Architectures et technologies WEB

Introduction aux technologies WEB

V01.00 17/08/2016 - MAURICE

Déroulement du cours

- *Chapitre 0 : Préambule*
- *Chapitre 1 : Les technologies WEB*
- **Chapitre 2 : La Programmation cliente**
- Chapitre 3 : Le modèle MVC / Les templates
- Chapitre 4 : La sécurité
- Chapitre 5 : Les bases de PHP
- Chapitre 6 : PHP 2ème partie / ORM
- Chapitre 7 : SOA / ROA
- Chapitre 8 : Le référencement

Plan de la séance

- Chapitre 2 : La programmation cliente
  - Angular

Angular

- 2010, 1ère version d'AngularJS lancée.
- Permet de créer plus facilement des Single Page Applications, des applications web qui imitent les applications natives
- Cette version souffre d'une syntaxe plutôt complexe ainsi que des limitations du JavaScript.
- Google choisit de complètement réécrire le framework pour sa version 2.
- Aujourd'hui, nous en sommes à Angular 6.x (maintenant appelé simplement "Angular") ;

Angular

- Angular est performant : rapide (5 fois plus rapide que Angular JS)
- Modulaire, l’application est consituée de composants et de serivces
- Améliore la productivité : en affichant une syntaxe expressive basée sur la syntaxe de ES2015/TypeScript
- S ’adpate aux mobiles

Les alternatives à Angular

- Il existe différents frameworks JavaScript très populaires aujourd'hui : React, Ember, Vue…
- Angular présente un niveau de difficulté légèrement supérieur
- Il utilise le TypeScript plutôt que JavaScript pur
- Angular est géré par Google, un gage de périnnité ?
- Le TypeScript — ce langage permet un développement beaucoup plus stable, rapide et facile.
- Le framework Ionic basé sur Angular permet le développement d'applications mobiles multi-plateformes à partir d'une seule base de code.

ES6

- **ES5:** publié en 2009
  ES6: publié en 2015, sous le nom ES2015
  ES7: publié en 2016, sous le nom ES2016
  ES8: publié en 2017, sous le nom ES2017
  ES9: Il est dejà en cours…

ES6

- ES6 apporte :
- Les Classes et l’héritage
- Les fonctions fléchées

```
const add = (a,b) => a+b;
```

- Le templating des chaines

Les applications SPA

- Une application SPA (Single Page Application) est une application constituée d'une seule page.
- Elle permet d'optimiser les temps de réponse en évitant les délais de rechargement.
- L'IHM est considérée comme un client riche ou la logique applicative s'éxécute coté client.

Installation

- Le CLI, ou “Command Line Interface” permet d'exécuter des commandes en mode console,
- C’est l’outil qui vous permet d’exécuter des scripts pour la création, la structuration et la production d’une application Angular.
- Nécéssite d’installer préalablement nodeJS et NPM
- NPM est un package manager qui permet l’installation d’outils et de libraries.
- Pour le développement, plusieurs IDE intégre bien la Syntaxe Angular : Atom, Sublime Text, Visual Studio Code ou WebStorm.
- Augury : Plugin Chrome spécifique pour le développement d’applications Angular

```
• nvm install 10
• npm install -g @angular/cli
```

Les commandes CLI

- Nouvelle application : `ng new <app-name>`
- Web server : `ng serve`
- Production Build : `ng build prod`
- Unit test : `ng test`

- Création d'une classe : `ng g class new-class`
- Création d'un component : `ng g component new-component`
- Création d'une directive : `ng g directive new-directive`
- Création d'une enum : `ng g enum new-enum`
- Création d'une Interface : `ng g interface new-interface`
- Création d'un Module : `ng g module new-module`
- Création d'une Pipe : `ng g Pipe new-pipe`
- Création d'une Service : `ng g Service new-Service`

Première application

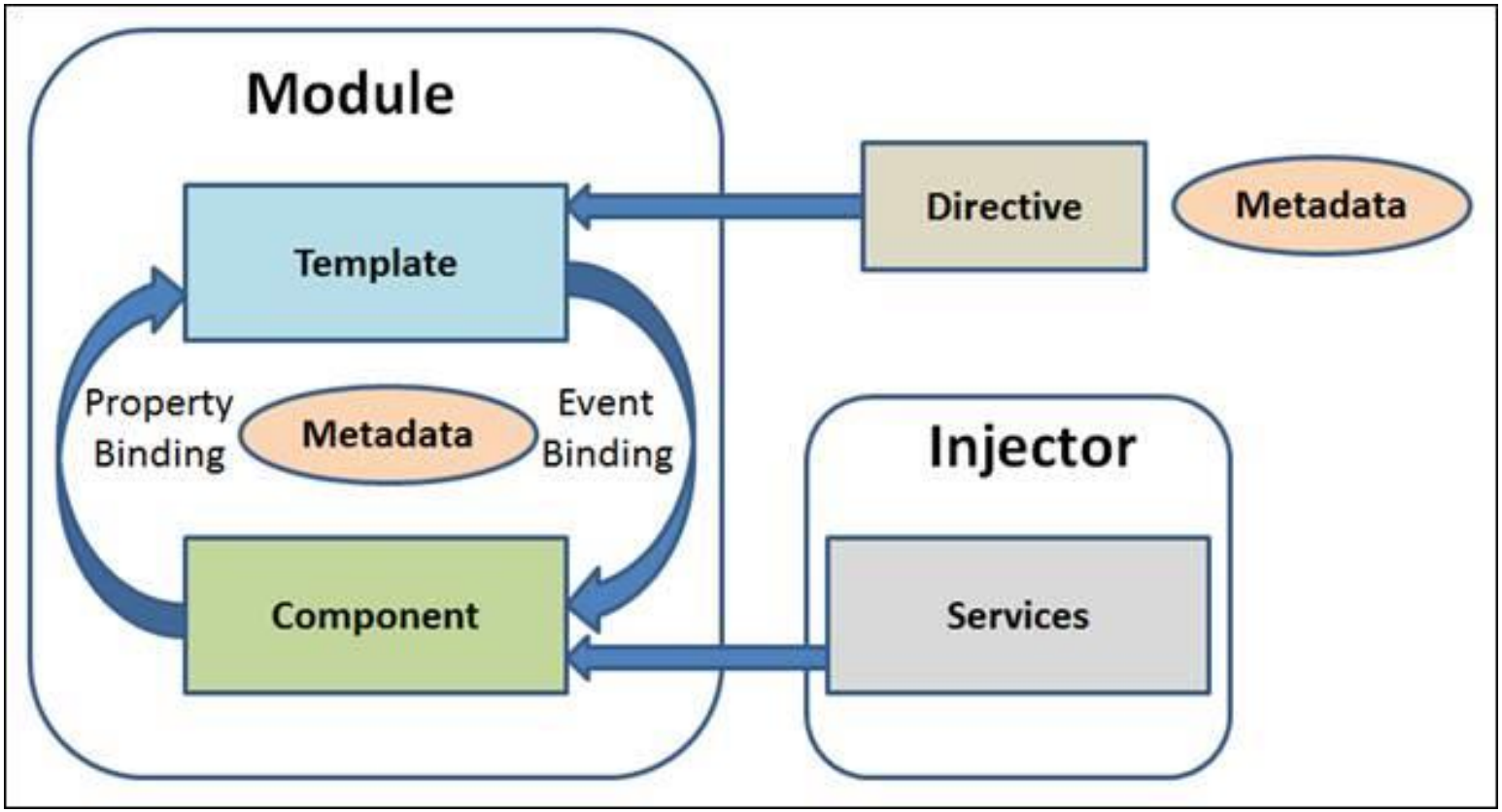
- Créer un nouveau projet Angular à l'aide de commandes CLI

```
• ng new mon-appli
• cd mon-appli
• ng serve --open
```

- Une structuration SCSS

```
• ng new mon-projet-angular --
  style=scss
```

L’architecture



Les Composants

- Angular propose une organisation de l'application en composants.
- La page `index.html` générée contient la structure de l'application
- La balide `<app-root> </app-root>` indique l'emplacement ou sera injecté le composant racine.

La page Index.hml

```
<!doctype html>
<html lang="en">
<head>
        <meta charset="utf-8">
        <title>Test</title>
        <base href="/">
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
        <app-root></app-root>
</body>
</html>
```

Les Composants

- Le composant principal de l'application générée se nomme : `app-component`
- Un composant est organisé en 3 parties :
  - le template en HTML,
  - la feuille de styles en SCSS,
  - le fichier TypeScript,
- Le fichier `app.module.ts` référence les différents composants connus par l'application.

Les Composants

- Le fichier ts contient le décorateur @Component() associé à un objet qui défini les éléments suivants :
  - selector : nom utiliser comme balise HTML pour insérer le component,
  - templateUrl : le chemin vers le code HTML à injecter ;
  - styleUrls : un array contenant les feuilles de styles de ce component ;
- Une classe contenant la logique applicative du composant.

Les Composants

- La génération d'un nouveau composant s'effectue en ligne de commande :

```
ng generate component mon-composant
```

Les composants

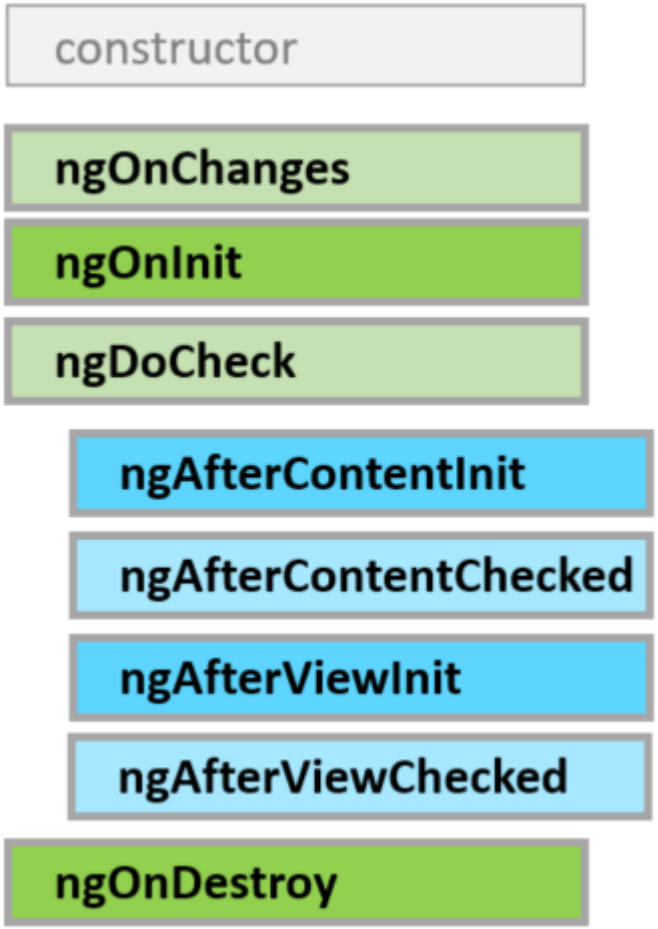
```
import { Component } from '@angular/core';


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


export class AppComponent {  title = 'test';}
```

Cycle de vie

- **Les Hooks**
- Après avoir créé un composant ou une directive en appelant son constructeur, Angular appelle les méthodes du cycle de vie dans la séquence suivante à des moments spécifiques :

ngOnChanges()	Appelé après un changement détecté dans un Input
ngOnInit()	Appelé après le binding
ngDoCheck()	Appelé pour personnaliser un changement suite un appel à ngOnChanges ou ngOnInit
ngAfterContentInit()	Appelé une fois après le 1er ngDoCheck
ngAfterContentChecked()	Appélé après ngAfterContentInit et après chaque ngDoCheck
ngAfterViewInit()	Appelé après l'initalisation de la vue une fois après ngAfterContentChecked
ngAfterViewChecked()	Appelé après la verification de la vue après ngAfterViewInit et chaque ngAfterContentChecked()
ngOnDestroy()	Appelé juste avant la destruction d'un composant.

Hooks


constructor
ngOnChanges
ngOnInit
ngDoCheck
ngAfterContentInit
ngAfterContentChecked
ngAfterViewInit
ngAfterViewChecked
ngOnDestroy

Le Databinding

- Le databinding permet la communication entre le code TypeScript et le template HTML.
- Cette communication peut s’effectuer dans les deux directions :
  - les informations venant du code seront affichées dans le navigateur:
    - Les deux principales méthodes sont le "string interpolation" et le "property binding" ;
  - les informations venant du template seront gérées par le code : l'utilisateur a rempli un formulaire ou clique sur un bouton : "event binding"

Le Databinding

- **String interpolation**
  - L'interpolation est la manière la plus simple de mettre à jour des données issues du code TypeScript et affichées dans la page HTML.

```
<div>{{nom}}</div>
```

- Ou nom est un propriété définie dans la classe du composant

Le Databinding

- **Property Binding**
- La liaison "property binding" permet de modifier dynamiquement les propriétés d'un élément du DOM en fonction du code TypeScript.
- **Pour lier une propriété du code TypeScript à un attribut HTML, il faut le mettre ce dernier entre crochets [ ]**

```
<button class="btn-submit"
[disabled]="!valid">Envoyer</button>
```

Le Databinding

- **Event Binding**
- Permet d'associer un attribut à un code TypeScript (une méthode de la classe ts).
- **Pour lier un attribut HTML à une méthode défini dans la classe ts, il faut mettre cet attribut entre ( )**

```
<button
(click)="onSubmit()">Valider</button>
```

Le Databinding

- **Two-way binding**
- Permet d'associer un attribut HTML à un code propriété de la classe ts et réciproquement
- **Pour lier un attribut HTML à une méthode défini dans la classe ts, il faut mettre cet attribut entre [( )]**
- NgModel est une directive qui permet à un formulaire de mettre à jour le model et le formulaire

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

Le Databinding

- **Communication entre composants**
- Il est possible de créer des propriétés personnalisées dans un component afin de pouvoir lui transmettre des données depuis le composant parent.
- il faut utiliser le décorateur @Input()

```
<app-composant [p1]="'valeurP1'"</ app-composant>
```

```
@Input() p1: string;
```

Les directives

- Les directives sont des instructions intégrées permettant de générer et manipuler des éléments dans le DOM.
- Angular propose 2 types de Directives :
  - les directives structurelles
  - les directives par attribut.
- Vous pouvez créer vos propres directives

Les directives

- **Les directives structurelles**

*ngIf : permet d'envoyer un élément en fonction d'une condition

```
  <div *ngIf="afficher == 'ok'">Bonjour</div>
```

*ngFor : permet d'itérer sur élément

```
<app-monComposant  *ngFor="let c of listC"
                    [nomC]="c.nom"
</app-monComposant>
```

Les directives

- **Les directives attributs**
- les directives par attribut permettent de modifier le comportement d'un objet déjà existant.
- **ngStyle**
- Cette directive permet d'appliquer des styles à un objet du DOM de manière dynamique

```
<h4 [ngStyle]="{color:
getColor()}">Bonjour {{ Nom }} </h4>
```

Les directives

- **ngClass**
- Cette directive permet d'appliquer une classe à un objet du DOM de manière dynamique

```
<div
[ngClass]="{'classe1': true,
            'classe2': valide === 'ok'}">
Exemple </div>
```

Les Pipes

- Les pipes transforment les données en entrée. Les pipes peuvent être sérialisés
- Plusieurs pipes sont proposés par Angular.
- Il est également possible de créer ses propres pipes

```
<p>Mis à jour : {{ lastUpdate | date: 'short' }}</p>
```

Les Pipes

- Les pipes transforment les données en entrée. Les pipes peuvent être sérialisés
- Plusieurs pipes sont proposés par Angular.
- Il est également possible de créer ses propres pipes

```
<p>Mis à jour : {{ lastUpdate | date: 'short' }}</p>

<p>Mis à jour : {{ lastUpdate | date: 'yMMMMEEEEd' }}</p>
<p>Mis à jour : {{ lastUpdate | date: 'yMMMMEEEEd' | uppercase }}</p>
```

Les Pipes

- Le pipe `async` permet de gérer des données asynchrones, lorsque des données doivent être récupérer sur un serveur.

```
<p>Mis à jour : {{ lastUpdate | async | date: 'yMMMMEEEEd' | uppercase }}</p>
```

Les services

- un service permet de centraliser des méthodes utilisées par plusieurs composants de l’application ou de manière globale par l'application entière.
- Un service permet d’organiser l’application fonctionnellement.

```
ng generate service monservice
```

L’injection

- Pour être utilisé un service doit être injecté.Il y a trois niveaux possibles :
  - dans AppModule : la même instance du service sera utilisée par tous les components de l'application et par les autres services ;
  - dans AppComponent : tous les components auront accès à la même instance du service mais pas les autres services ;
  - dans un autre component : le component lui-même et tous ses enfants auront accès à la même instance du service, mais le reste de l'application n'y aura pas accès.

L’injection

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

@Injectable(
        {
                providedIn: 'root',
        }
)

export class MonService {
        constructor() { }
}
```

```
constructor(private monService: MonService) { }
```

Les Observables

- Pour interagir avec des composants asynchrones il existe plusieurs techniques :
- Les callback
- Les promises (obsolètes)
- Les observables (RxJS)

A un objet Observable, un Observer est associé qui sera exécuté à chaque fois que l'Observable émet une information.

Les observables sont notamment utilisés dans les échanges HTTP.

Les Observables

- L'Observable émet trois types d'information : des données, une erreur, un message complète

```
{
 const counter = Observable.interval(1000);
 counter.subscribe(
          (value) => { this.secondes = value; },
          (error) => { console.log(error); },
          () => { console.log('Terminé!'); }
     );
}
```

HTTP

- Angular propose un service HttpClient qui dédié aux requêtes HTTP

```
private clients = [
  {id: 1, name: 'Toto'},
  {id: 2, name: 'Titi'},
  {id: 3, name: 'Tutu'}
];
```

HTTP

```
// Injection
constructor(private httpClient: HttpClient) { }


save (clients) {
this.httpClient.post('http://',clients).subscribe
(
     () => { console.log('terminé !'); },
     (error) => {console.log(error); }
     );
}
```

Le Routing

- Il s’agit de définir les routes qui permettent de définir la cinématique de l’application.
- Il s'agit des instructions d'affichage à suivre pour chaque URL, autrement dit, les component(s) à afficher pour une URL donnée.
- Le routing d'une application est déclaré dans le fichier `app.module.ts` ou dans un fichier spécifique généré:

```
ng generate module app-routing --flat --module=app
```

- Pour importer RouterModule depuis @angular/router , il doit être ajouté à l'array imports de AppModule , tout en l’associant à la méthode forRoot() et en lui passant l'array des routes crées

Le Routing

```
const appRoutes: Routes = [
  { path: 'client', component: clientComponent },
];
```

```
imports: [
    BrowserModule,
    FormsModule,
    RouterModule.forRoot(appRoutes)
],
```

Le Routing

- Angular affichera les components dans le template lorsque l'utilisateur naviguera vers la route déterminée à place de la balise <router-outlet>

```
<div class="container">
  <div class="row">
    <div class="col-xs-12">
      <router-outlet></router-outlet>
    </div>
  </div>
</div>
```

Le Routing

- Pour naviguer à l'intérieur de l'application, il est nécessaire de créer des liens référençant les routes définies dans le fichier `app.module.ts`
- L'attribut `routerLink` permet de définir ces liens.

```
<a
[routerLink]="[routerLinkVariable,dynamicPara
meter]"></a>
```

Le Routing

- Pour associer un parametre à un lien, il suffit de le préfixé de ‘:’ dans la table de routage du fichier `app.module.ts`

```
{ path: 'detail/:id', component: DetailCryptoComponent },
```

```
<a [routerLink] = "['/detail',coin.id]" > {{coin.id}} </a>
```

- Et de récupérer la valeur du paramètre à l’initialisation du composant après avoir injecté l’objet route dans le constructeur

```
this.route.snapshot.params.id;
```

Le Routing

- **Redirection**
- Dans certains cas, il peut être nécessaire de rediriger l'utilisateur sur un 404 par exemple.
- Créer un composant spécifique et ajouter les routes ad'hoc.

```
{ path: 'not-found', component: 404Component },
{ path: '**', redirectTo: 'not-found' }
```

Le Routing

- **Guards**
- Pour éxécuter une code avant l'exécution d'une route, il est possible d'utiliser le guard `canActivate`
- Un guard est un service exécuté au moment de la navigation.
- La méthode admet 2 paramètres : `ActivatedRouteSnapshot` et `RouterStateSnapshot`
- Elle retroune une valeur booléenne.
- Permet de conditionner l'éxécution d'un route (habilitation…)

```
@Injectable()
export class AuthGuard implements CanActivate {

  constructor(private authService: AuthService) { }

  canActivate(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean {

  }
}
```

Reactive Forms

- Angular propose deux technologies de création de formes:
  - les formes réactives
  - les formes gabarit
- Les deux technologies appartiennent à la bibliothèque @angular/forms et partagent un ensemble commun de classes de contrôle de formulaire.
- ils divergent le style de programmation et la technique.

Reactive Forms

- La programmation réactive facilite la gestion des formulaires et des contrôles de surface.
- Les objets de contrôle de formulaire sont directement accessible depuis la classe du composant.
- Le composant peut observer les changements d’état des contrôles du formulaire et réagir à ces changements.
- Les mises à jour de valeur et de validité sont toujours synchrones.

Reactive Forms

- La programmation réactive facilite la gestion des formulaires et des contrôles de surface.
- Les objets de contrôle de formulaire sont directement accessible depuis la classe du composant.
- Le composant peut observer les changements d’état des contrôles du formulaire et réagir à ces changements.
- Les mises à jour de valeur et de validité sont toujours synchrones.

Références

- node.js : https://nodejs.org/en/docs/
- npm : https://docs.npmjs.com/
- ng : https://cli.angular.io/
- Bulma : https://bulma.io/
- ES6 : http://es6-features.org/