DÉVELOPPEMENT DES APPLICATIONS MOBILES HYBRIDES MULTIPLATEFORMES

CH 7 - Architecture avancée d'une application Ionic / Angular

FORMATEUR : MAHAMANE SALISSOU YAHAYA : ysalissou@gmail.com

(TechnoLAB -ISTA)

4ième année

OCTOBRE 2020

- Dans ce chapitre nous allons aborder des notions un peu plus avancées du framework Ionic (Angular) comme les composants, les directives, les providers, les services, les pipes, les modules ou encore les plugins natifs.

- Nous en profiterons aussi pour introduire les frameworks React et VueJS, inclus depuis la version 4 de Ionic.

MODELS

OCTOBRE 2020

Model

Il est utilisé par le component, le template ainsi que pour transmettre les données au service

La création d'un nouveau model se fait simplement en saisissant la commande suivante à la racine de votre projet ionic ou Angular :

$ ng g class nom_du_model [--spec=false]

Composants

OCTOBRE 2020

Composants

La majorité des développements sous Angular, et comme vous l'aurez compris, sous Ionic également (Ionic étant en fin de compte un projet Angular), est effectué au niveau des composants. Nous avons déjà étudier le composant Root, dont l'arborescence est la suivante :

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

Nous avons aussi écrit un composant Etudiants qui nous permettait d'afficher la liste des étudiants.

La création d'un nouveau composant se fait simplement en saisissant la commande suivante à la racine de votre projet ionic ou Angular :

$ ionic g module components [--routing] # Création du module components qui stockera nos composants
$ ionic g c components/mon_composant # Création de notre composant
$ ionic g
> *ng generate component etudiants*

Composants

Il faut ensuite modifier le module (components.module.ts) de la manière suivante (Certaines versions de Ionic et Angular font ce travail pour vous):

src/app/etudiants/ etudiants.module.ts

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';

import { IonicModule } from '@ionic/angular';

import { EtudiantsPageRoutingModule } from './etudiants-routing.module';

import { EtudiantsPage } from './etudiants.page';

@NgModule({
  imports: [
    CommonModule,
    FormsModule,
    IonicModule,
    EtudiantsPageRoutingModule
  ],
  declarations: [EtudiantsPage]
})
export class EtudiantsPageModule {}
```

IonicModule : Contient tous les éléments de base de Ionic
CommonModule : Contient tous les éléments de base d'Angular : directives, pipes, NgIf,...
CUSTOM_ELEMENTS_SCHEMA : permet l'affichage des composants Ionic dans des composants personnalisés.

Composants

Chaque fois que vous devrez créer un composant, il vous suffira de le rajouter directement dans le fichier **src/app/etudiants/etudiants.module.ts.**

Rappelons que les modules sont chargés du bootstrapping (démarrage) de composants. C'est donc ce module qu'il faudra éventuellement déclarer dans le module principal de la page ou de composant qui l'appelle.

Directives

OCTOBRE 2020

Directives

Une directive est un élément qui va nous permettre d'étendre des fonctionnalité html.

Il en existe différents types :

- **Directive de type attribut** : vous en avez déjà vu, elles permettent de modifier du html. Citons par exemple **color**, une directive qui permet d'attribuer la couleur passée en paramètre à l'élément concerné, ou encore la directive **padding**, qui permet d'ajouter un padding à l'élément qui l'invoque.

- **Directive de type composant** : oui au risque de vous embrouiller un peu, un composant est en réalité une directive, mais dotée d'un template html. La directive est en quelque sorte l'atome, le composant la molécule.

- **Directive de type structure** : Ce type de directive est fait pour la manipulation du DOM et commence toujours par un "*****". On peut citer parmi celles-ci deux que nous avons déjà utilisées à savoir ***ngIf** et ***ngFor**.

la création d'une directive se fait simplement en saisissant la commande suivante :

$ ionic g directive maDirective

Directives

Créons par exemple une directive que nous appellerons **trombiborder** et qui permettra de rajouter des marges au sein de l'élément qui l'invoque.

$ ionic g directive directives/trombiborder

Ionic (disons plutôt Angular) utilisant par défaut le **Lazy loading** (chargement de code uniquement si nécessaire pour augmenter les performances générales), on ne peut plus vraiment déclarer notre directive de manière globale dans le module **src/app/app.module.ts**.

Pour être exploitable, la directive doit être appelée dans le module de la page ou du composant qui souhaite l'utiliser.

Si l'on souhaite par exemple appeler notre directive dans une page nommée **EtudiantDetailPage** affichant notre fil d'actualité, il suffit d'éditer le module de cette page comme ceci :

```
import { EtudiantDetailPage } from './etudiant-detail.page';
import { TrombiborderDirective } from '../directives/trombiborder.directive';

@NgModule({
  imports: [
    CommonModule,
    FormsModule,
    IonicModule,
    EtudiantDetailPageRoutingModule
  ],
  declarations: [EtudiantDetailPage,TrombiborderDirective]
})
export class EtudiantDetailPageModule {}
```

Et puisqu'il a été appelé ici, il faut donc le retirer dans le fichier **src/app/app.module.ts**, s'il s'y trouve.

Directives

Il ne reste plus qu'à éditer notre directive pour qu'il fasse ce que l'on souhaite, à savoir rajouter une bordure :

```
import { Directive, ElementRef, Renderer2 } from '@angular/core';

@Directive({
  selector: '[appTrombiborder]'
})
export class TrombiborderDirective {

  constructor(public element: ElementRef, public renderer: Renderer2) {
    console.log('ElementRef', element);
    this.renderer.setStyle(this.element.nativeElement, 'border', '1px solid #000000');
  }

}
```

On peut ainsi utiliser notre nouvelle directive dans le contenu html de notre page EtudiantDetailPage :

```
<ion-content>
  <p appTrombiborder>Nom : Moussa</p>
</ion-content>
```

Providers, Services

OCTOBRE 2020

Services/Providers

Qu'est-ce qu'un service Angular ?

Avec Angular, une dépendance est généralement l'instance d'une classe permettant de **factoriser certaines fonctionnalités** ou **d'accéder à un état** permettant ainsi aux composants de **communiquer entre eux**.

Dans le vocabulaire Angular, ces classes sont appelées "**services**".

Injection d'un service Angular

Un service Angular peut être injecté par n'importe quelle classe Angular (i.e. : composant, Directive, Service ou Pipe) via les paramètres de son constructeur.

```
@Component({
  selector: 'app-specialites',
  templateUrl: './specialites.component.html',
  styleUrls: ['./specialites.component.scss']
})
export class SpecialitesComponent {
  constructor(
    protected service: SpecialitesService,
    protected translate: TranslateService) { }

}
```

Services/Providers

Déclaration d'un Service

Pour déclarer un service Angular, il suffit de créer une classe TypeScript et de la décorer avec le décorateur **@Injectable().**

```
@Injectable({
  providedIn: 'root'
})
export class SpecialitesService {

}
```

Pour créer un service utiliser la commande **ng g service nom_du_service**

Afin de pouvoir instancier un service, Angular a besoin d'un "**provider**" lui indiquant comment produire l'instance de ce service.

Mais depuis Angular 6, il n'est plus nécessaire de définir les "providers" au niveau des modules pour vous services.

Services/Providers - Utilisation de HttpClient

Injection du service HttpClient

HttpClient est un service Angular ; on peut donc le récupérer avec la Dependency Injection.

```
@Injectable({
  providedIn: 'root'
})
export class SpecialitesService {
  constructor(protected httpClient: HttpClient) {
  }

}
```

On obtient l'erreur suivante No provider for HttpClient! car le service HttpClient n'est pas encore Tree-Shakable et il faut donc importer le module associé **HttpClientModule**.

Etant donné que le service HttpClient est stateless, nous pouvons importer le module HttpClientModule directement dans notre Module **AppModule**.

```
@NgModule({
  declarations: [
    AppComponent,
  ],
  imports: [
    HttpClientModule,
  ],
  providers: [HttpClient],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Services/Providers - Utilisation de HttpClient

Exécution de la requête

HttpClient.get

Nous pouvons donc récupérer les données par API

```
public findAll() {
  const url = encodeURI('http://localhost:9090/api-sih/Specialites');
  return this.httpClient
    .get(url, this.baseOption)
    .pipe(catchError(this.handleError));
}
```

apiUrl: l'adresse de la ressource à récupérer.
baseOption : le type de retour

```
public static baseOption() {
  const _headers = new HttpHeaders({
    'Content-Type': 'application/json',
  });
  return { headers: _headers };
}
```

Services/Providers - Utilisation de HttpClient

Exécution de la requête

HttpClient.post, HttpClient.put et HttpClient.delete

```
public save(entity: Specialites) {
  const url = encodeURI(this.apiUrl.concat(this.address()));
  return this.httpClient
    .post(url, JSON.stringify(entity), this.baseOption)
    .pipe(catchError(this.handleError));
}

public update(id: number, entity: Specialites) {
  const url = encodeURI(this.apiUrl.concat(this.address() + "/" + id));
  return this.httpClient
    .put(url, JSON.stringify(entity), this.baseOption)
    .pipe(catchError(this.handleError));
}

public delete(id: number) {
  const url = encodeURI(this.apiUrl + this.address() + "/" + id);
  return this.httpClient
    .delete(url, this.baseOption)
    .pipe(catchError(this.handleError));
}
```

Services/Providers - Utilisation de HttpClient

Déclenchement de la requête au subscribe

En inspectant le comportement du "browser“, on peut remarquer que la **requête n'est pas envoyée**.

En effet, les méthodes get, delete, patch, post, put, request etc**... retournent toujours un Observable**.

Cet Observable est "lazy" et il faut donc **subscribe** pour déclencher le traitement.

```
loadList() {
  this.service.findAll().subscribe(data => {
    this.list = data;
    console.log(data)
  }, error => {

  });
}
```

Traitement de la "response«

Lors de la compilation, TypeScript ne connait pas le type des données retournées par l'API.
Par défaut, la méthode get retourne un objet de type **Observable<Object>** .

Les méthodes de la classe HttpClient sont des méthodes génériques et il est donc possible de contrôler leur type de retour.

```
public static getDatas(data): any[] | any {
  return data !== null && data.hasOwnProperty('content') ? data.content : data;
}
```

Services/Providers – http ionic

Supposons que l'on souhaite récupérer la liste des professeur sous forme de tableau comme à partir d'un serveur distant accessible à l'adresse suivante : **https://jsonplaceholder.typicode.com/users**

Pipes

OCTOBRE 2020

Pipes

On en a déjà un peu parlé dans les chapitres précédents. Les pipes permettent de modifier la forme d'un contenu avant son affichage.

Ils sont utilisés uniquement côté html (ex : **src/mapage/mapage.page.html**) Citons quelques pipes intéressant :

- **currency** : permet de rajouter une devise avant la valeur sur laquelle on l'applique
- **date** : formatage de date
- **uppercase** : transforme du texte en majuscule
- **lowercase** : transforme du texte en minuscule
- **json** : affiche le contenu d'un objet ou d'un texte au format JSON

```
let maVariable = "Hello mmi";
```

Puis côté html, on peut appeler le pipe uppercase:

```
<span>{{maVariable | uppercase}}</span>
```

Pipes

Pour créer un nouveau Pipe, il suffit de faire :

$ ionic g module pipes // si le module Pipe n'existe pas déjà !
$ ionic g pipe pipes/monPipe

src/app/pipes/pipes.module.ts

Ajustons le module Pipes pour qu'il exporte (rendre disponible au grand public) notre nouveau composant.

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { MonPipePipe } from './mon-pipe.pipe';
@NgModule({
  declarations: [MonPipePipe],
  imports: [
    CommonModule
  ]
})
export class PipesModule { }
```

Pipes

Il vous faut ensuite importer le module Pipes dans le module de la page dans laquelle vous souhaitez l'utiliser (pour respecter le lazy loading) :

src/mapage/mapage.module.ts

```
import { EtudiantDetailPage } from './etudiant-detail.page';
import { TrombiborderDirective } from '../directives/trombiborder.directive';
import { PipesModule } from '../pipes/pipes.module';

@NgModule({
  imports: [
    CommonModule,
    FormsModule,
    IonicModule,
    PipesModule,
    EtudiantDetailPageRoutingModule
  ],
  declarations: [EtudiantDetailPage, TrombiborderDirective]
})
export class EtudiantDetailPageModule { }
```

Pipes

La logique du Pipe (comment il interagit avec l'élément qui l'appelle) est à gérer dans le fichier

src/app/pipes/mon-pipe.pipe.ts

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

@Pipe({
  name: 'monPipe'
})
export class MonPipePipe implements PipeTransform {

  transform(value: any, ...args: any[]): any {
    // value est le contenu à modifier
    // Ici on rajoutera un hashtag (#) avant la valeur
    return `#${value}`;
  }

}
```

Nous pouvons désormais appeler notre pipe sans problème depuis le html de notre page

src/mapage/mapage.page.html

```
<span>{{maVariable | monPipe}}</span>
```

Modules

OCTOBRE 2020

Modules

Un module angular permet de regrouper en un seul endroit des composants, des directives, des pipes et des services d'une application.

Prenons par exemple le module racine de notre application défini comme ceci :

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

import { IonicModule, IonicRouteStrategy } from '@ionic/angular';
import { SplashScreen } from '@ionic-native/splash-screen/ngx';
import { StatusBar } from '@ionic-native/status-bar/ngx';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { HTTP } from '@ionic-native/http/ngx';

@NgModule({
  declarations: [AppComponent],
  entryComponents: [],
  imports: [BrowserModule, IonicModule.forRoot(), AppRoutingModule],
  providers: [
    StatusBar,
    SplashScreen,
    HTTP,
    { provide: RouteReuseStrategy, useClass: IonicRouteStrategy }
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Modules

Déclarations

C'est un tableau des composants qui seront utilisés dans l'application.

```
declarations: [
  AppComponent
],
```

Import

C'est un tableau des différents modules de l'application. C'est dans cette section que l'on déclare le module regroupant l'ensemble des composants.

```
imports: [BrowserModule,
  IonicModule.forRoot(),
  AppRoutingModule
],
```

Providers

C'est ici que seront déclarés tous les services utilisés dans l'application.

```
providers: [
  StatusBar,
  SplashScreen,
  HTTP,
  { provide: RouteReuseStrategy, useClass: IonicRouteStrategy }
],
```

Bootstrap

On déclare ici le composant principal de l'application

Formulaires

OCTOBRE 2020

Formulaires

Il existe différentes façon d'implémenter les formulaires avec Angular.

Ce guide aborde **les deux approches les plus répandues** :

- **Template-driven Forms (à éviter)** : inspirée du "**two-way binding**" utilisé dans AngularJS, cette approche a de nombreuses limitations et s'avère rapidement **fastidieuse à implémenter**, **peu extensible** et **peu efficace**.

- **Reactive Forms (à adopter)** : cette approche vient appuyer le paradigme "**Reactive Programming**" qui fait parti des fondements d'Angular avec : une meilleure **séparation de la logique du formulaire et de la vue**, **une meilleure testabilité**, des Observables, **la génération dynamique de formulaires etc**...

Template-driven Forms

L'approche "**Template-driven Forms**" nécessite de mettre la majeure partie de la logique du formulaire dans le template. Cela donne une fausse impression de "**quick win**" au départ et le code devient rapidement difficile à maintenir et à tester.

ngModel

La Directive ngModel est au coeur des "Template-driven Forms". Elle permet principalement de "**binder**" dans les deux sens le "**model**" avec la "**view**". C'est ce que l'on appelle le "**Two-way Binding**".

Pour profiter de la directive NgModel, il faut importer le module **FormsModule** dans les modules contenant des composants qui en dépendent.

```
imports: [
  CommonModule,
  FormsModule,
  TablesRoutingModule,
  TranslateModule,
]
```

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

```
@Component({
  selector: 'app-specialites-add',
  templateUrl: './specialites-add.component.html',
  styleUrls: ['./specialites-add.component.scss']
})
export class SpecialitesAddComponent implements OnInit {
  title: string;
  constructor() { }
  ngOnInit(): void {
  }

}
```

A chaque modification du "model", la "view" sera mise à jour et inversement.

Template-driven Forms

L'approche "**Template-driven Forms**" nécessite de mettre la majeure partie de la logique du formulaire dans le template. Cela donne une fausse impression de "**quick win**" au départ et le code devient rapidement difficile à maintenir et à tester.

ngModel

La Directive ngModel est au coeur des "Template-driven Forms". Elle permet principalement de "**binder**" dans les deux sens le "**model**" avec la "**view**". C'est ce que l'on appelle le "**Two-way Binding**".

Pour profiter de la directive NgModel, il faut importer le module **FormsModule** dans les modules contenant des composants qui en dépendent.

```
imports: [
  CommonModule,
  FormsModule,
  TablesRoutingModule,
  TranslateModule,
]
```

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

```
@Component({
  selector: 'app-specialites-add',
  templateUrl: './specialites-add.component.html',
  styleUrls: ['./specialites-add.component.scss']
})
export class SpecialitesAddComponent implements OnInit {
  title: string;
  constructor() { }
  ngOnInit(): void {
  }

}
```

A chaque modification du "model", la "view" sera mise à jour et inversement.

Détection du "submit" avec ngSubmit

Pour détecter le "submit" du formulaire, il faut utiliser l'Output **ngSubmit** sur l'élément **form**.

```
<form (ngSubmit)="submitSpecialite()">
    <input name="title" [(ngModel)]="title">
    <button type="submit">SUBMIT</button>
</form>
```

```
@Component({
  selector: 'app-specialites-add',
  templateUrl: './specialites-add.component.html',
  styleUrls: ['./specialites-add.component.scss']
})
export class SpecialitesAddComponent implements OnInit {
  title: string;
  constructor() { }
  ngOnInit(): void {
  }
  submitSpecialite() {
    console.log(this.title);
    /* Reset specilite's title. */
    this.title = null;
}

}
```

Avantages des « Reactive Forms »

Pour remédier aux différentes limitations des Template-driven Forms, **Angular offre une approche originale et efficace nommée "Reactive Forms"** présentant les avantages suivants :

- La logique des formulaires se **fait dans le code TypeScript**. Le formulaire devient alors plus facile à **tester** et à **générer dynamiquement**.

- Les "Reactive Forms" utilisent des **Observables** pour faciliter et encourager le "**Reactive Programming**".

La boite à outils des "Reactive Forms"

FormControl
La classe FormControl est une indirection permettant de contrôler et d'accéder à l'état des "controls" de la vue (e.g. <input>).

```
@Component({
  selector: 'app-specialites-add',
  templateUrl: './specialites-add.component.html',
  styleUrls: ['./specialites-add.component.scss']
})
export class SpecialitesAddComponent implements OnInit {
  titleControl = new FormControl();
  descriptionControl = new FormControl();
  constructor() { }
  ngOnInit(): void {
  }
  submitSpecialite() {
    console.log(this.titleControl.value);
    console.log(this.titleControl.value);
  }

}
```

```
<form (ngSubmit)="submitSpecialite()">
    <input type="text" [formControl]="titleControl">
    <textarea [formControl]="descriptionControl"></textarea>
    <button type="submit">SUBMIT</button>
</form>
```

On obtient l'erreur suivante :
Can't bind to 'formControl' since it isn't a known property of 'input'.

... car l'Input formControl est ajouté via la "directive" FormControlDirective

La boite à outils des "Reactive Forms"

Pour profiter de la directive FormGroupDirective, il faut importer le module **ReactiveFormsModule** dans les modules contenant des composants qui en dépendent.

```
imports: [
  CommonModule,
  FormsModule,
  ReactiveFormsModule,
  TablesRoutingModule,
  TranslateModule,
]
```

Vous pouvez retirer l'import du module FormsModule.

FormControl fournit différentes propriétés et méthodes permettant de piloter le "control" :

- **value** : permet d'accéder à la valeur actuellement contenu dans le "control".
- **valueChanges** : est un Observable permettant d'observer les changements de valeur du "control".
- **reset**, **setValue** et **patchValue** permettent de modifier l'état et la valeur du "control".

La boite à outils des "Reactive Forms"

FormGroup

FormGroup permet de regrouper des "controls" afin de faciliter l'implémentation, récupérer la valeur groupée des "controls" ou encore appliquer des validateurs au groupe.

Le FormGroup est construit avec un "plain object" associant chaque propriété à un FormControl .
Il est alors possible d'accéder aux valeurs et aux "controls" via la propriété associée.

```
specialiteForm = new FormGroup({
  title: new FormControl(),
  description: new FormControl()
});
submitSpecialite() {
  console.log(this.specialiteForm.value);
}
```

```
<form>
    <input type="text" [formControl]="specialiteForm.controls.title">
    <textarea [formControl]="specialiteForm.controls.description"></textarea>
    <button type="submit" (click)="submitSpecialite()">SUBMIT</button>
</form>
```

La boite à outils des "Reactive Forms"

FormGroup

Pour éviter d'accéder aux controls via la propriété **FormGroup.controls**, les "Reactive Forms" implémentent également une directive **FormGroupDirective** qui permet **d'associer un FormGroup à un élément du DOM (form en général mais on peut utiliser d'autres éléments)**. Cela permet alors d'associer les "controls" via leur nom à l'aide de l'Input **formControlName**.

```
<form [formGroup]="specialiteForm" (ngSubmit)="submitSpecialite()">

    <input type="text" formControlName="title">

    <textarea formControlName="description"></textarea>

    <button type="submit">SUBMIT</button>

</form>
```

La classe FormGroup propose de nombreuses propriétés et méthodes similaires à celles rencontrées précédemment avec FormControl. En effet, FormControl et FormGroup héritent de la classe abstraite **AbstractControl**.

La boite à outils des "Reactive Forms"

FormBuilder

Le module ReactiveFormsModule implémente également un service **FormBuilder** qui permet d'ajouter un peu de "syntactic sugar" afin de simplifier la création des FormGroup ou FormControl.

```
export class SpecialitesAddComponent implements OnInit {

  specialiteForm: FormGroup;

  constructor(private _formBuilder: FormBuilder) {
    this.specialiteForm = this._formBuilder.group({
      title: null,
      description: null
    });
  }
  ngOnInit(): void {
  }

  submitSpecialite() {
    console.log(this.specialiteForm.value);
  }

}
```

Validation

"Validators"

Les constructeurs des "controls" (FormControl, FormGroup et FormArray) acceptent en second paramètre une liste de fonctions de validation appelées "validators".

Les "validators" natifs d'Angular sont regroupés sous forme de méthodes statiques dans la classe Validators.

```
constructor(private _formBuilder: FormBuilder) {
  this.specialiteForm = this._formBuilder.group({
    title: this._formBuilder.control(null, [Validators.required]),
    description: null
  });
}
```

Les "controls" disposent d'une série de propriétés et de méthodes permettant d'en vérifier l'état :
valid : Valeur booléenne indiquant si le "control" est valide. Dans le cas d'un FormGroup, le "control" est valide si les "controls" qui le composent sont tous valides.
errors : "plain object" combinant les erreurs de tous les validateurs. Vaut null si le "control" est valide.
touched : Valeur booléenne positionnée à true dès le déclenchement de l'événement blur (i.e. l'utilisateur change de "focus").
pristine : Valeur booléenne indiquant si le "control" a été modifié.

Validation

"Exemple de désactivation du "submit""

```
<button [disabled]="!specialiteForm.valid" type="submit">SUBMIT</button>
```

hasError & getError

Les méthodes hasError et getError sont deux méthodes "helpers" permettant d'accéder plus facilement aux informations d'erreur d'un "control".

```
<div *ngIf="shouldShowTitleRequiredError()">Title is required.</div>
```

```
shouldShowTitleRequiredError() {

  const title = this.specialiteForm.controls.title;

  return title.touched && title.hasError('required');

}
```

"Validator" personnalisé

Un "validator" est une fonction qui est appelée à chaque changement de la valeur du "control" afin d'en vérifier la validité. Si la valeur est valide, le "control" retourne null ou un objet d'erreur dans le cas contraire.

```
ValidatorDateDebClotureMin(): ValidatorFn {

  return (control: AbstractControl): { [key: string]: boolean } | null => {

    let currentDateTime = new Date(this.specialiteForm.controls.title.value)
    currentDateTime.setHours(0, 0, 0, 0);

    let controlValue = new Date(control.value);
    controlValue.setHours(0, 0, 0, 0);

    console.log(control.value)
    if (controlValue.getTime() > currentDateTime.getTime()) {
      return { 'ValidatorDateDebClotureMin': true };
    }

    return null;

  };
}
```

```
title: this._formBuilder.control(null, [Validators.required, this.ValidatorDateDebClotureMin()]),
```

Plugins natifs

OCTOBRE 2020

Plugins natifs

En plus de composants purement visuel qui vous permettent de mettre en forme votre application, Ionic propose également des plugins natifs cette fois-ci pour interagir avec les fonctions dites natives de votre téléphone mobile. Ces plugins Ionic sont en réalité des wrappers des plugins **Cordova/Phonegap**.

La liste de tous les plugins Ionic disponibles actuellement se trouve à l'adresse :
https://ionicframework.com/docs/native/

Avant d'installer un plugin, il faut s'assurer que le package Ionic native est bien disponible, ce qui devrait être le cas dans une installation Ionic classique. Mais au besoin, il suffit de faire :

$ npm install @ionic-native/core –save

C'est à l'intérieur de ce package (dossier) que seront installés les autres plugins Ionic.

Installation d'un plugin

Les plugins Ionic étant comme nous l'avons vu des versions remastérisées de plugins Cordova/Phonegap, l'installation se fera en deux étapes : installation de la version Ionic et installation de la version Cordova/Phonegap sur laquelle se base la précédente.

Pour installer la version "Ionic" du plugin il suffit de lancer une commande similaire à celle-ci :

$ npm install @ionic-native/mon_plugin_ionic --save

Puis, il ne reste plus qu'à installer le plugin Cordova/Phonegap correspondant :

$ ionic cordova plugin add mon_plugin_cordova

Chaque plugin possédant sa propre documentation il est recommandé de suivre les instructions d'installation à partir de la documentation de chaque plugin, car certains plugins nécessitent des étapes supplémentaires pour une installation complète.

Une fois le plugin installé, il faut le déclarer dans le module principal de l'application dans la section **providers**:

src/app/app.module.ts.

Utilisation de quelques plugins

Ionic proposant un nombre assez vaste de composant, je vous propose de n'en étudier que quelques-uns parmi les plus intéressants.

- **Camera**
- **Geolocation**
- **Network**
- **Device**

Camera

Documentation : https://ionicframework.com/docs/native/camera/

Ce plugin permet de prendre une photo ou d'enregistrer une vidéo en utilisant l'objet navigator.camera introduit avec l'arrivée du HTML5.

Installation
Pour l'installation, il suffit simplement de lancer les commandes suivantes :

$ ionic cordova plugin add cordova-plugin-camera
$ npm install --save @ionic-native/camera

On le déclare ensuite dans NgModule :

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

```
providers: [
  StatusBar,
  SplashScreen,
  Camera,
```