



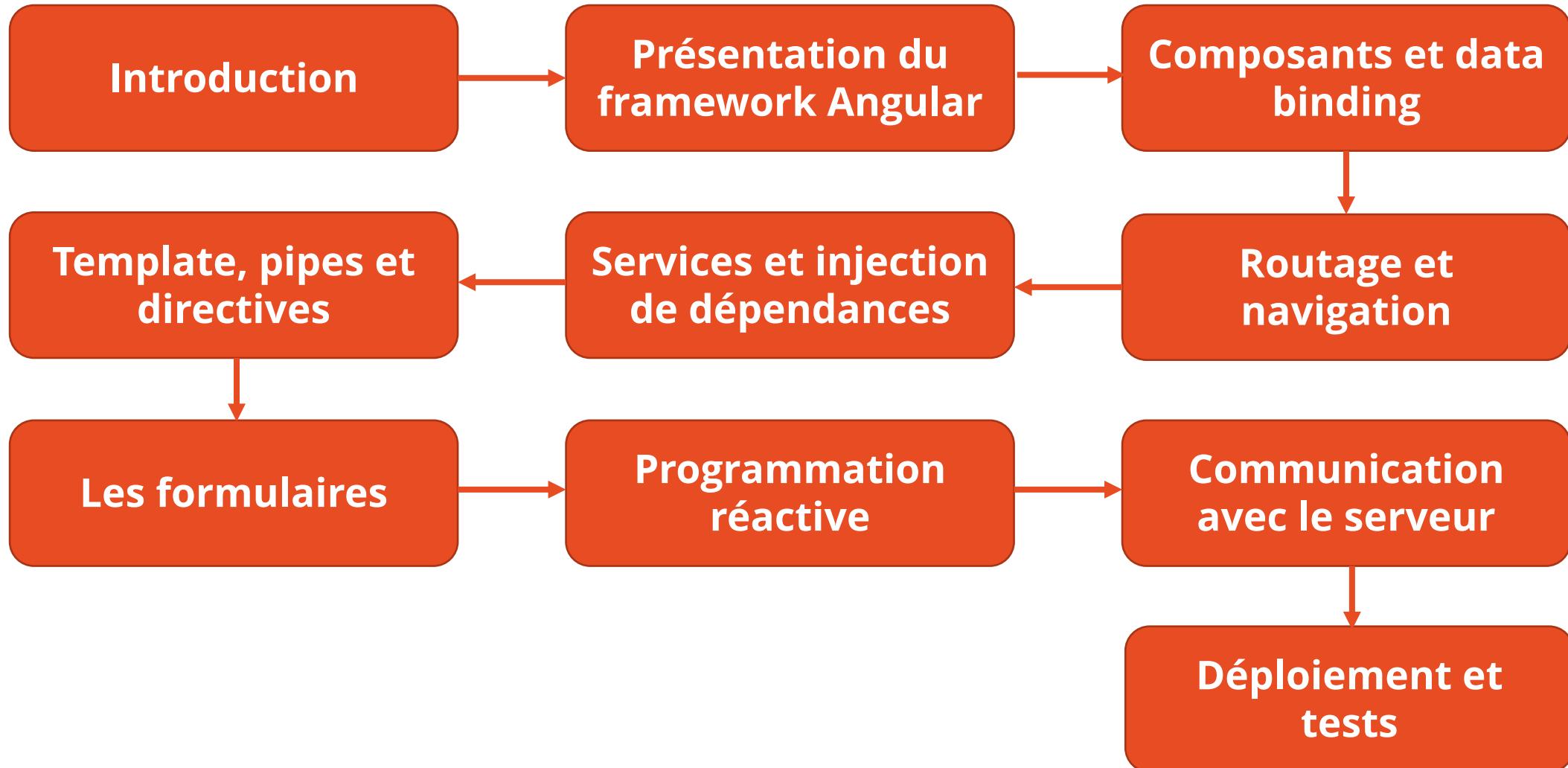
Formation : Angular, maîtriser le Framework Front-End de Google

Animée par : Mehdi M'tir

Mehdi.mtir@gmail.com

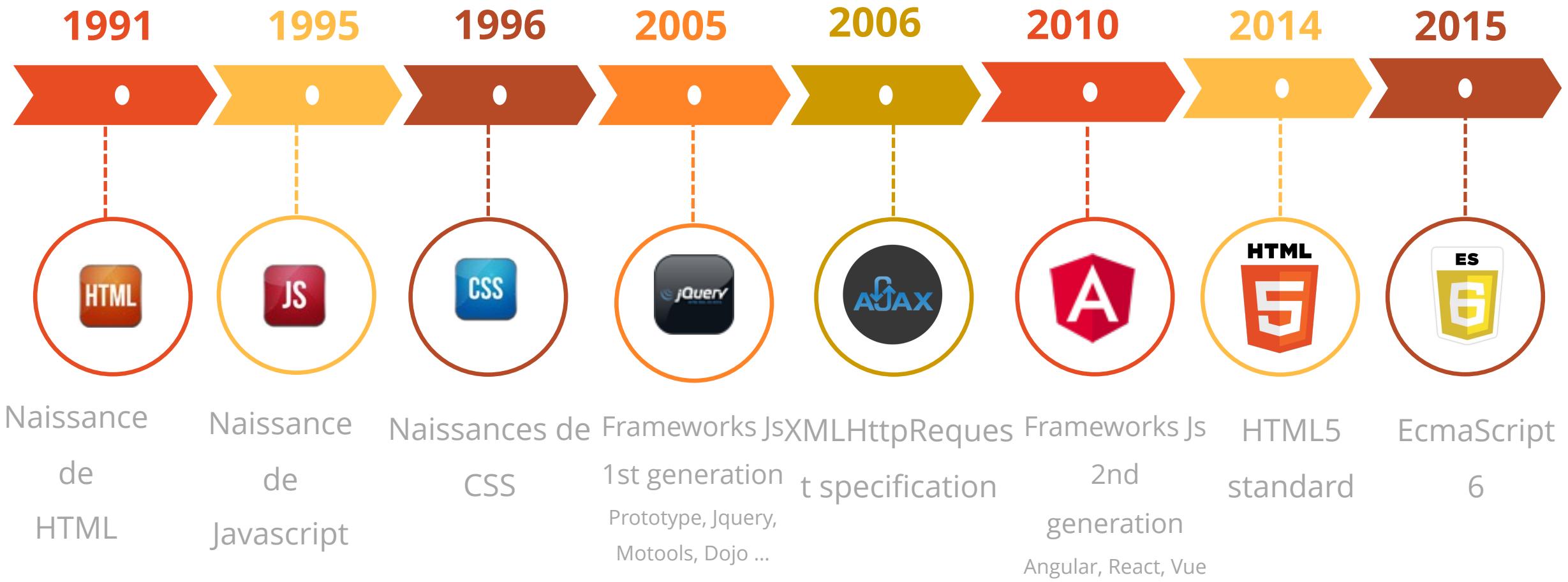


Plan de la formation



Introduction

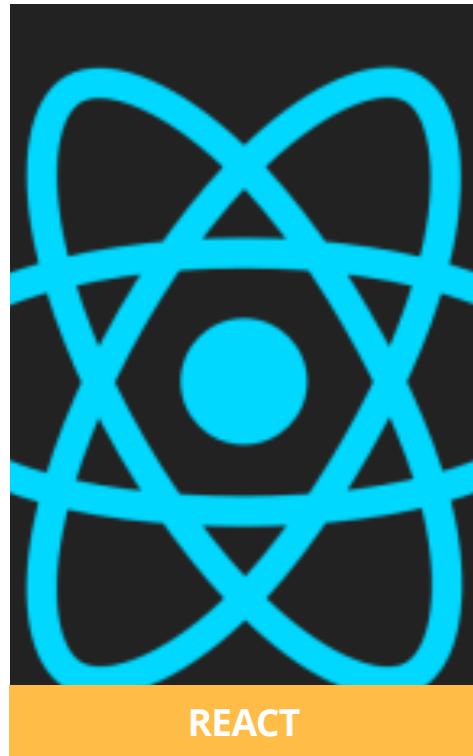
Bref Historique des technologies front-end



Principaux frameworks Javascript

Plusieurs frameworks Javascript MVC sont apparus au cours des dernières années

.....



www.angular.io

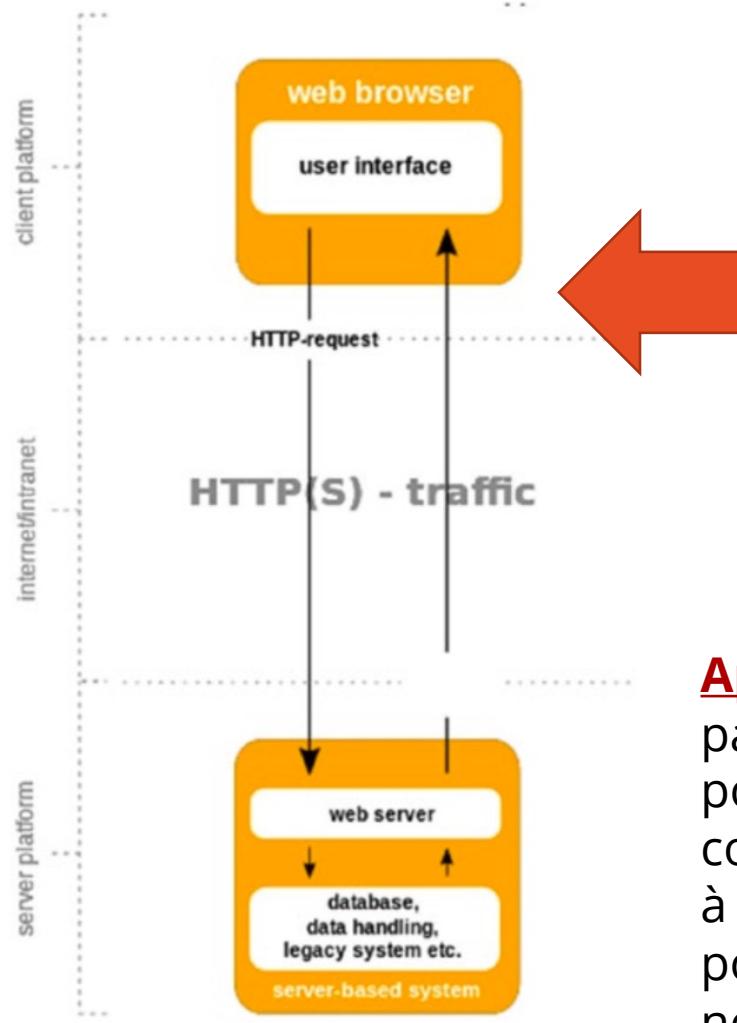
www.reactjs.org

www.vue.js

www.svelte.dev

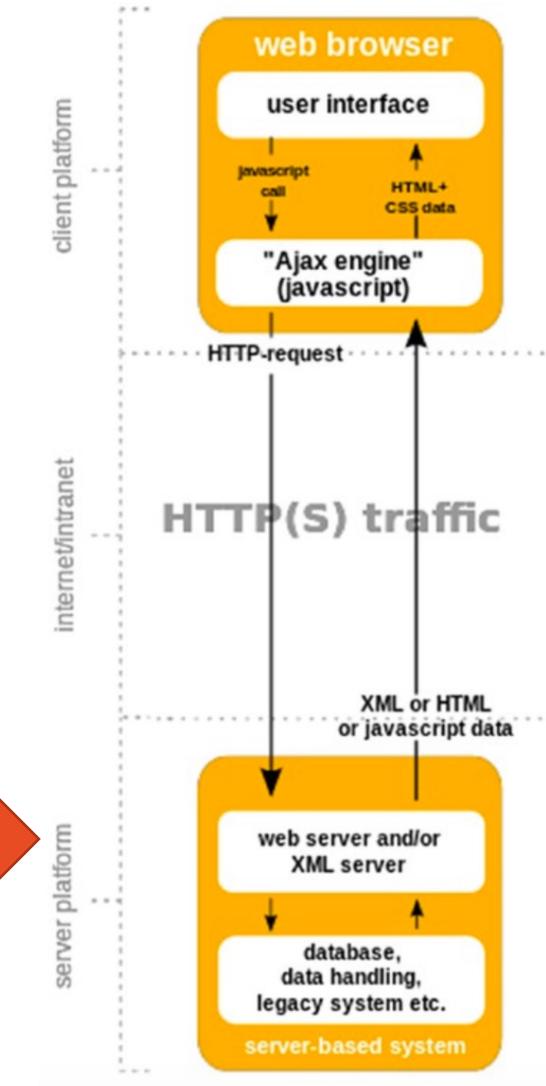
Approche classique vs Approche moderne

Différences entre les architectures classique et moderne des applications web



Approche classique : tout le travail est fait côté serveur. Le navigateur se contente d'afficher les fichiers reçus.

Approche moderne : Une partie du travail est déléguée pour le client. Celui-ci peut communiquer avec le serveur à travers des requêtes AJAX pour récupérer les données nécessaires.



Présentation du framework Angular



Framework Javascript



Open source



Développé par Google



Basé sur les composants



Single Page Applications



Développé avec Typescript

Les versions du framework Angular

.....

AngularJS (version 1)

Refonte complète!

Framework Angular (couvert dans cette formation)

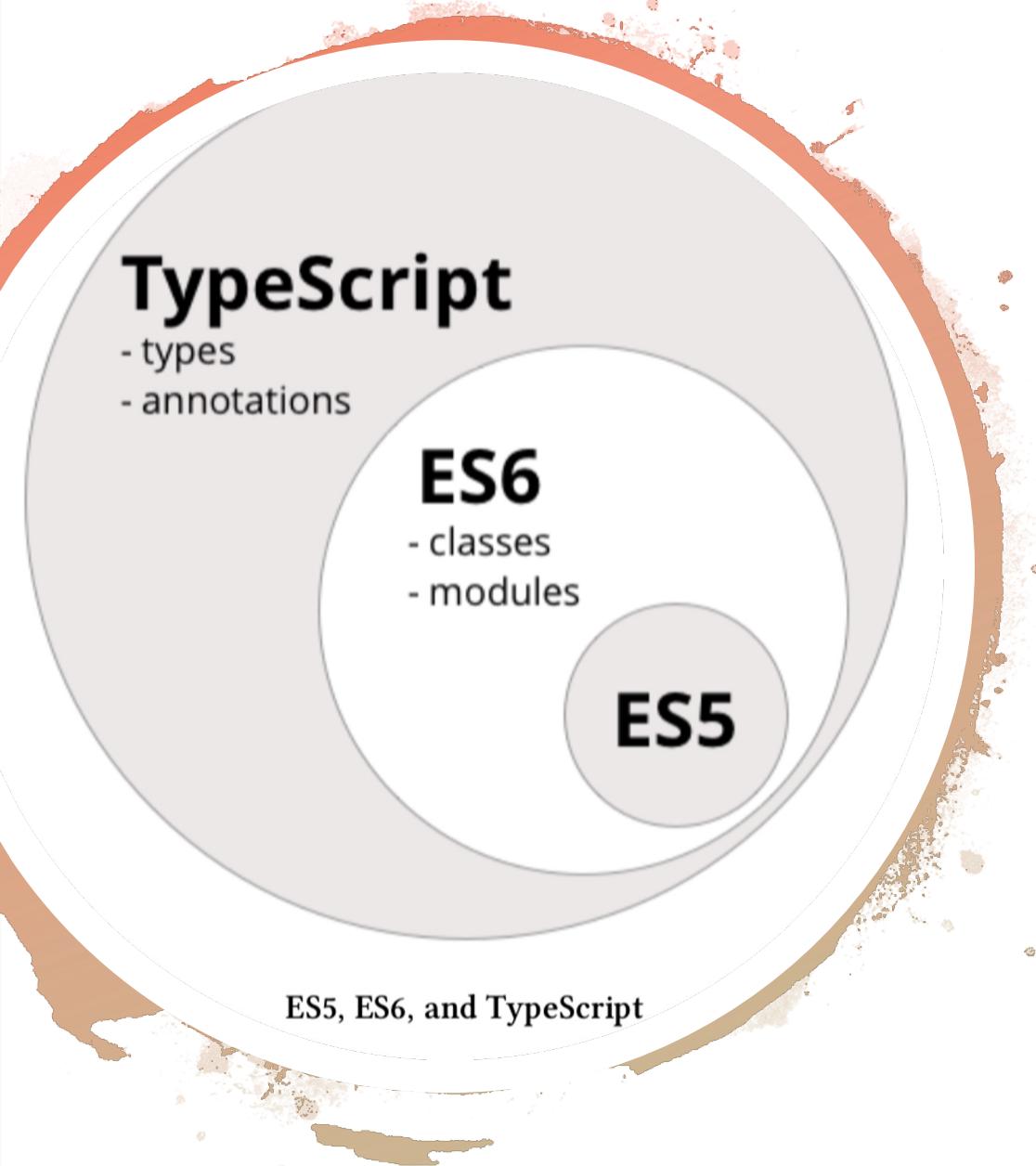
Angular 2

Angular 4

• • •

Angular 16

Une nouvelle version tous les 6 mois
changements mineurs, incrémentaux avec une compatibilité ascendante



Présentation de Typescript

- Conçu par Microsoft pour le développement d'applications JavaScript à grande échelle.
- Conçu comme un sur-ensemble typé de JavaScript.
- La compilation (avec un transpiler) permet de générer du code Javascript standard interprétable par un navigateur web.
- Les fichiers TypeScript ont l'extension **.ts** et le transpiler produit les fichiers **.js**.
- Site officiel :
<https://www.typescriptlang.org/>
- <https://github.com/Microsoft/TypeScript>

Avantages de Typescript par rapport à JS ES5 (1)

- ✓ Types : String, Number, Boolean, Array, Enums, Any, Void

```
var fullName: string;           function greetText(name: string): string {
                                return "Hello " + name;
}

```

- ✓ Classes : Propriétés, méthodes, constructeur, héritage...

```
class Person {                   // declare a variable of type Person
    first_name: string;       var p: Person;
    last_name: string;        // instantiate a new Person instance
    age: number;              p = new Person();
                               // give it a first_name
    greet() {                  p.first_name = 'Felipe';
        console.log("Hello", this.first_name);   // call the greet method
    }                           p.greet();
}

```

Avantages de Typescript par rapport à JS ES5 (2)

✓ Décorateurs :

Les décorateurs fournissent un moyen d'ajouter des annotations et de la métaprogrammation pour les déclarations de classes, méthodes, propriétés ...

```
@sealed
class Greeter {
    greeting: string;
    constructor(message: string) {
        this.greeting = message;
    }
    greet() {
        return "Hello, " + this.greeting;
    }
}
```

✓ Imports / Exports de modules :

Les modules en typescript sont exécutés dans leur propre scope, pas dans un scope global; cela signifie que les variables, fonctions, classes, etc. déclarées dans un module ne sont pas visibles en dehors du module sauf si elles sont explicitement exportées (**export**).

Inversement, pour consommer une variable, une fonction, une classe, une interface, etc. exportées depuis un module différent, il faut l'importer (**import**).

Avantages de Typescript par rapport à JS ES5 (3)

- Utilitaires de langage :

Fat Arrow function :

```
// ES5-like example
var data = ['Alice Green', 'Paul Pfifer', 'Louis Blakenship'];
data.forEach(function(line) { console.log(line); });
```

```
// Typescript example
var data: string[] = ['Alice Green', 'Paul Pfifer', 'Louis Blakenship'];
data.forEach( (line) => console.log(line) );
```

Variables in strings

```
var firstName = "Nate";
var lastName = "Murray";

// interpolate a string
var greeting = `Hello ${firstName} ${lastName}`;

console.log(greeting);
```

Multiline String

```
var template = `
<div>
  <h1>Hello</h1>
  <p>This is a great website</p>
</div>
`
```

Transpilation

- TypeScript est reconverti en JavaScript (compatible avec les navigateurs web) en utilisant un processus appelé transpilation.
- Un transpiler est un logiciel qui convertit le code source d'un langage de programmation vers un autre. Par exemple, TypeScript, CoffeeScript, Caffeine, Kaffeine et plus de deux douzaines de langues sont transcrits dans JavaScript.
- Voir des exemple sous www.typescriptlang.org/play/

The image shows a screenshot of the TypeScript playground interface. On the left, the TypeScript code is displayed:

```
1 class Greeter {
2     greeting: string;
3     constructor(message: string) {
4         this.greeting = message;
5     }
6     greet() {
7         return "Hello, " + this.greeting;
8     }
9 }
10 let greeter = new Greeter("world");
11 let button = document.createElement('button');
12 button.textContent = "Say Hello";
13 button.onclick = function() {
14     alert(greeter.greet());
15 }
16 document.body.appendChild(button);
```

On the right, the generated JavaScript code is shown:

```
1 var Greeter = /** @class */ (function () {
2     function Greeter(message) {
3         this.greeting = message;
4     }
5     Greeter.prototype.greet = function () {
6         return "Hello, " + this.greeting;
7     };
8     return Greeter;
9 })();
10 var greeter = new Greeter("world");
11 var button = document.createElement('button');
12 button.textContent = "Say Hello";
13 button.onclick = function () {
14     alert(greeter.greet());
15 };
16 document.body.appendChild(button);
```

The interface includes tabs for "Using Classes", "TypeScript", "Share", "Options", "Run", and "JavaScript".

Débogage et fichiers Map

- ✓ Le processus de transpilation décrit précédemment rend le débogage difficile puisque vous écrivez votre code dans un langage et le navigateur exécute un autre.
- ✓ Les fichiers Map sont très utiles pour remédier à ce problème.
- ✓ Les fichiers Map sont automatiquement générés par votre transpiler et donnent au navigateur les informations nécessaires pour mapper le code d'origine (TypeScript) au code déployé (JavaScript).
- ✓ Cela signifie que le débogueur JavaScript peut vous permettre de déboguer votre code source comme si le navigateur l'exécutait.
- ✓ Il est donc fortement recommandé d'activer les fichiers .Map dans votre navigateur pour vous aider dans la phase de débogage.
- ✓ Liens utiles pour Google chrome :
 - ✓ <https://developers.google.com/web/tools/chrome-devtools/javascript/source-maps>
 - ✓ <https://gist.github.com/jakebellacera/336c4982194bcb02ef8a>

Node.js® is a JavaScript runtime built on [Chrome's V8 JavaScript engine](#).

Download for macOS (x64)

16.13.0 LTS

Recommended For Most Users

17.1.0 Current

Latest Features

[Other Downloads](#) | [Changelog](#) | [API Docs](#) [Other Downloads](#) | [Changelog](#) | [API Docs](#)

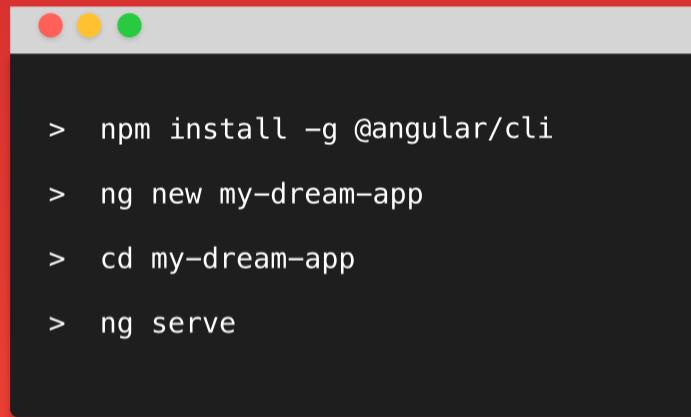
Installer Node JS et NPM

Gestion des paquets NPN

npm est le gestionnaire de paquets pour JavaScript et le plus grand registre de logiciels au monde.

npm permet d'installer, partager et distribuer du code source

npm facilite la gestion des dépendances dans votre projet.

A screenshot of a terminal window with a dark background and light text. It shows four command-line entries:

```
> npm install -g @angular/cli
> ng new my-dream-app
> cd my-dream-app
> ng serve
```

Angular CLI

A command line interface for Angular

[GET STARTED](#)

Installer Angular CLI

Command Line Interface for Angular

ng new : Créer une nouvelle application

ng generate : Générer un nouveau composant, service, route

...

ng serve : tester facilement l'application en cours

Version 1.17 is now available! Read about the new features and fixes from September.

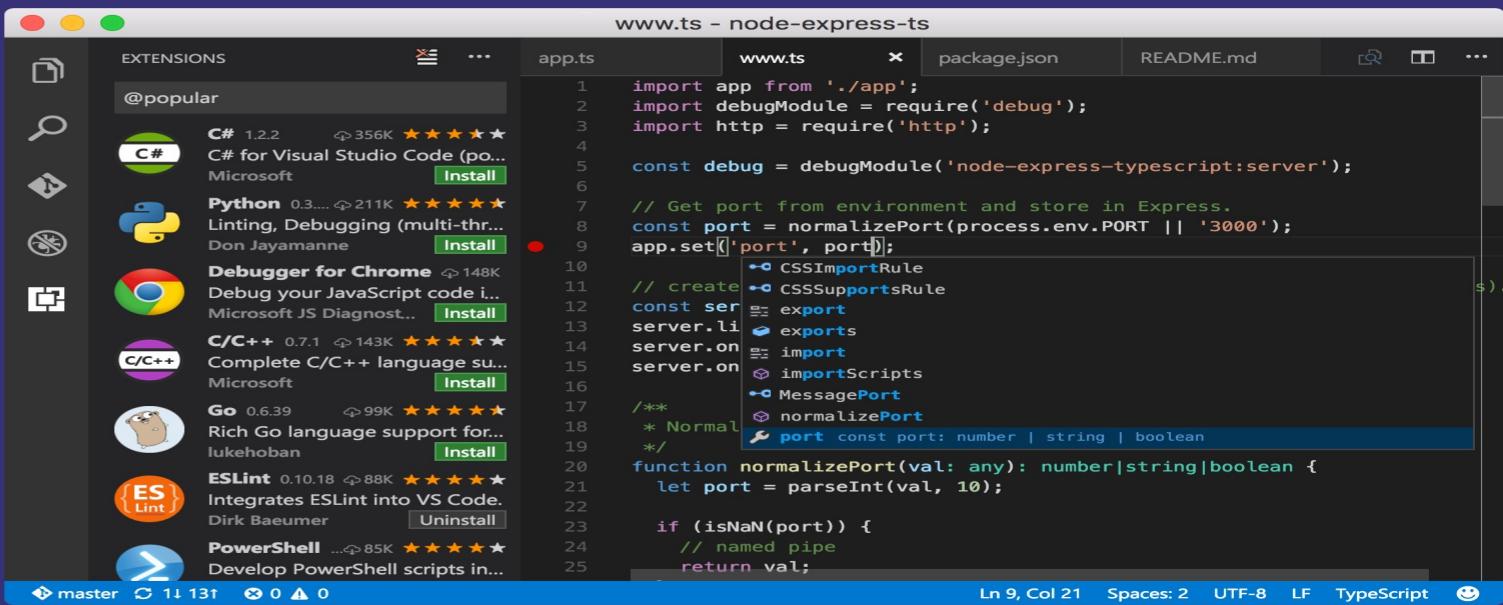
Code editing. Redefined.

Free. Open source. Runs everywhere.

Download for Mac
Stable Build

Other platforms and Insiders Edition

By using VS Code, you agree to its
license and privacy statement.



Installer Visual Studio Code

EDI Open source

Autres alternatives :

WebStorm

SublimeText

Atom

...

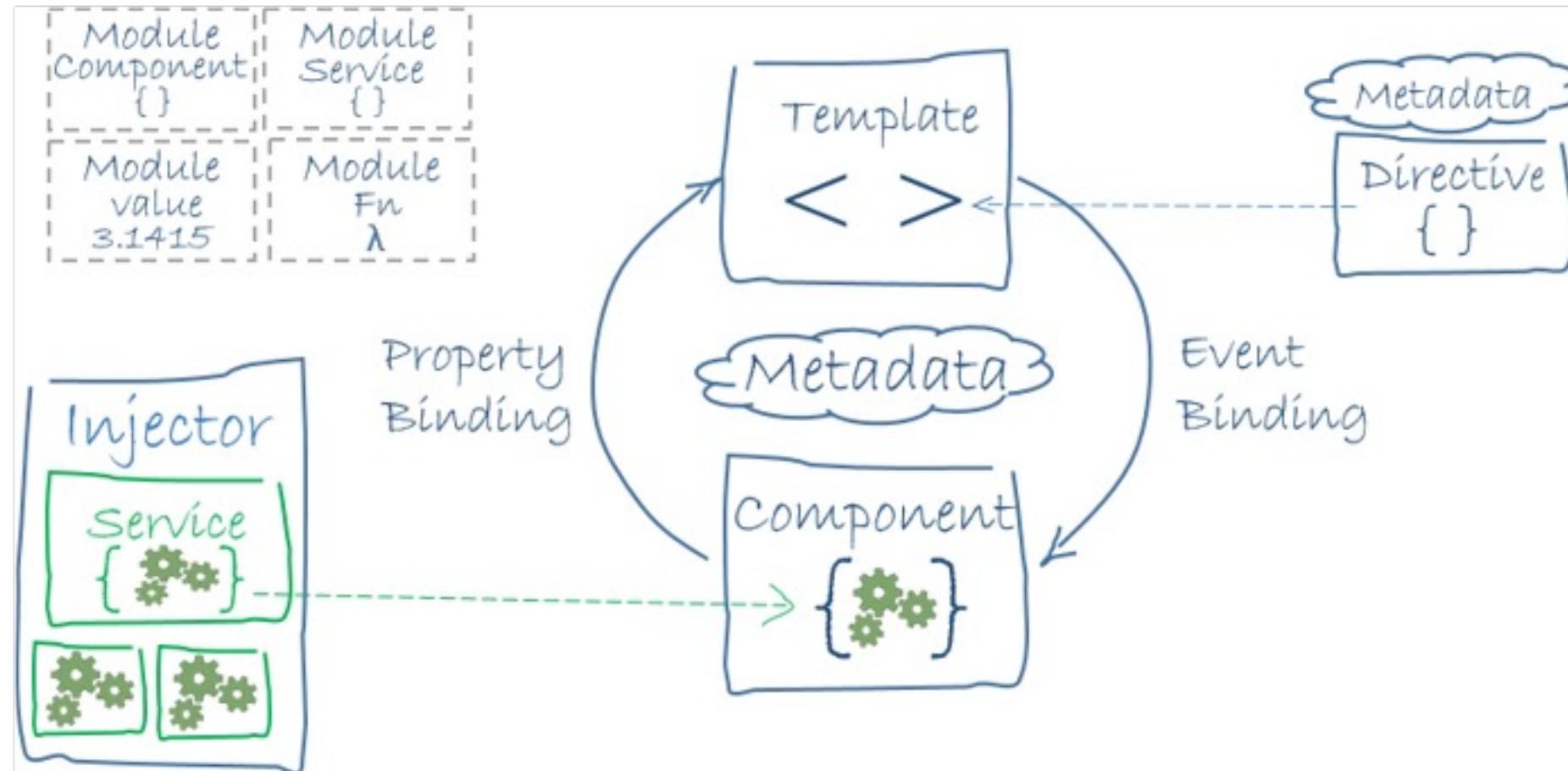
Hello World App

- Installer Node Js (`node -v` pour vérifier si une version est déjà installée)
- Installer Angular CLI (`npm install -g @angular/cli`)
- Créer un projet (`ng new angular-hello-world`)
- **Remarque : pendant la création du premier projet, nous n'allons pas intégrer Angular routing et nous allons utiliser CSS comme format de feuille de style.**
- Lancer le projet
 - Accéder au répertoire créé (`cd angular-hello-world`)
 - Compiler le projet et lancer le serveur (`ng serve`)
 - Accéder au projet dans le navigateur (**`http://localhost:4200`**)

Présentation du framework

Architecture

The big picture



<https://angular.io/guide/architecture>

Amorçage d'une application (Bootstrapping)

Ce que Angular CLI fait en coulisse

Chaque application a un point d'entrée principal. Cette application a été créée en utilisant Angular CLI (basé sur Webpack). Nous exécutons cette application en appelant la commande: `ng serve`

Etapes d'amorçage :

1. La commande `ng serve` cherche le fichier **angular.json**
2. **angular.json** spécifie un fichier "principal", qui dans ce cas est **main.ts**
3. **main.ts** est le point d'entrée de notre application et il amorce notre application
4. Le processus d'amorçage démarre un module Angular (**AppModule** spécifié dans `src /app/app.module.ts`)
5. **AppModule** spécifie le composant à utiliser en tant que composant de niveau supérieur (**AppComponent**).
6. **AppComponent** se charge de remplir le fichier **index.html** (affiché par défaut)

Modules (1)

Ng Modules

- ✓ Les applications Angular sont modulaires et Angular a son propre système de modularité appelé NgModules.
- ✓ Chaque application Angular contient au moins une classe NgModule (root module), conventionnellement appelée AppModule.
- ✓ Angular utilise les modules comme méthode d'assemblage de divers **composants, directives, pipes** et **services** dans une même unité cohérente.
- ✓ Pour créer un module, vous devez définir une classe et la décorer avec le mot clé @NgModule.
- ✓ Vous devrez commencer par importer deux modules d'Angular.

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

Modules (2)

Ng Modules

The screenshot shows a code editor interface with two main sections: an Explorer view on the left and a code editor on the right.

EXPLORATEUR

- ▶ ÉDITEURS OUVERTS
- ◀ MYFIRSTANGULAR
 - ▶ e2e
 - ▶ node_modules
 - ◀ src
 - ▶ app
 - # app.component.css
 - ↳ app.component.html
 - TS app.component.spec.ts
 - TS app.component.ts
 - TS app.module.ts
 - ▶ assets
 - ▶ environments
 - ★ favicon.ico

TS app.module.ts ×

```
1 import { BrowserModule } from '@angular/platform-browser';
2 import { NgModule } from '@angular/core';
3
4 import { AppComponent } from './app.component';
5
6 @NgModule({
7   declarations: [
8     AppComponent
9   ],
10  imports: [
11    BrowserModule
12  ],
13  providers: [],
14  bootstrap: [AppComponent]
15 })
16 export class AppModule { }
```

Modules (3)

NgModules vs. JavaScript modules

- ✓ Le NgModule d'Angular diffère et complète la notion de module javascript (ES6) servant à gérer une collection d'objets javascript.
- ✓ Le module déclare certains objets **publics** en les marquant avec le mot clé **export**.
- ✓ D'autres modules utilisent **import** pour accéder aux objets publics partagés.

Avantages :

- Organisation du code en unités fonctionnelles distinctes → Meilleur gestion des projets complexes.
- Possibilité de réutilisation du code.
- Lazy loading : charger uniquement les modules nécessaires → Démarrage plus rapide.

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

export class AppModule { }
```

Les bibliothèques (1)

Angular Libraries

- Angular est une collection de modules JavaScript. On peut les considérer comme des bibliothèques. Le nom de chaque bibliothèque Angular commence avec le préfixe @angular.
- On peut installer les bibliothèques Angular avec le gestionnaire de paquets npm et en importer des parties avec des instructions d'importation JavaScript.
- Par exemple, on peut importer le décorateur de composants d'Angular à partir de la bibliothèque **@angular/core** comme ceci:

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

Les bibliothèques (2)

Angular Libraries

- ✓ On importe également des NgModules à partir de bibliothèques Angular à l'aide d'instructions d'importation JavaScript:

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

- ✓ Pour accéder au contenu du module **BrowserModule**, il faut l'ajouter aux **imports** de métadonnées **@NgModule** comme ceci..

```
imports:      [ BrowserModule ],
```

→ De cette façon, vous utilisez les systèmes de modules Angular et JavaScript ensemble.

Composants et data binding

Les composants (1)

Présentation générale

- ✓ Un composant contrôle une partie d'un écran appelée une vue.
- ✓ Chaque composant doit être défini dans un fichier `.ts`.
- ✓ Ce fichier contient généralement 3 sections :
 - **La section import** : qui définit les autres composants qu'on veut utiliser (importer) dans notre composant.
 - **La section metadata** : qui décrit les détails du composant pour le framework Angular.
 - **La section contenant le code du composant.**

Les composants (2)

La section Import

- ✓ Syntaxe de base :

```
import { Component_name } from 'module location'
```

- ✓ Dans cette section, il faudra au moins importer le composant **Component** à partir du module core d'Angular. Ceci est nécessaire pour fournir les métadonnées de la section suivante .

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

- ✓ Il existe plusieurs composants qu'on peut importer des bibliothèques d'Angular. Parmi les plus utilisés, on peut citer :
 - **HttpClient** dans **@angular/common/http**: Opérations HTTP de base.
 - **Router** dans **@angular/router**: Lier les URLs aux composants correspondants.

Les composants (3)

La section Metadata

- ✓ La section Metadata (métadonnées) est constituée d'un ensemble de propriétés qui permettent de décrire le composant pour le framework Angular.
- ✓ Plusieurs propriétés peuvent être définies dans cette section mais les plus importantes sont : **selector** et **template** (les 2 propriétés obligatoires)
- ✓ La valeur de la propriété **selector** est une chaîne de caractères faisant référence à l'élément de la page HTML où le contenu du composant doit être affiché.
- ✓ La valeur de la propriété **selector** correspond le plus souvent au **nom d'un élément** HTML, mais elle peut également contenir un **sélecteur de classe** CSS ou un **sélecteur d'attribut**.
- ✓ La propriété **template** permet de définir le code HTML à insérer dans l'élément référencé par la propriété **selector**. Le contenu de cette propriété contient souvent, en plus du code HTML, des éléments spécifiques à Angular tels que les expressions `{{...}}`
- ✓ La propriété **template** peut être remplacée par **templateURL** pour permettre la définition du code HTML dans un fichier séparé.

Les composants (4)

Autres propriétés de la section Metadata

- ✓ **styles** : permet de définir des styles CSS à associer aux éléments du template. Les styles définis dans cette section doivent être délimités par des crochets [], ils ne s'appliqueront qu'au code HTML du composant. Si les styles sonts définis sur plusieurs lignes, il faut les délimiter par des backticks (`).
- ✓ **styleUrls** : comme pour templateUrl, ceci permet de définir les styles du composant dans un fichier séparé.
- ✓ **encapsulation** : permet à Angular de fonctionner avec des navigateurs qui ne supportent pas le shadow DOM. Cet attribut peut recevoir l'une des valeurs suivantes :
 - ✓ Emulated (valeur par défaut)
 - ✓ ShadowDom
 - ✓ None

Les composants (5)

Le code du composant

- ✓ Dans cette section, nous décrivons le code de la classe définissant le comportement du composant.
- ✓ Le nom de la classe doit être précédé par l'instruction **export class**.

```
'''  
export class AppComponent {  
    // Properties  
    ClubTitle: string;  
    LeagueTitle: string;  
    // Constructor  
    public constructor() {  
        this.ClubTitle = "The 422 Sportsplex";  
        this.LeagueTitle = "Adult Over 30 Leagues";  
    }  
    // Class methods  
    public ChangeLeagues( newLeague: string ) {  
        this.LeagueTitle = newLeague;  
    }  
}
```

Les composants (6)

Les propriétés de la classe

✓ Syntaxe :

```
[ public | private | protected ]  
    Property Name: data Type  
        [ = initial Value ]
```

- ✓ Par défaut les propriétés sont déclarés comme **public**.
- ✓ Le nom d'une propriété (Property Name) suit les mêmes règles de déclaration d'une variable typescript : peut contenir des caractères alphanumériques et des underscores mais ne doit pas commencer par un chiffre.
- ✓ Le type de données (data Type) est optionnel si une valeur est affectée à la propriété : le type sera affecté implicitement.
- ✓ Il est recommandé de déclarer les propriétés **private** et d'utiliser des accesseurs getter/setter pour y accéder de l'extérieur.

Les composants (7)

Le constructeur de la classe

- ✓ Le constructeur est une méthode publique (elle ne peut pas être privée ou protégée) qui est appelée chaque fois que la classe est instanciée.
- ✓ Si vous ne spécifiez pas de méthode constructeur, un constructeur par défaut est utilisé.
- ✓ Une classe ne peut avoir qu'un seul constructeur.
- ✓ La syntaxe est comme n'importe quelle fonction TypeScript void, en utilisant le nom **constructor**.
- ✓ Il est souvent utilisé pour initialiser des variables, ou peut-être faire un appel à une base de données pour définir diverses propriétés de l'objet.

Les composants (8)

Les méthodes de la classe

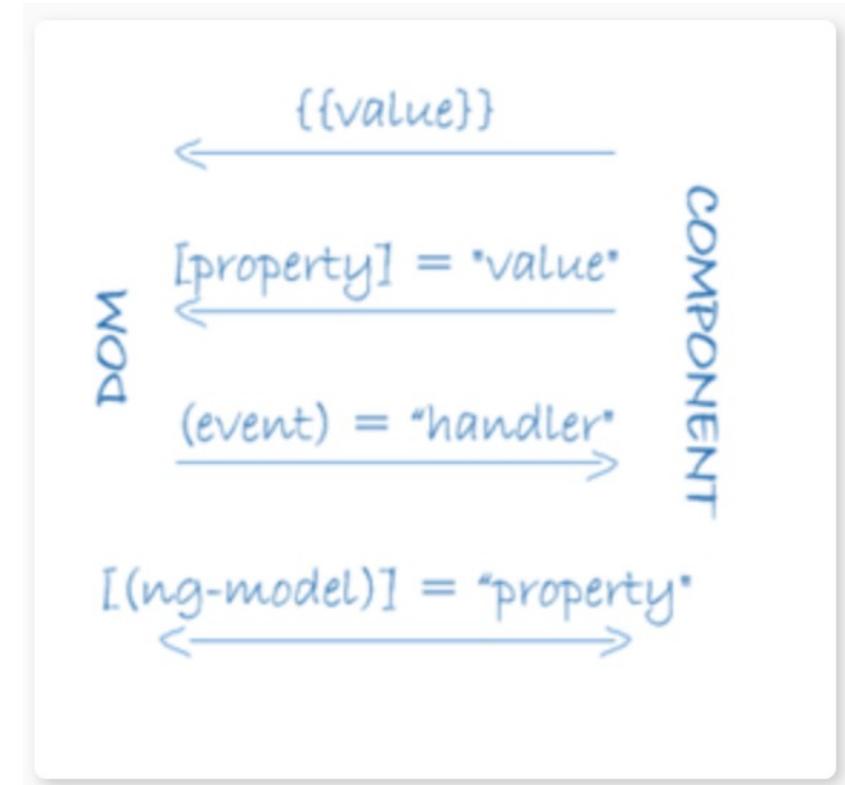
✓ Syntaxe :

```
[ public | private | protected ]  
    Method Name ( optional parameters )  
    : optional return type  
    {      body of method  
    }
```

- ✓ Par défaut une méthode est déclarée comme public.
- ✓ Si aucun type de retour n'est spécifié, undefined est assigné.
- ✓ Pour utiliser les propriétés de la classe, il faut les précéder par **this**.
- ✓ Il est possible de déclarer des variables à l'intérieur d'une méthode

Data binding

- Angular inclut le concept de **data binding**, qui consiste essentiellement à faire communiquer la partie métier (fichier typescript du composant) avec le template pour :
 - amener les propriétés et les expressions des composants dans le template généré.
 - intercepter les interactions de l'utilisateur avec le template et les lier à des méthodes du composant associé,



Data binding

- ✓ Sans framework, vous seriez responsable d'insérer les valeurs de données dans les balises HTML et de transformer les réponses des utilisateurs en actions et mises à jour de valeur.
- ✓ Écrire une telle logique push / pull à la main est fastidieux, sujette aux erreurs et un cauchemar à lire, comme n'importe quel programmeur jQuery expérimenté peut en attester.
- ✓ Angular prend en charge la liaison de données bidirectionnelle (two-way data binding), un mécanisme permettant de coordonner des parties d'un modèle avec des parties d'un composant.
- ✓ Un balisage spécial ajouté au code HTML permet d'indiquer à Angular comment connecter les deux côtés.

Cycle de vie d'un composant

- ✓ Un composant a un cycle de vie géré par Angular.
- ✓ Angular le crée, le restitue, crée et restitue ses enfants, le vérifie lorsque ses propriétés liées aux données changent et le détruit avant de le retirer du DOM.
- ✓ Angular propose des hooks qui offrent une visibilité et une capacité d'agir sur les moments clés du cycle de vie.
- ✓ Après la création d'un composant par son constructeur, Angular appelle les hooks du cycle de vie, à des moments spécifiques, selon la séquence

Hooks	Objectif et timing
ngOnChanges()	Appelé avant ngOnInit () et chaque fois qu'une ou plusieurs propriétés d'entrée liées aux données changent.

Cycle de vie d'un composant

Hooks	Objectif et timing
ngOnInit()	Appelé une fois, après le premier ngOnChanges () .
ngDoCheck()	Appelé à chaque exécution de détection de changement, immédiatement après ngOnChanges () et ngOnInit () .
ngAfterContentInit()	Appelé une fois après le premier ngDoCheck () .
ngAfterContentChecked()	Appelé après ngAfterContentInit () et chaque ngDoCheck () suivant.
ngAfterViewInit()	Appelé une fois après le premier ngAfterContentChecked () .
ngAfterViewChecked()	Appelé après ngAfterViewInit et chaque ngAfterContentChecked () suivant.
ngOnDestroy()	Appelé juste avant qu'angular ne détruise la directive / le composant.

Communication directe entre composants (1)

Parent => Child via @Input

Dans le code du composant fils :

- ✓ Impoter la bibliothèque **Input** à partir de @angular/core

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

- ✓ Déclarer la variable cible avec @Input()

```
@Input() message: string;
```

Dans le template du composant parent :

- ✓ Passer le paramètre correspondant à la variable cible à l'aide d'un binding de propriété.

```
<app-child [message]="message"></app-child>
```

Communication directe entre composants (2)

Child=> Parent via @Onput et EventEmiter

Dans le code du composant fils :

- ✓ Impoter les bibliothèques **Output** et **EventEmiter** à partir de

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

- ✓ Déclarer un objet EventEmitter à l'aide de @Output()

```
@Output() messageEvent = new EventEmitter<string>();
```

- ✓ Emettre un évènement à laide de la méthode emit(...)

```
sendMessage() {
  this.messageEvent.emit(this.message)
}
```

Dans le template du composant parent :

- ✓ Ajouter un binding pour l'évènement émis.

```
<app-child (messageEvent)="receiveMessage($event)"></app-child>
```

Dans le code du composant parent

```
receiveMessage($event) {
  this.message = $event
}
```

Templates, pipes et directives

Les templates

Déclaration

- ✓ Le template peut être inclus dans les métadonnées du composant (une bonne solution si le template est petit) ou stocké dans un fichier séparé et référencé via la propriété **templateUrl**.
- ✓ L'exemple suivant indique au composant de rechercher le fichier League.html dans le dossier app/Views pour le code du template associé au composant :

```
@Component({  
    selector: 'main-app',  
    moduleId: module.id,  
    templateUrl: 'app/Views/League.html',  
})
```

Templates et vues

Le HTML

- ✓ Un template définit la vue associée à un composant.
- ✓ Les templates sont formés de code HTML, avec quelques améliorations pour pouvoir les utiliser avec les composants Angular.
- ✓ La plupart des éléments HTML peuvent être utilisés dans un template, à l'exception notable de la balise `<script>`, qui sera ignorée. De même, le code HTML du template ne doit pas contenir les balises : `<html>`, `<header>` et `<body>`.

```
<h2>Hero List</h2>
<p>
  <i>Pick a hero from the
list</i>
</p>
<ul>
  <li *ngFor="let hero of
heroes"
    (click)="selectHero(hero)">
    {{hero.name}} </li>
</ul>

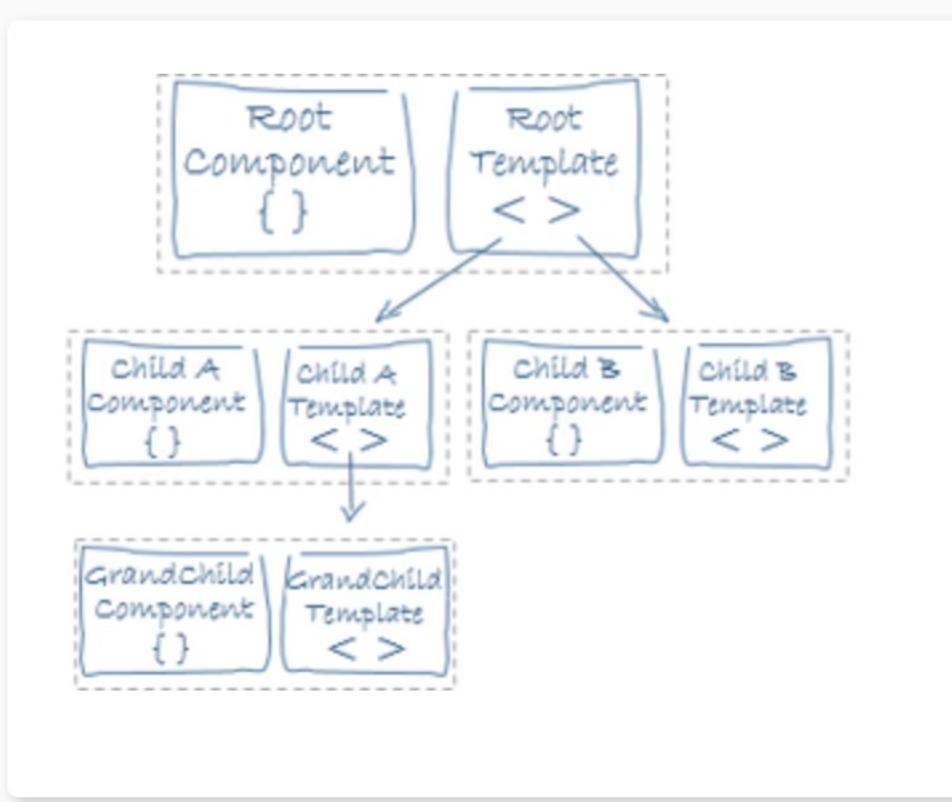
<app-hero-detail
*ngIf="selectedHero"
[hero]="selectedHero">
</app-hero-detail>
```

Exemple d'un template

Templates et vues

Le HTML

- ✓ Un template est souvent composé d'une hiérarchie de vues.
- ✓ Le template associé directement au composant est appelé host-view
- ✓ La hiérarchie de vues permet de modifier, d'afficher et de masquer des sections entières de l'interface utilisateur en tant qu'unité.
- ✓ Un composant peut intégrer des vues hébergées par d'autres composants.
- ✓ Une hiérarchie de vues peut inclure des vues provenant de composants dans le même NgModule, ou inclure des vues à partir de composants qui sont définis dans différents NgModules.



Directives

@Directive

- ✓ Les templates d'Angular sont dynamiques.
- ✓ Au moment de l'affichage, Angular transforme le DOM selon les instructions données par les directives.
- ✓ Une directive est une classe avec un décorateur @Directive.
- ✓ Un composant est techniquement une directive - mais les composants sont tellement distinctifs et centraux pour les applications qu'Angular définit le décorateur @Component, qui étend le décorateur @Directive avec des fonctionnalités orientées template.
- ✓ Tout comme pour les composants, les métadonnées d'une directive associent la classe à un sélecteur que vous utilisez pour l'insérer dans le code HTML.
- ✓ Dans les templates, les directives apparaissent généralement comme attributs.

Types de Directives

@Directive

- Il existe deux types de directives **en plus des composants: les directives structurelles et les directives d'attribut.**

- Directives structurelles :

```
<li *ngFor="let hero of heroes"></li>  
<app-hero-detail *ngIf="selectedHero"></app-hero-detail>
```

- Directive d'attribut :

```
<input [(ngModel)]="hero.name">
```

NgIf

- ✓ La directive ngIf est utilisée lorsque vous souhaitez afficher ou masquer un élément en fonction d'une condition.
- ✓ La condition est déterminée par le résultat de l'expression que vous transmettez dans la directive. Si le résultat de l'expression renvoie une valeur false, l'élément sera supprimé du DOM.
- ✓ Exemples :

```
<div *ngIf="false"></div>          <!-- never displayed -->
<div *ngIf="a > b"></div>          <!-- displayed if a is more than b -->
<div *ngIf="str == 'yes'"></div>    <!-- displayed if str is the string "yes" -->
<div *ngIf="myFunc()"></div>        <!-- displayed if myFunc returns truthy -->
```

NgSwitch

```
<div class="container">
  <div *ngIf="myVar == 'A'">Var is A</div>
  <div *ngIf="myVar == 'B'">Var is B</div>
  <div *ngIf="myVar != 'A' && myVar != 'B'">Var is something else</div>
</div>

<h4 class="ui horizontal divider header">
  Current choice is {{ choice }}
</h4>

<div class="ui raised segment">
  <ul [ngSwitch]="choice">
    <li *ngSwitchCase="1">First choice</li>
    <li *ngSwitchCase="2">Second choice</li>
    <li *ngSwitchCase="3">Third choice</li>
    <li *ngSwitchCase="4">Fourth choice</li>
    <li *ngSwitchCase="2">Second choice, again</li>
    <li *ngSwitchDefault>Default choice</li>
  </ul>
</div>
```

```
<div class="container" [ngSwitch]="myVar">
  <div *ngSwitchCase="'A'">Var is A</div>
  <div *ngSwitchCase="'B'">Var is B</div>
  <div *ngSwitchDefault>Var is something else</div>
</div>
```

NgStyle

- ✓ Avec la directive NgStyle, vous pouvez définir des propriétés CSS d'un élément DOM donné à partir d'expressions Angular.
- ✓ La façon la plus simple d'utiliser cette directive est de faire
[style. <Propriété css>] = "valeur"

✓ Exemples :

```
<div [style.background-color]="'yellow'">
  Uses fixed yellow background
</div>
```

```
<div [ngStyle]="{color: 'white', 'background-color': 'blue'}">
  Uses fixed white text on blue background
</div>
```

Valeur Statique

```
<div class="ui input">
  <input type="text" name="color" value="{{color}}" #colorinput>
</div>

<div class="ui input">
  <input type="text" name="fontSize" value="{{fontSize}}" #fontinput>
</div>

<button class="ui primary button" (click)="apply(colorinput.value, fontinput.value)">
  Apply settings
</button>
```

Valeur Dynamique

NgClass

- La directive NgClass, représentée par un attribut ngClass dans votre modèle HTML, vous permet de définir et de modifier dynamiquement les classes CSS pour un élément DOM donné.

Exemples :

```
.bordered {
  border: 1px dashed black;
  background-color: #eee; }
```

```
<div [ngClass]="{bordered: false}">This is never bordered</div>
<div [ngClass]="{bordered: true}">This is always bordered</div>
```

This is never bordered

This is always bordered

Utilisation simple

```
@Component({
  selector: 'app-ng-class-example',
  templateUrl: './ng-class-example.component.html'
})
export class NgClassExampleComponent implements OnInit {
  isBordered: boolean;
  classesObj: Object;
  classList: string[];

  constructor() {}

  ngOnInit() {
    this.isBordered = true;
    this.classList = ['blue', 'round'];
    this.toggleBorder();
  }

  toggleBorder(): void {
    this.isBordered = !this.isBordered;
    this.classesObj = {
      bordered: this.isBordered
    };
  }

<div [ngClass]="classesObj">
  Using object var. Border {{ classesObj.bordered ? "ON" : "OFF" }}
</div>
```

NgFor (1)

- ✓ Le rôle de cette directive est de répéter un élément DOM donné (ou une collection d'éléments DOM) et de passer un élément du tableau à chaque itération.
- ✓ La syntaxe est `*ngFor = "let item of items"`.
 - ✓ `let item` spécifie une variable qui récupère l'élément courant du tableau;
 - ✓ `items` fait référence à la collection d'éléments de votre contrôleur.

✓ `this.cities = ['Miami', 'Sao Paulo', 'New York'];`

Simple list of strings

```
<h4 class="ui horizontal divider header">
  Simple list of strings
</h4>
```

Miami

Sao Paulo

New York

```
<div class="ui list" *ngFor="let c of cities">
  <div class="item">{{ c }}</div>
</div>
```

Résultat

NgFor (2)

✓ Comment parcourir un tableau d'objets

```
this.people = [
  { name: 'Anderson', age: 35, city: 'Sao Paulo' },
  { name: 'John', age: 12, city: 'Miami' },
  { name: 'Peter', age: 22, city: 'New York' }
];
];
```

List of objects		
Name	Age	City
Anderson	35	Sao Paulo
John	12	Miami
Peter	22	New York

```
<h4 class="ui horizontal divider header">
  List of objects
</h4>
```

```
<table class="ui celled table">
  <thead>
    <tr>
      <th>Name</th>
      <th>Age</th>
      <th>City</th>
    </tr>
  </thead>
  <tr *ngFor="let p of people">
    <td>{{ p.name }}</td>
    <td>{{ p.age }}</td>
    <td>{{ p.city }}</td>
  </tr>
</table>
```

✓ Comment récupérer l'index de l'élément en cours ?

```
<div class="ui list" *ngFor="let c of cities; let num = index">
  <div class="item">{{ num+1 }} - {{ c }}</div>
</div>
```

NgNonBindable

- ✓ Nous utilisons ngNonBindable quand nous voulons dire à Angular de ne pas compiler ou lier une section particulière de notre page.
- ✓ Exemple :

```
<div class='ngNonBindableDemo'>
  <span class="bordered">{{ content }}</span>
  <span class="pre" ngNonBindable>
    &larr; This is what {{ content }} rendered
  </span>
</div>
```

Some text ← This is what {{ content }} rendered

Directives personnalisées

En plus des directives pré-définis par Angular, nous pouvons créer nos propres directives à l'aide du décorateur `@Directive`.

Créer une directive ressemble à la création d'un composant avec quelques différences :

- Les composants ont toutes les fonctionnalités des directives mais ont aussi une vue, c'est-à-dire du HTML qui est injecté dans le DOM lorsque nous l'utilisons.
- Un seul élément HTML ne peut être associé qu'à un seul composant. Cependant, un seul élément peut être associé à plusieurs directives.

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

```
...
```

```
@Directive({ selector: "[ccCardHover]" })
```

```
<div class="card card-block" ccCardHover>...</div>
```

```
class CardHoverDirective { ...}
```

Pipes prédéfinis (1)

Filtres prédéfinis

- ✓ L'opérateur de pipe (|) vous permet d'appliquer un modificateur qui peut contrôler la manière dont une propriété apparaît à l'écran.
- ✓ **Syntaxe**

```
 {{ property | pipe_name }}
```

- ✓ Parmi les filtres prédéfinis :
 - uppercase: convertit la valeur de la propriété en majuscule.
 - lowercase: convertit la valeur de la propriété en minuscules.
 - percent: Exprime la valeur décimale en pourcentage avec le signe%.
 - currency: convertie dans la devise spécifiée.
 - date: affiche la propriété sous la forme d'une chaîne de date.

Pipes prédéfinis (2)

Exemple avec les filtres de dates

```
<h4>New season starts on {{ SeasonStart | date:"fullDate" }}</h4>
```



Thursday, August 11, 2016

- Autres formats de dates :

- **medium** (Aug 25, 2016, 12:59:08 PM)
- **short** (8/25/2016, 12:59 PM)
- **fullDate** (Thursday, August 25, 2016)
- **longDate** (August 25, 2016)
- **mediumDate** (Aug 25, 2016)
- **shortDate** (8/25/2016)
- **mediumTime** (12:59:08 PM)
- **shortTime** (12:59 PM)

Pipes personnalisés

- ✓ Un pipe personnalisé est une classe décorée avec @Pipe.
- ✓ La classe du pipe implémente la méthode de transformation de l'interface PipeTransform qui accepte une valeur d'entrée suivie de paramètres facultatifs et renvoie la valeur transformée.
- ✓ Il y aura un argument supplémentaire à la méthode de transformation pour chaque paramètre passé au pipe.

<p>Super power boost: {{2 | exponentialStrength: 10}}</p>

```
import { Pipe, PipeTransform } from '@angular/core';
/*
 * Raise the value exponentially
 * Takes an exponent argument that defaults to 1.
 * Usage:
 *   value | exponentialStrength:exponent
 * Example:
 *   {{ 2 | exponentialStrength:10 }}
 * formats to: 1024
 */
@Pipe({name: 'exponentialStrength'})
export class ExponentialStrengthPipe implements PipeTransform {
  transform(value: number, exponent: string): number {
    let exp = parseFloat(exponent);
    return Math.pow(value, isNaN(exp) ? 1 : exp);
  }
}
```

Services et injection de dépendances

Services

- ✓ Le service englobe toute valeur, fonction ou fonctionnalité dont une application a besoin.
- ✓ Un service est généralement une classe avec un objectif spécifique et bien défini.



Objectif : rendre les classes de composants plus légères et plus efficaces pour assurer une meilleure expérience utilisateur. ➔ Le travail d'un composant doit se limiter à la liaison entre la vue et le modèle de l'application.

Les autres tâches (exemple : connexion au serveur) doivent être déléguées aux services.

Avantages :

- ✓ Réutiliser le code : services injectables dans plusieurs composants.
- ✓ Application plus adaptable : injecter différents fournisseurs du même type de service, selon les circonstances.
- ✓ Faciliter les tests.

Injection des dépendances



- ✓ Mécanisme permettant d'injecter des services, des fonctions ou des valeurs dans des composants, des services, des pipes ou des NgModules.
- ✓ Pour définir une classe en tant que service dans Angular, on utilise le décorateur **@Injectable** pour fournir les métadonnées qui permettent à Angular de l'injecter en tant que dépendance.
- ✓ De même, on utilise le décorateur **@Injectable** pour indiquer qu'un composant ou une autre classe (tel qu'un autre service, un pipe ou un NgModule) a une dépendance.
- ✓ L'injection des dépendance est une fonctionnalité de base dans Angular.
- ✓ L'injecteur est le mécanisme principal. Angular crée un injecteur à l'échelle de l'application pour vous pendant le processus d'amorçage.
- ✓ L'injecteur gère un conteneur d'instances de dépendance qu'il a déjà créées et les réutilise si possible.
- ✓ Un provider est nécessaire pour créer une dépendance. Pour un service, il s'agit généralement de la classe de service elle-même.

Fonctionnement de l'Injection des dépendances

- ✓ Lorsque Angular crée une nouvelle instance d'une classe de composants, il détermine les services ou autres dépendances dont ce composant a besoin en examinant les types de ses paramètres constructeurs.

```
constructor(private service: HeroService) { }
```

- ✓ Si Angular découvre qu'un composant dépend d'un service, il vérifie d'abord si l'injecteur a déjà des instances existantes de ce service.
- ✓ Si une instance de service demandée n'existe pas encore, l'injecteur crée une instance en utilisant le provider enregistré et l'ajoute à l'injecteur avant de renvoyer le service à Angular.
- ✓ Quand tous les services demandés ont été résolus et renvoyés, Angular peut appeler le constructeur du composant avec ces services en tant qu'arguments.

Fournir des services

- ✓ Vous devez enregistrer au moins un provider de tout service que vous allez utiliser. Vous pouvez enregistrer des providers dans des modules ou dans des composants.

Enregister un provider dans un module :

Si le provider est enregistré au module root, la même instance du service est disponible pour tous les composants de l'application.

src/app/app.module.ts (module providers)

```
providers: [
  BackendService,
  HeroService,
  Logger
],
```

Enregister un provider dans un composant:

Si le provider est enregistré au niveau du composant, une nouvelle instance du service est créée pour chaque instance du composant.

src/app/hero-list.component.ts (component providers)

```
@Component({
  selector: 'app-hero-list',
  templateUrl: './hero-list.component.html',
  providers: [ HeroService ]
})
```

Routage et navigation

Le routing

Les composants du Angular Routing :

- ✓ Il y a trois composants principaux que nous utilisons pour configurer le routage dans Angular:
 - ✓ **Routes** décrit les routes prises en charge par notre application
 - ✓ **RouterOutlet** est un composant "placeholder" qui indique à Angular où placer le contenu de chaque route
 - ✓ La directive **RouterLink** est utilisée pour lier aux routes

Imports :

- ✓ Afin d'utiliser le routeur dans Angular, nous importons des constantes du package **@angular/router**:

```
import {  
    RouterModule,  
    Routes  
} from '@angular/router';  
  
app.module.ts
```

Configurer les routes

```
const routes: Routes = [
  // basic routes
  { path: '', redirectTo: 'home', pathMatch: 'full' },
  { path: 'home', component: HomeComponent },
  { path: 'about', component: AboutComponent },
  { path: 'contact', component: ContactComponent },
  { path: 'contactus', redirectTo: 'contact' },

  // authentication demo
  { path: 'login', component: LoginComponent },
  {
    path: 'protected',
    component: ProtectedComponent,
    canActivate: [ LoggedInGuard ]
  },

  // nested
  {
    path: 'products',
    component: ProductsComponent,
    children: childRoutes
  }
];
```

app.module.ts

Remarques :

- **path** spécifie l'URL que cette route va gérer
- **component** est ce qui lie un chemin d'accès donné à un composant qui gérera l'itinéraire
- le **redirectTo** facultatif est utilisé pour rediriger un chemin donné vers un itinéraire existant

Installer les routes

1. Importer RouterModule

```
imports: [
  BrowserModule,
  FormsModule,
  HttpModule,
  RouterModule.forRoot(routes), // <-- routes
  // added this for our child module
  ProductsModule
],
```

app.module.ts

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  constructor(private router: Router) {
  }
}
```

app.component.ts

2. RouterOutlet :

<router-outlet> indique où le contenu du composant de route sera

```
<div class="page-header">
  <div class="container">
    <h1>Router Sample</h1>
    <div class="navLinks">
      <a [routerLink]="/home">Home</a>
      <a [routerLink]="/about">About Us</a>
      <a [routerLink]="/contact">Contact Us</a>
      |
      <a [routerLink]="/products">Products</a>
      <a [routerLink]="/login">Login</a>
      <a [routerLink]="/protected">Protected</a>
    </div>
  </div>
</div>

<div id="content">
  <div class="container">
    <router-outlet></router-outlet>
  </div>
</div>
```

app.component.html

3. RouterLink :

```
<h1>Router Sample</h1>
<div class="navLinks">
  <a [routerLink]="/home">Home</a>
  <a [routerLink]="/about">About Us</a>
  <a [routerLink]="/contact">Contact Us</a>
```

Affecter un style différent au menu actif

- ✓ Affecter la classe css à l'attribut **routerLinkActive**.
- ✓ Pour les URLs qui peuvent faire partie d'URLs plus longues, on peut forcer l'affectation du style uniquement lorsque l'URL exacte est sélectionnée à l'aide de **[routerLinkActiveOptions]**

```
<li role="presentation"
    routerLinkActive="active"
    [routerLinkActiveOptions]="{exact: true}">
    <a routerLink="/">Home</a>
</li>
```

Naviguer programmatiquement

- ✓ Import Router from @angular/Router

```
import { Router } from '@angular/router';
```

- ✓ Inject router in constructor

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

- ✓ Utiliser la méthode navigate du router pour aller vers une nouvelle URL :

```
this.router.navigate(['/servers']);
```

Stratégie de routing

- ✓ La façon dont l'application Angular analyse et crée des chemins depuis et vers les définitions de routes est appelée stratégie de routing.
- ✓ La stratégie par défaut est **PathLocationStrategy**, ce que nous appelons le routage HTML5. En utilisant cette stratégie, les routes sont représentées par des chemins réguliers, comme /home ou /contact.
- ✓ Nous pouvons changer la stratégie de localisation utilisée pour notre application en liant la classe **LocationStrategy** à une nouvelle classe de stratégie concrète.
- ✓ Au lieu d'utiliser la **PathLocationStrategy** par défaut, nous pouvons également utiliser **HashLocationStrategy**.
- ✓ La stratégie de hachage peut être choisie pour des raisons de compatibilité avec les anciennes versions de navigateurs (chemins avec #, ancrés).

Passer des paramètres dans la route

- ✓ Pour ajouter un paramètre à la configuration de notre routeur, nous spécifions le chemin de l'itinéraire comme ceci:

```
const routes: Routes = [
  { path: 'product/:id', component: ProductComponent },
];
```

- ✓ Pour utiliser les paramètres de route, nous devons d'abord importer **ActivatedRoute**:

```
import { ActivatedRoute } from '@angular/router';
```

- ✓ Ensuite, nous injectons l'ActivatedRoute dans le constructeur de notre composant. Par exemple, disons que nous avons un itinéraire qui spécifie ce qui suit:

```
const routes: Routes = [
  { path: 'product/:id', component: ProductComponent }
];
```

- ✓ Enfin, nous ajoutons l'ActivatedRoute comme l'un des paramètres du constructeur:

```
export class ProductComponent {
  id: string;

  constructor(private route: ActivatedRoute) {
    route.params.subscribe(params => { this.id = params['id']; });
  }
}
```

Protéger des URLs avec canActivate Guard (1)

```
export class AuthService {
  loggedIn = false;

  isAuthenticated() {
    const promise = new Promise(
      (resolve, reject) => {
        setTimeout(() => {
          resolve(this.loggedIn);
        }, 800);
      }
    );
    return promise;
  }

  login() {
    this.loggedIn = true;
  }

  logout() {
    this.loggedIn = false;
  }
}
```

Fake auth service

```
import {
  CanActivate,
  ActivatedRouteSnapshot,
  RouterStateSnapshot,
  Router,
  CanActivateChild
} from '@angular/router';
import { Observable } from 'rxjs/Observable';
import { Injectable } from '@angular/core';
import { AuthService } from './auth.service';

@Injectable()
export class AuthGuard implements CanActivate, CanActivateChild {
  constructor(private authService: AuthService, private router: Router) {}

  canActivate(route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean {
    return this.authService.isAuthenticated()
      .then(
        (authenticated: boolean) => {
          if (authenticated) {
            return true;
          } else {
            this.router.navigate(['/']);
          }
        }
      );
  }
}
```

Auth guard service

Protéger des URLs avec canActivate Guard (2)

```
const appRoutes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'users', component: UsersComponent, children: [
    { path: ':id/:name', component: UserComponent }
  ] },
  {
    path: 'servers',
    canActivate: [AuthGuard],
    component: ServersComponent,
  },
  // { path: 'not-found', component: PageNotFoundComponent },
  { path: 'not-found', component: ErrorPageComponent, data: {message: 'Page not found!'} },
  { path: '**', redirectTo: '/not-found' }
];
```

Ajouter le paramètre **canActivate** en spécifiant le **Guard** qui se chargera de vérifier les accès.

Note : ne pas oublier d'injecter les nouveaux services (définis dans le slide précédent dans

Les formulaires

Les formulaires

- ✓ Les formulaires sont probablement l'aspect le plus crucial de votre application Web.
- ✓ Pami les fonctionnalités d'un formulaire :
 - ✓ Les entrées de formulaire sont destinées à modifier les données, à la fois sur la page et sur le serveur
 - ✓ Les changements doivent souvent être reflétés ailleurs sur la page
 - ✓ Les utilisateurs ont beaucoup de marge de manœuvre dans ce qu'ils entrent, vous devez donc valider les valeurs.
 - ✓ L'interface utilisateur doit clairement indiquer les attentes et les erreurs, le cas échéant.
 - ✓ Les champs dépendants peuvent avoir une logique complexe
 - ✓ Nous voulons être en mesure de tester nos formulaires, sans compter sur les sélecteurs DOM

Angular et les formulaires

- ✓ Parmi les principaux outils offerts par Angular pour gérer les formulaires :
 - ✓ **FormControls** : encapsule les entrées dans nos formulaires et nous donne des objets pour travailler avec eux
 - ✓ **Les validateurs (validators)** nous donnent la possibilité de valider les entrées, comme nous le souhaiterions
 - ✓ **Les observateurs (observables)** nous permettent de rester à l'écoute des changements effectués sur notre formulaire et de réagir en conséquence

FormControls and FormGroups (1)

- ✓ Les deux objets fondamentaux dans les formulaires Angular sont **FormControl** et **FormGroup**:

✓ FormControl

- ✓ Un **FormControl** représente un champ d'entrée unique - c'est la plus petite unité d'un formulaire Angular.
- ✓ **FormControls** encapsule la valeur du champ et les états tels que : être valide (valid), modifié (dirty) ou a des erreurs.

✓ Exemple :

```
// create a new FormControl with the value "Nate"
let nameControl = new FormControl("Nate");
```

```
let name = nameControl.value; // -> Nate
```

```
// now we can query this control for certain values:
nameControl.errors // -> StringMap<string, any> of errors
nameControl.dirty // -> false
nameControl.valid // -> true
// etc.
```

```
<!-- part of some bigger form -->
<input type="text" [formControl]="name" />
```

FormControls and FormGroups (2)

● FormGroup

- ✓ La plupart des formulaires ont plus d'un champ, nous avons donc besoin d'un moyen de gérer plusieurs FormControls.
- ✓ Si nous voulions vérifier la validité de notre formulaire, il est fastidieux d'itérer sur un tableau de FormControls et de vérifier la validité de chaque FormControl.
- ✓ **FormGroup** résout ce problème en fournissant une interface wrapper autour d'une collection de FormControls.

● Exemple :

```
let personInfo = new FormGroup({  
  firstName: new FormControl("Nate"),  
  lastName: new FormControl("Murray"),  
  zip: new FormControl("90210")  
})
```

```
personInfo.value; // -> {  
  //   firstName: "Nate",  
  //   lastName: "Murray",  
  //   zip: "90210"  
  // }  
  
// now we can query this control group for certain values, which have sensible  
// values depending on the children FormControl's values:  
personInfo.errors // -> StringMap<string, any> of errors  
personInfo.dirty // -> false  
personInfo.valid // -> true  
// etc.
```

Deux approches

- **Template-driven approche** : Le formulaire est créé dans le template du composant puis Angular génère les objets du formulaire à partir du DOM.
- **Reactive approche** : Le formulaire est créé dans le code du composant puis synchronisé avec le DOM.

Démonstration

Username

[Suggest an Username](#)

Mail

Secret Questions

Your first teacher?



Your reply:

- male
- female

[Submit](#)

Programmation réactive

Rappel : programmation asynchrone

Callbacks

```
function printString(string, callback){  
    setTimeout(  
        () => {  
            console.log(string)  
            callback()  
        },  
        Math.floor(Math.random() * 100) + 1  
    )  
}  
  
function printAll(){  
    printString("A", () => {  
        printString("B", () => {  
            printString("C", () => {})  
        })  
    })  
}  
  
printAll()
```

Promise

```
function printString(string){  
    return new Promise((resolve, reject) => {  
        setTimeout(  
            () => {  
                console.log(string)  
                resolve()  
            },  
            Math.floor(Math.random() * 100) + 1  
        )  
    })  
}  
  
function printAll(){  
    printString("A")  
    .then(() => {  
        return printString("B")  
    })  
    .then(() => {  
        return printString("C")  
    })  
}
```

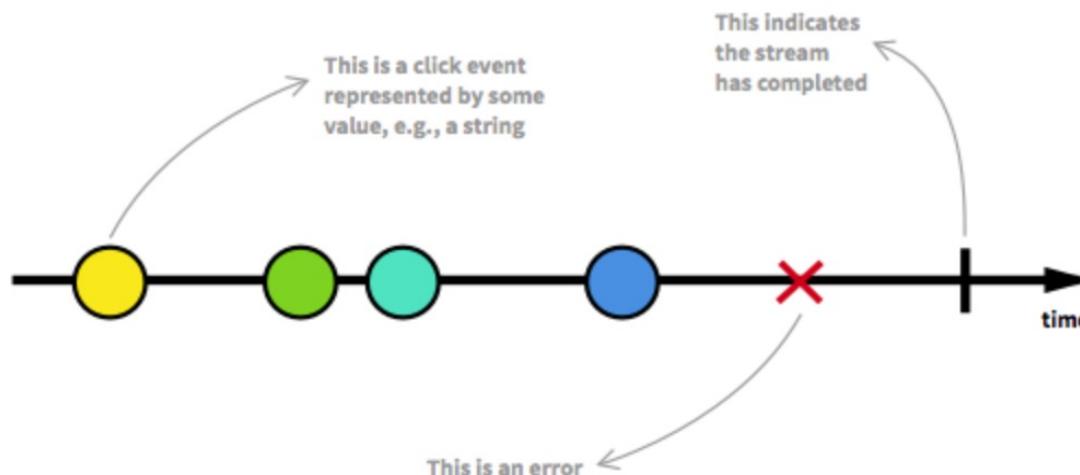
Async/Await

```
function printString(string){  
    return new Promise((resolve, reject) => {  
        setTimeout(  
            () => {  
                console.log(string)  
                resolve()  
            },  
            Math.floor(Math.random() * 100) + 1  
        )  
    })  
}  
  
async function printAll(){  
    await printString("A")  
    await printString("B")  
    await printString("C")  
}
```

```
printAll()
```

Qu'est-ce que la programmation réactive ?

- La programmation réactive est la programmation avec des flux de données (data streams) asynchrones.
- Un flux (stream) est une séquence d'événements en cours classés dans le temps.
- En plus de cela, vous disposez d'une incroyable boîte à outils de fonctions pour combiner, créer et filtrer n'importe lequel de ces flux.



```
--a----b-c---d---X---|-->
```

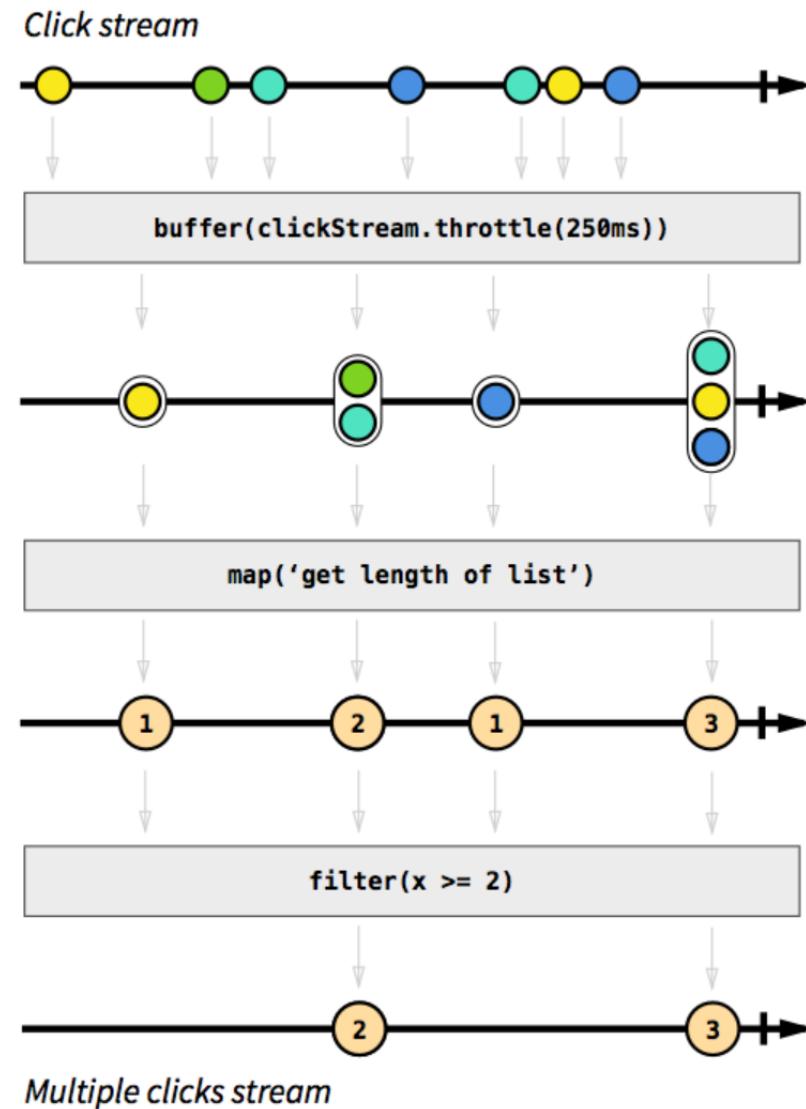
a, b, c, d are emitted values
X is an error
| is the 'completed' signal
---> is the timeline

Comment ça marche ?

- ✓ Nous capturons ces événements émis uniquement de manière asynchrone, en définissant une fonction qui s'exécutera lorsqu'une valeur est émise, une autre fonction lorsqu'une erreur est émise et une autre fonction lorsque « completed » est émise.
- ✓ Parfois, ces deux derniers peuvent être omis et vous pouvez simplement vous concentrer sur la définition de la fonction des valeurs.
- ✓ L'écoute du flux s'appelle l'abonnement (subscribe). Les fonctions que nous définissons sont des observateurs (observer). Le flux est le sujet observé (observable).
- ✓ C'est précisément l'Observer Design Pattern.

Exemple

nous allons créer de nouveaux flux d'événements de clic transformés à partir du flux d'événements de clic d'origine.



RxJS : la bibliothèque javascript pour la programmation réactive (1)

- ✓ RxJS est l'une des bibliothèques les plus en vogue dans le développement Web aujourd'hui.
- ✓ Offrant une approche puissante et fonctionnelle pour gérer les événements et les points d'intégration dans un nombre croissant de frameworks, de bibliothèques et d'utilitaires, les arguments en faveur de l'apprentissage de Rx n'ont jamais été aussi attrayants.
- ✓ Ajoutez à cela la possibilité d'utiliser vos connaissances dans presque tous les langages, une solide compréhension de la programmation réactive et de ce qu'elle peut offrir semble être une évidence.
- ✓ Cependant, apprendre RxJS et la programmation réactive est difficile. Il y a la multitude de concepts, la grande surface d'API et le changement fondamental de mentalité d'un style impératif à un style déclaratif.

Source : <https://www.learnrxjs.io/>

RxJS : la bibliothèque javascript pour la programmation réactive (2)

- ✓ RxJS est l'une des bibliothèques les plus en vogue dans le développement Web aujourd'hui.
- ✓ Offrant une approche puissante et fonctionnelle pour gérer les événements et les points d'intégration dans un nombre croissant de frameworks, de bibliothèques et d'utilitaires, les arguments en faveur de l'apprentissage de Rx n'ont jamais été aussi attrayants.
- ✓ Ajoutez à cela la possibilité d'utiliser vos connaissances dans presque tous les langages, une solide compréhension de la programmation réactive et de ce qu'elle peut offrir semble être une évidence.

Communication avec le serveur

Http et le module HttpClient pour communiquer avec un serveur

- ✓ 99% des projets Angular impliquent une communication entre un client (un navigateur) et un serveur distant.
- ✓ Normalement, cela se fait avec HTTP. Il est donc très important de savoir comment fonctionne la communication HTTP et comment vous pouvez écrire du code pour cela.
- ✓ Le protocole HTTP (HyperText Transfer Protocol) est conçu pour permettre les communications entre les clients et les serveurs.
- ✓ HTTP fonctionne comme un protocole de requête-réponse entre un client et un serveur.
- ✓ Les méthodes HTTP sont utilisées depuis longtemps dans les applications Web traditionnelles côté serveur et dans les applications Web AJAX côté client :
- ✓ Les méthodes HTTP les plus couramment utilisées sont les suivantes:
 - POST
 - GET

Http header

- ✓ Les en-têtes HTTP permettent au client et au serveur de transmettre des informations supplémentaires avec les requêtes ou les réponses.
- ✓ Une en-tête de requête HTTP est composée de son nom (insensible à la casse), suivi de deux points (:), suivi de sa valeur (sans saut de ligne).

Name	Headers	Preview	Response	Timing
font-check-blue.77286...				
analytics.js	General	Request URL: https://www.google-analytics.com/analytics.js Request Method: GET Status Code: 200 (from disk cache) Remote Address: [2607:f8b0:4002:807::200e]:443 Referrer Policy: no-referrer-when-downgrade		
fontawesome-webfont...				
OpenSans-Light-webf...				
OpenSans-Regular-we...				

Http header d'une requête

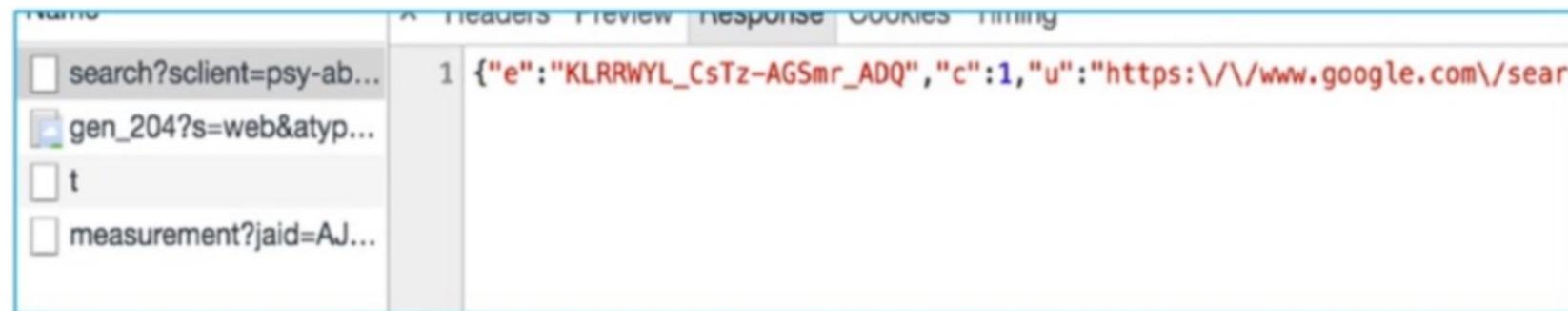
Response Headers
age: 1109
alt-svc: quic=":443"; ma=2592000; v="39,38,37,36,35"
cache-control: public, max-age=7200
content-encoding: gzip
content-length: 12343
content-type: text/javascript
date: Tue, 27 Jun 2017 00:44:41 GMT
expires: Tue, 27 Jun 2017 02:44:41 GMT
last-modified: Tue, 06 Jun 2017 00:25:39 GMT
server: Golfe2

Http header d'une réponse

Ces informations sont accessibles à partir de l'onglet network des outils du développeur intégrés au navigateur (F12).

Http body

- ✓ Le corps http permet au client et au serveur de transmettre des informations supplémentaires à la requête ou à la réponse après l'en-tête.
- ✓ Les corps HTTP ne sont pas toujours nécessaires car un ensemble d'informations n'est pas toujours nécessaire.
- ✓ Par exemple, les requêtes Http « Get" n'ont pas besoin d'inclure des informations dans le corps - toutes les informations sont déjà contenues dans l'en-tête.



Exemple de corps d'une requête HTTP

Passer des informations avec Http

- ✓ Il existe différentes manières de transmettre des informations du navigateur au serveur.

Query Parameters

- ✓ Exemple : <http://localhost:4200/sockjs-node/info?t=1498649243238>.

Matrix Parameters

- ✓ Exemple : <http://localhost:4200/sockjs-node/info;t=1498649243238>

Path Parameters

- ✓ Exemple : <http://localhost:4200/api/badges/9243238>

Passing Data in the Request Body

- ✓ Le client HTTP d'Angular vous permet de transmettre des informations au serveur dans le corps de la requête en utilisant la méthode POST du client HTTP.
- ✓ Vous pouvez transmettre plus de données dans le corps de la requête qu'en les transmettant dans l'URL.

REST

- ✓ Une application RESTful est une application serveur qui expose son état et ses fonctionnalités sous la forme d'un ensemble de ressources que les clients (navigateurs) peuvent manipuler. Exemple de ressources : une liste de clients ou leurs commandes.
- ✓ Toutes les ressources sont adressables de manière unique, généralement via des URI, bien que d'autres adresses puissent être utilisées. Par exemple, vous pouvez utiliser l'URL orders/23 pour accéder au numéro de commande 23.
- ✓ Toutes les ressources peuvent être manipulées via un ensemble contraint d'actions connues, généralement CRUD (créer, lire, mettre à jour, supprimer), représentés le plus souvent par les méthodes HTTP POST, GET, PUT et DELETE.
- ✓ Exemple: vous pouvez utiliser un HTTP DELETE pour orders/23 pour supprimer cette commande.
- ✓ Les données de toutes les ressources sont transférées via un nombre limité

Angular Http Client

- ✓ Le client HTTP d'Angular est un service que vous pouvez injecter dans vos classes pour effectuer une communication HTTP avec un serveur.
- ✓ Ce service est disponible via le nouveau module Http Client d'Angular 5 @angular/common/http, qui remplace l'ancien module Http d'Angular 4 @angular/common/http.
- ✓ Vous devrez modifier votre classe de module (celle de votre projet) pour importer ce module:

```
@NgModule({  
  imports: [  
    ...  
    HttpClientModule,  
    ...  
  ],  
  declarations: [ AppComponent ],  
  bootstrap: [ AppComponent ]  
})
```

Vous pouvez injecter le service HTTP directement dans vos composants de cette manière

```
@Injectable()  
class CustomerComponent {  
  ...  
  constructor(private http: HttpClient) {  
    ...  
  }  
}
```

Angular Http Client (bonnes pratiques)

- ✓ La solution vue précédemment est conseillée pour le prototypage, mais ce n'est pas conseillé pour la maintenabilité du code à long terme.
- ✓ Vous ne devriez pas utiliser HttpClient directement pour accéder aux données en dehors d'une classe de service.
- ✓ Vous devez écrire les classes de service qui utilisent le client HTTP, puis injecter ces classes dans votre code où vous avez besoin d'un accès aux données.
- ✓ Si vous consultez la documentation officielle Angular sur angular.io, vous verrez ce qui suit: *This is a golden rule: always delegate data access to a supporting service class.*
- ✓ Voici un exemple de classe de service utilisant le HttpClient:

```
@Injectable()  
class CustomerCommunicationService {  
  ...  
  constructor(private http: HttpClient) {  
    ...  
  }  
}
```

```
class CustomerComponent {  
  ...  
  constructor(private http: CustomerCommunicationService) {  
    ...  
    // perform data access  
  }  
}
```

Les Generics

- ✓ Avec Angular 5, le nouveau HttpClientModule nous permet d'utiliser des génériques lorsque nous appelons des requêtes HTTP.
- ✓ Les génériques nous permettent d'indiquer à Angular le type de réponse que nous attendons à recevoir de la requête HTTP.
- ✓ Le type de réponse peut être un "any" (pour autoriser tout type de réponse), un type de variable (par exemple un String), une classe ou une interface.
- ✓ Par exemple, le code ci-dessous effectue un "get" http, spécifiant la réponse attendue comme étant un tableau d'objets Language:

```
this._http.get<Array<Language>>('https://languagetool.org/api/v2/languages');
```
- ✓ Cela permet à Angular d'analyser la réponse pour nous afin que nous n'ayons pas à le faire.
- ✓ Plus besoin d'appeler JSON.parse pour convertir la chaîne de réponse en objet.

Opérations asynchrones

- ✓ En JavaScript, faire des requêtes HTTP est une opération asynchrone.
- ✓ Il envoie la requête HTTP à l'API et n'attend pas de réponse avant de poursuivre avec la ligne de code suivante.
- ✓ Lorsque l'API répond en millisecondes ou en secondes ou minutes plus tard, nous sommes avertis et nous pouvons commencer à traiter la réponse.
- ✓ Dans Angular, il existe deux manières de gérer ces opérations asynchrones: nous pouvons utiliser des promesses (Promise) ou des observables.
- ✓ Nous faisons des appels à nos classes de service de support, et ils renvoient le résultat asynchrone, que nous traitons dans le composant.

Aperçu de l'architecture des données sous Angular

- ✓ La gestion des données peut être l'un des aspects les plus délicats de l'écriture d'une application maintenable.
- ✓ Plusieurs façons permettent d'obtenir des données dans votre application:
 - ✓ Requêtes HTTP AJAX
 - ✓ Websockets
 - ✓ Indexdb
 - ✓ LocalStockage
 - ✓ Service workers
 - ✓ etc.

Aperçu de l'architecture des données sous Angular

- ✓ Le problème de l'architecture de données aborde des questions telles que:
 - ✓ Comment pouvons-nous agréger toutes ces différentes sources dans un système cohérent?
 - ✓ Comment pouvons-nous éviter les bogues causés par des effets secondaires involontaires?
 - ✓ Comment pouvons-nous structurer le code de façon judicieuse pour qu'il soit plus facile de maintenir et intégrer les nouveaux membres de l'équipe?
 - ✓ Comment pouvons-nous rendre l'application aussi rapide que possible lorsque les données changent?

Aperçu de l'architecture des données sous Angular

- Pendant de nombreuses années, MVC était un modèle standard pour l'architecture des données dans les applications:
 - ✓ les modèles contenaient la logique du domaine,
 - ✓ la vue affichait les données et
 - ✓ le contrôleur les reliait tous ensemble.
- Le problème est que nous avons appris que MVC ne se traduit pas bien dans les applications Web côté client.
- Il y a eu une renaissance dans le domaine des architectures de données et beaucoup de nouvelles idées sont explorées. Par exemple:
 - ✓ MVW / Two-way data binding
 - ✓ Flux-Redux-NgRx
 - ✓ Observables

Aperçu général des tests sous Angular

- ✓ Tout comme son prédecesseur AngularJs 1, Angular a été conçu avec la testabilité comme objectif principal.
- ✓ Quand nous parlons de tests dans Angular, nous parlons généralement de deux types de tests différents:
- ✓ **Tests unitaires**
 - ✓ Ceci est parfois appelé aussi test isolé. C'est la pratique de tester de petits morceaux de code isolés. Si votre test utilise une ressource externe, comme le réseau ou une base de données, ce n'est pas un test unitaire.

Aperçu général des tests sous Angular

● Tests fonctionnels

- Ceci est défini comme le test de la fonctionnalité complète d'une application. En pratique, avec les applications Web, cela signifie interagir avec votre application car elle fonctionne dans un navigateur, tout comme un utilisateur interagirait avec elle dans la vie réelle, c'est-à-dire via des clics sur une page. Ceci est également appelé **Test de bout en bout** ou **E2E (End to end)**.

Les tests unitaires avec Jasmine & Karma (1)

- ✓ Nous pouvons tester nos applications Angular de toutes pièces en écrivant et en exécutant des fonctions javascript pures.
- ✓ Créer des instances des classes pertinentes, appeler des fonctions et vérifier le résultat réel par rapport au résultat attendu.
- ✓ Mais puisque testing est une activité si courante avec javascript, il existe un certain nombre de bibliothèques de tests et de frameworks que nous pouvons utiliser pour réduire le temps nécessaire pour écrire des tests.
- ✓ Deux outils et frameworks de ce type qui sont utilisés lors des tests Angular sont **Jasmine** et **Karma**.

Les tests unitaires avec Jasmine & Karma (2)

- ✓ **Jasmine** : est un framework de test javascript qui prend en charge une pratique de développement logiciel appelée **Behaviour Driven Development**, ou **BDD** en abrégé. C'est une saveur spécifique du **développement piloté par les tests (TDD)**.
- ✓ Jasmine, et BDD en général, tente de décrire les tests dans un format lisible par l'homme afin que les personnes non techniques puissent comprendre ce qui est testé.
- ✓ La lecture des tests techniques au format BDD permet de comprendre plus facilement le code d'une application et facilite l'intégration de nouveaux membres dans l'équipe.

```
function helloWorld() {  
    return 'Hello world!';  
}
```

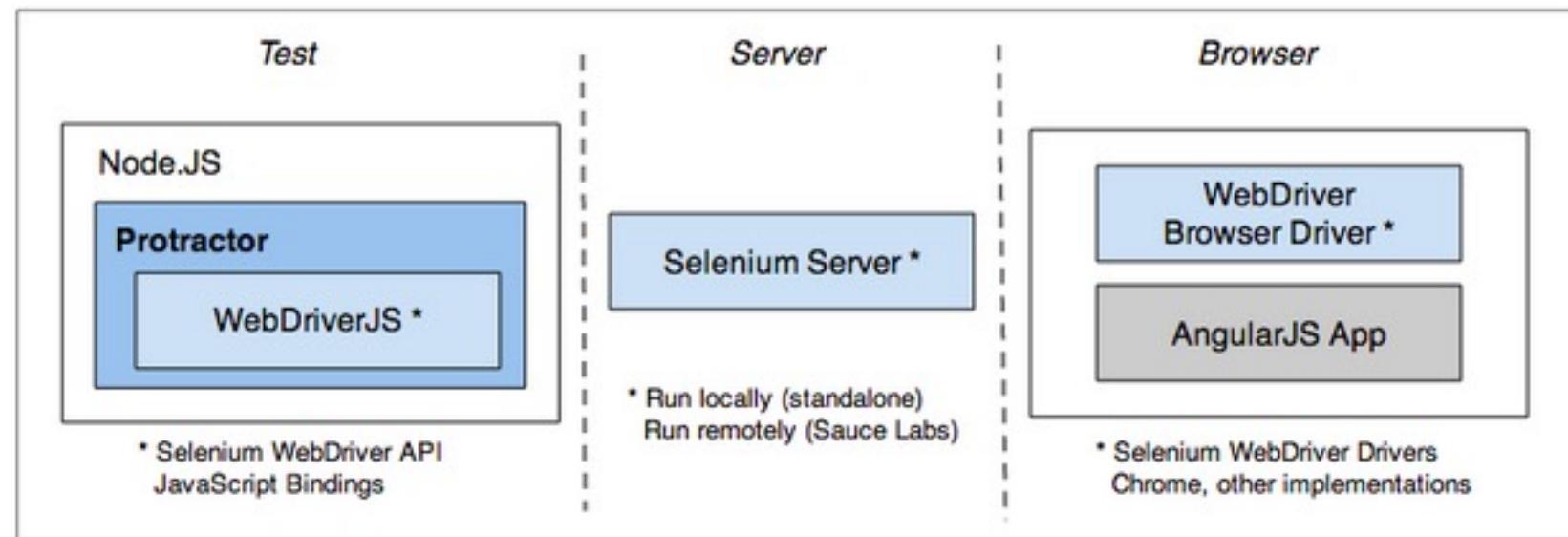
```
describe('Hello world', () => { ①  
    it('says hello', () => { ②  
        expect(helloWorld()) ③  
            .toEqual('Hello world!'); ④  
    });  
});
```

Les Built-in matchers de Jasmin

- ✓ expect(array).toContain(member);
 - ✓ expect(fn).toThrow(string);
 - ✓ expect(fn).toThrowError(string);
 - ✓ expect(instance).toBe(instance);
 - ✓ expect(mixed).toBeDefined();
 - ✓ expect(mixed).toBeFalsy();
 - ✓ expect(mixed).toBeNull();
 - ✓ expect(mixed).toBeTruthy();
 - ✓ expect(mixed).toBeUndefined();
 - ✓ expect(mixed).toEqual(mixed);
 - ✓ expect(mixed).toMatch(pattern);
 - ✓ expect(number).toBeCloseTo(number, decimalPlaces);
 - ✓ expect(number).toBeGreaterThanOrEqual(number);
 - ✓ expect(number).toBeLessThanOrEqual(number);
 - ✓ expect(number).toBeNaN();
 - ✓ expect(spy).toHaveBeenCalled();
 - ✓ expect(spy).toHaveBeenCalledTimes(number);
 - ✓ expect(spy).toHaveBeenCalledWith(...arguments);
- Pour des exemples concrets voir les docs de Jasmine ici:
http://jasmine.github.io/edge/introduction.html#section-Included_Matchers

Les tests fonctionnels avec Protractor (1)

- ✓ **Protractor** est une bibliothèque officielle à utiliser pour écrire des suites de tests E2E avec une application Angular. Ce n'est rien d'autre qu'une enveloppe au-dessus de l'Api WebDriverJS de Selenium qui traduit son code succinct et ses méthodes aux méthodes WebDriver JS. Cela dit, vous pouvez utiliser les méthodes WebDriverJS aussi dans votre script e2e.



Déploiement d'une application Angular

● Déploiement simple :

- Lancer la commande : **ng build**
- Copier le contenu du dossier **/dist**
- Si vous copiez les fichiers dans un sous-dossier de serveur, ajoutez l'indicateur de construction, --base-href et définissez <base href> de manière appropriée. Par exemple, si index.html est sur le serveur à /my/app/index.html, définissez la base href sur <base href = "/my/app/"> comme ceci : **ng build --base-href=/my/app/**
- Configurez le serveur pour rediriger les demandes de fichiers manquants vers index.html.

● Optimisation pour la version prod :

- Lancer la commande : **ng build --prod**

Déploiement d'une application Angular

✓ Déploiement simple :

- ✓ Lancer la commande : **ng build**
- ✓ Copier le contenu du dossier **/dist**
- ✓ Si vous copiez les fichiers dans un sous-dossier de serveur, ajoutez l'indicateur de construction, --base-href et définissez <base href> de manière appropriée. Par exemple, si index.html est sur le serveur à /my/app/index.html, définissez la base href sur <base href = "/my/app/"> comme ceci : **ng build --base-href=/my/app/**
- ✓ Configurez le serveur pour rediriger les demandes de fichiers manquants vers index.html.

✓ Optimisation pour la version prod :

- ✓ Lancer la commande : **ng build --prod**
- ✓ Pour réduire encore la taille du bundle, on peut utiliser :
ng build --prod --build-optimizer

Pour aller plus loin

- ✓ **Ebooks :**

Angular 2 Succently By Joseph D, Booth. Syncfusion Inc. 2017
Ng-book – The complete book on Angular 5.

- ✓ **Formation Vidéo :**

Udemy – **Angular - The Complete Guide (2023 Edition)**. By Maximilian Schwarzmüller.

- ✓ **Documentation officielle :**

<https://angular.io/docs>