Concepts Fondamentaux d'Angular

Définition d'Angular

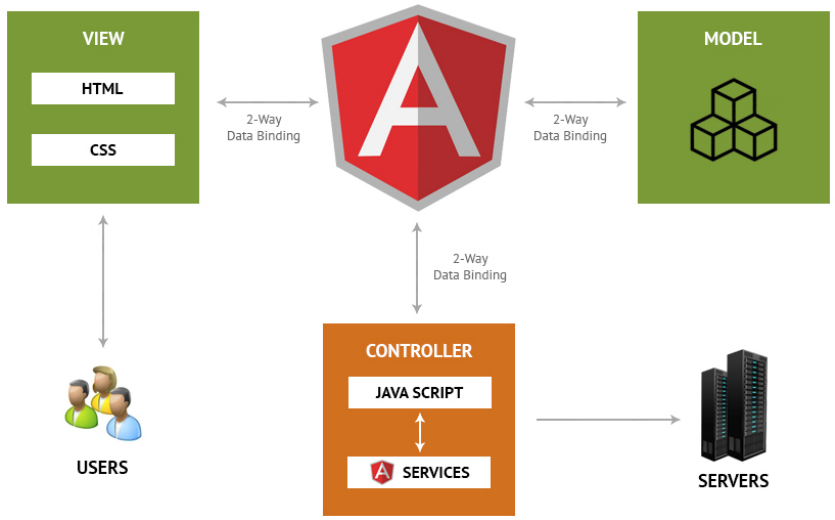
Angular est un framework open-source développé par Google pour la création d'applications web dynamiques. Il repose sur TypeScript et permet de structurer les applications de manière modulaire grâce à une architecture basée sur les composants, les services et les modules. Angular facilite la gestion des données, l'interaction avec des APIs, la gestion des états et la navigation au sein des applications web complexes.

Figure 1: Logo d'Angular

Architecture d'Angular

L'architecture d'Angular repose sur une séparation claire des responsabilités pour garantir la scalabilité et la maintenance des applications. Les principaux blocs de cette architecture sont : les **modules**, les **composants**, les **services**, et d'autres entités comme les **directives** et les **pipes**.

Figure 2: Architecture d'Angular



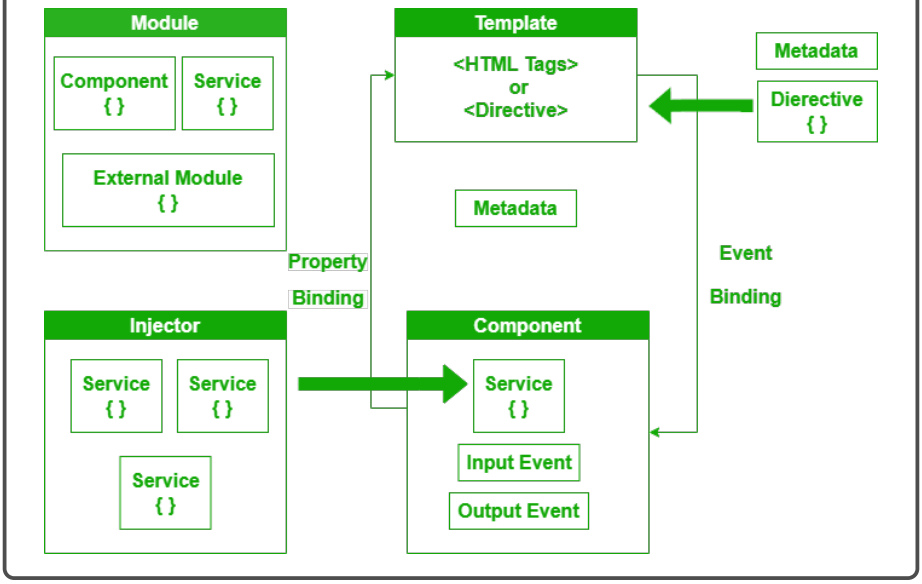
L'objectif de cette section est de comprendre comment ces éléments interagissent entre eux pour former une application Angular cohérente.

1 Vue d'Ensemble de l'Architecture d'Angular

Angular suit le **Modèle Vue Contrôleur (MVC)** amélioré, où :

- Les **composants** gèrent l'affichage et capturent les actions de l'utilisateur.
- Les **services** manipulent les données et la logique métier.
- Les **modules** structurent et organisent les différentes fonctionnalités.

Figure 3: Architecture d'Angular



Ce modèle favorise une séparation des responsabilités pour un développement modulaire.

2 Séparation des Responsabilités et Flux d'Information

2.1 Les Modules : La Structure Globale

- Les modules définissent le squelette de l'application.
- Ils regroupent les composants, services et autres entités.

Lien avec les autres parties : Les modules servent de conteneurs logiques pour regrouper les fonctionnalités connexes. Par exemple, un module dédié à l'authentification contiendra tous les composants et services liés à la gestion des utilisateurs.

Conseil : Commencez toujours par définir les modules avant de créer les composants et services associés.

2.2 Les Composants : L'Interface Utilisateur

- Chaque composant représente une partie de l'interface utilisateur (un bouton, un formulaire, une page).
- Un composant communique avec les services pour obtenir les données à afficher ou à traiter.

Flux : Les données transitent **du service vers le composant**, et les actions utilisateur remontent **du composant vers le service**.

Conseil : Une fois que le module est configuré, commencez à créer les composants pour concevoir l'interface utilisateur.

2.3 Les Services : La Logique Métier et les Données

- Les services sont des classes dédiées à la gestion des données et des processus métiers.
- Ils utilisent souvent le module HTTP pour communiquer avec des API externes.

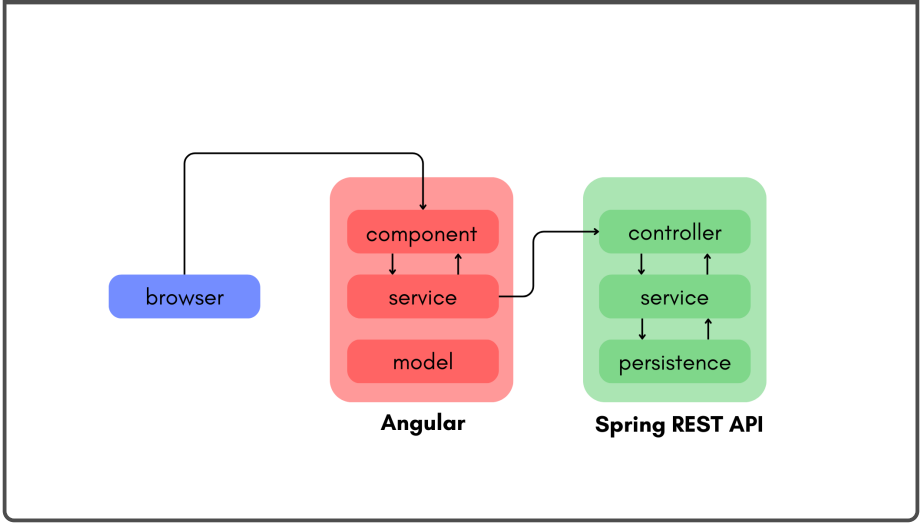
Flux : Les services récupèrent ou manipulent les données, puis les transmettent aux composants.

Conseil : Après avoir défini le modèle de données (interfaces/classes), développez les services pour centraliser la logique métier.

3 Exemple de Flux d'Information en Trois Étapes

1. **Définition des Modules :** Créez un module pour une fonctionnalité spécifique (e.g., gestion des produits).
2. **Création des Services :** Implémentez un service qui récupère les données des produits via une API.
3. **Conception des Composants :** Développez un composant qui utilise le service pour afficher une liste de produits.

Figure 4: Flux d'Information



4 Importance du Flux entre les Parties

L'ordre de développement est essentiel pour éviter des erreurs et des inefficacités :

1. **Module :** Structure la fonctionnalité et facilite son intégration.
2. **Service :** Gère les données et les processus liés à la fonctionnalité.
3. **Composant :** Présente les données à l'utilisateur et capture ses actions.

Ce flux garantit une architecture propre, avec des responsabilités bien définies et un code facile à maintenir.

Modules dans Angular : Structure et Fonctionnement

Dans Angular, les **modules** jouent un rôle clé dans l'organisation de l'application. Ils permettent de regrouper des composants, des services, des directives, des pipes et d'autres modules afin de rendre l'application modulaire, réutilisable et facilement maintenable. Chaque module est associé à un fichier TypeScript qui sert de point d'entrée pour le module, le plus important étant **AppModule**, le module principal.

1. AppModule : Le Point d'Entrée de l'Application

Le **AppModule** est le module de base de toute application Angular. C'est le module qui est chargé lors du démarrage de l'application. Il regroupe tous les autres modules nécessaires au bon fonctionnement de l'application.

Déclaration d'AppModule : AppModule est un module Angular classique qui contient les informations essentielles pour démarrer l'application, comme la déclaration des composants, l'importation des modules nécessaires, les services, et l'injection des dépendances.

Exemple :

```
1
2  import { NgModule } from '@angular/core';
3  import { BrowserModule } from '@angular/platform-browser';
4  import { AppComponent } from './app.component';
5
6  @NgModule({
7    declarations: [  // D clare tous les composants utilis s par ce
         module
8      AppComponent
9    ],
10   imports: [  // Importation d'autres modules n cessaires
11     BrowserModule
12   ],
13   providers: [],  // D claration des services
```

```
14    bootstrap: [AppComponent]  // Composant qui d marre l'
          application
15  })
16  export class AppModule { }
```

Listing 1: Using a service in a component

2. Structure des Modules Angular

Un module Angular est défini à l'aide du décorateur `@NgModule`, qui prend un objet avec plusieurs propriétés clés. Ces propriétés permettent de configurer le module en fonction des besoins spécifiques de l'application.

2.1 Declarations

La propriété `declarations` regroupe tous les **composants**, **directives** et **pipes** utilisés dans le module. Chaque composant, directive ou pipe doit être déclaré dans un module pour qu'il soit reconnu et utilisé.

- **Composants :** Représentent l'interface utilisateur.
- **Directives :** Modifient l'apparence ou le comportement d'un élément du DOM.
- **Pipes :** Permettent de transformer les données affichées à l'utilisateur.

Exemple :

```
@NgModule({
  declarations: [
    AppComponent,
    HeaderComponent,
    FooterComponent
  ]
})
```

2.2 Imports

La propriété `imports` contient une liste de **modules externes** que ce module utilise. Cela permet de regrouper les fonctionnalités d'autres modules, par exemple, le module de routage ou le module de formulaires.

- Par exemple, vous pouvez importer des modules comme `FormsModule`, `HttpClientModule`, `RouterModule`, etc.

Exemple :

```

import { FormsModule } from '@angular/forms';
import { RouterModule } from '@angular/router';

@NgModule({
  imports: [
    BrowserModule,  // Module de base pour le navigateur
    FormsModule,    // Module pour les formulaires
    RouterModule    // Module pour la gestion du routage
  ]
})
```

Listing 2: Using a service in a component

2.3 Providers

La propriété `providers` permet de déclarer les **services** que vous souhaitez injecter dans l'application. Cela permet à Angular d'injecter ces services dans les composants et autres services via l'injection de dépendances.

- Les services peuvent être fournis à l'échelle du module ou de l'application entière en utilisant des `providers` dans `@NgModule`.

Exemple :

```

import { ProductService } from './product.service';

@NgModule({
  providers: [
    ProductService  // D claration du service    utiliser dans l'
        application
  ]
})
```

Listing 3: Using a service in a component

2.4 Bootstrap

La propriété `bootstrap` permet de spécifier le ou les **composants principaux** qui démarrent l'application. C'est ici que vous déclarez le composant racine de votre application, généralement `AppComponent`.

- `AppComponent` est le composant qui sera utilisé pour initialiser l'interface utilisateur de l'application.

Exemple :

```
@NgModule({
  bootstrap: [AppComponent]  // Définition du composant qui démarre l'application
})
```

3. Exemple Complet d'un Module Angular

Voici un exemple d'un module complet avec les différentes parties que nous avons abordées :

```

import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';  // Pour les formulaires
import { RouterModule } from '@angular/router';  // Pour le routage

import { AppComponent } from './app.component';
import { HeaderComponent } from './header/header.component';
import { FooterComponent } from './footer/footer.component';
import { ProductService } from './services/product.service';

@NgModule({
  declarations: [
    AppComponent,  // Composant principal
    HeaderComponent,  // Composant de l'en-t te
    FooterComponent   // Composant du pied de page
  ],
  imports: [
    BrowserModule,  // Module de base pour le navigateur
    FormsModule,    // Module pour les formulaires
    RouterModule    // Module pour la gestion du routage
  ],
  providers: [
    ProductService  // Service pour g rer les produits
  ],
  bootstrap: [AppComponent]  // Composant principal qui d marre l'application
})
export class AppModule { }
```

Listing 4: Using a service in a component

4. Rôle de `AppModule` dans le Flux d'Information

Le **AppModule** sert de point d'entrée et de point de coordination pour toutes les fonctionnalités de l'application.

- Dès que l'application démarre, Angular charge le **AppModule**.
- Il **déclare** les composants, **importe** les autres modules nécessaires (comme ceux pour les formulaires, les services HTTP, etc.), **déclare les services** qu'il fournit à l'application et **initialise** le **composant principal (AppComponent)**.

5. Conclusion

L'architecture d'un module Angular permet de structurer et d'organiser les fonctionnalités de manière efficace et maintenable. La structure d'un module —

Declarations, **Imports**, **Providers**, et **Bootstrap** — est essentielle pour définir comment les différentes parties de l'application interagiront et seront accessibles à travers les autres modules et composants.

Conseil pratique : Commencez par définir vos modules pour organiser l'application en fonctionnalités indépendantes. Ensuite, ajoutez les composants et services nécessaires et assurez-vous qu'ils sont correctement injectés dans les modules.

Dans Angular, les **composants** sont les éléments fondamentaux pour construire l'interface utilisateur (UI). Chaque composant représente une **partie spécifique de l'UI**, comme un bouton, une page, ou un formulaire. Les composants permettent de diviser l'interface utilisateur en éléments modulaires, réutilisables et faciles à maintenir.

Chaque composant dans Angular est associé à un **décorateur Component** qui fournit des métadonnées sur le composant, y compris la structure HTML, le style CSS et le comportement en TypeScript.

1. Le Rôle d'un Composant dans Angular

Un **composant** sert à :

- **Gérer l'interface utilisateur** d'une section de l'application (ex : un formulaire, une barre de navigation).
- **Capturer les interactions** de l'utilisateur (ex : clics, saisie dans des champs de formulaire).
- **Communiquer avec d'autres composants ou services** pour récupérer ou manipuler des données.

En résumé, les composants sont les briques de base de l'application Angular, et leur rôle principal est de lier l'affichage et la logique métier.

2. Utilisation du Décorateur Component

Le décorateur Component est une fonction qui permet de configurer un composant Angular. Il est utilisé pour définir les métadonnées du composant, telles que le modèle HTML, le style CSS, et le comportement TypeScript. Il permet également de connecter un composant à un module spécifique.

Le décorateur Component prend un objet contenant plusieurs propriétés essentielles :

- **selector** : Déclare le nom de l'élément HTML personnalisé qui représente ce composant dans le DOM.
- **templateUrl** ou **template** : Lien vers le fichier HTML ou code HTML inline utilisé pour le modèle du composant.

- **styleUrls** ou **styles** : Lien vers le fichier CSS ou code CSS inline utilisé pour le style du composant.
- **providers** : Déclare les services utilisés dans ce composant.

Exemple de décorateur Component :

```
1
2  import { Component } from '@angular/core';
3
4  @Component({
5    selector: 'app-header',  // D clare un    lment    HTML
         personnalis
6    templateUrl: './header.component.html',  // Lien vers le mod le
         HTML
7    styleUrls: ['./header.component.css']   // Lien vers le fichier
         CSS du composant
8  })
9  export class HeaderComponent {
10   title: string = "Bienvenue sur notre site";
11 }
```

Listing 5: Using a service in a component

3. Structure d'un Composant

Un composant Angular est composé de trois fichiers principaux :

1. **HTML (Template)** : Définit l'interface utilisateur du composant.
2. **CSS (Styles)** : Définit les styles associés à l'interface du composant.
3. **TypeScript (Classe)** : Contient la logique métier du composant, gère les données et les événements utilisateur.

3.1 Le Fichier HTML (Template)

Le fichier HTML définit la structure de l'interface utilisateur. Il peut contenir des balises HTML de base ainsi que des directives Angular pour lier les données et gérer les événements.

Exemple de template HTML :

```
<div class="header">
  <h1>{{ title }}</h1>
  <button (click)="onClick()">Cliquez ici</button>
</div>
```

- `{ title }` : C'est une **liaison de données** (data binding) qui affiche la valeur de la propriété `title` du composant. - `(click)="onClick()"` : Cette syntaxe est un **binding d'événements** qui lie un événement du DOM (comme un clic) à une méthode TypeScript (ici `onClick()`).

3.2 Le Fichier CSS (Styles)

Le fichier CSS contient les styles spécifiques au composant. Angular utilise un système d'encapsulation de styles pour éviter que les styles d'un composant n'affectent d'autres composants.

Exemple de fichier CSS :

```
.header {
  background-color: #4CAF50;
  color: white;
  padding: 10px;
}
```

3.3 Le Fichier TypeScript (Classe)

Le fichier TypeScript contient la **classe du composant** qui gère les propriétés et les méthodes de ce composant. C'est ici que vous définissez la logique métier, l'état interne du composant, et les comportements en réponse aux interactions utilisateur.

Exemple de classe TypeScript :

```
1
2  import { Component } from '@angular/core';
3
4  @Component({
5    selector: 'app-header',
6    templateUrl: './header.component.html',
7    styleUrls: ['./header.component.css']
8  })
9  export class HeaderComponent {
10   title: string = "Bienvenue sur notre site";
11
12   onClick() {
13     console.log("Le bouton a   t   cliqu !");
14   }
15 }
```

Listing 6: Using a service in a component

- La **propriété** `title` contient des données que l'on veut afficher dans le template HTML. - La méthode `onClick()` gère l'événement de clic sur le bouton.

4. Flux de Données et Interactions dans un Composant

Les composants interagissent principalement avec :

- **Les propriétés et méthodes** dans la classe TypeScript.
- **Le template HTML**, via des liaisons de données.
- **Les services** pour manipuler les données ou faire des appels API.

4.1 Liaison de données (Data Binding)

Angular propose plusieurs types de liaison de données :

- **Liaison unidirectionnelle (One-way Binding)** : De la classe vers le template (ou inversement).
  - `{ value }` : Interpolation pour afficher des valeurs dans le template.
  - `[property]="value"` : Liaison des propriétés d'un élément HTML avec la propriété d'un composant.
  - `(event)="handler()"` : Lier un événement DOM à une méthode dans la classe.
- **Liaison bidirectionnelle (Two-way Binding)** : Utilisation de `[(ngModel)]` pour lier les données dans les formulaires.

Exemple de liaison bidirectionnelle :

```
<input [(ngModel)]="username">
<p>Bienvenue, {{ username }}!</p>
```

5. Exemple Complet d'un Composant

Voici un exemple complet d'un composant :

HTML (Template) :

```html
<div class="login-form">
  <h2>{{ title }}</h2>
  <input [(ngModel)]="username" placeholder="Entrez votre nom">
  <button (click)="onSubmit()">Se connecter</button>
</div>
```

Listing 7: HTML example of a login form

CSS (Styles) :

```css
.login-form {
  margin: 20px;
  padding: 15px;
  border: 1px solid #ccc;
}
input {
  padding: 5px;
  margin-bottom: 10px;
}
button {
  background-color: #4CAF50;
  color: white;
```

```
  padding: 10px 20px;
}
```

Listing 8: CSS styling for login form

TypeScript (Classe) :

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

@Component({
  selector: 'app-login',
  templateUrl: './login.component.html',
  styleUrls: ['./login.component.css']
})
export class LoginComponent {
  title: string = 'Connexion';
  username: string = '';

  onSubmit() {
    console.log(`Nom d'utilisateur: ${this.username}`);
  }
}
```

Listing 9: TypeScript class for login functionality

6. Conclusion

Les **composants** sont les éléments clés pour construire l'interface utilisateur dans Angular. Chaque composant est constitué de trois parties essentielles :

- **HTML** pour la structure de l'interface,
- **CSS** pour le style,
- **TypeScript** pour la logique et la gestion des données.

Le décorateur Component permet de lier ces trois parties et de définir comment le composant interagira avec d'autres composants ou services, tout en permettant une gestion efficace des événements et des données.

Conseil pratique : Lors de la création d'un composant, commencez toujours par définir le modèle HTML (structure), puis implémentez la logique nécessaire dans la classe TypeScript, et enfin, ajoutez les styles pour que l'interface soit propre et intuitive.

5 Liaison de Données : Unidirectionnelle et Bidirectionnelle

Dans Angular, le **template** d'un composant est responsable de l'interface utilisateur, tandis que la **liaison de données** permet de relier le modèle de données

(les propriétés et méthodes de la classe du composant) au DOM (le modèle de l'interface utilisateur).

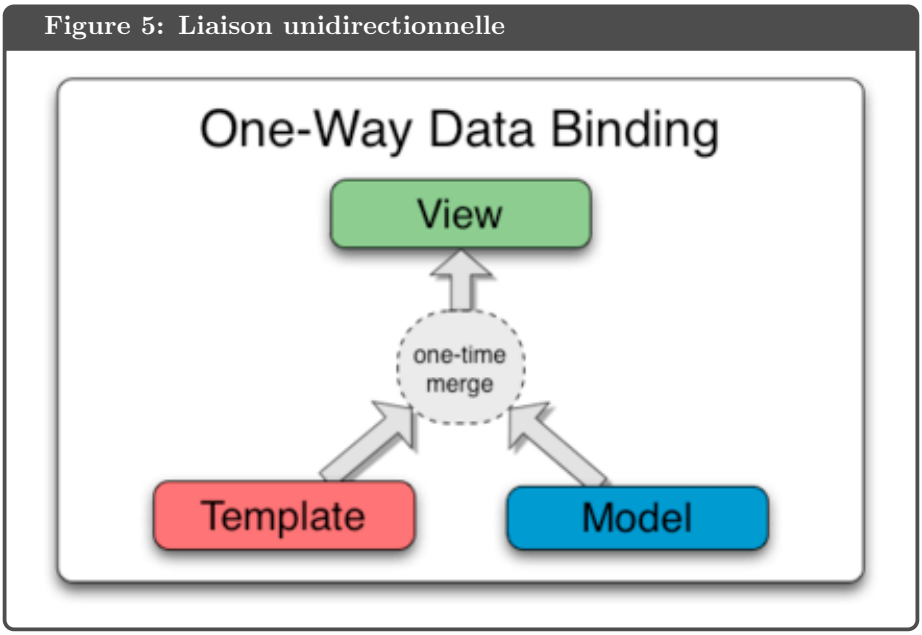
Il existe plusieurs types de liaison de données qui permettent à Angular de synchroniser les données entre le composant et la vue de manière efficace.

6 Liaison de Données : Unidirectionnelle et Bidirectionnelle

6.1 Liaison unidirectionnelle (One-way Binding)

Dans la liaison unidirectionnelle, les données circulent dans une seule direction : soit du composant vers la vue (affichage), soit de la vue vers le composant (interactions utilisateur).

Figure 5: Liaison unidirectionnelle



6.1.1 Interpolation (du composant vers la vue)

Permet de lier une valeur de la classe TypeScript à une section de l'HTML.

```
<p>{{ title }}</p>
```

Dans cet exemple, le contenu de `title` dans la classe du composant sera affiché dans le `<p>`.

6.1.2 Liaison de propriétés (du composant vers la vue)

Permet de lier une propriété d'un élément HTML à une propriété du composant.

```
<img [src]="imageUrl" alt="Image dynamique">
```

Ici, l'attribut `src` de l'image est lié à la propriété `imageUrl` du composant.

6.1.3 Liaison d'événements (de la vue vers le composant)

Permet de lier un événement DOM (comme un clic) à une méthode dans la classe du composant.

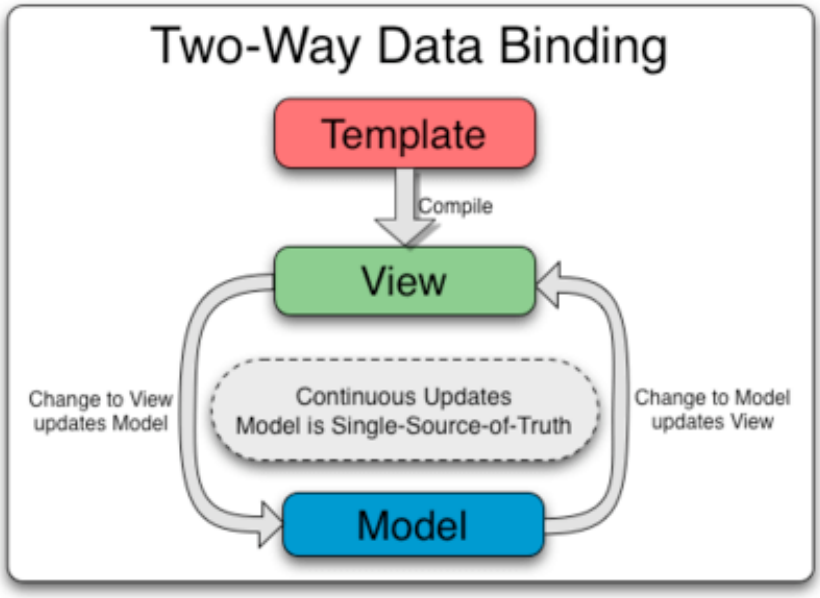
```
<button (click)="onClick()">Cliquez ici</button>
```

L'événement `click` déclenche la méthode `onClick()` du composant lorsqu'on clique sur le bouton.

6.2 Liaison bidirectionnelle (Two-way Binding)

La liaison bidirectionnelle permet de lier une donnée dans le modèle et la vue, de sorte que toute modification dans l'un des deux se reflète instantanément dans l'autre.

Figure 6: Liaison bidirectionnelle



Angular implémente cela principalement avec la directive `ngModel`, qui permet de lier un champ de formulaire avec une propriété de la classe du composant.

```
<input [(ngModel)]="username">
<p>Nom d'utilisateur : {{ username }}</p>
```

Dans cet exemple, la valeur de `username` est liée à l'input. Si l'utilisateur tape quelque chose, `username` est mis à jour automatiquement, et si la propriété `username` change dans le composant, l'input sera également mis à jour.

7 Utilisation de Directives Angular

Les **directives** sont des classes qui modifient le comportement des éléments du DOM. Elles sont utilisées pour créer des comportements réutilisables dans l'interface utilisateur.

7.1 Directives structurelles

Les directives structurelles modifient la structure du DOM en ajoutant ou en supprimant des éléments. Elles sont généralement utilisées pour **contrôler la visibilité** ou la **répétition d'éléments**.

7.1.1 *ngIf

Cette directive permet de conditionner l'affichage d'un élément du DOM en fonction d'une expression. Si l'expression est vraie, l'élément est affiché ; sinon, il est supprimé du DOM.

```
<div *ngIf="isVisible">Ce contenu est visible si isVisible est true.</div>
```

7.1.2 *ngFor

Cette directive permet de répéter un élément pour chaque élément d'une collection.

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

7.2 Directives attributs

Les directives attributs modifient l'apparence ou le comportement d'un élément sans changer sa structure.

7.2.1 ngClass

Cette directive permet d'ajouter ou de retirer des classes CSS d'un élément en fonction des conditions spécifiées dans l'expression.

```
<div [ngClass]="{'highlight': isHighlighted}">Ce texte est mis en surbrillance si isHighligh
```

7.2.2 ngStyle

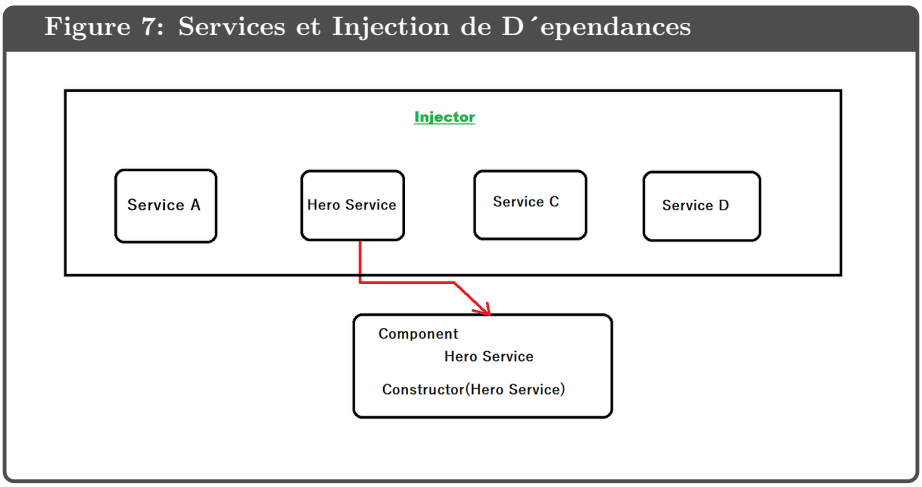
Cette directive permet d'ajouter ou de modifier dynamiquement les styles d'un élément.

```
<div [ngStyle]="{'color': color, 'font-size': fontSize + 'px'}">Ce texte a des styles dynami
```

8 Services et Injection de Dépendances

Les **services** jouent un rôle essentiel dans Angular pour gérer les **logiques métiers**, accéder aux données et effectuer des traitements externes. Ils sont utilisés pour fournir des fonctionnalités réutilisables dans toute l'application. Les services sont généralement utilisés pour communiquer avec des API, manipuler des données ou stocker des informations.

Figure 7: Services et Injection de D´ependances



8.1 Rôle des Services dans Angular

Un service est une **classe** qui encapsule une logique ou un comportement spécifique. Les services sont souvent utilisés pour :

- **Partager des données entre composants** : Un service peut stocker des données partagées et permettre aux composants de les lire ou de les modifier.
- **Communiquer avec des API** : Un service peut être utilisé pour gérer les appels HTTP et récupérer des données depuis un serveur.
- **Gérer l'état de l'application** : Un service peut contenir des informations qui doivent être accessibles à plusieurs composants, comme l'état d'authentification d'un utilisateur ou les paramètres de l'application.

8.2 Injection de Dépendances avec `@Injectable`

L'**injection de dépendances** (DI) est un concept fondamental dans Angular qui permet d'injecter des objets (comme des services) dans des composants ou d'autres services. Cela facilite la gestion des dépendances sans que les classes aient besoin de créer ces objets elles-mêmes.

8.2.1 `@Injectable()`

C'est un décorateur qui marque une classe comme un **service injectable**. Il indique à Angular que cette classe peut être **injectée** dans d'autres composants ou services.

Voici un exemple simple d'un service avec l'injection de dépendances :

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

@Injectable({
  providedIn: 'root' // This indicates that the service is
      available at the application level
})
export class DataService {
  private data: string[] = [];

  constructor() { }

  getData(): string[] {
    return this.data;
  }

  addData(newData: string): void {
    this.data.push(newData);
  }
}
```

Listing 10: TypeScript service example

8.3 Utilisation du service dans un composant

Dans le composant, vous **injectez** le service pour y accéder :

```
import { Component } from '@angular/core';
import { DataService } from './data.service';  // Import the
    service

@Component({
  selector: 'app-data-list',
  templateUrl: './data-list.component.html',
  styleUrls: ['./data-list.component.css']
})
export class DataListComponent {
  data: string[] = [];

  constructor(private dataService: DataService) { } // Inject the
      service
```

```
13
14    ngOnInit(): void {
15      this.data = this.dataService.getData();
16    }
17
18    addData(newData: string): void {
19      this.dataService.addData(newData);
20    }
21  }
```

Listing 11: Using a service in a component

Dans cet exemple, `DataService` est injecté dans `DataListComponent` via le constructeur. Vous pouvez appeler les méthodes du service pour récupérer ou manipuler les données.

9 Conclusion

- **Templates et Liaison de Données** : La liaison de données permet de synchroniser le modèle (données) avec la vue (interface utilisateur). Les liaisons unidirectionnelles et bidirectionnelles facilitent l'interaction entre le modèle et la vue.
- **Directives Angular** : Angular propose des directives puissantes comme `*ngIf`, `*ngFor`, `ngClass` et `ngStyle`, qui permettent de modifier la structure ou les styles des éléments du DOM de manière déclarative.
- **Services et Injection de Dépendances** : Les services encapsulent la logique métier et les données partagées. Grâce à l'injection de dépendances, les services peuvent être facilement injectés dans les composants, permettant de maintenir un code modulaire et réutilisable.

Ces concepts fondamentaux d'Angular sont essentiels pour développer des applications robustes et bien structurées.

Routage dans Angular

Le **routage** permet de naviguer entre différentes vues de l'application. Dans Angular, le routage est configuré en utilisant le module `RouterModule` et permet de lier des composants à des URL spécifiques.

1. Configuration des Routes dans `app-routing.module.ts`

Dans Angular, les routes sont définies dans un fichier de module appelé `app-routing.module.ts`. Ce module importe le `RouterModule` et utilise la méthode `RouterModule.forRoot(routes)` pour configurer les différentes routes.

Voici un exemple de configuration de routes :

```
1
2  import { NgModule } from '@angular/core';
3  import { RouterModule, Routes } from '@angular/router';
4  import { HomeComponent } from './home/home.component';
5  import { AboutComponent } from './about/about.component';
6
7  const routes: Routes = [
8    { path: '', component: HomeComponent },
9    { path: 'about', component: AboutComponent }
10 ];
11
12 @NgModule({
13   imports: [RouterModule.forRoot(routes)],
14   exports: [RouterModule]
15 })
16 export class AppRoutingModule { }
```

Listing 12: Using a service in a component

Dans cet exemple : - La route par défaut (`path: ''`) charge le `HomeComponent`. - La route `/about` charge le `AboutComponent`.

2. Navigation entre Composants et Gestion des Paramètres de Route

Pour naviguer entre les composants, on utilise la directive `routerLink` dans le template, qui permet de lier une URL à un élément.

Exemple :

```
1
2  <a routerLink="/about">   propos</a>
```

Listing 13: Using a service in a component

Gestion des paramètres de route : Parfois, on doit récupérer des paramètres dans l'URL. Cela peut être fait en utilisant `ActivatedRoute` dans le composant cible.

Exemple :

```
1
2  import { ActivatedRoute } from '@angular/router';
3
4  @Component({
5    selector: 'app-about',
6    templateUrl: './about.component.html',
7    styleUrls: ['./about.component.css']
8  })
9  export class AboutComponent implements OnInit {
10
11   constructor(private route: ActivatedRoute) { }
12
13   ngOnInit(): void {
14     this.route.params.subscribe(params => {
15       const id = params['id']; // R cup re un param tre 'id' de
             l'URL
```

```
      console.log(id);
    });
  }
}
```

Listing 14: Using a service in a component

Formulaires dans Angular

Les formulaires permettent de collecter des informations de l'utilisateur. Angular propose deux approches principales pour gérer les formulaires : **les formulaires basés sur des templates** et **les formulaires réactifs**.

1. Formulaires Basés sur des Templates

Dans cette approche, le formulaire et ses validations sont principalement définis dans le template HTML.

Exemple :

```

<form #myForm="ngForm" (ngSubmit)="onSubmit(myForm)">
  <input name="username" ngModel required />
  <button type="submit" [disabled]="!myForm.valid">Submit</button>
</form>
```

Listing 15: Using a service in a component

Dans ce cas, `ngForm` est une directive qui crée un objet de formulaire, et `ngModel` lie les champs à la classe du composant.

2. Formulaires Réactifs

Les formulaires réactifs sont plus puissants et flexibles. Ils sont définis dans le composant et sont utilisés pour créer des formulaires complexes avec des validations avancées.

Exemple :

```

import { Component, OnInit } from '@angular/core';
import { FormBuilder, FormGroup, Validators } from '@angular/forms'
    ;

@Component({
  selector: 'app-login',
  templateUrl: './login.component.html',
  styleUrls: ['./login.component.css']
})
export class LoginComponent implements OnInit {

  loginForm: FormGroup;

  constructor(private fb: FormBuilder) { }
```

```
15
16    ngOnInit(): void {
17      this.loginForm = this.fb.group({
18        username: ['', Validators.required],
19        password: ['', [Validators.required, Validators.minLength(6)
              ]]
20      });
21    }
22
23    onSubmit(): void {
24      if (this.loginForm.valid) {
25        console.log(this.loginForm.value);
26      }
27    }
28  }
```

Listing 16: Using a service in a component

Ici, nous utilisons `FormBuilder` pour créer un formulaire réactif et y appliquer des validations.

Pipes dans Angular

Les **pipes** sont utilisés pour transformer les données affichées dans l'interface utilisateur. Ils sont souvent utilisés pour formater les valeurs, comme les dates ou les devises.

1. Pipes Intégrés

Angular propose plusieurs pipes intégrés pour effectuer des transformations courantes : - `date` : Formate une date.

```
<p>{{ currentDate | date:'shortDate' }}</p>
```

- `currency` : Affiche une valeur sous forme de devise.

```
<p>{{ amount | currency:'EUR' }}</p>
```

2. Création de Pipes Personnalisés

Il est possible de créer des pipes personnalisés pour effectuer des transformations spécifiques aux besoins de l'application.

Exemple :

```
1
2  import { Pipe, PipeTransform } from '@angular/core';
3
4  @Pipe({
5    name: 'reverse'
6  })
```

```
export class ReversePipe implements PipeTransform {
  transform(value: string): string {
    return value.split('').reverse().join('');
  }
}
```

Listing 17: Using a service in a component

Dans le template :

```
<p>{{ 'Hello' | reverse }}</p> <!-- Affiche 'olleH' -->
```

Gestion des Données et HTTP

Angular permet d'effectuer des requêtes HTTP en utilisant le module `HttpClientModule`, ce qui facilite la consommation d'APIs externes.

1. Utilisation de `HttpClientModule`

Pour consommer des API REST, vous devez d'abord importer le module `HttpClientModule` dans le fichier `app.module.ts` :

```

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

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

Listing 18: Using a service in a component

Ensuite, dans votre service ou composant, vous utilisez `HttpClient` pour faire des requêtes HTTP.

Exemple d'appel GET :

```

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

@Component({
  selector: 'app-data',
  templateUrl: './data.component.html',
  styleUrls: ['./data.component.css']
})
export class DataComponent {

  constructor(private http: HttpClient) {}

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

```
  }
}
```

Listing 19: Using a service in a component

2. Consommation d'API REST et Gestion des Erreurs

Lors de la consommation d'APIs, il est important de gérer les erreurs avec `catchError` et de traiter les réponses asynchrones.

Exemple de gestion des erreurs :

```

import { catchError } from 'rxjs/operators';
import { throwError } from 'rxjs';

this.http.get('https://api.example.com/data')
  .pipe(catchError(error => {
    console.error('Error occurred:', error);
    return throwError(error);
  }))
  .subscribe(data => {
    console.log(data);
  });
```

Listing 20: Using a service in a component

RxJS et Observables

Les **Observables** sont une partie centrale d'Angular et sont utilisés pour gérer les flux de données asynchrones, comme les requêtes HTTP, les événements utilisateurs, ou les modifications de données.

1. Introduction à RxJS

RxJS (Reactive Extensions for JavaScript) est une bibliothèque pour composer des programmes asynchrones et basés sur des événements en utilisant des observables.

Un **observable** est un flux de données qui peut être observé. Un **subscriber** peut s'abonner à cet observable pour recevoir les valeurs émises.

Exemple de base avec un observable :

```

import { Observable } from 'rxjs';

const observable = new Observable(subscriber => {
  subscriber.next('Hello');
  subscriber.complete();
});

observable.subscribe(value => console.log(value));  // Affiche '
    Hello'
```

Listing 21: Using a service in a component

2. Utilisation des Observables pour Gérer les Flux de Données

Les observables permettent de gérer les flux de données de manière réactive, par exemple, pour manipuler les données reçues d'une API, les événements utilisateurs, etc.

Exemple avec une requête HTTP :

```

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

Listing 22: Using a service in a component

Lazy Loading et Optimisation

Le **Lazy Loading** permet de charger les modules Angular de manière différée, réduisant ainsi le temps de chargement initial de l'application.

1. Chargement Différé des Modules

La configuration de Lazy Loading est effectuée dans les routes. Au lieu de charger tous les modules dès le début, on charge uniquement ceux qui sont nécessaires pour la vue actuelle.

Exemple :

```

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

Listing 23: Using a service in a component

2. Optimisation des Performances

Il existe plusieurs techniques pour améliorer les performances d'une application Angular, comme : - **Réduction du bundle** en utilisant le Lazy Loading. - **Utilisation de ChangeDetectionStrategy.OnPush** pour réduire le nombre de vérifications de changement. - **Minification et compression des fichiers** lors du déploiement.

Tests et Déploiement

Les tests unitaires permettent de s'assurer que le code fonctionne comme prévu, et le déploiement consiste à mettre l'application sur un serveur ou une plateforme cloud pour qu'elle soit accessible par les utilisateurs.

1. Tests Unitaires avec Jasmine/Karma

Angular utilise **Jasmine** pour les tests unitaires et **Karma** pour l'exécution des tests dans un environnement de navigateur.

Exemple de test unitaire :

```
1
2  import { TestBed } from '@angular/core/testing';
3  import { DataService } from './data.service';
4
5  describe('DataService', () => {
6    let service: DataService;
7
8    beforeEach(() => {
9      TestBed.configureTestingModule({});
10     service = TestBed.inject(DataService);
11   });
12
13   it('should be created', () => {
14     expect(service).toBeTruthy();
15   });
16 });
```

Listing 24: Using a service in a component

2. Déploiement sur un Serveur ou une Plateforme Cloud

Une fois l'application développée et testée, il faut la déployer sur un serveur. Le processus de déploiement consiste généralement à : - Construire l'application en production avec `ng build --prod`. - Déployer les fichiers générés (dans le dossier `dist/`) sur un serveur ou une plateforme cloud comme **Firebase**, **AWS**, ou **Heroku**.

Conclusion

Ce cours couvre les concepts essentiels d'Angular pour la création d'applications robustes et réactives. Du routage à la gestion des données, en passant par les formulaires et l'optimisation des performances, Angular fournit une boîte à outils complète pour le développement d'applications web modernes.