



Préambule

Ce support de cours se présente sous la forme d'un guide pratique couvrant les grandes lignes des principes fondamentaux d'Angular, sous une forme synthétique, destiné à un cours de courte durée. En ce sens, il ne peut se substituer à la documentation officielle sur lequel il est basé : <https://angular.io/>

Le sujet traité ici est le framework « Angular », à ne pas confondre avec AngularJS qui est une librairie de vue dont le support s'est arrêté en janvier 2022.

Bien que ce support se base sur Angular 19, sorti officiellement en 2024, la plupart des principes évoquées dans ce document sont aussi valables pour les versions précédentes du framework (à partir de la version 2).

Certains aspects abordés relèvent de stratégies, politiques dont il existe de multiples variantes à adapter selon les cas. Ces-derniers seront donc présentés à titre d'exemple parmi les plus employés dans le monde Angular.

Table des matières

| | |
|---|----|
| Qu'est-ce que Angular ?..... | 6 |
| Le framework..... | 6 |
| Positionnement dans l'écosystème JS..... | 7 |
| Les technologies tierces mise en œuvre..... | 9 |
| NodeJS..... | 9 |
| NPM..... | 9 |
| TypeScript..... | 9 |
| RxJS..... | 9 |
| Angular CLI..... | 10 |
| Génération d'élément..... | 10 |
| Lancement de fonctionnalités..... | 10 |
| Gestion des dépendances..... | 11 |
| Les éléments de structure..... | 12 |
| Template interpolation..... | 12 |
| @if et *ngIf..... | 13 |
| @switch et NgSwitch..... | 14 |
| @for et *ngFor..... | 15 |
| <ng-container>..... | 17 |
| Les modules..... | 18 |
| Création d'un module..... | 18 |
| Morphologie des modules..... | 18 |
| Principaux paramètres d'un module..... | 19 |
| exports..... | 19 |
| imports..... | 19 |
| declarations..... | 19 |
| providers..... | 19 |
| bootstrap..... | 19 |
| Les composants..... | 20 |
| Création d'un composant..... | 20 |
| Principaux paramètres d'un module..... | 20 |
| selector..... | 20 |
| Property Binding (@input)..... | 21 |

| | |
|---|----|
| Event Binding (@output)..... | 22 |
| Two Way Binding..... | 23 |
| Attribute Binding..... | 24 |
| Évènements de cycle de vie..... | 25 |
| Les pipes..... | 26 |
| Utilisation..... | 26 |
| Création..... | 26 |
| Optimiser l’affichage à l’aide des pipes..... | 27 |
| Les directives..... | 28 |
| Création..... | 28 |
| Utilisation..... | 28 |
| Les signaux..... | 29 |
| Zoneless..... | 29 |
| Utilisation des signaux..... | 29 |
| Les « computed (valeurs calculées)..... | 30 |
| Les « effects »..... | 31 |
| Interaction avec les observables..... | 32 |
| Le routage..... | 33 |
| Configuration du routage..... | 33 |
| Les liens..... | 34 |
| Navigation via le code..... | 34 |
| Récupération des paramètres..... | 35 |
| Les guards..... | 36 |
| Lazy-loading..... | 37 |
| Le système d’injection (services)..... | 38 |
| Portée des services..... | 39 |
| Injection d’un service..... | 40 |
| Création d’un service..... | 41 |
| Tree-Shakable Services et providers..... | 41 |
| Le service HttpClient..... | 42 |
| Les formulaires..... | 43 |
| Template Driven Forms..... | 43 |
| Reactive Forms..... | 44 |
| Autres technologies à considérer..... | 46 |

| | |
|------------------------|----|
| Angular Materials..... | 46 |
| Jest..... | 46 |
| Volta..... | 46 |
| NgRx..... | 46 |
| ESLint..... | 46 |
| Prettier..... | 46 |
| NX..... | 47 |
| Compodoc..... | 47 |
| Electron..... | 47 |
| Angular DevTools..... | 47 |

Qu'est-ce que Angular ?

Le framework

Angular est une plateforme de développement destinée à créer des applications WEB de type Single Page Application ou PWA. Souvent comparé à des bibliothèques de vues, Angular est pourtant bien plus que ça car il renferme un framework complet comprenant notamment le routage, la gestion des formulaires, un système d'injection de dépendances, un client HTTP, un système de rendu SSR (rendu côté serveur), des outils de génération de code, etc.

Développé par Google sous licence Open Source, les publications de versions Angular avancent à un rythme soutenu (environ tous les 6 mois) offrant de nouvelles fonctionnalités et optimisations.

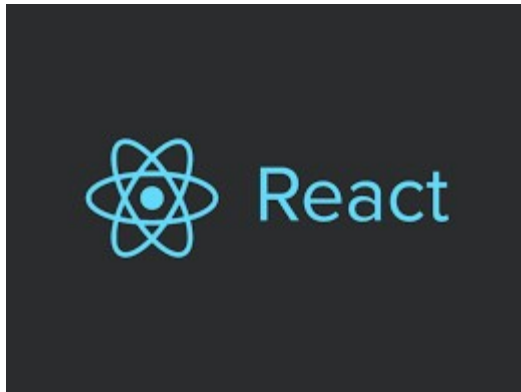


Historique des versions Angular, source calibrait.com

IMPORTANT : AngularJS a été entièrement refondu pour sa version 2 afin de correspondre à la philosophie que nous lui connaissons aujourd'hui. Afin de bien faire le distinguo avec l'ancienne librairie parue en 2010, la version 1 a été renommée « AngularJS ».

Positionnement dans l'écosystème JS

Angular est souvent mis en concurrence deux autre technologies JS qui sont React et VueJS.



- | | |
|---|--|
| <ul style="list-style-type: none">- Le plus populaire (avis développeurs, marché de l'emploi)- Développé par Facebook, License MIT- Forte communauté et contributeurs- Courbe d'apprentissage simple/moyenne | <ul style="list-style-type: none">- Outsider en prise de vitesse constante- A ce jour le plus rapide des trois (pour afficher les mêmes éléments)- Détient le plus grand nombre de projets- Courbe d'apprentissage la plus courte des trois |
|---|--|

A l'heure où sont rédigées ces lignes (fin 2025), ces deux produits constituent, avec Angular, le trio de tête des technologies frontend JS les plus reconnues (il en existe bien entendu d'autres qui mériteraient attention (Svelte par exemple) mais ce n'est pas l'objet de ce document). De manière générale, ils sont réputés pour être plus simple et plus léger qu'Angular.

Mais alors pourquoi choisir Angular ?

1. Angular est un framework complet là où React et VueJS sont des librairies de vues : afin de réaliser un SI, vous aurez donc toujours besoin de les coupler à d'autres librairies (Nuxt.js, Next.js, Remix, Axios ...) afin d'obtenir des fonctionnalités plus ou moins équivalentes à ce que peut proposer Angular. De plus, TypeScript, qui est embarqué nativement dans Angular, devient de plus en plus incontournable dans les projets récents. Ainsi, React et Vue, dans leurs versions récentes, prennent aussi en charge TypeScript. Avec ces éléments mis bout-à-bout, **la courbe d'apprentissage de ses deux libraires n'est plus aussi éloignée de celle d'Angular.**



Une fois associés à des librairies tierces, la courbe d'apprentissage de VueJS se rapproche de celle d'Angular

2. Comme évoqué ci-dessus, les piles utilisées avec Vue ou React peuvent être totalement différentes selon la composition de librairies tierces que vous utilisez. Ainsi, deux projets en VueJS (par exemple) peuvent être totalement différents : ils nécessiteront donc un temps d'adaptation, voire de formation. Angular est Angular ! Passer d'un projet à l'autre est relativement aisé car les fonctionnalités restent les mêmes. **On favorise ainsi l'interchangeabilité des développeurs et des équipes et on facilite les MCO/MCS.**
3. Angular vient avec une philosophie, une structure, une manière de faire. Tous ses outils sont spécifiquement conçus pour fonctionner ensemble et permettre une montée en puissance des projets de manière sobre et efficace. C'est pourquoi **Angular a la réputation d'être le plus « scalable » des trois.** Il est donc particulièrement recommandé pour les projets moyen/lourds.

IMPORTANT : Cela ne signifie en aucun cas que React et VueJS ne peuvent être utilisés sur de gros projets. Mais le fait d'être « non-opinionné » et d'avoir recours à plus de librairie tierce oblige à avoir une expertise avancée afin de bien structurer et configurer un projet sans tomber dans le syndrome « spaghetti ».

Les technologies tierces mise en œuvre



NodeJS

NodeJS est une plateforme logicielle qui permet d'exécuter du JavaScript. Sur la machine du développeur elle est indispensable à Angular CLI pour générer les projets, composants, services (etc.) mais aussi pour lancer l'environnement de test. En production, NodeJS permet d'utiliser la partie SSR d'Angular (rendu côté serveur, aussi appelé Angular Universal).



NPM

NPM est le gestionnaire de paquets JavaScript livré par défaut avec NodeJS. Il permet d'installer de nouveaux paquets (recensés sur le site officiel <http://npmjs.com>) et de les maintenir à jour facilement. Il peut bien entendu être remplacé par un autre gestionnaire comme « Yarn » ou « pnpm » selon les goûts des développeurs.



TypeScript

TypeScript est un langage de programmation libre qui se présente comme une « surcouche » au JavaScript standard afin de lui apporter un typage fort. Si cette technologie apporte une couche de complexité, la plupart des développements modernes intègre TypeScript pour la robustesse et la qualité qu'il apporte aux codes.



RxJS

RxJS est une bibliothèque de programmation réactive utilisant des Observables pour faciliter la composition de code asynchrone. Très riche, les fonctionnalités de cette librairie peuvent parfois rendre le code difficile à appréhender pour les non-initiés. Elle regorge pourtant d'outils puissants permettant de simplifier certaines tâches complexes dans la gestion des flux de données.

Angular CLI

Angular CLI est l'interface en ligne de commande d'Angular. Elle permet de réaliser diverses opérations comme la génération des composants, la compilation du projet ou le démarrage du serveur de développement. Voici une liste non exhaustive de commandes les plus utilisées :

Génération d'élément

```
ng new mon-premier-projet
```

Crée un nouveau projet Angular nommé « mon premier projet » à partir de 0.

```
ng g c (ou) ng g s (ou) ng g pipe (...)
```

Permet de générer un nouvel élément d'Angular : « c » pour composant, « s » pour service... En utilisant cette commande, Angular génère et préconfigure l'élément demandé. Pour le composant par exemple, un fichier TS sera géré pour le code, ainsi qu'un fichier SCSS pour les styles, un fichier « spec.ts » pour les tests unitaires... Le tout sera éventuellement déclaré dans le module parent.

Lancement de fonctionnalités

```
ng serve
```

Compile le projet et lance le serveur de développement. Par défaut celui-ci devient accessible à l'aide du navigateur à l'adresse <http://localhost:4200>.

```
ng build
```

Compile le projet et génère les fichiers prêts à être déployés sur un serveur (Apache par exemple). Par défaut, les livrables sont générés dans un dossier « dist » à la racine du projet.

```
ng lint
```

Permet de lancer le linter configuré par défaut (ESLint, SonarLint ou autre).

```
ng test
```

Lance les tests unitaires.

Gestion des dépendances

`ng update`

Met à jour les paquets Angular ainsi que les autres dépendances prises en charge. Cette commande est recommandée plutôt que « `npm update` » car elle permet d'apporter des modifications automatiquement au workspace si besoin.

`ng add`

Permet d'ajouter un paquet à votre projet. Si le paquet le prends en charge, cette commande est à favoriser au profit de « `npm install` » car elle intègre souvent des scripts qui configure automatiquement le workspace.

Il existe une multitude d'autres commandes et d'options qui sont détaillées dans la documentation officielle d'Angular <https://angular.io/cli>.

IMPORTANT : Les commandes d'Angular CLI peuvent être désactivées, ou masquées selon les technologies que vous utilisez sur votre projet. C'est le cas avec NX par exemple qui encapsule les fonctionnalités d'Angular CLI dans ses propres commandes préfixées par « `nx` ».

Les éléments de structure

Template interpolation

- Permet d'afficher la valeur d'une propriété du composant (cela peut aussi être un service injecté). La propriété doit être en modifier public (valeur par défaut).
- S'utilise à l'aide des doubles accolades {{ xxx }}
- Doit impérativement rester simple à évaluer pour le système : il faut éviter les traitements complexes sous peine d'avoir des lenteurs dans l'interface.
- Attention au type de la propriété.
- L'usage d'un « pipe » est souvent la solution pour afficher une donnée « complexe ».

header.component.ts

```
@Component({
  selector: 'jdm-header-component',
  templateUrl: './header.component.html',
})
export class HeaderComponent {
  welcomeText = 'Bienvenue sur votre application !';
}
```

header.comonent.html

```
<div>
  <p>{{ welcomeText }}</p>
</div>
```

@if et *ngIf

@if est l'élément conditionnel de base des vues Angular **depuis la version 17**. Il est voué à remplacer le ***ngIf** qui est l'opérateur historique. Celui-ci reste néanmoins disponible pour assurer la compatibilité ascendante. Bien que similaires, les deux éléments ont des fonctionnements légèrement différents.

Fonctionnement de @if

@if offre une écriture plus intuitive, facilitant la lecture du code et intégrant mieux le « else » que son prédécesseur. C'est donc la syntaxe à privilégier pour les nouveau développement (Angular >= 17).

header.component.html

```
@if (param1 > param2) {  
  <p>param1 est plus grand que param2 !</p>  
} @else if (param1 === param2) {  
  <p>param1 est égal à param2 !</p>  
} @else {  
  <p>param1 est inférieur à param2 !</p>  
}
```

*Fonctionnement de *ngIf*

Contrairement à son cadet, *ngIf se comporte comme un attribut et doit donc être obligatoirement associé à une balise HTML pour fonctionner.

app.component.html

```
<p *ngIf="param1 > param2">param1 est plus grand que param2 !</p>  
<p *ngIf="param1 === param2">param1 est égal à param2 !</p>  
<p *ngIf="param1 < param2">param1 est inférieur à param2 !</p>
```

Si cette syntaxe pourrait paraître plus concise de prime abord, elle est pourtant bien moins lisible car elle possède 3 conditions contre seulement 2 pour le **@if**.

Elle est, de plus, bien moins optimisée car, dans le cas où `param1 > param2`, la version n'évaluera que le premier prédicat **@if (param1 > param2)** et ignorera les deux autres, là où, dans le cas du ***ngIf** les prédicats seront systématiquement évalués.

*ngIf dispose aussi d'une mécanique lui permettant de prendre en charge les « else ». Néanmoins, cette méthode est très peu intuitive et peu lisible. Elle n'a donc jamais été appréciée par la communauté et reste peu employée.

@switch et NgSwitch

`@switch` permet d'afficher des blocs en fonction de la valeur d'une propriété. Disponible uniquement depuis la version 17, il est voué à remplacer le `*ngSwitch` qui est l'opérateur historique. Celui-ci reste néanmoins disponible pour assurer la compatibilité ascendante. Bien que similaires, les deux éléments ont des fonctionnements légèrement différents.

Fonctionnement de @switch

app.component.html

```
@switch (maVoiture.marque) {  
  @case ('Renault') {  
    La concession la plus proche est à 27km.  
  }  
  @case ('Peugot') {  
    La concession la plus proche est à 12km.  
  }  
  @default {  
    Aucune information disponible sur les concessions.  
  }  
}
```

Fonctionnement de NgSwitch

app.component.html

```
<div [ngSwitch]="maVoiture.marque">  
  <p *ngSwitchCase="'Renault'">La concession la plus proche est à 27km.</p>  
  <p *ngSwitchCase="'Peugot'">La concession la plus proche est à 12km.</p>  
  <p *ngSwitchDefault>Aucune information disponible sur les concessions.</p>  
</div>
```

Notons ici encore que l'ancienne formule, utilisant le `NgSwitch` souffre de contraintes sévères telles que son couplage obligatoire avec une balise HTML.

@for et *ngFor

@for est l'élément qui permet de boucler sur les éléments d'une collection (tableaux, etc.) depuis la version 17. Il est voué à remplacer le ***ngFor** qui est l'opérateur historique. Celui-ci reste néanmoins disponible pour assurer la compatibilité ascendante. Bien que similaires, les deux éléments ont des fonctionnements légèrement différents.

Fonctionnement du @for

app.component.ts

```
export class AppComponent {
  marquesVehicules = [
    {
      nom: 'Renault',
      lien: 'https://www.renault.fr/',
    },
    {
      nom: 'Peugeot',
      lien: 'https://www.peugeot.fr/',
    },
    {
      nom: 'Skoda',
      lien: 'https://www.skoda.fr/',
    },
    {
      nom: 'Tesla',
      lien: 'https://www.tesla.com/fr_fr',
    },
  ];
}
```

app.component.html

```
<ul>
  @for (marque of marquesVehicules; track marque.nom) {
    <li>{{ marque.nom }}</li>
  } @empty {
    <li>Aucune marque.</li>
  }
</ul>
```

Important : L'attribut « track » permet de spécifier la propriété qui permet de différencier chaque élément. S'il peut paraître anodin, il est pourtant un élément crucial car c'est grâce à cela qu'Angular déterminera s'il doit redessiner le composant ou pas. Cela jouera donc un rôle essentiel dans les performances d'affichage de l'application.

Fonctionnement du `*ngFor`

Contrairement à son cadet, `*ngFor` se comporte comme un attribut et doit donc être obligatoirement associé à une balise HTML pour fonctionner.

app.component.html

```
<ul>
  <li *ngFor="let marque of marquesVehicules">{{ marque.nom }}</li>
  <li *ngIf="marquesVehicules.length === 0">Aucune marque.</li>
</ul>
```

Encore une fois, si cette version peut paraître légèrement plus concise, elle souffre pourtant de nombreux défauts par rapport au `@for` :

- Elle est tributaire d'une balise HTML pour exister : ici c'est la balise « li ».
- Elle ne dispose pas d'un attribut « track » naturel. Elle possède bien une fonctionnalité équivalente « trackBy » mais bien complexe à exploiter car elle requiert une fonction dédiée.
- Elle ne prend pas en charge nativement le fait que la collection soit vide comme le fait `@if` avec la balise `@empty`. Dans cet exemple, pour y palier, nous avons dû y adjoindre un `*ngIf` qui complexifie le code et perd de son optimisation : le `@ngIf` sera évalué systématiquement contrairement à la balise `@empty`.
- Le tout reste moins lisible pour le développeur que la version avec `@for`.

<ng-container>

Le `<ng-container>` n'est pas un élément de structure à proprement parler mais il trouve sa place dans cette section car il est souvent utilisé avec ces derniers.

En effet, `<ng-container>` est un composant Angular basique qui s'utilise comme une balise HTML standard. Sauf qu'à la différence de celle-ci, `<ng-container>` n'existe plus du tout au moment du rendu de l'application. Cette mécanique s'avérerait donc particulièrement utile pour utiliser les éléments de structure ancienne génération (< Angular 17) : `*ngIf`, `*ngFor`...

Exemple sans <ng-container> :

app.component.html

```
<p>
  <span *ngIf="maVoiture.marque === 'Renault'">La concession la plus proche est à 27km.</span>
  <span *ngIf="maVoiture.marque === 'Peugot'">La concession la plus proche est à 12km.</span>
</p>
```

Code généré :

```
<p _ngcontent-ng-c3489358111>
  <span _ngcontent-ng-c3489358111>La concession la plus proche est à 27km.</span>
</p>
```

Notons qu'une balise `` est insérée uniquement pour héberger le `*ngIf`. Elle n'a aucun autre intérêt ici et complexifie inutilement le code.

Exemple avec <ng-container> :

app.component.html

```
<p>
  <ng-container *ngIf="maVoiture.marque === 'Renault'">La concession la plus proche est à 27km.</ng-container>
  <ng-container *ngIf="maVoiture.marque === 'Peugot'">La concession la plus proche est à 12km.</ng-container>
</p>
```

Code généré :

```
<p _ngcontent-ng-c3489358111>La concession la plus proche est à 27km.</p>
```

Le code généré est plus propre : aucune balise superflue n'a été nécessaire.

L'utilisation de `@ngIf` aurait donné exactement le même résultat mais avec un code plus lisible :

app.component.html

```
<p>
  @if (maVoiture.marque === 'Renault') {
    La concession la plus proche est à 27km.
  }
  @if (maVoiture.marque === 'Peugot') {
    La concession la plus proche est à 12km.
  }
</p>
```

Les modules

Les modules sont aux éléments Angular, ce que les dossiers sont aux fichiers : ils sont en quelque sorte des boîtes pouvant contenir toutes sortes de choses : des composants, des directives, des pipes, des services... Ils permettent ainsi d'organiser le projet en le découpant en plusieurs morceaux ce qui facilite la réutilisation. Mieux, ils seront automatiquement utilisés par Angular pour découper le projet en « chunks » utilisés pour le lazy-loading.

IMPORTANT : Il ne faut pas confondre les modules Angular avec les modules de JavaScript/TypeScript. Il est facile de faire l'amalgame car les deux sont souvent utilisés au sein d'un même projet mais ce sont bel et bien deux notions différentes.

Création d'un module

```
ng g m MonPremierModule
```

Morphologie des modules

Les modules se présentent sous la forme d'une simple classe muni d'un décorateur `@NgModule`.

```
import { CommonModule } from '@angular/common';
import { NgModule } from '@angular/core';
import { MatIconModule } from '@angular/material/icon';
import { SpinnerDialogComponent } from
'./components/dialogs/spinner-dialog/spinner-dialog.component';

@NgModule({
  imports: [CommonModule, MatIconModule],
  declarations: [SpinnerDialogComponent],
  exports: [SpinnerDialogComponent],
})
export class BaseModule {}
```

Le corps du module est vide la plupart du temps. En revanche le décorateur « NgModule » admet de multiples paramètres qui font office de configuration.

Principaux paramètres d'un module

exports

Définit la liste des éléments (composants, pipes...) pouvant être utilisés par les modules qui importeront celui-ci. A l'inverse, les éléments qui ne sont pas exportés ne seront utilisables que par les composants du module lui-même.

imports

Définit une liste de module à importer. Le contenu des modules de cette liste seront donc utilisables par tous les composants du module « importateur ».

declarations

Définit la liste des éléments contenus dans ce module. A noter qu'un même élément ne peut pas être déclaré dans plusieurs modules d'un même projet. Pour le réutiliser, il suffit de lui créer un module dédié.

providers

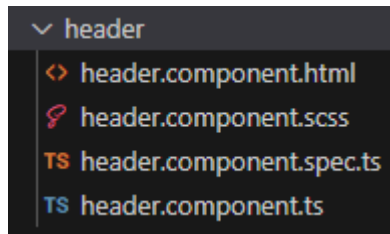
Définit une liste de providers (des services par exemple) mis à disposition de tous les éléments contenus dans le module.

bootstrap

Définit le composant qui sera utilisé comme racine : c'est le composant principal du module, il sera généré puis inséré dans la page index.html du projet.

Les composants

Le composant est l'élément visuel de base d'Angular. C'est un simple objet constitué de code TypeScript, de HTML et de SCSS. Le fichier « .spec.ts » est, pour sa part, le fichier contenant les test unitaires du composant.



Structure des fichiers d'un composant

```
@Component({
  selector: 'jdm-footer-component',
  templateUrl: './footer.component.html',
  styleUrls: ['./footer.component.scss'],
})
export class FooterComponent {
  @Input() urlMarianne!: string;
  versionCfg = versionConfig;

  onVersionClick() {
    console.log(this.versionCfg);
  }
}
```

Exemple d'implémentation d'un composant

Création d'un composant

```
ng g c components/MonPremierComposant
```

Principaux paramètres d'un module

Comme les modules, la partie TS des composants se présente sous la forme d'une simple classe muni d'un décorateur `@Component`. Ce décorateur admet divers paramètres permettant de « configurer » le composant. Voici les paramètres les plus fréquents :

selector

Définit le nom de balise qui sera associé à ce composant.

templateUrl

Donne le chemin vers le fichier de code HTML du composant.

styleUrls

Donne une liste de chemin vers des fichiers de style (CSS, SCSS...) pour le composant.

standalone

Définit si le composant doit être traité comme un module ou non. Si oui, le composant doit être importé dans les autres modules (ou composants standalone) pour être utilisé (comportement par défaut). Sinon il doit être déclaré dans un module.

imports

Utilisable que si le composant est défini comme standalone. Si c'est le cas, le composant ne dépendant

pas d'un module, il aura souvent besoin d'importer lui-même les ressources qu'il exploite (composants visuels, etc.). Ce paramètre permet de réaliser ces imports, comme le ferait un module.

changeDetection

Permet de spécifier la stratégie de détection des modifications qui conditionne le déclenchement du « re-dessin » de la vue.

encapsulation

Spécifie la stratégie d'encapsulation des styles du composant. Attention, le changement de cette valeur peut facilement provoquer des effets de bord : par exemple, `ViewEncapsulation.None` propage les styles du composant à tous les éléments du projet.

Property Binding (@input)

Le property binding est une mécanique permettant de passer des variables à un composant. Son utilisation s'inspire des propriétés des éléments DOM. Il s'implémente à l'aide du décorateur `@input`.

footer.component.ts

```
@Component({
  selector: 'jdm-footer-component',
  templateUrl: './footer.component.html',
  styleUrls: ['./footer.component.scss'],
})
export class FooterComponent {
  /**
   * Ceci est un property binding !
   */
  @Input() urlMarianne!: string;
}
```

app.component.html

```
<jdm-footer-component
  [urlMarianne]="https://www.gouvernement.fr/sites/default/files/chartegraphique_1000_marianne.gif"
></jdm-footer-component>
```

Grace à cela, le composant FooterComponent pourra être utilisé à plusieurs endroits, tout en personnalisant à chaque fois l'image de la Marianne utilisée dans le rendu.

Attention : Lorsqu'il est nécessaire de partager des valeurs entre de multiples sous-composants, il peut être tentant de multiplier les `@Input` afin de les transmettre. Néanmoins, lorsque des composants ont un rapport parent/enfant, il est souvent plus avantageux de tirer parti des services.

[Voir la section « Le système d'injection ».](#)

Event Binding (@output)

Si le property bindings permet de fournir des données à un composant, celui-ci est aussi capable d'envoyer. Cela s'articule dans un système d'évènement appelé « event binding » exploitable à l'aide du décorateur `@output`.

Imaginons deux boutons dans un composant « footer ».

footer.component.html

```
<button type="button" (click)="onButtonClick('bonjour')">Bonjour !</button>
<button type="button" (click)="onButtonClick('au revoir')">Au revoir !</button>
```

footer.component.ts

```
onButtonClick(buttonName: string): void {
  console.log(`Nom du bouton cliqué : ${buttonName}`);
}
```

La balise HTML `<button>` possède un événement natif nommé `click` que nous connectons à la méthode `onButtonClick` grâce à l'évènement binding d'Angular en lui passant une chaîne comme argument.

Notons que plusieurs événements peuvent être « bindés » à la même méthode du moment que ceux-ci ont les mêmes types d'arguments.

Dans le composant parent du footer (qui sera dans notre cas AppComponent) nous souhaitons réagir à cet événement et en récupérer la valeur. Nous définissons donc un `@Output` dans notre footer afin de pouvoir émettre cet événement.

footer.component.ts

```
export class FooterComponent {
  /**
   * Un bouton a été cliqué !
   */
  @Output() buttonClicked = new EventEmitter<string>();

  onButtonClick(buttonName: string): void {
    this.buttonClicked.emit(buttonName);
  }
}
```

Enfin nous connectons l'évènement dans le composant parent :

app.component.html

```
<jdm-footer-component (buttonClicked)="methodeALancer($event)"></jdm-footer-component>
```

app.component.ts

```
methodeALancer(nomDuBouton: string) {
  console.log(nomDuBouton);
}
```

Two Way Binding

Le two way binding (twb) permet de combiner le property binding et l'évent binding pour une même propriété. Comme la fonctionnalité se base sur les mécaniques déjà détaillées si dessus, nous allons simplement en énoncer les principes :

- Pour qu'une propriété soit en twb il faut qu'il ait une propriété `@Input` ainsi qu'une propriété `@Output` qui soient liés entre-elles.
- La propriété `@Output` doit porter le même nom que la propriété `@Input` mais avec le suffixe « Change » : `size` -> `sizeChange`
- Au besoin, une propriété en twb peut être utilisée simplement comme un `@Input` (en utilisant les crochets), comme un `@Output` (en utilisant les parenthèses), ou les deux en même temps.
- Pour exploiter une propriété en twb dans les deux sens, le composant parent utilisera la notation `[(size)] = "xxxx"`.

La syntaxe `[(xxx)]` peut être difficile à mémoriser. Un des membres de l'équipe Angular proposa un moyen mnémotechnique relativement efficace consistant à imaginer une boîte contenant un hot-dog. Les crochets représenteraient alors la boîte et les parenthèses seraient le pain.

Attribute Binding

Angular permet de gérer les attributs dynamiquement de manière très simple, en adoptant une mécanique similaire au property binding.

App.component.html

```
<table>
  <tr>
    <th [attr.colspan]="marquesVehicules.length">Liste des marques</th>
  </tr>
  <tr>
    @for (marque of marquesVehicules; track marque.nom) {
      <th>{{ marque.nom }}</th>
    }
  </tr>
  <tr>
    @for (marque of marquesVehicules; track marque.nom) {
      <td>{{ marque.lien }}</td>
    }
  </tr>
</table>
```

Tableau généré

| Liste des marques | | |
|---|---|---|
| Renault | Skoda | Tesla |
| https://www.renault.fr/ | https://www.skoda.fr/ | https://www.tesla.com/fr_fr |

Ici, l'attribut HTML **colspan** est défini par Angular selon le nombre d'éléments dans le tableau **marquesVehicules**. Si ce nombre change, les modifications seront automatiquement répercutées dans la vue.

Cette mécanique est exploitée par Angular pour gérer dynamiquement les classes...

```
<div [class.conflict]="isConflict">Cet élément est en conflit !</div>
```

...mais aussi les styles CSS.

app.component.html

```
<nav [style.background-color]="bgColor"></nav>
```

Si la valeur fournie est considérée comme « falsy » (false, 0, NULL, undefined), l'attribut sera tout simplement supprimé par Angular.

Évènements de cycle de vie

Le cycle de vie des composants Angular est une mécanique complexe qui ne pourra pas être détaillé sur support. Il existe en revanche deux évènements très importants dans la vie d'un composant. Il s'agit du `OnInit`, lors de son initialisation et du `OnDestroy` lors sa destruction.

Ces deux évènements sont extrêmement utilisés pour préparer les données à afficher par exemple, ou pour nettoyer des ressources, telles que les souscriptions à des observables.

dashboard.component.ts

```
export class DashboardComponent implements OnInit, OnDestroy {
  notificationSub!: Subscription;

  constructor(private readonly notifSrv: NotificationService) {}

  ngOnInit(): void {
    this.notificationSub = this.notifSrv.notification$.subscribe((notif) => {
      alert(notif);
    });
  }

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

Dans cet exemple, le composant s'abonne aux notifications du service `NotificationService` lors de son initialisation. Pour cela, la classe doit implémenter l'interface `OnInit` en implémentant une méthode `ngOnInit` qu'Angular exploitera automatiquement.

Idem, nous implémentons l'interface `OnDestroy` en implémentant une méthode `ngOnDestroy` afin de résilier l'abonnement aux notifications.

Important : Cette pratique est très courante dans Angular car les abonnements ne sont pas automatiquement nettoyés. Cela a pour effet de cumuler les souscriptions et de déclencher des traitements indésirables, provoquant ainsi effets de bord et pertes de performances.

PENSEZ A NETTOYER VOS SOUSCRIPTION !!!

Les pipes

Les pipes sont des éléments conçus pour transformer une valeur en vue de son affichage dans un composant. Plusieurs pipes sont livrés avec Angular out-of-the-box (async, date, uppercase, lowercase, currency, etc.). Tous les pipes sont chainables afin d'en combiner les résultats.

Utilisation

app.component.ts

```
<!-- Affichage d'une date sans pipe -->
<p>{{ today }}</p>
<!-- Résultat : Fri Feb 02 2024 09:13:41 GMT+0100 (heure normale d'Europe centrale) -->

<!-- Affichage d'une date avec le pipe "date" d'Angular -->
<p>{{ today | date }}</p>
<!-- Résultat : 2 févr. 2024 -->

<!-- Combinaison de deux pipes -->
<p>{{ today | date | uppercase }}</p>
<!-- Résultat : 2 FÉVR. 2024 -->

<!-- Utilisation d'un paramètre de pipe (ici le format d'affichage de la date) -->
<p>{{ today | date: 'dd/MM/yyyy' }}</p>
<!-- Résultat : 02/02/2024 -->
```

Création

ng g p pipes/UserWithTrigram

```
@Pipe({
  name: 'userWithTrigram',
})
export class UserWithTrigramPipe implements PipeTransform {
  constructor(private readonly userSrv: UserService, private readonly upperPipe: UpperCasePipe){}

  transform(user: User): string {
    const lastName = this.upperPipe.transform(user.lastName);
    const trigram = this.userSrv.computeTrigram(user);
    return `${lastName} ${user.firstName} (${trigram})`;
  }
}
```

Cet exemple nous avons créé un pipe permettant d'afficher un utilisateur avec son nom, prénom et trigramme. Il est intéressant de noter que pour transformer la valeur de l'objet passé en paramètre, une autre pipe et un service tiers a été injecté en toute simplicité (cf. injection de service).

Optimiser l’affichage à l’aide des pipes

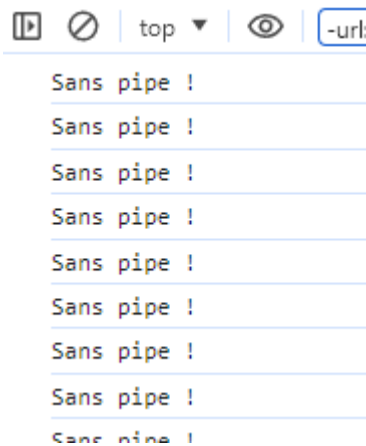
Si les pipes sont extrêmement utiles pour la mutualisation du code, ils sont un atout précieux pour l’optimisation des vues. Pour illustrer cela, nous allons utiliser l’exemple précédent et le comparer à la version sans pipe. Pour voir l’impact sur les traitements opérés par Angular, nous rajoutons juste un `console.log` dans le code de la méthode `computeTrigram`.

Sans Pipe

app.component.html

```
<p>
  {{ user.lastName }}
  {{ user.firstName }}
  ({{ userSrv.computeTrigram(user) }})
</p>
```

Sortie console

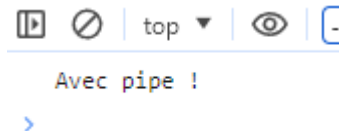


Avec pipe

app.component.html

```
<p>{{ currentUser | userWithTrigram }}</p>
```

Sortie console



Si le rendu HTML semble identique à l’utilisateur, le fonctionnement interne du rendu, lui, **est bien différent**. En effet, à chaque événement du navigateur (clic, déplacement de la souris, scroll, etc.), Angular doit déterminer les éléments à redessiner. Or, celui-ci n’a aucun moyen de savoir si la valeur retournée par la méthode `computeTrigram` a changé : il doit donc fatalement enchaîner les appels à cette fonction pour vérifier.

Cela peut rapidement avoir un impact non négligeable, voir catastrophique sur les performances du rendu si la méthode est complexe ou si elle est appelée de multiples fois dans la vue (comme dans une liste par exemple).

Avec le pipe en revanche, l’appel est bien réalisé qu’une seule fois car, par défaut un pipe est dit « pure ». Cela signifie que la valeur qu’il retourne ne change pas si la valeur de la variable qu’on lui transmet n’a pas changé non plus (ou la référence). Ainsi, Angular n’appellera la méthode `computeTrigram` que si la valeur de « user » change.

Il est possible de créer de pipes « impures » mais ceux-ci ne présentent que très peu d'intérêt.

Les directives

Les directives sont des éléments permettant de modifier le comportement de l'élément auquel il est associé (composant, simple élément DOM...). Visuellement il est similaire à un simple attribut HTML.

Création

```
ng g d directives/Highlight
```

Une directive est une simple classe possédant le décorateur `@Directive`.

```
highlight.directive.ts
@Directive({
  selector: '[appHighlight]',
  standalone: true,
})
export class HighlightDirective {}
```

Utilisation

Une des utilisations courantes des directives est de manipuler directement le DOM. Comme les composants, les directives doivent être déclarés ou importés dans les modules Angular pour être utilisés. Voici deux exemples simples d'application, avec ou sans valeur associée :

Sans valeur :

highlight.directive.ts

```
@Directive({
  selector: '[appHighlight]',
  standalone: true,
})
export class HighlightDirective implements OnInit {
  constructor(private _elementRef: ElementRef) {}

  ngOnInit() {
    this._elementRef.nativeElement.style.backgroundColor = 'red';
  }
}
```

app.component.html

```
<p>Ligne sans fond.</p>
<p appHighlight>Ligne à un fond rouge.</p>
```

Avec valeur :

highlight.directive.ts

```
@Directive({
  selector: '[appHighlight]',
  standalone: true,
})
export class HighlightDirective implements OnInit {
  @Input() appHighlight: 'red' | 'yellow' = 'red';
  constructor(private _elementRef: ElementRef) {}

  ngOnInit() {
    this._elementRef.nativeElement.style.backgroundColor = this.appHighlight;
  }
}
```

app.component.html

```
<p>Ligne sans fond.</p>
<p appHighlight="red">Ligne à un fond rouge.</p>
<p appHighlight="yellow">Ligne à un fond jaune.</p>
```

Les utilisations des directives peuvent être très variées, elles peuvent aussi, par exemple, implémenter des handlers pour des événements en utilisant `@HostListener` (voir documentation officielle <https://angular.io/guide/attribute-directives#handling-user-events>).

Les signaux

Historiquement, Angular embarquait systématiquement la librairie ZoneJS pour la gestion des événements du navigateur. Avec les nouvelles normes JS et leur adoption massive par les navigateurs, Angular a mis en place un système permettant de se passer de cette librairie, augmentant ainsi les performances du rendu tout en diminuant la taille du livrable : ainsi furent créés les « signaux ».

Zoneless

Bien que stable, le mode « zoneless » (sans ZoneJS) demeure, en Angular 19, encore en phase expérimentale (celle-ci devrait être probablement la norme en Angular 20 prévue pour le mai 2025). Ainsi, pour l'utiliser il faut l'ajouter manuellement aux providers.

app.config.ts

```
import { ApplicationConfig, provideExperimentalZonelessChangeDetection } from '@angular/core';

export const appConfig: ApplicationConfig = {
  providers: [provideExperimentalZonelessChangeDetection()],
};
```

Utilisation des signaux

Un signal est une variable qui informe le moteur de rendu lorsque sa valeur change. On le crée en utilisant la fonction `signal()`, puis on l'exploite avec les méthodes `get()` et `set()`.

app.component.ts

```
readonly acteurA = signal({
  nom: 'Thurman',
  prenom: 'Uma',
  derniereApparition: 2019,
});

ngOnInit(): void {
  setTimeout(() => {
    this.acteurA.set({
      nom: 'Hanks',
      prenom: 'Tom',
      derniereApparition: 2022,
    });
  }, 5000);
}
```

app.component.html

```
{{ acteurA().prenom }} {{ acteurA().nom }}
<small>{{ acteurA().derniereApparition }}</small>
```

Ce composant affichera « Uma Thurman 2019 » pendant 5s avant d'afficher «Tom Hanks 2022».

Les « computed (valeurs calculées)

Les « computed » sont des signaux particulier sous forme de variables calculées, capables de réagir automatiquement aux changement de valeur des signaux auxquels ils font référence.

Complétons l'exemple précédent :

app.component.ts

```
readonly acteurA = signal({
  nom: 'Thurman',
  prenom: 'Uma',
  derniereApparition: 2019,
});

readonly depuisA = computed(() => {
  return (new Date().getFullYear() - this.acteurA().derniereApparition);
})

ngOnInit(): void {
  setTimeout(() => {
    this.acteurA.set({
      nom: 'Hanks',
      prenom: 'Tom',
      derniereApparition: 2022,
    });
  }, 5000);
}
```

app.component.html

```
{{ acteurA().prenom }} {{ acteurA().nom }}
<small>il y a {{ depuisA() }} an(s)</small>
```

La valeur calculée « depuisA » fait référence au signal « acteurA ». Ainsi, lorsque la valeur de ce-dernier sera modifiée, la valeur du « computed » sera recalculée et émettra à son tour. Le moteur de rendu sera alors automatiquement informé qu'il doit redessiner son affichage.

Une valeur calculée ne peut pas être affectée. Sa valeur est automatiquement calculée à partir de la méthode fournie à sa création.

Un « computed » peut faire référence à un autre « computed » permettant ainsi de réaliser un chaînage optimisé pour l'affichage.

Les « effects »

Les « effects » permettent de réagir à l'émission d'un signal (et donc d'un « computed »). Ils sont à ce titre assez similaires aux « computed » sauf qu'ils n'émettent pas de signaux à leur tour.

Rajoutons un « effect » à l'exemple précédent :

app.component.ts

```
readonly acteurA = signal({
  nom: 'Thurman',
  prenom: 'Uma',
  derniereApparition: 2019,
});

readonly depuisA = computed(() => {
  return new Date().getFullYear() - this.acteurA().derniereApparition;
});

ngOnInit(): void {
  effect(() => {
    console.log(`Valeur modifiée : ${this.depuisA()}`);
  });

  setTimeout(() => {
    this.acteurA.set({
      nom: 'Hanks',
      prenom: 'Tom',
      derniereApparition: 2022,
    });
  }, 5000);
}
```

Ici l'« effect » écrira dans la console à chaque modification de la valeur du computed « depuisA » et donc, par extension du signal « acteurA ».

Dans les faits, la fonction `effect()` possède une valeur de retour. Mais dans cet exemple (comme la plupart du temps) cette valeur est rarement utilisée.

Important : Le système de signaux est avant tout conçu pour optimiser l'affichage. À ce titre, les « effects » ne doivent être utilisés que pour des opérations simples. Il est notamment fortement déconseillé de modifier la valeur d'un signal dans un « effects » sous peine de créer une boucle infinie.

Si toutefois vous devez réaliser des opérations complexes (asynchrones par exemple), la fonction « effect » admet un argument « `onCleanup` » qui permet de spécifier une méthode de nettoyage afin d'éviter les fuites mémoires.

Interaction avec les observables

Les signaux ne sont pas conçus pour gérer les fonctions asynchrones. Toutefois, il est possible de transformer un observable en signal (et inversement) ce qui permet de faciliter grandement et d'optimiser l'exploitation de l'asynchronisme de JS.

app.component.ts

```
// Service exploitant le backend
apiSrv = inject(ApiService);

// Retourne True si le backend répond, sinon False
healthCheck = toSignal(this.apiSrv.healthCheck());
```

app.component.html

```
Etat du backend : {{ healthCheck() ? 'ok' : 'ko' }}
```

Ici, la méthode « healthCheck » du service « ApiService » renvoie un observable. Plutôt que de devoir souscrire à celui-ci (et nettoyer la souscription), il suffit d'utiliser la fonction « toSignal ». Ainsi le signal « healthCheck » est directement exploitable par le moteur de rendu qui sera réactualiser à chaque émission.

Le cycle de vie des signaux sont alignés sur celui des composants. Ainsi, lorsque le composant est détruit, tous les signaux qu'il contient sont aussi nettoyés. C'est vraiment un gros avantage en comparaison des observables qui, eux, nécessite plus de code pour le même résultat.

La méthode « toObservable » permet de faire le chemin inverse en transformant un signal en observable. Il est ainsi possible, par exemple, d'exploiter les possibilités d'RxJS avec les signaux.

Important : Si ces méthodes peuvent rendre bien des services, il convient de toujours garder à l'esprit que les signaux sont avant tout conçus pour optimiser l'affichage.

Les équipes d'Angular conseillent de rester sur le système d'observables pour la gestion de l'asynchronisme dans le code métier (services etc.) et d'exploiter les signaux dans les vues.

Le routage

Configuration du routage

Avant tout, il convient de définir une liste de route pour notre application :

```
export const appRoutes: Route[] = [
  {
    path: 'accueil',
    component: HomeComponent,
  },
  {
    path: 'contact',
    component: ContactComponent,
  },
  {
    path: 'annonce/:annonceId',
    component: AdComponent,
  },
  {
    path: '**',
    redirectTo: 'accueil',
  },
];
```

- **path** : C'est le chemin pour accéder à la route depuis la racine de l'application. Les portions de route préfixés par deux points comme ici **:annonceId** représentent des paramètres que nous pourrions exploiter dans le composant de destination. Ce paramètre utilise la syntaxe WildCard pour résoudre les routes dans l'ordre. Ainsi, la route **'**'** intercepte toutes les routes qui ne correspondent à aucune des entrées spécifiées.

- **component** : C'est le composant qui sera affiché lorsque la route correspondante sera activée.

- **redirectTo** : Permet de rediriger une route vers une autre, évitant ainsi des duplications.

Il faut maintenant de fournir ces routes au **RouterModule** d'Angular. L'usage veut que l'on réalise cela dans un module dédié. C'est ce que fera Angular à l'initialisation du projet si vous vous demandez l'initialisation du routing.

app-routing.module.ts

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

Ce module ainsi créé devra être directement importé dans le module principal de votre application. Il suffit enfin d'afficher notre router dans l'application à l'aide de la balise **<router-outlet>**.

app.component.html

```
<main>
  <header>Mon header</header>
  <router-outlet></router-outlet>
  <footer>Mon footer</footer>
</main>
```

Important : Comme c'est Angular qui se chargera du routing, le serveur doit configurer la règle « rewrite » afin que toutes les URL renvoient vers la page index.html. Cela peut se faire par un simple fichier .htaccess par exemple.

Les liens

Afficher un lien qui pointe vers une route définie plus haut se fait très simplement à l'aide de l'attribut `routerLink`.

app.component.html

```
<!-- Lien simple vers une route statique -->
<a routerLink="/contact">Contact</a>

<!-- Lien sur un bouton avec une route dynamique -->
<button type="button" [routerLink]="['/annonce', idAnnonce]">Annonce n°{{ idAnnonce }}</button>

<!-- Lien sur un <span> avec une route dynamique (pas très joli !) -->
<span [routerLink]="'/annonce/' + idAnnonce2">Annonce n°{{ idAnnonce2 }}</span>
```

Comme tous les attributs Angular, nous pouvons utiliser le property binding (voir plus haut dans ce support) pour affecter des valeurs à l'attribut `routerLink`.

Navigation via le code

Afin de déclencher la navigation directement à partir du code TypeScript, suite à la validation d'un formulaire par exemple, Angular met à disposition un service `Router` qui admet une méthode `navigate`.

annonce-form.component.ts

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

onSubmitForm(idAnnonce: string) {
  // Ici le code de validation de formulaire par exemple...
  this.router.navigate(['/annonce', idAnnonce]);
}
```

Important : La méthode `navigate` est asynchrone. Pensez à utiliser `await` si vous souhaitez exploiter sa valeur de retour !

Récupération des paramètres

Lors des exemples précédents, nous avons défini une route `'annonce / :annonceId'` qui admet donc un paramètre `:annonceId`. Ce paramètre peut être récupéré à l'aide du service `ActivatedRoute`.

```
annonce.component.ts
export class AnnonceComponent implements OnInit, OnDestroy {
  paramMapSub!: Subscription;

  constructor(private readonly activatedRoute: ActivatedRoute) {}

  ngOnInit(): void {
    this.paramMapSub = this.activatedRoute.paramMap.subscribe((paramMap) => {
      console.log(paramMap.get('annonceId'));
    });
  }

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

Notez que la récupération des paramètres est asynchrone il faut donc y souscrire pour en récupérer la valeur.

Il existe une multitude de solutions pour récupérer la valeur des paramètres (en exploitant la puissance de la librairie RxJS ou bien en utilisant les resolvers, etc.) Il s'agit ici de donner un exemple simple à comprendre, répandu et efficace.

Les guards

Les « guards » sont des méthodes permettant d'autoriser l'accès à une route (`canActivate`) ou le départ depuis celle-ci (`canDeactivate`). Leur implémentation prend tout simplement la forme d'une fonction qui retourne `true` ou `false`.

Attention : S'ils s'y apparentent, les guards ne sont pas un mécanisme de sécurité ! Un frontend JS quel qu'il soit laisse libre accès à son code pour être exécuté sur la machine Client. Il faut donc TOUJOURS le considérer comme potentiellement compromis, contrairement au backend, qui a contrario, doit être solidement sécurisé pour anticiper d'éventuelles attaques.

Les guards ont donc juste vocation à améliorer l'ergonomie de l'application, rien de plus.

Exemple d'implémentation d'un guard :

is-current-user.guard.ts

```
export const isCurrentUserGuard : CanActivateFn = (route: ActivatedRouteSnapshot) => {  
  return inject(UserService).currentUser.id === route.params['userId'];  
};
```

Puis utilisation dans une nouvelle route :

app-routing.module.ts

```
{  
  path: 'profil/:userId',  
  component: ProfilComponent,  
  canActivate: [isCurrentUserGuard],  
},
```

Ici nous avons donc défini un guard, qui empêche l'accès à un profil si celui-ci n'est pas celui de l'utilisateur connecté.

Il existe plusieurs types de guard disponibles avec Angular consultables dans la documentation officielle du Router.

Lazy-loading

Les routes définies précédemment impliquent que tous les modules concernés par les routes sont importés en masse dès le chargement de l'application. A mesure que l'application grossit, la taille du paquet initial chargé par le navigateur peu devenir conséquent et ralentir dramatiquement le chargement de l'application (on parle de budget).

Afin de palier à cela, il est conseillé de ne charger que le stricte nécessaire au chargement de l'application, comme la page d'accueil par exemple, et de définir les autres routes en chargement différé : c'est le principe du lazy-loading !

Angular est capable naturellement, via son router, de différer le chargement de modules, et donc des composant.

Pour cet exemple, nous allons utiliser le composant `ProfilComponent` que nous allons faire passer en `standalone`. Comme vu précédemment, cela aura pour effet de le traiter comme un module (cf. chapitre Composant).

```
@Component({
  standalone: true,
  selector: 'app-profil',
  templateUrl: './profil.component.html',
  styleUrls: ['./profil.component.scss'],
})
export class ProfilComponent {}
```

Nous allons maintenant transformer notre ancienne route :

```
{
  path: 'profil/:userId',
  component: ProfilComponent,
  canActivate: [isCurrentUserGuard],
},
```

En route chargée de manière différée grâce au lazy-loading :

```
{
  path: 'profil/:userId',
  loadComponent: () => import('./components/profil/profil.component').then((mod) => mod.ProfilComponent),
  canActivate: [isCurrentUserGuard],
},
```

Le système d'injection (services)

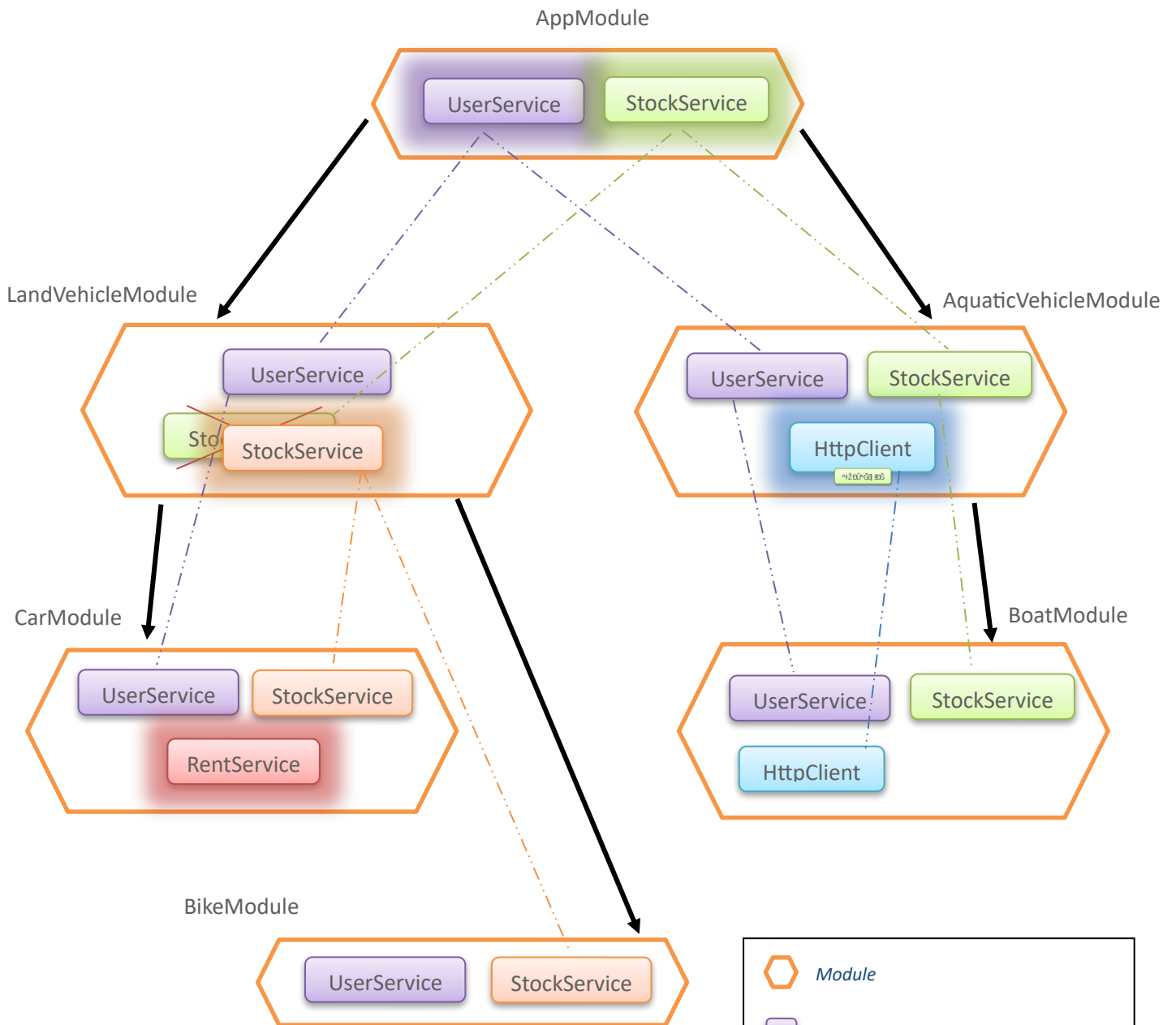
Angular intègre out-of-the-box un système d'injection qui permet de charger les dépendances de manière dynamique. C'est le principe d'inversion de contrôle que l'on retrouve dans de nombreux frameworks modernes.

Concepts généraux :

- Le mécanisme permettant d'injecter des services se nomme « **Injector** ».
- Chaque module possède son propre Injector (y compris les composants « standalone »). L'injecteur principal de l'application s'appelle le « **Root Injector** ». Les services qu'il fournit sont donc disponibles de partout dans l'application (sauf s'il sont réinjecté dans un autre module).
- Un « **provider** » est un élément qui permet d'instancier un service. Il est donc possible, pour chaque module, de définir une liste de providers afin de lui fournir des services.
- Un module importé reçoit automatiquement les services fournis par le module parent. On parle donc d'un **arbre d'injection**.
- Un « provider » peut être **identifié** par un nom de classe (par défaut), une clé sous forme de chaîne de caractère, un jeton, etc.
- La dépendance instanciée par un provider peut être une classe, une valeur (un objet, un tableau, etc.) ou même une factory.
- Si un service est « providé » dans un module, alors que celui-ci est déjà « providé » dans un de ses modules parents, alors une nouvelle instance de ce service est créée : attention aux effets de bords que cela peut engendrer ! (Cf. exemple ci-dessous)

Important : Il ne faut surtout **pas confondre** les mécaniques d'import/export des composants Angular et celles de l'injection de services.

Portée des services



Ce schéma décrit l'arbre d'injection d'une petite application :

- AppModule est le Root Injector, sa liste de providers sont constitués de UserService et StockService.
- AppModule importe LandVehicleModule qui hérite donc de ses deux services. Néanmoins, LandVehicleService injecte son propre StockService. Ainsi, le StockService de AppModule et LandVehicleModule ne sont plus du tous les mêmes : attention, si ces services contiennent des variables d'instance (la valeur du stock par exemple), ces valeurs seront donc différentes.
- LandVehicleModule importe CarModule qui hérite de ses deux services puis il en fournit un nouveau qui est le RentService. Il n'a pas accès au StockService de l'AppModule mais uniquement de celui de LandVehicleModule.
- LandVehicleModule importe BikeModule qui, lui, ne fournit pas de service supplémentaire. Il n'a pas accès au StockService de l'AppModule mais uniquement de celui de LandVehicleModule.
- Enfin, AppModule importe AquaticVehicleModule qui fournit un HttpClient qu'il transmettra au BoatModule.

Injection d'un service

app-component.ts

```
@Component({
  selector: 'app-home',
  templateUrl: './home.component.html',
  styleUrls: ['./home.component.scss'],
})
export class HomeComponent {
  constructor(private httpClient: HttpClient) {}
}
```

Ici, vous avons injecté le service `HttpClient` fourni par Angular dans le composant `home`. Dès lors, il sera disponible dans tout le composant en utilisant `this.httpClient`.

Création d'un service

```
ng g s services/User
```

Les services sont de simples classes possédant le décorateur `@Injectable`.

user.service.ts

```
@Injectable({
  providedIn: 'root',
})
export class UserService {
  /**
   * Connexion
   */
  login(): boolean {
    // (...code permettant de se connecter)
    return true;
  }
}
```

Tree-Shakable Services et providers

`providedIn` est un argument du décorateur `@Injectable` permettant de définir la portée du service. Généralement utilisé avec la valeur `root`, cela permet de spécifier à Angular que le service est directement fourni à l'injecteur racine, sans avoir besoin de le déclarer dans celui-ci.

Le gros avantage de l'argument `providedIn : 'root'` est réside dans le fait que celui-ci ne sera importé uniquement lors de sa première utilisation. S'il n'est pas utilisé du tout, celui-ci sera complètement supprimé de l'application lors de la compilation : c'est le fameux « Tree Shaking » d'Angular.

Dans le cas où `@Injectable` est utilisé sans argument, celui-ci devra être déclaré explicitement dans les providers d'un module pour pouvoir être injecté. C'est l'option qui serait choisie pour le « StockService » de l'exemple précédent car celui-ci doit pouvoir être injecté à la racine mais aussi directement dans le module « LandVehicleModule ».

Le service HttpClient

Le service HttpClient est l'un des services fournis « out-of-the-box » par Angular. Il permet, comme son nom l'indique, d'effectuer des appels HTTP à n'importe quel API. Il est donc particulièrement utile pour communiquer avec un backend en REST.

Comme tous les services, avant de l'utiliser il faut le « provider ». Cela se fait généralement dans le module principal d'Angular.

app.config.ts

```
export const appConfig: ApplicationConfig = {
  providers: [
    provideRouter(routes, withComponentInputBinding()),
    provideClientHydration(),
    provideAnimationsAsync(),
    provideHttpClient(withInterceptors([authInterceptor]))
  ],
};
```

Dans l'exemple ci-dessus, le HttpClient est fourni avec un intercepteur. Ce sont de simples fonction qui interceptent la requête http avant qu'elle soit exécutée afin d'y apporter des modifications. On peut ainsi, y ajouter à la volée des éléments tels que des headers.

Cela est très intéressant notamment dans le cadre de l'utilisation de JWT afin de transmettre le token de session afin de ne pas répéter cette opération dans tout le projet, à chaque utilisation du HttpClient.

auth.interceptor.ts

```
export const authInterceptor: HttpInterceptorFn = (req, next) => {
  const authSrv = inject(AuthService);

  const headers: Record<string, string> = { 'X-Parse-Application-Id': 'cours_angular' };

  if (authSrv.isLoggedIn) {
    headers['X-Parse-Session-Token'] = authSrv.currentSession.sessionToken as string;
  }

  // Ajout du token dans les entêtes de la requête
  const authReq = req.clone({
    setHeaders: headers,
  });

  // Envoi de la requête avec les nouvelles entêtes
  return next(authReq);
};
```

L'intercepteur ci-dessus ajoute systématiquement un header 'X-Parse-Application-Id' aux requêtes sortantes ainsi que le token de session dans le header 'X-Parse-Session-Token' si l'utilisateur est identifié.

Il est aussi possible de traiter la réponse http afin avant que celle-ci ne soit utilisée par le code appelant.

Les formulaires

Angular permet d'implémenter les formulaires de diverses façons mais les plus courantes sont les Template-driven Forms et les Reactive Forms.

Template Driven Forms

Cette pratique consiste à exploiter intensivement le two-way bindings afin de garnir les propriétés d'un composant avec les valeurs des champs. Cette association « propriété – valeur de champ » se fait à l'aide de la directive `ngModel` fournie par Angular en important `FormsModule`.

user.component.ts

```
export class UserFormComponent {
  userName = '';
  userPass = '';

  constructor(private readonly userSrv: UserService) {}

  onSubmit() {
    try {
      this.userSrv.saveUser(this.userName, this.userPass);
      alert('Utilisateur enregistré avec succès !');
    } catch (err) {
      alert('Erreur lors de l'enregistrement de l'utilisateur !');
      console.error(err);
    }
  }
}
```

user.component.html

```
<h2>Création d'un nouvel utilisateur :</h2>
<form (ngSubmit)="onSubmit()">
  <p>Nom :</p>
  <input name="title" type="text" [(ngModel)]="userName" />
  <p>Mot de passe :</p>
  <input name="password" type="password" [(ngModel)]="userPass" />
  <button type="submit">Enregistrer</button>
</form>
```

Sous cette forme, l'exploitation des formulaires est extrêmement simple et rapide à mettre en place. Si cela semble, à première vue, être la solution à privilégier, elle va en réalité très rapidement trouver ces limites et devenir compliqué à maintenir pour de multiples raisons :

- Le fort couplage entre la vue et le code
- La vue possède la plus grosse partie de logique du formulaire, sa réutilisabilité est donc très limitée
- Le formulaire n'a pas d'existence propre dans le code, on ne peut donc pas y associer directement des mécaniques tels que les validateurs ou la gestion d'erreur.
- La montée en puissance de ce genre de formulaire passe par la multiplication des variables dans le code, ce qui devient très vite compliqué à maintenir.

Si ce genre formulaire n'est pas à bannir complètement, il est fortement déconseillé de l'utiliser dans des formulaires complexes. L'usage de `ngModel` trouve plutôt son sens pour connecter de petits champs isolés dans une vue.

Reactive Forms

Cette approche permet de combler la principale faiblesse des template driven forms en restituant la maîtrise du formulaire au code Typescript. Ainsi, le formulaire est entièrement piloté par le code et le template HTML ne conserve que la partie visuelle. Pour cela, les formulaires réactifs utilisent les mécaniques asynchrones (observables, etc.).

Il existe plusieurs méthodes pour implémenter un formulaire réactif, nous allons nous focaliser ici sur l'utilisation du FormGroup qui est l'une des plus courantes.

user-form.component.ts

```
export class UserFormComponent {  
  
  userForm = new FormGroup({  
    email: new FormControl('', [Validators.email, Validators.required]),  
    password: new FormControl(''),  
  });  
  
  constructor(private readonly userSrv: UserService) {}  
  
  onSubmit() {  
    if (this.userForm.valid) {  
      try {  
        this.userSrv.saveUser(this.userForm.value.email ?? '', this.userForm.value.password);  
        alert('Utilisateur enregistré avec succès !');  
      } catch (err) {  
        alert("Erreur lors de l'enregistrement de l'utilisateur !");  
        console.error(err);  
      }  
    }  
  }  
}
```

Ici, nous avons créé un formulaire sous forme d'un **FormGroup** qui contiendra deux **FormControls** :

- email, dont la valeur est une chaîne vide par défaut mais dont une valeur sera forcément requise. De plus le **Validators.email** permettra de s'assurer que la valeur saisie dans le champ est bien une adresse e-mail.
- password, dont la valeur est une chaîne vide par défaut mais qui n'a aucune contrainte de validation

Le premier argument de **new FormControl()** est lui aussi optionnel mais lui donner une valeur permet à indiquer à Angular le type de données utilisée pour la valeur, ici un **string**.

FormGroup stock les valeurs dans un tableau stocké dans sa propriété **value**. Il est important de noter que la valeur d'un champ du DOM peut toujours être **null** ou **undefined**. D'où l'utilisation de l'opérateur **??** afin d'appeler notre méthode **saveUser** qui attend obligatoirement un **string**.

FormGroup possède, entre autres, une propriété **valid** extrêmement utile qui permet de savoir, à tout moment, si le formulaire est valide.

user-form.component.html

```
<h2>Création d'un nouvel utilisateur :</h2>
<form [formGroup]="userForm" (ngSubmit)="onSubmit()">

  <p>E-mail :</p>
  <input name="email" type="text" formControlName="email" />
  <small *ngIf="userForm.controls.password.hasError('required')"> L'e-mail est obligatoire </small>

  <p>Mot de passe :</p>
  <input name="password" type="password" formControlName="password" />

  <button type="submit" [disabled]="!userForm.valid">Enregistrer</button>
</form>
```

Au niveau de la vue, `ngModel` fait place à `formControlName` afin d'associer le `FormControl` à l'input du DOM. Le formulaire quant à lui, reçoit une propriété `formGroup` qui, comme son nom l'indique associe le `FormGroup` à la balise `<form>`.

A partir de là, notre formulaire est déjà fonctionnel. Néanmoins quelques améliorations ont été ajoutée afin d'améliorer l'ergonomie globale :

- Nous utilisons le property bindings pour désactiver le bouton de soumission de formulaire lorsque celui-ci n'est pas valide.
- Nous affichons un message lorsque l'email est vide. Nous aurions pu aussi signaler la non-conformité de la valeur saisie.

Le formulaire ainsi créé, pourrait très bien être affichée sous une autre forme dans une autre vue. La logique du formulaire (initialisation, validation...) est donc bien découplée de sa couche présentation, rendant son usage flexible et facilement réutilisable.

Autres technologies à considérer

Angular Materials

Angular Materials est une suite de composants visuels basé sur Material Design. Développé par les équipes Google, elle offre l'énorme avantage d'être systématiquement mise à jour parallèlement à Angular, assurant sa parfaite compatibilité lors d'une mise à jour majeure.

<https://material.angular.io/>

Jest

Jest est un outil dédié aux tests unitaires/intégration. Initialement conçu pour React, cette librairie fonctionne extrêmement bien avec les autres frameworks, y compris Angular. Il offre ainsi une alternative très intéressante à Karma, l'outil livré historiquement avec Angular.

<https://jestjs.io/fr/>

Volta

Volta est un utilitaire qui permet d'installer plusieurs instances de NodeJs parallèlement et d'utiliser automatiquement celles qui est définie dans le projet. Cela est très utile lorsque les développeurs gèrent des projets ne tournant pas sur les mêmes versions de NodeJS.

<https://volta.sh/>

NgRx

NgRx est un puissant « store manager » qui permet de structurer de manière globale les états de l'application. Cet outil est fortement inspiré de Redux, très populaire chez les utilisateurs de React.

<https://ngrx.io/>

ESLint

ESLint est le linter le plus répandu dans le milieu du JS. Il est rapide, très complet et dispose d'extension pour la majorité des IDE. De plus il dispose d'un plugin efficace pour l'intégration de TypeScript ce qui est parfait pour Angular.

<https://eslint.org/>

Prettier

Prettier permet de reformater le code selon des règles établies sous forme de fichier de configuration. Il permet ainsi d'uniformiser le code entre tous les développeurs, le rendant ainsi plus lisible.

<https://prettier.io/>

NX

NX est une suite d'outils destinés à la génération et la compilation des projets en mono-repos. Il offre diverses fonctionnalités intéressantes permettant de faciliter les interactions entre les projets, de gérer leur compilation et déploiement ou encore de mettre en cache les builds.

<https://nx.dev/>

Compodoc

<https://compodoc.app/>

Compodoc est un package qui permet de générer une documentation complète à partir des JsDoc fournis dans le code.

Electron

Electron permet de distribuer une application JS sous forme d'un client lourd. Pour cela, il embarque un NodeJs et Chromium qui permettent d'implémenter et de distribuer une application complète en JS sous forme d'exécutable.

<https://www.electronjs.org/fr/>

Angular DevTools

Angular DevTools est une extension Chrome et Firefox permettant d'inspecter les variables Angular directement depuis les outils de développement du navigateur. Il offre ainsi un outil de débogage extrêmement puissant et simple à mettre en œuvre.