Angular

Présentation

Angular est un framework front-end développé par Google qui a connu une évolution majeure avec sa version 17, simplifiant considérablement le développement tout en améliorant les performances. Cette version marque un tournant vers une approche plus légère et intuitive du développement web.

La Syntaxe de base d'Angular

Angular utilise une combinaison de HTML et TypeScript avec quelques ajouts spéciaux.

Les principaux délimiteurs syntaxiques d'Angular :

Les principaux délimiteurs syntaxiques d'Angular :

- {{ }} - Interpolation
  - Fonction : Afficher des valeurs dans le template
  - Exemple : Bienvenue, {{ username }}

- [ ] - Property binding
  - Fonction : Lier des propriétés d'éléments HTML à des valeurs
  - Exemple : <img [src]="imageUrl">

- ( ) - Event binding
  - Fonction : Réagir aux événements utilisateur
  - Exemple : <button (click)="sauvegarder()">Enregistrer</button>

- [( )] - Two-way binding
  - Fonction : Synchroniser les données dans les deux sens
  - Exemple : **<input [(ngModel)]="username">**

- Directives structurelles
  - Fonction : Modifier la structure DOM avec la syntaxe *ngX
  - Exemple : <div *ngIf="produit">{{ produit.nom }}
  - Exemple : <div *ngFor="let item of items">{{ item.nom }}

- Blocs de contrôle (Angular 17+)
  - Fonction : Alternative moderne aux directives structurelles
  - Exemple : @if (produit) { {{ produit.nom }} }
  - Exemple : @for (item of items; track item.id) { {{ item.nom }} }

- ngModel dans Angular Récent (17+)

```
<div class="input-examples">
  <h3>Différentes façons d'utiliser ngModel</h3>

  <!-- 1. Syntaxe traditionnelle avec [(ngModel)] -->
  <div class="input-group">
    <label>Méthode classique [(ngModel)]:</label>
    <input type="text" [(ngModel)]="demoText1" placeholder="Écrivez quelque
chose...">
    <p>Valeur: {{ demoText1 }}</p>
  </div>

  <!-- 2. Syntaxe décomposée traditionnelle -->
  <div class="input-group">
    <label>Méthode décomposée classique:</label>
    <input type="text" [ngModel]="demoText2" (ngModelChange)="demoText2 =
$event"
      placeholder="Écrivez quelque chose...">
    <p>Valeur: {{ demoText2 }}</p>
  </div>

  <!-- 3. Nouvelle syntaxe simplifiée (Angular 17+) -->
  <div class="input-group">
    <label>Nouvelle syntaxe simplifiée:</label>
    <input type="text" ngModel="{{demoText3}}" (ngModelChange)="demoText3 =
$event"
      placeholder="Écrivez quelque chose...">
    <p>Valeur: {{ demoText3 }}</p>
  </div>
</div>
```

L'organisation en Module et Composant

Angular : Un framework orienté composant

Angular est fondamentalement un framework orienté composant, **ce qui constitue l'un de ses principes architecturaux les plus importants :**

- Approche composant-first : Dans Angular, tout est composant - de la page entière aux plus petits éléments d'interface.
- Encapsulation : Chaque composant encapsule son template (HTML), sa logique (TypeScript) et ses styles (CSS/SCSS), créant ainsi des unités autonomes et réutilisables.
- Composition : Les applications Angular sont construites en composant des composants plus petits pour former des interfaces complexes.

Les modules

Évolution de l'architecture modulaire d'Angular:

I. L'approche traditionnelle par modules (pré-Angular 14)

Historiquement, Angular imposait une organisation stricte basée sur les *NgModules*.

Cette structure nécessitait :

- Déclaration de tous les composants dans un module
- Importation explicite de tous les modules nécessaires
- Organisation hiérarchique avec un AppModule racine
- Enregistrement des services dans les providers

Exemple:

```
// Exemple de NgModule traditionnel
@NgModule({ declarations: [ ProductListComponent, ProductDetailComponent ],
 imports: [ CommonModule, RouterModule, FormsModule ],
exports: [ ProductListComponent ],
providers: [ ProductService ] })
export class ProductModule { }
```

II. La révolution des composants autonomes (Angular 14+)

Avec l'introduction des composants autonomes (standalone) depuis Angular 14 et leur adoption complète dans Angular 17, le framework a connu une transformation majeure dans sa philosophie d'organisation du code.

Désormais, il est possible de fonctionner avec des composants autonomes qui ne nécessitent plus d'être déclarés dans un NgModule. Cette approche simplifie considérablement le développement en permettant à chaque composant de gérer ses propres dépendances directement.

Pourquoi cette évolution ?

Cette évolution offre plusieurs avantages significatifs :

- **Réduction du boilerplate** : Plus besoin de créer et maintenir des modules pour chaque fonctionnalité.
- **Meilleure encapsulation** : Chaque composant déclare explicitement ce dont il a besoin, rendant les dépendances plus claires et traçables.
- **Facilité de compréhension** : Pour les nouveaux développeurs, le concept de composant autonome est plus intuitif que la hiérarchie de modules.
- **Lazy-loading simplifié** : Le chargement à la demande devient plus naturel, sans configuration complexe de modules.

Le démarrage de l'application s'en trouve également simplifié. Au lieu d'un AppModule, l'application démarre directement avec un composant racine :

```
// main.ts moderne sans AppModule
bootstrapApplication(AppComponent, {
    providers: [
        provideRouter(routes),
        provideHttpClient(),
        provideAnimations()
        ]
});
```

Les services partagés sont désormais fournis au niveau de l'application ou injectés directement là où ils sont nécessaires. Cette approche favorise une architecture plus plate et plus flexible, où les composants peuvent être facilement déplacés, testés et réutilisés sans se soucier de leur appartenance à un module spécifique.

Pour les équipes qui adoptent Angular 17+, cette évolution représente une simplification bienvenue qui permet de se concentrer davantage sur les fonctionnalités métier plutôt que sur la configuration technique du framework.

Les composants

Source: https://angular.dev/essentials/components

Chaque composant comporte plusieurs choses essentielles :

1. Un **@Component** *decorator* qui contient une configuration utilisée par Angular.
2. Un **modèle HTML** qui contrôle ce qui est rendu dans le DOM.
3. Un **sélecteur CSS** (Optionnel) qui définit comment le composant est utilisé en HTML.
4. Une **classe TypeScript** avec des comportements, tels que la gestion des entrées utilisateur ou l'envoi de requêtes à un serveur.

Il est possible de tout avoir sur le même fichier :

- En utilisant les propriété ***'template'***, pour le modèle HTML et ***'style'*** pour le modèle CSS

Il est aussi possible de séparer ces fichiers en ayant un fichier contenant la classe typescript, un fichier contenant le modèle HTML et un fichier contenant le style :

- En utilisant les propriété ***'templateUrl'***, pour le modèle HTML et ***'styleUrl'*** pour le CSS

Il est possible d'importer plusieurs fichier css sur un même composant

- En utilisant la propriété ***'styleUrls'*** (il faut alors utiliser un tableau)

Importer un composant

Pour importer et utiliser un composant dans Angular, nous devons :

1. Dans notre fichier TypeScript du composant, ajoutez une instruction d'importation pour le composant que l'on souhaite utiliser.
   Exemple: import { MonComposant } from './chemin/vers/mon-composant';

2. **Dans notre décorateur @Component**, ajoutez une entrée au tableau **imports** pour le composant que l'on souhaite utiliser.
   Exemple : @Component({
   selector: 'app-parent',
   templateUrl: './parent.component.html',
   styleUrls: ['./parent.component.css'],
   imports: [MonComposant]

})

3. **Dans le template de notre composant**, ajouter un élément qui correspond au sélecteur du composant que l'on souhaite utiliser.

```
<div>
        <!-- Si le sélecteur du composant est 'app-mon-composant' -->
        <app-mon-composant></app-mon-composant>
</div>
```

Cette méthode s'applique aux composants autonomes (standalone components) dans Angular 14+ et est la méthode recommandée pour Angular 17+

Avant angular 14+ il était indispensable d'importer tous les composant dans un *module* angular.

Intégration du Routage dans l'Application

Pour activer le routage, il faut configurer l'application avec les routes définies.

- **provideRouter** est une fonction qui configure le système de routage
- Cette configuration est ensuite utilisée lors du démarrage de l'application
- Dans les versions modernes d'Angular, cette approche remplace l'ancien RouterModule

```
// app.component.ts
import { Component } from '@angular/core';
import { RouterOutlet, RouterLink, RouterLinkActive } from '@angular/router';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet, RouterLink, RouterLinkActive],
  template: `
    <nav>
      <a routerLink="/home" routerLinkActive="active">Accueil</a>
      <a routerLink="/products" routerLinkActive="active">Produits</a>
    </nav>

    <main>
      <router-outlet></router-outlet>
    </main>
  `
})
```

```
export class AppComponent { }
```

- **RouterOutlet** est l'emplacement où les composants routés seront insérés
- **RouterLink** est une directive qui transforme les liens en navigation sans rechargement
- **RouterLinkActive** ajoute une classe (ici "active") aux liens correspondant à la route active

Paramètres d'URL

Les paramètres d'URL permettent de passer des données via l'URL, ce qui est essentiel pour les pages de détail.

```
export const routes: Routes = [
  // ...
  { path: 'products/:id', component: ProductDetailComponent },
  // ...
];
```

Récupération du Paramètre dans le Composant

- Le paramètre est défini dans le chemin avec **:id**
- **ActivatedRoute** permet d'accéder aux informations de la route active
- **paramMap** est un Observable qui émet à chaque changement de paramètre
- Cette approche est particulièrement utile pour les pages de détail ou les vues spécifiques à un élément

```
// product-detail.component.ts
import { Component, OnInit, inject } from '@angular/core';
import { ActivatedRoute } from '@angular/router';

@Component({
  // ...
})
export class ProductDetailComponent implements OnInit {
  private route = inject(ActivatedRoute);
  productId: string | null = null;

  ngOnInit() {
    // Approche avec Observable (réactive aux changements)
    this.route.paramMap.subscribe(params => {
      this.productId = params.get('id');
      // Charger les données du produit avec cet ID
    });
  }
}
```

Query Parameters

Les *query parameters* sont des paramètres optionnels ajoutés à l'URL après un point d'interrogation.

- Les query parameters sont parfaits pour les filtres, la pagination, ou les termes de recherche
- Contrairement aux paramètres d'URL, ils sont optionnels et n'affectent pas la correspondance des routes
- queryParamMap fonctionne de manière similaire à paramMap mais pour les query parameters

- **Navigation avec Query Parameters**

```
// Dans un composant
import { Router } from '@angular/router';

export class SearchComponent {
  private router = inject(Router);

  search(term: string) {
    this.router.navigate(['/results'], {
      queryParams: { q: term, page: 1 }
    });
    // Résulte en /results?q=term&page=1
  }
}
```

- **Récupération des Query Parameters**

```
export class ResultsComponent implements OnInit {
  private route = inject(ActivatedRoute);

  ngOnInit() {
    this.route.queryParamMap.subscribe(params => {
      const searchTerm = params.get('q');
      const page = params.get('page');
      // Utiliser ces valeurs pour filtrer les résultats
    });
  }
}
```

Guards : Protection des Routes

Les guards permettent de contrôler l'accès aux routes en fonction de conditions spécifiques.

- Les guards sont des fonctions qui retournent **true**, **false**, ou un **UrlTree** pour la redirection
- **canActivate** vérifie si un utilisateur peut accéder à une route
- D'autres types de guards existent : **canDesactivate** (empêcher de quitter), **canLoad** (lazy loading), **resolve** (pré-chargement de données)
- Les guards fonctionnels (introduits dans Angular 14+) remplacent les classes guards et sont plus simples à utiliser
- Ils sont parfaits pour protéger des zones administratives ou des fonctionnalités premium

- **Création d'un Guard d'Authentification**

```
// guards/auth.guard.ts
import { inject } from '@angular/core';
import { CanActivateFn, Router } from '@angular/router';
import { AuthService } from '../services/auth.service';

export const authGuard: CanActivateFn = (route, state) => {
  const authService = inject(AuthService);
  const router = inject(Router);

  if (authService.isLoggedIn()) {
    return true; // Autorise l'accès
  }

  // Redirige vers la page de connexion
  return router.createUrlTree(['/login'], {
    queryParams: { returnUrl: state.url }
  });
};
```

- **Application du Guard à une Route**

```
export const routes: Routes = [
  // ...
  {
    path: 'admin',
    component: AdminComponent,
```

```
    canActivate: [authGuard]
  },
  // ...
];
```

Routes Imbriquées

Les routes imbriquées permettent de créer des hiérarchies de navigation, idéales pour les interfaces complexes avec des sous-sections.

- Les routes enfants sont définies dans la propriété **children**
- Le composant parent (**AdminLayoutComponent**) contient un **<router-outlet>** où les composants enfants seront affichés
- Les liens de navigation utilisent des chemins relatifs (**./**) qui sont relatifs à la route parent
- Cette approche permet de créer des layouts complexes avec des zones de navigation persistantes
- Idéal pour les interfaces administratives, les tableaux de bord, ou toute interface avec des sous-sections

```
export const routes: Routes = [
  {
    path: 'admin',
    component: AdminLayoutComponent,
    canActivate: [authGuard],
    children: [
      { path: '', redirectTo: 'dashboard', pathMatch: 'full' },
      { path: 'dashboard', component: DashboardComponent },
      { path: 'users', component: UserListComponent },
      { path: 'users/:id', component: UserDetailComponent }
    ]
  }
];
```

Composant Parent avec Router-Outlet Imbriqué

```
// admin-layout.component.ts
import { Component } from '@angular/core';
import { RouterOutlet, RouterLink } from '@angular/router';

@Component({
  selector: 'app-admin-layout',
```

```
  standalone: true,
  imports: [RouterOutlet, RouterLink],
  template: `
    <div class="admin-container">
      <nav class="sidebar">
        <a routerLink="./dashboard">Tableau de bord</a>
        <a routerLink="./users">Utilisateurs</a>
      </nav>

      <div class="content">
        <!-- Les composants enfants s'afficheront ici -->
        <router-outlet></router-outlet>
      </div>
    </div>
  `
})
export class AdminLayoutComponent { }
```

Lazy Loading

Le chargement paresseux (lazy loading) permet d'améliorer les performances en chargeant les parties de l'application uniquement lorsqu'elles sont nécessaires.

- **loadComponent** charge un composant à la demande
- **loadChildren** charge un groupe entier de routes à la demande
- Le lazy loading réduit la taille du bundle initial, accélérant le premier chargement
- Particulièrement utile pour les grandes applications avec de nombreuses fonctionnalités
- Angular gère automatiquement le préchargement intelligent des modules (configurable)

```
export const routes: Routes = [
  { path: 'home', component: HomeComponent },
  {
    path: 'products',
    loadComponent: () => import('./products/products.component')
      .then(m => m.ProductsComponent)
  },
  {
    path: 'admin',
    loadChildren: () => import('./admin/admin.routes')
      .then(m => m.ADMIN_ROUTES)
  }
];
```

```
// admin/admin.routes.ts
```

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

export const ADMIN_ROUTES: Routes = [
  { path: '', redirectTo: 'dashboard', pathMatch: 'full' },
  {
    path: 'dashboard',
    loadComponent: () => import('./dashboard/dashboard.component')
      .then(m => m.DashboardComponent)
  },
  // Autres routes admin...
];
```

Navigation Programmatique avec ***navigate*** dans Angular

La méthode navigate du service Router d'Angular permet de déclencher des navigations directement depuis votre code TypeScript, offrant un contrôle précis sur les transitions entre les routes de votre application.

```typescript
import { Component, inject } from '@angular/core';
import { Router } from '@angular/router';

@Component({ ...})
export class MyComponent {
  private router = inject(Router);

  goToHomePage() {
    this.router.navigate(['/home']);
  }
}
```

- **Navigation Simple sans Paramètres**
    - **Navigation vers une Route Statique**

```typescript
// Navigation vers une route absolue
this.router.navigate(['/products']);

// Navigation vers une route relative (par rapport à la route actuelle)
this.router.navigate(['../categories'], { relativeTo: this.route });

// Navigation vers une route enfant
this.router.navigate(['details'], { relativeTo: this.route });
```

- **Navigation avec Options Supplémentaires**

```
this.router.navigate(['/dashboard'], {
  // Remplace l'entrée dans l'historique du navigateur au lieu d'en ajouter
une nouvelle
  replaceUrl: true,

  // Restaure la position de défilement précédente ou définit une nouvelle
position
  scrollPositionRestoration: 'enabled',

  // Définit une position de défilement spécifique
  scrollOffset: [0, 0]
});
```

- **Navigation avec Paramètres de Route**

```
// Navigation vers la page de détail du produit avec ID 123
this.router.navigate(['/products', '123']); // Équivalent à: /products/123

// Pour une route comme: /shop/:category/:productId
this.router.navigate(['/shop', 'electronics', 'laptop-x1']); // Équivalent à:
/shop/electronics/laptop-x1
```

- **Navigation avec Query Parameters**

```
this.router.navigate(['/products'], {
  queryParams: { category: 'electronics', sort: 'price' }
});

// Résultat: /products?category=electronics&sort=price
```

- **Query Parameters => gestion des paramètres existants**

```
// Préserver les query params existants
this.router.navigate(['/products'], {
  queryParams: { page: 2 },
  queryParamsHandling: 'preserve'  // Garde tous les params existants
});

// Fusionner avec les query params existants
this.router.navigate(['/products'], {
  queryParams: { filter: 'new' },
```

```
  queryParamsHandling: 'merge'  // Fusionne avec les params existants
});

// Navigation sans query params
this.router.navigate(['/products'], {
  queryParams: {},
  queryParamsHandling: ''  // Ne préserve pas les params existants
});
```

Signals :
https://angular.dev/essentials/signals

https://angular.fr/signals/

Dans les recommandations récentes faites par angular, nous retrouvons l'élément *signals*.

Les Signals représentent un changement fondamental dans la façon dont Angular gère la réactivité. Contrairement aux variables classiques des composants qui sont automatiquement surveillées par le mécanisme de détection de changements basé sur Zone.js, les Signals offrent une approche explicite et plus performante.

Une variable classique dans Angular est surveillée à chaque cycle de détection de changements, ce qui peut entraîner des vérifications inutiles et des problèmes de performance. Lorsque vous modifiez une variable, Angular doit vérifier l'ensemble du composant et ses enfants pour détecter les changements, même si seule une petite partie de l'interface utilisateur dépend de cette variable.

Les Signals, en revanche, créent un graphe de dépendances précis. **Quand un Signal est modifié via .set() ou .update(), seules les parties de l'interface qui dépendent directement de ce Signal sont mises à jour. Cette granularité fine permet d'éviter des cycles de détection superflus et d'améliorer considérablement les performances, particulièrement dans les applications complexes.**

De plus, les Signals rendent le flux de données plus traçable et prédictible. Avec les variables classiques, il peut être difficile de suivre ce qui déclenche un rendu, alors qu'**avec les Signals, ces relations sont explicites**. Cette clarté facilite le débogage et améliore la maintenabilité du code**, tout en préparant Angular pour un avenir sans Zone.js, avec une réactivité plus légère et plus efficace.**

Le routage avec les composants *standalone* (angular 17+)

Le routage est un élément fondamental des applications Angular qui permet la navigation entre différentes vues sans rechargement de la page. Avec l'évolution d'Angular vers une approche "standalone", la configuration du routage a été simplifiée, éliminant le besoin de modules NgModule.

Configuration des Routes

Le routage dans Angular commence par la définition d'une collection de routes qui associent des chemins URL à des composants.

- **Chaque route est un objet avec au minimum un path** (chemin URL) **et un component** (composant à afficher)
- Le chemin ** est un wildcard qui capture toutes les routes non définies

```
// app.routes.ts
import { Routes } from '@angular/router';
import { HomeComponent } from './home/home.component';
import { ProductsComponent } from './products/products.component';
import { NotFoundComponent } from './not-found/not-found.component';

export const routes: Routes = [
  { path: 'home', component: HomeComponent },
  { path: 'products', component: ProductsComponent },
  { path: '**', component: NotFoundComponent }
];
```

Route par Défaut et Redirections

Cette configuration redirige automatiquement l'utilisateur vers la page d'accueil lorsqu'il accède à la racine de l'application

- redirectTo spécifie la destination de la redirection
- pathMatch: 'full' signifie que la redirection s'applique uniquement si le chemin URL complet est vide

```
export const routes: Routes = [
  // Redirection de la route vide vers /home
  { path: '', redirectTo: '/home', pathMatch: 'full' },
  { path: 'home', component: HomeComponent },
];
```

Les pipes

Les pipes dans Angular sont des outils puissants de transformation de données directement dans vos templates. Ils permettent de modifier l'affichage des valeurs sans altérer les données sources.

Exemple:

```
{{ expression | pipeName:param1:param2:... }}
```

Angular 19 continue d'améliorer l'API des pipes avec une meilleure performance et des fonctionnalités étendues.

Pipes Intégrés dans Angular 19

1. **Pipes de Transformation de texte:**

   UpperCasePipe* / *LowerCasePipe* et *TitleCasePipe

```
<p>{{ 'hello world' | uppercase }}</p> <!-- Affiche: HELLO WORLD -->
<p>{{ 'HELLO WORLD' | lowercase }}</p> <!-- Affiche: hello world -->
```

```
<p>{{ 'hello world' | titlecase }}</p> <!-- Affiche: Hello World -->
```

1. **Pipes de Formatage de Nombres**

2. ***DecimalPipe***

```
<!-- Format: {minIntegerDigits}.{minFractionDigits}-{maxFractionDigits} -->
<p>{{ 3.14159 | number:'1.2-4' }}</p> <!-- Affiche: 3.1416 -->
<p>{{ 42 | number:'3.0-0' }}</p> <!-- Affiche: 042 -->
```

3. ***CurrencyPipe***

```
<p>{{ 42.5 | currency }}</p> <!-- Affiche: $42.50 -->
<p>{{ 42.5 | currency:'EUR':'symbol':'1.1-2':'fr' }}</p> <!-- Affiche: 42,5 €
-->
```

4. ***PercentPipe***

```
<p>{{ 0.42 | percent }}</p> <!-- Affiche: 42% -->
<p>{{ 0.42 | percent:'2.2-2' }}</p> <!-- Affiche: 42.00% -->
```