# Développement JavaScript : rappels

# Bonnes pratiques Ecmascript 5

Javascript est un langage qui évolue en permanence, ceci dit, dans sa version Ecmascript 5, les bonnes pratiques restent globalement inchangées. Les voici en versions condensées, et bien entendu cette liste n'est en rien exhaustive.

- Eviter les variables globales
- Déclarer les variables locales à l'aide du mot clé "var"
- Le mode strict doit être utilisé
- Eviter la fonction eval()
- Organiser son code en module
- Une expression doit toujours se terminer par un point-virgule
- Le code doit être découpé en plusieurs fichiers

... Et bien d'autres que vous pourrez retrouver par exemple à l'adresse suivante : https://maxlab.fr/javascript/bonnes-pratiques-javascript-pour-lentreprise

# ES7/ES6/ES2015, présentation générale.

ECMAScript est un ensemble de normes concernant les langages de programmation de type script et standardisées par Ecma International dans le cadre de la spécification ECMA-262. Il s'agit donc d'un standard, dont les spécifications sont mises en œuvre dans différents langages de script, comme JavaScript ou ActionScript, ainsi qu'en C++ (norme 2011). C'est un langage de programmation orienté prototype.

Pour le langage Javascript, la norme ECMASCRIPT 5 est aujourd'hui la plus répandue, toutefois certains navigateurs, comme Chrome (Google Inc.), mettent déjà à disposition des développeurs une implémentation de Javascript respectant la norme ECMA 6 (ES2015).

Nommée ES2015, la dernière version d'ECMAScript a été publiée en juin 20156. Son support par les navigateurs évolue progressivement, mais il est possible d'utiliser un transcompilateur (tel que Babel.js8) vers ES5 pour développer dès aujourd'hui en ES6.

Première version publiée après le changement du processus de normalisation, l'ES7, ou ES2016, apporte peu de changements au langage. Le nouveau processus prévoit en effet de publier chaque année une nouvelle norme avec les ajouts qui ont eu lieu dans l'intervalle, afin d'éviter de publier des changements énormes comme ce fut le cas en 20159.

# ECMASCRIPT 6 et 7, les nouveautés

Les normes ECMASCRIPT 6 et 7 arrivent avec leurs lots de nouvelles fonctionnalités notamment les suivants ( mais pas que ):

- Les mot-clés let et const
- Les templates et chaînes de caractères
- Les paramètres par défaut
- Les promesses
- Les fonctions lambda ( arrow functions )
- Les classes et l'héritage

Nous allons détailler un peu plus chacune de ces fonctionnalités.

## Les mot-clés let et const

Le mot clé **let** permet de déclarer une variable limitée à la portée d'un bloc, c'est-à-dire qu'elle ne peut être utilisée que dans le bloc où elle a été déclarée, ce qui n'est pas le cas avec var.

Le mot-clé **const** sert à définir une valeur immuable, accessible uniquement en lecture seule.

```
function whoIsTheSithLord() {
    // la valeur ne peut plus être modifiée
    const MOVIE = "STAR WARS";

    if (true) {
        // existe en dehors du bloc conditionnel
        var sith_lord = "Palpatine";
        //existe uniquement dans le bloc conditionnel
        let padawan = "Dark Vador";
    }

    console.log(sith_lord); // displays Palpatine
    console.log(padawan); // Uncaught Reference Error
    conole.log(MOVIE);
}

whoIsTheSithLord();
```

# Templates Strings

Les "templates strings" permettent d'insérer plus facilement des valeurs de variables au sein d'une chaîne de caractère, ces dernières peuvent également être définies sur plusieurs lignes.

```
let jedi = {surname:"Obiwan", name:"Kenobi"};
let msg = `${jedi.name} ${jedi.surname} is the Jedi Master`;
console.log(msg);
```

## Paramètres par défaut

Il est dorénavant possible de donner une valeur par défaut aux paramètres, cela évite les lignes de codes superflues et facilite la lecture du code.

```
function whoIsTheSithLord(p_name='Dark Vador')
{
    console.log("The Sith lord is: ", p_name);
}

whoIsTheSithLord();
```

## Les fonctions fléchées/lambdas

Il s'agit d'une nouvelle façon d'écrire les fonctions anonymes, plus compacte et permettant de lier définitivement le contexte d'éxécution de la fonction (this).

```
function launchFunc(p_function) {
    p_function();
}

// ecma 5 style anonymous function
launchFunc(
    function () {
        console.log("I am The Sith Lord")
    }
);

// ecma6 arrow function
launchFunc(
    () => { console.log("I am The Sith Lord") }
);
```

# Les promesses

L'objet Promise (pour « promesse ») est utilisé pour réaliser des traitements de façon asynchrone. Une promesse représente une valeur qui peut être disponible maintenant, dans le futur voire jamais.

L'interface Promise représente un intermédiaire (proxy) vers une valeur qui n'est pas nécessairement connue au moment de sa création. Cela permet d'associer des gestionnaires au succès éventuel d'une action asynchrone et à la raison d'une erreur. Ainsi, des méthodes asynchrones renvoient des valeurs comme les méthodes synchrones, la seule différence est que la valeur retournée par la méthode asynchrone est une promesse (d'avoir une valeur plus tard).

```
new Promise((resolve, reject) => {
    const xhr = new XmlHttpRequest();
    xhr.open("GET", url);
    xhr.onload = () => resolve(xhr.responseText);
    xhr.onerror = () => reject(xhr.statusText);
});

function resolve(p_msg){
    console.log("resolve: ", p_msg);
}

function reject(p_msg){
    console.log("reject: ", p_msg);
}
```

## Classes et Héritages

Ecma 6 introduit des mécanismes Orienté Objet par classe grâce aux mots-clés **class** et **extends**, malheureusement ces derniers n'introduisent aucun changement dans la manière dont les objets sont réellements agencés et conçus au sein de la VM Javascript, il s'agit uniquement de sucre syntaxique.

```
class Personnage {
    constructor() {
        this.name = "";
    }

    sayMyName() {
        console.log(this.name);
    }
}

class JediKnight extends Personnage {
    constructor(p_name) {
        this.power = 250;
        this.name = p_name;
    }
}

let obiwan = new JediKnight("Obiwan Kenobi");
obiwan.sayMyName();
```

# Les Observables

Un Observable est un producteur de données qui peut être observé. On le mettra sous observation avec la méthode **subscribe** et cette observation sera exécutée par un objet de type Observer. Les Observables se sont imposés dans le monde de la programmation réactive ( reactive programming ) par le biais de la librairie Reactive X, abrégé RxJs pour sa version Javascript. Un Observable peut également produire des données de façon asynchrone et il est possible de fusionner plusieurs Observables entre eux.

Il faut bien noter que les Observables sont souvent associés aux promesses, voire décrits comme des super-promesses, ce qui n'est pas tout à fait exact, un article entier a d'ailleurs été rédigé à l'attention des développeurs à l'adresse suivante: https://medium.com/@benlesh/learning-observable-by-building-observable-d5da57405d87 et explique un peu plus en détail ce que sont les Observables.

```
Rx.Observable.create(function(observer) {
    setTimeout(() => observer.next("valeur A"), 700);
    setTimeout(() => observer.next("valeur B"), 400);
})
 .subscribe(e => console.log(e));

 // affiche "valeur A", puis "valeur B"  dans la console
```

# Reactive Extension For Javascript (RxJS)

Reactive Extensions (Rx) est une bilbiothèque permettant de développer des applications "orientées évènements" et utilisant des requêtes asynchrones en liant entre elles des séquences d'objets de type Observables.

Les séquences de données peuvent prendre plusieurs formes, telle qu'un flux de données issue d'un webservice, des requêtes vers des webservices, des notifications systèmes, ou bien encore une séquence d'évènements utilisateurs etc...

Rx represente toutes ces données sous la forme d'une séquence d'Observables, une application peut souscrire à ce flux de données et être notifiée chaque fois qu'une donnée est ajoutée au flux.

La comparaison avec les promesses est alors tentante et il est possible de se faire une idée plus précise de ce que sont les Observables et dans quels domaine les préférer par rapport aux Promises à l'adresse suivante: https://xgrommx.github.io/rx-book/index.html

# Les outils

L'une des choses les plus difficiles à déterminer lorsque l'on commence un développement en Angular est la stack complète de développement. Fort heureusement, une multitude d'outils tous plus performants les uns que les autres sont venus renforcer l'écocsystème Javascript ces derniers temps, et ce, grâce à **node.js** et son gestionnaire de dépendances spécialisé front-end **npm**. L'on peut donc compter au rang des outils quasiment indispensables:

- Node.Js et NPM (node package manager)
- Typescript ( langage poussé par Google pour le développement Angular )
- Un bon éditeur Typescript (Micorsoft Visual Code)
- Un outil de scaffolding ( génération de structure de projets et codes ) comme @angular/cli
- Un module loader, comme Webpack
- Un bon navigateur et/ou environnement d'éxécution ( Google Chrome, Ionic ... )

# Le langage Typescript

TypeScript est un langage de programmation libre et open source développé par Microsoft qui a pour but d'améliorer et de sécuriser la production de code JavaScript. C'est un sur-ensemble de JavaScript (c'est-à-dire que tout code JavaScript correct peut être utilisé avec TypeScript). Le code TypeScript est transcompilé en JavaScript, pouvant ainsi être interprété par n'importe quel navigateur web ou moteur JavaScript.

TypeScript permet un typage statique optionnel des variables et des fonctions, la création de classes et d'interfaces, l'import de modules, tout en conservant l'approche non-contraignante de JavaScript. Il supporte la spécification ECMAScript 6.

- Typage statique
- Typage générique
- Interfaces
- Classe, classe abstraite, expressions de classe
- Modules
- Mixin
- Enumérations
- Paramètres optionnels
- Symboles

... Et bien d'autres, d'ailleurs depuis la version 1.6, le format JSX est supporté.

# ES6 Module Pattern

La norme ES6 prévoit de founir au développeur la possibilité de regrouper et empaqueter des objets, classes, fonctionnalités et valeurs au sein de **modules**, dans la continuité des pattern modulaires déjà bien ancrés au sein de l'écosystème Javascript (AMD par exemple).

L'instruction export est utilisée pour permettre d'exporter des fonctions et objets ou des valeurs primitives à partir d'un fichier (ou module) donné. Ces fonctions et objets peuvent ensuite être utilisées dans d'autres fichiers grâce à import.

Cette fonctionnalité n'est pas encore implémentée nativement par la plupart des navigateurs, en revanche, elle est gérée par de nombreux transpileurs comme **Traceur, Babel ou Rollup.**

```
function whoIsTheSithLord() {
    console.log("Palpatine is the Sith Lord !");
}

// on exporte une fonction déclarée plus haut
export { whoIsTheSithLord };

// on exporte une constante
export const DARK_VADOR = "Dark Vador";
```

# ES7 Async functions

La déclaration async function définit une fonction asynchrone qui renvoie un objet AsyncFunction. On peut également définir des fonctions asynchrones grâce au constructeur AsyncFunction et via une expression de fonction asynchrone.

Lorsqu'une fonction asynchrone est appelée, elle renvoie une promesse. Lorsque la fonction asynchrone renvoie une valeur, la promesse est résolue avec la valeur renvoyée. Lorsque la fonction asynchrone lève une exception, la promesse est rompue avec la valeur de l'exception.

Une fonction asynchrone peut contenir une expression await qui permet d'interrompre l'exécution de la fonction asynchrone en attendant la résolution d'une promesse passée à l'expression. L'exécution de la fonction asynchrone reprend lorsque la promesse est résolue.

L'objectif des fonctions asynchrones avec await est de simplifier le comportement des promesses lors d'opérations synchrones et d'opérer sur des groupes de promesses. Si les promesses sont en quelque sorte des callbacks organisés, async/await permet de combiner les générateurs et les promesses. On peut trouver des exemples à l'adresse suivante: https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Instructions/async_function

# Module Loader & Webpack

Un module loader est un outil qui fournit une solution permettant de gérer des modules Javascript, en respectant la norme des ES6 Modules. Il s'agit en règle générale d'une suite logicielle composée d'un utilitaire en ligne de commande et de fichiers à configurer.

L'un des plus célèbres Module Loader de l'écosystème Javascript est Webpack, ce dernier permet, comme son nom l'indique, de créer des paquetages à partir de fichier Javascript ( mais pas que ) et de les mettre à disposition sous forme de modules au sein d'une application front-end.

Pour plus de renseignements sur le fonctionnement de Webpack, rendez-vous sur le site officiel du produit à l'adresse suivante: https://webpack.github.io/docs/

# Définition de composants

# Comprendre les Web Components. (standard, concepts, shadow DOM, scoped CSS...).

Lorsque l'on développe une application, plus on écrit de code, plus celui-ci a besoin d'être modulaire, réutilisable et encapsulé et pour cela, côté frond, les **Web Components** viennent à notre rescousse. Il s'agit avant tout d'une spécification du W3C en cours de rédaction mise en avant par les acteurs du Web (Les deux éditeurs de la spécification travaillent chez Google). Mais en vrai ça consiste en quoi ? En vrai, les Web Components c'est :

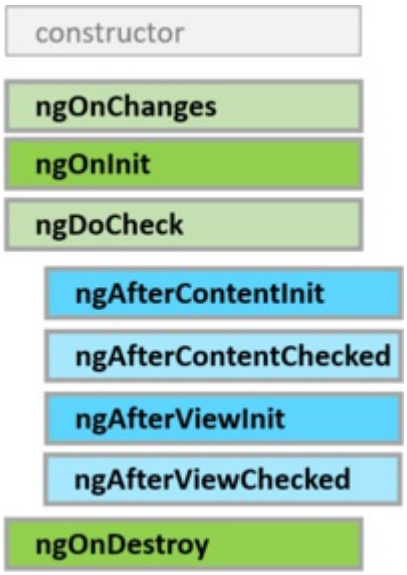
- Templates
- Éléments customs
- Shadow DOM
- Encapsulation de style
- Observers (pour le modèle ainsi que pour le DOM)
- Variables CSS

Pour en apprendre plus sur cette spécification, je vous invite à vous rendre à l'adresse suivante: https://developer.mozilla.org/fr/docs/Web/Web_Components. Il est important de noter que le système de composants d'Angular respecte au mieux les spécifications des Web Components.

# Cycle de vie dans l'application

Les instances de Directives et Composants instances disposent d'un cycle de vie au sein d'Angular à mesure que le framework crée, met à jour, et détruit lesdites instances. Les dévelopeurs peuvent attraper et intervenir sur divers moments de ce cyle de vie en implémentant des méthodes d'interfaces que l'on appelle des **Hooks**.

L'ensemble des **Hooks** disponibles pour un composant est listé ci-dessous, pour savoir à quoi chacun de ces hook correspond, rendez-vous à l'adresse suivante: https://angular.io/guide/lifecycle-hooks



# Angular Compiler : Change Detection

Pour ceux qui ont connu Angular JS, l'une des expériences les plus douloureuses fût l'expérimentation de la boucle de rendu, ou plus exactement l'expérimentation de la lenteur et de la surconsommation de ressources de la boucle de rendu. En effet, en plus d'offrir une API un peu bancale, cette boucle de rendu avait pour mauvaise habitude de consommer un peu (trop) de ressources au niveau du CPU, aujourd'hui Angular introduit un système différent, basé sur la librairie zone.js **sans déficit en termes de fonctionnalités**.

En vérité Angular utilise sa propre version un peu remaniée de la librairie zone.js que l'on peut appeller NgZone, cette librairie a pour but premier de détecter les changements opérés côté Model et côté View afin de permettre le relancement d'une boucle de rendu et une synchronisation simplifiée des données entre le Model et la View.

Chaque fois qu'une action est effectuée par le développeur, comme le chargement d'une ressource externe par exemple à l'aide d'un mécanisme asynchrone, ou le fait de taper dans un champ texte par exemple, NgZone va détecter ces opérations et relancer un cycle de rendu / synchronisation.

Pour un complément d'information sur les dessous programmatiques de cette librairie NgZone, je vous invite à vous rendre à l'adresse suivante: https://auth0.com/blog/understanding-angular-2-change-detection/

# Template Syntax

Angular intègre un système interne de templating, les templates sont des modèles HTML reliés à des données exposées via des instances de composants. Ce système de templating vient, comme tout bon système de templating, avec sa syntaxe propre.

On utilise la syntaxe suivante **{{...}}** pour faire référence à une propriété publique d'un composant, dans le but d'exposer sa valeur au sein de la boucle de rendu par le biais d'une **interpolation**. Cette syntaxe nous permet de lier les données dans un sens seulement, on parle de **one-way databinding**

On a également accès à des Templates Statements, qui sont en fait des moyens de faire de la programmation évenementielle au sein des templates, et donc de répondre à des évenements utilisateurs.

Angular nous donne également au data-binding bidirectionnel ou **two-ways databinding** à l'aide de la syntaxe [(...)] portant le nom de **banana in the box**.

Plus d'infos à l'adresse suivante: https://angular.io/guide/template-syntax#attribute-class-and-style-bindings

# Le symbole *

Pour les directives structurelles, le symbole * est utilisé, il s'agit de sucre syntaxique, càd une facilité programmatique accéder à quelque chose d'un peu plus compliqué en arrière plan. Angular utilise ce symbole et le décompose en 2 étapes.

Premièrement, dans le cas de la directive structurelle ( ou comportementale ) *ngIf, la directive va être traduite sous la forme d'un attribut de template: template='ngIf (...conditions...)'

Puis dans un second temps, cet attribut de template va ête traduit en élement <ng-template> </ng-template> qui va encapsuler l'élément hôte de la directive.

```
<div *ngIf="jedi">{{jedi.name}}<</div>
//....

<div template="ngIf jedi">{{jedi.name}}</div>

//.....

<ng-template [ngIf]="hero">
  <div>{{hero.name}}</div>
</ng-template>
```

## Les variables locales de template

Angular permet la création et l'utilisation de variables locales au sein des templates. Ces variables n'existent que le temps de leur utilisation ou le temps de vie complet du template. L'exemple le plus probant étant les variables que l'on déclare systématiquement lors de l'utilisation de la directive structurelle *ngFor. En effet l'on déclare systématiquement une variable temporaire chargée de stocker, à mesure que l'itération sur une liste évolue, la valeur de cette liste à un instant t.

```
// ici la variable warrior a une durée de vie limitée,
// qui est celle de la directive structurelle *ngFor,
// en de dehors de l'élement hôte ( la div ), la variable warrior
// sera inaccessible
<div *ngFor="let warrior of jedis">{{warrior.name}}</div>
```

## Les classes de composants

Les classes de composants sont des classes affûblées du décorateur @Component, disponible de base au sein d'Angular. Ce décorateur permet au développeur de définir des métadonnées qui donneront des informations à Angular sur comment instancier et utiliser le composant, mais également sur les différentes classes qu'il peut utiliser en tant que fournisseur de services (providers), le sélecteur HTML auquel il répond (selector), et surtout son template ou le chemin vers celui-ci.

```
// exemple d'une classe de composant JediComponent
@Component(
   {
      selector: 'jedi',
      template: 'Hello Jedi {{rank}} !'
   }
)
class JediComponent {
   rank: string = 'Master';
}
```

# EventEmitter

Angular propose la mise en place et la création d'évenements personnalisées, ainsi que leur diffusion à l'aide de la classe EventEmitter, utilisée surtout par les directives ( un composant est une directive jouissant d'un lien avec un template ) afin de gérer une programmation évenementielle customisée.

```
@Component({
  selector: 'jedi',
  template: `<button (click)="toggle()">Change Side Force</button>`})
export class JediComponent {

  isOnTheDarkSide: boolean = true;
  @Output() dark: EventEmitter<any> = new EventEmitter();
  @Output() light: EventEmitter<any> = new EventEmitter();

  toggle() {

    this.isOnTheDarkSide = !this.isOnTheDarkSide;

    if (this.isOnTheDarkSide) {
      this.open.emit(null);
    }
    else {
      this.close.emit(null);
    }
  }
}

//....

<jedi (dark)="darkHandler($event)" (light)="lightHandler"></jedi>
```

# Transclusion & Content Projection

# Angular et la Projection (Transclusion)

Dans le composant ci-dessous, nous cherchons à injecter du contenu par un biais autre que le décorateur @Input(), ce dernier ne se prêtant pas facilement à l'injection de HTML ou d'un autre composant. Pour ce faire, nous allons faire appel au mécanisme de la **Projection**, anciennement nommé **Transclusion** en AngularJS.

La balise <ng-content></ng-content> à pour vocation d'accueillir du contenu défini à l'intérieur de l'élement HTML correspondant au sélecteur du composant.

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

@Component({
  selector: 'jedi',
  template: `
    <div style="border: 1px solid blue; padding: 1rem;">
      <h4>Jedi Component</h4>
      <ng-content></ng-content>
    </div>`
})
export class JediComponent {}

// creates the component from the host component ...


<jedi>
  <p>I am the Jedi Master</p>
</jedi>
```

# Projection multiple

On peut également utiliser la projection multiple en ciblant à l'aide de l'attribut select:

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

@Component({
  selector: 'jedi',
  template: `
    <div>
      <ng-content select="jediname"></ng-content>
      <ng-content select="jedisurname"></ng-content>
    </div>`
})
export class JediComponent {}


<jedi>
    <jediname>
        <h1>Kenobi</h1>
    </jediname>
    <jedisurname>
        <h2>Obiwan</h2>
    </jedisurname>
</jedi>
```

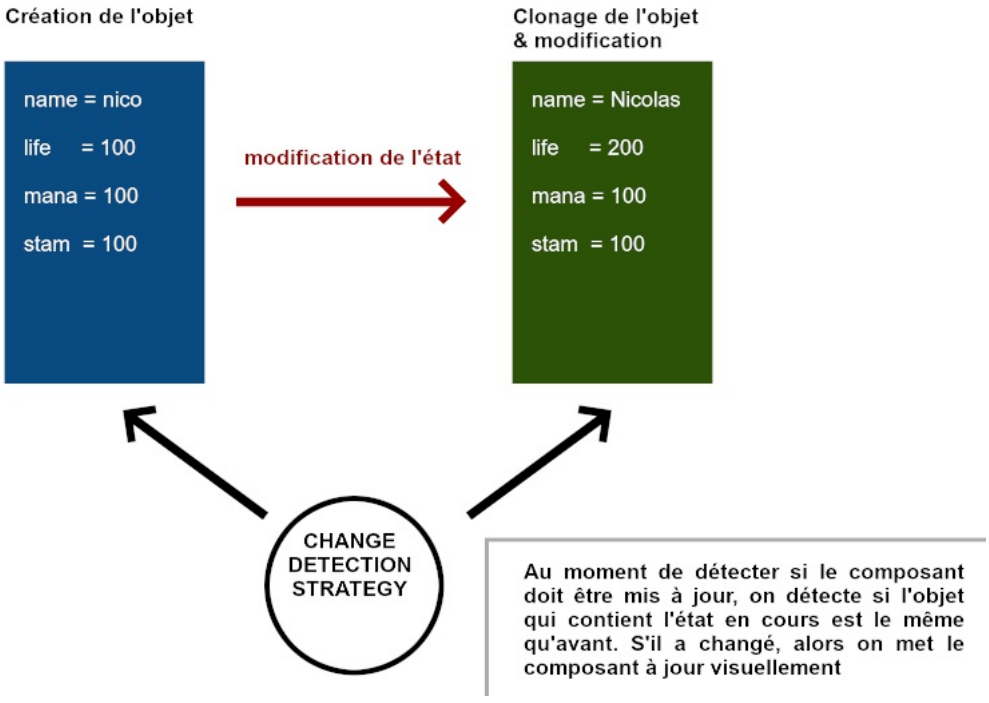
# Zones & change detection strategy

# Principes des zones

Une zone est un contexte d'exécution qui persiste dans les tâches asynchrones. Vous pouvez le considérer comme une espèce de 'boîte noire', une sorte de stockage local qui concerne avant tout une partie de votre application Angular. L'appellation de zone provient directement de la librairie zone.js, incluse directement au sein d'Angular. Cette librairie se charge de détecter des changements d'états en se reposant sur le pattern de l'immutabilité des objets.

Le pattern de l'immutabilité des objets repose sur le fait que l'état d'un composant est stocké sur un objet dont les valeurs sont figées. À chaque changement d'état, plutôt que de changer les valeurs des propriétés de cet objet (ce qui est de toutes façons impossible), on clone l'objet, puis on change les valeurs des propriétés du clone en question pour enfin le sceller, comme son prédecesseur.

Ce mécanisme permet de détecter facilement un changement d'état, car au lieu de vérifier l'état de chacune des propriétés (et des éventuelles sous propriétés de ces propriétés...), on vérifie seulement si la référence de l'objet a changé. Au sein d'Angular, cette mécanique est plutôt transparente, mais on peut la customiser à l'aide du Change Detection Strategy.



# Change Detection

Angular se repose sur deux mécanismes pour détecter les changements de modèle de données et actualiser les vues associées aux divers composants qui composent le programme:

Le premier est un **"Change Detector Tree"**, un arbre symbolisant l'ensemble des objets attachés à chacun des composants et dont le rôle est de détecter l'ensemble des changements opérés sur le composant associé.

Le second est **le principe d'immutabilité**, à chaque fois que l'on souhaite changer une valeur d'une des propriétés d'un des composants, Angular fait en sorte de créer un nouvel objet, qui représente les données associées à ce composant. Cet objet va se voir affecter la nouvelle valeur, ainsi lorsque qu'Angular va lancer son cycle de détection des changements, en partant **haut vers le bas** de l'arbre, le **Change Detector associé ne devra comparer que deux références d'objets, au lieu d'opérer une vérification profonde.**

Il est possible de court-circuiter le cycle de détection à un niveau précis, et de gérer soi-même la mécanique de rafraîchissement des données à l'aide de la stratégie **onPush**. Pour plus de détails sur le fonctionnement du cycle de détection et sur le contrôle total de celui-ci, vous pouvez vous rendre à l'adresse suivante: https://blog.thoughtram.io/angular/2016/02/22/angular-2-change-detection-explained.html#reducing-the-number-of-checks

Il est possible de faire d'opérer une stratégie de rendu différente pour un composant Angular. Prenons le cas du composant suivant:

```
@Component({
  template: `
    <h2>{{data.name}}</h2>
    <span>{{data.surname}}</span>
  `,
  changeDetection: ChangeDetectionStrategy.OnPush
})
class WarriorComponent {
  @Input() data;
}
```

Ce composant ne dispose d'aucune méthode interne lui permettant de changer ses propres valeurs, il attend qu'on les lui fournisse à l'aide du décorateur @Input(). En lui changeant sa stratégie et en la passant en mode OnPush, on fait en sorte que le rendu ne soit mis à jour que lorsque les données provenant de l'extérieur ( ou d'autres mécanismes déclenchant le rarfraîchissement ). Là l'exemple est simple, mais imaginez que ce composant dispose lui-même de sous composants etc ... Le gain de performances peut être énorme.

## Contrôle total du rendu

Admettons le cas ou l'on souhaite passer notre composant principal en strategie de rendu **OnPush**, aucun élement ne serait alors mis à jour à moins de le demander explicitement à Angular. Il est tout à fait possible d'avoir un contrôle total du rendu en le demandant directement à l'objet de type ChangeDetector associé à chacun des composants.

```
@Component({
  template: `
    <h2>{{data.name}}</h2>
    <span>{{data.surname}}</span>
  `,
  changeDetection: ChangeDetectionStrategy.OnPush
})
class WarriorComponent {

  @Input() data;

  constructor(private cd: ChangeDetectorRef) {}

  public changeName()
  {
     this.data.name = "Obiwan Kenobi";
     // en appellant la méthode markForCheck, Angular va réévaluer les valeurs associées
     // au composant et relancer un rendu de ce dernier
     this.cd.markForCheck();
  }
}
```

# Injection de dépendances & bonnes pratiques

# Principes de l'injection de dépendances

Le principe de l'injection de dépendance est assez simple, dans le cadre d'un développement MVC classique, les fichiers declasses embarquent des références directes aux classes tierces qu'ils utilisent, ce qui le rend le développement lourd à maintenir et empêche le développeur de réutiliser facilement son code. On dit que l'on fait du "monolithic development",dans le sens ou les couches logicielles sont constituées souvent de gros blocs de code.

L'injection de dépendances quand à elle, permet de déléguer à un objet tiers, la création des objets dont nos modules vont avoir besoin. Cet objet est appelée l'injecteur de dépendance, il va donc se charger de créer les objets demandés et les envoyer aux objets qui en ont besoin. Ainsi ces derniers pourront utiliser des dépendances sans avoir besoin d'avoirune référence directe à ces dernières.

De plus, cela permet de tester différentes versions de l'application facilement, en effet, puisque les composants ne créentplus eux-mêmes les objets dont ils ont besoin ( souvent des classes gérant des échanges de données par le biais de services), il est possible de demander à l'injecteur de tester une version de l'application avec tel dépendance injectée, puis avec une autre, et ce, en changeant uniquement les élements de configuration. Ce pattern s'intègre parfaitement dans le cadre d'une application développée avec des tests d'intégration continue.

La classe suivante est un service, le décorateur @Injectable() en fait un service qui peut être injecté, mais pas n'importe où ! Il vous faut configurer un injecteur de dépendance à l'aide d'un provider pour ce service. Comme dit plus haut, l'injecteur va se charger les instances de notre service. Avec Angular, on crée rarement nos propres injecteurs car le framework le fait pour nous dès le démarrage de l'application en créant le root injector. Le provider quand à lui, va expliquer à notre injecteur comment créer le service. On doit configurer un provider avant qu'un injecteur puisse créer le service. Le provider peut être le service lui-même, en ce cas l'injecteur aura juste à utiliser le mot-clé 'new'. On peut configurer à l'aide de méta-données à 3 endroits:

- Au sein du décorateur @Injectable(), à l'aide de l'option providerIn
- Au sein du décorateur @NgModule() à l'aide de l'option provider
- Au sein du décorateur @Component() à l'aide de l'option provider.

```
import { Injectable } from '@angular/core';
import { HEROES } from './mock-heroes';

@Injectable({
  // we declare that this service should be created
  // by the root application injector.
  providedIn: 'root',
})
export class HeroService {
  getHeroes() { return HEROES; }
}
```

# Gestion des modules: Bonnes pratiques

Un NgModule est une classe à laquelle on a ajouté le decorateur @NgModule, ce dernier prend en charge un objet décrivant des méta-données qui permettent à Angular de savoir comment compiler et démarrer le module. Au sein des méta-données sont également spécifiés les composants, directives, pipes, services etc ... Embarqués par le module et mis à disposition des autres modules.

Chaque Application Angular dispose au moins d'un module, le module d'application, qu'il faut configurer et lancer afin de démarrer l'application. Il est recommandé, lorsque des composants et services doivent être partagés entre plusieurs modules, de regrouper ces applications dans un module partagé (SharedModule) que les autres spécifieront comme étant une dépendance

Les modules peuvent être pré-compilés à l'aide du compilateur ngc (A.O.T compilation) ou tout simplement compilés à l'utilisation (J.I.T compilation), ils peuvent être tous chargés au démarrage de l'application, ou à la demande par le Router d'Angular.

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

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

## Module personnalisé: BookModule

Le module ci-dessous embarque des fonctionnalités liés à un objet de type livre. Il met à disposition du développeur un objet BookComponent, ainsi qu'un service nous permettant de manipuler les données liées aux livres.

```
import { NgModule }            from '@angular/core';
import {BookComponent }        from './book.component';
import {BookService}           from './book.service';

@NgModule({
  declarations:[BookComponent],
  providers:[BookService]
})
export class BookModule {}


import {NgModule } from '@angular/core';
import {BookModule} from './book.module';
import { AppComponent } from './app.component';

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

# RouterModule

Angular propose un système de **routing**, çàd qu'il met en relation ce que l'on appelle des urls profondes ( ou deeplinks ) avec des instances de composants et des templates qu'Angular va se charger de créer à la place du développeur.

Le RouterModule peut être importé plusieurs fois, tant que le router traite avec des ressources partagées et globales, nous ne pouvons avoir qu'une seule instance du router service active à la fois. C'est d'ailleurs pour cela qu'il existe 2 façons de créer le module: RouterModule.forRoot and RouterModule.forChild.

La méthode forRoot qui crée un module contenant les directives, les routes et le RouterService lui-même.

La méthode forChild fait exactement la même à l'exclusion du RouterService.

```
@NgModule({
  imports:      [
    BrowserModule,
    FormsModule,
    HttpModule,
    RouterModule.forRoot([{path:"home", component: AppComponent} ])
  ],
  declarations: [ AppComponent ],
  bootstrap:    [ AppComponent ],
  providers:[]
})
export class AppModule { }
```

# Injection de dépendances hiérarchique

Les services sont des singletons au de la portée de l'injecteur, ce qui veut dire qu'il y a tout au plus une seule instance d'un même service au sein d'un injecteur donné.

Il n'y a qu'un seul 'root injector' pour une application. Si cet injecteur fournit le service UserService, alors il y aura une seule instance de UserService partagée à travers toute l'application, sauf si vous configurez un autre provider avec un injecteur enfant.

Angular dispose d'un système d'injection hiérarchique, ce qui signifie que les injecteurs imbriqués peuvent créer leurs propres instances de service. Angular crée régulièrement des injecteurs imbriqués. Chaque fois qu'Angular crée une nouvelle instance d'un composant dont les providers sont spécifiés dans @Component (), il crée également un nouvel injecteur enfant pour cette instance. De même, lorsqu'un nouveau NgModule est chargé à l'aide du 'lazy loading' au moment de l'exécution, Angular crée un injecteur pour lui avec ses propres providers.

Les injecteurs des modules enfants et des composants sont indépendants les uns des autres et créent leurs propres instances séparées des services concernés. Quand Angular détruit une instance de NgModule ou un composant, il détruit également l'injecteur associé et toutes les instances de services créées par cet injecteur.

Grâce à l'héritage des injecteurs, on peut toujours injecter des services partagés à l'échelle de l'application au sein de ces composants. L'injecteur d'un composant spécifique est l'enfant de l'injecteur de son composant parent et hérite de toute la chaîne d'injecteurs jusqu'au 'root injector'. Angular peut donc injecter n'importe quel service de cette lignée.

```
// HeroService délivré par l'injecteur racine, instance partagée à travers toute l'application
import { Injectable } from '@angular/core';

@Injectable({providedIn: 'root'})
export class HeroService { constructor(){} }

// HeroService délivré par un injecteur de niveau composant, l'instance de base
// de HeroService, pour ce composant ne sera pas la même.
import { Component } from '@angular/core';
import { HeroService } from './hero.service';

@Component({
  selector: 'app-heroes',
  providers: [ HeroService ],
  template: `<p>HeroesComponent</p>`
})
export class HeroesComponent { }
```

# Injection token

Quand vous configurez un injecteur à l'aide d'un provider, vous associez en fait le provider avec un 'DI token', une sorte d'identifiant unique. L'injecteur maintient en son sein une carte d'association token-provider utilisée à chaque fois que l'on demande une dépendance, le token est utilisé ici en tant que clé. Dans la plupart des exemples, la valeur est une instance du service alors que la classe du service est utilisée comme clé. Mais on peut créer ses propres DI tokens afin d'injecter tout et n'importe quoi, et surtout des valeurs qui ne soient pas des objets.

```
// On crée un DI Token
import { InjectionToken } from '@angular/core';
export const MY_TOKEN = new InjectionToken<string>('MyToken');

// Ensuite, on associe ce DI Token à une valeur à injecter au niveau provider
@NgModule({
providers: [
   { provide: MY_TOKEN, useValue: 'Hello world' }
]
})
export class AppModule { }

// Puis ensuite, on peut injecter cette valeur au sein de notre composant
@Component({
  selector: 'my-component',
  template: '<h1>{{ value }}</h1>'
})
export class MyComponent {
  constructor(@Inject(MY_TOKEN) public value: string) { }
}
```

# Classifications des composants applicatifs

# RouterModule

Angular propose un système de **routing**, çàd qu'il met en relation ce que l'on appelle des urls profondes ( ou deeplinks ) avec des instances de composants et des templates qu'Angular va se charger de créer à la place du développeur.

Le RouterModule peut être importé plusieurs fois, tant que le router traite avec des ressources partagées et globales, nous ne pouvons avoir qu'une seule instance du router service active à la fois. C'est d'ailleurs pour cela qu'il existe 2 façons de créer le module: RouterModule.forRoot and RouterModule.forChild.

La méthode forRoot qui crée un module contenant les directives, les routes et le RouterService lui-même.

La méthode forChild fait exactement la même à l'exclusion du RouterService.

```
@NgModule({
  imports:      [
    BrowserModule,
    FormsModule,
    HttpModule,
    RouterModule.forRoot([{path:"home", component: AppComponent} ])
  ],
  declarations: [ AppComponent ],
  bootstrap:    [ AppComponent ],
  providers:[]
})
export class AppModule { }
```

# Guards

Angular permet de protéger certaines routes à l'aide du système **Guards**. Protéger certaines routes est très utile par exemple dans le cas où on souhaite restreindre l'accès à certaines zones du site, où même demander une confirmation pour quitter une zone du site. Il existe 4 types de Guard différents:

- **CanActivate** - Décide si une route peut être activée ou pas
- **CanActivateChild** - Décide si l'enfant d'une route peut être activée ou non
- **CanDeactivate** - Décide si une route peut être désactivée
- **CanLoad** - Décide si un module peut être chargé de manière 'fainénante' (à la demande uniquement)

On peut retrouver quelques exemples d'implémentations des Guards à l'adresse suivante: https://angular.io/guide/router#guards.

# Pipes

Un Pipe prend des données en entrée et les transforment afin de délivrer le contenu de sortie que l'on souhaite obtenir. Il s'agit aussi d'un décorateur qui s'applique à une classe, cette classe permet de définir le code qui va gérer cette transformation de données et donc de construire des Pipes customisés. On peut enchaîner les Pipes les uns à la suite des autres, et Angular procure un certains nombre de Pipes précodés.

```
/*
* Fournit un Pipe qui prend en charge un tableau de données numériques
* et retourne une moyenne de ces données numériques
*/

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

@Pipe({ name: 'average' })
export class AveragePipe implements PipeTransform {
   transform(values: number[] ): number {
      let avg:number = 0;
      let i:number = values.length;

      while( --i > -1 )
      {
         avg += values[i];
      }

      return avg / values.length;
   }
}
```

# Principes de l'injection de dépendances

Le principe de l'injection de dépendance est assez simple, dans le cadre d'un développement MVC classique, les fichiers declasses embarquent des références directes aux classes tierces qu'ils utilisent, ce qui le rend le développement lourd à maintenir et empêche le développeur de réutiliser facilement son code. On dit que l'on fait du "monolithic development",dans le sens ou les couches logicielles sont constituées souvent de gros blocs de code.

L'injection de dépendances quand à elle, permet de déléguer à un objet tiers, la création des objets dont nos modules vont avoir besoin. Cet objet est appelée l'injecteur de dépendance, il va donc se charger de créer les objets demandés et les envoyer aux objets qui en ont besoin. Ainsi ces derniers pourront utiliser des dépendances sans avoir besoin d'avoirune référence directe à ces dernières.

De plus, cela permet de tester différentes versions de l'application facilement, en effet, puisque les composants ne créentplus eux-mêmes les objets dont ils ont besoin ( souvent des classes gérant des échanges de données par le biais de services), il est possible de demander à l'injecteur de tester une version de l'application avec tel dépendance injectée, puis avec une autre, et ce, en changeant uniquement les élements de configuration. Ce pattern s'intègre parfaitement dans le cadre d'une application développée avec des tests d'intégration continue.

La classe suivante est un service, le décorateur @Injectable() en fait un service qui peut être injecté, mais pas n'importe où ! Il vous faut configurer un injecteur de dépendance à l'aide d'un provider pour ce service. Comme dit plus haut, l'injecteur va se charger les instances de notre service. Avec Angular, on crée rarement nos propres injecteurs car le framework le fait pour nous dès le démarrage de l'application en créant le root injector. Le provider quand à lui, va expliquer à notre injecteur comment créer le service. On doit configurer un provider avant qu'un injecteur puisse créer le service. Le provider peut être le service lui-même, en ce cas l'injecteur aura juste à utiliser le mot-clé 'new'. On peut configurer à l'aide de méta-données à 3 endroits:

- Au sein du décorateur @Injectable(), à l'aide de l'option providerIn
- Au sein du décorateur @NgModule() à l'aide de l'option provider
- Au sein du décorateur @Component() à l'aide de l'option provider.

```
import { Injectable } from '@angular/core';
import { HEROES } from './mock-heroes';

@Injectable({
  // we declare that this service should be created
  // by the root application injector.
  providedIn: 'root',
})
export class HeroService {
  getHeroes() { return HEROES; }
}
```

# Services Injectables

Au fur et à mesure que les applications se développent, vous avez de plus en plus le besoin que les composants accèdent à des données externes à l'application. Dans la philosophie Angular, la couche qui s'occupe de l'accès aux données distantes est dédiée aux services.

Et il se trouve que les services sont des objets qui sont injectables en Angular, çàd que l'on peut spécifier à la plateforme que l'on souhaite, en tant que composant, obtenir une instance de tel ou tel service et celui-ci va nous la fournir. Bien entendu pour cela il faut utiliser un décorateur, le décorateur @Injectable.

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

@Injectable()
export class HeroService {}
```

# Configuration de l'injecteur de dépendances

Le développeur n'a pas besoin de créer un injecteur de dépendances, Angular en fournit et en crée déjà un qui est accessible à travers toute l'application lors de la phase de démarre de l'application. On doit donc configurer l'injecteur de dépendances en spécifiant des **providers** qui crée les services dont l'application a besoin.

On peut soit définir des providers au sein d'un Module ou au sein de composants. Si vous spécifiez un service en tant que provider d'un module, tous les composants du même module se partageront la même instance du service, si vous le spécifiez uniquement en tant que provider d'un composant, alors il y aura une nouvelle instance à chaque fois.

```
import { Component }        from '@angular/core';
import { JediService }      from './jedi.service';

@Component({
  selector: 'jedi',
  providers: [JediService],
  template: "Jedi master !"
})
export class JediComponent {
    constructor(p_service:JediService){

    }
}
```

# Host, ContentChild, ViewChild, ContentChildren et ViewChildren

Les décorateurs @ContentChild, @ViewChild, @ContentChildren et @ViewChildren existent pour créer des liens entre l'élement hôte et les éléments qui sont imbriqués en son sein et qui font référence à des instances des composants.

Ces instances peuvent être imbriquées de deux façons, au sein du template du composant, ou elles peuvent être directement imbriquées au sein des données d'un composant hôte et ajoutées par projection de contenu sans avoir forcément besoin du template pour cela

Les ViewChildren font référence aux instances de composants crées à l'aide du template et les ContentChildren front eux référence aux données fournies par le composant lui-même par projection de contenu.

Les décorateurs nous servent donc à obtenir des références qui pointent vers les objets qui gèrent ces composants imbriqués. Pour plus de renseignement sur le sujet, vous pourrez trouver un excellent récapitulatif à l'adresse suivante: https://ng2.codecraft.tv/components/viewchildren-and-contentchildren/

# Gestion des formulaires, "Routing" et requête HTTP

# FormControl, FormGroup, Template Driven Form & Data Driven Forms

Angular nous procure une multitude d'outils nous permettant de créer et gérer des formulaires assez facilement et ce, toujours de façon efficace. La notion de FormControl y est omniprésente, un FormControl est tout simplement un objet amené à gérer l'un des éléments du formulaire, à traquer ses changements de valeur, la validation des données qui y sont associés etc...

Un FormGroup est tout simplement qui regroupe un ensemble d'objets de type FormControl, il est donc tout à fait possible à l'aide de ces derniers de mutualiser et réutiliser des logiques de validation et d'analyse des données gérées par les FormControls.

Vous pouvez construire des formulaires en écrivant des templates, à l'aide de la syntaxe template angular et des directives de formulaires et de certaines techniques décrites au sein de la documentation, c'est ce que l'on appelle des **templates driven forms**.

Il est également possible de décrire un modèle de données, organisé de façon arborescente, de créer les contrôles qui y sont associés au sein des composants eux-mêmes plutôt que de déléguer ce travail au moteur de template, et enfin de tout fournir à Angular afin qu'il puisse concrétiser cela sous la forme d'une interface utilisateur exploitable.

L'un des avantages à utiliser un data driven form (Ou Reactive Forms), est que l'on peut introduire beaucoup plus facilement à l'aide de la programmation, des mécanises de tests sur les validations des formulaires par exemple. Pour plus de renseignements sur ces 2 types de Forms, ainsi que sur la gestion des formulaires en règle générale au sein d'Angular, rendez-vous aux adresses suivantes: https://angular.io/guide/reactive-forms et https://angular.io/guide/forms

# Dynamic Input

Il est possible de créer ses formulaires autrement que par le biais du template seul ( template driven form ), en effet les objets de type FormGroup, FormControl, et FormBuilder peuvent nous aider à créer des formulaires dynamiques, pilotés côté modèle.

```
<form [formGroup]="myForm">
    <label>Prenom</label>
    </label><input type="text" formControlName="firstname">
    <fieldset formGroupName="details">
        <label>Age:</label>
        <input type="text" formControlName="age">
    </fieldset>
</form>

export class MyFormComponent implements OnInit {

    myForm: FormGroup;

    constructor(private formBuilder: FormBuilder) { }

    ngOnInit() {

        let data:Object = {
            firstname: '',
            details: this.formBuilder.group({ age: '', })
        };

        this.myForm = this.formBuilder.group(data);

    }
}
```

# Validators

Il est tout à fait possible d'ajouter à nos formulaires des validateurs (validators) et ce, directement au sein de notre modèle de données nous servant à le décrire:

```
export class MyFormComponent implements OnInit {

    myForm: FormGroup;

    constructor(private formBuilder: FormBuilder) { }

    ngOnInit() {

        let data:Object = {
            firstname: ['', Validators.required],
            details: this.formBuilder.group(
                {
                    age: ['', Validators.required],
                }
            )
        };

        this.myForm = this.formBuilder.group(data);

    }
}
```

## Validation asynchrone

Par défaut les validateurs crées au sein du template sont toujours asynchrones, mais il est également possible de créer, ses propres validateurs asynchrones au sein du modèle. Nota bene: Angular éxécute les validateurs asynchrones après les validateurs synchrones.

```
import { Directive, forwardRef } from "@angular/core";
import { NG_ASYNC_VALIDATORS, Validator, AbstractControl } from "@angular/forms";
import { Observable } from "rxjs";
@Directive({
    selector: '[isAdult][ngModel],[isAdult][formControl]',
    providers: [
        { provide: NG_VALIDATORS, useExisting: forwardRef(() => IsAdultValidator), multi:
true }
    ]
})
export class IsAdultValidator {

    constructor() { }

    validate(control: AbstractControl) {
        return new Observable(observer => {
            if ( parseInt(control.value >= 18) ) {
                observer.next(  { asyncInvalid: true }  );
            } else {
                observer.next(null);
            }
        });
    }
}
```

# RouterModule

Angular propose un système de **routing**, çàd qu'il met en relation ce que l'on appelle des urls profondes ( ou deeplinks ) avec des instances de composants et des templates qu'Angular va se charger de créer à la place du développeur.

Le RouterModule peut être importé plusieurs fois, tant que le router traite avec des ressources partagées et globales, nous ne pouvons avoir qu'une seule instance du router service active à la fois. C'est d'ailleurs pour cela qu'il existe 2 façons de créer le module: RouterModule.forRoot and RouterModule.forChild.

La méthode forRoot qui crée un module contenant les directives, les routes et le RouterService lui-même.

La méthode forChild fait exactement la même à l'exclusion du RouterService.

```
@NgModule({
  imports:      [
    BrowserModule,
    FormsModule,
    HttpModule,
    RouterModule.forRoot([{path:"home", component: AppComponent} ])
  ],
  declarations: [ AppComponent ],
  bootstrap:    [ AppComponent ],
  providers:[]
})
export class AppModule { }
```

# Paramètres de route et router-outlet

Afin de récupérer les paramètres des routes en Angular, il nous faut accéder à la route en cours à l'aide de l'objet ActivatedRoute et de l'objet RouteProvider. L'objet de type ActivatedRoute étant là pour symboliser la route active, et l'autre est censé donc nous fournir cette route.

Dans un second temps, une fois la route active identifiée et accessible, il nous pouvoir en récupérer les paramètres, c'est ici qu'Angular nous propose d'utiliser alors les Observables, de sorte à ce que l'on puisse gérer proprement l'ensemble des mécanismes asynchrones d'Angular.

Une fois cette route entièrement décortiquées et mise en relation avec un composant, il nous faut déterminer ou va s'afficher le contenu associé au composant et à son template, c'est ici que la balise <router-outlet></router-outlet> vient à notre secours. Elle permet de spécifier ou devra se trouver le contenu à ajouter / modifier / remplacer.

```
// configuration d'une route au format REST indiquant une base fixe (details)
// et une partie variable à récupérer (:id)
{path: 'detail/:id', component: JediComponent}

//.....
export class JediComponent implements OnInit {
ngOnInit(): void {
  this.route.paramMap
    .switchMap(
       (params: ParamMap) => console.log(params.get('id'))
    );
}
```

# "Routing" et requête HTTP

# Echange de données avec le service Http

Le module **HttpModule** n'est pas un module central d'Angular, il s'agit plus d'un module optionnel proposant une solution pour la gestion des requêtes AJAX via le protocole HTTP. Il existe donc en tant que module séparé du coeur d'Angular et est contenu dans le paquetage @/http.

Votre application dépendra donc du service Http, qui dépend lui-même d'autres petites choses, pour pouvoir l'utiliser, il vous suffit de spécifier le HttpModule en tant que dépendance du module princpal

```
@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule,
    AppRoutingModule
  ],
  declarations: [
    AppComponent,
    DashboardComponent,
    HeroDetailComponent,
    HeroesComponent,
  ],
  providers: [ HeroService ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

# RouterModule

Angular propose un système de **routing**, çàd qu'il met en relation ce que l'on appelle des urls profondes ( ou deeplinks ) avec des instances de composants et des templates qu'Angular va se charger de créer à la place du développeur.

Le RouterModule peut être importé plusieurs fois, tant que le router traite avec des ressources partagées et globales, nous ne pouvons avoir qu'une seule instance du router service active à la fois. C'est d'ailleurs pour cela qu'il existe 2 façons de créer le module: RouterModule.forRoot and RouterModule.forChild.

La méthode forRoot qui crée un module contenant les directives, les routes et le RouterService lui-même.

La méthode forChild fait exactement la même à l'exclusion du RouterService.

```
@NgModule({
  imports:      [
    BrowserModule,
    FormsModule,
    HttpModule,
    RouterModule.forRoot([{path:"home", component: AppComponent} ])
  ],
  declarations: [ AppComponent ],
  bootstrap:    [ AppComponent ],
  providers:[]
})
export class AppModule { }
```

# Paramètres de route et router-outlet

Afin de récupérer les paramètres des routes en Angular, il nous faut accéder à la route en cours à l'aide de l'objet ActivatedRoute et de l'objet RouteProvider. L'objet de type ActivatedRoute étant là pour symboliser la route active, et l'autre est censé donc nous fournir cette route.

Dans un second temps, une fois la route active identifiée et accessible, il nous pouvoir en récupérer les paramètres, c'est ici qu'Angular nous propose d'utiliser alors les Observables, de sorte à ce que l'on puisse gérer proprement l'ensemble des mécanismes asynchrones d'Angular.

Une fois cette route entièrement décortiquées et mise en relation avec un composant, il nous faut déterminer ou va s'afficher le contenu associé au composant et à son template, c'est ici que la balise <router-outlet></router-outlet> vient à notre secours. Elle permet de spécifier ou devra se trouver le contenu à ajouter / modifier / remplacer.

```
// configuration d'une route au format REST indiquant une base fixe (details)
// et une partie variable à récupérer (:id)
{path: 'detail/:id', component: JediComponent}

//.....
export class JediComponent implements OnInit {
ngOnInit(): void {
  this.route.paramMap
    .switchMap(
       (params: ParamMap) => console.log(params.get('id'))
    );
}
```

# Guards

Angular permet de protéger certaines routes à l'aide du système **Guards**. Protéger certaines routes est très utile par exemple dans le cas où on souhaite restreindre l'accès à certaines zones du site, où même demander une confirmation pour quitter une zone du site. Il existe 4 types de Guard différents:

- **CanActivate** - Décide si une route peut être activée ou pas
- **CanActivateChild** - Décide si l'enfant d'une route peut être activée ou non
- **CanDeactivate** - Décide si une route peut être désactivée
- **CanLoad** - Décide si un module peut être chargé de manière 'fainénante' (à la demande uniquement)

On peut retrouver quelques exemples d'implémentations des Guards à l'adresse suivante: https://angular.io/guide/router#guards.

# Resolver

Le réflexe que l'on peut avoir lorsque l'on développe une application Angular, c'est de définir, dans la méthode ngOnInit d'un composant, un subscribe à une requête de récupération de données, d'affecter cette donnée à un attribut du composant, et d'utiliser cet attribut dans un template. L'inconvénient ici c'est que le premier rendu va se faire avec des valeurs d'attribut vide, ce qui peut provoquer un désagréable effet de clignotement, voire, si la requête tarde à arriver, un instant de doute pour l'utilisateur. Pour éviter cela, nous pouvons faire patienter l'utilisateur jusqu'à ce que le rendu complet de la page soit disponible, à l'aide des resolvers. Les Resolver d'Angular permettent d'attendre le retour d'un observable avant d'initialiser / mettre à jour un composant après une mise à jour de l'url. Le Resolver est une classe que l'on associe à la route du composant. Leur utilisation classique est la récupération de données.

```
//users.resolver.ts

@Injectable()
export class UsersResolver implements Resolve<User[]> {
  constructor(private users: UserService) {}
  resolve(): Observable<User[]> {
    return this.users.getUsers();
  }
}

//app-routing.module.ts
{ path: 'admin',
  component: AdminComponent,
  resolve: {users: UsersResolver  } // on associe un resolver à la route
}

// admin.component.ts
@Component({})
class AdminComponent implements ngOnInit {

    constructor(private route: ActivatedRoute) {}

    ngOnInit() {
        this.route.data.subscribe(
          (data: { users: Users[] }) => this.users = data.users
        );
    }
}
```

# Performances au démarrage & Lazy loading

En configurant l'intégralité du routing de l'application dans un module, on serait amené à importer tous les modules de l'application avant son démarrage. A titre d'exemple, plus l'application sera riche, plus la page d'accueil sera lente à charger par effet de bord. Pour résoudre ces problèmes de performances au démarrage, Angular nous permet de charger les modules à la demande, c'est ce que l'on appelle le lazy loading (ou chargement fainéant). La configuration du lazy loading se fait au niveau du routing.

Seul le module AppRoutingModule importe le module RouterModule avec la méthode statique forRoot afin de définir le 'Routing' racine et la configuration du router via le second paramètre. Les 'Child Routing Modules' importent le RouterModule avec la méthode forChild. Une fois l'application démarrée et pour éviter la latence de chargement des 'Lazy Loaded Routes', il est possible de configurer le 'Routing' pour précharger tous les modules 'Lazy Loaded' juste après le démarrage de l'application.

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
// Ici, en utilisant loadChildren, on précise le chemin vers notre module
// qui contient notre composant associé à la route contact.
// En faisant cela, on déclenchera le chargement de notre module (et donc de notre
composant)
// uniquement lorsque cela est nécessaire.
const routes: Routes = [
  {
    path: 'contact',
    loadChildren: () => import('./contact.module')
      .then(mod => mod.ContactModule)
  }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule],
  declarations: []
})
export class AppRoutingModule { }

// .....
// Contact module
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { ContactComponent } from './contact.component';

const routes: Routes = [ { path: '', component: ContactComponent } ];

@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule]
})
export class ContactRoutingModule { }
```

## Shared Module

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. 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.

```
@NgModule({
   imports: SharedModule.MODULE_LIST,
   exports: SharedModule.MODULE_LIST
})
export class SharedModule {

   static readonly MODULE_LIST = [
      CommonModule,
      FlexLayoutModule,
      RouterModule
   ];

}
```

# Observables & Reactive Programming

# Les Observables

Un Observable est un producteur de données qui peut être observé. On le mettra sous observation avec la méthode **subscribe** et cette observation sera exécutée par un objet de type Observer. Les Observables se sont imposés dans le monde de la programmation réactive ( reactive programming ) par le biais de la librairie Reactive X, abrégé RxJs pour sa version Javascript. Un Observable peut également produire des données de façon asynchrone et il est possible de fusionner plusieurs Observables entre eux.

Il faut bien noter que les Observables sont souvent associés aux promesses, voire décrits comme des super-promesses, ce qui n'est pas tout à fait exact, un article entier a d'ailleurs été rédigé à l'attention des développeurs à l'adresse suivante: https://medium.com/@benlesh/learning-observable-by-building-observable-d5da57405d87 et explique un peu plus en détail ce que sont les Observables.

```
Rx.Observable.create(function(observer) {
    setTimeout(() => observer.next("valeur A"), 700);
    setTimeout(() => observer.next("valeur B"), 400);
})
 .subscribe(e => console.log(e));

 // affiche "valeur A", puis "valeur B"  dans la console
```

## Créer des Observables

Il est possible de créer des Observables en instanciant la classe Observable ou en utilisant des opérateurs dédiés (tels que of, from etc ... ).

```
// Importe la classe Observable ainsi que
// l'intégralité des opérateurs
import {Observable} from 'rxjs/RX';

// on peut créer un Observable à partir d'un jeu de données à l'aide d'opérateurs
let obs = Observable.from(["Dark Vador", "Obiwan", "Maître Yoda"]);
obs.subsribe(
    (character) => {
        console.log(character);
    }
);

// ou alors à l'aide de l'opérateur new
let obs = new Observable(
    (observer) => {
        observer.next("Chewbacca");
        observer.next("Han Solo");
        observer.next("Princesse Leïa");
    }
);
```

# Cold Observables

Un Cold Observable, est un Observable qui crée son propre producteur de données lors du déclenchement de la méthode |"subscribe". On utilise un Cold Observable de préférence uniquement lorsque la création répétée d'un producteur de données n'est pas dérangeante en termes de performances.

L'exemple ci-dessous nous donne l'exemple d'un Cold Observable qui crée son propre producteur de données à chaque subscribe.

```
// ici, à chaque fois que l'on appelle la méthode subscribe de l'observable
// on crée une connexion au serveur websocket et on renvoit une function
// permettant de refermer cette connexion

let obs = new Observable(

    (observer) => {
        let socket = new WebSocket('ws://someurl');
        socket.addEventListener(
            'message',
            (data) => observer.next(data)
        );

        return () => socket.close();
    }
);
```

# Hot Observables

Un Hot Observable, est un Observable dont le producteur de données est déclenché ou crée en dehors de la méthode "subscribe". ON utilisera de préférence un Hot Observable lorsqu'on souhaite écouter des données dont on peut considérer qu'elels peuvent être diffusées **avant et après** la souscription et la désinscription à l'Observable ( ex: MouseEvents ).

L'exemple ci-dessous nous donne l'exemple d'un Hot Observable dont le producteur de données est crée et activé en dehors de la méthode subscribe.

```
// ici on crée un hot Observable, le producteur de données est crée
// en dehors du subscribe, en revanche, on évite de fournir une méthode
// de désinscription qui détruira le producteur de données car d'autres observables
// être en fin d'écouter le même producteur.

let socket = new WebSocket('ws://someurl');

let source = new Observable(
    (observer) => {
        socket.addEventListener(
            'message',
            (e) => observer.next(e)
        );
    }
);
```

# ReplaySubject

Un objet de type Subject est un objet qui est à la fois un Observer et un Observable. En tant qu'Observer, il peut souscrire à un Observable, et en tant qu'Observable, il peut produire des données et recevoir la souscription d'Observers.

Un objet de type ReplaySubject met en cache ses valeurs et les réemet à chacun de ses Observers, même les plus tardifs.

```
function playSubject(p_subject)
{
    subject.onNext("Jedi");

    subject.subscribe(
        (data) => {
            console.log(data);
        }
    );

    subject.onNext("Obiwan");
    subject.onNext("Kenobi");
}
let subject1 = new Rx.Subject();
let subject2 = new Rx.ReplaySubject();

playSubject(subject1); // affiche: Obiwan Kenobi
playSubject(subject2); // affiche: Jedi Obiwan Kenobi
```

## Les opérateurs les plus funs

Il existe une multitude d'opérateurs livrés avec la librairie RxJS, ces derniers sont avant tout là pour nous faciliter la vie, en voici quelques-uns que l'on peut qualifier d'indispensables:

```
/*
   interval: crée un observable qui va produire un entier qui s'incrémente
   toutes les x millisecondes.

   do: un opérateur n'ayant aucune influence sur le flux de données, utilisé
   dans le but de vérifier certaines informations ou d'effectuer une action parralèle

   map: même opérateur qu'en javascript es6, opère un traitement sur chacune
   des valeurs retournées.

   take: permet de limiter la production de données, infinie par défaut, d'un
   observable crée à l'aide d'"interval".
*/
let msg = "STAR WARS VII: The Force Awakens";
let obs = Observable.interval(100)
                .map( (val) => msg[val])
                .do(
                   (val) => {
                      document.body.innerHTML += val;
                   }
                )
                .take(msg.length);
```

# Tests unitaires, bonnes pratiques et outils

# Préparationde l'environnement de test

Afin de pouvoir tester correctement notre application Angular, il nous faut créer un environnement propice aux tests, qui se composera de plusieurs couches:

- L'installation et la configuration de protractor end-to-end test runner (e2e)
- L'installation et la configuration de Jasmine, framework de test unitaire Javascript
- L'installation de Karma, test launcher qui va nous permettre de lancer la moulinette de tests unitaires Javascript sur l'ensemble des navigateurs désirés.
- Et enfin l'écriture de tests à proprement parler

Ce n'est qu'en respectant l'ensemble de ces étapes que nous aurons un environnement de test valable.

# Karma, Jasmine & Protractor

Karma est un outil de terminal JavaScript qui permet le lancement de navigateurs web. Une fois le navigateur lancé, Karma y charge le code de l'application et exécute vos tests. Il est possible d'utiliser Karma afin de lancer votre application sur plusieurs navigateurs (Chrome, Safari, IE, PhantomJS, …). Cela vous permet de vérifier que votre application fonctionne bien partout! Lors de l'exécution de vos tests, les résultats sont affichés directement dans votre terminal. Pour plus d'informations rendez-vous sur le site de karma à l'adresse suivante: https://karma-runner.github.io/1.0/index.html

Jasmine est un framework de tests unitaires Javascript orienté comportement. Il ne dépend d'aucune librairie tierce et ne requiert même pas l'accès à l'API du DOM et il possède une syntaxe claire, évidente, qui permet d'écrire rapidement et simplement des tests unitaires. Pour plus d'informations, rendez-vous sur le site de Jasmine à l'adresse suivante: https://jasmine.github.io/

Protractor est un framework de test de bout en bout pour Angular, qui permet au développeur de tester son application comme un utilisateur lambda mais avec des tâches de tests automatisées. Pour plus d'informations rendez-vous sur le site de protractor à l'adresse suivante: http://www.protractortest.org/#/

# Tester un composant Angular

Un composant Angular est la première chose que tout développeur voudra tester en priorité. Ci-dessous le composant "BannerComponent" est un exemple simple de composant qui peut nous servir comme point de départ. Son rôle est d'afficher le titre de l'application dans une balise h1.

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

@Component({
  selector: 'app-banner',
  template: '<h1>{{title}}</h1>'
})
export class BannerComponent {
  title = 'STAR WARS: THE RETURN OF THE JEDI';
}
```

# Test un composant Angular

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { By }                        from '@angular/platform-browser';
import { DebugElement }              from '@angular/core';
import { BannerComponent }           from './banner-inline.component';

describe('BannerComponent (inline template)', () => {

  let comp:    BannerComponent;
  let fixture: ComponentFixture<BannerComponent>;
  let de:      DebugElement;
  let el:      HTMLElement;

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [ BannerComponent ], // declare the test component
    });

    fixture = TestBed.createComponent(BannerComponent);

    comp = fixture.componentInstance; // BannerComponent test instance

    // query for the title <h1> by CSS element selector
    de = fixture.debugElement.query(By.css('h1'));
    el = de.nativeElement;
  });
});
```

# Tester un composant Angular

**TestBed** nous sert à créer un module de test Angular que l'on configure par la suite à l'aide de la méthode **configureTestingModule** afin de produire un environnement pour la classe à tester.

La méthode **configureTestingModule** prend en paramètre des metadonnées, au même titre que n'importe quel autre module.

Une fois le module de test configuré, on fait appel à la méthode TestBed.createComponent qui crée donc une instance de BannerComponent et qui retourne un objet de type **ComponentFixture**.

Un objet de type ComponentFixture est en fait un **gestionnaire** qui nous permet d'accéder à l'instance de **BannerComponent** ainsi qu'à un objet de type **DebugElement**, ce dernier est un gestionnaire du DOM associé à ce composant.

A l'aide de la méthode **query** de l'objet de type **DebugElement** il est possible d'accéder aux élements du DOM généré à l'aide du template du composant.

# Tester un service Angular

Voyons voir comment tester un service retournant des valeurs asynchrones, ci-dessous, le service WarriorService est censé retourner une Promesse. On voudrait vérifier qu'il y ait au moins 2 guerriers de retournés.

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

@Injectable()
export class WarriorService {

    public getAll(): Promise<Object[]> {
        return Promise.resolve(
            [
                { name: "kenobi", surname: "obiwan" },
                { name: "vador", surname: "dark" }
            ]
        );
    }
}
```

# Tester un service Angular

```
import { TestBed, inject } from '@angular/core/testing';

import { WarriorService } from './warrior.service';

describe('WarriorService', () => {
    beforeEach(() => {
        TestBed.configureTestingModule({
            providers: [WarriorService]
        });
    });

    it('should return at least 2 warriors',
        inject([WarriorService],
            (service: WarriorService) => {
                service.getAll().then(
                    (tab: Object[]) => {
                        expect(tab.length).toBeGreaterThan(1);
                    }
                );
            }
        )
    );
});
```

# Spies & Mocks

Si l'on veut tester un composant faisant appel à un ou plusieurs services, lesquels vont chercher des données de manière asynchrones sur un serveur quelconque, il est recommandé de tout faire pour que les appels en question ne soient pas réalisés.

En effet, le serveur cible peut être innateignable depuis l'environnement de test, ou tout simplement éteint, ce qui ferait échouer toute la chaîne de tests, il est donc courant:

Soit de fournir un faux service émulant les comportements du service injecté au composant.

Soit de court-circuiter les méthodes du service renvoyant des données asynchrones à l'aide de **spies et de mocks.**

Un spy comme son nom l'indique, est une fonction qui a pour but d'espionner l'appel à une autre fonction. Une fois la fonction espionnée appelée, l'espion prend le relais et opère un traitement différent, émulant ainsi le comportement souhaité.

Un mock consiste tout simplement en une série de données ressemblant trait pour trait aux données que le serveur nous aurait renvoyés. Le but est de disposer d'un jeu de données conforme aux attentes du test et ce, sans faire appel à un environnement extérieur.

## Spies & Mocks

```
beforeEach(() => {
  TestBed.configureTestingModule({
    declarations: [ WarriorComponent ],
    providers:    [ WarriorService ],
  });

  fixture = TestBed.createComponent(WarriorComponent);
  comp    = fixture.componentInstance;
  service = fixture.debugElement.injector.get(WarriorService);

  spy = spyOn(service, 'getAll')
        .and.returnValue(Promise.resolve(
          [{"name": "skywalker", "surname": "anakin"}]
        ));

  // WarriorComponent.refresh callas WarriorService.getAll()
  // and fills his public "warriors" property
  comp.refresh();

  expect(comp.warriors.length).toBeGreatThan(0);
});
```

# Webpack

# Webpack: présentation é & configuration

Webpack est ce que l'on peut appeler un empaqueteur de module. Un paquet est un fichier Javascript qui incorpore des contenus ayant un rapport les uns avec les autres et qui doivent être délivrés au client en une seule requête.

Un paquet peut être composé de fichiers Javascript, CSS, HTML, images etc... Pour empaqueter ces différents types de fichiers, Webpack traverse toute votre application à la recherche de références directes à des fichiers, qu'il charge par le biais de "loaders" (des unités logicielles prenant en charge un type de fichier spécifique), ou à des classes.

Il est bien entendu possible de configurer Webpack à l'aide du fichier **webpack.config.js**.

## Les loaders indispensables

- **awesome-typescript-loader**: Un loader permettant de prendre en charge les fichiers Typescript.
- **css-loader**: Un loader permettant de prendre en charge les fichiers css.
- **html-loader**: Un loader permettant de prendre en charge les fichiers HTML
- **raw-loader**: Un loader permettant de prendre en charge des données binaires brutes, images, vidéos etc...
- **style-loader**: Un loader permettant de prendre en charge des styles css embarqués.
- **file-loader**: Un loader permettant de prendre en charge des fichiers de type texte.

# Concepts avancés : lazy loading

# Performances au démarrage & Lazy loading

En configurant l'intégralité du routing de l'application dans un module, on serait amené à importer tous les modules de l'application avant son démarrage. A titre d'exemple, plus l'application sera riche, plus la page d'accueil sera lente à charger par effet de bord. Pour résoudre ces problèmes de performances au démarrage, Angular nous permet de charger les modules à la demande, c'est ce que l'on appelle le lazy loading (ou chargement fainéant). La configuration du lazy loading se fait au niveau du routing.

Seul le module AppRoutingModule importe le module RouterModule avec la méthode statique forRoot afin de définir le 'Routing' racine et la configuration du router via le second paramètre. Les 'Child Routing Modules' importent le RouterModule avec la méthode forChild. Une fois l'application démarrée et pour éviter la latence de chargement des 'Lazy Loaded Routes', il est possible de configurer le 'Routing' pour précharger tous les modules 'Lazy Loaded' juste après le démarrage de l'application.

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
// Ici, en utilisant loadChildren, on précise le chemin vers notre module
// qui contient notre composant associé à la route contact.
// En faisant cela, on déclenchera le chargement de notre module (et donc de notre
composant)
// uniquement lorsque cela est nécessaire.
const routes: Routes = [
  {
    path: 'contact',
    loadChildren: () => import('./contact.module')
      .then(mod => mod.ContactModule)
  }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule],
  declarations: []
})
export class AppRoutingModule { }

// .....
// Contact module
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { ContactComponent } from './contact.component';

const routes: Routes = [ { path: '', component: ContactComponent } ];

@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule]
})
export class ContactRoutingModule { }
```

## Shared Module

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. 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.

```
@NgModule({
   imports: SharedModule.MODULE_LIST,
   exports: SharedModule.MODULE_LIST
})
export class SharedModule {

   static readonly MODULE_LIST = [
      CommonModule,
      FlexLayoutModule,
      RouterModule
   ];

}
```

# Concepts avancés : ahead of time compilation

# Ahead Of Time compilcation VS Just In Time compilation

Une application Angular est principalement constituée de composants et des templates HTML qui y sont associés. Avant d'être rendue, ces derniers doivent être convertis en code Javascript éxécutable par le compilateur intégré à Angular, et à partir de ce moment là 2 solutions s'offrent à vous:

- Soit vous laissez le mode de compilation classique propre à la machine virtuelle Javascript opérer (J.I.T compiler), ce qui consiste en gros à faire en sorte que les composants et templates sont recompilés à l'éxécution.
- Soit vous pré-compilez ce code en amont (AOT compilation) à l'aide des outils fournis par Angular, ce qui constitue un gain en performance et vous permet également de découvrir les erreurs autrement qu'à l'éxécution.

Les avantages sont multiples à utiliser la compilation A.O.T, en production c'est le mode compilation à privilégier, lors du développement, cela reste à l'appréciation du développeur. Utiliser ce mode de compilation réclame cependant un surplus de configuration que vous pourrez trouver à l'adresse suivante: https://angular.io/guide/aot-compiler. (https://angular.io/guide/aot-compiler). Pour déclencher la compilation AOT, vous pouvez lancer la commande **ng build --aot** ou encore **ng build --prod**, il s'agit donc bien du mode par défaut pour la production !

# Concepts avancés : route guards & resolve

# Guards

Angular permet de protéger certaines routes à l'aide du système **Guards**. Protéger certaines routes est très utile par exemple dans le cas où on souhaite restreindre l'accès à certaines zones du site, où même demander une confirmation pour quitter une zone du site. Il existe 4 types de Guard différents:

- **CanActivate** - Décide si une route peut être activée ou pas
- **CanActivateChild** - Décide si l'enfant d'une route peut être activée ou non
- **CanDeactivate** - Décide si une route peut être désactivée
- **CanLoad** - Décide si un module peut être chargé de manière 'fainénante' (à la demande uniquement)

On peut retrouver quelques exemples d'implémentations des Guards à l'adresse suivante: https://angular.io/guide/router#guards.

# Resolver

Le réflexe que l'on peut avoir lorsque l'on développe une application Angular, c'est de définir, dans la méthode ngOnInit d'un composant, un subscribe à une requête de récupération de données, d'affecter cette donnée à un attribut du composant, et d'utiliser cet attribut dans un template. L'inconvénient ici c'est que le premier rendu va se faire avec des valeurs d'attribut vide, ce qui peut provoquer un désagréable effet de clignotement, voire, si la requête tarde à arriver, un instant de doute pour l'utilisateur. Pour éviter cela, nous pouvons faire patienter l'utilisateur jusqu'à ce que le rendu complet de la page soit disponible, à l'aide des resolvers. Les Resolver d'Angular permettent d'attendre le retour d'un observable avant d'initialiser / mettre à jour un composant après une mise à jour de l'url. Le Resolver est une classe que l'on associe à la route du composant. Leur utilisation classique est la récupération de données.

```
//users.resolver.ts

@Injectable()
export class UsersResolver implements Resolve<User[]> {
  constructor(private users: UserService) {}
  resolve(): Observable<User[]> {
    return this.users.getUsers();
  }
}

//app-routing.module.ts
{ path: 'admin',
  component: AdminComponent,
  resolve: {users: UsersResolver  } // on associe un resolver à la route
}

// admin.component.ts
@Component({})
class AdminComponent implements ngOnInit {

    constructor(private route: ActivatedRoute) {}

    ngOnInit() {
        this.route.data.subscribe(
          (data: { users: Users[] }) => this.users = data.users
        );
    }
}
```

# Concepts avancés : internationalisation i18n

# Internationalisation

L'internationalisation (i18n) est le processus de conception et de préparation de votre application pour qu'elle soit utilisable dans différents pays du monde. La localisation est le processus de création de versions de votre application pour différents paramètres régionaux, y compris l'extraction de texte à traduire dans différentes langues et le formatage des données pour des paramètres régionaux particuliers. Un paramètre régional identifie une région (comme un pays) dans laquelle les gens parlent une langue ou une variante de langue particulière. Les paramètres régionaux déterminent la mise en forme et l'analyse des dates, heures, nombres et devises, ainsi que les unités de mesure et les noms traduits pour les fuseaux horaires, les langues et les pays. Vous pouvez utiliser Angular pour internationaliser votre application en:

- Utilisant des pipes intégrés pour afficher des dates, nombres, pourcentages et monnaies dans un format local.
- En utilisant des marqueurs de texte au sein de vos composants.
- En utilisant des marqueurs de formes plurielles.
- En utilisant des marqueurs de textes alternatifs.

Après avoir préparé votre application pour l'internationalisation, utilisez angular CLI comme ceci:

- En extrayant les textes marqués vers un fichier source de langage.
- En créant une copie de ce fichier pour chaque langage et en l'envoyant à un traducteur.
- En fusionnant les fichiers de traduction finalisés à l'aide d'angular CLI.

Pour tirer avantage des fonctionnalités d'internationalisation d'Angular, utilisez angular CLI pour ajouter le package @angular/localize à votre projet:

```
ng add @angular/localize
```

Utilisez des ID (Unicode locale identifier) pour l'ensemble des langages. Pour trouver une liste des ID pour chaque langage, rendez-vous sur la page de la spécification ISO 639-2 (https://www.loc.gov/standards/iso639-2/)

La liste des locales, conforme au CLDR (Unicode Common Locale Data Respository), est disponible sur la page de la spécification CLDR (http://cldr.unicode.org/core-spec#Unicode_Language_and_Locale_Identifiers)

L'ID consiste en un identifiant de langage, comment par exemple 'en' pour l'anglais, 'fr' pour le français suivi d'un dash (-) et d'une extension désignant la locale (comme us pour les USA). Par exemple, l'ID en-US fait référence à l'anglais mais celui parlé aux USA et fr-CA fait référence au français du Canada. Angular utilise cet ID pour trouver la locale correspondante.

Angular utilise la locale en-US en tant que votre fichier source de locale par défault. L'extraction génère un fichier .xlf, qui est un standard dans le monde de la traduction, la commande pour extraire les textes marqués est la suivante:

```
ng xi18n <project> [options]
ng i18n-extract <project> [options]

<?xml version="1.0" encoding="UTF-8" ?>
<xliff version="1.2" xmlns="urn:oasis:names:tc:xliff:document:1.2">
  <file source-language="en-US" datatype="plaintext" original="ng2.template">
    <body>
      <trans-unit id="6226cbeebaffaec0342459915ef7d9b0e9e92977" datatype="html">
        <source>app is running!</source>
        <context-group purpose="location">
          <context context-type="sourcefile">src/app/app.component.html</context>
          <context context-type="linenumber">2</context>
        </context-group>
      </trans-unit>
    </body>
  </file>
</xliff>
```

# Internationalisation les marqueurs

Marquer les textes statiques au sein de vos composants à l'aide de l'attribut i18n. Placez-le sur chaque élément avec du texte destiné à être localisé. Dans l'exemple suivant, la balise h1 affiche un texte simple, écrit en anglais: "Hello i18n !". Afin de faire en sorte que ce texte soit localisable, on ajoute l'attribut i18 à la balise h1.

```
// Ce texte n'est pas marqué comme localisable
<h1>Hello I am not marked !</h1>

// Ce texte est marqué comme localisable
<h1 i18n>Hello i18n!</h1>

//i18n est un attribut customisé, reconnu par les outils d'angular et les compilateurs. Après traduction, le compilateur le retire. Il ne s'agit pas d'une directive d'Angular.

// Parfois, pour traduire efficacement, le traducteur a besoin de contexte, afin de
// pouvoir ajouter ce contexte facilement au fichier de traduction, vous pouvez donner
// une valeur à l'attribut i18n. Le traducteur pourrait également avoir besoin d'une
// supplémentaire, pour cela, il suffit de séparer le contexte et la description par un | .
<h1 i18n="First message|Just say hello">Hello i18n!</h1>

//Afin de pouvoir traduire un attribut, par exemple celui d'une image, il faut procéder comme suit:

<img [src]="logo" i18n-title title="Angular logo" />
```

L'outil d'extraction de texte génère un fichier ou chaque entrée correspond à un élément marqué de l'attribut i18n. À chaque entrée est associée un identifiant unique, basé sur le contexte et la description. Des éléments de textes similaires, mais avec des contextes différents seront notés comme étant deux entrées différentes au sein du fichier de traduction. Par exemple, si le mot 'right' est traduit par correct une première fois, il peut être traduit par 'droite' à un autre endroit, pour ce faire, il faut alors préciser des contextes différents au sein de l'attribut i18n. En revanche, si deux textes différents possèdent le même 'contexte', ils seront compris comme étant une seule et même entrée au sein du fichier de traduction.

# Internationalisation les pluriels et le singulier

Différentes langues ont des règles de pluralisation et des constructions grammaticales différentes qui peuvent rendre la traduction difficile. Pour simplifier la traduction, utilisez les clauses International Components for Unicode (ICU) avec des expressions régulières, telles que le pluriel pour marquer les utilisations de nombres au pluriel, et sélectionnez pour marquer les choix de texte alternatifs. Afin de marquer un pluriel, utilisez l'attribut 'plural' sur les expressions qui n'auraient pas de sens si elles étaient traduites mot à mot. Par exemple, si vous voulez afficher l'expression: 'mis à jour il y a x minutes' en anglais, vous pourriez vouloir afficher les valeurs 'à l'instant', 'il y a une minute', ou 'il y a x minutes' (x étant le vrai nombre de minutes). Certains langages pourraient exprimer cette pluralité de façon différente, l'exemple suivant comment utiliser une clause plurielle qui prend en charge ces trois options pour la locale en cours:

```
// Dans l'exemple en cours, le premier paramètre 'minutes' est lié à la propriété du composant
// du même nom, le second paramètre quand à lui, marque l'expression comme nécessitant la prise en charge
// de la pluralité.
// le troisième paramètre quand à lui définit le pattern de pluralisation et les valeurs correspondantes.

// Les différentes catégories de pluralisation sont les suivantes:
// =0 (or any other number)
// zero
// one
// two
// few
// many
// other

// ATTENTION, CERTAINES LOCALES NE SUPPORTENT PAS CERTAINES CATEGORIES.
<span i18n>Updated {minutes, plural, =0 {just now} =1 {one minute ago} other {{{minutes}} minutes ago}}</span>
```

L'outil d'extraction de texte génère un fichier ou chaque entrée correspond à un élément marqué de l'attribut i18n. À chaque entrée est associée un identifiant unique, basé sur le contexte et la description. Des éléments de textes similaires, mais avec des contextes différents seront notés comme étant deux entrées différentes au sein du fichier de traduction. Par exemple, si le mot 'right' est traduit par correct une première fois, il peut être traduit par 'droite' à un autre endroit, pour ce faire, il faut alors préciser des contextes différents au sein de l'attribut i18n. En revanche, si deux textes différents possèdent le même 'contexte', ils seront compris comme étant une seule et même entrée au sein du fichier de traduction.

# Internationalisation - fusionner les textes au sein de l'application

Pour fusionner les traductions dans l'application, utilisez angular CLI pour créer une copie des fichiers distribuables de l'application pour chaque paramètre régional. Le processus de génération remplace le texte d'origine par du texte traduit et définit le jeton LOCALE_ID pour chaque copie distribuable de l'application. Il charge et enregistre également les données locales.

Après la fusion, vous pouvez servir chaque copie distribuable de l'application à l'aide de la détection de langue côté serveur ou de différents sous-répertoires, comme décrit dans la section suivante sur le déploiement de plusieurs paramètres régionaux.

Le processus de construction utilise la compilation AOT pour produire une petite application rapide et prête à l'emploi. Avec Ivy dans Angular version 9, AOT est utilisé par défaut pour les versions de développement et de production, et AOT est requis pour localiser les modèles de composants.

Pour créer une copie distribuable distincte de l'application pour chaque paramètre régional, définissez les paramètres régionaux dans le fichier de configuration de votre projet angular.json. Cette méthode raccourcit le processus de génération en supprimant la nécessité d'effectuer une génération complète d'application pour chaque paramètre régional.

Vous pouvez ensuite générer des versions d'application pour chaque locale à l'aide de l'option '--localize' dans angular.json. Vous pouvez également créer à partir de la ligne de commande en utilisant la commande de construction Angular CLI avec l'option --localize.

Utilisez l'option de projet 'i18n' dans le fichier de configuration de construction de votre application (angular.json) pour définir les paramètres régionaux d'un projet. Les sous-options suivantes identifient la langue source et indiquent au compilateur où trouver les traductions prises en charge pour le projet:

**sourceLocale:** les paramètres régionaux que vous utilisez dans le code source de l'application (en-US par défaut)

**locales:** une carte des identificateurs de paramètres régionaux aux fichiers de traduction

Par exemple, l'extrait suivant d'un fichier angular.json définit les paramètres régionaux source sur en-US et fournit le chemin d'accès au fichier de traduction des paramètres régionaux fr (français):

```
{
   "build": {
      "builder": "@angular-devkit/build-angular:browser",
      "options": {
         "localize": true,
         "aot": true,
         // ....
         "projects": {
            "angular.io-example": {
               // ...
               "i18n": {
                  "sourceLocale": "en-US",
                  "locales": {
                     "fr": "src/locale/messages.fr.xlf"
                  }
               }
            }
         }
      }
   }
}
```

# Ecosystème : @ngrx/store

# Immutable data store avec @ngrx/store

Store est une gestion globale de l'état optimisée par RxJS pour les applications Angular, inspirée de Redux. Store est un conteneur à état contrôlé conçu pour aider à écrire des applications performantes et cohérentes sur Angular. Les concepts clés de Store sont:

- Les Actions décrivent des événements uniques qui sont distribués à partir de composants et de services.
- Les changements d'état sont gérés par des fonctions pures appelées 'reducers' qui prennent l'état actuel et la dernière action pour calculer un nouvel état.
- Les Selectors sont des fonctions pures utilisées pour sélectionner, dériver et composer des éléments d'état.
- L'état (State) est accessible via le magasin (Store), un observable du state et un observer des actions.

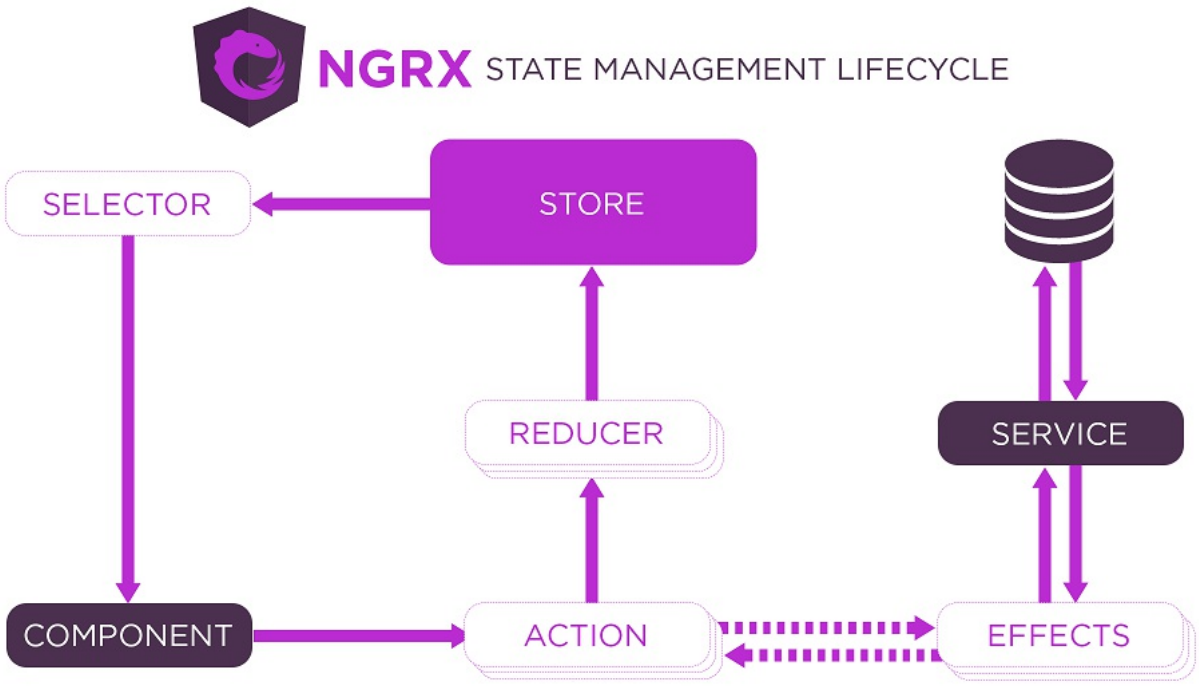
NgRx Store est principalement pensé pour gérer un état global à travers une application entière. Dans le cas où vous auriez besoin de gérer des états temporaires ou locaux, essayez d'utiliser NgRx ComponentStore.

Les changements d'état sont gérés par des fonctions pures appelées 'reducers' qui prennent l'état actuel et la dernière action pour calculer un nouvel état.

Les Selectors sont des fonctions pures utilisées pour sélectionner, dériver et composer des éléments d'état.

L'état (State) est accessible via le magasin (Store), un observable du state et un observer des actions.

Afin d'avoir un exemple d'implémentation de nrgx store au sein d'une application Angular, rdv à l'adresse suivante: https://ngrx.io/guide/store (https://ngrx.io/guide/store)


NGRX STATE MANAGEMENT LIFECYCLE
SELECTOR
STORE
REDUCER
SERVICE
COMPONENT
ACTION
EFFECTS

# Ecosystème : server side rendering

# Server Side Rendering avec Angular

Pour être visité par un grand nombre d'utilisateurs, un site web se doit de remplir deux conditions essentielles, à savoir s'afficher le plus rapidement possible et être bien référencé par les moteurs de recherche. Le problème, c'est qu'Angular est ce que l'on appelle une SPA (single page app), et que l'affichage du contenu référençable est géré par Javascript. Les robots de référencement ne comprennent pas le Javascript, il est donc impossible pour eux de référençer le moindre contenu de la sorte. Pour cela, nous devons donc opérer notre rendu côté serveur, et cette technique s'appelle le server side rendering (SSR). Pour pouvoir uriliser le SSR, nous allons utiliser Angular Universal. Pour pouvoir ajouter Angular Universal à notre projet, nous allons taper la commande suivante:

```
# Installation
ng add @nguniversal/express-engine

#Pour utiliser le SSR, lancez la commande suivante
npm run dev:ssr
```

Pour rappel angular CLI utilise via la directive ng add le principe des schematics pour modifier notre code et l'adapter à la nouvelle fonctionnalité (ici le ssr). De nombreuses opérations ont été effectuées automatiquement sur notre projet. Si nous avions dû réaliser cette opération manuellement voici les différentes étapes que nous aurions dû suivre.

- Installation des nouvelles dépendances nécessaires
- Modification du fichier main.ts
- Modification du fichier app.module.ts
- Modification du fichier angular.json
- Création du fichier src/app/app.server.module.ts
- Création du fichier src/main.server.ts
- Création du fichier server.ts
- Création du fichier tsconfig.server.json
- Modification du fichier angular.json
- Modification du fichier package.json

On peut constater qu'un certain nombre de dépendances ont ainsi été ajoutés de façon transparente, si nous avions du les installer à la main, voici ce que nous aurions du faire:

On peut constater qu'un certain nombre de dépendances ont ainsi été ajoutés de façon transparente:

```
# Installer les nouvelles dépendances dans package.json
npm install --save @angular/platform-server
npm install --save @nguniversal/express-engine
npm install --save express
npm install --save @nguniversal/builders
npm install --save @types/express
```

Notez bien que le package @angular/platform-server contient la plupart des dépendances que nous allons utiliser et qui nous permettent d'adapter notre code front-end ç un contexte back-end.

## Server bootstrap

Si nous allons faire un tour du côté du fichier server.ts, nous pouvons constater que beaucoup de code s'y trouve, mais la partie la plus intéréssante est celle-ci:

```
// Our Universal express-engine (found @
// https://github.com/angular/universal/tree/master/modules/express-engine)
server.engine('html', ngExpressEngine({
  bootstrap: AppServerModule,
}));
```

La fonction ngExpressEngine() function est un wrapper de la fonction renderModule() d'Universal. Ce wrapper transforme une requête client en une page HTML rendue côté serveur. Le wrapper accepte un objet avec les options suivantes:

- bootstrap: Le module de type NgModule racineà utiliser afin de démarrer notre application côté serveur, Ici, un AppServerModule a été généré par défaut, il s'agit du pont entre notre application Angular classique et notre application Angular Universal.
- extraProviders: Optionnel, vous laissez définir les éventuelles dépendances spécifiques à votre application côté serveur.

Vous noterez donc qu'il est tout à fait possible de générer des pages statiques et des pages dynamiques, en effet, notre premier rendu, et le téléchargement des données associés, est assuré par Angular Universal et retourné au client sous la forme d'une page web classique. Une fois le premier rendu opéré, c'est l'application côté client qui reprend la main, afin de faire bénéficier à notre client de la vélocité et de la légèreté d'une SPA. Cela résoud donc les problèmes des robots tout en gardant les avantages pour les clients. Vous constaterez que nous pouvons filtrer les requêtes qui seront traités côté Anggular Universal:

```
// Par exemple pour interdire uniquement les requêtes commençant par la chaîne 'api/**'
// il nous faut nous rendre dans le fichier server.ts à nouveau.
server.get('/api/**', (req, res) => {
  res.status(404).send('data requests are not yet supported');
});

// Pour rendre les fichiers statiques (tels les images) de façon correcte voici
// comment on peut s'y prendre.
server.get('*.*',
  express.static(
    distFolder, // chemin vers le dossier contenant les assets
    {maxAge: '1y'} // durée du cache = 1 an
  )
);
```

# Ecosystème : pwa

# Service Worker

Les Services Workers enrichissent le modèle de déploiement Web traditionnel et permettent aux applications de fournir une expérience utilisateur avec une fiabilité et des performances comparables à celles du code installé en mode natif. L'ajout d'un service worker à une application Angular est l'une des étapes permettant de transformer une application en application Web progressive (également appelée PWA). Dans sa forme la plus simple, un service worker est un script qui s'exécute dans le navigateur Web et gère la mise en cache d'une application.

Les Services Workers fonctionnent comme un proxy réseau. Ils interceptent toutes les requêtes HTTP sortantes faites par l'application et peuvent choisir comment y répondre. Par exemple, ils peuvent interroger un cache local et fournir une réponse mise en cache si elle est disponible. Le proxy ne se limite pas aux demandes effectuées via des API programmatiques, telles que la récupération; il inclut également des ressources référencées en HTML et même la requête initiale à index.html. La mise en cache basée sur le service worker est donc entièrement programmable et ne repose pas sur les en-têtes de mise en cache spécifiés par le serveur.

Contrairement aux autres scripts qui composent une application, tels que le bundle d'applications Angular, le service worker est conservé une fois que l'utilisateur ferme l'onglet. La prochaine fois que le navigateur charge l'application, le service worker se charge en premier et peut intercepter chaque demande de ressources pour charger l'application. Si le service worker est conçu pour le faire, il peut satisfaire complètement le chargement de l'application, sans avoir besoin du réseau.

Même sur un réseau rapide et fiable, les aller-retour peuvent introduire une latence importante lors du chargement de l'application. L'utilisation d'un service worker pour réduire la dépendance au réseau peut considérablement améliorer l'expérience utilisateur.

Les applications Angular, en tant que SPA, sont dans une position privilégiée pour bénéficier des avantages des Service Workers. À partir de la version 5.0.0, Angular est livré avec une implémentation de service worker. Les développeurs Angular peuvent tirer parti de ce service worker et bénéficier de la fiabilité et des performances accrues qu'il offre, sans avoir besoin de coder avec des API de bas niveau.

Le service worker d'Angular est conçu pour optimiser l'expérience de l'utilisateur final lors de l'utilisation d'une application sur une connexion réseau lente ou peu fiable, tout en minimisant les risques liés à la diffusion de contenu obsolète. Le comportement du Service Worker d'Angular suit cet objectif de conception:

- La mise en cache d'une application revient à installer une application native.
  L'application est mise en cache comme une unité et tous les fichiers sont mis à jour ensemble.

- Une application en cours d'exécution continue de s'exécuter avec la même version de tous les fichiers. Elle ne commence pas soudainement à recevoir les fichiers mis en cache d'une version plus récente, qui sont probablement incompatibles.
- Lorsque les utilisateurs actualisent l'application, ils voient la dernière version entièrement mise en cache. Les nouveaux onglets chargent le dernier code mis en cache.
- Les mises à jour se produisent en arrière-plan, relativement rapidement après la publication des modifications. La version précédente de l'application est servie jusqu'à ce qu'une mise à jour soit installée et prête.
- Le Service Worker conserve la bande passante lorsque cela est possible. Les ressources ne sont téléchargées que si elles ont changées.

Pour prendre en charge ces comportements, le Service Worker Angular charge un fichier 'manifest' à partir du serveur. Le manifeste décrit les ressources à mettre en cache et inclut les hashages du contenu de chaque fichier. Lorsqu'une mise à jour de l'application est déployée, le contenu du manifeste change, informant le Service Worker qu'une nouvelle version de l'application doit être téléchargée et mise en cache. Ce manifeste est généré à partir d'un fichier de configuration généré par CLI appelé ngsw-config.json.

L'installation du service worker Angular est aussi simple que l'inclusion d'un NgModule. En plus d'enregistrer le service worker Angular avec le navigateur, cela rend également disponibles pour l'injection quelques services qui interagissent avec le service worker et peuvent être utilisés pour le contrôler. Par exemple, une application peut demander à être notifiée lorsqu'une nouvelle mise à jour devient disponible, ou une application peut demander au Service Worker de vérifier le serveur pour les mises à jour disponibles. Les pré-requis à l'utilisation des Service Workers sont relativement nombreux:

- Pour utiliser toutes les fonctionnalités d'Angular Service Worker, il vous faut utiliser les dernières versions d'Angular et de la CLI Angular.
- Pour que Service Workers soient enregistrés, l'application doit être accessible via HTTPS et non via HTTP. Les navigateurs ignorent les Service Workers sur les pages qui sont servies via une connexion non sécurisée. La raison en est que les Service Workers sont assez puissants, donc des précautions supplémentaires doivent être prises pour s'assurer que le script du service worker n'a pas été falsifié. Il existe une exception à cette règle: pour faciliter le développement local, les navigateurs ne nécessitent pas de connexion sécurisée lors de l'accès à une application sur localhost.
- Pour bénéficier du service worker Angular, votre application doit s'exécuter dans un navigateur Web prenant en charge les Service Workers en général. Actuellement, les Service Workers sont pris en charge dans les dernières versions de Chrome, Firefox, Edge, Safari, Opera, UC Browser (version Android) et Samsung Internet. Les navigateurs comme IE et Opera Mini ne prennent pas en charge les Service Workers.

Du fait des nombreuses conditions requises, il est recommandé de faire en sorte que votre application fonctionne, même sans les Service Workers. Pour en savoir plus sur leur mise en place, rendez-vous à l'adresse suivante: https://angular.io/guide/service-worker-intro

(https://angular.io/guide/service-worker-intro)

# Les notifications Push

Les notifications Push sont basées sur deux standards navigateurs différents:

La Push API, qui permet à des messages provenant d'un serveur d'être poussés vers le navigateur.

Et la Notification API, qui permet d'afficher des notifications natives système à notre utilisateur.

Mais notez ici que l'on ne peut PAS pousser une notification directement depuis NOTRE serveur vers le navigateur. Au lieu de ça, seuls certain serveurs, que les développeurs de navigateurs choisissent (Google, Mozilla etc.), sont autorisés à pousser des notifications vers un navigateur. Ces serveurs sont connus sous le nom de **Browser Push Service**. Notez que que le Browser Push Service utilisé par Chrome n'est pas nécessairement le même que celui utilisé par Firefox etc. Cette sécurité est prévue par les constructeurs afin de prévenir les utilisateurs d'un trop grand nombre de notifications non justifiées. Elles choisissent donc de contrôler chacune d'entre elles. Dans le cas de Chrome, toutes les notifications passent pe le cloud de Firebase. Pour en savoir plus sur la mise en place de notifications Push au sein d'une application Angular, rendez-vous à l'adresse suivante: https://blog.angular-university.io/angular-push-notifications/ (https://blog.angular-university.io/angular-push-notifications/). Notez toutefois que pour bénéficier des notifications Push, il vous faudra avoir accès aux Service Workers.

# Ecosystème : @angular/material

# Angular Material

Angular Material est un package tiers utilisé sur les projets Angular pour faciliter le processus de développement grâce à la réutilisation de composants communs tels que des cartes, de belles entrées, des tables de données, etc. La liste des composants disponibles est longue et continue de s'allonger au fur et à mesure que nous parlons. Donc, pour une référence complète des composants avec des exemples, consultez le site officiel.

Avec Angular, l'application entière est une composition de composants et, au lieu de créer et de styliser des composants à partir du groupe, vous pouvez utiliser Angular Material qui fournit des composants de style prêts à l'emploi qui suivent les spécifications de conception de matériaux. Cette spécification est utilisée par Google dans le système d'exploitation Android et est également très populaire sur le Web en raison de ses magnifiques utilitaires d'interface utilisateur. Pour installer Angular Material sur votre projet angular, vous devez rentrer la commande suivante:

```
npm install @angular/material @angular/cdk

// ... other import statements ...
import {BrowserAnimationsModule} from '@angular/platform-browser/animations';
import {MaterialModule} from './material.module';

@NgModule({
  // ... declarations property ...
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    MaterialModule,
  ],
  // ... providers and bootstrap properties ...
})
export class AppModule {}
```

# Angular Material Demo & CDK

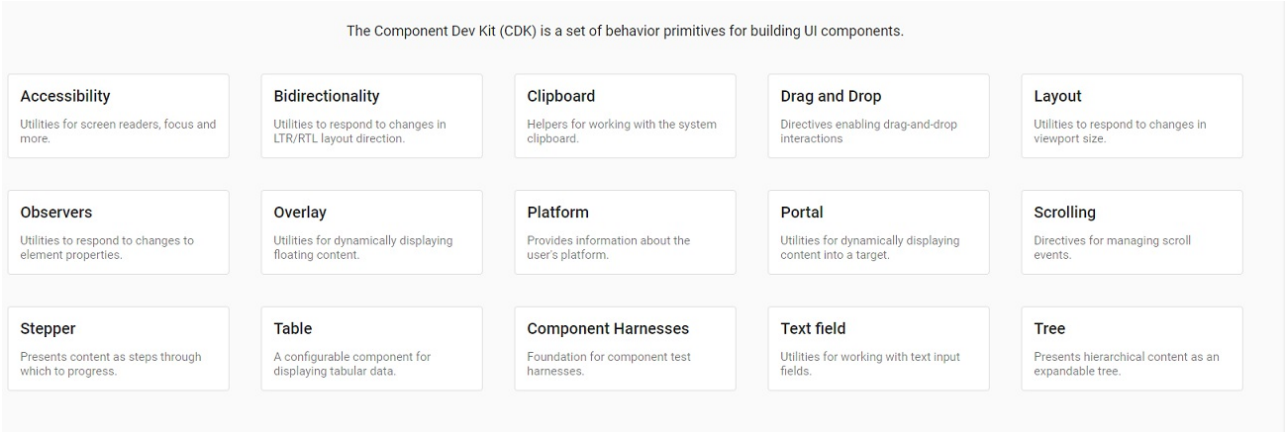
Voici comment créer un composant utilisant un composant Angular Material:

```
import {BrowserAnimationsModule} from '@angular/platform-browser/animations';
import {MatSliderModule} from '@angular/material/slider';

@NgModule({
  imports: [ BrowserModule, BrowserAnimationsModule, MatSliderModule ],
})
export class AppModule {}

<!-- The following component is now available in your app -->
<mat-slider min="1" max="100" step="1" value="1"></mat-slider>
```

Comme on peut le voir, il est assez facile d'utiliser les composants Angular Material. Angular Material vient donc avec un jeu de composants déjà précodés et prêts à l'emploi. Mais il vient également avec ce que l'on appelle le CDK (Component Dev Kit), un regroupement de comportements qui aident à créer des composants. Pour avoir la liste complète de ces comportements, rendez-vous à cette adresse: [https://material.angular.io/cdk/categories (https://material.angular.io/cdk/categories)](https://material.angular.io/cdk/categories).



# Communication en temps réel

# Problématiques et définition

La communication en temps réel permet à deux process d'un même programme ( ou de programmes différents ) de communiquer par le biais de messages, et ce, sans faire appel à des opérations bloquantes. Les messages sont envoyés, directement au destinataire ou à un programme tiers qui fait alors office de gestionnaire de message.

Les messages sont alors consommés par le destinataire, et celui-ci peut choisir d'y répondre ou non sans limite de délai. Bien entendu une telle façon de communiquer doit être accompagnée d'une façon de programmer bien précise. Le javascript, de part sa nature asynchrone, répond très à ce défi, notamment grâce à sa gestion de l'asynchronicité et à son modèle évenementiel.

Il existe plusieurs technologies, natives ou non, nous permettant de mettre en place une communication en temps réel inter processus en javascript. Redis, RabbitMQ, Socket.IO ou encore les websockets seules sont autant de solutions nous permettant de répondre à nos besoins en la matière.

# Websockets et Socket.IO

Socket.IO est une bibliothèque JavaScript pour les applications Web en temps réel. Il permet une communication bidirectionnelle en temps réel entre les clients Web et les serveurs. Il comporte deux parties: un côté client bibliothèque qui fonctionne dans le navigateur , et un côté serveur bibliothèque pour Node.js . Les deux composants ont une API presque identique . Comme Node.js , il est piloté par les événements.

Socket.IO utilise principalement le protocole WebSocket avec l'interrogation comme option de secours, tout en fournissant la même interface. Bien qu'il puisse être utilisé comme un simple wrapper pour WebSocket, il fournit de nombreuses autres fonctionnalités, notamment la diffusion vers plusieurs sockets, le stockage des données associées à chaque client et les E / S asynchrones. Il peut être installé avec l' outil npm.

socketio client

```
const io = require("socket.io-client");
const socket = io("ws://localhost:3000");

function start(){
    socket.send("classic client")
    socket.emit("clientEvent", "custom client");
}

socket.on("connect", start);
socket.on("message", console.log);
socket.on("serverEvent", console.log);
```

socketio server

```
const io = require("socket.io")(3000);

function init(socket){
    socket.on("message", console.log );
    socket.send("classic server");

    socket.on("clientEvent", console.log );
    socket.emit("serverEvent", "custom server");
}

io.on("connection", init);
```

# TP: Server Side Events & Websockets

Le but de ce TP est de créer un tchat en temps réel à l'aide de Websockets. Le tchat devra comporter les fonctionnalités suivantes:

- Il doit y avoir un salon principal, où tout le monde peut s'exprimer.
- Il doit y avoir la possibilité de créer un salon privé entre deux utilisateurs.
- Il doit y avoir la possibilité de fermer un salon privé entre deux utilisateurs.
- BONUS: Il doit y avoir la possibilité d'envoyer des emojis

L'utilisation de socket.io est recommandée quoique non obligatoire. La durée maximale du TP est de 1 heure.