WM

ESIR2

WEB ENGINEERING : FRONTEND

2024 - 2025

P L A N

Généralités et rappel

Développement Frontend

HTML, CSS et JavaScript

Utilisation de Framework : Angular

Conclusion

ORGANISATION

→ Suite cours backend de Stéphanie Challita (WM – ESIR2)

→ 5 séances de CM (7,5h)

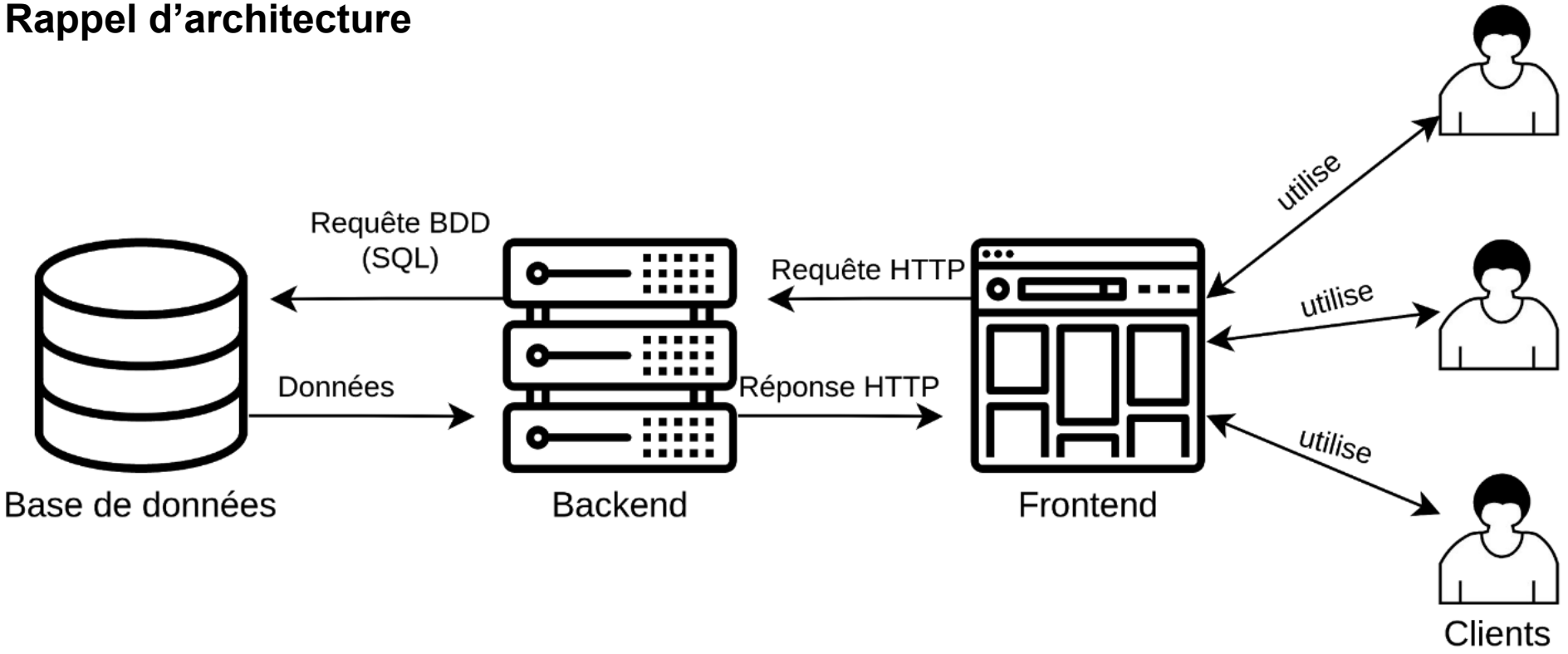
→ 2 séances de TP sur Angular (3h)

→ 6 séances projet final (9h)

→ 1 séance de TP pour l'évaluation du projet (3h)

G É N É R A L I T É S

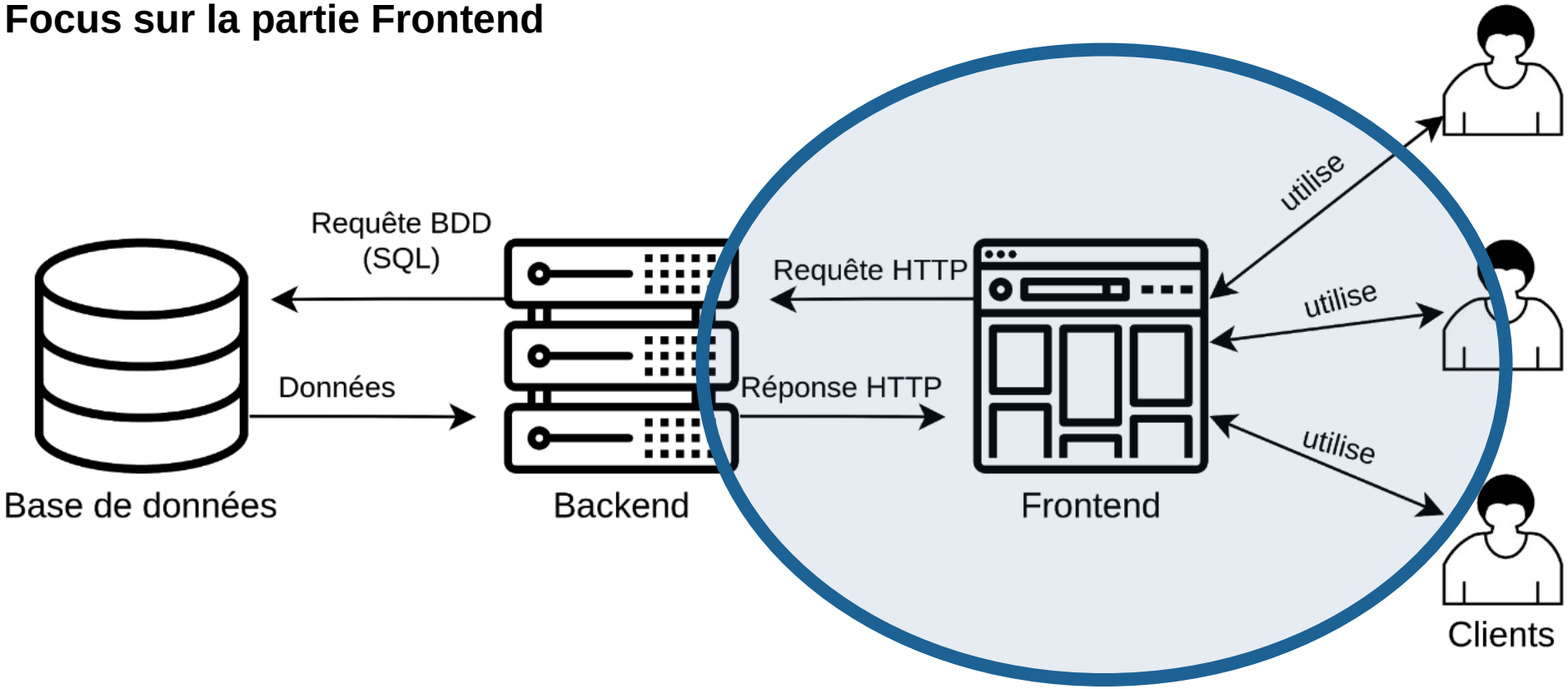
Rappel d'architecture



(Stéphanie Chalitta)

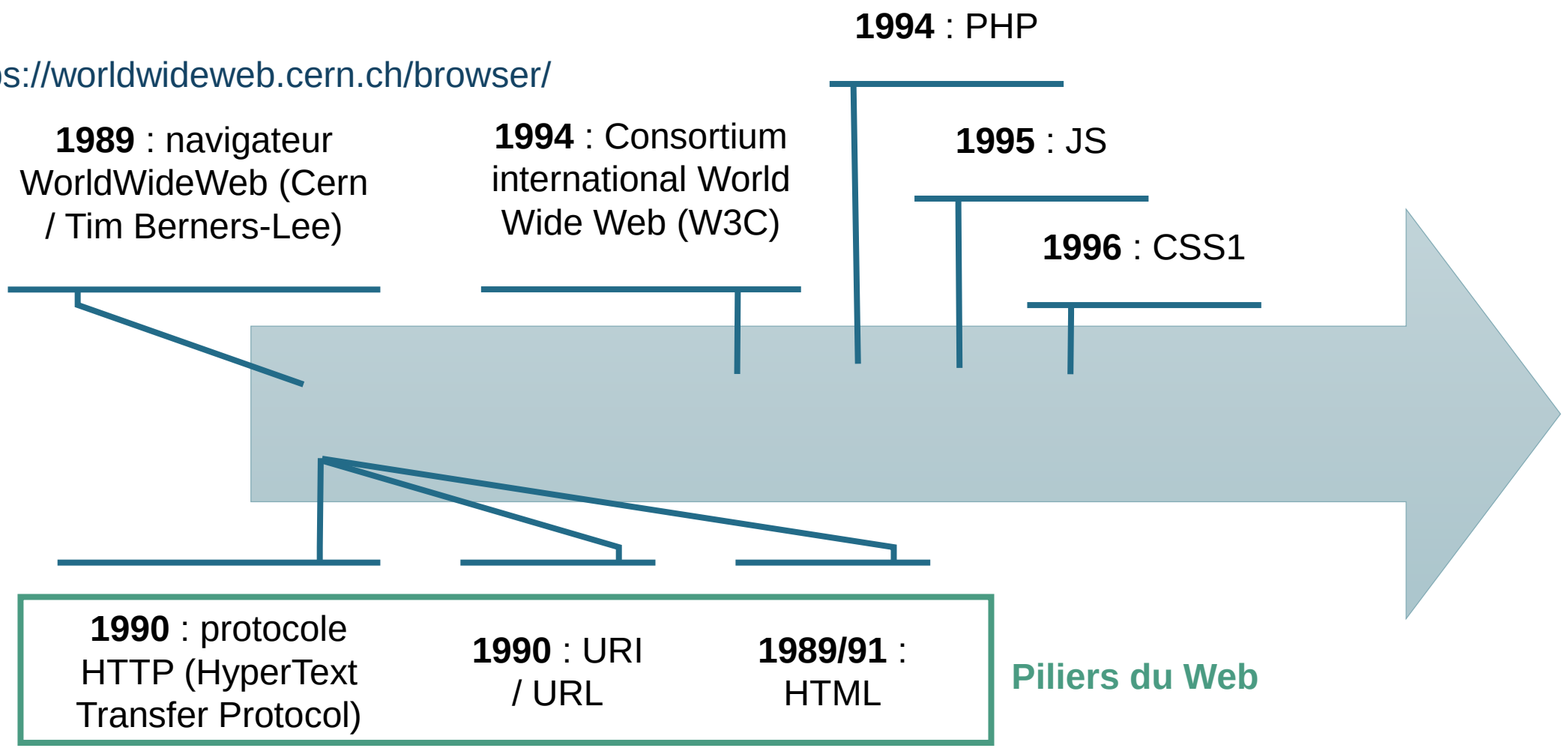
GÉNÉRALITÉS

Focus sur la partie Frontend



GÉNÉRALITÉS

https://worldwideweb.cern.ch/browser/



Piliers du Web

G É N É R A L I T É S

Evolution du Web

→ Séparation mise en forme (CSS)

→ Pages statiques vers interactions dynamiques

→ Cloud computing, Web sémantique, Web embarqué, etc.

→ HTML5 (Conteneur d’applications complexes)

G É N É R A L I T É S

Evolution du Web

→ **Web 2.0** (web collaboratif, wiki, blog, etc.)

Web 3.0 avec le web sémantique, l'IA et les objets connectés

→ **Moyens techniques**

Langage de scripts

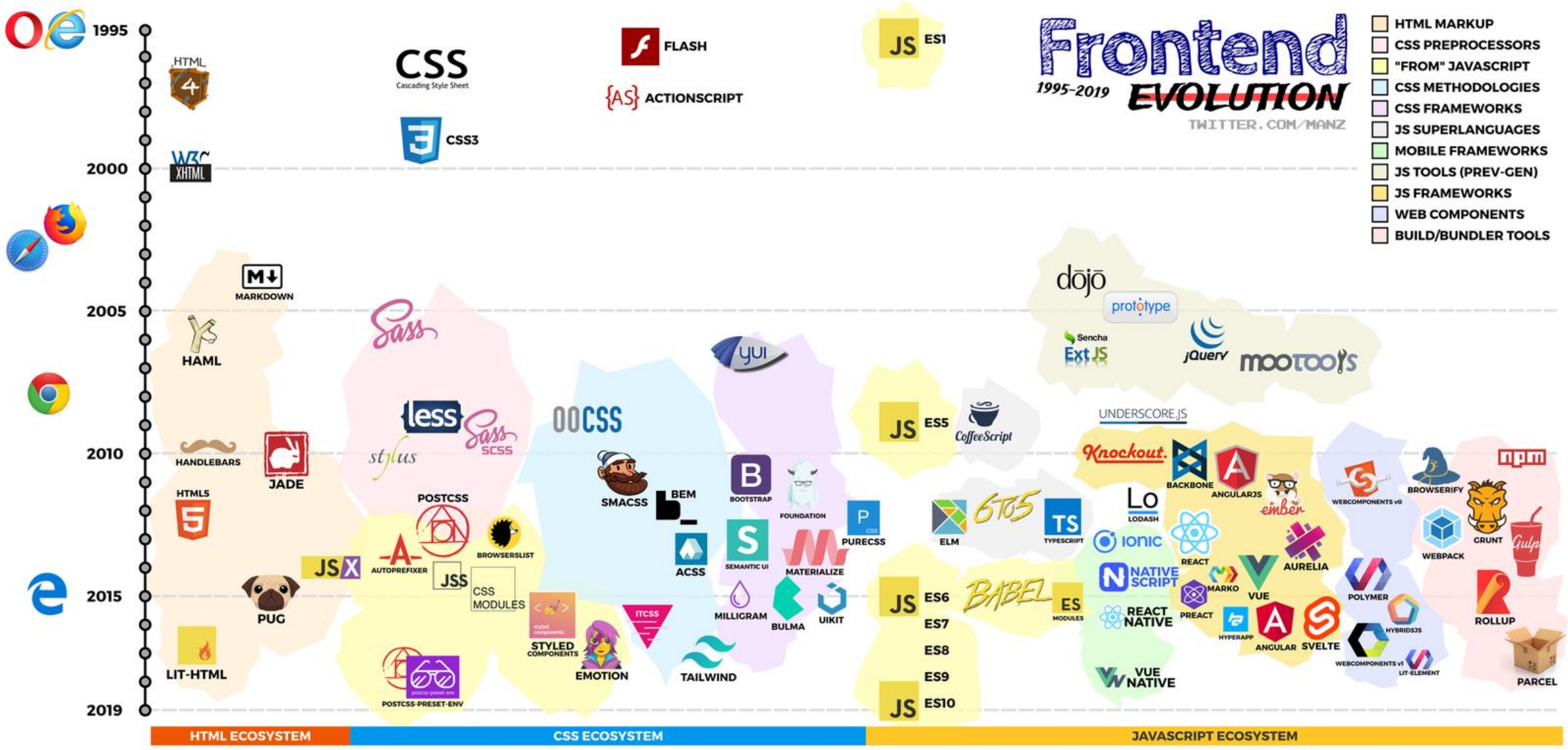
Côté serveur (PHP, ASP, C#, etc.)

Côté client (Javascript, flash, etc.)

→ **Services Web**

Échange de données entre applications (HTML / XML, JSON)

GÉNÉRALITÉS



G É N É R A L I T É S

Pourquoi utiliser des frameworks ?

→ Simplifier la vie du développeur et réduire le coup

Différents types :

→ Basé sur le style (Boostrap, Tailwind)

→ Basé sur des composants (Vue, Angular, React)

OBJECTIFS

→ Fondamentaux sur le frontend web (Single Page Application)

→ Rappel sur les technologies de base du web (HTML/CSS/JS)

→ Notion de composants web

→ Panorama des frameworks JavaScript (Angular)

DÉVELOPPEMENT FRONTEND

Les piliers du web

Architecture front-end : MPA - SPA

DÉVELOPPEMENT FRONTEND

→ **HTML / CSS / JavaScript**

→ De nombreuses variations à JavaScript existent.

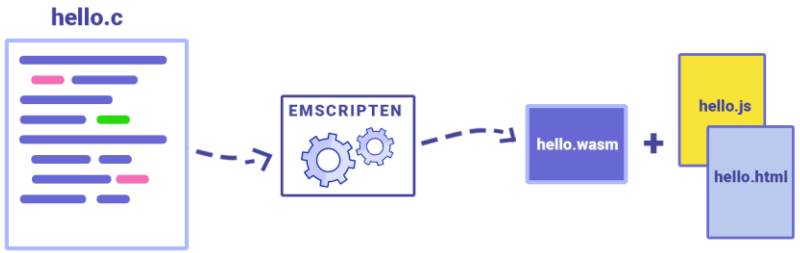
DÉVELOPPEMENT FRONTEND

Création du nouveau standard du web en 2019 : WebAssembly (wasm)

→ **Ne remplace pas** JavaScript mais devient complémentaire.

→ Bas niveau (de style assembleur) avec un format binaire (Code *C++/Rust* compilé au format WebAssembly dans un fichier binaire *.wasm* via *Emscripten/wasm-pack*)

→ Permet un gain de performance (format plus léger qu'un fichier JS, déjà compilé en amont, utile pour les tâches gourmandes en ressources, etc.)

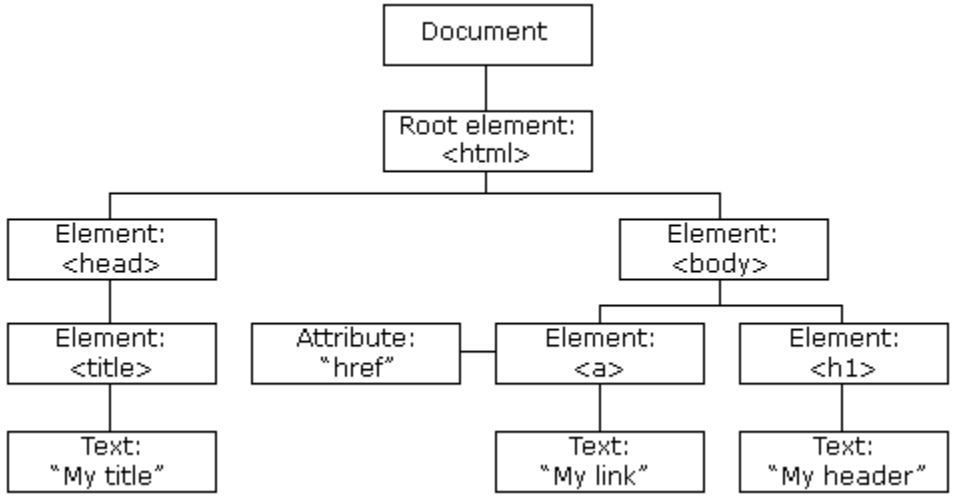


DÉVELOPPEMENT FRONTEND

DOM (Document Object Model)

→ Représentation *objet* des données qui composent la structure et le contenu d'un document sur le web.

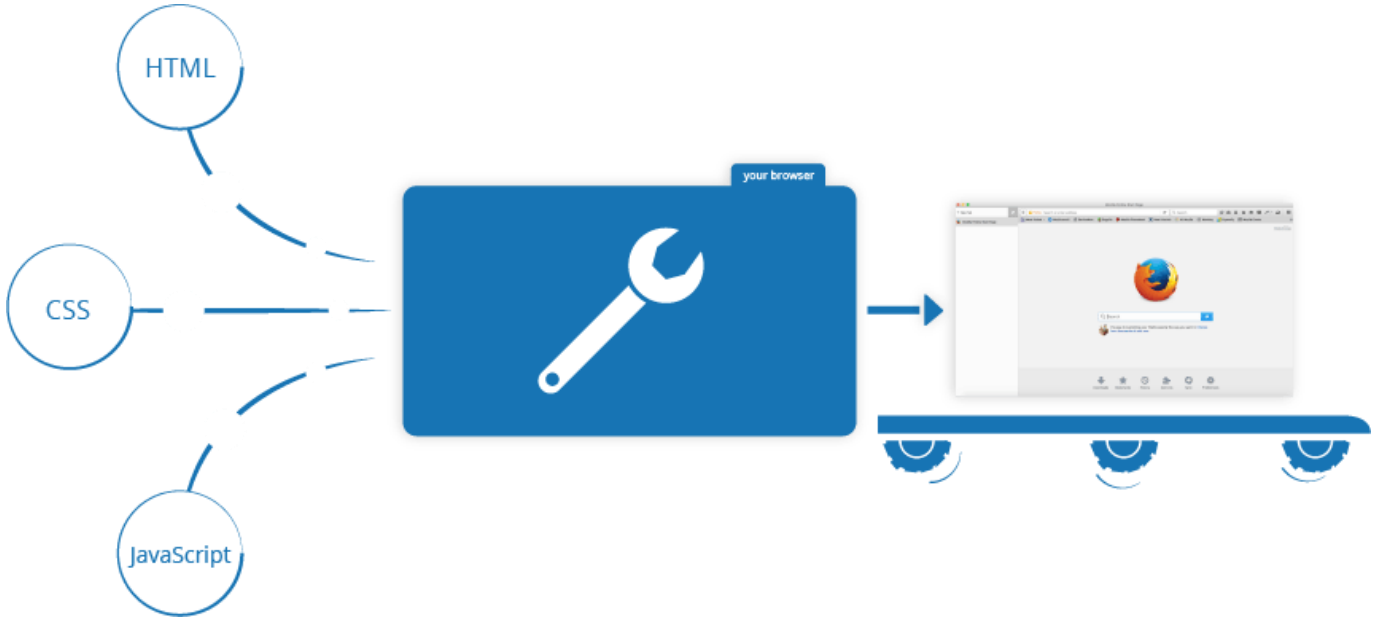
→ Il peut être manipulé à l'aide d'un langage script comme JavaScript.



DÉVELOPPEMENT FRONTEND

→ Lorsque la page web se charge dans un navigateur, **les codes HTML, CSS et JavaScript s'exécutent dans un environnement**.

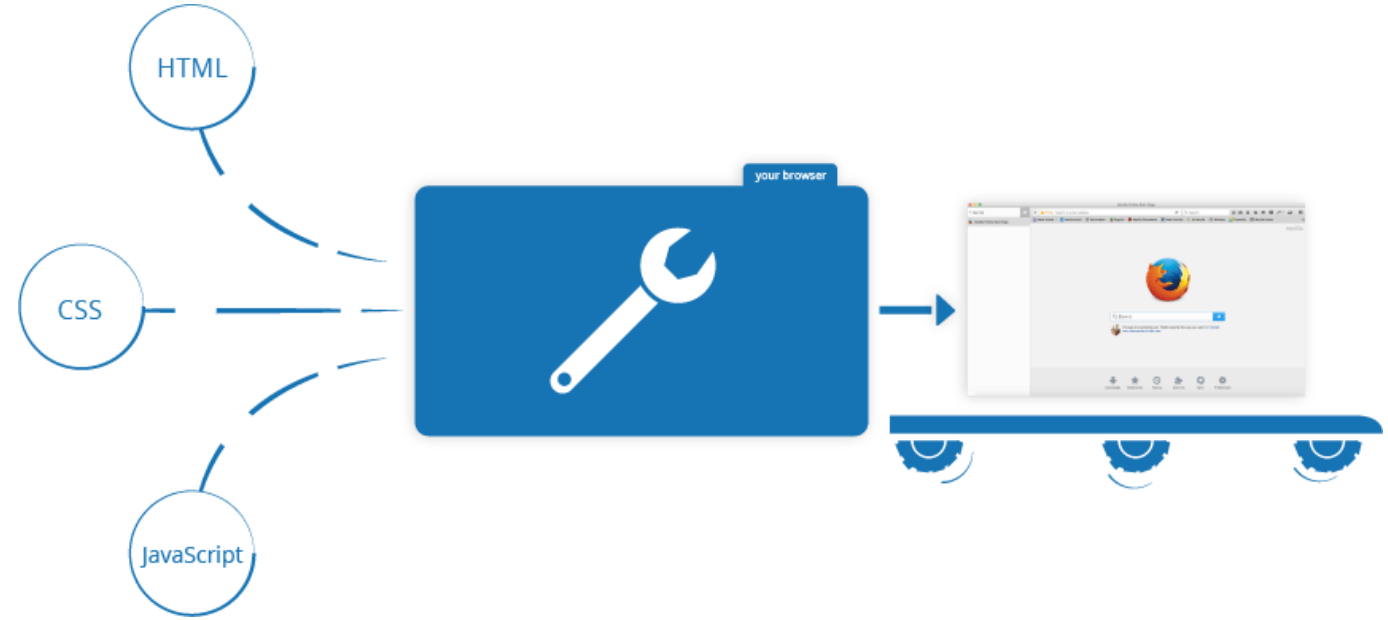
→ Le **JavaScript** est exécuté par le moteur JavaScript du navigateur, **après** que le HTML et le CSS ont été assemblés et combinés en une page web.



DÉVELOPPEMENT FRONTEND

→ Possibilité d'utiliser différentes technologies pour créer des applications web

- VanillaJS
- Langages existants (GWT)
- Nouveaux langages (DART)
- Extension de JS (TypeScript, Coffee)
- Frameworks JS (Angular, Vue, React)

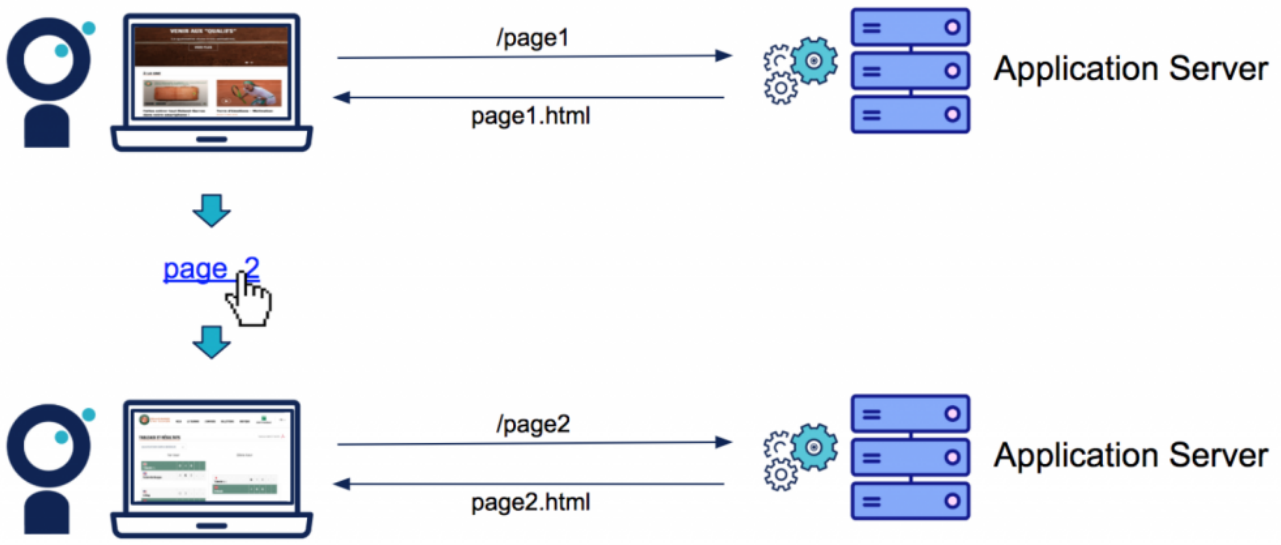


MULTIPAGES APPLICATION VS SINGLE PAGE APPLICATION

Multiple page application (MPA)

→ Chaque action de l'utilisateur déclenche une requête HTTP vers le serveur.

→ Rechargement complet de la page même si une partie du contenu reste inchangée.

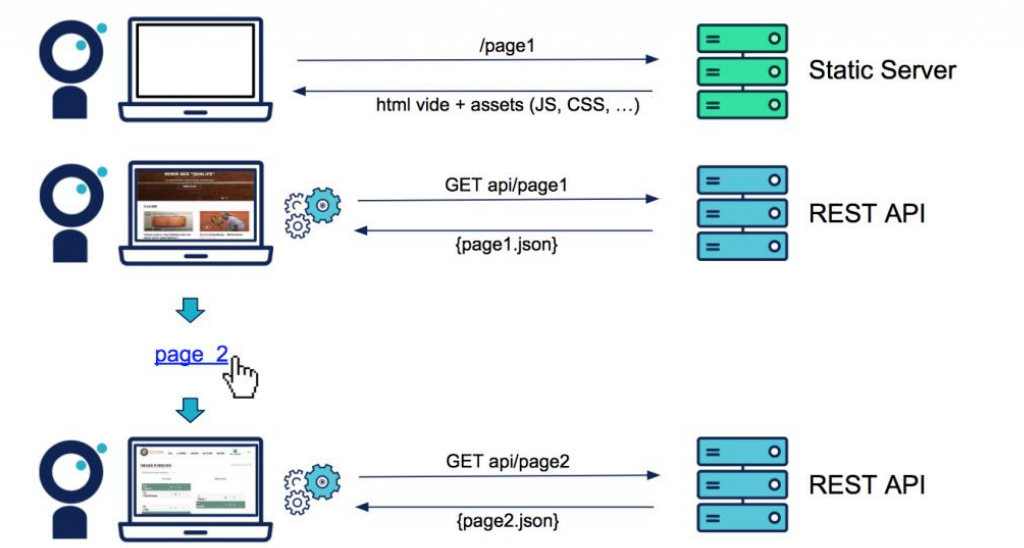


MULTIPAGES APPLICATION VS SINGLE PAGE APPLICATION

Single page application (SPA)

L'ensemble des éléments de l'application est chargé sur le navigateur. Tous les templates du site sont donc pré-chargés.

→ Navigation côté client uniquement (émulation d'une navigation).



MULTIPAGES APPLICATION VS SINGLE PAGE APPLICATION

Single page application (SPA)

→ Type de pattern apparu avec l'arrivée de nouvelle technologie (AJAX en 2004, jQuery en 2006, Google Chrome avec un nouveau moteur JavaScript V8, etc.).

→ Utilisé par les frameworks AngularJS et Backbone.js (2010), Ember (2011), React (2013), Vue (2014) ou encore Angular (2016).

H T M L

Langage de balises

Anatomie d’un document HTML

Rappel formulaire

Notion de template

HTML5 : GÉNÉRALITÉS

Cours plus détaillé ?

HTML pour **HyperText Markup Language**.

→ Fichier avec extension **.html**

→ Langage de **balise** (structure et contenu)

→ Evolution du langage (HTML2 en 1996, HTML3/4 en 1997, HTML5 en 2014).

HTML sert à définir la **structure** d'une page web.

→ Le navigateur lit le fichier et interprète les balises pour faire l'affichage

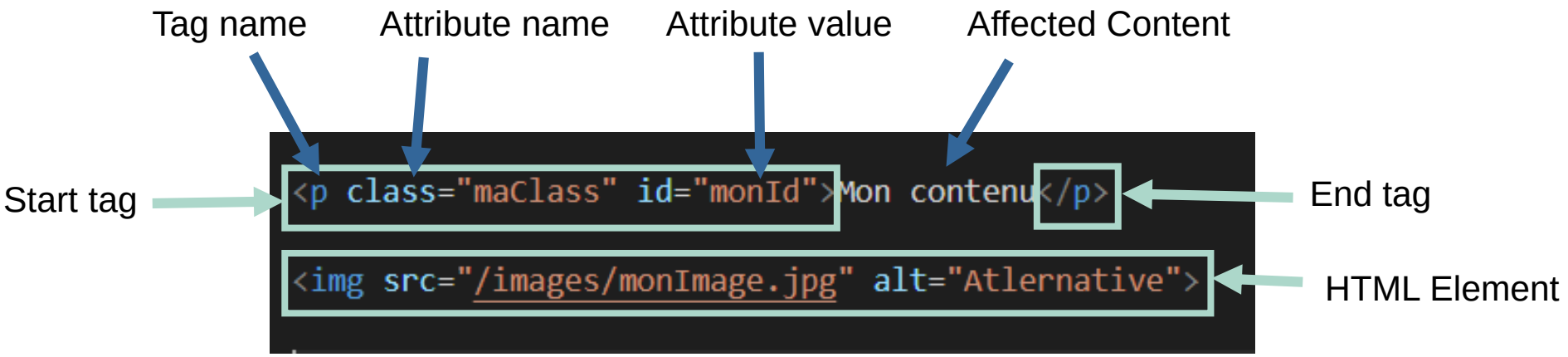
https://developer.mozilla.org/en-US/play

https://developer.mozilla.org/fr/docs/Web/HTML/Element

HTML5 : GÉNÉRALITÉS

La structure est indiquée à l'aide de balise :

→ Le **nom de la balise** indique le **type d'élément** que l'on ajoute (un titre, un paragraphe, une image, une liste, etc.).



https://developer.mozilla.org/fr/docs/Web/HTML/Element

HTML5 : STRUCTURE D'UNE PAGE

→ Structure simple d’une page HTML5 :

```html
<!DOCTYPE html>
<html lang="fr">
  <head>
    <!-- Metadata -->
    <meta charset="utf-8">
  </head>
  <body>
    <!-- Contenu -->
  </body>
</html>
```

HTML5 : STRUCTURE D'UNE PAGE

```html
<!DOCTYPE html>
<html lang="fr">
  <head>
    <!-- Metadata -->
    <meta charset="utf-8">
  </head>
  <body>
    <!-- Contenu -->
  </body>
</html>
```

Balise **doctype** spécifie le type de document :

→ Nécessaire pour préciser la version et la variante de la norme HTML afin que le navigateur interprète correctement le code fourni

→ Obligatoire pour valider son document (W3C Markup Validation Service)

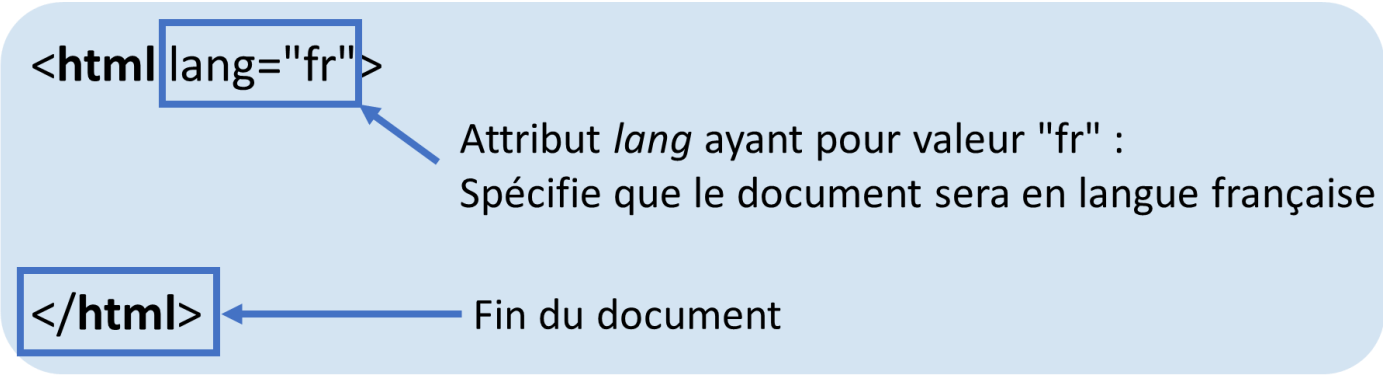
→ Balise simplifiée en HTML 5 :

```
<!DOCTYPE html>
```

HTML5 : STRUCTURE D'UNE PAGE

```html
<!DOCTYPE html>
<html lang="fr">
  <head>
    <!-- Metadata -->
    <meta charset="utf-8">
  </head>
  <body>
    <!-- Contenu -->
  </body>
</html>
```

→ La balise **<html>** spécifie le début du document



HTML5 : STRUCTURE D'UNE PAGE

```html
<!DOCTYPE html>
<html lang="fr">
  <head>
    <!-- Metadata -->
    <meta charset="utf-8">
  </head>
  <body>
    <!-- Contenu -->
  </body>
</html>
```

La balise **<head>** permet de spécifier :

→ Des **métadonnées** sur le document (utilisées par les navigateurs ou les moteurs de recherche pour l'indexation).

→ Des **scripts** ou **lien vers des scripts externes** (en javascript par exemple) contenant du code utilisable dans la partie body.

→ Des liens vers des feuilles de style permettant de gérer la mise en forme du document.

L'entête n'affiche aucun contenu dans la page web

HTML5 : STRUCTURE D'UNE PAGE

```html
<!DOCTYPE html>
<html lang="fr">
  <head>
    <!-- Metadata -->
    <meta charset="utf-8">
  </head>
  <body>
    <!-- Contenu -->
  </body>
</html>
```

La balise **<body>** permet de définir la **structure** et le **contenu** du document via l'utilisation d'autres balises.

→ Balises structurantes

→ Balises de contenu

Le corps du document peut contenir :

→ Un titre, sous titre, etc.

→ Un paragraphe,

→ Listes ordonnées (ou non),

→ Images,

→ Lien hypertextes,

→ Formulaires,

→ Composant, etc.

HTML5 : STRUCTURE D'UNE PAGE

Des balises apportent de la sémantique à la structure de la page (HTML5):

→ header

→ footer

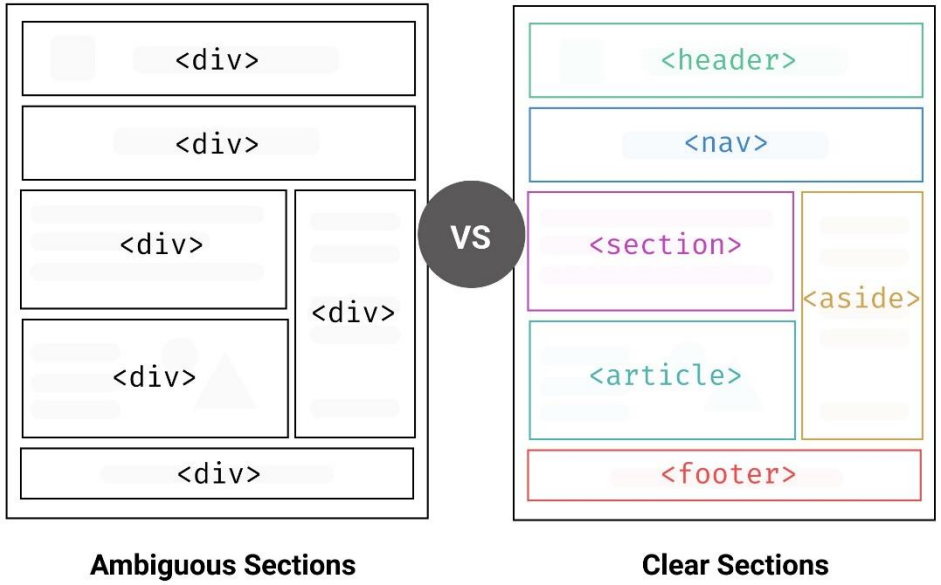
→ nav

→ section

→ article

→ aside

→ main



Attention, la mise en page et mise en forme est réalisée via le CSS !

HTML5 : STRUCTURE D'UNE PAGE

Le contenu d'un site HTML est organisé via des balises de contenu :

→ Un titre ou sous titre <h1></h1>

→ Un paragraphe

→ Une liste

→ Un tableau <table></table>

→ Une image <img />

→ Un lien hypertexte <a>

En HTML 5, il existe de nombreuses balises (>100)

https://developer.mozilla.org/fr/docs/Web/HTML/Element
https://html.spec.whatwg.org/multipage/

HTML5 : IDENTIFIANT

Il est possible d'associer un identifiant unique à une balise

→ Attribut **id**

→ Cela permet de référencer certaines balises et de manipuler plus facilement ces éléments (via un langage de programmation)

```
<p id="monParagraphe" > </p>
```

```
<img id="logo" src = "image.png" > </img>
```

HTML5 : CLASSE

Il est possible d'associer une classe à plusieurs balises pour les lier

→ Attribut **class**

→ Utile pour appliquer une mise en forme (CSS) identique à plusieurs balises et pour manipuler des éléments via un langage de programmation.

```
<p class="paragraphe1" ></p>
<p class="paragraphe1" ></p>
```

HTML5 : FORMULAIRE

```
<form action="url"   method="POST|GET"   name="nomForm" >
        ...
</form>
```


→ Utilisation des balises **<form></form>** pour déclarer un formulaire

→ Attribut **action** : page ou url vers laquelle le visiteur sera redirigé après soumission du formulaire (traitement des informations)

→ Attribut **name** optionnel

→ Le corps du formulaire contient les balises représentant les composants de saisie (champ texte, liste, etc.)

https://developer.mozilla.org/fr/docs/Learn/Forms

HTML5 : FORMULAIRE

```
<form action="url"   method="POST|GET"   name="nomForm" >
        …
</form>
```


→ Attribut **method** permet de définir la méthode de transmission des paramètres.

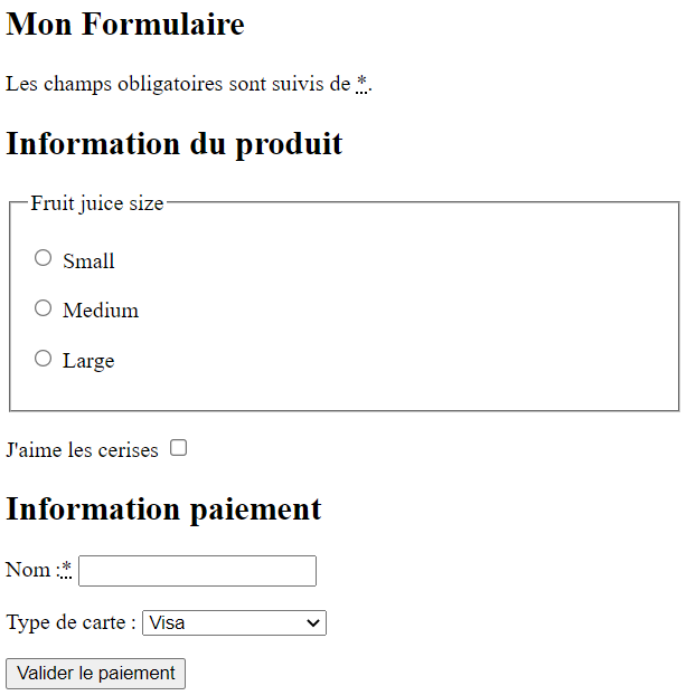
- **POST** : les paramètres sont transmis dans le corps de la requête HTTP
- **GET** : les paramètres sont transmis dans l'URL de la page

http://www.esir.com/index.php?param1=val1&param2=20

https://developer.mozilla.org/fr/docs/Learn/Forms

HTML5 : FORMULAIRE

→ Formulaire : différents composants




```html
<fieldset>
    <legend>Fruit juice size</legend>
    <p> <input type="radio" name="size" id="size_1" value="small"> <label for="size_1">Small</label> </p>
    <p> <input type="radio" name="size" id="size_2" value="medium"> <label for="size_2">Medium</label> </p>
    <p> <input type="radio" name="size" id="size_3" value="large"> <label for="size_3">Large</label> </p>
</fieldset>
```

```html
<label for="taste_1">J'aime les cerises</label>
<input type="checkbox" id="taste_1" name="taste_cherry" value="1">
```

```html
<label for="username">Nom :<abbr title="required">*</abbr></label>
<input id="username" type="text" name="username" required>
```

```html
<select id="card" name="usercard">
    <option value="visa">Visa</option>
    <option value="mc">Mastercard</option>
    <option value="amex">American Express</option>
</select>
```

```html
<button type="submit">Valider</button>
```

https://developer.mozilla.org/fr/docs/Learn/Forms

HTML5 : FORMULAIRE

Formulaire : validation des données côté HTML

Champ *required*

```
<label for="choose">Login ?</label>
<input id="choose" name="i_like" required>
```

Expression régulière

```
<label for="choose">Mot de passe ? </label>
<input id="choose" name="mdp" required pattern="[A-F][0-9]{5}">
```

Champ *input*

```
<label for="t2">Adresse électronique ?</label>
<input type="email" id="t2" name="email">
```

→ API de validation des contraintes disponible en JavaScript

HTML5 : FORMULAIRE

Formulaire : envoi du formulaire

→ A travers l’attribut action du formulaire.

→ En utilisant des requêtes AJAX ou l'API Fetch (e.g. via un objet FormData avec XmlHttpRequest).

HTML5 : TEMPLATE

Balise template :

→ Mécanisme utilisé pour stocker côté client du contenu HTML qui ne doit pas être affiché lors du chargement de la page.

→ Permet de stocker des modèles de code HTML pouvant être clonés et collés dans un document à l'aide de scripts JS (structure répétitive).


```
<template id="monParagraphe">
    <h2>JXC</h2>
</template>
```

```
let template = document.getElementById('monParagraphe');
let templateContent = template.content.cloneNode(true);
document.body.appendChild(templateContent);
```

DESIGN

CSS

Responsive design

Pré et Post-processeurs CSS (SASS, LESS)

Framework CSS (Bootstrap)

DESIGN

→ **Le webdesign doit être réfléchi** : UX et UI designer (e.g. expérience utilisateur à travers l'ergonomie et la navigation du site, réalisation de maquettes pour définir la chartre graphique).

→ **A éviter :** un design complexe ou chargé, des polices d'écriture trop petites, des couleurs trop marquées, une navigation peu ergonomique, des pop-ups ou bannières trop agressives.

C S S

Cours plus détaillé ?

→ **C**ascading **S**tyle **S**heets : code utilisé pour mettre en forme une page web.

→ Insertion d'une feuille de style dans un fichier html :

```
<link rel="stylesheet" href="css/monCSS.css" />
```


→ Code CSS inline **à éviter** :

```
<h1 style="color: blue;background-color: yellow;border: 1px solid black;">Hello World!</h1>
```

→ Possibilité d'appliquer un style à tous les éléments du même type, plusieurs éléments, un élément particulier.

```
p {
  color: red;
  width: 500px;
  border: 1px solid black;
}
```

```
p,li,h1 {
  color: red;
}
```

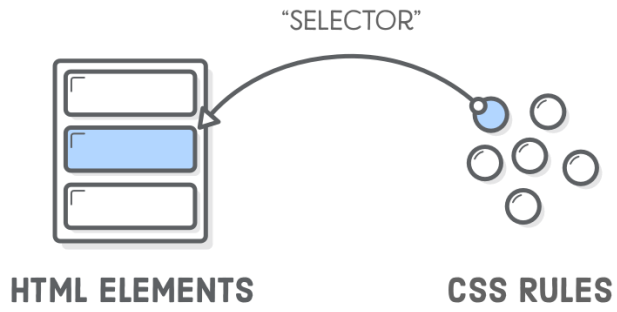
CSS : STRUCTURE

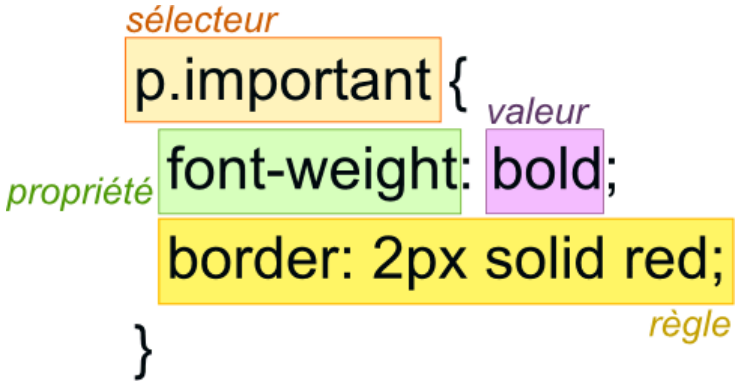
→ Sélecteur (élément HTML)

→ Déclaration (règle simple qui détermine les propriétés de l'élément mis en forme)

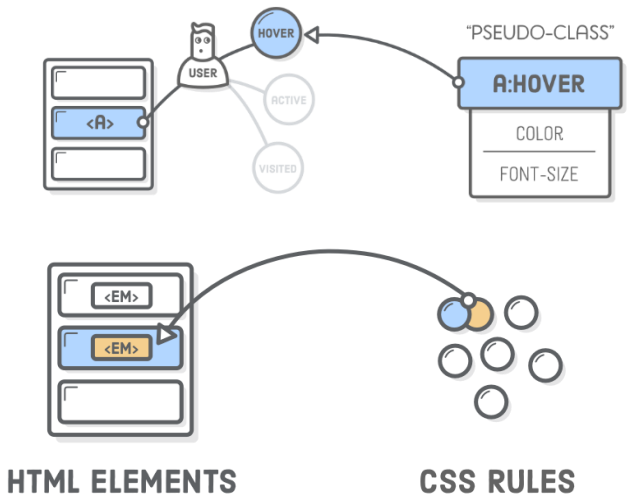
→ Propriétés

→ Valeur de la propriété





CSS : SÉLECTEUR




```css
#maDiv {
  background-color: #15DEA5;
}
.button {
  background-color: #DB464B;
}
p .maClass {
  background-color: steelblue;
}
a:hover{
  color: cornflowerblue;
}
```

Selector	Example
Type selector	`h1 { }`
Universal selector	`* { }`
Class selector	`.box { }`
id selector	`#unique { }`
Attribute selector	`a[title] { }`
Pseudo-class selectors	`p:first-child { }`
Pseudo-element selectors	`p::first-line { }`
Descendant combinator	`article p`
Child combinator	`article > p`
Adjacent sibling combinator	`h1 + p`
General sibling combinator	`h1 ~ p`

RESPONSIVE DESIGN

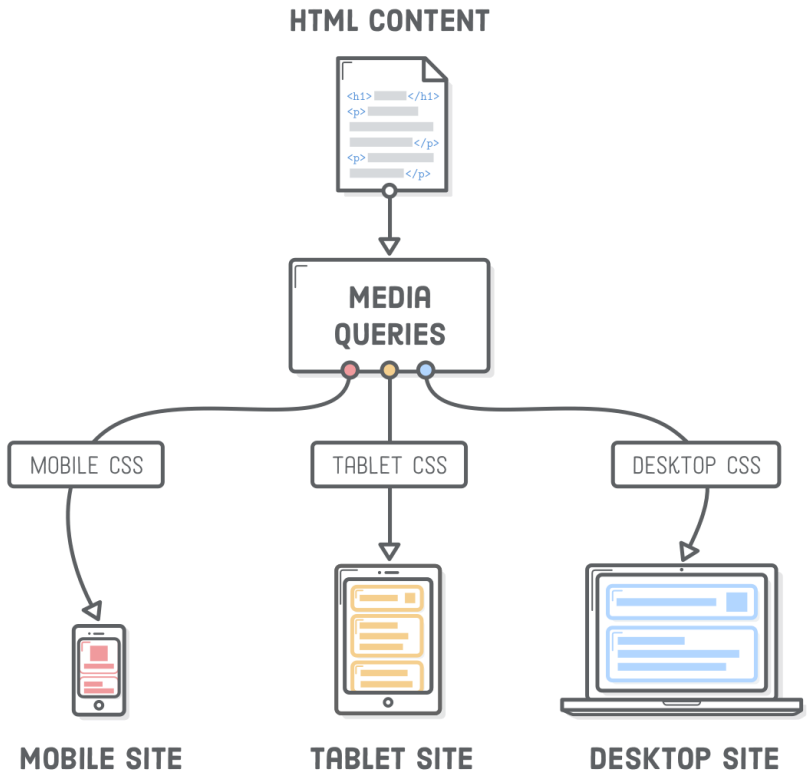
Comment adapter le design aux différents types de supports et résolution d'écran ?

→ Une solution : les **media queries**

Moyen d'appliquer conditionnellement des règles CSS.

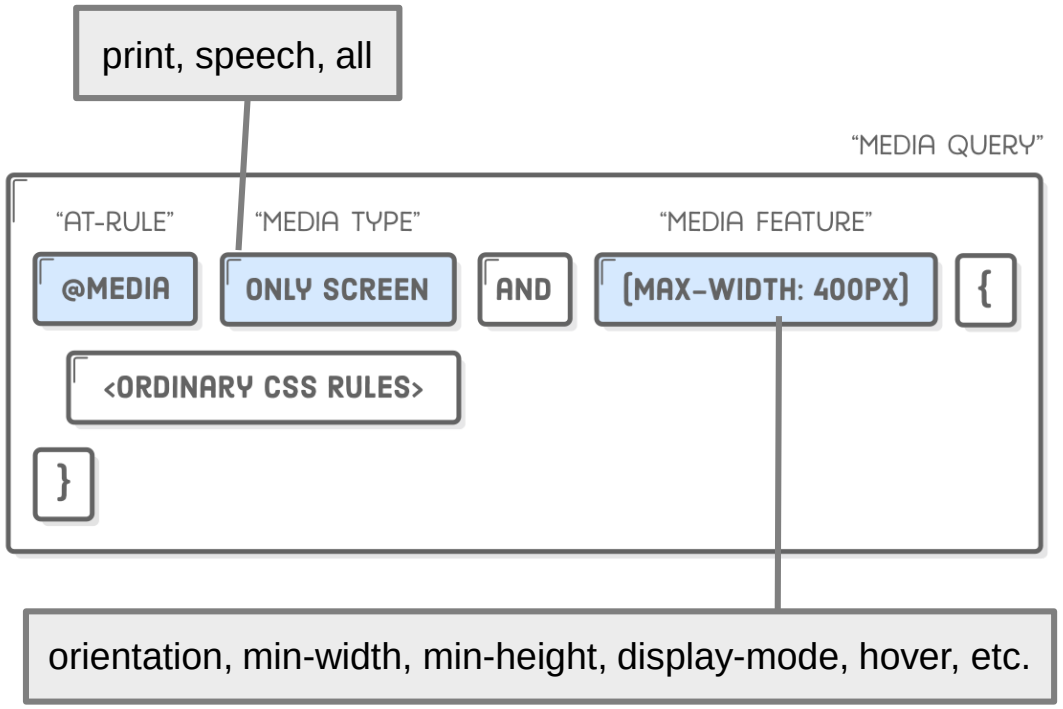
Utilisation des mêmes éléments HTML

→ Généralement débuter par la mise en page mobile.



RESPONSIVE DESIGN

→ **Media query**




```css
* {
  margin: 0;
  padding: 0;
  box-sizing: border-box;
}

/* Mobile Styles */
@media only screen and (max-width: 400px) {
  body {
    background-color: #F09A9D; /* Red */
  }
}

/* Tablet Styles */
@media only screen and (min-width: 401px) and (max-width: 960px) {
  body {
    background-color: #F5CF8E; /* Yellow */
  }
}

/* Desktop Styles */
@media only screen and (min-width: 961px) {
  body {
    background-color: #B2D6FF; /* Blue */
  }
}
```

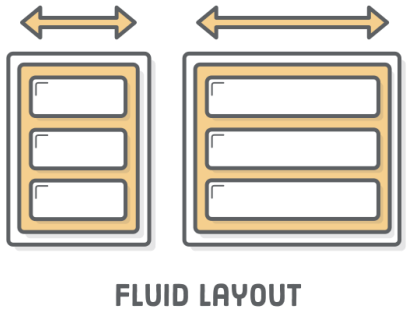
RESPONSIVE DESIGN

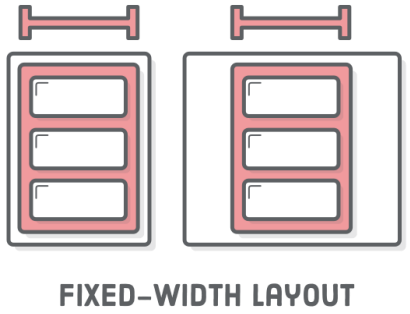
→ Zoom automatique sur les navigateurs mobiles pour ajuster la page dans la largeur.

→ Possibilité de le désactiver si le site est responsive :

```
<meta name="viewport" content="width=device-width, initial-scale=1">
```


→ Attention aux unités de longueur pour un design adaptable (pour la taille du texte par exemple), au type de positionnement des éléments et leur conteneur (Grid, FlexBox, etc.)





DESIGN

CSS

Responsive design

Pré et Post-processeurs CSS (SASS, LESS)

Framework CSS (Bootstrap)

PRE-PROCESSEUR : SASS

→ SASS (**S**yntactically **A**wesome **S**tyle**s**heets) permet d'ajouter à CSS diverse fonctionnalités permettant d'organiser de manière plus maintenable les feuilles de style.

→ Commande pour compiler les fichiers sass en css

```
sass --watch input.scss output.css
```

Outils à disposition pour organiser et gérer les styles :

→ Variables

→ Imbrication des sélecteurs

→ Fonctions

→ Directives @import

PRE-PROCESSEUR : SASS

→ Hiérarchie entre les sélecteurs

```
nav ul {
  margin: 0;
  padding: 0;
  list-style: none;
}
nav li {
  display: inline-block;
}
nav a {
  display: block;
  padding: 6px 12px;
  text-decoration: none;
}
```

```
nav {
    ul {
      margin: 0;
      padding: 0;
      list-style: none;
    }

    li { display: inline-block; }

    a {
      display: block;
      padding: 6px 12px;
      text-decoration: none;
    }
  }
```

PRE-PROCESSEUR : SASS

→ Variables (disponible également en CSS)

```css
body {
  font: 100% Helvetica, sans-serif;
  color: #333;
}
```

```css
:root{
    --main-color: #333;
}

body {
    font: 100% Helvetica, sans-serif;
    color: var(--main-color);
}
```

```scss
$font-stack: Helvetica, sans-serif;
$primary-color: #333;

body {
  font: 100% $font-stack;
  color: $primary-color;
}
```

PRE-PROCESSEUR : SASS

→ Utilisation de modules

```css
body {
  font: 100% Helvetica, sans-serif;
  color: #333;
}

.inverse {
  background-color: #333;
  color: white;
}
```

```scss
// _base.scss
$font-stack: Helvetica, sans-serif;
$primary-color: #333;

body {
font: 100% $font-stack;
color: $primary-color;
}
```

```scss
// styles.scss
@use 'base';

.inverse {
  background-color: base.$primary-color;
  color: white;
}
```

PRE-PROCESSEUR : SASS

→ Utilisation des mixins

```
.info {
  background: DarkGray;
  box-shadow: 0 0 1px rgba(169, 169, 169, 0.25);
  color: #fff;
}

.alert {
  background: DarkRed;
  box-shadow: 0 0 1px rgba(139, 0, 0, 0.25);
  color: #fff;
}

.success {
  background: DarkGreen;
  box-shadow: 0 0 1px rgba(0, 100, 0, 0.25);
  color: #fff;
}
```

```
@mixin theme($theme: DarkGray) {
    background: $theme;
    box-shadow: 0 0 1px rgba($theme, .25);
    color: #fff;
  }

  .info {
    @include theme;
  }
  .alert {
    @include theme($theme: DarkRed);
  }
  .success {
    @include theme($theme: DarkGreen);
  }
```

L E S S

→ LESS (**Le**aner **S**tyle **S**heets) est un préprocesseur CSS, influencé par SASS.

→ Permet également une plus grande facilité de gestion des feuilles de styles et supprime les répétitions de code.

→ Concept similaires (imbrication, variable, héritages, mixins, etc.)

→ Repose sur une syntaxe plus naturelle (ressemblant à CSS) par rapport à SASS qui utilise des mots clés (@use, @include, etc.).

BOOTSTRAP

→ Framework open source utilisant Sass, développé au début par une équipe de Twitter.

→ Compatible avec les différents navigateurs.

→ Prend en charge la conception réactive et propose des modèles de conception prédéfinis.

→ Pour inclure Bootstrap : CSS + JS (Bootstrap Bundle avec Popper)

```
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css" rel="stylesheet"
  integrity="sha384-1BmE4kWBq78iYhFldvKuhfTAU6auU8tT94WrHftjDbrCEXSU1oBoqyl2QvZ6jIW3" crossorigin="anonymous">
```

```
<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/js/bootstrap.bundle.min.js"
  integrity="sha384-ka7Sk0Gln4gmtz2MlQnikT1wXgYsOg+OMhuP+IlRH9sENBO0LRn5q+8nbTov4+1p" crossorigin="anonymous"></script>
```

→ Développé au début pour mobile (responsive design)

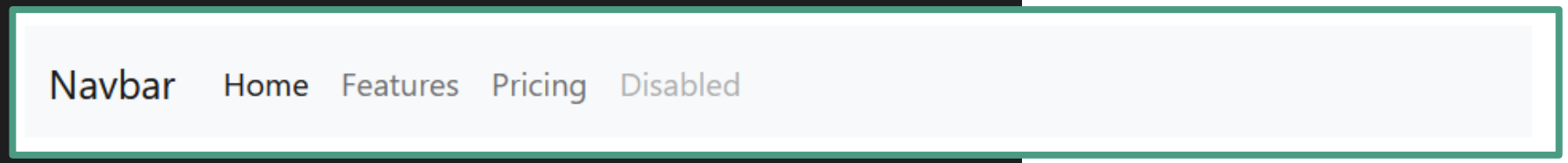
```
<meta name="viewport" content="width=device-width, initial-scale=1">
```

BOOTSTRAP : EXEMPLE

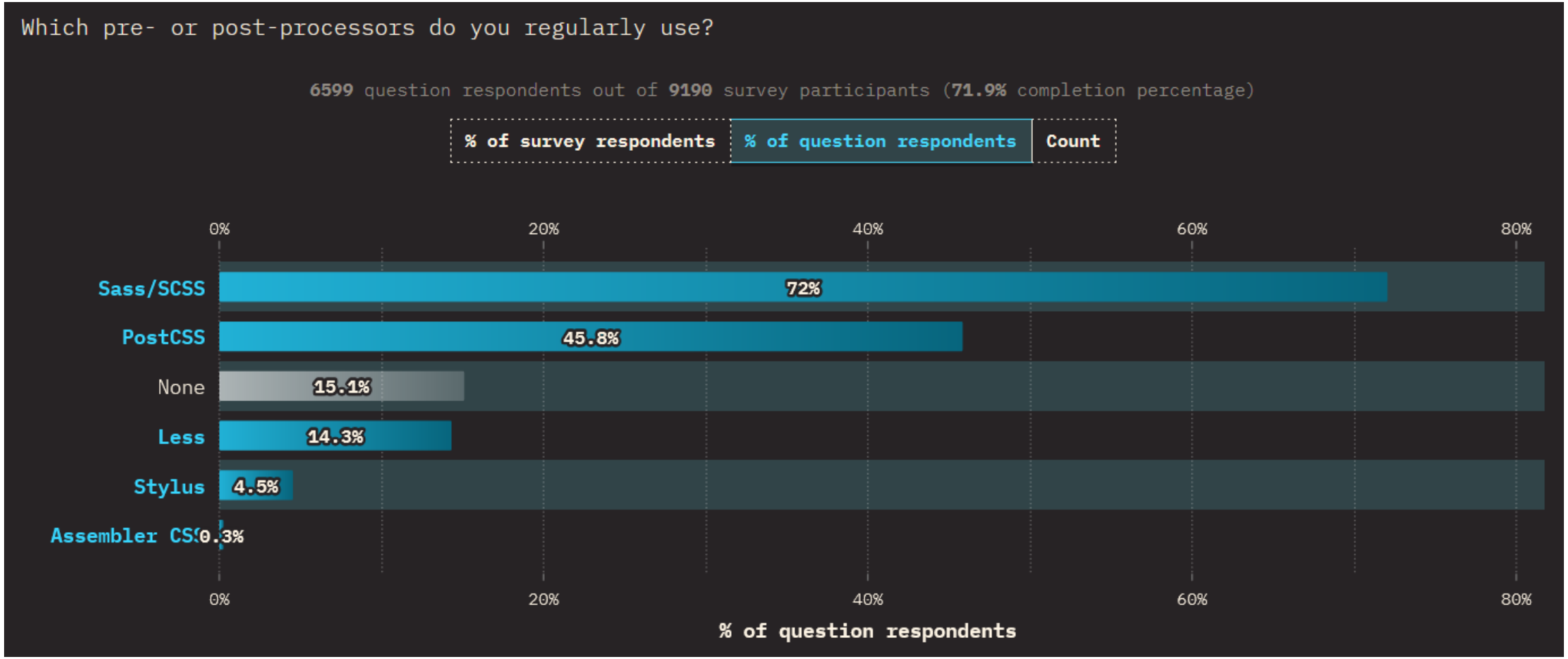
```
<div class="btn-group" role="group" aria-label="Basic mixed styles example">
  <button type="button" class="btn btn-danger">Left</button>
  <button type="button" class="btn btn-warning">Middle</button>
  <button type="button" class="btn btn-success">Right</button>
</div>
```

BOOTSTRAP : EXEMPLE

```
<nav class="navbar navbar-expand-lg navbar-light bg-light">
  <div class="container-fluid">
    <a class="navbar-brand" href="#">Navbar</a>
    <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarNav"
            aria-controls="navbarNav" aria-expanded="false" aria-label="Toggle navigation">
      <span class="navbar-toggler-icon"></span>
    </button>
    <div class="collapse navbar-collapse" id="navbarNav">
      <ul class="navbar-nav">
        <li class="nav-item">
          <a class="nav-link active" aria-current="page" href="#">Home</a>
        </li>
        <li class="nav-item">
          <a class="nav-link" href="#">Features</a>
        </li>
        <li class="nav-item">
          <a class="nav-link" href="#">Pricing</a>
        </li>
        <li class="nav-item">
          <a class="nav-link disabled">Disabled</a>
        </li>
      </ul>
    </div>
  </div>
</nav>
```

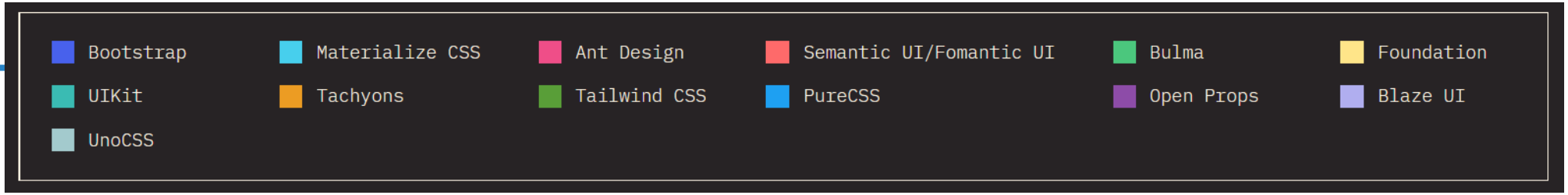


PRE-POSTPROCESSEUR CSS

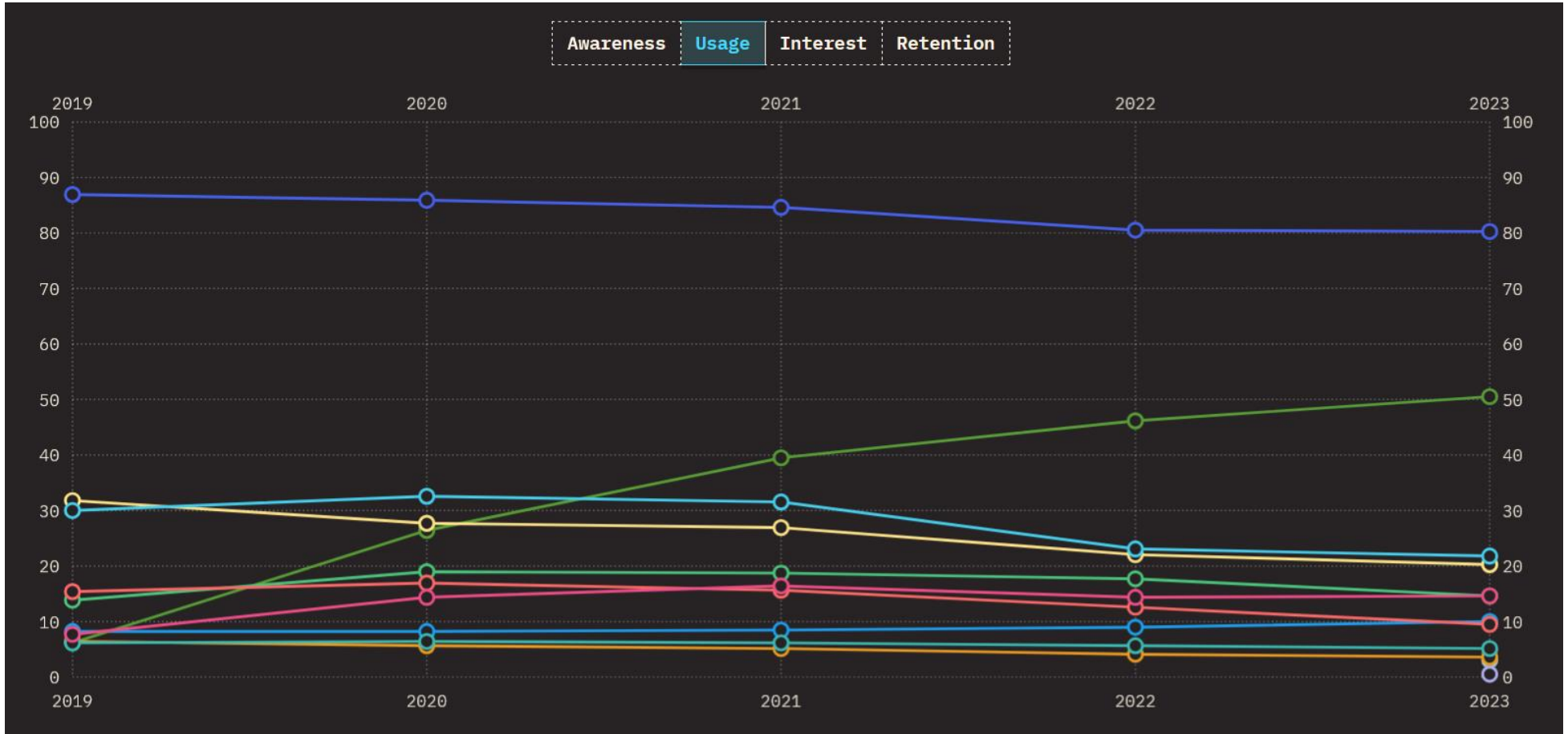


https://2023.stateofcss.com

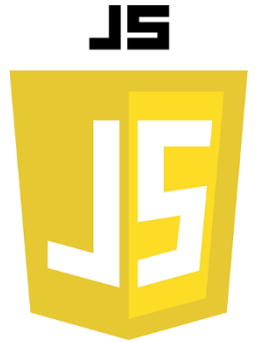
FRAMEWORK CSS



→ Ratio d'utilisation



JAVASCRIPT



Définition

Syntaxe

Objets

Evènements

JAVASCRIPT

→ Développé par Netscape et Sun dans les années 95 (principal développeur : Brendan Eich, co-fondateur du projet Mozilla, de la Mozilla Foundation et Mozilla Corporation)

→ Précédentes versions : Mocha (en interne), LiveScript, puis JavaScript

→ Conçu à l'origine comme un langage de script complémentaire à Java

→ Proposé à la standardisation à ECMA (*European Computer Manufacturers Association*) en 1996 (ECMA-262) :

- **ES1** en 1997
- **ES6** en 2015
- **ES2022** en juin 2022
- **ES2024**

JAVASCRIPT

Langage populaire :

→ Disponible dans tous les navigateurs (compatibilité)

→ La plupart des navigateurs modernes supporte le standard ES6 (98% à 100%)

→ Modules tiers disponibles grâce à NPM et GitHub : il existe des modules JavaScript pour quasiment chaque besoin.

Utilisation de JavaScript :

→ Historiquement exécuté sur le navigateur web

→ Node.js et NPM à la base du nouveau succès de JavaScript (serveur, local)

JAVASCRIPT

JavaScript permet :

→ De modifier l'apparence de la page

→ De communiquer avec le serveur

→ D'enregistrer les actions de l'utilisateur

→ De réagir aux évènements utilisateur

→ De sauvegarder des données

JAVASCRIPT

→ Langage orienté **prototype** et non orienté objet :

- Déclaration d'un objet générique (modèle), puis héritage
- Notion de classe depuis ECMAScript 6

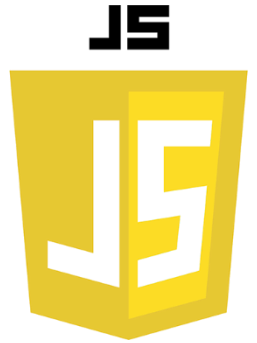
→ Dynamique :

- typage (faiblement typé),
- fonctions,

→ Evènementiel :

- paradigme de programmation, attente puis réaction aux actions utilisateur

JAVASCRIPT



Définition

Syntaxe

Objets

Evènements

JAVASCRIPT : SYNTAXE

Intégration dans la page :

→ En **interne** :

```
<script>alert("Hello World");</script>
```


→ En **externe** :

```
<script src="js/fichierJS.js"></script>
```

→ Attention au **positionnement des balises** dans le fichier html.

Quand le navigateur rencontre un bloc JavaScript, il l'exécute dans l'ordre. Le code est **interprété**, le résultat du code exécuté est envoyé directement.

JAVASCRIPT : SYNTAXE

Depuis **ES6** : mot-clé **async** et **defer**

```
<script src="js/monScript.js" defer></script>
```

→ **async** : le navigateur continu de traiter l'HTML et le fichier JS est téléchargé en parallèle. Une fois chargé, le contenu est exécuté.

→ **defer** : diffère l'exécution du JavaScript jusqu'à ce que la page soit complètement chargée.

JAVASCRIPT : SYNTAXE

→ **Syntaxe et opérateur semblable à C / C# / Java**

Operateurs (+, *, !+, etc.)

Variables

Chaînes de caractères et Array

Structures conditionnelles

Structures itératives

Fonctions

JAVASCRIPT : SYNTAXE

Portées des variables

→ **var :** permet de déclarer une variable dont la portée est le **contexte d'exécution courant** (**la fonction** qui contient la déclaration ou **le contexte global** si la variable est déclarée en dehors de toute fonction).

→ **let :** permet de déclarer une variable dont la portée est **le bloc courant**. *let* crée une variable globale alors que *var* ajoute une propriété à l'objet global au niveau le plus haut.

→ **const :** variable accessible qu'en lecture.

JAVASCRIPT : SYNTAXE

```
function varTest() {
    var x = 31;
    if (true) {
      var x = 71;
      console.log(x);
    }
    console.log(x);
  }

function letTest() {
    let x = 31;
    if (true) {
        let x = 71;
        console.log(x);
    }
    console.log(x);
}
```

```
var x = 'global';
let y = 'global2';
console.log(this.x);
console.log(this.y);
console.log(y);
```

JAVASCRIPT : SYNTAXE

Types de données :

→ type **booléen** (true et false)

→ Type **nul** (null)

→ Type **indéfinie** (undefined)

→ Type pour les **nombres** entiers ou réels (number)

→ Type pour les **chaînes de caractères** (string)

→ Type pour les **symboles** (depuis ES6 : type pour des données immuables et uniques)

→ Type pour les **objets** (Object, avec par exemple Array)

```
let x;          //undefined
x = 5;          //number
x = "Toto";     //string
let myArray = [1, 3.3, "toto"];
let myHashTable = {a: 5, c:9.5, c:"toto"};
```

JAVASCRIPT : SYNTAXE

Objet :

→ Chaque variable est considéré comme un objet.

→ Méthodes existantes pour les objets (string et array par exemple)

String :

```
let x = "toto tata titi tutu tyty tete";
console.log(x[3]);
console.log(x.charAt(8));
console.log("test".substring(1, 3));
console.log("test".toUpperCase());
```

```
let y = 16 + 4 + "Volvo";
let z = "Volvo" + 16 + 4;
```

```
parseInt("234");
parseInt("2.99abc");
```

```
console.log("toto"=="tata");
```

JAVASCRIPT : SYNTAXE

Liste

Pour appliquer un traitement à chaque élément d'un liste :

→ *forEach(<fonction de callback>)*

→ *map(<fonction de callback>)*

```
let arr = [1, 2, 3, 4, 5];
console.log(arr.length);
arr.push(6);
arr.splice(3, 0, 7);
console.log(arr.indexOf(7));
arr.sort();
arr.forEach(element => {
    console.log(element);
});
console.log(arr.map(element => element*2 ));
```

JAVASCRIPT : SYNTAXE

Structures conditionnelles

→ structure **if … else ….**

```
if (x > 50){
    // faire quelque chose
} else if (x > 5) {
    // faire autre chose
} else {
    // faire encore autre chose
}

/*
Greater than: >
Less than: <
Greater than or equal to: >=
Less than or equal to: <=
Equal: ==
Not equal: !=
*/
```

JAVASCRIPT : SYNTAXE

Structures conditionnelles

→ structure **switch … case …**

```
switch (variable) {
    case 1:
      // do something
      break;
    case 'a':
      // do something else
      break;
    case 3.14:
      // another code
      break;
    default:
      // something completely different
}
```

JAVASCRIPT : SYNTAXE

Structures itératives

→ boucles classiques :

for

while

do while

```
for(let counter=0 ; counter < 5 ; counter++){
    console.log(counter);
}
var i=0;
while(i<5){
    console.log(++i);
}
do{
    --i;
} while(i>0)
console.log(i);
```

JAVASCRIPT : SYNTAXE

Structures itératives

→ boucles **for … in …** et **for … of …**

```
let phones = {"type":"phone", "brand":"tomato", "name":"tomatoPhone"};
for(let key in phones){
    console.log(key);
    console.log(phones[key]);
}
let brandPhone = ["tomato", "pear"];
for(let i in brandPhone){
    console.log(i);
}
for(let elem of brandPhone){
    console.log(elem);
}
```

```
type
phone
brand
tomato
name
tomatoPhone
0
1
tomato
pear
```

JAVASCRIPT : SYNTAXE

Fonctions

Définition d'une fonction :

```
function nomFonction(arg1, arg2)
```

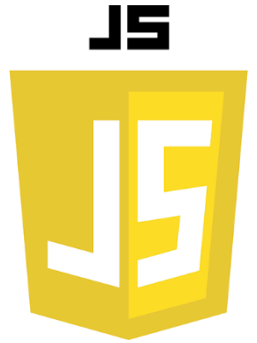

→ On ne donne pas le type des arguments ni celui de la valeur de retour éventuelle.

```
function average(a, b, c) {
    return ( a + b + c ) /3;
}
```

→ Possibilité d'écrire une fonction **anonyme :**

```
var result = function() { /* instructions */ }
```

JAVASCRIPT



Définition

Syntaxe

Objets

Evènements

JAVASCRIPT : OBJET

→ Notion d'objet est important en JavaScript : quasiment **tout est objet**.

Avant la normalisation ES6, il n'existait pas de notion de **class**.

Il est possible d'interagir à deux niveaux :

→ Au niveau du navigateur internet

→ Au niveau de la page affichée dans le navigateur

Tous les éléments HTML du DOM peuvent être manipulés en tant qu'objet.

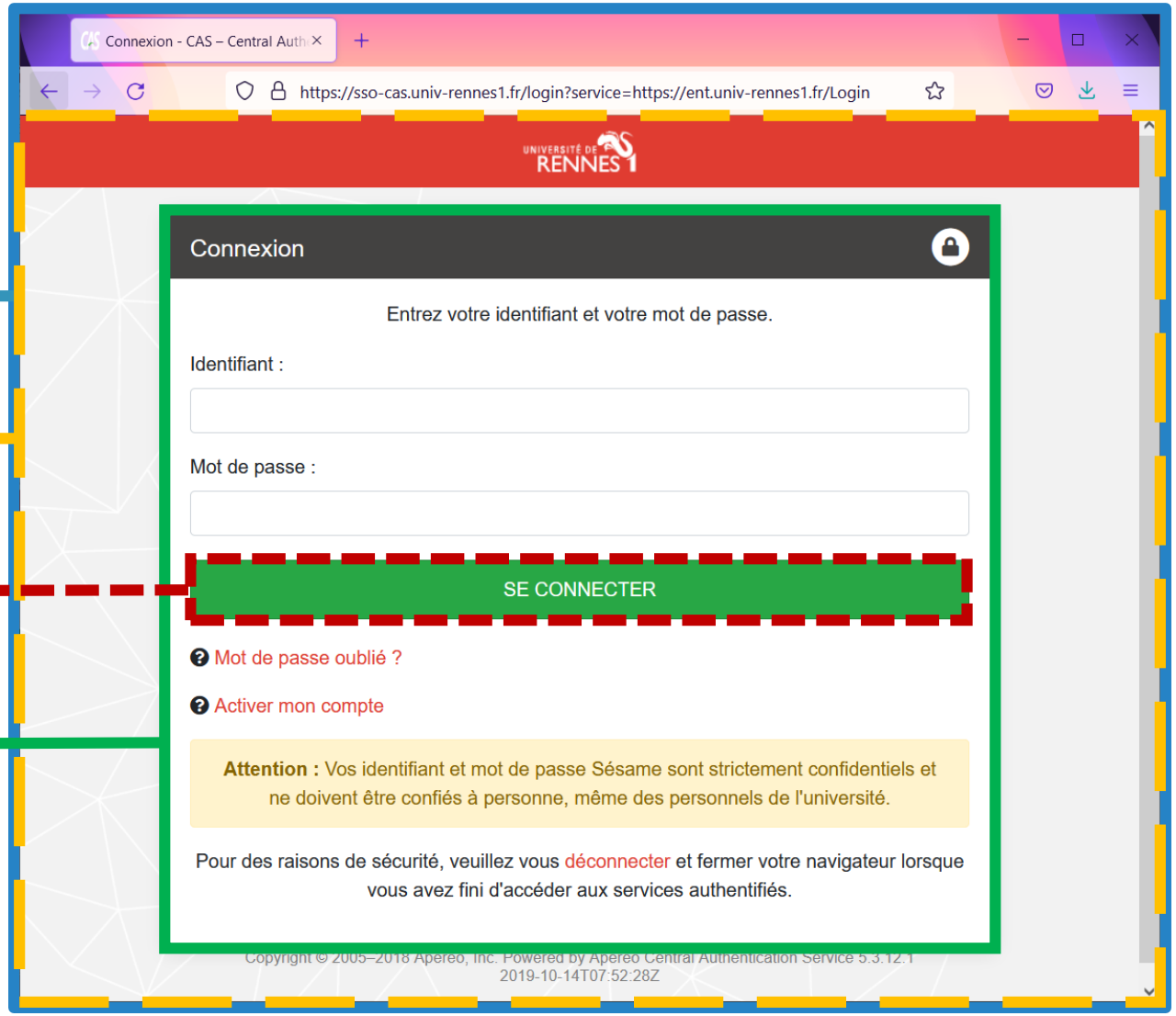
JAVASCRIPT : OBJET

Objet fenêtre

Objet document

Objet bouton

Objet formulaire



JAVASCRIPT : OBJET

→ L'accès se fait de façon hiérarchique.

```
window.document.forms["nomFormulaire"].nomElement
```

→ Par chaque objet il existe des méthodes et des attributs.


```
<form name="nomForm">
    <label for="login">Votre login :</label>
    <input type="text" name="nomLogin" id="login"/>
</form>
```

Votre login :

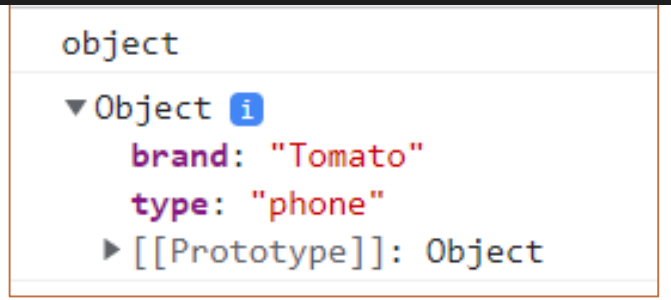
→ Par exemple, pour obtenir la valeur du champ login du formulaire :

```
<script>
    console.dir(window.document.forms["nomForm"].nomLogin);
    let userLogin = window.document.forms["nomForm"].nomLogin.value;
</script>
```

```
▼ input#login i
    accept: ""
    accessKey: ""
    align: ""
    alt: ""
    ariaAtomic: null
    ariaAutoComplete: null
    ariaBrailleLabel: null
    ariaBrailleRoleDescription: null
    ariaBusy: null
    ariaChecked: null
    ariaColCount: null
    ariaColIndex: null
```

JAVASCRIPT : OBJET

```
console.log(typeof myPhone);
console.dir(myPhone);
```



Possibilité de créer ses propres objets

→ **Objets littéraux (JSON)**

```
let myPhone = {"type": "phone", "brand": "Tomato" };
```


Une propriété d'un objet peut avoir n'importe quelle valeur :

→ Une valeur booléenne,

→ Une valeur scalaire,

→ Une liste,

→ Un objet,

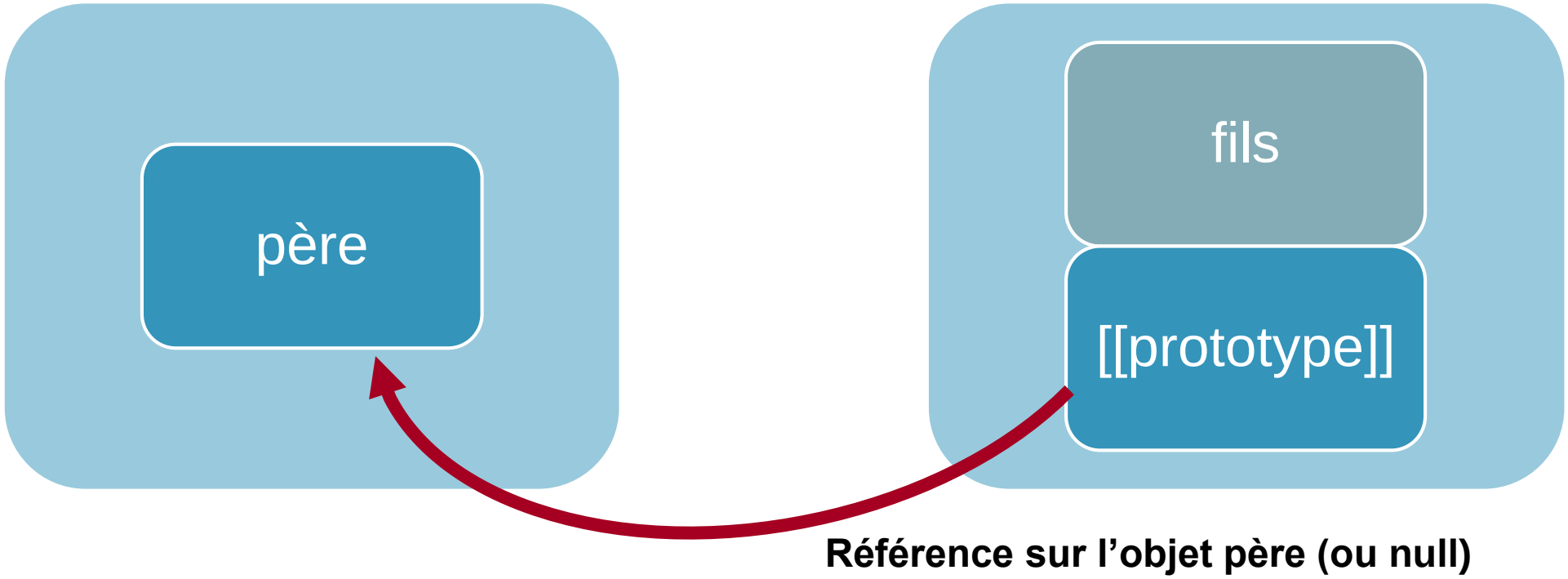
→ Un code implémentant une fonction ou une classe

JAVASCRIPT : OBJET

Propriété cachée

→ Héritage par chaînage de **prototype**

Création d'un nouveau objet héritant des propriétés d'un autre objet via **[[prototype]]**



JAVASCRIPT : OBJET

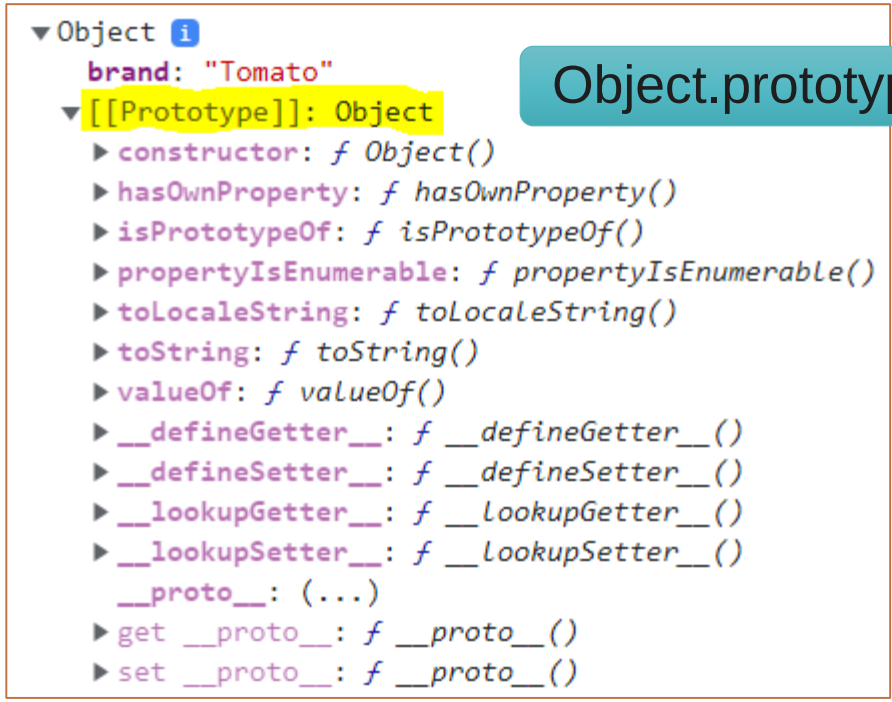
→ Héritage par chaînage de **prototype**

```
let tomatoPhone = { brand: "Tomato" };
console.dir(tomatoPhone);
```

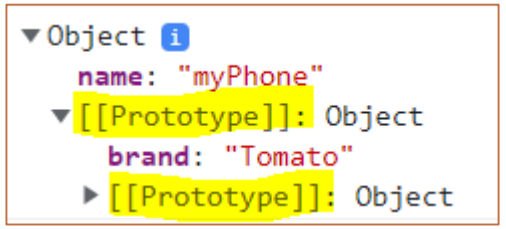
```
let myPhone = {
    name: "myPhone",
    __proto__ : tomatoPhone
};
console.dir(myPhone);
```

Obsolète : obj.__proto__

setPrototypeOf() et getPrototypeOf()



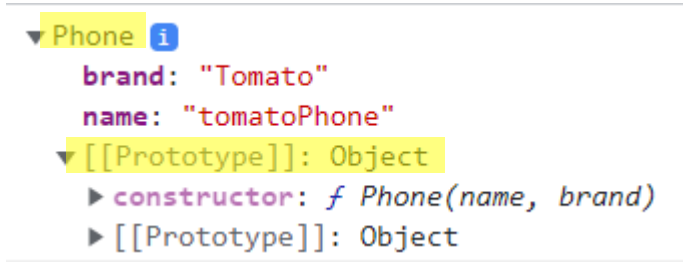
Object.prototype



__proto__ ≠ [[prototype]]

getter/setter de [[prototype]]

JAVASCRIPT : OBJET



→ Création d'un objet par une **fonction constructrice**

```
function Phone(name, brand) {
    this.name = name;
    this.brand = brand;
}

let tomatoPhone = new Phone("tomatoPhone", "Tomato");
console.dir(tomatoPhone);
```

→ Notion de **.prototype :** propriété que toutes les fonctions possèdent et qui est utilisée quand la fonction est utilisée comme fonction constructrice.

F.prototype contient une référence d'objet

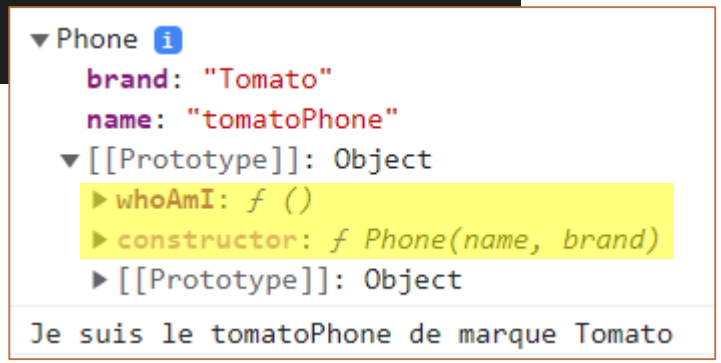
.prototype est défini comme le prototype du nouvel objet créé via la fonction constructrice

JAVASCRIPT : OBJET

→ Création d'un objet par une **fonction constructrice**

→ Possibilité **d'ajouter/supprimer des propriétés** au **prototype** sans modifier la propriété **constructor**.

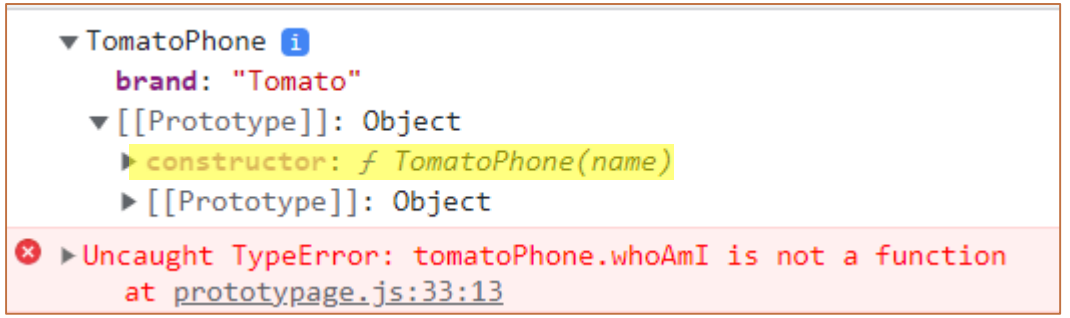

```
Phone.prototype.whoAmI = function() {
    console.log(`Je suis le ${this.name} de marque ${this.brand}`);
}

tomatoPhone.whoAmI();
```

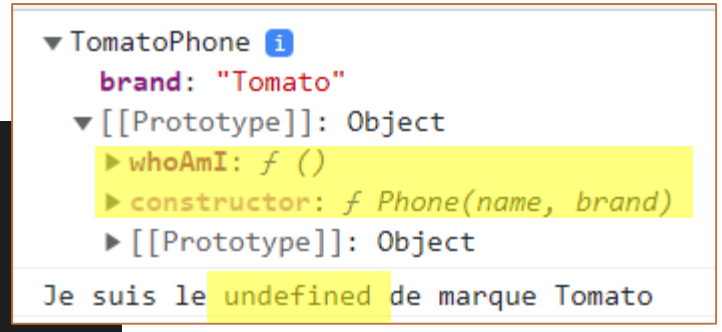


JAVASCRIPT : OBJET

→ L'héritage est implémenté par le **chaînage de prototypes**.

```
function Phone(name) {
    this.name = name;
}
Phone.prototype.whoAmI = function() {
    console.log("Je suis le " + this.name + " de marque " + this.brand);
}

function TomatoPhone(name) {

    this.brand = "Tomato";
}


var tomatoPhone = new TomatoPhone("tomatoPhone");
tomatoPhone.whoAmI();
```





JAVASCRIPT : OBJET

Identique à l'héritage par prototype, mais syntaxe plus simple pour créer des objets et manipuler l'héritage

→ Création de **classe** (ES6)

```
class Phone {
    constructor(name, brand){
        this.name = name;
        this.brand = brand;
    }
    whoAmI() {
        console.log("Je suis le " + this.name + " de marque " + this.brand);
    }
}
let tomatoPhone = new Phone("tomatoPhone", "Tomato");
tomatoPhone.whoAmI();
```

```
Je suis le tomatoPhone de marque Tomato
```

```
console.log(typeof Phone);
console.dir(tomatoPhone);
```

```
function
▼ Phone i
    brand: "Tomato"
    name: "tomatoPhone"
  ▼ [[Prototype]]: Object
    ▶ constructor: class Phone
    ▶ whoAmI: ƒ whoAmI()
    ▶ [[Prototype]]: Object
```

JAVASCRIPT : OBJET

→ Création de **classe** (ES6) :

Champ de classe : syntaxe permettant d'ajouter des propriétés uniquement à l'objet

```
class Phone {
    version = "test";
    constructor(name, brand){
        this.name = name;
        this.brand = brand;
    }
}
```

```
let tomatoPhone = new Phone("tomatoPhone", "Tomato");
console.log(tomatoPhone.version);
console.log(Phone.prototype.version);
```

```
test
undefined
```

JAVASCRIPT : OBJET

→ **Héritage** entre classe (ES6)

Utilisation du mot clé **extends**

Appel du constructeur de la classe mère avec **super()**

```
Je suis le tomatoPhone de marque Tomato
Je suis le super pearPhone de marque Pear
```

```
class Phone {
    constructor(name, brand){
        this.name = name;
        this.brand = brand;
    }
    whoAmI() {
        console.log("Je suis le " + this.name + " de marque " + this.brand);
    }
}
class SuperPhone extends Phone{
    constructor(name, brand){
        super(name, brand);
    }
    whoAmI() {
        console.log(`Je suis le super ${this.name} de marque ${this.brand}`);
    }
}
let tomatoPhone = new Phone("tomatoPhone", "Tomato");
tomatoPhone.whoAmI();
let smartPhone = new SuperPhone("pearPhone", "Pear");
smartPhone.whoAmI();
```

JAVASCRIPT : CLASS

→ Attention à la compatibilité des navigateurs avec les fonctionnalités récentes proposé par ECMASript.

	Chrome	Edge	Firefox	Opera	Safari	Chrome Android	Firefox for Android	Opera Android	Safari on iOS	Samsung Internet	WebView Android	Deno	Node.js
classes	✓ 49 …	✓ 13	✓ 45	✓ 36 …	✓ 9	✓ 49 …	✓ 45	✓ 36 …	✓ 9	✓ 5.0 …	✓ 49 …	✓ 1.0	✓ 6.0.0 …
constructor	✓ 49 …	✓ 13	✓ 45	✓ 36 …	✓ 9	✓ 49 …	✓ 45	✓ 36 …	✓ 9	✓ 5.0 …	✓ 49 …	✓ 1.0	✓ 6.0.0 …
extends	✓ 49 …	✓ 13	✓ 45	✓ 36 …	✓ 9	✓ 49 …	✓ 45	✓ 36 …	✓ 9	✓ 5.0 …	✓ 49 …	✓ 1.0	✓ 6.0.0 …
Private class fields	✓ 74	✓ 79	✓ 90	✓ 62	✓ 14.1	✓ 74	✓ 90	✓ 53	✓ 14.5	✓ 11.0	✓ 74	✓ 1.0	✓ 12.0.0
Private class fields 'in'	✓ 91	✓ 91	✓ 90	✓ 77	✓ 15	✓ 91	✓ 90	✓ 64	✓ 15	✓ 16.0	✓ 91	✓ 1.9	✓ 16.4.0
Private class methods	✓ 84	✓ 84	✓ 90	✓ 70	✓ 15	✓ 84	✓ 90	✓ 60	✓ 15	✓ 14.0	✓ 84	✓ 1.0	✓ 14.6.0
Public class fields	✓ 72	✓ 79	✓ 69	✓ 60	✓ 14.1 …	✓ 72	✓ 79	✓ 51	✓ 14.5 …	✓ 11.0	✓ 72	✓ 1.0	✓ 12.0.0
static	✓ 49 …	✓ 13	✓ 45	✓ 36 …	✓ 9	✓ 49 …	✓ 45	✓ 36 …	✓ 9	✓ 5.0 …	✓ 49 …	✓ 1.0	✓ 6.0.0 …
Static class fields	✓ 72	✓ 79	✓ 75	✓ 60	✓ 14.1	✓ 72	✓ 79	✓ 51	✓ 14.5	✓ 11.0	✓ 72	✓ 1.0	✓ 12.0.0
Class static initialization blocks	✓ 94	✓ 94	✓ 93	✓ 80	✗ No	✓ 94	✓ 93	✓ 66	✗ No	✓ 17.0	✓ 94	✓ 1.14	✓ 16.11.0

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Classes

JAVASCRIPT : OBJET

Mot clé this

Il se comporte légèrement différemment des autres langages de programmation.

→ **Dans le contexte global** :

this fait référence à l'objet global.

Dans le cas d'un navigateur

this = window

```
console.log(this === window);

this.a = 37;
console.log(window.a);

this.b = "JXC";
console.log(window.b);
console.log(b);

var c = 1;
console.log(this.c);
console.log(window.c);

let d = true;
console.log(window.d);
```

JAVASCRIPT : OBJET

Mot clé this

→ **Dans le contexte d'une fonction** :

la valeur de ***this*** dépend de la façon dont la fonction est appelée.

Quand une fonction est appelée comme **méthode d'un objet**, ***this*** correspond à l'objet possédant la méthode qu'on appelle.

```
var o = {
    prop: 37,
    f: function() {
        return this.prop;
    }
};

console.log(o.f()); // 37
```

Quand une fonction est utilisée comme un **constructeur** (avec le mot clef new), ***this*** sera lié au nouvel objet.

```
function Phone(){
    this.brand = "tomato";
}
var myPhone = new Phone();
console.log(myPhone.brand); // tomato
```

JAVASCRIPT : OBJET

Mot clé this

→ **Dans le contexte d'une fonction** :

Une fonction peut être appelée sans être liée à un objet :


```
function f1(){
    return this;
}
console.log(f1() === window); // true (objet global)
```

Mode use strict non activé

```
"use strict";

function f1(){
    return this;
}

console.log(f1()); //undefined
```

Cet appel peut être vue comme une erreur de programmation : si il y a un **this** dans une fonction, il s'attend à être appelé dans **un contexte d'objet**.

JAVASCRIPT : OBJET

Mot clé this

En utilisant les **fonctions fléchées**, ***this*** correspond à la valeur de this utilisée dans le contexte englobant (ces fonctions ne possèdent pas de this).


```
let objet = {
    i: [10, 20, 30],
    j: 100,
    b: () => console.log(this.j, this),
    c: function() {
        console.log(this.j, this);
    },
    showList() {
        this.i.forEach(
          elem => console.log(this.j + ': ' + elem)
        );
    }
}
```

```
objet.b();
objet.c();
objet.showList()
```

JAVASCRIPT : OBJET

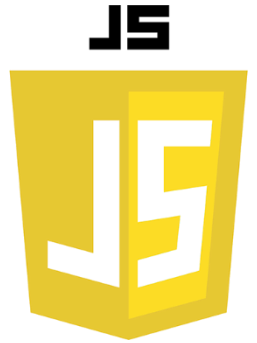
Mot clé this

Pour passer ***this*** d'un **contexte à un autre**, les fonctions suivantes peuvent être utilisées : **call()**, **apply()** et **bind()**.


```
"use strict";

function fonction(name) { this.name = name; }
fonction.prototype.methode = function(callback) {
    console.log('fonction ', this);
    callback();
}

let objet = new fonction("objet1");
objet.methode(function() {
    console.log('objet ', this);
})
```

```
fonction  ▶fonction {name: 'objet1'}
objet  undefined
```

JAVASCRIPT



Définition

Syntaxe

Objets

Evènements

JAVASCRIPT : EVENEMENT

L’action sur un élément de la page HTML se fait lors d’un évènement particulier :

→ clic sur un bouton,

→ champ input d’un formulaire qui change,

→ Redimensionnement de la fenêtre,

→ Formulaire en cours de soumission,

→ fin de chargement de la page HTML, etc.

Les **gestionnaires d’évènements** peuvent être utilisés pour gérer et vérifier les entrées utilisateur, les actions utilisateur et les actions du navigateur.

JAVASCRIPT : EVENEMENT

Exemple d'évènement possible pour une page Web :

→ click (onClick)

→ load (onLoad)

→ unload (onUnload)

→ mouseOver (onMouseOver)

→ mouseOut (onMouseOut)

→ focus (onFocus)

→ change (onChange)

→ submit (onSubmit)

https://developer.mozilla.org/fr/docs/Web/Events

JAVASCRIPT : EVENEMENT

→ Exemple d'utilisation de **onChange** avec un champ **input** d'un formulaire :

Fichier HTML

```
<body>
  <form name="monForm">
      <label for="login">Votre login :</label>
      <input type="text" name="login" id="login" onchange="afficherLogin();" />
      <input type="text" name="loginBis" id="loginBis" />
  </form>
</body>
```

Fichier .js

```
function afficherLogin(){
    window.document.forms["monForm"].loginBis.value =
        window.document.forms["monForm"].login.value;
}
```

Résultat

JAVASCRIPT : EVENEMENT

→ Exemple d'utilisation de **onChange** avec un champ **input** d'un formulaire :

Fichier HTML

```
<body>
  <form name="monForm">
      <label for="login">Votre login :</label>
      <input type="text" name="login" id="login" />
      <input type="text" name="loginBis" id="loginBis" />
  </form>
</body>
```

Fichier .js

```
var loginEvent = document.forms["monForm"].login.onchange = function() {
    window.document.forms["monForm"].loginBis.value =
        window.document.forms["monForm"].login.value;
}
```

Résultat

Méthode à privilégier

JAVASCRIPT : EVENEMENT

→ Utilisation de la méthode **addEventListener()** :

Enregistre un écouteur d'évènement sur un élément DOM.

Permet également d'enregistrer plusieurs gestionnaires pour le même écouteur.

Possibilité de supprimer un écouteur ajouté précédemment (*removeEventListener()*).


```html
<body>
  <form name="monForm">
      <label for="login">Votre login :</label>
      <input type="text" name="login" id="login" />
      <input type="text" name="loginBis" id="loginBis" />
  </form>
</body>
```

Fichier HTML

```js
var inputLogin = document.forms["monForm"].login;
inputLogin.addEventListener('change', function(){
    window.document.forms["monForm"].loginBis.value =
        window.document.forms["monForm"].login.value;
});
```

Fichier .js

JAVASCRIPT : EXEMPLE

Fichier HTML

```html
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Hello world</title>
    <link rel="stylesheet" href="css/monCSS.css" />
    <script src="js/mesFonctions.js" defer></script>
  </head>
  <body>
    <div id="maDiv"></div>
    <button type="button" id="btnvert">Vert</button>
    <button type="button" id="btnrouge">Rouge</button>
  </body>
</html>
```

JAVASCRIPT : EXEMPLE

Fichier CSS

```css
#maDiv{
    background-color: #635ead;
    width:200px;
    height:200px;
}
```

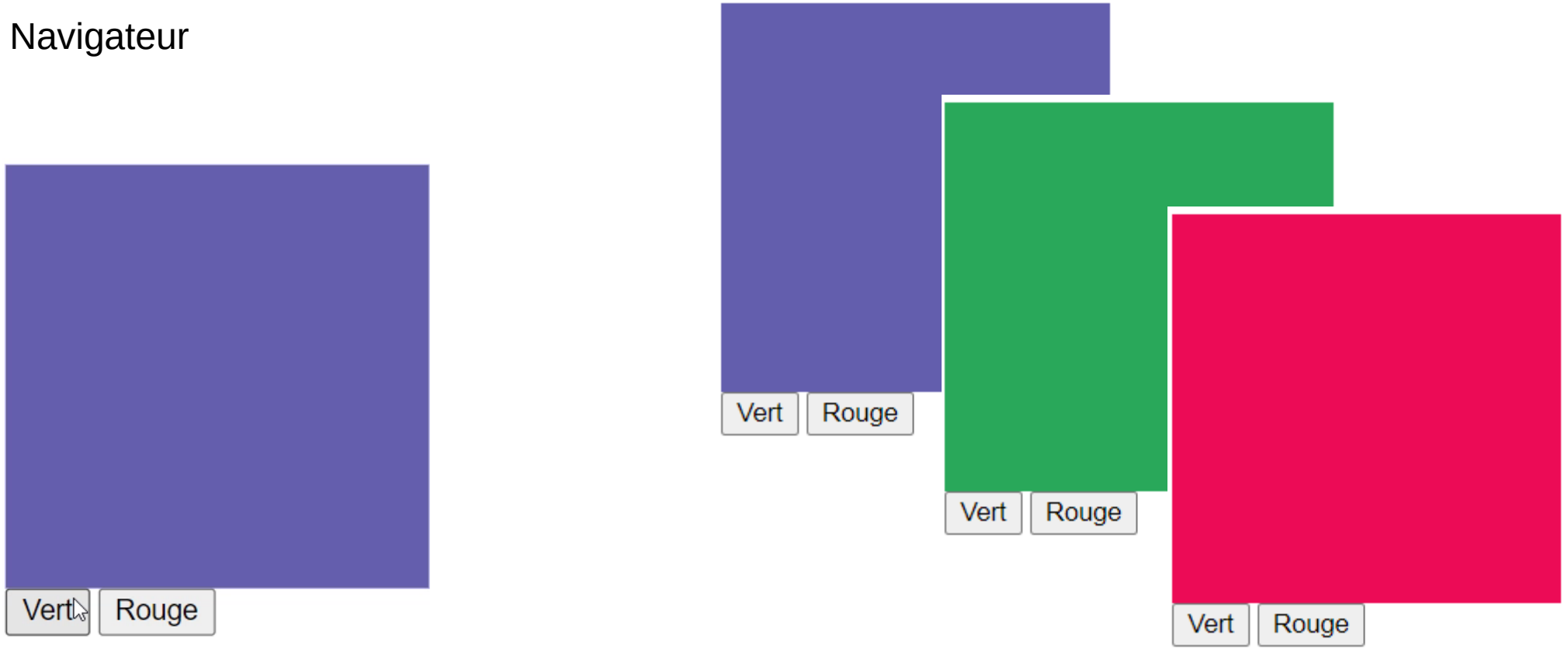
JAVASCRIPT : EXEMPLE

Fichier JS

```
function vert(){
    document.getElementById('maDiv').style.backgroundColor='#29a85a';
}
function rouge(){
    document.getElementById('maDiv').style.backgroundColor='#ec0b56';
}

document.getElementById('btnvert').addEventListener('click', vert);
document.getElementById('btnrouge').addEventListener('click', rouge);
```

JAVASCRIPT : EXEMPLE

Navigateur



JAVASCRIPT : ECMASCRIPT

→ **Lien vers les nouvelles spécifications ECMAScript :**

https://www.ecma-international.org/technical-committees/tc39/?tab=published-standards

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Language_Resources

→ **Table de compatibilité ECMAScript :**

http://kangax.github.io/compat-table/es6/

http://kangax.github.io/compat-table/es2016plus/

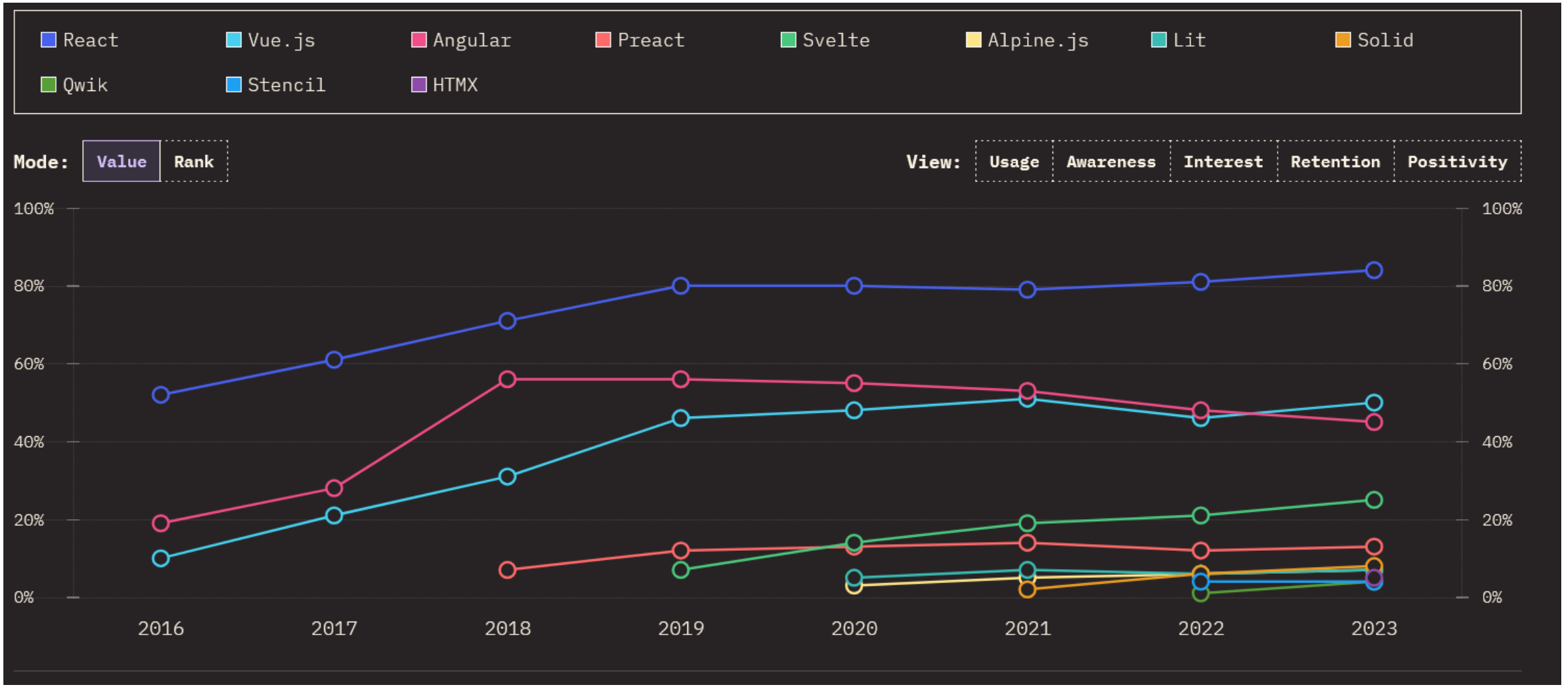
FRAMEWORK

Utilisation de Framework

Notion de composant web

Concept d’Angular

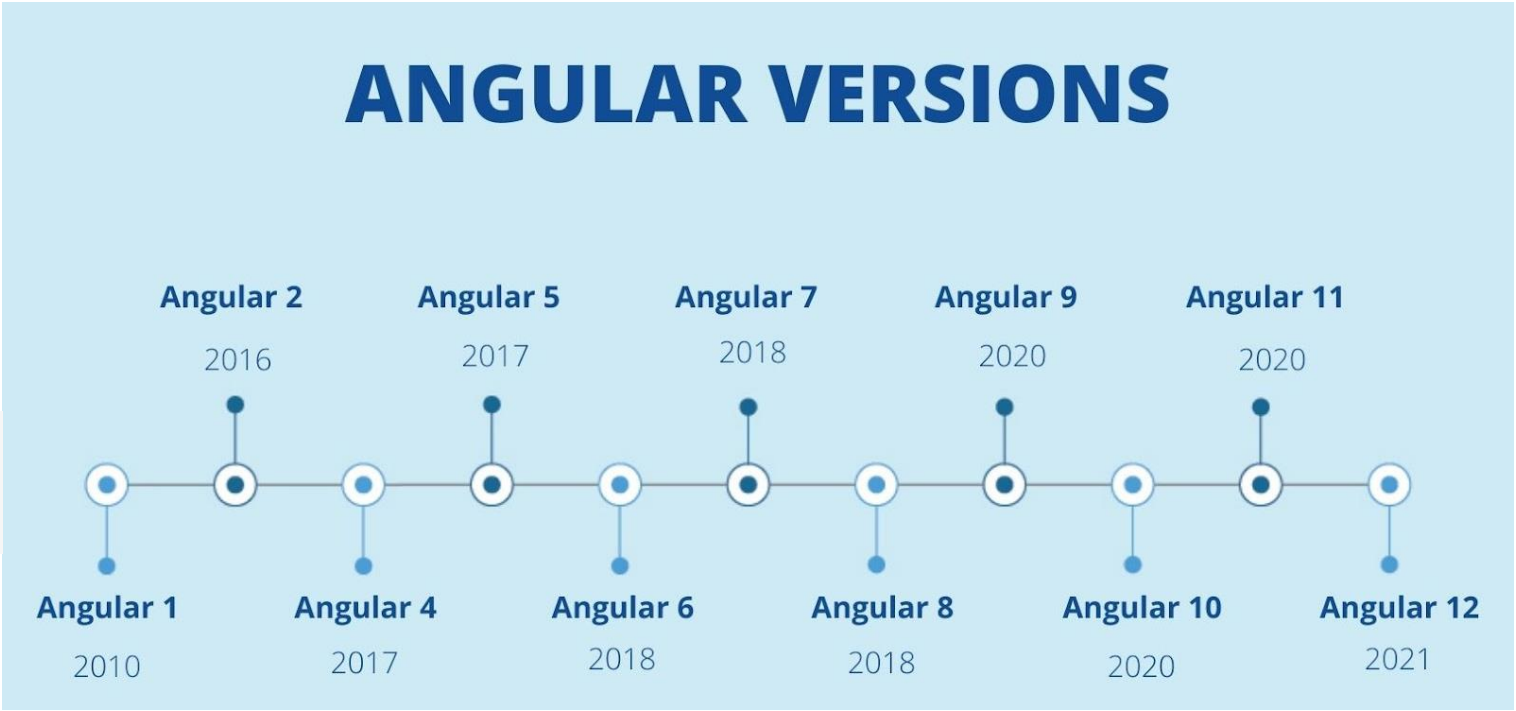
FRAMEWORK



https://2023.stateofjs.com/en-US/libraries/front-end-frameworks/

FRAMEWORK ANGULAR

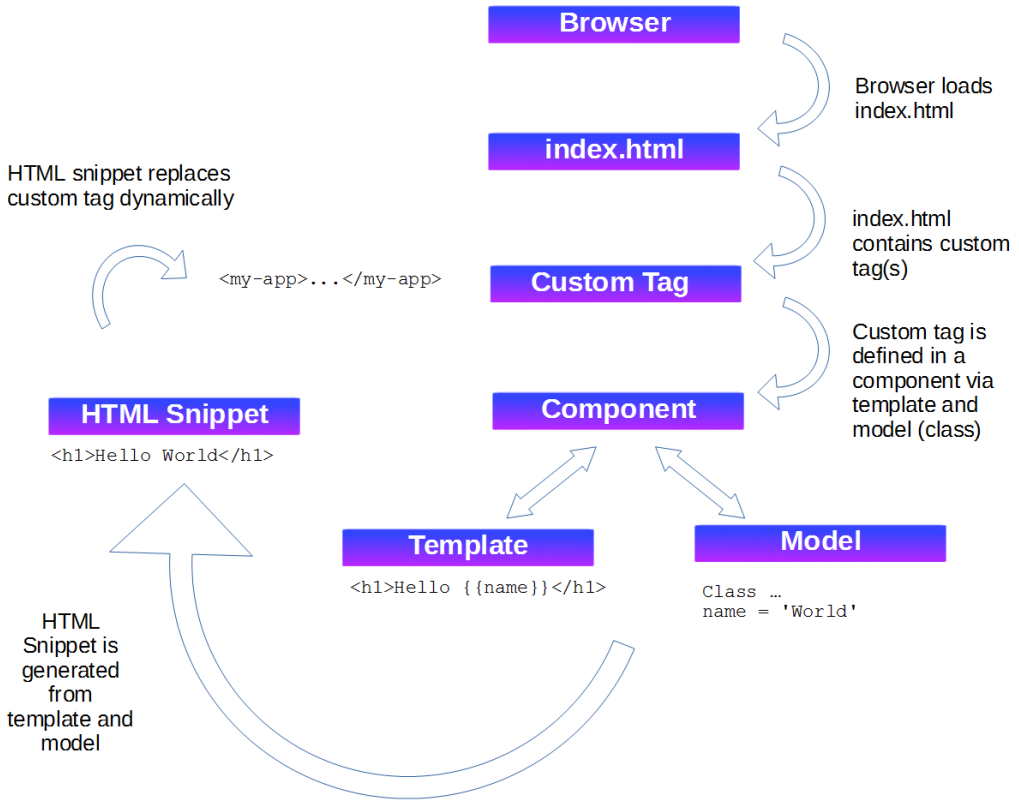
ANGULARJS



ANGULAR

FRAMEWORK ANGULAR

→ Angular '*compile*' HTML : utilisation de custom tag



FRAMEWORK : COMPOSANT WEB

→ Faciliter la réutilisation d'éléments dans la monde du web (création d'élément réutilisables, encapsulés et versatiles sans risquer une collision avec d'autres morceaux de code).

Standard du web :

→ **Custom Elements** (permet de créer et enregistrer des nouveaux éléments HTML)

→ **HTML Templates** (squelette pour créer des éléments HTML instanciables)

→ **Shadow DOM** (permet d'encapsuler le JavaScript et le CSS des éléments)

→ **HTML Imports** (abandonnées au profit des imports JavaScript)

FRAMEWORK : COMPOSANT WEB

→ Enrichi le web avec des nouveaux tag.

Principe :

Créer des nouveaux tags

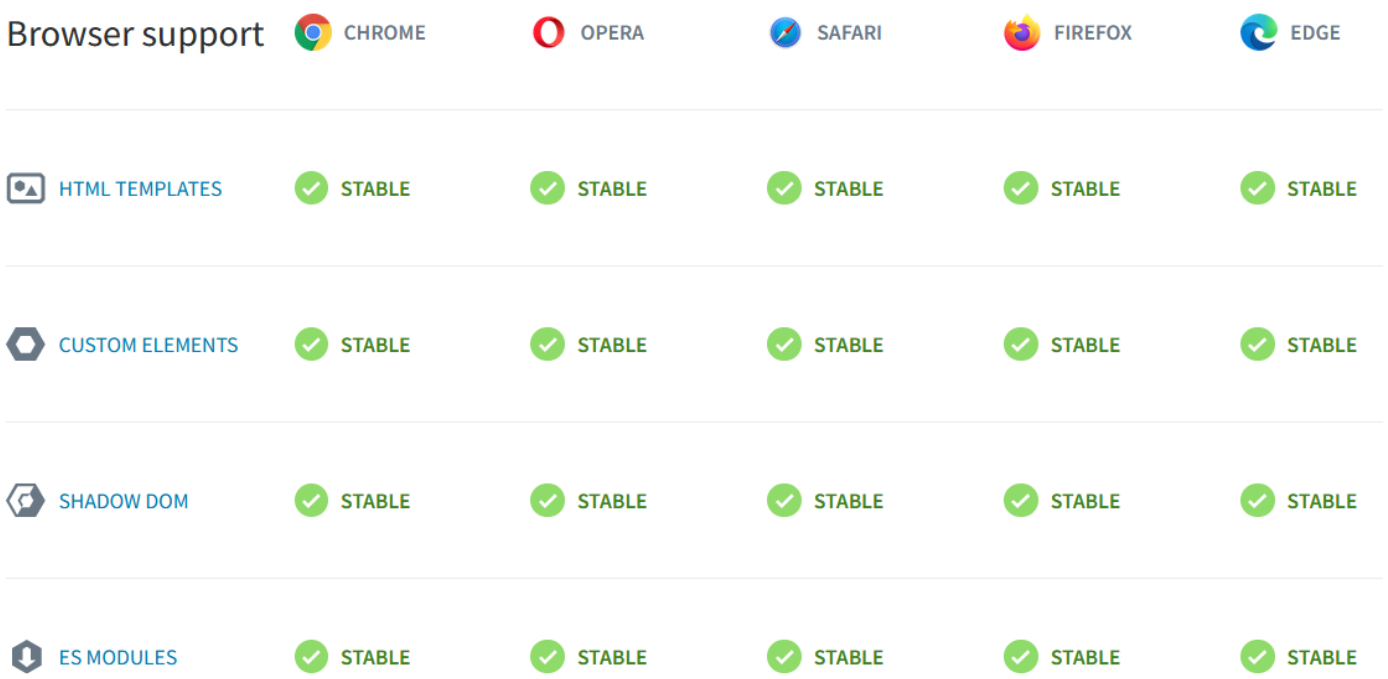
Encapsuler le code pour masquer et isoler sa complexité

Etre capable d’importer et de déclarer les tags dans d’autres pages ou projets.

FRAMEWORK : COMPOSANT WEB

→ Standardisation des composants web en 2012 par W3C.

→ Google via son projet ***Polymer*** à aider au développement des web componants.



COMPOSANT WEB : CUSTOM ELEMENT

→ **4 grandes étapes :**

Etape 1 : Créer une classe (syntaxe de classe ES6) dans laquelle est spécifiée la **fonctionnalité du composant web**.

Etape 2 : Enregistrer le nouvel élément personnalisé avec la commande ***define***.

Etape 3 : Connecter un shadow DOM à l'élément personnalisé, puis ajouter des éléments fils, des écouteurs d'évènements, etc.

Etape 4 : Possibilité de définir un **template HTML**. Utilisation des méthodes DOM pour cloner le template et le connecter au shadow DOM.

COMPOSANT WEB : CUSTOM ELEMENT

Etape 1 : Créer une classe (syntaxe de classe ES6) dans laquelle est spécifiée la **fonctionnalité du composant web**.

→ Hérite de ***HTMLElement***

→ Possibilité de définir des fonctions se déclenchant à différent moment du **cycle de vie de l'élément**.

→ Extension de l'API JavaScript *CustomElements* pour créer des nouveau éléments.

```
class HelloWorld extends HTMLElement {
    constructor(){
        super();
        console.log("constructor");
        //Fonctionnalité de l'élément
    }
    //Rappels du cycle de vie
    connectedCallback(){
        this.innerHTML = '<p>Composant connecté pour la 1ère fois au DOM : '+
                         'Ajout du contenu initial / fetch data</p>'
    }
    disconnectedCallback(){
        //l'élémentt personnalisé est déconnecté du DOM du document
    }
    adoptedCallback(){
        //l'élément personnalité est déplacé vers un nouveau document
    }
    attributeChangedCallback(){
        //un des attributs de l'élément personnalisé est ajouté, supprimé ou modifié
    }
}
```

COMPOSANT WEB : CUSTOM ELEMENT

Etape 2 : Enregistrer le nouvel élément personnalisé avec la commande ***define***.

→ Le nom de l'élément doit contenir un « - »

```
//customElements.define(name, constructor, options);
customElements.define('hello-world', HelloWorld);
```

→ Possibilité d'ajouté des options

```
customElements.define('word-count', WordCount, { extends: 'p' });
```

```
class WordCount extends HTMLParagraphElement { /*...*/ }
```

COMPOSANT WEB : CUSTOM ELEMENT

Deux types d'élément customisé :

→ Éléments customisés autonomes (ne dépend pas d'un autre élément HTML)

```
<hello-world></hello-world>
```

```
document.createElement('hello-world');
```


→ Eléments intégrés personnalisés (héritant d'un élément HTML de base)

```
<p is="word-count"></p>
```

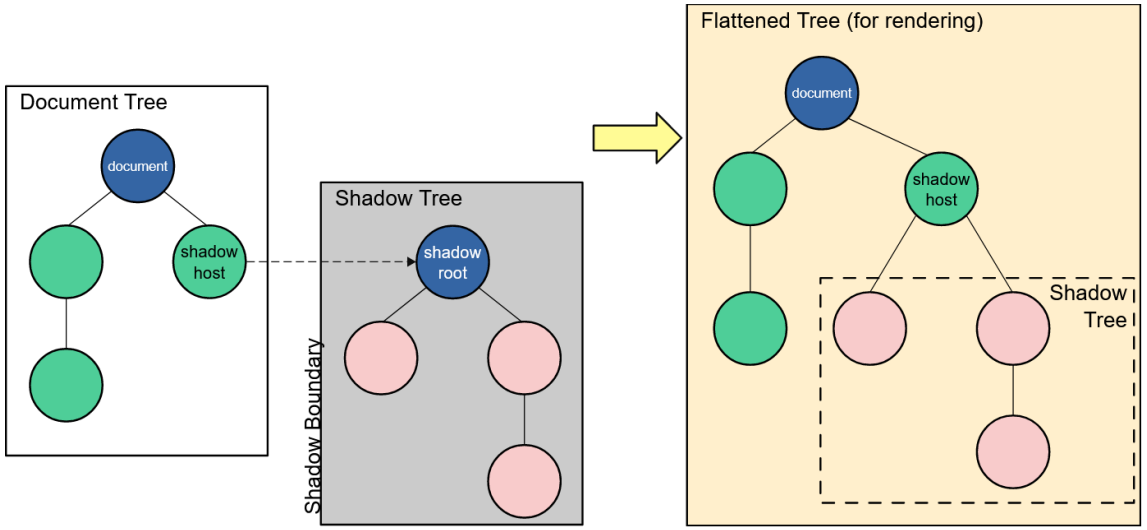
```
document.createElement("p", {is: "word-count"})
```

COMPOSANT WEB : SHADOW DOM

Etape 3 : Connecter un **shadow DOM** à l'élément personnalisé, puis ajouter des éléments fils, des écouteurs d'évènements, etc.

→ **Encapsulation** possible grâce au **Shadow DOM** : possible d'utiliser JavaScript et du CSS dans un élément customisé sans affecter les autres composants.

Lors de la création du **shadow DOM**, un sous arbre est rattaché à un élément du DOM.



COMPOSANT WEB : SHADOW DOM

Etape 3 : Connecter un **shadow DOM** à l'élément personnalisé, puis ajouter des éléments fils, des écouteurs d'évènements, etc.

→ Associer une racine fantôme à un élément :

```
let fantome = element.attachShadow({mode: 'open'});
let fantome = element.attachShadow({mode: 'closed'});
```

→ Ajout d'élément au Shadow DOM :

```
let paragraphe = document.createElement('p');
fantome.appendChild(paragraphe);
```

COMPOSANT WEB : SHADOW DOM

→ Possibilité d'ajouter une feuille de style externe au lieu de la balise style :

```
const linkElem = document.createElement('link');
linkElem.setAttribute('rel', 'stylesheet');
linkElem.setAttribute('href', 'style.css');

shadow.appendChild(linkElem);
```

COMPOSANT WEB : HTML TEMPLATE

Etape 4 : Possibilité de définir un **template HTML**. Utilisation des méthodes DOM pour cloner le template et le connecter au shadow DOM.

→ Constructeur de la classe *HelloWorld* :

```
constructor() {
    super();
    let template = document.getElementById('hello');
    let templateContent = template.content;

    const shadowRoot = this.attachShadow({mode: 'open'});
    shadowRoot.appendChild(templateContent.cloneNode(true));
}
```

```
<template id="hello">
  <p>Autre Hello World !</p>
</template>
```

COMPOSANT WEB : EXEMPLE

→ Résultat :

COMPOSANT WEB : EXEMPLE

→ Page HTML :

```html
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Hello world</title>
    <meta name="viewport" content="width=device-width, initial-scale=1">
  </head>
  <body>
    <template id="counter">
        <style>
          button {
            background-color: red;
            color: white;
            padding: 4px;
          }
        </style>
        <button>Click me</button>
        <span id="times">0</span>
    </template>
    <my-counter></my-counter>
    <button>Pas dans le même composant</button>
    <script src='js/composantWeb2.js'></script>
  </body>
</html>
```

COMPOSANT WEB : EXEMPLE

→ Page JavaScript :

```
class MyCounter extends HTMLElement {
    times = 0;
    constructor() {
      super();
        const template = document.getElementById('counter');
        const shadowRoot = this.attachShadow({mode: 'open'});
        shadowRoot.appendChild(template.content.cloneNode(true));
        this.onClick = this.onClick.bind(this);
        this.shadowRoot.querySelector('button')
            .addEventListener('click', this.onClick);
    }
    onClick() {
      this.times += 1;
      this.shadowRoot.querySelector('#times').textContent = this.times;
    }
  }
  customElements.define("my-counter", MyCounter);
```

FRAMEWORK ANGULAR

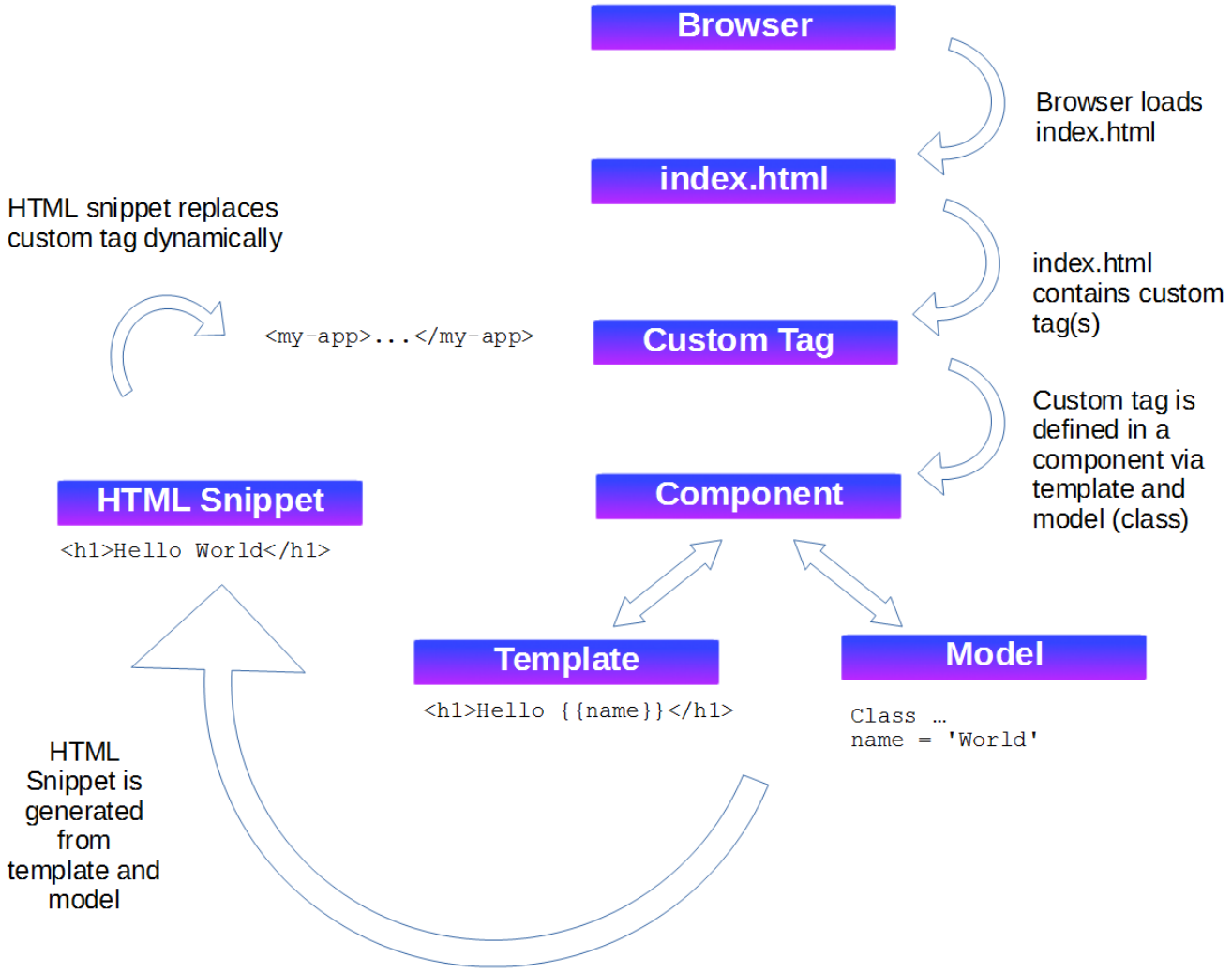
Utilisation de Framework

Notion de composant web

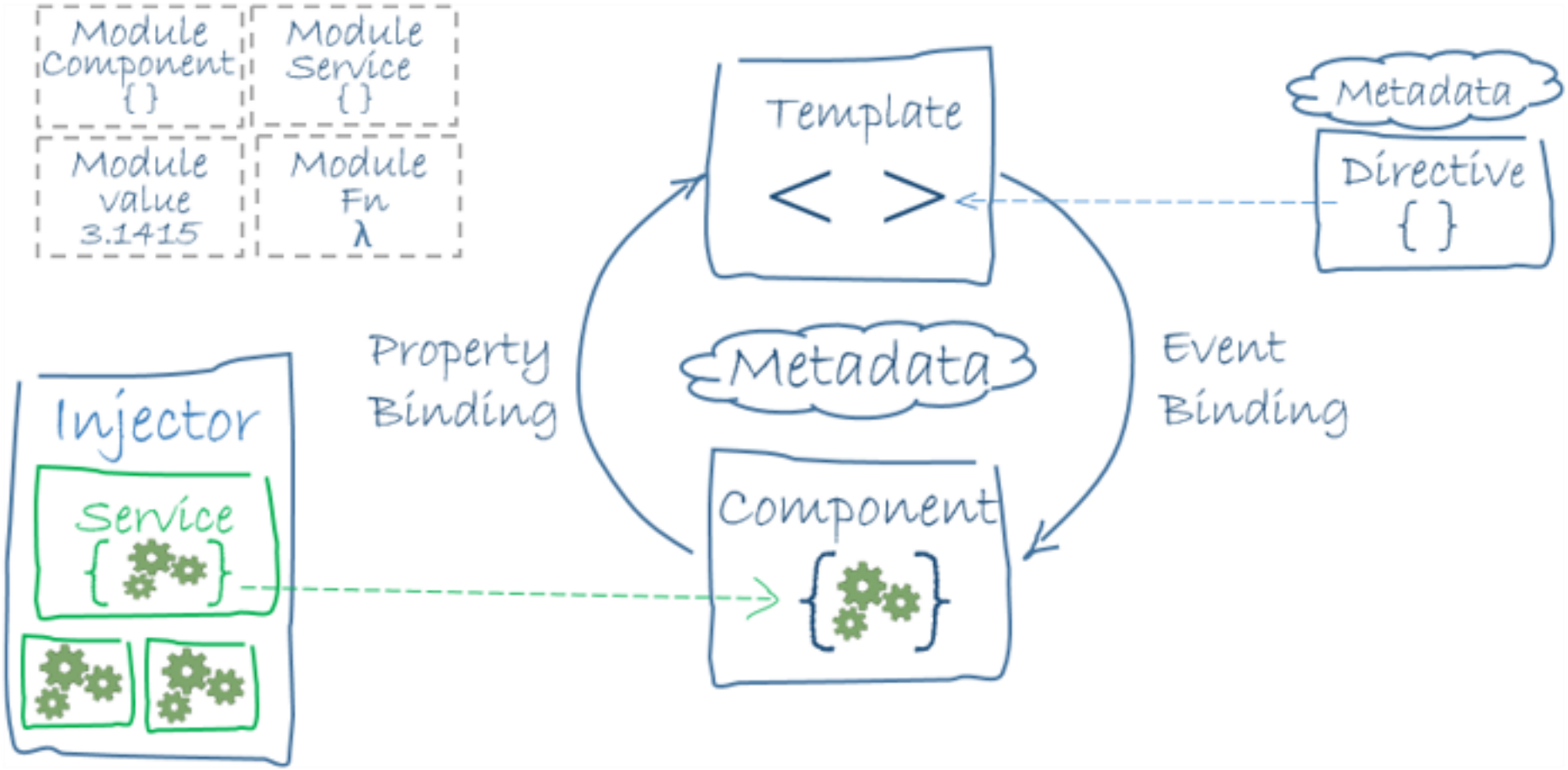
Concept d’Angular

ANGULAR

→ Angular s'appuie fortement sur la notion de **web component**



ANGULAR


Module
Component
{}
Module
Service
{}
Module
value
3.1415
Module
Fn
λ
Template
< >
Metadata
Directive
{}
Property
Binding
Metadata
Event
Binding
Injector
Service
{}
Component
{}

ANGULAR : CONCEPTS

→ **Components** : élément de base d'une application Angular, permet de définir la manière dont l'utilisateur interagit avec la vue.

→ **Templates** : rendu HTML du composant sur une page.

→ **Data bindings** : relation entre les données d'un composant (modèle) et les valeurs affichées dans le template.

→ **Metadata :** information permettant de relier des éléments Angular (template et component par exemple pour former la vue)

→ **Component interaction :** lien entre les différents composants (échange d'information)

ANGULAR : CONCEPTS

→ **Dependency injection / service** : implémentation du pattern IoC (inversion des contrôles) afin de gérer les dépendances d'une application.

→ **Routing** : gérer l'aspect navigation d'une SPA.

→ **Forms** : gestion de la saisie utilisateur.

→ **Pipe :** transformation de la valeur d'un élément avant de l'afficher dans la vue (e.g. date).

→ **Modules :** organisation d'une application en bloc fonctionnel.

ANGULAR : INSTALLATION

→ Vérifier la version de node.js (> 18.19.1) et npm

→ Installation : `$ npm install -g @angular/cli`

→ Création d'un projet Angular : `$ ng new <project-name>`

- Do you want to enable Server-Side Rendering (SSR) and Static Site Generation (SSG/Prerendering)? No *(pour débuter …)*

→ Playground : https://angular.dev/playground

→ Documentation : https://angular.dev/playground
https://angular.fr/get_started/installation.html

FRAMEWORK ANGULAR

Utilisation de Framework

Notion de composant web

Concept d’Angular

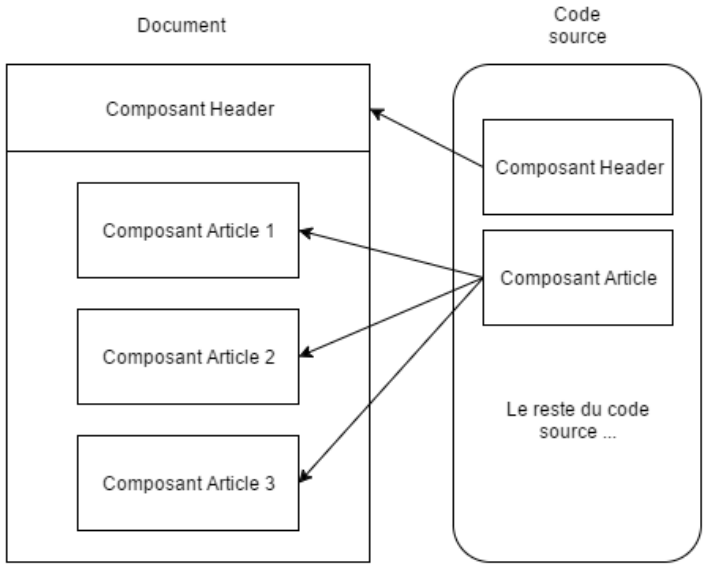
Component / Template

ANGULAR : COMPOSANT

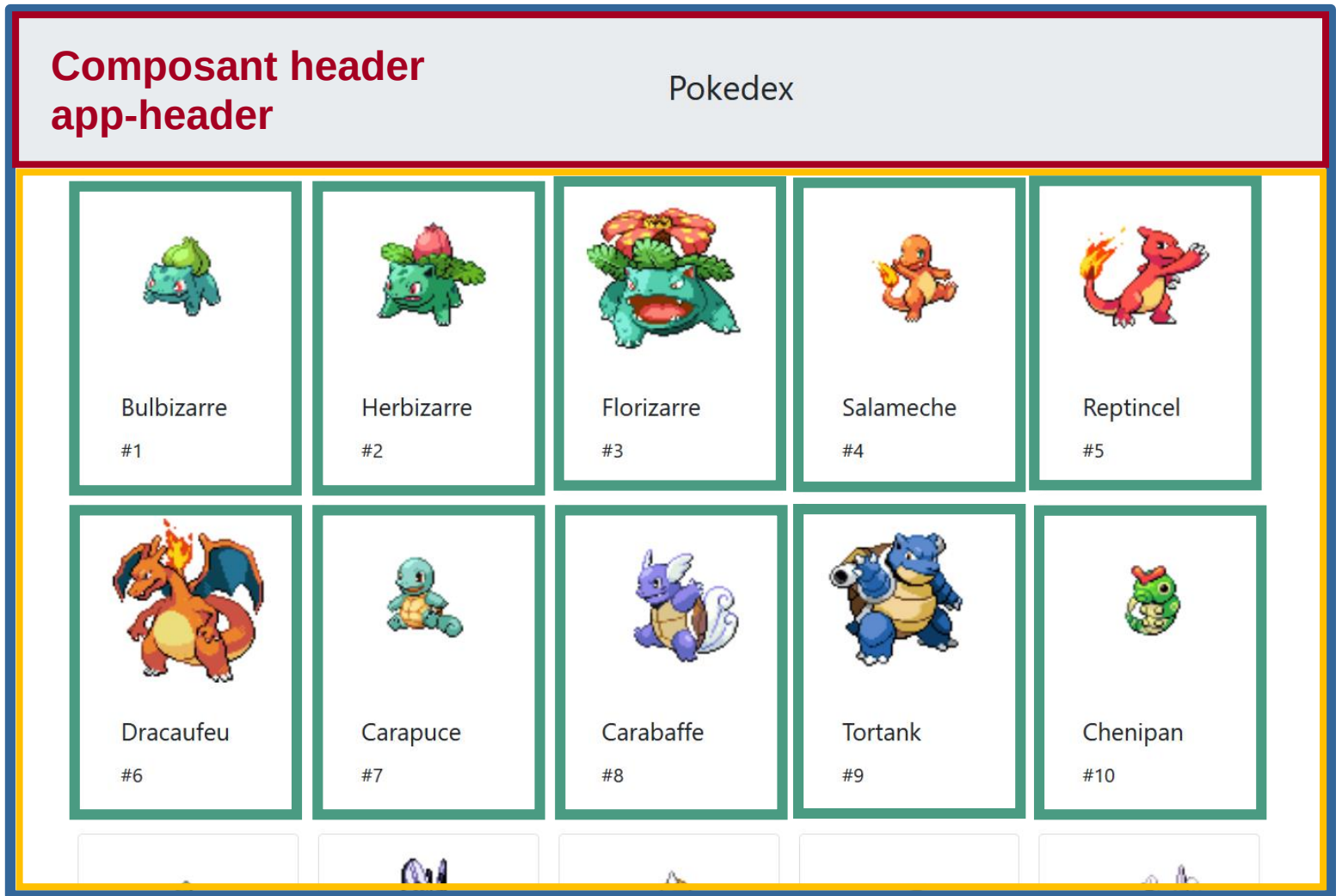
→ Structure fondamentale d'Angular (et d'autres frameworks et Web Component).

→ Une application est découpée en composant qui peuvent contenir eux-mêmes d'autres composants.

→ Avantages : réutilisation des composants et découpage logique.



ANGULAR : COMPOSANT



Composant principal (app component) app-root

Composant pokemon app-pokemon

Composant pokedex app-pokedex

ANGULAR : COMPOSANT STANDALONE

→ **Un composant Angular ?**

Interface utilisateur en HTML.

Métadonnées permettant de finir la classe comme composant Angular (directive @Component).

Classe Typescript avec des attributs, des méthodes logiques utilisée dans la vue.

ANGULAR : TEMPLATE

→ Correspond à la vue du composant dans la page de l'application Angular.

→ Possibilité d'utiliser des directives, de déclencher un appel d'évènement, d'afficher les données mises à jour des composants (databinding), d'instancier d'autres composants et bien d'autres.

ANGULAR : PREMIER COMPOSANT

Exemple pour la partie Angular basé sur Zelda :

→ Utilisation du composant racine du projet Angular (**composant standalone**).

→ Comment lancer notre application ?

```
ng serve --watch
```

ANGULAR : PREMIER COMPOSANT

→ Fichier app.component.ts

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

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

→ Fichier app.component.html

```html
<header>
  <h1>{{title}}</h1>
</header>
```

ANGULAR : PREMIER COMPOSANT

→ Fichier app.component.ts

Permet d'importer le décorateur Component

Décorateur **@Component** : permet de déclarer le composant

standalone: true/false

Classe liée au composant

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

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


→ Fichier app.component.html

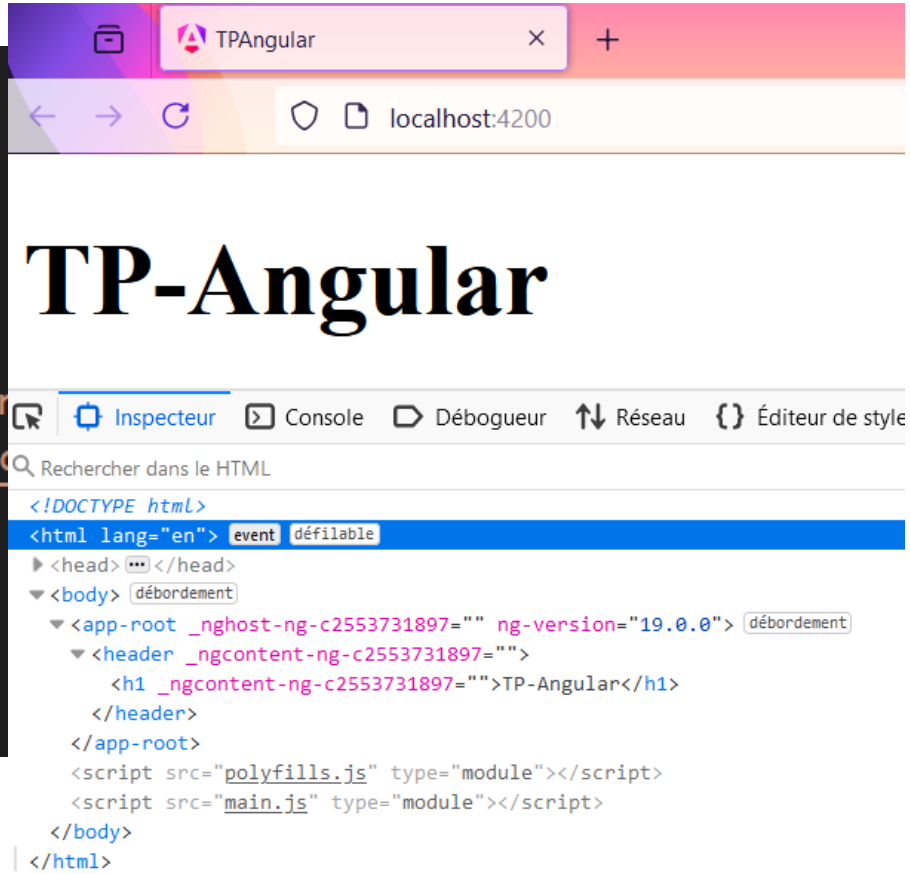
```html
<header>
  <h1>{{title}}</h1>
</header>
```

Interpolation du texte

ANGULAR : PREMIER COMPOSANT

→ Fichier **index.html** avec utilisation de notre composant racine (selecteur **app-root**).

```html
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>TPAngular</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, in
  <link rel="icon" type="image/x-icon" href="favicon.ic
</head>
<body>
  <app-root></app-root>
</body>
</html>
```



ANGULAR : PREMIER COMPOSANT

Exemple pour la partie Angular basé sur Zelda :

→ Utilisation du composant racine du projet Angular.

→ Nouveau composant Personnages :

ng generate component personnages

```
CREATE src/app/personnages/personnages.component.html (27 bytes)
CREATE src/app/personnages/personnages.component.spec.ts (650 bytes)
CREATE src/app/personnages/personnages.component.ts (245 bytes)
CREATE src/app/personnages/personnages.component.css (0 bytes)
```

ANGULAR : COMPOSANT

→ Possibilité de créer **des classes et des interfaces** pour les données afin de les utiliser dans les composants.

```typescript
export interface IPersonnage {
    id: number;
    name: string;
}
```

```typescript
import { Component } from '@angular/core';
import { IPersonnage } from '../interface/personnage.interface';
import { CommonModule } from '@angular/common';

@Component({
  selector: 'app-personnages',
  imports: [CommonModule],
  templateUrl: './personnages.component.html',
  styleUrl: './personnages.component.css'
})
export class PersonnagesComponent {
  hero: IPersonnage = {
    id: 1,
    name: 'Link'
  }
}
```

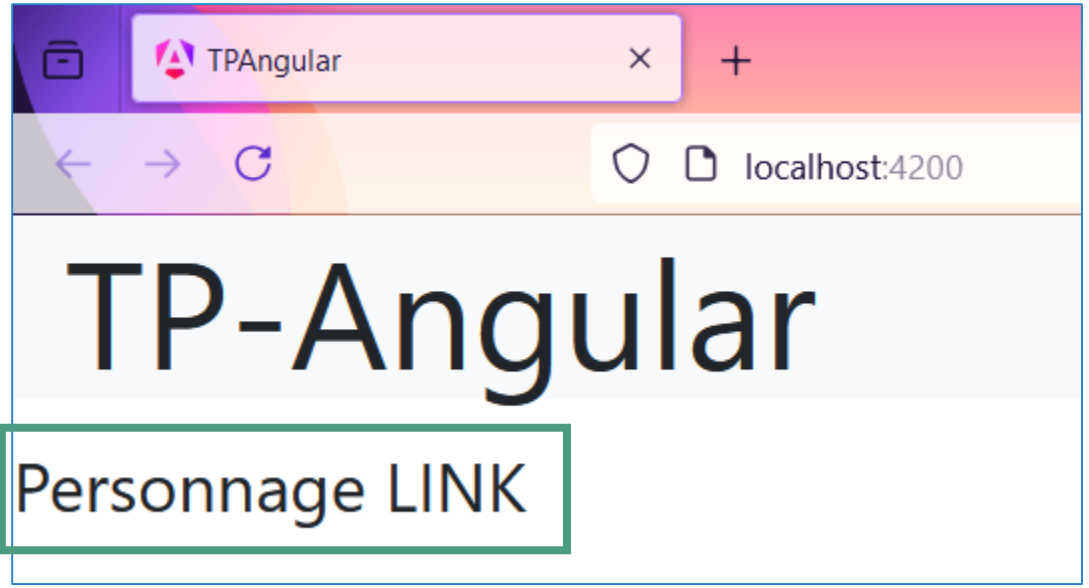
ANGULAR : COMPOSANT

→ Possibilité de créer **des classes et des interfaces** pour les données afin de les utiliser dans les composants.

```
export interface IPersonnage {
    id: number;
    name: string;
}
```

Possible d'utiliser ce pipe car CommonModule est importé

```
<p>Personnage {{hero.name | uppercase}}</p>
```



Balise `<app-personnages></app-personnages>` dans le fichier app.component.html

ANGULAR : PREMIER COMPOSANT

Exemple pour la partie Angular basé sur Zelda :

→ Utilisation du composant racine du projet Angular.

→ Nouveau composant Personnages.

→ Simuler des données d’un back pour les afficher dans le template de Personnages.

ANGULAR : COMPOSANT

→ Utilisation d'une liste de personnages dans le composant pour simuler la récupération des données depuis un serveur.

TP-Angular > src > app > TS personnages.mock.ts > ...

```ts
import { IPersonnage } from "./interface/personnage.interface";

export const PERSONNAGES: IPersonnage[] = [
    {id:1, name: 'Link'},
    {id:2, name: 'Zelda'},
    {id:3, name: 'Revali'},
    {id:4, name: 'Urbosa'},
    {id:5, name: 'Sidon'},
    {id:6, name: 'Mipha'},
]
```

ANGULAR : COMPOSANT

→ Utilisation d'une liste de personnages dans le composant pour simuler la récupération des données depuis un serveur.

TP-Angular > src > app > personnages > TS personnages.component.ts > ...

```
1   import { Component } from '@angular/core';
2   import { CommonModule } from '@angular/common';
3   import { PERSONNAGES } from '../personnages.mock';
```

```
11  export class PersonnagesComponent {
12    listPersonnages = PERSONNAGES;
13    MyHero = this.listPersonnages[1];
14  }
```

ANGULAR : COMPOSANT

Directive ***ngFor** **(avant Angular 17)**

TP-Angular > src > app > personnages > <> personnages.component.ht

Go to component

```
1   <p>Personnage : {{MyHero.name | uppercase}}</p>
2   <div class="container-fluid">
3       <p>Liste de tous les personnages :</p>
4       <ul class="list-group">
5           <li *ngFor="let hero of listPersonnages" class="list-group-item">
6               <span class="badge rouded-pill bg-info text-dark">
7                   {{hero.id}}
8               </span>
9               {{hero.name}}
10          </li>
11      </ul>
12  </div>
```

ANGULAR : COMPOSANT

Contrôle de flux **@for (à partir d'Angular 17)**

TP-Angular > src > app > personnages > personnages.component.html > ...

Go to component

```
<p>Personnage {{MyHero.name | uppercase}}</p>
<div class="container-fluid">
    <p>Liste de tous les personnages :</p>
    @for(hero of listPersonnages; track hero.id) {
        <ul class="list-group">
            <li class="list-group-item">
                <span class="badge rouded-pill bg-info text-dark">
                    {{hero.id}}
                </span>
                {{hero.name}}
            </li>
        </ul>
    }
    @empty {
        <p>Aucun personnage chargé ...</p>
    }
</div>
```

ANGULAR : VIEW ENCAPSULATION

Gestion de l'encapsulation

Possibilité de spécifier une *View Encapsulation* dans un composant.

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



```
@Component({
  selector: 'app-personnages',
  imports: [CommonModule],
  templateUrl: './personnages.component.html',
  styleUrl: './personnages.component.css',
  encapsulation: ViewEncapsulation.Emulated
})
```

→ **None** : aucune encapsulation de style réalisée, le style d'un composant est global à toute l'application.

→ **Emulated** : le Shadow DOM n'est pas utilisé mais une encapsulation émulée est fait pour les styles de composants (permet de limiter les styles des composants)

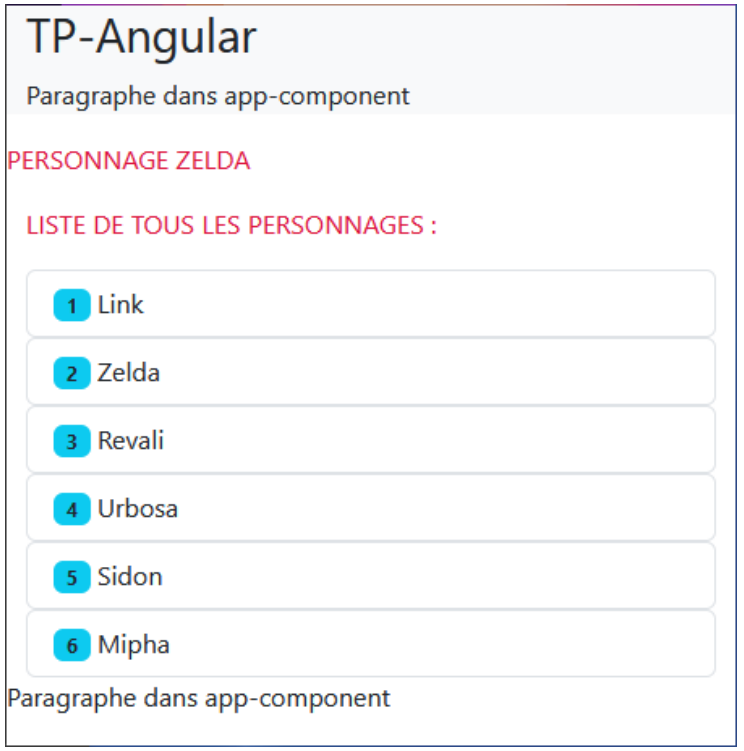
→ **ShadowDom** : utilisation du shadow DOM du navigateur (rendu plus rapide mais attention au support des navigateurs).

ANGULAR : VIEW ENCAPSULATION

→ **Exemple :** ajout de deux balises **<p>** dans app-component.html.

ajout d'un style pour la balise **<p>** du composant **Personnages**

```
@Component({
  selector: 'app-personnages',
  imports: [CommonModule],
  templateUrl: './personnages.component.html',
  styleUrl: './personnages.component.css',
  encapsulation: ViewEncapsulation.Emulated
})
```



ANGULAR : VIEW ENCAPSULATION

→ **Exemple :** ajout de deux balises **<p>** dans app-component.html.

ajout d'un style pour la balise **<p>** du composant **Personnages**

```
@Component({
  selector: 'app-personnages',
  imports: [CommonModule],
  templateUrl: './personnages.component.html',
  styleUrl: './personnages.component.css',
  encapsulation: ViewEncapsulation.None
})
```

ANGULAR : CYCLE DE VIE

Cycle de vie des composants

Possibilité d'intercepter les différentes étapes du cycle de vie d'un composant.

→ **ngOnChanges() :** appelée à la création du composant (après le constructeur), puis à chaque changement d'un attribut scalaire décoré par **@Input**.

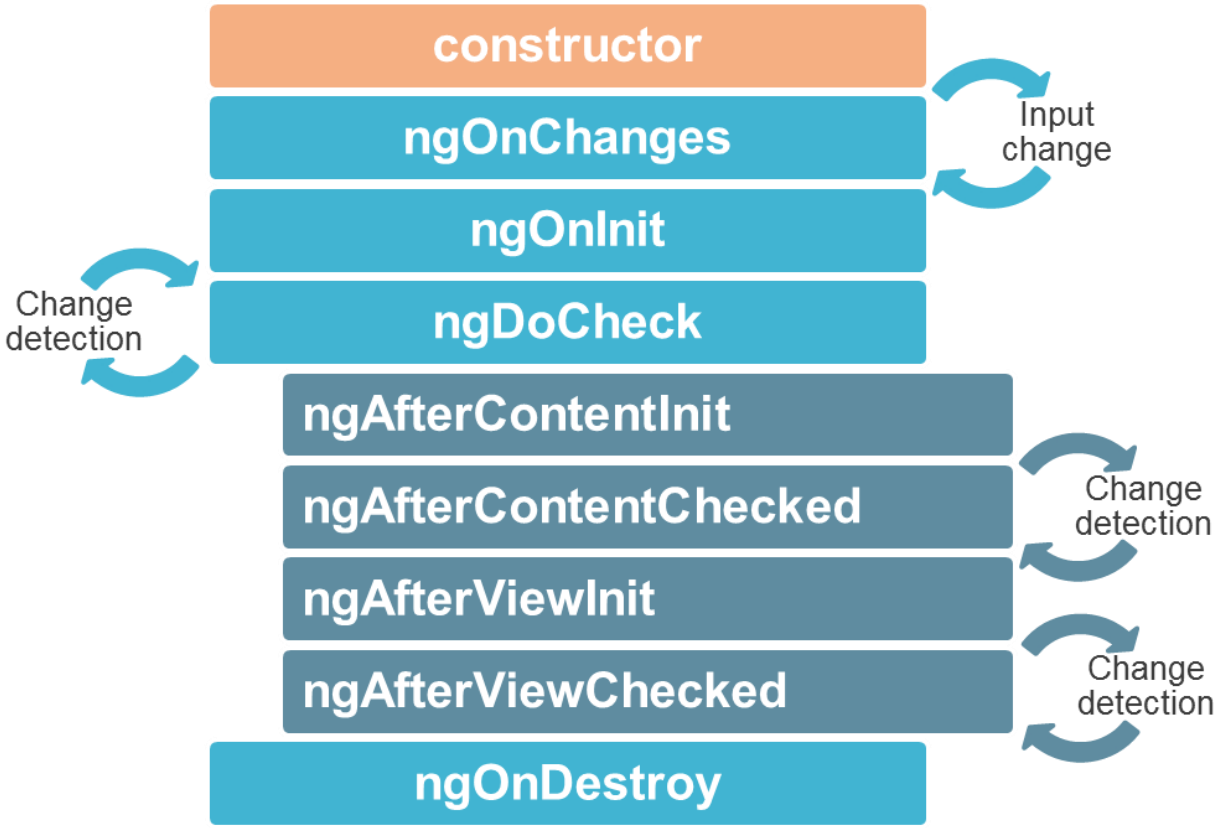
→ **ngOnInit() :** appelée une seule fois lors de la création d'un composant juste après le premier appel de *ngOnChange* (souvent appel aux API pour récupérer les données).

→ **ngDoCheck() :** mise en œuvre pour connaître les changements des valeurs internes d'objets ou de listes (ceux non identifiables par *ngOnChanges*)

→ **ngOnDestroy() :** appelée juste avant que le composant soit désalloué.

→ Et d'autres : *ngAfterContentInit*(), *ngAfterContentChecked*(), etc.

ANGULAR : CYCLE DE VIE



ANGULAR : CYCLE DE VIE

→ Exemple de fonction **ngOnInit** :

```
import { Component, OnInit } from '@angular/core';
import { PersonnagesComponent } from './personnages/personnages.component';

@Component({
  selector: 'app-root',
  imports: [PersonnagesComponent],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})
export class AppComponent implements OnInit{
  ngOnInit(): void {
    console.log(`OnInit AppComponent`);
  }
  title = 'TP-Angular';
}
```

FRAMEWORK ANGULAR

Utilisation de Framework

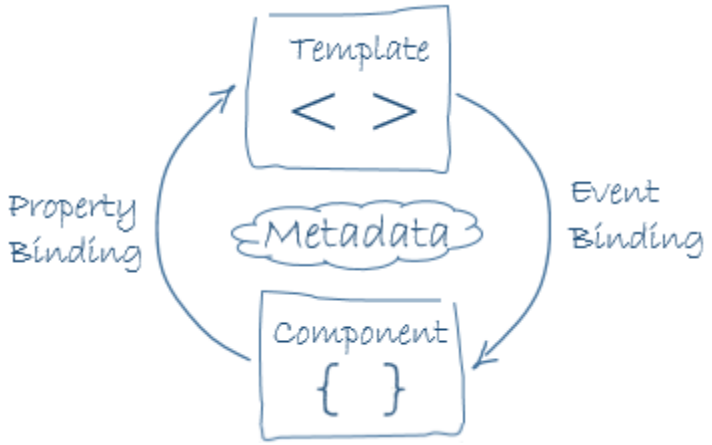
Notion de composant web

Concept d'Angular

- Component / Template
- **Data Binding**

ANGULAR : DATA BINDING

→ Permet de créer une relation entre les données d'un composant et les valeurs correspondantes affichées dans la vue.



Différentes catégories de binding :

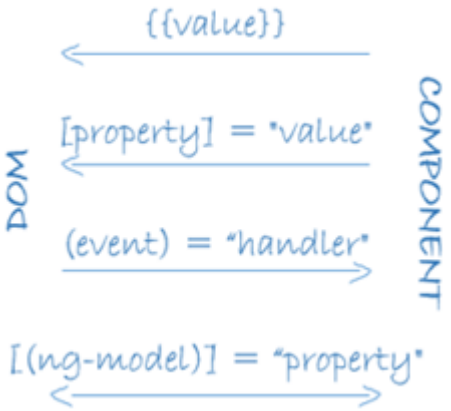
→ Interpolation **{{ }}**

→ Du composant à la vue **[ ]**

→ De la vue au composant **( )**

→ Et dans les deux directions - two-way binding **[( )]**

→ Control flow **@for, @if**, etc.



ANGULAR : DATA BINDING

Interpolation :

Une **variable scalaire** est injectée dans le **template**.

→ Si un composant modifie les données, la vue sera automatiquement mise à jour.

```
ular > src > app > personnages > personnages.component.html
Go to component
<p>Personnage {{MyHero.name | uppercase}}</p>
<img src="{{MyHero.urlImage}}"/>
```

ANGULAR : DATA BINDING

Property binding :

Si la variable à interpoler dans le template est la valeur d'un attribut d'une balise HTML, la gestion de cet attribut peut être délégué à Angular.

→ Attribut HTML encadré par des crochets, devient une directive de l'attribut.

```
gular > src > app > personnages > <> personnages.component.html > ...
  Go to component
  <p>Personnage {{MyHero.name | uppercase}}</p>
  <img src="{{MyHero.urlImage}}"/>
  <img [src]="MyHero.urlImage" alt="Image de {{MyHero.name}}"/>
```

ANGULAR : DATA BINDING

Property binding : **[target]="expression"**

Le target peut être une propriété d'un élément, d'un composant ou d'une directive.

```
<img [src]="heroImageUrl">
<app-hero-detail [hero]="currentHero"></app-hero-detail>
<div [ngClass]="{'special': isSpecial}"></div>
```


Cette syntaxe utilisé également pour les attributs, classe et style :

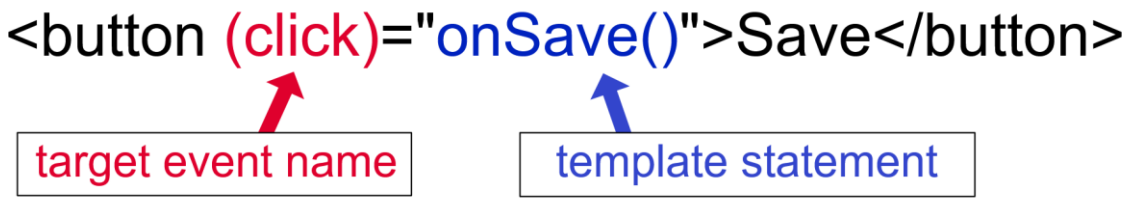
```
<button [attr.aria-label]="help">help</button>
<div [class.special]="isSpecial">Special</div>
<button [style.color]="isSpecial ? 'red' : 'green'"></button>
```

ANGULAR : DATA BINDING

Event binding :

Permet à Angular d'exécuter du code ou des actions lorsqu'un évènement est levé.

→ Clics, mouvements de souris, frappes au clavier, manipulations tactiles, etc.




```
<button (click)="onSave()">Save</button>
<app-hero-detail (deleteRequest)="deleteHero()"></app-hero-detail>
<div (myClick)="clicked=$event" clickable>click me</div>
```

→ Evènement sur un élément, un composant ou un directive.

ANGULAR : DATA BINDING

Two-way binding : [(target)]="expression"

Permet de lier une propriété de la classe TypeScript implémentant le composant avec une interface de saisie/sélection du template (input, select, textarea, etc.)

→ Une modification dans le template de la valeur de la zone de saisie met à jour immédiatement la valeur de la propriété dans le classe.

→ La mise à jour de la variable au sein de la propriété est immédiatement répercutée dans le template.

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

Plus de détail dans la suite du cours pour le data binging entre composants

ANGULAR : DATA BINDING

Exemple pour la partie Angular basé sur Zelda :

→ Utilisation du composant racine du projet Angular.

→ Nouveau composant Personnages.

→ Simuler des données d'un back pour les afficher dans le template de Personnages

→ Modification du comportement du composant Personnages (ajout databing).

ANGULAR : EXEMPLE

Modification de la classe TS du composant Personnages

```
11  export class PersonnagesComponent implements OnInit {
12
13    listPersonnages = PERSONNAGES;
14    myHero!: IPersonnage;
15
16    constructor() { }
17    ngOnInit(): void {
18    }
19
20    selectedPersonnage(hero: IPersonnage){
21      this.myHero = hero;
22    }
23  }
```

ANGULAR : DATA BINDING

Modification du fichier CSS

(afin de rendre plus visible les modifications…)

TP-Angular > src > app > personnages > # personnages.component.css >

```
1  .personnages li {
2      cursor: pointer;
3  }
4  .personnages li.selected {
5      background-color: rgb(176, 195, 220);
6      color: seagreen
7  }
```

ANGULAR : EXEMPLE

Modification du template du composant Personnages

TP-Angular > src > app > personnages > personnages.component.html > ...

Go to component

```
<div class="container-fluid">
    <p>Liste de tous les personnages :</p>
    @for(hero of listPersonnages; track hero.id) {
        <ul class="list-group personnages">
            <li class="list-group-item"
                [class.selected]="hero===MyHero"
                (click)="selectedPersonnage(hero)">
                <span class="badge rouded-pill bg-info text-dark">
                    {{hero.id}}
                </span>
                {{hero.name}}
            </li>
        </ul>
    }
    @empty {
        <p>Aucun personnage chargé ...</p>
    }
</div>
```

ANGULAR : DATA BINDI

→ Suite du template …

```
19  @if(MyHero){
20      <div>
21          <p>Personnage {{MyHero.name | uppercase}}</p>
22          <img [src]="MyHero.urlImage" alt="Image de {{MyH
23      </div>
24  } @else {
25      <div>
26          <p>Veuillez sélectionner un personnage.</p>
27      </div>
28  }
```

ANGULAR : DATA BINDING

```
getValue(event: Event): string {
  console.dir(event);
  return (event.target as HTMLInputElement).value;
}
```

Event binding :

→ Possibilité d'avoir l'objet **$event** comme paramètre à la méthode.

→ Le type de l'objet **$event** dépend du target (DOM element event)

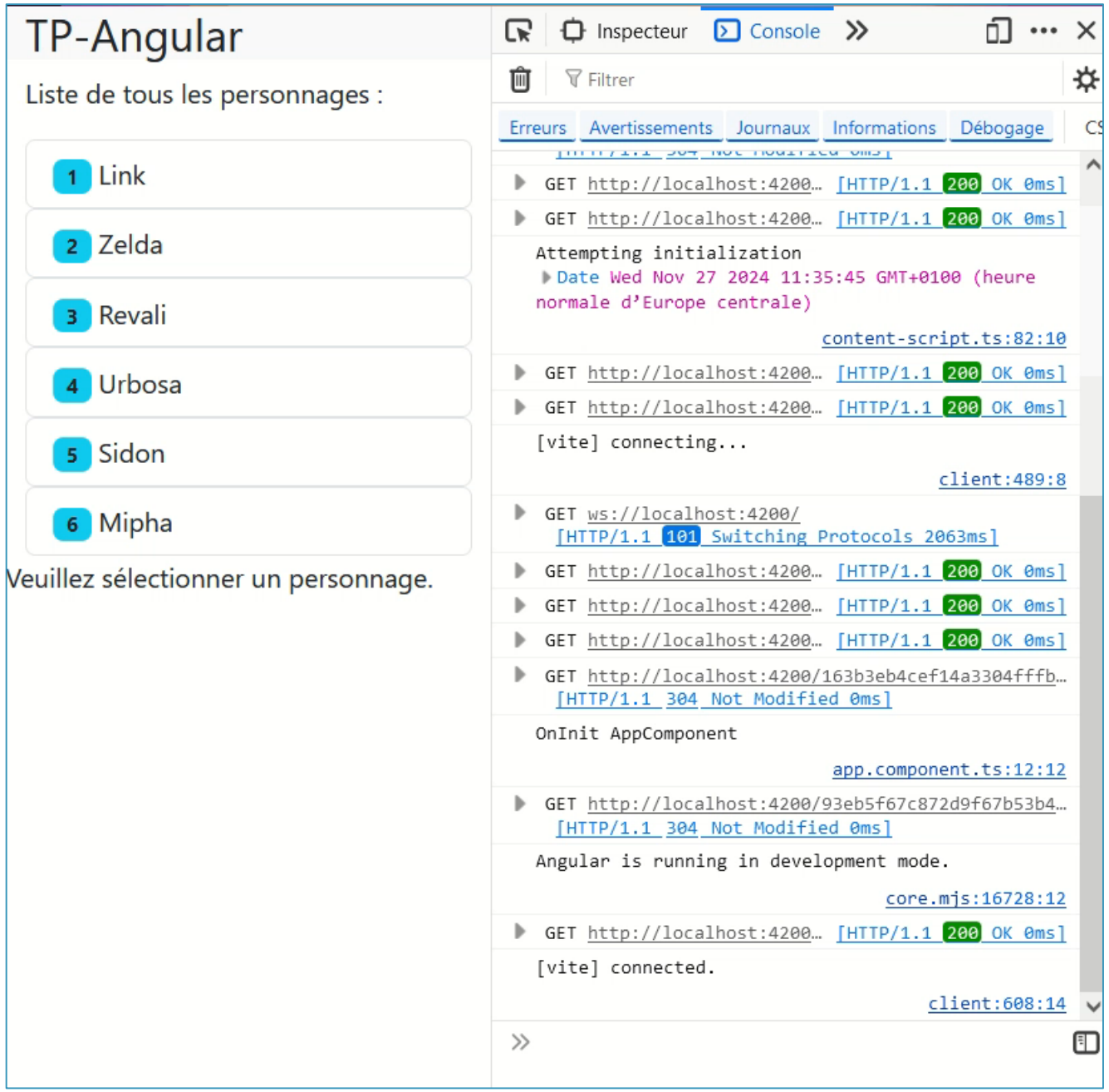
InputEvent

```
19  @if(MyHero){
20      <div>
21          <p>Personnage {{MyHero.name | uppercase}}</p>
22          <img [src]="MyHero.urlImage" alt="Image de {{MyHero.name}}"/>
23          <input [value]="MyHero.name"
24                 (input)="MyHero.name=getValue($event)"/>
25      </div>
26  } @else {
27      <div>
28          <p>Veuillez sélectionner un personnage.</p>
29      </div>
30  }
```

ANGULAR :

Event binding :

→ Résultat :



FRAMEWORK ANGULAR

Utilisation de Framework

Notion de composant web

Concept d'Angular

- Component / Template
- Data Binding
- **Component Interaction**

ANGULAR : INTERACTION ENTRE COMPOSANT

Une application *Angular* est composée de plusieurs composants.

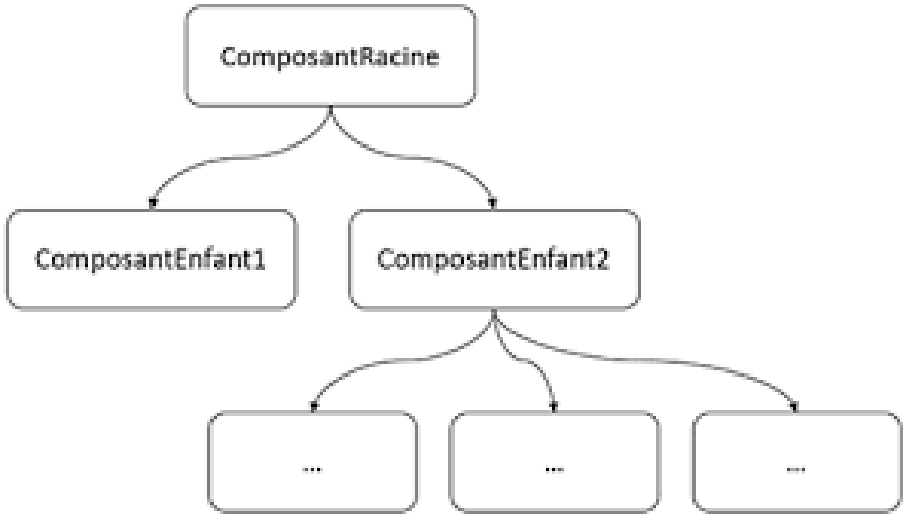
→ De quelle manière les composants peuvent interagir entre eux ?

→ **Déclarer les inputs et outputs d'un composant**

→ **EventEmitter**

→ **Variable locale**

→ **Décorateur ViewChild**



ANGULAR : NOTION DE SIGNAL

→ Concept de signal depuis Angular 16

→ Chaque nouvelle version apporte des nouveautés sur les signals (17.1, 18.2, etc.)

→ Représente une information qui peut changer au fil du temps

→ Wrapper autour d'un valeur permettant de signaler lorsque cette valeur change.

→ Un signal peut être en écriture ou en lecture seule

ANGULAR : NOTION DE SIGNAL

→ **Création d'un signal**

```
const count = signal(0);

// Signals are getter functions - calling them reads their value.
console.log('The count is: ' + count());
```


→ **Modification de la valeur d'un signal**

```
count.set(3);

// Increment the count by 1.
count.update(value => value + 1);
```

https://angular.fr/signals/

ANGULAR : NOTION DE SIGNAL

→ **Computed signal (dépend de la valeur d'autres signaux)**

```
const count: WritableSignal<number> = signal(0);
const doubleCount: Signal<number> = computed(() => count() * 2);
```


→ **Mise en place d'un effet (réagit à des évènements/changements spécifiques et déclenche une action)**

```
@Component({...})
export class EffectiveCounterComponent {
  readonly count = signal(0);
  constructor() {
    // Register a new effect.
    effect(() => {
      console.log(`The count is: ${this.count()}`);
    });
  }
}
```

https://angular.dev/guide/signals#what-are-signals

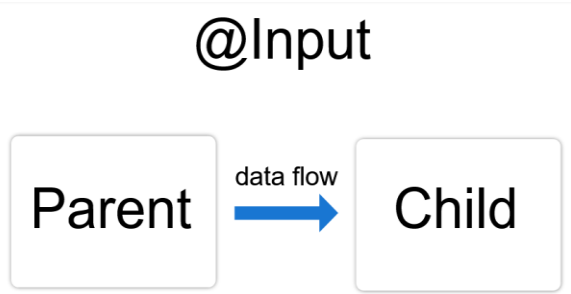
ANGULAR : INTERACTION ENTRE COMPOSANTS

→ **Notion de input et output entre composants**

→ Directive @Input ou signal input()

→ Directive @Output ou signal output()

ANGULAR : INPUT



@Input ou signal input()

→ Permet d'identifier une propriété du composant en tant qu'input.

→ Permet au **composant parent** de mettre à jour une donnée du **composant enfant.**

TS du composant enfant :

```ts
import { Component, Input } from '@angular/core';
export class ItemDetailComponent {
  @Input() item = '';
}
```

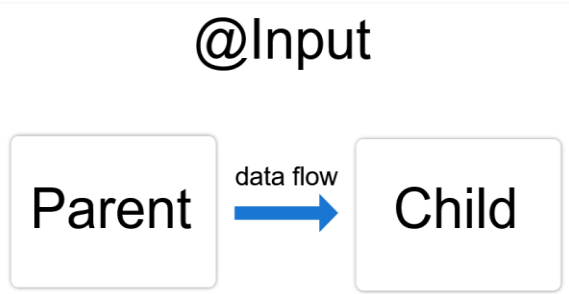
Template du composant parent :

Source (propriété du parent)

```html
<app-item-detail [item]="currentItem"></app-item-detail>
```

Target (propriété de l'enfant)

ANGULAR : INPUT



@Input ou signal **input()**

→ Permet d'identifier une propriété du composant en tant qu'input.

→ Permet au **composant parent** de mettre à jour une donnée du **composant enfant.**

TS du composant enfant :

```ts
import {Component, input} from '@angular/core';
export class CustomSlider {
  item = input('');
  item2 = input<String>();
}
```

Template du composant parent :

Source (propriété du parent)

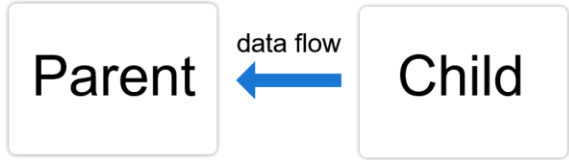
```
<app-item-detail [item]="currentItem"></app-item-detail>
```

Target (propriété de l'enfant)

https://angular.dev/guide/components/inputs

ANGULAR : OUTPUT

@Output



@Ouput ou le signal **output() :**

→ Permet de notifier le composant parent qu'un évènement s'est produit au sein du composant enfant.

→ Utilisation de la classe **EventEmitter** afin de notifier le composant parent (évènements personnalisés)

TS du composant enfant :

```ts
import { Output, EventEmitter } from '@angular/core';
export class ItemOutputComponent {

  @Output() newItemEvent = new EventEmitter<string>();

  addNewItem(value: string) {
    this.newItemEvent.emit(value);
  }
}
```

Template du composant parent :

```html
<app-item-output (newItemEvent)="addItem($event)"></app-item-output>
```

ANGULAR : OUTPUT

@Output

@Ouput ou le signal **output() :**

→ Permet de notifier le composant parent qu'un évènement s'est produit au sein du composant enfant.

→ Utilisation de la classe **EventEmitter** afin de notifier le composant parent (évènements personnalisés)

TS du composant enfant :

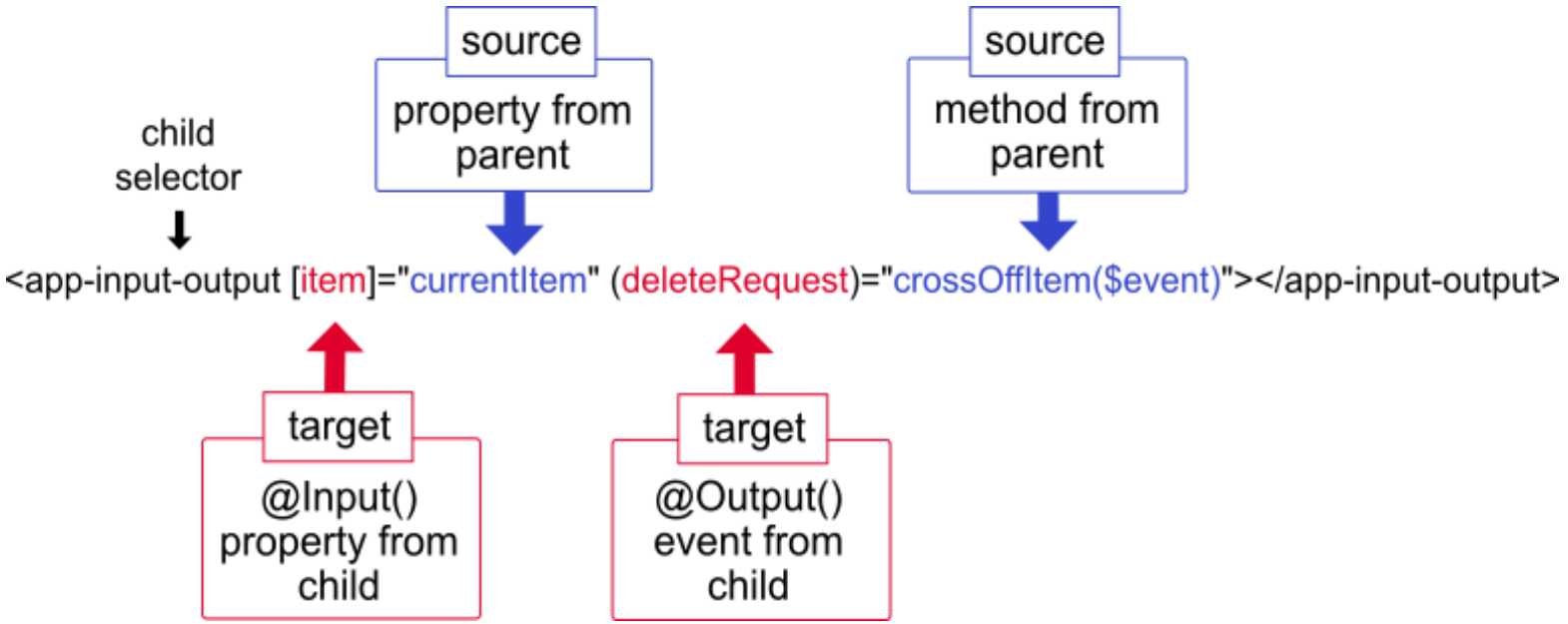
```ts
import { output } from '@angular/core'
export class ExpandablePanel {
  newItemEvent = output<string>();

  addNewItem(value: string) {
    this.newItemEvent.emit(value);
  }
}
```

Template du composant parent :

```html
<app-item-output (newItemEvent)="addItem($event)"></app-item-output>
```

ANGULAR : INPUT ET OUTPUT

Possibilité de combiner **@Input** et **@Output** :



ANGULAR : INPUT ET OUTPUT

→ **Notion de input et output entre composants**

→ Pour cet exemple : séparation de la liste des personnages et le détail d'un personnage

→ Un composant **personnages** et un nouveau composant **personnage-detail**

→ Ajout également d'une liste d'armes pour un personnage

ANGULAR : INPUT ET OUTPUT

→ Résultat attendu :

Composant Personnages

Composant Personnage-detail

TS du **composant enfant** (personnages-detail) :

TP-Angular > src > app > personnages-detail > TS personnages-detail.component.ts > ...


```ts
import { Component, input, output } from '@angular/core';
import { IPersonnage } from '../interface/personnage.interface';
import { CommonModule } from '@angular/common';

@Component({
  selector: 'app-personnages-detail',
  imports: [CommonModule],
  templateUrl: './personnages-detail.component.html',
  styleUrl: './personnages-detail.component.css'
})
export class PersonnagesDetailComponent {

  myHero = input.required<IPersonnage>();

  newWeaponEvent = output<string>();

  modifyWeapon(value: string){
    this.newWeaponEvent.emit(value);
  }

}
```

ANGULAR : EXEMPLE

Template du **composant enfant** (personnages-detail) :

TP-Angular > src > app > personnages-detail > personnages-detail.component.html

Go to component

```
@if(myHero()){
    <div>
        <p>Personnage {{myHero().name | uppercase}}</p>
        <img [src]="myHero().urlImage" alt="Image de {{myHero().name}}"/>
        <label for="weapon-input">Quelle arme ?</label>
        <input type="text" id="weapon-input" #newWeapon />
        <button (click)="modifyWeapon(newWeapon.value)">Modifier arme</button>
    </div>
} @else {
    <div>
        <p>Veuillez sélectionner un personnage.</p>
    </div>
}
```

ANGULAR : EXEMPLE

TS du **composant parent** (personnages) :

TP-Angular > src > app > personnages > TS personnages.component.ts > ...

```
14  export class PersonnagesComponent {
15    listPersonnages = PERSONNAGES;
16    myHero!: IPersonnage;
17
18    selectedPersonnage(hero: IPersonnage){
19      this.myHero = hero;
20    }
21    modifyWeapon(newWeapon: string){
22      this.myHero.weapons.push(newWeapon);
23    }
24  }
```

Template du **composant parent** (personnages) :

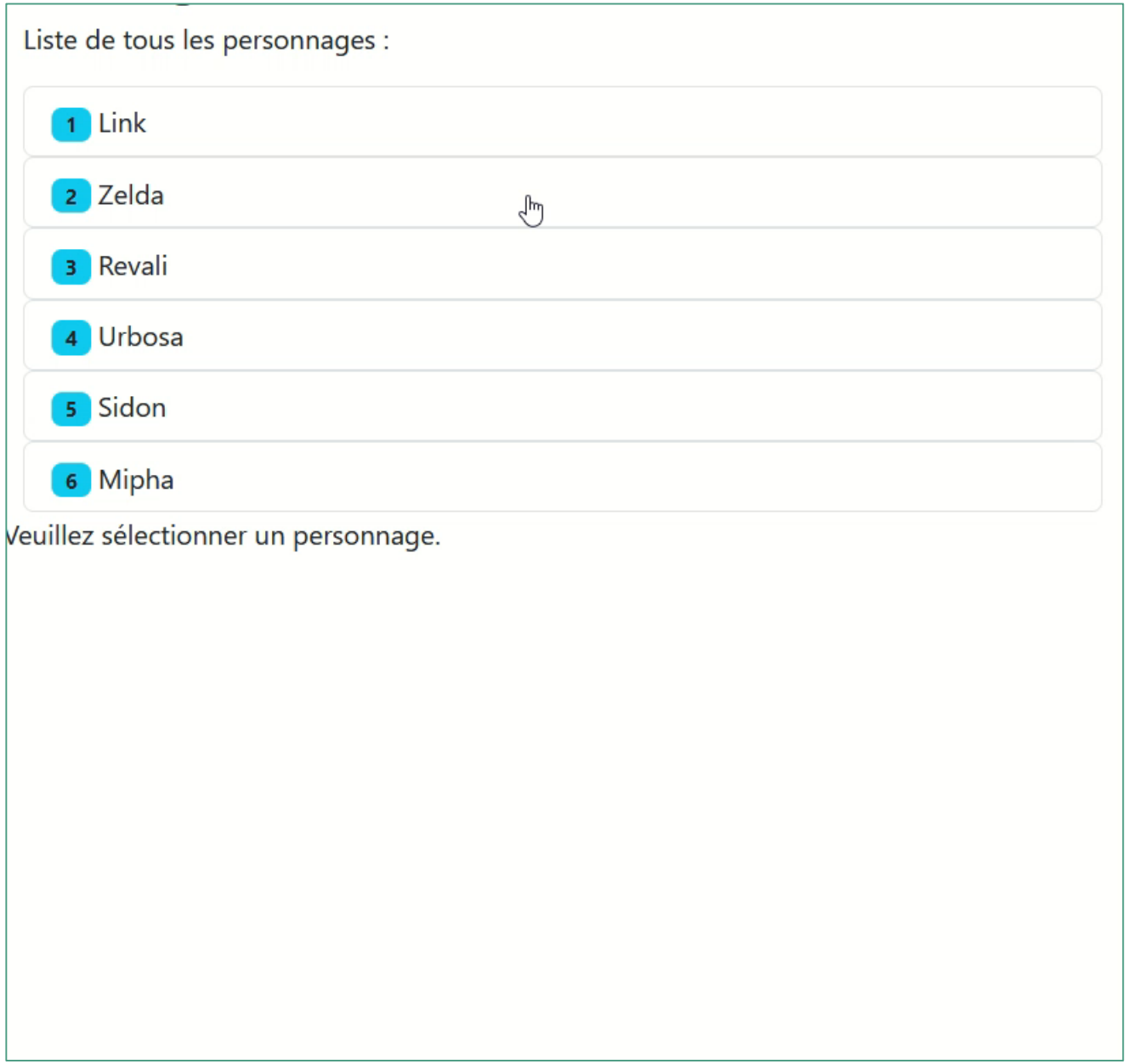
TP-Angular > src > app > personnages > personnages.component.html

Go to component

```
1  <div class="container-fluid">
2      <p>Liste de tous les personnages :</p>
3      @for(hero of listPersonnages; track hero.id) {
4          <ul class="list-group personnages">
5              <li class="list-group-item"
6                  [class.selected]="hero===myHero"
7                  (click)="selectedPersonnage(hero)">
8                  <span class="badge rouded-pill bg-info text-dark">
9                      {{hero.id}}
10                 </span>
11                 {{hero.name}}
12                 @for(arme of hero.weapons ; track $index) {
13                     <span>{{arme}}</span>
14                 }
15             </li>
16         </ul>
17     }
18     @empty {
19         <p>Aucun personnage chargé ...</p>
20     }
21 </div>
22 <app-personnages-detail [myHero]="myHero" (newWeaponEvent)="modifyWeapon($event)"></app-personnages-detail>
```

ANGULAR :

Résultat :



ANGULAR : INTERACTION ENTRE COMPOSANT

variable locale (template référence variable) :

→ Permet d'utiliser une donnée déclarée dans un **template** à un autre endroit de ce **template**.

→ Peut se référer à :

Un élément du DOM dans le template (e.g. input),

Un composant,

Une directive

#templateVariable

ANGULAR : INTERACTION ENTRE COMPOSANT

@ViewChild ou le signal viewChild() :

Permet de récupérer les données du premier composant fils à partir d'un composant parent (ou tout autre balise HTML présent dans le parent).

Composant disponible

```
export class PereComponent implements OnInit, AfterViewInit {
  @ViewChild(FilsComponent)
  private fils!: FilsComponent;
  constructor(){}
  ngAfterViewInit() {};
  ngOnInit(){};
}
```

@ViewChildren ou le signal viewChildren():

Possibilité à un composant parent de récupérer les données de ses composants enfants (QueryList).

ANGULAR : INTERACTION ENTRE COMPOSANT

@ViewChild ou le signal viewChild() :

Permet de récupérer les données du premier composant fils à partir d'un composant parent (ou tout autre balise HTML présent dans le parent).


```
detailPerso = viewChild(PersonnagesDetailComponent);
test = viewChild.required<ElementRef>("baliseP");

constructor() {
  effect( () => {
    console.log(this.detailPerso());
    console.log(this.test().nativeElement.innerText);
  } );
}
```

@ViewChildren ou le signal viewChildren():

Possibilité à un composant parent de récupérer les données de ses composants enfants (QueryList).

FRAMEWORK ANGULAR

Utilisation de Framework

Notion de composant web

Concept d'Angular

- Component / Template
- Data Binding
- Component Interaction
- **Service**

ANGULAR : SERVICE

→ Un composant doit se focaliser sur la représentation des données (user experience). Il ne doit pas se soucier des mécanismes et transformations mis en place pour récupérer ces données.

→ **Délégation du traitement des données au service** (fetching data, validation d'une saisie utilisateur, système de log).

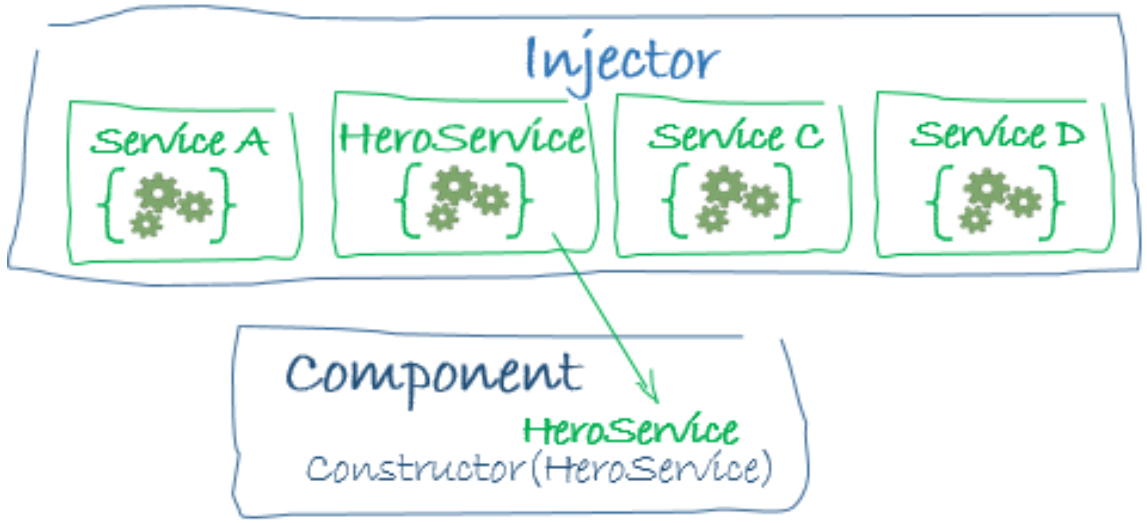
→ Un service permet de partager facilement des informations entre classes qui n'ont pas de lien.

→ **Permet d'augmenter la modularité et la réutilisation des composants.**

ANGULAR : INJECTION DE DÉPENDANCE

→ Les dépendances sont des services ou des objets dont une classe à besoin.

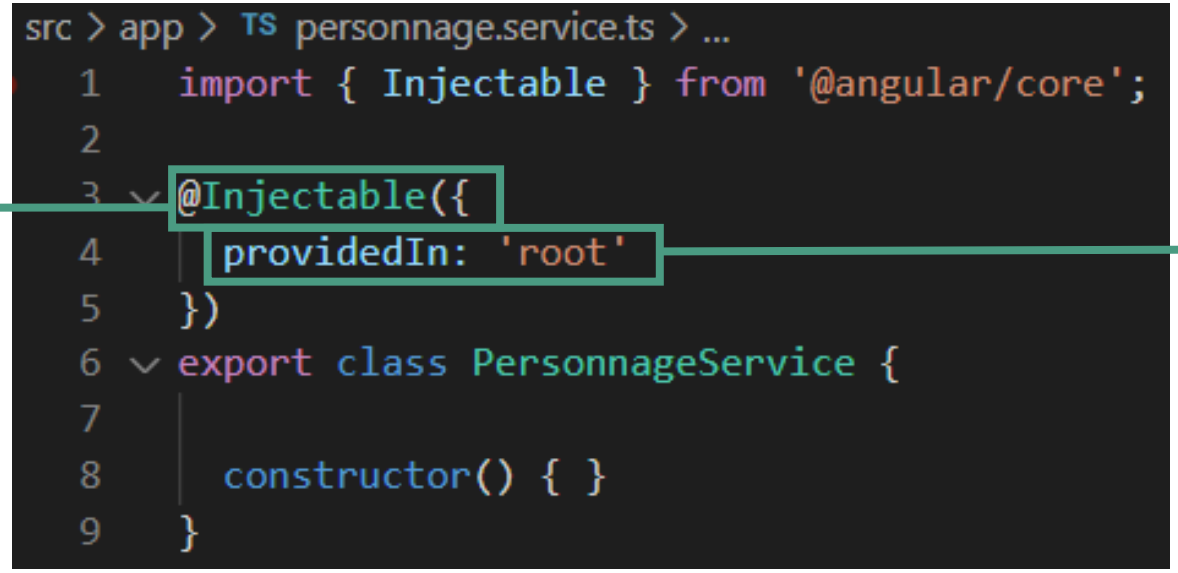
→ Les injections de dépendances (DI) est un design pattern.



ANGULAR : SERVICE

Déclaration d'un service

Décorateur : Angular peut utiliser cette classe dans le DI



Visible dans toute l'application.

Ajout au provider.

ANGULAR : INJECTION DE DÉPENDANCE

Autre manière possible (versions Angular inférieures à 6)

→ Utilisation de **providers** (metadata) pour spécifier les services dont le composant à besoin.

Dans un composant :

```
@Component({
  selector: 'app-personnages',
  templateUrl: './personnages.component.html',
  styleUrls: ['./personnages.component.css'],
  providers:[PersonnageService]
})
export class PersonnagesComponent implements OnInit {
```

ANGULAR : INJECTION DE DÉPENDANCE

Injection d'un service

→ Pas d'utilisation du **new**

→ Dans le constructeur du composant ayant besoin du service :

```
constructor(private persoService:PersonnageService) { }
```

→ Ou en utilisant la méthode **inject()**

ANGULAR : INJECTION DE DÉPENDANCE

Exemple pour la partie Angular basé sur Zelda :

→ Utilisation du composant racine du projet Angular.

→ Nouveau composant Personnages.

→ Simuler des données d'un back pour les afficher dans le template de Personnages

→ Modification du comportement du composant Personnages (ajout databing).

→ Création d'un nouveau composant : personnages-detail

→ Création de services pour récupérer les données des Personnages (et un simple logger).

ANGULAR : EXEMPLE

→ Récupération du **mock des personnages dans un service** et **utilisation d'un système de log** :

Service Logger :

TP-Angular > src > app > services > TS logger.service.ts > ...

```
1   import { Injectable } from '@angular/core';
2
3   @Injectable({
4     providedIn: 'root'
5   })
6   export class LoggerService {
7
8     logs: string[] = [];  //capture logs
9     constructor() { }
10    log(message: string){
11      this.logs.push(message);
12      console.log(message);
13    }
14    error(message: string){
15      console.error(message);
```

ANGULAR : EXEMPLE

Service Personnage :

TP-Angular > src > app > services > TS personnages.service.ts > ...

```ts
import { Injectable } from '@angular/core';
import { LoggerService } from './logger.service';
import { IPersonnage } from '../interface/personnage.interface';
import { PERSONNAGES } from '../personnages.mock';

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

  constructor(private loggerService: LoggerService) { }

  getPersonnages(): IPersonnage[] {
    this.loggerService.log('Getting personnages ...');
    return PERSONNAGES;
  }
}
```

Composant

Personnages :

TP-Angular > src > app > personnages > TS personnages.component.ts > ...

```
15  export class PersonnagesComponent {
16
17    listPersonnages!: IPersonnage[]; // = PERSONNAGES;
18    myHero!: IPersonnage;
19
20    detailPerso = viewChild(PersonnagesDetailComponent);
21    test = viewChild.required<ElementRef>("baliseP");
22
23    constructor(private personnagesService: PersonnagesService) {
24      effect( () => {
25        console.log(this.detailPerso());
26        console.log(this.test().nativeElement.innerText);
27      } );
28      this.getHeros();
29      console.log(this.listPersonnages);
30    }
31    getHeros(): void{
32      this.listPersonnages = this.personnagesService.getPersonnages();
33    }
```

ANGULAR : EXEMPLE

→ Résultat :

FRAMEWORK ANGULAR

Utilisation de Framework

Notion de composant web

Concept d'Angular

- Component / Template
- Data Binding
- Component Interaction
- Service
- **Observable (RXJS)**

ANGULAR : RXJS

Dans un réelle application : besoin de services asynchrones.

→ Utilisation de callback, de **Promise** ou de **Observable** (bibliothèque **RxJS**).

RxJS : Reactive extensions for javascript

→ combine Observer pattern et Iterator pattern.

- Observable, Observer, Subscription
- Operators
- Subject

ANGULAR : OBSERVABLE

Observable :

→ Objet permettant un échange d'information (stream).

→ Utilisé lors d'évènements, par la module HttpClient (get() retourne un Observable), etc.

→ Méthode *subscribe()* : abonne un traitement à l'observable.

→ Méthode *unsubscribe()* : désabonne un traitement à l'observable.

Un **observable** va émettre des évènements qui seront interceptés par les **observateurs**.

ANGULAR : OBSERVABLE

Méthode *subscribe() :* un seul paramètre (Observer)

Un Observer va « surveiller » l'Obervable et recevoir les valeurs émises.

→ **next** : se déclenche à chaque fois que l'Observable émet de nouvelles données (données en tant qu'argument)

→ **error** : se déclenche si l'Observable émet une erreur (erreur en tant qu'argument)

→ **complete** : se déclenche si l'Observable s'achève (aucun argument)

```
monObservable$.subscribe({
  next: (v) => console.log(v),
  error: (e) => console.error(e),
  complete: () => console.info('complete')
});
```

ANGULAR : OBSERVABLE

ANGULAR : OBSERVABLE

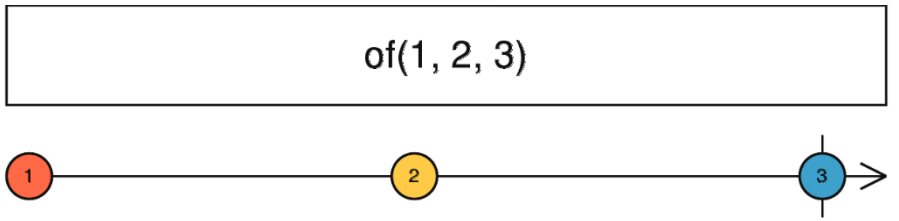
→ Plusieurs **opérateurs** de transformation disponibles :

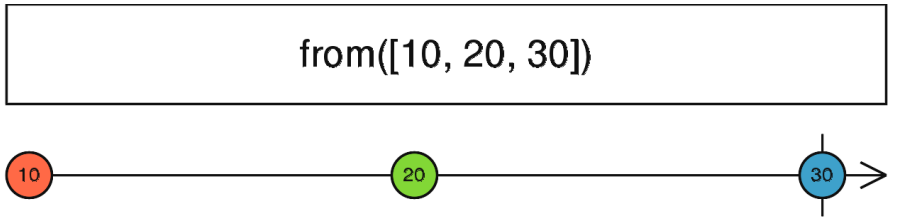
AREA	OPERATORS
Creation	`from`, `fromEvent`, `of`
Combination	`combineLatest`, `concat`, `merge`, `startWith` , `withLatestFrom`, `zip`
Filtering	`debounceTime`, `distinctUntilChanged`, `filter`, `take`, `takeUntil`
Transformation	`bufferTime`, `concatMap`, `map`, `mergeMap`, `scan`, `switchMap`
Utility	`tap`
Multicasting	`share`

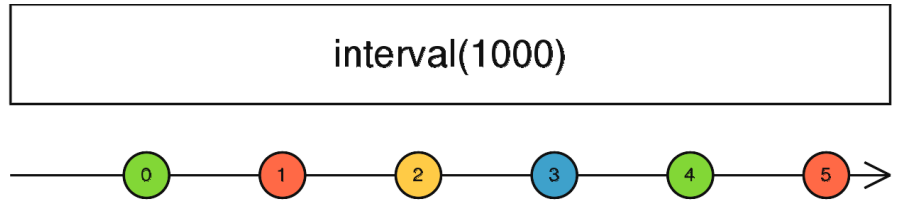
ANGULAR : OBSERVABLE

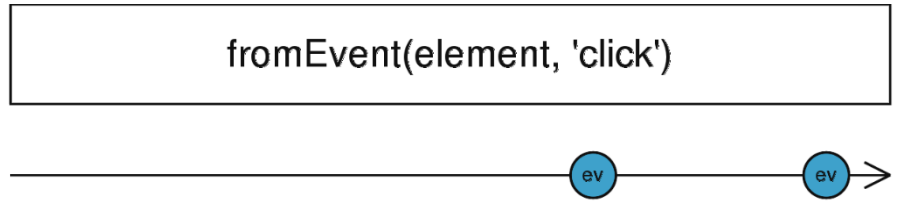
→ Plusieurs manières de **<u>créer</u>** un **Observable**.

```
let observableOf$ = of(1, 2, 3);
//emet chaque valeur/objet passé en paramètre
//output: 1, 2, 3
let observableFrom$ = from([1, 2, 3]);
//utiliser pour des array, promise
//emet le tableau comme une séquence de valeur
//output: 1, 2, 3
let observableInterval$ = interval(3000);
//emet toutes les 3 sec des valeurs
//output: 0, 1, 2, 3, 4 ....
let observableTimer$ = timer(1000,2000);
//commence à emettre la première valeur après 1 sec et emet toutes les 2 sec
//output: 0, 1, 2, 3, 4 ....
let observableEvent$ = fromEvent<MouseEvent>(document, 'mousemove');
//emet lorsque la souris est déplacée
//output: MouseEvent
```



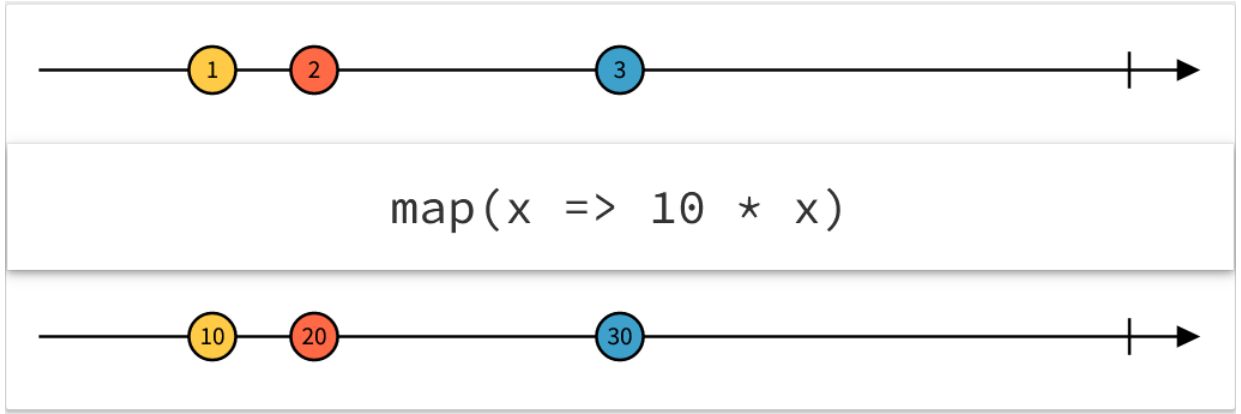




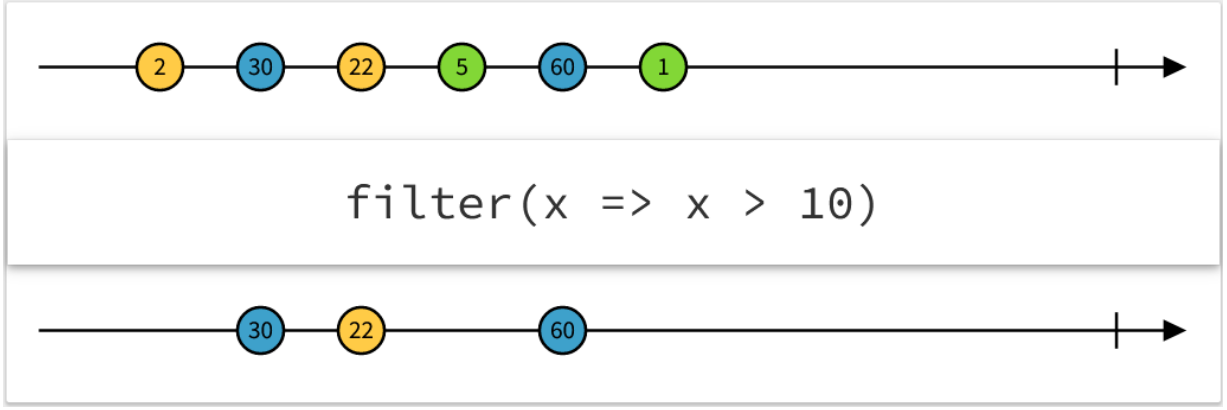


ANGULAR : OBSERVABLE

→ **Transformation**

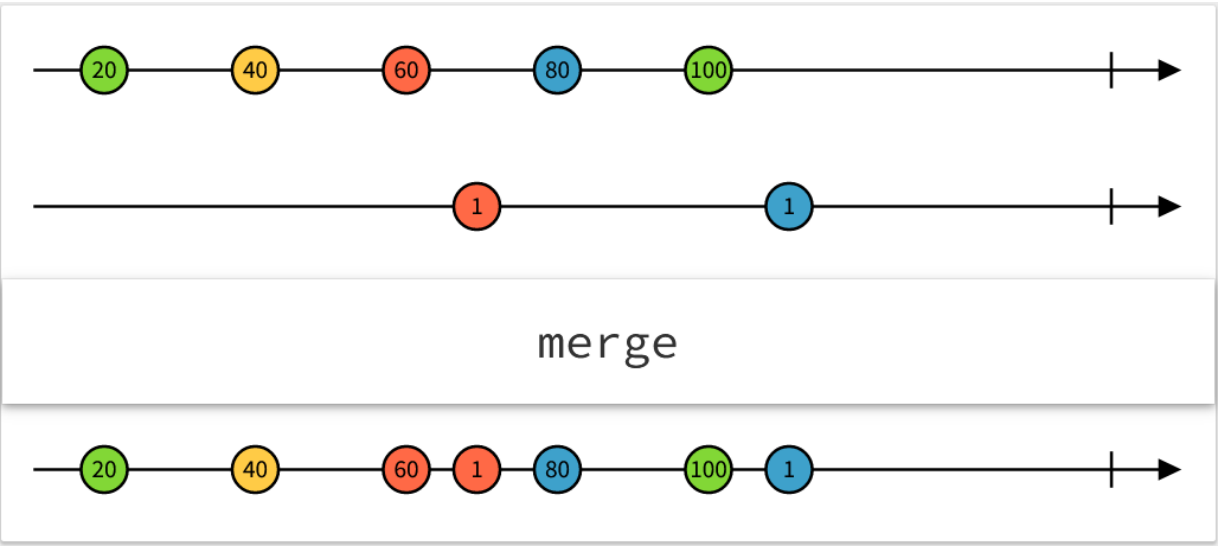


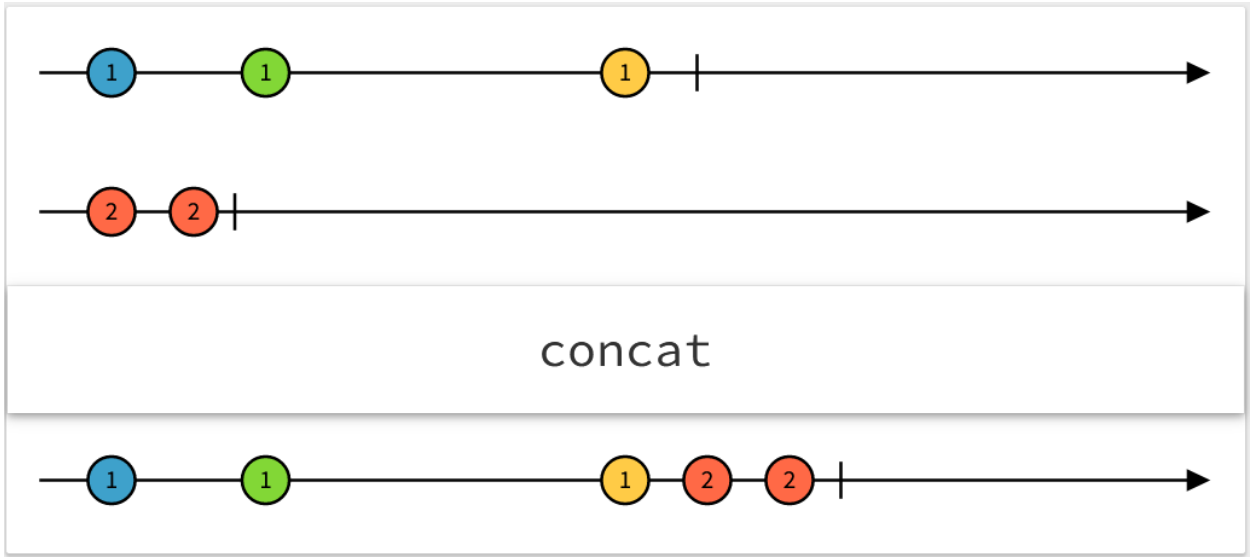
→ **Filtre**



ANGULAR : OBSERVABLE

→ **Combinaison**





ANGULAR : EXEMPLE

→ Modification de l'exemple précédent utilisant un **Observable** (asynchrone)

Service Personnage:

```
import { Observable, of } from 'rxjs';
```

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

  constructor(private loggerService: LoggerService) { }

  getPersonnages(): Observable<IPersonnage[]> {
    this.loggerService.log('Getting personnages ...');
    return of(PERSONNAGES);
  }
}
```

ANGULAR : EXEMPLE

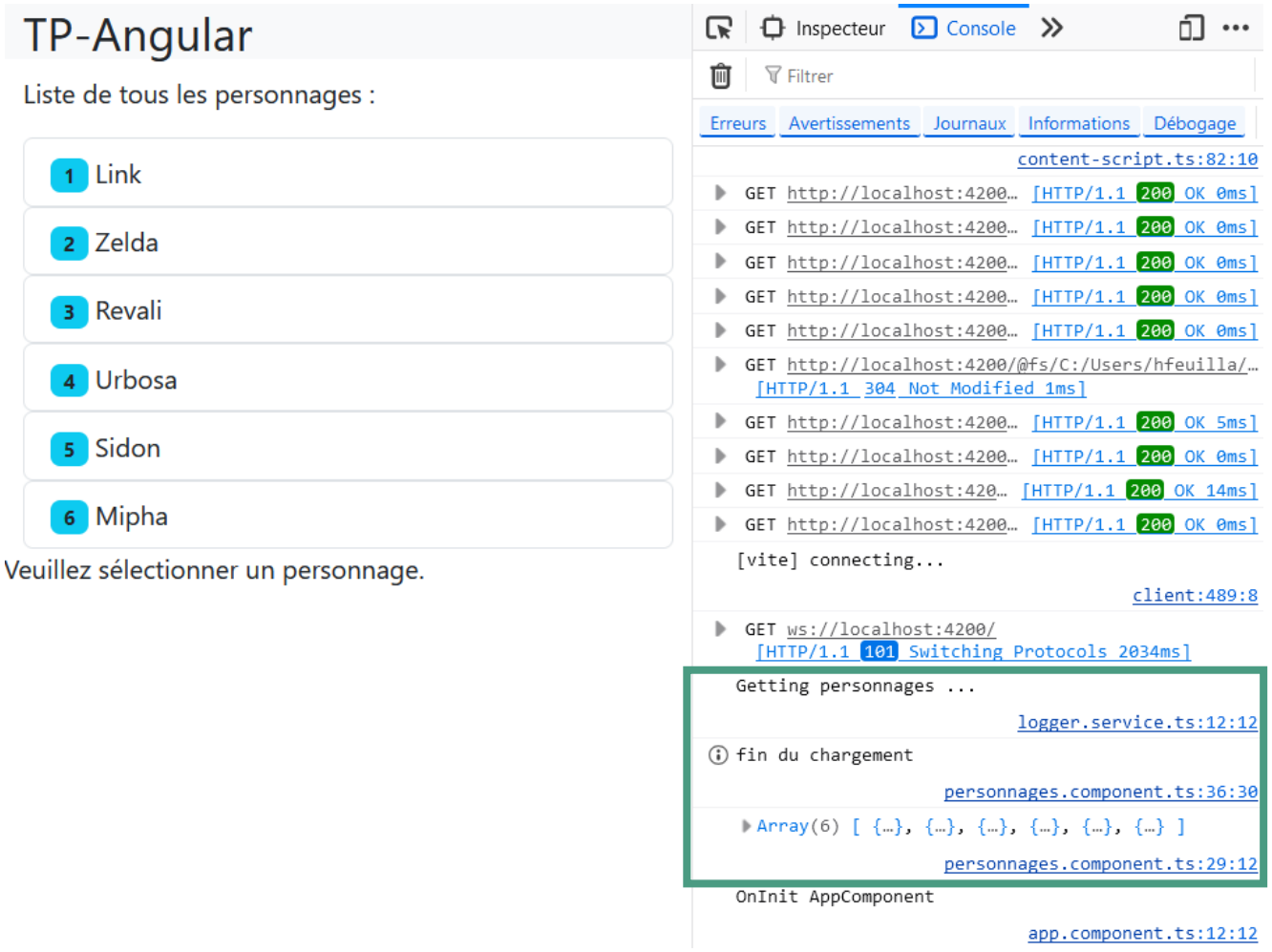
composant Personnage:

TP-Angular > src > app > personnages > TS personnages.component.ts > ...

```
15  export class PersonnagesComponent {
23    constructor(private personnagesService: PersonnagesService) {
24      effect( () => {
25        console.log(this.detailPerso());
26        console.log(this.test().nativeElement.innerText);
27      } );
28      this.getHeros();
29      console.log(this.listPersonnages);
30    }
31    getHeros(): void{
32      //this.listPersonnages = this.personnagesService.getPersonnages();
33      this.personnagesService.getPersonnages().subscribe({
34        next: personnages => this.listPersonnages = personnages,
35        error: error => console.error(error),
36        complete: () => console.info('fin du chargement')
37      })
38    }
```

ANGULAR : EXEMPLE

→ **Résultat identique :**



ANGULAR : SUBJECT

Subject

→ Type spécial d'Observable qui permet de partager les valeurs à plusieurs Observable.

→ Un **Subject** est un **Observable** (possibilité de souscrire à un Subject).

→ Un **Subject** est un **Observer** (object avec les callback next, error, complete).

→ **Plusieurs types :** subject, behabiorSubject, replaySubject, asyncSubject

ANGULAR : SUBJECT

```
const subject = new Subject<number>();
subject.subscribe({
next: (val) => console.log( `observerA: ${ v } ` )
});
subject.subscribe({
next: (val) => console.log( `observerB: ${ v } ` )
});
const observable = from([ 1 , 2 , 3 ]);
observable.subscribe(subject);
```

```
// observerA: 1
// observerB: 1
// observerA: 2
// observerB: 2
// observerA: 3
// observerB: 3
```

FRAMEWORK ANGULAR

Utilisation de Framework

Notion de composant web

Concept d'Angular

- Component / Template
- Data Binding
- Component Interaction
- Service
- Observable (RxJS)

Rooting

ANGULAR : ROUTING

→ Dans une **Single Page Application**, le contenu de la page est modifié au fur et à mesure (affichage de différents composants).

→ Notion de **navigation importante** pour les utilisateur (e.g. mimer le mécanisme de copier l'url, réaliser un retour en arrière dans le navigateur)

→ Solution : **créer des routes** afin de définir comment l'utilisateur navigue d'une partie de l'application à une autre.

https://angular.dev/guide/routing/router-reference

ANGULAR : ROUTING

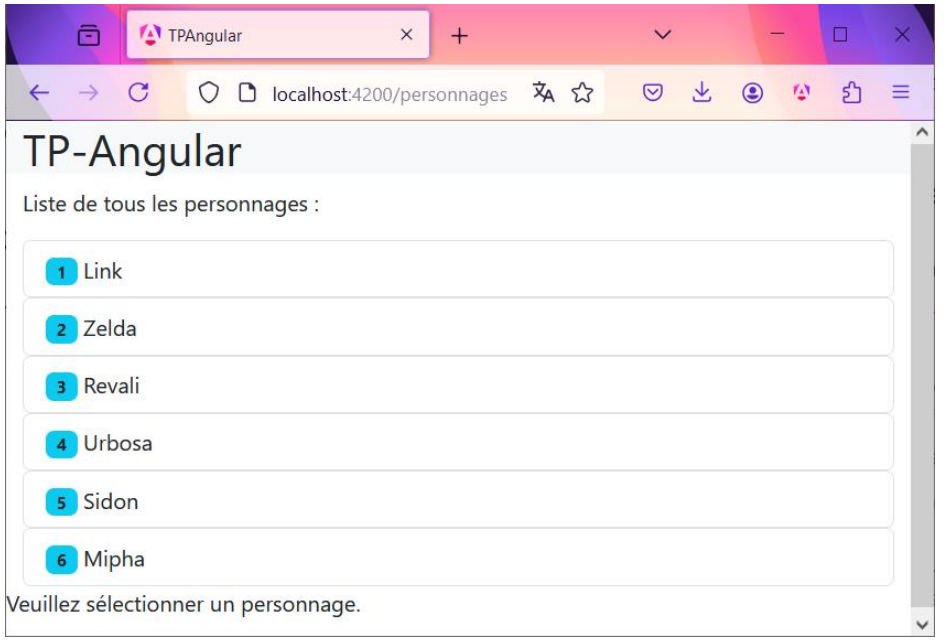
→ Mise en place d'un **routeur**

→ Fichier **app.config.ts**


```
import { ApplicationConfig, provideZoneChangeDetection } from '@angular/co
import { provideRouter } from '@angular/router';

import { routes } from './app.routes';

export const appConfig: ApplicationConfig = {
  providers: [
    provideZoneChangeDetection({ eventCoalescing: true }),
    provideRouter(routes)
  ]
};
```

Permet de charger le tableau de routes dans l'application

ANGULAR : ROUTING



→ Fichier **app.routes.ts** : définition de la liste de routes

Forme basique d'une route :

→ **Path** : chaîne de caractère pour l'URL

→ **Component** : le composant associé à l'URL

```
const routes: Routes = [
  {path: 'personnages', component: PersonnagesComponent}
];
```

→ Lorsque l'URL est demandée, le module de routage effectue le rendu du composant associé.

ANGULAR : ROUTING

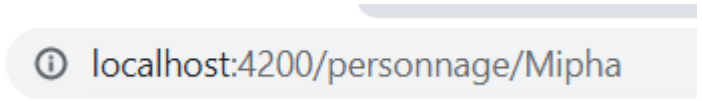
Attention à l'ordre des routes (utilisation de la première route correspondante)

→ Possibilité d'effectuer **une redirection** :

```
const routes: Routes = [
  {path: 'personnages', component: PersonnagesComponent},
  {path: '', redirectTo: 'personnages', pathMatch: 'full'}
];
```

→ Possibilité d'utiliser des URL **dynamiques** :

```
{path: 'personnage/:name', component: PersonnagesDetailComponent}
```



ANGULAR : ROUTING

→ Possibilité de rediriger l'utilisateur lorsqu'il rentre une URL qui n'existe pas (*wildcart route*)

→ Création d'une page 404 (nouveau composant)


```
{path: '**', component: PageNotFoundComponent}
```

→ **A ajouter en dernier dans la table de routage** !

ANGULAR : ROUTING

Comment intégrer les routes ?

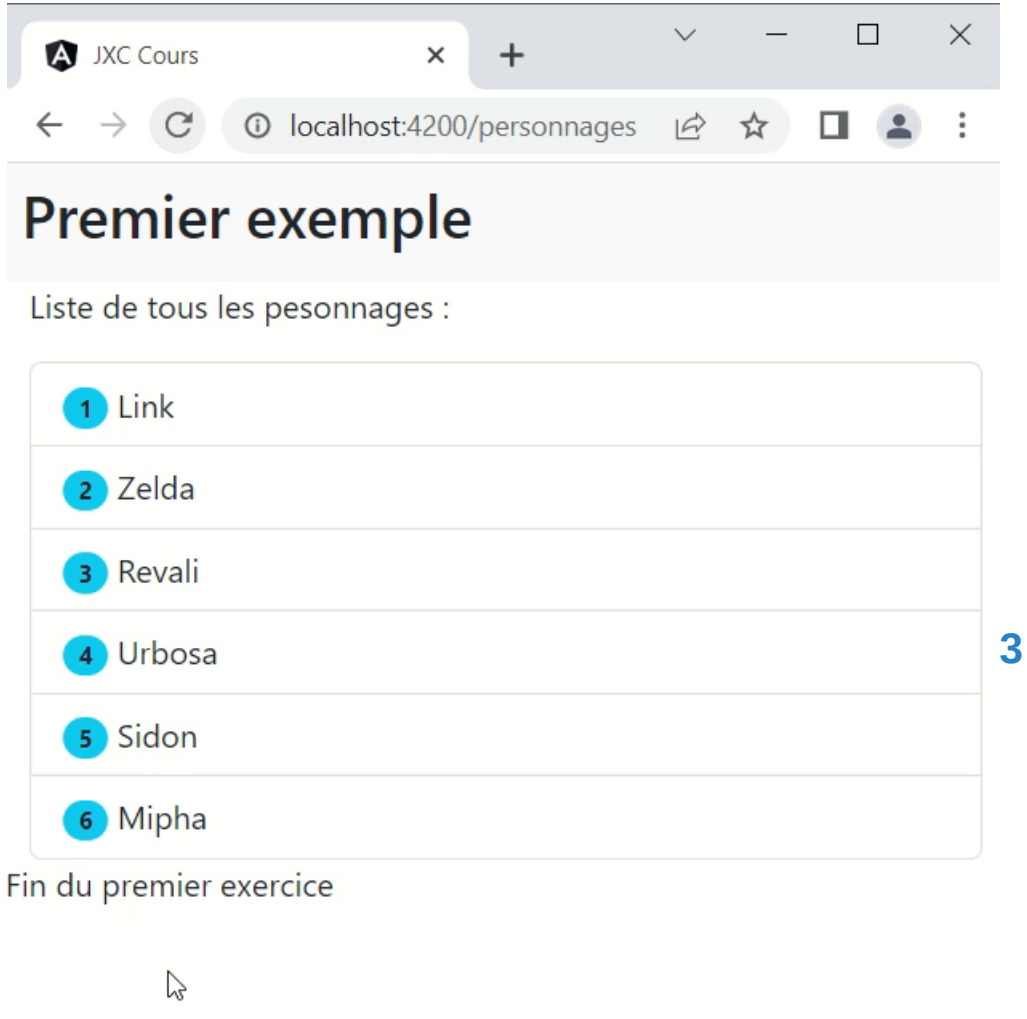
Directive **`RouterOutlet`**

→ Utilisée comme un composant

→ Permet de préciser à l'application où elle doit faire le rendu des composants associés aux routes.

```
src > app > app.component.html > ...
1   <header class="container-fluid bg-light p-2">
2     <h1>{{title}}</h1>
3   </header>
4   <!-- <app-personnages></app-personnages> -->
5   <router-outlet></router-outlet>
```

Le composant chargé sera affiché ici



ANGULAR : ROUTING

Comment naviguer dans l'application ?

1. A travers des liens : **`RouterLink`**
2. Directement dans le code du composant

ANGULAR : ROUTING

Problème : l'élément sera en gras pour toutes les pages visitées

Comment naviguer dans l'application ?

1. A travers des liens : **RouterLink**

```
.active {
    font-weight: bold;
}
```

Route active : on peut lui définir un style

→ Cas pour une **URL non dynamique** :

```
<nav>
    <ul>
      <li><a routerLink="/first-component" routerLinkActive="active">First Component</a></li>
      <li><a routerLink="/second-component" routerLinkActive="active">Second Component</a></li>
    </ul>
</nav>
```

→ Possibilité de créer un composant menu

ANGULAR : ROUTING

Comment naviguer dans l'application ?

1. A travers des liens : **RouterLink**

La classe est uniquement ajoutée lorsque la route correspond exactement à la valeur de *routerLink*

```
[routerLinkActiveOptions] = "{exact: true}"
```


→ Cas pour une **URL non dynamique** :

```
<nav>
    <ul>
      <li><a routerLink="/first-component" routerLinkActive="active">First Component</a></li>
      <li><a routerLink="/second-component" routerLinkActive="active">Second Component</a></li>
    </ul>
</nav>
```

→ Possibilité de créer un composant menu

ANGULAR : ROUTING

Comment naviguer dans l'application ?

1. A travers des liens : **RouterLink**

→ Cas pour une **URL dynamique**: → **/first-composant/:id**

/second-composant/:name

```
<nav>
    <ul>
      <li><a [routerLink]="['/first-composant', monObjet.id]" routerLinkActive="active">First Component ID</a></li>
      <li><a [routerLink]="['/second-composant', monObjet.name]" routerLinkActive="active">Second Component Name</a></li>
    </ul>
</nav>
```

ANGULAR : ROUTING

Comment naviguer dans l'application ?

1. A travers des liens : **RouterLink**

→ Cas pour une **URL dynamique**: → **/first-composant?id=1&nom:Zelda**

```
<a [routerLink]="[/personnage]" [queryParams]="{ id: '1', nom: 'Zelda'}">Personnage Zelda</a>
```

ANGULAR : ROUTING

Comment naviguer dans l'application ?

2. **Directement dans le code du composant**

→ Besoin d'injecter une instance de `Router` dans le composant

```
constructor(private router: Router) {}
```

→ Utilisation de la méthode **navigate / navigateByUrl**

```
goBack() {
  this.router.navigate(['previousComposant']);
}
```

Utile pour les retours !

ANGULAR : ROUTING

Comment naviguer dans l'application ?

2. Directement dans le code du composant

→ Autre possibilité pour aller à la vue précédente :
Utilisation du service **Location** de Angular.

```
constructor(private location: Location) {  }
```

```
goBack(): void {
  this.location.back();
}
```

ANGULAR : ROUTING

Récupération des données de routage ?

Possibilité d'utiliser une route pour passer des informations d'un composant à un autre.

→ Utilisation de l'interface **`ActivatedRoute`**.

→ Utilisation d'un objet de cette classe dans la méthode **`ngOnInit()`**

1) Les propriétés **snapshot.params / paramMap** contiennent **tous les paramètres passés à la route**.

Solution avec les snapshot (instantanée)

Solution avec les observables (asynchrone)

ANGULAR : ROUTING

Comment récupérer « 1 » dans l'URL /personnage/1 ?

Récupération des données de routage ?

→ Récupération de la route de la forme ***personnage/:id***


```
constructor(private route: ActivatedRoute,
  private router: Router, private persoService:PersonnageService) {}

ngOnInit(): void {
    const id = +this.route.snapshot.params.id;  //+ pour caster id en number
    this.persoService.getPersonnageObservable(id).subscribe(perso => this.perso = perso);
    //Observable pour récupérer le personnage avec le bon id
}
```

Snapshot.params

→ Paramètres de routage passés sous forme de chaînes de caractères.

ANGULAR : ROUTING

Comment récupérer « 1 » dans l'URL /personnage/1 ?

Récupération des données de routage ?

→ Récupération de la route de la forme ***personnage/:id***


```
constructor(private route: ActivatedRoute,
  private router: Router, private persoService:PersonnageService) {}

ngOnInit(): void {
  this.route.paramMap.subscribe( res => {
    const id = +res.get('id')!;
    this.perso = this.persoService.getPersonnage(id);
  });
}
```

paramMap

→ Paramètres de routage passés sous forme de chaînes de caractères.

ANGULAR : ROUTING

Récupération des données de routage ?

Possibilité d'utiliser une route pour passer des informations d'un composant à un autre.

→ Utilisation de l'interface `ActivatedRoute`.

→ Utilisation d'un objet de cette classe dans la méthode **`ngOnInit()`**

2) Les propriétés **snapshot.queryParams / queryParamMap** permettent de récupérer les paramètres de l'URL **sans définition d'une route**.

Solution avec les snapshot (instantanée)

Solution avec les observables (asynchrone)

ANGULAR : ROUTING

Récupération des données de routage ?

Quelle méthode choisir ?

→ **Snapshot** : Si la valeur initiale des paramètres est utilisée seulement à l'initialisation du composant et ne risque pas de changer.

→ **Observable** : Si la route risque de changer tout en restant dans le même composant (l'initialisation du composant à travers *ngOnInit()* ne sera pas appelée à nouveau mais l'observateur sera notifié lorsque l'URL est modifiée).

ANGULAR : INJECTION DE DÉPENDANCE

Exemple pour la partie Angular basé sur Zelda :

→ Utilisation du composant racine du projet Angular.

→ Nouveau composant Personnages.

→ Simuler des données d'un back pour les afficher dans le template de Personnages

→ Modification du comportement du composant Personnages (ajout databing).

→ Création d'un nouveau composant : personnages-detail

→ Création de services pour récupérer les données des Personnages (et un simple logger).

→ Création d'une barre de menu

ANGULAR : EXEMPLE

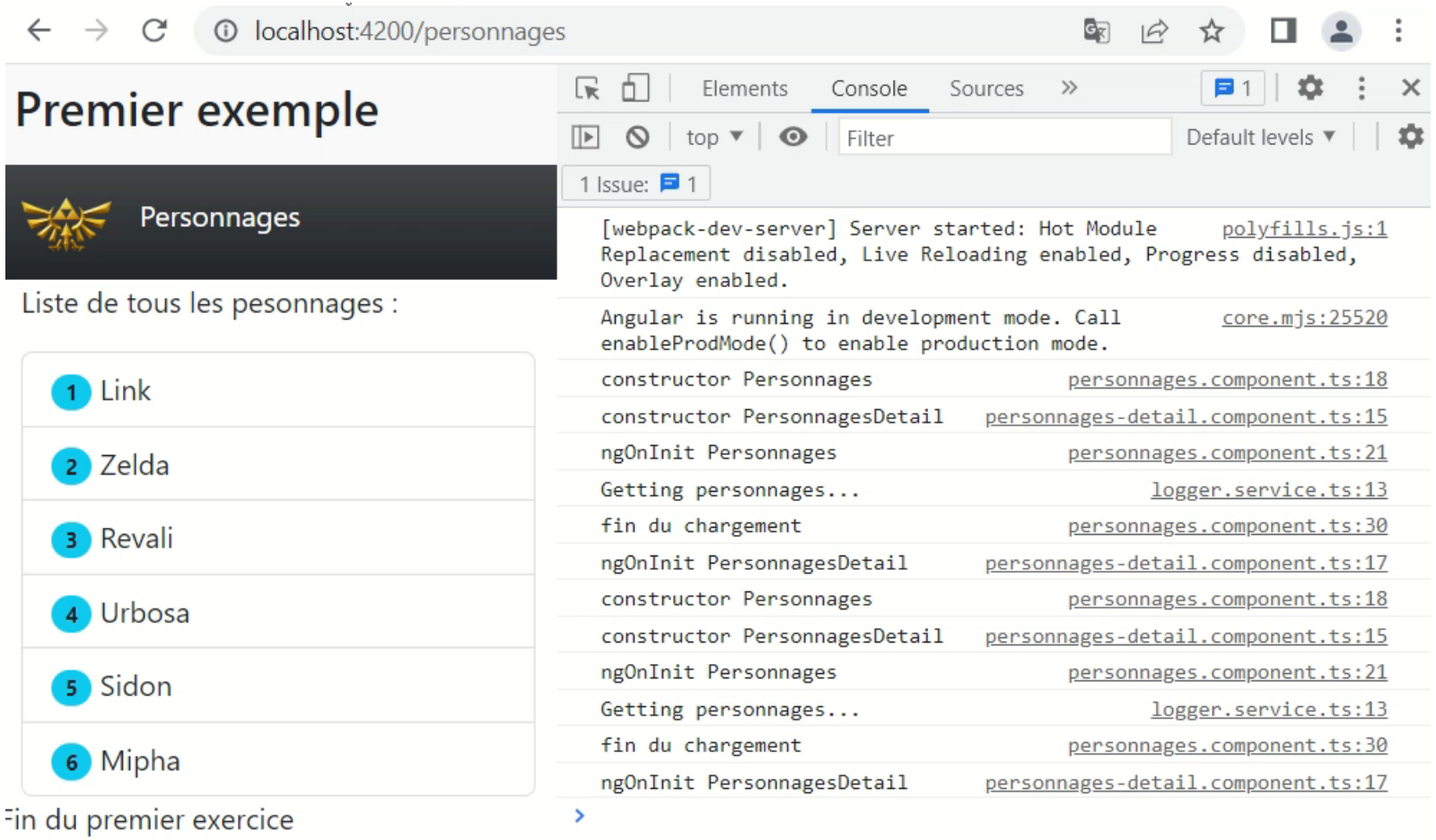
Exemple de résultat de navigation :

→ Création d'une barre de menu avec `routerLink` (/personnages)

Cours > jxc-cours > src > app > nav > nav.component.html > ...

```
<nav class="navbar navbar-expand-sm bg-dark navbar-dark bg-gradient bg-opacity-85 navbar-index sticky-top">
    <div class="container-fluid">
        <a class="navbar-brand" [routerLink]="['/home']" routerLinkActive="active">
            <img src="../assets/img/triforce.png" alt="Home" width="50">
        </a>
        <ul class="navbar-nav me-auto mb-2 mb-lg-0">
            <li class="nav-item">
                <a class="nav-link" aria-current="page"
                [routerLink]="['/personnages']" routerLinkActive="active">Personnages</a>
            </li>
        </ul>
    </div>
</nav>
```

ANGULAR : EXEMPLE



ANGULAR : EXEMPLE

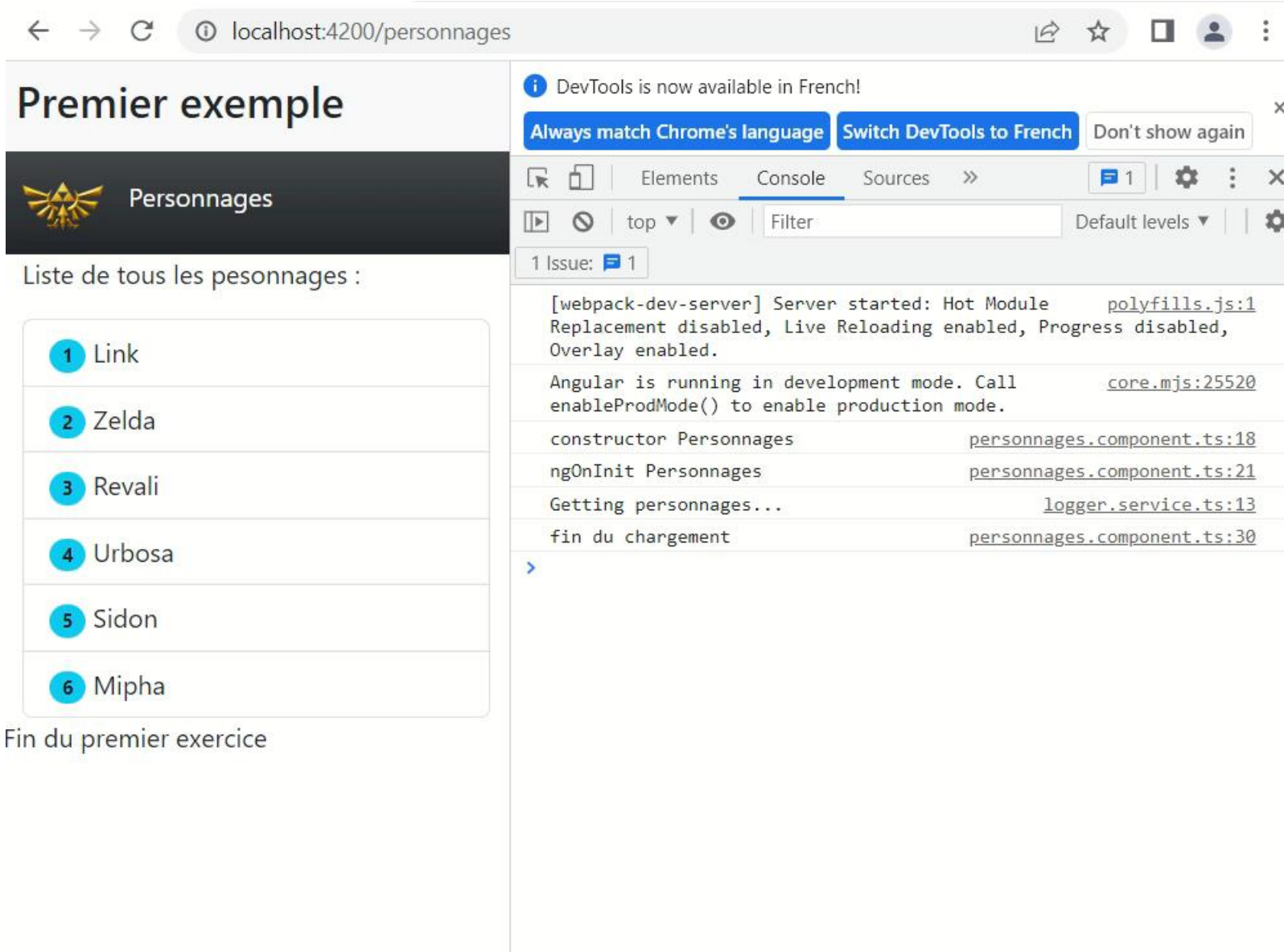
Exemple de résultat de navigation :

→ Lien dans le composant Personnage avec un `routerLink` et un id

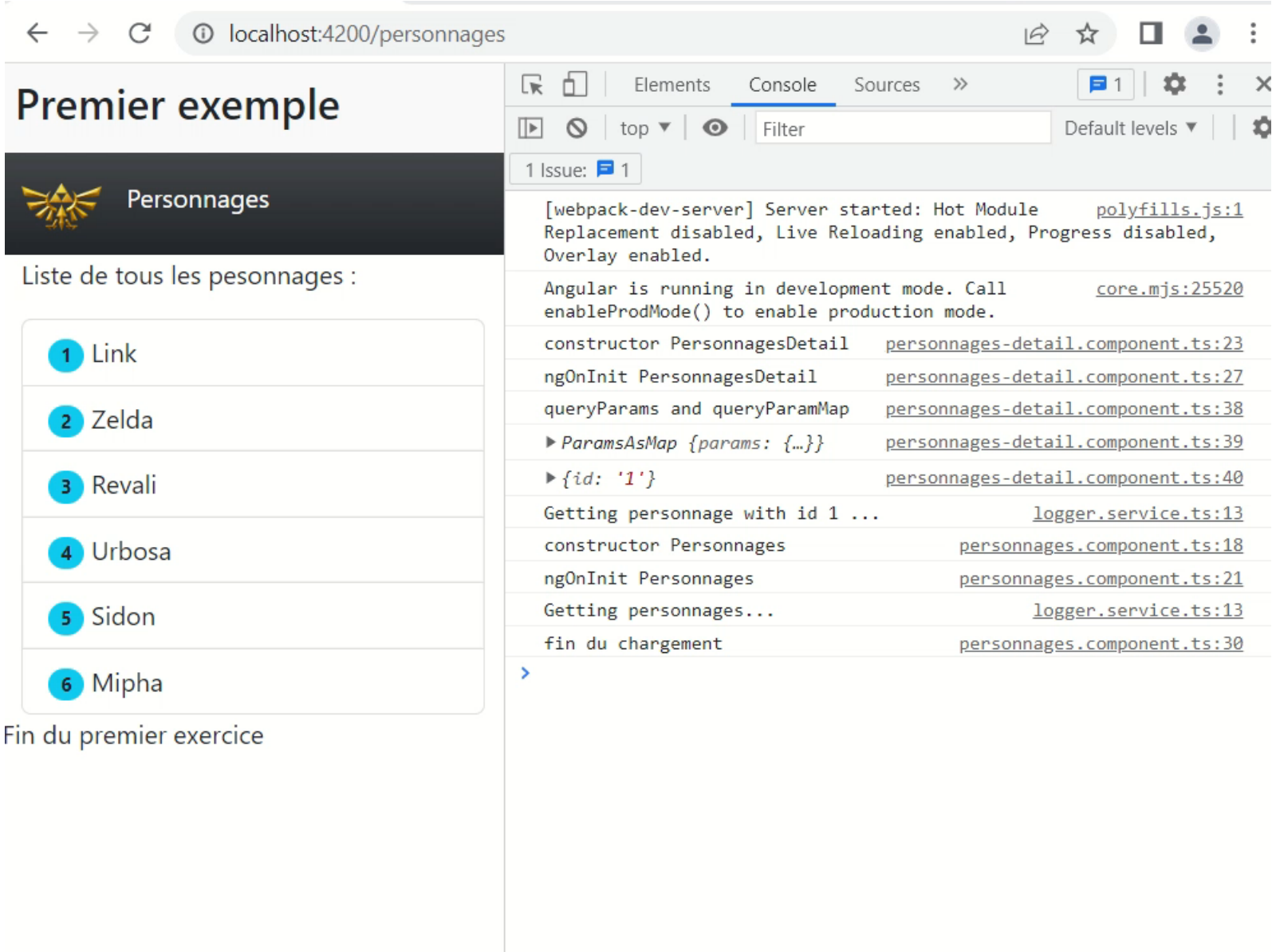
urs > jxc-cours > src > app > personnages > personnages.component.html > ...

```
<div class="container">
    <p>Liste de tous les pesonnages :</p>
    <ul class="list-group personnages">
        <li *ngFor="let hero of listPersonnages"
            class="list-group-item"
            [class.selected]="hero===myHero"
            (click)="selectedPersonnage(hero)"
            [routerLink]="['/hero', hero.id]" routerLinkActive="active">
            <span class="badge rounded-pill bg-info text-dark">{{hero.id}}</span>
            {{hero.name}}
            <span *ngFor="let arme of hero.arme"> {{arme}}</span>
    </ul>
</div>
<!-- <app-personnages-detail [myHero]="myHero" (newArmeEvent)="modifierArme($event)"></app-personnages-detail>
```

ANGULAR : EXEMPLE



ANGULAR : EXEMPLE



ANGULAR : ROUTING

Resolve

→ Lors de l’utilisation d’API, il peut y avoir un délai avant que les données qui doivent être affichées soient retournées du serveur (affichage d’une page blanche ou d’un message par défaut).

Utiliser **Resolve** permet :

→ d'attendre le retour d'un observable avant d'initialiser ou mettre à jour un composant après une mise à jour de l'url.

→ de passer un paramètre dynamique à une vue dans une route.

ANGULAR : ROUTING

ours > jxc-cours > src > app > personnages-detail > TS personnage.resolver.ts > ...

```ts
import { Injectable } from '@angular/core';
import { Resolve, RouterStateSnapshot, ActivatedRouteSnapshot } from '@angular/router';
import { Observable } from 'rxjs';
import { PersonnageService } from '../services/personnage.service';
import { IPersonnage } from '../personnages/IPersonnage';

@Injectable({
  providedIn: 'root'
})
export class PersonnageResolver implements Resolve<IPersonnage> {

  constructor(private personnageService: PersonnageService) { }

  resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable<IPersonnage> {
    console.log("resolve ...");
    const id = route.params['name'];
    return this.personnageService.getPersonnageById(+id);
  }
}
```

ANGULAR : ROUTING

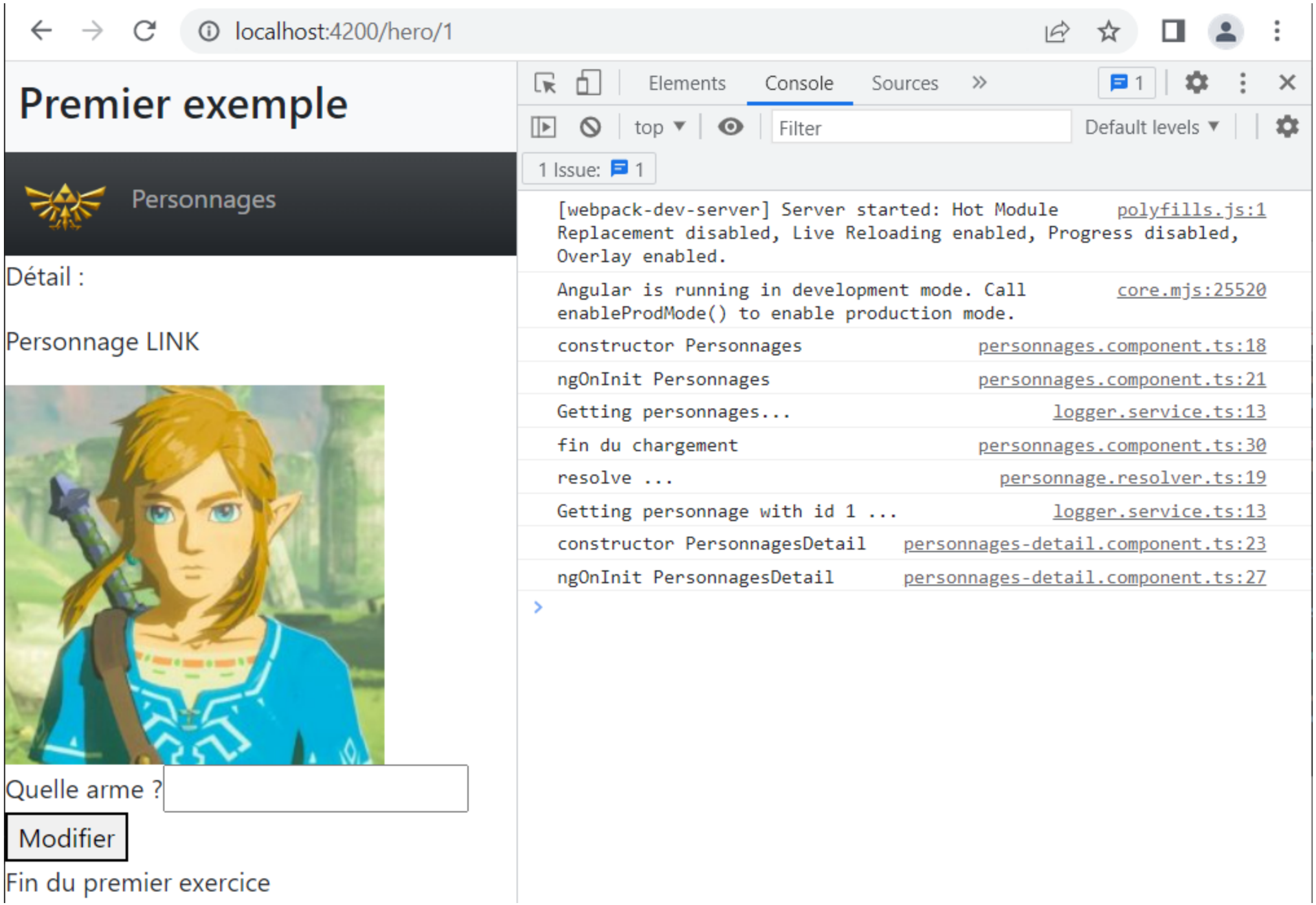
Resolve

Intégration dans la route :

```
const routes: Routes = [
  {path: 'hero/:name',
    component: PersonnagesDetailComponent,
    resolve: {hero: PersonnageResolver}
  },
```

Récupération des données :

```
ngOnInit(): void {
  console.log("ngOnInit PersonnagesDetail");
  this.route.data.subscribe(
    data => { this.myHero = data['hero']}
  );
  if(!this.myHero) this.router.navigate(['/personnages']);
}
```


localhost:4200/hero/1
Premier exemple
Personnages
Détail :
Personnage LINK
Quelle arme ?
Modifier
Fin du premier exercice
Elements
Console
Sources
top
Filter
Default levels
1 Issue: 1
[webpack-dev-server] Server started: Hot Module Replacement disabled, Live Reloading enabled, Progress disabled, Overlay enabled.
polyfills.js:1
Angular is running in development mode. Call enableProdMode() to enable production mode.
core.mjs:25520
constructor Personnages
personnages.component.ts:18
ngOnInit Personnages
personnages.component.ts:21
Getting personnages...
logger.service.ts:13
fin du chargement
personnages.component.ts:30
resolve ...
personnage.resolver.ts:19
Getting personnage with id 1 ...
logger.service.ts:13
constructor PersonnagesDetail
personnages-detail.component.ts:23
ngOnInit PersonnagesDetail
personnages-detail.component.ts:27

FRAMEWORK ANGULAR

Utilisation de Framework

Notion de composant web

Concept d'Angular

- Component / Template
- Data Binding
- Component Interaction
- Service
- Observable
- Rooting

Forms

ANGULAR : FORMULAIRE

→ Outils graphiques créé avec le langage **HTML**.

→ Permet à l'utilisateur d'interagir avec les données de l'application (se connecter, entrer des informations dans la base de données, mettre à jour un profil, etc.).

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

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

ANGULAR : FORMULAIRE

Angular propose deux approches :

→ **Template-driven forms**

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

→ **Reactive forms**

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

ANGULAR : FORMULAIRE

Différences principales entres les deux approches :

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

https://angular.io

ANGULAR : FORMULAIRE

Les **exemples suivants** seront basés sur une saisie de produits.

→ Interface **IProduit** :

exempleForm > src > app > TS IProduit.ts > ...

```ts
export interface IProduit{
    id: number,
    name: string,
    category: string
}
```


→ Composant **Products** :

exempleForm > src > app > products > <> products.component.html > ...

```html
<h2>Liste des produit :</h2>
<ul>
    <li *ngFor="let produit of produits">
        {{ produit.name }} {{ produit.category }}
    </li>
</ul>
```

exempleForm > src > app > products > TS products.component.ts > ...

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

@Component({
  selector: 'app-products',
  templateUrl: './products.component.html',
  styleUrls: ['./products.component.css']
})
export class ProductsComponent implements OnInit {

  produits: IProduit[] = [];
```

ANGULAR : TEMPLATE-DRIVEN FORMS

Première approche : Template-driven forms

→ Utilise la directive **`ngModel`** pour réaliser une **liaison bidirectionnelle**.

→ Cette directive permet de créer et manager une instance de **`FromControl`** pour un élément donné du formulaire.

→ Besoin d'importer **`FormsModule`** dans `app.module.ts` afin d'utiliser `ngModel` :

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

ANGULAR : TEMPLATE-DRIVEN FORMS

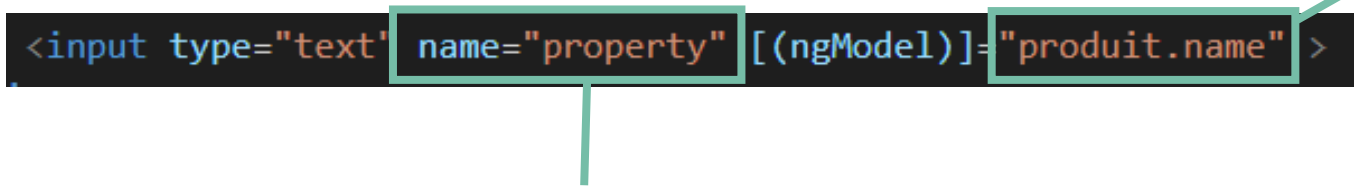
Création d'un composant :

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

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

@Component({
  selector: 'app-form-td',
  templateUrl: './form-td.component.html',
  styleUrls: ['./form-td.component.css']
})
export class FormTDComponent implements OnInit {

  categories = ["Légumes", "Fruits", "Viandes"];
  produit!: IProduit;

  constructor() { }

  ngOnInit(): void {
    this.produit = {
      id: 0,
      name: 'Pas de nom',
      category: this.categories[0]
    };
  }
}
```

ANGULAR : TEMPLATE-DRIVEN FORMS

Syntaxe ngModel :



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

→ Modification directe de la propriété `name` avec la nouvelle valeur saisie.

ANGULAR : TEMPLATE-DRIVEN FORMS

Exemple :

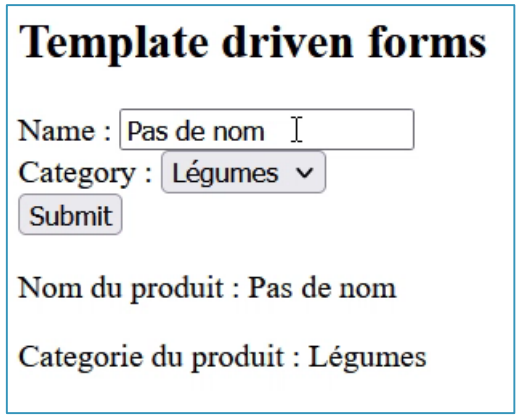
Un premier formulaire avec :

→ un champ texte pour saisir le nom du produit

→ un champ select pour sélectionner la catégorie du produit

→ un bouton submit

Résultat :



ANGULAR : TEMPLATE-DRIVEN FORMS

Directive `ngForm` attatchée à la balise `form`

(utilisation *template reference variable*)

exempleForm > src > app > form-td > form-td.component.html > ...

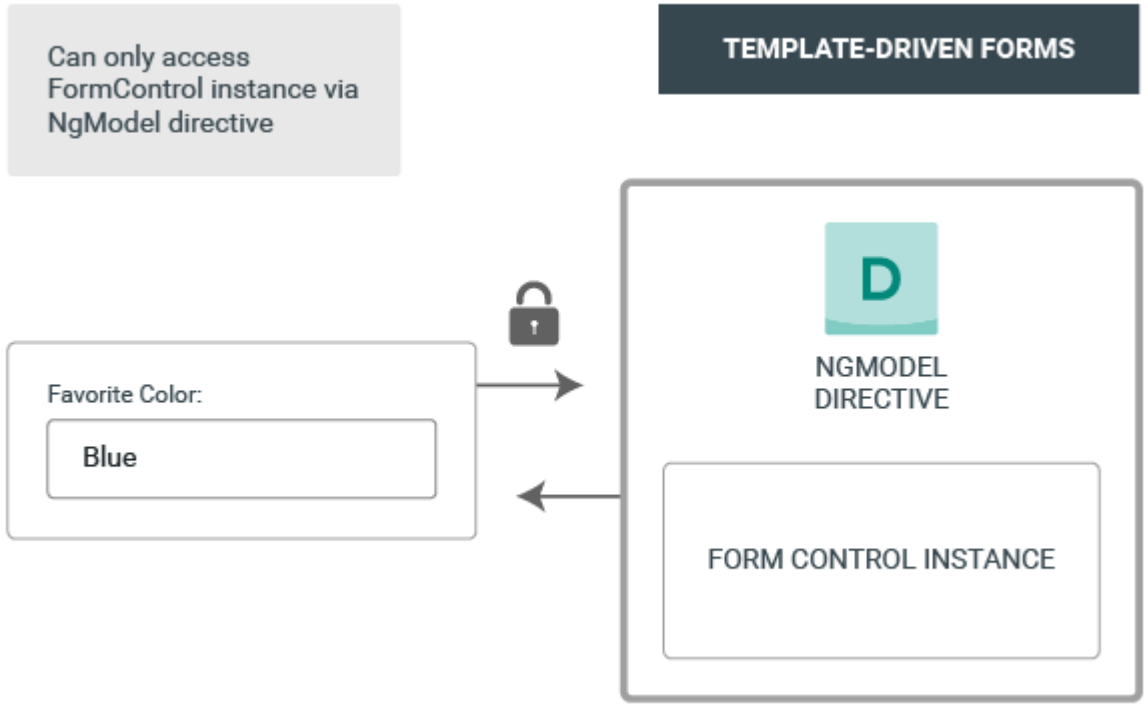
```
1   <h2>Template driven forms</h2>
2   <form #productFormTD="ngForm">
3       <div class="group">
4           <label for="name">Name : </label>
5           <input type="text" name="name" id="name" [(ngModel)]="produit.name" >
6       </div>
7       <div class="group">
8           <label for="category">Category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

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

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

ANGULAR : TEMPLATE-DRIVEN FORMS

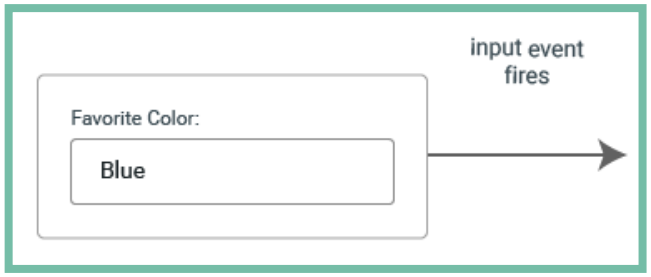
→ Avec ce type d'approche nous n'avons **pas un accès direct** à l'instance du **`FromControl`**.

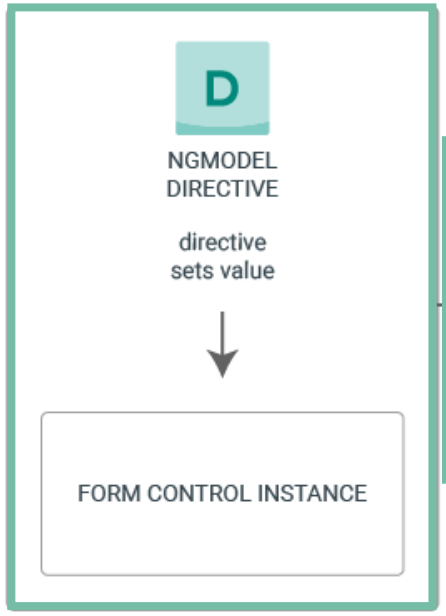


https://angular.io

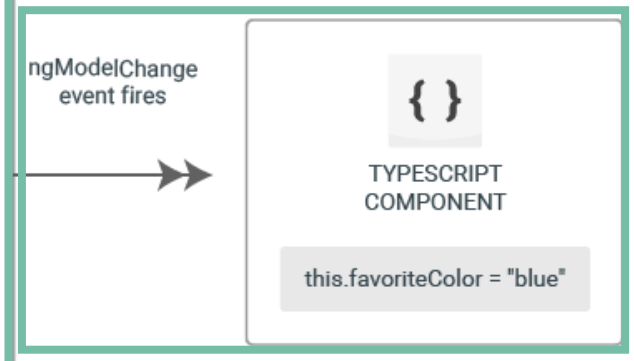
ANGULAR : TEMPLATE-DRIVEN FORMS

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



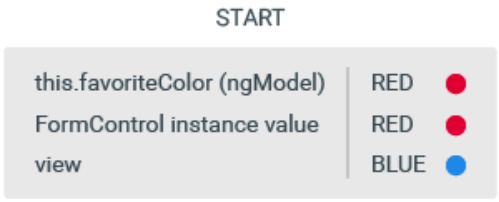


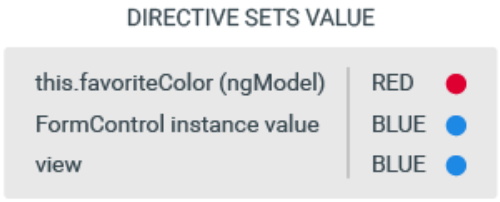
3. `ngModel.viewToModelUpdate()` est appelée et émet un `ngModelChange Event`.

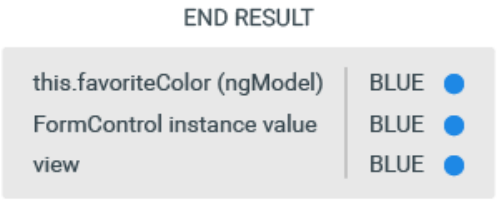


Emission de la nouvelle valeur via l'observable valueChanges

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



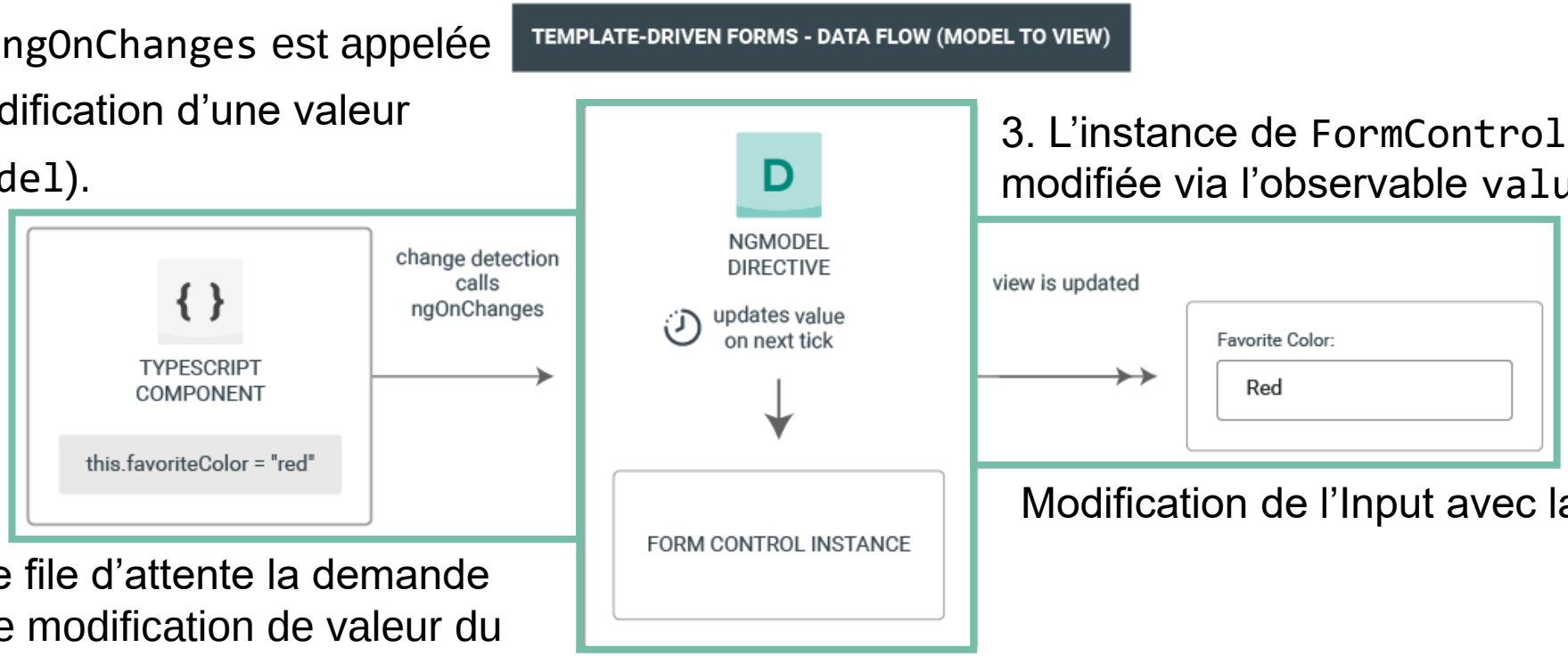




https://angular.io

ANGULAR : TEMPLATE-DRIVEN FORMS

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

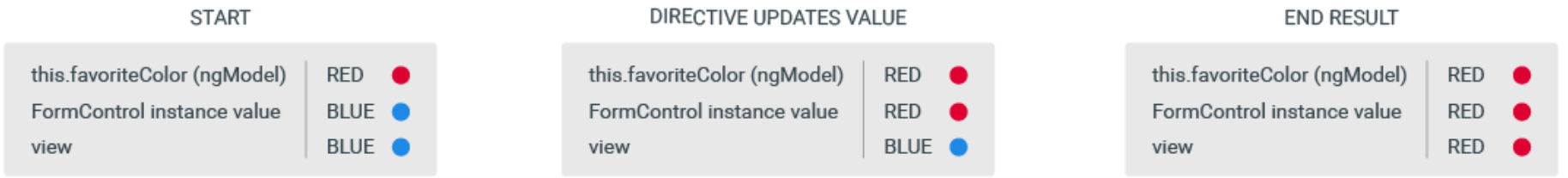


3. L'instance de `FormControl` émet la valeur modifiée via l'observable `valueChanges`.

Modification de l'Input avec la nouvelle valeur.

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

2. Changement de valeur



ANGULAR : TEMPLATE-DRIVEN FORMS

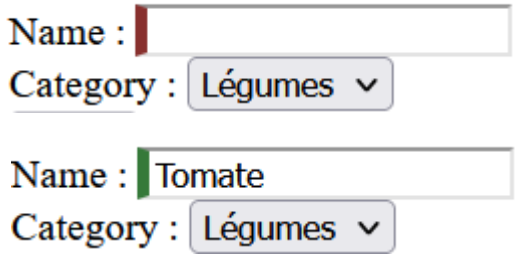
→ L'utilisation de la directive `ngModel` permet donc de suivre l'état d'un input à un moment donné.

→ Angular propose différentes classes CSS pour gérer les différents états :

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

ANGULAR : TEMPLATE-DRIVEN FORMS

Utilisation du couple `ng-valid` / `ng-invalid` :



→ CSS :

exempleForm > src > app > form-td > # form-td.component.css >

```css
.ng-valid[required] {
    border-left: 5px solid #357939;
}

.ng-invalid:not(form)  {
    border-left: 5px solid #88302e;
}
```

→ Modification du formulaire :

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

```html
<h2>Template driven forms</h2>
<form #productFormTD="ngForm">
    <div class="group">
        <label for="name">Name : </label>
        <input type="text" name="name" id="name" [(ngModel)]="produit.name" required>
    </div>
```

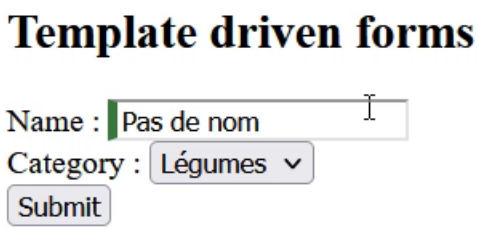
ANGULAR : TEMPLATE-DRIVEN FORMS

Soumission du formulaire

→ La directive `ngForm` possède une notion de validité du formulaire.

→ Il est par exemple possible de désactiver le bouton *submit* tant que le formulaire n'est pas valide :

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



→ Possibilité de créer ses propres validateurs.

ANGULAR : TEMPLATE-DRIVEN FORMS

Soumission du formulaire

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


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

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

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

ANGULAR : REACTIVE FORM

Deuxième approche : Reactive form

→ A l'inverse de la première approche, nous allons pouvoir manipuler les contrôles de chaque entité du formulaire.

→ Définition des contrôles directement dans le composant :
La directive `[formControl]` va directement lier l'instance de `FormControl` à un élément de la vue.

→ Besoin d'importer le module : **`ReactiveFormsModule`**

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

ANGULAR : REACTIVE FORM

Instance **`FormControl`** :

exempleForm > src > app > form-r > TS form-r.component.ts > ...

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

@Component({
  selector: 'app-form-r',
  templateUrl: './form-r.component.html',
  styleUrls: ['./form-r.component.css']
})
export class FormRComponent implements OnInit {

  name: string = 'defaut'
  nameControl = new FormControl('defaut');

  constructor() { }

  ngOnInit(): void {
  }
}
```

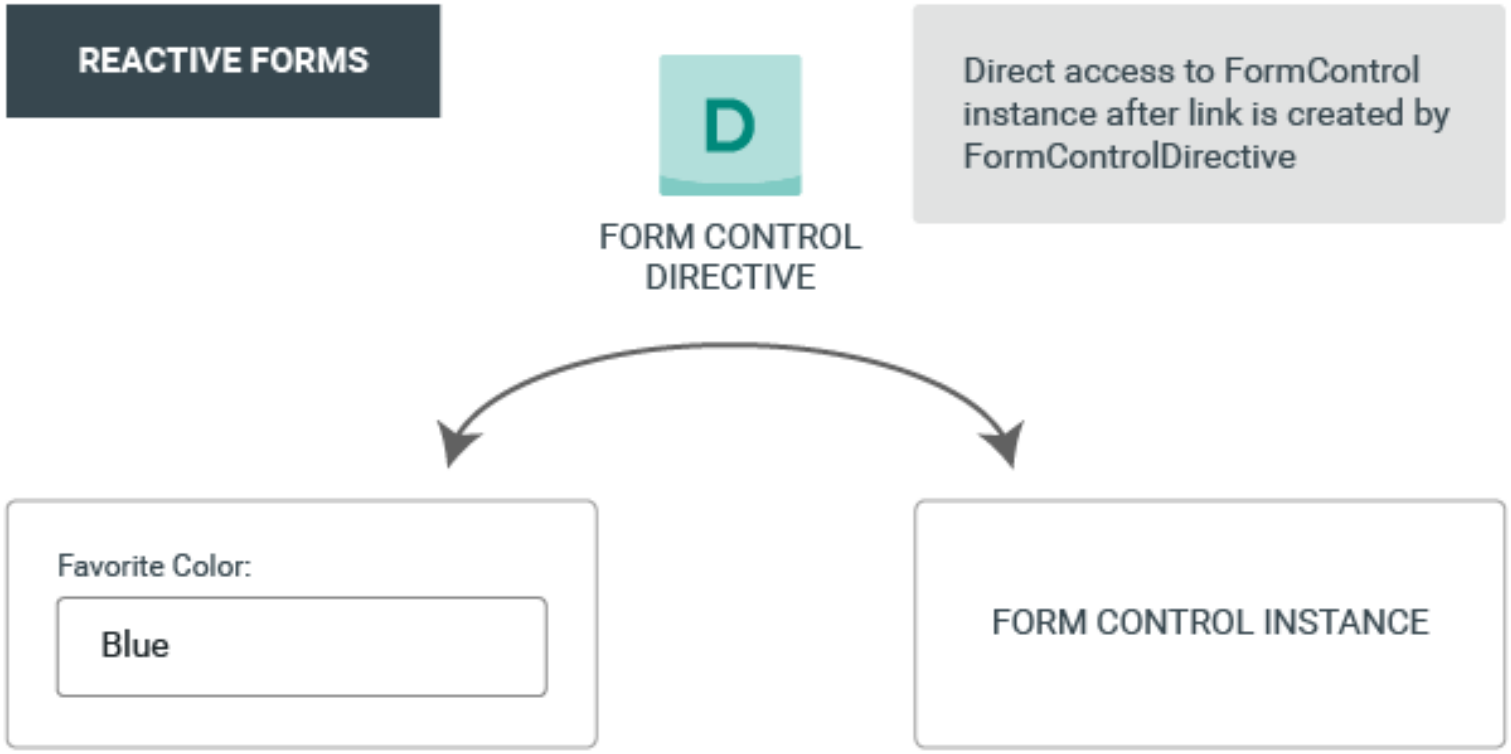
ANGULAR : REACTIVE FORM

Syntaxe de la directive **`formControl`** :

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

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

ANGULAR : REACTIVE FORM



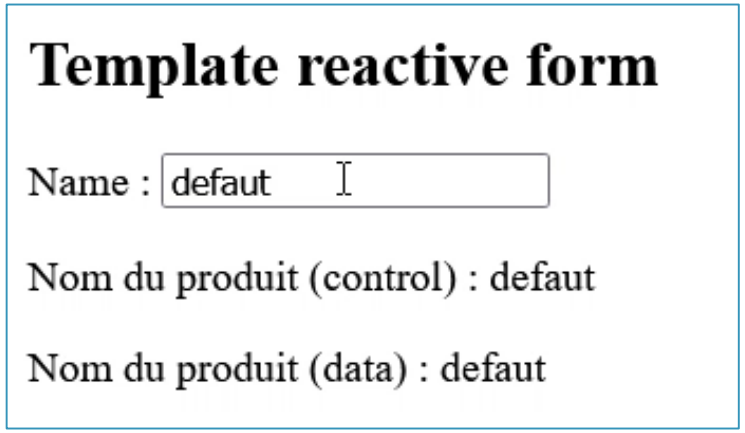
https://angular.io

ANGULAR : REACTIVE FORM

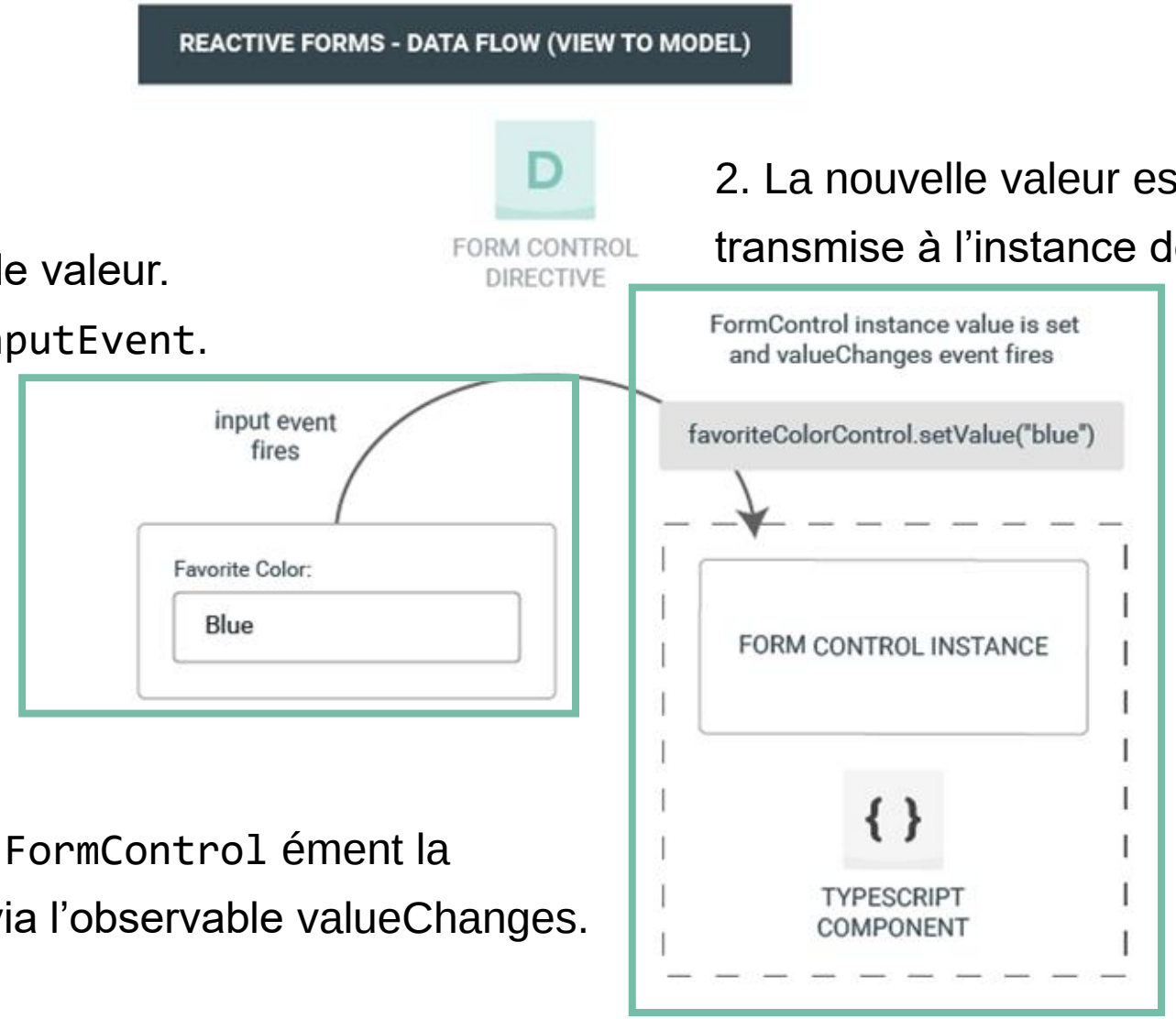
→ L'instance de `FormControl` à accès à la valeur courante de l'input associé.

→ **Aucune modification** de la donnée du composant est réalisé directement.

→ Les maj de la vue au modèle et inversement du modèle à la vue sont synchrones.



ANGULAR : REACTIVE FORM



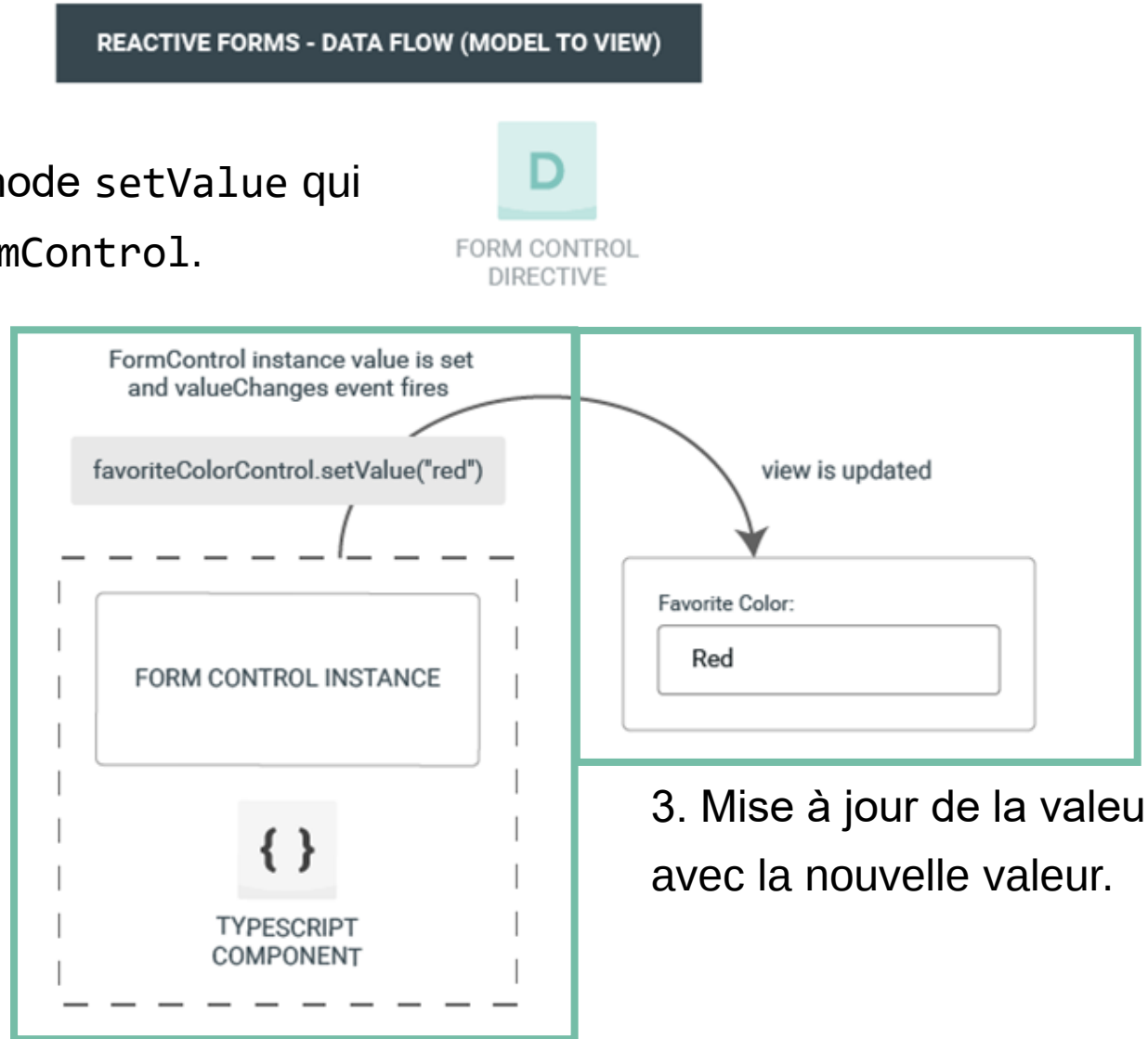
1. L'utilisateur saisit la nouvelle valeur.
   L'élément Input émet un `InputEvent`.

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

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

https://angular.io

ANGULAR : REACTIVE FORM



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

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

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

3. Mise à jour de la valeur de l'Input avec la nouvelle valeur.

https://angular.io

ANGULAR : REACTIVE FORM

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

exempleForm > src > app > form-r > TS form-r.component.ts > ...

```
export class FormRComponent implements OnInit {

  name: string = 'defaut'
  nameControl = new FormControl('defaut');

  constructor() { }

  ngOnInit(): void {
    this.nameControl.valueChanges.subscribe( res =>
      this.name = res
    )
  }
}
```


Template reactive form

Name : defaut

Nom du produit (control) : defaut

Nom du produit (data) : defaut

ANGULAR : REACTIVE FORM

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

exempleForm > src > app > form-r > TS form-r.component.ts > FormRComponent >

```
 9   export class FormRComponent implements OnInit {
10
11     name: string = 'defaut';
12     category: string = '';
13     categories = ["", "Légumes", "Fruits", "Viandes"];
14
15     //nameControl = new FormControl('defaut');
16     produitForm = new FormGroup({
17       nameControl: new FormControl(''),
18       categoryControl: new FormControl(this.categories[0]),
19     });
```

→ Peut également imbriquer plusieurs `FromGroup`.

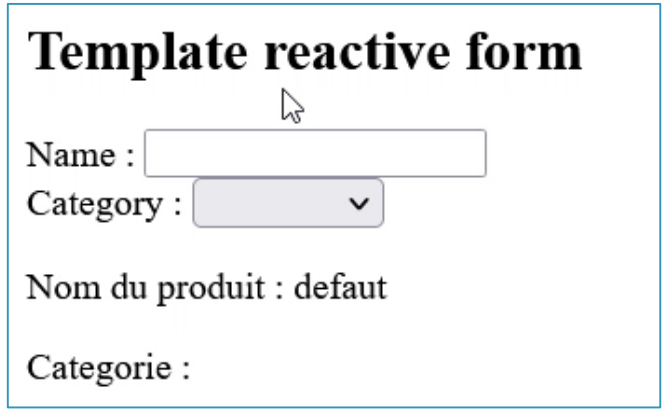
ANGULAR : REACTIVE FORM

formGroupName=…
Si imbrication d'un autre fromGroup

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

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

ANGULAR : REACTIVE FORM



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

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

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

ANGULAR : REACTIVE FORM

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

→ Utilisation de `setValue()` ou de `patchValue()`


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

ANGULAR : REACTIVE FORM

Soumission du formulaire

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

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


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

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

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

ANGULAR : REACTIVE FORM

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

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


exempleForm > src > app > form-r > TS form-r.component.ts > ...

```
11  export class FormRComponent implements OnInit {
12
13    produitFormFB = this.fb.group({
14      nameControl: [''],
15      categoryControl: ['']
16    })
17
18    constructor(private fb: FormBuilder) { }
```

ANGULAR : REACTIVE FORM

Fonctions de validation

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

- **Validateurs synchrones** (retourne directement les erreurs ou null, 2ème paramètre)
- **Validateurs asynchrones** (retourne un observable qui émet plus tard les erreurs ou null, 3ème paramètre)

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

ANGULAR : REACTIVE FORM

Fonctions de validation

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

required, maxLenght, minLenght, pattern, etc.


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

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

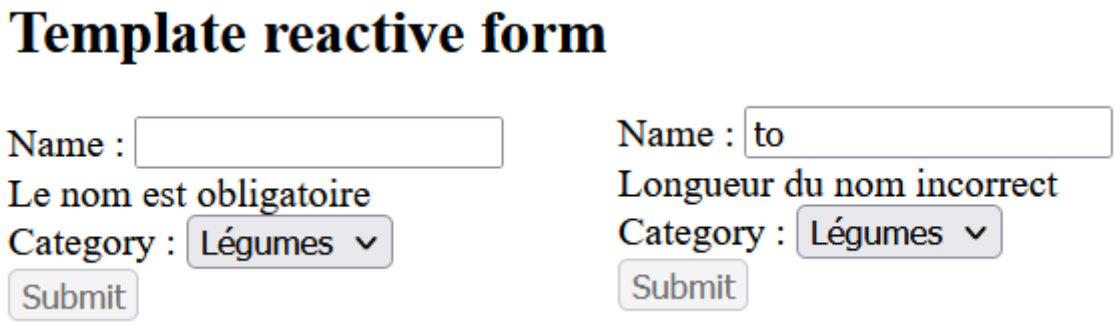
ANGULAR : REACTIVE FORM

Fonctions de validation

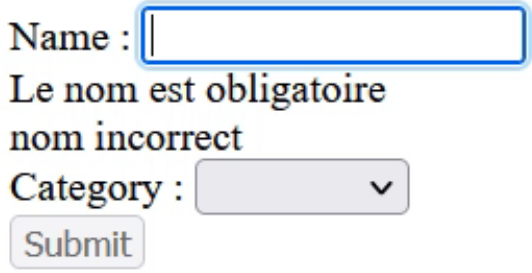
Affichage du message d'erreur dans le template :

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

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



ANGULAR : REACTIVE FORM



Fonctions de validation

→ Possibilité de créer ses propres fonctions de validation.

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

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

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

FRAMEWORK ANGULAR

Utilisation de Framework

Notion de composant web

Concept d'Angular

- Component / Template
- Data Binding
- Component Interaction
- Service
- Rooting
- Rooting
- Forms
- **HTTP**

ANGULAR : HTTP

→ Angular propose d'utiliser des services « bas niveau » afin de procéder à un échange de donnée avec un service web côté back.

→ Utilisation du **module HTTP** (**HttpModule** et depuis la V5 **HttpClientModule**) facilitant la réalisation de requêtes http via les classes suivantes :
HttpClient, HttpHeaders, HttpInterceptor, HttpRequest, etc.

→ Permet d'invoquer des services web via les différentes méthodes HTTP :
GET, POST, PUT, DELETE

ANGULAR : HTTP – JSON SERVER

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

Utilisation d'un serveur JSON

→ **json-server** permet d'imiter une API et de fournir un accès dynamique aux données.

→ Possibilité de lire, ajouter, mettre à jour et supprimer des données.
(GET, POST, PUT, DELETE).

→ Open-source

→ Utilise le port 3000 par défaut

ANGULAR : HTTP – JSON SERVER

Json-serveur

→ Comment l'installer ?

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

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


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

ANGULAR : HTTP – JSON SERVER

Json-serveur

→ Lancement du serveur

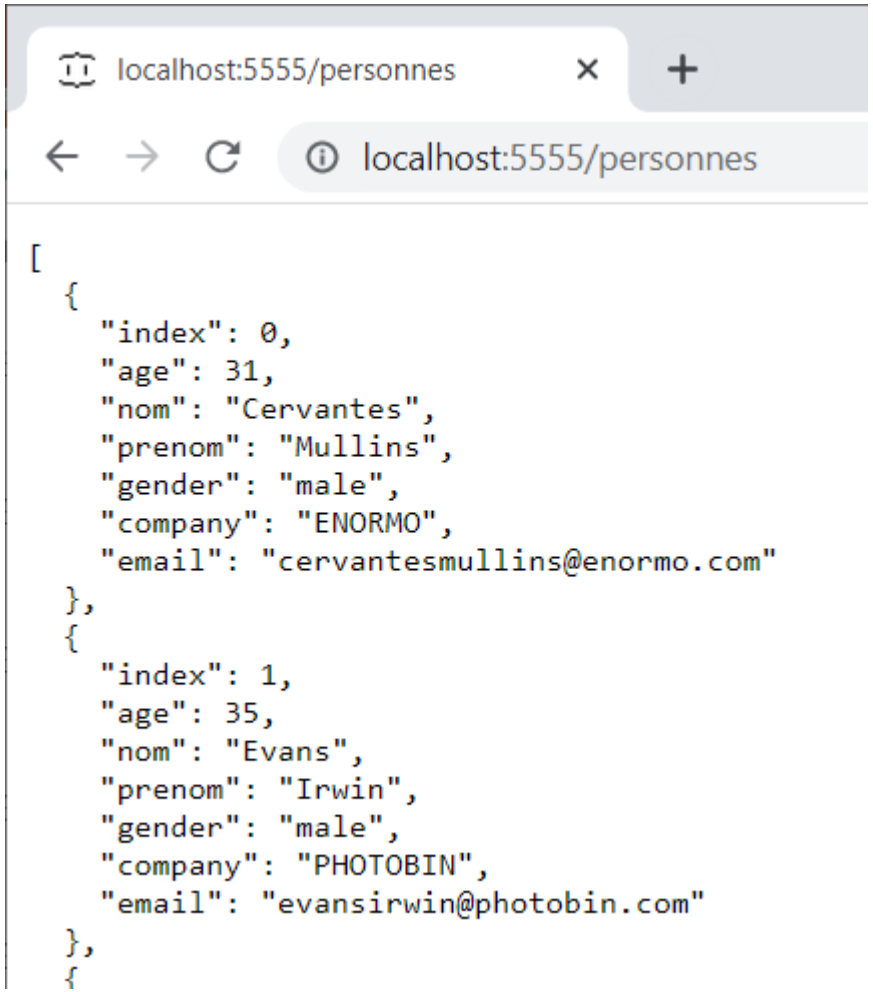
```
json-server db.json
```

URL utilisée par le client pour réaliser des requêtes HTTP

ANGULAR : HTTP – JSON SERVER

Json-serveur

→ Lancement du serveur



ANGULAR : HTTP – JSON SERVER

→ **Différentes requêtes HTTP possible** :

- Pour récupérer la liste de toutes les personnes :

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

- Pour récupérer une personne selon un identifiant :

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

- Pour ajouter une nouvelle personne :

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

- Pour modifier les valeurs d'une personnes :

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

- Pour supprimer une personne :

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

ANGULAR : HTTP

→ **Comment utiliser le module HTTP** ?

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

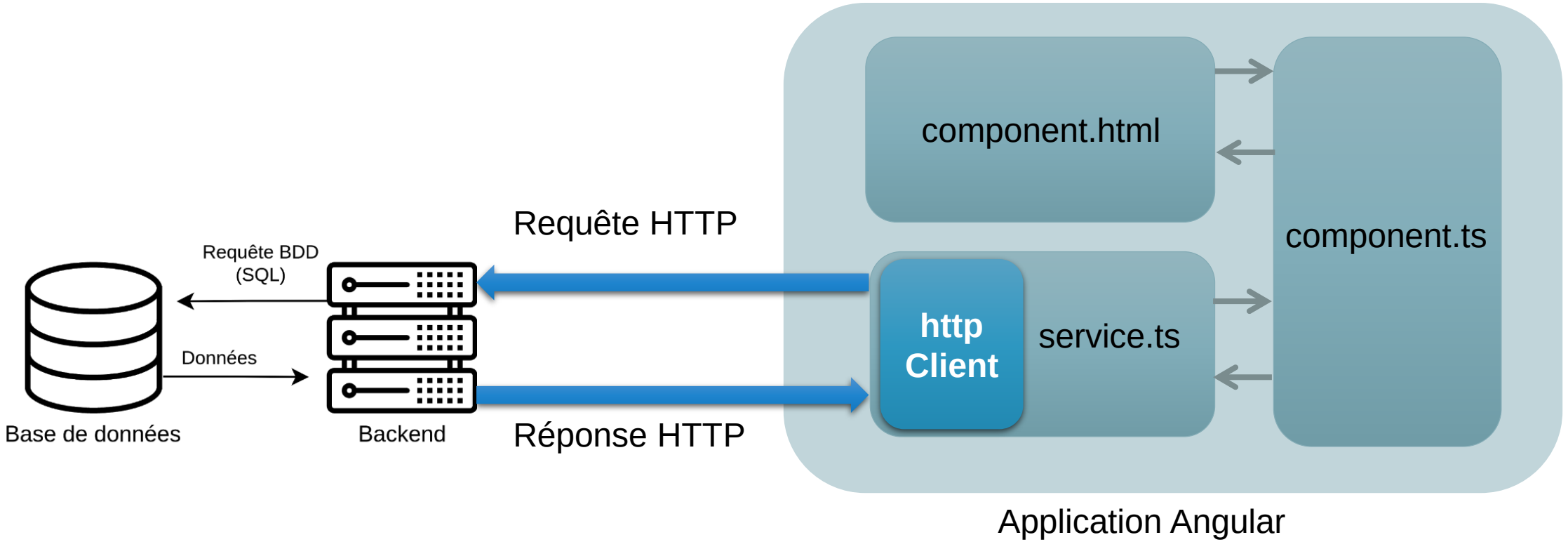
```
import { HttpClientModule } from '@angular/common/http';
```

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

ANGULAR : HTTP

→ **Comment utiliser le module HTTP** ?

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



ANGULAR : HTTP

→ **Comment utiliser le module HTTP** ?

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

ANGULAR : HTTP

→ **Comment utiliser le module HTTP** ?

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

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

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

URL de base pour les requêtes HTTP

Injection du service HTTP dans le service Personnage

ANGULAR : HTTP

→ **Comment utiliser le module HTTP** ?

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


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

Fichier personne.component.ts

ANGULAR : HTTP

1) Récupération de toutes les données Personne :

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

exempleHttp > src > app > TS IData.ts > ...

```ts
export interface IData {
    index: number;
    age: number;
    nom: string;
    prenom: string;
    gender: string;
    company: string;
    email: string;
}
```

ANGULAR : HTTP

1) Récupération de toutes les données Personne :

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

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

```
10  export class PersonneComponent implements OnInit {
11
12      personnes: IData[] = [];
```

exempleHttp > src > app > personne > <> personne.component.html >

```
1   <h2>Liste des personnes :</h2>
2   <ul>
3       <li *ngFor="let perso of personnes">
4           {{ perso.prenom }} {{ perso.nom }}
5       </li>
6   </ul>
```

ANGULAR : HTTP

1) Récupération de toutes les données Personne :

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

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

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

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

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

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

Endpoint URL

ANGULAR : HTTP

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

1) Récupération de toutes les données Personne :

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

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

Res est de type IData[]

```
(res:IData[])
```

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

ANGULAR : HTTP

1) Récupération de toutes les données Personne :

Résultat obtenu :

ANGULAR : HTTP

1) Récupération de toutes les données Personne :

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

→ Nouvelle **interface** :

exempleHttp > src > app > TS IDataLight.ts > ...

```ts
export interface IDataLight {
    index: number;
    age: number;
    lastname: string;   //dans IData : nom
    firstname: string;  //dans IData : prenom
}
```


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

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

```ts
export class PersonneComponent implements OnInit {

  personnes: IData[] = [];
  personnesBis: IDataLight[] = [];
```

exempleHttp > src > app > personne > personne.component.html > ...

```html
<h2>Liste des personnes IDATALIGHT:</h2>
<ul>
    <li *ngFor="let perso of personnesBis">
        {{ perso.firstname }} {{ perso.lastname }}
    </li>
</ul>
```

ANGULAR : HTTP

1) Récupération de toutes les données Personne :

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

→ **Problème :**

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

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

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

ANGULAR : HTTP

1) Récupération de toutes les données Personne :

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

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

Nous voulons la totalité de la réponse à la requête http


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

Par défaut : body et json

ANGULAR : HTTP

Headers

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

1) Récupération de toutes les données Personne :

Réponse entière ?

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

```
35    getAllResponse(): Observable<HttpResponse<IData[]>>{
36      return this.http.get<IData[]>(this.url, {observe: 'response'});
37    }
```

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

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

ANGULAR : HTTP

1) Récupération de toutes les données Personne :

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

304

→ Le serveur peut rejeter la requête HTTP (code de retour 404 ou 500 par exemple) : *error response*

→ Ou alors problème côté client (problème de réseau, exception déclenchée par un operateur RxJS, etc.). Les erreurs ont pour statut 0.

ANGULAR : HTTP

Exemple de gestion d'erreur :

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

```
18    getAll(): Observable<IData[]>{
19      return this.http.get<IData[]>(this.url).pipe(
20        catchError(this.handleError)
21      );
22    }
```

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

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

ANGULAR : HTTP

1) Récupération de toutes les données Personne :

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

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


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

ANGULAR : HTTP

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

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

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

Fonctionne de manière similaire à `get()`

Possibilité de spécifier un header

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

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

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

```
61    addPersonne(personne: IData): Observable<IData> {
62      return this.http.post<IData>(this.url, personne);
63    }
```

ANGULAR : HTTP

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

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

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

```
55    addPersonne() {
56      this.personneService.addPersonne(this.perso).subscribe( res => {
57          console.log(res);
58          this.personnes.push(res);
59      })
60    }
```

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

ANGULAR : HTTP

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

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

Résultat :

Liste des personnes IDATA:

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


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

ANGULAR : HTTP

Remarque :

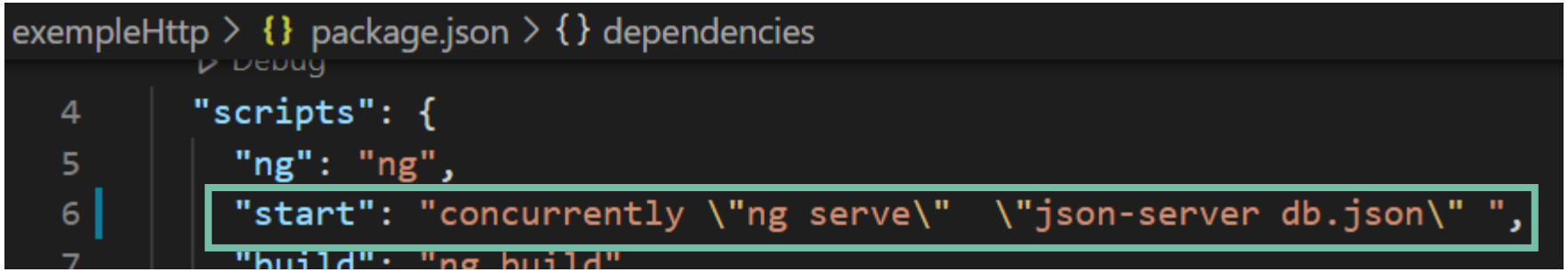
→ Possibilité d'utiliser le package **concurrently** pour ne plus à avoir démarrer séparément les deux serveurs Angular et json-server.

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


→ Package NodeJS permettant d'exécuter plusieurs commandes simultanément.

```
npm start
```

→ `concurrently "command1 arg" "command2 arg"`



ANGULAR : HTTP

→ Les ressources REST ont besoin d'un authentification et d'une autorisation.

→ JSON web tokens (JWT)
Généré par un web service et aide à la communication entre le client et le serveur.

→ Création d'un service pour enregistrer le **token** et le réutiliser.

→ Mise en place d'un **intercepteur** pour injecter des headers dans tous les requêtes d'authentification (et ne pas à avoir répéter du code)

→ Ajout d'une **garde** pour restreindre l'accès à certains composants aux personnes identifiés uniquement.

FRAMEWORK ANGULAR

Utilisation de Framework

Notion de composant web

Concept d'Angular

- Component / Template
- Data Binding
- Component Interaction
- Service
- Rooting
- Rooting
- Forms
- HTTP
- **Pipes**

ANGULAR : PIPES

→ Les données obtenues peuvent ne pas être affichées de la manière souhaitée dans la vue.
Par exemple : la date, la monnaie, l'ordre d'une liste, etc.

→ La solution avec Angular : les **pipes**.
Fonctions prenant en entrée une valeur et retournant la valeur transformée.

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

ANGULAR : PIPES

→ Exemple pour les dates :

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

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

Nous sommes le : Jan 7, 2022

Nous sommes le : Friday, January 7, 2022

Nous sommes le : 01/07/2022

→ Utilisation de la fonction `registerLocalData`

exempleForm > src > app > TS app.module.ts > AppModule

```
14  import { FormsModule } from '@angular/forms';
15  import { ReactiveFormsModule } from '@angular/forms';
16  import { registerLocaleData } from '@angular/common';
```

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

Nous sommes le : 7 janv. 2022

Nous sommes le : vendredi 7 janvier 2022

Nous sommes le : 01/07/2022

ANGULAR : PIPES

→ Possibilité de créer son propre pipe :

exempleForm > src > app > TS greeting.pipe.ts > ...

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

@Pipe({
  name: 'greeting'
})
export class GreetingPipe implements PipeTransform {

  transform(value: string, gender: string): string {
    if(gender === "M"){
      return `Bonjour monsieur ${value}`;
    } else if(gender === "F"){
      return `Bonjour madame ${value}`;
    } else {
      return `Bonjour ${value}`;
    }
  }

}
```

exempleForm > src > app > TS app.module.ts > ...

```ts
import { GreetingPipe } from './greeting.pipe';

@NgModule({
  declarations: [
    AppComponent,
    ProductsComponent,
    GreetingPipe
  ],
```

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

Bonjour monsieur John Doe

Bonjour John Doe

FRAMEWORK ANGULAR

Utilisation de Framework

Notion de composant web

Concept d'Angular

- Component / Template
- Data Binding
- Component Interaction
- Service
- Rooting
- Rooting
- Forms
- HTTP
- Pipes
- **Directives**

ANGULAR : DIRECTIVES

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

→ **Directives structurelles** : modification de l'arborescence du DOM.
`ngIf, ngFor, NgSwitch, etc.`

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

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

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

ANGULAR : DIRECTIVES

Possibilité de créer ses propres directives.

Exemple pour un directive d'attribut : Color.

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

exempleForm > src > app > TS color.directive.ts > ...

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

@Directive({
  selector: '[appColor]'
})
export class ColorDirective {

  constructor() {  }

}
```

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

@NgModule({
  declarations: [
    AppComponent,
    FormTDComponent,
    FormRComponent,
    ProductsComponent,
    GreetingPipe,
    ColorDirective
  ],
```

A intégrer dans le app module

ANGULAR : DIRECTIVES

Possibilité de créer ses propres directives.

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

exempleForm > src > app > TS color.directive.ts > ...

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

@Directive({
  selector: '[appColor]'
})
export class ColorDirective {

  constructor(private el: ElementRef) {
      el.nativeElement.style.background = 'red';
  }

}
```

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

Hello world

ANGULAR : DIRECTIVES

Possibilité de créer ses propres directives.

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

Hello world

exempleForm > src > app > TS color.directive.ts > ...

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

@Directive({
  selector: '[appColor]'
})
export class ColorDirective {

  constructor(private el: ElementRef) {  }

  @HostListener('mouseenter') onMouseEnter(): void {
    this.changerCouleur('red');
  }
  @HostListener('mouseleave') onMouseLeave(): void {
    this.changerCouleur('white');
  }
  changerCouleur(couleur: string){
    this.el.nativeElement.style.background = couleur;
  }
}
```

ANGULAR : DIRECTIVES

Possibilité de créer ses propres directives.

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

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

Hello world

```
export class ColorDirective {

  @Input('appColor') couleur = '';

  constructor(private el: ElementRef) {  }

  @HostListener('mouseenter') onMouseEnter(): void {
    this.changerCouleur(this.couleur);
  }
  @HostListener('mouseleave') onMouseLeave(): void {
    this.changerCouleur('white');
  }
  changerCouleur(couleur: string){
    this.el.nativeElement.style.background = couleur;
  }
}
```

FRAMEWORK ANGULAR

Utilisation de Framework

Notion de composant web

Concept d'Angular

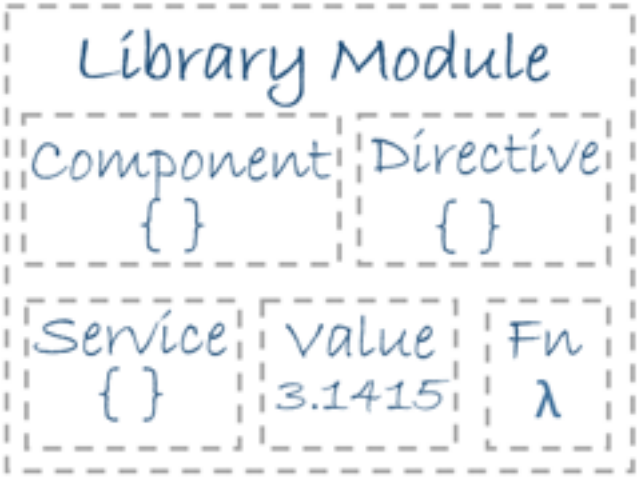
- Component / Template
- Data Binding
- Component Interaction
- Service
- Rooting
- Rooting Forms
- HTTP
- Pipes
- Directives
- **Module**

ANGULAR : MODULES

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

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

→ Possibilité de créer des sous modules dans une application.



ANGULAR : SOUS-MODULES

→ Nouveau sous module `Fruits` dans notre application.

→ Nouveau composant `Pomme`

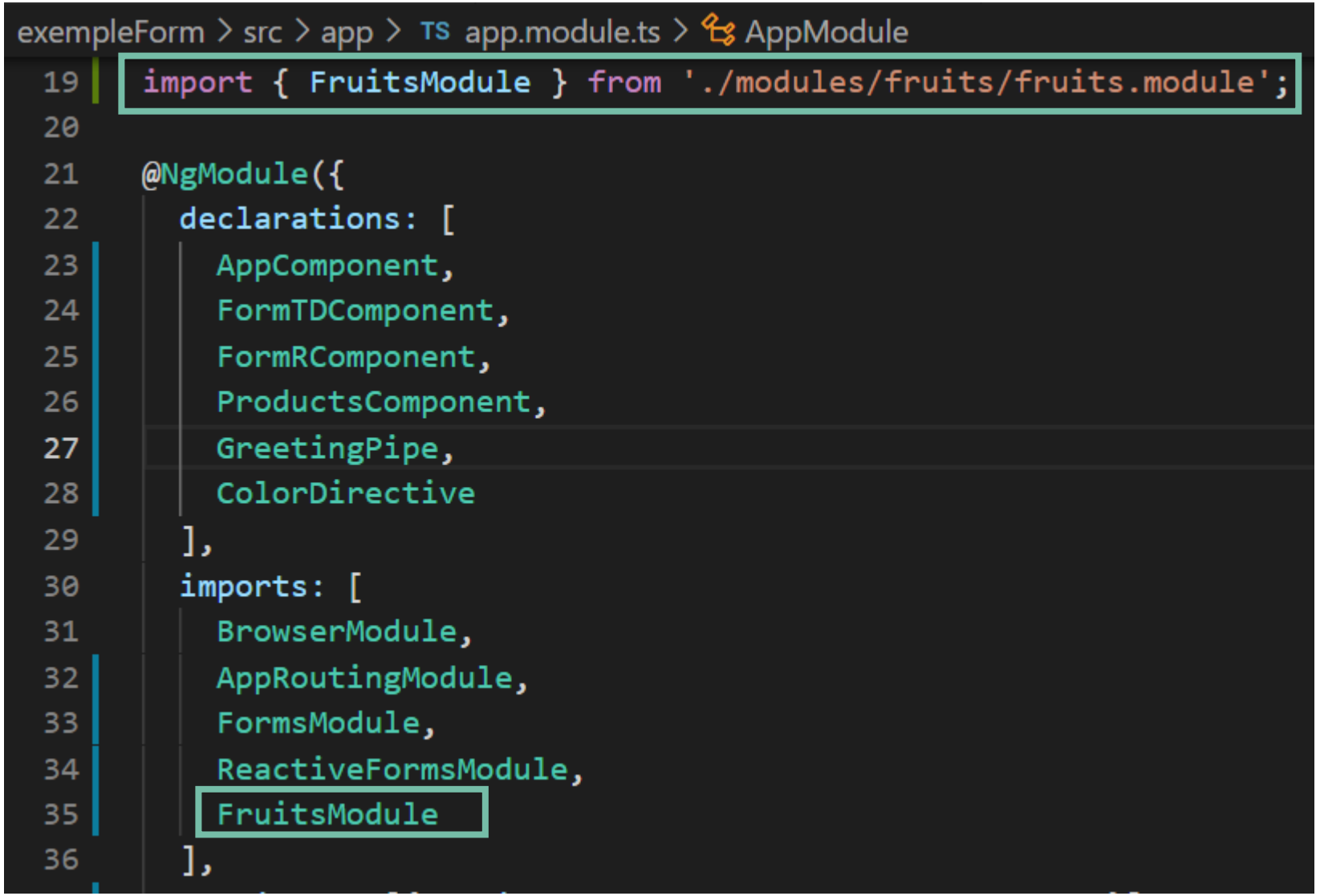
Module contenant les pipes et les directives

exempleForm > src > app > modules > fruits > TS fruits.module.ts > ...

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

import { FruitsRoutingModule } from './fruits-routing.module';
import { PommeComponent } from './pomme/pomme.component';

@NgModule({
  declarations: [
    PommeComponent
  ],
  imports: [
    CommonModule,
    FruitsRoutingModule
  ]
})
export class FruitsModule { }
```

ANGULAR : SOUS-MODULES

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

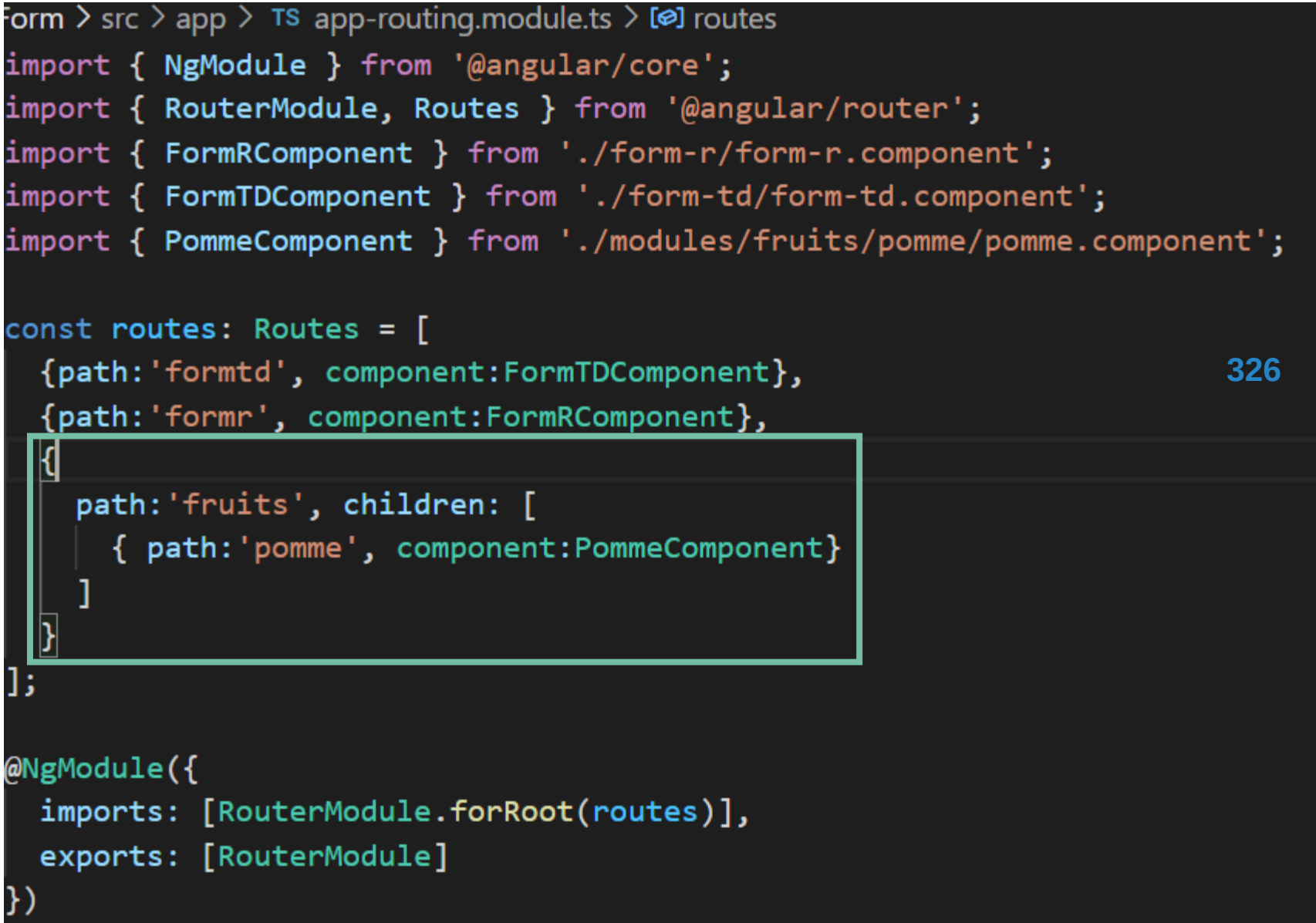


ANGULAR : SOUS-MODULES

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

→ **Eager loading** vs lazy loading

→ Dans le app routing module



ANGULAR : SOUS-MODULES

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

→ **Eager loading** vs lazy loading

→ Ou dans le rooting module du nouveau module.

exempleForm > src > app > modules > fruits > TS fruits-routing.module.ts > ...

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

ANGULAR : SOUS-MODULES

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

→ Eager loading vs **lazy loading**

→ Utilisation de **`loadChildren`** et des promesses.

exempleForm > src > app > TS app-routing.module.ts > AppRoutingModule

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

328

exempleForm > src > app > modules > fruits > TS fruits-routing.module.ts >

```
5  const routes: Routes = [
6    { path:'pomme', component:PommeComponent}
7  ];
```

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

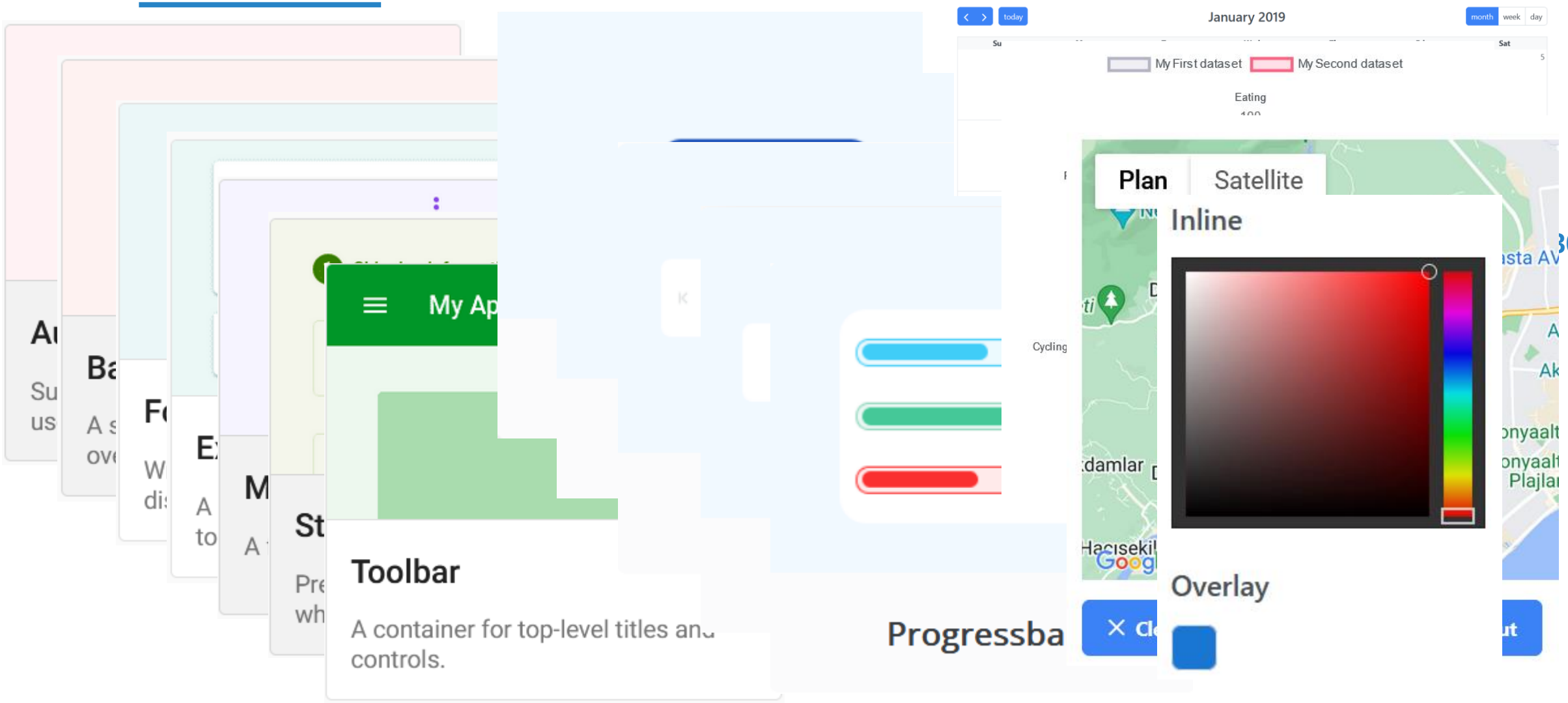
ANGULAR : MODULES

→ Lors de l'apprentissage d'Angular : utilisation de module et moins création de module.

→ Plusieurs modules sont déjà créés :

- Angular material (https://material.angular.io/)
- NGX-Bootstrap (https://valor-software.com/ngx-bootstrap)
- primeNG (https://www.primefaces.org/primeng/)
- Etc.

ANGULAR


today
January 2019
month
week
day
My First dataset
My Second dataset
Eating
Cycling
Plan
Satellite
Inline
Overlay
Toolbar
A container for top-level titles and controls.
Progressba

FRAMEWORK ANGULAR

Utilisation de Framework

Notion de composant web

Concept d’Angular

Tests

ANGULAR : TEST

Possibilité de créer et exécuter :

→ des tests unitaires grâce à **Karma** et **Jasmine**.

→ des tests end-to-end avec **Protractor** (anciennement) ou **Cypress**.

ANGULAR : TEST

Test unitaire :

→ Pour les exécuter avec Angular CLI : `ng test`.

Framework **Jasmine** :

→ Ils sont rédigés en TypeScript,

→ Utilisation du *task-runner* Karma

→ Configuration au niveau des fichiers *karma.conf.js* et *app/test.ts*

ANGULAR : TEST

→ Récent problème lors de l'exécution de la commande `ng test` (début 2022):

Présence d'une boucle infinie dans un des module npm `Colors` (version 1.4.2)

→ Boucle infinie insérée par le développeur

ANGULAR : TEST

Framework **Jasmine** :

→ Basé sur le *BDD* (*Behaviour Driven Development*).

Concepts en commun :

→ **Describe** : définit le titre du test et donne la fonction contenant les diverses spécifications.

→ **It** : possède le titre d'une spécification et contient les tests validant cette spécification (assertions).

→ **Expect** : assertion entre deux valeurs, se basant sur un matcher (ex: la valeur A DOIT être égale à la valeur B).

→ **Matcher** : aides mise à disposition pour rédiger les assertions (toEqual, contains, etc.).

ANGULAR : TEST

Couple `expect - matcher` :

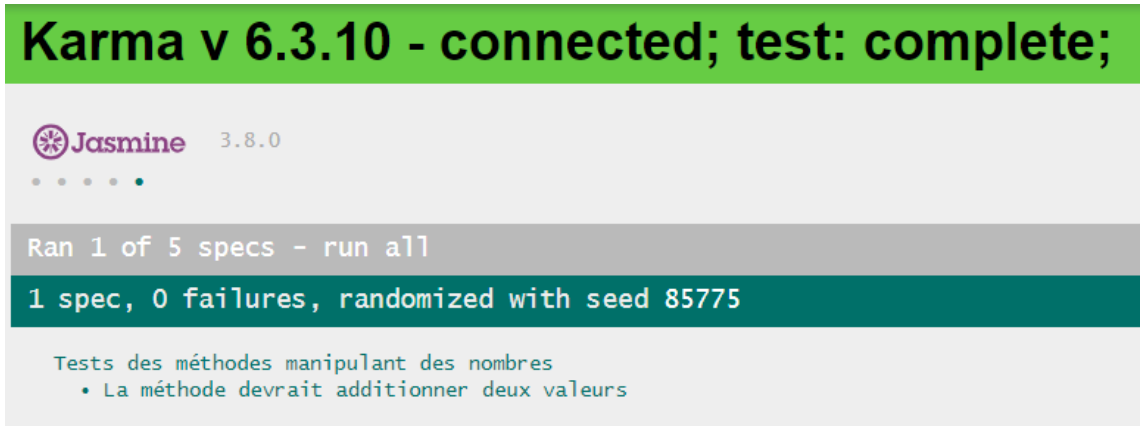
→ Large panel de matcher mis à disposition :

- toBe(), toBeNull(), toEqual(), toBeUndefined(), toBeFalsy(), etc.
- toThrow(), toThrowError(), etc.
- toContain()
- toBeGreaterThan()

→ Négation : expect(value).not.toEqual(otherValue)

→ Possibilité de créer des matchers personnalisés

ANGULAR : TEST



→ Test sur une fonction :

```
function additionner(a: number, b: number) {
  return a+b;
}
```


→ Créer un fichier *.spec.ts* avec le test suivant :

```
describe('Tests des méthodes manipulant des nombres', () => {
  it("La méthode devrait additionner deux valeurs", () => {
    let a = 1;
    let b = 2;
    let result = additionner(a, b);
    expect(result).toEqual(3);
  })
});
```

ANGULAR : TEST

→ Si le test est incorrect :

ANGULAR : TEST

→ Dans une méthode `describe`, plusieurs méthodes `it` peuvent être déclarée.

→ Attention à la redondance de code lors de l'initialisation de chaque test.

→ Solution : exécuter du code avant et après chaque test avec `beforeEach` et `afterEach`.

ANGULAR : TEST

```
describe('Tests des méthodes manipulant des nombres', () => {
  let value:number;
  beforeEach( () => {
    value = 1;
    //autres initialisations de données
  })
  beforeEach(async () => {
    //mot clé async à importer de @angular/core/testing
  })
  it("La méthode devrait additionner deux valeurs", () => {
    let a = value;
    let b = 2;
    let result = additionner(a, b);
    expect(result).toEqual(3);
  })
  afterEach(()=>{
  })
});
```

ANGULAR : TEST

Test sur les composants :

→ Besoin d'utiliser `TestBed` pour tester les composants Angular :
Point d'entrée des interfaces de tests

→ Configuration des tests réalisée via la méthode `configureTestingModule` de `TestBed`

→ Utilisation de la méthode `createComponent` pour créer le composant (pour les tests) qui renvoi un objet de type `ComponentFixture`.

Permet de récupérer l'instance du composant, les éléments HTML du composant, etc.

ANGULAR : TEST

Récupération des données dans les méthodes beforeEach()

Récupération de l'instance du composant à partir de `fixture`

test > src > app > first > TS first.component.spec.ts > ...

```ts
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { FirstComponent } from './first.component';

describe('FirstComponent', () => {
  let component: FirstComponent;
  let fixture: ComponentFixture<FirstComponent>;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [ FirstComponent ]
    })
    .compileComponents();
  });

  beforeEach(() => {
    fixture = TestBed.createComponent(FirstComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });
```

ANGULAR : TEST

→ Premier test : instanciation du composant

Dans la même fonction `describe` :

```
it('should create', () => {
  expect(component).toBeTruthy();
});
```


→ Deuxième test : utilisation des propriétés du composant

```
it('la propriété doit avoir la valeur Hello',async () => {
  expect(component.variable).toEqual('Hello');
})
```

ANGULAR : TEST

→ Vérifier la cohérence du rendu :


```
it('la balise div myId doit contenir la valeur de la propriété variable', async () => {
  fixture.detectChanges();
  expect(componentHTMLElement.querySelector('#myId')?.textContent)
    .toContain(component.variable)
})
```



```
beforeEach(() => {
  fixture = TestBed.createComponent(FirstComponent);
  component = fixture.componentInstance;
  componentHTMLElement = fixture.debugElement.nativeElement;
```

test > src > app > first > first.component.html > ...

```
1 <div id="myId">
2     {{variable}}
3 </div>
```

ANGULAR : TEST

Test e2e : Tests complémentaires aux tests unitaires.

Possibilité de tester de bout en bout une fonctionnalité en définissant une suite d'interaction à réaliser.

→ Jusqu'à la version **12** d'Angular :
Framework **Jasmine** pour écrire les tests,
Protractor pour exécuter les scripts (programme node.js utilisant Selenium pour permettre aux tests d'interargir avec le navigateur).

→ Maintenant : utilisation d'autres librairies comme **Cypress** (basée sur **Mocha**) ou encore **WebdriverIO, Nightwatch** ou **TestCafe**.

ANGULAR : TEST

Cypress :

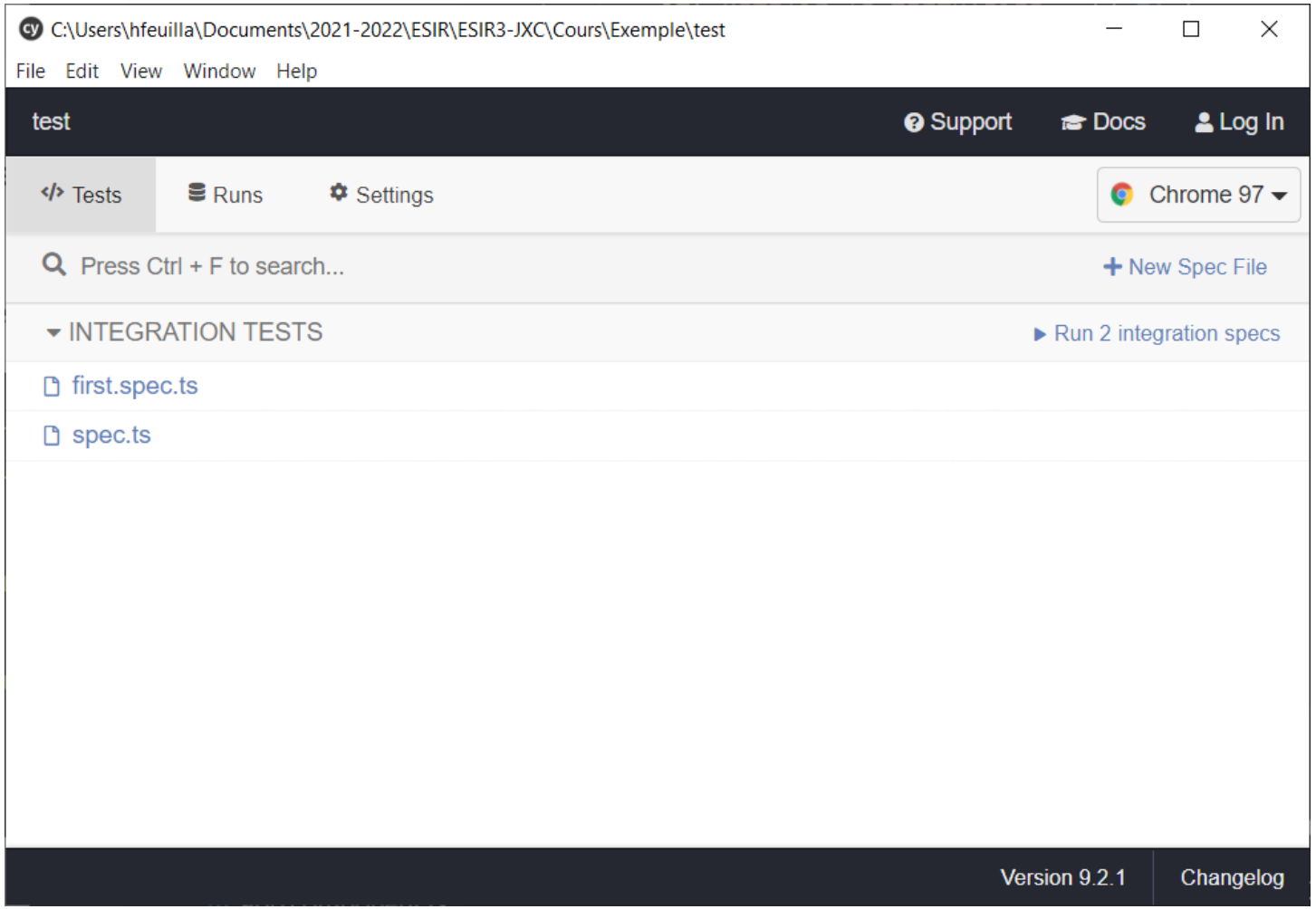
Installation de Cypress dans le projet Angular : **`ng add @cypress/schematic`**

→ Tests sont structurés avec le framework **Mocha** couplé avec **Chai** pour les assertions.

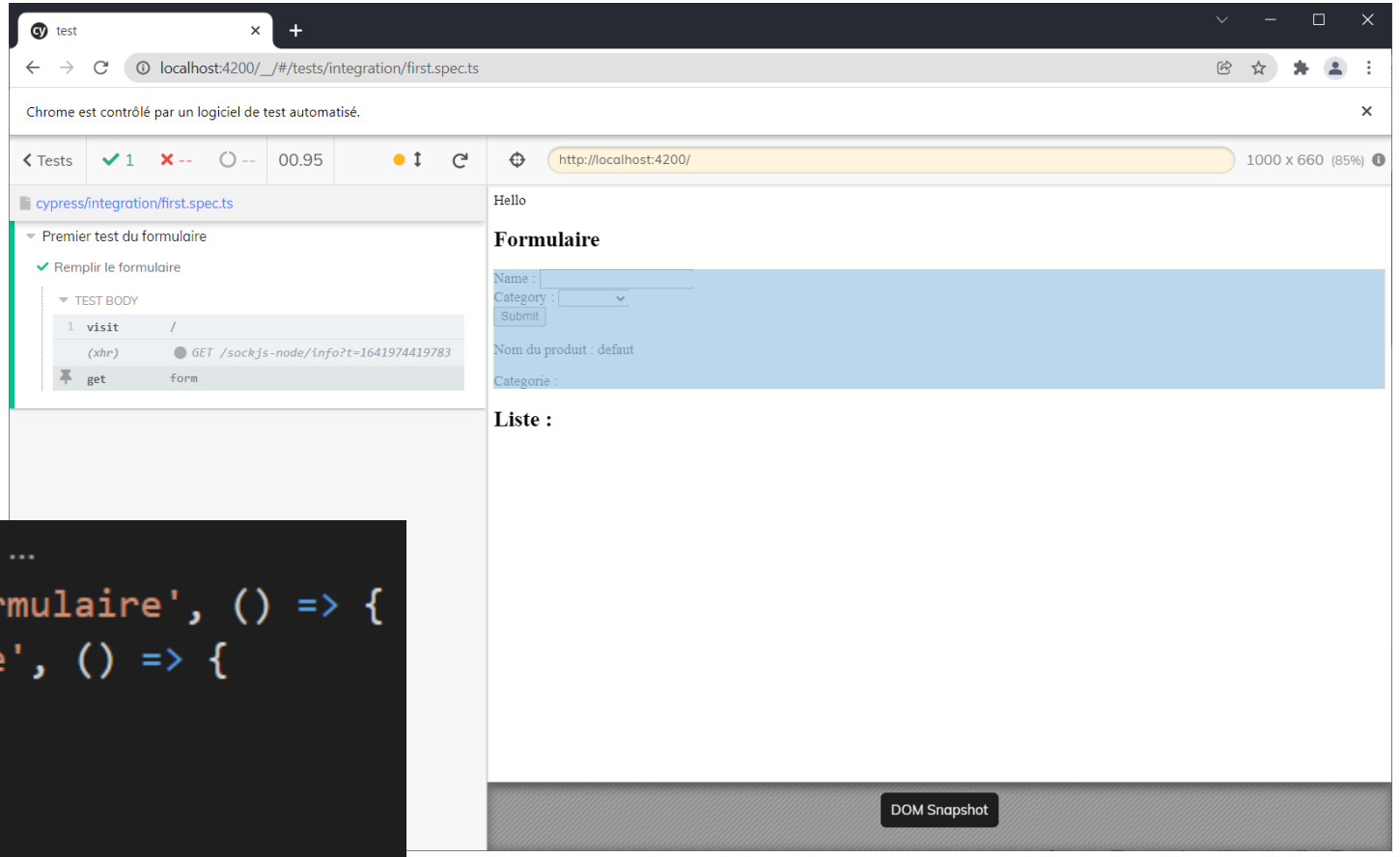
→ Similaire à la syntaxe de Jasmine.

→ Pour exécuter les tests e2e avec CLI : **`ng e2e`**

ANGULAR : TEST

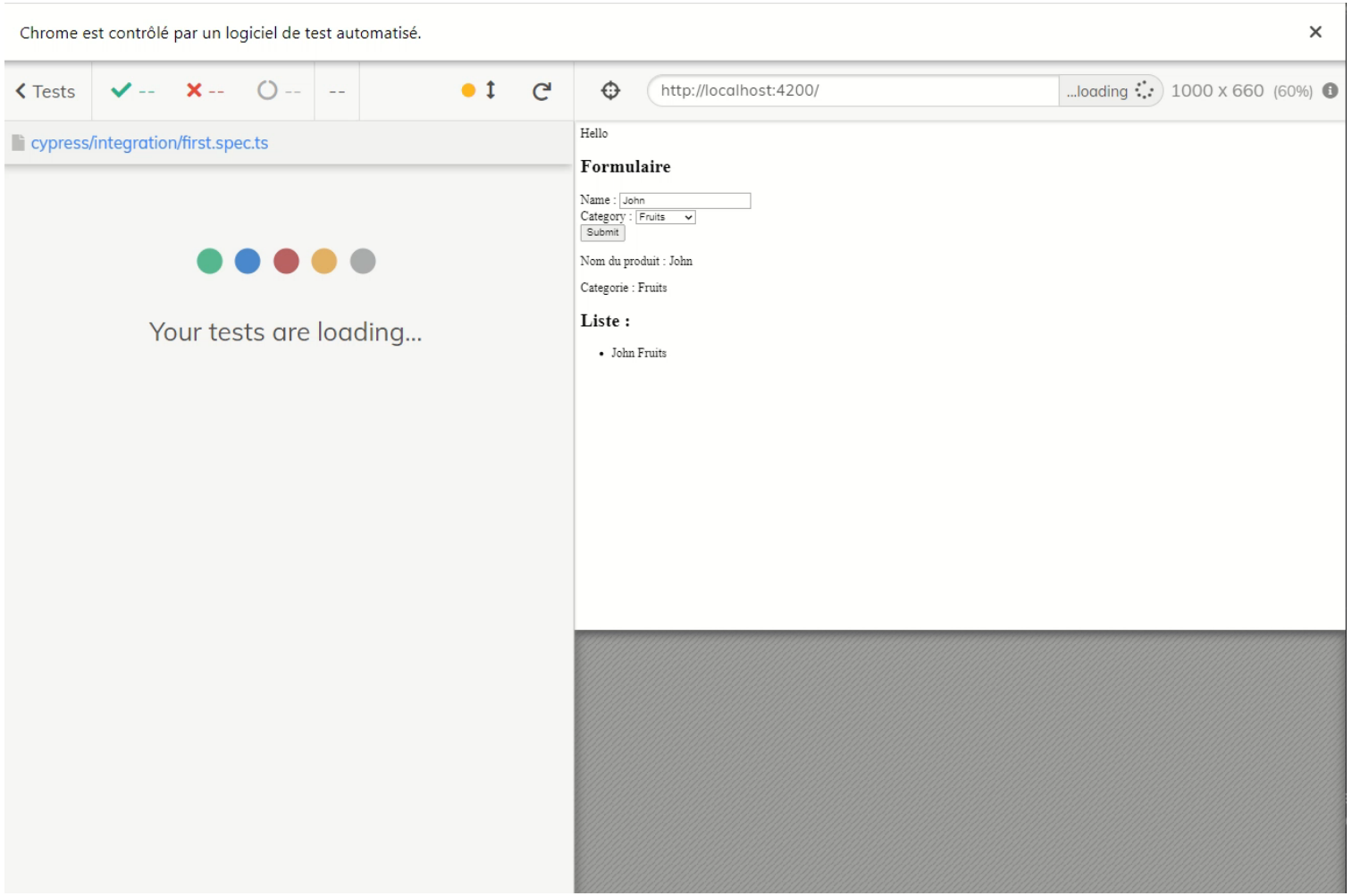


ANGULAR : TEST




```
test > cypress > integration > TS first.spec.ts > ...
1   describe('Premier test du formulaire', () => {
2       it('Remplir le formulaire', () => {
3           cy.visit('/')
4           cy.get("form")
5       })
6   })
```

ANGULAR : TEST



ANGULAR : TEST

→ Code du test précédent :

test > cypress > integration > TS first.spec.ts > ...

```
describe('Premier test du formulaire', () => {
    it('Remplir le formulaire', () => {
        cy.visit('/')
        cy.get("form")
        cy.get('input[name="name"]').type("John")
            .should("have.value", "John");
        cy.get('select[name="category"]').select('Fruits')
            .should("have.not.value", "Légumes");
        cy.get('button').should('be.visible')
        cy.get('form').find('button').should('not.have.class', 'disabled')
        cy.get('#btnSubmit').click()
    })
})
```

ANGULAR : TEST

Avec :

test > src > app > first > first.component.html > form > div.group > select#category

```
4   <h2>Formulaire</h2>
5   <form #productForm="ngForm" (ngSubmit)="ajouterProduit()">
6       <div class="group">
7           <label for="name">Name : </label>
8           <input type="text" name="name" id="name" [formControl]="nameControl" required>
9       </div>
10      <div class="group">
11          <label for="category">Category : </label>
12          <select id="category" name="category" [formControl]="categoryControl" >
13              <option *ngFor="let cat of categories" [value]="cat">{{cat}}</option>
14          </select>
15      </div>
16      <button id="btnSubmit" type="submit" [disabled]="!productForm.valid">Submit</button>
17      <div>
18          <p>Nom du produit : {{name}}</p>
19          <p>Categorie : {{category}}</p>
20      </div>
21  </form>
22  <h2>Liste :</h2>
23  <div id="Liste">
24      <ul>
25          <li *ngFor="let prod of produits">
26              {{ prod.name }} {{ prod.category }}
27          </li>
28      </ul>
```

ANGULAR : TEST

Possibilité de faire des tests visuels avec le plugin **snapshot** :

→ Installation du plugin dans le projet Angular

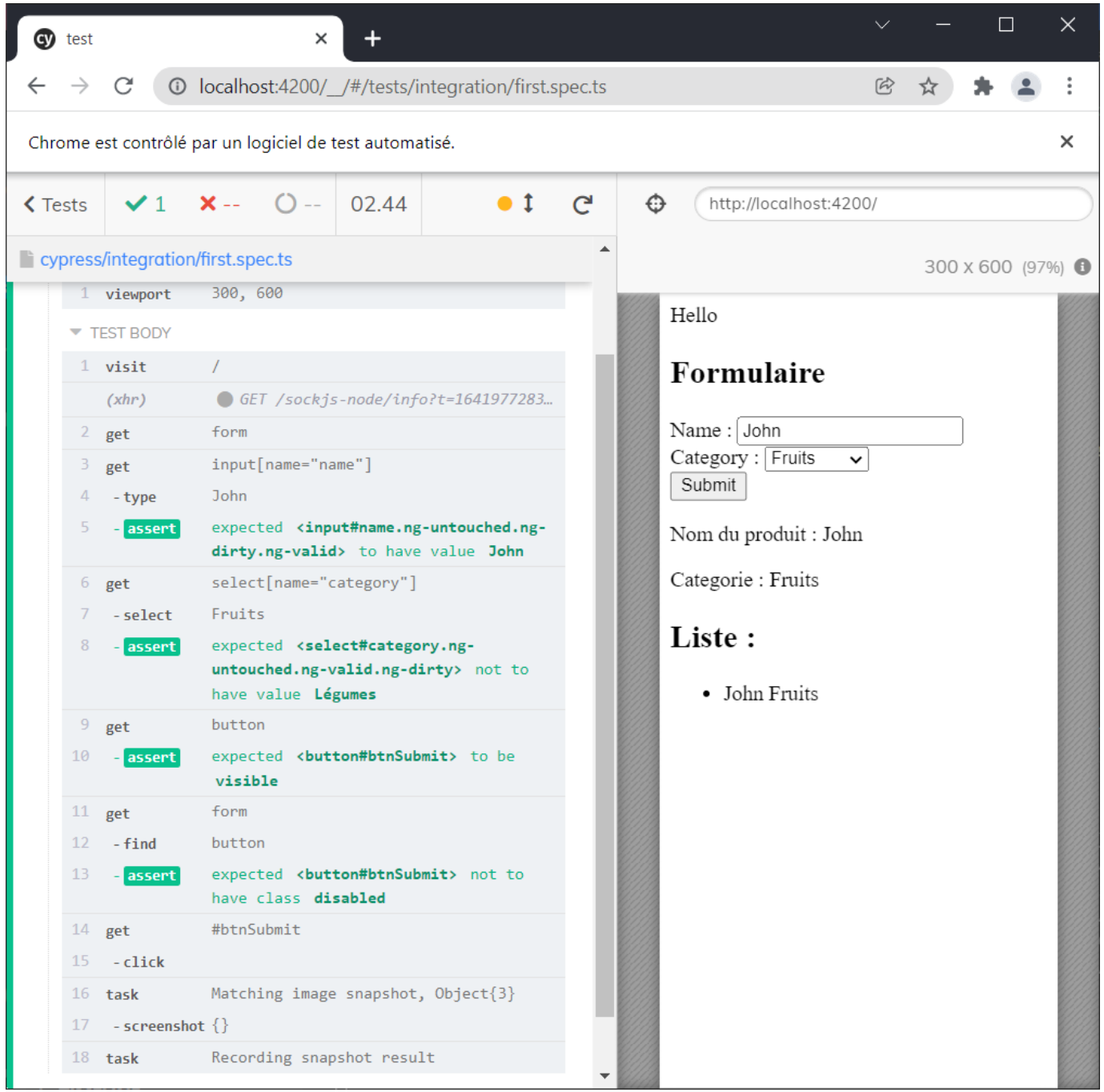
→ Enregistrer le plugin dans le fichier *cypress/plugins/index.js*

→ Penser à fixer le viewport

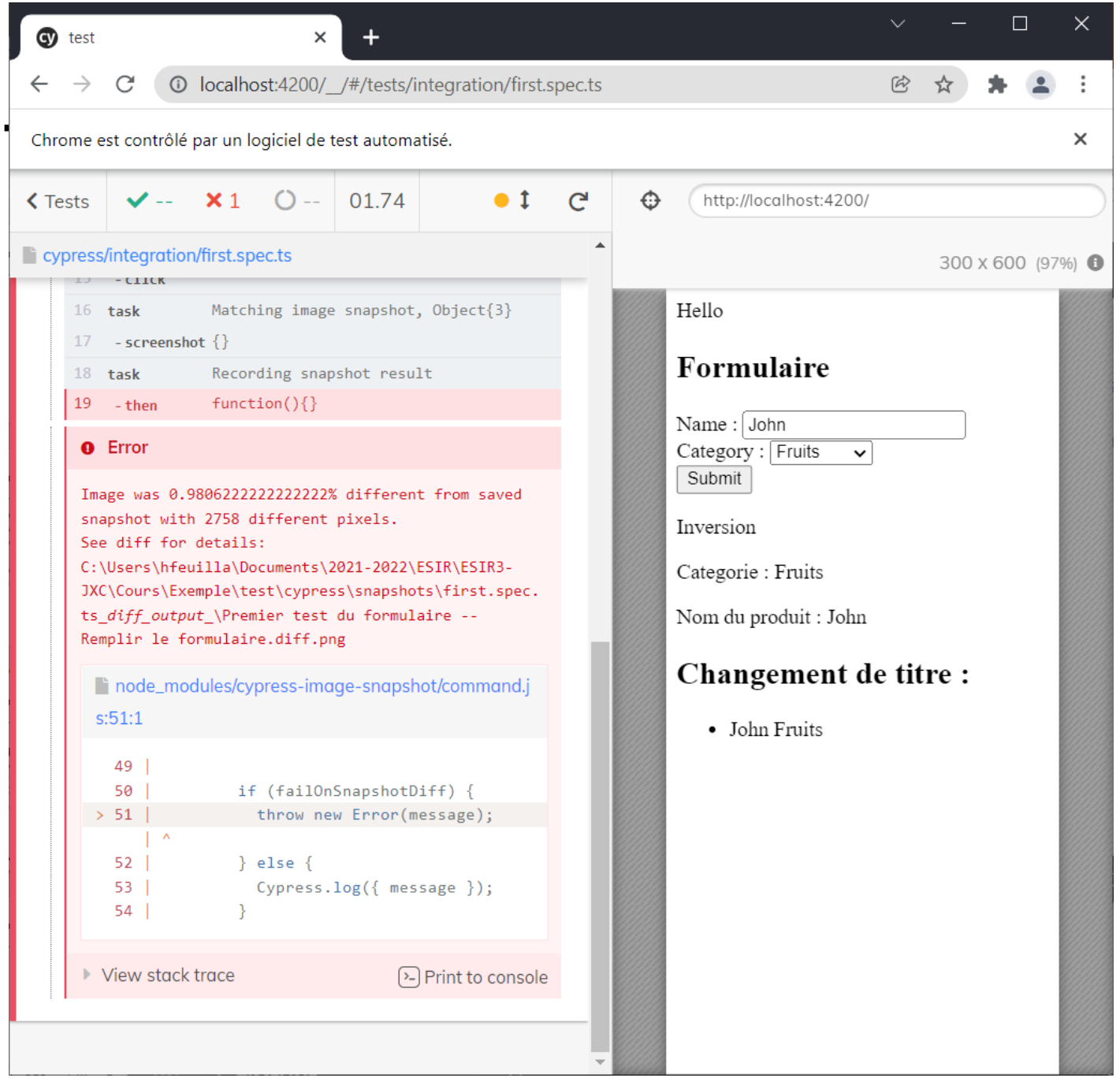
→ Fonction cy.matchImageSnapshot() a appeler pour réaliser le snapshot

ANGULAR :


test > cypress > snapshots > first.spec.ts > Premi
Hello
Formulaire
Name : John
Category : Fruits
Submit
Nom du produit : John
Categorie : Fruits
Liste :
John Fruits


test
localhost:4200/__/#/tests/integration/first.spec.ts
Chrome est contrôlé par un logiciel de test automatisé.
Tests
1
--
--
02.44
http://localhost:4200/
cypress/integration/first.spec.ts
300 x 600 (97%)
1 viewport 300, 600
TEST BODY
1 visit /
(xhr) GET /sockjs-node/info?t=1641977283…
2 get form
3 get input[name="name"]
4 - type John
5 - assert expected <input#name.ng-untouched.ng-dirty.ng-valid> to have value John
6 get select[name="category"]
7 - select Fruits
8 - assert expected <select#category.ng-untouched.ng-valid.ng-dirty> not to have value Légumes
9 get button
10 - assert expected <button#btnSubmit> to be visible
11 get form
12 - find button
13 - assert expected <button#btnSubmit> not to have class disabled
14 get #btnSubmit
15 - click
16 task Matching image snapshot, Object{3}
17 - screenshot {}
18 task Recording snapshot result
Hello
Formulaire
Name : John
Category : Fruits
Submit
Nom du produit : John
Categorie : Fruits
Liste :
John Fruits

ANGULAR :


test
localhost:4200/__/#/tests/integration/first.spec.ts
Chrome est contrôlé par un logiciel de test automatisé.
Tests
-- 1 -- 01.74
http://localhost:4200/
cypress/integration/first.spec.ts
300 x 600 (97%)
16 task Matching image snapshot, Object{3}
17 -screenshot {}
18 task Recording snapshot result
19 -then function(){}
Error
Image was 0.9806222222222222% different from saved snapshot with 2758 different pixels.
See diff for details:
C:\Users\hfeuilla\Documents\2021-2022\ESIR\ESIR3-JXC\Cours\Exemple\test\cypress\snapshots\first.spec.ts_diff_output_\Premier test du formulaire -- Remplir le formulaire.diff.png
node_modules/cypress-image-snapshot/command.js:51:1
49 |
50 | if (failOnSnapshotDiff) {
> 51 | throw new Error(message);
| ^
52 | } else {
53 | Cypress.log({ message });
54 | }
View stack trace
Print to console
Hello
Formulaire
Name : John
Category : Fruits
Submit
Inversion
Categorie : Fruits
Nom du produit : John
Changement de titre :
John Fruits

ANGULAR : TEST