes, propriétés ou paramètres pour par exemple indiquer "cette classe est un composant", "cette propriété est un champ input d'un composant"
- En TypeScript, les annotations sont préfixées par @, et peuvent être appliquées sur une classe, une propriété de classe, une fonction, ou un paramètre de fonction. Pas sur un constructeur en revanche, mais sur ses paramètres oui.

Les décorateurs

- Comment reconnaitre un décorateur ?
- on trouve des décorateurs sur différents objets comme les classes, propriétés ou méthodes.
- Par exemple, sur une classe :
  - @Component
  - @Directives
  - @Injectable()
- Sur un attribut
  - @Input()
  - @Output()

Atelier

- Définir une interface User ayant comme propriétés
  - firstname, lastname, email, age et liste de rôles.
- L'âge doit être marqué comme propriété optionnelle.
- Dans la classe du composant, créer un user et l'afficher sur la page (sans ses rôles).

Angular: Notions de base

Angular CLI

- Angular CLI est une interface de ligne de commande que vous pouvez utiliser lors de la création d'applications avec Angular.
- À partir de cet outil ligne de commande :
  - vous pouvez créer des projets
  - ajouter des fichiers à des projets existants
  - tester, déboguer et déployer vos applications Angular.
- le processus de génération d'un modèle générique d'application basé sur les normes de construction d'une application Angular.
- Une hiérarchie de dossiers de base est créée selon les meilleures pratiques de mise en page et de nommage.
- le CLI peut être installer en utilisant l'outil npm

npm install –g @angular/cli

Commandes du CLI

- Pour générer la structure d'un projet Angular, on utilise le CLI via sa commande **ng** suivie de l'option **new** et le nom du projet

  ng new my-app
- Cette commande génère les différents fichiers requis par une application basique Angular et installe aussi toutes les dépendances requises par le projet.
- La commande ng new peut être utilisé avec des options :
  - Annuler la création des fichiers de tests

    ng new my-app **--skip-tests=true**
  - crée des fichiers .scss pour les styles plutôt que des fichiers .css

    ng new --style=scss :

Commandes du CLI

- Pour tester un projet Angular, on exécute la commande **ng serve** à partir de la racine du projet.
- Cette commande compile le code source du projet pour transpiler le code TypeScript en JavaScript et en même temps démarre un serveur web local basé sur NodeJs pour déployer l'application localement.
- Pour tester le projet, il suffit de lancer le navigateur et accéder à l'URL:

http://localhost:4200

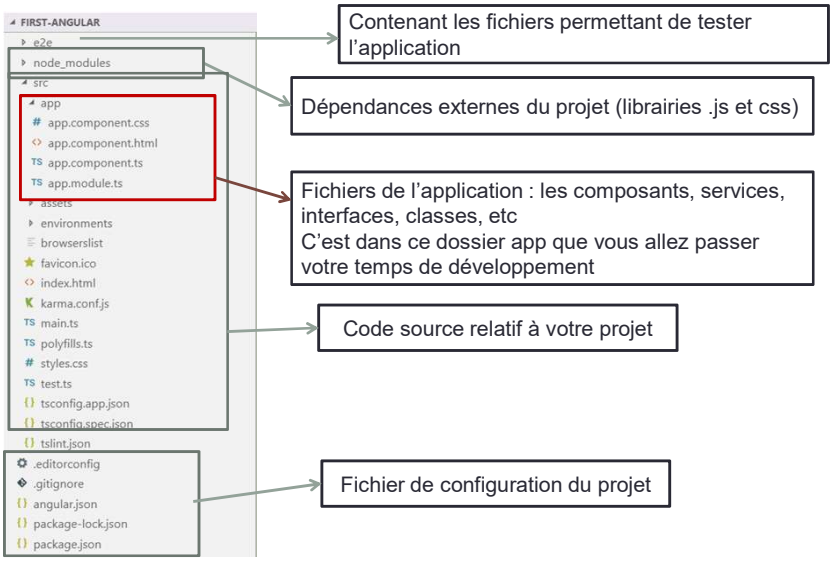
- modifier le port par défaut avec l'option --port

ng serve --port 5000

Commandes de génération

- le CLI permet aussi de créer facilement des composants de votre application sans avoir à vous soucier d'inclure correctement le code dans votre application.
- Le CLI génère les fichiers au format approprié et les modifie pour inclure correctement le composant nouvellement généré.
- Parmi les différents générateurs pouvant être exécutés par la CLI, citons:
  - Modules : **ng generate module** my-module
  - composants : **ng generate component** my-component
  - Interfaces : **ng generate interface** my-interface
  - Services : **ng generate service** my-service
  - Classes : **ng generate class** my-class
  - Guards ng **generate guard** my-guard
- Le CLI propose aussi d'autres commandes utiles :
  - Pour déployer l'application : **ng build** --target=production --base-href /
  - Pour exécuter les tests unitaires avec le framework jasmine: **ng test**
  - Pour exécuter des tests de bout en bout avec Protactor : **ng e2e**

Structure d'un projet Angular



Structure d'un projet Angular

Index.html

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

Structure d'un projet Angular

main.ts

```
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';
import { environment } from './environments/environment';

if (environment.production) {
  enableProdMode();
}

platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.error(err));
```

Structure d'un projet Angular

Puisque nous construisons une application pour le navigateur, le module racine devra importer BrowserModule.

app.module.ts

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

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

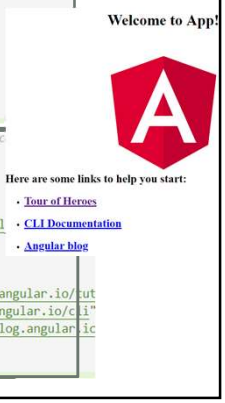
Structure d'un projet Angular

app.component.ts

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent{
    title : string = 'App';
```

app.component.html

```
<!--The content below is only a placeholder and can be replac
<div style="text-align:center">
  <h1>
    Welcome to {{ title }}!
  </h1>
  <img width="300" alt="Angular Logo" src="data:image/svg+xml
</div>
<h2>Here are some links to help you start: </h2>
<ul>
  <li>  <h2><a target="_blank" rel="noopener" href="https://angular.io/
  <li> <h2><a target="_blank" rel="noopener" href="https://angular.io/c
  <li> <h2><a target="_blank" rel="noopener" href="https://blog.angular.
</ul>
```



Structure d'un projet Angular

app.component.css

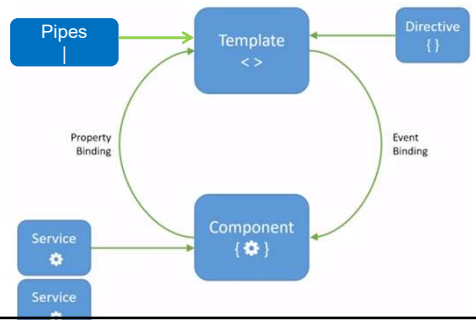
Ce fichier est généré vide, il contient la logique de présentation des données (règles css)

app.component.spec.ts

Les fichiers .spec sont des tests unitaires pour vos fichiers sources. La convention pour les applications Angular 2 est d'avoir un fichier .spec pour chaque fichier .ts. Ils sont exécutés à l'aide du Framework JavaScript de tests unitaires Jasmine via le programme de tâches Karma lorseque vous utiliser la commande **ng test**

Architecture d'Angular

- Une application Angular se compose de :
  - Un à plusieurs modules dont un est principal
  - Chaque module peut inclure :
  - Des composants web : la partie visible de l'application
  - Des services pour la logique applicative. Les composants peuvent utiliser les services via le principe de l'injection de dépendances.
  - Les directives : un composant peut utiliser des directives
  - Les pipes : utilisées pour formater
  - l'affichage des données



Modules

- Un module Angular, est un fichier de classe qui vous aide à organiser toutes les «pièces» de votre application. il vous permet d'organiser ces pièces en blocs.
- Les modules vous permettent également d'étendre les capacités de votre application via des bibliothèques externes.
- Les deux instructions d'importation comportaient deux modules Angular angular/core et le module angular/common

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

Vous pouvez également exporter votre module pour l'utiliser dans d'autres applications.

```
export class FirstModule { }
```

- Grâce aux modules Angular, vous pouvez diviser votre application en plusieurs composants, ce qui facilite sa mise à jour et sa maintenance.

Modules

- Chaque application angulaire comporte au moins un module appelé module racine.
- Le module racine est démarré pour charger l'application. Par convention, ce module racine s'appelle AppModule.
- @NgModule est un décorateur qui prend en paramètre un objet JavaScript qui contient les métadonnées dont les propriétés décrivent le module.
-

```
@NgModule({
declarations: [
    AppComponent,
    HomePageComponent,
    NotFoundComponent
],
imports: [
    HttpClientModule,
    FormsModule,
],
providers: [GitSearchService],
bootstrap: [AppComponent]
})
export class AppModule { }
```

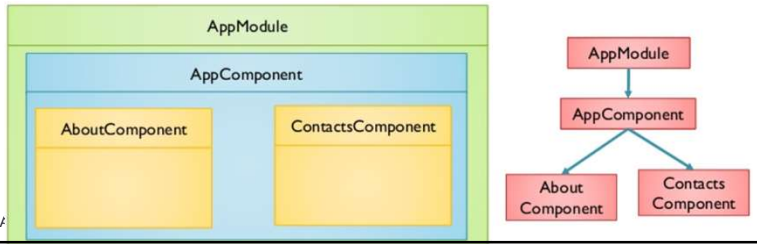
Syntaxe d'un module

- Les modules Angular consistent en un ensemble discret d'attributs que vous utilisez pour décrire le module.
- déclarations : il s'agit de la liste des composants faisant partie du module spécifique que vous créez. Tout ce qui se trouve dans une déclaration est visible pour les applications utilisant ce module, sans aucune exportation explicite.
- imports: à l'aide des instructions d'importation, vous pouvez rendre d'autres modules visibles dans ce module. L'importation d'autres modules ne constitue pas une exportation implicite de ces autres modules. En d'autres termes, si un autre développeur importe votre module, il se peut qu'il ne puisse pas voir les modules que vous avez importés.
- exports: il s'agit de la liste ou de toutes les déclarations que vous souhaitez rendre disponibles (visibles) aux autres modules susceptibles d'importer ce module.
- providers : utilisé pour lister les fournisseurs pour la configuration de l'injecteur, lorsque ce module est importé par d'autres modules.
- bootstrap : Principalement utilisé pour le chargement dynamique de composants.

Composants

Composants

- Un composant est utilisé par contrôler une vue ou une partie de l'écran.
- L'une des caractéristiques la plus utile de l'architecture d'Angular est l'utilisation de composants réutilisables pour l'affichage des données.
- En utilisant un composant, vous pouvez coupler la logique d'affichage des données, les styles et le HTML dans un conteneur facilement portable dans toute votre application.
- Le code que vous allez ajouter à ce code HTML est responsable de la définition des liaisons aux données et des directives au sein du code HTML fournissant les aspects dynamiques d'un composant Angular.



Génération d'un composant avec le CLI

- Pour créer un composant il faut taper la commande :

> ng generate component component-name : où component-name est le nom du composant à créer

- Exemple : ng generate component git-search
- le générateur va créé un dossier /src/app/git-search, ainsi qu'un fichier .css, .ts, .html et .spec.ts pour le composant git-search. Ces fichiers ont chacun un objectif.
  - Le fichier .ts : contient la logique TypeScript du composant et est l'endroit où vous placeriez les méthodes, l'intégration de service et toute autre logique.
  - Le fichier .css : est automatiquement injecté dans votre composant. On peut écrire automatiquement des feuilles de code CSS spécifiquement adaptées au composant, sans avoir à vous soucier de la réutilisation des classes.
  - Le fichier .html : est l'endroit où il faut écrire le code de la vue (template).
  - Le fichier .spec.ts : est l'endroit où on peut écrire des tests Karma pour le composant.
  - Les tests seront traités dans la partie Angular avancé.

Composants

- le fichier de classe créé lors de la génération d'un composant contiendra les propriétés et les méthodes qui seront utilisées par l'application.
- On trouve dans ce fichier également des métadonnées utilisées pour fournir des informations supplémentaires sur le composant .
- Les métadonnées sont définies à l'aide de décorateurs.

```
import { Component, OnInit} from '@angular/core';
@Component({
   selector: 'app-home-page',
   templateUrl: './home-page.component.html',
   styleUrls: ['./home-page.component.css']
})
export class HomePageComponent implements OnInit {

constructor() { }
```


Anatomie d'un composant Angular

- Les instructions d'importation sont utilisées pour accéder aux fonctionnalités d'autres codes tels que les bibliothèques et autres classes.

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

- Cette ligne est responsable de l'importation de l'objet composant à partir de la bibliothèque @ angular / core. C'est l'une des principales bibliothèques utilisées dans Angular.

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

- C'est le segment où vous définissez votre composant. Le @ est un symbole TypeScript qui nous aide à décrire un décorateur Angular.
- Dans ce cas, @Component est un décorateur qui indique à Angular que nous souhaitons créer un nouveau composant.
- Un décorateur en Angular est précédée du symbole @ et est considérée comme un mécanisme permettant d'ajouter des métadonnées à une classe, aux membres d'une classe ou aux arguments de méthode d'une classe.

Anatomie d'un composant Angular

- le décorateur @Component contient un objet avec des métadonnées et des informations pour ce composant.
  - **selector** : spécifie le nom d'une balise HTML qui contiendra ce composant lorsque l'application est en cours d'exécution. Dans le code HTML, vous trouverez les balises <app-home-page> </ app-home-page> pour ce composant.
  - **templateUrl** : l'utilisation de cette option vous permet de spécifier un modèle qui contiendra le code HTML pour ce composant. Vous pouvez également choisir d'utiliser l'option template: et spécifier le droit HTML "en ligne" dans le corps du composant.
  - **Template** : permet de définir à l'intérieur du décorateur le code HTML représentant la vue du composant.
  - **styleUrls** : renvoie à une feuille de style CSS externe qui sera appliquée au rendu HTML du composant.

Anatomie d'un composant Angular

- La déclaration d'exportation

```
export class HomePageComponent implements OnInit
```

- Le mot-clé export permet à votre composant d'être disponible pour être importé dans d'autres fichiers du projet.
- Le mot-clé class identifie ce fichier en tant que fichier de classe.
- L'identifiant HomePageComponent est le nom de la classe. Ce sera le nom que vous utiliserez dans une instruction import dans les autres fichier de votre application et vous fournira un identificateur clair pour cette classe.
- Le dernier mot-clé, implements, ne sera pas toujours présent sur tous les composants. Dans ce cas, cela indique que notre classe HomePageComponent implémentera également des fonctionnalités spécifiées dans un autre interface, appelé OnInit.

Atelier : mon premier composant

- Créer une classe Type script “FilmItem” définit par le titre, l’url de l’image du film, la description, l’année se sortie et l’image d’un film on peut ajouter aussi un score.
- Créer un composant “list-Film” qui affiche plusieurs films dans une page.

Templates

- La vue est la partie visuelle du composant. En utilisant l’attribut template du décorateur @Component, nous déclarons le modèle HTML que le composant utilisera:
- Deux solutions sont possibles:
  - Template inline :

```
@Component({
        selector: 'inventory-app-root',
         template: `
                <div class="inventory-app">
                </div> ` })
```

  - Dans cette exemple ci-dessus, notez que nous utilisons la syntaxe de la chaîne multiligne de TypeScript.
- Nous pouvons également déplacer notre template vers un fichier séparé et utiliser l’attribut templateUrl pour le référencer:

```
@Component({
        selector: 'inventory-app-root',
         templateUrl: './app.component.html'
})
```

Templates : bootstrap

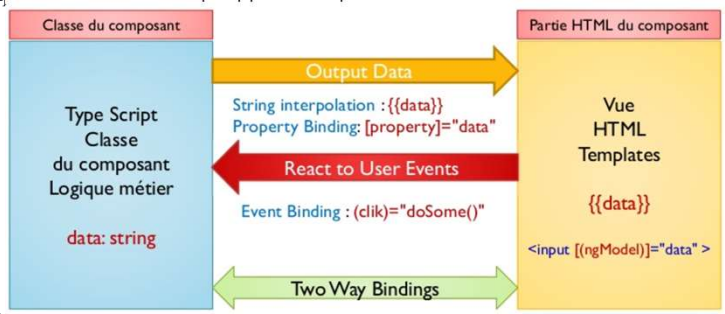
- On peut utiliser les classes de bootstrap dans les templates en utilisant le CLI
- **Installation** : npm install bootstrap --save
- Vérifier qu'un répertoire bootstrap a bien été crée dans nodes_modules.
- **Intégration dans le projet**
  - Ouvrir le fichier .angular.json (pour Angular 6 ou supérieur) situé à la racine de votre projet
- Dans la rubrique styles, ajouter le chemin relatif vers le fichier bootstrap.css (le chemin est "./node modules/bootstrap/dist/css/bootstrap.css" ,
- **Utilisation** : on peut utiliser maintenant bootstrap dans les component.html

```
<button class="btn btn-success"
(click)="onAllumer()">Tout allumer</button>
<button class="btn btn-danger"
(click)="onEteint()">Tout éteindre</button>
<ul class="list-group">
    <li class="list-group-item" *ngFor="let element of ensemble">
        {{ element }}
    </li>
</ul>
```


Databinding

Liaison de données (Databinding)

- La liaison est le terme utilisé pour décrire comment les données d'une application sont «liées» à l'interface utilisateur.
- La liaison est donc responsable de la communication entre nos données et notre interface utilisateur.
- La liaison nous permet de transmettre les données à l'interface utilisateur via l'utilisation de propriétés, dans la classe qui définit notre modèle, et l'utilisation de directives, qui utilisent un concept appelé interpolation.



Databinding

- Il existe quatre types de liaison de données que nous pouvons effectuer dans Angular.
  - Liaison d'interpolation :
  - liaison de propriété
  - liaison d'événement
  - liaison bidirectionnelle.
- **L'interpolation** est réalisée à l'aide de la syntaxe d'interpolation, également appelée moustache, où nous plaçons la propriété de données dans le code HTML à l'aide de la double accolade, {{}}.

Exemple : <h1> {{titre}} </ h1>

Databinding - Interpolation

- La classe UserComponent a une propriété, *firstName*. Et le template a été enrichi avec une balise <h1>, utilisant la fameuse notation avec double-accolades(les "moustaches") pour indiquer que cette expression doit être évaluée.
- Ce type de databinding est **l'interpolation**.

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

@Component({
selector: '[app-user]',
template: `<h1> User : {{firstName}} </h1>.`
})
export class UserComponent {
   firstName = 'Adil"
}
```


Databinding - Interpolation

Il faut cependant se rappeler une chose importante : si on essaye d'afficher une variable qui n'existe pas, au lieu d'afficher undefined, Angular affichera une chaîne vide. Et de même pour une variable null.

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

@Component({
selector: '[app-user]',
template: `<h1> User : {{firstName}} </h1>.`
})
export class UserComponent {
}
```

Databinding - Interpolation

- Que se passe-t-il si mon objet *user* est en fait récupéré depuis le serveur, et donc indéfini dans mon composant au début ? Que pouvons-nous faire pour éviter les erreurs quand le template est compilé ?.

```
@Component({
  selector: 'user-app',
  // user is undefined
  // but the ?. will avoid the error
  template: `
  <h1> User </h1>
  <h2> Welcome {{user?.name}} </h2>`
 })
export class UserComponent {
  user: any;
}
```


Databinding – binding de propriété

- En Angular, on peut écrire dans toutes les propriétés du DOM via des attributs spéciaux sur les éléments HTML, entourés de crochets [].

```
<p> {{user.name}} </p>
<p [textContent]="user.name"></p>
```

- La syntaxe à base de crochets permet de modifier la propriété textContent du DOM, et nous lui donnons la valeur user.name qui sera évaluée dans le contexte du composant courant, comme pour l'interpolation.
- Note que l'analyseur est sensible à la casse, ce qui veut dire que la propriété doit être écrite avec la bonne casse.

Databinding - binding de propriété

- Bien sûr, une expression peut aussi contenir un appel de fonction

```
<img src="http://localhost/images/{{user.getId()}}" />

<img [src]=" 'http://localhost/images/' + user.getId() " />
```


Databinding - Événements

- Les **événements** sont un mécanisme permettant de réagir aux interactions de l'utilisateur.
- La liaison d'événement : est réalisée lorsque nous associons un événement à une méthode de notre classe.
- Dans le code ci-dessous, nous entourons l'attribut d'événement click entre parenthèses de l'élément button et fournissons le nom de la méthode dans notre classe, qui gérera l'événement.

```
@Component({
 selector: 'app-users',
 template: `<h2>Users</h2>
   <button (click)="refreshUsers()">
            Refresh the users list
    </button>
   <p>{{users.length}} users</p>`
})
export class UsersComponent {
 users: any = [];
 refreshUsers() {
   this.users = [{ name: Rabat }, { name: 'Fès' }];
 }
```

Databinding - Expressions vs instructions

- Dans le premier cas de binding de propriété, la valeur *getSomething()* est une **expression**, et sera évaluée à chaque cycle de détection de changement pour déterminer si la propriété doit être modifiée.
- Dans le second cas de binding d'événement, la valeur *doSomething*() est une **instruction** (statement), et ne sera évaluée que lorsque l'événement est déclenché.

```
<!--Expressions évalués par Angular à chaque cycle -->
<component property="{{user.name}}"></component>
<component property="{{getSomething()}}"></component>

<component [property]="user.name"></component>
<component [property]="getSomething()"></component>

<!-- Instruction exécutée sur réception d'un event -->
<component (event)="doSomething()"></component>
```


Databinding – variables locales

- En analysant le template, Angular va regarder dans l'instance du composant pour trouver une variable et dans les variables locales.
- Les variables locales sont des variables qu'on peut déclarer dynamiquement dans le template avec la notation #

```
<input type="text" #name>
{{ name.value }}

<input type="text" #name>
<button (click)="name.focus()">
  Focus the input
</button>
```

Databinding - two-way databinding

- La directive ***NgModel*** met à jour la valeur de l'input à chaque changement du modèle lié user.username, d'où la partie **[ngModel]="user.name"**.
- La directive génère un événement depuis un output nommé **ngModelChange** à chaque fois que l'input est modifié par l'utilisateur, où l'événement est la nouvelle valeur, d'où la partie **(ngModelChange)="user.name = $event",** qui met donc à jour le modèle user.username avec la valeur saisie.

```
<input type="text" [(ngModel)]="user.name" />
{{ user.name }}
<input type="text"
    [ngModel]="user.name"
    (ngModelChange)="user.name = $event" />
```


Databinding - Utilisation de ngModel

- Afin de pouvoir utiliser la directive ***ngModel***, il faudrait ajouter le module ***FormsModule*** aux tableau imports[] défini dans le root module.
- Nous aborderons plus en détail le module FormsModule sur la partie des formulaires

```
import { FormsModule } from '@angular/forms';
import { AppComponent } from './app.component';
@NgModule({
declarations: [ AppComponent ],
imports: [
  BrowserModule,
  FormsModule
],
 bootstrap: [AppComponent]
})
export class AppModule { }
```

Databinding

- En conclusion
  - Le système de template d'Angular nous propose une syntaxe puissante pour exprimer les parties dynamiques de notre HTML. Elle nous permet d'exprimer du binding de données, de propriétés, d'événements, et des considérations de templating, d'une façon claire, avec des symboles propres :
    - {{}} pour l'interpolation,
    - [] pour le binding de propriété,
    - () pour le binding d'événement,
    - # pour la déclaration de variable locale

Propriétés et événements

Interaction entre composants

- Une application Angular est composée de plusieurs composants.
- En utilisant des formulaires et des liens, on peut envoyer des données d'un composant à un autre,
- On peut ajouter le sélecteur d'un premier composant dans la template d'un deuxième composant
  - on appelle le premier composant : composant fils
  - on appelle le deuxième composant : composant parent
- En utilisant les décorateurs @Input() et @Output() les deux composants peuvent échanger de données.
- @Input() : permet à un composant fils de récupérer des données de son composant parent,
- @Output() : permet à un composant parent de récupérer des données de son composant enfant,

Composants - Inputs

- Les entrées sont parfaites pour passer des données d'un élément supérieur à un élément inférieur.
- Par exemple, si on veut avoir un composant affichant une liste d'utilisateurs, il est probable qu'on ait un composant supérieur contenant la liste, et un autre composant inférieur affichant un utilisateur
- Le décorateur ***@Input()*** permet de déclarer une **propriété** comme une **entrée** d'un composant.
- Les entrées spécifient les paramètres que nous attendons de notre composant.
- Lorsque nous spécifions qu'un composant prend une entrée, il est attendu que la classe de définition aura une variable d'instance.

Interaction entre composants

```
@Component({
 selector: 'app-users',
 template: `<app-user [user]="selectedUser">
            </app-user>`
})
export class UsersComponent {
 selectedUser: User = { id: 1, name: 'Salman' });
}
```

```
@Component({
 selector: 'app-user',
 template: `<div>{{user.name}}</div
})
export class UserComponent {
 @Input() user: User;
}
```


Les entrées d'un composant - Alias

- Si on souhaite exposer la propriété par un terme différent de celui de la propriété, il faudrait ajouter l'alias en paramètre de l'appel du décorateur.

@Input('userInput') user: User;

```
@Component({
 selector: 'app-user',
 template: `<div>{{user.name}}</div>`
})
export class UserComponent {
 @Input('user') internalUser: User;
}
@Component({
 selector: 'app-users',
 template: `<app-user [user]="selectedUser">
            </app-user> `
})
export class UsersComponent {
 selectedUser: User = { id: 1, name: 'Salman' }
}
```

Composants – Événements

- **En Angular, les données entrent dans un composant via des propriétés, et en sortent via des événements.**
- Les événements sont un mécanisme permettant à un composant de notifier son parent et au parent de gérer la notification
- Les événements spécifiques sont émis grâce à un *EventEmitter*, et doivent être déclarés dans le décorateur, via l'attribut outputs. Comme l'attribut inputs, il accepte un tableau contenant la liste des événements que le composant peut déclencher.
- Et, comme pour les inputs, on préfère généralement le décorateur @Output()

Composants – Événements

- Si on veut émettre un événement appelé userSelected. Il y aura trois étapes à réaliser :
  1. déclarer l'événement en l'annotant du décorateur @Output()
  2. créer un *EventEmitter*
  3. émettre un événement quand le user est sélectionné avec la méthode emit

```
import { Component, Output } from '@angular/core';
@Component({
 selector: 'app-users',
 template: `
   <app-user (userSelected)="modifyUser($event)">
   </app-user>`
})
export class UsersComponent {
 modifyUser($event) {
    console.log('Selected user\'s email'+ $event.email);
 }
}
```

```
import { Component, Output } from '@angular/core';
@Component({
 selector: 'app-user',
 template: `
    <button (click)="clickedUser()">
      {{user.name}}
    </button>`
})
export class SelectableUserComponent {
 @Output() userSelected = new EventEmitter<User>();
 clickedUser() {
   this.userSelected.emit(this.user);
 }
}
```

Atelier : Binding de propriétés et Événements

- Refactorer l'application de gestion des films dans une approche one-way data flow:
- Les formulaires et listings doivent être mis dans des composants spécifiques
- Les composants d'alerte doivent être paramétrables par le message à afficher.

Directives

Directives

- Les directives sont chargées de fournir des fonctionnalités dans une application Angular et aident à transformer le DOM.
- Les directives sont essentiellement des commandes envoyées à Angular pour exécuter des traitements spécifiques sur des affichages de données ou de templates.
- Il existe deux types de directives
  - directives structurelles
  - directives d'attribut.
- Les directives structurelles vous permettent de modifier la mise en page de la page en manipulant le DOM.
- Une directive d'attribut modifiera simplement le comportement ou l'apparence d'un élément DOM existant.

Directives -- ngIf

- les directives structurelles manipulent la présentation HTML via le DOM. La manipulation consiste à ajouter, supprimer ou modifier un élément.
- Les deux directives structurelles les plus couramment utilisées sont ngIf et ngFor.
- La directive * ngIf est une directive intégrée fournie avec Angular
- Cette directive évaluera une expression booléenne et, en fonction du résultat, affichera ou masquera un élément HTML.

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-users',
  template: `
       <div *ngIf="users.length">
          <h2>Users</h2>
       </div>`
})
export class UsersComponent {
  users: Array<any> = [];
```

Directives - *ngIf else

- Au lieu de mettre une condition sur un template et son contraire pour afficher deux blocs dont un uniquement doit être affiché, la syntaxe est dorénavant plus simplifié,

Angular 2

```
<div *ngIf="condition">Mon Bloc OK</div>
<div *ngIf="!condition">Mon Autre Bloc</div>
```

Angular 4

```
<div *ngIf="condition; else elseBlock">
  Mon Bloc OK
</div>
<ng-template #elseBlock>
  Mon Autre Bloc
<ng-/template>
```


Directives -- *ngFor

- **ngFor** : utiliser pour afficher des données d'unne façon ittérative
- Exemple : afficher les éléments d'un tableau

```
@Component({
  selector: 'app-users',
  template: `
    <ul>
     <li *ngFor="let user of users">{{user.name}}</li>
    </ul>`
})
export class UsersComponent {
  users = [
      {name: 'ali'}, {name: adil}
   ];
}
```

Directives -- *ngFor

- Et pour avoir l'indice de l'itération courante

```
<app-appareil *ngFor="let appareil of appareils ; let i = index"

        [appareilName]="appareil.name"

        [appareilStatus]="appareil.status"
        [indexOfAppareil]= "i"
        [id]="appareil.id"></app-appareil>
```


Directives d'attributs -- ngStyle

- Si on veut changer plusieurs styles en même temps, nous pouvons utiliser la directive ngStyle :
- Note que la directive attend un objet dont les clés sont les styles à définir.

```
<h4 [ngStyle]="{color: getColor()}"> {{ product.name }}</h4>
```

Directives d'attributs -- ngClass

- La directive ngClass est conçue pour vous permettre d'appliquer dynamiquement CSS à un élément. La directive utilise une expression pour évaluer le CSS à appliquer.
- la directive ngClass permet d'ajouter ou d'enlever dynamiquement des classes sur un élément. Comme pour le style, on peut soit définir une seule classe avec le binding de propriété :

```
<li [ngClass]="{'list-group-item': true,
                'list-group-item-success': compteur > 0,
                'list-group-item-danger': compteur < 0,
                'list-group-item': compteur === 0}">
```


Atelier Binding et Directives

- Créer une mini application affichant une liste de films initialement stockés dans un tableau dans le composant list-film.
- Chaque film doit être passé comme paramètre au composant fils film-item,
- Afficher un message bon film en vert s'il le film est bien classé ou mauvais en rouge dans le cas contraire.
- En termes de structure :
- votre FilmListComponent contiendra l'array des films
- votre FilmListComponent affichera un FilmItemComponent pour chaque film dans l'array
- chaque FilmItemComponent affichera le titre, le score, l'année, et l'image du film dans le template, un lien pour la description.
- Chaque FilmItemComponent affichera un bouton (like) qui permet de d'ajouter le film dans les favoris.

L'injection de dépendances

L'injection de dépendance

- La documentation sur Angular définit l'injection de dépendance (DI) comme «un moyen pour fournir à une nouvelle instance d'une classe, les dépendances complètes qu'elle nécessite ».
- L'injection de dépendances est un **design pattern** bien connu. Un composant peut avoir besoin de faire appel à des fonctionnalités qui sont définies dans d'autres parties de l'application (un service, par exemple).
- C'est ce que l'on appelle une dépendance : **le composant dépend du service**. Au lieu de laisser au composant la charge de créer une instance du service, **l'idée est que le framework crée l'instance du service lui-même**, et **la fournisse au composant** qui en a besoin.
- C'est un concept largement utilisé côté serveur, mais AngularJS 1.x était un des premiers à l'apporter côté client.

L'injection de dépendance : Principe de fonctionnement

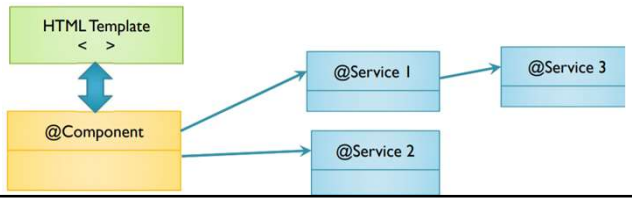
- Lorsque Angular crée un composant, il demande d'abord à un injecteur les services requis.
- Un injecteur maintient un conteneur d'instances de service qu'il a créé précédemment.
- Si une instance de service demandée n'est pas dans le conteneur,
- l'injecteur en fait une et l'ajoute au conteneur avant de renvoyer le service à Angular.
- Lorsque tous les services demandés ont été résolus et retournés, Angular peut appeler le constructeur du composant avec ces services comme arguments.
- Il s'agit d'une injection de dépendance

Injection de dépendances - Angular

- Pour faire de l'injection de dépendances, on a besoin :
  - d'une façon d'enregistrer une dépendance, pour la rendre disponible à l'injection dans d'autres composants/services.
  - d'une façon de déclarer quelles dépendances sont requises dans chaque composant/service.
- Le framework se chargera ensuite du reste. Quand on déclarera une dépendance dans un composant, il regardera dans le registre s'il la trouve, récupérera l'instance existante ou en créera une, et réalisera enfin l'injection dans notre composant.

Services

- Les services sont des classes spécifiques dans Angular qui sont conçus pour créer une logique réutilisable pouvant être injectée dans plusieurs composants.
- Un service est généralement une classe avec un but étroit et bien défini.
- Généralement, les composants se limite à l'affichage et à la gestion des événements utilisateurs dans la vue du composant.
- L'exécution des traitements en local ou en back end sont attribués aux services.
- Quand un événement survient dans la vue, le composant fait appel à des fonctions dans les services pour effectuer des traitements et fournir des résultats



Services – Création

- Le pattern DI peut être utilisé dans une application Angular pour créer une classe (appelé un service) et l'injecter dans d'autres composants de votre application.
- le service importe le décorateur @**Injectable**, ce qui permet d'injecter ce service dans d'autres composants.
- Par exemple, dans cette application on créer une classe ProductService qui va faire un traitement et doit partager le résultat avec plusieurs composants,
- Au lieu d'écrire ce code dans votre application plusieurs fois, vous avez décidé de créer une classe simple qui implémente cette logique:
- Pour créer un service il faut utiliser le CLI avec une simple commande:

```
ng generate service product
```

- Exemple

```
import { Injectable } from '@angular/core';
@Injectable()
  export class ProductService {
  constructor() {
  }
```

Services -- Enregistrement

- Pour utiliser un service, il faut préalablement enregistrer un fournisseur de ce service avec l'injecteur.
- Un fournisseur de service est une fabrique qui permet de gérer l'instanciation des services.
- Vous pouvez enregistrer les fournisseurs dans les modules ou dans les composants.
- En général, ajoutez des fournisseurs dans le module racine afin que la même instance d'un service soit disponible partout.
- Pour cela il faut mettre à jour le module **appModule** pour ajouter la classe *GitSearchService*

```
providers: [ProductService, AuthService, AuthGuardService],
```


Services -- Utilisation

- Enfin, vous pouvez utiliser la classe ProductService dans les composants
  - Ajouter une instruction d'importation pour la classe
  - Injecter une instance de classe via le constructeur en ajoutant un paramètre de constructeur correspondant au type de classe

```
export class ProductListComponent implements OnInit {
  products : Array<Product>;
  selectedProducts : Array<Product>;
  constructor(private productService : ProductService) {
  }

  ngOnInit() {
    this.products = this.productService.products;
  }
```

Atelier Services

- Créer un service FilmService exposant les méthodes suivantes:
  - add(film: Film): void
  - get(id: number): Film
  - getAll(): Array<Film>
- Refactorer l'application en centralisant la gestion du tableau des films dans le service
- Ajouter un composant FilmDetailComponent pour afficher les informations détaillées d'un Film à partir de la liste des films.

Le routage

Routage

- Angular utilise les composants pour gérer les fonctionnalités de routage. Les composants représentent les différentes «pages» auxquelles le routeur dirigera les utilisateurs.
- Pour rajouter la fonctionnalité de routage, vous devez suivre les étapes suivantes :

1. Importer le module ***RouterModule*** dans le module racine de l'application :

   ```
   import { RouterModule, Routes } from '@angular/router';
   ```

2. déclarez une variable constante avec un type de routes

   ```
   const appRoutes: Routes
   ```


Routage

3. initialiser la variable, qui est un tableau d'objets avec deux paramètres obligatoires: **path** et **composant**.

```
const appRoutes : Routes = [
  { path : 'produits', component : ProductListComponent},
  { path : 'auth', component : AuthComponent},
  {path : 'produits/:id', component : EditProductComponent},
  {path : 'panier', component : PanierComponent},
  {path : '', component : ProductListComponent},
  {path : 'admin', component : AdminComponent},
  {path  : 'not-found', component : FourOhFourComponent},
  {path : '**', redirectTo : '/not-found'}
]
```

- la route ** au bas de cette déclaration. Il s'agit de la route générique qui peut être utilisée pour afficher un NotFoundComponent - ou une page 404 - pour les routes qui ne correspondent pas
- Il est important de noter que l'ordre de déclaration des routes est très important.

Routage

4. vous ajouterez la déclaration RouterModule au tableau des importations sous AppModule pour que les routes que nous venons de configurer soient disponibles pour l'application.

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

5. modifier le fichier app.component.html pour pouvoir activer le routage. en rajoutant la balise **<router-outlet> </ router-outlet>.**

- Le tag router-outlet injectera le composant configuré pour la route présentée.

Routage -- navigation

- Problème : Pour accéder à un composant, l'utilisateur doit connaître son chemin défini dans le tableau de routes (or ceci n'est pas vraiment très pratique)
- Solution : On peut plutôt définir un menu contenant des liens vers nos différents composants
- Ici, on utilise l'attribut *routerLink* à la place de *href* pour faire référence à un *path* défini dans notre tableau de routes.

```
<ul>
<li><a routerLink=''>Accueil</a></li>
<li><a routerLink='/about'> About</a></li>
<li><a routerLink='/contact'>Contact</a></li>
</ul>
<router-outlet></router-outlet>
```

Routage – navigation dans le code

- Il est également possible de naviguer entre les différentes routes en utilisant la méthode navigate() du service Router. Pour cela le service Router doit être injecté dans la classe composant.
- L'exemple suivant montre le code de la classe du composant AppComponent : app.component.ts

```
<button class ="btn btn-success" *ngIf ="!authStatus"
        (click)="onSignIn()">se connecter</button>

onSignIn(){
this.authService.signIn().then(()=> {
  //console.log('connexion résussie');
  this.authStatus = this.authService.iAuth;
  this.router.navigate( commands: ['produits']);
  });
}
```


Routage – Module de routes

- Il est plus utile de créer un module séparé pour configurer les routes au de le faire dans le module principal.
- Il suffit de créer un fichier app/app-routing.module.ts
- Ensuite importer ce module dans le module principal

app.routing.module.ts

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import {RouterModule, Routes} from '@angular/router';
import { AboutComponent } from '../about/about.component';
import { ContactsComponent } from '../contacts/contacts.component'

const appRoutes : Routes= [
  {path : 'about', component : AboutComponent} ,
  {path: 'contacts', component : ContactsComponent}
];
@NgModule({
  declarations: [],
  imports: [
    CommonModule,RouterModule.forRoot(appRoutes)
  ],
  exports :[RouterModule]
})
```

app.module.ts

```
@NgModule({
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
})
export class AppModule { }
```

Routage -- Paramètres de routes

- Pour ajouter un paramètre de route, il suffit de créer la route appropriée à l'aide de path: identifier, puis d'inclure le paramètre, tel que id, dans cette entrée, comme illustré dans l'exemple suivant:

```
const appRoutes : Routes = [
  { path : 'produits', component : ProductListComponent},
  { path : 'auth', component : AuthComponent},
  {path : 'produits/:id', component : EditProductComponent},
```

- La dernière entrée de cette section Routes est produits /: id.
- La partie: id est le paramètre de route et sera remplacée par la valeur transmise à l'URL
- Exemple : http: //localhost:4200/produits/12545

Routage -- Paramètres de routes

- Pour récupérer les paramètre d'une route de la forme : products/:id
  - aller dans le composant concerné (ici, EditProductComponent.)
  - faire une injection de dépendance du service Angular ***ActivatedRoute***
  - utiliser ce service dans la méthode *ngOnInit*() pour récupérer l'objet ***snapshot*** puis récupérer les paramètres

```
constructor(private productService : ProductService,
            private route : ActivatedRoute,
            private panierService : PanierService) { }
ngOnInit() {
  const id : number = this.route.snapshot.params['id'];
  this.product = this.productService.getProductById(+id);
}
```

Routage -- Guards

- Certaines routes de l'application ne devraient pas être accessibles à tous. L'utilisateur est-il authentifié ?
- A-t-il les permissions nécessaires ? Bien sûr, il faut désactiver ou masquer les liens pointant sur ces routes inaccessibles.
- Le backend doit aussi empêcher l'accès aux ressources interdites à l'utilisateur courant. Mais cela n'empêchera pas l'utilisateur d'accéder aux routes qui lui sont interdites, simplement en entrant leur URL dans la barre d'adresse du navigateur.

Routage -- Guards

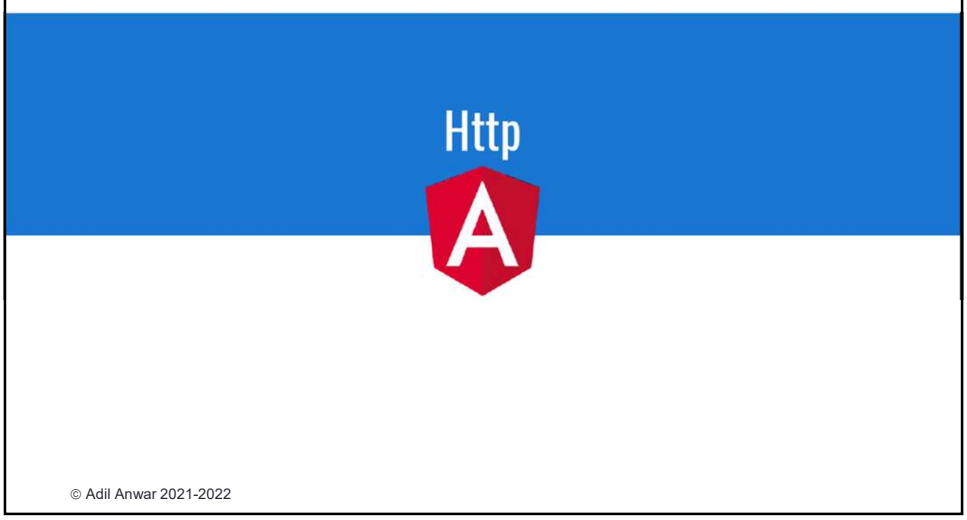
- C'est là que le guard ***CanActivate*** intervient, lorsqu'un tel guard est appliqué à une route, il peut empêcher l'activation de la route

```
export class AuthGuardService implements  CanActivate{
  canActivate(route: ActivatedRouteSnapshot,
              state: RouterStateSnapshot):
   Observable<boolean> | Promise<boolean>  | boolean  {
  if(this.authService.iAuth){return  true; }
  else {this.router.navigate( commands: ['/auth']);}
  }
```

Routage -- Guards

- Le Guard doit être appliqué sur une route donnée. Ceci est indiqué en configuration de la route.

```
const appRoutes : Routes = [
  { path : 'produits', component : ProductListComponent},
  { path : 'auth', component : AuthComponent},
  {path : 'produits/:id', component : EditProductComponent},
  {path : 'panier', component : PanierComponent},
  {path : '', component : ProductListComponent},
  {path : 'admin', component : AdminComponent, canActivate :[AuthGuardService]}
  {path  : 'not-found', component : FourOhFourComponent},
  {path : '**', redirectTo : '/not-found'}
]
```

Communication avec HttpClient

- Dans les communications entre les clients front-end et les services back-end en utilisant le protocole HTTP, nous constatons que les navigateurs modernes utiliseront le protocole XMLHttpRequest ou l'API fetch ().
- **HttpClient** est une bibliothèque HTTP asynchrone intégrée à Angular, remplaçant d'autres méthodes telles que XMLHttpRequest, $ .ajax () de jQuery ou l'Api fetch.
- Angular 4.3 et versions ultérieures utilisent la bibliothèque **HttpClient** comme mécanisme permettant de gérer les requêtes http dans une application Angular.

Communication avec HttpClient

- Avant d'utiliser HttpClient, il faut d'abord importer le module **HttpClientModule** dans le module racine de l'application.

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

- Une fois que vous avez importé le module, vous pouvez l'utiliser dans votre propre composant en injectant HttpClient au service que nous créons.

```
export class ProductService {
  constructor(private  http : HttpClient) { }

  getProducts() {
    return this.http.get( url: 'http://localhost:3000/api/products')
```

Programmation réactive avec RxJs

- Pour réagir à des événements ou à des données de manière asynchrone, Angular utilise la bibliothèque **RxJS**,
- La bibliothèque la plus populaire de programmation réactive dans l'écosystème JavaScript est **RxJS**. Et c'est celle choisie par Angular.
- un Observable est un objet qui émet des informations auxquelles on souhaite réagir. un observable est comme une collection (tableau). Mais c'est une collection asynchrone, dont les éléments arrivent au cours du temps.
- Ces informations peuvent venir de la communication avec un serveur : le service Angular **httpClient** emploie les Observables.
- À cet Observable, on associe un Observer — un bloc de code qui sera exécuté à chaque fois que l'Observable **émet** une information.
- L'Observable émet trois types d'information :
  - des données (une nouvelle donnée !)
  - des erreurs (quand il y en a)
  - des terminaisons (fin du flux)

Programmation réactive avec RxJs

- il existe un type d'Observable qui permet non seulement de réagir à de nouvelles informations, mais également d'en émettre.
- Imaginez une variable dans un service, par exemple, qui peut être modifié depuis plusieurs components ET qui fera réagir tous les components qui y sont liés en même temps. Voici l'intérêt des Subjects.
- Un observable ne peut avoir qu'un seul observateur
- Un subject pour avoir plusieurs observateurs : Il autorise la souscription de plusieurs observateurs

Programmation réactive avec RxJs

- Exemple

1. Déclaration d'un objet de type subject dans le service

```
activites: Array<any>= [];
activitesSubject = new Subject<any>();
```

2. créer une méthode qui, quand le service reçoit de nouvelles données, fait émettre ces données par le Subject et appeler cette méthode dans toutes les méthodes qui en ont besoin ;

```
getActivitesFromServer(){
  this.http.get( url: BACKEND_URL+'activites').subscribe( next: (activites : any[])=>{
    this.activites = activites;
    this.emitActivitesSubject();
  });
}
   emitActivitesSubject() {
     this.activitesSubject.next(this.activites.slice());
   }
```


Programmation réactive avec RxJs

- Exemple

3. souscrire à ce Subject depuis ActivitesComponent pour recevoir les données émises, émettre les premières données

```
export class ActivitesComponent implements OnInit, OnDestroy {
  activiteSubscription : Subscription;
  activites: Array<any> = [];

ngOnInit(): void {
  this.activiteService.getActivitesFromServer();
  this.activiteSubscription = this.activiteService.activitesSubject.subscribe( next: (result : any)=>{
    this.activites = result;
  })
  this.activiteService.emitActivitesSubject();
}
```

4. implémenter OnDestroy pour détruire la souscription.

```
ngOnDestroy(): void {
    this.activiteSubscription.unsubscribe();
}
```

Communication avec HttpClient

- Par défaut, le service HttpClient réalise des requêtes AJAX avec XMLHttpRequest.
- Il propose plusieurs méthodes, correspondant au verbes HTTP communs :
- Chaque méthode renvoie un objet spécial de type **Observable** permettant à votre code d'application d'attendre de manière **asynchrone** la fin de la requête distante.
  - **get** : http.get(url: string) : Observable<Response>
  - **delete** : http.delete(url: string) : Observable<Response>
  - **post** :http.post(url: string, body: any) : Observable<Response>
  - **put** : http.put(url: string, body: any) : Observable<Response>

Communication avec HttpClient

- L'appel à httpClient.get retourne un **Observable**, on peut **s'abonner** à cet observable pour obtenir la réponse. Pour cela, vous utiliserez la fonction **subscribe**()
- La réponse est un objet Response, avec quelques champs et méthodes bien pratiques. On peut ainsi facilement accéder au code status, au headers, contenu en format json etc.
- Le corps de la réponse est la partie la plus intéressante. Mais pour y accéder, il faudra utiliser une méthode :

```
getProducts() {
  return this.http.get( url: 'http://localhost:3000/api/products')
    .subscribe( next: (response : Response)=>{
    console.log(response.status);
    console.log(response.headers);
    console.log(response.json());
    }
  );
}
```

Communication avec HttpClient

- Envoyer des données est aussi trivial.
- Il suffit d'appeler la méthode **post**(), avec l'URL et l'objet à poster :

```
ajouterProduct(product : Product){
    this.http.post(
      url: 'http://localhost:3000/products',
      product);
}
```


Communication http avec les promises

- Il existe une deuxième alternative à la bibliothèque RxJs qui coniste à convertir une réponse HTTP de type Observable en un objet JavaScript de type promise.
- Exemple : http.get(requestUrl).toPromise();
- Avec l'objet promise on peut ensuite ajouter des gestionnaires d'événements en cas d'erreur ou si l'événement asynchrone est exécuté avec succès:

```
http.get('http://localhost:3000/api/products')
  .toPromise()
  .then((response) => {
          jsonResult = response.json();
}, (error )=>{ console.log('Error Occurred:', error)
});
```

Les formulaires

Formulaires

- En Angular ou en développement web en général, les formulaires sont très utiles pour valider les saisies de l'utilisateur, afficher les erreurs correspondantes, ou aussi pour déclarer des champs obligatoires ou non, ou qui dépendent d'un autre champ,
- Elles permettent de réagir sur les changements de certains, etc. On a aussi besoin de tester ces formulaires,
- En Angular, il y a deux grandes méthodes pour créer des formulaires :
- La méthode « piloté par le template »
  - vous créez votre formulaire dans le template, et Angular l'analyse pour comprendre les différents inputs et pour en mettre à disposition le contenu ,
  - utile pour des formulaires simples, sans trop de validation.
- la méthode réactive ou « piloté par le code »
  - vous créez votre formulaire en TypeScript et dans le template, puis vous en faites la liaison manuellement à l'aide de directives
  - approche est plus complexe, mais elle permet beaucoup plus de contrôle et de validation
  - approche dynamique car ça permet de générer des formulaires dynamiquement.

Formulaires -- template

- Dans cette méthode, on va mettre en oeuvre un ensemble de directives dans notre formulaire
- Toutes ces directives sont incluses dans le module FormsModule, nous devons donc l'importer dans notre module racine.

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

const appRoutes : Routes= [ ···
@NgModule({
  declarations: [],
  imports: [
    FormsModule,
    CommonModule,RouterModule.forRoot(appRoutes)
  ],
```

- FormsModule contient les directives pour la façon "pilotée par le template".
- il existe un autre module, ReactiveFormsModule, dans le même package @angular/forms, qui est nécessaire pour la façon "pilotée par le code".

Formulaires -- template

- **ngForm**: cette directive transforme l'élément <form> en sa version Angular correspondante
- **ngModel** :La directive ngModel met à jour la valeur de l'input à chaque changement du modèle lié à l'input.
- elle génère aussi un événement à chaque fois que l'input est modifié par l'utilisateur
- cette directive est utilisé pour faire la liaison bidirectionelle « two way binding »

```
<label>Username</label><input name="username" [(ngModel)]="user.username">
```

- **ngSubmit** : est émis par la directive form lors de la soumission du formulaire. Cela invoquera une méthode écrite dans le composant qui gère la template

```
<form (ngSubmit)="register()">

<button type="submit">Register</button>

</form>
```

Formulaires -- template : ngForm

- Exemple : version 1

```
<h2>Sign up</h2>
<form (ngSubmit)="register(userForm)" #userForm="ngForm">
  <div>
    <label>Username</label>
    <input name="username" required minlength="3"
           [(ngModel)]="user.username" >
  </div>
  <div>
    <label>Password</label>
    <input type="password" required name="password"
           [(ngModel)]="user.password">
  </div>
  <button type="submit">Register</button>
</form>
```


Formulaires -- template

- Dans l'exemple précédent, on crée une variable locale userForm et lui assigner l'objet NgForm créé par Angular pour référencer le formulaire.
- C'est possible parce que la directive exporte l'instance de la directive NgForm,
- Notre méthode register est désormais appelée avec la valeur du formulaire en paramètre :

```
register(form : NgForm) {
  const username : string = form.value['username'];
  const password : string = form.value['password'];
}
```

Formulaires --template

- Angular nous offre une possibilité de lier un champ et une expression qui se mettait à jour automatiquement dès que l'utilisateur entrait une valeur dans le champ. C'est la « liaison Bi-directionnelle« ou le two-way binding

```
<h2>Sign up</h2>
<form (ngSubmit)="register(f)" #f="ngForm">
  <div>
    <label>Username</label>
    <input name="username"  minlength="3" required
           [(ngModel)]="user.username" >
  </div>
  <div>
    <label>Password</label>
    <input type="password"  name="password" required
           [(ngModel)]="user.password">
  </div>
  <button type="submit">Register</button>
</form>
```


Formulaires – Validation

- La validation de données est traditionnellement une partie importante de la construction de formulaire. Certains champs sont obligatoires, certains dépendent d'autres, certains doivent respecter un format spécifique….
- Commençons par ajouter quelques règles basiques : **tous nos champs sont obligatoires,**

```
<h2>Sign up</h2>
<form (ngSubmit)="register(f)" #f="ngForm">
  <div>
    <label>Username</label>
    <input name="username"
           required minlength="3"
           [(ngModel)]="user.username">
    <small>{{ user.username }}</small>
  </div>
  <div>
    <label>Password</label>
    <input type="password" required
           name="password" [(ngModel)]="user.password">
  </div>
  <button type="submit">Register</button>
</form>
```

Formulaires – Validation

- Un élément du formulaire a plusieurs attributs, dont :
  - **valid** : si le champ est valide
  - **invalid** : si le champ est invalide
  - **pristine** : l'opposé de dirty.
  - **dirty** : false jusqu'à ce que l'utilisateur modifie la valeur du champ.
  - **untouched** : l'opposé de touched.
  - **touched** : false jusqu'à ce que l'utilisateur soit entré dans le champ.
  - **value** : la valeur du champ.
  - **errors** : un objet contenant les erreurs du champ.

Formulaires - Template - Erreurs de soumission

- Maintenant il nous faut afficher les erreurs sur chaque champ.
- on peut donc créer une variable locale pour accéder aux erreurs :

```
<h2>Sign up</h2>
<form (ngSubmit)="register(f)" #f="ngForm">
  <div>
    <label>Username</label>
    <input name="username"
           required minlength="3"
           [(ngModel)]="user.username" #userName="ngModel">
    <small>{{ user.username }}</small>
  </div>
  <div *ngIf="userName.dirty
&& userName.hasError( errorCode: 'required')">
    User name is required
  </div>
```

Formulaires - Template - Erreurs de soumission

- Evidemment, on veut que l'utilisateur ne puisse pas soumettre le formulaire tant qu'il reste des erreurs, et ces erreurs doivent être parfaitement affichées.

```
<h2>Sign up</h2>
<form (ngSubmit)="register(f)" #f="ngForm">
  <div>
    <label>Username</label>
    <input name="username"
           required minlength="3"
           [(ngModel)]="user.username">
    <small>{{ user.username }}</small>
  </div>
  <div>
    <label>Password</label>
    <input type="password" required
           name="password" [(ngModel)]="user.password">
  </div>
  <button type="submit" [disabled]="f.invalid">Register</button>
</form>
```


Formulaires – Piloté par le code

- Angular introduit une déclaration impérative, qui permet de construire les formulaires programmatiquement.
- Désormais, on peut manipuler les formulaires directement depuis le code. C'est plus verbeux mais plus puissant.
- Pour construire un formulaire dans notre code, nous allons utiliser les abstractions dont nous avons parlé: FormControl et FormGroup

Formulaires – Piloté par le code : FormBuilder

- Avec ces briques de bases on peut construire un formulaire dans notre composant. Mais au lieu d'utiliser new FormControl() ou new FormGroup(), on va utiliser une classe utilitaire, FormBuilder, qu'on peut s'injecter.
- FormBuilder est une classe utilitaire, avec plein de méthodes bien pratiques pour créer des contrôles et des groupes.

Classe du composant

```
export class RegisterFormComponent {
   userForm: FormGroup;
   constructor(fb: FormBuilder) {
       this.userForm = fb.group({
           username: '',
           password: ''
       });
   }
}
```

Template du composant

```
<h2>Sign up</h2>
<form (ngSubmit)="register()" [formGroup]="userForm">
 <div>
   <label>Username</label>
   <input formControlName="username" />
 </div>
 <div>
   <label>Password</label>
   <input type="password" formControlName="password" />
 </div>
 <button type="submit">Register</button>
</form>
```

Formulaires – Piloté par le code

- On utilise la notation avec crochets [formGroup]="userForm" pour relier notre objet userForm à formGroup.
- Chaque input reçoit la directive formControlName avec pour valeur le nom du contrôle auquel il est relié. Si l'on indique un nom qui n'existe pas, on aura une erreur. Comme on passe une valeur (et pas une expression), on n'utilise pas les [] autour de formControlName.

Validation du formulaire

- Pour spécifier que chaque champ est requis, on va utiliser Validator.
- Un validateur retourne une map des erreurs, ou null si aucune n'a été détectée.Quelques validateurs sont fournis par le framework
  - Validators.required pour vérifier qu'un contrôle n'est pas vide
  - Validators.minLength(n) pour s'assurer que la valeur entrée a au moins n caractères
  - Validators.maxLength(n) pour s'assurer que la valeur entrée a au plus n caractères
  - Validators.pattern(p) pour s'assurer que la valeur entrée correspond à l'expression régulière p définie.

Validation - configuration

- Les validateurs peuvent être multiples, en passant un tableau, et peuvent s'appliquer sur un FormControl ou un FormGroup.
- Comme on veut que tous les champs soient obligatoires, on peut ajouter le validator required sur chaque contrôle, et s'assurer que le nom de l'utilisateur fait 3 caractères au minimum.

```
export class RegisterFormComponent {
    userForm: FormGroup;
    constructor(fb: FormBuilder) {
        this.userForm = fb.group({
            username: fb.control('', [
              Validators.required,
              Validators.minLength(3)
            ]),
            password: fb.control('', Validators.required)
        });
    }
}
```

Validation - Contrôle de soumission

- Nous avons ajouté un champ userForm, du type FormGroup , à notre composant. Ce champ fournit une vision complète de l'état du formulaire et de ses champs, incluant ses erreurs de validation.

```
<h2>Sign up</h2>
<form (ngSubmit)="register()" [formGroup]="userForm">
   <div>
       <label>Username</label>
       <input formControlName="username">
   </div>
   <div>
       <label>Password</label>
       <input type="password" formControlName="password">
   </div>
   <button type="submit"
          [disabled]="!userForm.valid">Register</button>
</form>
```


Validation - Affichage des erreurs

```
<form (ngSubmit)="register()" [formGroup]="userForm">
   <label>Username</label>
   <input formControlName="username">
    <div  *ngIf="userForm.get('username').hasError('required')">
        Username is required
    </div>
    <label>Password</label>
    <input type="password" formControlName="password">
  <button [disabled]="!userForm.valid">Register </button>
</form>
```

Validateurs spécifiques

- On souhaite inscrire un utilisateur que s'il est agé de plus de 12 ans. Comment fait-on cela ? En créant un validateur spécifique. Pour ce faire, il suffit de créer une méthode qui accepte un FormControl, teste sa value, et retourne un objet avec les erreurs, ou null si la valeur est valide

```
export class MyValidators {
  static isOldEnough(control: FormControl) {
    const bd = new Date(control.value); // 10/10/1990
    bd.setFullYear(bd.getFullYear() + minAge);
    return bd < new Date() ? null : { tooYoung: true };
  }
}
```


Validateurs spécifiques : utilisation

```
...
import { MyValidators} from './validators';
const minAge = 12;
@Component(...)
export class RegisterFormComponent {
   birthdateCtrl: FormControl;
   userForm: FormGroup;
   constructor(fb: FormBuilder) {
       this.birthdateCtrl =
         fb.control('', [MyValidators.isOldEnough]);

       this.userForm = fb.group({
           birthdate: this.birthdateCtrl
       });
   }
}
```

Réagir aux modifications

- Avec un formulaire piloté par le code : on peut facilement réagir aux modifications, grâce à l'observable **valueChanges**.
- Par exemple, disons que notre champ de mot de passe doit afficher un indicateur de sécurité.
- On veut en calculer la robustesse lors de chaque changement du mot de passe

```
this.passwordCtrl.valueChanges
 .subscribe(newValue => {
    this.passwordStrength = this.calcStrentgh(newValue.length)
);
```


Déployer votre application

Stratégies de déploiement

- Lors de la mise en production d'une application Angular/NodeJS, deux scénarios ou options peuvent être choisis pour le déploiement. Chacun de ces scénario présente des avantages et des inconvénients.
  1. Scénario 1 : Déploiement séparé
     - deux applications séparates qui s'exécutent sur deux serveurs (machines) différents.
  2. Scénario 2 : Déploiement combiné
     - une seule application Node Rest API qui va juste rendre l'application Angular

Déploiement séparé

- Deux applications s'exécutant sur deux domaines/ports différents
- L'application Angular en Front-end tourne sur un host statique qui n'exécute pas un code serveur, ce host doit juste être capable d'exécuter un code HTML, CSS et JS.
  - Exemple : Apache, AWS S3, Firebase hosting (google),
- L'application back-end nodeJS/Express tourne sur un host qui est capable d'exécuter un code nodeJs
  - Exemple : AWS EC2/ EB, Heroku
- Il est important d'ajouter les entêtes CORS dans la partie NodeJS/express (domaines différents)

Déploiement combiné

- Une seule application qui s'exécute sur un host capable d'exécuter un code NodeJs et qui retournent les fichiers Angular.
- Pas besoin de régler les entêtes CORS vu que Angular s'exécute sur le même domaine que la partie NodeJs/express

Étapes de déploiement dans Heroku

1. Builder l'application Angular
   - La première étape consiste à builder le projet Angular en utilisant le commande ng build –prod
   - Ceci génère un dossier nommé **dist** que vous pouvez optionnellement renommer (ex, Angular) et transporter tel quel dans un des dossiers gérés par votre serveur préféré (Apache, Node.js, http-server...).
2. Déplacer le dossier **dist** générer dans le dossier racine de l'application NodeJs/express.
3. Préparer le code dans votre contrôleur principal (server.js) pour rediriger les appels vers http vers l'index html.

```
app.get('', (req, res) => {
  res.sendFile(path.join(__dirname, 'angular','index.html'));
});
```

Étapes de déploiement

4. Installation de Heroku CLI et Création d'un compte
5. Installation du gestionnaire de version git
6. Transfert du code au serveur Heroku via git

```
$ heroku login
        Email : anwar@emi.ac.ma
        Password : #########
$ heroku create arram-demo
$ git init
$ git add .
$ git commit -m 'my first commit‘
$ heroku git:remote –a arram-demo
$ git push heroku master
```