

Développement Frontend

Angular

Installation - 1ère étape

Installation de Node.js et Yarn

Pour les utilisateurs de MacOS, je vous conseille d'utiliser le gestionnaire de package [Homebrew](https://brew.sh/) (<https://brew.sh/>)

Une fois que vous avez Homebrew sur votre machine, afin d'installer Node.js et Yarn, tapez dans votre terminal

```
$ brew install yarn
```

Procédure expliquée ici => <https://classic.yarnpkg.com/en/docs/install#mac-stable>

Installation de Node.js et Yarn

Pour les utilisateurs de Ubuntu ou Debian, vous devez d'abord installer Node.js v16.x comme indiqué dans ce [lien](https://github.com/nodesource/distributions/blob/master/README.md#installation-instructions) (<https://github.com/nodesource/distributions/blob/master/README.md#installation-instructions>)

Une fois que vous avez Node.js sur votre machine, afin d'installer Yarn, tapez dans votre terminal

```
$ curl -o- -L https://yarnpkg.com/install.sh | bash
```

Procédure expliquée ici => <https://classic.yarnpkg.com/en/docs/install/#alternatives-stable>

Installation de @angular/cli

Une fois Node.js et Yarn installés sur votre machine, vérifiez les versions en tapant dans votre terminal

```
$ yarn -v  
=> 1.22.15
```

```
$ node -v  
=> v16.10.0
```

Vous pouvez maintenant installer @angular/cli grâce à Yarn en tapant dans votre terminal

```
$ yarn global add @angular/cli
```

Configuration de @angular/cli

Une fois @angular/cli installé sur votre machine, vérifiez les versions en tapant dans votre terminal

```
$ ng version  
▸ @angular/cli : 12.2.8  
▸ node : 16.10.0
```

Il faut maintenant indiquer à @angular/cli d'utiliser Yarn comme package manager en tapant dans le terminal

```
$ ng config --global cli.packageManager yarn
```

Déroulement de la formation

Github de la formation

<https://www.github.com/akanass/nwt-school-front-ng12>

Déroulement de la formation

```
$ git clone https://www.github.com/akanass/nwt-school-front-ng12
```

Installation des dépendances du projet

Une fois que vous avez cloné le projet dans votre répertoire de travail, récupérez les PDFs dans le dossier « documents » et copiez les à un autre endroit sur votre disque

Maintenant, vous devez passer sur la branche step-01 du projet en tapant dans votre terminal

```
$ git checkout -f step-01
```

Une fois sur cette branche, vous devez installer les dépendances du projet en tapant dans votre terminal

```
$ yarn install
```

Installation des dépendances du projet

Une fois que vous avez installé les dépendances du projet, vous pouvez vérifier toutes les versions en tapant dans votre terminal

```
$ ng version
```

```
Angular CLI: 12.2.8
Node: 16.10.0 (Unsupported)
Package Manager: yarn 1.22.15
OS: darwin x64

Angular: 12.2.8
  ... animations, cdk, cli, common, compiler, compiler-cli, core
  ... forms, material, platform-browser, platform-browser-dynamic
  ... router

Package                      Version
-----
@angular-devkit/architect    0.1202.8
@angular-devkit/build-angular 12.2.8
@angular-devkit/core          12.2.8
@angular-devkit/schematics    12.2.8
@schematics/angular           12.2.8
rxjs                          7.3.1
typescript                   4.3.5
```

Déroulement de la formation

Un concept clé **Angular**

Un TP

- Une branche d'exercice : **step-XX**
- Une branche de solution : **step-XX-solution**

Quickstart

La stack « officielle »

- **Angular CLI** : <https://github.com/angular/angular-cli>
- Fichier de configuration : **angular.json**

\$ **ng new my-awesome-app**

- génère l'arborescence de l'application
- initialise un repo Git + 1er commit
- installe les deps NPM

La stack « officielle »

\$ ng generate component user

- génère les fichiers d'un composant
 - src/app/user/user.component.css
 - src/app/user/user.component.html
 - src/app/user/user.component.spec.ts
 - src/app/user/user.component.ts
- met à jour le fichier src/app/app.module.ts avec la référence du nouveau composant

La stack « officielle »

```
$ ng generate service user
```

- génère les fichiers d'un service
 - src/app/user.service.ts
 - src/app/user.service.spec.ts

```
$ ng generate service shared/user
```

- génère les fichiers d'un service
 - src/app/shared/user.service.ts
 - src/app/shared/user.service.spec.ts
- met à jour le fichier src/app/app.module.ts avec la référence du nouveau service

La stack « officielle »

\$ ng generate directive

\$ ng generate pipe

\$ ng generate class

\$ ng generate guard

\$ ng generate interface

\$ ng generate enum

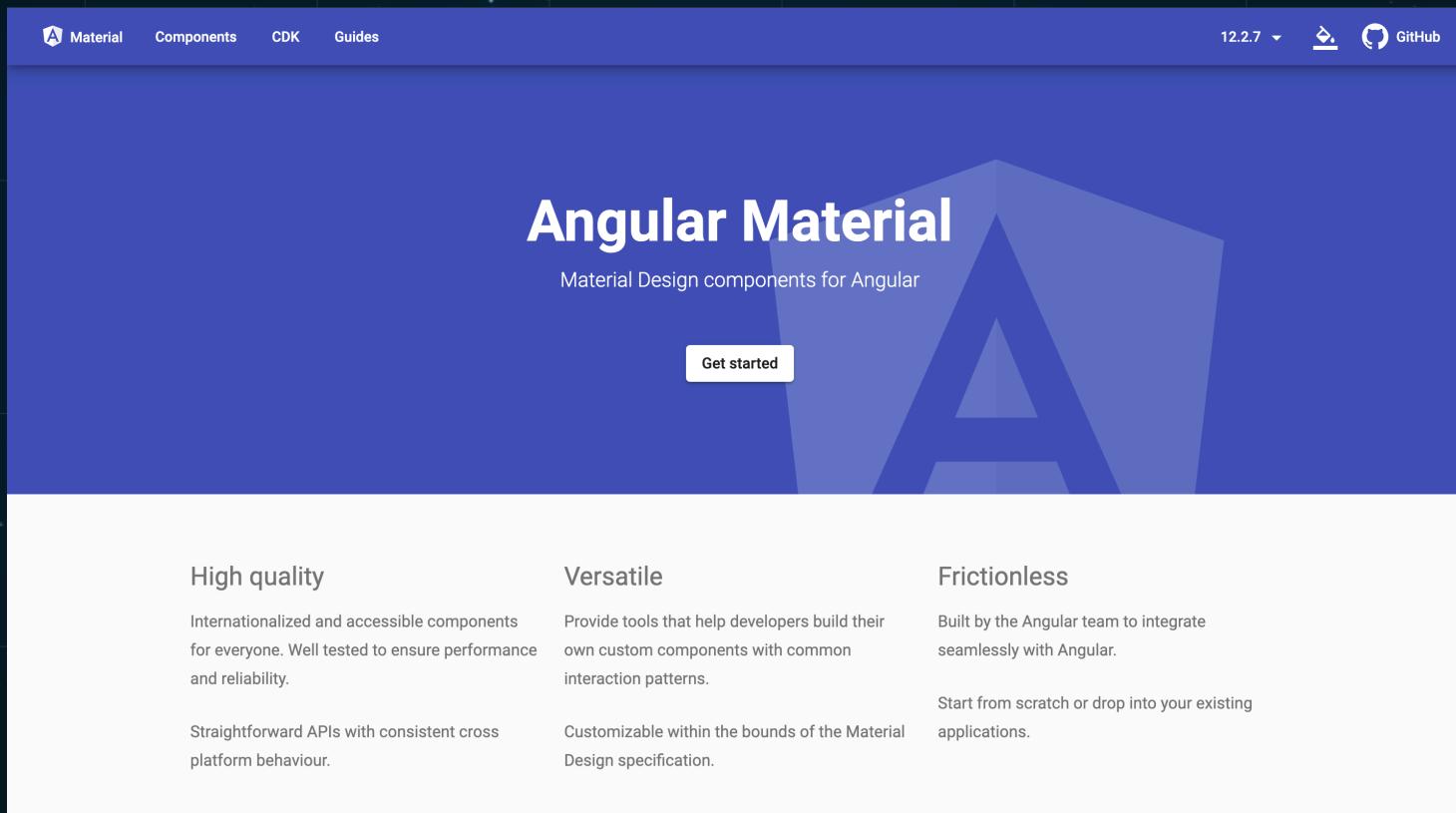
\$ ng generate module

\$ ng build –prod

\$ ng doc directive

...

Un mot sur le Material Design



The screenshot shows the Angular Material homepage. At the top, there's a navigation bar with links for 'Material', 'Components', 'CDK', and 'Guides'. On the right side of the bar are icons for '12.2.7', a bell, and 'GitHub'. The main title 'Angular Material' is prominently displayed in large white letters, with a subtitle 'Material Design components for Angular' below it. A large, stylized 'A' logo is visible on the right. A 'Get started' button is located in the middle-left area. Below the main title, there are three sections: 'High quality', 'Versatile', and 'Frictionless', each with a brief description and a small icon.

High quality
Internationalized and accessible components for everyone. Well tested to ensure performance and reliability.

Straightforward APIs with consistent cross platform behaviour.

Versatile
Provide tools that help developers build their own custom components with common interaction patterns.

Customizable within the bounds of the Material Design specification.

Frictionless
Built by the Angular team to integrate seamlessly with Angular.

Start from scratch or drop into your existing applications.

Un mot sur le Material Design

- Le TP utilise des composants **Material Design**
 - mat-toolbar
 - button[mat-fab], button[mat-button]
 - mat-card
 - mat-input
 - mat-checkbox
 - ...
- Ce n'est pas une formation sur **Material Design**
- Vous n'êtes pas obligé d'utiliser ces composants

Langage utilisé

TypeScript

- ES6+
- Types (optionnels)
- Annotations

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

@Component({
  selector: 'nwt-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'app';
}
```

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>NWTSchoolFront</title>
    <base href="/">

    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="icon" type="image/x-icon" href="favicon.ico">
  </head>
  <body>
    <nwt-root>Loading ...</nwt-root>
  </body>
</html>
```

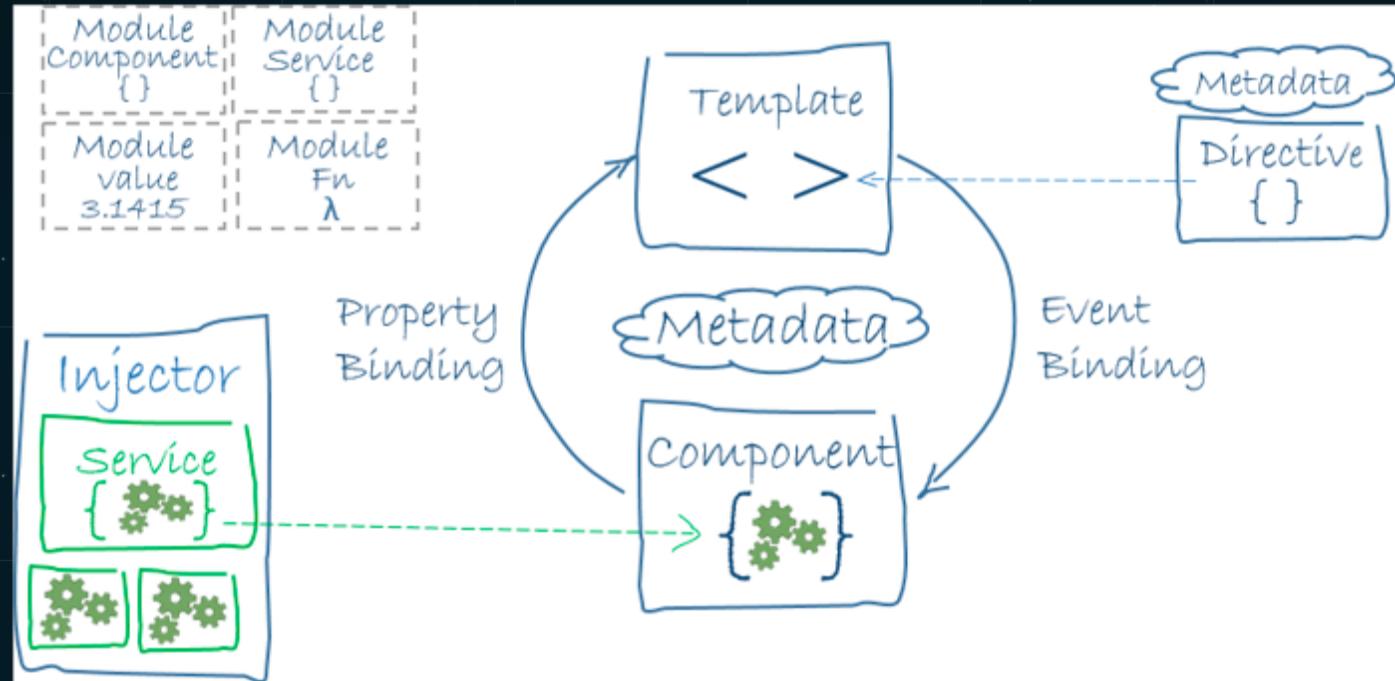
index.html

Webpack

- Bundle en **JavaScript**
- Hot reload
- Choix par défaut d'**Angular**

```
$ yarn run start
```

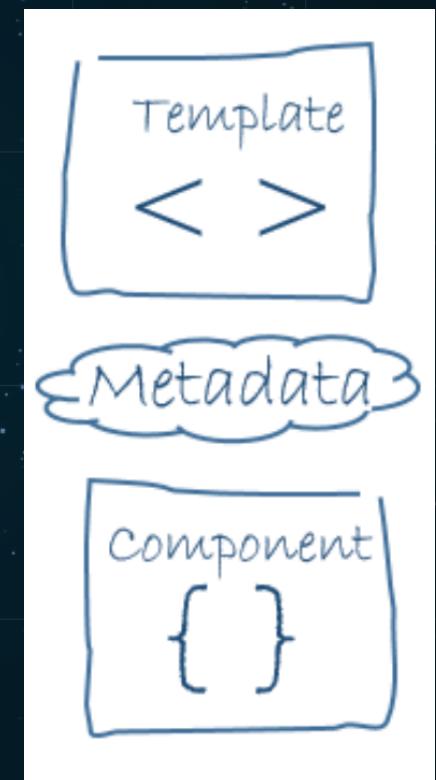
Architecture Globale



Architecture Globale

Votre application: un composant

3 concepts de base



Composant : annotation + classe

- ▶ **Les annotations** : comment afficher le composant dans la page
- ▶ La logique du composant utilise la syntaxe de **classe ES6**

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

@Component({
  selector: 'nwt-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
```

```
export class AppComponent {
  title: string;

  constructor() {
    this.title = 'Angular';
  }
}
```

NgModule

Un module

- Permet de regrouper des fonctionnalités
- Au moins un module par application
- Peut être chargé de façon asynchrone
- Différents types de modules
 - Root (App) module
 - Feature Module
 - Shared module
 - Core module

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

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

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

Exemple d'un module

```
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';
import { environment } from './environments/environment';

if (environment.production) {
  enableProdMode();
}

platformBrowserDynamic().bootstrapModule(AppModule)
  .catch( onrejected: err => console.log(err));
```

Bootstrap

Exercice 1 : prise en main

```
git checkout -f step-01
```

Mise en place de votre premier composant (à la main)

Exercice 1 : prise en main

- Créer un composant AppComponent (**à la main**)
 - src/app/app.component.ts
 - <nwt-root></nwt-root>
- Utiliser le template et le CSS fournis
 - src/app/app.component.html
 - src/app/app.component.css
- Configurer le module de l'application : src/app/app.module.ts
- Afficher le texte static de index.html dans la variable {{ name }}
- Décommenter le contenu du fichier src/main.ts

Solution

```
git checkout -f step-01-solution
```

Exercice 2 : utiliser le CLI

```
git checkout -f step-02
```

Mise en place de votre premier composant avec le CLI

Exercice 2 : utiliser le CLI

- Créer un composant HomeComponent avec le CLI
 - `ng g c home --skip-tests`
- Examiner les fichiers générés dans `src/app/home/`
- Compléter `src/app/home.component.html` et `src/app/home.component.ts`
 - afficher par exemple la chaîne "Hello {{ .name }}"
- Utiliser HomeComponent dans `src/app.module.ts`
 - remplacer l'ancien composant AppComponent dans "bootstrap"
- Changer le nom de l'élément HTML dans `src/index.html`
 - utiliser le sélecteur de HomeComponent

Solution

```
git checkout -f step-02-solution
```

Databinding & Template

```
<html>
    Bonjour <span id="name"></span>
    <input type="text"/>
</html>

window.onload = function(){
    var span = document.querySelector('name');
    var input = document.getElementsByTagName('input')[0];

    input.onkeyup = function(){
        if (span.textContent || span.textContent === "") {
            span.textContent = input.value;
        } else if(span.innerText || span.innerText === "") { // IE
            span.innerText = input.value;
        }
    };
};
```

JavaScript

```
<html>
    Bonjour <span id="name"></span>
    <input type="text"/>
</html>

$(document).ready(function() {
    var $input = $('input');
    var $span = $('#name');

    $input.keyup(function (event) {
        $span.text(event.target.value);
    });
});
```

jQuery

```
<div>  
    <input type="text" name="myName" [(ngModel)]="myName">  
    <p>Bonjour {{myName}}</p>  
</div>
```

Angular

Properties, events et références

```
<div> My name is {{ name }} </div>
<div>
  <input #newname type="text">

  <button (click)="changeName(newname.value)"
    [disabled]="newname.value == 'Angular 4'">Change Name

  </button>
</div>
```

Syntaxe

Anatomie d'un binding

`target="expression"`

Interpolation et expression

- ▶ Interpolation

```
<div>Hello {{name}}</div>

```

- ▶ Les expressions

- ▶ dans le contexte du composant
- ▶ du JS mais
 - ▶ pas d'affectation (sauf pour les events)
 - ▶ pas d'accès aux variables globales (window, document..)
 - ▶ pour les opérateurs logiques, tout est évalué
 - ▶ pas de new, ++, --

3 catégories de binding

Direction	Syntaxe	Type
Unidirectionnel Depuis le model vers la vue	<code>{{ expression }}</code> <code>[targetFooBar] = "expression"</code> <code>bindTargetFooBar = "expression"</code>	Interpolation Propriétés Classe Attribut Style
Unidirectionnel Depuis la vue vers le model	<code>(targetFooBar) = "expression"</code> <code>onTargetFooBar = "expression"</code>	Événements
Bidirectionnel	<code>[(targetFooBar)] = "expression"</code> <code>bindOnTargetFooBar = "expression"</code>	Bidirectionnel

Mais avant : attributs vs propriétés

- Les attributs c'est du **HMTL**, les propriétés c'est du **DOM**
 - mapping strict (`id`)
 - attribut sans propriété (`colspan`)
 - propriété sans attribut (`textContent`)
 - les 2
- La plupart du temps, les **attributs** servent à initialiser les **propriétés** mais **ne sont pas modifiés** si la **propriété change**
- Des attributs sans valeurs : `<bouton disabled>Click!!</bouton>`
- Un attribut : une chaîne de caractère

Que des propriétés ...

- Un monde sans attributs
- Avec le binding, nous travaillons sur les **propriétés** (des éléments, composants ou directives)

```
<bouton [disabled] = "true" >Click!!</bouton>
```

- Permet de passer des objets

Property binding

Type	Cible	Exemple
Propriété	Attribut d'élément	<code></code>
	Attribut de composant	<code><my-component [data]="currentData"></my-component></code>
	Attribut de directive	<code><div [ngClass]="{{selected: isSelected}}></div></code>

Ok mais si je veux un attribut ...

- ▶ Les éléments n'ont pas forcément la propriété (ex: **aria**, **svg**, **colspan**)

- ▶ On peut cibler un attribut en précédant le nom de **attr**.

```
<td [attr.colspan] = "1+1">a cell!!</td>
```

- ▶ Pour les classes, on précède le nom de la classe par **class**.

```
<div [class.isSpecial] = "isSpecial">special class</div>
```

- ▶ Pour les styles, on précède le nom de la propriété par **style**.

```
<div [style.color] = "isSpecial ? 'red' : 'green'">Special class</div>
```

Event binding

Type	Cible	Exemple
Événement	Événement d'élément	<code><button (click)="onSave()"></button></code>
	Événement de composant	<code><hero-detail (deleted)="onDeleted(\$event)"> </hero-detail></code>
	Événement de directive	<code><input (change)="firstName = \$event" /></code>

2 way binding

Type	Cible	Exemple
Bidirectionnel	Propriété Événement de directive	<code><input name="firstName" [(ngModel)]="firstName" /></code>

- équivalent à
`<input [ngModel]="firstName" (ngModelChange)="firstName=$event">`
- Note : **ngModel** est fourni par le package **@angular/forms**

Variables locales

- ▶ Variables :
- ▶ Une valeur (**let**)

```
<movie-detail  
  *ngFor="let movie of movies">  
</movie-detail>
```

- ▶ Référence :
- ▶ l'élément (#)
- ▶ disponible dans :
- ▶ tout le template
- ▶ le composant

```
<input #phone >  
<button (click)="click(phone.value)">Call</button>
```

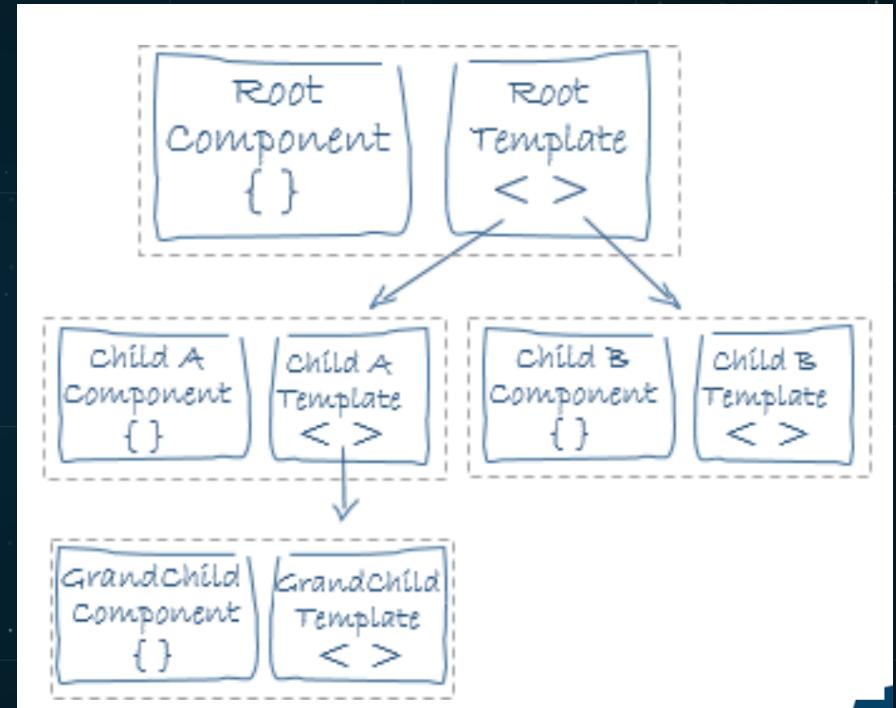
Composants

2 types de composants

- La "**Directive**" permet d'enrichir un élément HTML...
- Le "**Composant**" est une **directive** avec une vue et des styles CSS

Composants : arbre

- ▶ Arbre de composants
- ▶ Les “enfants” sont ajouté au parent s’ils apparaissent dans son template
- ▶ Les composants doivent être déclarés dans le module



Composant : @Component()

Component

- selector
- templateUrl ou template
- providers
- ...

```
@Component({  
  selector: 'ng2-app',  
  templateUrl: home.component.html',  
  ...  
})
```

Composant : la Classe

Des annotations

- `@Input()`
- `@Output()`
- ...

```
class FormComponent {  
  @Input() name: string;  
  @Output('personAdd') personAdd$: EventEmitter;  
  
  constructor(){  
    this.personAdd$ = new EventEmitter<any>();  
  }  
}
```

```
<app-form (personAdd)="addPerson($event)" [name]="formTitle"></app-form>
```

```
import { Component, Output, EventEmitter } from '@angular/core';
@Component({
  selector: "app-child"
})
export class ChildComponent {
  @Output() childEvent$: EventEmitter<string>;
  constructor() {
    this.childEvent$ = new EventEmitter<string>();
  }
  raiseEvent(){
    this.childEvent$.emit("event from child");
  }
}
```

Communication entre composants par événements

Impliquer les composants

Lorsqu'un composant parent utilise des composants enfants :

- Ils doivent être référencés
- Ils doivent être déclarés dans les déclarations du `@NgModule()`

```
import { HomeComponent } from './app/home/';
import { FooDirective } from './app/shared/';

@NgModule({
  declarations: [HomeComponent, FooDirective]
})
```

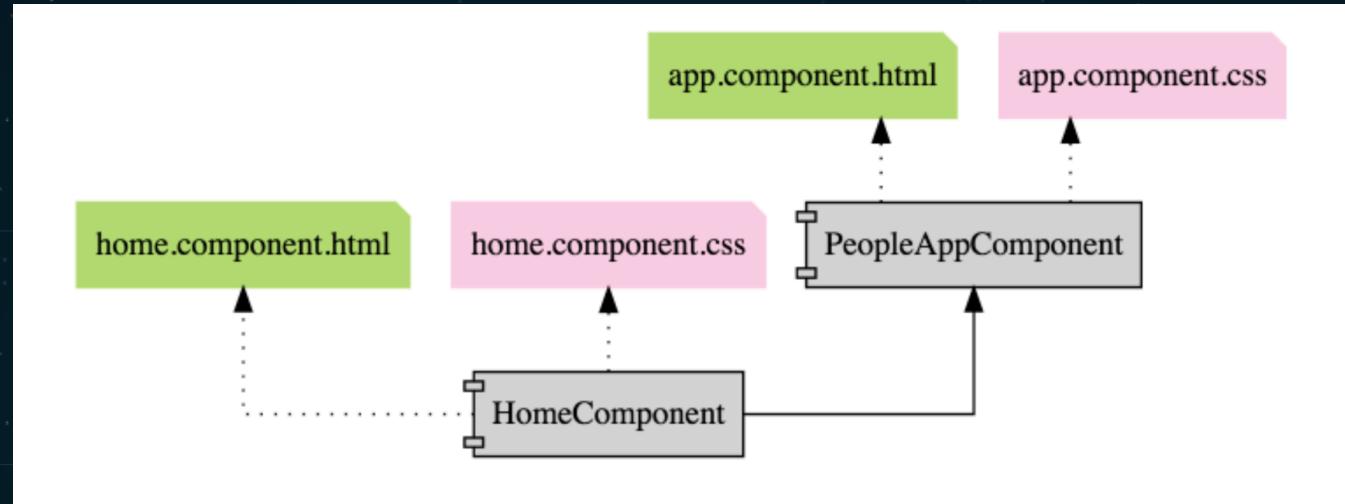
Exercice 3

`git checkout -f step-03`

• Imbriquer les composants

Exercice 3 : imbriquer les composants

- Faites en sorte que le composant `AppComponent` utilise `HomeComponent`
- Voir le contenu du fichier `src/app/app.component.html`
- Mettre à jour les fichiers suivants :
 - `src/index.html`
 - `src/app/app.module.ts`
 - `src/app/app.component.html`
 - `src/app/app.component.ts`



Exercice 3 : imbriquer les composants

Solution

```
git checkout -f step-03-solution
```

Exercice 4

`git checkout -f step-04`

Afficher une personne

Exercice 4 : afficher une personne

- Créer un composant `PersonComponent` pour afficher les détails d'une personne
 - `ng g c person --skip-tests`
- Utiliser les fichiers suivants comme exemple pour le contenu :
 - `src/app/_static/person.component.html`
 - `src/app/_static/person.component.css`
 - `src/app/_static/people.ts`
- Regarder le contenu du composant principal (=> une explication sera donnée)
- Utiliser les données de `people.ts` dans `src/app/person.component.ts`
- Afficher ce nouveau composant dans `src/home/home.component.html`

Leanne Woodard
BIOSPAN

✉ Leanne.Woodard@BIOSPA...
☎ 0784112248

Manager : Erika
Location : DEVLAND



📍 🖊 🗑

Exercice 4 : afficher une personne

Solution

```
git checkout -f step-04-solution
```

Gestion des événements du DOM

Les événements

- ▶ Nom de l'événement entre ()
- ▶ Fait référence à une fonction de la classe
- ▶ Pour récupérer les détails de l'event : **\$event**

```
export class MyComponent {  
  values: string = '';  
  
  constructor(){}
  
  
  onClick(event){  
    this.values += event.target.value + ' | '|';  
  }
}
```



The diagram illustrates the flow of an event from the component's event handler to the input element. A curved arrow originates from the 'onClick' method in the component's code and points to the '(click)' attribute in the input element's binding.

Exercice 5

`git checkout -f step-05`

Gérer un clic

Exercice 5 : gérer un clic

- Un bouton “random” a été ajouté dans `src/app/person.component.html`
- Ajouter un clic sur ce bouton pour afficher une personne au hasard
 - exploiter le tableau `PEOPLE` fourni

Solution

```
git checkout -f step-05-solution
```



Fin de la 1ère journée

Si vous avez apprécié la formation, envoyez un Tweet !

#NewWebTechnologies #Angular @_akanass_ @TaDaweb

Communication avec le serveur

```
1. // app.module.ts:
2.
3. import {NgModule} from '@angular/core';
4. import {BrowserModule} from '@angular/platform-browser';
5.
6. // Import HttpClientModule from @angular/common/http
7. import {HttpClientModule} from '@angular/common/http';
8.
9. @NgModule({
10.   imports: [
11.     BrowserModule,
12.     // Include it under 'imports' in your application module
13.     // after BrowserModule.
14.     HttpClientModule,
15.   ],
16. })
17. export class AppModule {}
```

Intégrer HttpClientModule dans NgModule

```
{  
  "results": [  
    "Item 1",  
    "Item 2",  
  ]  
}
```



Récupérer des données JSON par requête HTTP

```
1. @Component(...)
2. export class MyComponent implements OnInit {
3.
4.   results: string[];
5.
6.   // Inject HttpClient into your component or service.
7.   constructor(private http: HttpClient) {}
8.
9.   ngOnInit(): void {
10.     // Make the HTTP request:
11.     this.http.get('/api/items').subscribe(data => {
12.       // Read the result field from the JSON response.
13.       this.results = data['results'];
14.     });
15.   }
16. }
```

Récupérer des données JSON par requête HTTP

```
interface ItemsResponse {  
    results: string[];  
}
```



```
http.get<ItemsResponse>('/api/items').subscribe(data => {  
    // data is now an instance of type ItemsResponse, so you can do this:  
    this.results = data.results;  
});
```



Typer les réponses

```
http
  .get<MyJsonData>('/data.json', {observe: 'response'})
  .subscribe(resp =>
    // Here, resp is of type HttpResponse<MyJsonData>.
    // You can inspect its headers:
    console.log(resp.headers.get('X-Custom-Header'));
    // And access the body directly, which is typed as MyJsonData as requested.
    console.log(resp.body.someField);
  );

```

Accéder à la réponse complète

```
http
  .get<ItemsResponse>('/api/items')
  .subscribe(
    // Successful responses call the first callback.
    data => {...},
    // Errors will call this callback instead:
    err => {
      console.log('Something went wrong!');
    }
);
```

Gestion des erreurs

```
1. http
2.   .get<ItemsResponse>('/api/items')
3.   .subscribe(
4.     data => {...},
5.     (err: HttpErrorResponse) => {
6.       if (err.error instanceof Error) {
7.         // A client-side or network error occurred. Handle it accordingly.
8.         console.log('An error occurred:', err.error.message);
9.       } else {
10.         // The backend returned an unsuccessful response code.
11.         // The response body may contain clues as to what went wrong,
12.         console.log(`Backend returned code ${err.status}, body was: ${err.error}`);
13.       }
14.     }
15.   );
```

Accéder au détail des erreurs

Réponse différente que du JSON

- Le serveur peut renvoyer autre chose que du JSON :
 - text
 - binaire
- L'option **responseType** doit être fournie :
 - json (défaut)
 - text
 - arraybuffer
 - blob

```
http
  .get('/textfile.txt', {responseType: 'text'})
  // The Observable returned by get() is of type Observable<string>
  // because a text response was specified. There's no need to pass
  // a <string> type parameter to get().
  .subscribe(data => console.log(data));
```

```
const body = {name: 'Brad'};

http
  .post('/api/developers/add', body)
  // See below - subscribe() is still necessary when using post().
  .subscribe(...);
```



Envoyer des données au serveur

Un monde d'Observable

- **HttpClient** utilise **RxJS** pour effectuer les traitements de manière **asynchrone**.
- Toutes les fonctions retournent des **Observables**.
- **Rien n'est exécuté** si on ne **souscrit** pas à l'**Observable** avec la méthode **.subscribe()**
- A **chaque appel** de cette méthode, une **nouvelle requête** est exécutée.
- <https://angular.io/guide/observables>

Configuration de la requête

On peut ajouter des **Headers** à notre requête :

- ▶ En instantiant l'objet `HttpHeaders()` et en lui passant des paramètres dans le constructeur :
 - ▶ `{ headers : new HttpHeaders({ 'Content-Type': 'application/json' }) }`
- ▶ En instantiant l'objet `HttpHeaders()` et en utilisant ses assesseurs :
 - ▶ `{ headers : new HttpHeaders().set('Content-Type', 'application/json') }`

```
http
  .post('/api/items/add', body, {
    headers: new HttpHeaders().set('Authorization', 'my-auth-token'),
  })
  .subscribe();
```

Exercice 6

`git checkout -f step-06`

Récupérer les personnes à partir du serveur

Exercice 6 : fini les données en dur

- Dans une autre fenêtre de votre terminal, lancez le serveur :
`$ yarn run server`
- Utiliser `HttpClient` pour récupérer les données depuis le serveur
- GET `http://localhost:3000/people`
 - retourne un tableau de PEOPLE
- GET `http://localhost:3000/people/random`
 - retourne une personne au hasard
- L'URL `http://localhost:3000/documentation`
 - Affiche la documentation du server dans votre navigateur

Solution

```
git checkout -f step-06-solution
```

Navigation

Configuration (simple)

- **path** : l'URL de route (ex: /people/:id)
- **component** : le composant associé à cette route (ex: PeopleComponent)
- **redirectTo** : le fragment d'URL vers lequel rediriger route courante (ex: '/home')
- **pathMatch** : stratégie de redirection (full / prefix)
 - **full** : tente une reconnaissance depuis la racine de la route
 - **prefix**: tente une reconnaissance partielle de la route

Configuration (complète)

Voir la documentation officielle pour toutes les autres options possibles :

<https://angular.io/api/router/Route#description>

```
// app-routing.module.ts
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { HomeComponent } from './home/home.component';

const routes: Routes = [
  { path: '', redirectTo: 'home', pathMatch: 'full' },
  { path: 'home', component: HomeComponent }
];

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

Exemple : définitions des routes

```
// app.module.ts
import { AppRoutingModule } from './app-routing.module';

@NgModule({
  imports: [
    AppRoutingModule
  ]
})
export class AppModule {
```

Exemple : utilisation des routes

Stratégie de navigations

- Par “**Path**”: PathLocationStrategy (Mode HTML5 et pushState => Par défaut)
 - ex: localhost/people/1
 - {useHash: false}
- Par “**Hash**”: HashLocationStrategy
 - ex: localhost/#/people/1
 - {useHash: true}

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

@Component({})
export class FooComponent implements OnInit {

    constructor(private _route: ActivatedRoute, private _router: Router) { }

    ngOnInit() {
        this._route.params.subscribe(params => {
            let id = params['id'];
        });
    }

    go() { this._router.navigate(['/people/']); }
}
```

Utilisation dans un composant : Typescript

```
<a class="btn btn-info" [routerLink]=["/people"]>Movies Liste</a>
<a class="btn btn-info" [routerLink]=["/people/", obj.id]>Edit</a>

<section class="container">
  <router-outlet></router-outlet>
</section>
```

Utilisation dans un composant : HTML

Exercice 7

`git checkout -f step-07`

Une navigation côté client

Exercice 7 : une navigation côté client

- Compléter le fichier `src/app/app-routing.module.ts` avec la configuration des routes
- Mettre à jour le fichier `src/app/app.module.ts`
- Utiliser le `<router-outlet></router-outlet>` dans le fichier :
 - `src/app/app.component.html`

Solution

```
git checkout -f step-07-solution
```



Ajoutons des fonctionnalités

Itérer sur une collection avec *ngFor

- Itère dans une collection et génère un template par élément
- **index, odd, even, last** à utiliser en alias dans des variables

```
<ul>
  <li *ngFor="let fruit of fruits; let i=index">
    <!-- répétition du template li par élément fruit -->
    {{ i }} : {{ fruit.name }}
  </li>
</ul>
```

Exercice 8

```
git checkout -f step-08
```

Afficher la liste des personnes en utilisant la directive *ngFor

Exercice 8 : répéter les personnes

- Créer un composant `PeopleComponent` pour afficher la liste des personnes
 - `ng g c people --skip-tests`
- Lui associer une route `/people` accessible depuis le lien List (en haut dans la toolbar)
- Pour la liste vous pouvez répéter le contenu de :
 - `src/app/person/person.component.html`
 - note : il est inutile de copier le bouton "random"
 - `src/app/person/person.component.css`
- Nous verrons comment améliorer cela lors de la prochaine question

Solution

```
git checkout -f step-08-solution
```

Exercice 9

`git checkout -f step-09`

Créer un composant pur pour éviter la duplication

Exercice 9 : réutilisation des composants

- Créer un composant `CardComponent` qui affichera les détails d'une personne
 - `ng g c shared/card --skip-tests`
 - **note** : Ce composant sera créé dans `src/app/shared` car il sera utilisé à plusieurs endroits
 - Y transférer les contenus HTML et CSS du composant `PersonComponent`
 - Faire en sorte que le composant `PersonComponent` utilise `CardComponent` :

```
<nwt-card [person]="person"></nwt-card>
```

Solution

```
git checkout -f step-09-solution
```

Exercice 10

`git checkout -f step-10`

La même chose pour les People

Exercice 10

- Même question que précédemment...
- Faire en sorte que le composant PeopleComponent utilise CardComponent

Solution

```
git checkout -f step-10-solution
```

Exercice 11

`git checkout -f step-11`

Supprimer une personne

Exercice 11 : supprimer une personne

- Ajouter un événement clic sur le bouton de suppression dans CardComponent
- Propager l'événement jusqu'au PeopleComponent
 - l'événement devra s'appeler deletePerson
 - astuce : utiliser @Output()
- L'API à utiliser est celle-ci :
 - DELETE <http://localhost:3000/people/:id>
 - retourne 204
 - vous devez mettre à jour le tableau de personnes

Solution

```
git checkout -f step-11-solution
```

Les Formulaires avec Angular

Template Driven Forms
Model Driven Forms

Template Driven Forms

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

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

Template Driven Forms : FormsModule

Template Driven Forms : Syntaxe

- `#f="ngForm"`
 - déclarer une référence sur un formulaire (`#f` est un exemple de réf)
- `f.value`
 - **JSON** avec les valeurs de tous les champs de ce formulaire

```
<form #f="ngForm" (ngSubmit)="onSubmit(f.value)">
  ...
</form>
```

Template Driven Forms : Syntaxe

- **ngModel** : le binding d'un contrôle f.value
- **name** : associer un contrôle au champ (obligatoire avec un **ngModel**)

```
<form #f="ngForm">

    <input name="title" ngModel>

    <input ngModel name="city" #city="ngModel"></input>

    <input [(ngModel)]="postalCode" name="postalCode"></input>

</form>
```

Template Driven Forms : Syntaxe

- **ngModelGroup** : regrouper des contrôles (optionnel)

```
<form #f="ngForm">

  <p ngModelGroup="address">
    <input ngModel name="city" #city="ngModel"></input>
    <input [(ngModel)]="postalCode" name="postalCode"></input>
  </p>

  <button type="submit" [disabled]="!f.valid">Modifier</button>

</form>
```

Exercice 12

`git checkout -f step-12`

Ajouter une personne

Exercice 12 : ajouter une personne (1/2)

- Petite contrainte afficher dans une modale
- Le composant `PeopleComponent` a été complété avec une méthode :
 - `showDialog()` : permet d'afficher la modale
- Le template a été complété avec un bouton pour ajouter une personne (affiche la modale d'ajout)
- On a ajouter le composant `DialogComponent` pour gérer la modale :
 - dans `src/app/shared/dialog`
 - `onCancel()` : permet de fermer la modale sans traitement
 - `onSave()` : permet de fermer la modale en passant des données à son parent et elles seront accessibles dans l'observer `dialogRef.afterClosed()`

Exercice 12 : ajouter une personne (2/2)

- Créer un composant `FormComponent` (dans shared)
 - `ng g c shared/form --skip-tests`
 - vous trouverez dans `app/_static` les fichiers HTML et CSS à utiliser
 - **Sorties** : (`cancel`), (`addPerson`)
- Mettre à jour le composant `DialogComponent` en y intégrant `FormComponent`
- Implémenter dans `PeopleComponent` la méthode privée `_add()` qui ajoute un contact et qui sera appelée dans l'observer `dialogRef.afterClosed()`:
 - L'API à utiliser est celle-ci :
 - POST `http://localhost:3000/people`
 - retourne la personne créée

Solution

```
git checkout -f step-12-solution
```

Exercice 13

`git checkout -f step-13`

Modifier une personne

Exercice 13 : modifier une personne (1/2)

- Créer un composant `UpdateComponent`
 - `ng g c update --skip-tests`
- Ce composant doit être accessible via l'url `/edit/:id`
 - pensez à mettre à jour `src/app/app-routing.module.ts`
 - ainsi que `src/app/shared/card/card.component.html`
- Récupérer le paramètre `id` depuis la route (`ActivatedRoute`)
- Utiliser l'API `/people/:id` (GET)

Exercice 13 : modifier une personne (2/2)

- Appeler `DialogComponent` dans `UpdateComponent` afin d'afficher au chargement la modale avec le formulaire contenant les données de la personne courante
- Mettre à jour `FormComponent`
 - Entrée : [`model`]
 - Sorties : (`cancel`), (`submit`)
 - ajouter un “mode édition” afin d’utiliser le même formulaire pour l’édition et la création
 - Si `model` alors `isUpdateMode=TRUE` sinon `isUpdateMode=FALSE...`
- Utiliser le Template Driven Forms, `ngModel`
- La mise à jour se fait sur l’API `/people/:id` (PUT)
- Retourner sur la liste à la fin du traitement

Solution

```
git checkout -f step-13-solution
```



Fin de la 2ème journée

Si vous avez apprécié la formation, envoyez un Tweet !

#NewWebTechnologies #Angular @_akanass_ @TaDaweb

Template Driven Forms Validation

Les états d'un contrôle (ou groupe)

- **control.pristine** : l'utilisateur n'a pas interagi avec le contrôle.
- **control.dirty** : l'utilisateur a déjà interagi avec le contrôle.
- **control.valid** : le contrôle est valide.
- **control.invalid** : le contrôle n'est pas valide.
- **control.touched** : le contrôle a perdu le focus.
- **control.unouched** : le contrôle n'a pas encore perdu le focus.

Les classes CSS

Class name disponible pour le skin :

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

La gestion des erreurs

- Pour un **groupe de contrôles** ou un **contrôle**
 - `control.valid`, `control.invalid` ...
 - `control.errors`
- Par exemple :
 - `f.valid`
 - `f.hasError('minlength')`

```
<form #f="ngForm">
  <div>
    <input name="user" ngModel #user="ngModel" required>

    <div *ngIf="user.dirty && user.hasError('required')">
      <span class="help-block">Ce champ est obligatoire</span>
    </div>

  </div>
</form>
```

Exemple avec gestion des erreurs

Exercice 14

`git checkout -f step-14`

Valider votre formulaire

Exercice 14 : valider votre formulaire

- Valider les champs
 - Firstname : required + min 2 lettres
 - Lastname : required + min 2 lettres
 - Email : required
 - Address : required
 - City : required
 - PostalCode : required
 - Phone : required et numéro FR ⇒ `(0|\+33)\d{9}`
- Afficher des messages en fonction des erreurs
 - utiliser la directive `<mat-error></mat-error>`

Solution

```
git checkout -f step-14-solution
```

Model Driven Forms

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

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

Model Driven Forms : ReactiveFormsModule

Model Driven Forms : Syntaxe

- Référence au modèle de formulaire via `formGroup`
- Mapping de controls via `formControlName`

```
<form [formGroup]="editForm">
  <input type="text" formControlName="name">

  <div *ngIf="editForm.get('name').valid">...</div>
  <button type="submit" [disabled]="editForm.invalid">
    Modifier
  </button>
</form>
```

Model Driven Forms : Syntaxe

- `[formGroup] = "f"`
- déclarer une référence sur un modèle de formulaire "f"

```
<form [formGroup] = "f">  
    ...  
</form>
```

Model Driven Forms : Syntaxe

- **formControlName** : le binding d'un contrôle

```
<form [FormGroup]="f">

  <input name="title" formControlName="title">

  <div name="address" formGroupName="address">...</div>

</form>
```

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

@Component({})
export class FormComponent {
  form: FormGroup;

  constructor() {
    this.form = new FormGroup({
      id: new FormControl(''),
      firstname: new FormControl('', Validators.compose([
        Validators.required, Validators.minLength(2)
      ])),
      ...
    });
  }
}
```

Dans la classe

```
<form [formGroup]="form">
  <div>
    <input name="user" formControlName="user">

    <div *ngIf="form.get('user').dirty && form.get('user').hasError('required')">
      <span class="help-block">Ce champ est obligatoire</span>
    </div>

  </div>
</form>
```

Exemple avec la gestion des erreurs

Exercice 15

`git checkout -f step-15`

Valider votre formulaire

Exercice 15 : valider votre formulaire

- Transformer `FormComponent` en mode `Model-Driven`
- Tenez compte du mode "création" aussi
- Astuces :
 - Utiliser la méthode `.patchValue()` de l'objet `FormGroup` pour mettre à jour le formulaire avec les données du modèle
 - `Validators.pattern('(0|\\+33)\\d{9}')`
 - `Validators.minLength(2)`

Solution

```
git checkout -f step-15-solution
```

Model Driven Forms

Validation personnalisée

Créer ses propres validateurs

Créer sa fonction de validation :

- Si **OK** : retourne **NULL**
- Si **NOK** : retourne un objet sous la forme { **errorName: true** }

```
CustomEmailValidator = (c: FormControl) => (c.value.indexOf('@') !== -1) ? null : {email: true}
```

```
this.formModel = new FormGroup({  
    email: new FormControl('', CustomEmailValidator),  
    // or  
  
    email: new FormControl('', Validators.compose([  
        Validators.required, CustomEmailValidator  
    ]))  
});
```

Utilisation du validateur

Exercice 16

`git checkout -f step-16`

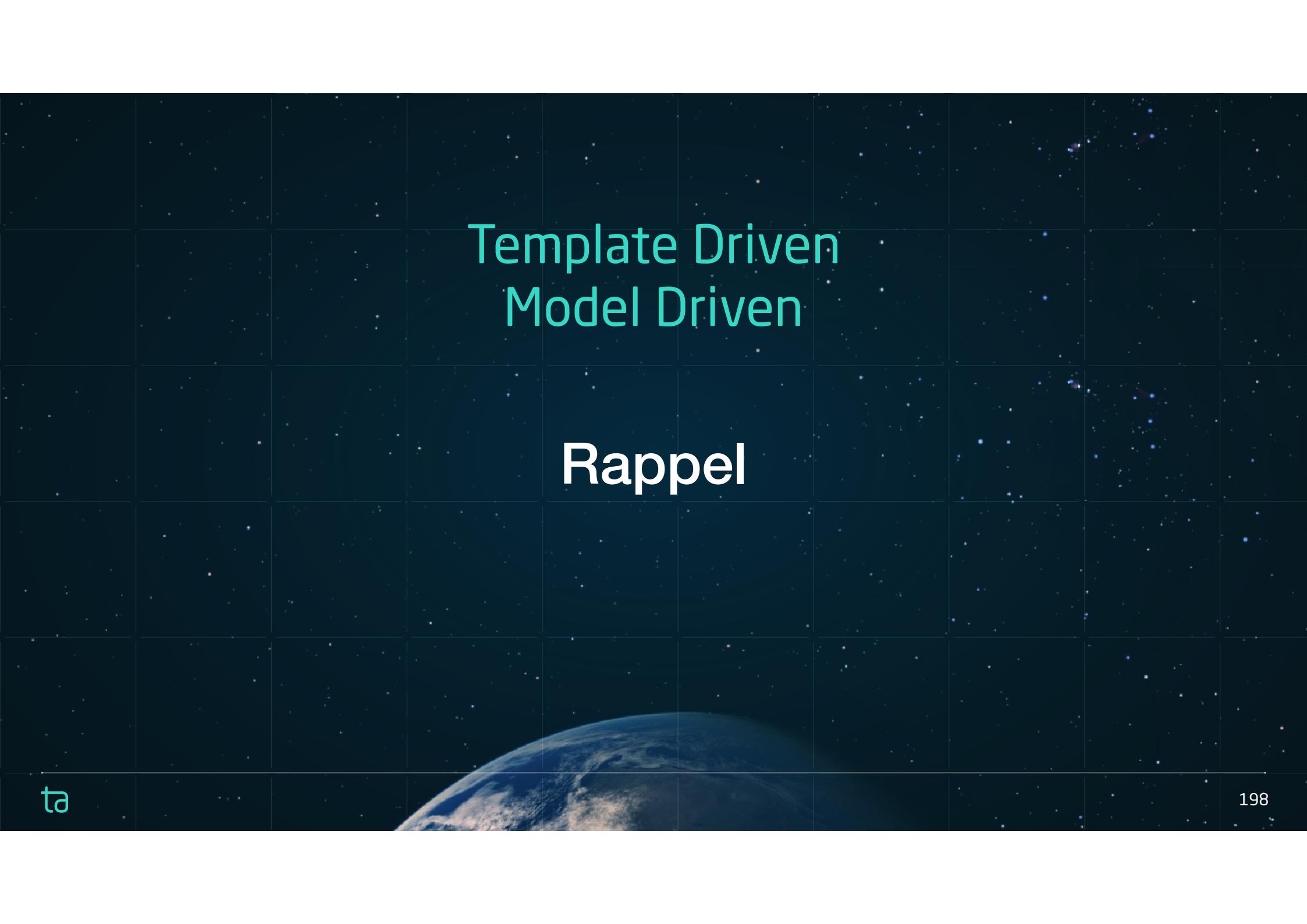
Votre validateur

Exercice 16 : votre validateur

- Générer une classe `CustomValidators` dans `shared/form`
 - `ng g class shared/form/custom-validators --skip-tests`
- Créer un validateur (méthode statique) qui vérifie l'adresse e-mail
 - respectant le format "name.firstname@gmail.com"
- Associer ce validateur au control "mail"
- Afficher dans le HTML le message correspondant au type d'erreur

Solution

```
git checkout -f step-16-solution
```



Template Driven
Model Driven

Rappel

Template-Driven vs Model-Driven

- ▶ La classe expose le modèle de données
- ▶ La classe expose le modèle du formulaire

```
class MyComponent {  
    model: any;
```

```
class MyComponent {  
    formModel: FormGroup;
```

Template-Driven vs Model-Driven

- La classe expose le modèle de données
- Binding et validation se font dans la vue
- La classe expose le modèle du formulaire
- Binding et validation se font dans la classe

```
<input type="text"  
      name="name"  
      ngModel  
      required>
```

```
class MyComponent {  
  formModel: FormGroup;  
  constructor() {  
    this.formModel = new FormGroup({  
      name: new FormControl('', Validators.required)  
    });  
  }  
}
```

Template-Driven vs Model-Driven

- La classe expose le modèle de données
- Binding et validation se font dans la vue
- la vue contient le data binding
- La classe expose le modèle du formulaire
- Binding et validation se font dans la classe
- La vue contient le mapping

```
<input type="text"  
      name="name"  
      ngModel  
      required>
```

```
<input type="text"  
      formControlName="name">
```

Avantages du Model-Driven

- La logique est dans le code et non dans le template
- Plus facile à tester
- Prêt pour de futurs scénarios (Data-Driven-Form)

L'injection de dépendances

```
import { Engine, Tires, Doors } from './shared';

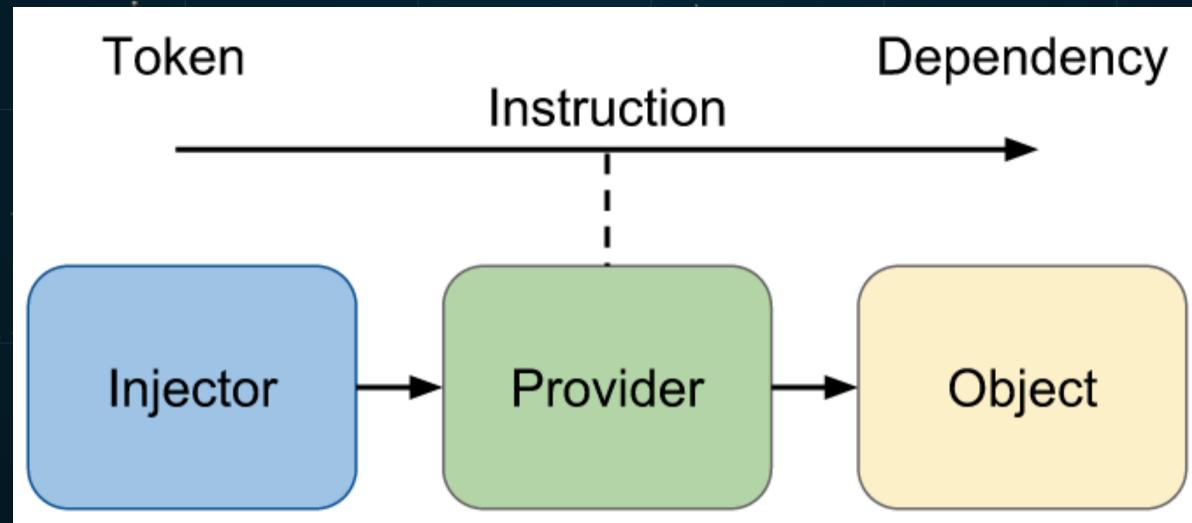
class Car {

    constructor(
        private _engine: Engine,
        private _tires: Tires,
        private _doors: Doors
    ) {
        //...
    }
}
```

Exemple

Principe de la DI

- L'**Injector** expose l'API pour **créer** des instances de dépendances.
- Le **Provider** indique à l'**Injector** comment créer la dépendance.
- La dépendance est le **type** d'objet à créer.



```
import { NameService } from './shared/';
@Component({
  providers: [NameService] ←
})
class LocalComponent {}
```

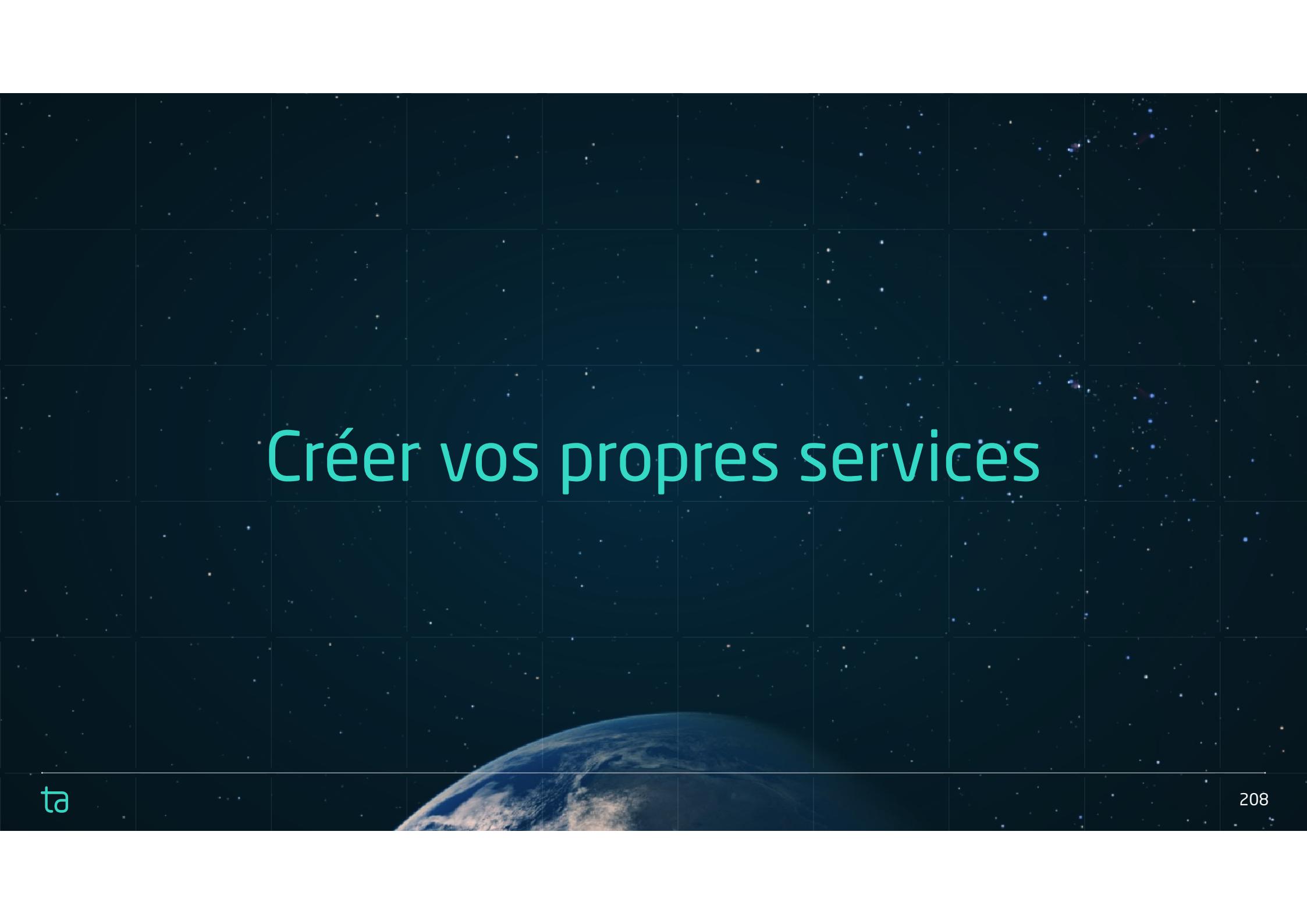
Ou

```
import { NameService } from './shared/';
@Component({
  providers: [
    {provide:NameService, {
      useClass: NameService
    }}
  ]
})
class LocalComponent {}
```

Les providers : Injection locale

```
class NameService {  
    constructor() {  
        this.name = 'Hello';  
    }  
    getName() {  
        return this.name;  
    }  
}  
  
@NgModule({ providers: [NameService] ...  
// ou  
@NgModule({ providers: [  
    {provide: NameService, {useClass: NameService}}  
] ...
```

Les providers : Injection globale



Créer vos propres services

Définition d'un service

- ▶ Une simple classe exportée
- ▶ Un décorateur `@Injectable()`

```
@Injectable()  
export class NameService {  
  
    constructor() {  
        this.name = 'Hello';  
    }  
  
    getName() {  
        return this.name;  
    }  
}
```

Exercice 17

`git checkout -f step-17`

Créez votre propre service

Exercice 17 : créez votre propre service

Actuellement, nous utilisons l'API `HttpClient` dans les composants et nous dupliquons à chaque fois l'URL de l'API.

Améliorons ceci en créant notre propre service pour nous connecter au back-end pour les opérations de CRUD sur les personnes.

- Créer un service `PeopleService` dans le répertoire `src/app/shared/services/`
 - `ng g s shared/services/people --skip-tests`
- Dans le composant, au lieu d'appeler `http.get("/people")`
 - On voudrait appeler `peopleService.fetch().subscribe(...);`
- De manière analogue, idem pour les autres opérations CRUD
- Ne pas ajouter ce service dans les providers du `@NgModule` car il est global: **explication à suivre**

Solution

```
git checkout -f step-17-solution
```

Pipes

hello

```
{ { "hello" | uppercase } }
```

HELLO

Exemples

1368730200

```
{{ "1368730200" | date }}
```

Jeudi 16 mai 2013 20H50

Exemples

1234.562342

```
{ { "1234.562342" | currency } }
```

1 234,56 €

Exemples

Syntaxe

- A la suite d'une expression

```
{{ expression | filter1 }}
```

- On peut les chaîner

```
{{ expression | filter1 | filter2 }}
```

- On peut leur passer des paramètres

```
{{ expression | filter1:param1:param2 }}
```

Pipes existants

- currency
- date
- lowercase
- uppercase
- limitTo
- async
- json
- decimal
- percent

```
this.amount = 1234.56;

<!-- 1234.56 -->
<div>{{amount}}</div>
<!-- USD1,234.56 -->
<div>{{amount | currency}}</div>
<!-- USD$1,234.56 -->
<div>{{amount | currency:"USD$"}}</div>
<!-- €1,234.56 -->
<div>{{amount | currency:"EUR":true}}</div>
```

Exemples

Pipe date

- Formate une date selon un certain format et selon une locale

```
<div>{{uneDate | date:format}}</div>
```

- Ce filtre accepte un format (string) en argument.

Exercice 18

`git checkout -f step-18`

Améliorer la lisibilité des données

Exercice 18

- ▶ Une date de naissance a été ajoutée
- ▶ Afficher cette date sous le format : dd/MM/yyyy

Solution

```
git checkout -f step-18-solution
```

```
import {Pipe} from '@angular/core';
@Pipe({
  name: 'mypipe'
})
export class MyPipe implements PipeTransform {
  transform(value: number, args: any[]) {
    return newValue;
  }
}

@NgModule({
  ...
  declarations: [MyPipe],
  ...
})
```

Créer ses propres pipes

Exercice 19

`git checkout -f step-19`

Je veux des N/A

Exercice 19

- Créer un NAPipe avec le CLI dans le répertoire shared
 - `ng g p shared/pipes/na --skip-tests`
- Afficher "N/A" s'il n'y a pas de manager

Solution

```
git checkout -f step-19-solution
```

Les directives

Directive *ngFor (rappel)

- Itère dans une collection et génère un template par élément
- **index, odd, even, last** à utiliser en alias dans des variables

```
<ul>
  <li *ngFor="let fruit of fruits; let i=index">
    <!-- répétition du template li par élément fruit -->
    {{ i }} : {{ fruit.name }}
  </li>
</ul>
```

Directive *ngSwitch

Change la structure du DOM de manière conditionnel

```
<div [ngSwitch]="expression">  
  <p *ngSwitchCase="whenExpression1">...</p>  
  <p *ngSwitchCase="whenExpression1">...</p>  
  <p *ngSwitchDefault>...</p>  
  
</div>
```

Directive *ngIf

Change la structure du DOM de manière conditionnel

```
<div *ngIf="errorCount > 0" class="error">
  {{ errorCount }} errors detected
</div>
```

C'est quoi ces étoiles ?

- pour NgFor, NgIf et NgSwitch
- sucre syntaxique
- indique que l'on utilise un <template>
- Attention à ne pas oublier les [] si vous utilisez les templates

```
<div *ngIf="errorCount > 0"></div>

//SAME AS
<template [ngIf]="errorCount > 0">
  <div></div>
</template>
```

Exercice 20

`git checkout -f step-20`

Une liste plus compacte

Exercice 20 : une liste plus compacte

- Un bouton ainsi qu'une nouvelle liste viennent d'être ajoutés dans `PeopleComponent` (la vue HTML)
- Le composant `PersonComponent` doit être modifié pour afficher les détails d'une personne ⇒ Créer une nouvelle route
- Utiliser la directive `NgSwitch` pour changer le mode d'affichage : "card", "list"
- Modifier les icônes à l'aide d'un `Nglf`
- Lors d'un clic sur `mat-list-item`, rediriger vers `/person/:id`

Solution

```
git checkout -f step-20-solution
```



Créer vos propres directives

Rappel

- Le **component** est une **directive** avec une vue
- La **directive** est une classe sans vue

3 types de directives

- **structurelle** qui modifie le layout en ajoutant, supprimant ou remplaçant des éléments du DOM
- **attribute** qui altère l'apparence ou le comportement d'un élément
- **composant** qui est juste une directive avec un template

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

@Directive({})
export class MyDirective {}
```

Définition

Plusieurs types d'invocations possibles

Juste un sélecteur CSS3

- **element-name**: pour restreindre à un élément
- **[attribute]**: pour restreindre à un attribut
- **.class**: pour restreindre à une classe
- **[attribute=value]**: restreint à un attribut avec une certaine valeur
- **:not(sub_selector)**: si l'élément ne match pas le sous-sélecteur

```
import {Directive, Input} from '@angular/core';
@Directive({
  selector: '[foobar]'
})
export class MyDirective {

  myProp: string;
  @Input('alias') myProp2: string;

}
```

Properties (rappel)

- ▶ Grâce à l'annotation `@Input()`
- ▶ peuvent être aliasées
- ▶ **Comme pour les Composants**

```
import {Directive, Input} from '@angular/core';
@Directive({
  selector: '[foobar]'
})
export class MyDirective {

  myProp: string;
  @Input('alias') myProp2: string;

}
```

Eléments DOM

- `this._element.nativeElement` référence directe de l'élément DOM
- `this._renderer` pour interagir avec le DOM

```
import {  
    Directive, ElementRef, Renderer2  
} from '@angular/core';  
  
@Directive({  
    selector: '[foobar]'  
)  
export class MyDirective {  
    constructor(  
        private _element: ElementRef,  
        private _renderer: Renderer2  
    ) {}  
}
```

Interaction avec le DOM

- Préférez l'utilisation du **Renderer** au lieu du **ElementRef**
- Pas de dépendance direct au DOM
- Permet d'exécuter l'application dans d'autres environnements

:/

```
this._element.nativeElement.style.color = 'orange';  
this._element.nativeElement.innerHTML = ':/';
```

:)

```
this._renderer.setStyle(this._element.nativeElement, 'color', '#0f0');  
  
this._renderer.setProperty(  
  this._element.nativeElement, 'innerHTML', ':)'  
);
```

Interaction avec le DOM

Événements (rappel)

- **@HostListener** pour écouter des évènements
- **@Output()** pour propager des évènements
- **Comme pour les Composants**

```
@Directive({})
export class MyDirective {

  @Output() somethingChange$: EventEmitter<any>;

  constructor(){
    this.somethingChange$ = new EventEmitter();
  }

  @HostListener('click', '$event')
  onClick($event) {
    this.somethingChange$.emit({$event, data});
  }
}
```

Exercice 21

`git checkout -f step-21`

Manipuler le DOM

Exercice 21 : manipuler le DOM

- Créer une directive `NwtBadge` dans le répertoire `shared/badge`
 - `ng g d shared/directives/badge --skip-tests`
- Cette directive affiche une icône si une personne est un manager
 - `<i class="material-icons">supervisor_account</i>`
- Elle doit pouvoir s'utiliser comme ceci dans la `vue liste` du template du `PeopleComponent`
 - ``

Solution

```
git checkout -f step-21-solution
```



Fin de la 3ème journée et du développement Frontend

Si vous avez apprécié la formation, envoyez un Tweet !

#NewWebTechnologies #Angular @_akanass_ @TaDaweb