



ANGULAR

APPROFONDISSEMENT

Godwin AVODAGBE

SOMMAIRE

- 1** **Routing**
- 2** **Forms**
- 3** **Observable**
- 4** **Lifecycle**
- 5** **Production**
- 6** **Projet de Fin**

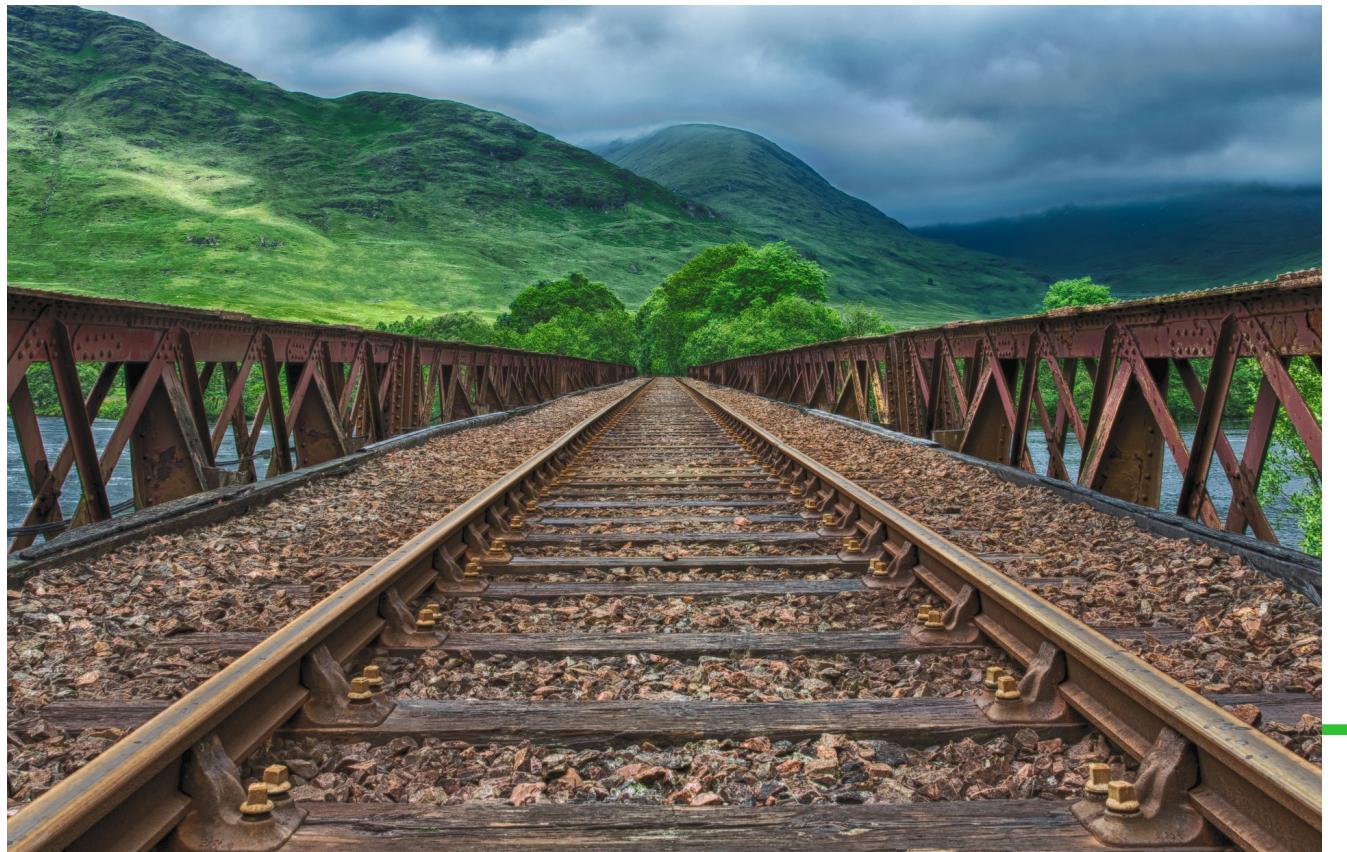


Afin de **permettre aux utilisateurs de garder leurs habitudes de navigation** en visitant une Single Page Application ou une Progressive Web Application, il est nécessaire d'utiliser un système de « Routing ».

Grâce au système de "Routing", les utilisateurs peuvent :

- utiliser l'historique de leur navigateur (e.g. les boutons *back* et *next*),
- partager des liens,
- ajouter une vue à leurs favoris,
- ouvrir une vue dans une nouvelle fenêtre via le menu contextuel,
- ...

Angular fournit nativement un module de "Routing" pour répondre à ce besoin.



ROUTING

Mise en place

Rappel
Configuration des routes

Lazy Loading

Route Guards

MISE EN PLACE

1. Tag

Avant toute chose, il est nécessaire d'ajouter le tage **base** au **head** du fichier **index.html** de l'application :

```
<!doctype html>
<html>
  <head>
    <b><base href="/"></b>
    ...
  </head>
  <body>
    ...
  </body>
</html>
```

MISE EN PLACE

1. Tag

Ce tag indique la base du « **path** » partir de laquelle le "Routing" Angular rentre en jeu.

Cette valeur est généralement personnalisée dans le cas où plusieurs applications Angular sont hébergées sur un **même nom de domaine mais avec des paths différents**.

MISE EN PLACE

2. Configuration de l'appmodule

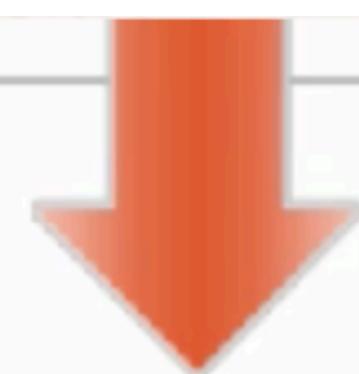
app.module.ts

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

@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    HttpClientModule,
    RouterModule.forRoot([])
  ],
  declarations: [
    ...
  ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

MISE EN PLACE

2. Configuration de l'appmodule

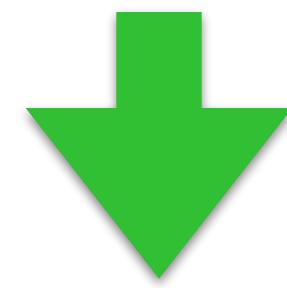
```
[  
  { path: 'products', component: ProductListComponent },  
  { path: 'products/:id', component: ProductDetailComponent },  
  { path: 'welcome', component: WelcomeComponent },  
  { path: '', redirectTo: 'welcome', pathMatch: 'full' },  
  { path: '**', component: PageNotFoundComponent }]
```

MISE EN PLACE

2. Configuration de l'appmodule

PS : Si vous acceptez la génération du routing à la création du nouveau projet Angular, vous aurez le routing défini comme ci-dessous :

```
[MBP-de-Ekoura:JS_ANGULAR trainingekoura$ ng new TrainingAngulars  
? Would you like to add Angular routing? (y/N) █
```



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

const routes: Routes = [];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

MISE EN PLACE

2. Configuration de l'appmodule

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

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

MISE EN PLACE

3. Configuration des liens

En utilisant des liens natifs ``, le navigateur va lancer une requête HTTP de type « GET » ce qui aura pour conséquence de recharger l'application.

Pour éviter ce problème, le module de "Routing" Angular fournit la directive « `routerLink` » qui permet d'intercepter l'évènement « `click` » sur les liens et de changer de "route" sans recharger toute l'application.

```
app.component.ts
...
@Component({
  selector: 'pm-root',
  template: `
    <ul class='nav navbar-nav'>
      <li><a [routerLink]=[" '/welcome' "]>Home</a></li>
      <li><a [routerLink]=[" '/products' "]>Product List</a></li>
    </ul>
  `
})

```

MISE EN PLACE

4. <router-outlet></router-outlet>

La configuration du "Routing" permet de définir quel composant afficher en fonction de la route mais cela n'indique pas à Angular où **injecter le composant** dans la page.

Pour indiquer l'emplacement d'insertion du composant, il faut utiliser la directive `<router-outlet>` directement dans le "root component"

```
app.component.ts
...
@Component({
  selector: 'pm-root',
  template:
    <ul class='nav navbar-nav'>
      <li><a [routerLink]=["['/welcome']">Home</a></li>
      <li><a [routerLink]=["['/products']">Product List</a></li>
    </ul>
    <router-outlet></router-outlet>
})
```

MISE EN PLACE

4. <router-outlet></router-outlet>

En fonction de la "route" visitée, le **composant associé sera alors injecté en dessous du tag router-outlet** (et non à l'intérieur ou à la place du tag contrairement à ce que l'on pourrait supposer).

MISE EN PLACE

5. Construction Dynamique

La "route" peut être construite dynamiquement et passée à l'`Input` `routerLink`.

```
this.route = '/search';
this.routeName = 'Search';
```

```
<a [routerLink]="route">{{ routeName }}</a>
```

MISE EN PLACE

6. Passage des paramètres

app.module.ts

```
@NgModule({
  imports: [
    ...,
    RouterModule.forRoot([
      { path: 'products', component: ProductListComponent },
      { path: 'products/:id', component: ProductDetailComponent },
      { path: 'welcome', component: WelcomeComponent },
      { path: '', redirectTo: 'welcome', pathMatch: 'full' },
      { path: '**', redirectTo: 'welcome', pathMatch: 'full' }
    ])
  ],
  declarations: [...],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

MISE EN PLACE

6. Passage des paramètres

product-list.component.html

```
<td>
  <a [routerLink]=["/products", product.productId]>
    {{product.productName}}
  </a>
</td>
```

app.module.ts

```
{ path: 'products/:id', component: ProductDetailComponent }
```

Il est également possible de passer des paramètres optionnels via l'input « **queryParams** »

```
<a
  routerLink="/search"
  [queryParams]={{keywords: 'Scrum Master'}}>
  Scrum Master
</a>
```

MISE EN PLACE

6. Passage des paramètres (optionnels)

Il est possible de passer plusieurs paramètres. Tous les paramètres non déclarés dans les accolades sont considérés optionnels.

```
<a [routerLink]="['./WishDetail', {wishId: 1234, edit: true}]>EDIT</a>
```

L'URL générée sera « /wishes;wishId=1234;edit=true ». Il est possible de définir plusieurs paramètre obligatoires :

```
...
path: '/wishes/:wishId/:edit'
...
```

MISE EN PLACE

7. Accès aux Paramètres

Le service **ActivatedRoute** décrit l'état actuel du router. Il permet au composant associé à la route de récupérer les paramètres via les propriétés **paramMap** et **queryParamMap**.

Les propriétés paramMap et queryParamMap sont des Observables, pour des raisons d'optimisation, quand vous naviguez vers la même route mais avec des paramètres différents, Angular ne recharge pas le composant mais propage les nouveaux paramètres via ces Observables

product-detail.component.ts

```
import { ActivatedRoute } from '@angular/router';

constructor(private _route: ActivatedRoute) {
  console.log(this._route.snapshot.paramMap.get('id'));
}
```

MISE EN PLACE

8. Navigation à partir du code

product-detail.component.ts

```
import { Router} from '@angular/router';
...
constructor(private _router: Router) { }

onBack(): void {
  this._router.navigate(['/products']);
}
```

MISE EN PLACE

8. Navigation à partir du code et passage de paramètres obligatoire

```
goPrevious(){
    //this.selectedId = parseInt(this._route.snapshot.paramMap.get('id')) - 1 ;
    let prev = this.selectedId - 1 ;
    this.router.navigate(['/catalogue', prev])
}

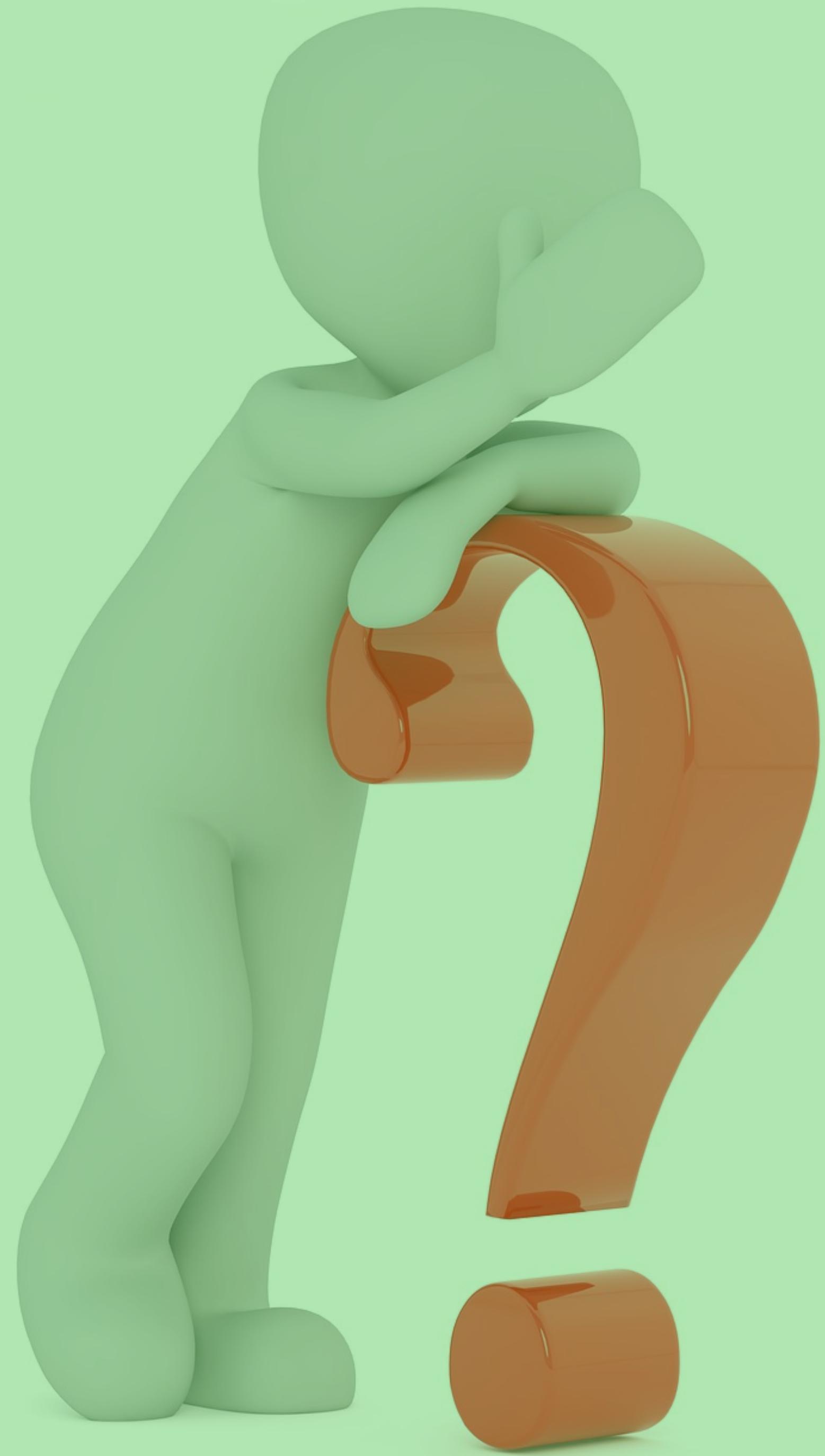
goNext(){
    //this.selectedId = parseInt(this._route.snapshot.paramMap.get('id')) + 1 ;
    let nextv = this.selectedId + 1 ;
    this.router.navigate(['/catalogue', nextv, 56,86,89])
}
```

MISE EN PLACE

8. Navigation à partir du code et passage de paramètres optionnels

```
goPrevious(){
    //this.selectedId = parseInt(this._route.snapshot.paramMap.get('id')) - 1 ;
    let prev = this.selectedId - 1 ;
    this.router.navigate(['/catalogue', prev])
}

goNext(){
    //this.selectedId = parseInt(this._route.snapshot.paramMap.get('id')) + 1 ;
    let nextv = this.selectedId + 1 ;
    this.router.navigate(['/catalogue', nextv, { test : 'Aéroport', horaire : '12h50' }])
}
```



TP

- 1- Créer la page de détail d'un produit**
- 2 - Implémenter les boutons « Précédent » « Suivant » « Back »**

LAZY LOADING

En configurant l'intégralité du Routing de l'application dans le module AppRoutingModule, on serait amené à importer tous les modules de l'application avant son démarrage.

A titre d'exemple, plus l'application sera riche, plus la page d'accueil sera lente à charger par effet de bord.

Pour éviter ces problèmes de "**scalability**", Angular permet de charger les modules à la demande (i.e. : "**Lazy Loading**") afin de ne pas gêner le chargement initial de l'application.

LAZY LOADING

1. Configuration

Création d'un module de routing (**CatalogueRoutingModule**) qui va être contenir tous les liens rattachés à la fonctionnalité du catalogue :

```
export const catalogueRouteList: Routes = [
  { path: 'catalogue', component: CatalogueComponent },
  { path: 'catalogue/:id', component: ProductDetailComponent },
  {
    path: '**',
    redirectTo: 'catalogue'
  }
];

@NgModule({
  imports: [
    RouterModule.forChild(catalogueRouteList),
    CommonModule,
    FormsModule
  ],
  declarations: [
    CatalogueComponent,
    ProductDetailComponent,
    CustomStringPipe,
    RatingComponent
  ]
})
export class CatalogueRoutingModule {}
```

LAZY LOADING

1. Configuration

Modification du fichier **appmodule.ts** de sorte à prendre en compte notre nouvelle configuration des routes :

```
RouterModule.forRoot([
  { path: 'home', component: HomeComponent },
  {
    path : 'catalogue' ,
    loadChildren : './site/catalogue/catalogue-routing.module#CatalogueRoutingModule'
    //loadChildren : () => import('./site/catalogue/catalogue-routing.module')
    //      .then(module => module.CatalogueRoutingModule)
  },
  { path: 'contact', component: ContactComponent },
  { path: 'connexion', component: ConnexionComponent },
  { path: 'inscription', component: InscriptionComponent },
  { path: '', redirectTo: 'home', pathMatch: 'full'},
  { path: '**', redirectTo: 'home', pathMatch: 'full'}
  //{ path: 'catalogue', component: CatalogueComponent },
  //{ path: 'catalogue/:id', component: ProductDetailComponent },
])
)
```

LAZY LOADING

1. Configuration

Modification du fichier **catalogue.component.html** de sorte à prendre en compte notre nouvelle configuration des routes :

```
<a [routerLink]="'./catalogue', product.productId">
|   {{ product.productName }}
</a>
```

Les "routes" doivent être préfixées par "/", "./" ou "../".

- / : Racine de l'application.
- ./ : "Routes" relatives à la "route" actuelle.
- ../ : "Routes" relatives à la "parent route".

LAZY LOADING

2. **forRoot** vs **forChild**

Seul le module **AppModule** (dans certaines application le **AppRoutingModule**) importe le module **RouterModule** avec la méthode statique **forRoot** afin de définir le "Routing" racine et la configuration du router via le second paramètre.

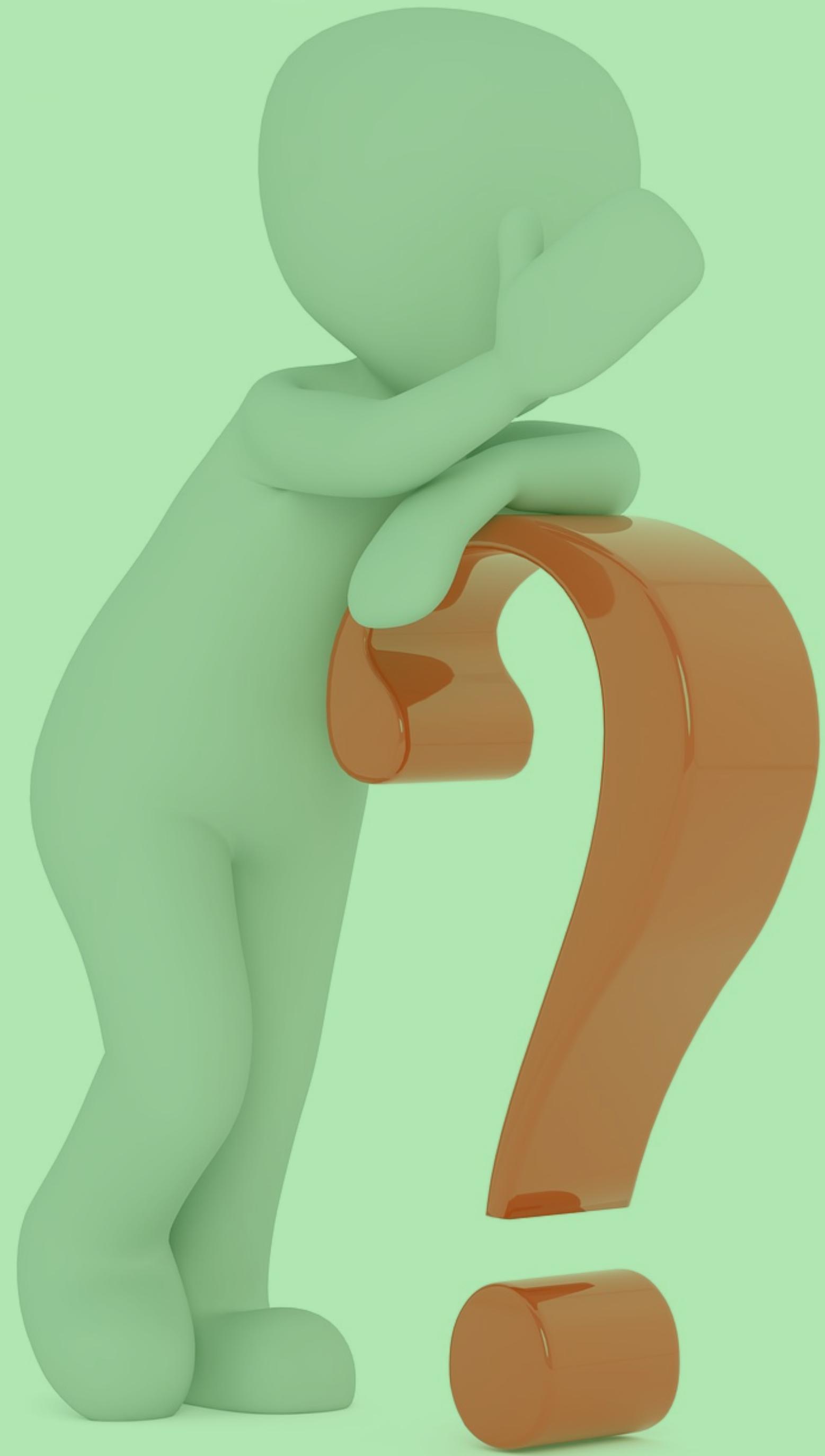
Les "**Child Routing Modules**" importent le **RouterModule** avec la méthode **forChild**.

LAZY LOADING

3. Preloading Strategy

Une fois l'application démarrée et pour éviter la latence de chargement des "**Lazy Loaded Routes**", il est possible de configurer le "Routing" pour précharger tous les modules "**Lazy Loaded**" juste après le démarrage de l'application.

```
RouterModule.forRoot(appRouteList, {  
  preloadingStrategy: PreloadAllModules  
})
```



TP

-
- 1- Réorganiser les différentes routes du TP précédent**
 - 2 - Implémenter la création d'un produit**

ROUTE GUARDS

Les "**Guards**" permettent de contrôler l'accès à une "route" (e.g. **autorisation**) ou le départ depuis une "route" (e.g. *enregistrement ou publication obligatoire avant départ*).

Attention : Les "Guards" ne doivent en aucun cas être considérés comme un mécanisme de sécurité.
La gestion de permission des accès aux ressources doit se faire au niveau des APIs HTTP.

Les "Guards" servent à améliorer la "User eXperience" en évitant par exemple l'accès à des "routes" qui ne fonctionneraient pas car l'accès aux données serait rejeté par l'API.

ROUTE GUARDS

1. Configuration

Les "Guards" sont ajoutés au niveau de la configuration du "Routing" :

- Crédation d'un guard => ng g guard guard-name

```
import { Injectable } from '@angular/core';
import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot } from '@angular/router';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class ProductGuard implements CanActivate {
  canActivate(
    next: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean {
    return true;
  }
}
```

ROUTE GUARDS

1. Configuration

```
RouterModule.forRoot([
  { path: 'home', component: HomeComponent },
  {
    path : 'catalogue',
    canActivate : [ProductGuard],
    loadChildren : './site/catalogue/catalogue-routing.module#CatalogueRoutingModule',
    //loadChildren : () => import('./site/catalogue/catalogue-routing.module')
    //      .then(module => module.CatalogueRoutingModule)
  },
])
```

ROUTE GUARDS

1. CanActivate

Une "Guard" d'activation est un service qui implémente l'interface **CanActivate**.

Ce service doit donc implémenter la méthode **canActivate**.

Cette méthode est appelée à chaque demande d'accès à la "route" ; elle doit alors retourner une valeur de type boolean ou Promise<boolean> ou Observable<boolean> indiquant si l'accès à la "route" est autorisé ou non.

Il est donc possible d'attendre le résultat d'un traitement **asynchrone** pour décider d'autoriser l'accès ou non.

ROUTE GUARDS

1. CanActivate

```
import { Injectable } from '@angular/core';
import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot } from '@angular/router';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class ProductGuard implements CanActivate {
  canActivate(
    next: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean {
    return true;
  }
}
```

ROUTE GUARDS

1. CanActivate

- ⓘ En cas de refus d'accès, il est possible de **rediriger l'utilisateur vers une autre "route"** "manuellement" en injectant le service "Router" par exemple :

```
1  constructor(private _router: Router,  
2               private _session: Session) {  
3  
4  
5      canActivate(route: ActivatedRouteSnapshot, state: RouterState  
6  
7          const isSignedIn = this._session.isSignedIn();  
8  
9          if (isSignedIn !== true) {  
10              this._router.navigate([...]);  
11          }  
12  
13          return isSignedIn;  
14  
15      }
```

ROUTE GUARDS

1. CanDeactivate

Les "Guards" de désactivation sont couplées aux composants car elles doivent communiquer avec le composant pour établir leur décision d'accès.

Une "Guard" de désactivation est un service qui implémente l'interface **CanDeactivate**.

Ce service doit donc implémenter la méthode **canDeactivate**.

Cette méthode est appelée à chaque fois que l'utilisateur souhaite quitter la route (clic sur un lien ou déclenchement automatique) ; elle doit alors retourner une valeur de type boolean ou Promise<boolean> ou Observable<boolean> indiquant si l'accès à la "route" est autorisé ou non.

Contrairement au "Guards" d'activation, cette "Guard" prend en premier paramètre l'instance du composant. C'est pour cette raison que l'interface **CanDeactivate** est générique.

ROUTE GUARDS

1. CanDeactivate

```
export interface IsDirty {
    isDirty(): boolean | Observable<boolean>;
}

@Injectable({
    providedIn: 'root'
})
export class IsNotDirtyGuard implements CanDeactivate<IsDirty> {

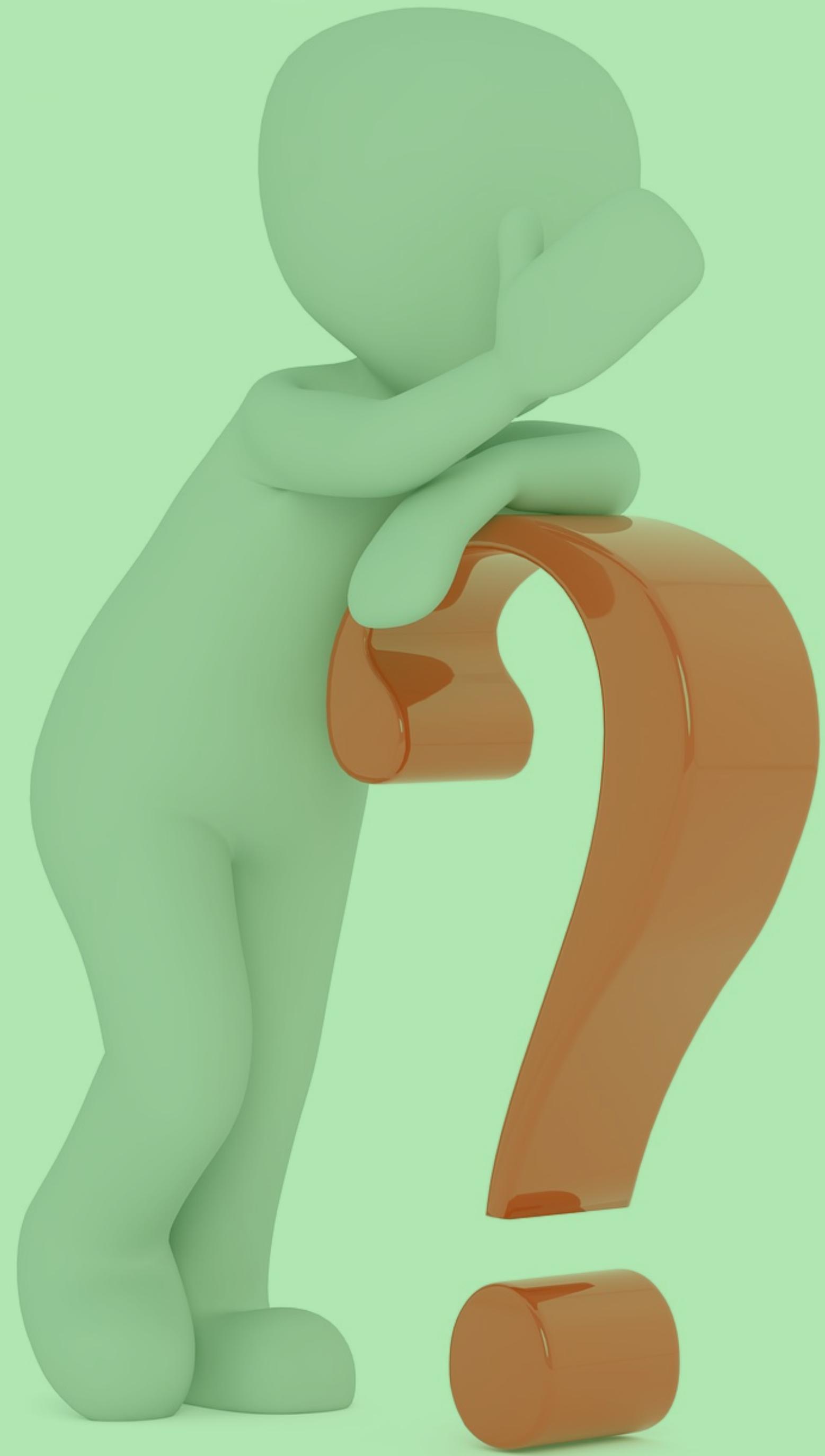
    canDeactivate(component: IsDirty,
                  currentRoute: ActivatedRouteSnapshot,
                  currentState: RouterStateSnapshot,
                  nextState?: RouterStateSnapshot): Observable<bo
        return component.isDirty();
    }

}
```

ROUTE GUARDS

1. CanDeactivate

```
export class ProfileViewComponent implements IsDirty {  
  
    isDirty() {  
        return false;  
    }  
  
}
```



TP

Dans le mécanisme d'ajout d'un produit, ne pas pouvoir changer de lien tant que l'enregistrement ne s'est pas fait.



ROUTING

Pour aller plus loin, il faut voir :

- Navigation au sein d'un même composant
- ServiceResolver
- RouterEvents
- Animation

SOMMAIRE

- 1** **Routing**
- 2** **Forms**
- 3** **Observable**
- 4** **Lifecycle**
- 5** **Production**
- 6** **Projet de Fin**



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

Ce cours 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...



FORMULAIRE

1- Template-driven Forms

2- Reactive Forms

TEMPLATE-DRIVEN FORMS

1. 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**".

```
<form>
  <input
    name="title"
    type="text"
    [(ngModel)]="userTitle">
  <input
    name="title"
    type="text"
    [(ngModel)]="userTitle">
  <div>{{ userTitle }}</div>
</form>
```

TEMPLATE-DRIVEN FORMS

2. Détection du "submit" avec **ngSubmit**

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

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

TEMPLATE-DRIVEN FORMS

2. Détection du "submit" avec **ngSubmit**

Attention : Préférez le "listener" de submit (via l'Output **ngSubmit**) sur le formulaire plutôt que le "listener" de click sur les boutons.

Bien qu'il soit possible d'utiliser l'Event Binding sur l'événement **submit** natif du formulaire, il est fortement recommandé d'utiliser l'Output **ngSubmit** ajouté par le module **FormsModule**, de sorte à éviter le comportement de base des formulaires web qui recharge la page.

REACTIVE FORMS

1. Avantages

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** ».

REACTIVE FORMS

2. FormControl

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

```
import { FormControl } from "@angular/forms";

export class UserFormComponent {

  userTitleControl = new FormControl();
  userDescriptionControl = new FormControl();

  submitUser() {
    console.log(this.userTitleControl.value);
    console.log(this.userDescriptionControl.value);
  }
}
```

REACTIVE FORMS

2. FormControl

```
<input  
  type="text"  
  [FormControl]="userTitleControl">  
  
<textarea [FormControl]="userDescriptionControl"></textarea>  
  
<button  
  type="submit"  
  (click)="submitUser()">SUBMIT</button>
```

Si nous testons tel quel, il va y avoir une erreur. Il nous faut importer le module « **ReactiveFormsModule** » dans le appmodule.ts

REACTIVE FORMS

2. FormControl

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 ».

REACTIVE FORMS

3. 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.

```
import { FormGroup, FormControl } from "@angular/forms";

export class UserFormComponent {
  userForm = new FormGroup({
    title: new FormControl(),
    description: new FormControl()
  );
  submitUser() {
    console.log(this.userForm.value);
  }
}
```

REACTIVE FORMS

3. FormGroup

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

L'objet **this.userForm.value** présente l'ensemble des valeurs des controls via un "plain objet" : **{title: 'TITLE', description: null}**

REACTIVE FORMS

3. 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]="userForm"  
  (ngSubmit)="submitUser()">  
  
<input  
  type="text" class="form-control"  
  formControlName="title">  
  
<textarea formControlName="description" class="form-control" ></textarea>  
  
<button class="btn btn-success" type="submit" >SUBMIT</button>  
  
</form>
```

REACTIVE FORMS

3. FormGroup

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**.

Un FormGroup est donc également un "control" et il est donc possible de construire une arborescence de **FormGroups** afin de regrouper différentes parties d'un formulaire de taille importante ou dont des parties sont réutilisables dans d'autres contextes.

```
new FormGroup({  
    address: new FormGroup({  
        street: new FormControl()  
    })  
});
```

REACTIVE FORMS

4. FormArray

FormArray héritant également de la classe abstraite AbstractControl, permet de créer un "control" contenant une liste de "controls" (e.g. FormControl ou FormGroup) ordonnés (contrairement à FormGroup qui permet de créer un groupe de "controls" accessibles avec des clés sur un "plain object").

La valeur contenu dans le FormArray est de type Array.

REACTIVE FORMS

5. FormBuilder

Le module **ReactiveFormsModule** implémente également un service **FormBuilder** qui permet de simplifier la création des FormGroup, FormArray ou FormControl.

```
bookForm: FormGroup;

constructor(private _formBuilder: FormBuilder) {
  this.bookForm = this._formBuilder.group({
    title: null,
    description: null
  });
}
```

REACTIVE FORMS

6. Validator

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.

```
import { FormGroup, FormControl, Validators } from "@angular/forms";
export class UserFormComponent {
  userForm = new FormGroup({
    title: new FormControl(null, [
      Validators.required
    ]),
    description: new FormControl()
  );
  submitUser() {
    console.log(this.userForm.value);
  }
}
```

REACTIVE FORMS

6. Validator

Attention : Remarquez que l'ajout du "validator" ne change pas le comportement du composant : la méthode **submitBook** continue à être appelée bien que la contrainte de validation ne soit pas respectée.

C'est au composant de décider de l'action à mener en fonction de l'état des « controls ».

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** ou **FormArray**, 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.

Exemple de validation :

```
<button [disabled]="!userForm.valid" type="submit" class="btn btn-success">SUBMIT</button>
```

REACTIVE FORMS

7. Validator (hasError & getError)

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

Code HTML

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

Code TS

```
shouldShowTitleRequiredError() {  
  const title = this.userForm.controls.title;  
  return title.touched && title.hasError('required');  
}
```

REACTIVE FORMS

8. Validator « personnalisé »

A "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.

Code TS : valid-user-title.validator.ts

```
import { ValidatorFn } from '@angular/forms';
export const validUserTitle: ValidatorFn = (control) => {

  /* Is not valid. */
  if (/learn .*? in one day/i.test(control.value)) {
    return {
      'validUserTitle': {
        reason: 'blacklisted',
        value: control.value
      }
    };
  }

  /* Is valid. */
  return null;
};
```

REACTIVE FORMS

8. Validator « personnalisé »

Code TS : Component d'utilisation du formulaire

```
userForm = new FormGroup({
  title: new FormControl(null, [
    Validators.required,
    validUserTitle
  ]),
  description: new FormControl()
});
```

Les informations d'erreur (reason et value) sont alors accessibles grâce à la méthode getError.

```
const error = this.userForm.getError('validUserTitle', ["title"]);
if (error != null) {
  console.log(error.reason);
  console.log(error.value);
}
```

REACTIVE FORMS

9. Validator « paramétré »

Tels que le "validator" **minLength**, certains "validators" ont besoin de paramètres pour personnaliser leur comportement.

Dans ce cas, il suffit d'implémenter une "factory" de "validators" (i.e. : une fonction qui retourne des fonctions de type « validator»).

Code TS : valid-pattern.validator.ts

```
export type ValidPattern = (args: {blacklistedPatternList: RegExp[]}) => ValidatorFn;
export const validPattern: ValidPattern = ({blacklistedPatternList}) => control => {
  for (const blacklistedPattern of blacklistedPatternList) {
    const result = blacklistedPattern.exec(control.value);
    if (result != null) {
      return {
        'validPattern': {
          reason: 'blacklisted',
          match: result[0],
          value: control.value
        }
      };
    }
  }
  return null;
};
```

REACTIVE FORMS

9. Validator « paramétré »

Le validator est alors personnalisé à l'utilisation

Code TS : Dans le component

```
userForm = new FormGroup({
  title: new FormControl(null, [
    Validators.required,
    validUserTitle,
    validPattern({
      blacklistedPatternList: [
        /^learn .*? in one day/i,
        /hero/i,
        /ninja/i
      ]
    })
  ]),
  description: new FormControl()
});
```

REACTIVE FORMS

9. Validator « paramétré »

Même l'erreur contient alors des informations personnalisées :

```
1 const error = bookForm.getError('validPattern', 'title');
2 if (error != null) {
3     console.log(error.reason); // blacklisted
4     console.log(error.value); // Become an Angular Ninja
5     console.log(error.match); // Ninja
6 }
```

REACTIVE FORMS

10.Async Validators

Il est parfois nécessaire d'implémenter des "validators" **asynchrones** (e.g. vérification distante via une API).

Un "validator" **asynchrone** se comporte de la même façon qu'un "validator" synchrone mais au lieu de retourner **null** ou un objet d'erreur, il doit retourner un **Observable**.

- ✓ Les "validators" asynchrones peuvent être transmis au "control" par paramètre ordonné (*3ème paramètre après la valeur initiale et les validators synchrones*) mais il est préférable d'utiliser un **objet** plus explicite en guise de second paramètre :

```
1 new FormControl(null, {  
2   validators: [Validators.required],  
3   asyncValidators: [myAsyncValidator]  
4 })
```



REACTIVE FORMS

10.Async Validators

Il est également possible de faire de la validation par injection de dépendance.

11.Personnalisation

Afin de faciliter la personnalisation du CSS en fonction de la validité des éléments du formulaire, Angular ajoute automatiquement les classes CSS suivantes en fonction de l'état du "control" :

- .ng-valid
- .ng-invalid
- .ng-pending
- .ng-pristine
- .ng.dirty
- .ng-untouched
- .ng-touched

```
input.ng-touched.ng-invalid {  
    border-width: 1px;  
    border-left: red solid 5px;  
}
```

REACTIVE FORMS

12. Observation des Changements

La propriété `valueChanges` est l'une des propriétés les plus importantes des "controls". Il s'agit d'un **Observable** permettant d'adopter une approche réactive en détectant tous les changements apportés aux données du "control" (le **FormGroup** regroupant tous les "controls" est lui-même un "control").

A titre d'exemple, le code suivant permet de construire un Observable contenant la liste des résultats associés à la recherche de l'utilisateur.

L'opérateur **debounceTime** permet d'attendre 100ms d'inactivité avant de produire une requête.

Grâce à l'opérateur **switchMap**, chaque requête en cours est annulée avant d'en produire une nouvelle.

```
this.bookList$ = this.bookForm.valueChanges
  .pipe(
    debounceTime(100),
    switchMap(value => this._bookRepository.search({keywords: value.title}))
);
```



FORMS

Pour aller plus loin, il faut voir :
- Les styles sur les erreurs



TRAINING

mashroom⁶



THANK YOU

GODWIN AVODAGBE

godwin.avodagbe@ekoura.com

Phone: 0033 6 16 43 70 00

Ekoura, 1 Boulevard Victor 75015 Paris

