

# INTRODUCTION À ANGULAR

# HISTORIQUE ET VERSIONS

# ANGULARJS

- Premier framework sorti en **2010**
- Conçu pour des applications **mono-page (SPA)**
- Développé par **Google**

# ANGULAR 2+

- **Refonte complète** d'AngularJS
- Introduit en **2016**
- **Architecture modulaire et orientée composants**

## DERNIÈRE VERSION STABLE

- Toujours **maintenue** et **mise à jour** par Google
- Consultez [Angular.io](#) pour la dernière version

# CONCEPTS CLÉS

- **Composants**
- **Directives**
- **Services**
- **Modules**

Concept	Description
Composants	Ils contiennent le code logique, le style et le contenu HTML d'une partie de l'application Angular.
Directives	Elles permettent de manipuler le DOM et de modifier le comportement des éléments lors de différents événements.
Services	Ils sont utilisés pour partager des données, des méthodes et des fonctionnalités communes entre différents composants de l'application Angular.
Modules	Ils organisent et regroupent différentes fonctionnalités de l'application Angular pour assurer une meilleure structure et lisibilité du code.

# AVANTAGES ET INCONVÉNIENTS

## COMPARAISON AVEC D'AUTRES FRAMEWORKS

- **React**: plus souple, moins de convention
- **Vue**: plus simple, taille réduite
- **Angular**: plus structuré, implémente le modèle **MVVM**

# CAS D'UTILISATION APPROPRIÉS

- **Applications d'entreprise**
- Applications nécessitant des **fonctionnalités complexes**
- Équipes de développement avec des **développeurs expérimentés**

# COMPOSANTS

# STRUCTURE D'UN COMPOSANT

Un composant **Angular** est constitué de trois parties principales :

1. **Template** : le code HTML pour afficher les données
2. **Classe** : le code TypeScript pour définir les données et la logique
3. **Métadonnées** : des informations supplémentaires pour Angular

# TEMPLATE

Le **template** d'un composant est le code **HTML** qui structure son apparence. Il peut être défini dans un fichier séparé (`.html`) ou directement dans le fichier `.ts` du composant.

```
<div>
  <h1>Hello, {{ name }}!</h1>
  <button (click)="onClick()">Click me</button>
</div>
```

# CLASSE

La **classe** d'un composant est définie en **TypeScript**. Elle déclare les **données** et les **méthodes** pour le template.

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

@Component({ ... })
export class MyComponent {
  name = "Angular";
  onClick() {
    console.log("Button clicked");
  }
}
```

# MÉTADONNÉES

Les métadonnées sont définies par un décorateur `@Component` et incluent des informations pour **Angular**, telles que le **sélecteur** et l'**emplacement du template**.

```
@Component ({  
  selector: "app-my-component",  
  templateUrl: "./my-component.component.html",  
})
```

# INTERACTION ENTRE COMPOSANTS

# ENTRÉES (INPUTS)

Les composants peuvent recevoir des données de leurs parents en utilisant les **inputs** déclarés avec le décorateur `@Input()`.

```
// parent.component.ts
import { Component } from "@angular/core";

@Component({ ... })
export class ParentComponent {
  data = "Hello from parent";
}

// child.component.ts
import { Component, Input } from "@angular/core";

@Component({ ... })
export class ChildComponent {
  @Input() data: string;
}
```

# SORTIES (OUTPUTS)

Les composants peuvent émettre des événements vers leurs parents en utilisant les **outputs** déclarés avec le décorateur `@Output()` et la classe `EventEmitter`.

```
// child.component.ts
import { Component, Output, EventEmitter } from "@angular/core";

@Component({ ... })
export class ChildComponent {
  @Output() clicked = new EventEmitter<void>();

  onClick() {
    this.clicked.emit();
  }
}

// parent.component.ts
import { Component } from "@angular/core";
```

# CONTENU TRANSCLUS (NG-CONTENT)

Le contenu transclus permet d'insérer du contenu **HTML** depuis le parent directement dans le template du **composant enfant** à l'aide de la balise `<ng-content>`.

```
<!-- child.component.html -->
<div>
  <ng-content></ng-content>
</div>

<!-- parent.component.html -->
<app-child>
  <p>Hello from parent</p>
</app-child>
```

# CYCLE DE VIE D'UN COMPOSANT

# HOOKS ET LEUR ORDRE D'EXÉCUTION

Les **composants Angular** ont un cycle de vie qui commence par la création, puis l'initialisation des données, la mise à jour et enfin la destruction. Les **hooks** sont des méthodes spéciales qui sont appelées à différentes étapes de ce cycle.

1. ngOnChanges: appelé lorsqu'un input est modifié
2. ngOnInit: appelé après la création du composant et l'initialisation des inputs
3. ngDoCheck: appelé après chaque cycle de détection des changements
4. ngAfterContentInit: appelé après l'initialisation du contenu transclus (ng-content)
5. ngAfterContentChecked: appelé après chaque vérification du contenu transclus
6. ngAfterViewInit: appelé après l'initialisation des vues enfants
7. ngAfterViewChecked: appelé après chaque vérification des vues enfants
8. ngOnDestroy: appelé avant la destruction du composant

# UTILISATION AVEC NGONCHANGES

Le hook `ngOnChanges` est souvent utilisé pour réagir aux changements d'**inputs**. Il est appelé avec un objet `SimpleChanges` qui contient les valeurs précédentes et actuelles des inputs modifiés.

```
import {  
  Component,  
  Input,  
  OnChanges,  
  SimpleChanges,  
} from '@angular/core';  
  
@Component({ ... })  
export class MyComponent implements OnChanges {  
  @Input() data: string;  
  
  ngOnChanges(changes: SimpleChanges) {  
    console.log("Data changed:", changes.data.previousValue, "->", changes.data.currentValue);  
  }  
}
```

# DIRECTIVES

# TYPES DE DIRECTIVES

Dans **Angular**, il existe deux types de directives :

- **Directives attributs**
- **Directives structurelles**

# DIRECTIVES ATTRIBUTS

Les **directives attributs** modifient la structure et le comportement des éléments du **DOM**. Syntaxe :

```
<element [directive]="expression">
```

# DIRECTIVES STRUCTURELLES

Les **directives structurelles** modifient le layout du **DOM** en ajoutant, supprimant ou remplaçant des éléments. Syntaxe :

```
<element *directive="expression">
```

Directives courantes	Utilisation
*ngIf	Conditionnellement afficher ou masquer des éléments
*ngFor	Créer des éléments en fonction d'une collection
*ngSwitch	Afficher des éléments en fonction d'une expression

# DIRECTIVES INTÉGRÉES

# NGIF

La directive **ngIf** permet d'ajouter ou de supprimer un élément du **DOM** selon une condition. Syntaxe :

```
<element *ngIf="condition">
```

# NGFOR

La directive **ngFor** permet de répéter un élément du DOM pour chaque élément d'une collection. Syntaxe :

```
<element *ngFor="let item of items; index as i">
```

# NGSWITCH

La directive **ngSwitch** fonctionne comme un `switch case` pour afficher des éléments du **DOM** selon une condition. Syntaxe :

```
<element [ngSwitch] = "expression">
  <element *ngSwitchCase = "caseValue"></element>
  <element *ngSwitchDefault></element>
</element>
```

# CRÉATION DE DIRECTIVES PERSONNALISÉES

# STRUCTURE ET MÉTADONNÉES

Pour créer une **directive personnalisée**, créez une classe TypeScript avec le décorateur `@Directive`.  
Syntaxe :

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

@Directive({
  selector: '[appCustomDirective]'
})
export class CustomDirective {
  // Implementation goes here
}
```

# INTERACTION AVEC LES ÉLÉMENTS DU DOM

Pour interagir avec le DOM, injectez l'élément **ElementRef** dans le constructeur et modifiez les **propriétés** ou les **styles** de l'élément. Exemple :

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

@Directive({
  selector: '[appCustomDirective]'
})
export class CustomDirective {
  constructor(private el: ElementRef) {
    this.el.nativeElement.style.backgroundColor = 'yellow';
  }
}
```

# SERVICES

# UTILISATION DES SERVICES

Les **services** sont des classes qui fournissent des fonctionnalités **réutilisables** et centralisent la **logique métier**.

- Créer un service :

```
ng generate service mon-service
```

- Utilisation d'un service dans un composant :

```
import { MonService } from './mon-service.service';

constructor(private monService: MonService) { }
```

- Exemple d'un service pour la gestion des utilisateurs :

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

@Injectable({
  providedIn: 'root'
})
export class UserService {
  users = [
    { id: 1, name: 'Alice' },
    { id: 2, name: 'Bob' }
  ];

  getUsers() {
    return this.users;
  }
}
```



# INJECTION DE DÉPENDANCES

Angular utilise l'**injection de dépendances** pour fournir des instances de **services** à d'autres éléments de l'application, tels que les **composants**.

Avantages	Exemple d'utilisation
Réutilisation de code	Partage d'un service commun
Réduction de couplage	Avoir plusieurs implémentations
Testabilité	Créer des mocks pour les tests

# FOURNISSEURS (providers)

Pour rendre un **service** disponible dans l'ensemble de l'**application**, il faut l'enregistrer en tant que fournisseur (provider) dans un **module**.

## Etapes Description

---

1. Créer un service avec la CLI : `ng generate service my-service`
2. Importer le service dans le module
3. Ajouter le service à la propriété providers du décorateur @NgModule

# CRÉATION DE SERVICES PERSONNALISÉS

# STRUCTURE ET MÉTADONNÉES

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

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

**Import** : Les services sont importés à l'aide de la déclaration `@Injectable()`. Celle-ci est utilisée pour déclarer qu'un service peut être injecté dans une autre classe (par exemple, un composant).

**Métadonnées** : Les métadonnées `providedIn: 'root'` déterminent que le service est disponible au niveau de toute l'application. Ainsi, il peut être injecté dans n'importe quel composant, directive ou autre service.

# UTILISATION AVEC HTTPCLIENT

Injectez HttpClient dans votre service pour effectuer des **requêtes HTTP** :

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

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

# INTERACTION AVEC LES API REST

Les **services** permettent de centraliser la communication avec les **API REST**, en utilisant `HttpClient` pour effectuer des requêtes et gérer les réponses.

Exemple de requête GET :

```
getArticles() {  
    return this.http.get<Article[]>(`${API_URL}/articles`);  
}
```

# MODULES

- Les **modules** sont une manière d'organiser et de regrouper des fonctionnalités similaires en Angular.
- Un module Angular est composé de **composants**, **directives** et **services**.
- Le module racine est généralement appelé AppModule.

## Caractéristiques des Modules

---

Contient des composants

---

AppModule, SharedModule

---

Importe d'autres modules

---

BrowserModule, FormsModule, HttpClientModule

---

Fournit un service

## Exemples

| AppService, UserService |

# INTRODUCTION AUX MODULES

Angular utilise des **modules** pour organiser et séparer les différents blocs fonctionnels de l'application. Les modules facilitent la structuration, le développement et le test de l'application.

## Avantages des modules   Exemples concrets

---

Structuration

Organisation des composants par fonctionnalité

Développement

Facilité de maintenabilité et d'évolution du code

Test

Isolation des fonctionnalités pour les tests unitaires

# ORGANISATION DU CODE AVEC LES MODULES

- Les **modules** permettent de **séparer** et **grouper** différentes parties du code
- Chaque fichier .ts peut être considéré comme un **module**
- On utilise `import` pour **utiliser** du code depuis un autre module
- On utilise `export` pour permettre à d'autres modules d'**accéder** à une partie du code

```
// Exemple d'import et export

// Fichier avec un code à exporter
export class MaClasse {
    //...
}

// Fichier utilisant la classe exportée
import { MaClasse } from './chemin/vers/le/fichier';
```

# NGMODULE

NgModule est une **classe** décorée avec le **décorateur** @NgModule. Il prend un objet de **métadonnées** qui fournit des informations sur le module.

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

@NgModule({
  imports: [], // Liste des modules importés
  declarations: [], // Liste des composants, directives et pipes du module
  providers: [], // Liste des services fournis par le module
  exports: [] // Liste des composants, directives et pipes exportés par le module
})
export class MonModule {}
```

# DÉCLARATIONS, IMPORTATIONS ET EXPORTATIONS

- **declarations** : liste des **composants**, **directives** et **pipes** qui appartiennent à ce module
- **imports** : liste des **modules** dont les composants et directives exportés sont utilisés dans les templates des composants de ce module
- **exports** : liste des **composants**, **directives** et **pipes** qui peuvent être utilisés dans d'autres modules qui importent ce module

# LAZY-LOADING

Le **Lazy-loading** permet de charger des **modules** à la demande plutôt que de les charger au démarrage de l'application, ce qui améliore les **performances**.

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';

const routes: Routes = [
  {
    path: 'example',
    loadChildren: () => import('./example/example.module').then(m => m.ExampleModule)
  }
];

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

# ROUTERMODULE

Le **RouterModule** permet de configurer les **routes** et les **paramètres de navigation** d'une application **Angular**.

RouterModule	Objectif
Déclaration des routes	Configure les itinéraires avec les composants correspondants
Navigation	Gère l'historique de navigation et la navigation
Paramètres	Permet de transmettre des informations entre les pages

- Les routes sont déclarées dans un tableau
- Un exemple simple de configuration de router :

```
import { RouterModule, Routes } from '@angular/router';
import { Page1Component } from './page1/page1.component';
import { Page2Component } from './page2/page2.component';

const routes: Routes = [
  { path: 'page1', component: Page1Component },
  { path: 'page2', component: Page2Component }
];

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

# MODULES INTÉGRÉS

- **HttpClientModule** : Effectuer des requêtes HTTP
- **FormsModule, ReactiveFormsModule** : Gérer les formulaires
- **RouterModule** : Gérer la navigation entre les pages
- **BrowserAnimationsModule** : Ajouter des animations aux éléments du DOM

# BROWSERMODULE

Le **BrowserModule** importe la plate-forme de **navigateur** et les **fournisseurs** nécessaires pour exécuter l'application sur un navigateur.

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

@NgModule({
  imports: [ BrowserModule ],
  // ...
})
export class AppModule { }
```

# FORMSMODULE

Le **FormsModule** fournit des directives pour créer et gérer les **formulaires** à l'aide de la méthode **pilotée par le template**.

Directives	Description
ngForm	Crée un groupe de contrôles à l'échelle du formulaire.
ngModel	Enregistre un contrôle de formulaire pour un élément donné.
ngModelGroup	Crée un sous-groupe de contrôles au sein d'un formulaire.

# HTTPCLIENTMODULE

Le **HttpClientModule** permet d'interagir avec les **API REST** en utilisant l'objet **HttpClient**.

1. Importer HttpClientModule:

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

2. Ajouter HttpClientModule aux imports :

```
@NgModule({
  imports: [
    HttpClientModule,
    // ... autres imports
  ],
  // ... autres propriétés
})
export class AppModule { }
```

# CRÉATION DE MODULES PERSONNALISÉS

1. Utilisez la commande CLI pour générer un module personnalisé :

```
ng generate module custom-module
```

2. Importez les **composants** et/ou **services** nécessaires dans votre module personnalisé :

```
```javascript
import { NgModule } from '@angular/core'; import { CommonModule } from '@angular/common';
import { CustomComponent } from './custom-component/custom-component.component';
```

```
@NgModule({ declarations: [ CustomComponent ], imports: [ CommonModule ], exports: [
CustomComponent ] }) export class CustomModule {}
```

3. Ajoutez le **module** personnalisé dans le `NgModule` principal :

```
```javascript
import { CustomModule } from './custom-module/custom-module.module';

@NgModule({
  imports: [ // ..., CustomModule ],
})
export class AppModule {}
```

# STRUCTURE ET MÉTADONNÉES

Un module personnalisé doit avoir une classe décorée avec le décorateur `@NgModule` et un objet de métadonnées définissant les **declarations**, **imports** et **exports** du module.

Métadonnée	Description
Declarations	Liste des composants, directives et pipes du module
Imports	Liste des modules utilisés par ce module
Exports	Liste des composants, directives et pipes qui seront accessibles aux autres modules

# PARTAGE DE FONCTIONNALITÉS

Les **modules personnalisés** permettent de partager des fonctionnalités entre différentes parties de l'application et facilitent la **réutilisation du code** et la **séparation des responsabilités**.

Avantages	Exemples
Réutilisation du code	Directives
Séparation claire	Services
Maintenance simplifiée	Composants réutilisables

# FORMULAIRES

# MODÈLE PILOTÉ PAR LE TEMPLATE (TEMPLATE-DRIVEN FORMS)

Les **formulaires pilotés par le modèle** sont basés sur les **directives Angular** pour gérer la logique de formulaire directement dans le template.

# ngModel

Permet de lier un élément de formulaire (**input**, **select**, etc.) à une propriété du **composant**.

```
<input [(ngModel)]="nom" />
```

# VALIDATION INTÉGRÉE

Utilisation de **validateurs HTML5** et de **directives Angular** spécifiques.

- required
- minlength
- maxlength
- pattern

Exemple :

```
<form>
  <input type="text" ngModel required minlength="3" maxlength="10" pattern="[a-zA-Z0-9]*" />
</form>
```

# MODÈLE PILOTÉ PAR LE CODE (REACTIVE FORMS)

Les **formulaires réactifs** sont basés sur la création et la gestion de la **logique de formulaire** dans la classe du **composant**.

# FormControl, FormGroup, FormArray

Ce sont les classes utilisées pour créer et gérer la **structure** du formulaire.

```
form = new FormGroup({  
  nom: new FormControl(''),  
  prenom: new FormControl(''),  
});
```

# VALIDATION PERSONNALISÉE

Création de **fonctions de validation personnalisées** pour gérer les **erreurs de formulaire**.

# VALIDATION

# VALIDATION INTÉGRÉE

Angular propose des **validateurs intégrés** basés sur les attributs **HTML5** :

- **required**
- **minlength**
- **maxlength**
- **pattern**

# VALIDATION PERSONNALISÉE

Créez vos propres **fonctions de validation** pour répondre à des besoins spécifiques :

```
function validateurMonChamp(control: FormControl) {  
  const value = control.value;  
  // Vérifiez la valeur et retournez un objet d'erreur ou null  
}
```

Ajoutez les **validateurs personnalisés** à vos FormControl et FormGroup.

# ROUTAGE

# CONFIGURATION DU ROUTAGE

# ROUTES ET ROUTERMODULE

Pour configurer le **routage**, définissez un tableau de routes avec les chemins et les **composants** associés.  
Importez la classe RouterModule pour définir les routes.

```
import { RouterModule, Routes } from '@angular/router';

const routes: Routes = [
  { path: 'chemin1', component: Composant1 },
  { path: 'chemin2', component: Composant2 },
];

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

# NAVIGATION ENTRE LES COMPOSANTS

Utilisez la directive **routerLink** pour naviguer entre les composants.

```
<a routerLink="/chemin1">Composant 1</a>
<a routerLink="/chemin2">Composant 2</a>

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

# PARAMÈTRES DE ROUTE

# PARAMÈTRES OBLIGATOIRES ET FACULTATIFS

Les **paramètres obligatoires** sont définis dans les routes et les **paramètres facultatifs** sont passés via la directive `routerLink`.

```
{ path: 'chemin/:id', component: Composant }
```

```
<a [routerLink]="/chemin, id">Composant</a>
```

# RÉCUPÉRATION ET UTILISATION DES PARAMÈTRES

Pour récupérer les paramètres de la route, importez `ActivatedRoute` et utilisez `snapshot` ou `queryParams`.

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

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

# GUARDS

# CANACTIVATE

Utilisez canActivate pour protéger les **routes** en fonction de certaines conditions, comme l'**authentification**.

```
{ path: 'chemin', component: Composant, canActivate: [AuthGuard] }
```

# CANDEACTIVATE

canDeactivate empêche la **navigation** vers d'autres routes si certaines **conditions** ne sont pas remplies (par exemple, si les données doivent être enregistrées).

```
{ path: 'chemin', component: Composant, canDeactivate: [CanDeactivateGuard] }
```

# CANACTIVATECHILD

canActivateChild protège les **routes enfants** en fonction de certaines conditions.

```
{ path: 'chemin', component: Composant, canActivateChild: [AuthGuard], children: [...] }
```

# COMMUNICATION AVEC LES API REST

# HTTPCLIENT

Pour interagir avec les **API REST**, Angular fournit le module **HttpClient** qui permet d'envoyer des requêtes **HTTP**.

Exemple d'utilisation de HttpClient:

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

@Component({ ... })
export class AppComponent implements OnInit {
  constructor(private http: HttpClient) {}

  ngOnInit() {
    this.http.get('https://api.example.com/data').subscribe(data => {
      console.log(data);
    });
  }
}
```

# SYNTAXE ET UTILISATION

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

Injectez **HttpClient** dans le constructeur d'un service :

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

# REQUÊTES HTTP (GET, POST, PUT, DELETE)

Pour effectuer des requêtes HTTP en Angular, utilisez les méthodes disponibles sur l'instance d'**HttpClient**:

- `get(url: string)`: **récupérer des données**
- `post(url: string, body: any)`: **ajouter des données**
- `put(url: string, body: any)`: **modifier des données**
- `delete(url: string)`: **supprimer des données**

# OBSERVABLE ET ASYNC PIPE

Le module **HttpClient** renvoie des instances de la classe **Observable**, qui permettent d'effectuer des opérations de manière **asynchrone**.

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

// ...

constructor(private http: HttpClient) {}

ngOnInit() {
  this.http.get('https://api.example.com/data').subscribe((data) => {
    console.log(data);
  });
}
```

- Utilisation de l'**async pipe**:

```
<ul>
  <li *ngFor="let item of items | async">{{ item }}</li>
</ul>
```

# UTILISATION AVEC LES SERVICES

Dans les services, utilisez **HttpClient** pour récupérer ou manipuler des données :

```
getUsers ()  {
    return this.http.get('https://api.example.com/users');
}
```

# MANIPULATION DES DONNÉES

Dans les **composants**, utilisez la méthode `**subscribe () **` pour gérer les données récupérées depuis les **services** :

```
ngOnInit () {  
    this.userService.getUsers () .subscribe (users => {  
        this.users = users;  
        // Faites ce que vous voulez avec les données récupérées  
    });  
}
```

Aussi, pour éviter les **fuites de mémoire**, pensez à se désabonner des observables dans la méthode `**ngOnDestroy () **` du composant.

# BONNES PRATIQUES

# ORGANISATION DU CODE (ARCHITECTURE)

- Utiliser une **architecture modulaire**
- Structurer le code par **fonctionnalités**
- Séparer les **composants, services** et **directives** dans des dossiers distincts
- Utiliser des **conventions de nommage** cohérentes (ex: kebab-case pour les fichiers, PascalCase pour les classes)

# TESTS UNITAIRES ET D'INTÉGRATION

# JASMINE ET KARMA

**Jasmine** et **Karma** sont les frameworks de test par défaut pour **Angular**. Jasmine est un framework de test comportemental (**BDD**) tandis que Karma est un test runner qui permet d'exécuter les tests dans des navigateurs.

# TESTBED

**TestBed** est un module d'**Angular** spécifiquement conçu pour faciliter les **tests d'intégration**. Il permet de créer des modules de test avec les dépendances et les composants nécessaires.

# PERFORMANCES ET OPTIMISATION

# CHANGEDETECTIONSTRATEGY

- Utiliser la stratégie **\*\*OnPush\*\*** pour les composants **performants**
- Limiter les déclencheurs de **détection de changement**

## Stratégie de détection de changement

	Description
Default	La détection de changement est automatiquement vérifiée à chaque événement
OnPush	La détection de changement est vérifiée seulement quand les propriétés d'entrée changent

# LAZY-LOADING DES MODULES

- Utiliser le **lazy-loading** pour charger les modules uniquement lorsqu'ils sont **nécessaires**

# MINIFICATION ET CONCATÉNATION DES FICHIERS

- Utiliser **Webpack** ou d'autres outils de build pour **minifier** et **concaténer** les fichiers

# OUTILS ET RESSOURCES COMPLÉMENTAIRES

# ANGULAR CLI

**Angular CLI** est un outil en ligne de commande qui facilite la **création** et la **gestion** d'applications Angular.

# INSTALLATION ET UTILISATION

Pour installer **Angular CLI**, utilisez la commande suivante :

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

Pour créer une nouvelle **application Angular**, exéutez :

```
ng new my-app
```

# GÉNÉRATION DE COMPOSANTS, DIRECTIVES, SERVICES, ETC.

Angular CLI permet de générer rapidement des éléments avec des commandes préfixées par `ng generate` ou `ng g`:

Élément	Commande
<b>Composant</b>	<code>ng generate component</code>
<b>Directive</b>	<code>ng generate directive</code>
<b>Service</b>	<code>ng generate service</code>
<b>Module</b>	<code>ng generate module</code>
<b>Pipe</b>	<code>ng generate pipe</code>
<b>Guard</b>	<code>ng generate guard</code>
<b>Classe</b>	<code>ng generate class</code>
<b>Interface</b>	<code>ng generate interface</code>
<b>Enum</b>	<code>ng generate enum</code>

# DOCUMENTATION OFFICIELLE

La **documentation officielle** d'**Angular** est une ressource précieuse pour comprendre et approfondir le framework. Elle est disponible sur le site [angular.io](https://angular.io).

# TUTORIELS ET ARTICLES EN LIGNE

Il existe de nombreux **tutoriels** et **articles** en ligne pour apprendre et maîtriser **Angular**. Voici quelques sites recommandés :

- [Angular University](#)
- [Scotch.io](#)
- [Coursetro](#)
- [Codecraft](#)

