

Formation Angular

Ihab ABADI / UTOPIOS

Programme Partie 1

- **Rappel Es6+ Javascript**
- **TypeScript et typage en Javascript**
- **Philosophie du Framework Angular**
- **Notion de templating**
- **Angular CLI et environnement de dev**
- **Structure d'une application Angular**
- **Notion de composant**
- **Passer des données à un composant**
- **Utilisation des événements dans un composant**
- **TwoWay Binding**
- **Utilisation des projections**
- **Notion de directives**
- **Directive d'attribut**
- **Directive de structure**

Rappel ES6 Javascript

- **Introduction des classes :**
 - ES6 a introduit la possibilité d'instanciation d'objet à partir d'une classe
 - La possibilité d'utilisation d'héritage comme tout autre langage de POO
- **Arrow Functions:**
 - La possibilité d'écrire des fonctions sans état avec une écriture fléchée
- **Template Strings**
 - Mécanisme d'interpolation des chaînes de caractères
- **Variable muable et mutable et block scoped**
- **Opérateur ...spread et ...rest**
 - La possibilité de récupérer les éléments d'un tableau, objet ou passer un nombre illimité de paramètres
- **Déstructuration des objets**
 - La possibilité d'extraire des variables à partir d'un objet ou tableau plus rapidement
- **Utilisation des modules**
 - Utilisation des modules donne la possibilité d'exporter d'importer des fonctionnalités, objet, variable

TypeScript

- **TypeScript est un transpiler de Javascript**
- **TypeScript offre la possibilité d'utiliser un langage fortement typé**
- **TypeScript peut utiliser plusieurs type (boolean, number, string, [], {}, undefined, enum, any...)**
- **TypeScript donne la possibilité d'utiliser des classes**
- **TypeScript donne la possibilité d'utiliser des interfaces**
- **TypeScript donne la possibilité d'utiliser des Decorators (Annotation)**
- **Démo**

Philosophie Angular

- **Angular est un (Vrai) Framework développé par Google**
- **Pourquoi utiliser Angular ?**
- **TypeScript**
- **Facilité d'apprentissage**
- **Une multitude de fonctionnalité disponible directement dans le framework**
 - **Routage**
 - **Formulaire**
 - **Observable**
 -

Angular Templating

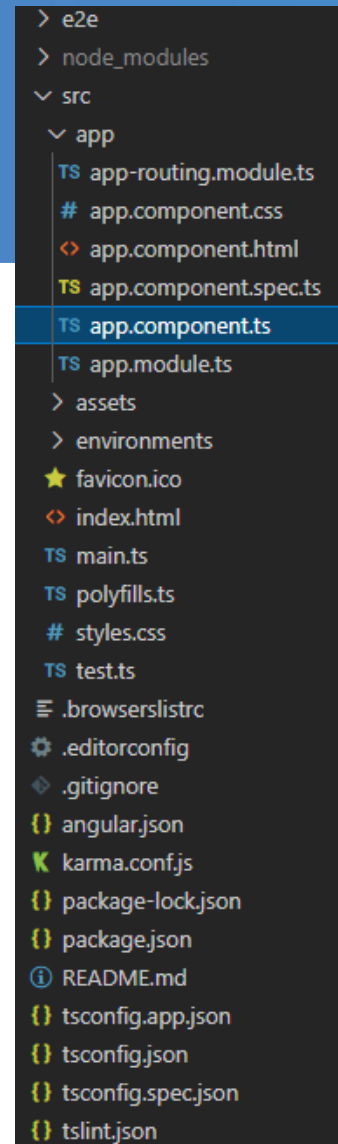
- **La gestion du rendu dans angular se fait directement dans le DOM réel**
- **Angular utilise le moteur rendu Ivy**
- **La compilation et la génération du code en AOT**
- **L'amélioration des debug**
- **Diminution du poids des Js**

Environnement de dev et Angular CLI

- **Pour le développement en Angular, on utilisera :**
 - **Un ide comme visual studio**
 - **NodeJS**
- **Angular fournit une commande Line interface pour la simplification de développement**
- **Installation du CLI se fait à l'aide d'un package npm**
- **Quelque exemple de commande**

Structure d'un projet Angular

- Dossier node_module
- Fichier de configuration (node, typescript)
- Fichier de configuration de test
- Fichier Modules
- Fichier composants



```
> e2e
> node_modules
v src
  v app
    TS app-routing.module.ts
    # app.component.css
    <> app.component.html
    TS app.component.spec.ts
    TS app.component.ts
    TS app.module.ts
  > assets
  > environments
  ★ favicon.ico
  <> index.html
  TS main.ts
  TS polyfills.ts
  # styles.css
  TS test.ts
  ≡ .browserslistrc
  ⚙ .editorconfig
  ⚡ .gitignore
  {} angular.json
  📄 karma.conf.js
  {} package-lock.json
  {} package.json
  ⓘ README.md
  {} tsconfig.app.json
  {} tsconfig.json
  {} tsconfig.spec.json
  {} tslint.json
```


Notion de composants

- Le concept de base de toute application angular est le composant.
- En effet, toute l'application peut être modélisée sous la forme d'un arbre de ces composants.
- Fondamentalement, un composant est tout ce qui est visible par l'utilisateur final et qui peut être réutilisé plusieurs fois dans une application.
- Création d'un composant en Angular se fait à l'aide de l'annotation `@Component`
- Un composant est défini par plusieurs méta-data (selector, template,...)
- Un Composant peut partagé ses propriétés avec le template

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

@Component({
  selector: "rio-hello",
  template: "<p>Hello, {{name}}!</p>",
})
export class HelloComponent {
  name: string;

  constructor() {
    this.name = "World";
  }
}
```

Notion de composants

Un moyen utile de conceptualiser la conception d'applications angulaires consiste à la considérer comme une arborescence de composants imbriqués, chacun ayant une portée isolée

```
<rio-todo-app>
  <rio-todo-list>
    <rio-todo-item></rio-todo-item>
    <rio-todo-item></rio-todo-item>
    <rio-todo-item></rio-todo-item>
  </rio-todo-list>
  <rio-todo-form></rio-todo-form>
</rio-todo-app>
```

Utilisation des événements dans un composant

- Un gestionnaire d'événements est spécifié à l'intérieur du modèle à l'aide de crochets pour désigner la liaison d'événements.
- Ce gestionnaire d'événements est ensuite codé dans la classe pour traiter l'événement

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

@Component({
  selector: 'rio-counter',
  template: `
    <div>
      <p>Count: {{num}}</p>
      <button (click)="increment()">Increment</button>
    </div>
  `
})
export class CounterComponent {
  num = 0;

  increment() {
    this.num++;
  }
}
```

Utilisation des évènements dans un composant

- Dans Angular, on peut créer nos propres évènements pour remonter des données à un composant parent.
- La mise en place d'un évènement se fait par une propriété bindable et annoter avec **@Output** de type **EventEmitter**

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

@Component({
  selector: 'rio-app',
  templateUrl: 'app/app.component.html'
})
export class AppComponent implements OnChange {
  num = 0;
  parentCount = 0;

  ngOnChange(val: number) {
    this.parentCount = val;
  }
}
```

```
<div>
  Parent Num: {{ num }}<br>
  Parent Count: {{ parentCount }}
  <rio-counter [count]="num" (result)="ngOnChange($event)">
  </rio-counter>
</div>
```

```
import { Component, EventEmitter, Input, Output } from '@angular/core';

@Component({
  selector: 'rio-counter',
  templateUrl: 'app/counter.component.html'
})
export class CounterComponent {
  @Input() count = 0;
  @Output() result = new EventEmitter<number>();

  increment() {
    this.count++;
    this.result.emit(this.count);
  }
}
```

```
<div>
  <p>Count: {{ count }}</p>
  <button (click)="increment()">Increment</button>
</div>
```

Two way binding

- Two way binding combine à la fois le Input et OutPut avec la directive ngModel

```
<input [(ngModel)]="name" >
```

Utilisation des projections

- La projection est un concept très important dans Angular.
- Il permet aux développeurs de créer des composants réutilisables et de rendre les applications plus évolutives et flexibles.
- Pour illustrer cela, supposons que nous ayons un ChildComponent comme

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

@Component({
  selector: 'rio-child',
  template: `
    <div style="border: 1px solid blue; padding: 1rem;">
      <h4>Child Component</h4>
      <ng-content></ng-content>
    </div>
  `
})
export class ChildComponent {
}
```

```
...
<rio-child>
  <p>My <i>projected</i> content.</p>
</rio-child>
...
```

Directives

- Une directive modifie le DOM pour changer l'apparence, le comportement ou la disposition des éléments DOM.
- Les directives sont l'un des principaux blocs de construction utilisés par Angular pour créer des applications. En fait, les composants angular sont en grande partie des directives avec des modèles.
- Il existe trois principaux types de directives dans Angular:
- Composant - directive avec un modèle.
- Directives d'attribut - directives qui modifient le comportement d'un composant ou d'un élément mais n'affectent pas le modèle
- Directives structurelles - directives qui modifient le comportement d'un composant ou d'un élément en affectant la façon dont le modèle est rendu

Directives d'attribut

- **Les directives d'attribut sont un moyen de modifier l'apparence ou le comportement d'un composant ou d'un élément DOM natif. Idéalement, une directive devrait fonctionner d'une manière indépendante des composants et non liée aux détails de mise en œuvre.**
- **Par exemple, Angular a des directives d'attribut intégrées telles que `ngClass` et `ngStyle` qui fonctionnent sur n'importe quel composant ou élément**
- **demo**

Directives de structure

- Les directives structurelles sont un moyen de gérer le rendu d'un composant ou d'un élément via l'utilisation de la balise de template.
- Cela nous permet d'exécuter du code qui décide de la sortie finale rendue. Angular a quelques directives structurelles intégrées telles que `ngIf`, `ngFor` et `ngSwitch`
- demo

Programme Partie 2

- **Injection de dépendance en Angular, Injection et utilisation de services**
- **Utilisation des observables**
- **Utilisation de module en Angular**
- **Module de routing**
- **Module ReactiveForms**
- **Module HttpClient**

Injection de dépendance en Angular

- L'injection de dépendances (DI) était une fonctionnalité de base dans Angular.
- DI est un concept de programmation antérieur à Angular.
- Le but de DI est de simplifier la gestion des dépendances dans les composants logiciels. En réduisant la quantité d'informations dont un composant a besoin pour connaître ses dépendances, les tests unitaires peuvent être simplifiés et le code a plus de chances d'être flexible.
- Le système DI d'Angular est (principalement) contrôlé via `@NgModule`. Plus précisément, via un tableau de providers

```
@NgModule({  
  declarations[ ChatWidget, AuthWidget ]  
  providers: [ AuthService, ChatSocket ],  
})
```

Injection de dépendance en Angular

- L'interaction avec DI Angular se fait à l'aide d'annotation `@Inject` et `@Injectable`
- `@Inject ()` est un mécanisme manuel pour faire savoir à Angular qu'un paramètre doit être injecté.
- `@Injectable ()` fait savoir à Angular qu'une classe peut être utilisée avec l'injecteur de dépendances.
- `@Injectable ()` n'est pas strictement requis si la classe a d'autres décorateurs angulaires dessus ou n'a pas de dépendances. Ce qui est important, c'est que toute classe qui va être injectée avec Angular soit décorée.
- La meilleure pratique consiste à décorer les injectables avec `@Injectable ()`, car cela a plus de sens pour le lecteur

```
constructor(@Inject(ChatWidget) private chatWidget) { }
```

```
@Injectable()  
export class ChatWidget {  
  constructor(  
    public authService: AuthService,  
    public authWidget: AuthWidget,  
    public chatSocket: ChatSocket) { }  
}
```

Injection de dépendance en Angular

- **DI Angular renvoie un singleton si l'injection se fait au niveau du module principale**
- **Angular offre la possibilité de définir le scope d'injection au niveau d'un composant, un sous module, ..**

```
@Component({  
  selector: "child-own-injector",  
  template: `<span>{{ value }}</span>`,  
  providers: [Unique],  
})
```

Utilisation des observables

- Une nouvelle fonctionnalité intéressante utilisée avec Angular est l'Observable.
- Il ne s'agit pas d'une fonctionnalité spécifique à Angular, mais plutôt d'une norme proposée pour la gestion des données asynchrones.
- Les observables ouvrent un canal de communication continu dans lequel plusieurs valeurs de données peuvent être émises au fil du temps.
- À partir de là, nous obtenons un modèle de traitement des données en utilisant des opérations de type tableau pour analyser, modifier et maintenir les données.
- Angular utilise largement les observables

Utilisation des observables

- Création d'un observable à l'aide d'une librairie RxJS

```
this.data = new Observable(observer => {  
  setTimeout(() => {  
    observer.next(42);  
  }, 1000);  
  
  setTimeout(() => {  
    observer.next(43);  
  }, 2000);  
  
  setTimeout(() => {  
    observer.complete();  
  }, 3000);  
});  
  
let subscription = this.data.subscribe(  
  value => this.values.push(value),  
  error => this.anyErrors = true,  
  () => this.finished = true  
);
```

Utilisation des observables VS Promise

- Les promesses et les observables nous fournissent des abstractions qui nous aident à gérer la partie asynchrone de nos applications.
- Cependant, il existe des différences importantes entre les deux:
- Les observables peuvent définir à la fois les aspects de configuration et de démontage du comportement asynchrone.
- Les observables sont annulables.
- De plus, Observables peut être retenté à l'aide de l'un des opérateurs de relance fournis par l'API, tels que `retry` et `retryWhen`.
- les promesses exigent que l'appelant ait accès à la fonction d'origine qui a renvoyé la promesse afin d'avoir une capacité de nouvelle tentative