Table of Contents

Pratique

Pourquoi une version 2 ?

Les besoins évoluent, les frameworks JS évoluent, le language lui-même évolue. Il faut donc un framework utilisant cette évolution du langage et de ce besoin.

Quels sont les besoins ?

- Créer une application web
- Créer un logiciel avec Electron ou WebKit
- Créer une application mobile avec Cordova
- Etc

Le Javascript n'est plus maintenant utilisé seulement pour dynamiser quelques pages web !

Javascript a t-il évolué ?

Oui ! Mi-2015, une nouvelle version est sortie : EcmaScript 6. Une vague de nouveautés a apportée un coup de jeunesse au langage.

source code

Préface

Pour qui ?

Ce e-book est destiné aux développeurs voulant se familiariser avec le framework AngularJS. Plusieurs points sont abordés pour créer une application web.

Quel est le niveau requis ?

Il n'est as requis d'avoir des connaissances dans la première version d'Angular. Idéologie est toujours présente mais la conception est différente. Cependant, il très recommandé de connaître Javascript. Même si nous abordons ES6 (ou ES2015), savoir certaines subtilités de Javascript permet d'éviter quelques blocages sur l'écriture du code.

CopyRight

ECMAScript6 et TypeScript

Pour utiliser AngularJS 2, il faut préalablement savoir utiliser ES6 et TypeScript. Plusieurs fonctionnalités sont présentes :

- Classes
- Les fonctions fléchées
- Les chaînes de caractères en format "Template"
- Constantes et Scoped-Variables
- Déclaractions déstructurées des variables
- Modules
- Spread et Rest
- Différentes syntaxes des objets et des tableaux

Transformer ES6 en ES5 avec BabelJS

ES6 est une nouvelle manière d'écrire du Javascript, mais certains navigateurs ne lisent pas cette nouvelle syntaxe. En tant que développeur, nous n'allons pas nous bloquer à cause de ces navigateurs web, nous utilisons déjà ES6. Pour que tous les utilisateurs puissent lire Javascript, nous allons transformer le code ES6 vers ES6. Nous parlons alors de transpilation.

Pour cela, nous utilisons un outils nommé BabelJS

Le lien ci-contre indique tous systèmes supportant BabelJS : https://babeljs.io/docs/setup.

Juste pour tester, nous allons utiliser Babel CLI. Une simple ligne de commande pour transpiler notre code.

Tout d'abord, installez :

```
npm install --save-dev babel-cli
```

Ensuite, si votre dossier contenant les fichiers Javascript se nomme `app` , ajoutez dans le fichier `package.json` , une nouvelle commande :

```
{
  "name": "my-project",
  "version": "1.0.0",
   "scripts": {
     "build": "babel src -d app"
   },
  "devDependencies": {
    "babel-cli": "^6.0.0"
  }
}
```

Ici, la ligne est `"build": "babel src -d app"` . Vous pouvez exécuter la ligne en tapant :

```
npm run build
```

let, Portée des variables

EcmaScript6 ajoute un mot-clé : `let`. Il s'utilise lors de la déclaration d'un variable.

La portée en ES5

Voici un code avec EcmaScript5 :

```
var foo = 1;

if (true) {
  var foo = 2;
  console.log(foo);
}

console.log(foo);
```

Que vaut le premier `console.log()` ? `2`, puisque nous redéclaration la variable `foo`. Pour le deuxième `console.log()`, `foo` affichera encore `2`.

Une portée plus précise avec let

Voici un code avec `let` :

```
let foo = 1;

if (true) {
  let foo = 2;
  console.log(foo);
}

console.log(foo);
```

Le premier `console.log()` affiche `2`. Le deuxième `console.log()` affiche `1`. On remarque que la portée de la variable est plus précise, c'est à dire que `foo` vaut 2 seulement dans le scope de la condition. En dehors, `foo` vaut `1`.

Utilité avec l'asynchrone

`let` devient très utile avec l'asynchrone. Prenons une problématique. En HTML, nous avons ceci :

```
<!doctype html>
<html>
    <head>

    </head>
    <body onload="load()">
        <button id="test0">Test 0</button>
        <button id="test1">Test 1</button>
        <button id="test2">Test 2</button>
    </body>
</html>
```

Une page avec 3 boutons. Le premier affiche `0`, le deuxième `1` et le troisième `2`. En utilisant une boucle, le code pourrait être le suivant :

```
function load() {
   for (var i=0 ; i < 2 ; i++) {
     var el = document.getElementById('test' + i);
     el.addEventListener("click", () => {
       console.log(i);
     }, false);
   }
 }
```

Si vous testez, ça ne fonctionne pas car, peut importe le bouton cliqué, `3` est affiché. Pourquoi ? Car la boucle est synchrone est s'éxécute 3 fois. Lorsque la boucle est terminée, `i` vaut `3`. C'est ensuite **après** que l'événement `click` se déclenche. Donc `i` va valoir toujours `3`. Comment remédier à ce problème ? En ES5, nous pouvons changer la portée de la variable de `i` en le passant en paramètre d'une fonction :

```
function click(i) {
  var el = document.getElementById('test' + i);
  el.addEventListener("click", () => {
    console.log(i);
  }, false);
}

function load() {
   for (var i=0 ; i < 2 ; i++) {
     click(i);
   }
 }
```

Mais `let` évite la fonction :

```
function load() {
   for (let i=0 ; i < 2 ; i++) {
     let el = document.getElementById('test' + i);
     el.addEventListener("click", () => {
       console.log(i);
     }, false);
   }
 }
```

Les classes

Si vous utilisez les classes dans d'autres languages comme PHP, Java et cie, vous n'avez pas vraiment de surprises sur l'utilisation des classes dans le JS.

La syntaxe est la suivante :

```
class Warrior {

    // constructeur
    constructor(name) {
        this.name = name;
    }

    attack() {

    }

}
```

L'indication de méthode publique, privé ou `protected` n'existe pas. Une convention de nommage peut être utilisée pour indiquer qu'une méthode est privé. Pour cela, on préfixe le nom d'une méthode avec `_` :

```
class Warrior {

    // méthode privée
    _refresh() {

    }

}
```

Accesseur et modificateur

Voici une simple classe.

```
class Warrior {

    constructor(name) {
      this.name = name;
    }

}

var warrior = new Warrior('Link');
console.log(warrior.name);
```

Nous aimerions mettre un accesseur et un modificateur :

```
class Warrior {

    constructor(name) {
      this.setName(name);
    }

    getName() {
      return this._name;
```

```
    }

    setName(name) {
      this._name = name;
    }

}

var warrior = new Warrior('Link');
console.log(warrior.getName());
```

Nous mettons un `_` avant le nom pour indiquer que la propriété est privé. Notre classe est bien écrite mais nous pouvons aller plus loin avec ES6 :

```
class Warrior {

    constructor(name) {
      this.name = name;
    }

    get name() {
      return this._name;
    }

    set name(name) {
      this._name = name;
    }

}

var warrior = new Warrior('Link');
console.log(warrior.name);
```

Nous accédons à la propriété `name`, qui a été encapsulée dans notre classe.

Héritage

Pour l'héritage, nous utilisons le mot clé `extends` :

```
class Character {

  constructor(name) {
    this.name = name;
  }

  get name() {
    return this._name;
  }

  set name(name) {
    this._name = name;
  }

}

class Warrior extends Character {

}

var warrior = new Warrior('Link');
console.log(warrior.name);
```

Nous pouvons accéder à la méthode parente avec le mot clé `super` :

```
class Character {

  constructor(name) {
    this.name = name;
  }

  refresh() {

  }

  get name() {
    return this._name;
  }

  set name(name) {
    this._name = name;
  }

}

class Warrior extends Character {

  constructor(name) {
    super(name);
    this.hp = 10;
  }

  refresh() {
    super.refresh();
  }

}

var warrior = new Warrior('Link');
console.log(warrior.name);
```

Les fonctions fléchées

Mettons une problématique pour se rendre compte de l'utilité des fonctions fléchées.

```
class Warrior {

    constructor() {
        this.name = 'Clad';
    }

    displayName() {

        var fnName = function() {
           console.log(this.name); // 2
        }

        console.log(this.name); // 1

        fnName();

    }

}
```

Le premier log exécuté va afficher le nom du guerrier, soit `Clad`. Mais, si vous connaissez un peu le JS, vous savez que le deuxième va afficher `undefined`. Pourquoi ?

Car le `this` du premier log fait référence à la classe, alors que, le `this` du deuxième log fait référence à la fonction anonyme.

Solution ? Oui, avec les fonctions fléchées :

```
class Warrior {

 constructor() {
   this.name = 'Clad';
 }

 displayName() {

   var fnName = () => {
     console.log(this.name); // 2
   }

   console.log(this.name); // 1
   fnName();

 }

}
```

Les objets

Nous avons l'habitude d'utiliser des objets :

```
var obj = {};
```

L'écriture en ES6 des objets permet de simplifier plusieurs points

Racourcir l'écriture d'une paire clé/valeur

Il arrive très souvent que nous souhaitons passer une variable dans un objet :

```
let url = 'angular.fr';
let obj = {url: url};
```

Lorsque le nom de la variable est équivalente au nom de la clé de l'objet, nous pouvons raccourcir l'écriture :

```
let url = 'angular.fr';
let obj = {url};
```

Racourcir l'écriture d'une paire clé dynamique/valeur

Si le nom de la clé d'un objet se trouve dans une variable, nous devons d'abord initialiser l'objet et lui donner une clé et sa valeur :

```
let obj = {};
let nameProp = 'url';
obj[nameProp] = 'angular.fr';
```

En ES6, nous pouvons raccourcir cette écriture :

```
let nameProp = 'url';
let obj = {
  [nameProp]: 'angular.fr'
}
```

Parcourir un tableau

Nous avons beaucoup l'habitude de parcourir un tableau avec la boucke `for` :

```
let names = ['Sam', 'Jim', 'Ana'];
for (let i=0 ; i < names.length ; i++) {
  let name = names[i];
}
```

Si nous n'avons pas besoin de la variable `i` (sauf pour accéder à la valeur du tableau), nous pouvons raccourcir cette écriture par :

```
let names = ['Sam', 'Jim', 'Ana'];
for (let name of names) {
```

```
}
```

> La méthode `forEach()` sur un tableau n'est pas vraiment conseillé. Elle est beaucoup moins performante qu'une boucle `for`

Les templates

Pour concaténer deux chaînes de caractères, nous faisons habituellement :

```
console.log('Bonjour ' + 'Sam');
```

et avec une variable :

```
var name = 'Sam';
console.log('Bonjour ' + name);
```

Avec ES6, nous simplifions cette écriture avec les chaînes de caractères de templating :

```
var name = 'Sam';
console.log(`Bonjour ${name}`);
```

Notez qu'on utilise l'accent grave (touche 7) et non des simples ou doubles guillemets. Cette notation est très utile pour un texte avec des sauts de ligne :

```
console.log(`Exsistit autem hoc loco quaedam quaestio subdifficilis,
  num quando amici novi, digni amicitia, veteribus sint anteponendi,
  ut equis vetulis teneros anteponere solemus. Indigna homine dubitatio!
  Non enim debent esse amicitiarum sicut aliarum rerum satietates; veterrima quaeque,
  ut ea vina, quae vetustatem ferunt, esse debet suavissima; verumque illud est,
  quod dicitur, multos modios salis simul edendos esse, ut amicitiae munus expletum sit.`);
```

Opérateurs Spread et Rest

Opérateur Spread

En EcmaScript5, la fonction `apply()` permet d'exécuter une fonction en envoyant un tableau de paramètres :

```
var add = function(a, b) {
  return a + b;
}

add.apply(add, [10, 5]);
```

En ES6, nous utilisons l'opérateur Spread :

```
let add = function(a, b) {
  return a + b;
}

let args = [10, 5];
add(...args);
```

Les éléments du tableau sont donc décomposés en paramètres. Nous pouvons faire cette action pour fusionner deux tableaux :

```
let numbers1 = [1, 2, 3];
let numbers2 = [...numbers1, 4, 5, 6];
// [1, 2, 3, 4, 5, 6]
```

Cela peut fonctionner pour un objet :

```
let data = {url: 'angular.fr', port: 80}
let config = {password: 'azerty', ...data};
```

Opérateur Rest

L'opérateur Rest permet de récupérer des arguments indéterminés

```
function add(...numbers) {
  return numbers[0] + numbers[1];
}

add(10, 5);
```

Dans la même idée, nous pouvons parcourir les paramètres :

```
function cal(type, ...numbers) {
  let result = 0;
  if (type == 'add') {
    for (let n of numbers) {
      result += n;
    }
  }
  return result;
}
```

```
cal('add', 10, 5, 8, 2);
```

Importer / Exporter des modules

> NodeJS et les navigateurs ne lisent pas encore nativement les modules en ES6. Cependant, la syntaxe est équivalente pour TypeScript

Un fichier peut renvoyer n'importe quelle valeur :

Exporter une classe :

```
export class User {

}
```

Dans un fichier différent, nous pouvons récupérer cette classe en important le fichier :

```
import {User} from './user.js'
```

L'exportation peut être une valeur de votre choix :

Une constante :

```
export const URL = 'angular.fr';
```

Un objet :

```
export var config = {
  url: 'angular.fr'
}
```

Une fonction :

```
export function config() {
  return 'angular.fr';
}
```

Les promesses

Pourquoi les promesses ?

Imaginons la scène suivante.

Vous souhaitez faire du pain. Malheursement, il vous manque de la farine. Vous demandez donc à votre ami de chercher de la farine au commerce du coin. Pendant ce temps, vous n'allez pas attendre son retour, vous allez à faire le ménage. Votre ami revient du commerce avec la farine. Enfin, vous pouvez continuer le pain ! Mais il manque la levure. Vous allez redemander à votre ami de repartir au commerce pour chercher la levure.

Les promesses se basent sur cette idée : gérer des difficultés asynchrones. En effet, si votre ami revient mes mains vides, il est impossible de continuer le pain. La promesse de faire du pain échoue.

Nous pouvons rédiger l'algorithme de la façon suivante :

```
Fonction (Faire du Pain)

  -> Chercher la farine
  -> Ensuite
  -> Chercher la levure
  -> Ensuite
  -> Faire le pain
  -> Retourner notre pain
```

Les instructions doivent s'effectuer fur et à mesure. Mais nous sommes dans un cheminement asynchrone. Les promesses permettent de résoudre ça

Ecrire une promesse

```
new Promise((resolv, reject) => {

});
```

Une promesse s'exécute automatiquement. Deux callbacks sont à déclenchées pour indique que la promesse est résolu ou rejeté.

```
new Promise((resolv, reject) => {
    setTimeout(resolv, 2000)
}).then(() => {
  return 42;
}).then((val) => {
  console.log(42);
});
```

Dans la promesse, nous effectuons une instruction asynchrone (`setTimeout`). Lorsque le timer est terminé, la promesse est résolu et passe au prochaine `then()` . Ce dernier renvoie une promesse ou une valeur synchrone.

Avec notre histoire sur le pain, nous pourrions imaginer une promesse de ce style :

```
function chercher(type) {
  return new Promise((resolv, reject) => {
      http.get(type, (err, ret) => {
        if (err) {
```

```
          reject(err);
        }
        resolv(ret);
      })
  });
}

function fairePain() {
  return chercher('farine').then(() => {
    return chercher('levure');
  })
}

fairePain().then(() => {
  console.log('terminé');
});
```

En admettant que `http` est un module pour faire une requête HTTP, nous piuvons créer plusieurs promises, les rendre modulables dans des fonctions externes et chaîner les promesses.

Récupérer les erreurs d'une promesse

Nous avons vu que nous utilisons `reject()` pour rejeter une promesse. Cela a pour but d'arrêter l'exécution de la promesse et de lancer une fonction `catch()` :

```
  new Promise((resolv, reject) => {
      setTimeout(() => {
        reject(new Error('erreur'));
      }, 2000);
  }).then(() => {
    console.log(42);
  }).catch((err) => {
    console.log(err);
  });
```

Le nombre `42` n'est pas affiché. La promesse passe directement dans `catch()` .

Pourquoi utiliser TypeScript ?

TypeScript permet d'étendre la syntaxe d'ES6. Nous avons des types mais aussi d'autres notions comme les décorateurs.

Pour utiliser TypeScript, vous pouvez l'installer avec NPM :

```
npm install -g typescript
```

Créez ensuite un fichier avec l'extension `.ts`. Nous devons ensuite transpiler le code avec du Javascript. Créez un fichier `main.ts` et tapez :

```
tsc main.ts
```

Si vous souhaitez transpiler à chaque modification sauvegardée, tapez :

```
tsc -w main.ts
```

Les types

TypeScript ajoute la notion de typage :

Boolean

```
let myvar:boolean = false;
```

Nombre

```
let myvar:number = 0;
```

Chaîne de caractères

```
let myvar:string = 'hello';
```

Tableau

```
let myvar:number[] = [1, 2, 3];
```

ou

```
let myvar:Array<number> = [1, 2, 3];
```

Objet

```
interface User {
  name: string
}

let myvar:User = {name: 'Sam'}
```

N'importe quelle valeur

```
let myvar:Array<number> = [1, 2, 3];
```

Dans une classe

Le constructeur ou une méthode contient des paramètres :

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

  }
}
```

Dans le cas où le paramètre est une autre classe ou interface :

```
class User {
  constructor(foo: Foo) {

  }
}
```

Les interfaces

Une interface permet juste de déclarer le schéma d'un objet. Lors de la transpilation, les interfaces ne sont pas utilisées en Javascript, elles serviront juste à TypeScript de valider le schéma d'un objet.

```
interface User {
  username: string;
}

let obj: User = {
  username: 'Sam'
}
```

L'interface donne la structure de notre objet `obj`

Les décorateurs

Un décorateur est une fonction invoquée avec le préfixe `@` et se trouve avant d'une classe, paramètre ou méthode. Elle permet de donner des informations à propos de cette classe, paramètre ou méthode.

Ainsi la classe suivante configurée avec la propriété `name` :

```
class Warrior {
  constructor() {
    this.name = 'Sam';
  }
}
```

Sera écrite avec un décorateur :

```
@MyDecorator({
  name: 'Sam'
})
class Warrior {

}
```

Nous écrivons avec une syntaxe de méta-programmation

Démarrer une application avec AngularJS.

Une application est le mélange de plusieurs modules composés de composants. Ces modules sont des sortes de paquets indépendants. Cela permet de les distribuer, de les réutiliser et de - surtout - de structurer l'application.

Pour démarrer à partir de sources existantes, utilisez le dépot Angular QuickStart.

Récupérer le dépot avec Git :

```
git clone  https://github.com/angular/quickstart  project
cd project
```

Installer les dépendances et démarrer l'application:

```
npm install
npm start
```

Démarrer avec Angular CLI

Si vous souhaitez profiter Angular CLI, installez-le avec

```
npm install -g @angular/cli
```

Après l'installation, tapez :

```
ng new PROJECT_NAME
```

où `PROJECT_NAME` est le nom de votre projet.

Rendez-vous dans le dossier et lancez le serveur pour vérifier que votre application fonctionne :

```
cd PROJECT_NAME
ng serve
```

Structure de l'application

La documentation d'AngularJS nous propose une structure (https://angular.io/docs/ts/latest/guide/style-guide.html#!#04-06) :

```
<project root>
  app
    core
      core.module.ts
      exception.service.ts|spec.ts
      user-profile.service.ts|spec.ts

    heroes
      hero
        hero.component.ts|html|css|spec.ts
      hero-list
        hero-list.component.ts|html|css|spec.ts
      shared
        hero-button.component.ts|html|css|spec.ts
        hero.model.ts
        hero.service.ts|spec.ts
      heroes.component.ts|html|css|spec.ts
      heroes.module.ts
      heroes-routing.module.ts

    shared
      shared.module.ts
      init-caps.pipe.ts|spec.ts
      text-filter.component.ts|spec.ts
      text-filter.service.ts|spec.ts

    villains
      villain
      ...
      villain-list
      ...
      shared
      ...
      villains.component.ts|html|css|spec.ts
      villains.module.ts
      villains-routing.module.ts

  app.component.ts|html|css|spec.ts
  app.module.ts
  app-routing.module.ts

main.ts
index.html
```

Le dossier `app` contient tous les modules :

- core
- heroes
- shared
- villains

Le dossier des modules contient des sous-dossiers. Ils correspondent aux composants.

Le dossier des composants contient

- Le fichier TypeScript qui définit le composant
- Le template en HTML
- Le CSS

- Le test unitaire (`*.spec.ts`)

Le dossier `shared` contient des fichiers utilisés sur les différents composants ou module (des services, pipes, etc.)

Le dossier `core` est le coeur de notre application avec des fonctions élémentaires pour le bon fonctionnement du produit.

Nous verrons plus en détails les différents parties de l'application. Retenez principalement la structure des dossiers.

Fichier de bootstraping

Au départ, il faut indiquer quel est module de démarrage. Autrement dit, quel est le point de démarrage dans notre application, dit `bootstraping`.

C'est donc dans le fichier `main.ts` que cela s'applique :

```
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';
import {AppModule} from 'app/app.module';

platformBrowserDynamic().bootstrapModule(AppModule);
```

Nous récupérons le module racine `AppModule` et nous indiquons à Angular de démarrer dessus. `platformBrowserDynamic` désigne que nous démarrons le module sur un navigateur. Effectivement, nous pourrions plus tard avoir un autre point de démarrage sur du mobile avec Cordova par exemple.

Créer un module

Créons notre premier module. C'est le module racine (ou module parent). Il va charger tous les autres modules et le premier composant.

Notre structure de l'application est la suivante :

```
<project root>
  app
    app.module.ts
  main.ts
  index.html
```

Le fichier `app.module.ts` contient le code suivant :

```
import {BrowserModule} from '@angular/platform-browser';
import {NgModule} from '@angular/core';

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

}
```

On déclare un module avec le décorateur `NgModule` . Trois propriétés sont requises :

Importer d'autres modules avec `imports`

Les autres modules vont construire les fonctionnalités de notre application. C'est un tableau contenant les différentes classes.

Nous avons importé le module `BrowserModule` . Il intègre les directives, pipes et services que nous utiliserons par la suite (Par exemple, l'utilisation directive `ngIf`)

Déclarer les composants du module avec `declarations`

Tous les composants créés doivent être déclarés dans ce tableau. Autrement dit, nous avons un tableau de composants. Le prochain chapitre expliquera les composants plus en détails

Composant racine avec `bootstrap`

Au départ, nous devons bien créer un premier composant. Ce composant est le parent de tous les composants. C'est lui qui indiquera quel selecteur dans notre page HTML définera la racine de l'application.

Nous indiquons donc le composant dans le tableau `bootstrap` . A 99% des cas, nous avons qu'un seul composant dans ce tableau.

Pour que module fonctionne, il faut un composant obligatoirement.

Création du composant racine

Nous allons créer le premier composant indispensable pour notre application. C'est le composant présent dans le module racine. Voici la structure de notre application :

```
<project root>
  app
    app.component.ts
    app.module.ts
  main.ts
  index.html
```

Le fichier `app.component.ts` contient le code suivant :

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

@Component({
  selector: 'app',
  template: '<p>Hello World</p>'
})
export class AppComponent {

}
```

Créer un composant est très simple. C'est le décorateur `Component` qui définit :

- Le sélecteur dans le DOM avec `selector`
- Le template avec `template`

Rappelez-vous, dans `index.html`, nous avons :

```
<app>Loading...</app>
```

Le contenu sera remplacé par le template et le composant s'éxécutera.

Ajouter le composant dans le module déjà créé.

Maintenant qu'on a créé le composant, ajoutons le dans le module racine présent dans le fichier `app.module.ts` :

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

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

}
```

Puisque c'est le module racine, on ajoute le composant dans `bootstrap`. Enfin, puisque c'est le composant fait partie du module, on l'ajoute dans `declarations`.

Création d'un module fonctionnel.

Passons maintenant sur la création d'un module fonctionnel. Le principe est très similaire à un module racine. La différence est que module est exportable et peut être utiliser dans d'autre projet.

Nous allons créer un module nommé "Navbar". C'est un module qui gére la barre de navigation. Nous créons donc un dossier nommé `navbar`. La structure du projet sera la suivante :

```
<project root>
  app
    navbar
      navbar.module.ts
      navbar.component.ts
    app.component.ts
    app.module.ts
  main.ts
  index.html
```

Voici le code de `navbar.component.ts` :

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

@Component({
  selector: 'navbar',
  template: '<p>Barre de navigation</p>'
})
export class NavbarComponent {

}
```

Rien de particuler, passons au module dans `navbar.module.ts` :

```
import {CommonModule} from '@angular/common';
import {NgModule} from '@angular/core';
import {NavbarComponent} from './navbar.component';

@NgModule({
  imports: [CommonModule],
  declarations: [NavbarComponent],
  exports: [NavbarComponent]
})
export class NavbarModule {

}
```

Deux choses vont changer par rapport au module racine :

1. Nous utilisons le module `CommonModule`
2. Nous exportons le composant racine du module

En fait, le module `BrowserModule` d'un module racine intègre déjà `CommonModule` avec d'autres services pour le rendu dans le navigateur. Ici, nous voulons surtout un module indépendant, réutilisable. Raison pour laquelle, nous utilisons seulement le module `CommonModule`.

Enfin, nous exportons le composant racine nommé `NavbarComponent`. Ce composant sera utilisable dans les autres modules. Comment ? La suite l'expliquera.

Déclaration du module dans notre application

Jusqu'à maintenant, nous avons créé un module indépendant. Nous pouvons le mettre dans n'importe quelle application utilisant AngularJS 2. Et justement, nous voulons mettre cette brique dans notre application courante.

Allons dans `app.module.ts` :

```
import {BrowserModule} from '@angular/platform-browser';
import {NavbarModule} from 'app/navbar/navbar.module';
import {NgModule} from '@angular/core';
import {AppComponent} from 'app/app.component';

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

}
```

Si vous avez compris le chapitre sur les modules, ce code ne devrait pas vous surprendre. Deux points :

1. On importe le module `NavbarModule`
2. On l'ajoute dans la propriété `imports`

Utiliser le module Navbar dans notre application

Maintenant qu'on l'a importé, nous pouvons utiliser le composant dans notre application. Par exemple, allez `app.component.ts` et mettez le sélecteur `<navbar>` dans le template :

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

@Component({
  selector: 'app',
  template: '<navbar></navbar> <p>Hello World</p>'
})
export class AppComponent {

}
```

Composants et directives

Comme nous l'avons vu, créer un composant n'est pas très complexe. Cependant un compostant est dynamique et change selon des données reçues.

Lire des propriétés dans le template

Notre structure :

```
<project root>
  app
    navbar
      navbar.module.ts
      navbar.component.ts
    app.component.ts
    app.module.ts
  main.ts
  index.html
```

Voici notre fichier `app.component.ts` :

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

@Component({
  selector: 'navbar',
  template: '<p>Bienvenue {{name}}</p>'
})
export class NavbarComponent {

  name: string;

  constructor() {
    this.name = 'Sam';
  }

}
```

Nous définissons la propriété `name`. Dans le construteur, nous lui donnons la valeur `Sam`. Remarquez comment nous lisons la valeur dans le template :

`{{nom de la propriété}}`

Passer des données dans le composant

Il est utile parfois que le composant reçoive des données externes. Voici comment le code est écrit :

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

@Component({
  selector: 'navbar',
  template: '<p>Bienvenue {{name}}</p>'
})
export class NavbarComponent {

  @Input() name: string;

}
```

Notez :

1. L'importation de la classe `Input`
2. L'ajout du décorateur sur la propriété `name` ;

Pour utiliser le composant dans un template :

```
<navbar name="Sam"></navbar>
```

Par exemple, dans notre fichier `app.component.ts` , nous pouvons assigner des valeurs au composant :

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

@Component({
  selector: 'app',
  template: '<navbar name="Sam"></navbar> <p>Hello World</p>'
})
export class AppComponent {

}
```

Si nous voulons passer des variables, l'écriture sera légèrement différente :

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

@Component({
  selector: 'app',
  template: '<navbar [name]="person.name"></navbar> <p>Hello World</p>'
})
export class AppComponent {

  person: any;

  constructor() {
    this.person = {
      name: 'Sam',
      age: 35
    }
  }

}
```

Les événements

Dans notre composant, nous souhaitons effectuer un événement. Par exemple, quand l'utilisateur clique sur un bouton, un compteur s'incrémente.

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

@Component({
  selector: 'count',
  template: `
    <p>{{n}}</p>
    <button (click)="up()"></button>
  `
})
export class CountComponent {

  n: number;

  up() {
    this.n++;
  }

}
```

Nous retrouvons des similitudes avec Angular1. La syntaxe change un peu. Nous mettons des parenthèses autour du nom de l'événement pour indiquer une valeur de sortie.

Envoyer la valeur en dehors du composant

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

@Component({
  selector: 'count',
  template: `
    <p>{{n}}</p>
    <button (click)="up()">Up</button>
  `
})
export class CountComponent {

  n: number = 0;
  @Output() numberChange: EventEmitter = new EventEmitter();

  up() {
    this.n++;
    this.numberChange.emit(this.n);
  }

}
```

Deux importations :

- EventEmitter : Emettre l'événement aux composants parents
- Output : Décorateur indiquant la propriété envoyant l'information.

La propriété `numberChange` est donc utilisé dans le template pour recevoir l'événement :

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

```
@Component({
  selector: 'app',
  template: `
    <div>
        <count (numberChange)="multiplicate($event)"></count>
      {{result}}
      </div>
  `
})
export class AppComponent {

  result: number;

  constructor() {
    this.result = 0;
  }

  multiplicate(val: any) {
    this.result = val * 2;
  }
}
```

Nous appelons donc `multiplicate``qui va juste multiplier la valeur du compteur par deux.

Tester

Envoyer la valeur en dehors du composant

Faire du double data-binding.

Le double data-binding consiste à mettre à jour une variable peu importe où l'information a été modifiée.

Par exemple, si nous avons le nom de l'utilisateur affiché sur les deux composants A et B. Si on modifie le nom sur A, il le sera aussi sur B et réciproquement.

En fait, nous avons une entrée *et* une sortie.

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

@Component({
  selector: 'count',
  template: `
    <p>{{n}}</p>
    <button (click)="up()">Up Counter</button>
  `
})
export class CountComponent {

  @Input() n: number = 0;
  @Output() nChange: EventEmitter = new EventEmitter();

  up() {
    this.n++;
    this.nChange.emit(this.n);
  }

}
```

Le composant du compteur ne change pas vraiment. Nous avons juste ajouté le décorateur `Input` pour récupérer la valeur en entrée.

Lorsque vous faites du data-binding, la propriété de sortie est le même nom que la propriété d'entrée suffixée de `Change`. Par exemple, ici :

- Propriété d'entrée : `n`
- Propriété de sortie : `nChange`

Pourquoi ? Car nous n'indiquerons pas l'attribut `nChange` dans le template. Cela se fera automatiquement par AngularJS. Voyez sur le composant parent :

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

@Component({
  selector: 'app',
  template: `
    <div>
      <p>{{val}}</p>
      <button (click)="up()">Up Parent</button>
      <count [(n)]="val"></count>
      </div>
  `
})
export class AppComponent {

  val: number = 0;

  up() {
    this.val++;
  }
```

```
}
```

Remarquez que nous retrouvons le bouton qui incrémente la propriété `val`. Cette dernière est propre au composant `AppComponent`. Regardez que nous faisons du data-binding en combinant l'entrée et la sortie : `[(n)]`. Ainsi, si vous cliquez sur l'un des boutons, la valeur se synchronise.

Cela revient à faire :

```
  <count (n)="val" [nChange]="up()"></count>
```

Tester

Faire du double data-binding

Les directives

Une directive permet de changer l'apparence et le comportement d'un composant

Changer l'apparence avec ngClass

La directive `ngClass` permet de changer la classe `class` et donc, de changer l'apparence

Voici un composant simple.

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

@Component({
  selector: 'app',
  template: `
    <p [ngClass]="{red: result%2, green: !(result%2)}">{{result}}</p>
    <button (click)="up()">Up</button>
  `
})
export class AppComponent {

  result: number;

  constructor() {
    this.result = 0;
  }

  up() {
    this.result++;
  }
}
```

Lorsque l'utilisateur clique, un compteur s'incrémente. Quand le nombre est pair, il est coloré en rouge sinon en vert. Pour réaliser cela, nous avons un fichier `style.css` à la racine :

```
.red {
  color: red;
}
.green {
  color: green;
}
```

La directive `ngClass` peut avoir deux valeurs :

- soit un tableau de classes. Par exemple `['red']` indiquera que l'élément aura la classe `red` ;
- soit un objet avec les classes en clé et un booléen en valeur. Par exemple : `{red: true}` indiquera que l'élément aura la classe `red` ;

Tester

Changer l'apparence avec ngClass

Gérer l'apparence avec ngStyle

Comme avec `ngClass`, nous pouvons changer l'apparence avec `ngStyle`. La différence est qu'on modifie l'attribut `style`. Pas besoin de fichier CSS externe :

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

@Component({
  selector: 'app',
  template: `
    <p [ngStyle]="{color: color}">{{result}}</p>
    <button (click)="up()">Up</button>
  `
})
export class AppComponent {

  result: number;
  color: string = 'red';

  constructor() {
    this.result = 0;
  }

  up() {
    this.result++;
    this.color = this.result % 2 ? 'red' : 'green';
  }
}
```

`ngStyle` est un objet avec la propriété CSS avec sa valeur CSS.

> Deux moyens d'écrire les valeurs CSS : Soit en kebab case : `font-weight` Soit en camel case : `fontWeight`

Tester

Gérer l'apparence avec ngStyle

Afficher et désafficher un élément avec ngIf

La directive `ngIf` va modifier la structure du DOM. Prenons un exemple :

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

@Component({
  selector: 'app',
  template: `
    <p>{{result}}</p>
    <button (click)="up()">Up</button>
    <p *ngIf="result%2==0">Pair</p>
  `
})
export class AppComponent {

  result: number;

  constructor() {
    this.result = 0;
  }

  up() {
    this.result++;
  }
}
```

Notre composant est un simple compteur. La syntaxe de `ngIf` est préfixée d'une astérisque pour indiquer que la balise `<p>` et ses enfants font partie un template à part entière.

Ici, nous envoyons un booléons pour qu'il affiche ou désaffiche l'élément. Plus précisément, il créé et supprime l'élément. `ngIf` peut consommer de la performance s'il était utilisé d'une manière importante.

Tester

Afficher et désafficher un élément avec ngIf

Répéter un affichage avec ngFor

`ngFor` est une directive importante pour votre application. Pourquoi ? Car vous recevez généralement des données de la part du serveur, et vous devez les afficher côté frontend :

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

@Component({
  selector: 'app',
  template: `
    <p *ngFor="let user of users">{{user.name}} : {{user.age}}</p>
  `
})
export class AppComponent {

  users: any[] = [
    {name: 'Sam', age: 45},
    {name: 'Jim', age: 33},
    {name: 'Ana', age: 17},
    {name: 'Lou', age: 4},
  ]

}
```

Ici, nous avons un tableau d'utilisateurs (directement dans le code, mais nous pouvons imaginer que ces données seront récupérées sur un serveur par la suite).

La valeur de `ngFor` reprend la syntaxe d'une boucle Javascript. La variable locale `user` n'est utilisable que dans l'élément ayant `ngFor`

Tester

Répéter un affichage avec ngFor - Partie 1

Utiliser d'autres variables locales

Il est parfois très utiliser de connaitre l'index, par exemple, pour afficher une numérotation à chaque tour de boucle. Il existe donc 5 variables locales :

- `index` : position de l'item. Commence à 0.
- `first` : booléen indiquant si l'item est le premier de l'itération
- `last` : booléen indiquant si l'item est le dernier de l'itération
- `even` : booléen indiquant si la position de l'item est paire
- `odd` : booléen indiquant si la position de l'item est impaire

Voici un code illustrant leur utilisation :

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

@Component({
  selector: 'app',
  styles: [
    `
      .red {
        color: red;
      }
```

```
    `
  ],
  template: `
    <p *ngFor="let user of users ; let i = index ; let isEven = even" [ngClass]="{red: isEven}">
     N°{{i}} --> {{user.name}} : {{user.age}}
    </p>
  `
})
export class AppComponent {

  users: any[] = [
    {name: 'Sam', age: 45},
    {name: 'Jim', age: 33},
    {name: 'Ana', age: 17},
    {name: 'Lou', age: 4},
  ]


}
```

Les variables locales sont présentes dans `ngFor` . Nous déclarons des nouvelles variables pour utiliser ces fameuses variables locales. Nous mixons notre boucle avec d'autres directives. Ici, lorsque l'index est pair alors l'élément est coloré en rouge.

Tester

Répéter un affichage avec ngFor - Partie 2

Utiliser trackBy pour améliorer la performance

Nous allons utiliser un exemple pour se rendre compte de l'utilité de `trackBy` .

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

@Component({
  selector: 'app',
  template: `
    <p *ngFor="let user of users">
      <user>
        {{user.name}} : {{user.age}}
      </user>
    </p>
    <button (click)="add()">Ajouter</button>
  `
})
export class AppComponent {

  users: any[] = [
    {name: 'Sam', age: 45, id: 1},
    {name: 'Jim', age: 33, id: 2},
    {name: 'Ana', age: 17, id: 3},
    {name: 'Lou', age: 4,  id: 4},
  ];

  add() {
    let newIndex = this.users.length+1;
    // Create immutable object
    this.users = this.users.map((obj) => {
      return Object.assign({}, obj);
    })
    this.users.push({name: `Test${newIndex}`, age: 15, id: newIndex});
  }
```

```
}
```

Notre composant permet d'afficher les utilisateurs. Rien de nouveau. Nous avons ajouté un bouton ajoutant un nouvel utilisateur.

Le tableau `users` a été cloné (en créant un objet `immutable`). Le tableau n'étant plus le même que l'initial, Angular va supprimer tous les éléments dans le DOM et les récréer d'une manière itérative. Pour se rendre compte, nous avons créer un composant enfant :

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

@Component({
  selector: 'user',
  template: `
    <ng-content></ng-content>
  `
})
export class UserComponent {

  ngOnInit() {
    console.log('Utilisateur créé')
  }
  ngOnDestroy() {
    console.log('Utilisateur supprimé')
  }

}
```

Effectivement, qiand nous ajoutons un utilisateur, les logs sont appelés plusieurs fois : 4 suppressions et 5 créations ensuite. Imaginez la même chose mais avec 10000 utilisateurs...

Tester : ngFor sans trackBy

Pour éviter cela, nous utilisons `trackBy` qui mémorise les items selon une propriété unique. Très généralement, cette propriété est l'ID. Voici comment nous l'utilisons :

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

@Component({
  selector: 'app',
  template: `
    <p *ngFor="let user of users ; trackBy: trackById">
      <user>
        {{user.name}} : {{user.age}}
      </user>
    </p>
    <button (click)="add()">Ajouter</button>
  `
})
export class AppComponent {

  users: any[] = [
    {name: 'Sam', age: 45, id: 1},
    {name: 'Jim', age: 33, id: 2},
    {name: 'Ana', age: 17, id: 3},
    {name: 'Lou', age: 4,  id: 4},
  ];

  add() {
    let newIndex = this.users.length+1;
    // Create immutable object
    this.users = this.users.map((obj) => {
      return Object.assign({}, obj);
    })
```

```
    this.users.push({name: `Test${newIndex}`, age: 15, id: newIndex});
  }

  trackById(index: number, obj: any): number {
    return obj.id;
  }


}
```

Nous ajoutons donc `trackBy` dans `ngFor` avec la méthode à appeler (ici, `trackById`). Cette dernière envoie l'identifiant de l'item.

Si vous testez et regardez les logs, vous remarquez que nous avons qu'une création lors d'un ajout d'un utilisateur. Bien mieux !

Tester : ngFor avec trackBy

Combiner ngIf et ngFor

Cycle de vie

Des hooks sont présents sur un composant. Cela permet de déclencher une fonction selon l'état du composant. Soit :

- `ngOnChanges` - appelé quand une valeur dans un champ change
- `ngOnInit` - appelé lors de l'initialisation du composant
- `ngDoCheck` - appelé à chaque changement détecté
  - `ngAfterContentInit` - après l'initialisation des dans les composants
  - `ngAfterContentChecked` - après les changements des dans les composants
  - `ngAfterViewInit` - après l'initialisation des vues des composants et composants enfants
  - `ngAfterViewChecked` - après les changements des vues des composants et composants enfants
- `ngOnDestroy` - Quand le composant est détruit

Exemple :

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

@Component({
  selector: 'count',
  template: `
    <p>{{n}}</p>
    <button (click)="up()"></button>
  `
})
export class CountComponent implements OnInit {

  n: number;

  ngOnInit() {
    console.log('init')
  }

  up() {
    this.n++;
  }

}
```

Accéder à composant enfant

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

@Component({
  selector: 'my-child-view',
  template: '<button (click)="up()">Up child</button>'
})
export class ChildViewComponent {
  n: number = 0;
  up() {
    this.n++;
  }
}


@Component({
  selector: 'count',
  template: `
    <p>{{n}}</p>
    <button (click)="up()">Up</button>
```

```
    <my-child-view></my-child-view>
  `
})
export class CountComponent implements OnInit {

  @ViewChild(ChildViewComponent) viewChild: ChildViewComponent;

  n: number = 0;

  ngOnInit() {
    console.log('init')
  }

  up() {
    this.n = this.viewChild.n + 2;
  }

}
```

Tester

https://embed.plnkr.co/qOyZ9zFRfw3W54KcKfGj/

Utilisation des Pipes

Un Pipe est un filtre en AngularJS 1. Le but est récupérer une donnée en entrée et de la transformer en sortie.

Utilisation basique des Pipes : changer un nombre en une devise monétaire

Nous utilisons le Pipe dans le template :

```
{{variable | nom_du_pipe}}
```

Voici un exemple :

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

@Component({
  selector: 'app',
  template: `<p>Prix : {{ price | currency }}</p>`
})
export class AppComponent {
  price: number = 15.2;
}
```

Cela va afficher `USD15.20` ;

Tester : Mettre en devise

Envoyer des paramètres

Comme vous pouvez le constater, ça envoie le nombre préfixé de `USD` . Nous pouvons changer cette information en envoyant une donnée en paramètre du Pipe :

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

@Component({
  selector: 'app',
  template: `<p>Prix : {{ price | currency:'EUR' }}</p>`
})
export class AppComponent {
  price: number = 15.2;
}
```

Soit le format :

```
nom_du_pipe: paramètre
```

Nous pouvons ajouter plusieurs paramètres sous le format :

```
nom_du_pipe: paramètre_1 : paramètre_2 : paramètre_3 ...etc
```

D'après la documentation de CurrencyPipe, nous voyons qu'il possible de mettre 3 parmètres :

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

@Component({
  selector: 'app',
  template: `
```

```
  <p>
    Prix 1 : {{ price | currency:'EUR':true:'1.2-4' }} <br>
    Prix 2 : {{ price | currency:'EUR':false }} <br>
    Prix 3 : {{ price2 | currency:'EUR':false:'1.2-2' }}
  </p>`
})
export class AppComponent {
  price: number = 15.23789;
  price2: number = 3;
}
```

Les paramètres :

1. Le nom de la devise
2. Si on doit afficher le symbole € ou $
3. Reprise du Pipe `DecimalPipe` (voir prochain chapitre)

Nous avons donc comme résultat :

```
Prix 1 : €15.2379
Prix 2 : EUR15.24
Prix 3 : EUR3.00
```

Tester

Tester : Mettre en devise avec paramètres

Utiliser le pipe Decimal

La documentation de DecimalPipe nous donne un exemple de ce pipe :

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

@Component({
  selector: 'app',
  template: `
  <div>
    <p>e (no formatting): {{e}}</p>
    <p>e (3.1-5): {{e | number:'3.1-5'}}</p>
    <p>pi (no formatting): {{pi}}</p>
    <p>pi (3.5-5): {{pi | number:'3.5-5'}}</p>
  </div>`
})
export class AppComponent {
   pi: number = 3.141592;
   e: number = 2.718281828459045;
}
```

L'idée est d'arrondir un nombre ou d'afficher des 0 en plus avant ou après la virgule. Ce n'est pas difficile mais il faut se familiariser avec le format :

```
minChiffre.minFraction-maxFraction
```

- `minChiffre` désigne le nombre de chiffres minimum avant la virgule.
- `minFraction` désigne le nombre de chiffres minimum après la virgule.
- `maxFraction` désigne le nombre de chiffres maximum après la virgule.

Par exemple, si on applique `3.1-5` sur PI :

- `3` : il faut au moins 3 chiffres avant la virgule soit : `003.141592`
- `1` : il faut au moins 1 chiffre après la virgule soit `003.141592` (ça ne change pas)
- `5` : il faut maximum 5 chiffres après la virgule soit `003.14159`. Le dernier chiffre (`2`) est enlevé et le nombre final est arrondi.

Donc `{{pi | number:'3.1-5'}}` vaut `003.14159`

Tester

https://plnkr.co/edit/4hYIwzO0Dx21x84dD2v6

Créer des Pipes personnalisés

Nous souhaitons faire ceci :

```
{{'Sam' | author}}
```

renvoie :

```
Ecrit par Sam
```

On pourrait bien sûr se passer du Pipe. L'idée est de montrer comment créer un Pipe personnalisé simplement. Nous créons un fichier pour notre Pipe nommé : `author.pipe.ts` :

app app.component.ts app.module.ts author.pipe.ts main.ts index.html

Le code :

```
import {Pipe, PipeTransform} from '@angular/core';

@Pipe({
  name: 'author'
})
export class AuthorPipe implements PipeTransform {

  transform(name: string): string {
    return `Ecrit par ${name}`;
  }

}
```

1. Nous importons le décorateur `Pipe` et l'interface `PipeTransform` .
2. Dans le décorateur, nous donnons le nom du Pipe
3. Dans notre classe, nous utilisons la méthode `transform` . Le premier paramètre de la méthode est la donnée en entrée. La méthode retourne la donnée transformée.

Dans le module, n'oublions pas d'importer le Pipe :

```
import {BrowserModule} from '@angular/platform-browser';
import {NgModule} from '@angular/core';
import {AppComponent} from './app.component';
import {AuthorPipe} from './author.pipe';

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

}
```

Que ça soit un module racine ou fonctionnel, le Pipe est envoyé dans le tableau `declarations`

Avec des paramètres

Si nous avons d'autres paramètres :

```
{'Sam' | author:'Admin'}}
```

Nous ajoutons alors des paramètres supplémentaires dans méthode `transform` :

```
import {Pipe, PipeTransform} from '@angular/core';

@Pipe({
  name: 'author'
})
export class AuthorPipe implements PipeTransform {

  transform(name: string, label:string): string {
    return `Ecrit par ${name} - ${label}`;
  }

}
```

Tester

Créer des Pipes personnalisés

Les formulaires

Avant tout

Pour utiliser les formulaires, vous devez importer les modules :

- `FormsModule`
- `ReactiveFormsModule` pour avoir les directives `formControl` et `formGroup`

Importez ces modules dans `imports` :

```
import {BrowserModule} from '@angular/platform-browser';
import {NgModule} from '@angular/core';
import {AppComponent} from './app.component';
import {ReactiveFormsModule, FormsModule} from '@angular/forms';

@NgModule({
  imports: [BrowserModule, FormsModule, ReactiveFormsModule],
  declarations: [AppComponent],
  bootstrap: [AppComponent]
})
export class AppModule {

}
```

Formulaire HTML

Dans le template, il suffit d'utiliser la forme HTML5 :

```
  <label for="name">Nom</label>
  <input type="text" name="username" id="name">
```

Le plus important est de mettre une valeur à l'attribut `name`. Nous pourrons controler le formulaire et ses champs grâce à cette attribut.

Créer un simple formulaire

Voici le code d'un composant ayant un formulaire :

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

@Component({
  selector: 'app',
  template: `
  <form #form="ngForm" (ngSubmit)="submit(form)">
    <label for="username">User name
      <input type="text" name="username" id="username" ngModel>
    </label>
     <button type="submit">Ok</button>
  </form>
  Result :
  {{result}}
  `
})
export class AppComponent {

  result: string;

  submit (form: NgForm) {

    this.result = form.value.username;
    // ou
    // this.result = form.controls['username'].value;
  }
}
```

Nous utilisons la directive `ngForm` sur le formulaire. Remarquez que nous créons une variable locale `form` pour accéder aux propriétés de `ngForm` dans le template. Ainsi, lorsque le formulaire est soumis, l'évènement `ngSubmit` est exécuté et appelle la méthode `submit` de notre composant avec la variable locale `form` en paramètre.

Dans la méthode `submit`, nous pouvons récupérer la valeur du champ `username`. Pour cela, ce dernier doit avoir la directive `ngModel`.

Tester

Créer un simple formulaire

Utiliser ngModel

Reprenons le code précédent mais complétons la directive `ngModel` :

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

@Component({
  selector: 'app',
  template: `
  <form #form="ngForm" (ngSubmit)="submit(form)">
    <label for="username">User name
      <input type="text" name="username" id="username" [(ngModel)]="propUsername">
    </label>
     <button type="submit">Ok</button>
  </form>
  <p>Data-Binding : {{propUsername}}</p>
  <p>Result :  {{result}}</p>
  `
})
export class AppComponent {

  result: string;
  propUsername: string;

  submit (form: NgForm) {

    this.result = form.value.username;

  }
}
```

Le but est de récupérer la saisie de l'utilisateur dans le champ (l'entrée) et modifier la variable `propUsername` dans le composant (la sortie). Nous faisons donc du `two-way data-binding`. Rappellez vous la syntaxe : `[(ngModel)]`

Ainsi, lorsque l'utilisateur rentre une information, la donnée s'affiche aussi dans le composant

Tester

Utiliser ngModel

Regrouper des champs d'un formulaire ngModelGroup

Vous remarquez cependant que nous avons une propriété pour chaque champ. Mais quelque fois, nous avons un objet. Par exemple, les données de l'utilisateur récupérées sur le serveur :

```
{first: 'Nancy', last: 'Drew'}
```

La directive `ngModelGroup` permet de regrouper des champs dans le formulaire :

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

@Component({
  selector: 'app',
  template: `
  <form #form="ngForm" (ngSubmit)="submit(form)">
      <div ngModelGroup="name">
        <input name="first" [(ngModel)]="name.first">
```

```
        <input name="last" [(ngModel)]="name.last">
      </div>
</form>
  `
})
export class AppComponent {

  result: string;
  name = {first: 'Nancy', last: 'Drew'};

  submit (form: NgForm) {


  }
}
```

Nous donnons l'objet `name` à `ngModelGroup` et appliquons les propriétés de l'objet aux différents champs. Nous sommes pas obligé d'utiliser `ngModelGroup` quand nous avons un objet. L'idée de pouvoir vérifier que ce groupe de champs est valide indépendamment du reste du formulaire

```
<form #form="ngForm" (ngSubmit)="submit(form)">
      <div ngModelGroup="name" #nameCtrl="ngModelGroup">
        <input name="first" [(ngModel)]="name.first" minlength="2">
        <input name="last" [(ngModel)]="name.last" required>
      </div>
      Valide ? : {{nameCtrl.valid}}
</form>
```

Il suffit de créer une variable locale `nameCtrl` et de tester sa propriété `valid` .

Le chapitre suivant expliquera plus pleinement la notion de validation.

Tester

Regrouper des champs d'un formulaire ngModelGroup

Exercice

Créez un formulaire où :

1. L'objet suivant est envoyé dès le départ dans le formulaire

   {country: 'France', city: 'Lyon'}

2. Le titre du formulaire (dans un élément `<h1>`) affiche la propriété `city` de l'objet
3. Lorsque l'utilisateur modifie le champ `city` , il ne faut pas que le titre soit modifié. C'est seulement lorsque le formulaire est validé que le titre est changé. Conseil : pensez à la syntaxe du data-binding.
4. Lorsque le formulaire est soumis, une `<div>` s'affiche avec un texte `Modification effectuée`

Construire le formulaire

Pour construire le formulaire en Javascript, nous avons besoin d'un module supplémentaire : `ReactiveFormsModule`. Cela ajoute des directives et providers :

- `FormBuilder`
- `FormGroup`
- `FormControl`

Voici le code de notre composant :

```
import {Component} from '@angular/core';
import {
  FormBuilder,
  FormControl,
  FormGroup
} from '@angular/forms';


@Component({
  selector: 'app',
  template: `
  <form [formGroup]="form" (ngSubmit)="submit()">
    <label for="username">User name
      <input type="text" name="username" [formControl]="username">
    </label>
     <button type="submit">Ok</button>
  </form>

  `
})
export class AppComponent {

  form: FormGroup;
  username: FormControl;

  constructor (private builder: FormBuilder) {

  }

  ngOnInit() {
    this.username = new FormControl('', []);
    this.form = this.builder.group({
      username: this.username
    });
  }

  submit () {
    console.log(this.form.value);
  }
}
```

Le service `FormBuilder` permet de constuire un formulaire et définir un groupe avec la méthode `group()`. Ce dernier possède un paramètre : un objet définissant les champs présent dans notre formulaire

Dans le template, nous définissons le formulaire créé avec la directive `formGroup`. Chaque champ intègre la directive `formControl` avec la clé du champ définit dans l'objet.

Le service `FormControl` permet de mettre un contrôle sur un champ. Deux paramètres sont présents :

1. La valeut initiale

2. Un tableau de validateurs

Tester

Construire le formulaire

Mettre des validateurs

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


@Component({
  selector: 'app',
  template: `
  <form [formGroup]="form" (ngSubmit)="submit()">
    <label for="username">User name
      <input type="text" name="username" [formControl]="username">
      <div [hidden]="username.untouched">
        <div [hidden]="!username.hasError('required')">Champ requise</div>
      </div>
    </label>
     <button type="submit">Ok</button>
  </form>

  `
})
export class AppComponent {

  form: FormGroup;
  username: FormControl;

  constructor (private builder: FormBuilder) {

  }

  ngOnInit() {
    this.username = new FormControl('', [
        Validators.required
    ]);
    this.form = this.builder.group({
      username: this.username
    });
  }

  submit () {
    console.log(this.form.value);
  }
}
```

Voici les étapes ajoutées :

1. Le service `Validators` en plus
2. Nous déclarons les validateurs dans le tableau de l'objet `FormControl`
3. Nous pouvons utiliser des méthodes de `FormControl` dans le template comme `hasError()`

Tester

Mettre des validateurs

Créer ses propres validateurs

Créer un validateur, c'est juste une fonction renvoyant :

- null, si aucune erreur
- un objet d'erreurs

Exemple :

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

export function beginLetter(input: FormControl): ValidatorFn {
  return /^S/.test(input.value) ? null : {letter: true};
}


@Component({
  selector: 'app',
  template: `
  <form [formGroup]="form" (ngSubmit)="submit()">
    <label for="username">User name
      <input type="text" name="username" [formControl]="username">
      <div [hidden]="username.untouched">
        <div [hidden]="!username.hasError('required')">Champ requise</div>
        <div [hidden]="!username.hasError('letter')">First Letter is not S</div>
      </div>
    </label>
     <button type="submit">Ok</button>
  </form>

  `
})
export class AppComponent {

  form: FormGroup;
  username: FormControl;

  constructor (private builder: FormBuilder) {

  }

  ngOnInit() {
    this.username = new FormControl('', [
        Validators.required,
        beginLetter
    ]);
    this.form = this.builder.group({
      username: this.username
    });
  }

  submit () {
    console.log(this.form.value);
  }
}
```

Le validateur vérifie si le nom d'utilisateur commence par un `S`. Si ce n'est pas le cas, on envoie un objet `{letter: true}` indiquant une erreur se nommant `letter`. Du coup, nous pouvons l'utiliser dans le template avec `username.hasError('letter')`

Tester

Créer ses propres validateurs

Avec des paramètres

Il suffit de créer une fonction renvoyant la fonction de validation :

```
export function beginLetter(letter) {

  return function(input: FormControl) {
    let reg = new RegExp(`^${letter}`)
    return reg.test(input.value) ? null : {letter: true};
  }

}
```

Nous pouvons ajouter la fonction dans le tableau de validateurs :

```
[...]
  ngOnInit() {
    this.firstLetter = 'U';
    this.username = new FormControl('', [
        Validators.required,
        beginLetter(this.firstLetter)
    ]);
    this.form = this.builder.group({
      username: this.username
    });
  }
[...]
```

Tester

Créer ses propres validateurs avec des paramètres

Injection des dépendances

Le Design Pattern d'injection de dépendances

Créer un service

Avant de parler des injections de dépendances, nous allons créer un service :

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

@Injectable()
export class MyService {

  getTitle() {
    return "Formation AngularJS2";
  }

}
```

Oui, un service est tout simplement une classe avec le décorateur `Injectable`

Problèmes :

1. Comment nous l'avons vu avant, nous pouvons pas utiliser `new` car si le constructeur change, il faut changer le changer partout.
2. Nous créons à chaque fois une instance avec `new`

La solution : L'injection de dépendance (DI).

Injection de dépendance :

3 éléments : Injector : l’objet qui nous fournit les API permettant de créer les instances des dépendances ; Provider : une sorte de recette qui dit à l’injector comment créer une instance de dépendance donnée ; Dépendance : un type dont une nouvelle instance (nouvel objet) doit être créée.

Lorsqu'on nous l'utilisons dans une autre classe, nous utilisons le décorateur `@Inject` . Cela signifie que nous injections la classe créée dans une autre classe pour pouvoir l'utiliser :

```
import {Component, Inject} from '@angular/core';
import {MyService} from './app.service.ts';

@Component({
  selector: 'app',
  template: '<h1>{{title}}</h1>'
})
export class AppComponent {

  title: string;

  constructor(@Inject(MyService) myservice) {
      this.title = myservice.getTitle();
  }
}
```

Et en privilégiant le hook `ngOnInit` avec l'injection raccourci de TypeScript :

```
import {Component, Inject, OnInit} from '@angular/core';
import {MyService} from './app.service.ts';
```

```
@Component({
  selector: 'app',
  template: '<h1>{{title}}</h1>'
})
export class AppComponent implements OnInit {

  constructor(private myservice:MyService) {

  }

  ngOnInit() {
    this.title = this.myservice.getTitle();
  }

}
```

Remarque sur private

```
class AppComponent {
  constructor(private myservice:MyService) {

  }
}
```

Revient à faire :

```
class AppComponent {
  myservice:MyService

  constructor(myservice:MyService) {
    this.myservice = myservice;
  }
}
```

Le provider

Bien entendu, cela n'est pas suffisant. Ici, nous n'utilisons pas le mot clé `new`. Normal, puisque nous utilisons le design pattern DI. C'est le provider qui se charge de créer les instances. Il faut donc les déclarer dans le module :

```
import {BrowserModule} from '@angular/platform-browser';
import {NgModule} from '@angular/core';
import {AppComponent} from './app.component';
import {MyService} from './app.service.ts';

@NgModule({
  imports: [BrowserModule],
  declarations: [AppComponent],
  bootstrap: [AppComponent],
  providers: [MyService]
})
export class AppModule {

}
```

C'est donc avec la propriété `providers` que nous déclarons les services à instancier.

Tester

Le Design Pattern d'injection de dépendances

La programmation réactive

Une promesse.

Les promesses sont intégrées nativement dans Javascript et permettent de gérer des appels asynchrones. Prenons un exemple.

1. Vous avez besoin de récupérer des données sur un utilisateur.
2. Vous récupérez ensuite les messages de l'utilisateur

L'action n°2 est dépendante de l'action n°1. Pourquoi ? Car dans la deuxième requête HTTP, nous devons envoyer sûrement une information récupéré (comme un token) de la première requête.

Comment programmer cela sans les promises ?

```
http.get(`users/${userId}`, function(user) {
  http.get(`posts/${user.token}`, function(posts) {
      console.log('Posts', posts)
  });
});
```

Avez vous remarquez le problème ? L'enfer des callbacks et un code compliqué à maintenir. Admettons que nous voulons créer des services, notre application serait difficilemet modulable avec tous ces callbacks. Les promesses rentrent en jeu et apporte un autre concept :

1. Une promesse se lance directement
2. La promesse doit ne résoudre pour effectuer une autre promesse
3. La promesse peut capturer une erreur

D'après le point 2, nous pouvez comprendre que les promesses peuvent s'enchainer. Comment ? Voyons un code utilisant nativement les promesses :

```
var p = new Promise((resolv, reject) => {
  setTimeout(resolv, 1000);
});
```

La promesse s'éxécute mais nous indiquons à quel moment elle doit se résoudre. Ici, au bout d'une seconde. L'interêt ?

```
var p = new Promise((resolv, reject) => {
  setTimeout(resolv, 1000);
}).then(() => {
  return "hello"
});

p.then((str) => {
  console.log(str);
});
```

D'enchainer les promesses ! Le premier `then` sera déclenché au bout d'une seconde. Nous pouvons soit renvoyer une promesse, soit une valeur.

Remarquez que `p` est une promesse. Nous pouvons détacher notre promesse. Très utile si nous souhaitons créer des modules indépendants renvoyant des promesses.

Et si les requêtes http renvoyaient des promesses ?

```
http.get(`users/${userId}`).then((user) => {
  return http.get(`posts/${user.token}`);
}).then((posts) => {
  console.log('Posts', posts);
```

```
})
```

N'est ce pas mieux ? Si nous créons une méthode dans une classe, nous pourrions faire :

```
class User {
  getPosts(userId) {
    return http.get(`users/${userId}`).then((user) => {
      return http.get(`posts/${user.token}`);
    });
  }
}
```

Puisqu'il renvoie une promesse, nous pouvons les chainer avec d'autres promesses de l'application. Très pratique ! Et surtout, bien plus maintenable et modulable !

Les erreurs

Une promesse peut renvoyer une erreur. Par exemple, une requête HTTP n'aboutissant pas. Comment ?

```
var p = new Promise((resolv, reject) => {
   requestAsynchrone((err) => {
     if (err){
       reject(err);
     }
     else {
       resolv();
     }
   })

});
p.then(() => {

}).catch((err) => {
  console.log(err)
});
```

Si nous avons une requête asynchrone (ici, une fausse fonction nommée `requestAsynchrone`) qui n'aboutit pas elle renvoie une erreur. Nous utilisons donc la fonction `reject` qui rejecte la promesse. Une exception est levée et la méthode `catch` est exécutée. ça fonctionne donc comme les exceptions avec `try {} catch`

Dans Angular2

Flux observé

Un Observable est un flux d'événement observé. D'une manière simple, le flux peut être un tableaut :

```
    [1, 2, 3]
```

Nous avons l'événement 1, 2 et 3. Sur ce tableau, nous avons parfois le besoin de filtrer ou de transformer l'information :

```
[1, 2, 3].filter((x) => x > 1) // [2, 3]
[1, 2, 3].map((x) => x * 2) // [2, 4, 6]
```

Un flux d'événement, c'est quasi-identique sauf c'est un processus asynchrone. Nous avons des valeurs dans le temps. L'interêt est de pouvoir observer ce flux, voir comment il évolue dans le temps et effectuer des opérations dessus. Ces opérations ne vont pas transformer le flux, mais en créer un nouveau.

Un Observable comporte 3 états : dans 3 callbacks appelés :

1. Une valeur renvoyée pour chaque événement
2. Une erreur envoyée
3. Indication que le flux est terminé

Les différence avec les promises ?

1. Une promesse s'enchaine et c'est tout. Pour les Observables, comme nous pouvons le voir, nous pouvons ajouter des aspects de démontage sur le flux
2. Une promesse se déclenche directement et s'éxécute jusqu'à la fin. L'Observable est `lazy`, c'est à paresseux. Il faut souscrire au flux pour le lancer. Enfin il peut être arrété à tout moment.
3. Un Observable peut être relancé avec `retry` alors qu'une promesse demande de relancer la fonction d'origine

En pratique

- Avec les promises : http://embed.plnkr.co/8ap1Lm/
- Avec les Observables : http://embed.plnkr.co/SIltBL/

Dans Angular2

Angular2 utilise RxJS pour les Observables. Voici notre service :

```
import {Injectable} from '@angular/core';
import {Observable} from 'rxjs/Observable';
import 'rxjs/add/operator/filter';

@Injectable();
export class PostService {

  getPosts() {
    return new Observable(observer => {
          setTimeout(() => {
              observer.next(4);
          }, 1000);

          setTimeout(() => {
              observer.next(40);
          }, 2000);
```

```
          setTimeout(() => {
              observer.complete();
          }, 3000);
      }).filter(x => x > 10);

  }

}
```

Ici, nous créons un Observable. Chaque seconde, il envoie un événement avec une valeur (`4` et ensuite `40`). Au bout de 3 secondes, nous indiquons que le flux est terminé. La méthode `filter` va recréer un flux mais seulement avec les événements où la valeur est supérieur à `10`

Nous notre composant, nous pouvons récupérer le service et souscrire à Observable :

```
import {Component, Inject} from '@angular/core';
import {PostService} from './post.service';

@Component({
  selector: 'app',
  template: '<h1>{{title}}</h1>'
})
export class AppComponent {

  title: string;

  constructor(@Inject(PostService) post) {
      post.getPosts().subscribe(
          value => console.log(value),
          error => console.log(error),
          () => console.log('finish')
      );
  }
}
```

Nous appelons la méthode `getPosts` et appliquons les 3 callbacks. Réutilisons ce composant avec une requête HTTP. Le prochain chapitre l'expliquera.

> PostService a été déclarée préablement dans le module et ajouté au provider.

Tester

Flux observé

Requête HTTP

Avec le serveur, nous devons récupérer ou envoyer des données. Angular2 intègre le service Http pour communiquer avec le serveur avec l'architecture REST.

Requête HTTP

Avant tout

Nous devons charger le module `HttpModule` dans notre module :

```
import {BrowserModule} from '@angular/platform-browser';
import {NgModule} from '@angular/core';
import {HttpModule} from '@angular/http';
import {AppComponent} from './app.component';
import {PostService} from './post.service';

@NgModule({
  imports: [BrowserModule, HttpModule],
  declarations: [AppComponent],
  bootstrap: [AppComponent],
  providers: [PostService]
})
export class AppModule {

}
```

Service de messages

Voici un service récupérant des messages sur un serveur :

```
import {Injectable, Inject} from '@angular/core';
import {Http, Response} from '@angular/http';
import 'rxjs/Rx';

@Injectable();
export class PostService {

  url: string = 'https://jsonplaceholder.typicode.com/posts';

  constructor(@Inject(Http) private http) {

  }

  getPosts() {
    return this.http
               .get(this.url)
               .map((r: Response) => r.json());
  }

}
```

1. Nous importons le service `Http` que nous injectons dans notre service `PostService` ;
2. Nous importons les méthodes de `rxjs` (pour utiliser `map`)
3. Nous définissons l'URL

La méthode `getPosts` va récupérer tous les messages. Le service `http` utilise les verbe de l'architecture REST :

- get
- put
- post
- delete

- patch

La requête retourne un objet Observable permettant d'observer le flux. Ainsi, nous transformons la réponse en JSON.

Le composant

Dans notre composant, nous utilisons le service pour afficher une liste de messages :

```
import {Component, Inject} from '@angular/core';
import {PostService} from './post.service';

@Component({
  selector: 'app',
  template: `
    <ul>
      <li *ngFor="let post of posts">{{post.title}}</li>
    </ul>
  `
})
export class AppComponent implements OnInit {


  constructor(@Inject(PostService) private post) {

  }

  ngOnInit() {
    this.post.getPosts().subscribe((posts) => {
      this.posts = posts;
    });
  }

}
```

Avec `ngOnInit`, nous appelons directement la requête lorsque le composant est initialisé. Rappelez vous que `getPosts` renvoie un Observable. Cela signifie qu'il faut y souscrire avec `subscribe` pour lancer le flux et l'observer. Le premier paramètre récupérer la donnée de l'événement du flux (les messages) et les affiche dans le template (merci le data-binding)

Tester : Requête HTTP

Créer, mettre à jour et supprimer une donnée

Créer avec la méthode POST

```
import {Injectable, Inject} from '@angular/core';
import {Http, Headers, RequestOptions, Response} from '@angular/http';
import 'rxjs/Rx';

@Injectable();
export class PostService {

  url: string = 'https://jsonplaceholder.typicode.com/posts';
  headers: Headers = new Headers({ 'Content-Type': 'application/json' });

  constructor(@Inject(Http) private http) {

  }

  createPost(obj) {
    let body = JSON.stringify(obj);
    let options = new RequestOptions({ headers: this.headers });

    return this.http.post(this.url, body, options)
                    .map((r: Response) => r.json());
  }

}
```

Deux services nous seront utiles pour envoyer des données :

1. `Headers` pour gérer les entêtes HTTP
2. `RequestOptions` pour définir les options de la requêtes.

Nous envoyons les données en méthode `POST` au serveur.

Tester

Routage

Pourquoi le routage ? Le routage permet de diriger l'utilisateur vers une vue en particulier selon l'URL. L'interêt ?

1. De récupérer des identifiants dans l'URL (ID d'un message, d'une personne, etc) pour le CRUD
2. L'URL peut être partagé. L'utilisateur arrive sur la vue directement sans passer par la home.
3. L'utilisateur peut revenir en arrière avec le bouton précédent
4. Plus esthétique que rien du tout :)

Créer un simple routeur

Dans le module, nous définissons les chemins pour les composants :

```
import { RouterModule, Routes } from '@angular/router';

import {OtherComponent} from './other.component';
import {MainComponent} from './main.component';

const routes: Routes = [
  { path: '', redirectTo: 'main', pathMatch: 'full' },
  { path: 'main', component: MainComponent },
  { path: 'other', component: OtherComponent }
];

export const routing = RouterModule.forRoot(routes);
```

On admet que nous avons 2 composants : `MainComponent` et `OtherComponent` . Nous créons donc un tableau définissant les différentes routes.

Lorsque le chemin entier de l'URL ne possède pas de nom, alors on redirige l'utilisateur vers `main` .

Nous exportons le module `RouterModule` configuré avec nos routes.

Chargeons le module routing

Nous ajoutons le module configuré dans le tableau `imports` :

```
import {BrowserModule} from '@angular/platform-browser';
import {routing} from './app.routes';
import {NgModule} from '@angular/core';

import {AppComponent} from './app.component';
import {OtherComponent} from './other.component';
import {MainComponent} from './main.component';

@NgModule({
  imports: [BrowserModule, routing],
  declarations: [AppComponent, OtherComponent, MainComponent],
  bootstrap: [AppComponent]
})
export class AppModule {

}
```

Utiliser les directives pour naviguer

Le composant parent utilise des directives pour naviguer avec le routeur :

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

@Component({
  selector: 'app',
  template: `
    <ul>
      <li><a [routerLink]="['/main']">main</a></li>
      <li><a [routerLink]="['/other']">other</a></li>
```

```
    </ul>
    <router-outlet></router-outlet>
  `
})
export class AppComponent {
}
```

La directive `routerLink` indique le chemin du composant. Ce dernier est inséré dans l'élément `<router-outlet>` dynamiquement

Tester

Créer un simple routeur

Utiliser des paramètres

configuration du routeur

Dans la configuration du routeur, nous ajoutons où se trouve les paramètres dans l'URL :

```
import { RouterModule, Routes } from '@angular/router';

import {OtherComponent} from './other.component';
import {MainComponent} from './main.component';

const routes: Routes = [
  { path: '', redirectTo: 'main', pathMatch: 'full' },
  { path: 'main', component: MainComponent },
  { path: 'other/:id', component: OtherComponent }
];

export const routing = RouterModule.forRoot(routes);
```

`:id` indique que nous pouvons avoir un identifiant.

Naviguer avec un paramètre

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

@Component({
  selector: 'app',
  template: `
    <ul>
      <li><a [routerLink]="['/main']">main</a></li>
      <li><a [routerLink]="['/other', 1337]">other</a></li>
    </ul>
    <router-outlet></router-outlet>
  `
})
export class AppComponent {

}
```

Dans le template, nous ajoutons le paramètre (ici, l'id est `1337`)

Récupérer le paramètre dans le composant

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

@Component({
  selector: 'other',
  template: `
    Other with id {{id}}
    <button (click)="nav()">MainComponent</button>
  `
})
export class OtherComponent implements OnInit {
```

```
  id: number;

  constructor(private route: ActivatedRoute, public router: Router) {

  }

  ngOnInit() {
    this.route.params.subscribe(params => {
       this.id = +params.id;
    });
  }

  nav() {
    this.router.navigate(['/main']);
  }

}
```

Nous chargeons et initialisons le service `ActivatedRoute`. Nous récupérons le paramètre avec un Observable lorsque le composant est initialisé.

Angular utilise un Observable car le composant est construit (`constructor`) qu'une fois alors que le paramètre, lui, peut changer. Il faut donc observer l'événement avec un Observable.

Tester

Utiliser des paramètres

Des routes enfants

Appelé aussi "routes imbriqués", les routes enfants permettent de mettre une navigation dans une vue.

Par exemple :

```
http://localhost/users/1337/profile
```

Affichera un composant `Profile` pour le profil de l'utilisateur

```
http://localhost/users/1337/roles
```

Affichera un composant `Roles` pour les différents rôles de l'utilisateur Cela peut s'apparenter à un menu avec des boutons changeant une page.

Nous pouvons donc créer un fichier `user.router.ts` que nous plaçons dans le dossier `user`. Notre structure est la suivante :

```
<root>
  user
    user.component.ts
    user.module.ts
    user-profile.component.ts
    user-role.component.ts
    user.router.ts
  app.router.ts
  app.component.ts
  app.module.ts
```

Le fichier `user.router.ts` a ce code :

```
import {RouterModule, Routes} from '@angular/router';
import {UserComponent} from './user.component';
import {UserProfileComponent} from './user-profile.component';
import {UserRoleComponent} from './user-role.component';

const routes:Routes = [
  {
    path: 'users/:id',
    component:  UserComponent,
    children: [
          {
            path: 'profile',
            component: UserProfileComponent
          },
          {
            path: 'roles',
            component: UserRoleComponent
          }
      ]
  },
];

export const UserRouting = RouterModule.forChild(routes);
```

Le code est très ressemblant. Notez juste que nous avons la propriété `children` qui définit les différents composants pour le routeur enfant. Nous configurons le module avec `forChild()`

Ajouter au module

Comme pour le routeur parent, nous ajoutons aux modules racines (`app.module.ts`) :

```
import {BrowserModule} from '@angular/platform-browser';
import {NgModule} from '@angular/core';

import {UserModule} from './user/user.module';

import {AppComponent} from './app.component';

import {routing} from './app.router';
import {UserRouting} from './user/user.router';

@NgModule({
  imports: [BrowserModule, UserModule, routing, UserRouting],
  declarations: [AppComponent],
  bootstrap: [AppComponent]
})
export class AppModule {

}
```

Router-Outlet

Le premier `<router-outlet>` se trouve dans `AppComponent` comme d'habitude. Le deuxième se trouve `UserComponent` .

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

@Component({
  selector: 'other',
  template: `
    Utilisateur {{id}}
    <router-outlet></router-outlet>
  `
})
export class OtherComponent implements OnInit {

  id:number;

  constructor(private route: ActivatedRoute) {

  }

  ngOnInit() {
    this.route.params.subscribe(params => this.id = +params.id);
  }

}
```

Encore une chose à penser

Ici, nous avons un module nommé `user.module.ts` . Ce module intègre les 3 composants (`UserComponent` , `UserProfileComponent` et `UserRoleComponent`). Cependant, dans `UserComponent` ci-dessous, nous utilisons les directives du routeur. Pour que Angular puisse les utiliser, il faut importer le module routeur dans le module `UserModule` :

```
import {BrowserModule} from '@angular/platform-browser';
import {NgModule} from '@angular/core';
```

```
import {UserComponent} from './user.component';
import {UserProfileComponent} from './user-profile.component';
import {UserRoleComponent} from './user-role.component';

import {UserRouting} from './user/user.router';

@NgModule({
  imports: [BrowserModule, UserRouting],
  declarations: [UserComponent, UserProfileComponent, UserRoleComponent],
  exports: [UserComponent]
})
export class UserModule {

}
```

Installer les modules pour les tests unitaires

Pour les tests unitaires, nous avons besoin de plusieurs fichiers:

1. karma.conf.js : Configuration de Karma
2. karma-test-shim.js : Faire fonctionner Karma sur l'environnement Angular2

Ainsi que deux frameworks pour le tests :

1. Karma
2. Jasmine

Mettre en place l'environnement

Deux moyens pour mettre en place un environnement de test :

1. Avec Angular CLI (que nous verrons plus tard)
2. Récupérer les fichier sur le dépots github

Récupérer le dépot avec Git :

git clone https://github.com/angular/quickstart project cd project

Installer les dépendances et démarrer l'application:

npm install npm start

Pour effectuer des tests unitaires, nous allons installer karma :

npm install karma-cli -g

Simple test

Chaque composant possèdeun des test unitaires. Le fichier doit être nommé de la façon suivante : `<nom composant>.component.spec.ts`

Par exemple :

```
<project root>
  app
    app.component.ts
    app.component.spec.ts
    app.module.ts
  main.ts
  index.html
```

Voici un simple test de `app.component.spec.ts`

```
describe('Test calculs', () => {
  it('2+2 = 4 ?', () => {
    expect(2+2).toEqual(4);
  });
});
```

Nous utilisons Jasmine pour écrire le test. Le chapitre suivant expliquera plus amplement l'utilisation de Jasmine.

Lancez les tests unitaires avec :

npm test

Les fichiers TypeScripts vont être compilés en Javascript. Ensuite, Karma est lancé et lit les fichiers `.spec.js`. Une fenêtre Google Chrome se lance. Ignorez la. Elle permet juste exécuter le Javascript. C'est la console qui vous montre si les tests unitaires sont bien passés ou non.

Jasmine

La documentation se trouve sur http://jasmine.github.io. Cela permet de faire des tests BDD (=écrire les tests après le code de l'application).

Le site nous donne un code très simple :

```
describe("A suite is just a function", function() {
  var a;

  it("and so is a spec", function() {
    a = true;

    expect(a).toBe(true);
  });
});
```

Deux fonctions reviennent très souvent :

- `describe()`
- `it()`

Ces fonctions permettent d'organiser nos tests unitaires. Nous pouvons comparer `describe()` à une liste et `it()` à chaque point de la liste.

Prenons un cas concret, nous souhaitons tester le `login` :

Login

- Vérifier si le composant vérifie bien la dureté du mot de passe
- Vérifier si le composant vérifie bien que le champ utilisateur est bien remplie
- Etc.

Nous transformons cette liste en code :

```
describe("Login", function() {
  it("Vérifier si le composant vérifie bien la dureté du mot de passe", function() {

  });
  it("Vérifier si le composant vérifie bien que le champ utilisateur est bien remplie", function() {

  });
});
```

Tests avant chaque item ou suite

Un autre interêt des fonctions citées plus haut et de pourvoir faire des tests récurrents avant chaque item ou suite

> describe() est une suite it() est un item

beforeEach()

```
describe("Login", function() {

  beforeEach(function() {
```

```
    // code qui remet les champs du formulaire vides
  });

  it("Vérifier si le composant vérifie bien la dureté du mot de passe", function() {

  });
  it("Vérifier si le composant vérifie bien que le champ utilisateur est bien remplie", function() {

  });
});
```

La fonction `beforeEach()` intègre un callback qui exécuter au début de chaque item.

Si le code est asynchrone (une requête HTTP par exemple), le callback de `beforeEach()` contient callback à exécuter quand le code asynchrone est terminé :

```
describe("Login", function() {

  beforeEach(function(done) {
    setTimeout(() => {
      done()
    }, 3000);
  });

  // [...]
});
```

Ici, l'item est exécuté après un code asynchrone durant 3 secondes.

before()

De la même manière, nous pouvons exécuter un code avant chaque suite :

```
describe("Login", function() {

  beforeAll(function() {
    // Exécuté avant tous les tests
  });

  describe("Tests sur le mot de passe", function() {

    var password;

    beforeAll(function() {
      password = 'abcdef1';
      // Exécuté avant tous les tests sur le mot de passe
    });

    it("Vérifier si le mot de passe contient minimum 8 caractères", function() {

    });
    it("Vérifier si le mot de passe contient au moins un chiffre", function() {

    });
    // [...]

  });

  it("Vérifier si le composant vérifie bien que le champ utilisateur est bien remplie", function() {

  });

});
```

Pour l'asynchrone, le principe est le même que `beforeEach()` :

```
beforeAll(function(done) {

});
```

Tester

Pour le moment, nous avons juste indiquer les items et les suites. Dans chaque item, nous devons tester des valeurs. Comment ? Avec `expect()` .

Sur l'exemple au début du chapitre, nous avons :

```
var a = 1;
expect(a).toBe(true);
```

`expect()` récupère une valeur et le teste avec une autre valeur. Ici, nous vérifions si la valeur de `a` vaut bien `true` . Si c'est le cas, le test unitaire est validé.

Il existe plusieurs méthodes :

- Egalité : `expect(12).toEqual(12);`
- Expression régulière : `expect('barman').toMatch(/bar/);`
- Est défini : `expect(a).toBeDefined();`
- Vaut Null : `expect(null).toBeNull();`
- Plus grand que : `expect(2).toBeGreaterThan(1);`
- Etc

La négation peut être faite avec la proprité `not` :

```
expect(false).not.toBe(true);
expect('barman').not.toMatch(/bar/);
// etc.
```

Regardez la documentation pour d'autres méthodes.

Tester un composant

Admettons que notre composant soit le suivant :

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

@Component({
    selector: 'my-app',
    template: `<h1>{{message}}</h1>`
})
export class AppComponent {

  message: string = 'test';

}
```

Voici le code permettant de le tester :

```
// 1
import { AppComponent } from './app.component';
// --

// 2
import { async, ComponentFixture, TestBed } from '@angular/core/testing';
import { By }           from '@angular/platform-browser';
import { DebugElement } from '@angular/core';
// --

describe('AppComponent', function () {

  // 3
  let comp: AppComponent;
  let fixture: ComponentFixture<AppComponent>;
  // --

  // 4
  beforeEach(() => {
    TestBed.configureTestingModule({
       declarations: [ AppComponent ]
    });
    fixture = TestBed.createComponent(AppComponent);
    comp = fixture.componentInstance;
  });
  // --

  // 5
  it('Composant créé', () => expect(comp).toBeDefined() );

  it('Propriété', () => {
    expect(comp.message).toBe('test');
  });
  // --
});
```

Etapes sur la création du code :

1. Nous importons notre composant
2. Nous importons des classes qui nous serviront pour les tests
3. Nous déclarons 2 variables :
   - `comp` : une instance de notre composant
   - `fixture` : Accéder au composant

4. Nous créons un module test (plus tard, nous pourrons ajouter les providers).Nous créons le composant à partir du module test et une instance du composant
5. Nous vérifions que le composant existe (est défini) et que la propriété `message` du composant vaut bien `test`.

Si les templates et CSS sont externes

Si notre composants contient des templates et CSS externes :

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

@Component({
    selector: 'my-app',
    templateUrl: 'path/to/app.component.html',
    stylesUrl: ['path/to/app.component.css']
})
export class AppComponent {

  message: string = 'test';

}
```

L'étape 4 sera la suivante :

```
beforeEach(async(() => {
 TestBed.configureTestingModule({
    declarations: [ AppComponent ]
  })
  .compileComponents();
}));

beforeEach(() => {
  fixture = TestBed.createComponent(AppComponent);
  comp = fixture.componentInstance;
});
```

Nous ajoutons un `beforeEach()` avec le module test. La méthode `compileComponents` permet de compiler d'une manière asynchrone les fichiers CSS et HTML externes. Pensez à utiliser la méthode `async()` pour l'asynchrone.

Tester un élément

Maintenant, nous souhaîtons tester en récupérant un élément dans le DOM du composant. Le but est de vérifier que le titre vaut bien `test` sans passer par la propriété mais par élément.

```
import { AppComponent } from './app.component';

import { async, ComponentFixture, TestBed } from '@angular/core/testing';
import { By }           from '@angular/platform-browser';
import { DebugElement } from '@angular/core';

describe('AppComponent', function () {

  let comp: AppComponent;
  let fixture: ComponentFixture<AppComponent>;
  // 1
  let de: DebugElement;
  let el: HTMLElement;
  // --

  beforeEach(() => {
```

```
    TestBed.configureTestingModule({
       declarations: [ AppComponent ],
       providers: [
         { provide: ComponentFixtureAutoDetect, useValue: true }
       ]
    });
    fixture = TestBed.createComponent(AppComponent);
    comp = fixture.componentInstance;
  });

  it('Titre est `test`', () => {
    // 2
    de = fixture.debugElement.query(By.css('h1'));
    el = de.nativeElement;
    // --
    // 3
    fixture.detectChanges();
    // --
    // 4
    expect(el.textContent).toBe('test');
    // --
  });
});
```

1. Nous déclarons deux variables de type `DebugElement` et `HTMLElement`
2. Nous récupérons l'élément avec une requête `query()` . La requête s'effectue avec une expression CSS
3. Au début, le titre ne contient rien et `TestBed.createComponent` ne déclenche pas le changement. Nous faisons manuellement avec `detectChanges()`
4. Nous testons le contenu de l'élément. Renvoie `true` si le titre est bien `test`

> Nous pouvons demander à Angular de détecter les changements automatiquement avec `ComponentFixtureAutoDetect` :
>
> ```
> ```js
> import { async, ComponentFixture, ComponentFixtureAutoDetect, TestBed } from '@angular/core/testing';
> [...]
> TestBed.configureTestingModule({
>    declarations: [ AppComponent ],
>    providers: [
>      { provide: ComponentFixtureAutoDetect, useValue: true }
>    ]
> });
> ```
> ```

Vérifier le changement après un événement

Admettons que notre composant soit maintenant le suivant :

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

@Component({
    selector: 'my-app',
    template: `<h1>{{message}}</h1>

    <button (click)="modified()">Change</button>

    `
})
export class AppComponent {

  message: string = 'test';

  modified() {
```

```
    this.message = 'foo';
  }

}
```

Ici, le titre devient `foo` lorsque l'utilisateur clique sur le bouton. Nous devons faire ce test. Voici donc l'item que nous ajoutons dans notre fichier de tests :

```
beforeEach(() => {
  de = fixture.debugElement.query(By.css('h1'));
  el = de.nativeElement;
  fixture.detectChanges();
});

it('Titre est `foo` après le clique sur le bouton', () => {
  // 1
  let deButton = fixture.debugElement.query(By.css('button'));
  // 2
  deButton.triggerEventHandler('click', null);
  // 3
  fixture.detectChanges();
  // 4
  expect(el.textContent).toBe('foo');
});
```

1. Nous récupérons le bouton
2. Nous effectuons un événement clique. Le deuxième paramètre est un objet `Event` que nous pouvons personnaliser (par exemple, dire que c'est un clique-droit). Nous le mettons à `null`
3. Nous faisons le changement manuellement
4. Nous vérifions le titre

Tester un composant avec un service injecté

Voici notre composant

```
import { Component, OnInit } from '@angular/core';
import {AppService} from './app.service';

@Component({
    selector: 'my-app',
    template: `<h1>{{title}}</h1> `
})
export class AppComponent implements OnInit {

  title: string;

  constructor(private appService: AppService) { }

  ngOnInit(): void {
    this.title = this.appService.name;
  }
}
```

Un simple composant qui récupère le nom de l'application présent dans le service `AppService` et qui l'affiche en titre.

Création d'un faux service

```
import { AppComponent } from './app.component';
// 1
import {AppService} from './app.service';
// --

import { async, ComponentFixture, TestBed } from '@angular/core/testing';
import { By }           from '@angular/platform-browser';
import { DebugElement } from '@angular/core';

// 2
class AppServiceMock {
  name:string = 'Mon App';
}
// --

describe('AppComponent', function () {

  let comp: AppComponent;
  let fixture: ComponentFixture<AppComponent>;
  let de: DebugElement;
  let el: HTMLElement;
  let appService: AppService;

  beforeEach(() => {
    TestBed.configureTestingModule({
       declarations: [ AppComponent ],
       // 3
       providers:    [ {provide: AppService, useClass: AppServiceMock } ]
       // --
    });
    fixture = TestBed.createComponent(AppComponent);
    comp = fixture.componentInstance;
    // 4
    appService = TestBed.get(AppService);
    // --
```

```
    de = fixture.debugElement.query(By.css('h1'));
    el = de.nativeElement;
    fixture.detectChanges();
  });


  it('Titre est `Mon App`', () => {
    // 5
    expect(el.textContent).toBe('Mon App');
    // --
  });

  it('Titre devient `foo`', () => {
    appService.name = 'foo';
    fixture.detectChanges();
    expect(el.textContent).toBe('foo');
  });

});
```

Explication :

1. Nous importons le service.
2. Nous créons un faux service. Et c'est ce service que nous utilisons !
3. Nous déclarons au provider que `AppService` utilise en fait notre faux service `AppServiceMock`
4. Nous pouvons récupérer le service avec `TestBed.get()` pour l'utiliser plus tard.
5. Nous testons le titre.

Tester un service asynchrone

Notre service et composant

Voici notre service :

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

@Injectable()
export class AppService {

  run() {
    return new Promise((resolv, reject) => {
      setTimeout(resolv, 3000);
    });
  }

}
```

Et voici notre composant :

```
import { Component, OnInit } from '@angular/core';
import {AppService} from './app.service';

@Component({
    selector: 'my-app',
    template: `
      <h1>{{msg}}</h1>
    `
})
export class AppComponent implements OnInit {

  msg: string = '...';

  constructor(private appService: AppService) { }

  ngOnInit(): void {
    this.appService.run().then(() => {
      this.msg = 'ok';
    })

  }
}
```

Le composant affiche `ok` dans le titre après 3 secondes.

Le test asynchrone

L'item sera le suivant :

```
it('Après le Timeout', async(() => {
  fixture.whenStable().then(() => {
    fixture.detectChanges();
    expect(el.textContent).toBe('ok');
  });
}));
```

Nous utilisons la fonction `async()` pour tester un code asynchrone. Il n'est pas possible de récupérer la promise présent dans `ngOnInit()`. Par contre, la promise est interceptée dans la `async test zone`. La fonction `async` récupère toutes les promises et `whenStable()` se déclenche quand ces promises sont terminées.

Personnaliser des directives

Une directive permet de changer l'état d'un élément du DOM. Par exemple, nous pourrions afficher une bulle sur un élément ayant l'attribut `tooltip` : `<div tooltip></div>`.

Très utile, car nous avons juste à utiliser l'attribut `tooltip` dans notre projet pour afficher une infobulle.

Créer une directive

Créons une simple directive où le but est d'afficher une boite de dialogue lorsqu'on clique sur un bouton.

```
<bouton alert></bouton>
```

La directive est la suivante :

```
import {Directive, HostListener} from '@angular/core';

@Directive({
  selector: `[alert]`
})
export class AlertDirective {
  @HostListener('click', ['$event'])
  run(event: Event) {
    alert('Hello World')
  }
}
```

Nous avons besoin du décorateur `Directive` pour indiquer que nous classe est un directive. Cette directive a une propriété `selector` indiquant les selecteurs CSS concerné par cette directive

Attention ! Le nom peut être trompeur. Nous parlons bien **d'un selecteur CSS** ici ! Ainsi, nous avons des crochets : `[alert]` pour indiquer que nous souhaitons prendre les éléments ayant l'attribut `alert`. Vous l'auriez compris, si nous souhaitons prendre les éléments ayant la classe `alert`, nous aurions mis `.alert`.

Nous utilisons ensuite le décorateur `@HostListener` pour indiquer que la méthode en-dessous (`run()`) est éxécutée au clique. Dans le tableau, nous indiquons le paramètre `$event` : l'événement du clic récupérable dans la méthode `run()`

Avec une valeur en entrée :

Maintenant, je souhaite passer une valeur en entrée dans ma directive :

```
<bouton alert="Hello World"></bouton>
```

Nous utiliserons le décorateur `Input` comme les composants :

```
import {Directive, HostListener, Input} from '@angular/core';

@Directive({
  selector: `[alert]`
})
export class AlertDirective {

  @Input('alert') text:string;
```

```
  @HostListener('click', ['$event'])
  run(event: Event) {
    alert(this.text)
  }
}
```

Puisque le nom `alert` est déjà utilisé pour le nom de la directive, nous le faisons passer en paramètre du décorateur `Input` .

Avec d'autres paramètres

Admettons que nous souhaitons avoir ceci :

```
<bouton alert="Hello World" [go]="clickFn"></bouton>
```

`clickFn` est une fonction de notre composant à appeler quand on clique le sur bouton.

```
import {Directive, HostListener, Input} from '@angular/core';

@Directive({
  selector: `[alert]`
})
export class AlertDirective {

  @Input('alert') text:string;
  @Input() go:Function = () => {};

  @HostListener('click', ['$event'])
  run(event: Event) {
    alert(this.text)
    go();
  }
}
```

Le principe est équivalent à un composant. Nous indiquons la proprété en entrée. Ici, c'est une fonction. Lorsque la méthode `run()` est exécutée, on déclenche la fonction `go()`

Tester

https://embed.plnkr.co/25zYEtxmig1GDmCRkagZ/

Gérer le rendu

```
import {Directive, HostListener, ElementRef, Renderer} from '@angular/core';

@Directive({
  selector: `[highlight]`
})
export class HighlightDirective {
  constructor(private el: ElementRef, private renderer: Renderer) {}

  @HostListener('document:click', ['$event'])
  handleClick(event: Event) {
    if (this.el.nativeElement.contains(event.target)) {
      this.highlight('yellow');
    } else {
      this.highlight(null);
    }
  }

  highlight(color) {
    this.renderer.setElementStyle(this.el.nativeElement, 'backgroundColor', color);
  }
}
```

Régler des propriétés

```
import { Directive, HostBinding, HostListener } from '@angular/core';

@Directive({
  selector: '[buttonPress]'
})
export class ButtonPressDirective {
  @HostBinding('attr.role') role = 'button';
  @HostBinding('class.pressed') isPressed: boolean;

  @HostListener('mousedown') hasPressed() {
    this.isPressed = true;
  }
  @HostListener('mouseup') hasReleased() {
    this.isPressed = false;
  }
}
```

Pratique

Le but va être de réaliser une application avec Angular.

(ensemble) Création de la barre de navigation

Création du module users.

Vous devez créer un module nommé `users`. C'est une page contenant la liste des utilisateurs. Le template doit avoir ce code :

```
<ul>
  <li> Jim </li>
  <li> Ana </li>
  <li> Ben </li>
</ul>
```

(ensemble) Créer barre de recherche dans la navbar

1. Data-binding : nom de l'application
2. Création du composant `search`
3. Valeur en entrée (`navbar` -> `search`). Mettre un prénom par défaut dans l'attribut `value` du champ de recherche
4. Clic sur le bouton
5. Valeur en sortie (`search` -> `navbar`)
6. Double data-binding avec `ngModel`
7. Utilisation de ngIf
8. Utilisation de ngFor

Créer un composant Card

Chaque carte - ou vignette - est un sous composant du composant `UsersComponent`.

1. Créer le composant `user-card`
2. Ajouter la propriété `users` dans le composant `UsersComponent` avec la valeur dans ce lien : `https://jsonplaceholder.typicode.com/users`
3. Afficher autant de fois le composant `user-card` qu'il y a d'utilisateurs dans le tableau
4. Le composant `` `user-card `` à son propre template avec les données de l'utilisateur à afficher

```
<h3>Nom utilisateur ici  </h3>
<p>
  <span> Adresse email ici </span>
</p>
```

Changer la langue d'un bouton

1. Ajouter un bouton `Supprimer` dans le composant `Card`
2. Créer un Pipe qui va afficher `Supprimer` ou `Delete` selon la langue envoyée en paramètre (`fr` ou `en`). Voici comment appliquer le Pipe dans le template :

```
{{ 'REMOVE' | lang:'fr' }}
```

`REMOVE` est un identifiant qui sera remplacé par le bon mot selon la langue

Trier les utilisateurs selon leur extension

Créer une liste déroulante des extensions des emails des utilisateurs sur la page `UsersComponent` . On a donc un tableau de ce style :

```
extensions: Array<string> = ['tv', 'biz', 'io', 'me']
```

Lorsqu'on selectionne une extension, ça affiche que les utilisateurs ayant un meail terminant par cette extension .

Créer le routeur pour aller sur la page Login

1. (ensemble) Créer un module `login`
2. On doit pouvoir aller sur la page du Login en tapant `/login` dans l'URL. Attention, nous ne devons pas avoir la barre de navigation et menu dans la page Login, contrairement à la Home.

(ensemble) Créer un service

Au lieu d'avoir le tableau d'utilisateurs dans `UsersComponent` , le mettre dans un service.

(ensemble) Requête HTTP d'utilisateurs

Requête sur `https://jsonplaceholder.typicode.com/users`

Problématique - Changer l'Observable en Promise

Sans utiliser `toPromise()` de RxJS !

Commencer à concevoir le service AuthService

1. Créer un fichier `auth.service.ts`
2. Dans le service, avoir une méthode `login()` effectuant une requête sur `https://reqres.in/api/login` (en `POST`)
3. Avoir un bouton de connexion dans le composant `LoginComponent`
4. Au clic, sur le bouton, appeler la méthode `login()` précédemment créée

Supprimer un utilisateur

Avoir un bouton pour supprimer un utilisateur. Comment actualiser l'interface après la suppression ?

Aide

```
[1, 2, 3].splice(index, nb)
[1, 2, 3].splice(1, 1) // [1, 3]
```

```
[1, 2, 3].splice(0, 2) // [3]
```

```
[1, 2, 3].findIndex(item => item == 2) // 1
[{ name: 'sam'}, { name: 'jim'}].findIndex(item => item.name == 'jim') // 1
```

Formulaire pour modifier un utilisateur

1. Récupérer le paramètre (ID) dans l'URL et afficher l'utilisateur en question.
2. Avoir un titre avec le nom de l'utilisateur

```
<h1> Nom de l'utilisateur ici </h1>
```

1. Avoir 3 champs pour modifier les propriétés `name` , `username` , `email`
2. Ecrire le validateur qui vérifie l'email

> Note, depuis Angular 4, le validateur `Email` est intégré dans Angular

1. Au clic, sur le bouton `Modifier` , effectuer une requête `PUT` pour mettre à jour l'utilisateur
2. Afficher les erreurs (email invalide ou champs vides)

(optionnel) Barre de recherche

L'utilisateur rentre des mots clé dans la barre de recherche. Seulement les utilisateurs concernés doivent s'afficher.

Directive : supprimer un média lors d'une confirmation

Nous supprimons un utilisateur, seulement lorsque la confirmation est positive.

(optionnel) Bloquer la page `UsersComponent` si utilisateur non connecté

1. (ensemble) créer le service `AuthGard`
2. Effectuer une requête sur `https://reqres.in/api/login` et enregistrer le token en LocalStorage :

```
localStorage.setItem('angular-token', token)
```

1. Bloquer l'utilisateur si le token n'existe pas

Récupérer token

```
localStorage.getItem('angular-token')
```

> On peut aller plus loin en vérifiant l'intégrité du token côté serveur

Tester un Pipe dans les tests unitaires

Sur un nouveau projet :

1. (ensemble) Créer un Pipe pour mettre en majuscule un texte (nommez le `myuppercase`)
2. Effectuer deux tests :
3. Est-ce que le titre dans le template est bien en majuscule ?
4. Est-ce que la méthode `transform()` du Pipe fonctionne bien ?