



ANGULAR 7

Durée : 5 jours



Modules

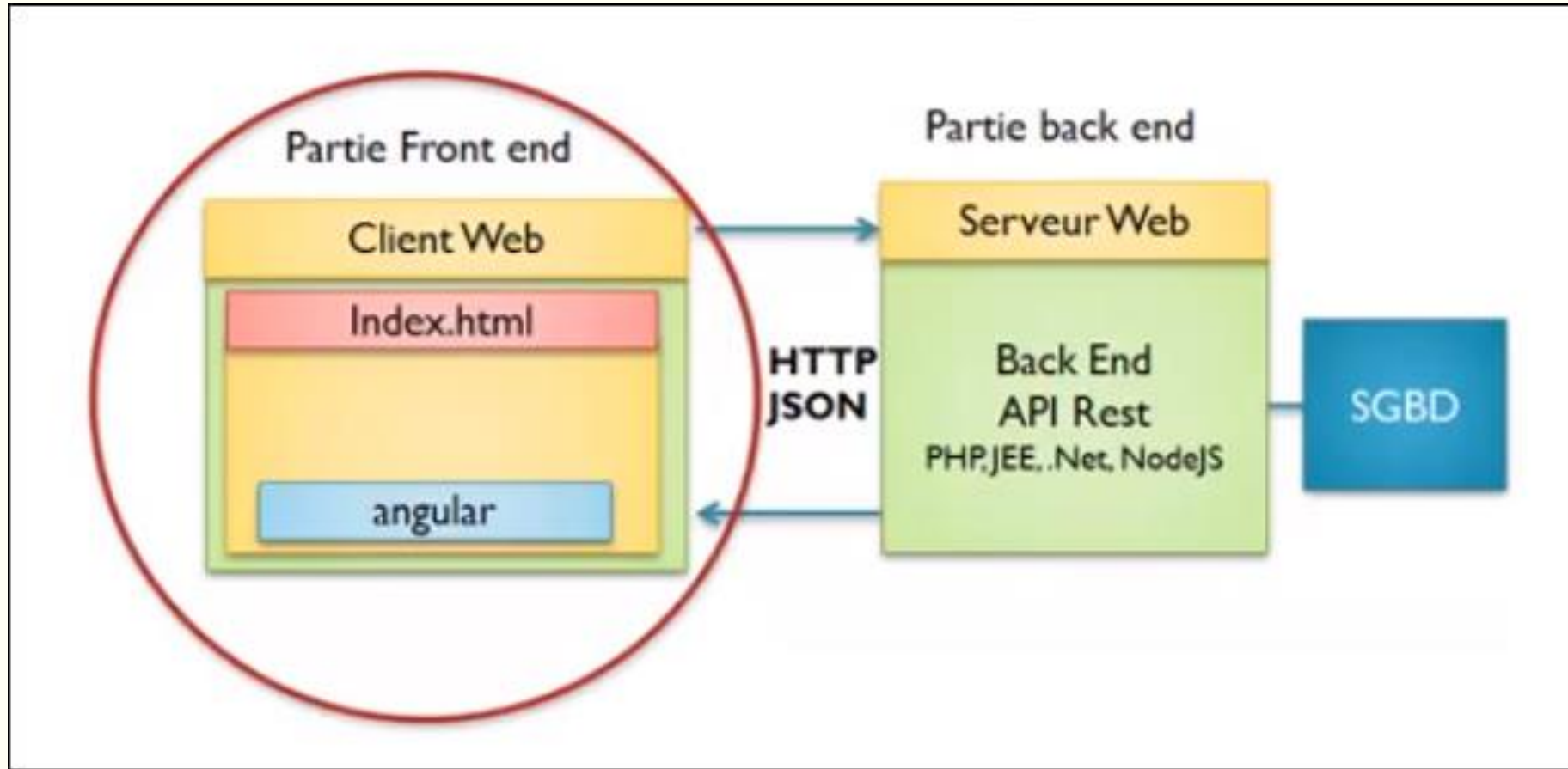
- **Module 1 : Introduction**
- **Module 2 : Ma première application**
- **Module 3 : Fondamentaux d'Angular**
- **Module 4 : TypeScript**
- **Module 5 : Les composants**
- **Module 6 : Pipe**
- **Module 7 : Directive**
- **Module 8 : Formulaire**
- **Module 9 : Service**
- **Module 10 : Http**
- **Module 11 : Routes**
- **Module 12 : Ionic**



MODULE 1

Introduction

Introduction



Introduction

- Angular est un framework permettant de créer des applications clientes web.
 - Plateforme de développement permettant de créer des applications web et mobiles.
 - Aujourd'hui, la version d'Angular est la 7.
-
- Ce framework s'appuie sur plusieurs principes présentés dans les sections suivantes.
 1. Organisation par composants
 2. TypeScript
 3. Les spécifications ES6
 4. DOM Virtuel

Introduction

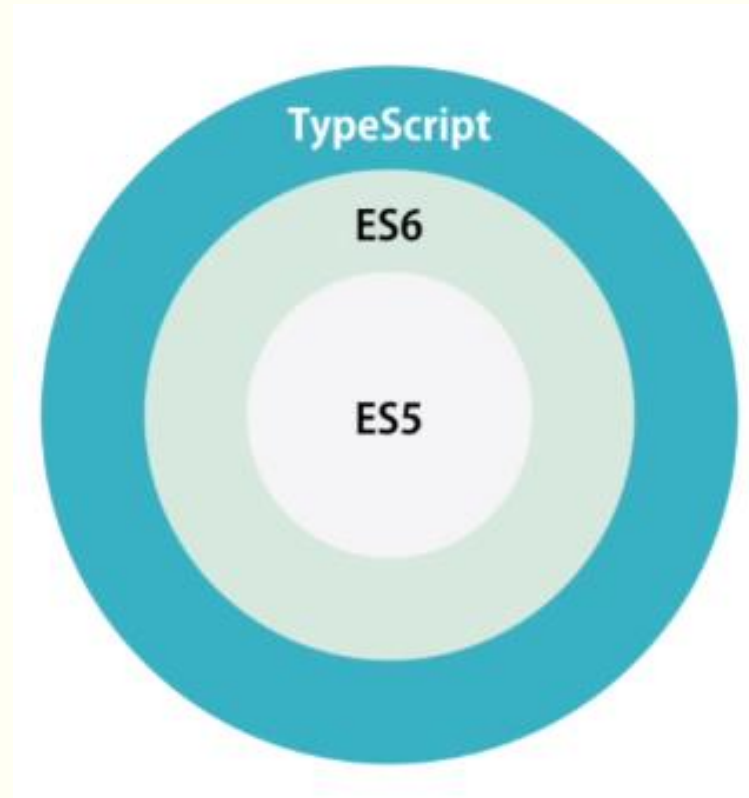
Organisation par composants

- L'organisation d'une application Angular se fait par composants.
- Un composant correspond à un élément réutilisable, indépendant et responsable d'une seule action métier.
- De cette manière, une application sera faite de l'assemblage d'un ensemble de composants.
- Cela permet une meilleure organisation, une meilleure testabilité et donc une meilleure maintenabilité.

Introduction

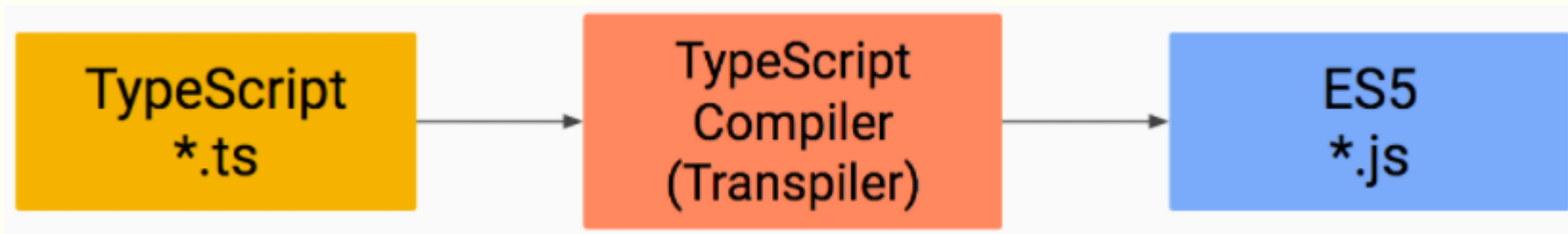
TypeScript

- Angular est écrit en TypeScript et il est conseillé de faire son développement d'applications également en TypeScript.
- TypeScript. Inventé par Microsoft
- Sur-ensemble de JavaScript ajoutant
 - ✓ Les types
 - ✓ Valeurs énumérées (enum)
 - ✓ Return types
 - ✓ Interfaces
 - ✓ Paramètre optionnel
 - ✓ Classes



Introduction

TypeScript



Introduction

Les spécifications ES6

- **EcmaScript 6**, la spécification JavaScript en cours de support par les navigateurs, apporte un ensemble d'éléments :
 - mot-clé let,
 - template de chaîne de caractères,
 - paramètres par défaut,
 - gestion des modules, etc.
- Angular a basé son architecture sur ces spécifications.
- C'est notamment le cas pour l'utilisation des modules ES6, à ne pas confondre avec les modules Angular, qui permettent de déclarer des éléments puis de les importer.

Introduction

DOM Virtuel

- Le **DOM** représente la structure d'une page HTML sous la forme d'un arbre
- L'une des tâches les plus impactantes en termes de performances dans le développement *front* est la manipulation du **DOM**.
- Chaque opération visant à interagir, que ce soit en lecture ou en écriture, avec les nœuds HTML est coûteuse et doit être minimisée au maximum.
- La librairie React a introduit une nouvelle façon de gérer cette problématique : le **DOM Virtuel**. Angular utilise ce mécanisme.

Introduction

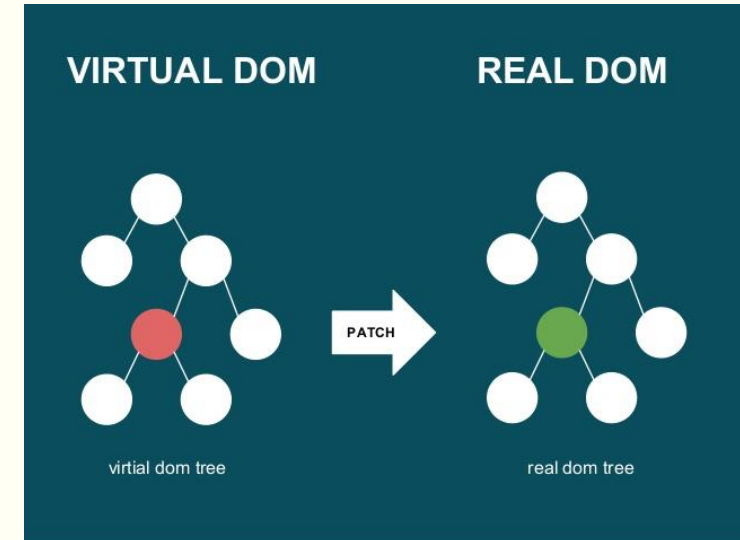
DOM Virtuel

- L'idée du DOM Virtuel est d'avoir une représentation en mémoire du DOM.
- Chaque modification, au lieu d'être effectuée sur le DOM physique, sera effectuée sur le DOM Virtuel, donc de manière beaucoup plus performante.

Le framework Angular s'occupera ensuite de synchroniser les modifications effectuées sur le DOM Virtuel vers le DOM physique.

Les modifications peuvent ainsi être appliquées de manière différentielle, en une seule fois.

Le gain de performance est majeur.



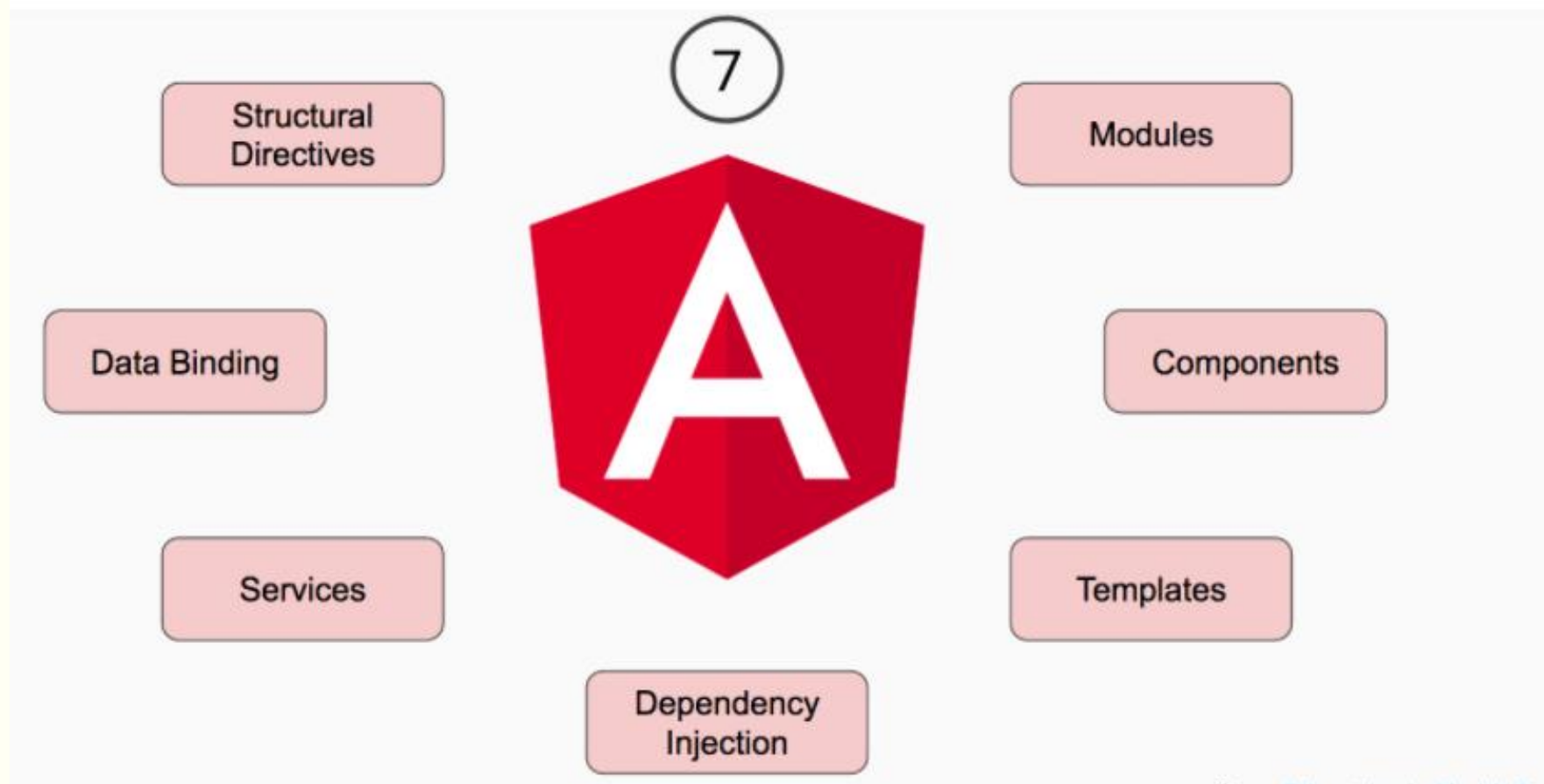
Introduction

Rendu côté serveur possible

- Depuis l'arrivée de Node.js, il est possible d'exécuter du code JavaScript côté serveur, c'est-à-dire en dehors du navigateur.
- Grâce à NodeJS, combiné à la fonctionnalité de DOM Virtuel, Angular peut être exécuté côté serveur afin de renvoyer une vue déjà pré-calculée.

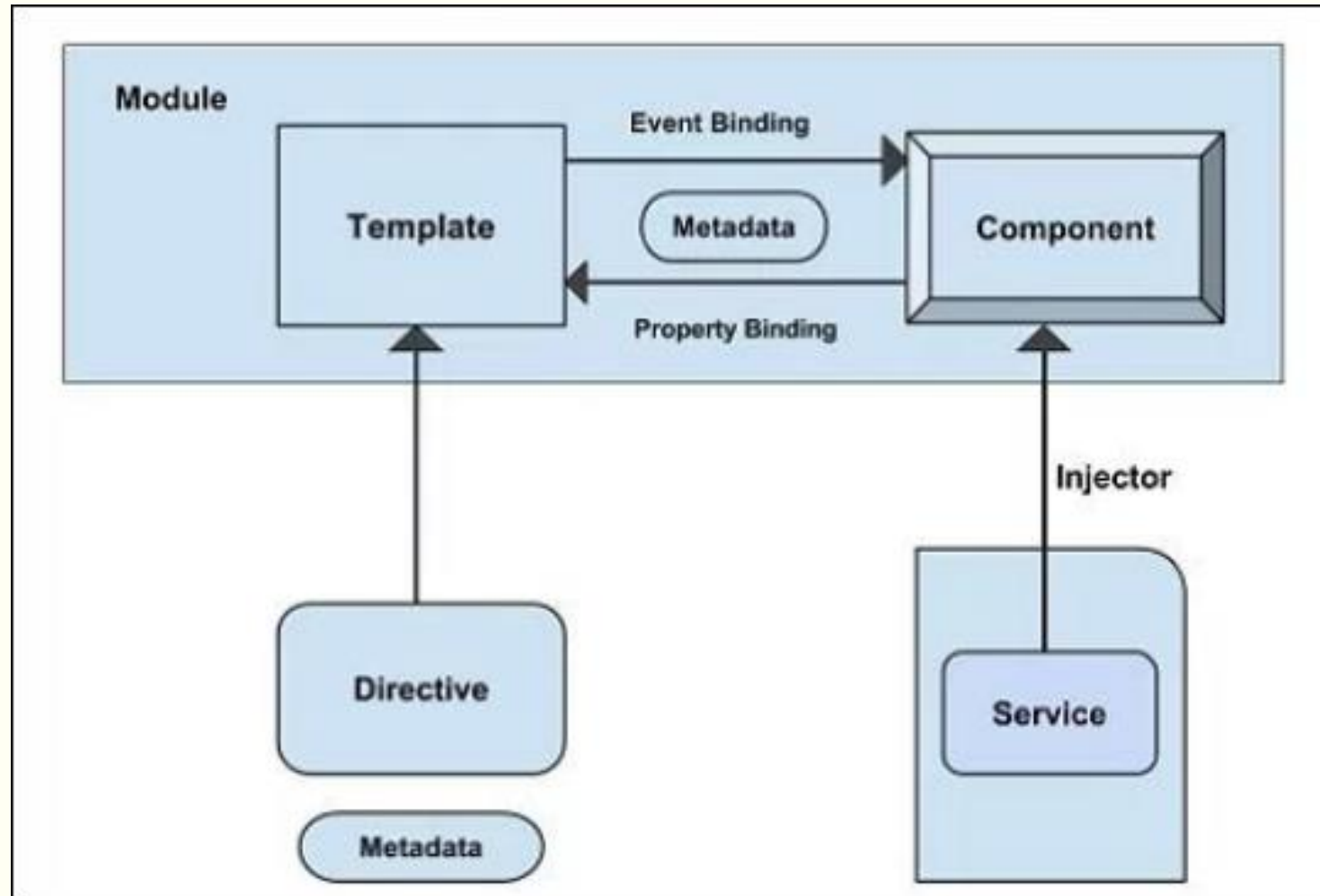
Introduction

7 Briques



Introduction

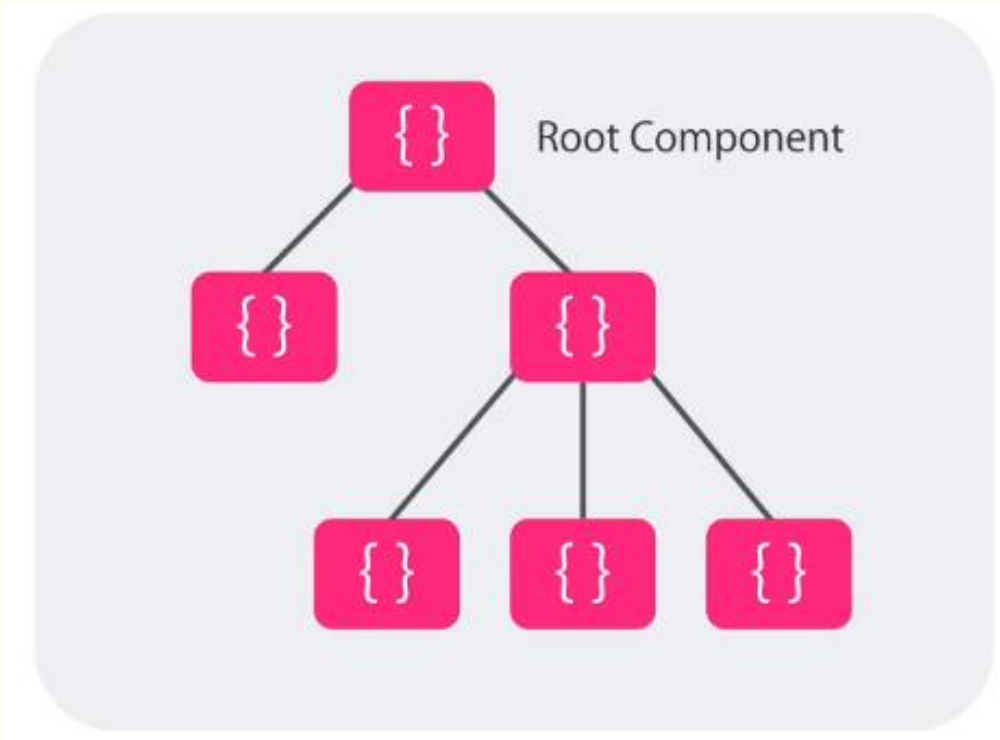
Architecture



Introduction

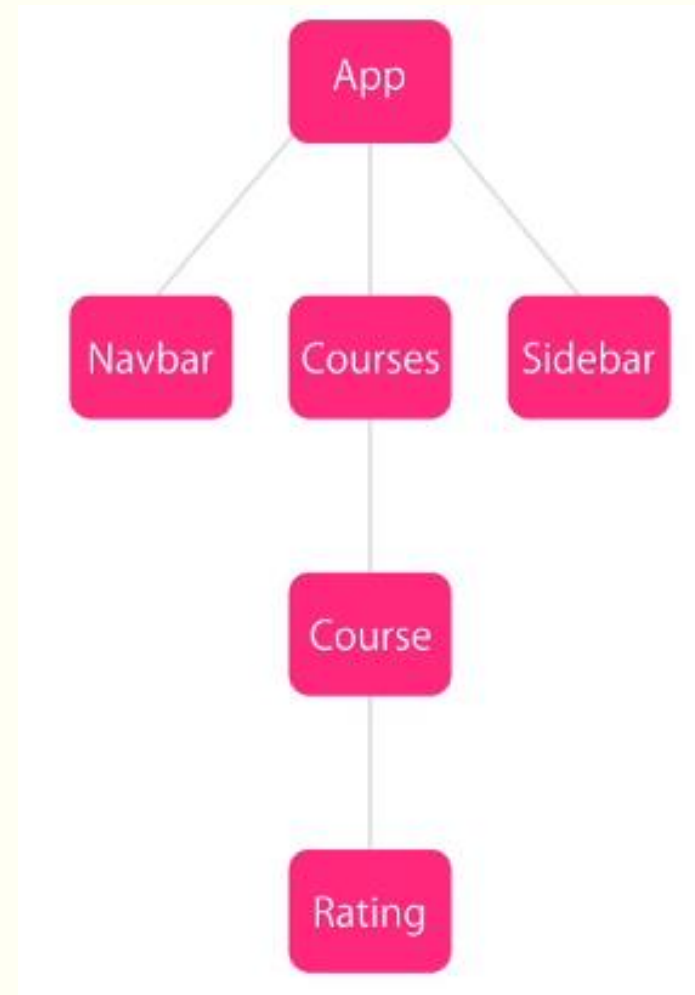
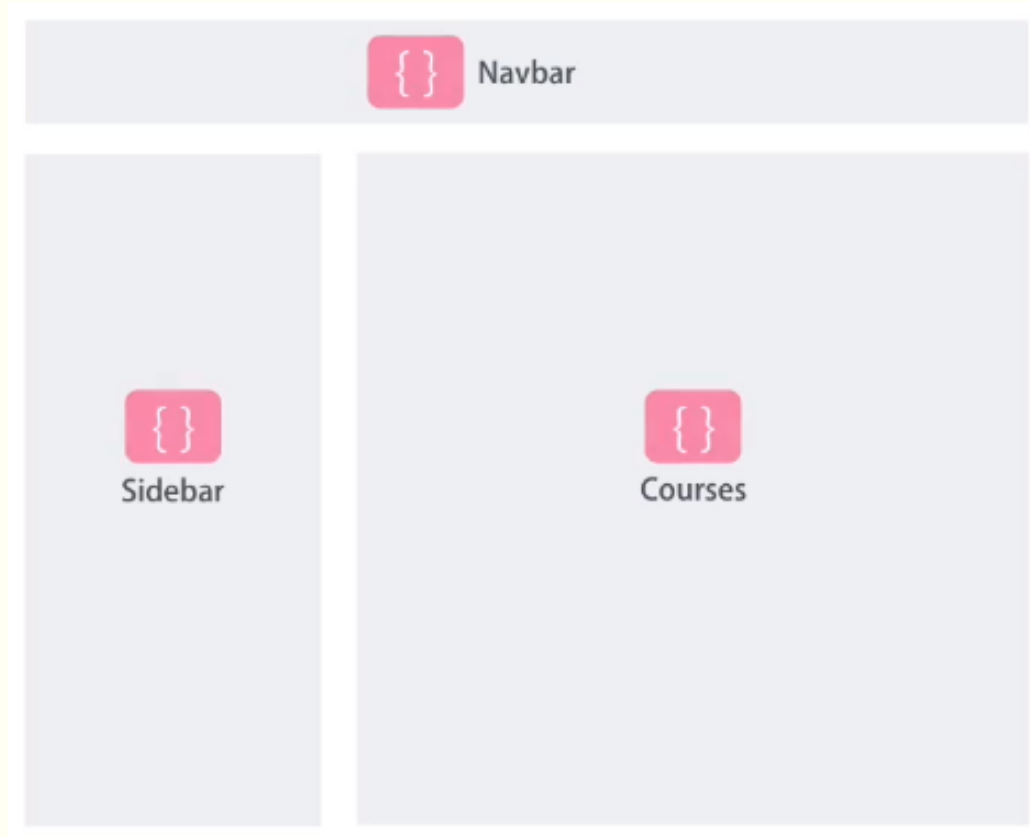
Composant

Dans Angular 7, un composant est un élément réutilisable de l'application, constitué d'une vue et d'un ensemble de traitements associés à cette vue.



Introduction

Exemple



Introduction

Service

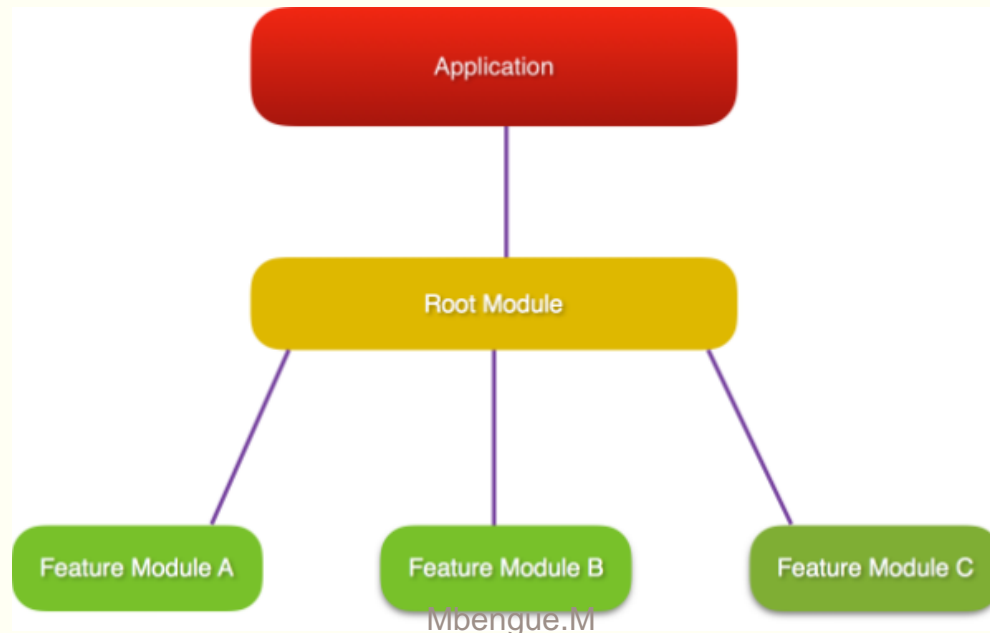
L'objectif d'un service est de contenir toute la logique fonctionnelle et/ou technique de l'application.



Introduction

Les modules

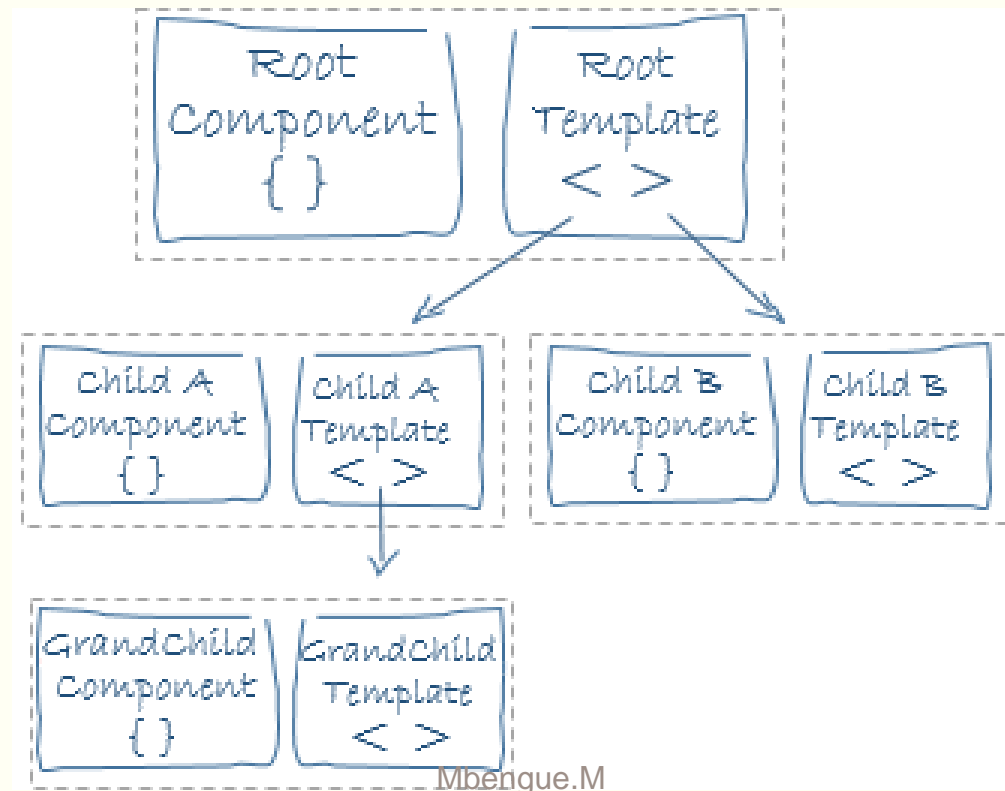
- Un module permet d'organiser une application en blocs fonctionnels ou techniques.
- Il est également le point d'entrée d'une application Angular.
- De la même manière qu'un composant, un module se déclare via une classe TypeScript.



Introduction

Les templates

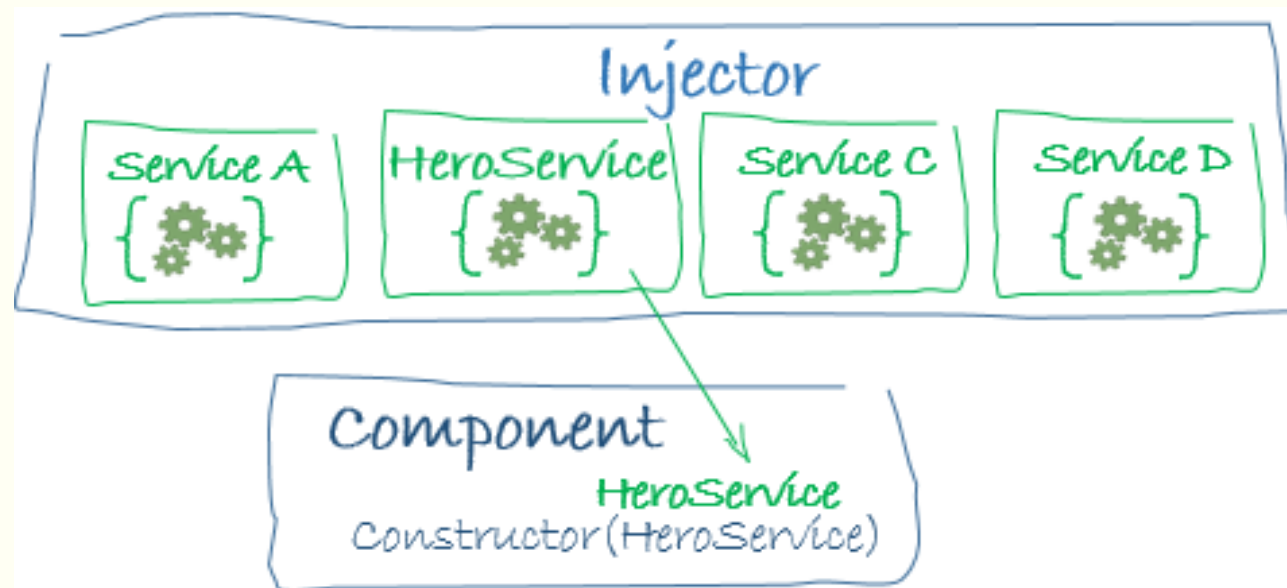
- Un template est du HTML représentant une vue. Cette vue décrit à Angular comment doit être rendu le composant auquel il est associé.



Introduction

L'injection de dépendances

- L'injection de dépendances est une implémentation du pattern IoC (*Inversion of Control* ou inversion des contrôles, en français). Le principe de ce pattern est de casser les dépendances entre les éléments.

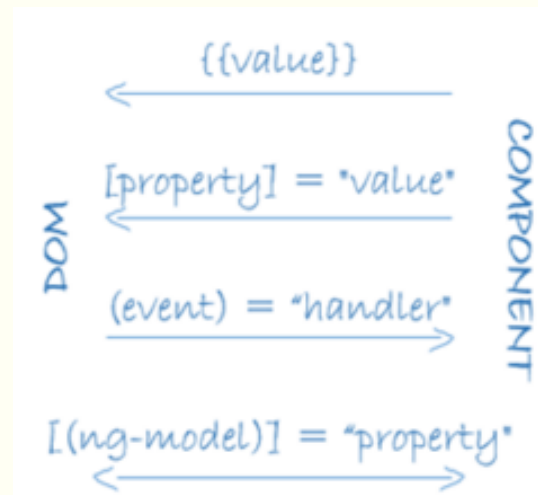


Introduction

Data binding

Permet de lier deux flux de données :

- Le premier flux est celui des données de l'interface, en somme les champs de saisie manipulés par l'utilisateur.
- Le second concerne les données de l'application en elle-même, c'est-à-dire les différentes propriétés que le développeur manipule/utilise dans le code.

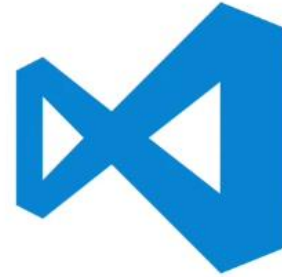


Introduction

Mise en place



nodejs.org



code.visualstudio.com



google.com/chrome



POSTMAN

Introduction

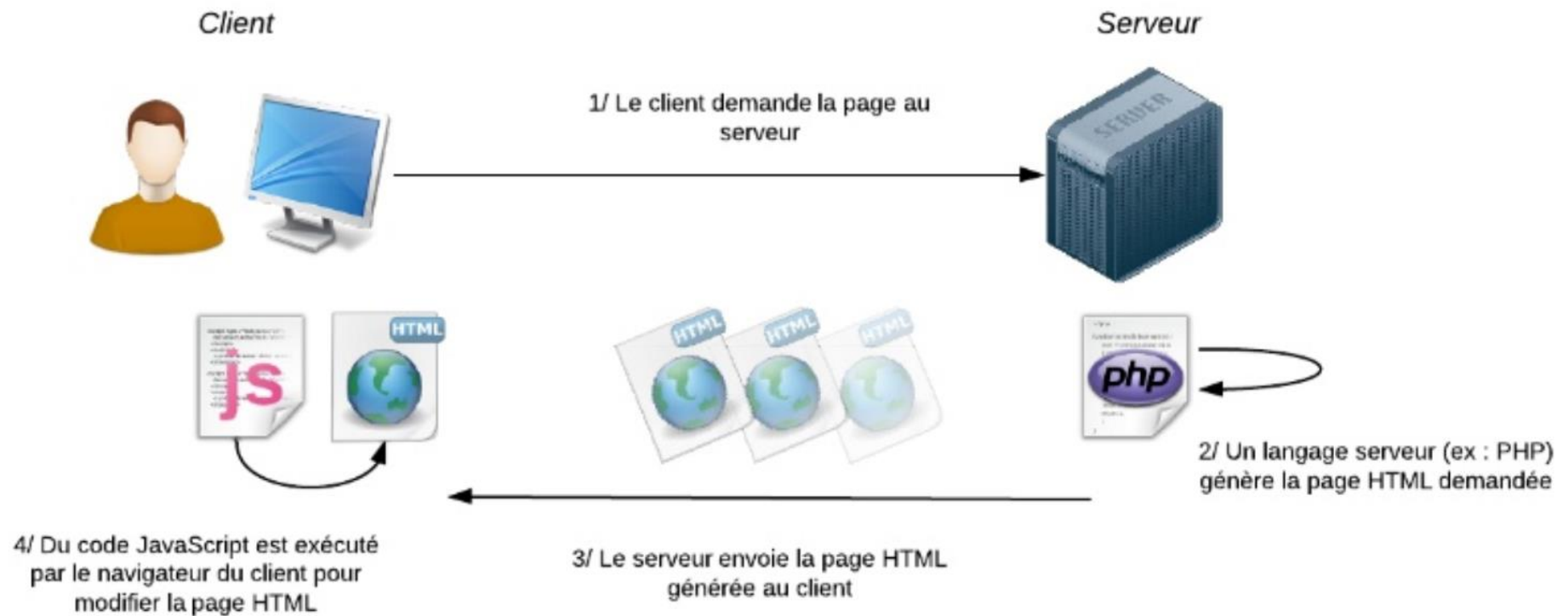
NodeJS

- environnement d'exécution de code JS
- applications serveur et réseau (mais aussi scripting général)
- créé par Ryan Dahl et première release le 27 mai 2009
- dernière version stable : 8.9.2
- dernière version : 9.2.0

Introduction

NodeJS

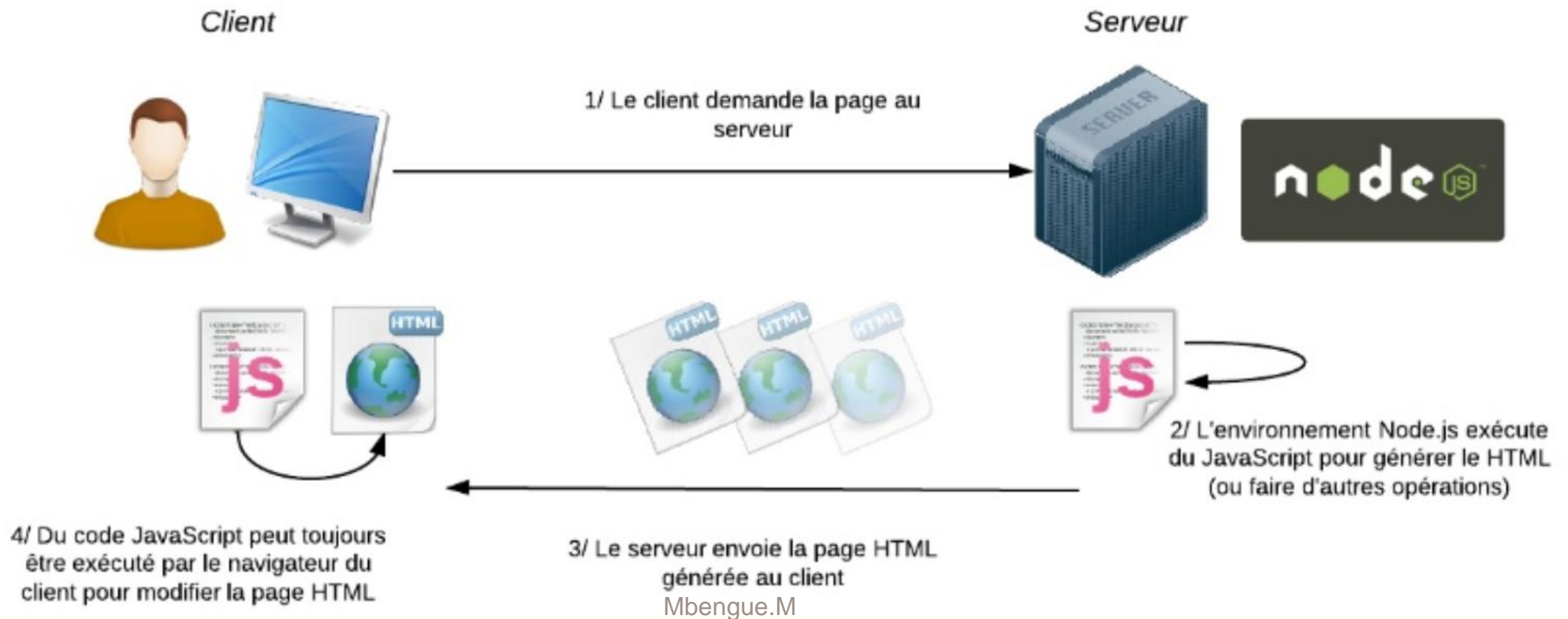
Javascript au début est utilisé coté client



Introduction

NodeJS

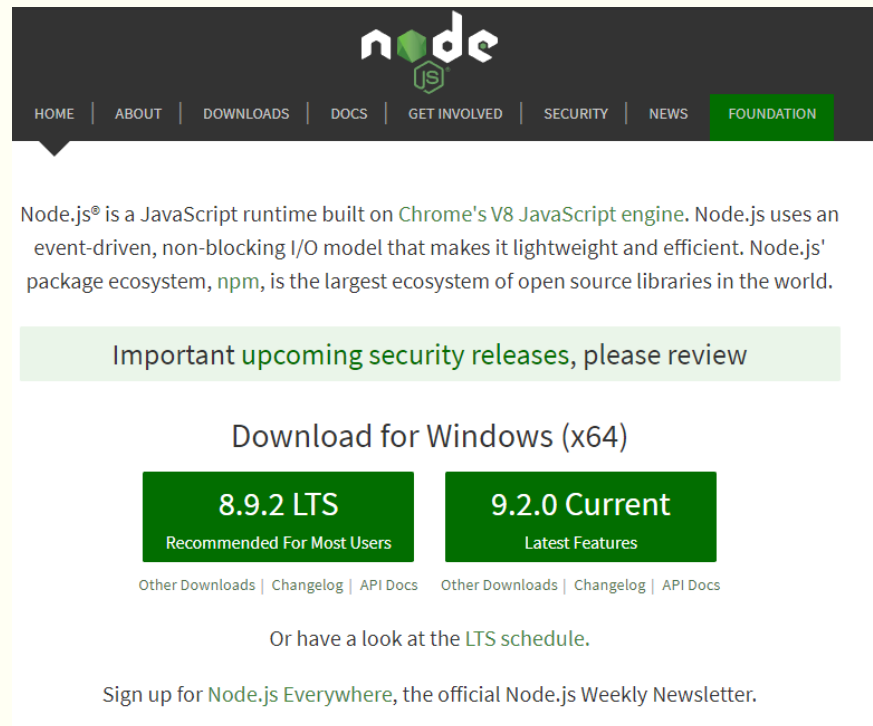
Maintenant javascript est exécuté coté serveur aussi



Introduction

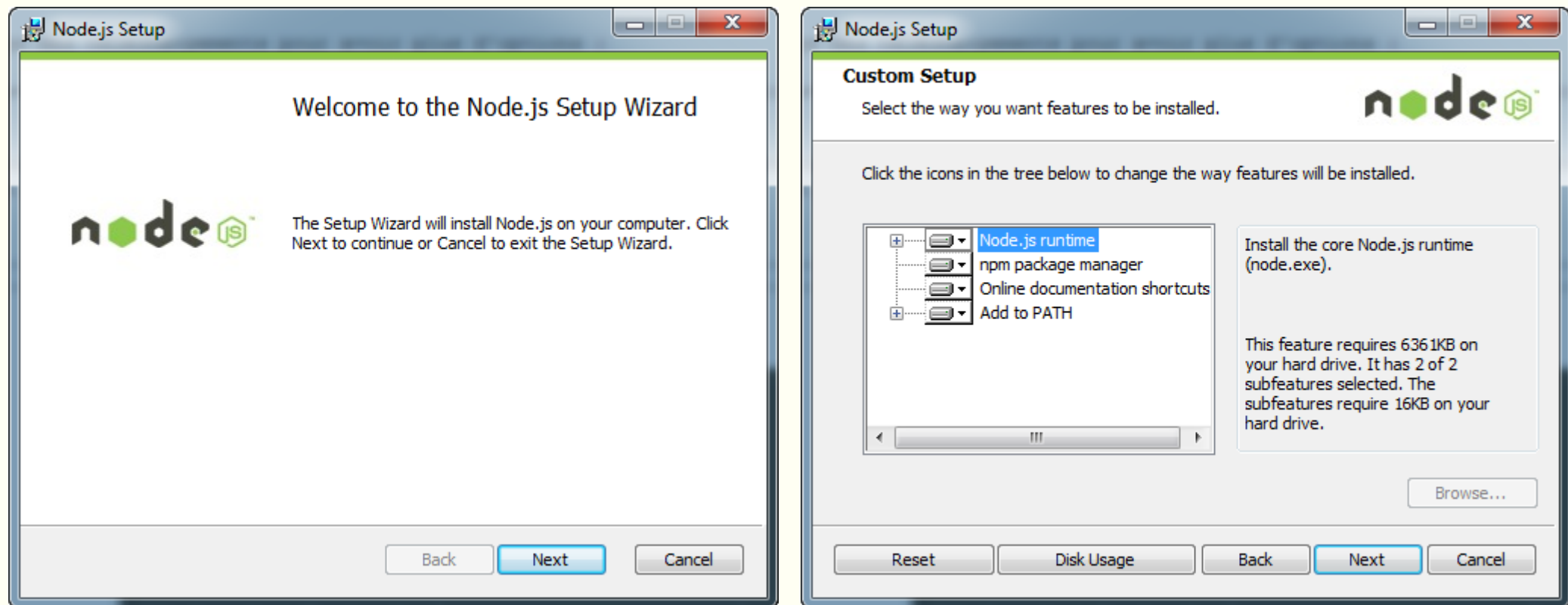
NodeJS : Installation - Windows

Node est téléchargeable via l'installateur Windows disponible sur la page de téléchargement du projet : <https://nodejs.org/en/>



Introduction

NodeJS : Installation - Windows



Il est conseillé de laisser toutes les options par défaut du programme d'installation, cela prend une trentaine de mégaoctets d'espace disque au maximum.

Introduction

NodeJS : Installation – Windows

- Afin de vérifier que Node est bien installé, on se propose de réaliser un premier script très simple qui se contente d'afficher un texte sur la sortie standard.
- Il suffit pour cela de créer un fichier JavaScript, appelé index.js, avec le contenu suivant :

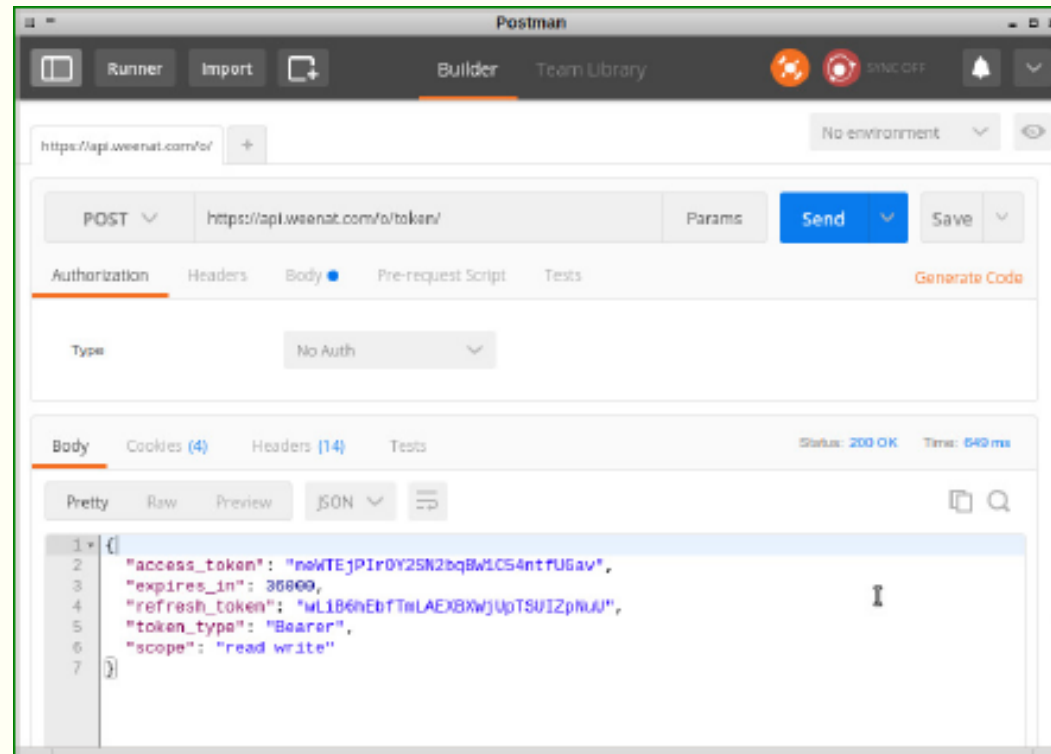
```
console.log('Bonjour tout le monde !');
```

- Il ne reste plus qu'à lancer le script avec l'interprète node :

```
$ node index.js  
Bonjour tout le monde !
```

Introduction

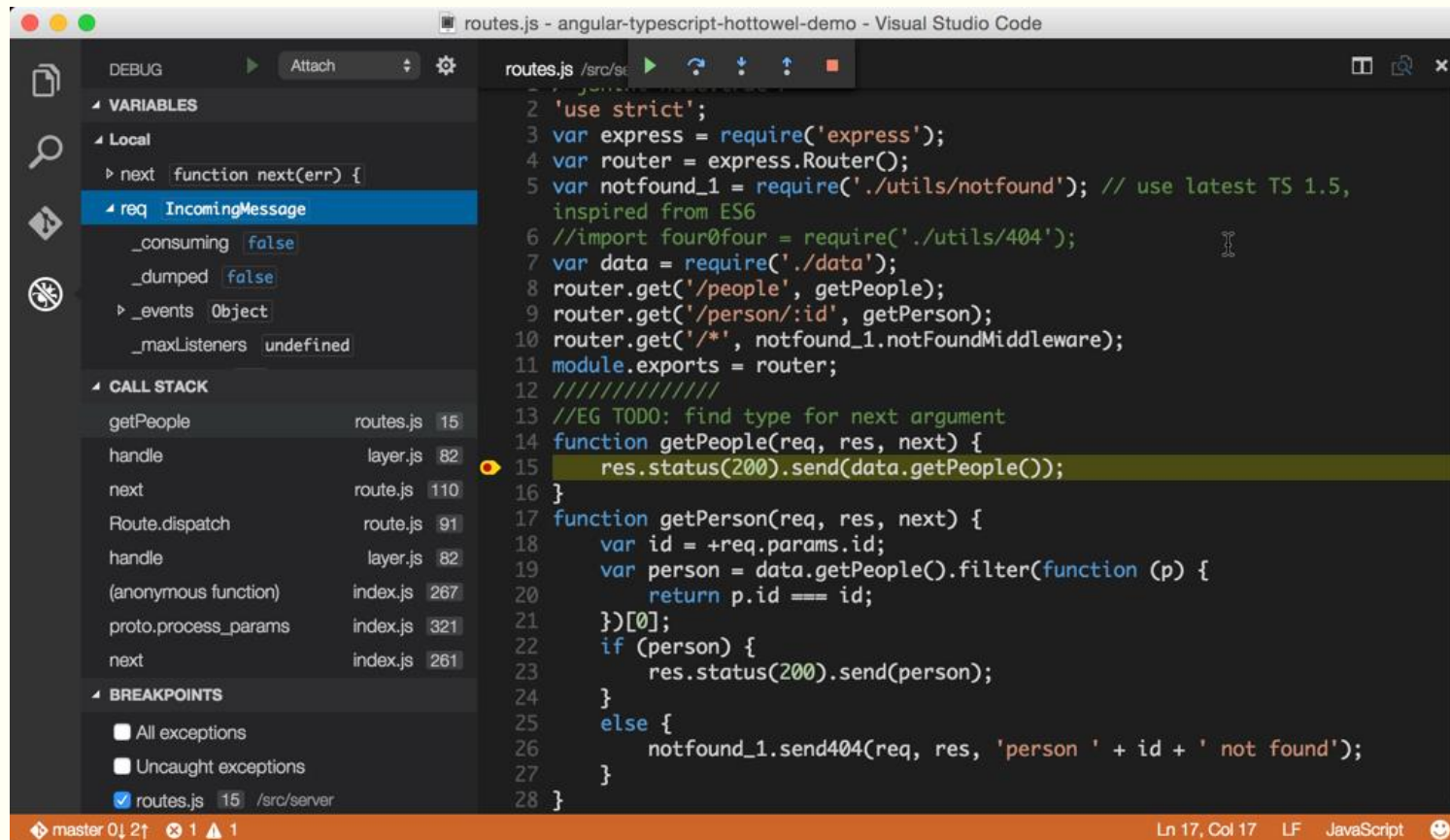
Les outils – client : POSTMAN



- définition de requêtes HTTP (méthode, url, headers, . . .)
- affichage des réponses (json, raw, . . .)

Introduction

Les outils – client : VSC



Disposition de titre et de contenu avec liste

Question ?

Disposition de titre et de contenu avec liste

TP

Installation de node

TP

Installation de VSC



MODULE 2

Ma première application

Ma première application

Ce module va vous permettre de créer votre première application Angular.

Nous y verrons :

- comment le faire **manuellement**
- puis comment le faire avec **les outils proposés par l'équipe d'Angular**.

Ma première application

Installation de TypeScript et initialisation du projet

- Le développement d'une application Angular se fait via le langage **TypeScript**.
- TypeScript est exposé sous la forme d'un **paquet npm** et s'installe via la commande suivante :

```
npm install -g typescript
```

- *Le paramètre -g de la commande précédente permet d'installer le paquet typescript en global c'est-à-dire accessible sur toute la machine.*
- L'environnement de la machine est maintenant prêt. La création de l'application en elle-même peut débuter.

Ma première application

Installation de TypeScript et initialisation du projet

- *Les commandes suivantes seront exécutées dans un répertoire dans lequel sera contenue l'application.*
- La première commande à exécuter permet d'initialiser le projet TypeScript.

```
tsc --init --target es5 --sourceMap --experimentalDecorators --emitDecoratorMetadata
```

- Le paramètre `--target es5` permet de compiler en ES5 pour rendre l'application compatible avec tout navigateur.
- Cette commande génère un fichier **tsconfig.json**. Dans ce fichier, il est nécessaire d'ajouter la propriété suivante à la racine :

```
"exclude":  
  "node_modules"  
]
```

Ma première application

Installation des dépendances Angular

Une application Angular se base sur un **ensemble de dépendances**.

Toutes les dépendances sont exposées sous la forme de paquets npm.

On va donc initialiser la configuration npm de l'application, via la commande suivante :

```
npm init
```

Cette commande va créer un fichier `package.json` contenant la liste des dépendances npm de l'application.

Ma première application

Installation des dépendances Angular

Il faut maintenant installer ces dépendances via la commande suivante :

```
npm install --save @angular/core @angular/compiler @angular/common  
@angular/platform-browser @angular/platform-browser-dynamic  
reflect-metadata zone.js rxjs
```

Le paramètre --save permet d'indiquer que l'on souhaite sauvegarder ces dépendances dans le fichier package.json.

Les dépendances installées sont les suivantes :

- **@angular** pour tous les paquets Angular.
- **reflect-metadata** pour pouvoir utiliser les décorateurs.
- **rxjs**, framework JavaScript permettant d'effectuer de la programmation reactive.
- **zone.js** pour permettre à Angular de détecter les changements.

Ma première application

Installation des dépendances Angular

Les dépendances sont maintenant récupérées.

Il reste toutefois à installer les fichiers *typings*, permettant au langage TypeScript d'avoir accès aux types et de pouvoir les vérifier à la compilation.

```
npm install --save-dev @types/core-js
```

Le paramètre --save-dev permet d'indiquer que l'on souhaite sauvegarder cette dépendance dans le fichier package.json en tant que dépendance de développement.

Ma première application

Mon premier composant

Ma première application

Mon premier composant

- La notion de composant sera vue plus en détail. dans le module Fondamentaux d'Angular.
- Dans le même répertoire que précédemment, il faut créer un fichier nommé **app.component.ts** contenant la classe TypeScript suivante :

```
export class MyFirstAppComponent {  
}
```

- Cette classe va représenter notre première application.
- Pour compiler ce fichier, il faut lancer la commande suivante : `tsc --watch`
- *Le paramètre --watch permet de relancer une compilation à chaque modification d'un fichier.*

Ma première application

Il faut maintenant indiquer à Angular que la classe **MyFirstAppComponent** est un composant.

Pour cela, il faut utiliser le décorateur Component :

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

@Component({})
export class MyFirstAppComponent {
}
```

Dans ce décorateur sera décrite toute la configuration du composant.

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

@Component({
  selector: 'my-first-app',
  template: '<h1>My First Angular App, hand made</h1>'
})
export class MyFirstAppComponent {
}
```

Ma première application

Mon premier module

Un module représente un conteneur regroupant l'ensemble des éléments de l'application.

Un module est représenté par une classe TypeScript décorée par NgModule, que l'on peut nommer app.module.ts :

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

@NgModule({
})
export class AppModule {
}
```

Ma première application

Le décorateur NgModule permet d'indiquer trois informations importantes :

- **Les imports** : liste des modules que notre module va importer. Cela permet de pouvoir utiliser tous les éléments inclus dans les modules importés.
- **Les déclarations** : liste des éléments à inclure dans le module.
- **Le bootstrap** : élément sur lequel l'application Angular va démarrer.

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

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

}
```

Ma première application

Lancement de l'application

La dernière étape consiste à créer le fichier qui va lancer l'application.

Ce fichier, que l'on va nommer **main.ts**, contient le code suivant :

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';  
import { AppModule } from './app.module';  
  
platformBrowserDynamic().bootstrapModule(AppModule);
```

L'import de `platformBrowserDynamic` permet d'avoir accès à la méthode permettant de démarrer le module sur un navigateur.

Ma première application

Lancement de l'application

L'application peut démarrer, il faut maintenant créer le fichier **index.html** correspondant au fichier qui sera affiché dans le navigateur :

```
<html>
<head></head>
<body>
  <my-first-app>Launching NG app</my-first-app>
</body>
</html>
```

Afin de pouvoir exécuter l'application sur tous les navigateurs, on va utiliser un gestionnaire de modules pour prendre le relais : **SystemJS**.

L'installation de ce gestionnaire se fait de manière classique via npm :

```
npm install --save systemjs
```

Ma première application

Il faut maintenant modifier la page HTML pour lui dire comment charger les fichiers JS de l'application.

```
<html>
<head>
  <script src="node_modules/zone.js/dist/zone.js"></script>
  <script src="node_modules/reflect-metadata/Reflect.js"></script>
  <script src="node_modules/systemjs/dist/system.js"></script>
  <script>
    System.config({
      map: {
        '@angular/core':
'node_modules/@angular/core/bundles/core.umd.js',
        '@angular/common':
'node_modules/@angular/common/bundles/common.umd.js',
        '@angular/compiler':
'node_modules/@angular/compiler/bundles/compiler.umd.js',
        '@angular/platform-browser':
'node_modules/@angular/platform-browser/bundles/platform-browser.umd.js',
        '@angular/platform-browser-dynamic':
'node_modules/@angular/platform-browser-dynamic/bundles/platform-browser-
dynamic.umd.js'
        'rxjs': 'node_modules/rxjs'
      },
      packages: {
        '.': {}
      }
    });

    System.import('main');
  </script>
</head>

<body>
  <my-first-app>Launching NG app</my-first-app>
</body>
</html>
```

La liste `System.import` permet de booter sur le fichier `main.ts`.

Ma première application

L'application est maintenant terminée. Pour la tester, on va installer un serveur HTTP, toujours via npm :

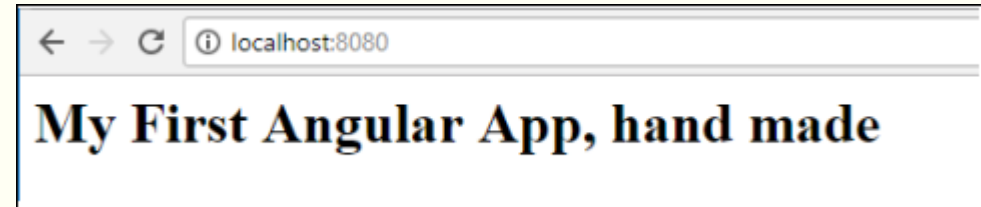
```
npm install http-server -g
```

***http-server** est un module npm servant de serveur de fichiers statiques.*

Le démarrage du serveur HTTP se fait alors simplement via la commande :

```
http-server
```

```
C:\Users\sebas\Desktop\ngalamain>http-server
Starting up http-server, serving ./
Available on:
  http://169.254.80.80:8080
  http://10.0.0.194:8080
  http://127.0.0.1:8080
Hit CTRL-C to stop the server
```



Disposition de titre et de contenu avec liste

TP

Premier Projet Manuel

Ma première application

Angular CLI à la rescousse

- Comme vu précédemment, l'initialisation d'un projet Angular est assez coûteux et peut s'avérer assez fastidieux.
- L'équipe Google a décidé de proposer directement un outil interne répondant à ces problématiques.
- Cet outil prend la forme d'une **CLI**(*Command Line Interface*) exposant un ensemble de commandes permettant d'accélérer le processus d'initialisation du projet.
- La CLI d'Angular est disponible sous la forme d'un paquet npm :

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

Ma première application

Angular CLI à la rescousse

Une fois installée, la CLI expose une commande **ng new** permettant de créer un nouveau projet :

```
ng new ngviacli
```

La commande précédente va créer une application Angular dans un répertoire nommé ngviacli. Cette commande est assez longue à s'exécuter puisqu'elle installe un ensemble de paquets npm pour initialiser le projet.

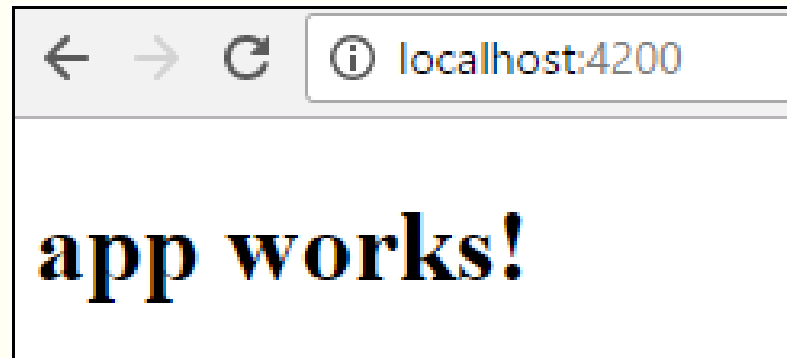
La CLI expose également une commande **ng serve** permettant de lancer l'application sur un serveur HTTP de développement.

```
cd ngviacli  
ng serve
```

Ma première application

Angular CLI à la rescousse

L'application est alors disponible sur l'URL `http://localhost:4200`.



Angular CLI

Generating Components, Directives, Pipes and Services

Scaffold	Usage
Component	<code>ng g component my-new-component</code>
Directive	<code>ng g directive my-new-directive</code>
Pipe	<code>ng g pipe my-new-pipe</code>
Service	<code>ng g service my-new-service</code>
Class	<code>ng g class my-new-class</code>
Guard	<code>ng g guard my-new-guard</code>
Interface	<code>ng g interface my-new-interface</code>
Enum	<code>ng g enum my-new-enum</code>
Module	<code>ng g module my-module</code>

Angular CLI

Updating Angular CLI

If you're using Angular CLI `1.0.0-beta.28` or less, you need to uninstall `angular-cli` package. It should be done due to changing of package's name and scope from `angular-cli` to `@angular/cli`:

```
npm uninstall -g angular-cli
npm uninstall --save-dev angular-cli
```

To update Angular CLI to a new version, you must update both the global package and your project's local package.

Global package:

```
npm uninstall -g @angular/cli
npm cache clean
# if npm version is > 5 then use `npm cache verify` to avoid errors (or to avoid using --force)
npm install -g @angular/cli@latest
```

Local project package:

```
rm -rf node_modules dist # use rmdir /S/Q node_modules dist in Windows Command Prompt; use rm -r -fo node_modules, dist
npm install --save-dev @angular/cli@latest
npm install
```

Disposition de titre et de contenu avec liste

TP

Premier Projet CLI

Disposition de titre et de contenu avec liste

Question ?

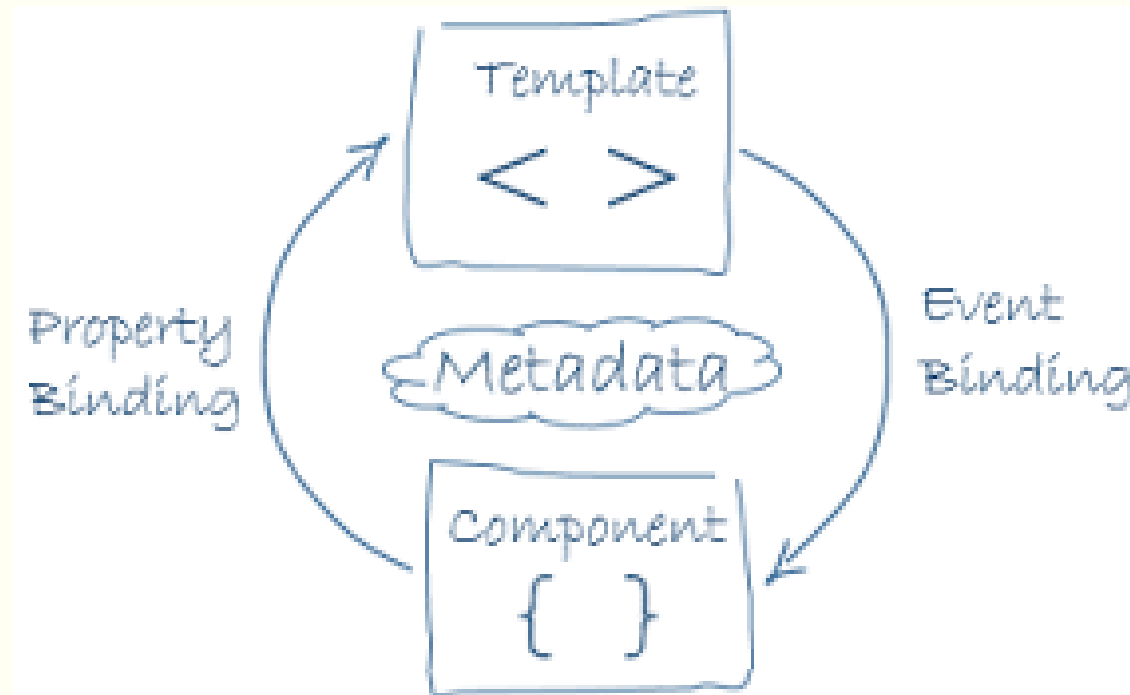


MODULE 3

Fondamentaux d'Angular

Les composants

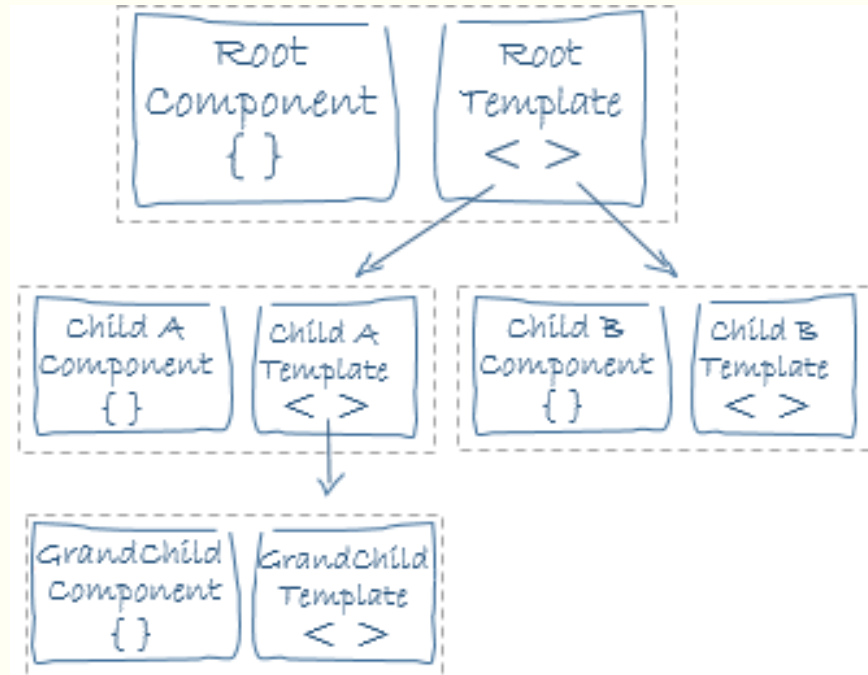
Un composant correspond à une classe exposant une vue et définissant la manière dont l'utilisateur interagit avec cette vue.



Les composants

Un composant peut être lui-même composé d'un ou de plusieurs composants, une application correspondant donc à un assemblage de ces éléments.

L'idée est qu'un composant doit être indépendant pour être réutilisable, peu importe les composants le référençant.



Les composants

Un composant Angular correspond à une classe TypeScript.

Cette classe peut déclarer des propriétés et des méthodes qui seront exposées en tant que modèle du composant et donc accessibles depuis la vue.

```
export class TodoListComponent {  
}
```

Pour indiquer à Angular que cette classe doit être considérée comme un composant, il faut lui ajouter une décoration.

Les décorations

La décoration **@Component** permet de considérer la classe associée comme un composant.

Cet attribut possède un ensemble de propriétés notamment :

- **selector** : indique le sélecteur CSS identifiant le composant dans le template. À chaque fois que ce sélecteur sera rencontré, Angular créera une nouvelle instance du composant correspondant.
- **template** ou **templateUrl** : template associé (en *inline* ou dans un fichier séparé).

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

@Component({
  selector: 'app-todolist',
  template: '<h1>Ma Todo List</h1>'
});
export class TodolistComponent {
  [...]
}
```

Les templates

Un **template** est du HTML représentant une vue.

Cette vue décrit à Angular comment doit être rendu le composant auquel il est associé.

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

@Component({
  selector: 'app-todolist',
  template: '<h1>Ma Todo List</h1>'
});
export class TodolistComponent {
  [...]
}
```

Déclarer un template directement
dans l'attribut

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

@Component({
  selector: 'app-todolist',
  templateUrl: './todo-list.component.html',
});
export class TodoListComponent {
  [...]
}
```

Définir une URL vers un fichier
correspondant au template

Les modules

- Un module permet d'organiser une application en blocs fonctionnels ou techniques.
- Il est également le point d'entrée d'une application Angular.
- Un module se déclare via une classe TypeScript ,décoré par l'attribut **@NgModule** :

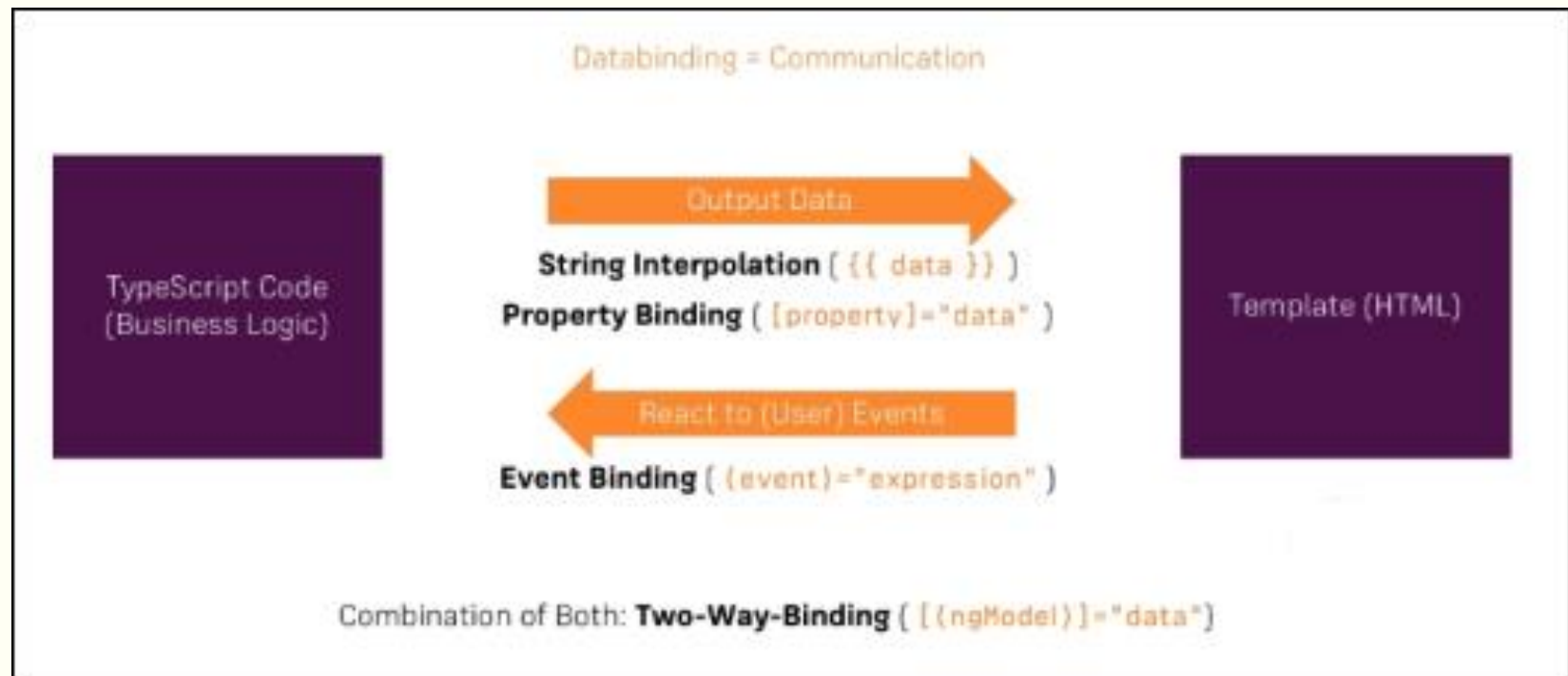
```
@NgModule({  
  imports: [BrowserModule],  
  declarations: [AppComponent],  
  bootstrap: [AppComponent],  
})  
export class AppModule { }
```

- Cet attribut possède notamment les propriétés suivantes :
 - **imports** : permet d'importer des modules, utiliser dans le module courant.
 - **declarations** : définit tous les éléments embarqués dans le module (composants, directives,, etc.).
 - **bootstrap** : définit le composant racine de l'application.
 - **providers** : définit les éléments (services par exemple) qui seront injectables dans tous les composants du module.

Binding

3 types de binding dans angular

- Interpolation
- Event Binding
- Property Binding



Binding - Interpolation

La déclaration d'un binding se fait dans la vue, via une syntaxe Angular `{{xx}}`.

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

@Component({
  selector: 'app-todolist',
  template: '<h1>{{listName}}</h1>',
});
export class TodolistComponent {
  listName = 'Ma Todo List';
}
```

- *Le template associé injecte la valeur de cette propriété dans le nœud <h1>.*
- La "**double moustache**" permet d'injecter du contenu dans la vue.
- Une relation de type **one-way binding** est alors créée.
- Cette relation implique que si le composant modifie les données, la vue sera automatiquement mise à jour.
- **Par contre l'inverse n'est pas vrai**, si la vue change, le composant ne le sera pas.

Binding - Property Binding

Il est également possible de rendre le contenu d'un attribut HTML dynamique en profitant du binding. Pour cela, il faut entourer l'attribut de crochet :

```
<button [disabled]="!isDirty">Save</button>
```

L'exemple de code précédent permet de désactiver le bouton si le flag `isDirty` est à `true`.

```
<img [src]="thumbnailUrl">
```

L'exemple de code précédent permet d'afficher une image en utilisant la propriété `thumbnailUrl` du modèle.

Binding - Event Binding

Angular permet de **réagir aux événements JavaScript déclenchés depuis la vue**, afin de notifier le composant.

Ce type de binding, permettant à un élément HTML de déclencher une action d'un composant, est appelé ***l'event binding***.

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

@Component({
  selector: 'app-todolist',
  template:
    `<h1>{{listTitle}}</h1>
    <form (submit)='createTodo()'>
      <input type='text' name='todoLabel' [(ngModel)]='newTodo' />
    </form>`,
})
export class TodolistComponent {
  listTitle = "Ma Todo List";
  todos: string[] = [];
  newTodo: string = '';

  createTodo() {
    if (this.newTodo) {
      this.todos.push(this.newTodo);
      this.newTodo = '';
    }
  }
}
```

Les directives

Les templates Angular permettent également d'itérer sur des collections du composant en définissant la manière dont sera affiché chaque élément à l'aide d'un template.

```
<li *ngFor='let todo of todos'>  
  ...  
</li>
```

La balise HTML portant l'attribut ngFor et son contenu seront dupliqués autant de fois qu'il y a d'éléments dans la collection.

Les directives

Angular permet d'afficher ou de cacher un élément de plusieurs manières.

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

@Component({
  selector: 'app-todolist',
  template:
    `

# {{listTitle}}


    <ul>
      <li *ngFor='let todo of todos; let i = index'>
        <hr *ngIf='i>0' />
        {{todo}}
      </li>
    </ul>`,
})
export class TodolistComponent {
  listTitle = "Ma Todo List";
  todos: string[] = [];
}
```

La directive **ngIf** permet ici d'afficher le hr uniquement si l'index est supérieur à zéro, donc pour tous les éléments sauf le premier.

Disposition de titre et de contenu avec liste

TP

Projet Basics



MODULE 4

TypeScript

JavaScript

- Le langage JavaScript s'est imposé depuis quelques années comme le langage indispensable au développement d'applications web.
- JavaScript est un langage interprété, le code écrit étant directement exécuté sans phase de compilation préalable.
- JavaScript est également un langage dynamiquement typé, c'est-à-dire qu'un élément peut changer de type en cours d'exécution.
- *Exemple du typage dynamique JavaScript. La variable est initialisée en tant que fonction puis prend une valeur entière.*

```
var maFonction = function() {  
    console.log("I'm a function");  
}  
maFonction = 27;
```


JavaScript

- Enfin, JavaScript est un langage par nature monothreadé, fortement asynchrone et basé sur un mécanisme d'événements non bloquants.
- Lorsque l'on développe une application en JavaScript, sa forte dynamicité devient rapidement un handicap.
- Comment faire pour s'assurer que le code que l'on écrit est syntaxiquement valide ?
- Comment effectuer des phases de refactoring sans apporter des régressions ?

JavaScript

- Dans ce code, il y a une erreur d'orthographe à l'appel de la méthode **maFonction**. L'erreur ne sera visible qu'à l'exécution du code.

```
var maFonction = function() {  
    console.log("I'm a function");  
}  
maFunction();
```

- Dans le code suivant, la méthode **maFonction** est appelée sans paramètre. À l'exécution, la console affichera le message "I'm a function with a param undefined".

```
var maFonction = function(param) {  
    console.log("I'm a function with a param " + param);  
}  
maFonction();
```

- Dans le code précédent, on affecte un entier à la variable **maFonction**, qui était une fonction. L'exécution de ce code renverra l'erreur suivante : "maFonction is not a function".

```
var maFonction = function() {  
    console.log("I'm a function");  
}  
maFonction = 4;  
maFonction(); Mbengue.M
```

TypeScript

- L'une des solutions à ces problématiques a été la création de langages qui se **transcompilent** en JavaScript, c'est-à-dire que la compilation génère des fichiers JavaScript.
- C'est notamment le cas de TypeScript, porté par Microsoft et utilisé par **Angular**.
- D'autres langages similaires existent, comme **Dart** ou **CoffeeScript**, mais TypeScript est sans doute aujourd'hui l'un des plus complets.
- TypeScript est donc un langage compilé et typé fortement qui génère du **JavaScript compréhensible par tous les navigateurs**.

TypeScript

- Le typage fort de ce langage va permettre d'associer un type à un élément et empêcher cet élément de changer de type.

```
var id: number;
```

- La phase de compilation de TypeScript permet de valider syntaxiquement le code.
- Les phases de refactoring, qui s'avéraient extrêmement compliquées et dangereuses, sont beaucoup plus simples.
- De manière générale, le code sera beaucoup plus robuste et stable.
- De nombreux éditeurs, notamment **Visual Studio**, **VSCode** ou **SublimeText**, comprennent le langage TypeScript et proposent de nombreux outils, comme **l'autocomplétion, les erreurs de compilation directement dans l'éditeur, etc.**

TypeScript - Syntaxe

Variables

La déclaration d'une variable se fait de la façon suivante :

```
var maVariableBooleenne : boolean;
```

Il existe un ensemble de types de base : boolean, number, string, [] pour un tableau, etc

Dans l'optique de garder la possibilité de profiter du typage dynamique de JavaScript, TypeScript introduit le type **any**..

```
var maVariableDynamique : any;
```

- Dans ce cas, TypeScript ne vérifiera pas le type de cette variable à la compilation.
- **Si le type d'une variable n'est pas précisé, TypeScript la considérera comme étant de type any.**

TypeScript - Syntaxe

Fonctions

La déclaration d'une fonction se fait de manière classique.

```
function maFonction() : void {  
    ...  
}
```

- Le type de retour de la fonction se définit de manière identique au type d'une variable. Le type **void** indique qu'il n'y a aucun retour.
- Si la fonction prend des paramètres en entrées, ces paramètres se typent de manière identique aux variables :

```
function maFonction(monParametre : string, monAutreParametre :  
boolean) : void {  
    ...  
}
```

TypeScript - Syntaxe

Classes

La notion de classe est apparue en JavaScript avec EcmaScript 6.

```
class Person {  
  name : string;  
  age: number;  
  
  constructor(name : string, age: number) {  
    this.name = name;  
    this.age = age;  
  
    toString() {  
      return `Hi I'm ${this.name} and I'm ${this.age} years old!`;  
    }  
  }  
}
```

Le mot-clé class permet d'indiquer que l'on est en train de créer une classe.

TypeScript - Syntaxe

Classes

Il est également possible de créer de l'héritage entre classes :

```
class Developer extends Person {  
    constructor(name, age, language) {  
        super(name, age);  
        this.language = language;  
    }  
  
    toString() {  
        return super.toString() + ` :: I'm a Developer who likes  
        ${this.language}`;  
    }  
}
```

La relation d'héritage se déclare via le mot-clé `extends`. Il est ensuite possible d'accéder aux éléments de la classe parente via la méthode `super`.

TypeScript - Syntaxe

Classes

TypeScript introduit également la notion de visibilité sur les propriétés.

Il est possible de déclarer une propriété private, qui ne sera accessible que depuis la classe courante, protected, qui ne sera accessible que depuis la classe courante ou les classes héritant de la classe courante, et enfin une propriété public qui sera accessible depuis l'extérieur de la classe.

Par défaut, si aucune visibilité n'est spécifiée, une propriété sera public.

```
class Modifier {  
    public myPublicProperty: string;  
    protected myProtectedProperty: string;  
    private myPrivateProperty: string;  
}
```

TypeScript

Union type

Les unions types permettent de définir une valeur qui peut avoir plusieurs types.

Cela permet de profiter de la dynamicité du langage JavaScript tout en gardant un typage pour plus de sécurité.

```
var pet : Fish | Bird;
```

La variable pet peut avoir comme type soit Fish soit Bird.

TypeScript

Interface

TypeScript permet également d'apporter des notions n'existant pas en JavaScript. C'est notamment le cas avec les interfaces.

```
interface ISender {  
    send(message: string): boolean;  
}
```

Un point intéressant à noter sur les interfaces est que **TypeScript ne générera pas de code JavaScript correspondant**. En effet, la notion d'interface n'existant pas en JavaScript, celle-ci est uniquement utilisée pour définir des contrats qui seront validés à la compilation et ne nécessite pas de génération de code.

TypeScript

Interface

Les interfaces TypeScript sont également utilisées pour représenter les modèles de notre application. L'avantage par rapport à l'utilisation de classe est que l'on ne va pas générer de code inutile.

```
interface Customer {  
  name: string;  
  firstName: string;  
  email: string;  
  age: number;  
}
```

```
var c : Customer = {  
  name: 'Ollivier',  
  firstName: 'Sébastien',  
  email: 'sebastien.ollivier@test.fr',  
  age: 31  
}
```

Le code généré par ces deux portions de TypeScript sera le code suivant :

```
var c = {  
  name: 'Ollivier',  
  firstName: 'Sébastien',  
  email: 'sebastien.ollivier@test.fr',  
  age: 31  
};
```

TypeScript

Générique

Il est également possible d'utiliser des génériques en TypeScript, notion n'existant pas en JavaScript.

```
class Greeter<T> {  
    greeting: T;  
  
    constructor(message: T) {  
        this.greeting = message;  
    }  
    greet() {  
        return this.greeting;  
    }  
}
```

L'instanciation de ce type générique se fait simplement en spécifiant le type souhaité :

```
let greeter = new Greeter<string>("Hello, world");
```

TypeScript

Les décorateurs

Les décorateurs (proposition d'EcmaScript 7 disponible en TypeScript) sont des annotations qui permettent de modifier une classe et ses propriétés au moment de son implémentation.

```
@Component({
  selector: 'my-component',
  template: '<div>Hello my name is {{name}}.</div>'
})
class MyComponent {
  constructor() {
    this.name = 'Sébastien'
  }
}
```

Dans l'exemple précédent, la décoration @Component d'Angular permet d'indiquer que la classe MyComponent correspond à un composant.

Disposition de titre et de contenu avec liste

TP

Projet TypeScript



MODULE 5

Les composants

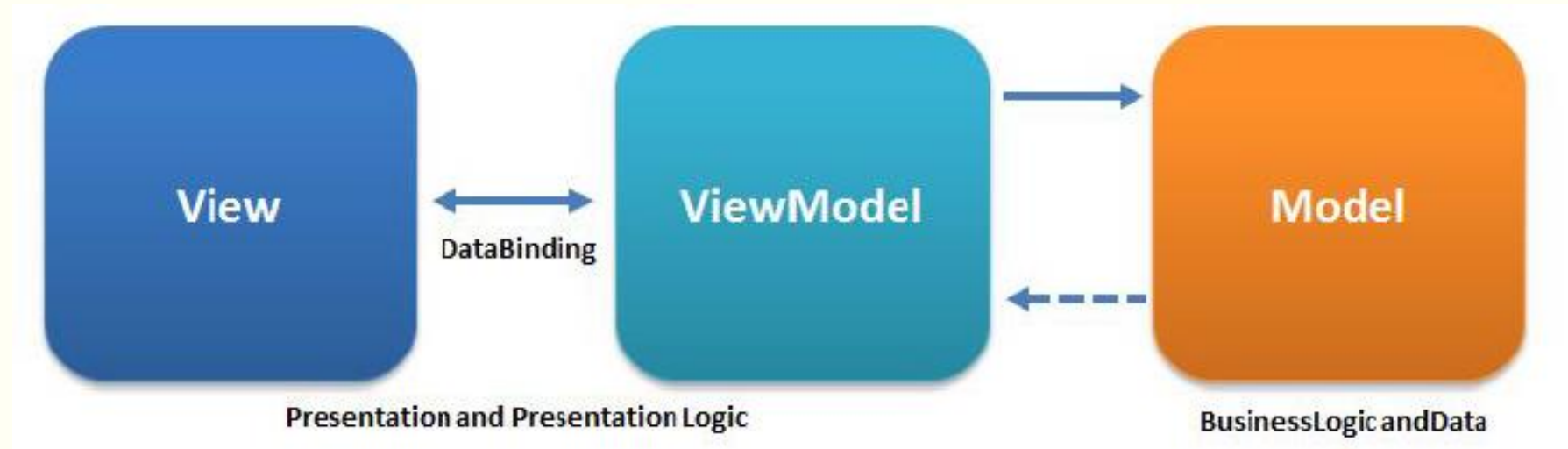
Les composants

Les constituants principaux d'un composants

Une chaîne de caractères
qui représente
le template du composant

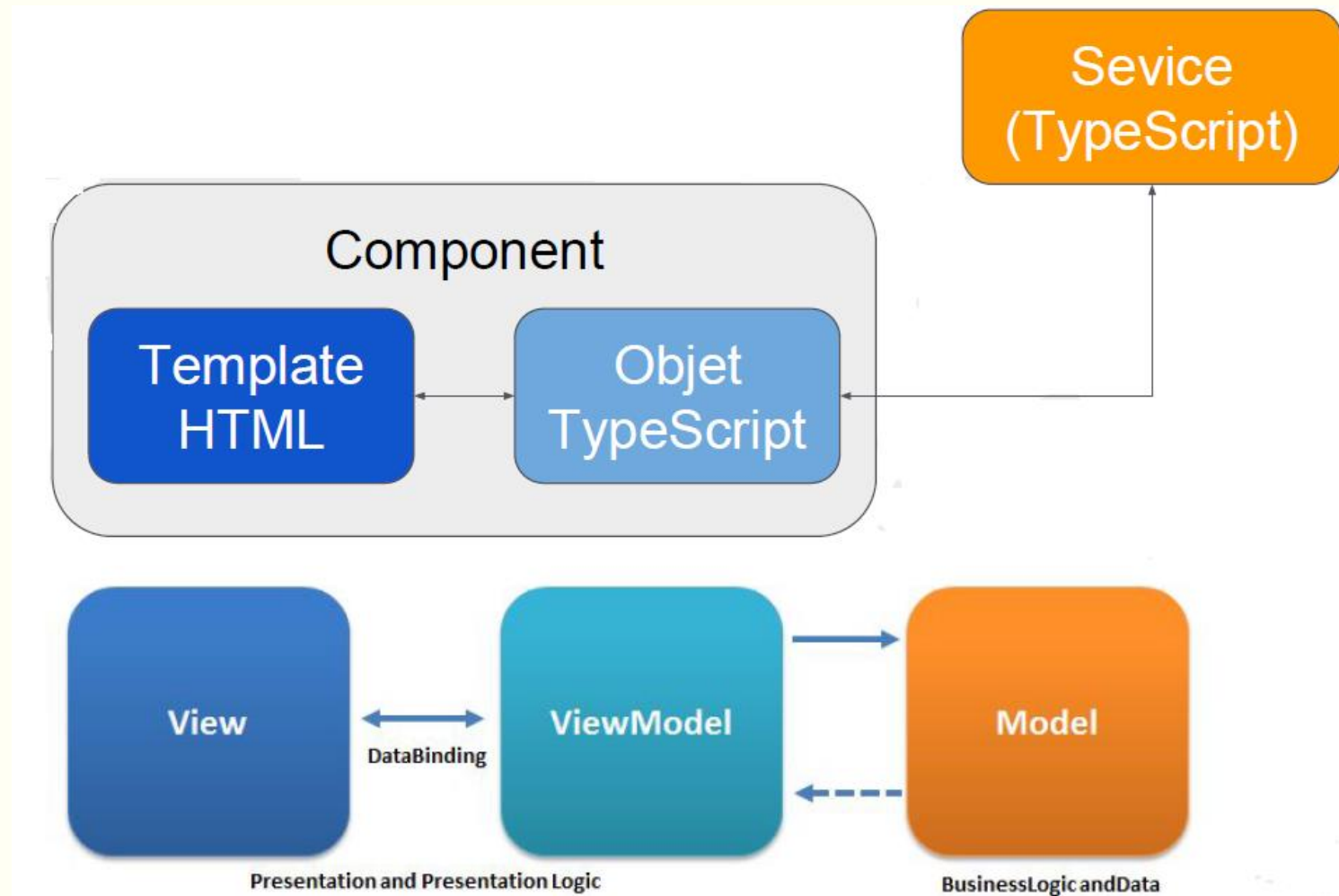
Une classe TypeScript
qui gère
la logique du composant

Parfois des services
qui permettent
d'agir sur le noyau



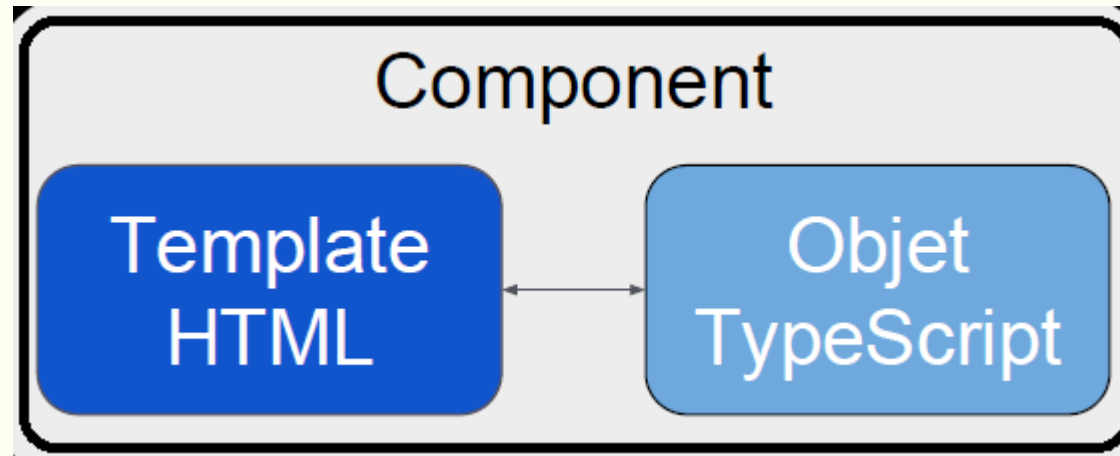
Le *two-way databinding* (liaison de données bidirectionnelle) permet de lier deux flux de données.

Les composants



Les composants

Le composant est la structure fondamentale d'une application Angular.



Les inputs d'un composant

Déclarer une variable en tant qu'Input

Pour qu'une application basée sur les composants puisse fonctionner, il est essentiel de pouvoir rendre les composants paramétrables.

Pour cela, Angular fournit le décorateur **@Input**, celui-ci va permettre d'identifier une propriété du composant en tant qu'input dudit composant.

L'input peut être de n'importe quel type TypeScript, un number, un string, ou même une classe/interface que vous aurez créée.

Les inputs d'un composant

Exemple

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

@Component({
  selector: 'app-todo',
  templateUrl: './todo.component.html',
  styleUrls: ['./todo.component.css']
})
export class TodoComponent {
  @Input()
  text: string;

  @Input()
  isChecked: boolean;
}
```

- On importe le décorateur **@Input** depuis **@angular/core**, et on l'utilise sur les propriétés **text** et **isChecked**.
- Ces deux propriétés sont donc des inputs du composant Todo et il est maintenant possible de les initialiser dans la balise **app-todo**.

Les inputs d'un composant

Exemple suite ...

Propriétés renseignées en dur

```
<app-todo text="Faire les courses" isChecked="true">  
</app-todo>
```

Propriétés du composant parent

```
todoChecked: boolean = true;  
todoName: string = "Faire les courses";
```

Il est alors possible de donner ces deux propriétés au composant Todo. Pour cela, il faut utiliser la syntaxe appropriée, et ce sont des crochets qui viennent entourer les propriétés dans le HTML.

Ainsi, dès que les propriétés sont modifiées par le composant parent, le composant Todo se mettra à jour en conséquence, car il recevra les nouvelles valeurs.

Les inputs d'un composant

Donner un nom personnalisé à son input

Parfois, il est intéressant d'avoir un nom spécifique et concis pour son utilisation dans la vue HTML, sauf que cette contrainte n'en est pas une pour le code TypeScript en soi.

C'est pourquoi il est possible de nommer différemment ses inputs en passant le nom public voulu en paramètre du décorateur Input.

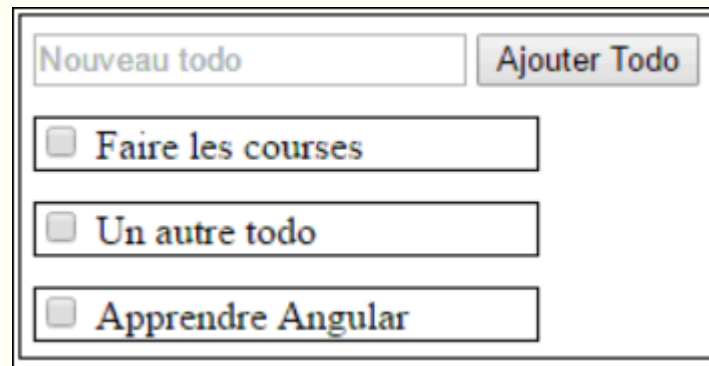
Cela donnerait par exemple :

```
@Input("todo")
todoModel: TodoModel;
```

Ainsi, le composant appelant va devoir passer un paramètre nommé todo, mais le composant lui va continuer d'utiliser la propriété todoModel.

Les outputs d'un composant

Supposons le cas d'une liste de Todo comme suit:



The image shows a web interface for a todo list. At the top, there is a text input field labeled 'Nouveau todo' and a button labeled 'Ajouter Todo'. Below this, there is a list of three todo items. Each item consists of a checkbox on the left and a text input field on the right. The items are: 'Faire les courses', 'Un autre todo', and 'Apprendre Angular'.

Maintenant nous voulons pouvoir supprimer des *Todos*. Il faut ajouter un bouton de suppression, et lorsque celui-ci est activé, supprimer le *Todo* de la liste.

Une première solution serait d'ajouter le bouton au niveau du `*ngFor` et d'appeler une méthode qui supprime le *Todo* de la liste.

```
<div *ngFor="let todo of todoList">
  <app-todo [todo]="todo">
  </app-todo>
  <button (click)="deleteTodo(todo)">Supprimer</button>
</div>
```

Mbengue.M

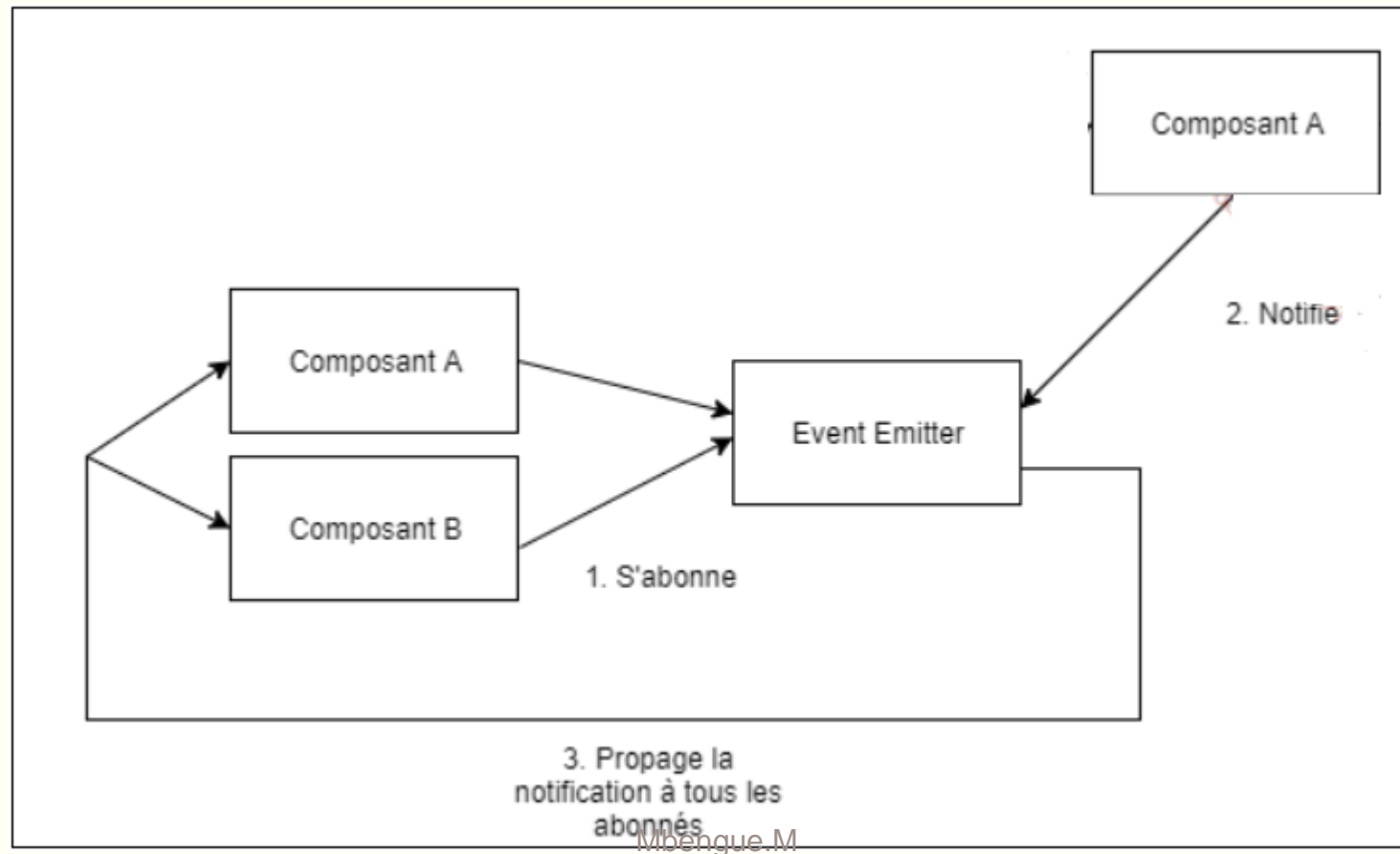
Les outputs d'un composant

Déclarer un output

- La méthode précédente fonctionne très bien, mais un problème demeure. Il serait préférable d'avoir le bouton de suppression dans le `TodoComponent` lui-même, le découpage dans ce cas serait plus intéressant. Toutefois, notre liste de *Todo* se trouve dans le composant `AppComponent` parent.
- C'est pour ces situations, en partie, que les outputs ont été conçus.
- **L'objectif est de notifier le composant parent qu'un événement s'est produit au sein du composant enfant.**

Les outputs d'un composant

Pour pouvoir notifier le composant parent, Angular met à disposition la classe **EventEmitter**.



Les outputs d'un composant

L'**EventEmitter** est typé selon un modèle, qui peut être une classe, une interface ou bien un type valeur directement.

C'est une instance de ce type qui va transiter à chaque notification.

Pour lancer une notification, il faut exécuter la méthode **emit** de l'**EventEmitter** en lui passant la valeur/instance à transiter.

Les outputs d'un composant

Exemple:

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

@Component({
  selector: 'app-todo',
  templateUrl: './todo.component.html',
  styleUrls: ['./todo.component.css']
})
export class TodoComponent {
  @Input()
  todo: TodoModel;

  @Output()
  deleteTriggered: EventEmitter<TodoModel> =
new EventEmitter<TodoModel>();

  deleteTodo() {
    this.deleteTriggered.emit(this.todo);
  }
}
```

Les outputs d'un composant

Exemple suite... coté parent

```
<app-todo [todo]="todo" (deleteTriggered)="deleteTodo($event)">  
</app-todo>
```

Ici, le (**deleteTriggered**) permet donc de réagir à l'output en appelant la méthode **deleteTodo**.

Le paramètre **\$event** est la valeur qui a été transmise par le biais de la méthode emit sur l'EventEmitter delete Triggered, ce qui permet donc de fournir le todo correspondant à la méthode deleteTodo.

Les outputs d'un composant

Donner un nom personnalisé à son output

Tout comme les inputs, il est possible de spécifier un nom public différent du nom privé, interne au composant.

```
@Output('deleteTriggered')
deleteEventEmitter: EventEmitter<TodoModel> =
new EventEmitter<TodoModel>();
```

Disposition de titre et de contenu avec liste

TP

Lab_ Input/Output

Passer une donnée du parent vers l'enfant à l'aide d'un input

- Le parent possède deux propriétés : **name** et **email** qu'il va transmettre au composant enfant à l'aide d'un input de ce composant enfant.
- Pour de l'affichage uniquement, un **getter** data renvoie un objet qui contient ces deux propriétés, pour que l'on puisse les afficher en JSON dans une balise **pre**, à l'aide du pipe **json**, intégrée à Angular.
- Le composant app-children est affiché et le parent renseigne les deux propriétés de l'enfant..

Passer une donnée du parent vers l'enfant à l'aide d'un input

Composant parent

```
@Component({
  selector: 'app-parent',
  template: `<div>
    Le parent possède :
    <pre>{{ data | json }}</pre>
    <hr />
    <!-- Les propriétés du parent sont passées au composant enfant -->
    <app-children [name]="name" [email]="email"></app-children>
  </div>`,
  styles: [
    :...`
  ]
})
export class ParentComponent {
  name: string = "Parent";
  email: string = "parent@angular.com";

  get data() {
    return { name: this.name, email: this.email };
  }
}
```

Passer une donnée du parent vers l'enfant à l'aide d'un input

- Au niveau de l'enfant, deux propriétés sont donc déclarées, la propriété **name** et l'autre propriété **mail**, mais cette dernière propriété possède un décorateur **@Input** qui spécifie que son nom public est email.
- C'est pour cela que le parent renseigne une propriété email et non mail.

Composant enfant

```
@Component(  
  selector: 'app-children',  
  template: `  
    <pre>{{ data | json }}</pre>`  
  )  
export class ChildrenComponent {  
  @Input()  
  name: string;  
  @Input('email')  
  mail: string;  
  
  get data()  
  {  
    |   return { name: this.name, email: this.mail };  
  }  
}
```

Passer une donnée du parent vers l'enfant à l'aide d'un input

Un simple input est déjà très intéressant, toutefois **il est parfois nécessaire de savoir quand cette valeur change.**

En **découpant** la déclaration d'une propriété en **getter/setter**, on peut alors exécuter le code que l'on veut lorsque le composant reçoit une nouvelle information.

```
@Component({
  selector: 'app-parent',
  template: `<div>
    <input [(ngModel)]="text" placeholder="Texte" />
    <hr />
    <app-children [name]="text"></app-children>
  </div>`,
  styles: [`
    :host{
      display:inline-block;
    }`
  ])
export class ParentComponent {
  text: string;
```

Passer une donnée du parent vers l'enfant à l'aide d'un input

- Le composant enfant déclare alors un input nommé text sous la forme d'un *getter* et d'un *setter*. Dans le *setter*, nous allons tout simplement pousser dans un tableau les nouvelles valeurs que le composant reçoit.
- Au niveau du template HTML, le composant se contente d'afficher la valeur courante de la propriété text, puis une directive **ngFor* affiche toutes les valeurs du tableau dans une liste.

```
@Component({
  selector: 'app-children',
  template: `<div>
    <div> Propriété name : {{ name }} </div>
    Historique des valeurs :
    <ul>
      <li *ngFor="let oldName of nameHistory">{{oldName}}</li>
    </ul>
  </div>`
})
export class ChildrenComponent {
  nameHistory: string[] = [];

  private _name: string;

  get name(): string { return this._name; }

  @Input()
  set name(value: string) {
    this.nameHistory.push(value);
    this._name = value;
  }
}
```

Notifier le parent à l'aide d'un EventEmitter en output

Cette méthode est une simple utilisation d'output.

Le parent s'abonne à un événement personnalisé du composant enfant grâce à une combinaison:

- d'Output,
- d'EventEmitter,
- et d'exécution de méthode lorsque l'enfant émet une notification.

Composant parent

```
3 @Component({
4   selector: 'app-parent2',
5   template: `
6     <div>
7       <app-children2
8         (sendMessage)="onSendMessage($event)"
9         (pokeEmitter)="onSendPoke()">
10      </app-children2>
11      <li *ngFor="let message of messages">{{message}}</li>
12    </div>
13  `,
14   styles: []
15 })
16 export class Parent2Component implements OnInit {
17
18   messages: string[] = [];
19
20   onSendMessage(message: string): void {
21     this.messages.push(message);
22   }
23
24   onSendPoke() {
25     this.messages.push("Poke reçu !");
26   }
27 }
```

Sur la méthode `onSendMessage`, le tableau est rempli avec le message renvoyé par l'enfant, que l'on récupère avec la syntaxe `$event`.

Notifier le parent à l'aide d'un EventEmitter en output

Composant enfant

```
2
3 @Component({
4   selector: 'app-children2',
5   template: `
6     <div>
7       <input [(ngModel)]="text" placeholder="Nouveau message" />
8       <button (click)="sendMessage()">Envoyer le message</button>
9       <button (click)="sendPoke()">Envoyer un poke</button>
10    </div>
11  `,
12  styles: []
13 })
14 export class Children2Component implements OnInit {
15
16   text: string;
17
18   @Output()
19   pokeEmitter: EventEmitter<void> = new EventEmitter<void>();
20
21   @Output('sendMessage')
22   sendMessageEmitter: EventEmitter<string> = new EventEmitter<string>();
23
24   sendMessage() {
25     this.sendMessageEmitter.emit(this.text);
26     this.text = "";
27   }
28
29   sendPoke() {
30     this.pokeEmitter.emit();
31   }
32 }
```

Observer les changements d'input avec ngOnChanges

- Il existe une méthode dans le cycle de vie d'une application Angular qui est appelée lorsque les inputs du composant sont modifiés.
- Cette méthode est `ngOnChanges(changes: SimpleChanges);`
- Implémenter, la classe abstraite **OnChanges** depuis le module **@angular/core**.
- La méthode ngOnChanges prend en paramètre un objet SimpleChanges qui est un dictionnaire.
- Chaque objet de ce dictionnaire est une instance de SimpleChange qui contient trois propriétés :
 - **currentValue** : la nouvelle valeur de la propriété.
 - **previousValue** : l'ancienne valeur de la propriété.
 - **isFirstChange** : un booléen qui indique si c'est le premier changement de la propriété.

Observer les changements d'input avec ngOnChanges

Composant enfant

```
3 @Component({
4   selector: 'app-children3',
5   template: `
6 <div>
7   <div> Propriétés : <pre> {{ data | json }} </pre> </div>
8   Historique des valeurs :
9   <ul> <li *ngFor="let changeInfo of changeHistory">{{changeInfo}}</li> </ul>
10 </div>
11 ` ,
12   styles: []
13 })
14 export class Children3Component implements OnInit, OnChanges {
15
16   changeHistory: string[] = [];
17
18   @Input()
19   firstName: string;
20
21   @Input()
22   lastName: string;
23
24   ngOnChanges(changes: SimpleChanges) {
25     for (let propName in changes) {
26       let changedProp: SimpleChange = changes[propName];
27       let newVal = JSON.stringify(changedProp.currentValue);
28       if (changedProp.isFirstChange()) {
29         this.changeHistory.push(`[${propName}] Valeur initiale : ${newVal}`);
30       } else {
31         let previousVal =
32           JSON.stringify(changedProp.previousValue);
33         this.changeHistory.push(`[${propName}] ${previousVal} -> ${newVal}`);
34       }
35     }
36   }
37
38   get data() {
39     return { firstName: this.firstName, lastName: this.lastName };
40   }
41 }
```


Observer les changements d'input avec ngOnChanges

Ce code récupère tous les changements d'inputs et les affiche via une liste de changements (le tableau "**changeHistory**").

Pour chaque changement, on récupère les différentes informations mises à disposition, comme le "**isFirstChange**" qui permet de savoir si c'est le premier changement de la propriété, ou bien le "**previousValue**" qui est la valeur précédente.

Enfin, la nouvelle valeur est fournie elle aussi dans la propriété "**currentValue**".

Utiliser une variable locale

Il est possible d'utiliser une variable locale d'un composant au sein du template d'un autre composant.

En créant une **template reference** variable pour l'élément voulu, il est possible, au sein du même template où la variable est créée, **d'accéder aux propriétés** de cet élément ainsi que **d'utiliser ses méthodes**.

La syntaxe d'une template reference variable est tout simplement le nom de la variable précédé d'un dièse :

```
<component #myComponent></component>
```

Utiliser une variable locale

Prenons l'exemple d'un composant parent qui possède :

- Un bouton.
- Un composant enfant.
- Une variable locale faisant référence sur ce composant enfant.

Le composant parent peut alors afficher la propriété text du composant enfant et même lancer la méthode showMessage du composant enfant.

Grâce au **#children** placé sur la balise **app-children**, une référence vers le composant enfant est gardée dans le template.

Cette référence est alors une variable qui est utilisable partout dans le template.

```
{{children.text}}  
children.showMessage()
```

Utiliser une variable locale

```
3 @Component({
4   selector: 'app-parent4',
5   template: `
6 <div>
7   <button (click)="children.showMessage()">Appeler children.showMessage()</button>
8   Propriété 'text' de l'enfant : {{children.text}}
9   <app-children4 #children></app-children4>
10 </div>
11 ` ,
12   styles: []
13 })
14 export class Parent4Component implements OnInit {
15
16   text: string;
```

Composant parent

Composant enfant

```
3 @Component({
4   selector: 'app-children4',
5   template: `
6 <div>
7   <input [(ngModel)]="text" placeholder="Texte de l'enfant" />
8 </div>
9 ` ,
10   styles: []
11 })
12 export class Children4Component implements OnInit {
13
14   text: string;
15
16   showMessage():void {
17     alert(this.text);
18   }
19 }
```

Disposition de titre et de contenu avec liste

TP

Lab_Intéraktion



MODULE 6

Les pipes

Les pipes

Dans une application Angular, il n'est pas rare d'effectuer une transformation sur un élément avant de l'afficher.

L'exemple le plus simple est celui d'une **date**.

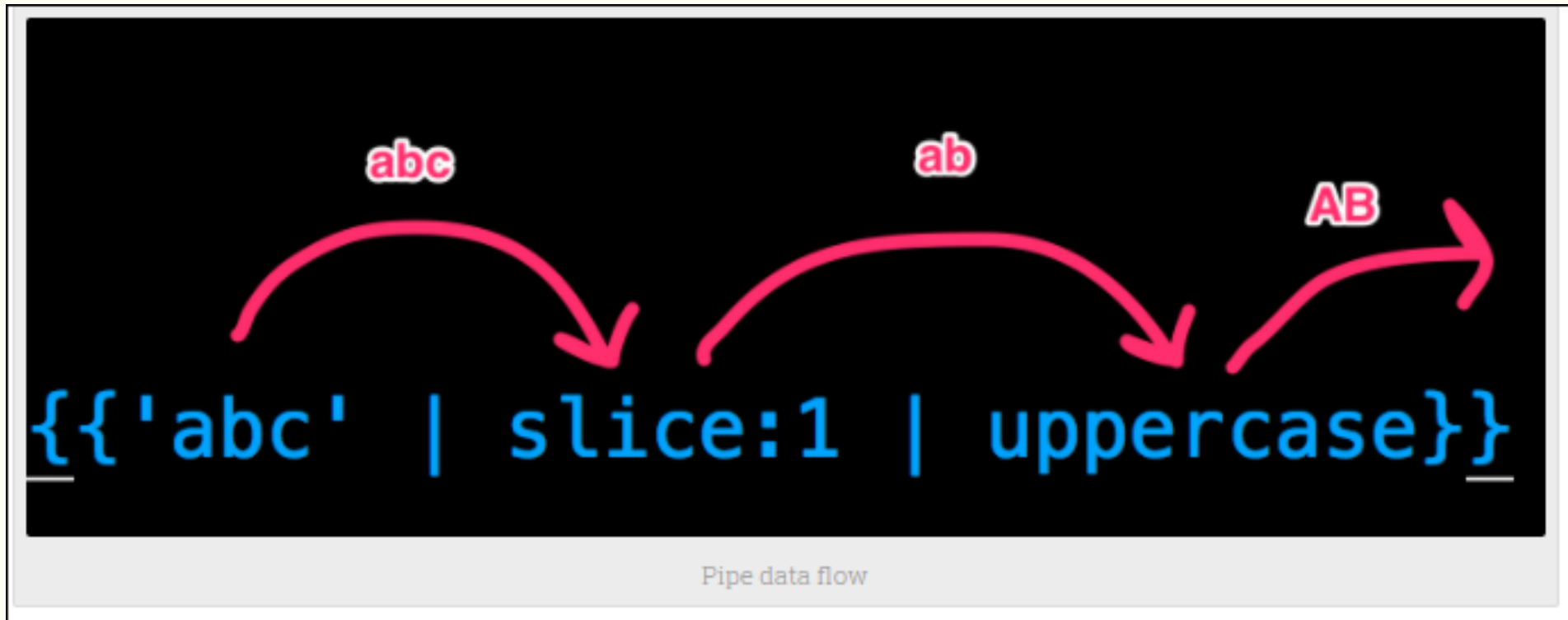
Lorsqu'une date est affichée, c'est rarement sous le **format traditionnel JavaScript**.

En effet, une date JavaScript s'affiche au format suivant : **Sun Jul 10 2016 17:03:07 GMT+0200 (Paris, Madrid (heure d'été))** alors qu'un utilisateur a plutôt l'habitude de la lire sous un format plus accessible, comme **10/06/2017**.

Le pipe est l'élément qui va permettre de gérer ces transformations.

Les pipes

▪



Les pipes - Utiliser un pipe

- Un pipe s'utilise directement dans le template d'un composant.
- Il suffit d'appliquer l'opérateur pipe '|' entre la valeur à transformer et le pipe que l'on souhaite utiliser.
- Dans l'exemple ci-dessous, c'est le pipe date (un pipe fourni par Angular) qui est utilisé afin de rendre la date du jour plus lisible.

```
1  import { Component } from '@angular/core';
2
3  @Component({
4    selector: 'app-today',
5    template: `
6      <p>Nous sommes le {{today | date}}</p>
7    `,
8    styles: []
9  })
10 export class TodayComponent {
11   today = new Date();
12 }
```

- Cependant, dans cet exemple, le template généré ressemblera à cela : Nous sommes le Jul 10, 2016. En effet, Angular va utiliser par défaut le format de date américain.

Les pipes -

- Pour spécifier le format à appliquer à la date, il faut passer des paramètres au pipe.
- Les paramètres se placent directement derrière le nom du pipe séparés par un double point (:). Pour modifier le format d’affichage de la date, il faut lui passer en paramètre le format à appliquer :

```
<p>Nous sommes le {{today | date:"dd/MM/yyyy"}}</p>
```

- Avec ce template, le composant retournera maintenant : Nous sommes le 07/10/2016.

Les pipes - Les pipes du framework

Pipe	
date	mettre en forme une date
async	prend en entrée une promise ou un observable et en affiche le résultat
uppercase et lowercase	permettent de mettre une valeur respectivement en majuscule et en minuscule.
json	transforme l'élément qui lui est passé en chaîne de caractères le représentant au format json. utile en phase de développement pour afficher graphiquement la valeur d'un objet
decimal	permet de mettre en forme un objet de type number.
percent	permet l'affichage d'un nombre sous la forme d'un pourcentage.
currency	affiche les nombres dans un format monétaire. Il possède un paramètre permettant de spécifier la monnaie à utiliser.

<https://angular.io/guide/pipes> pour une liste complète

Les pipes - Créer un pipe

Afin de faire son propre pipe, il faut créer une classe qui implémente l'interface **PipeTransform**.

Cette interface contient une seule méthode avec la signature suivante :

transform(value: any, ...args: any[]): any

Il est donc nécessaire d'implémenter cette méthode.

Le **premier paramètre est la valeur** sur laquelle est appliqué l'opérateur pipe (|).

Les suivants sont ceux qui sont passés en tant que paramètres lors de l'appel au pipe (séparé par un double point).

Le retour de cette méthode est ce qui sera **utilisé** dans le binding **où est appelé le pipe**.

Les pipes - Créer un pipe

Il est également nécessaire d'ajouter le décorateur **@Pipe** la classe.

Il faut **spécifier le nom de notre pipe** dans ce décorateur.

C'est ce nom qui sera utilisé ensuite dans le template pour faire appel à ce pipe.

```
1  import { Pipe, PipeTransform } from '@angular/core';
2
3  @Pipe({
4    name: 'greeting'
5  })
6  export class GreetingPipe implements PipeTransform {
7
8    transform(value: any, args?: any): any {
9      return null;
10     }
11
12   }
13
```

Les pipes - Créer un pipe

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'greeting'
})
export class GreetingPipe implements PipeTransform {

  transform(value: string, isMale: boolean = true): string {
    if (isMale) {
      return `Bonjour monsieur ${value}`;
    } else {
      return `Bonjour madame ${value}`;
    }
  }
}
```

Dans l'exemple ci-dessous, le pipe retourne une phrase d'accueil en fonction d'un nom. Ce pipe possède également un paramètre optionnel permettant de faire la distinction entre homme et femme.

Les pipes - Filtre

- **Jusqu'à présent**, nous n'avons vu que des **pipes retournant des chaînes de caractères après avoir effectué une transformation**.
- Cependant, la **signature** de la méthode **transform** de l'interface **PipeTransform** nous **permet de retourner n'importe quel type** (any).
- **Les pipes peuvent donc être utilisés autrement que pour la transformation**.
- En effet, **il est possible d'utiliser un pipe pour effectuer un filtre**, en l'associant à un ngFor par exemple.
- Pour cela, il suffit que le pipe prenne en paramètre un tableau et retourne un tableau du même type, mais filtré.

Les pipes - Filtre

```
4  @Pipe({
5    name: 'maleOnly'
6  })
7  export class MaleOnlyPipe implements PipeTransform {
8
9    transform(value: Personne[]): Personne[] {
10     return value.filter(x => x.isMale);
11   }
12 }
```

```
4  @Component({
5    selector: 'app-personnes',
6    template: `
7      <p *ngFor="let personne of personnes | maleOnly">{{ personne.name }}</p>
8    `
9  })
10 export class PersonnesComponent {
11   personnes: Personne[] = [
12     { name: "John", isMale: true },
13     { name: "Sarah", isMale: false },
14     { name: "Michel", isMale: true },
15   ]
16 }
```


Les pipes - Filtre

- Dans l'exemple ci-dessus, on crée un pipe MaleOnlyPipe.
- Ce pipe prend en entrée un tableau de Personne. Ce pipe utilise la méthode filter exposée par le type Array en JavaScript afin de retourner un tableau ne contenant que les objets dont la propriété isMale est vrai.
- Le pipe maleOnly est ensuite utilisé dans le template du composant PersonnesComponent.
- La directive ngFor dans ce template ne va donc itérer que sur les Personne dont le booléen isMale est vrai (ici John et Michel).

Disposition de titre et de contenu avec liste

TP

Lab_Pipe



MODULE 7

Directives

Directive - Qu'est-ce qu'une directive ?

Une directive est un élément du framework qui va interagir directement avec le DOM de notre page HTML.

Elles permettent donc d'associer à un élément HTML un comportement JavaScript.

Il existe trois types de directives.

- **les directives d'attribut :**

- Celles-ci ne modifient pas l'arborescence du DOM. Autrement dit, elles ne vont ni créer ni supprimer des éléments HTML, mais vont agir sur les attributs et les propriétés des balises HTML existantes.

- **les directives structurelles**

- Ces directives modifient directement l'arborescence du DOM HTML. Elles peuvent donc ajouter, modifier ou supprimer des éléments au cours de l'exécution de l'application.

- **les composants**

- En effet les composants sont des directives qui possèdent la spécificité d'être associées à un template.

Directive - Directives communes

- **NgIf**

- NgIf est une directive structurelle.
- Son rôle est de rendre ou non un élément HTML en fonction d'une condition passée en paramètre.

```
<div *NgIf="condition">Hello World</div>
```

- Pour obtenir un comportement similaire, il est possible d'utiliser un *binding* sur l'attribut hidden de l'élément HTML à cacher.

```
<div [hidden]="condition">Hello World</div>
```

- Dans le cas de la directive NgIf, l'élément HTML est supprimé du DOM lorsque la condition n'est pas vraie. Le *binding* sur l'attribut hidden ne fait en revanche que cacher visuellement l'élément.

Directive - Directives communes

NgFor

- NgFor est une directive structurelle.
- Son rôle est d'itérer sur une collection et d'appliquer un template sur chaque élément de cette collection.

```
<ul>  
  <li *ngFor="let item of ['un', 'deux', 'trois']">{{item}}</li>  
</ul>
```

- *Le résultat de ce template est le suivant :*

```
<ul>  
  <li>un</li>  
  <li>deux</li>  
  <li>trois</li>  
</ul>
```

Directive - Qu'est-ce qu'une directive ?

NgStyle

- NgStyle est une directive d'attribut. Elle permet d'appliquer un ou plusieurs styles CSS à un élément HTML.

```
<div [ngStyle]="{color: 'blue'}">Hello World !</div>
```

▪ NgClass

- NgClass est une directive d'attribut qui permet d'ajouter ou de supprimer des classes CSS sur un élément HTML. Il existe plusieurs syntaxes afin d'ajouter ou non des classes via cette directive.

- Il est possible de les lister, séparées par espaces.

```
<div [ngClass]=" 'class1 class2' ">exemple</div>
```

- De les affecter à un tableau JavaScript.

```
<div [ngClass]=" ['class1', 'class2'] ">exemple</div>
```

Directive - Les directives d'attribut

Créer une directive d'attribut

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

@Directive({
  selector: '[appBordure]'
})
export class BordureDirective {

  constructor(
    private el: ElementRef,
    private renderer: Renderer
  ) { }
}
```

Afin de créer une directive d'attribut, il faut créer une classe et lui ajouter:

- le décorateur **@Directive**
- injecter au constructeur
 - la référence à l'élément sur lequel la directive est appliquée `ElementRef`
 - ainsi qu'un *renderer* qui va permettre d'effectuer des opérations sur cet élément.

Directive - Les directives d'attribut

Créer une directive d'attribut

Déclarer cette directive dans le module de l'application

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

Directive - Les directives d'attribut

Créer une directive d'attribut

Voici un exemple de directive qui va ajouter une bordure noire à l'élément sur lequel elle est appliquée :

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

@Directive({
  selector: '[appBordure]'
})
export class BordureDirective {

  constructor(
    private el: ElementRef,
    private renderer: Renderer
  ) {
    this.renderer.setStyle(
      this.el.nativeElement,
      'border',
      '1px solid black'
    );
  }
}
```

Directive - Les directives d'attribut

Créer une directive d'attribut

Pour appliquer cette directive, il suffit d'ajouter l'attribut appBordure à un élément HTML. Par exemple, pour l'instruction suivante :

```
<div appBordure>Hello World !</div>
```

Le résultat est le suivant :

Hello World !

Directive - Les directives d'attribut

Interagir avec les événements du DOM

- Puisqu'une référence à l'élément est injectée dans le constructeur de la directive, il est possible d'ajouter directement des *listeners* (un mécanisme d'écoute d'événements) sur celui-ci.

```
@Directive({
  selector: '[appBordure]'
})
export class BordureDirective {
  constructor(private el: ElementRef, private renderer:
    Renderer) {
    this.el.nativeElement
      .addEventListener('click', () => alert('click !'));
  }
}
```

- Cependant, cela implique de gérer la suppression de ceux-ci lorsque la directive est supprimée afin d'éviter d'éventuelles fuites mémoire.
- De plus, l'une des *best practices* d'Angular est de ne jamais modifier directement les éléments du DOM.

Directive - Les directives d'attribut

Interagir avec les événements du DOM

C'est pour cela que le framework met à disposition un décorateur @HostListener.

Grâce à ce décorateur, il est possible d'ajouter des méthodes à la classe de directive qui ne s'exécuteront que lorsqu'un événement est déclenché sur l'élément sur lequel est appliquée celle-ci.

Ce décorateur prend en paramètre le nom de l'événement auquel la méthode doit se lier.

```
import { Directive, ElementRef, Renderer, HostListener } from
 '@angular/core';

@Directive({
  selector: '[appBordure]'
})
export class BordureDirective {

  ...

  @HostListener('click')
  toggleBorders() {...}

}
```

Directive - Passer des valeurs aux directives d'attribut

Interagir avec les événements du DOM

Il est souvent nécessaire de passer des valeurs aux directives pour les configurer ou avoir un comportement dynamique.

Pour cela, il faut utiliser le décorateur **@Input**.

Si l'on reprend l'exemple précédent en rendant configurable l'épaisseur de la bordure, voici le résultat :

```
@Directive({
  selector: '[appBordure]'
})
export class BordureDirective {

  @Input("appBordure") thickness: number;
  ...
}
```

Le paramètre d'épaisseur utilise la même chaîne de caractères que le sélecteur de la directive. Cela permet de rendre plus simple l'utilisation de la directive. En effet, en un seul attribut, il est possible d'appliquer une directive tout en lui passant des paramètres.

Directive - Passer des valeurs aux directives d'attribut

- Ce qui a pour résultat :



- Cependant, dans cet exemple, si la propriété en Input de la directive change, celle-ci est bien mise à jour, mais les bordures ne sont pas redessinées.
- Il faut donc un moyen d'être notifié du changement de la propriété.
- Pour pallier cela, utiliser : les *setters* et *getters*.
- Un *setter* est une méthode qui s'exécute lorsqu'une modification est effectuée sur une propriété. Ils s'utilisent généralement de la manière suivante :

```
private _thickness: number;

set thickness(thickness: number){
  this._thickness = thickness;
};
```

Directive - Passer des valeurs aux directives d'attribut

- Pour l'exemple précédent, il est possible de se servir d'un *setter* sur la propriété *thickness* afin de redessiner la bordure lorsque celle-ci change.
- La directive devient donc :

```
import { Directive, ElementRef, Renderer, Input, HostListener }
from '@angular/core';

@Directive({
  selector: '[appBordure]'
})
export class BordureDirective {
  ...
  @Input("appBordure") set thickness(thickness: number){
    this._thickness = thickness;
    this.addOrRemoveBorder();
  };

  get thickness(){
    return this._thickness;
  }
  ...
}
```


Directive - Les directives structurelles

La balise <template> et l'astérisque

- Avant toutes choses, il est important de comprendre d'où vient l'astérisque qui préfixe généralement les directives structurelles.
- En effet, ces directives utilisent en réalité un template à appliquer. Par exemple, la directive `NgIf` va rendre ou non un template, `NgFor` va appliquer un template sur une liste d'objets, etc.
- Pour utiliser `NgIf`, il faudrait en réalité utiliser la syntaxe suivante :

```
<template [ngIf]="condition">
  <div>
    Cette div est rendue ou non en fonction de la condition NgIf
  </div>
</template>
```

Directive - Les directives structurelles

La balise <template> et l'astérisque

Cependant, cette syntaxe est un peu lourde.

C'est pourquoi il existe un sucre syntaxique qui permet de l'alléger dans le framework.

C'est l'astérisque qui préfixe l'attribut de ces directives.

En utilisant cette syntaxe sur l'exemple précédent, celui-ci devient :

```
<div *ngIf="condition">  
  Cette div est rendue ou non en fonction de la condition NgIf  
</div>
```

Directive - Les directives structurelles

Créer une directive structurelle

Afin de créer une directive structurelle, comme pour les directives d'attributs il faut créer une classe et lui ajouter le décorateur **@Directive**.

Contrairement aux directives d'attributs, ce n'est pas une référence vers l'élément sur lequel est appliquée la directive qui est injectée dans le constructeur, mais une référence au template de type `TemplateRef` qui va être utilisée.

Le deuxième élément qui est injecté dans le constructeur est une référence vers un conteneur de vue de type `ViewContainerRef`.

Cet objet va nous permettre d'écrire du HTML dans un contexte Angular à l'emplacement de la directive.

Directive - Les directives structurelles

Créer une directive structurelle

Voici comment créer un clone de la directive NgIf.

```
import { Directive, TemplateRef, ViewContainerRef, Input } from
'@angular/core';

@Directive({
  selector: '[appIf]'
})
export class IfCloneDirective {
  constructor(
    private templateRef: TemplateRef<any>,
    private viewContainerRef: ViewContainerRef
  ) { }

  @Input() set appIf(condition: boolean) {
    if (condition) {
      this.viewContainerRef.createEmbeddedView(this.templateRef);
    } else {
      this.viewContainerRef.clear();
    }
  }
}
```

Directive - Les directives structurelles

Créer une directive structurelle

Pour utiliser cette directive, il faut écrire :

```
<template [appIf]="1 == 1">  
  <div>  
    Cette div est rendue ou non en fonction de la condition appIf  
  </div>  
</template>
```

Disposition de titre et de contenu avec liste

TP

Lab_Directive



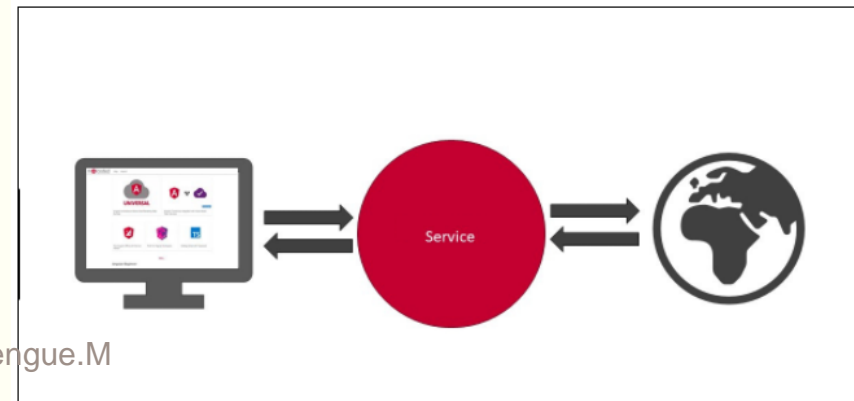
MODULE 8

Service

Service

Qu'est-ce qu'un service ?

- L'objectif d'un service est de contenir toute la logique fonctionnelle et/ou technique de l'application. Au contraire des composants qui ne doivent contenir que la logique propre à la manipulation de leurs données.
- Un des exemples très souvent utilisés est d'encapsuler les interactions avec une API dans un service pour que le composant n'ait pas à se soucier des aspects techniques liés à la communication.
- Un composant consommant un service doit simplement demander des objets et les récupérer. Il ne doit pas avoir à se soucier des mécanismes et transformations mis en place pour récupérer ces données.



Service

Déclarer son service

La déclaration d'un service se fait en créant une **simple classe TypeScript**.

Les **méthodes publiques** de cette classe seront alors **exposées et utilisables** par les autres éléments de l'application.

Si le service est lui-même dépendant d'autres services, il est nécessaire de lui ajouter le décorateur **@Injectable**.

Bonne pratique: le mettre dans tous les cas

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

@Injectable()
export class NumberService{
  getFirstFiveNumbers(): number[]{
    return [0,1,2,3,4];
  }
}
```

Service

Utiliser son service

Il n'est pas souhaitable de créer une instance du service à l'aide du mot-clé **new**.

Il est donc préférable de l'injecter

Pour commencer, il faut:

ajouter un paramètre privé dans le **constructeur** du composant.

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

préciser au framework où trouver ce service

```
@Component({  
  ...,  
  providers: [Service]  
})
```

Service

Rendre son service asynchrone

La plupart du temps, un service va avoir pour rôle de récupérer des données sur un serveur.

Il ne faut cependant pas bloquer le programme pendant la récupération de ces données.

Pour cela, voici **deux méthodes** afin de **rendre** un service **asynchrone**.

- Les promises
- Les observables

Service - Les promises

- Les promises sont un composant de la spécification ECMAScript 6.
- Elles sont des "promesses" qui permettent d'exécuter du code seulement lorsqu'elles ont terminé leur travail.
- Elles fonctionnent avec un **callback** qui va recevoir les informations retournées par ces promises., un **deuxième callback** permettant la gestion des erreurs.

```
promise.then((result) => console.log(result),  
             (error) => console.log(error));
```

- Les promises ne possèdent que trois états.

pending : état indiquant que la promise est en cours d'exécution

fulfilled, lorsque tout s'est bien passé,

rejected lorsqu'une erreur est survenue pendant le traitement.

Service - Les promises

Les promises exposent des méthodes **then** et **catch**.

Ces deux méthodes prennent en paramètre un *callback* qui sera respectivement appelé lorsque la promise passera à l'état "**fulfilled**" ou "**rejected**".

```
3 @Injectable()
4 export class NumberService{
5     private numbers : number[];
6
7     getFirstFiveNumbers(): Promise<number[]>{
8         if(this.numbers){
9             return Promise.resolve(this.numbers);
10        } else {
11            return new Promise( (resolve,reject) => {
12                /* get numbers from server */
13                let numbers = [11,2,3,4,5];
14                this.numbers = numbers;
15                resolve(numbers);
16            });
17        }
18    }
19 }
```

Exemple de service utilisant une promise

Service - Les promises

Pour créer une nouvelle promise, il suffit d'appeler son constructeur.

Celui-ci prend en entrée une méthode à deux paramètres, le *callback* de succès et celui d'erreur.

```
new Promise ((resolve, reject) => resolve("résolue"));
```

- Il existe également quelques *helpers* permettant de retourner un objet simple sous forme de promise.
- Le *helper* `resolve` sert à créer une promise résolue immédiatement à partir d'une valeur.

```
let promise = Promise.resolve("hello world");
```

- Le *helper* `reject` sert à créer une promise en échec immédiatement en fournissant une raison.

```
let promise = Promise.reject("not connected");
```

Service - Les observables

Une autre manière de rendre les services asynchrones est d'utiliser le jeu de bibliothèques *Reactive Extensions for JavaScript* (**RxJS**).

Pour consommer un observable, il faut utiliser la méthode **subscribe**.

Cette méthode prend en paramètre **trois callbacks**.

Le premier callback est celui de succès, dans lequel est reçu le résultat attendu à chaque fois qu'une nouvelle valeur est envoyée.

Le second callback est celui d'erreur. Il n'est appelé que lorsqu'une erreur s'est produite dans le traitement de notre service.

Enfin, **le dernier callback** est celui qui est **appelé dans tous les cas** à la fin du traitement d'un Observable.

Service - Les observables

Exemple de composant consommant un service observable

```
@Component({
  selector: 'numbers',
  templateUrl: 'numbers.html',
  providers: [NumberService]
})
export class NumberComponent {
  numbers: number[];

  constructor(private numberService : NumberService) {
    this.numberService.getFirstFiveNumbers()
      .subscribe(numbers => this.numbers = numbers,
        error => console.log(error),
        () => console.log('finished'));
  }
}
```

Dans cet exemple, le NumberService retourne un observable. Afin de récupérer les données, on se sert donc ici de la méthode subscribe. Dans le callback de succès, on affecte le résultat de l'Observable dans la propriété numbers du composant. Dans celui d'erreur, on affiche l'erreur dans la console. Enfin dans le callback de fin de traitement, on affiche "finished" dans la console.

Service

Exemple de service utilisant une observable

```
5 @Injectable()
6 export class NumberService {
7     private numbers: number[];
8
9     getFirstFiveNumbers(): Observable<number[]> {
10         /* get numbers from server */
11
12         return Observable.create(
13             (observer: Subscriber<number[]>) => {
14                 observer.next([1, 2, 3, 4, 5]);
15             }
16         );
17     }
18 }
```

Disposition de titre et de contenu avec liste

TP

Lab_Service



MODULE 9

Http

Http - Obtenir et envoyer des données

- Afin d'envoyer et de recevoir des données,
- Angular fournit un module **HttpClient**.
- Le module **HttpClient** expose plusieurs méthodes permettant d'envoyer et recevoir des données via des requêtes HTTP.
- Afin d'utiliser ce module, il faut commencer par l'importer dans le module de l'application.

Pour cela, il suffit d'ajouter la classe **HttpClient** dans la propriété imports du décorateur **@NgModule** du module principal de l'application :

```
4 | import {HttpClientModule} from '@angular/common/http';
5 |
6 | @NgModule({
7 |   declarations: [
8 |     ...
9 |   ],
10 |   imports: [
11 |     BrowserModule,
12 |     HttpClientModule
13 |   ],
14 |   providers: [...],
15 |   bootstrap: [...],
16 | })
17 | export class AppModule { }
```

Http - Une API typée

Appel permettant de récupérer une série d'articles

```
Observable<Article[]> articles$ = this.http.get<Article[]>('api.learn-angular.fr/articles');
```

Il est maintenant possible de définir dès le début le type d'objet que l'on attend :

Http - Un support complet des verbes HTTP

Récupération de données

Pour ajouter des paramètres à votre requête pour récupérer un article précis grâce à l'objet **HttpParams** :

```
const params = new HttpParams().set('id', '13');  
  
Observable<Article> article$ = this.http.get<Article>('api.learn-angular.fr/article', {params});
```

Cela aura pour effet d'appeler l'url api.learn-angular.fr/article?id=13

De la même façon, vous pouvons passer des **Headers** spécifiques à votre requête grâce à la propriété headers de ce même objet de configuration :

```
const headers = new HttpHeaders().set('Authorization', 'my-auth-token');  
  
Observable<Article[]> article$ = this.http.get<Article[]>('api.learn-angular.fr/articles', {headers});
```

Http - Envoi de données

Pour l'envoi de données à votre backend, la méthode **post()** est toujours préconisée. Grâce à elle, vous allez pouvoir envoyer un simple objet JavaScript. Voyons cela :

```
const body = {title: 'HttpClient'};

this.http
  .post('api.learn-angular.fr/article/add', body)
  .subscribe(...);
```

Comme pour la méthode `get()`, le second paramètre vous permet de passer des paramètres et des *headers* à votre requête.

Http - Récupération des erreurs

Les appels HTTP sont source de nombreuses erreurs dues à l'usage du réseau, du *backend* et même du *frontend*.

Il est donc nécessaire de gérer ces cas grâce à la callback d'erreur de la souscription de votre Observable.

```
this.http.get<Article>('api.learn-angular.fr/articles/12').subscribe(  
  data => {  
    console.log("Titre: " + data.title);  
  },  
  (err: HttpResponse) => {  
    if (err.error instanceof Error) {  
      // A client-side or network error occurred. Handle it accordingly.  
      console.log('An error occurred:', err.error.message);  
    } else {  
      // The backend returned an unsuccessful response code.  
      // The response body may contain clues as to what went wrong,  
      console.log(`Backend returned code ${err.status}, body was: ${err.error}`);  
    }  
  }  
);
```


Disposition de titre et de contenu avec liste

TP

Lab_Http

<https://jsonplaceholder.typicode.com/>



MODULE 10

Formulaire

Formulaire

- Avec Angular, il est possible de créer des formulaires avec des contrôles utilisateurs qui vont utiliser le *data binding* (liaison de données).
- Ce *data binding* va permettre de synchroniser les données entre les composants et les vues.
- Une modification d'une propriété d'un composant sera automatiquement reflétée sur l'interface, et inversement une modification d'un contrôle HTML mettra à jour les propriétés du composant.
- Dans la suite du chapitre, le formulaire permettra l'édition d'une entité de type Product, avec les propriétés suivantes :

```
export class Product {  
  id: number;  
  name: string;  
  category: string;  
}
```

Formulaire - Créer un composant formulaire

Le composant

- Comme un composant classique, un composant formulaire contient deux parties.
 - **la vue**, qui est composée du template HTML et du style CSS,
 - la deuxième, quant à elle, est **le code** lui-même.
- Ce code va gérer les données et les interactions de l'utilisateur.

```
export class ProductFormComponent implements OnInit {  
  
  categories = ["Légumes", "Fruits", "Viandes"];  
  product: Product;  
  
  ngOnInit() {  
    this.product =  
      {  
        id: 0,  
        category: this.categories[0],  
        name: "Produit A"  
      };  
  }  
}
```

Formulaire - La vue et le data binding

La syntaxe ngModel

```
<input type="text" name="property" [(ngModel)]="model.property" />
```

ngModel en détail

- Pour comprendre exactement ce qui se passe lorsqu'on utilise l'instruction `[(ngModel)]`, il faut décortiquer la structure de sa syntaxe.
- **Les deux crochets** représentent le *property binding* (la liaison à la propriété). Cela signifie que la valeur circule depuis le du modèle vers la vue. C'est tout simplement le **one-way (unidirectionnel) data binding** du modèle vers la vue.
- Les deux parenthèses, quant à elles, représentent l'*event binding* (la liaison à l'événement). C'est l'inverse du *property binding*. C'est un **one-way data binding** de la vue vers le modèle dans ce cas.
- Ainsi, en utilisant les deux ensembles, un *two-way data binding* est créé et le flux circule dans les deux sens.

Formulaire - La vue et le data binding

L'utilisation de ngModel dans un cas concret

```
<div class="product-container">
  <h1>Mon Formulaire</h1>
  <form>
    <div class="group">
      <label for="name">Name</label>
      <input type="text" id="name" name="name" required [(ngModel)]="product.name">
    </div>
    <div class="group">
      <label for="category">Catégorie</label>
      <select id="category" name="category" [(ngModel)]="product.category">
        <option *ngFor="let category of categories" [value]="category">{{category}}</option>
      </select>
    </div>

    <button type="submit">Submit</button>
  </form>
</div>
```

Nous pouvons observer, ici, un premier groupe qui utilise ngModel pour lier la valeur de l'input avec la propriété du composant product.name.

Formulaire - La vue et le data binding

L'utilisation de ngModel dans un cas concret

Dans le deuxième groupe, c'est sur le select que la propriété `product.category` est lié, toujours avec `ngModel`.

Ensuite, un `ngFor` permet de générer toutes les option de ce select en bouclant sur la liste `categories`.

Chaque option possède en valeur avec `[value]` la catégorie qui sera appliquée au `ngModel` si l'option est sélectionnée.



The screenshot shows a web browser window with the address bar displaying 'localhost:4200'. The page content includes a title 'Mon Formulaire' in a large, bold, black font. Below the title, there are two form fields: a text input labeled 'Name' containing the text 'Produit A', and a dropdown menu labeled 'Catégorie' with 'Légumes' selected and a downward arrow. Below these fields is a 'Submit' button. In the bottom right corner of the browser window, the text 'Mbengue.M' is visible.

Formulaire - Les états et la validité d'un champ

Les états d'un input

- **Utiliser ngModel apporte**, en plus du *two-way data binding*, **la connaissance des états d'un input à un instant donné.**
- Par exemple,
 - si un champ n'a jamais été visité, c'est-à-dire si l'utilisateur n'a jamais interagi avec celui-ci. Angular va ajouter au champ la classe `ng-untouched`.
 - À l'inverse, si l'utilisateur a interagi avec le champ, ce sera la classe `ng-touched` qui sera appliquée.
- Si le champ est **valide**, la classe **ng-valid** sera présente, si le champ est **invalide**, ce sera la classe **ng-invalid**.

Formulaire - Les états et la validité d'un champ

Les états d'un input

- Angular apporte également une gestion du "**dirty**", c'est-à-dire qu'en plus de la validité et la notion de "*touched*", l'application sait si le champ, "*touched*" ou "*untouched*", a été modifié.
- **Lorsque le champ est modifié**, la classe **ng-dirty** est appliquée, autrement c'est la classe **ng-pristine** qui l'est.
- Voici un tableau qui récapitule ces informations :

Formulaire - Les états et la validité d'un champ

Les états d'un input

Voici un tableau qui récapitule ces informations :

Condition	Classe si la condition est vraie	Classe si la condition est fausse
Le champ est valide	ng-valid	ng-invalid
Le champ est modifié	ng-dirty	ng-pristine
Il y a eu interaction avec le champ	ng-touched	ng-untouched

Formulaire - Les états et la validité d'un champ

Styliser selon la validité

Angular met à disposition les classes ng-valid et ng-invalid. À partir de là, il est très facile de styliser ses formulaires en conséquence.

```
.ng-valid{  
    border: solid #82af3c 1px;  
}  
  
.ng-invalid {  
    border: solid red 1px;  
}
```

Avec ce style, lorsque ng-valid est présent, une bordure verte sera ajoutée au contrôle. À l'inverse, si ng-invalid est présent, ce sera une bordure rouge qui sera ajoutée.

Formulaire – Soumettre le formulaire

La syntaxe de la directive est la suivante

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

Ainsi, la méthode onSubmit sera appelée lorsque le formulaire sera soumis.

Il faut donc ajouter cette méthode dans le composant :

```
onSubmit() {  
  | console.log(`Le produit ${this.product.name} a été soumis, il appartient à la catégorie ${this.product.category}`);  
  }  
}
```

Formulaire – FormControl

- Les formulaires Angular fonctionnent principalement avec deux types de composants : les FormControl (contrôles) et les FormGroup (groupes de contrôle).
- Dans les exemples précédents, c'est uniquement ngModel qui était réellement utilisé dans les formulaires, puis Angular s'occupe du reste sans que le développeur s'en soucie.
- L'objectif pour la suite est de prendre la main sur ces entités de formulaire et de les manipuler nous-mêmes, ce qui permet d'aller plus loin.

Formulaire – FormControls

- **Les contrôles sont des entités qui possèdent une valeur et un statut de validité**, qui est déterminé par une fonction de validation qui, elle, est optionnelle.
- **Un contrôle peut être lié à un champ**, et à sa création, il prend trois paramètres, toutes optionnelles.
 - Le premier paramètre est la valeur par défaut,
 - le second est un validateur
 - et le troisième est aussi un validateur, mais asynchrone.

Formulaire – FormControl

Pour utiliser un contrôle, il faut donc l'instancier dans un premier temps.

Ensuite, il faut **ajouter** les entités **FormGroup** et **FormBuilder** d'Angular, des classes qui permettent de manipuler les **FormControl**.

Le FormGroup et ses FormControl vont être stockés en propriété du composant, pour pouvoir les lier à la vue.

```
export class ProductFormComponent{

  login: FormControl;
  email: FormControl;

  loginForm: FormGroup;

  constructor(private builder: FormBuilder) {
    this.login = new FormControl('', [Validators.required, Validators.minLength(4)]);
    this.email = new FormControl('', Validators.required);

    this.loginForm = builder.group({
      login: this.login,
      email: this.email
    });
  }
}
```

Mbengue.M

Formulaire – FormControl

Cela change donc la vue HTML en conséquence.

Ce n'est plus **[(ngModel)]** qui va être utilisé mais **[formGroup]** et **[formControl]**.

```
<form [formGroup]="loginForm">
  <div class="group">
    <label for="login">Login</label>
    <input required type="text" [formControl]="login" />
    <div *ngIf="login.dirty && !login.valid">
      <p *ngIf="login.errors.minlength">
        | Le login doit être composé d'au moins 4 caractères !
      </p>
    </div>
  </div>
  <button type="submit">Submit</button>
</form>
```


Formulaire – Les validateurs intégrés

Angular propose trois validateurs par défaut qui peuvent autant être appliqués en utilisant un `FormControl` que par le biais de propriétés HTML.

- `required`
- `minLength`
- `maxLength`

Pour afficher un message d'erreur associé en cas de non-respect des validateurs, il suffit d'aller chercher l'information dans le **FormControl** associé, dans la propriété **`errors.minlength`**, par exemple.

Formulaire – Les validateurs intégrés

Exemple: si le login a été modifié et qu'il n'est pas valide (car la règle qui demande quatre caractères au moins n'est pas respectée), nous allons afficher le message correspondant.

```
<form [formGroup]="loginForm">
  <div class="group">
    <label for="login">Login</label>
    <input required type="text" [formControl]="login" />
    <div *ngIf="login.dirty && !login.valid">
      <p *ngIf="login.errors.minlength">
        Le login doit être composé d'au moins 4 caractères !
      </p>
    </div>
  </div>
  <button type="submit">Submit</button>
</form>
```

En accédant à la propriété `errors` du FormControl `login`, il est possible de savoir si tel ou tel validateur est en erreur. Si c'est le cas, une propriété du nom du validateur existera sur cet objet `errors`. C'est pourquoi le paragraphe dans l'exemple utilise un `ngIf` sur `login.errors.minlength` pour afficher un message d'erreur.

Formulaire – Créer un validateur personnalisé

Il est tout aussi possible de créer un validateur soi-même. Pour cela, il suffit de créer une classe qui va posséder une méthode statique, voire plusieurs.

Cette méthode doit prendre en paramètre un FormControl, qui sera lui-même injecté au moment de la validation.

```
interface ValidationResult {  
    [key: string]: boolean;  
}  
  
class LoginValidator {  
    static containsAdmin(control: FormControl): ValidationResult {  
        if (control.value.indexOf('admin') != -1) {  
            return { "containsAdmin": true };  
        }  
  
        return null;  
    }  
}
```

La méthode statique containsAdmin représente un validateur. Elle possède un paramètre control qui est le FormControl injecté automatiquement par Angular.

La méthode doit renvoyer un **ValidationResult** avec un booléen vrai si la valeur du contrôle possède bien le texte "admin". Si ce n'est pas le cas, c'est null qui est renvoyé.

Disposition de titre et de contenu avec liste

TP

Lab_Form



MODULE 11

Routes

Routes - Définir les routes d'une application

- La navigation au sein d'une application Angular s'effectue grâce à un système de route qui doit être définie.
- Une route dans sa forme la plus basique est l'association d'un composant et d'une URL.
- Lorsque cette URL est demandée, le module de routage effectue le rendu du composant associé.
- Pour effectuer cette association d'une URL à un composant, il faut définir une variable de type Routes.
- Ce type est déclaré comme étant un tableau dont chaque élément est un objet de type Route.
- Une route est donc composée d'un attribut path qui représente l'URL (relative ou absolue) associée à cette route et d'un attribut component qui est le composant à charger lorsque cette route est appelée.

Routes - Définir les routes d'une application

▪ Exemple de routes

```
7 export const ROUTES: Routes = [  
8   { path: '', redirectTo: 'home', pathMatch: 'full' },  
9   { path: 'home', component: HomeComponent },  
10  { path: 'manager', component: ManagerComponent },  
11  { path: 'admin', component: AdminComponent },  
12  { path: 'about', component: AboutComponent }  
13];
```

- Il est également possible d'effectuer des redirections.
 - Pour cela, au lieu d'utiliser l'attribut `component` de l'objet `Route`, il faut utiliser l'attribut `redirectTo`.
 - Lorsque cet attribut est utilisé, il est également nécessaire de remplir l'attribut **`pathMatch`**.
 - Cet attribut peut prendre la valeur **`full`**, dans ce cas, le routeur ne fera la redirection que si l'URL appelée est exactement la même que celle définie dans l'attribut `redirectTo`.
 - L'autre valeur possible pour cet attribut est **`prefix`**: le routeur effectuera la redirection si l'URL appelée commence comme celle définie dans l'attribut `redirectTo`.

Routes - Le rendu de composant

- Afin que le routeur puisse effectuer le rendu du composant, il faut en tout premier lieu enregistrer les routes déclarées dans le tableau ROUTES.

```
import { RouterModule } from '@angular/router';
import { ROUTES } from './app.routes';
import ...

@NgModule({
  declarations: [
    ...
  ],
  imports: [
    BrowserModule,
    RouterModule.forRoot(ROUTES)
  ],
  providers: [...],
  bootstrap: [...]
})
export class AppModule { }
```

- Dans l'exemple de déclaration de module ci-dessus, l'import du module **RouterModule** se fait en utilisant un objet de type Routes venant du fichier **app.routes.ts**.

Routes - Le rendu de composant

Il est maintenant nécessaire de préciser au module de routage où il doit faire le rendu des composants associés aux routes.

Pour cela, il faut utiliser la directive RouterOutlet dans le composant principal de l'application, en ajoutant la balise **router-outlet** dans le template.

Le rendu du composant de la route active se fera donc au niveau de cette balise :

```
<header>
  <h1> Pomme app !</h1>
</header>
<router-outlet></router-outlet>
<footer>@2017</footer>
```

Routes - Naviguer dans son application

Afin de naviguer dans une application en utilisant le module de routage, il existe deux solutions.

- **Premièrement**, il est possible de naviguer en utilisant des liens hypertextes HTML (balise « a »). Pour ce faire, il faut utiliser la directive **RouterLink**.

- Exemple:

```
1 <ul class="nav nav-pills">
2   <li><a routerLink="/home" routerLinkActive="active" >Home</a></li>
3   <li><a routerLink="/manager" routerLinkActive="active">Manager</a></li>
4   <li><a routerLink="/admin" routerLinkActive="active">Admin</a></li>
5   <li><a routerLink="/about" routerLinkActive="active">About</a></li>
6 </ul>
7 <router-outlet></router-outlet>
```

- Il est également possible d'utiliser l'attribut natif href de la balise HTML « a » cependant, cela impliquerait un chargement de la page, rendant l'expérience utilisateur moins agréable.

Routes - Naviguer dans son application

- **La deuxième manière** d'effectuer une navigation est de le faire directement dans le code d'un composant.
- Pour cela, il faut injecter une instance de l'objet Router au sein du composant.
- Cet objet dispose d'une méthode navigate qui prend exactement le même paramètre que la directive RouterLink.

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

@Component({...})
export class PommeDetailsComponent{

  pomme: Pomme;
  constructor(private router: Router) { }

  goBack() {
    this.router.navigate(['pommes']);
  }
}
```

Routes - Récupération des données de routage

- Afin d'obtenir des informations concernant la route active au sein du composant, il faut injecter l'objet **ActivatedRoute**.
- Cet objet contient notamment des informations sur les paramètres passés à la route lors de la navigation.
- Pour récupérer ces informations, il est possible de passer par la propriété snapshot de cet objet.
- Cette propriété contient elle-même une propriété **params** contenant tous les paramètres passés à la route lors de la navigation.

Routes - Récupération des données de routage

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

@Component({...})
export class PommeDetailsComponent implements OnInit {

  pomme: Pomme;
  constructor(private pommeService: PommeService,
               private route: ActivatedRoute,
               private router: Router) { }

  ngOnInit() {
    const id = +this.route.snapshot.params['id'];
    this.pommeService.getPomme(id)
      .subscribe(pomme => this.pomme = pomme);
  }

  goBack() {
    this.router.navigate(['pommes']);
  }
}
```

- Dans l'exemple, la classe `ActivatedRoute` est injectée dans le constructeur du composant. Dans la méthode d'initialisation, on récupère le paramètre `'id'` dans l'objet `snapshot.params` de l'`ActivatedRoute`. Le symbole « + » au niveau de l'affectation à la constante `id` sert à caster le paramètre `id` en `number`. En effet, les paramètres de routage sont passés sous forme de chaînes de caractères.

Routes - Guard

Routes -

Routes -

Disposition de titre et de contenu avec liste

TP

Lab_Routes