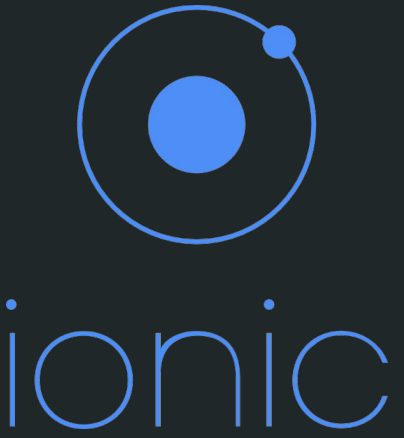
Formation Ionic



ES6 / TypeScript / Angular / Cordova

Jérémie Amsellem - lp1.eu

Présentation

Jérémie Amsellem

- Ethical Hacker
- Dev Python/JS (Back-end / Front-end)
- Frameworks Back : Flask, Django, Express.js
- Frameworks Front : Vue.js, Angular

Objectifs de la formation

- Maîtriser les outils en CLI d'**Ionic** et **TypeScript**
- Créer et débugger des projets **Ionic**
- Maîtriser les paradigmes du langage **TypeScript**
- Comprendre les interactions entre les différents éléments du framework **Angular**
- Maîtriser le comportement d'une application **Ionic** faisant du **CRUD** sur une API **REST**
- Savoir gérer une information de session (JWT Token e.g)

Sommaire

Jour 1 - ES6 / TypeScript

Jour 2 - Ionic-cli / Les bases d'Angular (Composants, Directives, Services)

Jour 3 - Les autres éléments d'Angular et Ionic (Modules, Composants++, Pipes, [...])

Jour 4 - Routing, Formulaires, et JWTs

Jour 5 - Routing avancé, Sécurité, Intercepteurs

Prérequis conseillés

- De bonnes bases en développement Web (**HTML/CSS/JavaScript**)
- Une bonne prise en main de **NodeJS** et **npm** sur votre environnement

Jour 1

ES6 et TypeScript

ECMAScript 6

- **ECMAScript** et **JavaScript**
- **ES6** (ou **ES2015**)
- La dernière spécification utilisée par les navigateurs Chrome, Firefox et Edge
- Ajoute des paradigmes issus de la **POO** et **Programmation Fonctionnelle**

0 - Les variables ES6

ES6 Introduit deux nouveaux mots-clés permettant de définir des variables :

let et const.

Let vise à remplacer var par une alternative plus 'sûre' pour les développeurs et évitant les débordements involontaires dans des scopes parents.

Const permet de définir des constantes, chose jusque là impossible dans le langage.

1 - Les Arrow Functions

Seconde fonctionnalité primordiale : la possibilité de définir des "**arrow functions**" (aussi appelées lambdas dans d'autres langages)

Cela permet de prototyper des fonctions beaucoup plus courtes qu'avec la syntaxe **function() {}**

La syntaxe est la suivante

(params) => { //logique }

2 - Les templates dans les chaînes de caractères

On peut désormais utiliser des chaînes de texte templatées pour interpoler (remplacer dynamiquement) des variables dans du texte.

Pour définir une chaîne de caractères contenant des templates, il faut utiliser des **backquotes** au lieux des simples et doubles quotes habituels.

On utilisera ensuite la syntaxe **${expression}** pour indiquer le contenu à remplacer par une expression JavaScript.

Par exemple :

`Aujourd'hui, nous sommes le ${new Date().toLocaleString()}`

3 - Sets

Des nouvelles structures de données ont également fait leur apparition dans ES6, notamment les très utiles Sets et Maps.

Un Set permet de stocker de valeurs **uniques** de **tous les types**.

Par exemple :

```
const set = new Set()

set.add('test').add(42).add('hello').add({})

set.has(42) // true
```

4 - Maps

Les Maps permettent de stocker des variables en **clé/valeur** (comme un objet JavaScript classique).

En revanche, celles-ci peuvent utiliser **n'importe quel type d'objet** comme clé.

Également, il est très simple d'obtenir la taille d'une Map, en utilisant l'attribut **Map.size**, contrairement aux objets JavaScript.

5 - Les promesses (ou promises)

Les promesses sont une manière plus explicite et claire de définir un comportement asynchrone.

Une promesse est un objet à usage unique, c'est une portion de code qui sera exécutée de manière asynchrone et renverra dans un laps de temps indéterminé des informations à la fin de cette exécution.

On définit une promesse en instanciant un nouvel objet Promise et en y ajoutant notre comportement.

Une fois notre promesse définie, on peut la déclencher en appelant ses méthodes then et catch.

6 - For ...in

Deux nouveaux mot-clés permettent d'itérer plus facilement sur des tableaux et des objets. Ce sont les mots-clés **for ...in et for ...of.**

For ... in permet d'itérer sur les clés d'un **objet** JavaScript.

Par exemple :

```
for (const key in {key1: 1, key2: 2}) {

    console.log(key) // key1, key2

}
```

7 - For ...of

For ...of permet quand à lui d'itérer facilement sur les éléments d'un tableau.

Par exemple :

```
For (const item of [3, 2, 1]) {
    console.log(item) // 3, 2, 1
}
```

8 - Nouvelles méthodes des Strings JavaScript

- String.includes : Retourne true si la chaîne passée en paramètre de includes est contenue dans la String
- String.repeat(n) : Retourne la chaîne multipliée n fois
- String.startsWith : Retourne true si la String commence par la chaîne passée en paramètre
- String.endsWith : Retourne true si la String termine par la chaîne passée en paramètre

9 - Nouvelles méthodes des tableaux JavaScript

- Array.forEach : Permet d'itérer sur chacun des éléments d'un tableau
- Array.filter : Permet de filtrer un tableau JavaScript
- Array.map : Permet de transformer les valeurs d'un tableau
- Array.reduce : Permet de réduire un tableau à un seul nombre
- Array.find : Permet de trouver une valeur dans un tableau (et non son index)

10 - Default

Il est possible de définir des paramètres par défaut dans une fonction dans le cas où ceux-ci ne seraient pas remplis. Par exemple :

```
Function createUser(name, age, location="France') {

    console.log(`${name / ${age} / ${location}`)

}
```

Attention, les paramètres par défaut sont **toujours** les derniers paramètres de votre fonction!

11 - Rest

Rest permet de stocker des paramètres en nombre indéfini dans un tableau. Par exemple :

```
Function test(name, age, ...extra) {

        console.log(extra) // Affiche tout ce qui est passé en plus de name et age

}
```

12 - Spread

Spread, quand à lui permet d'utiliser un tableau en tant que paramètres pour une fonction. Par exemple

Var array = ['Bob', 'TestMan', 'Paris']

Function createUser(firstName, lastName, location) {

}

createUser(...array) // Équivaut à faire createUser(array[0], array[1], array[2])

13 - Getters et Setters

Autre nouvelle possibilité, cette fois-ci concernant les objets JavaScript, on peut désormais définir simplement nos propres **setters** et **getters** sur des attributs :

```
var obj = { get size() {

        //calcul de la taille de l'objet

        }

}

obj.size // Retourne la taille de mon objet
```

TypeScript

Introduction à TypeScript

TypeScript - Introduction

Qu’est-ce que TypeScript ?

- Un langage **Libre** (sous licence GNU/GPL) sorti en 2012
- Développé par **Microsoft**
- **Orienté Objet**
- Transcompilé (ou "transpilé") en **JavaScript** avec l'aide de **babel**
- Accompagné d’un IDE : **Visual Studio Code**
- Utilisé par le framework Angular depuis **Angular 2**

Les annotations de type

En TypeScript on peut annoter nos variables avec différents types

```
1  let primitiveNum: number = 123;
2
3  function square(primitiveNum: number): number{
4      return primitiveNum;
5  }
```

Les Types Primitifs

TypeScript comprend 5 types primitifs :

- Les nombres (number)
- Les chaînes de caractères (string)
- Les booléens (boolean)
- Void
- Null
- Undefined

```
1  var number: number;
2  var string: string;
3  var boolean: boolean;
4
5  number = 123;
6  number = 123.456;
7  number = '123'; // Error
8
9  string = '123';
10 string = 123; // Error
11
12 boolean = true;
13 boolean = false;
14 boolean = 'false'; // Error
```

Le Type "any"

Le type any est un "supertype", il désigne **n'importe quel** type de variable.

Utiliser **any** comme type de variable ou de retour revient à **désactiver la vérification** de typage.

Les Tableaux en TypeScript

Ils sont similaires aux tableaux en JavaScript, à la différence qu'ils peuvent être également sujets aux annotations de type.

```
1  let numberArray: number[];
2
3  numberArray = [0, 1, 2];
4  console.log(numberArray[1]); // 1
5  console.log(boolArray.numberArray); // 3
6  numberArray[1] = 2; // [0, 2, 2]
7
8  numberArray[1] = '1'; // Error!
9  boolArray = [0, '1', 2]; // Error!
10
```

Les Interfaces

Elles ont le même rôle qu'en Java : établir un **modèle de donnée** dont tous les tiers qui l'utilisent devront respecter la structure.

Notez qu'il est possible d'ajouter un champ Optionnel au champ d'une interface en ajoutant un '?' après son nom.

```
interface Person{
    name: string;
    age: number;
}

let person: Person;
person = {
    name: 'John',
    age: 24
};

person = {           // Error : `age` is missing
    name: 'John'
};

person = {           // Error : `age` is the wrong type
    name: 'John',
    age: '24'
};
```

Les Classes

- Constructeur
- Méthodes
- Instanciation

```
1  class Point {
2      x: number;
3      y: number;
4
5      constructor(x: number, y: number) {
6          this.x = x;
7          this.y = y;
8      }
9      add(point: Point) {
10         return new Point(this.x + point.x, this.y + point.y);
11     }
12 }
13
14 var p1 = new Point(0, 10);
15 var p2 = new Point(10, 20);
16 var p3 = p1.add(p2); // {x:10,y:30}
```

L'héritage

- Extends
- Super

```
class Point {
    x: number;
    y: number;

    constructor(x: number, y: number) {
        this.x = x;
        this.y = y;
    }
    add(point: Point) {
        return new Point(this.x + point.x, this.y + point.y);
    }
}

class Point3 extends Point {
  z: number;

  constructor(x: number, y: number, z: number) {
    super(x, y);
  }
  add(point: Point3) {
    return new Point3(this.x + point.x, this.y + point.y, this.z + point.z);
  }
}
```

Les Modificateurs d'Accès

- Public (par défaut)
- Protected
- Private

```
class Person {
    private nickname: string;
    protected name: string;
    public age: number;

    constructor(name: string, nickname: string, age: number) {
       this.name = name;
       this.nickname = nickname;
       this.age = age;
     }
}

class Developer extends Person {
    private language: string;

    constructor(name: string, nickname: string, age: number, language: string)
        super(name, nickname, age);
        this.language = language;
    }

    public getFavoriteLanguage() {
        return `My favorite language is ${this.language} !`;
    }
}
```

Les Génériques

Dans le but de pouvoir créer du code facile à ré-utiliser,

nous avons tout comme en Java, C# ou C++ des génériques permettant d'utiliser des types variables dans les paramètres ou la valeur de retour d'une fonction.

Par exemple :

```
7   function compare<T>(a: T, b: T) {
8       return a > b;
9   }
```

Les Modules

- L'export
- Export as
- L'import

```
13  export class Developer extends Person {
14      private language: string;
15
16      constructor(name: string, nickname: string, age: number, language: string) {
17          super(name, nickname, age);
18          this.language = language;
19      }
20
21      public getFavoriteLanguage() {
22          return `My favorite language is ${this.language} !`;
23      }
24  }
25
26  export { Developer as SoftwareEngineer };
```

```
1  import {Person, Developer, SoftwareEngineer} from './export' // Ou import {*}
2
3  let lp1 = new SoftwareEngineer('Jeremie', 'lp1', 24, 'Python')
```

Les Enums

On peut (contrairement au JavaScript), créer des enums en TypeScript.

Notez que ceux-ci ne commencent pas forcément à 0 ou ne se suivent pas obligatoirement :

```
enum Direction {
    Up = 1,
    Down,
    Left,
    Right,
}
```

Jour 2

ionic

Les bases d'Ionic et Angular (Ionic CLI et éléments de base du Framework)

Introduction à Angular - Sommaire

- Qu’est-ce que le framework Angular ?
- Pourquoi Angular ?
- Comprendre les versions d'Angular
- Travailler avec **Angular-CLI / Ionic-CLI**
- Angular : Principes de base (Composants, Directives et Services)

Qu'est-ce qu'Angular ?

- Créé à la suite d’**AngularJS**, il est aussi appelé Angular2+
- Angular 2.0.0 est mis en production en **Mai 2016**
- Le projet est **Open Source** (sous license MIT)
- Maintenu par **Google** et la communauté Angular

Les Versions d'Angular

- La séparation AngularJS/Angular2+
- Où est passé Angular3 ?
- La gestion de version sémantique (Semantic Versioning)

Pourquoi utiliser Angular ?

- Une **suite d'outils complète** (angular-cli, tests, debug) accompagné d'un langage puissant (**TypeScript**)
- Malgré le fait qu'il soit très complet, le framework affiche des **performances** très correctes par rapport aux autres (React.js, Vue.js)
- Le projet Angular est soutenu par une grande communauté (+25 000 stars sur GitHub, ~500 contributeurs sur le projet)
- Angular permet aussi bien de développer des applications pour mobiles que pour desktop grâce au framework **Ionic**

Ressources : https://medium.com/unicorn-supplies/angular-vs-react-vs-vue-a-2017-comparison-c5c52d620176

Qu'est-ce qu'Ionic ?

Ionic est un framework utilisant **Cordova** et **Angular** permettant de créer des applications mobiles hybrides pour de nombreux supports.

Cordova crée une application native, contenant une WebView affichant le contenu du dossier **www** de l'application.

L'application affichée est en réalité une **application web** écrite en **HTML/CSS/JS**, c'est ce dont on parle quand on parle **d'application hybride**.

Qu'est-ce qu'Ionic ?

Ionic n'est donc en réalité qu'un ensemble de **plugins Cordova**, **services, directives et composants Angular** gérés par un **client en ligne de commande**.

Prérequis

Il est nécessaire, si vous souhaitez pouvoir tester votre application hybride sur un support en particulier (e.g Android ou iOS) d'installer les outils de développement spécifiques à ce support (**Android Studio ou au minimum Android SDK, XCode pour iOS**).

Il sera également nécessaire d'avoir **node et npm** installés sur votre machine.

Travailler avec Ionic-CLI

- Installation des outils Ionic-CLI
  - **npm install -g ionic cordova**
- Utilisation des outils Ionic-CLI
  - Créer un nouveau projet : **ionic start MessageBoard**
  - Lancer un serveur web avec du LiveReload
  - Générer des ressources Angular
- Anatomie du projet généré par **Ionic-CLI**
- Débogage des applications Ionic
- Travailler avec les outils Android
- Travailler avec les outils iOS

Travailler avec Ionic-CLI

- Ionic cordova run <platform>
- Ionic cordova build <platform>
- Ionic cordova emulate <platform>

Hands-On - Création d'un nouveau projet

- Création d'un nouveau projet
- Anatomie d'un projet Ionic
- Création de Composants, Templates et ajout de métadonnées
- Création et utilisation de Services

Les Composants Angular

En Angular, un composant est une partie de notre projet associant des **éléments visuels** à une **logique** leur étant propre.

Le template est représenté par du **HTML** et du **CSS** pour le style,
la logique est quand à elle écrite avec du code **TypeScript**.

Un composant peut contenir un ou plusieurs autres composants, on peut également les réutiliser pour éviter de dupliquer du code.

Les Métadonnées d'un composant

- **Selector**: Sélecteur **CSS** permettant à Angular d'identifier notre composant
- **Template**: Le template **HTML** de notre composant, sous forme de string
- **TemplateURL**: Le chemin vers le fichier **HTML** de template de notre composant
- **Style**: Le style de notre composant, sous forme de string[ ]
- **StyleUrls**: Les chemins des feuilles de styles de notre composant

La liaison de données dans un composant

Il existe plusieurs moyen de lier dynamiquement de la donnée dans un template.

Le plus simple consiste à entrer une expression TypeScript entre des double accolades {{ **expression** }}, celle-ci sera interprétée avant d'être affichée dans le navigateur du client.

Les informations vont de notre composant vers son template, on parle de "**one-way binding**" dans ce cas là.

Les Directives de base d'Angular

Les directives sont des outils permettant d'ajouter de la **logique** à un **template HTML existant**.

Une directive fonctionne comme un composant, mais ne peut **pas comporter de template**.

Certaines directives sont pré-définies dans le framework Angular, elles permettent d'effectuer certaines opérations concernant la logique d'affichage de nos informations.

La directive nglf

nglf permet d'afficher ou non un élément de notre template en fonction d'une condition. Elle s'utilise comme ceci :

<tag *nglf="condition === true"></tag>

La directive ngFor

ngFor permet d'itérer sur les éléments d'un tableau et d'associer un élément de template à chacun de ces éléments. Il s'utilise comme ceci :

```
<tag *ngFor="let n of [1,2,3]">{{n}}</tag>
```

Les Services Angular

Les services sont des **singletons** partagés au sein de toute notre application Angular.

Ils permettent généralement de sauvegarder et **partager** des informations entre différents **composants** Angular.

Les Services Angular

- Les services sont des **singletons** au sein de notre application
- Ils s'injectent dans le **constructeur** d'un élément Angular
- Ils sont définis dans une factory qui les instanciera
- Ils permettent de garder en mémoire de la donnée qui pourra être **partagée** entre les différents éléments Angular
- On affectera tous les traitements "lourds" de donnée à des services plutôt qu'à des composants

Jour 3

Utilisation avancée d'Angular et Ionic (Cycle de vie des composants, inputs, outputs et HTTP, Ionic Native, Storage)

Le Cycle de Vie des Composants Angular

Les composants peuvent implémenter des **interfaces** donnant accès à des méthodes **'crochet'** permettant d'assigner des **comportements** à des **moments précis** de la vie de notre composant.

Nous avons déjà utilisé le crochet **onInit**, mais il en existe d'autres nous permettant d'avoir un contrôle plus précis sur les différentes actions à effectuer en fonction de ce qu'Angular effectue comme changement sur notre composant.

Le Cycle de Vie des Composants Angular

- **Constructor** : Appelé quand Angular instancie l'objet composant
- **ngOnChanges** : Appelé à chaque changement sur la donnée liée au composant
- **ngOnInit** : Appelé après la 1ere fois que la donnée est liée au composant
- **ngDoCheck** : Appelé à chaque cycle de vérification sur la data liée au composant
- **ngOnDestroy** : Appelé lorsque le composant ne fait plus partie du DOM

Le Cycle de Vie des Composants Ionic

Ionic possède en plus du cycle de vie d'Angular son propre cycle de vie :

- **ionViewDidLoad**: Déclenché une fois que la vue est chargée en mémoire.
- **ionViewWillEnter**: Déclenché avant qu'une vue soit chargée comme vue active
- **ionViewDidEnter**: Déclenché après qu'une vue soit chargée comme vue active
- **ionViewWillLeave**: Déclenché avant qu'une vue active soit remplacée par une autre.
- **ionViewDidLeave**: Déclenché après qu'une vue active soit remplacée par une autre.
- **ionViewWillUnload**: Déclenché avant qu'une vue soit déchargée en mémoire.

La liaison de donnée avancée

Il existe plusieurs manières de lier de la donnée à un composant.

Nous avons déjà vu l'utilisation de **double accolades**, il est également possible de définir des **inputs** au sein d'un composant.

Ce sont des informations qui seront liées à l'intérieur du **template** dans lequel est instancié notre composant, au travers **d'attributs** HTML.

Les Inputs

Pour récupérer la valeur d'un **Input**, on utilise la métadonnée @Input :

```
export class Component {

 @Input() name: string;

}
```

Et du côté du template, on peut spécifier l'input ainsi :

```
<component [name]="test"></component>
```

Les Outputs

Les **Inputs** permettent de faire transiter une information d'un **composant parent** vers un **composant enfant**.

Mais il est également possible de faire remonter une information d'un **composant enfant** vers un **composant parent**, en utilisant la métadonnée **@Output** :

```
export class ChildComponent {

    @Output() onParentAction = new EventEmitter<boolean>();

}
```

```
<child-component
(onParentAction)="parentCallback($event)"></child-component>
```

Les Modules Angular

Un module permet de lier/ranger des éléments Angular ayant des logiques communes ensemble.

- Il y en a toujours au minimum un dans une application
- **Déclarations** : Spécifier quels Composants, Directives et Pipes appartiennent au module
- **Imports** : Les autres modules dont notre module dépend
- **Providers** : Les services appartenant au module
- **Bootstrap** : Le(s) composant(s) qui seront créés en premier lieu dans le module

Modèles architecturaux

Lorsque vous travaillez sur un projet il est important que son architecture reflète ses fonctionnalités et la manière dont elles sont hiérarchisées.

Nous allons étudier un modèle architectural simple, qui consiste à découper votre projet en modules, chaque module correspondant à une fonctionnalité.

Pour plus d'informations :

- **Le Style Guide Angular** https://angular.io/guide/styleguide

Utilisation de Composants dans un Composant

Pour pouvoir utiliser un composant custom au sein d'un autre composant en Angular il est nécessaire d'ajouter ce composant dans les déclarations de votre module.

Il est ensuite possible d'instancier directement ce composant dans un autre à l'aide du sélecteur HTML indiqué dans le champ selector de celui-ci.

Création de Directives

- Les **directives** sont les classes parentes des **composants**
- Elles permettent d'assigner des **comportements** à des éléments graphiques
- Elles contiennent une option **(hosts): {}** permettant d'injecter des **attributs** dans l'élément du DOM sur lequel elles sont attachées
- La métadonnée **@HostListener('eventname')** permet d'attacher une méthode à un évènement JavaScript

Les Pipes

Les pipes permettent de **transformer** une manière dont va être **affichée** une donnée.

- Un Pipe ne change pas la valeur des variables de son composant
- Il fonctionne en implémentant une méthode **transform**, prenant **une valeur en paramètre** et **retournant la valeur transformée**
- Il peut prendre un nombre illimité de paramètres

Par exemple, avec le pipe 'date' permettant de formater une date :

<tag>{{ '2017-01-01' | date:'short' }}</tag>

Le pattern observer

Le pattern observer est une manière de concevoir les **changements d'états** dans une application.

Dans celui-ci, un **sujet** est un objet dont on va observer les changements.

On appelle **observer** l'entité dans notre code qui souhaitera observer ces changements d'états.

Un **sujet** utilisera des objets appelés **Observables** pour notifier **l'observer** des changements de son état.

Pour être notifié de ces changements, il faut s'abonner à un **Observable.**

S'abonner revient à définir une **fonction à exécuter** lorsqu'il y aura un **changement** sur notre **sujet**.

RxJS

Angular utilise de base la bibliothèque **RxJS** qui implémente le **pattern observer**.

Une fois importés depuis Rx, nous avons accès (entre autres) à des objets **Subject** et **Observable.**

Il est possible d'observer un objet de notre projet ainsi à l'aide de RxJS :

```
let subject = new Subject();

let observable = subject .asObservable()

observable.subscribe((donnée) => {//  traitement des changements de donnée })
```

Les requêtes HTTP

Les objets **Angular** permettant de faire des requêtes HTTP sont définis dans le module **HttpClientModule**.

Celui-ci ne fait pas partie du module "**global**" du framework Angular, il faut importer et ajouter celui-ci au module principal de notre application depuis **@angular/common/http** avant de s'en servir !

Une fois correctement ajouté aux dépendance du projet, on peut importer et utiliser le service **HttpClient,** mettant à disposition des méthodes **get, post, put, delete, patch, options et head**.

Ces méthodes, une fois invoquées retournent un **Observable RxJS** qui permet d'observer le retour de notre requête.

Interaction avec des champs de formulaire

Tout comme le module HttpClientModule, le module permettant l'interaction avec des champs de formulaire ne fait **pas partie** du **module global Angular**, il est donc nécessaire de l'importer en ajoutant **FormsModule** au module principal de notre application.

Une fois ce module importé, nous pouvons utiliser la directive ngModel, permettant de lier un champ de formulaire à une variable TypeScript :

```
<input [(ngModel)]="login"/>
```

Template Reference Variables (#)

Il est possible, au sein d'un template de définir des variables **associées à certains éléments visuels dans le DOM**.

Il faut pour cela ajouter, comme un attribut dans les balises concernées un **#** suivi du **nom de notre variable** (ici userInput) :

<input #userInput (click)="printVariable(userInput)"/>

ngElse

Nous avons vu qu'il était possible d'ajouter des conditions à l'affichage avec **ngIf**,

depuis la version 5 d'Angular il est également possible d'avoir un comportement proche de conditions **if/else**.

Il faut en premier lieu créer un template Angular, et le stocker dans une **template reference variable** :

<ng-template #falseData>Data is false</ng-template>

Il est ensuite possible d'y faire référence dans notre ngIf comme ceci :

**<div *ngIf="data === true; else falseData"> **

ngSwitch

Autre possibilité permettant de gérer différentes conditions directement dans un template HTML : le ngSwitch :

```
<div [ngSwitch]="user.group">

<button *ngSwitchCase="admin">Admin Area</button>

<button *ngSwitchCase="user">User Area</button>

</div>
```

Ionic Native

Ionic propose une suite de modules permettant d'**intéragir avec des composants natifs** dans votre application (Stockage, GPS, Bluetooth, WiFi, etc...).

Ils font partie du groupe "**Ionic Native**", (@ionic-native sur NPM) comportant des modules composés d'un **plugin Cordova** et d'**APIs Angular** permettant d'envoyer des requêtes "natives" vers l'appareil utilisé.

Pour installer un plugin Ionic Native, on utilisera la commande :

```
npm install @ionic-native/camera #Pour installer les APIs Angular

Suivie de

ionic cordova plugin add cordova-plugin-camera #Pour installer le plugin cordova
```

Ionic Native

Une fois le plugin Ionic Native correctement installé il restera à l'ajouter dans la liste de providers du module où vous souhaitez utiliser le plugin :

```
...

import { Camera } from '@ionic-native/camera';

...

@NgModule({
  ...

  providers: [
    ...
    Camera
    ...
  ]
  ...
})
export class AppModule { }
```

Jour 4

Utilisation Avancée d’Ionic

Gestion des comptes / sessions

Une session est une information envoyée par le serveur permettant d'identifier un utilisateur connecté.

Il existe plusieurs stratégies de gestions de sessions, celle que nous utilisons dans le cours fonctionne en stockant celle-ci dans le localStorage de notre application.

Le localStorage est un espace permettant de stocker des informations en clé/valeur partagé au sein de notre WebView.

WebSockets

Une **WebSocket** permet d'établir une connexion **continue** entre un serveur et ses clients. C'est une implémentation volontairement **proche** du fonctionnement des **socket TCP/IP Unix,** basée sur le protocole HTTP.

Pour se connecter à la WebSocket d'un **serveur**, il suffit en **JavaScript** de créer un objet WebSocket auquel on passe en **paramètre l'URL** de la WebSocket de destination.