



Angular 18/19

Une formation complète sur le développement d'applications web modernes avec Angular.

Appuyez sur espace pour la page suivante →



SOMMAIRE

Voici le sommaire de cette formation sur Angular
18/19:

 Introduction à Angular

 TypeScript Essentiels

 Configuration de l'environnement

 Les Bases d'Angular

 Composants Angular

 Syntaxe des templates

 Cycle de vie des composants

 Injection de dépendances

 Services

 Routing et Navigation

 Formulaires et Validation

 RxJS et Observables

 HTTP Client et API REST

 Signals (Nouveauté Angular 18)

 Directives et Pipes

 Performance et Optimisation

 Tests unitaires et E2E

 Déploiement

 Bonnes pratiques

 Code source du projet



Introduction à Angular

Qu'est-ce qu'Angular ?

Imaginez Angular comme un kit complet pour construire une maison moderne :

- Les **fondations** (le framework core)
- Les **outils** (CLI, DevTools)
- Les **plans** (architecture)
- Les **matériaux** (composants)

Exemple concret de ces termes :

- **Fondations :**

- Framework core
- TypeScript
- RxJS
- CLI
- DevTools

- **Plans :**

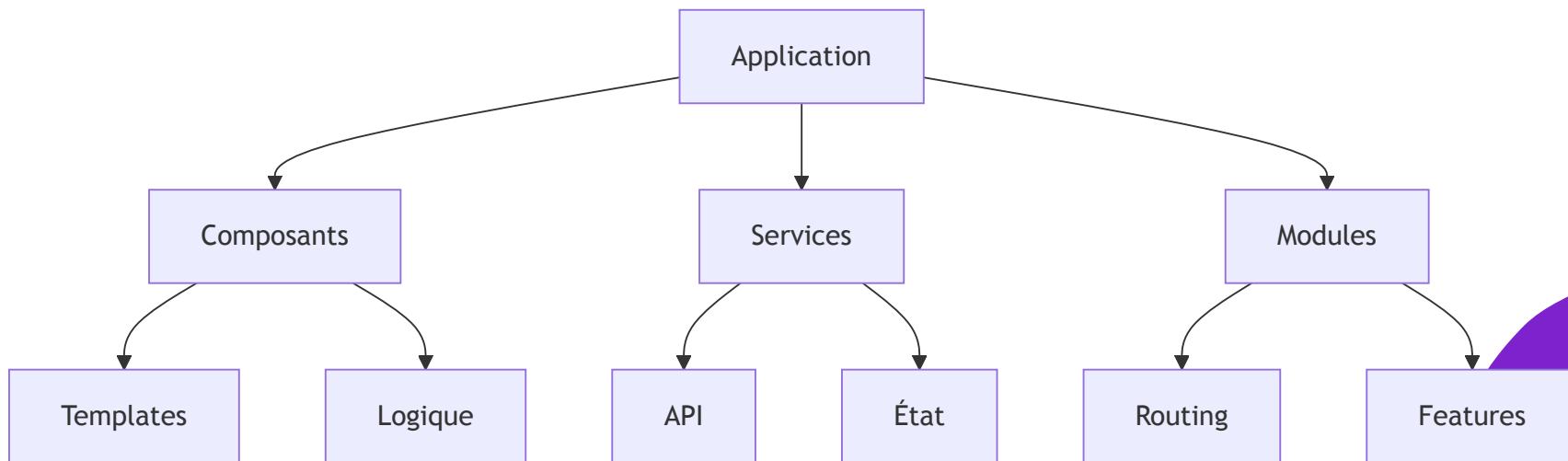
- Architecture orientée composants
- Routage / Routing
- State management avec RxJS / NgRx etc

- **Matériaux :**

- Composants : boutons, formulaires, tables, etc.
- Services : API, authentification, notifications, etc.

Architecture d'une application Angular

Structure typique d'un projet :



Les piliers d'Angular

1. Composants

Comme les LEGO® de votre application :

- Réutilisables
- Autonomes
- Combinables

2. Services

Comme les employés d'une entreprise :

- Spécialisés
- Partagés
- Indépendants

3. Dependency Injection

Comme un système de livraison automatique :

- Efficace
- Flexible
- Testable

Un exemple concret :

```
@Injectable({
  providedIn: 'root'
})
export class UserService {
  constructor(private http: HttpClient) {}
}
```

Ce qui veut dire que le service `UserService` est injectable dans n'importe quel composant

Et que c'est un service singleton

C'est à dire que toutes les instances de `UserService` sont la même instance.

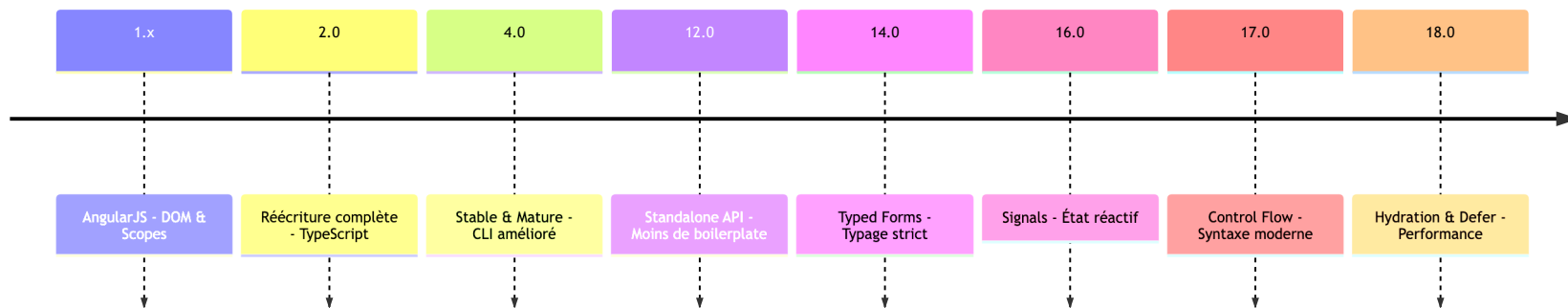
[Revenir au sommaire](#)

Évolution d'Angular

De AngularJS à Angular Moderne

- **AngularJS (1.x)**
 - Basé sur le DOM et les scopes
 - Directives comme composants
 - JavaScript vanilla
- **Angular 2+ : La révolution**
 - Réécriture complète en TypeScript
 - Architecture orientée composants
 - Performance améliorée
 - Injection de dépendances repensée

Historique d'évolution de Angular en timeline



Évolutions majeures

Angular 12-14 : Simplification

- Introduction des Standalone Components
- Suppression progressive des NgModules
- Amélioration du CLI
- Formulaires typés

Angular 15-16 : Réactivité

- Signals pour la gestion d'état
- Meilleure détection des changements
- SSR amélioré
- Hydratation intelligente

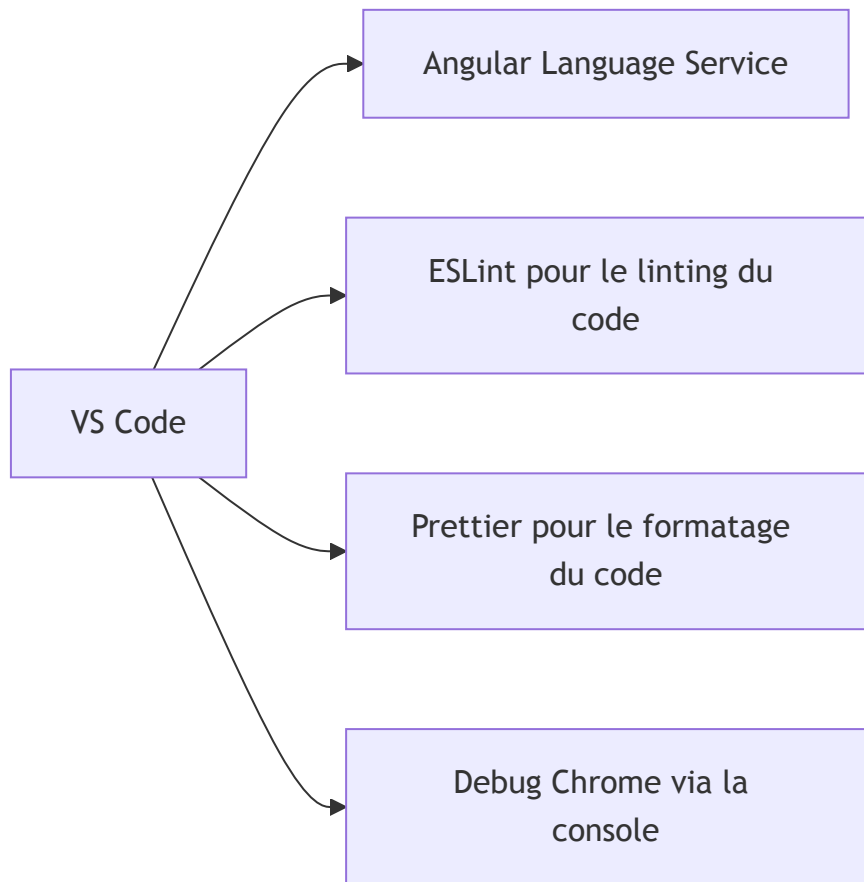
Angular 17-18 : Modernisation

- Nouveau Control Flow (@if, @for)
- Defer Loading intégré
- Build system avec Vite & ESBuild
- Developer experience améliorée

Comparaison avec d'autres frameworks

Caractéristique	Angular	React	Vue
Architecture	Full-framework	Bibliothèque	Progressive
Courbe d'apprentissage	Plus raide	Modérée	Douce
Tooling	Complet	Flexible	Intermédiaire
TypeScript	Natif	Optionnel	Optionnel

Outils essentiels



Extensions indispensables

- Angular Language Service pour la complétion de code
- Angular Snippets pour les snippets de code
- ESLint pour le linting du code
- Prettier pour le formatage du code

Prérequis techniques

Pour bien démarrer avec Angular, vous devez connaître :

✅ Fondamentaux

- HTML/CSS
- JavaScript moderne
- TypeScript basique
- Programmation orientée objet

❌ Pas nécessaire

- Backend development
- Mobile development
- WebAssembly



TypeScript Essentiels

Types de base

```
// Types primitifs
let name: string = 'John';
let age: number = 25;
let isActive: boolean = true;

// Arrays
let numbers: number[] = [1, 2, 3];
let names: Array<string> = ['John', 'Jane'];

// Tuple
let tuple: [string, number] = ['John', 25];
```

Interfaces et Types

```
// Interface
interface User {
  id: number;
  name: string;
  email?: string; // Propriété optionnelle
}

// Type
type UserRole = 'admin' | 'user' | 'guest';

// Utilisation
const user: User = {
  id: 1,
  name: 'John',
  email: 'john@example.com'
}
```

Décorateurs TypeScript

```
// Décorateur de classe
function Logger(target: any) {
  console.log('Class decorated:', target);
}

// Décorateur de propriété
function Required(target: any, propertyKey: string) {
  console.log('Property decorated:', propertyKey);
}

@Logger
class Example {
  @Required
  name: string;
}
```

Generics

```
// Fonction générique
function getFirst<T>(array: T[]): T {
  return array[0];
}



// Classe générique
class DataContainer<T> {
  private data: T;

  constructor(data: T) {
    this.data = data;
  }

  getData(): T {
    return this.data;
  }
}
```

Utility Types

```
interface Todo {  
  title: string;  
  description: string;  
  completed: boolean;  
}  
  
// Partial - Toutes les propriétés deviennent optionnelles  
type PartialTodo = Partial<Todo>;  
  
// Pick - Sélectionne certaines propriétés  
type TodoPreview = Pick<Todo, 'title' | 'completed'>;  
  
// Omit - Omet certaines propriétés  
type TodoWithoutDescription = Omit<Todo, 'description'>;
```



Configuration de l'environnement

Prérequis

```
# Installation de Node.js via nvm (recommandé)
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.0/install.sh | bash
nvm install --lts

# Installation du CLI Angular
npm install -g @angular/cli@latest
```

Création d'un projet Angular moderne

```
# Création d'un nouveau projet
ng new my-app --standalone

# Options recommandées
✓ Would you like to add routing? Yes
✓ Which stylesheet format would you like to use? SCSS
✓ Would you like to enable Server-Side Rendering (SSR)? No
```

standalone : pour utiliser les composants sans NgModules par défaut donc pas besoin de préciser

Structure du projet moderne

```
my-app/  
├── src/  
│   ├── app/  
│   │   ├── components/  
│   │   ├── services/  
│   │   ├── app.config.ts  
│   │   ├── app.routes.ts  
│   │   └── app.component.ts  
│   ├── assets/  
│   └── main.ts  
├── package.json  
└── angular.json
```

Configuration TypeScript

```
// tsconfig.json
{
  "compileOnSave": false,
  "compilerOptions": {
    "strict": true,
    "noImplicitOverride": true,
    "noPropertyAccessFromIndexSignature": true,
    "noImplicitReturns": true,
    "noFallthroughCasesInSwitch": true,
    "sourceMap": true,
    "declaration": false,
    "experimentalDecorators": true,
    "moduleResolution": "node",
    "importHelpers": true,
    "target": "ES2022",
```

Configuration des environnements

```
// src/environments/environment.ts
export const environment = {
  production: false,
  apiUrl: 'http://localhost:3000/api'
};

// src/environments/environment.prod.ts
export const environment = {
  production: true,
  apiUrl: 'https://api.monapp.com'
};
```

Scripts NPM utiles

```
// package.json
{
  "scripts": {
    "start": "ng serve",
    "build": "ng build",
    "build:ssr": "ng build && ng run my-app:server",
    "dev:ssr": "ng run my-app:serve-ssr",
    "lint": "ng lint",
    "test": "ng test",
    "e2e": "ng e2e"
  }
}
```

Exercice : Configuration initiale

1. Créez un nouveau projet Angular :

```
ng new exercice-app  
cd exercice-app
```

2. Ajoutez ESLint et Prettier :

```
ng add @angular-eslint/schematics  
npm install prettier prettier-eslint --save-dev
```

3. Configurez les environnements :

```
// src/app/environments/environment.ts
export const environment = {
  production: false,
  apiUrl: 'http://localhost:3000',
  features: {
    darkMode: true,
    analytics: false
  }
};
```

4. Testez votre configuration :

```
ng serve
ng lint
ng test
```

Cette configuration vous donnera une base solide pour développer des applications Angular modernes.

Configuration avec Vite (Angular 18/19)

Nouvelle configuration de build

```
// vite.config.ts
import { defineConfig } from '@angular-devkit/build-angular/vite';

export default defineConfig({
  build: {
    target: 'es2022'
  },
  server: {
    port: 4200
  }
});
```

Avantages de Vite & ESBuild

- Build plus rapide
- Hot Module Replacement amélioré
- Meilleure gestion des dépendances
- Consommation mémoire réduite



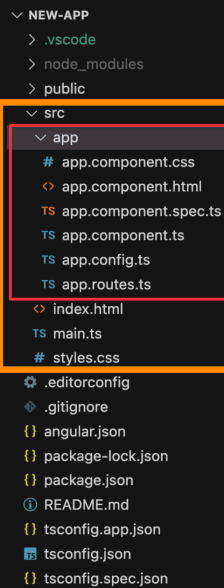
Les Bases d'Angular

Architecture fondamentale

- **Structure d'un projet Angular**
 - Le fichier angular.json
 - Les dossiers src/, app/, assets/
 - Les fichiers de configuration
- **Concepts clés**
 - Modules (NgModule)
 - Composants
 - Services
 - Directives
 - Pipes

SRC : Votre dossier source de votre projet

app : votre app , avec votre composant par défaut
la config et les routes



```

  NEW-APP
  ├── .vscode
  ├── node_modules
  ├── public
  └── src
      ├── app
      │   ├── app.component.css
      │   ├── app.component.html
      │   ├── app.component.spec.ts
      │   ├── app.component.ts
      │   ├── app.config.ts
      │   ├── app.routes.ts
      │   ├── index.html
      │   ├── main.ts
      │   └── styles.css
      ├── .editorconfig
      ├── .gitignore
      ├── angular.json
      ├── package-lock.json
      ├── package.json
      ├── README.md
      ├── tsconfig.app.json
      ├── tsconfig.json
      └── tsconfig.spec.json

```

tout les fichiers en dehors du "marquage" sont à
la racine du projet

Comprendre le bootstrapping

```
// main.ts
import { bootstrapApplication } from '@angular/platform-browser';
import { AppComponent } from './app/app.component';

bootstrapApplication(AppComponent, {
  providers: [
    // Configuration globale
  ]
}).catch(err => console.error(err));
```

Initialisation de l'application

C'est ici que l'on initialise l'application Angular. Si nous voulons par exemple utiliser un router ou utiliser `provideHttpClient`, nous pouvons le faire ici.

Je vous ai mis en évidence les lignes 3 et 4 qui sont les plus importantes.

```
bootstrapApplication(AppComponent, {  
  providers: [  
    provideRouter(routes),  
    provideHttpClient()  
  ]  
}).catch(err => console.error(err));
```

Structure d'un composant de base

```
// hello.component.ts
@Component({
  selector: 'app-hello',
  // le selector est le nom de la balise html qui va être utilisée pour utiliser le composant
  standalone: true,
  // standalone: true pour utiliser le composant sans NgModule, le composant vis en autonomie
  template: `
    <h1>Hello {{ name }}</h1>
    <button (click)="sayHello()">
      Click me
    </button>
  `,
  // template: ` pour le template du composant , en clair la vue :)
})
export class HelloComponent {
```

Data Binding fondamental

- **One-way binding (Liaison à sens unique)**

Interpolation: `{{ expression }}`

- Affiche des données du composant dans le template
- Exemple: `{{ user.name }}` affiche le nom de l'utilisateur
- Supporte les expressions simples: `{{ 1 + 1 }}` , `{{ user.firstName + ' ' + user.lastName }}`

- **Property binding (Liaison de propriété)**

- Syntaxe: `[property]="value"`
- Permet de lier une propriété d'un élément HTML à une valeur du composant
- Exemple: `[disabled]="isLoading"` désactive un bouton selon l'état
- Très utilisé pour les attributs HTML: `[src]` , `[href]` , `[class]` , etc.

- **Event binding (Liaison d'événement)**

- Syntaxe: `(event)="handler()"`
- Réagit aux événements de l'utilisateur
- Exemples courants:
 - `(click)="onClick()"`
 - `(submit)="onSubmit()"`
 - `(input)="onInput($event)"`
- `$event` donne accès aux données de l'événement

- **Two-way binding (Liaison bidirectionnelle)**

- Syntaxe banana in a box: `[(ngModel)]="property"`
- Combine property binding et event binding
- Met à jour automatiquement la vue ET le composant (synchro)
- Parfait pour les formulaires
- Exemple: `[(ngModel)]="user.name"` synchronise un champ input avec une propriété
- Nécessite d'importer `FormsModule` ou le `NgModel` dans le composant standalone



Exercice : Création du projet Mini-Blog

1. Créez un nouveau projet Angular :

```
ng new mini-blog --standalone --routing --style=scss  
cd mini-blog
```

2. Configurez la structure initiale :

```
mkdir src/app/features  
mkdir src/app/shared  
mkdir src/app/core
```

3. Créez le composant principal :

```
// app.component.ts
@Component({
  selector: 'app-root',
  template: `
    <header>
      <h1>{{ title }}</h1>
    </header>
    <main>
      <router-outlet />
    </main>
  `,
})
export class AppComponent {
  title = 'Mini Blog';
}
```

4. Testez l'application :

```
ng serve
```



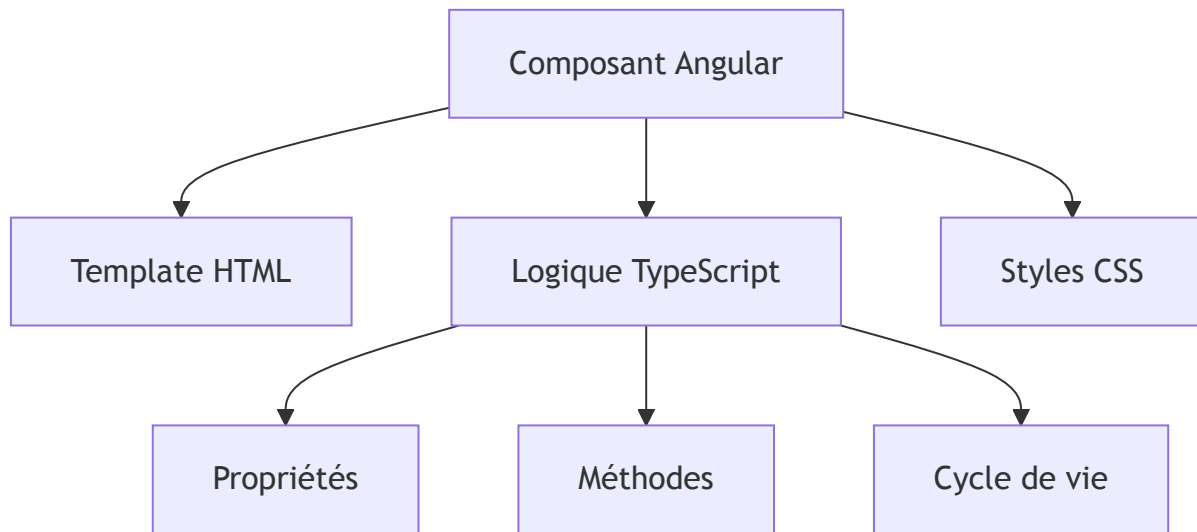
Les Composants dans Angular

Qu'est-ce qu'un composant ?

Un composant est comme une brique LEGO® de votre application :

- Une partie de l'interface utilisateur (UI)
- Autonome et réutilisable
- Avec sa propre logique et son propre template

Structure d'un composant



Composants Standalone (Angular 18/19)

```
@Component({  
  selector: 'app-user-card',  
  standalone: true, // Plus besoin de NgModule !  
  imports: [CommonModule],  
  template: `...`  
})  
export class UserCardComponent {}
```

Le décorateur @Component – Partie 1

```
@Component({
  // Nom de la balise HTML pour utiliser ce composant
  selector: 'app-user-profile',

  // Template HTML du composant
  template: `
    <div class="user-profile">
      <h2>{{ userName }}</h2>
    </div>
  `
})
```

Le décorateur @Component – Partie 2

```
@Component({  
  // Styles spécifiques au composant  
  styles: [`  
    .user-profile {  
      padding: 20px;  
      border: 1px solid #ccc;  
    }  
  `]  
})
```

Communication Parent → Enfant

Entrées (@Input)

```
@Component({
  selector: 'app-user-card',
  template: `
    <div class="card">
      <h3>{{ userName }}</h3>
      <p>{{ userRole }}</p>
    </div>
  `,
})
export class UserCardComponent {
  @Input() userName: string;
  @Input() userRole: string;
}
```

Utilisation des @Input

```
<app-user-card  
  userName="John Doe"  
  userRole="Admin"  
>
```

Communication Enfant → Parent

Sorties (@Output)

```
@Component({
  selector: 'app-counter',
  template: `
    <div>
      <h2>{{ count() }}</h2>
      <button (click)="increment()">+</button>
    </div>
  `,
})
export class CounterComponent {
  count = signal(0);
  @Output() countChange = new EventEmitter<number>();
}
```

Gestion des événements @Output

```
export class CounterComponent {  
  increment() {  
    this.count.update(n => n + 1);  
    this.countChange.emit(this.count());  
  }  
}
```

Utilisation :

```
<app-counter  
  (countChange)="handleCountChange($event)"  
>
```

Styles des composants – Partie 1

```
@Component({  
  selector: 'app-styled-button',  
  template: `  
    <button class="custom-btn">  
      <ng-content></ng-content>  
    </button>  
  `,  
})
```


Styles des composants – Partie 2

```
@Component({
  styles: [`
    .custom-btn {
      padding: 10px 20px;
      border-radius: 4px;
      border: none;
      background: #007bff;
      color: white;
      cursor: pointer;
    }
  `],
  encapsulation: ViewEncapsulation.ShadowDom
})
```

Projection de contenu - Structure

```
@Component({
  selector: 'app-card',
  template: `
    <div class="card">
      <div class="header">
        <ng-content select="[header]"></ng-content>
      </div>
      <div class="content">
        <ng-content></ng-content>
      </div>
    </div>
  `,
})
```

Projection de contenu - Utilisation

```
<app-card>  
  <h2 header>Mon titre</h2>  
  <p>Contenu principal</p>  
  <button footer>Action</button>  
</app-card>
```

Bonnes pratiques – À faire

- Un composant = une responsabilité unique
- Garder les composants petits et focalisés
- Utiliser des interfaces pour typer les inputs
- Documenter les inputs/outputs importants

Bonnes pratiques – À éviter ❌

- Trop de logique dans les templates
- Composants trop complexes
- Duplication de code entre composants
- Couplage fort entre composants

Exercice – Interface

```
interface User {  
  id: number;  
  name: string;  
  email: string;  
  role: 'admin' | 'user';  
}
```

Exercice - Composant

```
@Component({
  selector: 'app-user-list',
  template: `
    <div class="user-list">
      <h2>Utilisateurs</h2>
      @for (user of users(); track user.id) {
        <app-user-card
          [user]="user"
          (userClick)="onUserSelect($event)"
        />
      }
    </div>
  `,
})
```

Exercice - Logique

```
export class UserListComponent {  
  users = signal<User[]>([]);  
  @Output() userSelect = new EventEmitter<User>();  
  
  onUserSelect(user: User) {  
    this.userSelect.emit(user);  
  }  
}
```




Exercice : Composants du Blog

1. Créez le composant PostList :

```
// features/posts/post-list.component.ts
@Component({
  selector: 'app-post-list',
  standalone: true,
  template: `
    <div class="posts">
      @for (post of posts; track post.id) {
        <article class="post-card">
          <h2>{{ post.title }}</h2>
          <p>{{ post.excerpt }}</p>
        </article>
      }
    </div>
  `,
  styles: [`
```



Syntaxe des Templates

Interpolation et expressions

Nous avons déjà vu rapidement il y a quelques instants l'interpolation avec la syntaxe `{{ expression }}` mais nous allons revoir des cas concrets différents.

```
@Component({
  template: `
    <!-- Interpolation basique -->
    <h1>{{ title }}</h1>

    <!-- Expressions -->
    <p>Total: {{ price * quantity }}</p>

    <!-- Méthodes -->
    <div>{{ getMessage() }}</div>

    <!-- Chaînage de propriétés -->
    <span>{{ user?.address?.city }}</span>
  `,
})
```

Bindings de propriétés et d'événements

```
@Component({
  template: `
    <!-- Property binding -->
    <img [src]="imageUrl" [alt]="imageAlt">
    <button [disabled]="isDisabled">Click me</button>

    <!-- Event binding -->
    <button (click)="handleClick($event)">
      Click count: {{ clickCount }}
    </button>

    <!-- Two-way binding -->
    <input [(ngModel)]="userName">
    <p>Hello, {{ userName }}!</p>
  `
})
```

Directives structurelles modernes

```
@Component({
  template: `
    <!-- If moderne -->
    @if (isLoggedIn()) {
      <nav>Menu utilisateur</nav>
    } @else {
      <auth-form />
    }

    <!-- For moderne -->
    @for (item of items(); track item.id) {
      <item-card [data]="item" />
    }

    <!-- Switch moderne -->
```

Pipes et formatage

```
@Component({
  template: `
    <!-- Pipes de base -->
    <p>{{ date | date:'shortDate' }}</p>
    <p>{{ price | currency:'EUR' }}</p>
    <p>{{ text | uppercase }}</p>

    <!-- Chaînage de pipes -->
    <p>{{ data | json | async }}</p>

    <!-- Pipes avec paramètres -->
    <p>{{ number | number:'1.0-2' }}</p>

    <!-- Pipe personnalisé -->
    <p>{{ text | highlight:searchTerm }}</p>
```


Références de template et variables

```
@Component({
  template: `
    <!-- Référence locale -->
    <input #nameInput type="text">
    <button (click)="greet(nameInput.value)">
      Saluer
    </button>

    <!-- ViewChild -->
    <div #content>
      Contenu dynamique
    </div>

    <!-- Variables de template -->
    @for (item of items; track item.id; let i = $index) {
```

Cet exercice vous à permis de maîtriser :

- La syntaxe d'interpolation
- Les différents types de binding
- Les directives structurelles modernes
- L'utilisation des pipes
- Les références de template



Exercice : Template du Post

1. Créez le composant PostCard :

```
// features/posts/post-card.component.ts
@Component({
  selector: 'app-post-card',
  standalone: true,
  template: `
    <article class="post-card">
      <h2>{{ post.title }}</h2>
      <p class="meta">
        Par {{ post.author }}
        le {{ post.date | date:'shortDate' }}
      </p>
      <p [innerHTML]="post.excerpt"></p>
      <div class="actions">
        <button (click)="onRead()">Lire</button>
        @if (isAuthor) {
          <button (click)="onEdit()">Éditer</button>
        }
      </div>
    </article>
  `,
  styles: [`
    .post-card {
      padding: 1rem;
      border: 1px solid #ddd;
      border-radius: 4px
    }
  `]
```



Cycle de Vie des Composants

Hooks essentiels

- **ngOnInit:** Après la première initialisation des propriétés
- **ngOnDestroy:** Juste avant que le composant soit détruit
- **ngOnChanges:** Quand une propriété liée aux données change
- **ngAfterViewInit:** Après l'initialisation de la vue

Exemple d'implémentation

```
@Component({
  selector: 'app-lifecycle',
  template: `
    <h1>{{ title }}</h1>
    <p>{{ message }}</p>
  `,
})
export class LifecycleComponent implements OnInit, OnDestroy {
  @Input() title: string;
  message: string;

  ngOnInit() {
    console.log('Component initialized');
    this.message = 'Component is ready!';
  }
}
```


Détection des changements d'Input

```
@Component({
  selector: 'app-child',
  template: `
    <div>Data: {{ data }}</div>
  `
})
export class ChildComponent implements OnChanges {
  @Input() data: any;

  ngOnChanges(changes: SimpleChanges) {
    if (changes['data']) {
      console.log(
        'Previous:', changes['data'].previousValue,
        'Current:', changes['data'].currentValue
      );
    }
  }
}
```

AfterView et AfterContent

```
@Component({
  selector: 'app-view-child',
  template: `
    <div #contentDiv>
      <ng-content></ng-content>
    </div>
  `,
})
export class ViewChildComponent implements AfterViewInit {
  @ViewChild('contentDiv') contentDiv: ElementRef;

  ngAfterViewInit() {
    console.log('View initialized:', this.contentDiv.nativeElement);
  }
}
```

Bonnes pratiques

1. Initialisation

- Utiliser `ngOnInit` pour l'initialisation des données
- Éviter les opérations lourdes dans le constructeur

2. Nettoyage

- Toujours implémenter `ngOnDestroy` pour nettoyer les souscriptions
- Libérer les ressources (timers, listeners, etc.)

3. Performance

- Éviter les calculs lourds dans `ngOnChanges`
- Utiliser `ChangeDetectionStrategy.OnPush` quand possible



Injection de Dépendances

Principes fondamentaux

- **Qu'est-ce que l'injection de dépendances ?**
 - Pattern de conception logicielle
 - Gestion des dépendances entre composants
 - Facilite les tests et la maintenance
- **Avantages**
 - Découplage du code
 - Réutilisabilité
 - Testabilité améliorée

Création d'un service injectable

```
// user.service.ts
@Injectable({
  providedIn: 'root' // Service singleton au niveau application
})
export class UserService {
  private users: User[] = [];

  constructor(private http: HttpClient) {}

  getUsers(): Observable<User[]> {
    return this.http.get<User[]>('/api/users');
  }

  addUser(user: User): Observable<User> {
    return this.http.post<User>('/api/users', user);
  }
}
```

Hiérarchie d'injection

```
// Component-level provider
@Component({
  selector: 'app-feature',
  providers: [FeatureService] // Scope limité à ce composant
})
export class FeatureComponent {
  constructor(private featureService: FeatureService) {}
}

// Module-level provider
@NgModule({
  providers: [
    GlobalService,
    {
      provide: API_URL,
```

Tokens d'injection et providers

```
// Token d'injection personnalisé
export interface AppConfig {
  apiUrl: string;
  theme: 'light' | 'dark';
}

export const APP_CONFIG = new InjectionToken<AppConfig>('app.config');

// Configuration des providers
export const appConfig: ApplicationConfig = {
  providers: [
    {
      provide: APP_CONFIG,
      useValue: {
        apiUrl: 'https://api.example.com',
```


Utilisation avancée

```
@Injectable({
  providedIn: 'root'
})
export class CacheService {
  private cache = new Map<string, any>();

  constructor(
    @Optional() @SkipSelf() parentCache: CacheService,
    @Inject(CACHE_SIZE) private maxSize: number
  ) {
    if (parentCache) {
      this.cache = new Map(parentCache.cache);
    }
  }
}
```

Injection moderne avec inject()

```
// Avant (constructor injection)
@Component({
  template: `...`
})
class OldComponent {
  constructor(
    private userService: UserService,
    private router: Router,
    @Inject(APP_CONFIG) private config: AppConfig
  ) {}
}

// Après (fonction inject)
@Component({
  template: `...`
```

Avantages de inject()

- Plus concis
- Utilisable en dehors du constructor
- Parfait pour les composants standalone
- Meilleure inférence de type

```
@Component({
  standalone: true,
  template: `
    @if (user()) {
      <h1>Bienvenue {{ user().name }}</h1>
    }
  `,
})
class WelcomeComponent {
  // Injection et Signal en une ligne
  private auth = inject(AuthService)
  user = this.auth.currentUser

  // Injection dans une méthode
  logout() {
```

Injection conditionnelle avec inject()

```
@Component({
  template: `...`
})
class FeatureComponent {
  // Injection optionnelle
  private logger = inject(LoggerService, { optional: true })

  // Injection avec fallback
  private analytics = inject(AnalyticsService, {
    optional: true,
    self: true
  }) ?? new NoopAnalyticsService()

  // Injection au niveau parent
  private parentCache = inject(CacheService, {
```

Injection dans les services

```
@Injectable({ providedIn: 'root' })
class ModernService {
  // Injection directe sans constructor
  private http = inject(HttpClient)
  private config = inject(APP_CONFIG)

  // Computed basé sur injection
  private apiUrl = computed(() =>
    `${this.config.baseUrl}/api`
  )

  getData() {
    return this.http.get(`${this.apiUrl()}/data`)
  }
}
```



Services Angular

Introduction aux services

Voyons un service moderne avec Signals

Avant tout, je vous rapelle que vous n'êtes pas obligé d'utiliser Signals pour faire des services. Vous pouvez utiliser rxjs et les Observables comme Subject, BehaviorSubject, ReplaySubject, etc.

```
@Injectable({
  providedIn: 'root'
})
export class UserService {
  private users = signal<User[]>([]);
  private loading = signal(false);
  private error = signal<Error | null>(null);

  // Computed values
  readonly sortedUsers = computed(() =>
    [...this.users()].sort((a, b) => a.name.localeCompare(b.name))
  );

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


Injection de dépendances moderne

```
// Service avec configuration
@Injectable({
  providedIn: 'root',
  useFactory: (config: AppConfig) => {
    return new ApiService(config.apiUrl);
  },
  deps: [APP_CONFIG]
})
export class ApiService {
  constructor(private baseUrl: string) {}
}

// Token d'injection
const APP_CONFIG = new InjectionToken<AppConfig>('app.config');
```

Service avec état partagé

```
@Injectable({
  providedIn: 'root'
})
export class ThemeService {
  private darkMode = signal(false);

  // API publique en lecture seule
  readonly isDarkMode = this.darkMode.asReadonly();

  // Computed values
  readonly theme = computed(() =>
    this.darkMode() ? 'dark' : 'light'
  );

  toggleTheme() {
```

Utilisation dans les composants

```
@Component({
  selector: 'app-theme-switcher',
  template: `
    <button (click)="toggleTheme()">
      Mode {{ themeService.isDarkMode() ? 'Clair' : 'Sombre' }}
    </button>
  `,
})
export class ThemeSwitcherComponent {
  constructor(public themeService: ThemeService) {}

  toggleTheme() {
    this.themeService.toggleTheme();
  }
}
```

Service avec gestion d'état avancée

```
interface AppState {  
  user: User | null;  
  preferences: UserPreferences;  
  notifications: Notification[];  
}  
  
@Injectable({  
  providedIn: 'root'  
})  
export class StateService {  
  private state = signal<AppState>({  
    user: null,  
    preferences: defaultPreferences,  
    notifications: []  
  });  
};
```

Service avec effets

```
@Injectable({
  providedIn: 'root'
})
export class AuthService {
  private user = signal<User | null>(null);

  constructor() {
    // Effet pour synchroniser avec le localStorage
    effect(() => {
      const currentUser = this.user();
      if (currentUser) {
        localStorage.setItem('user', JSON.stringify(currentUser));
      } else {
        localStorage.removeItem('user');
      }
    })
  }
}
```

Service avec injection hiérarchique

```
// Service au niveau du module/composant
@Injectable()
export class FeatureService {
  private data = signal<any[]>([]);

  // Méthodes spécifiques à la fonctionnalité
}

@Component({
  selector: 'app-feature',
  providers: [FeatureService], // Injection locale
  template: `
    <div>
      Feature Component
      <child-component></child-component>
    </div>
  `
})
```



Exercice : Service de Blog

1. Créez l'interface Post :

```
// features/posts/post.model.ts
export interface Post {
  id: number
  title: string
  content: string
  excerpt: string
  author: string
  date: Date
}
```


2. Créez le service PostService :

```
// features/posts/post.service.ts
@Injectable({
  providedIn: 'root'
})
export class PostService {
  private posts = signal<Post[]>([])
  readonly allPosts = this.posts.asReadonly()

  constructor(private http: HttpClient) {}

  loadPosts() {
    return this.http.get<Post[]>('/api/posts').pipe(
      tap(posts => this.posts.set(posts))
    )
  }
}
```



Routing et Navigation

Configuration des routes modernes

```
// app.routes.ts
export const routes: Routes = [
  {
    path: '',
    component: HomeComponent
  },
  {
    path: 'products',
    loadChildren: () => import('./products/routes')
  },
  {
    path: 'profile',
    canActivate: [authGuard],
    loadChildren: () => import('./profile.component')
  }
]
```

Routes avec Signals

```
// products/routes.ts
export default [{
  path: '',
  component: ProductListComponent,
  resolve: {
    products: () => inject(ProductService).getProducts()
  }
}, {
  path: ':id',
  component: ProductDetailComponent,
  resolve: {
    product: (route: ActivatedRoute) => {
      const id = route.paramMap.pipe(map(params => params.get('id')));
      return inject(ProductService).getProduct(id);
    }
  }
}]
```

Navigation programmatique

```
@Component({
  template: `
    <nav>
      <a routerLink="/products">Produits</a>
      <a routerLink="/profile">Profil</a>
    </nav>

    <button (click)="goToProduct(123)">
      Voir Produit
    </button>
  `,
})
export class NavComponent {
  constructor(private router: Router) {}
}
```

Guards modernes

```
// auth.guard.ts
export const authGuard: CanActivateFn = (
  route: ActivatedRouteSnapshot,
  state: RouterStateSnapshot
) => {
  const authService = inject(AuthService);
  const router = inject(Router);

  if (authService.isAuthenticated()) {
    return true;
  }

  return router.createUrlTree(['/login'], {
    queryParams: { returnUrl: state.url }
  });
};
```

View Transitions API

```
@Component({
  template: `
    <div @routeAnimations>
      <router-outlet></router-outlet>
    </div>
  `,
  animations: [
    trigger('routeAnimations', [
      transition('* <=> *', [
        style({ position: 'relative' }),
        query(':enter, :leave', [
          style({
            position: 'absolute',
            top: 0,
            left: 0,
```

Lazy Loading avec Preloading

```
// Configuration du preloading
bootstrapApplication(AppComponent, {
  providers: [
    provideRouter(routes,
      withPreloading(PreloadAllModules),
      withViewTransitions()
    )
  ]
});

// Route avec lazy loading
{
  path: 'dashboard',
  loadComponent: () => import('./dashboard/dashboard.component')
    .then(m => m.DashboardComponent),
```


Paramètres de route avec Signals

```
@Component({
  template: `
    <h1>Produit {{ productId() }}</h1>
    @if (product()) {
      <div>
        <h2>{{ product().name }}</h2>
        <p>{{ product().description }}</p>
      </div>
    }
  `,
})
export class ProductDetailComponent {
  private route = inject(ActivatedRoute);
  private productService = inject(ProductService);
```

Nested Routes avec Outlets multiples

```
// Configuration des routes
export const routes: Routes = [{
  path: 'dashboard',
  component: DashboardComponent,
  children: [{
    path: '',
    component: DashboardOverviewComponent
  }, {
    path: 'stats',
    component: StatsComponent,
    outlet: 'sidebar'
  }]
}];

// Template avec multiple outlets
```



Exercice : Routes du Blog

1. Configurez les routes :

```
// app.routes.ts
export const routes: Routes = [
  {
    path: '',
    component: HomeComponent
  },
  {
    path: 'posts',
    children: [
      {
        path: '',
        component: PostListComponent,
        resolve: {
          posts: () => inject(PostService).loadPosts()
        }
      }
    ]
  }
]
```

2. Créez le guard d'authentification :

```
// core/guards/auth.guard.ts
export const authGuard: CanActivateFn = () => {
  const authService = inject(AuthService)
  const router = inject(Router)

  if (authService.isAuthenticated()) {
    return true
  }

  return router.createUrlTree(['/login'], {
    queryParams: { returnUrl: router.url }
  })
}
```



Formulaire Angular

Types de formulaires

Formulaires dans Angular 18/19

Formulaires réactifs modernes avec Signals

```
@Component({
  selector: 'app-signup',
  template: `
    <form [formGroup]="form" (ngSubmit)="onSubmit()">
      <div>
        <label for="email">Email</label>
        <input id="email" type="email" formControlName="email">
        @if (emailErrors()) {
          <span class="error">{{ emailErrors() }}</span>
        }
      </div>

      <div>
        <label for="password">Mot de passe</label>
        <input id="password" type="password" formControlName="password">
      </div>
    </form>
  `
})
```

Validation personnalisée

```
// Valideur personnalisé
function passwordStrength(): ValidatorFn {
  return (control: AbstractControl): ValidationErrors | null => {
    const value = control.value;

    if (!value) return null;

    const hasNumber = /\d/.test(value);
    const hasUpper = /[A-Z]/.test(value);
    const hasLower = /[a-z]/.test(value);
    const hasSpecial = /[!@#$%^&*]/.test(value);

    const valid = hasNumber && hasUpper && hasLower && hasSpecial;

    return valid ? null : {
```


Formulaires dynamiques

```
interface DynamicField {  
  name: string;  
  label: string;  
  type: 'text' | 'email' | 'number';  
  validators: ValidatorFn[];  
}  
  
@Component({  
  template: `  
    <form [formGroup]="form" (ngSubmit)="onSubmit()">  
      @for (field of fields; track field.name) {  
        <div>  
          <label [for]="field.name">{{ field.label }}</label>  
          <input  
            [id]="field.name"
```

Formulaires imbriqués

```
@Component({
  template: `
    <form [formGroup]="form" (ngSubmit)="onSubmit()">
      <div formGroupName="personal">
        <input formControlName="firstName">
        <input formControlName="lastName">
      </div>

      <div formGroupName="address">
        <input formControlName="street">
        <input formControlName="city">
        <input formControlName="zipCode">
      </div>

      <div formArrayName="phones">
```

Exercice : Formulaire d'inscription complet

Créez un formulaire d'inscription avec validation avancée :

```
interface RegistrationForm {  
  personal: {  
    firstName: string;  
    lastName: string;  
    email: string;  
  };  
  credentials: {  
    password: string;  
    confirmPassword: string;  
  };  
  preferences: {  
    newsletter: boolean;  
    notifications: string[];  
  };  
}
```

Cet exercice vous à permis de pratiquer :

- Les formulaires imbriqués
- La validation personnalisée
- La validation croisée
- La gestion des FormArray
- Les états de chargement avec Signals



Exercice : Formulaire de Post

1. Structure du Template

```
// features/posts/post-form.component.ts
@Component({
  selector: 'app-post-form',
  standalone: true,
  template: `
    <form [formGroup]="form" (ngSubmit)="onSubmit()">
      <div class="form-field">
        <label for="title">Titre</label>
        <input id="title" type="text" formControlName="title">
        @if (titleErrors()) {
          <span class="error">{{ titleErrors() }}</span>
        }
      </div>

      <div class="form-field">
```

2. Configuration du Formulaire

```
export class PostFormComponent {  
  private postService = inject(PostService)  
  private router = inject(Router)  
  
  form = new FormGroup({  
    title: new FormControl('', [  
      Validators.required,  
      Validators.minLength(3)  
    ]),  
    content: new FormControl('', [  
      Validators.required,  
      Validators.minLength(50)  
    ])  
  })  
}
```

3. Gestion des Erreurs

```
titleErrors = computed(() => {
  const control = this.form.get('title')
  if (control?.errors && control.touched) {
    if (control.errors['required']) return 'Le titre est requis'
    if (control.errors['minlength']) return 'Le titre doit faire au moins 3 caractères'
  }
  return null
})

contentErrors = computed(() => {
  const control = this.form.get('content')
  if (control?.errors && control.touched) {
    if (control.errors['required']) return 'Le contenu est requis'
    if (control.errors['minlength']) return 'Le contenu doit faire au moins 50 caractères'
  }
  return null
})
```


4. Soumission du Formulaire

```
async onSubmit() {  
  if (this.form.valid) {  
    this.saving.set(true)  
    try {  
      await firstValueFrom(this.postService.createPost({  
        ...this.form.value,  
        author: 'Utilisateur actuel',  
        date: new Date(),  
        excerpt: this.form.value.content?.slice(0, 100) + '...'  
      })))  
      this.router.navigate(['/posts'])  
    } finally {  
      this.saving.set(false)  
    }  
  }  
}
```



RxJS et Observables

Concepts de base

Bon déjà c'est quoi un Observable ?

Un Observable est un objet qui représente une séquence de valeurs, émises à des moments différents.

Imaginons je fais un appel api , regardons du marble "testing" pour mieux comprendre :

```
const apiCall$ = new Observable(observer => {  
  fetch('https://api.example.com/data')  
    .then(response => response.json())  
    .then(data => observer.next(data))  
    .catch(error => observer.error(error))  
    .finally(() => observer.complete())  
})
```

Donc ce flux : --1-2-3-4-5-

ce qui veut dire :

- 1 première étape : j'aurais donc une réponse de l'api
- 2 deuxième étape : j'aurais donc une autre réponse de l'api
- 3 troisième étape : j'aurais donc une autre réponse de l'api
- 4 quatrième étape : j'aurais donc une autre réponse de l'api
- 5 cinquième étape : j'aurais donc une autre réponse de l'api

Observer

Un observer est un objet qui écoute les événements d'un Observable.

```
// Observable simple
const numbers$ = of(1, 2, 3, 4, 5)

// Observer
numbers$.subscribe({
  next: value => console.log(value),
  error: err => console.error(err),
  complete: () => console.log('Terminé')
})
```

Types de Subjects

```
// Subject basique
const subject = new Subject<string>()
subject.subscribe(value => console.log('A:', value))
subject.subscribe(value => console.log('B:', value))
subject.next('Hello') // A: Hello, B: Hello

// BehaviorSubject - Garde la dernière valeur
const behavior = new BehaviorSubject<number>(0)
behavior.subscribe(value => console.log('Valeur actuelle:', value))
console.log('Valeur stockée:', behavior.value)

// ReplaySubject - Rejoue X valeurs
const replay = new ReplaySubject<string>(2)
replay.next('Un')
replay.next('Deux')
```

Opérateurs essentiels - Filtrage

```
const source$ = interval(1000)

// filter - Garde uniquement les valeurs qui passent le prédicat
source$.pipe(
  filter(n => n % 2 === 0)
) // 0, 2, 4, 6...

// take - Prend X valeurs puis complète
source$.pipe(
  take(3)
) // 0, 1, 2, complete

// takeUntil - Émet jusqu'à ce qu'un autre Observable émette
const stop$ = timer(5000)
source$.pipe(
```

Opérateurs essentiels - Transformation

```
// map - Transforme chaque valeur
source$.pipe(
  map(n => n * 2)
) // 0, 2, 4, 6...

// switchMap - Annule l'Observable précédent
searchTerm$.pipe(
  switchMap(term =>
    this.http.get(`/api/search?q=${term}`)
  )
)

// mergeMap - Fusionne tous les Observables
userIds$.pipe(
  mergeMap(id =>
```


Opérateurs essentiels - Combinaison

```
// combineLatest - Combine les dernières valeurs
combineLatest({
  user: userProfile$,
  preferences: userPrefs$,
  theme: themeSettings$
}).subscribe(({ user, preferences, theme }) => {
  console.log('État complet:', { user, preferences, theme })
})

// merge - Fusionne plusieurs Observables
merge(
  clicks$,
  keypresses$,
  touches$
).subscribe(event => {
```

Cas d'utilisation concrets

1. Recherche en temps réel

```
@Component({
  template: `
    <input [ngModel]="searchTerm()"
      (ngModelChange)="searchTerm$.next($event)">
    <div *ngFor="let result of results()">
      {{ result.name }}
    </div>
  `,
})
class SearchComponent {
  private searchTerm$ = new BehaviorSubject<string>('')

  results = toSignal(
    this.searchTerm$.pipe(
      debounceTime(300),
```

2. Gestion des websockets avec reconnexion

```
class WebSocketService {  
  private wsSubject = new BehaviorSubject<WebSocket | null>(null)  
  private messagesSubject = new Subject<any>()  
  private reconnectAttempts = 0  
  
  connect() {  
    const ws = new WebSocket('ws://api.example.com')  
  
    ws.addEventListener('message', event => {  
      this.messagesSubject.next(JSON.parse(event.data))  
    })  
  
    ws.addEventListener('close', () => {  
      this.reconnect()  
    })  
  }  
}
```

Introduction à RxJS

Intégration avec Signals

```
@Component({
  template: `
    <div>Données: {{ data() }}</div>
    <div>Statut: {{ status() }}</div>
  `,
})
export class DataComponent {
  private dataService = inject(DataService);

  // Conversion d'Observable en Signal
  data = toSignal(
    this.dataService.getData().pipe(
      catchError(error => {
        console.error('Erreur:', error);
        return of(null);
      })
    )
  );
}
```

Opérateurs RxJS modernes

```
@Injectable({
  providedIn: 'root'
})
export class SearchService {
  private http = inject(HttpClient);

  search(term: string): Observable<Result[]> {
    return of(term).pipe(
      // Attendre que l'utilisateur arrête de taper
      debounceTime(300),

      // Ignorer si le terme est le même
      distinctUntilChanged(),

      // Annuler la requête précédente
    );
  }
}
```

Gestion des souscriptions

```
@Component({
  template: `
    <input [ngModel]="searchTerm()"
      (ngModelChange)="searchTerm.set($event)">

    @if (results(); as items) {
      @for (item of items; track item.id) {
        <div>{{ item.name }}</div>
      }
    }
  `,
})
export class SearchComponent implements OnDestroy {
  private searchService = inject(SearchService);
  private destroy$ = new Subject<void>();
```

Combinaison d'Observables

```
@Component({
  template: `
    <div>
      Utilisateur: {{ userData()?.name }}
      Préférences: {{ userData()?.preferences }}
      Notifications: {{ userData()?.notifications }}
    </div>
  `,
})
export class UserDashboardComponent {
  private userService = inject(UserService);
  private prefService = inject(PreferencesService);
  private notifService = inject(NotificationService);

  userData = toSignal(
```

Gestion des erreurs avancée

```
@Injectable({
  providedIn: 'root'
})
export class ErrorHandlingService {
  private errorSubject = new Subject<Error>();

  error$ = this.errorSubject.asObservable().pipe(
    // Grouper les erreurs similaires
    groupBy(error => error.message),
    // Limiter les notifications
    mergeMap(group => group.pipe(
      debounceTime(1000),
      take(3)
    ))
  );
}
```


WebSocket avec RxJS

```
@Injectable({
  providedIn: 'root'
})
export class WebSocketService {
  private socket$ = new WebSocket('ws://example.com');
  private messagesSubject = new Subject<any>();

  messages$ = this.messagesSubject.asObservable().pipe(
    // Reconnexion automatique
    retryWhen(errors => errors.pipe(
      tap(error => console.error('WebSocket error:', error)),
      delay(1000)
    )),
    // Filtrage et transformation des messages
    filter(msg => msg.type === 'data'),
```



Exercice : Recherche de Posts

1. Créez le composant de recherche :

```
// features/posts/post-search.component.ts
@Component({
  selector: 'app-post-search',
  standalone: true,
  template: `
    <div class="search">
      <input
        type="text"
        [ngModel]="searchTerm()"
        (ngModelChange)="searchTerm.set($event)"
        placeholder="Rechercher..."
      >

    @if (loading()) {
      <spinner />
    }
  `
})
```

Opérateurs courants

```
// Création d'un Observable
const numbers$ = new Observable<number>(subscriber => {
  subscriber.next(1);
  subscriber.next(2);
  subscriber.next(3);
  subscriber.complete();
});

// Souscription
numbers$.subscribe({
  next: value => console.log(value),
  error: err => console.error(err),
  complete: () => console.log('Terminé')
});
```

Opérateurs courants

```
// map, filter, tap
const numbers$ = of(1, 2, 3, 4, 5);

numbers$.pipe(
  map(x => x * 2),
  filter(x => x > 4),
  tap(x => console.log('Valeur:', x))
).subscribe();
```

Combinaison d'Observables

```
// combineLatest, merge, forkJoin
const user$ = this.userService.getUser();
const posts$ = this.postService.getPosts();

combineLatest({
  user: user$,
  posts: posts$
}).subscribe(({ user, posts }) => {
  console.log('User:', user);
  console.log('Posts:', posts);
});
```

Gestion des erreurs

```
this.http.get('/api/data').pipe(  
  catchError(error => {  
    console.error('Erreur:', error);  
    return of({ error: true }); // Valeur par défaut  
  }),  
  retry(3), // Réessaie 3 fois  
  timeout(5000) // Timeout après 5s  
)  
.subscribe();
```

Subjects et BehaviorSubjects

```
@Injectable({
  providedIn: 'root'
})
export class ThemeService {
  private themeSubject = new BehaviorSubject<'light' | 'dark'>('light');
  theme$ = this.themeSubject.asObservable();

  setTheme(theme: 'light' | 'dark') {
    this.themeSubject.next(theme);
  }
}
```


RxJS avec Signals

```
@Component({
  template: `
    @if (users()); as users) {
      @for (user of users; track user.id) {
        <div>{{ user.name }}</div>
      }
    }
  `,
})
export class UserListComponent {
  private userService = inject(UserService);

  users = toSignal(
    this.userService.getUsers().pipe(
      catchError(() => of([])),
      shareReplay(1)
    ),
    { initialValue: [] }
  );
}
```

HTTP et Communication

Le HttpClient

HTTP Client dans Angular 18/19

Configuration moderne du HttpClient

```
// app.config.ts
export const appConfig: ApplicationConfig = {
  providers: [
    provideHttpClient(
      withInterceptors([authInterceptor]),
      withFetch(), // Nouvelle API Fetch
      withJsonpSupport()
    )
  ]
};
```

Service HTTP avec Signals

```
@Injectable({
  providedIn: 'root'
})
export class ApiService {
  private http = inject(HttpClient);
  private baseUrl = 'https://api.example.com';

  // États avec Signals
  private loading = signal(false);
  private error = signal<string | null>(null);

  // Getters publics en lecture seule
  readonly isLoading = this.loading.asReadonly();
  readonly currentError = this.error.asReadonly();
```

Intercepteurs modernes

```
// auth.interceptor.ts
export const authInterceptor: HttpInterceptorFn = (req, next) => {
  const authToken = localStorage.getItem('token');

  if (authToken) {
    const authReq = req.clone({
      headers: req.headers.set('Authorization', `Bearer ${authToken}`)
    });

    return next(authReq).pipe(
      catchError(error => {
        if (error.status === 401) {
          // Redirection vers login
          inject(Router).navigate(['/login']);
        }
      })
    );
  }
}
```

Gestion des erreurs HTTP

```
@Injectable({
  providedIn: 'root'
})
export class ErrorHandlingService {
  handleHttpError(error: HttpResponse): Observable<never> {
    let errorMessage = 'Une erreur est survenue';

    if (error.error instanceof ErrorEvent) {
      // Erreur côté client
      errorMessage = error.error.message;
    } else {
      // Erreur côté serveur
      switch (error.status) {
        case 404:
          errorMessage = 'Ressource non trouvée';
```

Requêtes parallèles

```
@Component({
  template: `
    @if (data(); as result) {
      <div>
        <h2>Utilisateur: {{ result.user.name }}</h2>
        <p>Posts: {{ result.posts.length }}</p>
        <p>Commentaires: {{ result.comments.length }}</p>
      </div>
    }
  `,
})
export class UserDataComponent {
  private http = inject(HttpClient);

  data = toSignal(
```

Upload de fichiers

```
@Injectable({
  providedIn: 'root'
})
export class FileUploadService {
  private http = inject(HttpClient);

  uploadFile(file: File) {
    const formData = new FormData();
    formData.append('file', file);

    return this.http.post('/api/upload', formData, {
      reportProgress: true,
      observe: 'events'
    }).pipe(
      map(event => {
```


Cache HTTP avec Signals

```
@Injectable({
  providedIn: 'root'
})
export class CacheService {
  private cache = signal<Map<string, any>>(new Map());
  private http = inject(HttpClient);

  getData<T>(url: string, options: { ttl?: number } = {}) {
    const cached = this.cache().get(url);

    if (cached && (!options.ttl || Date.now() - cached.timestamp < options.ttl)) {
      return of(cached.data);
    }

    return this.http.get<T>(url).pipe(
```



Exercice : Service API du Blog

1. Créez le service API :

```
// core/services/api.service.ts
@Injectable({
  providedIn: 'root'
})
export class ApiService {
  private http = inject(HttpClient)
  private baseUrl = 'https://api.blog.com'

  private loading = signal(false)
  private error = signal<string | null>(null)

  readonly isLoading = this.loading.asReadonly()
  readonly currentError = this.error.asReadonly()

  constructor() {
```

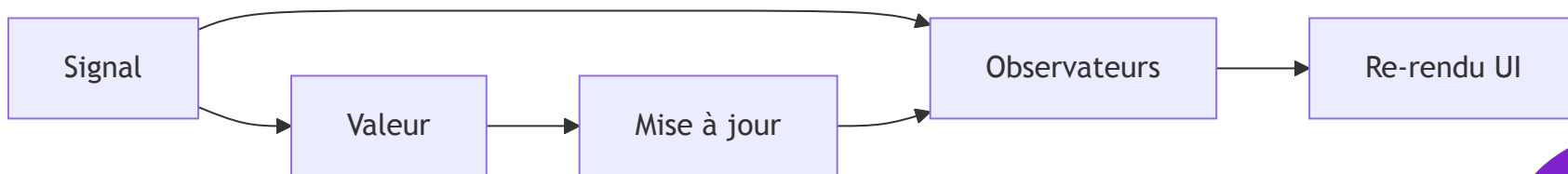


Signals

Introduction aux Signals

Un Signal est comme une "boîte réactive" qui :

- Contient une valeur
- Notifie automatiquement quand cette valeur change
- Permet un suivi précis des dépendances



Comparaison avec les variables classiques

Imaginons un thermomètre :

Sans Signal :

```
// La température change mais personne n'est notifié
class Thermometer {
    temperature = 20;

    update() {
        this.temperature++; // L'affichage ne sait pas qu'il doit se mettre à jour
    }
}
```

Avec Signal :

```
class Thermometer {
    temperature = signal(20);

    update() {
        this.temperature.update(t => t + 1); // Notification automatique !
    }
}
```

[Revenir au sommaire](#)

Les différents types de Signals

1. Signal Writable

```
// Comme une variable qu'on peut lire et modifier
const counter = signal(0);

// Lecture
console.log(counter()); // 0

// Écriture
counter.set(5);           // Remplacement direct
counter.update(n => n + 1); // Mise à jour basée sur valeur précédente
```

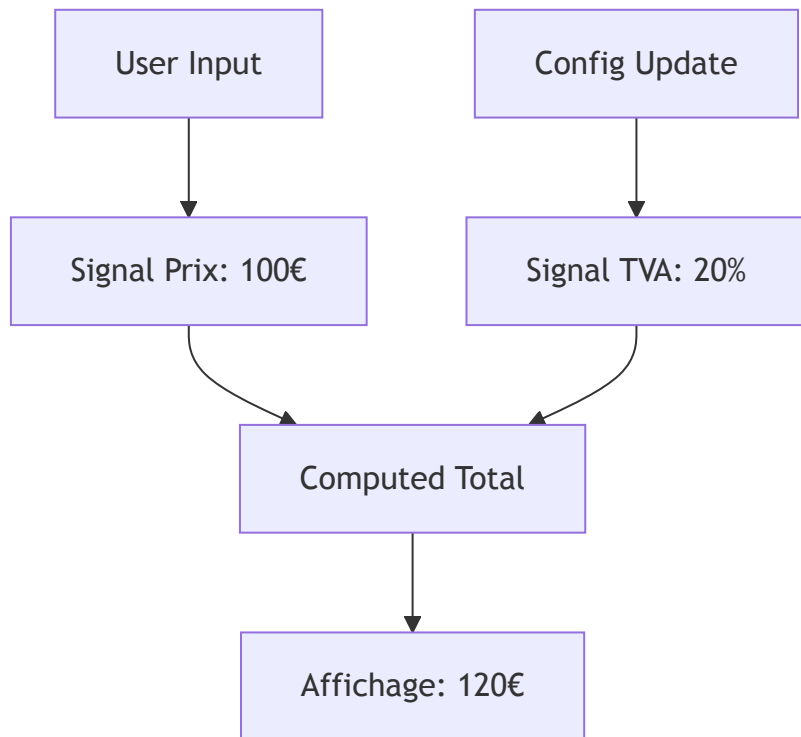
2. Signal Computed

```
// Comme une formule Excel qui se recalcule automatiquement
const price = signal(100);
const taxRate = signal(0.2);

const total = computed(() => {
  const basePrice = price();
  const tax = basePrice * taxRate();
  return basePrice + tax;
});
```

[Revenir au sommaire](#)

Visualisation du flux de données



Exemple concret : Panier d'achat

```
@Component({
  template: `
    <div class="cart-summary">
      <h2>Votre panier</h2>

      <!-- Affichage réactif des totaux -->
      <div class="totals">
        <p>Sous-total: {{ subtotal() }}€</p>
        <p>TVA ({{ vatRate() * 100 }}%): {{ vatAmount() }}€</p>
        <p class="grand-total">Total: {{ total() }}€</p>
      </div>

      <!-- La mise à jour d'un produit recalcule automatiquement tous les totaux -->
      @for (item of cartItems(); track item.id) {
        <cart-item
```

Bonnes pratiques avec les Signals

✓ À faire

- Utiliser `computed()` pour les valeurs dérivées
- Préférer `.update()` à `.set()` pour les modifications basées sur l'état précédent
- Garder les signals privés quand possible

✗ À éviter

- Créer des signals dans des boucles
- Appeler des signals dans des setters
- Modifier plusieurs signals de manière non atomique

Computed Signals

```
const count = signal(0);  
const doubled = computed(() => count() * 2);  
const isEven = computed(() => count() % 2 === 0);
```

Effects

```
@Component({
  template: `
    <button (click)="increment()">
      Compteur: {{ count() }}
    </button>
  `,
})
export class CounterComponent {
  count = signal(0);

  constructor() {
    // L'effet se déclenche à chaque changement de count
    effect(() => {
      console.log(`Nouvelle valeur: ${this.count()}`);
      localStorage.setItem('count', this.count().toString());
    });
  }
}
```

Signals avec objets et tableaux

```
interface User {  
  id: number;  
  name: string;  
  email: string;  
}  
  
@Component({  
  template: `  
    <div>  
      <h2>{{ user().name }}</h2>  
      @for (friend of friends(); track friend.id) {  
        <div>{{ friend.name }}</div>  
      }  
    </div>  
  `,  
})
```

Signals avec async data

```
@Component({
  template: `
    @if (loading()) {
      <spinner />
    } @else if (error()) {
      <error-message [message]="error()" />
    } @else if (data(); as users) {
      @for (user of users; track user.id) {
        <user-card [user]="user" />
      }
    }
  `,
})
export class UsersComponent {
  private userService = inject(UserService);
```

Signal Inputs

```
@Component({
  selector: 'app-user-profile',
  template: `
    <div>
      <h2>{{ name() }}</h2>
      <p>Age: {{ age() }}</p>
      <p>Score: {{ score() }}</p>
    </div>
  `,
})
export class UserProfileComponent {
  name = input.required<string>();
  age = input<number>(18); // Valeur par défaut
  score = input<number>(); // Optionnel
}
```

Signals avec formulaires

```
@Component({
  template: `
    <form [formGroup]="form" (ngSubmit)="onSubmit()">
      <input formControlName="name">
      <input formControlName="email">

      <div>Form Value: {{ formValue() | json }}</div>
      <div>Valid: {{ isValid() }}</div>

      <button type="submit" [disabled]="!isValid()">
        Envoyer
      </button>
    </form>
  `,
})
```

Gestion de l'État avec Signals

Avant les Signals

```
@Component({
  template: `
    <h1>{{ count }}</h1>
    <button (click)="increment()">+1</button>
  `,
})
class CounterComponent {
  count = 0

  increment() {
    this.count++ // Déclenche détection globale
  }
}
```


Avec les Signals

```
@Component({
  template: `
    <h1>{{ count() }}</h1>
    <button (click)="increment()">+1</button>
  `,
})
class CounterComponent {
  count = signal(0)

  increment() {
    this.count.update(n => n + 1) // Mise à jour granulaire
  }
}
```



Exercice : État Global du Blog

1. Créez le service d'état :

```
// core/state/blog.state.ts
interface BlogState {
  posts: Post[]
  selectedPost: Post | null
  filters: {
    search: string
    category: string
  }
  loading: boolean
  error: string | null
}

@Injectable({ providedIn: 'root' })
export class BlogStateService {
  // État privé
```

2. Utilisez l'état dans un composant :

```
@Component({
  selector: 'app-post-list',
  template: `
    <div class="filters">
      <input
        type="text"
        [ngModel]="searchTerm()"
        (ngModelChange)="onSearch($event)"
        placeholder="Rechercher..."
      >
      <select
        [ngModel]="category()"
        (ngModelChange)="onCategoryChange($event)"
      >
        <option value="all">Toutes catégories</option>
        <option value="tech">Tech</option>
        <option value="lifestyle">Lifestyle</option>
      </select>
    </div>

    @if (loading()) {
      <spinner />
    } @else if (error()) {
      <error-message [message]="error()" />
    } @else {
```



Directives

Qu'est ce qu'une directive ?

Une directive est un élément qui modifie le comportement ou la structure d'un élément HTML.

A vrai dire, depuis le début de la formation, vous en avez utilisé plusieurs :

- `*ngIf` maintenant `@if`
- `*ngFor` maintenant `@for`
- `*ngSwitch` maintenant `@switch`
- `*ngStyle` maintenant `[ngStyle]`
- `*ngClass` maintenant `[ngClass]`

Mais aussi des pipes sont des directives :

- `date`
- `uppercase`
- `lowercase`
- `currency`
- `slice`

Types de directives

Il existe plusieurs types de directives :

- Directives d'attributs personnalisées
- Directives de structure
- Pipes

Mais vous en avez d'autres, des directives personnalisées, des pipes personnalisés, des directives avec injection de dépendances, etc. Ce que je sous entend par là, c'est aussi que vous pouvez en créer vous même.

Types de directives

```
// Nouvelle syntaxe de control flow (Angular 18+)
@Component({
  template: `
    @if (isLoggedIn()) {
      <nav>Menu utilisateur</nav>
    } @else {
      <auth-form />
    }

    @for (item of items(); track item.id) {
      <item-card [data]="item" />
    }

    @switch (status()) {
      @case ('loading') {
```


Directives d'attributs personnalisées

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

  constructor(private el: ElementRef) {}

  @HostListener('mouseenter')
  onMouseEnter() {
    this.highlight(this.highlightColor || 'yellow');
  }
}
```

Utilisation des directives

```
@Component({
  selector: 'app-root',
  template: `
    <div appHighlight="lightblue">
      Survolez-moi !
    </div>

    <p [appHighlight]="color">
      Couleur dynamique
    </p>
  `,
  imports: [HighlightDirective]
})
export class AppComponent {
  color = 'lightgreen';
}
```

Pipes personnalisés

```
// time-ago.pipe.ts
@Pipe({
  name: 'timeAgo',
  standalone: true
})
export class TimeAgoPipe implements PipeTransform {
  transform(value: Date | string): string {
    const date = new Date(value);
    const now = new Date();
    const seconds = Math.floor((now.getTime() - date.getTime()) / 1000);

    if (seconds < 60) {
      return 'Il y a quelques secondes';
    }

    if (seconds < 3600) {
```

Utilisation des pipes

```
@Component({
  selector: 'app-post',
  template: `
    <article>
      <h2>{{ title | uppercase }}</h2>
      <p>Publié {{ date | timeAgo }}</p>
      <p>Prix: {{ price | currency:'EUR' }}</p>
      <p>{{ content | slice:0:100 }}...</p>
    </article>
  `,
  imports: [TimeAgoPipe]
})
export class PostComponent {
  title = 'Mon article';
  date = new Date();
}
```

Pipes avec Signals

```
@Pipe({
  name: 'filter',
  standalone: true
})
export class FilterPipe implements PipeTransform {
  transform(items: Signal<any[]>, searchTerm: Signal<string>): Signal<any[]> {
    return computed(() => {
      const term = searchTerm().toLowerCase();
      return items().filter(item =>
        item.name.toLowerCase().includes(term)
      );
    });
  }
}
```

Directive avec Injection de dépendances

```
@Directive({
  selector: '[appTooltip]',
  standalone: true
})
export class TooltipDirective implements OnDestroy {
  @Input('appTooltip') text = '';

  private tooltip: HTMLElement | null = null;

  constructor(
    private el: ElementRef,
    private renderer: Renderer2,
    private zone: NgZone
  ) {}
```

Exercice : Création d'une directive de validation

Créez une directive de validation personnalisée pour les formulaires :

```
@Directive({
  selector: '[appPasswordStrength]',
  standalone: true,
  providers: [{
    provide: NG_VALIDATORS,
    useExisting: PasswordStrengthDirective,
    multi: true
  }]
})
export class PasswordStrengthDirective implements Validator {
  validate(control: AbstractControl): ValidationErrors | null {
    const value = control.value;

    if (!value) {
      return null;
    }
  }
}
```

Cet exercice vous a permis de comprendre comment créer des directives de validation personnalisées et les intégrer avec les formulaires Angular.

Nouveau Control Flow (Angular 18/19)

```
@Component({  
  template: `  
    @if (isLoading()) {  
      Loading...  
    } @else {  
      @for (item of items(); track item.id) {  
        {{ item }}  
      }  
    }  
  `,  
})
```




Performance et Optimisation

Stratégies d'optimisation

```
// Avant (avec Zone.js)
@Component({
  template: `
    <div>{{ value }}</div>
    <button (click)="increment()">+</button>
  `,
})
class OldComponent {
  value = 0;

  increment() {
    this.value++; // Déclenche la détection des changements globale
  }
}
```

Optimisation des templates

```
@Component({
  template: `
    <!-- Utilisation de @if au lieu de *ngIf -->
    @if (showContent()) {
      <heavy-component />
    }

    <!-- Utilisation de @for au lieu de *ngFor -->
    @for (item of items(); track item.id) {
      <item-component [data]="item" />
    }

    <!-- Defer loading -->
    @defer {
      <expensive-component />
    }
  `
})
```

Lazy Loading moderne

```
// app.routes.ts
export const routes: Routes = [{
  path: 'admin',
  loadChildren: () => import('./admin/routes'),
  canActivate: [AuthGuard],
  providers: [
    importProvidersFrom(AdminModule)
  ]
}];
```

```
// Composant lazy-loadé
@Component({
  standalone: true,
  imports: [CommonModule],
  template: `
```

Optimisation des images

```
@Component({
  template: `
    <img
      ngSrc="{{ imageUrl }}"
      width="300"
      height="200"
      priority
      loading="eager"
    />

    @for (image of images(); track image) {
      <img
        ngSrc="{{ image.url }}"
        [width]="image.width"
        [height]="image.height"
```

Memoization avec Signals

```
@Component({
  template: `
    <div>Résultat filtré: {{ filteredData() }}</div>
  `,
})
export class MemoizedComponent {
  data = signal<number[]>([]);
  threshold = signal(10);

  // Computed value avec memoization intégrée
  filteredData = computed(() => {
    console.log('Calcul coûteux effectué');
    return this.data().filter(n => n > this.threshold());
  });
}
```

Optimisation des requêtes HTTP

```
@Injectable({
  providedIn: 'root'
})
export class OptimizedApiService {
  private cache = new Map<string, any>();
  private http = inject(HttpClient);

  getData<T>(url: string, options: { ttl?: number } = {}) {
    const cached = this.cache.get(url);
    if (cached && (!options.ttl || Date.now() - cached.timestamp < options.ttl)) {
      return of(cached.data);
    }

    return this.http.get<T>(url).pipe(
      tap(data => {
```

Virtual Scrolling

```
@Component({
  template: `
    <cdk-virtual-scroll-viewport
      itemSize="50"
      class="viewport"
    >
      @for (item of items(); track item.id; let i = $index) {
        <div class="item">
          {{ i }} - {{ item.name }}
        </div>
      }
    </cdk-virtual-scroll-viewport>
  `,
  styles: [
    .viewport {
```


Web Workers

```
// worker.ts
/// <reference lib="webworker" />

addEventListener('message', ({ data }) => {
  const result = expensiveCalculation(data);
  postMessage(result);
});

// component.ts
@Component({
  template: `
    <button (click)="startCalculation()">
      Calculer
    </button>
    @if (result()) {
```

Exercice : Optimisation d'une liste de produits

Créez une liste de produits optimisée avec :

- Virtual scrolling
- Lazy loading des images
- Mise en cache des données
- Filtrage optimisé

```
interface Product {  
  id: number;  
  name: string;  
  price: number;  
  image: string;  
  description: string;  
}  
  
@Component({  
  template: `  
    <div class="filters">  
      <input  
        [ngModel]="searchTerm()"  
        (ngModelChange)="searchTerm.set($event)"  
        placeholder="Rechercher..."
```

Cet exercice vous à permis de pratiquer :

- L'utilisation du virtual scrolling
- L'optimisation des images
- La gestion efficace des filtres avec Signals
- Le lazy loading des composants
- La mise en cache des données

Hydraton (Nouveauté Angular 18/19)

```
// app.config.ts
export const appConfig: ApplicationConfig = {
  providers: [
    provideClientHydration() // Active l'hydraton
  ]
};
```

Comment fonctionne l'Hydratation ?

- Réutilise le HTML du SSR
- Restaure l'état de l'application
- Évite le re-rendu complet
- Améliore le First Contentful Paint (FCP)

Defer Loading (Nouveauté Angular 18/19)

```
@Component({
  template: `
    @defer {
      <heavy-component />
    } @loading {
      <spinner />
    } @error {
      <error-message />
    } @placeholder {
      <div>Chargement...</div>
    }
  `,
})
```



Tests dans Angular

Introduction aux tests

Tests unitaires avec Jasmine et Karma

```
// user.service.spec.ts
describe('UserService', () => {
  let service: UserService;
  let httpMock: HttpTestingController;

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [HttpClientTestingModule],
      providers: [UserService]
    });

    service = TestBed.inject(UserService);
    httpMock = TestBed.inject(HttpTestingController);
  });
});
```

Tests de composants avec Signals

```
// counter.component.spec.ts
describe('CounterComponent', () => {
  let component: CounterComponent;
  let fixture: ComponentFixture<CounterComponent>;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      imports: [CounterComponent]
    }).compileComponents();

    fixture = TestBed.createComponent(CounterComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });
```

Tests d'intégration

```
// app.component.integration.spec.ts
describe('AppComponent (integration)', () => {
  let component: AppComponent;
  let fixture: ComponentFixture<AppComponent>;
  let userService: UserService;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      imports: [
        AppComponent,
        HttpClientTestingModule,
        RouterTestingModule
      ],
      providers: [UserService]
    }).compileComponents();
```

Tests E2E avec Cypress

```
// cypress/e2e/login.cy.ts
describe('Login Flow', () => {
  beforeEach(() => {
    cy.visit('/login');
  });

  it('should login successfully', () => {
    cy.intercept('POST', '/api/login', {
      statusCode: 200,
      body: { token: 'fake-token' }
    }).as('loginRequest');

    cy.get('[data-testid=email-input]')
      .type('user@example.com');
```

Tests de services avec Signals

```
// data.service.spec.ts
describe('DataService', () => {
  let service: DataService;

  beforeEach(() => {
    TestBed.configureTestingModule({
      providers: [DataService]
    });
    service = TestBed.inject(DataService);
  });

  it('should update data and notify subscribers', () => {
    // Test du signal
    expect(service.data()).toEqual([]);
  });
});
```

Tests de formulaires réactifs

```
// registration.component.spec.ts
describe('RegistrationComponent', () => {
  let component: RegistrationComponent;
  let fixture: ComponentFixture<RegistrationComponent>;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      imports: [
        ReactiveFormsModule,
        RegistrationComponent
      ]
    }).compileComponents();

    fixture = TestBed.createComponent(RegistrationComponent);
    component = fixture.componentInstance;
```

Exercice : Tests complets d'une fonctionnalité

Créez une suite de tests complète pour une fonctionnalité de panier d'achat :

```
// cart.service.spec.ts
describe('CartService', () => {
  let service: CartService;

  beforeEach(() => {
    TestBed.configureTestingModule({
      providers: [CartService]
    });
    service = TestBed.inject(CartService);
  });

  it('should add items to cart', () => {
    const item = { id: 1, name: 'Test', price: 10 };

    service.addItem(item);
```

Cet exercice vous à permis de pratiquer :

- Les tests unitaires de services
- Les tests de composants
- L'utilisation des Signals dans les tests
- Les tests d'intégration
- Les mocks et les spies



Déploiement

Build de production

```
# Build standard  
ng build --configuration production  
  
# Build avec SSR  
ng build && ng run my-app:server
```

Configuration des environnements

```
// environment.prod.ts
export const environment = {
  production: true,
  apiUrl: 'https://api.production.com',
  features: {
    analytics: true,
    monitoring: true
  }
};

// app.config.ts
import { environment } from './environments/environment';

export const appConfig: ApplicationConfig = {
  providers: [
```

Optimisation du bundle

```
// angular.json
{
  "projects": {
    "my-app": {
      "architect": {
        "build": {
          "builder": "@angular-devkit/build-angular:browser",
          "options": {
            "outputPath": "dist/my-app",
            "index": "src/index.html",
            "main": "src/main.ts",
            "polyfills": ["zone.js"],
            "tsConfig": "tsconfig.app.json",
            "assets": [
              "src/favicon.ico",
```

Server-Side Rendering (SSR)

```
// server.ts
import { APP_BASE_HREF } from '@angular/common';
import { CommonEngine } from '@angular/ssr';
import express from 'express';
import { fileURLToPath } from 'url';
import { dirname, join, resolve } from 'path';
import bootstrap from './src/main.server';

const app = express();
const port = process.env.PORT || 4000;
const __dirname = dirname(fileURLToPath(import.meta.url));
const distFolder = join(__dirname, '../dist/my-app/browser');

// Servir les fichiers statiques
app.use(express.static(distFolder));
```

Docker

```
# Dockerfile
# Stage 1: Build
FROM node:20-alpine as builder

WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build

# Stage 2: Run
FROM node:20-alpine

WORKDIR /app
COPY --from=builder /app/dist ./dist
```

CI/CD avec GitHub Actions

```
# .github/workflows/deploy.yml
name: Deploy

on:
  push:
    branches: [ main ]

jobs:
  deploy:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v4

      - name: Setup Node.js
```

Nginx Configuration

```
# nginx.conf
server {
    listen 80;
    server_name example.com;
    root /usr/share/nginx/html;
    index index.html;

    # Gzip compression
    gzip on;
    gzip_types text/plain text/css application/json application/javascript;
    gzip_min_length 1000;

    # Cache control
    location ~* \.(js|css|png|jpg|jpeg|gif|ico)$ {
        expires 1y;
```


Monitoring et Analytics

```
// app.config.ts
import { ApplicationConfig } from '@angular/core';
import { provideClientHydration } from '@angular/platform-browser';
import { provideRouter } from '@angular/router';
import { routes } from './app.routes';

export const appConfig: ApplicationConfig = {
  providers: [
    provideRouter(routes),
    provideClientHydration(),
    // Monitoring
    {
      provide: ErrorHandler,
      useClass: GlobalErrorHandler
    },
  ],
}
```

Exercice : Déploiement complet

Configurez un déploiement complet avec :

- Build optimisé
- SSR
- Docker
- CI/CD
- Monitoring

```
# 1. Préparation du build
npm run lint
npm test
npm run build:ssr

# 2. Construction de l'image Docker
docker build -t my-angular-app .
docker run -p 4000:4000 my-angular-app

# 3. Déploiement avec monitoring
```

Monitoring

```
// monitoring.service.ts
@Injectable({
  providedIn: 'root'
})
export class MonitoringService {
  private errorCount = signal(0);
  private performanceMetrics = signal<PerformanceMetrics>({
    fcp: 0,
    lcp: 0,
    cls: 0
  });

  trackError(error: Error) {
    this.errorCount.update(count => count + 1);
    // Envoi à un service de monitoring
  }
}
```



Déploiement avec Vercel

Configuration Vercel

1. Installez Vercel CLI :

```
npm i -g vercel
```

2. Connectez-vous à votre compte :

```
vercel login
```

Déploiement automatique

1. Créez un fichier `vercel.json` à la racine :

```
{
  "version": 2,
  "builds": [
    {
      "src": "dist/*",
      "use": "@vercel/static"
    }
  ],
  "routes": [
    {
      "src": "/(.*)",
      "dest": "/index.html"
    }
  ]
}
```

2. Configurez le build :

```
{
  "scripts": {
    "build": "ng build --configuration production"
  }
}
```

[Revenir au sommaire](#)

Intégration Continue

1. Connectez votre repo GitHub à Vercel
2. Activez les déploiements automatiques :
 - Sur chaque push sur main
 - Preview sur les Pull Requests
 - Rollback automatique en cas d'erreur
3. Variables d'environnement :

```
vercel env add PRODUCTION_API_URL
```

Optimisations Vercel

- Edge Functions pour l'API
- Image Optimization
- Analytics intégrées
- Monitoring temps réel
- Certificats SSL automatiques

```
// next.config.js
module.exports = {
  images: {
    domains: ['assets.example.com'],
  },
}
```


Commandes utiles

```
# Déploiement manuel  
vercel  
  
# Déploiement en production  
vercel --prod  
  
# Voir les logs  
vercel logs  
  
# Lister les déploiements  
vercel list  
  
# Supprimer un déploiement  
vercel remove <deployment-id>
```



Bonnes Pratiques

Architecture et organisation

```
// ❌ À éviter
@Component({
  template: `
    <div>
      <h1>{{ title }}</h1>
      <div *ngFor="let item of items">
        <!-- Logique complexe -->
      </div>
    </div>
  `,
})
class BigComponent {
  // Trop de responsabilités
}
```

Gestion de l'état

```
// ❌ État global mutable
class StateService {
    public data = [];

    updateData(newData) {
        this.data = newData; // Mutation directe
    }
}

// ✅ État immutable avec Signals
@Injectable({ providedIn: 'root' })
class StateService {
    private _data = signal<Data[]>([]);
    readonly data = this._data.asReadonly();
}
```

Performance

```
// ❌ Calcule inutiles
@Component({
  template: `
    <div>{{ heavyComputation() }}</div>
  `,
})
class SlowComponent {
  heavyComputation() {
    // Recalculé à chaque cycle
    return this.data.reduce((acc, val) => acc + val, 0);
  }
}

// ✅ Optimisé avec computed
@Component({
```

Gestion des formulaires

```
// ❌ Validation non structurée
@Component({
  template: `
    <form (ngSubmit)="onSubmit()">
      <input [(ngModel)]="email">
      <span *ngIf="!isValidEmail">Email invalide</span>
    </form>
  `,
})
class FormComponent {
  isValidEmail = true;

  validateEmail() {
    this.isValidEmail = /^[^@]+@[^@]+\.[^@]+$/.test(this.email);
  }
}
```

Gestion des erreurs

```
// ❌ Gestion d'erreur basique
@Injectable()
class ApiService {
  getData() {
    return this.http.get('/api/data').pipe(
      catchError(error => {
        console.error(error);
        return of(null);
      })
    );
  }
}
```

```
// ✅ Gestion d'erreur robuste
@Injectable()
```

Organisation du code

```
// ❌ Fichier surchargé
// huge-file.ts
@Component({
  // 500+ lignes de template
})
class HugeComponent {
  // 1000+ lignes de logique
}

// ✅ Organisation modulaire
// feature/
// └─ components/
//     └─ feature-list.component.ts
//     └─ feature-item.component.ts
//     └─ feature-filter.component.ts
```


Exercice : Refactoring

Refactorisez ce composant selon les bonnes pratiques :

```
// ❌ Avant
@Component({
  template: `
    <div>
      <h1>{{ title }}</h1>
      <div *ngFor="let user of users">
        <div (click)="selectUser(user)">
          {{ user.name }}
        </div>
      </div>
    </div>
  `
})
```