



Angular



Haithem KHALIFA

Senior Microsoft Consultant

Formateur C# / ASP.NET / Azure / Angular



<https://www.linkedin.com/in/haithem-khalifa-51356865/>

Un petit tour de table :)

Votre nom + Votre expérience et vécu NG + Vos attentes + Des projets ?

Javascript ?

Typescript ?

Rxjs ?

Angular ?

Déroulement du cours N'hésitez pas à interrompre
ou à intervenir Si un chapitre ne vous intéresse
pas, on peut le sauter

Introduction

Programm

1. e Présentation d'Angular
2. e ECMAScript 6
3. TypeScript
4. Angular CLI
5. Les composants Angular
6. Les templates
7. Les pipes
8. Les Observables
9. Plus loin dans les composants
10. Les services
11. Les formulaires
12. Le routing
13. HTTP
14. Migration de AngularJS vers Angular

Périmètre d'Angular

L'objectif d'Angular est de devenir le framework n°1 du web

- Multi-plateforme
- Développements mobiles (le framework *Ionic* se base sur Angular)

Objectifs (par rapport à AngularJS) :

- Amélioration des performances
- Diminuer l'utilisation de frameworks tiers (dépendances et librairies sous AngularJS)

Renforcer la présence sur mobile

Capitaliser sur la réputation d'AngularJS pour développer des applications robustes, performantes et simples à coder

Utilisation des nouveaux standards

- TypeScript avec l'utilisation des objets

Angular 2, Angular 4, Angular 5, Angular 6, Angular

Un historique

- 14/09/2016 : sortie de Angular 2.0.0
- 20/12/2016 : sortie de Angular 2.4.0
- 23/03/2017 : sortie de Angular 4.0.0
- 15/09/2017 : sortie de Angular 4.4.1
- 01/11/2017 : sortie de Angular 5.0.0
- 03/05/2018 : sortie de Angular 6.0.0
- 18/10/2018 : sortie de Angular 7.0.0 - Version LTS
- 28/05/2019 : sortie de Angular 8.0.0
- 06/02/2020 : sortie de Angular 9.0.0
- <https://github.com/angular/angular/blob/master/CHANGELOG.md>

Utilisation du versionnage sémantique pour numéroter chaque nouvelle version

Des cycles de sorties planifiés à l'avance

- <https://angular.io/guide/releases>

Une politique de dépréciation pour se préparer aux fonctionnalités qui vont disparaître Une distinction claire entre les APIs stables et expérimentales

Versionnage sémantique

Les chiffres d'un numéro de version sont chargés de sens

Une version donnée d'Angular comporte 3 chiffres (ex : angular 5.1.1).

Ces trois chiffres évoluent de manière à refléter le type de changements inclus dans chaque version :

- Les patches releases ne modifient pas les fonctionnalités d'Angular. Dans le numéro de version, elles ne changent que le dernier chiffre.
- Les versions mineures ne contiennent que des fonctionnalités supplémentaires. Elles modifient le chiffre du milieu.
- Les versions majeures contiennent des changements non rétro-compatibles et modifient le premier chiffre.

Pas de version 3 ?

Le module de routing d'Angular était déjà en version 3 alors qu'Angular était toujours en version 2

Les équipes ont donc décidé d'uniformiser les numéros de version et ainsi de passer directement à la version 4

Donc pas de version 3 d'Angular

On ne parle plus d'Angular 2 ou d'Angular 5

Le framework est désigné sous le simple nom d'Angular

Dates clés - Angular

Angular 2 – Septembre 2016

Angular 4 – Mars 2017

Angular 5 – Novembre 2017

Angular 6 – Mai 2018

Angular 7 – Octobre 2018

Angular 8 – Mai 2019

Angular 9 – Février 2020

Angular 10 – Juin 2020 ?

À noter : les deux frameworks AngularJS et Angular ont chacun un site et une documentation bien séparés :

- `angularjs.org` pour AngularJS
- `angular.io` pour Angular

Présentation

Framework JavaScript pour créer des applications monopages (SPA - *Single Page Application*), web et mobiles

Plusieurs langages supportés (ES5, ES6, TypeScript, Dart)

Angular est un framework complet:

- Inclut toutes les briques nécessaires à la création d'une appli professionnelle : routeur, requêtage, HTTP, gestion des formulaires, internationalisation,...
- Modulaire : Le framework lui-même est découpé en sous-paquets correspondant aux grandes aires fonctionnelles (core, routeur, http,...)
- Rapide

Angular adopte une approche composant

- Un composant est la brique de base d'une application Angular

Les applications monopages(SPA)

- Angular permet de développer des applications Web de type SPA.
- Une SPA(Single Page Application) est une application web accessible via une page web unique.
- Le but est d'éviter le chargement d'une nouvelle page à chaque action demandée et d'améliorer ainsi l'expérience utilisateur (meilleure fluidité)
- La différence entre une SPA et un site web classique réside dans leur structure et dans la relation qu'ils établissent entre le navigateur et le serveur:
 - Une SPA est donc composée d'une seule page.
 - Le rôle du browser (front-end) est beaucoup plus important : toute la logique applicative y est déportée.
 - Le serveur (back-end) est "seulement" responsable de la fourniture des ressources à l'application et surtout de l'exposition des données.

SPA: Pourquoi on en parle ?

- Les frameworks JS/TS comme Angular participent à la popularité des SPA.
- Les SPA s'appuyant sur de tels framework sont en général comme avantage d'être:
 - Testables (unitairement et fonctionnellement)
 - Fluides (pas de rechargement d'url etc)
 - Bien organisées
 - Maintenables et évolutives...

Différences avec AngularJS

Disparition des contrôleurs et de tout ce qui est `scope`

Approche composant (*component*), complètement objet (utilisation de `this`)

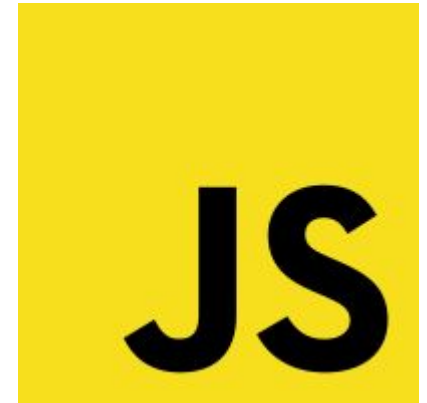
Les services deviennent de simples classes (plus de notion de `factory`) qui exposent des méthodes et sont utilisées par les composants

Remplacement des filtres par la notion de pipes

Les composants forment la représentation partielle d'une page

- Complètement autonome
- Ancienne directive AngularJS associée à un template
- Des propriétés (des entrées) et des évènements (des sorties)
- Un cycle de vie

Utilisation de TypeScript



ES6

ECMAScript 6

La version 5 d'ECMAScript a été publiée en décembre 2009

C'est encore la version la plus utilisée aujourd'hui

La version 6 d'ECMAScript a été publiée en juin 2015

- On parle d'ECMAScript 6, ES6, ES2015 ou encore ECMAScript 2015

La compatibilité des moteurs (et donc des navigateurs) évolue pour être compatible avec ES6

- Table de compatibilité : <http://kangax.github.io/compat-table/es6/>

Un transpileur peut être utilisé pour convertir du code ES6 en code ES5

- Babel (<https://babeljs.io/>)
- Traceur (<https://github.com/google/traceur-compiler>)

Les nouveautés ES6 - let

L'utilisation de `let` permet de déclarer une nouvelle variable

Contrairement à `var`, il n'y a plus de hoisting avec `let`

On retrouve ainsi le fonctionnement classique de nombreux langages de programmation

Utiliser `let` au maximum et oublier `var`

```
function calculateVar(n1, n2, op) {  
  if(op === '+')  
    var res = n1 +  
n2;  else if(op ===  
  '-')  
    var res = n1 - n2  
  
  return res;  
}
```

OK

```
function calculateLet(n1, n2, op) {  
  if(op === '+') {  
    let res2 = n1 + n2;  
  }  
  else if(op === '-') {  
    let res2 = n1 -  
n2;  
  }  
  return res2;  
}
```

Erreur

Les nouveautés ES6 - const

Comme son nom l'indique, `const` permet de définir une constante dans un bloc (comme avec `let`, il n'y a pas de *hoisting*)

Une constante doit être initialisée et ne pourra pas être modifiée

Si une constante contient un objet, on retrouve le même comportement qu'avec les langages objets :

- On ne peut pas modifier la constante
- Les attributs de l'objet restent modifiables

```
const maxWeight = 120;  
maxWeight = 80; // ERREUR
```

```
const client = { nom: 'Patrick'};  
client = { nom: 'André'}; // ERREUR
```

```
const client = { nom: 'Patrick'};  
client.nom = 'André'; // OK
```

Les nouveautés ES6 –

Template de string

En ES5

```
let nomComplet = 'Mr ' + prenom + ' ' + nom;
```

En ES6, on peut utiliser les templates de string en utilisant le caractère backtick ``` (Alr Gr + 7)

```
let nomComplet = `Mr ${prenom} ${nom}`;
```

Les templates de string supportent le multi-ligne, ce qui sera très utile avec Angular

```
let nomComplet = `

<h1>Bonjour</h1>
</div>`;


```

Les nouveautés ES6 - Objets

La création d'objet est simplifiée en ES6

```
function getClient() {  
    let prenom = 'Patrick';  
    let nom = 'Foucault';  
    return { prenom, nom };  
}
```

Les nouveautés ES6 - Fonctions

ES6 permet de mettre des valeurs par défaut aux paramètres des fonctions

```
function add(a, b = 1) {  
    return a + b;  
}
```

Les valeurs par défaut peuvent être des appels de fonction, d'autres variables ou encore d'autres paramètres de la même fonction (à condition de respecter l'ordre de définition)

```
function add(a, b = a + 1)  
{  
    return a + b;  
}
```

Les valeurs par défaut peuvent être utilisées dans les affectations par décomposition

```
let {prenom, nom = 'Foucault'} = client;
```

Les nouveautés ES6 –

Paramètre du reste

Les paramètres du reste permettent de définir un nombre indéfini d'arguments

Cela évite l'utilisation d'`arguments` avec ES5

```
function add(...nombres)
{
  let somme = 0;
  for( let nombre of nombres)
    {
      somme += nombre;
    }
  return somme;
}
```

Les nouveautés ES6 – Arrow functions

Les *arrow functions* se révèlent très pratiques pour les *callbacks*

```
let nombres = [1, 2, 3, 4, 5, 6, 7, 8, 9];

let es5Filtering = nombres.filter(function(number) {
  return number % 2;
});

let es6Filtering = nombres.filter(number => number % 2);
```


Les nouveautés ES6 –

Affectations déstructurées

Syntaxe ES5

```
var client = { prenom: 'Patrick', age: 45 };  
...  
var clientPrenom = client.prenom;  
var clientAge = client.age;
```

Syntaxe ES6 avec les affectations déstructurées

```
let client = { prenom: 'Patrick', age: 45 };  
...  
let { prenom , age} = client;
```

Attention à ne pas se tromper dans l'ordre (la clé est la propriété de l'objet et la valeur le nom de la variable à affecter)

Les nouveautés ES6 –

Affectations déstructurées

Il est possible d'écrire plus simplement

```
let client = { prenom: 'Patrick', age: 45 };  
...  
let { prenom, age } = client;
```

Cela crée deux variables `prenom` et `age` contenant les valeurs des attributs de même nom dans l'objet

On a le même fonctionnement avec les objets imbriqués

```
let client = { identite: { prenom: 'Patrick' }, age: 45 };  
...  
let { identite: { prenom }, age } = client;
```

Cela crée deux variables `prenom` et `age`

Les nouveautés ES6 –

Affectations déstructurées

La même chose est possible avec des tableaux

```
let prenom1 = 'Patrick', prenom2 = 'Jean-Marc', prenom3 = 'Blandine';  
...  
let [prenom1, prenom2] = prenom1;
```

prenom1 a pour valeur 'Patrick' et prenom2 a pour valeur 'Jean-Marc'

Les nouveautés ES6 –

Affectations déstructurées

Grâce aux affectations déstructurées, il devient possible de faire retourner plusieurs valeurs à une fonction

```
function getClient() {  
    let prenom = 'Patrick'; let age = 50;  
    return { prenom, age };  
}
```

Il est alors possible d'écrire

```
let { prenom, age } = getClient();
```

Les nouveautés ES6 –

Promises

On retrouvait déjà les Promises dans AngularJS

L'idée des Promises est de clarifier la programmation asynchrone en n'utilisant plus les fonctions de callback qui rendent rapidement le code illisible

Le code devient un code à plat

```
getClient(id)
    .then(function(client) {
        return getCommandes(client);
    })
    .then(function(commandes) {
        return getCA(commandes);
    })
```

Explication : récupération d'un utilisateur selon son id, puis récupération de ses commandes, puis calcul du chiffre d'affaire généré par ces commandes

Les nouveautés ES6 –

Promises

Une promise est un objet **thenable**, (il a une méthode `then`)

La méthode `then` prend 2 arguments :

- un callback de succès
- un callback d'erreur

Une promise a 3 états :

- **pending** ("en cours") : quand la promise n'est pas réalisée (appel serveur pas encore terminé)
- **fulfilled** ("réalisée") : quand la promise s'est réalisée avec succès (appel HTTP serveur a retourné un status 200-OK)
- **rejected** ("rejetée") : quand la promise a échoué (appel HTTP serveur a retourné un status 404-NotFound)

Le callback de succès est invoqué quand la promise est bien réalisée (**fulfilled**)

- En argument on obtient le résultat

Le callback d'erreur est invoqué quand la promesse est rejetée (**rejected**)

- En argument on obtient la valeur rejetée ou une erreur

Les nouveautés ES6 –

Promises

Pour créer une promise, on instancie la classe `Promise` avec comme paramètre une fonction qui accepte le callback `resolve` et le `reject` callback

```
let getClient = function (id) {  
    return new Promise(function(resolve, reject) {  
        ...  
        if(response.status === 200)  
            resolve(response.data);  
        else  
            reject('Non existant');  
    });  
};
```

On peut avoir une gestion d'erreur globale sur une chaîne de Promise grâce à la méthode `catch`

```
getClient(id)  
    .then(function(client) { return getCommandes(client); })  
    .then(function(commandes) { return getCA(commandes); })  
    .catch(function (error) { console.log(error); })
```

Les nouveautés ES6 – Promises

L'ECMAScript 8 (2017) introduit officiellement la notion de constructions `async / await` liées aux promesses

```
function resolveAfter2Seconds() {  
    return new Promise(resolve => {    setTimeout(()  
        => {  
            resolve('resolved');  
        }, 2000);  
    });  
}  
  
async function asyncCall() {  
    console.log('calling');  
    const result = await resolveAfter2Seconds();  
    console.log(result);  
}  
  
asyncCall();
```


Les nouveautés ES6 - Classes

ES6 permet enfin la définition de classes en JavaScript !

```
class Personne {  
    constructor(nom, prenom) {  
        this.nom = nom;  this.prenom =  
        prenom;  
    }  
  
    getNomComplet() {  
        return this.nom + ' ' + this.prenom;  
    }  
}  
  
let p1 = new Personne('Foucault', 'Patrick');  
console.log(p1.getNomComplet());
```

Les nouveautés ES6 - Classes

Les méthodes `static` sont associées à la classe et non aux instances

```
class Voyageur {  
    ...  
    static getStandardWeight() {  
        return 75;  
    }  
}  
  
console.log(Voyageur.getStandardWeight());
```

Les nouveautés ES6 - Classes

Des accesseurs et des mutateurs peuvent être définis simplement

```
class Personne {  
    ...  
    get age() {  
        console.log('in get age()'); return  
        this._age;  
    }  
    set age(newAge) {  
        console.log('in set age(newAge)'); this._age  
        = newAge;  
    }  
}  
...  
console.log(p1.age);  
p1.age = 60;  
console.log(p1.age);
```

Les nouveautés ES6 - Classes

L'héritage est grandement simplifié

```
class Client extends Personne {  
  constructor(nom, prenom, age, ca) {  
    super(nom, prenom, age);  
    this.ca = ca;  
  }  
}
```

L'utilisation du mot-clé `super` permet d'accéder à la classe mère

Les nouveautés ES6 – Modules

Les modules permettent de ranger ses fonctions

Dans le fichier du module, il faut utiliser `export`

```
export function sayBonjour(client) { console.log(`Bonjour  
    ${client.nom}`);  
}  
export function sayBonsoir(client) { console.log(`Bonsoir  
    ${client.nom}`);  
}
```

Dans le fichier qui utilise le module, il faut utiliser `import` pour importer les fonctions

```
import { sayBonjour, sayBonsoir as bonsoir } from './11_modules_ex';  
...  
sayBonjour(client);
```

Il est possible de tout importer avec

```
import * from './11_modules_ex';
```

TP - Transformer en utilisant au maximum les nouveautés d'ES6

```
function getCategorie(age) {
  if (age >= 18) {
    var categorie = "Majeur";
  } else {
    var categorie = "Mineur";
  }

  var resultat = "La personne est " + categorie;
  return resultat;
}

var resultat = getCategorie(19);
console.log(resultat);

function processAnnee(annee, cb) {
  var age = new Date().getFullYear() - annee; return cb(age);
}

processAnnee(1990, function (age) {
  console.log("Tu as " + age + " ans");
})
```



TypeScript

TypeScript

TypeScript, développé par Microsoft, est apparu en 2012 et constitue un sur-ensemble de JavaScript

TypeScript supporte ES6

Lors du développement , les fichiers TypeScript (extension `.ts`) sont transpilés en JavaScript (généralement ES5)

Installation du transpileur TypeScript

```
npm install -g typescript
```

Exemple d'utilisation

```
tsc --init
```

```
tsc --watch
```


TypeScript – Les types

TypeScript permet de typer ses variables

Syntaxe

```
let variable: type;
```

Exemples

```
let age: number = 25;  
const prenom: string = 'Patrick';  
let client: Client = new Client();  
let clientele: Client[] = [new Client()];
```

Grâce aux types, le compilateur pourra nous avertir en cas d'erreur

TypeScript propose le type `any` pour spécifier une variable sans

type particulier

```
let var: any = 'Californie';
```

Il est possible de combiner les types

```
let n: number|boolean = 51;
```

TypeScript - enum

Il est possible de créer des énumérations avec TypeScript

En réalité, derrière les énumérations se cache une valeur numérique qui commence par 0

```
enum Constructeur { Airbus, Boeing, Embraer }  
let avion = new Avion();  
avion.constructeur = Constructeur.Airbus;
```

TypeScript - Types de retour

Lors de la déclaration des fonctions, il est possible de spécifier le type de retour de la fonction

```
function getNom(): string {  
    return 'Patrick';  
}
```

Le mot-clé `void` est utilisé si la fonction ne retourne rien

```
function setNom(nom: string): void {  
    this.nom = nom;  
}
```

TypeScript - Paramètres optionnels

En TypeScript, les paramètres d'une fonction sont obligatoires, contrairement à JavaScript

Pour spécifier un paramètre optionnel, on utilise le caractère ?

```
function getTotal(a: number, b?:number, c?:number): number
{
    b = b || 0;
    c = c || 0;
    return a + b +
    c;
}
```

TypeScript - Interfaces

TypeScript permet l'utilisation d'interfaces (explicites ou non)

```
function addMasseToChargement(  
  chargement: { masse: number; },  
  masse: number): void {  
    chargement.masse += masse;  
}
```

- On informe ici que le paramètre `chargement` doit avoir une propriété `masse`

On peut écrire cela sous la forme d'une interface

```
interface HasMasse { masse: number; }  
  
function addMasseToChargement(chargement: HasMasse,  
                               masse: number): void {  
    chargement.masse += masse;  
}
```

TypeScript - Interfaces

On retrouve le même fonctionnement avec les fonctions

```
function faireCrier(personnage)
    {   personnage.crier();
}
```

On peut définir une interface

```
interface PeutCrier {
    crier(): void;
}
```

```
function faireCrier(personnage: PeutCrier): void {
    personnage.crier();
}
```

```
let lion = {
    crier: () => console.log('Grrrr');
};

faireCrier(lion);
```

TypeScript - Interfaces

Une classe peut implémenter une interface

```
class Lion implements PeutCrier {  
    crier() : void {  
        console.log('Grrrrr');  
    }  
}
```

TypeScript - Classes

Une classe en TypeScript peut avoir des propriétés, ce qui n'est pas le cas en ES6

```
class Lion implements PeutCrier {  
    age: number = 15;  
    crier() {  
        console.log(`Grrrr, j'ai ${age} ans`);  
    }  
}
```

Par défaut, une propriété est `public`. Il est possible d'utiliser le mot-clé `private`

```
class Lion implements PeutCrier {  
    private age: number = 15;  
    ...  
}
```


TypeScript - Classes

L'utilisation des constructeurs est très utile pour déclarer les propriétés d'un objet

```
class Lion implements PeutCrier {  
    constructor(public taille: number, private age:number) { }  
  
    crier() {  
        console.log(`Grrrrr, j'ai ${age} ans`);  
    }  
}
```

Ce constructeur a pour effet de déclarer 2 propriétés mais aussi de faire automatiquement l'affectation lors de la construction d'un objet

TypeScript - Décorateurs

Les décorateurs sont très utilisés dans Angular

Ils permettent d'ajouter de la métadonnée aux éléments

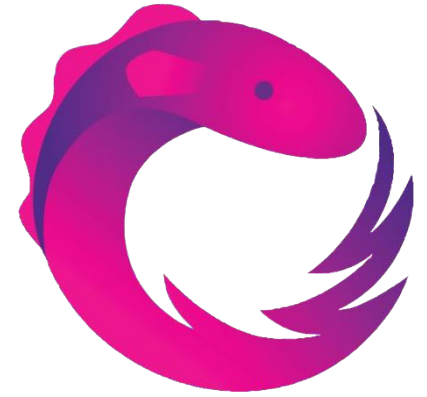
Les décorateurs sont préfixés par le caractère @

```
const Log = function () {  
  return (target: any, name: string, descriptor: any) =>  
    { console.log(`appel de ${name}`);  
  };  
};
```

```
class ClientService {  
  @Log()  
  getClients() { ...  
  } @Log()  
  getClient(id) { ... }  
}
```

TP - TypeScript

- Reprendre le code ES6 du TP précédent et l'écrire en TypeScript
- Installer typescript sur la machine et compiler le script TypeScript en script JavaScript



Observable et RxJS



RxJS

RxJS (*Reactive Extensions for JavaScript*) est une librairie qui permet de développer des applications asynchrones et basées sur des événements en utilisant des séquences observables

Cette librairie fournit

- Un type principal : `Observable`
- Des types secondaires : `Observer`, `Subject`,...
- Des opérateurs qui permettent de manipuler les événements (`map`, `filter`,...)

Un `Observable` peut être comparé à une collection de futures valeurs ou événements

Un `Observer` écoutera et réagira aux événements liés à

l'`Observable` Une `Subscription` représente une exécution d'un `Observable`

Un `Subject` est un équivalent à `EventEmitter` ; il permet d'envoyer des données dans des `Observable`

RxJS

Création d'un Observable et souscription

```
import { Observable } from 'rxjs';

const observable = new Observable(subscriber => {
  subscriber.next(0);
  subscriber.next(1);
  subscriber.next(2);
  subscriber.complete();
});

const sub = observable.subscribe({
  next(x) { console.log('Valeur: ' + x); },
  error(err) { console.error('Erreur: ' + err); },
  complete() { console.log('Terminé'); }
});
```

Version plus compacte

```
observable.subscribe( v => console.log('Value : ' + v));
```

RxJS

```
import { interval } from 'rxjs'; const
numbers = interval(1000);
numbers.subscribe(v => console.log(v));
```

La méthode `interval` permet de créer un flux qui génère une nouvelle valeur à toutes les millisecondes spécifiées

Un `Observable` est inactif (*cold*) tant qu'il n'a pas d'abonnés (*subscriber*)
Dès qu'un `Observable` a un abonné, il devient actif

(*hot*) L'appel à `subscribe` :

1. Rend l'`Observable` actif (*hot*)
2. Permet de spécifier un callback pour réagir à ce qui est poussé dans le flux de la chaîne

RxJS

Création d'un Observable depuis un tableau

```
import { from } from 'rxjs';  
const numbers = from([10, 21, 40, 53]);  
  
numbers.subscribe(v => console.log(v));
```

Création d'un Observable depuis une suite de valeurs

```
import { of } from 'rxjs';  
const numbers = of(10, 21, 40, 53);  
  
numbers.subscribe(v =>  
  console.log(v));
```


RxJS

Quelques fonctions proposées (voir les *marble diagrams* sur <http://reactivex.io/documentation/operators> + <https://www.learnrxjs.io/learn-rxjs/operators>)

Nom	Description
<code>map (fn)</code>	Applique la fonction <code>fn</code> pour tous les éléments et retourne le résultat
<code>filter (predicate)</code>	Ne laisse passer que les éléments qui respectent le prédicat
<code>debounce (time)</code>	N'émet que si un certain temps <code>time</code> est passé
<code>distinct ()</code>	Enlève les éléments en double
<code>first ()</code>	Ne garde que le premier élément émis
<code>take (n)</code>	Ne garde que les <code>n</code> premiers éléments
<code>last ()</code>	Ne garde que le dernier élément émis
<code>takeLast (n)</code>	Ne garde que les <code>n</code> derniers éléments

RxJS

Les méthodes peuvent être chaînées avec `pipe`

```
import { from } from 'rxjs';

import { filter, map } from 'rxjs/operators';
const numbers = from([10, 21, 40, 53]);

numbers.pipe(

  filter(x => x % 2 == 0),

  map(x => x / 2)

).subscribe(v => console.log(v));
```

RxJS et Angular

Angular utilise RxJS et permet d'utiliser cette librairie. Le framework met à disposition un adaptateur autour de Observable sous la forme de EventEmitter

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

EventEmitter dispose d'une méthode subscribe()

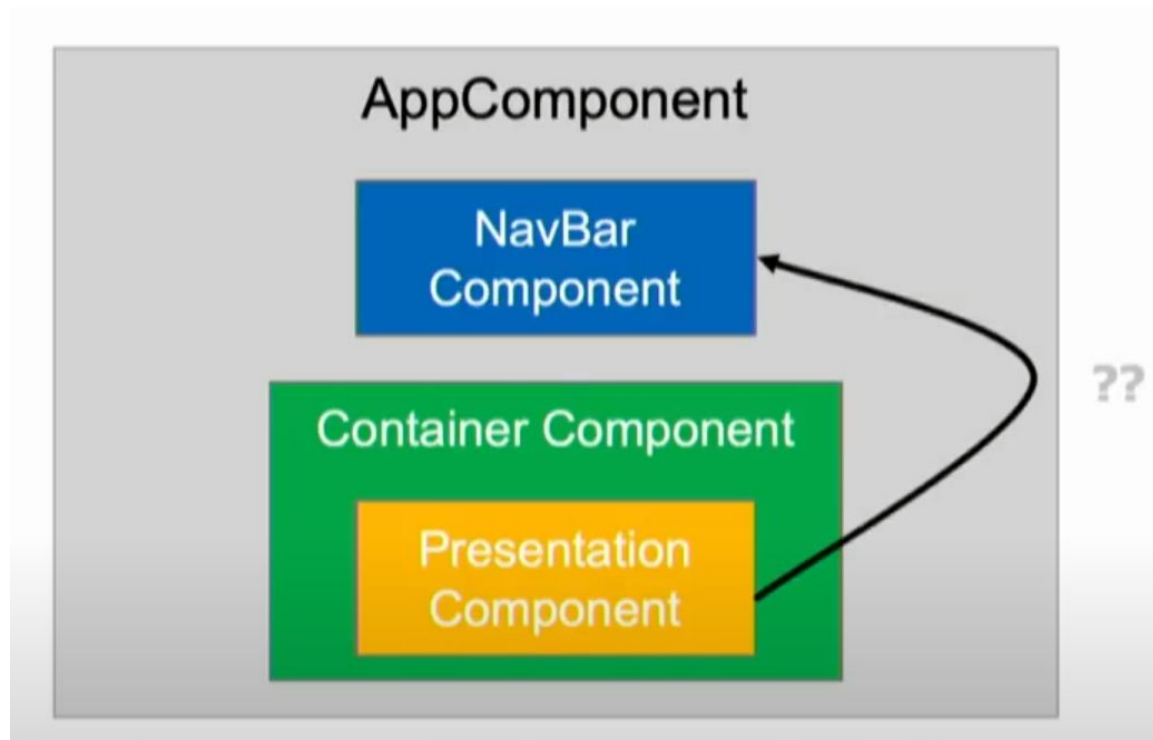
qui comporte 3 paramètres :

1. Une méthode pour réagir aux événements
2. Une méthode pour réagir aux erreurs
3. Une méthode pour réagir à la terminaison

```
let emitter = new  
EventEmitter();  
emitter.subscribe(  
    valeur =>  
        console.log(valeur), erreur  
=> console.log(erreur), ()  
=> console.log('Terminé !')  
);  
emitter.emit('coucou')  
; emitter.complete();
```

Communication Options in RxJS

Comment communiquer entre composants ?



Subject

Send data to subscribed observers.
Any previously emitted data is not
sent to new observers.

BehaviorSubject

Send last data value to new observers.

BehaviorSubject

Send last data value to new observers.

ReplaySubject

Previously sent data can be “replayed” to new observers.

AsyncSubject

Emits the last value (and only the last value) to observers when the sequence is completed.

Communications Options

Event Bus

Dashboard widgets
Mediator pattern

Observable Service

Inventory watcher
Observer pattern

EventBus

```
@Injectable()
export class EventBusService {
  private subject$ = new Subject();

  emit(event: EmitEvent) {
    this.subject$.next(event);
  }

  on(event: Events, action: any): Subscription {
    return this.subject$.pipe(
      filter((e: EmitEvent) => e.name === event),
      map((e: EmitEvent) => e.value))
      .subscribe(action);
  }
}
```

EventBus

```
export class HeaderComponent implements OnInit, OnDestroy {  
  customer: Customer;  
  subsink = new SubSink();  
  constructor(private eventbus: EventBusService) { }  
  
  ngOnInit() {  
    this.subsink.sink =  
      this.eventbus.on(Events.CustomerSelected,  
        (cust => this.customer = cust);  
  }  
  
  ngOnDestroy() {  
    this.subsink.unsubscribe();  
  }  
}
```

Observable Service

```
@Injectable()
export class InventoryService {
  latestProduct: Product;

  private inventorySubject$ =
    new BehaviorSubject<Product>(this.latestProduct);

  inventoryChanged$ = this.inventorySubject$.asObservable();

  addToInventory(product: Product) {
    this.latestProduct = product;
    this.inventorySubject$.next(product);
  }
}
```

Observable Service

```
export class InventoryComponent implements OnInit, OnDestroy {  
  @Input() products: Products[] = [];  
  
  constructor(private inventoryService: InventoryService) { }  
  
  ngOnInit() {  
    this.subsink.sink =  
      this.inventoryService.inventoryChanged$.  
        .subscribe(prod => this.products.push(prod));  
  }  
  
  ngOnDestroy() {  
    this.subsink.unsubscribe();  
  }  
}
```

Si on resume

Subject

Behavior
Subject

Replay
Subject

Async
Subject

Event Bus

Observable
Service

Observable Store

Single source
of truth

Immutable
state

Store change
notifications

Track state
history

Single source of truth

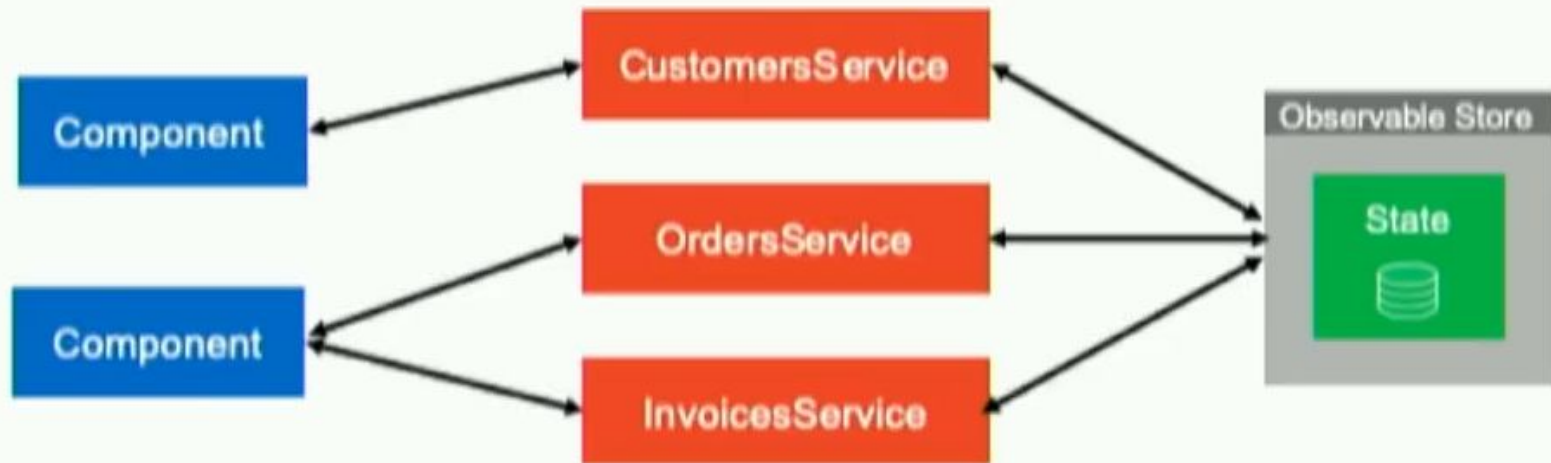
Lorsque vous créez une application Angular, l'état est généralement divisé et géré dans plusieurs services. Au fur et à mesure de la croissance de votre application, le suivi de vos changements d'état commence à devenir compliqué et difficile à déboguer et à maintenir. Avoir une seule source de vérité résout ce problème car l'état n'est géré que dans un seul objet et à un seul endroit, donc le débogage ou l'ajout de modifications devient beaucoup plus facile.

Immutable State

Le Store est construit sur une structure de données unique et immuable qui fait de la détection des changements une tâche relativement simple en utilisant la stratégie **OnPush**

Observable Store

Observable Store State Flow



Source :

Mastering the Subject: Communication
Options in RxJS | Dan Wahlin

https://www.youtube.com/watch?v=_q-HL9YX_pk

TP - Ecrire un service Bus

- Ecrire un service bus
- Ecrire une classe `AbstractComponent` avec une propriété `Children`
- Les Classe **A** , **B** et **C** hérité de `AbstractComponent`
- **B** fait partie des children de **A** et **C** des children de **B**
- Envoyer et afficher(`console.log`) un string du composant **C** vers **A**

Injection de dépendances

Injection de dépendances

L'injection des dépendances est un pattern de conception permettant de faciliter la gestion des dépendances, améliorer l'extensibilité d'une application et faciliter les tests-unitaires.

```
class UserStore {  
    getUser(userId: string): Observable<User> {  
        let restApi = new RestApi(new ConnectionBackend(), new RequestOptions({headers: ...}));  
        return restApi.users.get(userId);  
    }  
}
```

Sans injection de dépendance, Il faut savoir comment instancier `RestApi` ?

Comment factoriser ? Comment contrôler l'implémentation de la classe `RestApi` ?

Injection de dépendances

Angular dispose d'un "injector" qui implémente une factory permettant d'instancier des classes et maintenir les instances.

Lors du "bootstrap", Angular crée le "rootinjector" qui sera chargé d'injecter les dépendances de l'application.

On indique qu'une dépendance est injectable à l'aide du "decorator" `@Injectable`. Angular créera alors une instance unique de la dépendance disponible dans toute l'application.

```
const injector = new Injector([RestApi]);  
const restApi1 = injector.get(RestApi);  
const restApi2 = injector.get(RestApi); // restApi2 is the same instance as restApi1.
```


Portée des services

InjectorTree

Angular ne dispose pas que d'un seul "injector" mais d'un arbre d'"injectors".

RootInjector

Tous les "**providers**" définis par les modules importés directement ou indirectement par l'AppModule sont injectés par le "rootInjector" et sont donc accessibles dans toute l'application.

Component Injector

Chaque composant dispose d'un "injector" et peut définir des "providers" via la propriété providers de sa configuration.

Ces "providers" vont écraser les "providers" parents (ceux des composants parents ou du "rootinjector") et définir de nouvelles instances des services associés pour chaque instance du composant.

Il est préférable d'éviter cette approche et d'utiliser des Inputs /Outputs pour interagir avec les "childcomponents".

Autrement, les composants auront des comportements différents et parfois imprédictibles en fonction de l'endroit où ils sont utilisés

Component Injector

```
@Component({  
  providers: [  
    BookRepository  
  ]  
})  
export class BookPreviewComponent {  
}
```



Angular CLI

Angular CLI - Présentation

A l'origine, une application Angular était lourde à initialiser et à paramétrer

L'équipe Angular a travaillé sur un outil permettant de simplifier ces étapes

L'outil Angular CLI est né

Il permet :

- D'initialiser un projet Angular (ng new app-ex)
- D'ajouter des éléments au projet (ng add)
- D'avoir un serveur pour tester l'application au cours du développement (ng serve)
- De lancer des tests unitaires (ng test)
- De construire une application (ng build)

Angular CLI - Présentation

L'équipe d'Angular CLI a fait le choix d'utiliser Webpack pour packager l'application

- Les nombreux fichiers (js, css,...) sont packagés en quelques fichiers js
- Webpack détecte les dépendances entre modules et va incorporer ces modules au fichier de sortie

Cela allège grandement l'application par rapport à SystemJS qui était précédemment utilisé

- SystemJS est un chargeur dynamique de modules
- Se révèle particulièrement lent avec Angular

Ivy +Le tree-shaking

Ivy est le nom de code du pipeline de compilation et de rendu de nouvelle génération d'Angular. Avec la version 9 d'Angular, le nouveau compilateur et les instructions d'exécution sont utilisés par défaut à la place de l'ancien compilateur et du moteur d'exécution, connus sous le nom de View Engine.

Tree shake - Terme populaire utilisé pour désigner une étape du processus de construction où le code inutilisé n'est pas inclus dans le bundle, ce qui réduit le bundle global.

Tree Shaking Providers (TSP)

Tout ce qui sera déclaré au niveau du module sera packagé

- Déclarer les services avec `providedIn` plutôt que dans la partie `providers` du module

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

@Injectable({
  providedIn: 'root' // singleton
})
export class SharedService {
  constructor() {}
}
```


Angular CLI - Installation

L'installation d'Angular CLI s'effectue par le biais de NPM (Node Package Manager)

Installation de Node.js

- Programme d'installation standard

Installation d'Angular CLI

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

L'installation d'Angular CLI ajoute la commande `ng` au PATH

La commande `ng` est au cœur de l'utilisation d'Angular CLI

```
>ng version
@angular/cli          10.2.0
rxjs                  6.6.3
typescript             4.0.5
```

Angular CLI - Création d'une application

La création d'une application avec Angular CLI s'effectue avec la commande suivante à exécuter dans le répertoire où créer l'application

```
ng new <nom_application>
```

Exemple :

```
ng new formappli
```

Un nouveau répertoire est alors créé et ce dernier contient la structure d'un projet de base Angular

Pour tester cette application, lancer le serveur Angular CLI dans le répertoire de l'application via la commande

```
ng serve
```

Les différents modules de l'application sont traités et regroupés dans quelques fichiers JavaScript. Par ailleurs, dès qu'une partie de l'application est modifiée, l'application est mise à jour et testable directement.

On peut alors accéder à l'application en allant sur l'URL :

```
http://localhost:4200
```

TP Angular – Création d'une application

- Installer sur la machine les outils nécessaires, en particulier AngularCLI
- Créer une application « FlightViewWeb ».
- Choisir le mode de styles SCSS
- Ne pas utiliser le module router
- Tester l'application pour s'assurer que tout est bon.

Angular CLI - La structure

Structure d'un répertoire Angular CLI

Nom	Description
dist	Buils de production ou de développement de l'application
src	Sources de l'application
e2e	Description des tests end-to-end

Le fichier `package.json` contient les différentes dépendances du projet

Angular CLI - La structure

Dans le répertoire `src`, on retrouve différents éléments intéressants

Nom	Description
Répertoire <code>app</code>	Composants de l'application
Répertoire <code>assets</code>	Fichiers ou répertoires à copier tels quels dans le projet
Répertoire <code>environments</code>	Réglages pour les différents environnements (dev, prod,...)
Fichier <code>index.html</code>	Fichier HTML principal de l'application
Fichier <code>main.ts</code>	Fichier Typescript principal de l'application
Fichier <code>test.ts</code>	Prépare l'environnement de test et exécute tous les tests unitaires
Fichier <code>tsconfig.app.json</code>	Configuration Typescript de l'application
Fichier <code>typings.d.ts</code>	Fichier de description des types Typescript

Angular CLI - La structure

Angular CLI, à la création d'un projet, a créé un composant

On retrouve la description de ce composant dans le fichier

`src/app/app.component.ts`

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

Le composant possède un sélecteur `app-root`, son propre template HTML dans le fichier `app.component.html` et sa propre CSS dans le fichier `app.component.css`

Angular CLI - La structure

Dans le fichier HTML principal de l'application (`src/index.html`), on retrouve le sélecteur du composant précédent

```
<body>
  <app-root></app-root>
</body>
```

Le fichier `src/app/app/module.ts` contient la configuration haut-niveau de notre module Angular nommé `AppModule`

Le fichier `src/main.ts` permet l'importation du module principal Angular (`AppModule`) et l'initialisation d'Angular par la ligne

```
platformBrowserDynamic().bootstrapModule(AppModule);
```

Angular CLI - La construction

La commande `ng serve` permet uniquement de tester l'application en local

Pour construire l'application, Angular CLI propose la commande `ng build`

`ng build` construit l'application en mode développement

`ng build --prod` construit l'application en mode production

On retrouve les fichiers dans le répertoire `dist` du projet

La construction en production

- Ajoute une chaîne de caractères aléatoire à la fin des fichiers pour empêcher le cache des fichiers
- Les fichiers sont minifiés et uglifiés

Angular CLI - La construction

Par défaut, Angular utilise le compilateur JIT (*Just-in-Time*)

- Les composants et les templates sont compilés côté client

Angular propose aussi la compilation AOT (*Ahead-of-Time*)

- Les composants et les templates sont compilés lors du build
- Avantages :
 - Rendu plus rapide
 - Téléchargement plus rapide (pas besoin de télécharger le compilateur)
 - Détection des erreurs de template

Pour forcer la compilation AOT

- `ng serve --aot`
- `ng build --aot`

Note : La construction de l'application en production (`ng build --prod`) utilise la compilation AOT par défaut

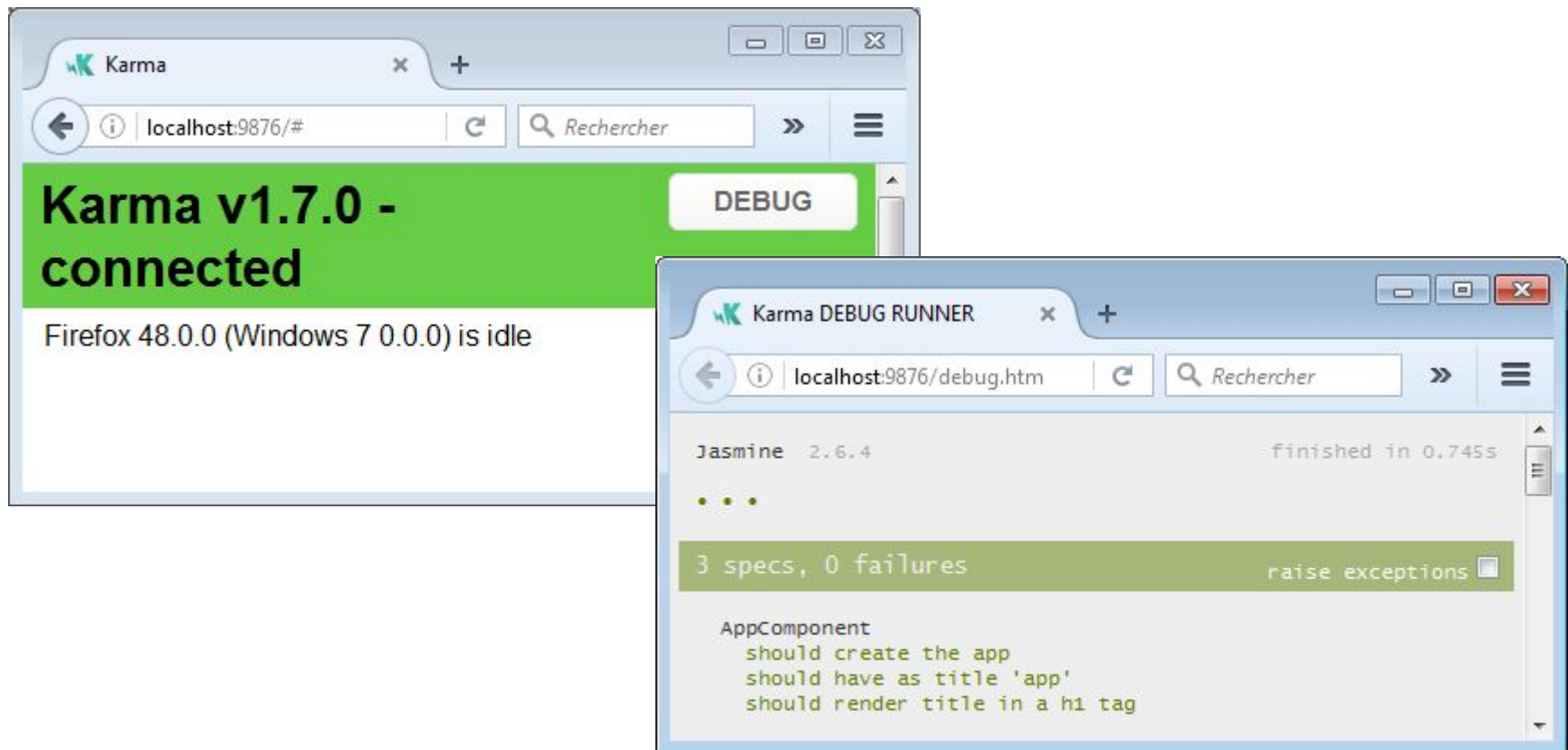
Depuis Angular 9, la compilation AOT est toujours utilisée par défaut grâce au nouveau moteur Ivy

Pour plus de détails sur la compilation AOT :

<https://angular.io/guide/aot-compiler>

Angular CLI - Les tests

Les tests unitaires peuvent être lancés via la commande `ng test`



Angular CLI - L'utilisation de librairies tierces

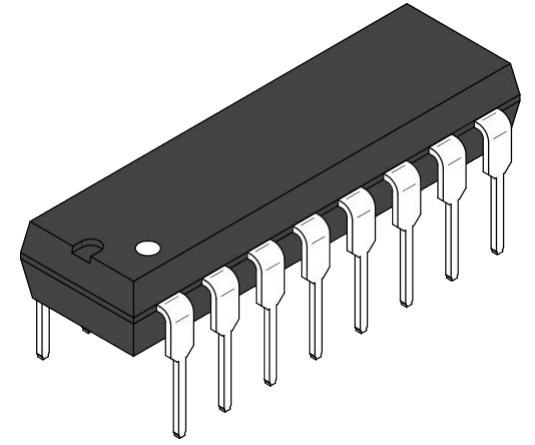
Dans le cas de l'utilisation de librairies tierces (par exemple moment, D3.js,...), il peut être utile d'importer les fichiers de définition de type TypeScript de la librairie installée

Exemple :

```
npm install d3 --save  
npm install @types/d3 --save-dev
```

Il convient alors de déclarer l'utilisation de ces fichiers dans la configuration TypeScript du projet (fichier `src/tsconfig.app.json`)

```
"compilerOptions": {  
    ...  
    "types": ["d3"]  
}
```



Les composants

Les composants

Les composants forment les briques d'une application Angular

Ils peuvent contenir eux-mêmes d'autres composants

Une application Angular est simplement un arbre de composants

A la racine de cet arbre on trouve le composant de haut-niveau, ou le composant racine (root component)

Lors de l'initialisation d'une application Angular, le composant racine est généré, ce qui engendre par cascade la génération des composants qu'il contient

Les composants

Pour construire une application Angular il convient de :

1. Séparer l'application en plusieurs composants
2. Décrire les responsabilités de chaque composant
3. Décrire les entrées et les sorties de chaque composant, ce qui compose en finalité son interface

La définition des entrées et des sorties permet de lier les composants entre eux en leur permettant de communiquer dans la structure de l'arbre des composants

Création d'un composant

Pour créer un composant, Angular CLI propose la commande

```
ng generate component <nom_composant>
```

Exemple : `ng generate component Client`

Le composant `ClientComponent` est alors généré par Angular CLI

```
installing component
  create src/app/client/client.component.css
  create src/app/client/client.component.html
  create
src/app/client/client.component.spec.ts
  create src/app/client/client.component.ts
  update src/app/app.module.ts
```

Le nouveau composant est également ajouté via un `import` et une entrée `declarations` dans le fichier principal à notre module Angular (`src/app/app.module.ts`)

Le composant

Un composant est caractérisé par le décorateur `@Component`

Ce décorateur permet de paramétrer le composant via un objet comportant des attributs :

- `selector` : Le sélecteur HTML du composant
- `templateUrl` : Le nom du fichier template HTML du composant
- `template` : Le template HTML du composant
- `styles` : Les styles CSS du composant
- `styleUrls` : Les fichiers CSS du composant

Le composant `ClientComponent` possède ce décorateur

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


Utilisation d'un composant

Pour utiliser un composant, il suffit d'utiliser son sélecteur au sein du template d'un autre composant (par exemple dans le composant principal)

```
<app-client></app-client>
```

Il est possible de modifier le paramétrage du composant

```
@Component({
  selector: 'app-client',
  template: `<h1>Client</h1>
              Prénom: Patrick`,
  styles: [
    `
      h1 { background-color : red; }
    `,
  ],
})
```



Les templates

Les templates

Dans le template on peut afficher les propriétés du composant en utilisant la syntaxe moustache `{{ propriété }}`

L'utilisation d'expressions est possible dans `{{ ... }}`

```
@Component({
  selector: 'app-client',
  templateUrl:
    './client.component.html', styleUrls:
    ['./client.component.css']
})
export class ClientComponent implements OnInit {
  prenom: string;

  constructor() {
    this.prenom = 'Patrick';
  }


  ngOnInit() {
  }
}
```

`<p>`
Prénom du client : `{{ prenom }}`
`</p>`

Les templates

On retrouve la même syntaxe pour les objets

```
@Component({
  selector: 'app-client',
  templateUrl:
    './client.component.html', styleUrls:
    ['./client.component.css']
})
export class ClientComponent {
  client: any = { prenom: 'Patrick' };
}
```



```
<p>
Prénom du client : {{ client.prenom
}}
</p>
```

Les templates

En cas d'affichage d'une variable `undefined` ou `null`,
Angular affichera une chaîne vide

Par contre, en cas d'accès à une propriété d'un objet qui n'existe pas,
une erreur est lancée

```
<p>  
Prénom du client : {{ client s.prenom }}  
</p>
```

Pour éviter cela dans le cas d'un objet récupéré plus tard (depuis un serveur) et indéfini au chargement du composant, on dispose de la syntaxe comportant le caractère ?

```
<p>  
Prénom du client : {{ client ?.prenom }}  
</p>
```

Utiliser des composants

Il est bien évidemment possible d'utiliser un composant dans un template

```
@Component({
  selector: 'app-client',
  templateUrl:
    './client.component.html', styleUrls:
    ['./client.component.css']
})
export class ClientComponent {
  client: any = { prenom: 'Patrick' };
}
```

```
<p>
Prénom du client : {{ client.prenom }}
</p>
<app-salutations></app-salutations>
```

```
@Component({
  selector: 'app-salutations',
  templateUrl:
    './salutations.component.html', styleUrls:
    ['./salutations.component.css']
})
export class SalutationsComponent { }
```

```
<h3>Salutations !!!</h3>
t
```

Binding de propriété

Avec Angular, il est possible d'accéder aux propriétés du DOM d'un composant HTML et ainsi de venir les modifier

Pour utiliser une entrée d'un composant, on utilise la syntaxe [. . .]

```
<p [style.color]="client.couleur">{{ client.prenom }}</p>  
<p [hidden]="!client.vip">Client VIP</p>
```

Ceci est équivalent à

```
<p [style.color]="client.couleur" [textContent]="client.prenom"></p>  
<p [hidden]="!client.vip">Client VIP</p>
```

Il sera possible d'utiliser cette syntaxe afin de transmettre des informations à ses propres composants

Evènements

Un évènement particulier peut être capturé par un composant Angular (par exemple le clic, l'appui sur une touche du clavier ou encore le mouvement de la souris)

Pour réagir à un évènement, on utilise la syntaxe (. . .)

```
<p>{{ client.prenom }}</p>  
<p [hidden]="!client.vip">Client VIP</p>  
<button (click)="client.vip = !client.vip">Modifier statut</button>
```

L'instruction peut être une fonction

```
export class ClientComponent {  
  ...  
  modifierStatut() {  
    this.client.vip = !this.client.vip;  
  }  
  ...  
}
```

```
<p>{{ client.prenom }}</p>  
<p [hidden]="!client.vip">Client VIP</p>  
<button (click)="modifierStatut()">Modifier  
statut</button>
```

Dans ses propres composants, il sera possible de créer ses propres évènements

Evènements

Les évènements gardent leur propriété bouillonnante (les évènements se propagent du bas vers le haut)

```
<p>{{ client.prenom }}</p>
<p [hidden]="!client.vip">Client VIP</p>
<div (click)="modifierStatut()">
    <button>Modifier statut</button>
</div>
```

Il est possible de récupérer les informations de l'évènement grâce à `$event`

```
<div (click)="modifierStatut($event)">
    <button>Modifier statut</button>
</div>
```

Cela permet de piloter l'évènement dans la fonction de prise en charge

```
modifierStatut(event) {
    console.log(event);
    event.preventDefault();
    event.stopPropagation();
    this.client.vip = !this.client.vip;
}
```

Variables locales

Dans les templates, il est possible de définir des variables locales grâce au caractère #

```
<input type="text" #nom>
{{ nom.value }}
<button (click)="nom.focus()">Donner le focus</button>
```

Il est possible d'utiliser cette syntaxe pour faire référence à un de ses composants

```
<app-salutations #sal></app-salutations>
<button (click)="sal.changerTexte()">Modifier</button>
```

Les directives

Pour enrichir les vues de nos composants, Angular met à notre disposition des directives structurelles

Ces directives permettent de construire des modèles qui s'appuient sur la balise HTML `<template>`

Les directives - ngIf

La directive structurelle `ngIf` permet d'instancier un template que si une condition est vérifiée

```
<ng-template [ngIf]="client.vip">  
  <h3>Client VIP !</h3>  
</ng-template>
```

Pour raccourcir les saisies, Angular utilise une syntaxe avec `*`

```
<h3 *ngIf="client.age < 18">Client mineur !</h3>
```

Les directives - ngFor

La directive `ngFor` permet d'instancier un template pour chaque élément d'une collection

```
<ul>  
  <li *ngFor="let client of clients">{{ client.nom }}</li>  
</ul>
```

Les directives - ngFor

Avec la directive `ngFor` on dispose avec cette directive de variables exportées

Nom	Description
<code>index</code>	Indice de l'élément en cours (commence par 0)
<code>even</code>	Booléen à <code>true</code> si l'index est pair
<code>odd</code>	Booléen à <code>true</code> si l'index est impair
<code>first</code>	Booléen à <code>true</code> si il s'agit du premier élément de la collection
<code>last</code>	Booléen à <code>true</code> si il s'agit du dernier élément de la collection

Pour récupérer l'indice de l'élément courant, on utilise `index` (commence par 0)

```
<ul>
  <li *ngFor="let client of clients; let n=index">{{ n }}. {{ client.nom }}</li>
</ul>
```

Les directives - ngSwitch

La directive `ngSwitch` est utilisée pour implémenter une structure de type switch. Le template varie selon une condition

```
<div [ngSwitch]="client.sexe">
  <p *ngSwitchCase="'H'" class="homme">Homme</p>
  <p *ngSwitchCase="'F'" class="femme">Femme</p>
  <p *ngSwitchDefault class="indefini">Indéfini</p>
</div>
```

Les directives - ngStyle

On utilise la directive `ngStyle` pour venir modifier plusieurs styles sur un élément

Cette directive permet de simplifier l'utilisation de la notation

`[style.<propriete>]`

```
<div *ngFor="let client of clients; let
  pair=even" [style.color]="pair ? 'red' :
  'blue'">
  <p [style.color]="client.couleur">{{ client.prenom }}</p>
</div>
```

Avec `ngStyle`, on utilise un objet qui décrit les styles appliqués

```
<p *ngFor="let client of clients; let pair=even"
  [ngStyle]="{ 'color': pair ? 'yellow' : 'purple',
               'background-color': couleurFond,
               'font-size.px': taillePolice }">
  {{ client.nom }}
</p>
```


Les directives - ngClass

La directive `ngClass` permet d'appliquer une ou des classes selon le cas

Cette directive permet de simplifier l'utilisation de la notation
`[class.<nom-classe>]`

```
<p [class.femme]="client.sexe == 'F'"  
    [class.homme]="client.sexe == 'H'" >{{ client.prenom }}</p>
```

Avec `ngClass`, on utilise un objet qui décrit les classes appliquées

```
<p [ngClass]="{'femme': client.sexe == 'F',  
              'homme': client.sexe == 'H'}">  
    {{ client.prenom }}  
</p>
```

A la place d'une comparaison, on peut appeler une fonction

TP Angular – Création de composants

* Sur la base du TP Angular 1, implémenter dans le projet une interface Flight contenant les attributs suivants :

Id : number

Numéro de vol : string

Code de l'aéroport de départ : string

Code de l'aéroport d'arrivée : string

Date et heure de départ : Date

Date et heure d'arrivée : Date

Retard : boolean

Prix : number

* Créer un composant FlightListComponent

-Y créer comme attribut un tableau de type Flight[]

-Afficher la liste des vols (1 div par vol) avec son numéro et les codes des aéroports

-Changer le style d'un élément de la div (par exemple sa couleur de fond)

lorsque le vol a son attribut Retard à true



Les pipes

Les pipes

Les pipes, combinés aux directives et aux expressions, permettent de manipuler et de transformer les données

Utilisation des pipes

```
{{ expression | pipe }}
```

Les filtres peuvent être chaînés

```
{{ expression | pipe1 | pipe2 }}
```

Angular fournit un ensemble de filtre déjà prêts à l'emploi

Le principe des pipes est le même que celui des filtres sous AngularJS

Le pipe json

Le pipe `json` permet d'afficher un objet au format JSON

```
<p>{{ clients | json }}</p>
```

Le pipe slice

Le pipe `slice` permet d'extraire une partie d'une collection, comme la fonction JavaScript du même nom

2 paramètres :

- Un indice de début (inclus)
- Un indice de fin facultatif (exclu)

```
<p>{{ clients | slice:1:3 | json }}</p>
```

```
<ol>
  <li *ngFor="let client of clients | slice:0:2">
    {{ client.prenom }}
  </li>
</ol>
```

Les pipes

Filtre lowercase

- Permet de mettre en minuscule

l'expression

```
{{ expression | lowercase }}
```

Filtre uppercase

- Permet de mettre en majuscule

l'expression

```
{{ expression | uppercase }}
```

Le pipe number

Filtre number

- Le pipe `number` permet de formater un nombre

```
{{ expression | number:<format>:<locale> }}
```

- Le format voulu est une chaîne écrite sous la forme

```
{minIntegerDigits}.{minFractionDigits}-{maxFractionDigits}
```

- `minIntegerDigits` : nombre minimal de chiffres dans la partie entière (1 par défaut)
- `minFractionDigits` : nombre minimal de chiffres dans la partie décimale (0 par défaut)
- `maxFractionDigits` : nombre maximal de chiffres dans la partie décimale (3 par défaut)

Le pipe percent

Filtre percent

- Permet d'afficher un pourcentage

```
{{ expression | percent:<precision>:<locale> }}
```

- On retrouve la même notation que pour le filtre `number`

```
{{ 0.4 | percent }} affiche 40%  
{{ 0.4266545999999 | percent:'3.3-8' }} affiche 042.66546%
```

Le pipe currency

Le pipe `currency` permet de formater un nombre selon une monnaie `{{ montant | currency }}`

Ce pipe a grandement changé suite aux modifications de la version 5 portant sur l'internationalisation

4 paramètres (tous optionnels) :

- Le code ISO de la devise
- Le mode d'affichage
 - `'code'` : Code de la monnaie (ex. : USD)
 - `'symbol'` : Symbole de la monnaie (ex. : \$)
 - `'symbol-narrow'` : Pour certains pays, le symbole étroit (ex. : CA\$ est le symbole du dollar canadien alors que \$ est son symbole étroit)
- Le format : voir le pipe `number`
- La locale : utilise `LOCALE_ID` par défaut

```
{{ 124.41 | currency:'EUR' }}  
{{ 124.41 | currency:'EUR':'code' }}
```

Le pipe date

Le pipe `date` permet de formater une date selon un format spécifié

```
{{ dateAchat | date }}
```

Le filtre accepte 3 paramètres optionnels :

- Le format : Le format d'affichage
- Le fuseau horaire : Le fuseau horaire (utilise le fuseau du navigateur si non spécifié)
- La locale : utilise `LOCALE_ID` par défaut

Le format indique le format de sortie sous la forme d'une chaîne (voir les possibilités sur <https://angular.io/api/common/DatePipe>)

```
{{ dateAchat | date:'shortDate' }}  
{{ dateAchat | date:'full':'-630' }}
```

Les pipes et l'i18n

Les pipes `DatePipe`, `CurrencyPipe`, `DecimalPipe` et `PercentPipe` utilisent par défaut `LOCALE_ID` pour connaître la locale à utiliser

Sans paramétrage, `LOCALE_ID` prend la valeur `'en-US'`

Pour utiliser une autre locale dans un pipe, on peut utiliser son dernier paramètre :

```
{{ 124.41 | currency:'EUR':'symbol':'.2-2':'fr-FR' }}
```

La locale concernée doit alors être chargée au sein du module (`app.module.ts`)

```
import { registerLocaleData } from '@angular/common';
import localeFr from '@angular/common/locales/fr';

registerLocaleData(localeFr, 'fr');
```

Les pipes et l'i18n

La locale à utiliser par défaut peut être spécifiée en donnant une valeur à `LOCALE_ID` dans le fichier `app.module.ts`

```
import { LOCALE_ID } from '@angular/core';
@NgModule({
  ...
  providers: [ { provide: LOCALE_ID, useValue: 'fr-FR' } ],
  ...
})
```

En compilation AoT, il est possible de spécifier la locale par défaut directement dans la ligne de commande

```
ng serve --aot --locale fr
ng build --prod --locale zh
```

Le pipe async

Le pipe `async` permet d'afficher une donnée obtenue de façon asynchrone (via une Promise ou un Observable)

```
{{ valeurAsync | async }}
```



```
export class ValeurComponent {  
  valeurAsync = new Promise(resolve =>  
  {  
    window.setTimeout(() => resolve('Bonjour !'), 1000);  
  });  
}
```

Les pipes personnalisés

Angular permet de créer ses propres pipes

Avec AngularCLI :

```
ng generate pipe Range
```

Un fichier `range.pipe.ts` est créé avec le décorateur `@Pipe`

```
@Pipe({  
  name: 'range'  
})
```

Le pipe est déclaré dans le fichier `app.module.ts`

```
declarations: [  
  ...  
  RangePipe  
,
```

Les pipes personnalisés

La fonction `transform` dans le fichier du pipe est cruciale

Elle accepte plusieurs paramètres :

- La valeur à transformer
- Les paramètres du pipe (si besoin)

```
transform(valeur: number, min: number = 0, max: number = 100): number
{
  if(min > max) max = min;
  if(valeur > max) return max;
  return valeur < min ? min :
  valeur;
}
```

```
{{ 585 | range:10:451 }}
{{ 2 | range:10 }}
{{ -10 | range }}
```


Les pipes personnalisés - Détection du changement

Par défaut un pipe Angular est marqué comme étant pur

- Le pipe (sa méthode `transform`) n'est exécuté que si la référence de l'objet lié est modifiée ou si un de ses arguments a été modifié

Un pipe pur ne sera donc pas exécuté de nouveau lorsqu'une propriété de l'objet transformé est modifiée

- Il sera par contre exécuté si on change l'objet

Angular propose de rendre un pipe impur en le spécifiant dans son décorateur avec la propriété `pure`

```
@Pipe({  
  name: 'concatParts',  
  pure: false  
})
```

Les pipes dans les composants

Pour utiliser un pipe dans un composant :

1. Importer le pipe dans le fichier du module (`app.module.ts`)

```
import { UpperCasePipe } from '@angular/common';
```

2. Ajouter le nom du pipe dans le tableau providers du même fichier

```
providers: [UpperCasePipe],
```

3. Importer le pipe dans le fichier du composant

```
import { UpperCasePipe } from '@angular/common';
```

4. Utiliser le pipe en l'injectant dans le constructeur (l'utiliser directement ou le stocker dans une variable privée pour une utilisation ultérieure)

```
constructor(upperCasePipe: UpperCasePipe, rangePipe: RangePipe) {  
  this.chaine = upperCasePipe.transform("chaine");  
}
```

TP Angular – Utilisation des pipes

Utiliser les pipes pour ajouter les dates et heures au format français



Plus loin dans les composants


Les entrées

Les entrées permettent à un composant de recevoir de la donnée de la part de son parent

Le décorateur `@Input` est utilisé dans le composant fils pour indiquer une propriété alimentée


```
@Input() nom : string
```

nom est une propriété
pouvant être alimentée par
[nom]

A horizontal arrow points from the text 'nom est une propriété pouvant être alimentée par [nom]' to the 'nom' part of the '@Input() nom : string' code snippet.

```
@Input('name') nom : string
```

nom est une propriété
pouvant être alimentée par
[name]

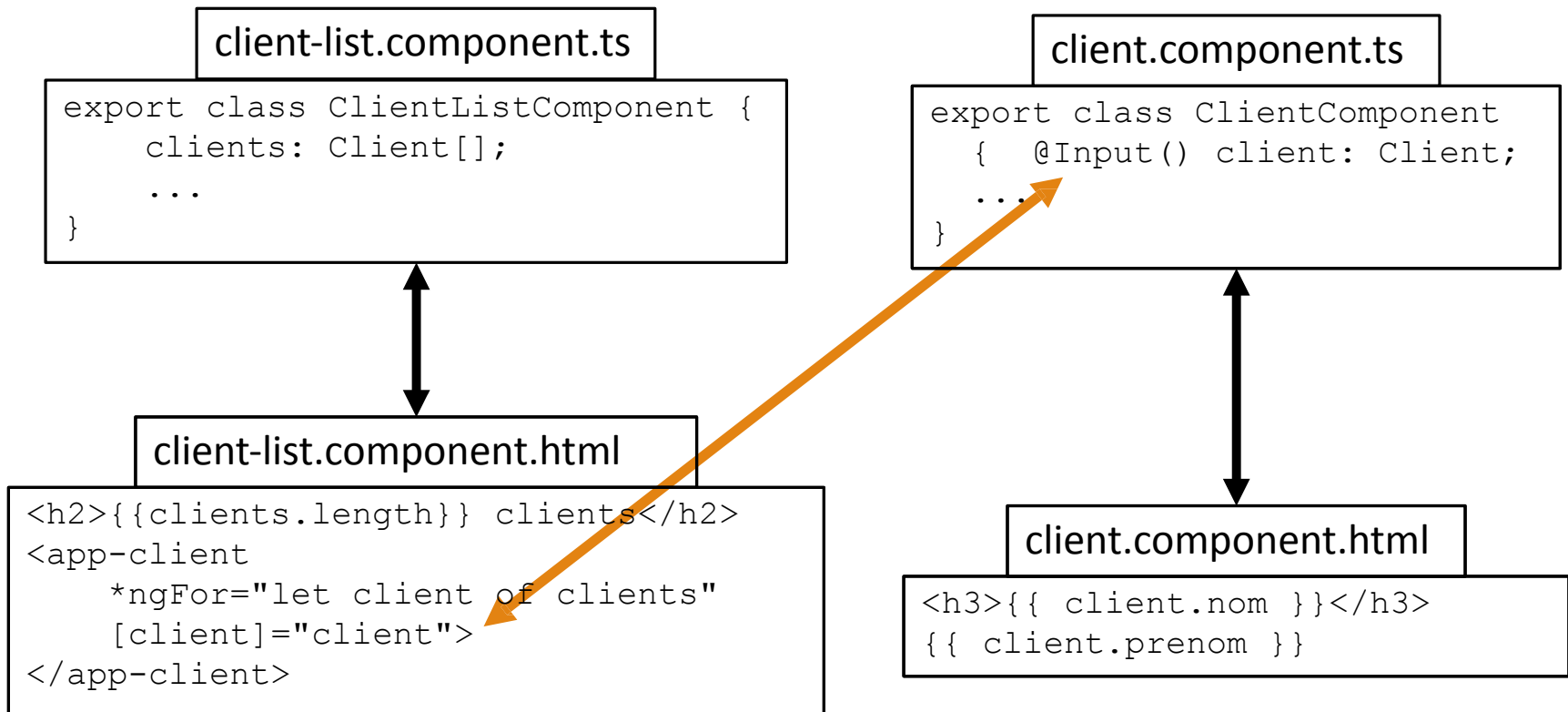
A horizontal arrow points from the text 'nom est une propriété pouvant être alimentée par [name]' to the 'name' part of the '@Input('name') nom : string' code snippet.

Ne pas oublier d'importer Input dans le fichier du composant

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

Les entrées

Exemple



Les sorties

Les sorties permettent à un composant de communiquer avec son parent

Avec Angular, les données entrent dans un composant via des propriétés, et en sortent via des évènements

Les évènements sont émis grâce à `EventEmitter` et sont déclarés avec le décorateur `@Output` dans le composant fils

```
@Output() clientSelected = new EventEmitter<Client>();
```

Ne pas oublier les importations nécessaires dans le composant fils

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

Les sorties

Pour émettre l'évènement, on utilise la méthode `emit` de `EventEmitter` dans le composant

```
this.clientSelected.emit(this.client)
```

Côté composant père, dans la balise du composant fils, on capture l'évènement grâce à la syntaxe `()` pour le traiter

Par défaut, le nom de l'évènement est le nom de la variable identifiée par `@Output`

```
@Output() clientSelected = new EventEmitter<Client>();
```

Il est possible, comme pour `@Input`, de spécifier un autre nom d'évènement

```
@Output('cs') clientSelected = new EventEmitter<Client>();
```


Les sorties

Exemple

client-list.component.ts

```
deleteClient(client) {  
  this.clients = this.clients.filter(  
    c => c.id !== client.id;  
  }  
}
```

client-list.component.html

```
<h2>{{clients.length}} clients</h2>  
<app-client  
  *ngFor="let client of clients"  
  [client]="client"  
  (clientDeleted)="deleteClient($event)"  
>  
</app-client>
```

client.component.ts

```
export class ClientComponent {  
  @Output() clientDeleted  
    = new EventEmitter<Client>();  
  
  deleteClient() {  
    this.clientDeleted.emit(this.client);  
  }  
  ...  
}
```

client.component.html

```
<button (click)="deleteClient()">  
  Supprimer</button>
```

Le cycle de vie

Les directives suivent un cycle de vie

Il est possible d'utiliser les différentes phases du cycle de vie afin de faire des traitements

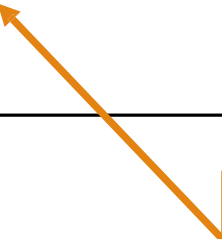
Nom	Description
<code>ngOnInit</code>	Appelé après le premier chargement (utile pour de l'initialisation)
<code>ngOnChanges</code>	Appelé quand une propriété bindée est modifiée (reçoit une map contenant les changements)
<code>ngOnDestroy</code>	Appelé quand le composant est supprimé
<code>ngDoCheck</code>	Appelé à chaque cycle de changement
<code>ngAfterContentInit</code>	Appelé quand les bindings du composant Contenu ont été vérifiés pour la première fois
<code>ngAfterContentChecked</code>	Appelé quand les bindings du composant Contenu ont été vérifiés, même quand ils n'ont pas changé
<code>ngAfterViewInit</code>	Appelé quand les bindings des directives enfants Vue ont été vérifiés pour la première fois
<code>ngAfterViewChecked</code>	Appelé quand les bindings des directives enfant Vue ont été vérifiés, même quand ils n'ont pas changé

Le cycle de vie

Pour utiliser cela, il convient d'implémenter l'interface du cycle voulu

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

export class ClientComponent implements OnInit {
  ...
  ngOnInit() {
    console.log(this.client);
  }
}
```



On aurait undefined si on faisait cela dans le constructeur

ChangeDetectionStrategy.Default

Par défaut, Angular ne fait aucune hypothèse sur ce dont dépend le composant. Il faut donc qu'il soit prudent et qu'il vérifie chaque fois que quelque chose a changé, c'est ce qu'on appelle la vérification sale. De manière plus concrète, il effectuera des vérifications pour **chaque événement du navigateur, minuteries, XHR et promesses**.

Cela peut être problématique lorsque vous commencez à avoir une grande application avec de nombreux composants, surtout si vous êtes concentré sur les performances.

ChangeDetectionStrategy.onPush

Avec onPush, le composant ne dépend que de ses entrées et embrasse l'immuabilité, la stratégie de détection des changements entrera en vigueur lorsque:

- La référence d'entrée change;
- Un événement provient du composant ou de l'un de ses enfants;
- Exécutez la détection de changement explicitement (`componentRef.markForCheck ()`);
- Utilisez le tube asynchrone dans la vue.

ViewChild

Le décorateur `ViewChild` permet d'obtenir une référence vers un élément du template

- Dans le template

```
<label #nomLabel for="nom">Nom</label>
```

- Dans le composant

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

...
@ViewChild('nomLabel') private nomLabel: ElementRef<HTMLInputElement>;
...
ngAfterViewInit() {
    this.nomLabel.nativeElement.style.backgroundColor = 'red';
}
```

Il est possible aussi d'interroger un type de composant

```
@ViewChild(ClientRowComponent) clientRow: ClientRowComponent;
```

ViewChildren

Le décorateur `ViewChildren` permet d'obtenir une référence vers plusieurs éléments du template sous la forme de `QueryList`

```
import {AfterViewInit, QueryList, ViewChildren } from '@angular/core';

...
@ViewChildren(ClientRowComponent) clientsRows: QueryList<ClientRowComponent>;
...
ngAfterViewInit() {
  this.clientsRows.changes.subscribe(
    newClientsRows => console.log(newClientsRows)
  )
}
```

`QueryList` a les attributs suivants :

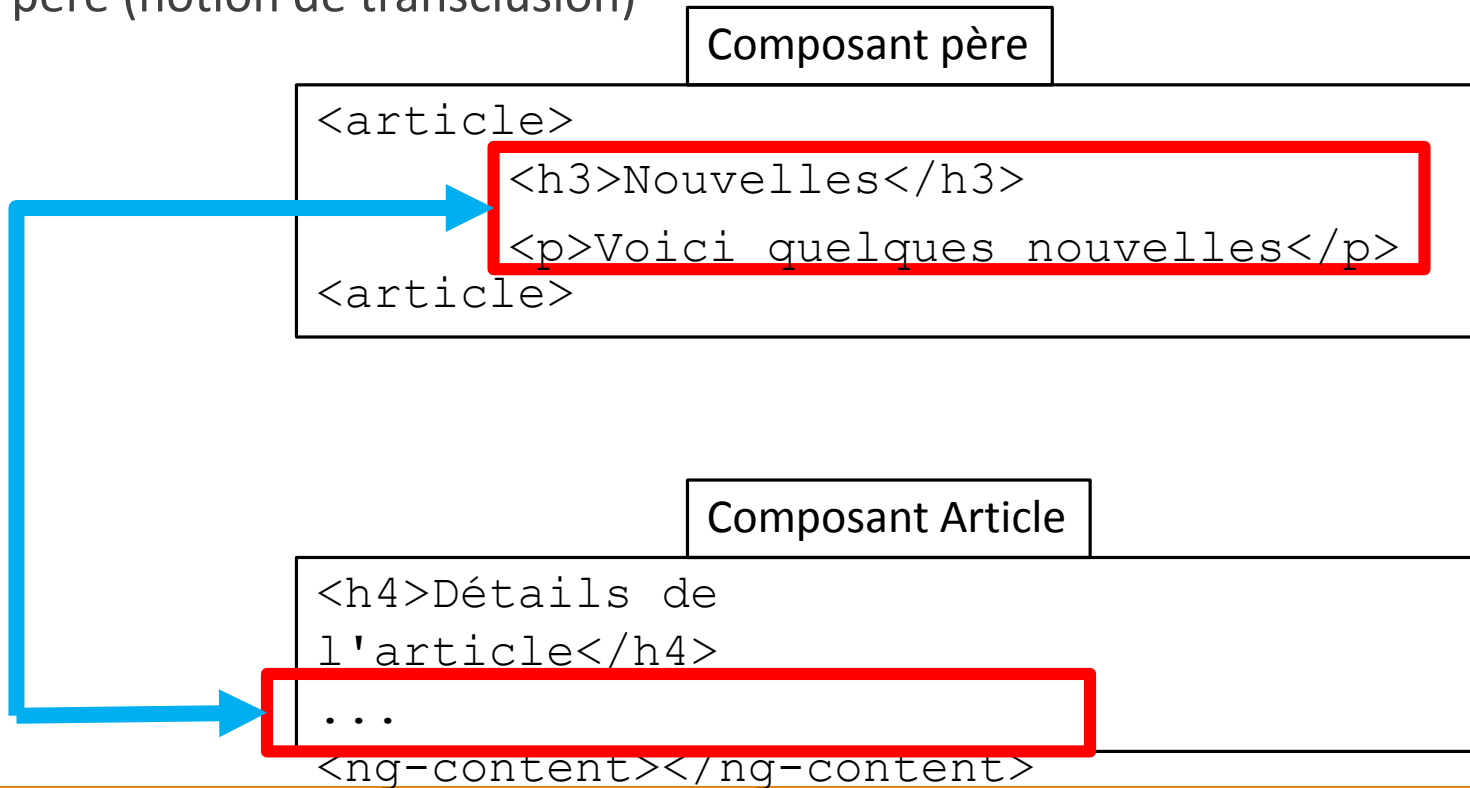
- `first` : Premier élément
- `last` : Dernier élément
- `length` : Nombre d'éléments
- `changes` : Observable qui émet la nouvelle `QueryList` quand elle est modifiée

`QueryList` possède des méthodes permettant d'accéder à la liste, par exemple :

- `toArray()`
- `forEach()`

ng-content

La balise `<ng-content>` permet de définir dans un composant un emplacement fixe qui sera alimenté dynamiquement par le composant père (notion de transclusion)



ContentChild / ContentChildren

Les décorateurs `@ContentChild` et `@ContentChildren` permettent quant à eux d'obtenir une référence vers un ou des éléments du contenu projeté

```
@ContentChildren(ActualitesComponent) newsList:
  QueryList<ActualitesComponent>;

ngAfterContentInit() {
  console.log(this.newsList);
}
```

TP Angular n°4 – Création de composants (suite)

Plutôt que d'avoir un composant FlightListComponent qui fasse tout, on souhaite plutôt avoir un composant FlightListComponent qui utilise un composant FlightComponent pour chaque vol à afficher

Modifier le TP Angular pour répondre à cette problématique :

- Créer le composant FlightComponent et l'implémenter
- Modifier le composant FlightListComponent en conséquence

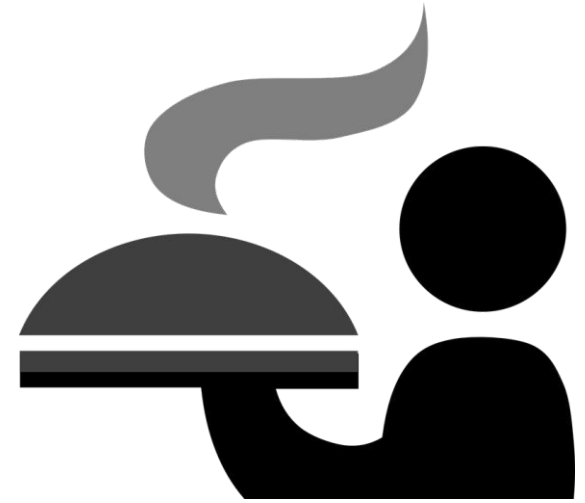
TP Angular n°5 – Création de composants (suite)

Créer un composant FlightDetailsComponent affichant toutes les informations d'un vol (dont le prix en euros) transmis en entrée

Ajouter à FlightComponent un bouton « Détails »

Grâce à un système d'Output, faire en sorte que

FlightListComponent utilise FlightDetailsComponent pour afficher le vol choisi



Les services

Les services

Les services sont des classes pouvant être injectées et utilisées dans d'autres classes

Un service pourra être utile afin de gérer par exemple les objets métier d'une application

L'écriture d'un service s'effectue par l'écriture d'une simple classe

- Dans les classes qui utiliseront le service, la même instance de ce service sera injectée si le service est déclaré au niveau module

L'écriture d'un service

Avec Angular CLI, un service peut être créé grâce à la commande

```
ng generate service <nom_service>
```

Exemple (pour un service permettant de gérer une collection)

```
ng generate service Client
```

Angular CLI ajoute aux services créés le décorateur `@Injectable()`

L'écriture d'un service

Exemple d'un service permettant de gérer une collection

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

'./client'; @Injectable()

export class ClientService {
  private clients:

  Client[]; constructor()

  {

    this.clients = [
      new Client(0, "Foucault", "Patrick", 52,
        false), new Client(1, "Renard", "Benoit", 23,
        false), new Client(2, "Grandu", "Marie", 37,
        true),

    ]
  }

  getClients(): Client[]
  { return
    this.clients;
  }
}
```

```
getClient(id: number): Client {
  return this.clients.find( c => c.id == id);
}
```

L'utilisation du service

Avant Angular 6, pour utiliser un service, il convient d'importer et d'enregistrer ce service dans notre module pour le rendre disponible (fichier `app.module.ts`)

```
import { ClientService } from './client.service';  
  
providers: [ClientService],
```

En enregistrant le service au niveau module, le service devient disponible pour tous les composants du module sous la forme d'un singleton

- Il est possible d'enregistrer un service pour seulement un composant ou quelques composants en ajoutant l'entrée `providers` dans le décorateur `@Component`. Attention ! Le service devient un singleton pour le composant

L'utilisation du service

A partir d'Angular 6, cette étape n'est plus nécessaire grâce à l'attribut `providedIn` du décorateur `@Injectable()`

```
@Injectable({  
  providedIn: 'root'  
})  
export class ClientService {  
  
}
```

La valeur mentionnée dans `providedIn` indique la portée :

- `'root'` : Racine de l'application
- `NomModule` : Module

Il est possible d'enregistrer un service pour seulement un composant ou quelques composants en ajoutant l'entrée `providers` dans le décorateur `@Component`. Attention ! Le service devient un singleton pour le composant

L'utilisation du service

Ensuite, pour utiliser le service, on procède par injection de dépendance

- Importer le service dans la classe utilisant ce service

```
import { ClientService } from '../client.service';
```

- Injecter le service dans le constructeur de la classe

```
constructor(private clientService: ClientService) {  
}
```

L'utilisation du service

Le service devient alors utilisable dans le composant dans lequel il est injecté

```
export class ClientListComponent implements OnInit {
  clients: Client[];

  constructor(private clientService: ClientService) {
  }

  getClients() {
    this.clients = this.clientService.getClients();
  }

  ngOnInit() {
    this.getClients();
  }
}
```

TP Angular n°6 – Services

Exporter la liste des vols dans un service FlightService



Les formulaires

Les formulaires

Angular propose de nombreuses possibilités pour gérer ses formulaires

Deux approches :

- *Template driven* (piloté par le template)
- *Model driven* (piloté par le modèle / code)

Template driven

- Seules les directives utilisées dans le template permettent de définir le formulaire
- Utile pour les formulaires simples

Model driven

- Le formulaire est décrit dans le code du composant
- Des directives sont utilisées pour lier ce code au template

FormControl

Quelque soit la méthode choisie, des objets `FormControl`, représentant les champs du formulaire, seront utilisés

Les objets `FormControl` possèdent plusieurs attributs

Attribut	Signification
<code>valid</code>	Le champ est valide
<code>errors</code>	Il y a des erreurs dans le champ (objet)
<code>touched</code>	<code>false</code> initialement, <code>true</code> dès que l'utilisateur entre dans le champ
<code>untouched</code>	<code>true</code> initialement, <code>false</code> dès que l'utilisateur entre dans le champ
<code>pristine</code>	<code>true</code> initialement, <code>false</code> dès que l'utilisateur modifie le champ
<code>dirty</code>	<code>false</code> initialement, <code>true</code> dès que l'utilisateur modifie le champ
<code>value</code>	La valeur du champ
<code>valueChanged</code>	Observable qui émet à chaque changement du champ

FormGroup

Un objet `FormGroup` est un regroupement d'objets `FormControl`

Les objets `FormGroup` possèdent plusieurs attributs

Attribut	Signification
<code>valid</code>	Le groupe (dont tous les champs) est valide
<code>errors</code>	Il y a des erreurs dans le groupe (objet - clé : erreur / valeur : tableau des champs avec cette erreur)
<code>touched</code>	<code>false</code> initialement, <code>true</code> dès que l'utilisateur entre dans un champ du groupe
<code>untouched</code>	<code>true</code> initialement, <code>false</code> dès que l'utilisateur entre dans un champ du groupe
<code>pristine</code>	<code>true</code> initialement, <code>false</code> dès que l'utilisateur modifie un champ du groupe
<code>dirty</code>	<code>false</code> initialement, <code>true</code> dès que l'utilisateur modifie un champ du groupe
<code>value</code>	Objet avec en clé les champs et en valeur la valeur du champ
<code>valueChanged</code>	Observable qui émet à chaque changement d'un champ du groupe

Formulaires Model Driven

Le formulaire est représenté dans le composant comme un modèle composé de `FormGroup` de `FormControl`

On le construit grâce à `FormBuilder`

```
import { FormBuilder, FormGroup, FormControl } from '@angular/forms';
export class ClientFormComponent implements OnInit {

  clientForm: FormGroup;

  constructor(private fb: FormBuilder) {
    this.clientForm = fb.group({
      identite: fb.group({
        prenom: fb.control(''),
        nom: fb.control(''),
      }),
      age: fb.control(''),
      vip: fb.control('')
    })
  }
}
```

Formulaires Model Driven

Dans le template, il convient

- de lier le `<form>` au `FormGroup` créé dans le modèle avec `[formGroup]`
- de lier chaque contrôle au `FormControl` correspondant avec `formControlName`
- de lier chaque `FormGroup` encapsulé avec `formGroupName`

Il est nécessaire également d'importer le module `ReactiveFormsModule` dans le module racine

```
import { ReactiveFormsModule } from '@angular/forms';
...
imports: [
  BrowserModule,
  ReactiveFormsModule
],
```

Formulaires Model Driven

Template du formulaire

```
<h2>Ajout d'un client</h2>
<form [formGroup]="clientForm">
  <fieldset formGroupName="identite">
    <div>
      <label>Prénom</label>
      <input type="text" formControlName="prenom">
    </div>
    <div>
      <label>Nom</label>
      <input type="text" formControlName="nom">
    </div>
  </fieldset>
  <div>
    <label>Age</label>
    <input type="number" formControlName="age">
  </div>
  <div>
    <label>VIP</label>
    <input type="checkbox" formControlName="vip">
  </div>

  <div>{{ clientForm.value |json }}</div>
</form>
```

Formulaires Model Driven

Pour la soumission du formulaire, la directive `ngSubmit` permet de spécifier l'opération à effectuer lors de l'appui sur un bouton `submit`

```
<form [formGroup]="clientForm" (ngSubmit)="ajouterClient()">
  ...
  <button type="submit">Ajouter</button>
</form>
```

Dans les formulaires *model driven*, il est possible de réinitialiser le formulaire dans le composant avec la méthode `reset()`

```
export class ClientFormComponent {
  ...
  ajouterClient() {
    console.log(this.clientForm.value);
    let formValues = this.clientForm.value;
    let c = new Client(-1, formValues.identite.nom, formValues.identite.prenom,
                                                                formValues.age, formValues.vip);

    this.clientService.addClient(c);
    this.clientForm.reset();
  }
}
```

Formulaires Model Driven

La validation de ces formulaire se fait par le biais de `Validators`

Les `Validators` sont des règles que le contrôle doit respecter

Dans les formulaires *model driven*, les validateurs sont ajoutés lors de la création des `FormControl`

Angular fournit un jeu de validateurs qui sont semblables aux attributs standards de validation HTML

- `required`
- `minlength`
- `maxlength`
- `pattern`

On communique au `FormControl` ses validateurs comme deuxième paramètre lors de sa création

Formulaires Model Driven

Importation de
Validators

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

Ajout des règles

```
this.clientForm =  
  fb.group({ identite:  
    fb.group({  
      prenom: fb.control('', Validators.required),  
      nom: fb.control('', [Validators.required, Validators.minLength(3)]),  
    }),  
    age: fb.control('', Validators.required),  
    vip: fb.control('')  
  })  
});
```

```
email: fb.control('',  
  [  
    Validators.required,  
    Validators.pattern("[^ @]*@[^ @]*")  
  ])
```

Formulaires Model Driven

Dans le template, il est possible d'accéder aux attributs des champs

```
{{ clientForm.controls.identite.controls.prenom.dirty }}  
{{ clientForm.controls.identite.controls.nom.valid }}  
{{ clientForm.controls.identite.valid }}  
{{ clientForm.controls.age.touched }}
```

Combiné à `ngClass`, cela permet par exemple de paramétrer les classes du champ (ou d'un autre élément) selon les attributs du champ

```
<div class="form-group" [ngClass]="{  
  'has-danger': myform.controls.email.invalid && myform.controls.email.dirty,  
  'has-success': myform.controls.email.valid && myform.controls.email.dirty  
}">
```

Formulaires Model Driven

Avec `ngIf`, grâce à la validation, il devient possible d'afficher des messages à destination de l'utilisateur (messages d'erreurs,...)

```
<div *ngIf="nom.invalid && nom.dirty">Nom  
  invalide</div>
```

Les messages peuvent être plus précis

```
<div *ngIf="nom.errors && (nom.dirty || nom.touched)">  
  <p *ngIf="nom.errors.required">Le nom est obligatoire</p>  
  <p *ngIf="nom.errors.minlength">  
    Le nom est trop court ({{ nom.errors.minlength.actualLength }}  
    au lieu de minimum {{ nom.errors.minlength.requiredLength  
    }})</p>  
</div>
```

Syntaxe alternative : `nom.hasError('required')`

Avec `disabled`, il est possible de désactiver des éléments, par exemple le bouton `submit`

```
<button type="submit"  
  [disabled]="!clientForm.valid">Ajouter</button>
```


Formulaires Template Driven

Le formulaire *template driven* se construit directement dans le template via des directives

Angular prend alors en charge la représentation du formulaire en créant les `FormControl` et `FormGroup` nécessaires

Le module `FormsModule` doit être importé dans le module principal de l'application

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

```
imports: [  
    ...  
    FormsModule  
],
```

Formulaires Template Driven

Les champs du formulaire sont enrichis par la directive `ngModel`

- Permet d'automatiser la création des `FormControl`

L'attribut `name` de chaque champ sera utile pour la construction automatique du `FormGroup`

La balise `<form>` automatise la création du `FormGroup`

Un formulaire soumis émet l'évènement `ngSubmit`

Pour récupérer les valeurs saisies, on définit pour le formulaire une variable locale qui référencera le formulaire

Formulaires Template Driven

Exemple de formulaire *template driven*

```
<h2>Ajout d'un client</h2>
<form
  (ngSubmit)="ajouterClient(clientForm.value)
  " #clientForm="ngForm">
  <div>
    <label>Prénom</label>
    <input type="text" name="prenom" ngModel>
  </div>
  <div>
    <label>Nom</label>
    <input type="text" name="nom" ngModel>
  </div>
  <div>
    <label>Age</label>
    <input type="number" name="age" ngModel>
  </div>
  <div>
    <label>VIP</label>
    <input type="checkbox" name="vip" ngModel>
  </div>
  <button type="submit">Ajouter</button>
</form>
```

Formulaire Template Driven

Il est possible d'avoir un binding bi-directionnel en liant le formulaire au modèle

Deux notations possibles

- La notation verbeuse

```
<input type="text" name="nom"  
      [ngModel]="client.nom"  
      (ngModelChange)="client.nom = $event">
```

- La "*banana in a box*" [()] (boîte à banane)

```
<input type="text" name="nom" [(ngModel)]="client.nom">
```

Avec le binding bi-directionnel, les valeurs du formulaires deviennent accessibles "en live" dans le template

```
{{ client.nom }}
```

Formulaire Template Driven

Exemple :

Template

Composant

```
<form (ngSubmit)="ajouterClient()">
  <div>
    <label>Prénom</label>
    <input type="text" name="prenom"
           [(ngModel)]="client.prenom">
  </div>
  <div>
    <label>Nom</label>
    <input type="text" name="nom"
           [(ngModel)]="client.nom">
  </div>
  <div>
    <label>Age</label>
    <input type="number" name="age"
           [(ngModel)]="client.age">
  </div>
  <div>
    <label>VIP</label>
    <input type="checkbox" name="vip"
           [(ngModel)]="client.vip">
  </div>
  <button type="submit">Ajouter</button>
</form>
```

```
client = new Client();

constructor(private clientService: ClientService)
{ }

ajouterClient() {
  this.clientService.addClient(this.client)
  ; this.client = new Client();
}
```

Formulaire Template Driven

La validation dans les formulaires *template driven* s'effectue par l'intermédiaire de plusieurs directives à ajouter aux `input` permettant de spécifier les règles de validation

- `required`
- `minlength`
- `maxlength`
- `pattern`

```
<form (ngSubmit)="ajouterClient()" #clientForm="ngForm">

  <input type="text" name="nom"
        [(ngModel)]="client.nom"
        required
        minlength="3">

  <button type="submit"
        [disabled]="clientForm.invalid">Ajouter</button>
```

Comme pour les formulaires *model driven*, on peut accéder aux informations de validation dans le template

```
{{ clientForm.form.controls.nom?.valid }} / {{ clientForm.controls.nom?.pristine }}
{{ clientForm.form.valid }} / {{ clientForm.valid }}
```

Astuce :

```
<input type="text" name="nom"
      [(ngModel)]="client.nom" #nom="ngModel">
On peut alors utiliser directement {{ nom.valid }} ou autre
```

? car le contrôle peut être null quand Angular construit la page

Pas besoin en *model driven* car le modèle est créé dans le composant

TP Angular n°7 – Formulaire

Créer un composant FlightFormComponent permettant d'ajouter un vol dans le tableau géré par le service

- Y implémenter un formulaire piloté par le code(Reactive forms)

Afficher ce composant dans le composant FlightListComponent



Le routing

Le routing

Le routing permet de lier l'URL à un état particulier de l'application

Il se révèle très utile dans le cadre d'une application SPA (*Single Page Application*)

Sous Angular, cela s'effectue par l'intermédiaire d'un module nommé *RouterModule*

La configuration

Ajouter dans le index.html la balise `base` contenant la partie statique du site

```
<head>
  <base href="/">
  ...
</head>
```

Une route Angular associe une URL à un composant qui sera affiché

Fichier `app.routes.ts`

URL

```
import { Routes } from
 '@angular/router';
export const appRoutes: Routes = [
  { path: 'view', component: ClientListComponent
  }, { path: 'add', component: ClientFormComponent },
  { path: '', redirectTo: '/view', pathMatch:
  'full' }, { path: '**', component: NotFoundComponent
  }
];
```

Composant

full : matching total
prefix : matching partiel
Obligatoire pour les redirections

La configuration

Une fois les routes configurées, il convient d'importer le routeur dans notre module principal

```
import { RouterModule } from '@angular/router';
import { appRoutes } from './app.routes';
...
imports: [
  ...
  RouterModule.forRoot(appRoutes),
  ...
],
```

Pour logger les routes

```
RouterModule.forRoot(appRoutes, { enableTracing: true })
```

Ensuite, la balise `<router-outlet>` permet d'activer le routeur sur le composant voulu (en général le composant principal du module)

```
<router-outlet></router-outlet>
```

La navigation

La navigation s'effectue au moyen de la directive `routerLink` en donnant la route voulue

Cette route sera mise en place dans le `router-outlet`

```
<a routerLink="/view" routerLinkActive="active">Liste</a>  
<a routerLink="/add" routerLinkActive="active">Ajouter</a>
```

La directive `routerLinkActive` (facultative) permet d'ajouter une classe particulière au lien lorsque sa route est active

La navigation

Dans le code d'un composant, on peut naviguer vers une route :

- Injecter le service dans le composant

```
import { Router } from '@angular/router';  
  
constructor(..., private router: Router) {  
  
}
```

- Utiliser le service Router

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

Les paramètres

Les routes peuvent comporter des paramètres

- Les paramètres, dans l'URL de la route sont préfixés par le caractère ':'

```
{ path: 'view/:id', component: ClientViewComponent },
```

Récupérer le paramètre via un *snapshot* (ne fonctionnera pas si le paramètre change à l'intérieur du même composant)

```
constructor(private clientService: ClientService,  
             private route: ActivatedRoute) { }  
  
ngOnInit() {  
    const id =  
        this.route.snapshot.paramMap.get("id");  
    this.client = this.clientService.getClient(id);  
}
```

Les paramètres

Autre possibilité plus avancée pour la gestion des paramètres

Récupérer le paramètre dans le composant concerné via un Observable appelé `paramMap` fourni par `ActivatedRoute`

```
this.client =  
    this.route.paramMap.pipe(  
        switchMap((params: ParamMap) =>  
            this.service.getClient(params.get('id'))  
        )  
    ).subscribe();
```

Les paramètres

La navigation dans le template peut se faire de deux façons

```
<a [routerLink]="['/view/',1]" routerLinkActive="active">Voir 1</a>
```

```
<a routerLink="/view/1" routerLinkActive="active">Voir 1</a>
```

Navigation par le code

```
this.router.navigate(['view', client.id]);
```


Les guards

Les *guards* permettent de gérer la sécurité de l'application côté *front*

- Il s'agit par exemple d'empêcher l'utilisateur d'accéder à une route particulière

4 guards sont mis à notre disposition

- `CanActivate` : Gère l'accès à une route
- `CanDeactivate` : Permet d'empêcher l'utilisateur de quitter la route actuelle
- `CanActivateChild` : Gère l'accès aux routes enfants
- `CanLoad` : Gère l'accès à une route qui utilise le *lazy loading*. Empêche le chargement du module même

Il convient d'implémenter les interfaces qui portent le même nom que les *guards* qu'on souhaite utiliser

Les guards

Avec Angular CLI

- `ng generate guard auth/auth`
- Génèrera un *guard* `AuthGuard` implémentant `CanActivate` dans le repertoire `auth`

Pour le guard `CanActivate`, la méthode `CanActivate` sera appelée

L'accès à la route sera autorisé si la méthode retourne :

- `true`
- un `Observable` émettant `true`
- une promesse résolue à `true`

Si la méthode retourne un objet `UrlTree`, alors il y aura redirection vers l'`UrlTree`

- `this.router.parseUrl('/login');`
- `this.router.createUrlTree(['/login']);`

CanActivate

Ecriture du *guard*

```
export class AuthGuard implements CanActivate {  
  constructor(private authService: AuthService) {  
  
  }  
  
  canActivate(  
  
    next: ActivatedRouteSnapshot,  
    state: RouterStateSnapshot): boolean {  
    console.log(this.authService.isLoggedIn);  
    return this.authService.isLoggedIn;  
  }  
}
```

Utilisation du *guard* dans la définition d'une route

```
{ path: 'add', component: ClientFormComponent, canActivate: [AuthGuard] }
```

CanActivate

Il est possible d'injecter le service Router pour faire une redirection au cas où l'accès n'est pas autorisé

```
export class AuthGuard implements CanActivate {  
  
  constructor(private authService: AuthService,  
               private router: Router) { }  
  
  canActivate(  
    next: ActivatedRouteSnapshot,  
    state: RouterStateSnapshot): boolean {  
    console.log(this.authService.isLoggedIn);  
    if(this.authService.isLoggedIn)  
      return true;  
  
    this.router.navigate(['/login']);  
    return false;  
  }  
}
```

CanActivate

On retrouve deux paramètres dans la méthode `CanActivate`

- `next: ActivatedRouteSnapshot` : La future route qui sera activée si l'accès est autorisé
- `state: RouterStateSnapshot` : **Futur RouterState** si l'accès est autorisé
 - L'attribut `url` est intéressant car il contient l'URL de la route testée qui peut être stockée éventuellement pour faire une redirection après le login

CanActivateChild

Le *guard* `CanActivateChild` fonctionne de manière similaire au *guard* `CanActivate`, Il permet d'accorder ou non l'accès aux routes enfants

```
export class AuthGuard implements CanActivate, CanActivateChild {
  ...
  canActivateChild(route: ActivatedRouteSnapshot, state: RouterStateSnapshot)
  {
    // Pas d'autorisation aux routes enfants
    return false;
  }
  ...
}

{ path: 'view/:id',
  component: ClientViewComponent,
  children: [
    { path: 'ca', component: ClientCADetailsComponent },
    { path: 'factures', component: ClientFacturesDetailsComponent }
  ],
  canActivate: [AuthGuard], canActivateChild: [AuthGuard]
}
```

CanDeactivate

Ce *guard* permet d'empêcher de sortir d'une route

```
export class AuthGuard
implements CanActivate, CanActivateChild, CanDeactivate<ClientFormComponent> {
  ...
  canDeactivate(component: ClientFormComponent): boolean {
    return window.confirm("Sûr de vouloir quitter ?");
  }
  ...
}
```

```
{ path: 'add',
  component: ClientFormComponent,
  canActivate: [AuthGuard],
  canDeactivate: [AuthGuard]
},
```

CanLoad

Ce *guard* permet d'empêcher le chargement d'un module configuré en *lazy loading*

Il implémente l'interface CanLoad

```
export class ModulesGuard implements CanLoad {  
  
    canLoad(route: Route, segments: UrlSegment[]): boolean {  
        ...  
        return false;  
    }  
}
```

- Le *guard* peut aussi retourner une promesse résolue en booléen ou un Observable émettant un booléen

```
{ path: 'produits',  
  loadChildren:  
    './produits/produits.module#ProduitsModule', canLoad:  
    [ModulesGuard] },
```


Stratégies de routage

Angular propose deux stratégies de routage

- *HashLocationStrategy*
- *PathLocationStrategy*

Par défaut, c'est *PathLocationStrategy* qui est utilisée

Pour utiliser *HashLocationStrategy*, dans le module principal

```
RouterModule.forRoot(appRoutes, {useHash: true}),
```

En *HashLocationStrategy*, la route est écrite derrière un hash #

- Tout ce qui est derrière # n'est pas envoyé au serveur par le navigateur
- Une seule URL à gérer pour le serveur

En *PathLocationStrategy*, on utilise l'API HTML5 *History*

- Le serveur doit pouvoir retourner l'application pour plusieurs URL
- Stratégie utilisée pour le rendu côté serveur (*Angular Universal*)

TP Angular n°8 – Routing

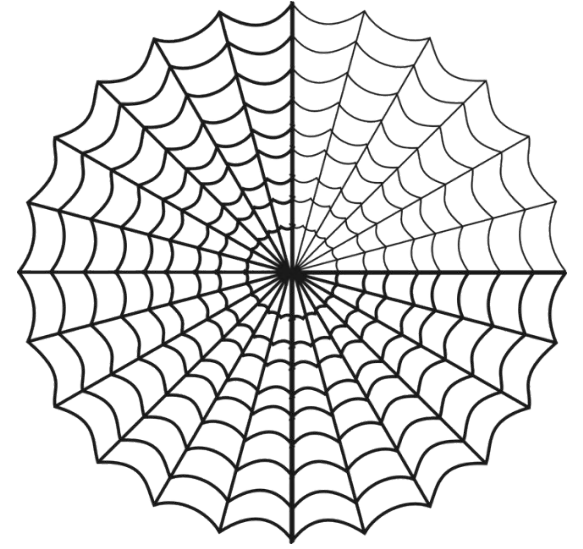
Configurer le module de routing avec 3 routes :

- Une route /flights affichant la liste des vols (composant FlightListComponent)
- Une route /add permettant d'ajouter un vol (composant FlightFormComponent)
- Une route /edit/:id permettant d'ajouter un vol (composant FlightFormComponent)
- Une route paramétrée /flight/ :id permettant d'afficher un vol particulier (composant FlightDetailsComponent)

Modifier le projet pour y ajouter une barre de navigation avec 2 menus

- Voir les vols
- Ajouter un vol

Modifier le bouton « Détails » afin d'aller vers la route paramétrée



HTTP

Le module HttpClientModule

Dès la première version, Angular était livré avec un module dédié aux opérations HTTP

Il s'agit du module HttpClientModule

```
import { HttpClientModule } from
  '@angular/common/http';

imports: [
  ...
  HttpClientModule,
  ...
]
```

Le service HttpClient

Pour envoyer et recevoir de la donnée, Angular fournit un service appelé `HttpClient` via le module `HttpClientModule`

On trouve tout cela dans le package `@angular/common/http`

Il convient d'importer le module `HttpClientModule` dans le module principal de l'application

```
import { HttpClientModule } from '@angular/common/http';

imports: [
    ...
    HttpClientModule,
    ...
]
```

Le service HttpClient

Le service `HttpClient` fournit les outils permettant de réaliser des requêtes AJAX en fournissant des méthodes qui correspondent aux méthodes HTTP classiques, en particulier :

- `get`
- `post`
- `put`
- `delete`

Toutes les méthodes du service `HttpClient` retournent un `Observable` auquel il faudra s'inscrire

Pour utiliser le service `HttpClient`, il convient de l'injecter dans les éléments qui l'utiliseront (composants, services,...)

```
import { HttpClient } from '@angular/common/http';

constructor(private http: HttpClient) {
    ...
}
```

L'utilisation de HttpClient

Pour utiliser ce service, il convient d'appeler la méthode voulue

```
http.get<Client[]>(`${this.apiRoot}/api/clients.json`)
  .subscribe(clients =>
    {
      console.log(clients)
    });
```

On récupère directement le résultat

Pour faire du POST, on ajoute à la méthode `post` de `HttpClient`

l'objet à poster

```
http.post(`${this.apiRoot}/api/clients`,
  newClient)
```

Note : les requêtes HTTP ne seront pas exécutées tant qu'une inscription n'aura pas été faite sur l'Observable

```
const req = http.post(`${this.apiRoot}/api/clients`, newClient);
req.subscribe();
```

L'utilisation de HttpClient

Pour gérer l'erreur, on peut communiquer à `subscribe` un deuxième callback en cas d'erreur

```
http.get<Client[]>(`${this.apiRoot}/api/clients.json`)
  .subscribe(
    data => console.log(data),
    err => console.log('Erreur !');
  );
```

Un troisième callback peut être mentionné. Ce callback est exécuté quand l'Observable est terminé

```
http.get<Client[]>(`${this.apiRoot}/api/clients.json`)
  .subscribe(
    data => console.log(data),
    err => console.log(err),
    () => console.log('completed')
  );
```

En cas d'erreur, l'opérateur RxJS `retry` peut être utilisé pour relancer la requête un certain nombre de fois

```
import 'rxjs/add/operator/retry';
...
http.get<Client[]>(`${this.apiRoot}/api/clients.json`)
  .pipe(retry(2))
  .subscribe(data => console.log(data));
```


L'utilisation de HttpClient

L'erreur renvoyé est de type `HttpErrorResponse`

2 types d'erreur :

- Erreur côté client (problème RxJs ou problème réseau)
- Erreur côté serveur

```
http.get<Client[]>(`${this.apiRoot}/api/clients.json`)
  .subscribe(
    data => console.log(data),
    (err: HttpErrorResponse) => {
      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}`);
      }
    }
  );
```

L'utilisation de HttpClient

Par défaut, `HttpClient` retourne le résultat de la requête

Parfois, on souhaite obtenir plus d'informations sur le résultat retourné d'un point de vue HTTP

On utilise pour cela l'option `observe`

```
http.get(`${this.apiRoot}/api/clients.json`, {observe: 'response'})  
  .subscribe(res => console.log(res));
```

Le résultat se présente alors sous la forme d'un objet `HttpResponse` qui contient des informations complémentaires, entre autres :

- `status` : le code de retour HTTP
- `body` : la réponse en tant que telle

Le traitement du résultat

Exemple de traitement du résultat

client.service.ts

```
import { Observable } from 'rxjs/Observable';

constructor(private http: HttpClient) { }

getClients(): Observable<Client[]> {
    return this.http.get<Client[]>(`${this.apiRoot}/api/clients.json`);
}
```

Composant : possibilité 1

```
clients: Client[];

constructor(private clientService: ClientService) {
}

getClients() {
    this.clientService.getClients(
    )
        .subscribe ( res => this.clients = res);
}

ngOnInit() {
    this.getClients();
}
```


<app-client *ngFor="let client of clients"
[client]=client></app-client>

Composant : possibilité 2

```
clients: Observable<Client>;

constructor(private clientService: ClientService) {
}

getClients() {
    this.clients = this.clientService.getClients();
}

ngOnInit() {
    this.getClients();
}
```


<app-client *ngFor="let client of clients |
async"
[client]=client></app-client>

Le passage de paramètres

Il est possible de passer des paramètres dans l'URL avec `HttpClient`

On utilise pour cela `HttpParams`

```
import { HttpParams } from '@angular/common/http';
...
return http.get(`${this.apiRoot}/api/clients.json`,
  {
    params: new HttpParams()
      .set('limit', '20')
      .set('offset', '80')
  });
```

JSONP

Angular propose des outils permettant de traiter du JSONP (*JSON Padding*), en particulier pour éviter les problèmes de *Same Origin Policy*

Auparavant, un module `JsonpModule` était proposé pour cela

- Ce module est aujourd'hui déprécié depuis la version 4.3

L'utilisation du JSONP passe maintenant par `HttpClient` qui propose une méthode `jsonp` qui accepte 2 paramètres :

- L'URL de la ressource
- Le nom du callback

Pour pouvoir utiliser le JSONP, il convient cependant d'importer le module `HttpClientJsonpModule`. Ce module permet d'intercepter la requête et ainsi de gérer le traitement du JSONP

JSONP

Exemple

- Fichier `app.module.ts`

```
import { HttpClientJsonpModule } from '@angular/common/http';  
...  
imports: [  
    ...  
    HttpClientJsonpModule,  
    ...  
],
```

- Dans le service ou le composant

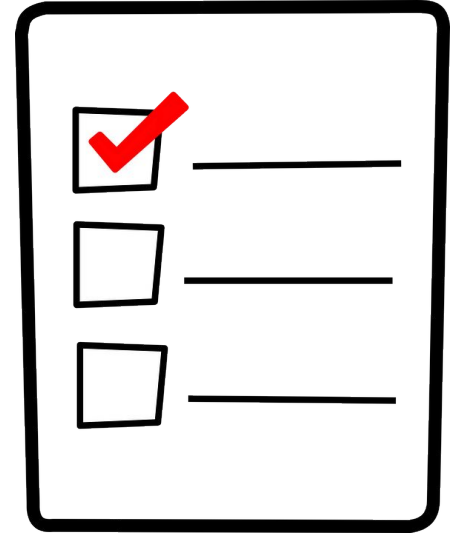
```
this.http.jsonp("https://archive.org/index.php?output=json&callback=JSONP_CALLBACK", 'JSONP_CALLBACK')  
    .subscribe(  
        res => console.log(res)  
    );
```

TP Angular n°9 – HTTP

Modifier le service FlightService afin que ce dernier puisse exploiter une API REST

Modifier les composants en conséquence

Les tests



Les tests

Le framework Angular propose deux types de tests :

- Les tests unitaires
- Les tests e2e (*end-to-end*)

Chacun de ces deux types de tests a une finalité particulière

Les tests unitaires

Les tests unitaires permettent de tester unitairement un morceau de code

- Ils sont courts donc très rapides
- Ils permettent de tester des cas très spécifiques, voire improbables

Il peut concerner un composant, un service ou tout autre élément Angular

L'idée est de s'assurer qu'il fonctionne correctement lorsqu'il est isolé

Lorsqu'on écrit un test unitaire, on s'efforce de neutraliser les dépendances

- Il conviendra d'écrire des *mocks objects* (objets fictifs) qui seront utilisés pour le test

Les tests unitaires

Pour écrire les tests unitaires, on utilise le framework Jasmine

- <https://jasmine.github.io/>

En association à la création de nos différents éléments, Angular CLI a créé un fichier `spec.ts`

- Ce dernier contient les tests unitaires associés à notre élément

`app.component.spec.ts`

```
describe("Test de Jasmine", () => {  
  it("is working", () => expect(true).toBe(true));  
});
```

Les tests unitaires

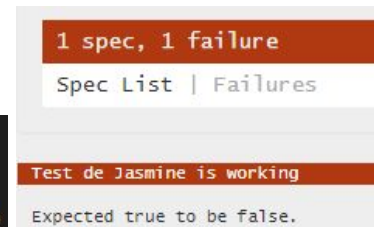
Pour lancer les tests de l'application, on utilise la commande `ng test`. Cette commande lance Karma pour exécuter les tests et le résultat est aussi indiqué sur la console.



```
Chrome 71.0.3578 (Windows 10 0.0.0): Executed 1 of 1 SUCCESS (0.122 secs / 0.013 secs)
```

Si un des tests n'est pas concluant, une information est aussi communiquée.

```
Chrome 71.0.3578 (Windows 10 0.0.0) Test de Jasmine is working FAILED
  Expected true to be false.
    at Object.<anonymous> (http://localhost:9876/src/app/app.component.spec.ts?:2:39)
    at ZoneDelegate../node_modules/zone.js/dist/zone.js.ZoneDelegate.invoke (http://localhost:9876/node_modules/zone.js/dist/zone.js?:388:1)
    at ProxyZoneSpec.push../node_modules/zone.js/dist/proxy.js.ProxyZoneSpec.onInvoke (http://localhost:9876/node_modules/zone.js/dist/proxy.js?:128:1)
Chrome 71.0.3578 (Windows 10 0.0.0): Executed 1 of 1 (1 FAILED) ERROR (0.101 secs / 0.086 secs)
```





Karma

Karma (<https://karma-runner.github.io>) est l'outil qui permet d'exécuter les tests et il peut être configuré, par exemple, pour les exécuter sur plusieurs navigateurs

La configuration de Karma se trouve dans le fichier

karma.conf.js Par exemple, pour ajouter FireFox dans les navigateurs testés

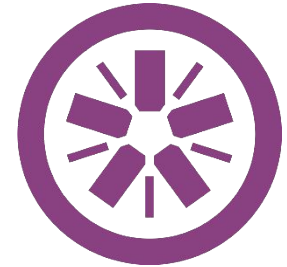
- Installation du plugin FireFox de Karma

```
npm install karma-firefox-launcher --save-dev
```

- Modification du fichier de configuration

```
require('karma-firefox-launcher'),  
...  
browsers: ['Chrome', 'Firefox'],
```

Karma exécute tous les tests dès qu'il détecte une modification dans les fichiers de l'application



Jasmine

Le framework Jasmine met à notre disposition plusieurs méthodes afin de décrire nos tests unitaires

Méthode	Utilité
<code>describe(description, function)</code>	Grouper des tests
<code>beforeEach(function)</code>	Tâche à effectuer avant chaque test
<code>afterEach(function)</code>	Tâche à effectuer après chaque test
<code>it(description, function)</code>	Décrit un test
<code>expect(value)</code>	Identifie le résultat d'un test

```
describe("Test de Jasmine", () => {  
  it("test numérique", () => expect(13).toBeLessThan(30));  
  it("test chaîne", () => expect("Bonjour").not.toContain('ov'));  
  it("test regex", () => expect("Bonjour").toMatch(/^Bon/));  
});
```

Jasmine

Pour la gestion du résultat attendu, Jasmine met à disposition plusieurs possibilités sous la forme de méthodes (voir la liste sur <https://jasmine.github.io/api/3.3/matchers.html>)

Méthode	Utilité
<code>toBe(value)</code>	Egalité stricte <code>===</code>
<code>toEqual(object)</code>	Egalité (avec parcours en profondeur pour les objets)
<code>toMatch(regex)</code>	Correspondance avec une expression régulière
<code>toBeDefined()</code>	Défini
<code>toBeUndefined()</code>	Non défini
<code>toBeNull()</code>	Null
<code>toBeTruthy()</code>	Vérité
<code>toBeFalsy()</code>	Non vérité (<code>false</code> , <code>0</code> , <code>""</code> , <code>null</code> , <code>undefined</code> , <code>NaN</code>)
<code>toContain(substring)</code>	Contient une sous-chaîne
<code>toBeLessThan(value)</code>	Inférieur à
<code>toBeGreaterThan(value)</code>	Supérieur à

Jasmine

```
class Point {
  constructor(public x: number, public y:number) { }
  addX() { this.x++; }
}
describe("Test de Point", () => {

  const initValue = 3;

  let point: Point;

  beforeEach( () => {
    point = new Point(initValue, initValue);
  })

  it("be initialized", () => {
    expect(point.x).toEqual(initValue);
    expect(point.y).toEqual(initValue);
  });

  it("add one to X", () => {
    point.addX();
    expect(point.x).toEqual(initValue+1);
    expect(point.y).toEqual(initValue);
  });
});
```


Jasmine - Test de composants

Une brique d'une application Angular, par exemple un composant, ne peut pas être testée de façon isolée car elle dépend de fonctionnalités internes à Angular pour son fonctionnement

Ainsi, pour tester un composant, il est nécessaire de créer un environnement minimal

Pour cela, la classe `TestBed` est mise à notre disposition

Elle permet de simuler l'environnement Angular

- Initialisation d'un composant
- Injection de services

Jasmine - Test de composants

Utilisation de TestBed

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { BonjourComponent } from '../bonjour.component';

describe('BonjourComponent', () => {
  let component: BonjourComponent;
  let fixture: ComponentFixture<BonjourComponent>;

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [ BonjourComponent ]
    });
    fixture = TestBed.createComponent(BonjourComponent);
    component = fixture.componentInstance;
  });

  it('is defined', () => {
    expect(component).toBeDefined();
  });
});
```

Configure le module test Angular

Crée une instance du composant

Jasmine - Test de composants

L'objet de type `ComponentFixture` propose plusieurs méthodes et attributs

Méthode / Attribut	Utilité
<code>componentInstance</code>	L'objet composant
<code>debugElement</code>	Elément hôte du composant
<code>nativeElement</code>	DOM de l'élément hôte du composant
<code>detectChanges()</code>	Lance la détection du changement pour mettre à jour le template
<code>whenStable()</code>	Retourne une promise résolue quand une opération a été complètement appliquée

Jasmine - Injection de dépendances

Lorsqu'on doit tester un composant qui utilise l'injection de dépendance, il convient de créer des *mocks* (simulacres) et de les injecter lors de l'initialisation du composant

```
describe('ArticlesComponent', () =>
{
  let component:
  ArticlesComponent;
  let fixture: ComponentFixture<ArticlesComponent>;

  let mockArticlesService =
  {
    getArticles:
    function() {
      return [
        { libelle: 'Tomates', prix: 1.54, categorie: 'Alimentaire' },
        { libelle: 'Pantalon', prix: 29.65, categorie: 'Habillement' },
        { libelle: 'Biscuit', prix: 2.96, categorie: 'Alimentaire' },
      ]
    }
  }

  beforeEach(() => {
    TestBed.configureTestingModule(
    {
      declarations: [ ArticlesComponent ],
      providers: [{provide: ArticlesService, useValue: mockArticlesService}]);
    fixture = TestBed.createComponent(ArticlesComponent);
    component = fixture.componentInstance;

  });

  it('filters categories', () => {
    component.categorie = 'Alimentaire';
    expect(component.getArticles().length).toBe(2);   component.categorie = 'Jardinage';
    expect(component.getArticles().length).toBe(0);   component.categorie =
    'Habillement'; expect(component.getArticles().length).toBe(1);
  });
});
```

Jasmine - Test de composants

Pour tester le composant, on va le créer, initialiser ses Input puis lancer la détection du changement manuellement

```
it('filters categories', () => {
  // Contexte
  TestBed.configureTestingModule({
    declarations:
      [ArticlesComponent], providers: [{provide: mockArticlesService }
      ]
  });
  const fixture =
    TestBed.createComponent(ArticlesComponent); const
    component = fixture.componentInstance;
    component.categorie = 'Alimentaire';

  // Détection du
  changement
  fixture.detectChanges();

  const element = fixture.debugElement;
  const bindingElement: HTMLSpanElement =
    element.query(By.css('span#nba')).nativeElement;
```

```
<h1>Bonjour !</h1>
  <p>Nombre
    d'articles</p></span>{{ this.getArticles().length
  </p></span>
```

```
expect(bindingElement.textContent).toBe('2');
```

Jasmine - Test de composants

Un objet de type `DebugElement` représente un élément

Méthodes / Attributs	Utilité
<code>nativeElement</code>	Objet qui représente l'élément HTML dans le DOM
<code>children</code>	Un tableau de <code>DebugElement</code> enfants
<code>query(selector)</code>	Retourne le premier <code>DebugElement</code> qui correspond au critère
<code>queryAll(selector)</code>	Retourne tous les <code>DebugElement</code> qui correspondent au critère
<code>triggerEventHandler(name, event)</code>	Déclenche un évènement

Pour la sélection avec `query` et `queryAll` on utilise la classe

`By`

Méthodes	Utilité
<code>By.all()</code>	Matche tous les éléments
<code>By.css(selector)</code>	Matche selon un sélecteur CSS
<code>By.directive(type)</code>	Matche selon une directive

Jasmine - Test de composants

Après création d'un composant il est possible de modifier certains de ses attributs (template, styles ,dépendances,...)

```
TestBed.configureTestingModule({
  declarations:
  [ArticlesComponent], providers:
  [
    { provide: ArticlesService, useValue: mockArticlesService }
  ]
});
TestBed.overrideComponent(ArticlesComponent, { set: { template: '<h3>{{this.getArticles().length }}</h3>' }
}); const fixture = TestBed.createComponent(ArticlesComponent);
```

Particulièrement pour le template

```
TestBed.configureTestingModule({
  declarations:
  [ArticlesComponent], providers:
  [
    { provide: ArticlesService, useValue: mockArticlesService }
  ]
});
TestBed.overrideTemplate(ArticlesComponent, '<h3>{{this.getArticles().length
}}</h3>'); const fixture = TestBed.createComponent(ArticlesComponent);
```

Jasmine - Test de composants

Pour tester les Output il convient de poster des espions avec la fonction `spyOn`

```
let fixture:
ComponentFixture<ArticlesComponent>; let
debugElement: DebugElement;
beforeEach(() => {
  TestBed.configureTestingModule(
    {
      declarations:
        [ArticlesComponent], providers:
        [
          { provide: ArticlesService, useValue: mockArticlesService }
        ]
    });

  fixture =
    TestBed.createComponent(ArticlesComponent);
  debugElement = fixture.debugElement;
});

it('refresh', () => {
  const component = fixture.componentInstance;
  const button =
    debugElement.query(By.css('button'));
  spyOn(component.refresh, 'emit');
```

```
button.nativeElement.click();
expect(component.refresh.emit).toHaveBeenCalled(2)
```


Code coverage




L'option `--code-coverage` ajoutée à la commande `ng test` permet d'avoir un rapport de *code coverage*

```
ng test --watch=false --code-coverage
```

Un nouveau dossier `coverage` est alors créé dans le dossier du projet, qui contient lui-même un fichier `index.html` qui contient le

All files

97.3% Statements 36/37 100% Branches 0/0 88.89% Functions 8/9 96.88% Lines 31/32

File ▲		Statements ▾		Branches ▾		Functions ▾		Lines ▾	
src		100%	15/15	100%	0/0	100%	1/1	100%	15/15
src/app		85.71%	6/7	100%	0/0	66.67%	2/3	80%	4/5
src/app/articles		100%	15/15	100%	0/0	100%	5/5	100%	12/12

Code coverage

Il est possible de générer automatiquement le rapport à chaque lancement des tests en ajoutant l'option adéquate dans le fichier `angular.json`

```
"test": {  
  "options": {  
    "codeCoverage":  
      true  
  }  
}
```

Les tests e2e

Dans un test e2e on va réellement simuler l'exécution de l'application dans un navigateur

Cependant, ils sont plus lents que les tests unitaires et ils ne permettront bien souvent que de tester des cas normaux

Pour la gestion des tests e2e Angular utilise Protractor (<http://www.protractortest.org>)



Protractor

Protractor met à notre disposition différentes méthodes et objets pour travailler, en particulier l'objet `browser`

Dans une application Angular, on retrouve tous les tests e2e dans le répertoire `e2e`

La configuration de Protractor se trouve dans le fichier `protractor.conf.js`

- L'option `directConnect` positionnée à `true` permet de communiquer directement avec le navigateur (Chrome et parfois Firefox supportés). Sinon, il faut passer par un serveur Selenium

Pour lancer les tests Protractor on utilise la commande `ng e2e`

Protractor

```
import { browser, by, element } from  
'protractor';
```

```
export class ClientsCRUDPage  
{ navigateTo() {  
  return browser.get('/');  
}
```

```
getTitrePrincipal() {  
  return element(by.css('.container  
} h1')).getText();  
}
```

app.po.ts

app.e2e-spec.ts

```
import { ClientsCRUDPage } from '../app.po';
```

```
describe('clients-crud App', () => {  
  let page: ClientsCRUDPage;
```

```
  beforeEach(() => {  
    page = new ClientsCRUDPage();  
  });
```

```
  it('should display main title', () => {  
    page.navigateTo();  
    expect(page.getTitrePrincipal()).toEqual('Gestion des clients');  
  });  
});
```

Protractor

Avec Protractor, il "suffit" de récupérer des éléments dans la page avec `element`

Pour récupérer des éléments on utilise `by`, par exemple (voir la liste sur <https://www.protractortest.org/#/api?view=ProtractorBy>)

Méthode	Utilité
<code>by.className (classe)</code>	Sélection par classe CSS
<code>by.css (selecteur)</code>	Sélection par sélecteur CSS
<code>by.id (identifiant)</code>	Sélection par id
<code>by.name (nom)</code>	Sélection par name
<code>by.binding (binding)</code>	Sélection par binding (ex : <code>'client.prenom'</code>)
<code>by.buttonTest (texte)</code>	Sélection de bouton par texte

Protractor

Une fois l'élément récupéré, il devient alors possible de le manipuler, avec par exemple (voir la liste sur

<https://www.protractortest.org/#/api?view=ElementFinder>)

Méthode	Utilité
<code>getText()</code>	Retourne le <code>innerText</code>
<code>getSize()</code>	Retourne la taille de l'élément
<code>sendKeys(texte)</code>	Envoie la séquence de caractères
<code>isEnabled()</code>	Pour savoir si l'élément est activé
<code>submit()</code>	Envoie le formulaire
<code>click()</code>	Clique sur l'élément

Ensuite les tests sont écrits avec Jasmine

Les bonnes pratiques

Angular/RxJS bonnes pratiques

- Éviter les fuites de mémoire `.subscribe()`
- Pas de subscribe dans un subscribe (utiliser `switchMap`, `mergeMap`)
- Interdire les fonctions dans les templates
- Utiliser NgRx avec précaution
 - Le cas typique est lors de la réception d'une notification dans un composant, une donnée dans un composant "éloigner" (ni parent, ni enfant) a besoin d'être rafraîchi.
- Éviter `ngModel` (déprécié à partir d'Angular 6)
- Petits composant et dumb/ container (smart)
- Découper en Sous-modules
- Prettier et TSLint (Ajouter les règles `TrackBy` et `function call from template`)
- Lazy loading

SharedModule

SharedModule est un simple module Angular, dont le rôle est de déclarer et exporter tous les composants, directives et pipes susceptibles d'être réutilisés partout dans le projet.

Le nom “SharedModule” est une simple convention, et vous n'êtes pas obligé de créer un tel module dans votre projet.

Le SharedModule contient généralement des éléments d'interface réutilisables (barre de navigation, HTML pour afficher un champ de formulaire ou un tableau...) ou des directives et pipes très génériques

SharedModule

Certains modules sont utilisés par quasiment tous les composants de l'application (e.g.: CommonModule, FlexLayoutModule, RouterModule). Les importer dans chaque module peut s'avérer pénible. Il est courant de factoriser ces imports en rassemblant toutes ces dépendances dans un module SharedModule importé par quasiment tous les autres modules de l'application

```
@NgModule({  
  imports: [CommonModule],  
  declarations: [NavbarComponent, TableComponent, HighlightDirective, MarkdownPipe...],  
  exports: [NavbarComponent, TableComponent, HighlightDirective, MarkdownPipe...]  
})  
export class SharedModule {}
```

SharedModule



Attention à ne pas trop surcharger ce module et en faire un "God Module".

N'y importez que les modules nécessaires pour quasiment tous les composants de l'application.

PWA : Services Workers

Un Service Worker est un script chargé parallèlement aux scripts de la page et qui va s'exécuter en dehors du contexte de la page web.

Bien que le Service Worker n'ait pas accès au DOM ou aux interactions avec l'utilisateur, il va pouvoir communiquer avec vos scripts via l'API **postMessage**.

Il se place en proxy de votre Web App, interceptant toutes les requêtes serveur et propose par exemple d'y répondre avec un cache ou en récupérant des données du LocalStorage ou d'IndexedDB. Il rend donc votre application disponible offline.

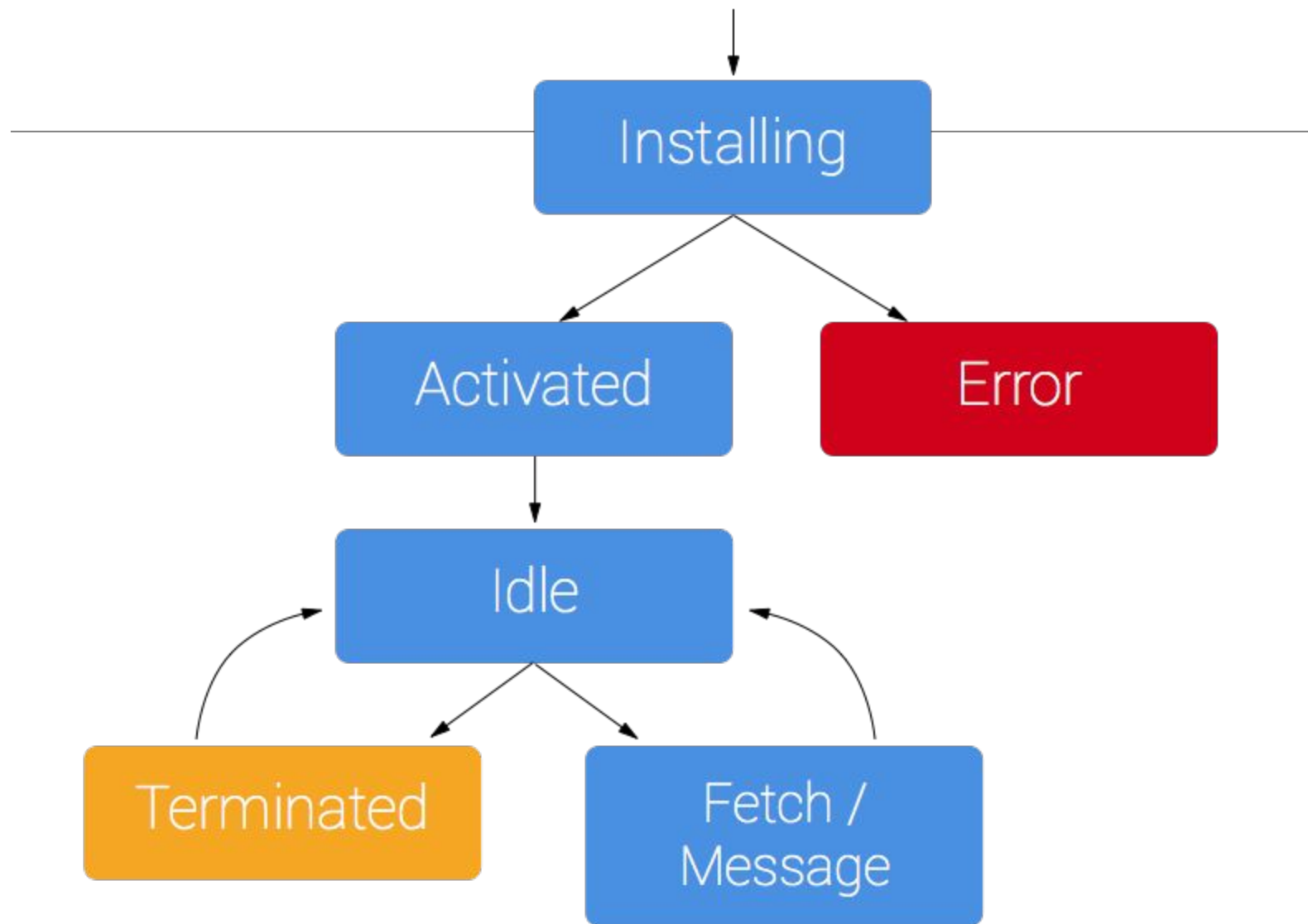
PWA : Services Workers

Un Service Worker est un script chargé parallèlement aux scripts de la page et qui va s'exécuter en dehors du contexte de la page web.

Bien que le Service Worker n'ait pas accès au DOM ou aux interactions avec l'utilisateur, il va pouvoir communiquer avec vos scripts via l'API **postMessage**.

Il se place en proxy de votre Web App, interceptant toutes les requêtes serveur et propose par exemple d'y répondre avec un cache ou en récupérant des données du LocalStorage ou d'IndexedDB. Il rend donc votre application disponible offline.

No Service
Worker





Références et Bibliographie

Références et Bibliographie

Le site officiel d'Angular et sa documentation

- <https://angular.io/>

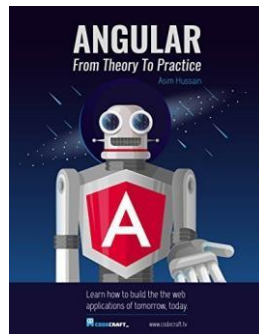
Le guide des bonnes pratiques Angular

- <https://angular.io/guide/styleguide>

Le site officiel d'Angular CLI

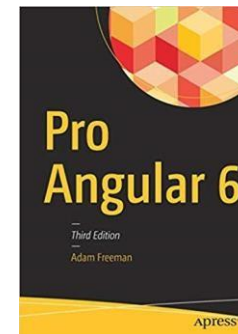
- <https://cli.angular.io/>

Un livre excellent (et gratuit !)



Angular : From Theory To Practice
Asim Hussain
<https://codecraft.tv/>

Un autre bon livre



Pro Angular 6
Adam Freeman

