Résumé formation angular

Ce document a pour objectif de résumer les éléments vus pendant la semaine de formation. Il reste un COMPLÉMENT de la documentation officielle, documentation qui évolue au cours du temps.

Table des matières

Présentation d'Angular

Qu'est-ce que Angular ?

Angular est plus qu'un simple framework frontend avec lequel créer des SPA (Single Page Applications). Il s'agit d'une plate-forme de développement à part entière construite en **TypeScript** qui comprend :

- **Un framework** basé sur des composants pour créer des applications Web scalables.
- **Une collection de librairies bien intégrées qui couvrent une grande variété de fonctionnalités, notamment le routage, la gestion des formulaires, la communication client-serveur, les progressive web apps, etc.**
- Une suite d'outils de développement pour vous aider à développer, créer, tester et mettre à jour votre code via le CLI Angular.

Angular est développé et maintenu par Google et sa sortie initiale remonte à septembre 2016. C'est une réécriture complète d'AngularJS et en tant que tel, Angular est un framework à part.

Un peu d'histoire

AngularJS, développé par Google et lancé en 2010, était un framework pionnier pour la création d'applications web dynamiques utilisant HTML et JavaScript. Ses fonctionnalités clés, telles que le two-way data-binding et l'injection de dépendances, ont révolutionné le développement front-end en permettant aux développeurs de construire des applications complexes de manière plus efficace. En 2016, Google a publié Angular 2, une réécriture complète d'AngularJS, introduisant une architecture basée sur les composants et des améliorations significatives de la performance. Cela a marqué la transition vers le framework moderne "Angular" (sans le "JS"). Les versions suivantes, commençant par Angular 4, ont continué à améliorer la performance, à introduire de nouvelles fonctionnalités comme Angular Universal pour le rendu côté serveur, et à améliorer les outils avec Angular CLI.

source: https://worldline.github.io/angular-training/fr/presentation/

Data Binding

Le data binding dans Angular connecte les données de l'application au DOM, gardant les deux synchronisés. Les types incluent l'interpolation {{ data }}, le property binding [property]="data", l'event binding, et le two-way binding.

Event Binding

L'event binding permet de répondre aux événements utilisateur comme les clics ou les frappes au clavier....

Syntaxe : (event)="handler()"

Exemple : <button (click)="onClick()">Cliquez-moi</button>

Les directives

Avantages de la nouvelle syntaxe @

- Meilleure performance par rapport aux directives ng*
- Vérification de type plus stricte
- Syntaxe plus claire et plus lisible
- Support natif pour les blocs @empty et @else
- Fait partie du nouveau moteur de rendu d'Angular (Ivy)

Pour utiliser ces nouvelles directives, assurez-vous d'avoir Angular 17 ou une version plus récente.

@for (Remplace *ngFor)

La directive @for permet d'itérer sur des collections et de générer des éléments DOM pour chaque élément.

```
@for (item of items; track item.id; let i = $index, let isFirst = $first, let isLast = $last) {
<div>{{i}} - {{item.name}}</div>
}
```

```
@empty {
<div>Aucun élément trouvé</div>
}
```

@if (Remplace *ngIf)

La directive @if permet d'afficher conditionnellement des éléments dans le DOM.

```
@if (condition) {
<!-- Contenu affiché si condition est vraie -->
} @else if (autreCondition) {
<!-- Contenu affiché si autreCondition est vraie -->
} @else {
<!-- Contenu affiché si toutes les conditions sont fausses -->
}
```

@switch (Remplace [ngSwitch])

La directive @switch permet d'afficher différents contenus selon la valeur d'une expression.

```
@switch (expression) {
  @case (valeur1) {
  <!--Contenu si expression === valeur1-- >
  }
@case (valeur2) {
  <!--Contenu si expression === valeur2-- >
  }
@default {
  <!--Contenu par défaut-- >
  }
}
```

Passage de données Parent vers Enfant

Les composants parents transmettent des données aux enfants en utilisant le property binding avec le décorateur @Input().

```
// Composant enfant
@Input() data: string;

// Template du parent
<app-child [data]="parentData" > </app-child>
```

Passage de données Enfant vers Parent

Les enfants communiquent avec les parents en utilisant le décorateur @Output() avec EventEmitter.

```
// Composant enfant
@Output() notify = new EventEmitter<string>();
this.notify.emit('Message au parent');

// Template du parent
<app-child (notify)="onNotify($event)" > </app-child>
```

Le Constructor des composants

Le constructeur initialise les membres de la classe et injecte des dépendances via l'injection de dépendances.

```
constructor(
  private service: MonService,
  private http: HttpClient
) { }
```

Les Services

Les services sont des objets singleton qui fournissent des fonctionnalités à travers les composants. **Ils sont idéaux pour le partage de données, la logique métier et les interactions externes**

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

HttpClient et les requêtes HTTP

HttpClient fournit des méthodes pour la communication HTTP (GET, POST, etc.) et retourne des Observables.

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

getData() {
  return this.http.get<Data[]>('api/donnees');
}
```

HttpClient: configurer un intercepteur de requêtes

1.Créer en premier un intercepteur

-> commande avec angular CLI : ng generate interceptor auth

```
// auth.interceptor.ts
import { HttpInterceptorFn } from '@angular/common/http';
```

```
export const authInterceptor: HttpInterceptorFn = (req, next) => {
  // Clone la requête et ajoute un token d'authentification
  const authReq = req.clone({
    headers: req.headers.set('Authorization', `Bearer ${getToken()}`)
  });

  return next(authReq);
};

function getToken(): string {
  // Récupérer votre token depuis le localStorage ou un service
  return localStorage.getItem('token') || '';
}
```

2. Mettre à jour votre fichier "*app.config.ts*"

```
// app.config.ts
import { ApplicationConfig, provideZoneChangeDetection } from '@angular/core';
import { provideRouter } from '@angular/router';
import { provideHttpClient, withInterceptors } from '@angular/common/http';

import { routes } from './app.routes';
import { provideClientHydration, withEventReplay } from '@angular/platform-browser';
import { authInterceptor } from './interceptors/auth.interceptor';

export const appConfig: ApplicationConfig = {
  providers: [
    provideZoneChangeDetection({ eventCoalescing: true }),
    provideRouter(routes),
    provideClientHydration(withEventReplay()),
    provideHttpClient(
      withInterceptors([authInterceptor])
    )
  ]
};
```

3. Points importants

1. Utilisez provideHttpClient() au lieu de l'ancien HttpClientModule
2. Utilisez withInterceptors() pour enregistrer vos intercepteurs
3. Les intercepteurs sont maintenant des fonctions pures (HttpInterceptorFn) et non plus des classes
4. Vous pouvez chaîner plusieurs intercepteurs : withInterceptors([authInterceptor, loggingInterceptor, ...])

Cette nouvelle approche fonctionnelle est plus légère, plus facile à tester et mieux optimisée pour la compilation AOT et le tree-shaking.

Si vous avez besoin de fonctionnalités avancées comme la gestion des erreurs ou des intercepteurs conditionnels, vous pouvez également utiliser withInterceptorsFromDi() pour les cas plus complexes.

Les formulaires avec ngModel

Les formulaires basés sur les templates utilisent ngModel pour le two-way binding dans les formulaires.

```
// Composant
utilisateur = { nom: '' };

// Template
<input [(ngModel)]="utilisateur.nom" name = "nom" >
```

Les formulaires avec 'ReactiveForm'

Les formulaires réactifs offrent une approche dirigée par le modèle avec création explicite de contrôles de formulaire.

1. Imports nécessaires

```
import { ReactiveFormsModule } from '@angular/forms';
import { FormGroup, FormControl, Validators } from '@angular/forms';

@NgModule({
  imports: [
    // ...
    ReactiveFormsModule
  ]
})
```

2. Création du FormGroup et des FormControl

```
// Composant
formulaire = new FormGroup({
  nom: new FormControl('', Validators.required),
  email: new FormControl('', [Validators.required, Validators.email])
});

// Template
<form [formGroup]="formulaire" >
  <input formControlName="nom" >
</form>
```

3. Liaison avec le template

```
<form [formGroup]="formulaire" (ngSubmit)="onSubmit()">
  <input formControlName="nom">
```

```
  <input formControlName="email" type="email">

  <button type="submit" [disabled]="formulaire.invalid"> Envoyer </button>
</form>
```

4. Méthode *submit*

```
onSubmit() {
  if (this.formulaire.valid) {
    console.log(this.formulaire.value);
    // Traitement du formulaire
  }
}
```

5. Accès aux contrôles pour avertir l'utilisateur

-> ajout des messages d'erreurs quand les champs sont remplis

```
@if (formulaire.get('nom')?.invalid && formulaire.get('nom')?.touched) {
<div>Le nom est requis</div>
}
```

Pour l'accès aux contrôles, il est possible de créer des getter dans notre class de notre fichier typescript pour simplifier la lisibilité du code

Le cycle de vie (onInit, onDestroy et onChanges)

OnInit

Utilisation: Initialisation des données, appels API, configuration initiale du composant.

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

@Component({
  selector: 'app-example',
  template: '<div>{{ data }}</div>'
})
export class ExampleComponent implements OnInit {
  data: string;

  ngOnInit(): void {
    // Exécuté une seule fois après l'initialisation des propriétés liées
    // Idéal pour les initialisations, chargement de données, abonnements
    this.data = 'Données initialisées';
  }
}
```

OnChanges

Utilisation: Réagir aux changements des propriétés d'entrée (@Input()), transformer des données en fonction des inputs.

Onchanges ne fonctionne que sur les 'input' des composants (donc uniquement sur les variables '@input'

```
@Component({
  selector: 'app-example',
  template: '<div>{{ processedData }}</div>'
})
export class ExampleComponent implements OnChanges {
  @Input() inputData: string;
  processedData: string;

  ngOnChanges(changes: SimpleChanges): void {
    // Exécuté à chaque fois qu'une propriété @Input change
    if (changes['inputData']) {
      console.log('Ancienne valeur:', changes['inputData'].previousValue);
      console.log('Nouvelle valeur:', changes['inputData'].currentValue);
      this.processedData = this.inputData + ' (modifié)';
    }
  }
}
```

OnDestroy

Utilisation: Nettoyage des ressources, désabonnement, arrêt des timers, libération de mémoire.

```
@Component({
  selector: 'app-example',
  template: '<div>{{ data }}</div>'
})
export class ExampleComponent implements OnDestroy {
  private subscription: Subscription;
  data: string;

  constructor(private dataService: DataService) {
    this.subscription = this.dataService.getData().subscribe(result => {
      this.data = result;
    });
  }

  ngOnDestroy(): void {
    // Exécuté juste avant que le composant soit détruit
    // Idéal pour nettoyer les ressources, désabonnements
    if (this.subscription) {
```

```
    this.subscription.unsubscribe();
  }
}
```

Ces hooks du cycle de vie sont essentiels pour gérer correctement les ressources et le comportement des composants Angular à différentes étapes de leur existence.

Routage dans les applications Angular modernes

Création des routes (Angular 16+)

1. Dans les nouvelles applications Angular standalone, les routes sont définies dans un fichier app.routes.ts :

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

export const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'about', component: AboutComponent },
  {
    path: 'products',
    loadComponent: () => import('./products/products.component').then(m => m.ProductsComponent)
  },
  { path: '**', component: NotFoundComponent }
];
```

2. Puis enregistrées dans la configuration de l'application :

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

export const appConfig: ApplicationConfig = {
  providers: [
    provideRouter(routes)
  ]
};
```

Navigation avec les liens <a>

La navigation déclarative utilise la directive routerLink :

```
<!-- Navigation simple -->

<a routerLink="/about">À propos</a>

<!-- Navigation avec tableau de segments -->
<a [routerLink]="['/products', 'electronics']">Produits électroniques</a>

<!-- Classe active -->
<a routerLink="/about" routerLinkActive="active-link" [routerLinkActiveOptions]="{exact: true}">À
propos</a>
```

Navigation programmatique avec *Router*

```
@Component({ ...})
export class NavComponent {
  constructor(private router: Router) { }

  goToProducts() {
    // Navigation simple
    this.router.navigate(['/products']);
  }

  goToProductDetail(productId: string) {
    // Navigation avec paramètres
    this.router.navigate(['/products', productId]);

    // Alternative avec queryParams
    this.router.navigate(['/products'], {
      queryParams: { id: productId, category: 'electronics' }
    });
  }
}
```

Paramètres dynamiques dans les routes

1. Définition des routes avec paramètres

```
export const routes: Routes = [
  // Paramètre simple
  { path: 'products/:id', component: ProductDetailComponent },

  // Multiples paramètres
  { path: 'category/:categoryId/product/:productId', component: ProductDetailComponent }
];
```

2. Récupération des paramètres

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

@Component({ ...})
export class ProductDetailComponent implements OnInit {
  productId: string;

  constructor(private route: ActivatedRoute, private productService: ProductService) { }

  ngOnInit() {
    // Méthode 1: snapshot (valeur ponctuelle)
    this.productId = this.route.snapshot.paramMap.get('id');

    // Méthode 2: observable (réactif aux changements)
    this.route.paramMap.subscribe(params => {
      this.productId = params.get('id');
      // Charger les données du produit
    });

    // Méthode 3: observable avec switchMap (recommandée)
    this.route.paramMap.pipe(
      switchMap(params => {
        const id = params.get('id');
        return this.productService.getProduct(id);
      })
    ).subscribe(product => {
      // Utiliser le produit
    });
  }
}
```

3. Redirection

```
export const routes: Routes = [
  // Redirection simple
  { path: '', redirectTo: '/home', pathMatch: 'full' },

  // Redirection conditionnelle (via Guard)
  {
    path: 'admin',
    component: AdminComponent,
    canActivate: [adminGuard]
  }
];
```

4. Routes enfants

```
export const routes: Routes = [
  {
    path: 'dashboard',
    component: DashboardComponent,
    children: [
      { path: '', component: DashboardOverviewComponent },
      { path: 'stats', component: StatsComponent },
      { path: 'settings', component: SettingsComponent }
    ]
  }
];
```

Composants Standalone et Lazy Loading dans Angular moderne

Composants Standalone

Les composants standalone, introduits dans Angular 14 et améliorés dans les versions ultérieures, permettent de créer des composants sans dépendre d'un NgModule.

Avantages des composants standalone

1. Réduction de la complexité (pas besoin de NgModule)
2. Meilleure encapsulation des dépendances
3. Facilité de réutilisation
4. Meilleure compatibilité avec les outils modernes
5. Simplification du **lazy loading**

Le concept de Lazy Loading de composants dans Angular

Le lazy loading (chargement paresseux) de composants est une technique d'optimisation qui consiste à ne charger certaines parties de votre application que lorsqu'elles sont réellement nécessaires, plutôt que de les inclure dans le bundle initial.

Principe fondamental

Définition simple : Charger du code uniquement quand on en a besoin, pas avant.
Avantages

1. **Temps de chargement initial réduit** : L'application démarre plus rapidement car seuls les composants essentiels sont chargés
2. **Meilleure performance** : Réduction de la taille du bundle principal (main.js)
3. **Utilisation optimisée des ressources** : Économie de bande passante et de mémoire
4. **Expérience utilisateur améliorée** : L'application est utilisable plus rapidement

Explication du Server-Side Rendering (SSR) dans un projet Angular

Analyse des fichiers de config et de leurs composants clés :

1. Configuration serveur -> app.config.server.ts

```
import { mergeApplicationConfig, ApplicationConfig } from '@angular/core';
import { provideServerRendering } from '@angular/platform-server';
import { provideServerRouting } from '@angular/ssr';
import { appConfig } from './app.config';
import { serverRoutes } from './app.routes.server';
```

```
const serverConfig: ApplicationConfig = {
  providers: [
    provideServerRendering(),
    provideServerRouting(serverRoutes)
  ]
};

export const config = mergeApplicationConfig(appConfig, serverConfig);
```

Rôle : Ce fichier configure l'application pour qu'elle s'exécute sur le serveur.

- provideServerRendering(): Active le rendu côté serveur
- provideServerRouting(serverRoutes): Configure le routage spécifique au serveur
- mergeApplicationConfig: Fusionne la configuration client avec la configuration serveur

2. Configuration client

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

export const appConfig: ApplicationConfig = {
  providers: [
    provideZoneChangeDetection({ eventCoalescing: true }),
    provideRouter(routes),
    provideClientHydration(withEventReplay())
  ]
};
```

Rôle : Ce fichier configure l'application côté client.

- provideZoneChangeDetection: Configure la détection de changements avec optimisation via eventCoalescing
- provideRouter(routes): Configure le routage côté client
- provideClientHydration(withEventReplay()): Active l'hydratation, permettant au client de "reprendre" l'application rendue par le serveur sans recréer le DOM

3. Routes serveur

```
import { RenderMode, ServerRoute } from '@angular/ssr';

export const serverRoutes: ServerRoute[] = [
  {
    path: '**',
    renderMode: RenderMode.Prerender
  }
];
```

Rôle : Ce fichier définit comment les routes doivent être rendues côté serveur.

- path: '**': S'applique à toutes les routes
- renderMode: RenderMode.Prerender: Indique que toutes les routes doivent être pré-rendues

Explication des concepts clés

Server-Side Rendering (SSR)
Le SSR permet de générer le HTML côté serveur avant de l'envoyer au client. Cela améliore :

- Le SEO (les moteurs de recherche voient le contenu complet)
- Les performances perçues (contenu visible plus rapidement)
- L'expérience utilisateur sur des connexions lentes

Hydratation

Après que le serveur envoie le HTML rendu, le client "hydrate" ce HTML en attachant les gestionnaires d'événements et en reprenant le contrôle de l'application. withEventReplay() permet de capturer les événements utilisateur survenus pendant l'hydratation.

Modes de rendu serveur

- **Prerender**: Génère le HTML à l'avance (build-time)
- **Dynamic**: Génère le HTML à chaque requête (runtime)

Zone.js et détection de changements

provideZoneChangeDetection configure comment Angular détecte les changements d'état. L'option eventCoalescing optimise les performances en regroupant plusieurs mises à jour en une seule.

En résumé

Cette configuration met en place une application Angular moderne utilisant le Server-Side Rendering avec hydratation, ce qui offre les avantages suivants :

1. **Meilleur SEO**
2. Temps de chargement initial plus rapide
3. Meilleure expérience utilisateur
4. Pré-rendu de toutes les pages
5. Transition fluide entre le contenu rendu par le serveur et l'application interactive

C'est une approche moderne qui combine le meilleur des deux mondes : la vitesse et le SEO du rendu côté serveur avec l'interactivité des applications SPA.