

# FRAMEWORK ANGULAR

---

Utilisation de Framework

Notion de composant web

## Concept d'Angular

Component / Template

Data Binding

Component Interaction

Service

Rooting

Forms

# ANGULAR : FORMULAIRE

---

- Outils graphiques créé avec le langage **HTML**.
- Permet à l'utilisateur d'interagir avec les données de l'application (se connecter, entrer des informations dans la base de données, mettre à jour un profil, etc.).

- **Angular facilite la gestion d'un formulaire :**

- La récupération des données saisies,
- La validation et le contrôle des valeurs saisies,
- La gestion d'erreur,
- Et bien d'autres.

# ANGULAR : FORMULAIRE

---

**Angular propose deux approches :**

→ **Template-driven forms**

- Basé sur `FormsModule`.
- Facile à utiliser et conseillé pour les formulaires simples.

→ **Reactive forms**

- Basé sur `ReactiveFormsModule`.
- Robuste et évolutif, conçu pour des applications nécessitant des contrôles particuliers (Form Group et Form Builder).

# ANGULAR : FORMULAIRE

---

Différences principales entre les deux approches :

	REACTIVE	TEMPLATE-DRIVEN
Setup of form model	Explicit, created in component class	Implicit, created by directives
Data model	Structured and immutable	Unstructured and mutable
Data flow	Synchronous	Asynchronous
Form validation	Functions	Directives

237

# ANGULAR : FORMULAIRE

---

Les exemples suivants seront basés sur une saisie de produits.

→ Interface **IProduit** :

```
exempleForm > src > app > TS IProduit.ts > ...
1   export interface IProduit{
2       id: number,
3       name: string,
4       category: string
5   }
```

→ Composant **Products** :

```
exempleForm > src > app > products > <> products.component.html > ...
1   <h2>Liste des produit :</h2>
2   <ul>
3       <li *ngFor="let produit of produits">
4           {{ produit.name }} {{ produit.category }}
5       </li>
6   </ul>
```

```
exempleForm > src > app > products > TS products.component.ts > ...
1   import { Component, OnInit } from '@angular/core';
2   import { IProduit } from '../IProduit';
3
4   @Component({
5       selector: 'app-products',
6       templateUrl: './products.component.html',
7       styleUrls: ['./products.component.css']
8   })
9   export class ProductsComponent implements OnInit {
10
11       produits: IProduit[] = [];
```

# ANGULAR : TEMPLATE-DRIVEN FORMS

---

## Première approche : Template-driven forms

- Utilise la directive **ngModel** pour réaliser une **liaison bidirectionnelle**.
- Cette directive permet de créer et manager une instance de **FormControl** pour un élément donné du formulaire.
- Besoin d'importer **FormsModule** dans `app.module.ts` afin d'utiliser `ngModel` :

```
import { FormsModule } from '@angular/forms';
```

# ANGULAR : TEMPLATE-DRIVEN FORMS

## Création d'un composant :

```
exempleForm > src > app > form-td > TS form-td.component.ts > ...
1  ∨ import { Component, OnInit } from '@angular/core';
2  import { IProduit } from '../IProduit';
3
4  ∨ @Component({
5    selector: 'app-form-td',
6    templateUrl: './form-td.component.html',
7    styleUrls: ['./form-td.component.css']
8  })
9  ∨ export class FormTDComponent implements OnInit {
10
11    categories = ["Légumes", "Fruits", "Viandes"];
12    produit!: IProduit;
13
14    constructor() { }
15
16  ∨ ngOnInit(): void {
17  ∨    this.produit = {
18      id: 0,
19      name: 'Pas de nom',
20      category: this.categories[0]
21    };
22  }
23 }
```

# ANGULAR : TEMPLATE-DRIVEN FORMS

---

## Syntaxe ngModel :

```
<input type="text" name="property" [(ngModel)]="produit.name" >
```

Dans le composant il existe une propriété name à l'objet produit.

- Angular crée des FormControl et les enregistre avec une directive ngForm qu'Angular attache à une balise form.  
Chaque FormControl est enregistré avec le nom de l'input associé (name).

- Modification directe de la propriété name avec la nouvelle valeur saisie.



# ANGULAR : TEMPLATE-DRIVEN FORMS

---

## Exemple :

Un premier formulaire avec :

- un champ texte pour saisir le nom du produit
- un champ select pour sélectionner la catégorie du produit
- un bouton submit

Résultat :

**Template driven forms**  
Name :

Category :

Nom du produit : Pas de nom

Categorie du produit : Légumes

# ANGULAR : TEMPLATE-DRIVEN FORMS

Directive  
ngForm  
attaché à la  
balise form

(utilisation  
template  
reference  
variable)

```
exempleForm > src > app > form-td > <> form-td.component.html > ...
1  <h2>Template driven forms</h2>
2  <form #productFormTD="ngForm">
3    <div class="group">
4      <label for="name">Name : </label>
5      <input type="text" name="name" id="name" [(ngModel)]="produit.name" >
6    </div>
7    <div class="group">
8      <label for="category">Category : </label>
9      <select id="category" name="category" [(ngModel)]="produit.category" >
10       <option *ngFor="let cat of categories" [value]="cat">{{cat}}</option>
11     </select>
12   </div>
13   <button type="submit">Submit</button>
14 </div>
15   <p>Nom du produit : {{produit.name}}</p>
16   <p>Categorie du produit : {{produit.category}}</p>
17 </div>
18 </form>
```

La propriété  
produit.category  
est lié au select  
avec ngModel.

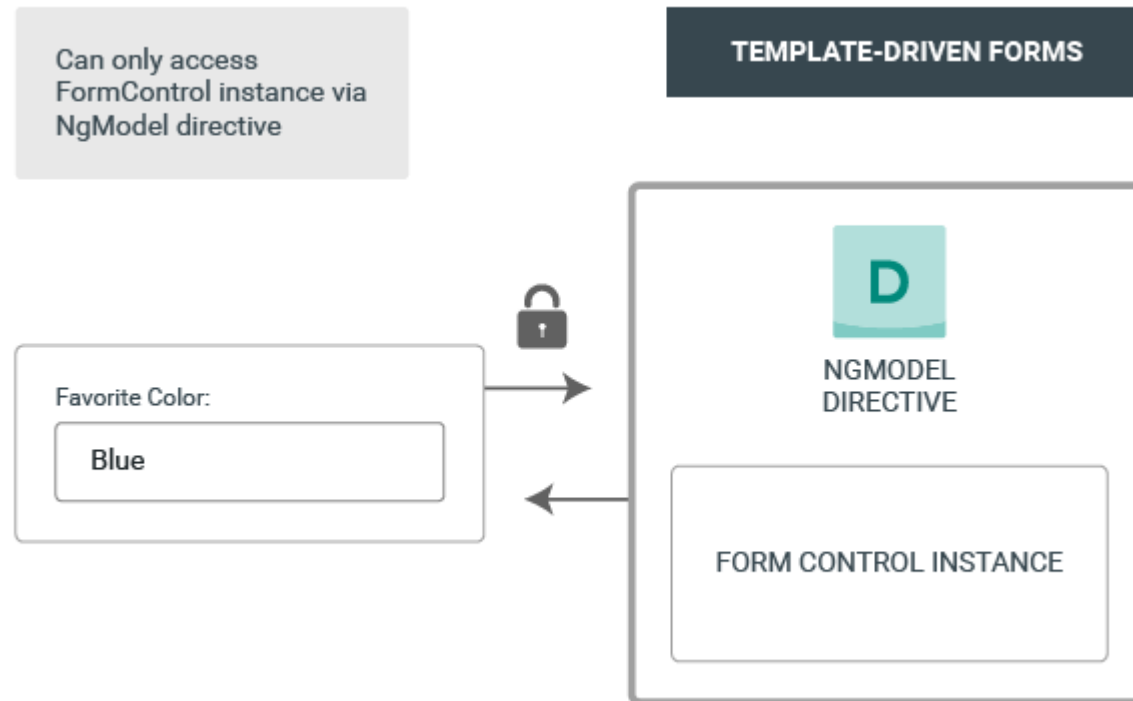
243

Pas besoin de  
cliquer pour  
envoyer la valeur  
saisie dans le  
champ texte.

# ANGULAR : TEMPLATE-DRIVEN FORMS

---

→ Avec ce type d'approche nous n'avons **pas un accès direct** à l'instance du **FormControl**.

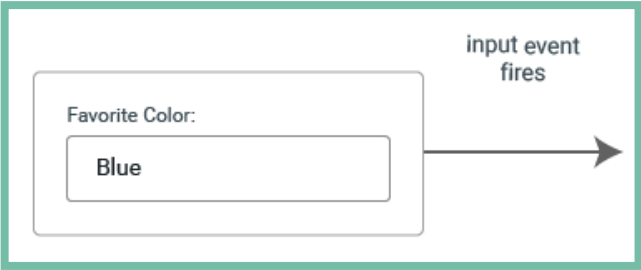


244

# ANGULAR : TEMPLATE-DRIVEN FORMS

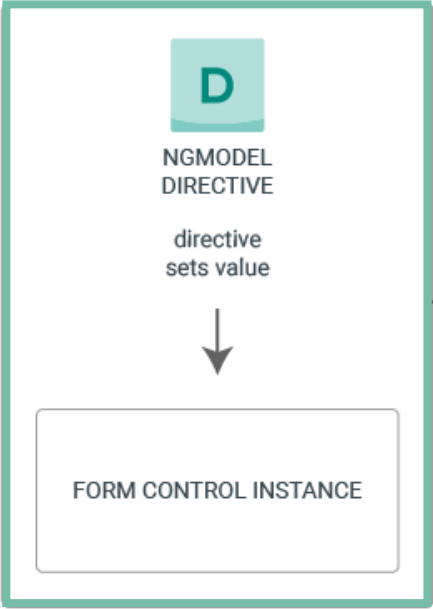
## TEMPLATE-DRIVEN FORMS - DATA FLOW (VIEW TO MODEL)

1. L'élément Input émet un **InputEvent**

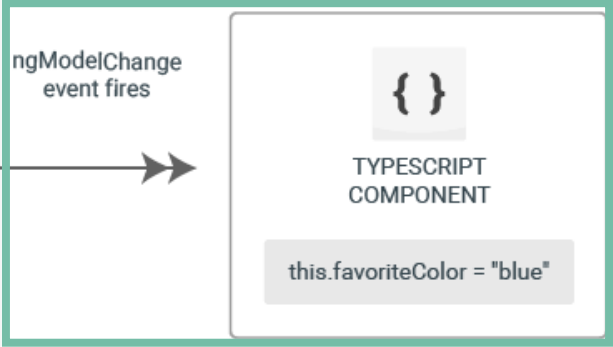


Emission de la nouvelle valeur via l'observable `valueChanges`

2. Déclenchement de la méthode `setValue()` sur l'instance de `FormControl`



3. `ngModel.viewToModelUpdate()` est appelée et émet un `ngModelChange` Event.



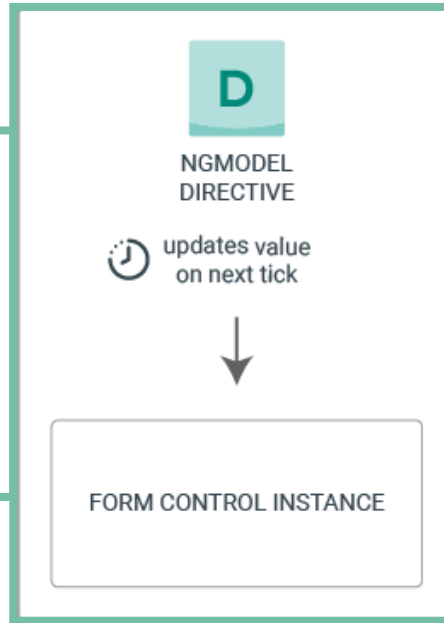
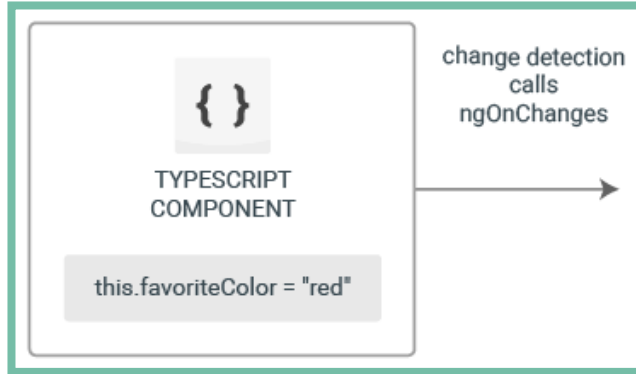
245

START			DIRECTIVE SETS VALUE			END RESULT		
this.favoriteColor (ngModel)	RED	●	this.favoriteColor (ngModel)	RED	●	this.favoriteColor (ngModel)	BLUE	●
FormControl instance value	RED	●	FormControl instance value	BLUE	●	FormControl instance value	BLUE	●
view	BLUE	●	view	BLUE	●	view	BLUE	●

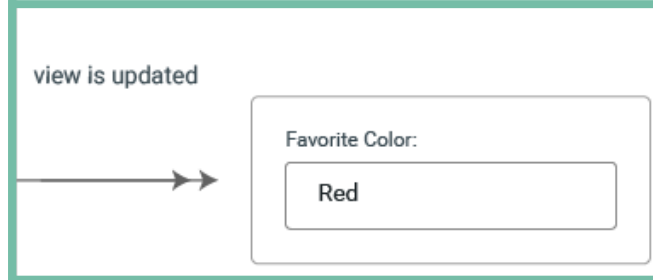
# ANGULAR : TEMPLATE-DRIVEN FORMS

1. La méthode `ngOnChanges` est appelée suite à une modification d'une valeur (directive `ngModel`).

## TEMPLATE-DRIVEN FORMS - DATA FLOW (MODEL TO VIEW)



3. L'instance de `FormControl` émet la valeur modifiée via l'observable `valueChanges`.



Modification de l'Input avec la nouvelle valeur.

Mise dans une file d'attente la demande asynchrone de modification de valeur du `FormControl`.

2. Changement de valeur

START		
this.favoriteColor (ngModel)	RED	●
FormControl instance value	BLUE	●
view	BLUE	●

DIRECTIVE UPDATES VALUE		
this.favoriteColor (ngModel)	RED	●
FormControl instance value	RED	●
view	BLUE	●

END RESULT		
this.favoriteColor (ngModel)	RED	●
FormControl instance value	RED	●
view	RED	●

# ANGULAR : TEMPLATE-DRIVEN FORMS

---

- L'utilisation de la directive `ngModel` permet donc de suivre l'état d'un input à un moment donné.
- Angular propose différentes classes CSS pour gérer les différents états :

State	Class if true	Class if false
The control has been visited.	<code>ng-touched</code>	<code>ng-untouched</code>
The control's value has changed.	<code>ng-dirty</code>	<code>ng-pristine</code>
The control's value is valid.	<code>ng-valid</code>	<code>ng-invalid</code>

# ANGULAR : TEMPLATE-DRIVEN FORMS

Utilisation du couple ng-valid / ng-invalid :

→ CSS :

```
exempleForm > src > app > form-td > # form-td.component.css >
1  .ng-valid[required] {
2    border-left: 5px solid #357939;
3  }
4
5  .ng-invalid:not(form) {
6    border-left: 5px solid #88302e;
7  }
```

→ Modification du formulaire :

```
exempleForm > src > app > form-td > <> form-td.component.html > ...
1  <h2>Template driven forms</h2>
2  <form #productFormTD="ngForm">
3    <div class="group">
4      <label for="name">Name : </label>
5      <input type="text" name="name" id="name" [(ngModel)]="produit.name" required>
6    </div>
```

Name :

Category : Légumes ▾

Name :

Category : Légumes ▾

# ANGULAR : TEMPLATE-DRIVEN FORMS

---

## Soumission du formulaire

- La directive `ngForm` possède une notion de validité du formulaire.
- Il est par exemple possible de désactiver le bouton *submit* tant que le formulaire n'est pas valide :

```
<button type="submit" [disabled]=!productFormTD.valid>Submit</button>
```

## Template driven forms

Name :

Category : Légumes ▼

- Possibilité de créer ses propres validateurs.



# ANGULAR : TEMPLATE-DRIVEN FORMS

---

## Soumission du formulaire

→ La directive `ngSubmit` permet de soumettre le formulaire:  
(event binding)

```
<form #productFormTD="ngForm" (ngSubmit)=ajouterProduit(>
```

```
exempleForm > src > app > form-td > TS form-td.component.ts > ...
```

```
27   ajouterProduit(): void{  
28     this.produits.push(this.produit);  
29     this.produit = { id: 0, name: 'Pas de nom', category: this.categories[0] }  
30   }
```

# ANGULAR : REACTIVE FORM

---

## Deuxième approche : Reactive form

- A l'inverse de la première approche, nous allons pouvoir manipuler les contrôles de chaque entité du formulaire.
- Définition des contrôles directement dans le composant :  
La directive [formControl] va directement lier l'instance de FormControl à un élément de la vue.
- Besoin d'importer le module : **ReactiveFormsModule**

```
import { ReactiveFormsModule } from '@angular/forms';
```

# ANGULAR : REACTIVE FORM

Instance FormControl :

```
exempleForm > src > app > form-r > TS form-r.component.ts > ...
1  import { Component, OnInit } from '@angular/core';
2  import { FormControl } from '@angular/forms';
3
4  @Component({
5    selector: 'app-form-r',
6    templateUrl: './form-r.component.html',
7    styleUrls: ['./form-r.component.css']
8  })
9  export class FormRComponent implements OnInit {
10
11    name: string = 'default'
12    nameControl = new FormControl('default');
13
14    constructor() { }
15
16    ngOnInit(): void {
17    }
18  }
```

# ANGULAR : REACTIVE FORM

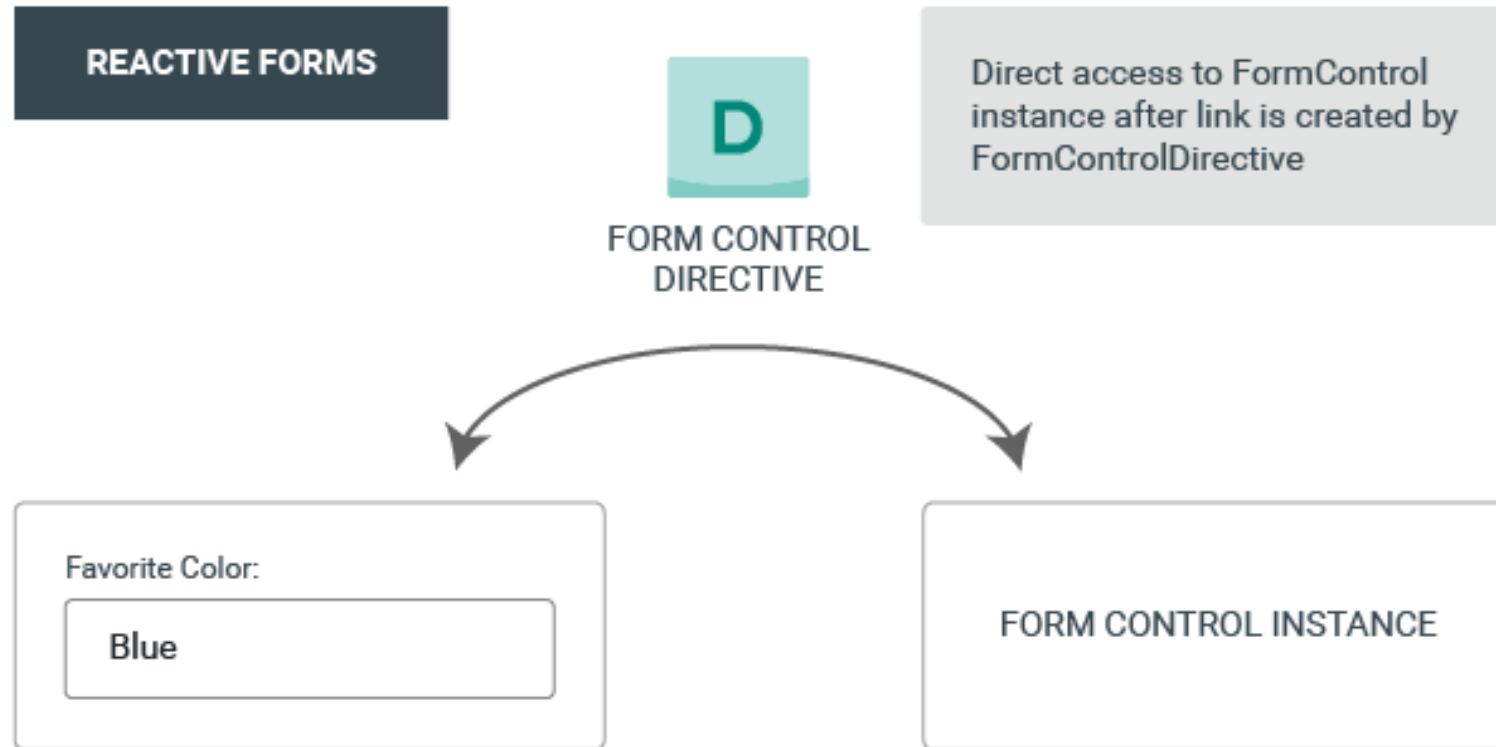
---

Syntaxe de la directive `formControl` :

```
exempleForm > src > app > form-r > <> form-r.component.html > ...
1  <h2>Template reactive form</h2>
2  <form #productFormTD="ngForm">
3      <div class="group">
4          <label for="name">Name : </label>
5          <input type="text" name="name" id="name" [formControl]="nameControl">
6      </div>
7      <div>
8          <p>Nom du produit (control) : {{nameControl.value}}</p>
9          <p>Nom du produit (data) : {{name}}</p>
10     </div>
11 </form>
```

# ANGULAR : REACTIVE FORM

---



254

# ANGULAR : REACTIVE FORM

---

- L'instance de FormControl à accès à la valeur courante de l'input associé.
- **Aucune modification** de la donnée du composant est réalisé directement.
- Les maj de la vue au modèle et inversement du modèle à la vue sont synchrones.

255

## Template reactive form

Name :

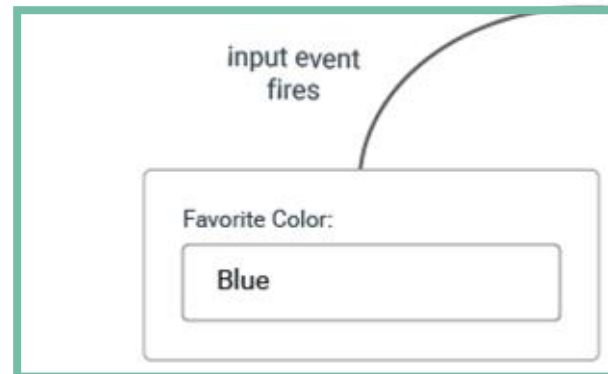
Nom du produit (control) : default

Nom du produit (data) : default

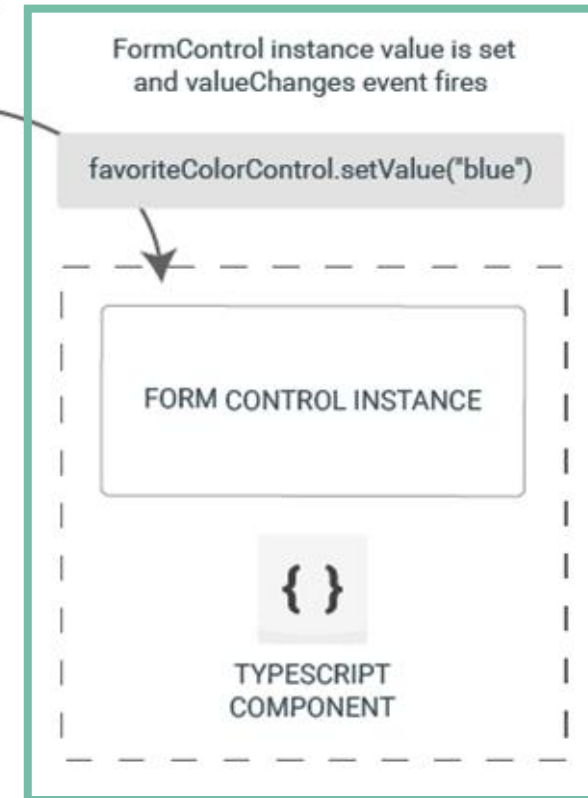
# ANGULAR : REACTIVE FORM

## REACTIVE FORMS - DATA FLOW (VIEW TO MODEL)

1. L'utilisateur saisit la nouvelle valeur.  
L'élément Input émet un `InputEvent`.



2. La nouvelle valeur est immédiatement transmise à l'instance de `FormControl`.



3. L'instance de `FormControl` émet la nouvelle valeur via l'observable `valueChanges`.

# ANGULAR : REACTIVE FORM

## REACTIVE FORMS - DATA FLOW (MODEL TO VIEW)

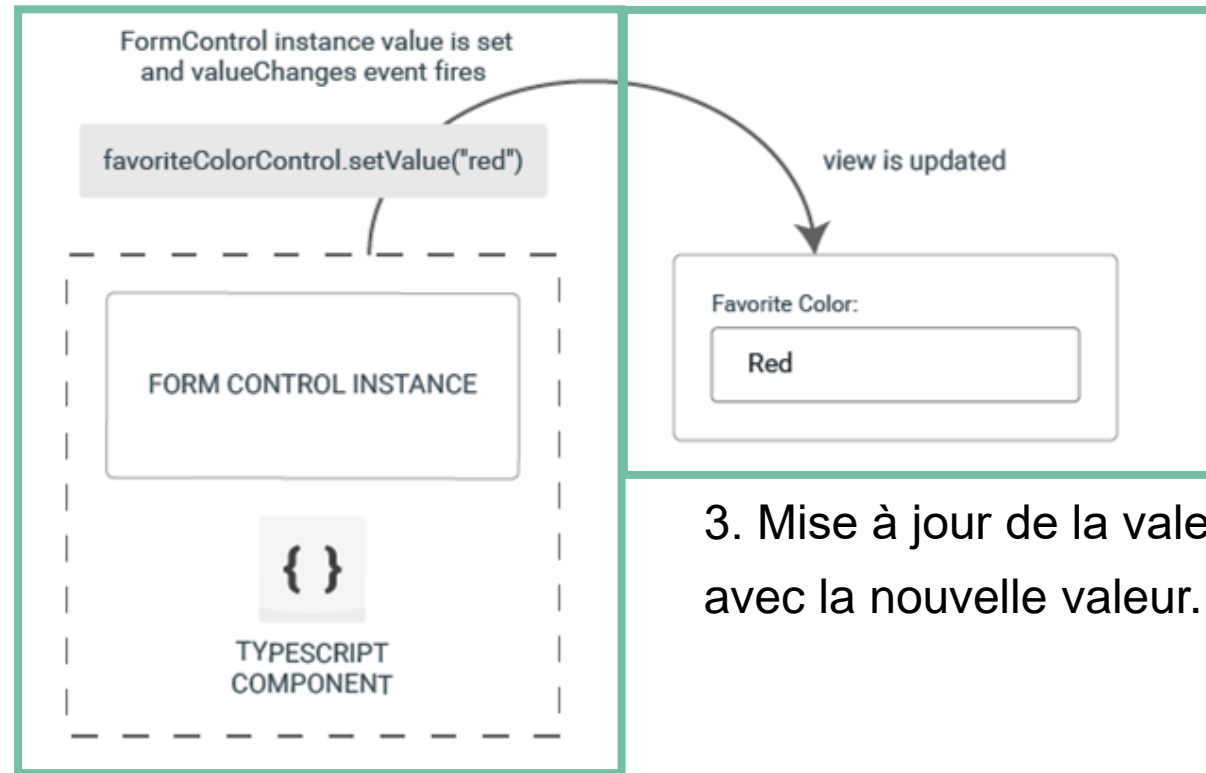
1. L'utilisateur appelle la méthode `setValue` qui met à jour la valeur du `FormControl`.



FORM CONTROL  
DIRECTIVE

2. L'instance de `FormControl` émet la nouvelle valeur avec l'observable `valueChanges`.

→ Une souscription à cet observable permet de recevoir la nouvelle valeur.



257



# ANGULAR : REACTIVE FORM

---

Souscription à l'observable `valueChanges` pour récupérer la nouvelle valeur :

```
exempleForm > src > app > form-r > TS form-r.component.ts > ...  
9   export class FormRComponent implements OnInit {  
10  
11     name: string = 'default'  
12     nameControl = new FormControl('default');  
13  
14     constructor() { }  
15  
16     ngOnInit(): void {  
17         this.nameControl.valueChanges.subscribe( res =>  
18             this.name = res  
19         )  
20     }  
21 }
```

258

## Template reactive form

Name :

Nom du produit (control) : default

Nom du produit (data) : default

# ANGULAR : REACTIVE FORM

---

Possibilité de regrouper plusieurs `FormControl` dans un `FormGroup`.

```
exempleForm > src > app > form-r > TS form-r.component.ts > FormRComponent >  
 9  export class FormRComponent implements OnInit {  
10  
11      name: string = 'default';  
12      category: string = '';  
13      categories = ['', 'Légumes', 'Fruits', 'Viandes'];  
14  
15      //nameControl = new FormControl('default');  
16      produitForm = new FormGroup({  
17          nameControl: new FormControl(''),  
18          categoryControl: new FormControl(this.categories[0]),  
19      });
```

259

→ Peut également imbriquer plusieurs `FromGroup`.

# ANGULAR : REACTIVE FORM

formGroupName=...

Si imbrication d'un autre fromGroup

exempleForm > src > app > form-r > <> form-r.component.html > ...

```
1  <h2>Template reactive form</h2>
2  <form #productFormTD="ngForm" [formGroup]="produitForm">
3    <div class="group">
4      <label for="name">Name : </label>
5      <input type="text" name="name" id="name" formControlName="nameControl">
6    </div>
7    <div class="group">
8      <label for="category">Category : </label>
9      <select id="category" name="category" formControlName="categoryControl" >
10       <option *ngFor="let cat of categories" [value]="cat">{{cat}}</option>
11     </select>
12   </div>
13   <div>
14     <p>Nom du produit : {{name}}</p>
15     <p>Categorie : {{category}}</p>
16   </div>
17 </form>
```

# ANGULAR : REACTIVE FORM

---

## Template reactive form

Name :

Category :

Nom du produit : default

Categorie :

Possibilité également de souscrire à l'observable `valueChange` pour chacun des `FormControl` du groupe :

```
ngOnInit(): void {  
  this.produitForm.get('nameControl')?.valueChanges.subscribe( res =>  
    | this.name = res  
  )  
  this.produitForm.get('categoryControl')?.valueChanges.subscribe( res =>  
    | this.category = res  
  )  
}
```

261

→ Le groupe traque chaque modification de ses contrôles. Lorsqu'un contrôle change, le parent émet également une nouvelle valeur.

# ANGULAR : REACTIVE FORM

---

Comment mettre à jour une valeur du modèle (formGroup) ?

→ Utilisation de setValue() ou de patchValue()

```
updateProduit(): void{  
  this.produitForm.patchValue({  
    nameControl: "Toto"  
  })  
  this.produitForm.setValue({  
    nameControl: "Toto",  
    categoryControl: this.categories[1]  
  })  
}
```

# ANGULAR : REACTIVE FORM

---

## Soumission du formulaire

Soumission du formulaire si celui-ci est valide (par rapport au groupe)

```
<button type="submit" [disabled]="!produitForm.valid">Submit</button>
```

263

Comme pour les *template-driven forms* : utilisation de la directive `ngSubmit`.

```
<form [formGroup]="produitForm" (ngSubmit)="ajouterProduit()">
```

```
ajouterProduit(): void{  
  console.dir(this.produitForm.value);  
  this.produit.name = this.name;  
  this.produit.category = this.category;  
  this.produits.push(this.produit);  
}
```

# ANGULAR : REACTIVE FORM

---

→ Possibilité d'utiliser le service **FormBuilder** pour faciliter la création des contrôles.

```
import { FormBuilder } from '@angular/forms';
```

```
exempleForm > src > app > form-r > TS form-r.component.ts > ...  
11   export class FormRComponent implements OnInit {  
12  
13     produitFormFB = this.fb.group({  
14       nameControl: [''],  
15       categoryControl: ['']  
16     })  
17  
18     constructor(private fb: FormBuilder) { }
```

# ANGULAR : REACTIVE FORM

---

## Fonctions de validation

→ Possibilité d'ajouter directement à un `formControl` des fonctions de validations.

- **Validateurs synchrones** (retourne directement les erreurs ou null, 2<sup>ème</sup> paramètre)
- **Validateurs asynchrones** (retourne un observable qui emit plus tard les erreurs ou null, 3<sup>ème</sup> paramètre)

→ Les validateurs asynchrones sont réalisés que si les validateurs synchrones retournent aucun problème.



# ANGULAR : REACTIVE FORM

---

## Fonctions de validation

→ Angular propose des fonctions de validation : voir la classe **Validators**.

required, maxLenght, minLenght, pattern, etc.

```
import { Validators } from '@angular/forms';
```

```
produitForm = new FormGroup({  
  nameControl: new FormControl('', [Validators.required, Validators.minLength(3)]),  
  categoryControl: new FormControl(this.categories[0])  
});
```

# ANGULAR : REACTIVE FORM

---

## Fonctions de validation

Affichage du message d'erreur dans le template :

```
<div class="group">
  <label for="name">Name : </label>
  <input type="text" name="name" id="name" formControlName="nameControl" required>
</div>
<div *ngIf="nameControl()?.invalid && (nameControl()?.dirty || nameControl()?.touched)">
  <div *ngIf="nameControl()?.errors?.required">Le nom est obligatoire</div>
  <div *ngIf="nameControl()?.errors?.['minlength']">Longueur du nom incorrect</div>
</div>
```

267

## Template reactive form

```
nameControl(): AbstractControl | null{
  return this.produitForm.get('nameControl');
}
```

Name :

Le nom est obligatoire

Category : Légumes ▼

Name :

Longueur du nom incorrect

Category : Légumes ▼

# ANGULAR : REACTIVE FORM

## Fonctions de validation

→ Possibilité de créer ses propres fonctions de validation.

```
checkNameValidator(control: FormControl): object | null {  
  const str: string = control.value;  
  if (str[0] >= 'A' && str[0] <= 'Z') {  
    return null;  
  } else {  
    return { checkNameValidator: 'Nom non valide' };  
  }  
}
```

```
produitForm = new FormGroup({  
  nameControl: new FormControl('', [Validators.required, Validators.minLength(3), this.checkNameValidator]),  
  categoryControl: new FormControl(this.categories[0])  
});
```

```
<div *ngIf="nameControl()?.errors?.['checkNameValidator']">nom incorrect</div>
```

Name :

Le nom est obligatoire  
nom incorrect

Category :

# FRAMEWORK ANGULAR

---

Utilisation de Framework

Notion de composant web

## Concept d'Angular

Component / Template

Data Binding

Component Interaction

Service

Rooting

Forms

HTTP

# ANGULAR : HTTP

---

- Angular propose d'utiliser des services « bas niveau » afin de procéder à un échange de donnée avec un service web côté back.
- Utilisation du **module HTTP** (**HttpModule** et depuis la V5 **HttpClientModule**) facilitant la réalisation de requêtes http via les classes suivantes :  
HttpClient, HttpHeaders, HttpInterceptor, HttpRequest, etc.
- Permet d'invoquer des services web via les différentes méthodes HTTP :  
GET, POST, PUT, DELETE

# ANGULAR : HTTP – JSON SERVER

---

- Pour pouvoir réaliser une démonstration, nous allons **créer un serveur** afin de pouvoir transférer des données.

## Utilisation d'un serveur JSON

- **json-server** permet d'imiter une API et de fournir un accès dynamique aux données.
- Possibilité de lire, ajouter, mettre à jour et supprimer des données.  
(GET, POST, PUT, DELETE).
- Open-source
- Utilise le port 3000 par défaut

# ANGULAR : HTTP – JSON SERVER

---

## Json-serveur

→ Comment l'installer ?

```
npm install -g json-server
```

→ Création de notre fichier **db.json** (possibilité d'utiliser un générateur aléatoire de json)

```
{
  "personnes": [
    {
      "index": 0,
      "age": 31,
      "nom": "Cervantes",
      "prenom": "Mullins",
      "gender": "male",
      "company": "ENORMO",
      "email": "cervantesmullins@enormo.com"
    },
  ],
}
```

# ANGULAR : HTTP – JSON SERVER

## Json-serveur

→ Lancement du serveur

```
json-server db.json
```

URL utilisée par le client pour  
réaliser des requêtes HTTP

```
json-server -p 5555 db.json
```

```
\{^_^}/ hi!
```

```
Loading db.json
```

```
Done
```

### Resources

```
http://localhost:5555/personnes
```

### Home

```
http://localhost:5555
```

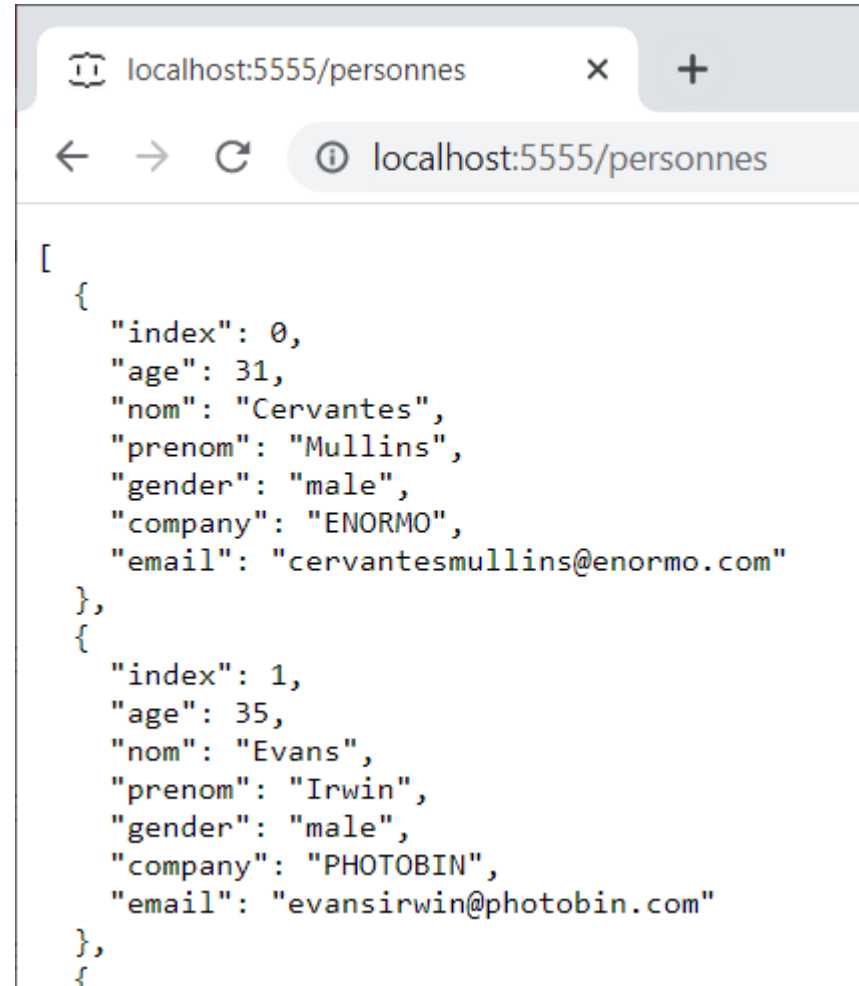
```
Type s + enter at any time to create a snapshot of the database
```



# ANGULAR : HTTP – JSON SERVER

## Json-serveur

→ Lancement du serveur



A screenshot of a web browser window. The address bar shows 'localhost:5555/personnes'. The page content displays a JSON array of two objects. The first object has an index of 0, age of 31, name 'Cervantes', first name 'Mullins', gender 'male', company 'ENORMO', and email 'cervantesmullins@enormo.com'. The second object has an index of 1, age of 35, name 'Evans', first name 'Irwin', gender 'male', company 'PHOTOBIN', and email 'evansirwin@photobin.com'.

```
[
  {
    "index": 0,
    "age": 31,
    "nom": "Cervantes",
    "prenom": "Mullins",
    "gender": "male",
    "company": "ENORMO",
    "email": "cervantesmullins@enormo.com"
  },
  {
    "index": 1,
    "age": 35,
    "nom": "Evans",
    "prenom": "Irwin",
    "gender": "male",
    "company": "PHOTOBIN",
    "email": "evansirwin@photobin.com"
  },
  {

```

# ANGULAR : HTTP – JSON SERVER

---

## → Différentes requêtes HTTP possible :

- Pour récupérer la liste de toutes les personnes :

GET `http://http://localhost:5555/personnes`

- Pour récupérer une personne selon un identifiant :

GET `http://http://localhost:5555/personnes/2`

- Pour ajouter une nouvelle personne :

POST `http://http://localhost:5555/personnes/32`

- Pour modifier les valeurs d'une personnes :

PUT `http://http://localhost:5555/personnes/2`

- Pour supprimer une personne :

DELETE `http://http://localhost:5555/personnes/2`

# ANGULAR : HTTP

---

→ Comment utiliser le module HTTP ?

Dans le **module principale** de l'application :

```
import { HttpClientModule } from '@angular/common/http';
```

276

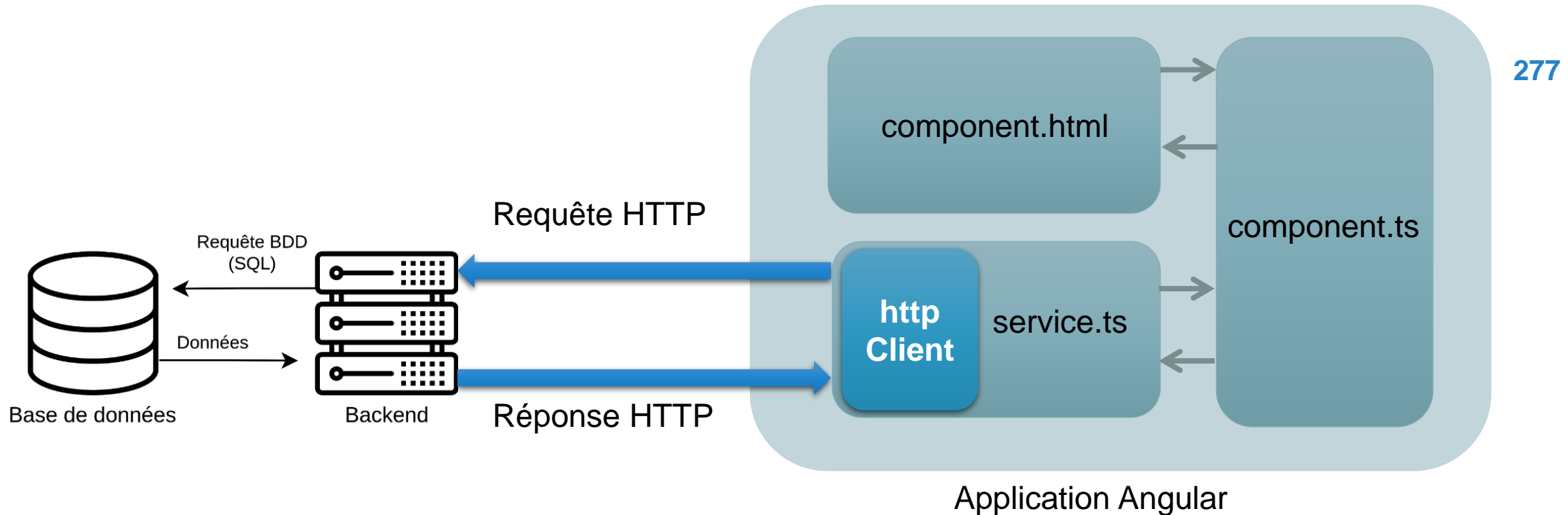
```
@NgModule({  
  declarations: [  
    AppComponent  
  ],  
  imports: [  
    BrowserModule,  
    AppRoutingModule,  
    HttpClientModule  
  ],  
})
```

# ANGULAR : HTTP

---

→ **Comment utiliser le module HTTP ?**

Utilisation d'un **service** pour gérer les transferts de données vers un serveur :



# ANGULAR : HTTP

---

## → Comment utiliser le module HTTP ?

- Les données sont saisies dans le **template** d'un composant (*component.html*)
- La **classe du composant** (*component.ts*) peut récupérer les données du template pour les passer au service (ou inversement récupérer du service pour les envoyer au template).
- Grâce à l'**injection de dépendance du service** (*service.ts*) dans la classe du composant, ce dernier peut l'utiliser pour persister ou récupérer des données.
- En faisant une injection de dépendance de la classe **HttpClient** dans le service, ce dernier peut effectuer des requêtes HTTP en précisant chaque fois la méthode et l'URL.

# ANGULAR : HTTP

---

→ Comment utiliser le module HTTP ?

Utilisation d'un **service** pour gérer les transferts de données vers un serveur :

```
exempleHttp > src > app > TS personne.service.ts > ...  
1  import { HttpClient } from '@angular/common/http';  
2  import { Injectable } from '@angular/core';  
3  
4  @Injectable({  
5    providedIn: 'root'  
6  })  
7  export class PersonneService {  
8  
9    url = "http://localhost:3000/personnes";  
10  
11    constructor(private http: HttpClient) { }  
12  }
```

279

URL de base pour les  
requêtes HTTP

Injection du service HTTP  
dans le service Personnage

# ANGULAR : HTTP

---

→ Comment utiliser le module HTTP ?

Dans notre nouveau composant **Personne** : Injection du service **PersonneService**.

```
9   export class PersonneComponent implements OnInit {  
10  
11       constructor(private personneService: PersonneService) { }  
12  
13       ngOnInit(): void {  
14           //récupération des données du serveur  
15       }  
16   }
```

Fichier `personne.component.ts`

# ANGULAR : HTTP

---

## 1) Récupération de toutes les données Personne :

→ Avant tout, nous avons créé une interface **IData** contenant toutes les caractéristiques de notre personnage (indépendant du contenu du fichier json) :

```
exempleHttp > src > app > TS IData.ts > ...  
1   export interface IData {  
2       index: number;  
3       age: number;  
4       nom: string;  
5       prenom: string;  
6       gender: string;  
7       company: string;  
8       email: string;  
9   }
```



# ANGULAR : HTTP

---

## 1) Récupération de toutes les données Personne :

→ Et ajouté à notre composant **Personne** afin de pouvoir visualiser les données récupérées du serveur :

```
exempleHttp > src > app > personne > TS personne.component.ts > ...  
10  ∨ export class PersonneComponent implements OnInit {  
11  
12      personnes: IData[] = [];
```

```
exempleHttp > src > app > personne > <> personne.component.html >  
1    <h2>Liste des personnes :</h2>  
2  ∨ <ul>  
3  ∨   <li *ngFor="let perso of personnes">  
4      |       {{ perso.prenom }} {{ perso.nom }}  
5      </li>  
6  </ul>
```

# ANGULAR : HTTP

## 1) Récupération de toutes les données Personne :

→ Utilisation de la méthode **HttpClient.get()**.

→ Utilise les **Observables** (rxjs). *(revoir la partie sur les services pour le rappel)*

283

exempleHttp > src > app > TS personne.service.ts > ...

```
11 export class PersonneService {
12
13     url = "http://localhost:3000/personnes";
14
15     constructor(private http: HttpClient) { }
16
17     getAll(): Observable<IData[]>{
18         return this.http.get<IData[]>(this.url);
19     }
20 }
```

La réponse de l'appel HTTP est un observable non typé par défaut.

**Observable de IData[]**  
(attention le serveur peut envoyer autre un autre type de données)

Endpoint URL

# ANGULAR : HTTP

Pas besoin de transformer les données reçu du serveur dans cet exemple

## 1) Récupération de toutes les données Personne :

```
exempleHttp > src > app > personne > TS personne.component.ts > ...
10  export class PersonneComponent implements OnInit {
11
12      personnes: IData[] = [];
13
14      constructor(private personneService: PersonneService) { }
15
16      ngOnInit(): void {
17          //récupération des données du serveur
18          this.personneService.getAll().subscribe(res => {
19              this.personnes = res;
20          })
21      }
22  }
```

284

Res est de type IData[]

(res:IData[])

Souscription au retour de la méthode *getAll()* du service

# ANGULAR : HTTP

---

## 1) Récupération de toutes les données Personne :

Résultat obtenu :

### Liste des personnes :

- Mullins Cervantes
- Irwin Evans
- Wagner Eunice
- Sykes Bean
- Robertson Luisa
- Case Byers

# ANGULAR : HTTP

---

## 1) Récupération de toutes les données Personne :

Comment faire si la réponse de notre requête ne correspond pas à nos structures de données ?

→ Nouvelle **interface** :

```
exempleHttp > src > app > TS IDataLight.ts > ...
1  export interface IDataLight {
2      index: number;
3      age: number;
4      lastname: string; //dans IData : nom
5      firstname: string; //dans IData : prenom
6  }
```

→ Modification du composant **Personne** pour utiliser cette interface :

```
exempleHttp > src > app > personne > TS personne.component.ts > ...
11  export class PersonneComponent implements OnInit {
12
13      personnes: IData[] = [];
14      personnesBis: IDataLight[] = [];
```

```
exempleHttp > src > app > personne > <> personne.component.html > ...
7    <h2>Liste des personnes IDATALIGHT:</h2>
8    <ul>
9        <li *ngFor="let perso of personnesBis">
10            {{ perso.firstname }} {{ perso.lastname }}
11        </li>
12    </ul>
```

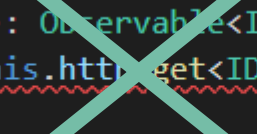
# ANGULAR : HTTP

---

## 1) Récupération de toutes les données Personne :

→ On souhaite avoir une méthode `get()` retournant un `Observable<IDataLight[]>`

→ **Problème :**



```
getAllBis(): Observable<IDataLight[]>{  
  return this.http.get<IData[]>(this.url);  
}
```

→ Utilisation de l'opérateur **map** de **RxJS** pour transformer la réponse (avec l'async **pipe**)

```
getAllBis(): Observable<IDataLight[]>{  
  return this.http.get<IData[]>(this.url).pipe(  
    map( res => res.map( data => {  
      return {  
        index: data.index,  
        age: data.age,  
        lastname: data.nom,  
        firstname: data.prenom  
      } as IDataLight  
    })  
  ));  
}
```

# ANGULAR : HTTP

---

## 1) Récupération de toutes les données Personne :

Possibilité d'ajouter diverses options à la méthode `HttpClient.get()`.

```
options: {  
  ...  
  observe?: 'body' | 'events' | 'response',  
  ...  
  responseType?: 'arraybuffer' | 'blob' | 'json' | 'text',  
  ...  
}
```

Nous voulons la totalité de  
la réponse à la requête http

288

```
getAll(): Observable<IData[]>{  
  return this.http.get<IData[]>(this.url, { responseType: 'json' });  
}
```

Par défaut : body et json

# ANGULAR : HTTP

---

## Headers

- cache-control: no-cache
- content-type: application/json; charset=utf-8
- expires: -1
- pragma: no-cache

### 1) Récupération de toutes les données Personne :

Réponse entière ?

```
exempleHttp > src > app > TS personne.service.ts > ...  
35   getAllResponse(): Observable<HttpResponse<IData[]>>{  
36   |   return this.http.get<IData[]>(this.url, {observe: 'response'});  
37   |   }  
37   }
```

```
exempleHttp > src > app > personne > TS personne.component.ts > ...  
32   showAllResponse() {  
33   |   this.personneService.getAllResponse().subscribe( res => {  
34   |   |   //headers (string[])  
35   |   |   const keys = res.headers.keys();  
36   |   |   this.headers = keys.map(key =>  
37   |   |   |   `${key}: ${res.headers.get(key)}`);  
38   |   |   //body (IData[])  
39   |   |   this.personnes = res.body!;  
40   |   |   })  
41   |   }  
42   }
```



# ANGULAR : HTTP

---


## 1) Récupération de toutes les données Personne :


Possibilité d'avoir des **erreurs** lors des requêtes HTTP

- Le serveur peut rejeter la requête HTTP (code de retour 404 ou 500 par exemple) :  
*error response*
- Ou alors problème côté client (problème de réseau, exception déclenchée par un opérateur RxJS, etc.). Les erreurs ont pour statut 0.

# ANGULAR : HTTP

Exemple de gestion d'erreur :

```
exempleHttp > src > app > TS personne.service.ts >  PersonneService  
18     getAll(): Observable<IData[]>{  
19         return this.http.get<IData[]>(this.url).pipe(  
20             catchError(this.handleError)  
21         );  
22     }
```

```
exempleHttp > src > app > TS personne.service.ts >  PersonneService  
41     private handleError(error: HttpResponse) {  
42         if (error.status === 0) {  
43             // A client-side or network error occurred. Handle it accordingly.  
44             console.error('An error occurred:', error.error);  
45         } else {  
46             // The backend returned an unsuccessful response code.  
47             // The response body may contain clues as to what went wrong.  
48             console.error(  
49                 `Backend returned code ${error.status}, body was: `, error.error);  
50         }  
51         // Return an observable with a user-facing error message.  
52         return throwError(  
53             'Something bad happened; please try again later.');  
54     }
```

# ANGULAR : HTTP

---

## 1) Récupération de toutes les données Personne :

RxJS permet également de refaire une requête http si celle-ci a échouée.

→ `retry()` permet de souscrire une nouvelle fois à un Observable.

```
getAll(): Observable<IData[]>{  
  return this.http.get<IData[]>(this.url).pipe(  
    retry(3),  
    catchError(this.handleError)  
  );  
}
```

# ANGULAR : HTTP

---

## 2) Ajout d'une personne dans le fichier json :

Comment ajouter une nouvelle personne dans le fichier *db.json* ?

→ Utilisation de la méthode `HttpClient.post()`

Fonctionne de manière similaire à `get()`

Possibilité de spécifier un header

Attention, le fichier *db.json* doit posséder un attribut « id » (automatiquement rempli avec `post`)

→ Les données de la personne peuvent être récupérée à partir d'un formulaire.

```
exempleHttp > src > app > TS personne.service.ts > ...  
61     addPersonne(personne: IData): Observable<IData> {  
62         |     return this.http.post<IData>(this.url, personne);  
63     }
```

# ANGULAR : HTTP

---

## 2) Ajout d'une personne dans le fichier json :

Comment ajouter une nouvelle personne dans le fichier *db.json* ?

```
exempleHttp > src > app > personne > TS personne.component.ts > ...  
55     addPersonne() {  
56         this.personneService.addPersonne(this.perso).subscribe( res => {  
57             console.log(res);  
58             this.personnes.push(res);  
59         })  
60     }
```

→ Possibilité d'ajouter une gestion d'erreur comme pour `get()`

# ANGULAR : HTTP

---

## 2) Ajout d'une personne dans le fichier json :

Comment ajouter une nouvelle personne dans le fichier *db.json* ?

Résultat :

### Liste des personnes IDATA:

- Mullins Cervantes
- Irwin Evans
- Wagner Eunice
- Sykes Bean
- Robertson Luisa
- Case Byers
- Titi Toto

295

```
► Object { id: 7, age: 55, nom: "Toto", prenom: "Titi", gender: "No", company: "Esir", email: "Esir@Esir.com" }
```

# ANGULAR : HTTP

---

## Remarque :

- Possibilité d'utiliser le package **concurrently** pour ne plus à avoir démarrer séparément les deux serveurs Angular et json-server.

```
npm install concurrently --save
```

- Package NodeJS permettant d'exécuter plusieurs commandes simultanément.

```
npm start
```

- concurrently "command1 arg" "command2 arg"

```
exempleHttp > {} package.json > {} dependencies
  Debug
4   "scripts": {
5     "ng": "ng",
6     "start": "concurrently \"ng serve\" \"json-server db.json\" ",
7     "build": "ng build"
```

# ANGULAR : HTTP

---

- Les ressources REST ont besoin d'une authentification et d'une autorisation.
- JSON web tokens (JWT)  
Généré par un web service et aide à la communication entre le client et le serveur.
- Création d'un service pour enregistrer le **token** et le réutiliser.
- Mise en place d'un **intercepteur** pour injecter des headers dans tous les requêtes d'authentification (et ne pas à avoir répéter du code)
- Ajout d'un **guard** pour restreindre l'accès à certains composants aux personnes identifiés uniquement.



# FRAMEWORK ANGULAR

---

Utilisation de Framework

Notion de composant web

## Concept d'Angular

Component / Template

Data Binding

Component Interaction

Service

Rooting

Forms

HTTP

Pipes

# ANGULAR : PIPES

---

→ Les données obtenues peuvent ne pas être affichées de la manière souhaitée dans la vue.

Par exemple : la date, la monnaie, l'ordre d'une liste, etc.

→ La solution avec Angular : les **pipes**.

Fonctions prenant en entrée une valeur et retournant la valeur transformée.

→ Plusieurs fonctions existent déjà : DatePipe, UpperCasePipe, LowerCasePipe, CurrencyPipe, DecimalPipe, PercentPipe etc.

# ANGULAR : PIPES

---

→ Exemple pour les dates :

```
today = new Date();
```


```
<p>Nous sommes le : {{ today | date }}</p>  
<p>Nous sommes le : {{ today | date:"fullDate" }}</p>  
<p>Nous sommes le : {{ today | date:"MM/dd/yyyy" }}</p>
```

Nous sommes le : Jan 7, 2022

Nous sommes le : Friday, January 7, 2022 **300**

Nous sommes le : 01/07/2022

→ Utilisation de la fonction registerLocaleData

```
exempleForm > src > app > TS app.module.ts >  AppModule  
14 import { FormsModule } from '@angular/forms';  
15 import { ReactiveFormsModule } from '@angular/forms';  
16 import { registerLocaleData } from '@angular/common';
```

```
providers: [{provide: LOCALE_ID, useValue: "fr-FR"}],
```

Nous sommes le : 7 janv. 2022

Nous sommes le : vendredi 7 janvier 2022

Nous sommes le : 01/07/2022

# ANGULAR : PIPES

→ Possibilité de créer son propre pipe :

```
exempleForm > src > app > TS greeting.pipe.ts > ...
1  import { Pipe, PipeTransform } from '@angular/core';
2
3  @Pipe({
4    name: 'greeting'
5  })
6  export class GreetingPipe implements PipeTransform {
7
8    transform(value: string, gender: string): string {
9      if(gender === "M"){
10        return `Bonjour monsieur ${value}`;
11      } else if(gender === "F"){
12        return `Bonjour madame ${value}`;
13      } else {
14        return `Bonjour ${value}`;
15      }
16    }
17
18  }
```

```
exempleForm > src > app > TS app.module.ts > ...
17  import { GreetingPipe } from './greeting.pipe';
18
19  @NgModule({
20    declarations: [
21      AppComponent,
22      ProductsComponent,
23      GreetingPipe
24    ],
```

301

```
<p>{{ name | greeting:"M" }}</p>
<p>{{ name | greeting:"" }}</p>
```

Bonjour monsieur John Doe

Bonjour John Doe

# FRAMEWORK ANGULAR

---

Utilisation de Framework

Notion de composant web

## Concept d'Angular

Component / Template

Data Binding

Component Interaction

Service

Rooting

Forms

HTTP

Pipes

**Directives**

# ANGULAR : DIRECTIVES

---

Au cours des différents exemples du cours, deux types de directives Angular ont été utilisées :

→ **Directives structurelles** : modification de l'arborescence du DOM.

ngIf, ngFor, NgSwitch, etc.

```
<div *ngIf="condition">Hello World</div>
<ul>
  <li *ngFor="let item of ['un', 'deux', 'trois']">{{item}}</li>
</ul>
```

→ **Directives d'attributs** : pas de modification du DOM, mais des attributs et propriétés des balises HTML existantes.

ngStyle, ngClass, ngModel, etc.

```
<div [ngStyle]="{color: 'blue'}">Hello World</div>
<div [ngClass]="isSpecial ? 'special' : ''">This div is special</div>
```

# ANGULAR : DIRECTIVES

---

**Possibilité de créer ses propres directives.**

Exemple pour un directive d'attribut : Color.

On souhaite modifier la couleur de fond d'un élément s'il est survolé.

304

```
exempleForm > src > app > TS color.directive.ts > ...
1  import { Directive } from '@angular/core';
2
3  @Directive({
4    selector: '[appColor]'
5  })
6  export class ColorDirective {
7
8    constructor() { }
9
10 }
```

```
import { ColorDirective } from './color.directive';

@NgModule({
  declarations: [
    AppComponent,
    FormTDComponent,
    FormRComponent,
    ProductsComponent,
    GreetingPipe,
    ColorDirective
  ],
  imports: [
    BrowserModule,
    FormsModule,
    ReactiveFormsModule,
    HttpClientModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

A intégrer dans le app module

# ANGULAR : DIRECTIVES

---

## Possibilité de créer ses propres directives.

Injection de dépendance de ElementRef pour référencer les éléments concernés par la directive.

```
exempleForm > src > app > TS color.directive.ts > ...
1  import { Directive, ElementRef } from '@angular/core';
2
3  @Directive({
4    selector: '[appColor]'
5  })
6  export class ColorDirective {
7
8    constructor(private el: ElementRef) {
9      el.nativeElement.style.background = 'red';
10   }
11
12 }
```

305

```
<p appColor>Hello world</p>
```

Hello world



# ANGULAR : DIRECTIVES

---

Possibilité de créer ses propres directives.

Utilisation du décorateur @HostListener pour rattacher le changement de couleur à un évènement.

Hello world

```
exempleForm > src > app > TS color.directive.ts > ...
1  import { Directive, ElementRef, HostListener } from '@angular/core';
2
3  @Directive({
4    selector: '[appColor]'
5  })
6  export class ColorDirective {
7
8    constructor(private el: ElementRef) { }
9
10   @HostListener('mouseenter') onMouseEnter(): void {
11     this.changerCouleur('red');
12   }
13   @HostListener('mouseleave') onMouseLeave(): void {
14     this.changerCouleur('white');
15   }
16   changerCouleur(couleur: string){
17     this.el.nativeElement.style.background = couleur;
18   }
19 }
```

# ANGULAR : DIRECTIVES

---

Possibilité de créer ses propres directives.

Utilisation du décorateur @Input pour que la couleur soit un paramètre de l'attribut appColor.

```
<p appColor="blue">Hello world</p>
```

Hello world

```
export class ColorDirective {  
  @Input('appColor') couleur = '';  
  constructor(private el: ElementRef) { }  
  @HostListener('mouseenter') onMouseEnter(): void {  
    this.changerCouleur(this.couleur);  
  }  
  @HostListener('mouseleave') onMouseLeave(): void {  
    this.changerCouleur('white');  
  }  
  changerCouleur(couleur: string){  
    this.el.nativeElement.style.background = couleur;  
  }  
}
```

# FRAMEWORK ANGULAR

---

Utilisation de Framework

Notion de composant web

## Concept d'Angular

Component / Template

Data Binding

Component Interaction

Service

Rooting

Forms

HTTP

Pipes

Directives

**Module**

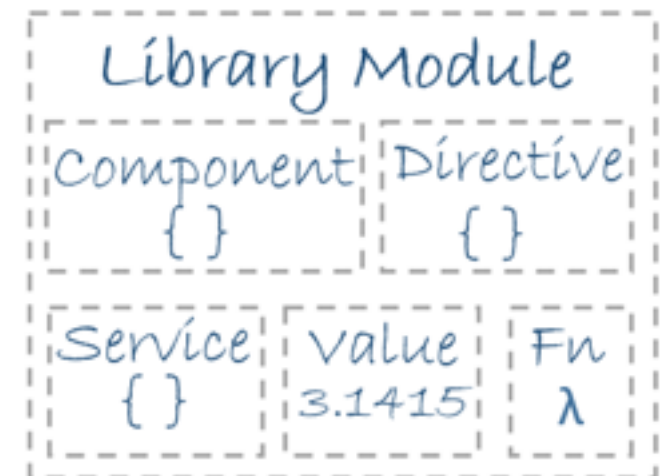
# ANGULAR : MODULES

---

→ Un module correspond à une partie de l'application que l'on veut importer ou exporter.

Décorateur `@ngModule` avec plusieurs propriétés possibles : `declarations`, `exports`, `imports`, `providers`, `bootstrap` (pour le root module uniquement).

→ Possibilité de créer des sous modules dans une application.



# ANGULAR : SOUS-MODULES

---

→ Nouveau sous module Fruits dans notre application.

→ Nouveau composant Pomme

Module contenant les pipes et les directives

```
exempleForm > src > app > modules > fruits > TS fruits.module.ts > ...
1  import { NgModule } from '@angular/core';
2  import { CommonModule } from '@angular/common';
3
4  import { FruitsRoutingModule } from './fruits-routing.module';
5  import { PommeComponent } from './pomme/pomme.component';
6
7  @NgModule({
8    declarations: [
9      PommeComponent
10   ],
11   imports: [
12     CommonModule,
13     FruitsRoutingModule
14   ]
15 })
16 export class FruitsModule { }
```

310

# ANGULAR : SOUS-MODULES

→ Besoin d'importer ce nouveau module dans le module principal de notre application.

```
exempleForm > src > app > TS app.module.ts > AppModule
19 | import { FruitsModule } from './modules/fruits/fruits.module';
20 |
21 | @NgModule({
22 |   declarations: [
23 |     AppComponent,
24 |     FormTDComponent,
25 |     FormRComponent,
26 |     ProductsComponent,
27 |     GreetingPipe,
28 |     ColorDirective
29 |   ],
30 |   imports: [
31 |     BrowserModule,
32 |     AppRoutingModule,
33 |     FormsModule,
34 |     ReactiveFormsModule,
35 |     FruitsModule
36 |   ],
```

311

# ANGULAR : SOUS-MODULES

---

- Possibilité d'avoir un module de routage pour ce sous-module.
- **Eager loading** vs lazy loading
- Dans le app routing module

```
Form > src > app > TS app-routing.module.ts > [🔗] routes
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { FormRComponent } from '../form-r/form-r.component';
import { FormTDComponent } from '../form-td/form-td.component';
import { PommeComponent } from '../modules/fruits/pomme/pomme.component';

const routes: Routes = [
  {path: 'formtd', component: FormTDComponent},
  {path: 'formr', component: FormRComponent},
  {
    path: 'fruits', children: [
      { path: 'pomme', component: PommeComponent}
    ]
  }
];

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

# ANGULAR : SOUS-MODULES

---

- Possibilité d'avoir un module de routage pour ce sous-module.
- **Eager loading** vs lazy loading
- Ou dans le rooting module du nouveau module.

```
exempleForm > src > app > modules > fruits > TS fruits-routing.module.ts > ...  
1  import { NgModule } from '@angular/core';  
2  import { RouterModule, Routes } from '@angular/router';  
3  import { PommeComponent } from './pomme/pomme.component';  
4  
5  const routes: Routes = [  
6    { path: 'pomme', component: PommeComponent }  
7  ];  
8  
9  @NgModule({  
10    imports: [RouterModule.forChild(routes)],  
11    exports: [RouterModule]  
12  })  
13  export class FruitsRoutingModule { }
```



# ANGULAR : SOUS-MODULES

---

→ Possibilité d'avoir un module de routage pour ce sous-module.

→ Eager loading vs **lazy loading**

→ Utilisation de `loadChildren` et des promesses.

```
exempleForm > src > app > TS app-routing.module.ts > AppRoutingModule
6  const routes: Routes = [
7    {
8      path: 'fruits',
9      loadChildren: () => import('./modules/fruits/fruits.module')
10     .then(m => m.FruitsModule)
11   }
12 ];
```

```
exempleForm > src > app > modules > fruits > TS fruits-routing.module.ts >
5  const routes: Routes = [
6    { path: 'pomme', component: PommeComponent }
7  ];
```

+ Suppression du module `FruitsModule` dans les imports du app module.

# ANGULAR : MODULES

---

- Lors de l'apprentissage d'Angular : utilisation de module et moins création de module.
- Plusieurs modules sont déjà créés :
  - Angular material (<https://material.angular.io/>)
  - primeNG (<https://www.primefaces.org/primeng/>)
  - Etc.