A la découverte d'Angular (JS / 2)

Hello!

Nicolas Payot
Front-End dev @Zenika Lyon

>> nicolaspayot.github.io <<

Sommaire

- Rappels
- AngularJS (1.5)
- TypeScript
- Angular 2

1

Rappels

EcmaScript 2015

- **JavaScript** : développé en 10 jours par **Brendan Eich** (Netscape, Mozilla, Brave) en avril 1995
- **EcmaScript** : spécification (1ère édition en juin 1997)
- **EcmaScript 2015** : 6ème édition (juin 2015)
- Supporté à +90% par les navigateurs les plus récents (voir https://kangax.github.io/compat-table/es6/)

EcmaScript 2015

➔ let et const

```
let var1 = 'Hello, World';
const var2 = ['Hello', 'World'];
```

- var1 et var2 ont un scope de type *block*
- var2 ne peut pas être réassignée

EcmaScript 2015

➔ Arrow functions

```
const helloFn1 = () => 'Hello, World!';

const helloFn2 = (name) => {
  console.log(name);
  return name.toUpperCase();
}
```

- Attention, la valeur du **this** est déterminée par le **contexte de définition** de la fonction, et non de son appel

EcmaScript 2015

➔ Default parameter

```
const multiply = (a, b = 2) => a * b;
multiply(5); // 10
```

- La valeur par défaut est utilisée si l'argument n'est pas passé ou **undefined**

EcmaScript 2015

➔ Rest parameter

```
const f = (a, b, ...others) => {
  others.forEach(n => console.log(n));
};
f('Hello', 'World', 1, 2, 3, 4); // 1, 2, 3, 4
```

- Si le dernier paramètre d'une fonction est préfixé par **...**, il devient un tableau qui contient la liste des arguments qui lui sont passés

EcmaScript 2015

➔ Spread operator

```
const a = [1, 2, 3];
const b = ['Hello', 'World', ...a];
// b = ['Hello', 'World', 1, 2, 3]
```

- L'élément préfixé par **...** doit être un tableau
- Dans un tableau, les **...** permettent d'effectuer une concaténation

EcmaScript 2015

➔ String interpolation

```
const name = 'John Doe';
const str = `Hello, ${name}!`; // Hello, John Doe!
```

- Interprétation d'une expression dans une chaîne de caractères
- La chaîne entière doit être entre des **back-ticks : `**
- Possibilité d'interpréter une chaîne multi-lignes

EcmaScript 2015

➔ Property shorthand

```
const a = 1, b = 2;
const obj = { a, b }; // obj = { a: a, b: b }

console.log(obj.a); // 1
console.log(obj.b); // 2
```

- Les clés de l'objet doivent avoir le même nom que les variables associées

EcmaScript 2015

➔ Destructuring assignment

```
const numbers = [1, 2, 3, 4];
const [a, b] = numbers; // a = 1, b = 2
const [x, , , y] = numbers; // a = 1, y = 4
```

- Déstructuration d'un tableau par des variables individuelles pendant l'assignation

EcmaScript 2015

➔ Destructuring assignment

```
const obj = { a: 1, b: 2 };
const { a, b } = obj; // a = 1, b = 2
```

- Fonctionne aussi avec des objets
- Les variables doivent avoir le même nom que les clés

EcmaScript 2015

➔ Classes

```
class Hello {
  constructor(name) { this.name = name; }
  sayHi() { console.log(`Hello, ${this.name}!`); }
}
const hello = new Hello('John Doe');
hello.sayHi(); // Hello, John Doe!
```

- *Sucre syntaxique* sur les fonctions et leurs prototypes
- Offrent une façon plus simple de créer des objets et de reproduire le principe d'héritage de la POO

EcmaScript 2015

➔ Modules

```
// file hello.js
export class Hello { ... }
export const sayHi = () => { ... };
export const ONE = 1;

// file index.js
import { Hello, sayHi, ONE } from './path/to/hello';
// ou
import * as hello from './path/to/hello';
```

- Attention, cette fonctionnalité n'est pas implémentée dans les navigateurs (utiliser un système de chargement de modules, type **webpack**)

npm

- Gestionnaire de paquets Node (npmjs.org)
- Existe pratiquement depuis la création de Node.js
- Outil en ligne de commande
- Commandes les plus courantes :
  - `npm install`
  - `npm install -g`
  - `npm remove / uninstall`
  - `npm update`

npm

➔ package.json

- npm se base sur un fichier descripteur du projet :
  - `name`
  - `version (node-semver)`
  - `description`
  - `dependencies`
  - `scripts (start, test, …)`

npm

➔ package.json : dépendances

- `dependencies`
  nécessaires à l'exécution
- `devDependencies`
  nécessaires au développement
- `peerDependencies`
  nécessaires au bon fonctionnement
- `optionalDependencies` (rare)
  pas indispensables
- `bundledDependencies` (rare)
  publiées, livrées avec le module

npm

➔ package.json : versions

- Structure : `MAJOR.MINOR.PATCH`
- `MAJOR` : changement d'API incompatible
- `MINOR` : ajout de fonctionnalités rétro-compatibles
- `PATCH` : correction de bugs

- Pour spécifier une version :
  - `version` exacte
  - `~` ou `^` (approximativement, compatible)

webpack

- Gestionnaire de modules
- Supporte les différents systèmes de modules
  - **ES2015, CommonJS, AMD**, …
- Disponible sur npm : `npm install -g webpack`
- Génère un **bundle** à chaque modification des sources
  - Fichier JavaScript contenant tous les modules importés de l'application)
- Serveur web de développement disponible

Questions ?

2

AngularJS (1.5)

Présentation

- Développé par **Google (2009, Miško Hevery)**
- Version actuelle : **1.5.8** (support IE9)
- Site : https://angularjs.org
- Code : https://github.com/angular/angular.js
- Documentation :
  - API : https://docs.angularjs.org/api
  - Blog : http://blog.angularjs.org

Présentation

- Framework full-stack
  - Routage par URL
  - Composants (JS + template HTML)
  - Directives (composants personnalisés)
  - Gestion des ressources REST
  - Injection de dépendances
  - Tests unitaires, end 2 end
  - ...

➔ Points forts

- JavaScript pur (**ES2015**)
  - Adhérence technique limitée
  - Réactivité de l'interface
- **Two-way** data binding
- Testabilité
- Écosystème très développé
  - Librairies / Modules *3rd party*
  - Documentation
  - Communauté

➔ Points faibles

- Commence à dater par rapport aux nouveaux frameworks
- Router natif incomplet
- i18n géré partiellement
- Courbe d'apprentissage difficile
  - Directives
  - Beaucoup de façons de faire une même chose
- Performances (si page lourde, non optimisée)

➔ Concepts

- `component`
  - Définit par une balise (ou un attribut) HTML
  - Association d'un contrôleur (objet JavaScript) + un template HTML
- `controller`
  - Données et logique d'affichage
- `directive`
  - Plus personnalisable qu'un composant
  - Permet de manipuler le DOM (**seul endroit sur !**)

➔ Concepts

- `service`
  - Singleton, injecté dans les contrôleurs (DI)
  - Fonctionnalités métiers
- `filter`
  - Transformation de la vue d'une donnée
- `module`
  - Agrégation logique (par fonctionnalité) de services, composants, contrôleurs, directives et filtres

Premiers pas

- Utiliser un générateur
  - **FountainJS,** générateur **Yeoman** d'applications web modernes (http://fountainjs.io/)

```
$ npm install -g yo generator-fountain-webapp
```

```
$ yo fountain-webapp
```

- Répondre aux questions
  - **Angular 1, Webpack, ES2015, Just a Hello World**

Premiers pas

- Module principal : `/src/index.js`
- Fichiers sources : `src/app/*.(html|js)`
- Bootstrap de l'application : `/src/index.html`
- Lancement de l'application

```
$ npm run serve
```

  - Server de développement (`localhost:3000`)
  - Live reload (rechargement automatique de la page après modification des sources)

Premiers pas

➔ Template et bindings

- Template
  - Fichier ou chaîne de caractères HTML
  - Représente la vue d'un composant (component)
- Binding
  - Représentation dynamique d'une donnée au sein d'un template. Interprétation de l'expression entre les **{{ }}**, préfixée par **`$ctrl`** : controller du composant (**seulement à partir d'Angular 1.5**)

```
<span>Hello, {{$ctrl.name}}!</span>
```

Premiers pas

➔ Directives utiles

- `ng-model`
  - Two-way data binding (vue `<->` model)
  - Utilisable sur les balises `input, select textarea`

```
<span>Hello, {{$ctrl.name}}!</span>
<input type="text" ng-model="name">
```

- `ng-repeat`
  - Affichage de listes

```
<!-- names = ['John', 'Bob', 'Mike'] -->
<ul><li ng-repeat="name in names">{{$ctrl.name}}</li></ul>
```

Premiers pas

➔ Directives utiles

- ng-if
  - Ajout ou suppression d'une partie du DOM

```
<div ng-if="$ctrl.isVisible()">Hello, World!</div>
```

- ng-class
  - Ajout ou suppression de classes CSS

```
<div ng-class="{ c1: $ctrl.addClass1, c2: $ctrl.addClass2 }"></div>
<!-- Avec addClass1 = true et addClass2 = false
le html généré sera : <div class="c1"></div> -->
```

Modules

➔ Déclaration

- Un module permet d'encapsuler un ensemble de fonctionnalités (**attention, il ne s'agit pas de module ES2015**)
- Une application AngularJS est composé d'un module principal et (bien souvent) de sous modules
- Pas obligatoire, mais bonne pratique de déclarer et configurer ses modules dans des fichiers index.js

```
// fichier src/app/feature1/index.js (sous module)
angular.module('znkApp.feature1', []);

// fichier src/index.js (module principal)
angular.module('znkApp', ['znkApp.feature1']);
```

Modules

➔ Configuration

- La directive **ng-app** permet de spécifier le module principal

```
<html ng-app="znkApp">
```

- Un module instancie et expose différents types d'objets
  - `.service('NameService', configFn)`
  - `.component('name', configObj)`
  - `.controller('NameController', configFn)`
  - `.directive('name', configFn)`
  - `.filter('name', configFn)`

Components

➔ Déclaration

```
// fichier znk-hello.component.js
class ZnkHelloController { ... }

export const znkHello = {
  template: `<h1>Hello, World!</h1>`,
  controller: ZnkHelloController
}

// fichier index.js
import { znkHello } from './znk-hello.component';

angular
  .module('znkApp', [])
  .component('znkHello', znkHello);
```

Components

➔ Déclaration

```
// fichier znk-hello.component.js
export const znkHello = {
  templateUrl: 'src/app/znk-hello.html',
  controller: 'ZnkHelloController'
}

// fichier index.js
import { znkHello } from './znk-hello.component';
import { ZnkHelloController } from './znk-hello.controller';

angular
  .module('znkApp', [])
  .component('znkHello', znkHello)
  .controller('ZnkHelloController', ZnkHelloController);
```

Components

➔ Utilisation

```
<!-- fichier index.html -->
<html ng-app="znkApp">
  <head></head>
  <body>
    <!-- Attention à la conversion camelCase / spinal-case -->
    <znk-hello></znk-hello>
  </body>
</html>
```

Components

➔ Bindings

```
// fichier znk-hello.component.js
export const znkHello = {
  template: `<h1>Hello, {{$ctrl.znkName}}!</h1>`,
  controller: 'ZnkHelloController',
  bindings: { znkName: '<' }
}

<!-- fichier index.html -->
<znk-hello znk-name="'John Doe'"></znk-hello>
<!-- Attention la aussi à la conversion camelCase / spinal-case -->
```

- La propriété **znkName** sera accessible dans la classe **ZnkHelloController**, bindée sur le **this**

Controllers

➔ Déclaration

```
// fichier znk-hello.controller.js
export class ZnkHelloController {
  constructor() { this.greetings = 'Hello, World!'; }

  // Exécutée lorsque les données bindées sont initialisées
  $onInit() { ... }
}

// fichier index.js
import { ZnkHelloController } from './znk-hello.controller';

angular
  .module('znkApp', [])
  .controller('ZnkController', ZnkController);
```

Services et DI

➔ Déclaration

```
// fichier znk.service.js
export class ZnkService {
  constructor() { ... }

  fetchUsers() { return fetch('/api/users'); }
}

// fichier index.js
angular
  .module('znkApp', [])
  .service('ZnkService', ZnkService);
```

Services et DI

➔ Injection de dépendances

- Permet d'implémenter le principe de l'IoC (Inversion de Contrôle)
- IoC : design pattern, permet de déléguer la mise en relation entre chaque objets à une couche au dessus de l'application (ici, le framework)
- DI (Injection de Dépendances) : un objet ne se préoccupe pas de l'instanciation de ses dépendances. Il ne se contente que de les utiliser.

Services et DI

➔ Injection de dépendances

```
// fichier znk-hello.controller.js
export class ZnkHelloController {
  'ngInject';
  // Permet au plugin ng-annotate de résoudre la DI
  // lors de la minification des fichiers

  // Injection du service dans le constructeur
  constructor(ZnkService) { this.ZnkService = ZnkService; }

  $onInit() {
    this.ZnkService.fetchUsers().then(users => this.users = users);
  }
}
```

REST

➔ Service $http

- Permet d'exécuter des requêtes Ajax sur une architecture REST (`GET, POST, PUT, DELETE, ...`)
- Basé sur les **promesses**
  - Représente le résultat (éventuel) d'une opération asynchrone
  - Peut avoir 3 états : **pending / resolved / rejected**
  - Evite le "callback hell"

REST

➔ Service $http

- API
  - `get(url, [config])`
  - `post(url, data, [config])`
  - `put(url, data, [config])`
  - `delete(url, [config])`
- Objet de configuration (optionnel)
  - `headers`
  - `params` (query string)

REST

➔ Service $http

```
// fichier znk.service.js
export class ZnkService {
  'ngInject'
  constructor($http) { this.$http = $http; }

  fetchUsers() { return this.$http.get('/api/users'); }
}
```

```
// fichier znk-hello.controller.js
$onInit() {
  this.ZnkService.fetchUsers().then(
    (users) => { this.users = users; }, // resolved
    (error) => { console.log(error); }  // rejected
  );
}
```

Filtres

- Permet de mettre en forme une donnée
  - Conserve un modèle de donnée propre
  - Pilote la mise en forme depuis la vue

```
<span>{{ data | filter1 filter2:param1 | filter3:param1:param2 }}</span>
```

- Filtres proposés par AngularJS
  - `currency, date, filter, json, limitTo, lowercase, number, orderBy, uppercase`

Filtres

➔ filter

- Le filtre **`filter`** est très important et modulable
- Il permet de filtrer un tableau de données de 3 façons ≠

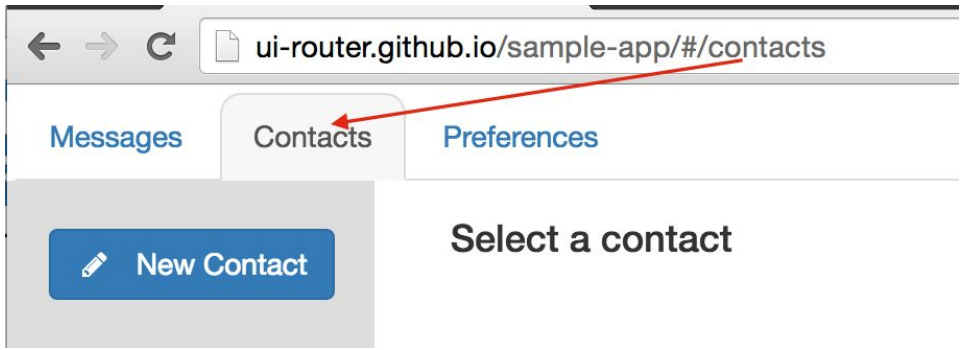
```
<h1>{{ data | filter:'value' }}</h1>
<h1>{{ data | filter:{ prop1: 'value1', prop2: 'value2' } }}</h1>
<h1>{{ data | filter:$ctrl.search }}</h1>
```

```
// fichier znk-hello.controller.js
search(item) { return item === 'hello'; }
```

Router

➔ UI-Router

- Permet de mettre à jour l'URL du navigateur en fonction de la navigation dans la **SPA** (Single Page App)
- Changements d'URL □ changements dynamiques de vues (sans rechargement de la page !)



Router

➔ UI-Router

```
npm install --save angular-ui-router
```

- Représentation des vues (templates HTML) + URLs par des états

```
// fichier znk.states.js
export function znkStates($stateProvider) {
  'ngInject';
  $stateProvider
    .state({ name: 'zenika', url: '/zenika', template: '<znk-hello></znk-hello>' })
    .state({ ... });
}
// fichier index.js
angular
  .module('znkApp', ['ui.router'])
  .config(znkStates);
```

Router

➔ UI-Router

- **Viewport** : lorsqu'un état est activé, le template correspondant est chargée dans la balise **<ui-view>**

```
<body ng-app="znkApp">
  <ui-view></ui-view>
</body>
```

- **Links** : utilisation de la directive **ui-sref** avec un état au lieu d'une URL

```
<a ui-sref="home">Home</a>
<a ui-sref="zenika">Zenika</a>
```

Questions ?

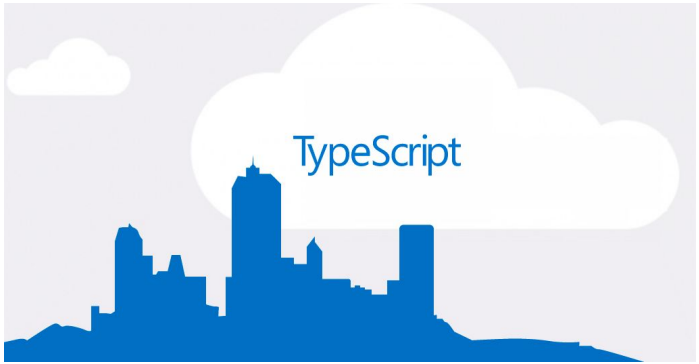
3

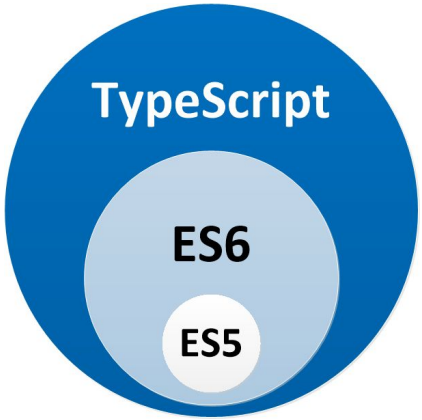
TypeScript

TypeScript

➔ Introduction

- Langage créé par **Anders Hejlsberg** en 2012
- Open-source, maintenu par Microsoft (actuellement **v 2.0**)
- Phase de compilation : fichiers **.ts** □ fichiers **.js**
- Typage statique fort





TypeScript

➔ Types

- Déclarer une variable

```
const done: boolean = true;
const height: number = 10;
const name: string = 'John Doe';
let something: any;
let people: string[] = ['Alice', 'Bob'];
```

- Inférence de type

```
const name = 'John Doe'; // type string
const numbers = [1, 2, 3]; // type number[]
```

TypeScript

➔ Fonctions

- Paramètres typés

```
function multiply(a: number, b: number) {
  return a * b;
}
multiply(2, 5); // ok
multiply(2, '5'); // erreur de compilation
```

- Signature typée

```
function multiply(a: number, b: number): number { ... }
multiply(2, 5).toUpperCase(); // erreur de compilation
```

TypeScript

➔ Classes

- Système de classes et interfaces similaire à la POO
- Le code JS généré utilisera le système de **prototype** (si < ES6)
- Propriétés / Méthodes accessibles via **this**

```
class User {
  name: string;
  constructor(name: string) {
    this.name = name;
  }
}
const john = new User('John');
```

TypeScript

➔ Classes

- 3 scopes de visibilité pour les propriétés
  - **`private`** / **`protected`** / **`public`** (par défaut)
- Possibilité de définir des propriétés statiques (**`static`**)

```
class User {
  private name: string;
  ...
}
const john = new User('John');
console.log(john.name); // erreur de compilation
```

TypeScript

➔ Classes

- Seconde version pour initialiser des propriétés

```
class User {
  constructor(public name: string) {}
}

// équivalent à

class User {
  name: string;
  constructor(name: string) { this.name = name; }
}
```

➔ Interfaces

- Utilisées par le compilateur pour vérifier la cohérence des différents objets
- Plusieurs cas d'utilisation possibles
  - Vérification des paramètres d'une fonction
  - Vérification de la signature d'une fonction
  - **Vérification de l'implémentation d'une classe**

TypeScript

➔ Interfaces

- Vérification de l'implémentation d'une classe
- Erreur de compilation tant que la classe ne respecte pas le contrat défini par l'interface

```
interface Shape {
  setColor(color: string);
}

class Circle implements Shape {
  private color: string;
  setColor(color: string) { this.color = color; }
}
```

Questions ?

4

Angular 2

Présentation

- Framework créé par **Google** (annoncé en **2014**)
- Réécriture totale avec reprise de certains concepts
- 1ère bêta : **octobre 2014**, version officielle : **septembre 2016**
- Architecture orientée **composants**
- Optimisé pour les mobiles
- http://angular.io

➔ Points forts

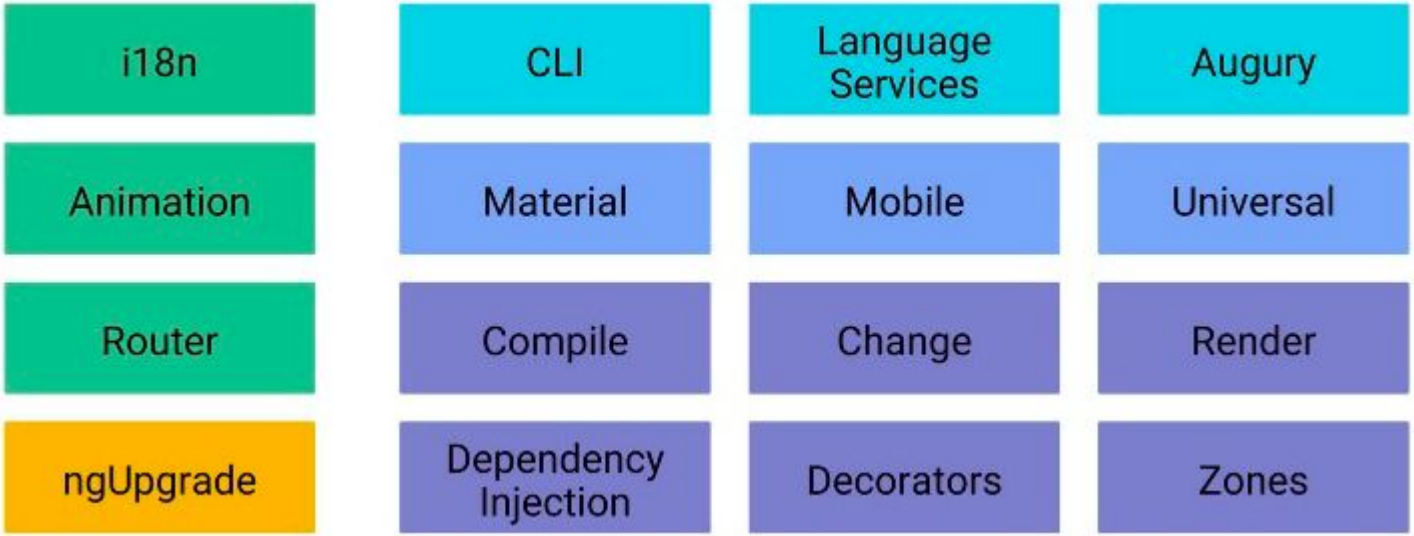
- API plus simple qu'avec AngularJS
- Basé sur les standards du web
- Performance de l'**API Change Detection**
- Server Side Rendering (**Universal**)
- Migration AngularJS Angular 2 : `ngUpgrade`
- Développé avec **TypeScript**

➔ Points faibles

- Nouvelle phase d'apprentissage
- Faible écosystème (pour le moment…)
- Applications AngularJS incompatibles
  - Mais collaboration possible avec `ngUpgrade`
- Nouveaux concepts à apprendre
  - `Zone`
  - `RxJS`

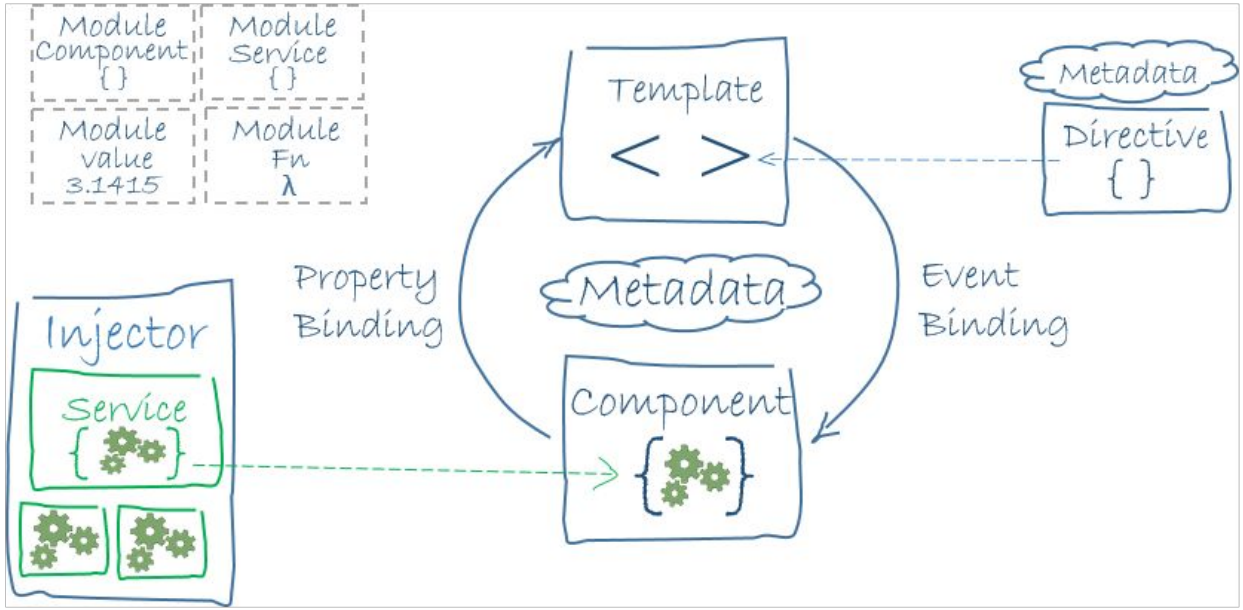
Présentation

➔ Une plateforme

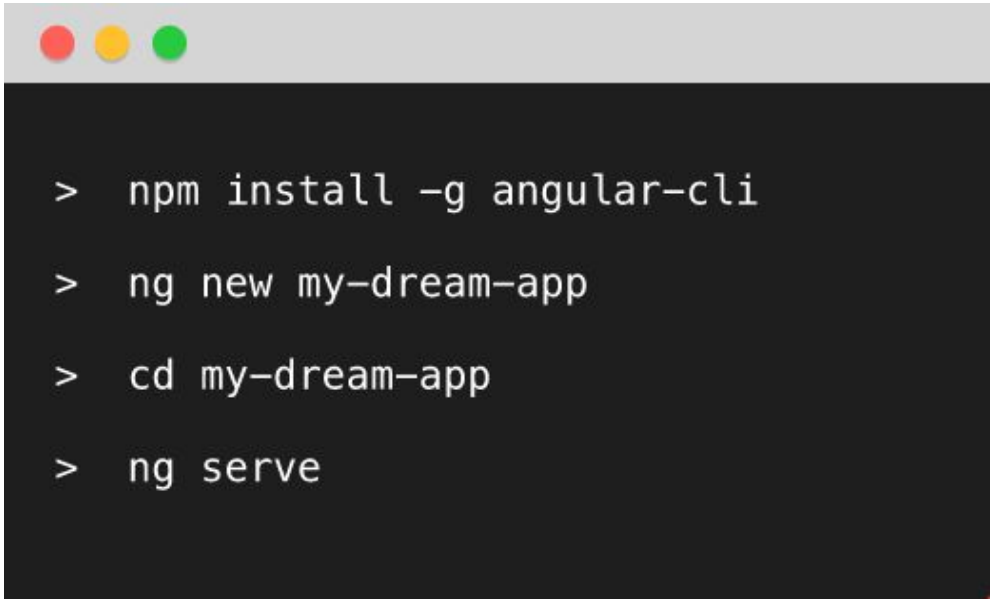


Présentation

➔ Architecture



Premiers pas



Premiers pas

- Module principal : `/src/app/app/module.ts`
- Fichiers sources : `src/app/*.(html|ts)`
- Bootstrap de l'application : `/src/main.ts`
- Lancement de l'application

```
$ npm start (ou ng serve)
```

  - Server de développement (`localhost:4200`)
  - Live reload

Template

➔ Property binding

- Affichage d'une donnée : `{{ expression }}`
- Possibilité de définir une valeur pour une propriété du DOM
- Un attribut HTML est statique, une propriété est dynamique
- Syntaxe : `[propertyName] = "expression"`

```
<span>Hello, my name is {{ user.firstName }}</span>
<input [value]="user.firstName">

<button [disabled]="isFormValid()">Save</button>
<a [href]="homeURL">Go to home</a>

<hero [data]="currentHero"></hero>
```

Template

➔ Event binding

- Association d'une expression à un évènement
  - HTML natif (**`click`**, **`blur`**, …)
  - Créé spécialement pour l'application
- Les méthodes et propriétés utilisées **doivent** être définies dans la classe associée
- Syntaxe : **`(eventName) = "expression"`**

```
<button (click)="save()">Save</button>
<hero (deleted)="onHeroDeletedEvent()"></hero>
```

Template

➔ Evénements

- Possibilité de récupérer le contexte de l'événement (données) via l'objet **$event**
- Cet objet peut être utilisé dans les expressions Angular 2

```
<input [value]="user.firstName"
       (input)="user.firstName = $event.target.value">
```

Template

➔ “Banana in the Box”

- Two-way data-binding

```
<input [(ngModel)]="user.firstName">
```

- Sucre syntaxique pour l’expression suivante

```
<input [ngModel]="user.firstName"
       (ngModelChange)="user.firstName = $event">
```

Components

➔ Input

- Utilisation du décorateur `@Component`

```
import { Component, Input } from '@angular/core';
import { Product } from './product/product';

@Component({
  selector: 'znk-product', // Sélecteur CSS
  template: `<article class="red">{{ product.name }}</article>`
  styles: [`.red { color: #F00 }`]
})
export class ZnkProductComponent {
  @Input() product: Product;
}

// Utilisation dans un template
<znk-product [product]="products[0]"></znk-product>
```

Components

➔ Input

- Avec `templateUrl` et `styleUrl`

```
import { Component, Input } from '@angular/core';
import { Product } from './product/product';

@Component({
  selector: 'znk-product',
  templateUrl: './znk-product.component.html'
  styleUrl: './znk-product.component.css'
})
export class ZnkProductComponent {
  @Input() product: Product;
}
```

Components

➔ Output

- Définition d'un événement personnalisé

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

@Component({
  selector: 'znk-timestamp',
  template: `<button (click)="timestampEvent.emit(Date.now())"></button>`
})
export class ZnkTimestampComponent {
  @Output() timestampEvent = new EventEmitter<number>();
}

// Utilisation dans un template
<znk-timestamp (timestampEvent)="onTimestampEvent($event)"></znk-timestamp>
```

Components

➔ NgModule

- Tous les composants nécessaires à l'application doivent être déclarés dans un module (`@NgModule`)

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppComponent } from './app.component';
import { ZnkProductComponent } from './product/znk-product.component';

@NgModule({
  declarations: [AppComponent, ZnkProductComponent],
  imports: [BrowserModule]
})
export class AppModule {}
```

Directives Angular 2

➔ ngClass

- Ajoute ou enlève des classes CSS
- [ngClass] = "{ class1: hasClass1, class2: hasClass2 }"

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

@Component({
  selector: 'znk-hello',
  template: `<button [ngClass]="{ disabled: isDisabled }"
    (click)="toggle(!isDisabled)">Click me!</button>
  `
})
export class ZnkHelloComponent {
  isDisabled: boolean = false;
  toggle(disabled) { this.isDisabled = disabled; }
}
```

Directives Angular 2

➔ ngIf

- Ajoute ou enlève des éléments HTML en fonction d'une condition

```
<div *ngIf="isVisible()">
  <span>Hello, World!</span>
</div>
```

Directives Angular 2

➔ ngFor

- Permet de dupliquer un template HTML pour chaque élément d'une collection
- Correspond à la directive **ng-repeat d'AngularJS**

```
<li *ngFor="let user of users; let i = index">
  {{ user.firstName }}
</li>
```

- Angular 2 met à disposition 4 variables
  - `index, last, even, odd`

Services et DI

➔ Injecteurs

- Élément utilisé pour injecter des services
- Possibilité de configurer un injecteur par composant (impossible avec AngularJS)
- Configuration d'un injecteur
  - De manière **globale** : **`providers (@NgModule)`**
  - De manière **locale** : **`providers (@Component)`**
- Services (**singletons**) injectés via le constructeur de la classe

Services et DI

➔ Injecteurs

- Configuration globale

```
// fichier app.module.ts
import { ZnkService } from './znk.service';
@NgModule({
  providers: [ZnkService]
})
export class AppModule {}
```

- Configuration locale

```
// fichier app.component.ts
import { ZnkService } from './znk.service';
@Component({
  providers: [ZnkService]
})
export class AppComponent {}
```

Services et DI

➔ DI dans un composant

```
// fichier app.component.ts
import { ZnkService } from './znk.service'

@Component({
  selector: 'znk-app',
  template: `<h1>Hello, World!</h1>`,
  providers: [ZnkService]
})
export class AppComponent {
  constructor(private znkService: ZnkService) {
    console.log(znkService.hello());
  }
}
```

Services et DI

➔ DI dans un service

- Utilisation du décorateur `@Injectable`

```
import { Injectable } from '@angular/core';
import { LoggerService } from './logger.service';

@Injectable()
export class ZnkService {
  constructor(private loggerService: LoggerService) {}

  hello() {
    this.loggerService.debug('hello method called');
    return 'Hello, World!';
  }
}
```

Http

➔ Observable

- Le service **`Http`** est basé sur le pattern **`Observable`**
- Permet de traiter des flux de données asynchrones
  - Requêtes Ajax
  - WebSocket
  - Evénements JavaScript
- Utilisation des méthodes dérivées de la PF
  - `map`
  - `filter`
  - `reduce`
  - …

Http

➔ RxJS

- `RxJS` permet de manipuler les `Observables`
- https://github.com/Reactive-Extensions/RxJS

```
fetchDataFromRemote1()
  .debounce(300)
  .filter(response => response != null)
  .flatMap(response => fetchDataFromRemote2(response))
  .map(response => response.json())
  .subscribe(data => console.log(data));
```

- `subscribe` : méthode à appeler pour récupérer les données

Http

➔ Appels REST

- Service disponible via **`HttpModule`** (à importer dans le module applicatif)

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

@Injectable()
export class ContactService {
  constructor(private http: Http) {}

  fetchContacts() { return this.http.get('/api/contacts'); }
}
```

Http

➔ Appels REST

- Requête de type `POST` avec surcharge des `Headers`

```
import { Http, Headers } from '@angular/http';

export class ContactService {
  constructor(private http: Http) {}

  save(contact) {
    const headers = new Headers();
    headers.set('Content-Type', 'application/json');
    return this.http.post('/api/contacts',
      JSON.stringify(contact), { headers });
  }
}
```

Http

➔ Appels REST

- Récupération des données dans un service (via **Http**) et affichage dans un composant

```
@Component({ ... })
export class AppComponent implements OnInit {
  contacts: Contact[];
  constructor(private contactService: ContactService) {}

  ngOnInit() {
    this.contactService.fetchContacts()
      .map(response => response.json())
      .subscribe(contacts => this.contacts = contacts);
  }
}
```

Pipes

- Permet la manipulation / mise en forme d'une donnée avant son affichage (similaires au filtres d'AngularJS)
- Pipes proposés par Angular 2 (`@angular/common`)
  - `lowercase, uppercase, currency, decimal, percent, date, json, slice, i18nPlural, i18nSelect, async`
- Possibilité de chaîner les pipes les uns à la suite des autres via le caractère `|`
- Ajout de paramètres via le caractère `:`

Pipes

```
<span>{{ birthDate | date:'YYYY-MM-DD' | uppercase }}</span>
<span>{{ price | currency:'EUR':true:'2.2-4' }}
<span>
  {{ messages.length | i18nPlural: { '=0': 'None',
                                     '=1': 'One message',
                                     '=other': '# messages' }

  }}
</span>
<!-- contact$ est un Observable -->
<span>{{ contact$ | async }}</span>
```

Router

- Permet de naviguer dans l'application sans rechargement de la page
- 3ème implémentation depuis le début d'Angular 2
- Prise en compte des différents cas d'utilisation : authentification, login, permissions, ...
- Router orienté composant
  - Association d'un composant avec une URL

Router

➔ Configuration

- Utilisation de la méthode `RouterModule.forRoot`
- `forRoot` prend en paramètre un tableau de `Route`

```
// fichier app.routing.ts
import { ModuleWithProviders } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';

const appRoutes: Routes = [
  { path: '', redirectTo: 'home', path: 'full' },
  { path: 'home', component: HomeComponent },
  { path: 'products', component: ProductsComponent }
];

export const routing: ModuleWithProviders = RouterModule.forRoot(appRoutes);
```

Router

➔ Configuration

- Import du module **routing** dans le module principal

```
// fichier app.module.ts
import { routing } from './app.routing';

@NgModule({
  imports: [
    BrowserModule,
    routing
  ],
  ...
})
export class AppModule {}
```

Router

➔ router-outlet

- La directive **router-outlet** est le point d'insertion des composants dans le template du composant principal

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

@Component({
  selector: 'znk-app',
  template: `
    <header><h1>Hello, ZnkApp!</h1></header>
    <router-outlet></router-outlet>
  `
})
export class AppComponent {}
```

Router

➔ routerLink

- La directive **routerLink** permet de naviguer d'une route à une autre

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

@Component({
  selector: 'znk-home',
  template: `
    <a [routerLink]=['products']>Products</a>
    <a [routerLink]=['products', 1]>Product 1</a>
  `
})
export class ZnkHomeComponent {}
```

Forms

- Basé sur les mécanismes standards des formulaires HTML
- Supporte les types de champs de saisie habituels et les validations natives
  - `input[text], input[text], input[text], input[text], input[text], input[text]`
  - `Select`
  - `textarea`

Forms

➔ Principe

- Associer des champs de saisie à des propriétés du composant avec **`ngModel`**
- Nommer les champs grâce à l'attribut **`name`**
- Ajouter des validateurs (**`required`**, **`pattern`**, etc.)
- Appeler une méthode du composant pour traiter le formulaire (event **`ngSubmit`**)

Forms

➔ NgForm

- La directive **NgForm** est automatiquement associée à chaque balise **<form>**
- Accès au formulaire dans le DOM grâce à l'écriture **#znkForm = "ngForm"**

```
<form #znkForm="ngForm" novalidate>
  <button type="submit" [disabled]="!znkForm.valid">Save</button>
</form>
```

Forms

➔ Exemple complet

```
<form #znkForm="ngForm" novalidate (ngSubmit)="onSubmit(znkForm.value)">

  <input type="text"
    [(ngModel)]="contact.firstName"
    name="firstName"
    #firstName="ngModel" required>

  <span *ngIf="firstName.dirty && !firstName.valid">
    First name is required
  </span>

  <button type="submit" [disabled]="!znkForm.valid">Save</button>
</form>
```

Forms

➔ Validation

- Angular 2 expose 5 propriétés au niveau du formulaire et de chacun des champs de saisie
  - `valid`
  - `pristine / dirty`
  - `untouched / touched`
- Des classes CSS correspondantes sont appliquées aux éléments
  - `ng-valid, ng-invalid, ng-pristine, ng-dirty, ng-untouched, ng-touched`

Questions ?