



FORMATIO N

- Management
2025

Bienvenue sur cette formation Angular

- Clément Hamon
- Développeur depuis 5 ans
- 3 ans de AngularJs et Angular 2 en entreprise
- 2 ans de formation
- clement.hamon35@gmail.com

Horaire et convocations

- 9h30 - 17h30
- 15 mn de pause le matin
- 1h de pause déjeuner
- 15 mn de pause l'après midi

- QCM à la fin pour vous évaluer

Déroulé et structure de la formation et formalités

- Emargement le matin et l'après-midi
- Google Forms de demi-journée pour la validation des acquis et l'adaptabilité
- Notions théoriques suivis de mise en pratique

Tour de table et pré-requis

- Qui êtes-vous et quel est votre expérience dans l'informatique ?
- Vos attentes à l'issue de cette formation ?
- Bonne connaissance de Javascript et Typescript

Structure et versionning Github

- Pour chaque notion montrée, un commit sera disponible sur ce git
- <https://github.com/ClementHamonDev/Formation-Angular-Juin>
- Support de cours sur Teams et Github

Objectifs de la formation

- 1.** Découverte de la mise en place d'un environnement de développement web
- 2.** Découverte du JavaScript Moderne (ECMASCRIPT 5 et +)
- 3.** Développer le front d'une application avec Angular

Installation des outils

- Installation Node.js - v^18.19.1 or newer
- Visual Code Studio

1. ES6



Rappels ES6

structuration

```
//array structuration
let arrayStructuration = [1, 2]
console.log(arrayStructuration); // [1, 2]

//array copy
let arrayCopy = [...arrayStructuration, 3, 4]
console.log(arrayCopy); // [1, 2, 3, 4]

//array destructuration
let [a, b] = arrayCopy;
console.log(a); // 1
console.log(b); // 2
```

structuration

```
//object structuration
let obj = {
  message: "hello world"
}
console.log(obj); // {message: "hello world"}

//object add key
obj.type = "success";
console.log(obj); // {message: "hello world", type: "success"}

//object copy
let copy = {...obj}

//object destructuration
const {message} = obj;
console.log(message); // "hello world"
```



Rappels ES6

• • •

var vs let

```
function varTest() {  
  var x = 1;  
  if (true) {  
    var x = 2; // c'est la même variable !  
    console.log(x); // 2  
  }  
  console.log(x); // 2  
}  
  
function letTest() {  
  let x = 1;  
  if (true) {  
    let x = 2; // c'est une variable différente  
    console.log(x); // 2  
  }  
  console.log(x); // 1  
}  
  
varTest();  
letTest();
```



Rappels ES6

Cheat sheet: <https://htmlcheatsheet.com/js/>

fonctions fléchées

```
//from this
function foo() {
  console.log("bar");
}

//to this
const foo = () => {
  console.log("bar");
}

//or this
const foo = () => console.log("bar");
```

array functions

```
//from this
const array = [1, 2, 3];

for (let i = 0; i < array.length; i++) {
  console.log(array[i]);
}

//to this
const array = [1, 2, 3];

array.map(i => console.log(i));
```



Cheat sheet fonction fléchées

```
● ● ● Uploaded using RayThis Extension  
  
const a = (param) => param; // paramètre unique, return implicite  
  
const b = param => param; // paramètre unique (parenthèse non requise), return implicite  
  
const c = (param1, param2) => param1 + param2; // paramètres multiples (parenthèses requises), return implicite
```

```
● ● ● Uploaded using RayThis Extension  
  
const a = () => {  
    return "hello"  
} // multi-lignes, return explicite  
  
const b = () => "hello" // une seule ligne, return implicite  
  
const c = () => (  
    "hello"  
)// multi ligne, return implicite
```



Threads

- █ Task 1
- █ Task 2
- █ Task 3

Single-threaded
synchronous (example)



Multi-threaded
synchronous (example)



Single-threaded
Asynchronous (example)



Multi-threaded
Asynchronous (example)





Rappels ES6

```
Uploaded using RayThis Extension

//retourne une promesse qui se résoudra après ms millisecondes
const sleep = (ms) => new Promise(resolve => setTimeout(resolve, ms));

const synchroneFunction = () => {
  console.log("start");
  sleep(3000);
  console.log("end");// executé immédiatement :-(

}

synchroneFunction();

const asynchroneFunction = async () => {
  console.log("start");
  await sleep(3000);
  console.log("end");// executé après 3 secondes :-D

}

asynchroneFunction();
```

```
Uploaded using RayThis Extension

//retourne une promesse qui se résoudra après ms millisecondes
const sleep = (ms) => new Promise(resolve => setTimeout(resolve, ms));

const synchroneFunction = () => {
  console.log("start");
  sleep(3000).then(() => {
    console.log("end");// executé après 3 secondes
  })
}

synchroneFunction();
```



Rappels ES6

```
function asyncFunction() {
  return new Promise((resolve, reject) => {
    // Simulation d'une opération asynchrone
    setTimeout(() => {
      const randomNumber = Math.random();

      if (randomNumber < 0.5) {
        // Résoudre la promesse avec le nombre aléatoire
        resolve(randomNumber);
      } else {
        // Rejeter la promesse avec une erreur
        reject(new Error('Une erreur s\'est produite !'));
      }
    }, 1000);
});
```

```
...  
  
// Utilisation de try/catch pour capturer l'erreur  
async function executeAsyncFunction() {  
  try {  
    const result = await asyncFunction();  
    console.log('Résultat :', result);  
  } catch (error) {  
    console.error('Erreur :', error.message);  
  }  
}
```

```
...  
  
// Utilisation de .catch pour capturer l'erreur  
asyncFunction()  
  .then(result => {  
    console.log('Résultat :', result);  
  })  
  .catch(error => {  
    console.error('Erreur :', error.message);  
});
```



Rappels ES6

Waterfall

```
async function asyncOperation1() { /* ... */ }
async function asyncOperation2() { /* ... */ }
async function asyncOperation3() { /* ... */ }

async function executeAsyncOperations() {
  try {
    const result1 = await asyncOperation1();
    const result2 = await asyncOperation2(result1);
    const result3 = await asyncOperation3(result2);
    console.log('Résultat final : ', result3);
  } catch (error) {
    console.error('Erreur : ', error);
  }
}

executeAsyncOperations()
```

Parallel

```
async function asyncOperation1() { /* ... */ }
async function asyncOperation2() { /* ... */ }
async function asyncOperation3() { /* ... */ }

async function executeParallelOperations() {
  try {
    const [result1, result2, result3] = await Promise.all([
      asyncOperation1(),
      asyncOperation2(),
      asyncOperation3()
    ]);
    console.log('Résultat 1 : ', result1);
    console.log('Résultat 2 : ', result2);
    console.log('Résultat 3 : ', result3);
  } catch (error) {
    console.error('Erreur : ', error);
  }
}

executeParallelOperations();
```

Parallel

```
const asyncQueue = [];

function addToQueue(asyncTask) {
  asyncQueue.push(asyncTask);
  if (asyncQueue.length === 1) {
    processQueue();
  }
}

function processQueue() {
  const asyncTask = asyncQueue[0];
  asyncTask()
    .then(result => {
      console.log('Résultat de la tâche : ', result);
      asyncQueue.shift();
      if (asyncQueue.length > 0) {
        processQueue();
      }
    })
    .catch(error => {
      console.error('Erreur de la tâche : ', error);
      asyncQueue.shift();
      if (asyncQueue.length > 0) {
        processQueue();
      }
    });
}

// Exemple d'utilisation
addToQueue(asyncOperation1);
addToQueue(asyncOperation2);
```

2. Différence Angular et autres frameworks

Angular vs React

- Angular est souvent comparé à React, un autre framework populaire développé par Google.
- Contrairement à React, Angular est un **framework complet** avec une multitude d'outils intégrés : formulaires, routes, services....
- React, plus léger, se concentre uniquement sur la partie "V" (**Vue**) du modèle MVC

Angular vs Vue

- Vue.js est souvent considéré comme un compromis entre Angular et React.
- Comme React, Vue.js est axé sur la création de composants et propose un système similaire de Virtual DOM. Cependant, Vue.js est un peu plus **opinionated** (il impose plus de conventions) que React, offrant une approche intégrée pour la gestion des états et des routes, mais sans la complexité d'Angular.
- Angular peut sembler plus compliqué mais offre plus de flexibilité et d'évolutivité pour les projets de grande envergure.

Performance

- React et Vue.js sont souvent assez proches grâce à l'utilisation du Virtual DOM.
- Angular, étant un framework plus lourd avec plus de fonctionnalités intégrées, peut être plus lent dans certaines situations si toutes ses fonctionnalités ne sont pas nécessaires.
- Angular est donc un excellent choix si vous travaillez sur une application dans une entreprise à grande échelle et avez besoin d'un framework robuste et riche en fonctionnalités. Dans ce cas Angular est un excellent choix mais il est aussi largement utilisable et utilisé pour des projets de plus petites échelles.

3. Principes de base de Angular

Initiation du projet

- npm install -g @angular/cli
- ng new <project-name>
- cd <project-name>
- npm start
- http://localhost:4200/

Composant

- Angular repose entièrement sur l'idée de **composants**.
- Partie autonome et indépendante d'une interface utilisateur
- Réutilisable et imbriquable dans d'autres composants
- Propre état et cycle de vie

Composant

Chaque composant possède ces parties principales :

- Un décorateur `@Component` qui contient certaines configurations utilisées par Angular.
- Un template HTML qui contrôle ce qui est rendu dans le DOM.
- Un sélecteur CSS qui définit comment le composant est utilisé dans le HTML.
- Une classe TypeScript avec des comportements, comme la gestion des interactions utilisateur ou l'envoi de requêtes à un serveur.

Composant

Fichier Typescript

```
● ● ● Uploaded using RayThis Extension  
  
import { Component } from '@angular/core';  
  
@Component({  
  selector: 'app-user',  
  imports: [],  
  templateUrl: './user.component.html',  
  styleUrls: ['./user.component.css']  
})  
export class UserComponent {  
  
}
```

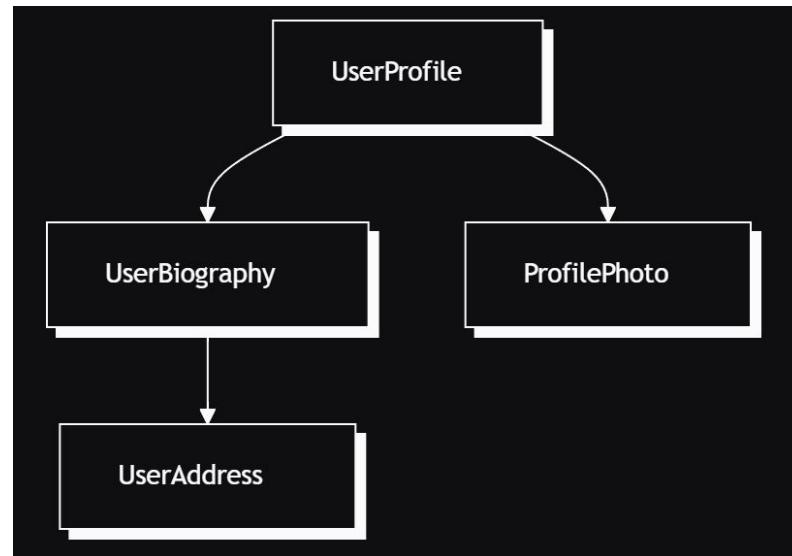
Fichier HTML

```
● ● ● Uploaded using RayThis Extension  
  
<p>Ceci est le composant de User</p>
```

Composant

Vous construisez une application en assemblant plusieurs composants ensemble.

Par exemple, si vous créez une page de profil utilisateur, vous pouvez diviser la page en plusieurs composants comme ceci :



Imbrication des composants

- Component Parent qui contient un ou des components Enfant



Exercice

Exercice 1 : Crédation d'un composant

1. Crée un composant `HelloWorld` qui retourne un message "Hello World!" dans un élément `<p>`.
2. Affiche ce composant dans `app`.

Exercice 2 : Crédation de composants fonctionnels

1. Crée un composant `Welcome` qui retourne un message de bienvenue
2. Dans `app`, affiche trois fois le composant `Welcome`.

Exercice 3 : Imbrication de composants

1. Crée un composant `Titre` qui affiche un titre `<h3>`.
2. Crée un composant `Page` qui retourne les composants `Titre` et `Welcome` (créé dans l'ex2) imbriqués ensemble.
3. Affiche le composant `Page` dans `app`.

Bonus : (Re)découvrir et faire du CSS en angular

Signals

Créez et gérez des données dynamiques.

- Utilisez la fonction `signal` pour créer un signal permettant de stocker un état local
- Angular suit les endroits où les signaux sont lus et les met à jour automatiquement lorsqu'ils changent.
- Writable

```
counter = signal(0); // Crée un signal avec une valeur initiale de 0

constructor() {
  this.init();
}

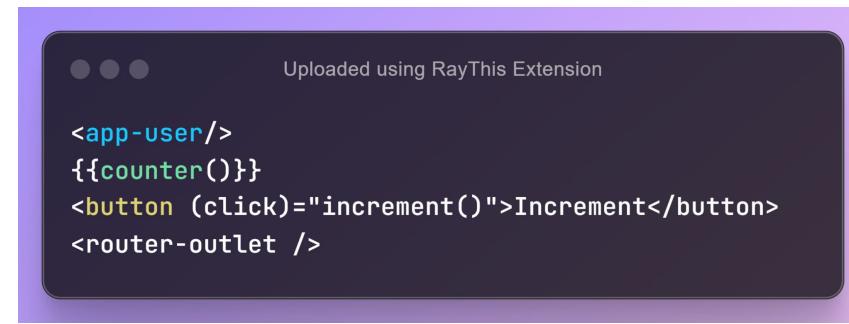
init() {
  // Pour accéder à la valeur du signal
  console.log(this.counter()); // Affiche 0

  // Pour modifier la valeur du signal
  this.counter.set(1);
  console.log(this.counter()); // Affiche 1
}
```

Signals

Les utiliser dans un composant :

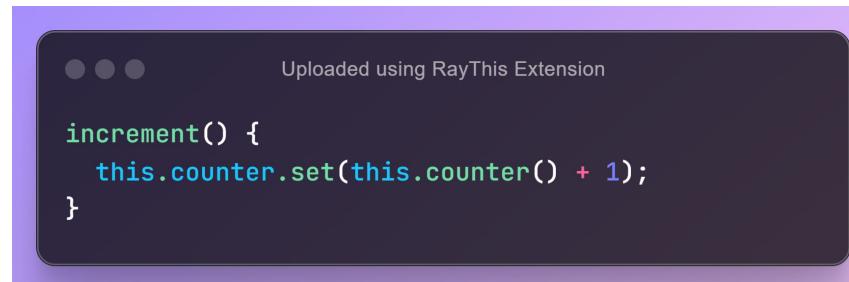
HTML



Uploaded using RayThis Extension

```
<app-user/>
{{counter()}}
<button (click)="increment()">Increment</button>
<router-outlet />
```

TypeScript



Uploaded using RayThis Extension

```
increment() {
  this.counter.set(this.counter() + 1);
}
```

Computed

C'est une valeur qui dépend d'un ou plusieurs signals et qui est recalculée automatiquement lorsque ces derniers changent.

Utilisation principale : éviter les recalculs inutiles et optimiser les performances.

Ne déclenche pas d'effets secondaires, c'est une fonction pure.



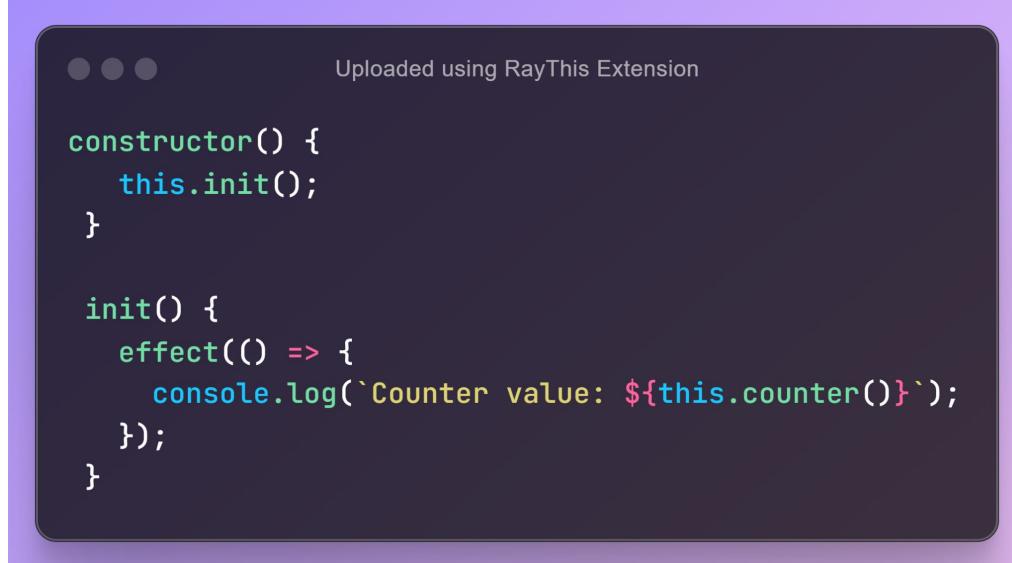
Uploaded using RayThis Extension

```
firstName = signal('Morgan');
firstNameCapitalized = computed(() => this.firstName().toUpperCase()); //Crée un computed
qui retourne le prénom en majuscule
```

Effect

Gérer les effets secondaires

- Fonction qui s'exécute chaque fois qu'un **signal** utilisé à l'intérieur change.
- Utilisation principale : déclencher du code avec des effets de bord (logs, requêtes HTTP, mises à jour DOM, etc.).
- Pas de valeur renvoyée, il ne stocke rien.



Uploaded using RayThis Extension

```
constructor() {
  this.init();
}

init() {
  effect(() => {
    console.log(`Counter value: ${this.counter()}`);
  });
}
```

Exercice

Exercice 1 : Utilisation des `signals`

1. Dans composant `WelcomeWithname` , crée un signal `name` et affiche "Bienvenue, {{name()}}".
2. Utilise ce composant dans `app`.

Exercice 2 : Utilisation des `computed`

3. Crée un composant `UserInfo` qui a deux `signals` : `name` et `age`, et les affiche avec un petit texte de présentation.
1. Crée un computed qui regarde si l'utilisateur est majeur.
2. Utilise ce composant dans `app` .

Exercice 3 : Utilisation de `effect`

1. Crée dans `UserInfo` un effect pour afficher dans la console le nom et l'âge de l'utilisateur et un effect qui envoie dans la console un message d'erreur si l'utilisateur est mineur.

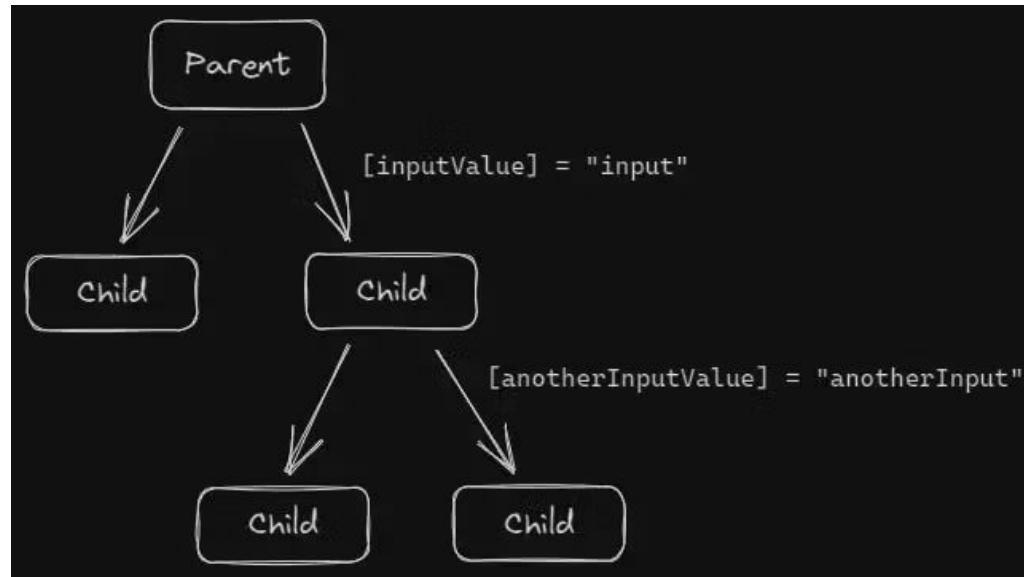
Exercice 4 : Tout ensemble

1. Crée une page utilisateur avec au moins 5 informations pertinentes et des boutons d'éditions qui mettent à jour ces valeurs. A chaque mise à jour, fais une récapitulatif des informations dans la console.

Bonus : Fais du CSS pour mettre en forme cette page

Input

Passer des données d'un composant parent à un composant enfant

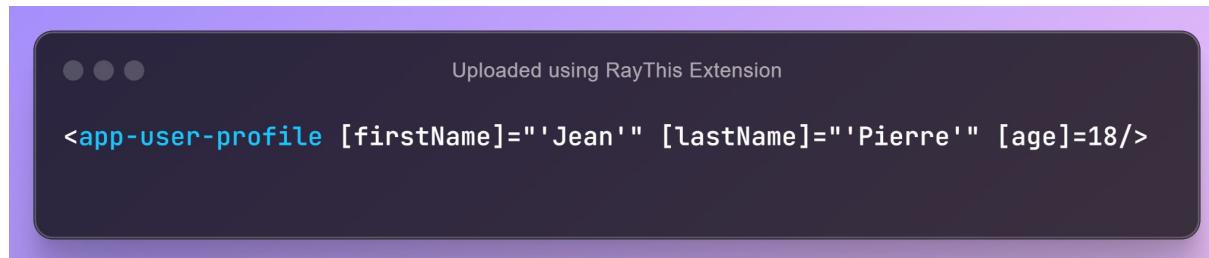


Input

Passer des données d'un composant parent à un composant enfant

Les changements de la valeur dans le parent sont répercutés dans l'enfant.

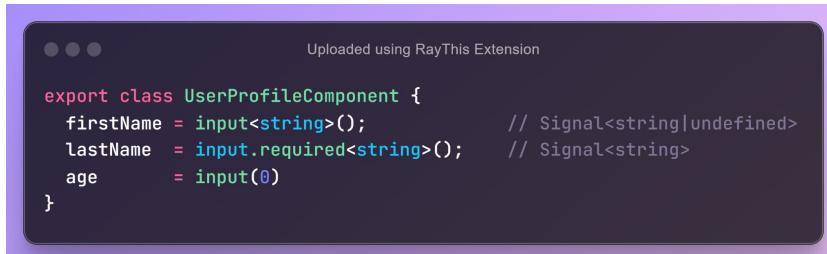
HTML Parent



Uploaded using RayThis Extension

```
<app-user-profile [firstName]="'Jean'" [lastName]="'Pierre'" [age]=18/>
```

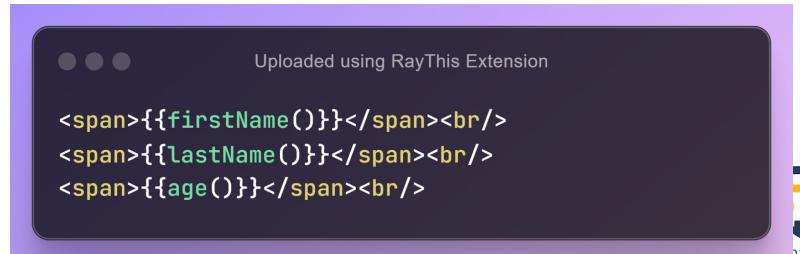
TS Enfant



Uploaded using RayThis Extension

```
export class UserProfileComponent {  
  firstName = input<string>(); // Signal<string|undefined>  
  lastName = input.required<string>(); // Signal<string>  
  age      = input(0)  
}
```

HTML Enfant



Uploaded using RayThis Extension

```
<span>{{firstName()}}</span><br/>  
<span>{{lastName()}}</span><br/>  
<span>{{age()}}</span><br/>
```

Output

Passer des données d'un composant enfant à un composant parent

TS Enfant

```
Upgraded using RayThis Extension

nameChange = output<string>(); // OutputEmitterRef<string>

sendData() {
  this.nameChange.emit('Message de l'enfant !');
}
```

TS Parent

```
Upgraded using RayThis Extension

export class UserComponent {
  receivedMessage: string = '';

  receiveData(message: string) {
    this.receivedMessage = message;
    console.log('Données reçues de l'enfant:', message);
  }
}
```

HTML Parent

```
Upgraded using RayThis Extension

<app-user-profile (nameChange)="receiveData($event)">
<p>Message reçu : {{ receivedMessage }}</p>
```

Relation bidirectionnelle

Donnée en commun entre le composant parent et le composant enfant

TS Enfant

```
...  
Uploaded using RayThis Extension  
  
export class BidirectionnalChildComponent {  
  name = input<string>();  
  nameChange = output<string>();  
  
  onInputChange(event: Event) {  
    const newValue = (event.target as HTMLInputElement).value;  
    this.nameChange.emit(newValue); // Informe le parent du changement  
  }  
}
```

HTML Enfant

```
...  
Uploaded using RayThis Extension  
  
<input [value]="name()" (input)="onInputChange($event)">
```

TS Parent

```
...  
Uploaded using RayThis Extension  
  
export class BidirectionnalParentComponent {  
  userName = 'Clément'; // Valeur initiale  
}
```

HTML Parent

```
...  
Uploaded using RayThis Extension  
  
<app-bidirectionnal-child [(name)]="userName"></app-bidirectionnal-child>  
<p>Nom mis à jour : {{ userName }}</p>
```

Relation bidirectionnelle

Donnée en commun entre le composant parent et le composant enfant

TS Enfant

```
...  
Uploaded using RayThis Extension  
  
page = model(1);
```

HTML Enfant

```
...  
Uploaded using RayThis Extension  
  
<p>Page numéro : {{ page() }}</p>
```

TS Parent

```
...  
Uploaded using RayThis Extension  
  
currentPage = signal(1);  
incrementPage(){  
    this.currentPage.set(this.currentPage() + 1);  
}
```

HTML Parent

```
...  
Uploaded using RayThis Extension  
  
<app-pagination [(page)]="currentPage" />  
<br/>  
<button (click)="incrementPage()">Page suivante</button>
```

Exercice

Exercice 1 : Utilisation de `input()`

1. Crée un composant **ChildComponent** qui affiche un message reçu du parent.
2. Le composant **ParentComponent** envoie un message à **ChildComponent**.
3. Fais la même chose avec des valeurs numériques et booléennes.

Exercice 2 : Utilisation de `output()`

1. Crée un composant **ChildComponent** qui contient un bouton "Envoyer au parent".
2. Lorsque l'utilisateur clique sur le bouton, **ChildComponent** envoie "Donnée envoyée !" au parent.
3. Le **ParentComponent** affiche le message reçu.
4. Fais la même chose avec des valeurs numériques et booléennes.

Exercice 3 : Relation bidirectionnelle

1. Ajoute un champ `<input>` dans **ChildComponent**.
2. Le parent envoie une valeur initiale, et toute modification dans l'enfant doit être répercutée dans le parent.
3. Affiche la valeur modifiée dans le **ParentComponent**.

Bonus : Fais du CSS pour mettre en forme cette page

Listes

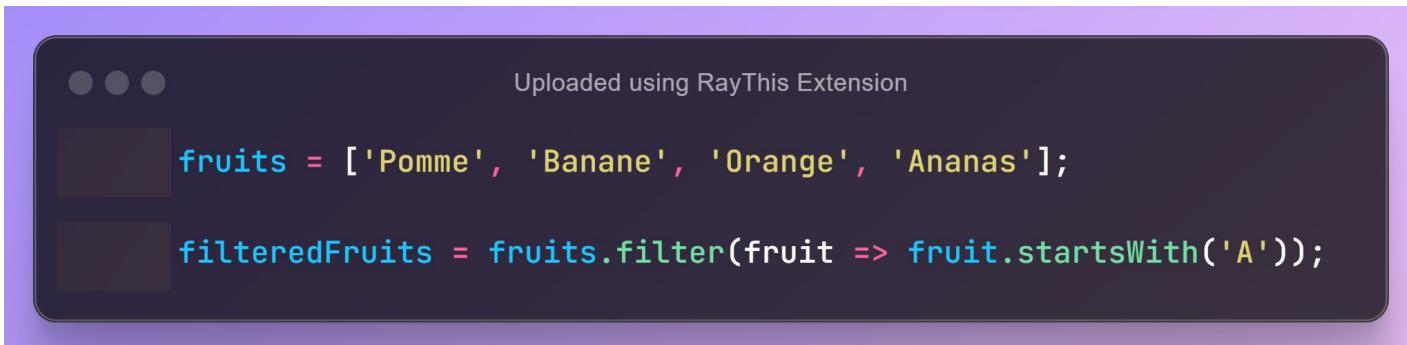
- Affichage d'une liste simple avec `@for`

```
    @for (item of items; track item.name) {  
      <li>{{ item.name }}</li>  
    } @empty {  
      <li>There are no items.</li>  
    }
```

- Si la liste est mise à jour fréquemment, il est recommandé d'utiliser `track` pour éviter les re-rendus inutiles.
- Optimise Angular en identifiant les éléments, évitant les recréations inutiles.

Fonctions

- `filter()` : Filtre une liste d'éléments
- La méthode `filter()` est utilisée pour créer un nouveau tableau contenant uniquement les éléments qui satisfont une condition.



Uploaded using RayThis Extension

```
fruits = ['Pomme', 'Banane', 'Orange', 'Ananas'];

filteredFruits = fruits.filter(fruit => fruit.startsWith('A'));
```

Exercice

Exercice 1 : Utilisation de `Liste` pour rendre une liste d'éléments

1. Crée un composant `ItemList` qui prend un input `items`, un tableau de string.
2. Affiche les éléments pour transformer ce tableau en une liste d'éléments `` dans une liste ``.
3. Utilise ce composant dans `app` avec un tableau d'exemple de chaînes de caractères.

Exercice 2 : Utilisation de `filter()` pour afficher des éléments filtrés

1. Crée un composant `FilteredItemList` qui prend deux input:
 - o `items`, un tableau de chaînes de caractères.
 - o `query`, une chaîne de caractères pour filtrer les éléments.
2. Utilise la méthode `filter()` pour ne garder que les éléments qui contiennent la chaîne `ng`.
3. Affiche les éléments filtrés en une liste d'éléments `` dans une liste ``.
4. Utilise ce composant dans `app` avec un tableau d'exemple et une chaîne de filtrage.

Passer un composant comme Content Projection

Si tu veux afficher dynamiquement un composant dans un autre

Avantage : Très utile pour créer des composants génériques (ex: **modals**, **cartes**, **layouts**).

list.component.html

```
<div>
  <ng-content></ng-content> <!-- Affichage du composant enfant -->
</div>
```

app.component.html

```
<app-list>
  <app-user-photo />
</app-list>
```

Passer un composant via ngComponentOutlet

Si tu veux afficher dynamiquement un composant dans un autre

Avantage : Très utile pour créer des composants génériques (ex: **modals, cartes, layouts**).

app.component.html



Uploaded using RayThis Extension

```
<ng-container *ngComponentOutlet="selectedComponent"></ng-container>
```

app.component.ts



Uploaded using RayThis Extension

```
import { CommonModule } from '@angular/common';
```



Uploaded using RayThis Extension

```
selectedComponent = UserPhotoComponent; // Choix du composant à afficher
```

Exercice

Exercice 1 : Passer un composant comme contenu avec `ng-content`

1. Crée un **composant CardComponent** qui contient un `<div>` avec une classe CSS `card`.
2. Ajoute `<ng-content>` dans `CardComponent` pour qu'il puisse contenir d'autres composants.
3. Dans `AppComponent`, affiche un `CardComponent` qui contient un composant avec du texte et un autre carte avec une image.

Exercice 2 :: Affichage dynamique d'un composant avec `ngComponentOutlet`

1. Crée **trois composants enfants** avec des textes différents.
2. Dans un **composant parent**, affiche les 3 composants.

Exercice 3 : Affichage dynamique d'un composant avec `ngComponentOutlet`

3. Crée **trois composants enfants** avec des textes différents.
4. Dans un **composant parent**, ajoute une liste déroulante (`<select>`) pour choisir quel composant afficher.
5. Passe le composant sélectionné en tant qu'`input()` et utilise `ngComponentOutlet` pour l'afficher.

Exercice

Exercice 4 : Mix des deux techniques

Crée un composant `ModalComponent` qui :

- Accepte un composant en `input()` et l'affiche avec `ngComponentOutlet`.
- Contient aussi un `<ng-content>` pour ajouter un composant avec du texte.

Dans `AppComponent`, affiche une **modale** qui contient :

- Un titre et une description passés en `ng-content`.
- Un composant dynamique choisi par l'utilisateur.

Bonus : Ajoute un bouton pour fermer la modale en modifiant une variable `isOpen`.

Exercice bonus : Affichage de composants avec une liste

1. Crée un composant `User` qui prend un input `name` et affiche un élément `<div>` avec le nom de l'utilisateur.
2. Crée un composant `UserList` qui affiche une liste de composant `User`.
3. Utilise ce composant dans `app` avec un tableau d'exemple d'utilisateurs.

4. Les événements et conditions

Evénements

- Un événements est une fonction qui est définie pour réagir à une action spécifique

Exemple avec une liste dynamique

```
...  
Uploaded using RayThis Extension  
  
users = [  
  { id: 1, name: 'Alice' },  
  { id: 2, name: 'Bob' }  
];  
  
addUser() {  
  const newId = this.users.length + 1;  
  this.users.push({ id: newId, name: 'User ' + newId });  
}  
  
removeUser(index: number) {  
  this.users.splice(index, 1);  
}
```

```
...  
Uploaded using RayThis Extension  
  
<button (click)="addUser()">Ajouter un utilisateur</button>  
<ul>  
  <li *ngFor="let user of users; let i = index">  
    {{ user.name }}  
    <button (click)="removeUser(i)">X</button>  
  </li>  
</ul>
```

Dans cet exemple, (`click`) est l'événement de clic qui déclenche la fonction `addUser` et `removeUser`.

- Quand utiliser un signal pour ajouter et supprimer des éléments dynamiquement dans un tableau ?

Angular détecte les changements dans un tableau **si la référence du tableau change !!**

Si tu modifies directement le contenu du tableau **sans changer sa référence**, Angular peut **ne pas détecter** le changement.

```
this.users.push({ id: 4, name: 'Eve' }); // La référence ne change pas !
```

```
this.users = [...this.users, { id: 4, name: 'Eve' }]; // Nouvelle référence !
```

Dans le cas où on modifie une valeur déjà présente alors il faut déclarer le tableau en tant que signal

Événements

Voici quelques événements couramment utilisés en Angular :

- click : déclenché lorsque l'utilisateur clique sur un élément.
- input : déclenché lorsque la valeur d'un champ d'entrée change (utile pour les formulaires).
- submit : déclenché lors de la soumission d'un formulaire.
- mouseenter / mouseleave : déclenchés lorsque la souris entre/sort d'un élément.
- keydown / keyup : déclenchés lorsqu'une touche est enfoncee/relâchée.

Exercice

Exercice 1 : Gestion d'un événement `click`

1. Crée un composant `Button` qui affiche un bouton avec le texte "Cliquez-moi".
2. Ajoute un événement `click` pour afficher un `alert` avec un message personnalisé lorsqu'on clique sur le bouton.

Exercice 2 : Événement `mouseover`

1. Crée un composant `HoverBox` qui affiche un `div` avec le texte "Survolez-moi".
2. Ajoute un événement `mouseover` pour changer la couleur du texte lorsque la souris passe dessus.

Exercice 3 : Événement `mouseleave`

3. Ajoute un événement `mouseleave` pour changer la couleur du texte lorsque la souris sort de la `div` avec le texte "Survolez-moi" ..

Exercice 4 : Gestion des événements `onkeyup` et `onkeydown`

1. Crée un composant `KeyTracker` qui affiche un champ de texte `<input>`.
2. Ajoute deux événements :
 - o Un événement `onkeydown` qui affiche un message dans la console indiquant quelle touche a été enfoncée.
 - o Un événement `onkeyup` qui affiche un message indiquant que la touche a été relâchée.

Exercice

Exercice Bonus : Cherche et utilise d'autres événements !

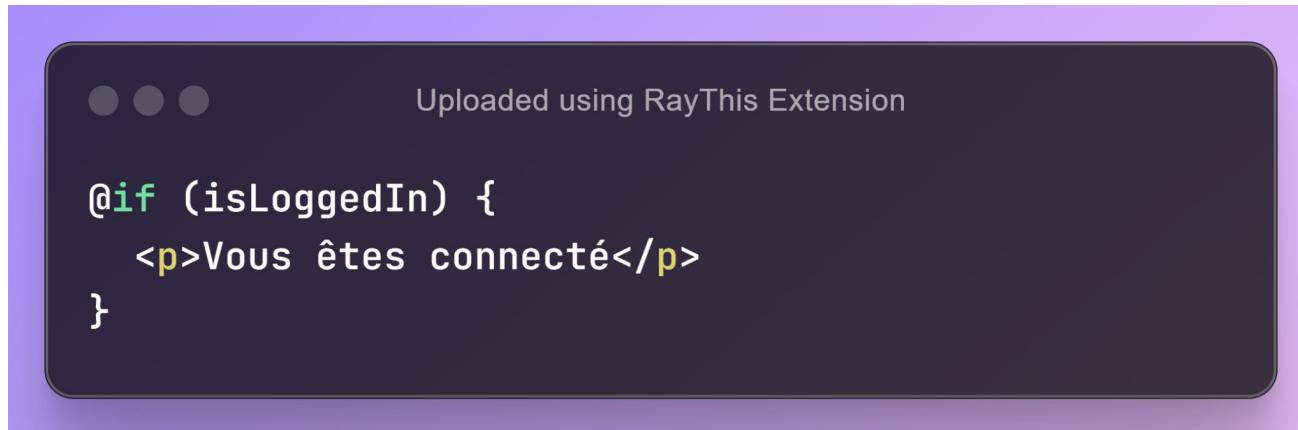
1. Crée un composant `NewEvent`.
2. Ajoute des événements de ton choix et utilise les de façon créative ! (ex : `resize`, `scroll...`)

Conditions

- Opérateur logique @if
- Dans cet exemple :

Si `isLoggedIn` est `true`, le paragraphe "Utilisateur identifié" sera affiché.

Si `isLoggedIn` est `false`, le paragraphe ne sera pas rendu.



Conditions

- Opérateur logique @else



Uploaded using RayThis Extension

```
@if (isLoggedIn) {  
    <p>Vous êtes connecté</p>  
} @else {  
    <p>Vous n'êtes pas connecté</p>  
}
```

Conditions

- Opérateur logique @switch @case
- Dans cet exemple :

Ce qui est affiché dépend de la valeur de *condition* défini dans le fichier ts.

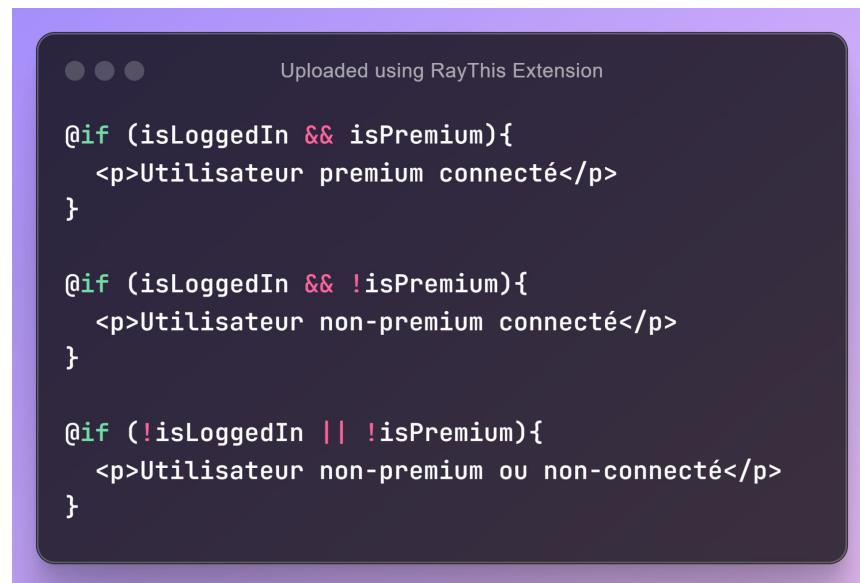


The screenshot shows a code editor window with a dark theme. At the top, there are three small circular icons. To the right of them, the text "Uploaded using RayThis Extension" is displayed. The main area contains the following TypeScript code:

```
@switch (condition) {
  @case ("a") {
    <p>Case A.</p>
  }
  @case ("b") {
    Case B.
  }
  @default {
    Default case.
  }
}
```

Conditions - &&, || et ! dans les expressions Angular

- && => et logique
- ! => inversion
- || => ou logique



Uploaded using RayThis Extension

```
@if (isLoggedIn && isPremium){  
    <p>Utilisateur premium connecté</p>  
}  
  
@if (isLoggedIn && !isPremium){  
    <p>Utilisateur non-premium connecté</p>  
}  
  
@if (!isLoggedIn || !isPremium){  
    <p>Utilisateur non-premium ou non-connecté</p>  
}
```

Exercice

Exercice 1 : Gestion de connexion avec @if

1. Dans un nouveau composant, crée une variable `isLoggedIn` dans le `component.ts`.
2. Affiche un message "**Bienvenue, utilisateur !**" si l'utilisateur est connecté.
3. Sinon, affiche "**Veuillez vous connecter**".
4. Ajoute un **bouton** qui permet d'inverser l'état de connexion.

Exercice 2 : Utilisation de @switch

Utilise `@switch` pour afficher :

- "Bienvenue, Admin !"
 - "Bonjour, Éditeur"
 - "Bonjour, Utilisateur"
 - Un message **par défaut** si le rôle est inconnu.
1. Ajoute un **menu déroulant (<select>)** pour changer le rôle dynamiquement.

Exercice 3: Combiner plusieurs conditions

1. Crée trois variables :
 - `isLoggedIn (true/false)`
 - `hasSubscription (true/false)`
 - `isAdmin (true/false)`
2. Affiche les messages suivants avec `@if` et `&&` / `||` :
 - Si l'utilisateur est connecté et abonné → "Accès au contenu premium"
 - Si l'utilisateur est admin → "Accès admin activé"
 - Si l'utilisateur n'est pas connecté → "Veuillez vous connecter"
3. Ajoute des boutons pour modifier ces variables.

Bonus : Ajoute du CSS sur la page

5. Formulaire

Formulaire avec ngModel

```
••• Uploaded using RayThis Extension  
  
nom: string = ''; // Variable liée au champ input
```

```
••• Uploaded using RayThis Extension  
  
<input type="text" [(ngModel)]="nom" placeholder="Entrez votre nom">  
<p>Bonjour, {{ nom }} !</p>
```

Formulaire avec ngModel

```
Uploaded using RayThis Extension

<form (ngSubmit)="onSubmit()">
    <input type="text" [(ngModel)]="utilisateur.nom" name="nom" placeholder="Nom" required>
    <input type="email" [(ngModel)]="utilisateur.email" name="email" placeholder="Email" required>
    <p>Nom : {{ utilisateur.nom }}</p>
    <p>Email : {{ utilisateur.email }}</p>

    <label>
        <input type="radio" [(ngModel)]="genre" name="genre" value="Homme"> Homme
    </label>
    <label>
        <input type="radio" [(ngModel)]="genre" name="genre" value="Femme"> Femme
    </label>
    <p>Genre sélectionné : {{ genre }}</p>

    <select [(ngModel)]="pays" name="pays">
        <option value="France">France</option>
        <option value="Belgique">Belgique</option>
        <option value="Suisse">Suisse</option>
    </select>
    <p>Pays sélectionné : {{ pays }}</p>

    <button type="submit">Envoyer</button>
</form>
```

```
Uploaded using RayThis Extension

onSubmit() {
    console.log('Formulaire soumis !', this.utilisateur, this.genre, this.pays);
    alert(`Nom: ${this.utilisateur.nom} \nEmail: ${this.utilisateur.email}`);
}
```

Exercice

Exercice 1 : Création de formulaire

1. Crée un formulaire de contact avec :
 - Un champ **Nom** (obligatoire et minimum 3 caractères).
 - Un champ **Email** (obligatoire et format valide).
 - Un champ **Message** (obligatoire et minimum 10 caractères).
 - Un champ **Langue préférée** (avec une liste déroulante pour sélectionner entre **Français**, **Anglais**, ou **Espagnol**).
2. Affiche un message de confirmation si le formulaire est soumis avec succès.
3. Affiche des messages d'erreur si un champ est mal rempli.

Exercice Bonus :

1. Ajout de validations personnalisées
2. Affichage d'une prévisualisation du message
3. Ajout d'un bouton de réinitialisation
4. Ajout de champs supplémentaires
5. Affichage des erreurs en temps réel
6. Limite de caractères sur les champs
7. Ajout de CSS

Formulaire

```
● ● ● Uploaded using RayThis Extension

export class FormulaireComponent {
  formulaire: FormGroup;

  constructor(private fb: FormBuilder) {
    // Création du formulaire avec des contrôles
    this.formulaire = this.fb.group({
      nom: ['', [Validators.required, Validators.minLength(3)]],
      email: ['', [Validators.required, Validators.email]]
    });
  }

  // Méthode pour soumettre le formulaire
  onSubmit(): void {
    if (this.formulaire.valid) {
      console.log(this.formulaire.value);
    } else {
      console.log("Formulaire invalide !");
    }
  }
}
```

Exercice

Exercice 1 : Création de formulaire

1. Crée un formulaire de contact avec :
 - Un champ **Nom** (obligatoire et minimum 3 caractères).
 - Un champ **Email** (obligatoire et format valide).
 - Un champ **Message** (obligatoire et minimum 10 caractères).
 - Un champ **Langue préférée** (avec une liste déroulante pour sélectionner entre **Français**, **Anglais**, ou **Espagnol**).
2. Affiche un message de confirmation si le formulaire est soumis avec succès.
3. Affiche des messages d'erreur si un champ est mal rempli.

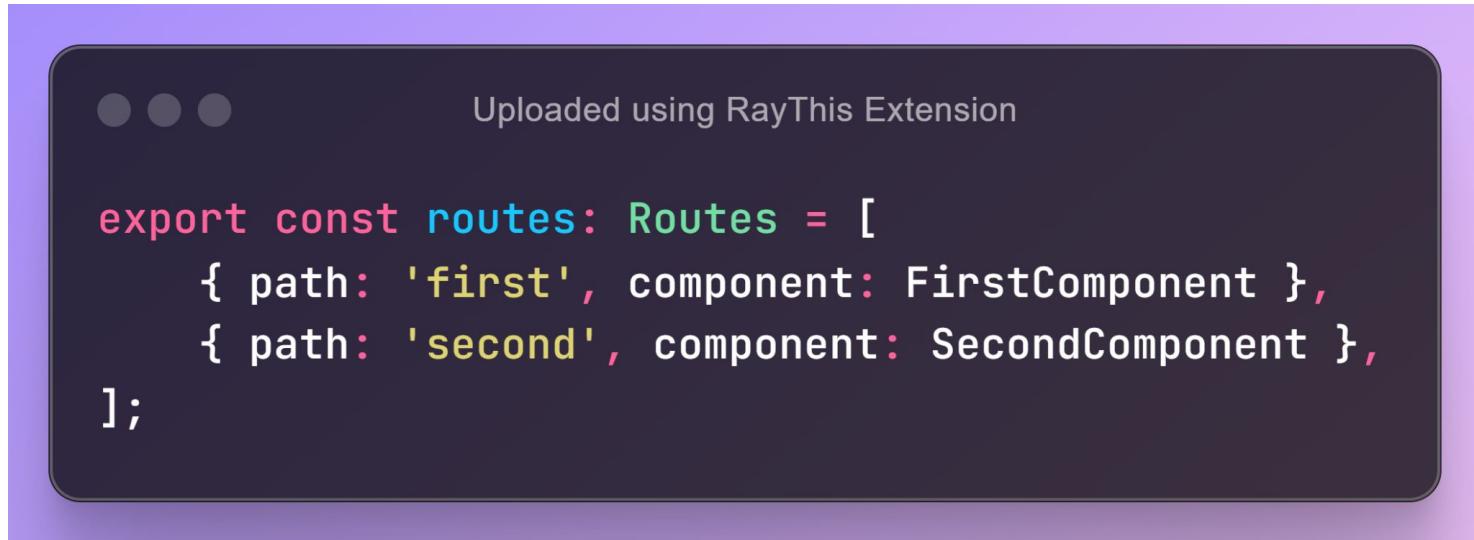
Exercice Bonus :

1. Ajout de validations personnalisées
2. Affichage d'une prévisualisation du message
3. Ajout d'un bouton de réinitialisation
4. Ajout de champs supplémentaires
5. Affichage des erreurs en temps réel
6. Limite de caractères sur les champs
7. Ajout de CSS

6. Le routing et la navigation

Les routes

- app.routes.ts



Uploaded using RayThis Extension

```
export const routes: Routes = [
  { path: 'first', component: FirstComponent },
  { path: 'second', component: SecondComponent },
];
```

Liens de Navigation

```
<h1>Angular Router App</h1>
<nav>
  <ul>
    <li><a routerLink="/first" routerLinkActive="active" ariaCurrentWhenActive="page">First Component</a></li>
    <li><a routerLink="/second" routerLinkActive="active" ariaCurrentWhenActive="page">Second Component</a></li>
  </ul>
</nav>
<router-outlet />
```

Les paramètres d'URL

- `:id` est un **paramètre d'URL** dynamique. Dans votre composant `First`, vous pouvez ensuite y accéder.

Uploaded using RayThis Extension

```
{ path: 'first/:id', component: FirstComponent },
```

Uploaded using RayThis Extension

```
userId: string | null = null;

constructor(private route: ActivatedRoute) {
  this.userId = this.route.snapshot.paramMap.get('id');
}
```

localhost:4200/first/51

URL first 51

Les routes imbriquées

- Lorsque l'utilisateur navigue vers `/second`, le composant `Second` est affiché.
- Lorsque l'utilisateur clique sur le lien "Afficher third", il est redirigé vers `/second/third`, et la route imbriquée `ThirdComponent` est rendue à l'intérieur du composant `User` grâce à l'élément `<router-outlet></router-outlet>`



Uploaded using RayThis Extension

```
{ path: 'second', component: SecondComponent, children : [  
    {path: 'third', component: ThirdComponent}  
] },
```



Uploaded using RayThis Extension

```
<nav>  
    <a routerLink="third">Afficher third</a>  
</nav>  
  
<router-outlet></router-outlet>
```

Exercice

Exercice 1 : Création de routes simples

1. Crée un composant `Home` qui affiche un message "Accueil".
2. Crée un composant `About` qui affiche un message "À propos".
3. Crée deux routes : `/home` pour le composant `Home` et `/about` pour le composant `About`.
4. Affiche les composants en fonction de la route actuelle.

Exercice 2 : Routes avec paramètres

1. Crée un composant `UserComponent` qui affiche l'ID de l'utilisateur.
2. Récupère l'ID dans `UserComponent` à l'aide d'`ActivatedRoute`.
3. Ajoute un lien dynamique vers un utilisateur spécifique (ex : `/user/42`).

Exercice 3 : Routes imbriquées

1. Crée une route parent `/dashboard` avec un composant `Dashboard`.
2. Ajoute deux routes enfants sous `/dashboard` :
 - `/dashboard/stats` → `StatsComponent`
 - `/dashboard/settings` → `SettingsComponent`
3. Dans `DashboardComponent`, ajoute un `<router-outlet>` pour afficher les enfants.

Exercice

Exercice 4 : Routes avec paramètres + Routes imbriquées

1. **Créer une route dynamique pour les utilisateurs** affichant un composant `UserComponent`.
2. **Ajouter des routes imbriquées pour chaque utilisateur**
 - `user/:id/info` → Composant `UserInfoComponent` (affiche les infos de l'utilisateur).
 - `user/:id/settings` → Composant `UserSettingComponent` (affiche ses settings).
3. **Afficher les sous-pages dans `UserComponent`**

Exercice Bonus : Gestion des erreurs et redirections

1. Crée une route "404" pour les pages non trouvées :
 - Crée un composant fonctionnel `NotFound` qui affiche un message "Page non trouvée".
 - Affiche le composant `NotFound` quand aucune route ne correspond.
2. Mise en place d'une redirection conditionnelle :
 - Crée un composant fonctionnel `Login` qui affiche un formulaire de connexion (pas besoin de le rendre fonctionnel).
 - Crée une route `/profile` qui nécessite que l'utilisateur soit "connecté" pour y accéder.
 - Si l'utilisateur n'est pas connecté (simule cela avec une simple variable `isLoggedIn`), redirige-le vers la page `/login`
 - Si l'utilisateur est connecté, affiche une page "Profil de l'utilisateur connecté".

7. Lifecycle

Lifecycle

- Permettent aux composants de gérer des événements tout au long du cycle de vie.

Lifecycle

- `ngOnInit()` : Initialiser des données (ex. récupération de données depuis une API).
- `ngOnChanges()` : Réagir aux changements d'inputs (ex. modification des données envoyées).
- `ngDoCheck()` : Déetecter manuellement des changements complexes.
- `ngAfterViewInit()` : Manipuler le DOM après que la vue ait été initialisée (ex. donner le focus à un champ).
- `ngAfterViewChecked()` : Effectuer des actions après chaque mise à jour de la vue.
- `ngOnDestroy()` : Effectuer un nettoyage avant la destruction du composant (ex. annulation d'abonnements ou de timers).

Lifecycle

- **ngOnInit()** : La méthode ngOnInit s'exécute après qu'Angular ait initialisé tous les entrées du composant avec leurs valeurs initiales. Le ngOnInit d'un composant s'exécute exactement une fois. Utile notamment pour les données qui ont besoin d'appel API ou d'un service.
- La méthode **ngOnChanges** s'exécute après toute modification des inputs du composant pour ajouter de la réactivité
 - Agit de la même façon qu'un signal, mais celui ci est utilisé pour la réactivité interne tandis que onChange est utile pour les valeurs passées de parent à enfant.
 - **input() + ngOnChanges** Utile quand un **parent envoie des données** à un composant enfant.
 - **signal()** Idéal pour la **gestion d'état locale**, sans dépendre du système de détection de changement d'Angular.

Si ton but est d'améliorer la performance et éviter la surcharge de **ngOnChanges**, les **signals sont la meilleure approche**

onChange

Uploaded using RayThis Extension

```
export class PriceDisplayComponent implements OnChanges {
    price = input(0); // Prix reçu du parent
    previousPrice: number | null = null; // Stocker la valeur précédente

    ngOnChanges(changes: SimpleChanges): void {
        if (changes['price']) { // Vérifie si le prix a changé
            this.previousPrice = changes['price'].previousValue;
            console.log(`Ancien prix : ${this.previousPrice}, Nouveau prix :
${changes['price'].currentValue}`);
        }
    }
}
```

Uploaded using RayThis Extension

```
<h1>Affichage du prix</h1>
<app-price-display [price]="productPrice"></app-price-display>
<button (click)="changePrice()">Changer le prix</button>
```

Exercice

Exercice 1 : Utilisation de onInit

1. Ajoute un onInit qui instancie des valeurs à un objet `user` contenant les champs name, age et address .

Exercice 2 : Utilisation de onChange

1. Crée un composant `StockDisplayComponent` qui reçoit la quantité d'un produit et combien on ajoute ou enlève au stock, via `input()`.
2. Utilise `ngOnChanges` pour :
 - Afficher la quantité actuelle.
 - Afficher l'ancienne quantité lorsqu'un changement est détecté.
 - Indiquer si le stock est en baisse ou en augmentation.
3. Crée un composant parent qui met à jour le stock via des boutons.
4. Ajoute une erreur si le stock va dans les négatifs

Exercice 3:

1. Crée un composant `OrderStatusComponent` qui reçoit un statut de commande via input
2. Affiche l'ancien et le nouveau statut lorsqu'un changement est détecté.
3. Crée un composant parent qui change le statut via des boutons
4. Désactive un bouton si le statut correspondant est déjà actif.

8. Les actions, les reducers et le store

```
ng add @ngrx/store
```

```
npm install @ngrx/store @ngrx/effects @ngrx/entity @ngrx/router-store
```

- **Actions** : Objets décrivant un événement. Chaque action a un *type* (qui identifie l'action) et, souvent, une charge utile (*payload*) avec des données supplémentaires.
- **Reducers** : Ce sont des fonctions pures qui reçoivent l'état *actuel* et une *action*, et renvoient un *nouvel état modifié*. Ils déterminent comment l'état de l'application change en réponse à une action donnée.
- **Store** : Le store est l'objet central qui contient l'état global de l'application. Il gère l'état, permet l'envoi des actions, et s'assure que les composants sont informés des changements d'état.

ngRx - Définir les Actions



Uploaded using RayThis Extension

```
import { createAction, props } from '@ngrx/store';

export const increment = createAction('[Counter] Increment');
export const decrement = createAction('[Counter] Decrement');
export const reset = createAction('[Counter] Reset');
```

ngRx - Définir le Reducer

```
••• Uploaded using RayThis Extension

import { createReducer, on } from '@ngrx/store';
import {increment, decrement, reset} from './counter.actions';

export const initialState = 0;

const _counterReducer = createReducer(
  initialState,
  on(increment, (state) => state + 1),
  on(decrement, (state) => state - 1),
  on(reset, () => initialState)
);

export function counterReducer(state: any, action: any) {
  return _counterReducer(state, action);
}
```

ngRx - Provider dans main.ts



Uploaded using RayThis Extension

```
bootstrapApplication(AppComponent, {
  providers: [
    provideStore({ count: counterReducer }),
    provideEffects([]),
    ...appConfig.providers
  ]
}).catch((err) => console.error(err));
```

ngRx - Utiliser le store

Uploaded using RayThis Extension

```
constructor(private store: Store<AppState>) {
  this.count$ = store.select('count');
}
```

TypeScript

Uploaded using RayThis Extension

```
count$: Observable<number>;  
  
increment() {  
  this.store.dispatch(increment());  
}  
  
decrement() {  
  this.store.dispatch(decrement());  
}  
  
reset() {  
  this.store.dispatch(reset());  
}
```

HTML

Uploaded using RayThis Extension

```
<h1>Counter: {{ count$ | async }}</h1>
```

Exercice

Exercice 1 : Configuration de ngRx

1. Configure un store pour gérer un compteur.

Consignes :

- Crée les actions `increment`, `decrement`, `reset`, `incrementBy10` et `divideBy2`.
- Configure le store
- Crée un composant `Counter` qui affiche le compteur et des boutons pour tester les méthodes du compteur.

Exercice 2 : Gestion de stock

Configure un store pour gérer les stocks.

Consignes :

- Crée les actions `addToStock` et `removeFromStock`
- Configure le store
- Crée un composant pour ajouter, supprimer et afficher le stock.

Exercice

Exercice 3 : Création d'actions avec paramètres

1. Crée un store pour gérer des tâches.
2. Les tâches ont toutes un titre
3. Mettre en place un composant pour interagir avec le store (afficher les tâches et pouvoir ajouter et supprimer des actions)
4. Bonus : Rajoute des paramètres pertinents pour des tâches (ex: une date deadline, une personne assignée, etc...)

Bonus : Fais du CSS pour la mise en page

8. Tests

Les tests

- Les **tests unitaires** sont essentiels pour garantir la qualité et la *stabilité* de votre code.
- Ils permettent de vérifier qu'une petite portion de code (une unité, comme une fonction ou un composant) fonctionne correctement en *isolation*.

Test

```
ng test --no-watch --code-coverage
```

```
===== Coverage summary =====
Statements : 43.2% ( 35/81 )
Branches   : 0% ( 0/3 )
Functions   : 9.09% ( 2/22 )
Lines       : 39.18% ( 29/74 )
=====
```

Concepts clés

- **describe** : Permet de regrouper des tests liés dans un bloc, souvent utilisé pour organiser les tests d'un composant ou d'une fonction.
- **it** : Définit un cas de test. Chaque cas de test doit vérifier un aspect spécifique du comportement.
- **expect** : Une assertion pour vérifier que le résultat correspond à ce qui est attendu.

Test

- Exemple simple :
- npm run test

```
export class AppComponent {
    title = 'ProjetTest';

    add = (a: number, b: number) => a + b;

    majuscule = (str: string) => {
        return str.toUpperCase();
    }
}
```

```
describe('AppComponent', () => {
    let component: AppComponent;
    let fixture: ComponentFixture<AppComponent>;

    beforeEach(async () => {
        await TestBed.configureTestingModule({
            imports: [AppComponent],
        }).compileComponents();
    });

    beforeEach(() => {
        fixture = TestBed.createComponent(AppComponent);
        component = fixture.componentInstance;
        fixture.detectChanges();
    });

    it('devrait retourner la somme de deux nombres', () => {
        expect(component.add(2, 3)).toBe(5);
    });
});
```

Cypress

```
npm install cypress --save-dev
```

```
npx cypress open
```



Uploaded using RayThis Extension

```
import { defineConfig } from "cypress";

export default defineConfig({
  e2e: {
    baseUrl: "http://localhost:4200",
    setupNodeEvents(on, config) {
      // implement node event listeners here
    },
  },
});
```

Cypress

```
● ● ● Uploaded using RayThis Extension

describe("Page d'accueil", () => {
  it("devrait afficher le titre", () => {
    cy.visit("/"); // Va sur la page d'accueil
    cy.contains("Angular Router App").should("be.visible"); // Vérifie que le texte est
présent
  });
});
```

(Run Finished)

Spec	Tests	Passing	Failing	Pending	Skipped
✓ price.cy.ts	667ms	1	1	-	-
✓ All specs passed!	667ms	1	1	-	-

Cypress

```
● ● ● Uploaded using RayThis Extension

describe("Compteur", () => {
  it("devrait incrémenter le compteur au clic", () => {
    cy.visit("/");
    cy.get('button[id="increment-btn"]').click(); // Clic sur le bouton
    cy.get("#counter").should("have.text", "Counter: 1"); // Vérifie que le compteur est à
    1
  });
});
```

```
● ● ● Uploaded using RayThis Extension

<h1 id="counter">Counter: {{ count$ | async }}</h1>
<button id="increment-btn" (click)="increment()">Increment</button>
```

Spec	Tests	Passing	Failing	Pending	Skipped
✓ price.cy.ts	00:01	2	2	-	-
✓ All specs passed!	00:01	2	2	-	-

Exercice

Exercice 1 : TestSimple

1. Fonction de Modification de Texte :

```
majuscule(str) {  
    return str.toUpperCase();  
}
```

Exercice 2 : Fonction filterAndSortUsers :

- **Description** : Filtre les utilisateurs pour ne conserver que ceux ayant un âge supérieur ou égal à `minAge`, puis trie les utilisateurs restants par nom en ordre alphabétique.
- **Paramètres** :
 - `users` : Un tableau d'objets utilisateurs, chacun avec un `name` et un `age`.
 - `minAge` : L'âge minimum requis pour qu'un utilisateur soit inclus dans le résultat.
- **Retour** : Un tableau d'objets utilisateurs filtrés et triés.

Tests :

- **Premier test** : Vérifie que les utilisateurs sont correctement filtrés par âge et triés par nom.
- **Deuxième test** : Vérifie le comportement lorsque aucun utilisateur ne correspond au critère d'âge (doit retourner une liste vide).

```
/** * Filtre les utilisateurs par âge et les trie par nom.  
 * @param {Array} users - Liste d'utilisateurs, chaque utilisateur étant un objet avec  
`name` et `age`.  
 * @param {number} minAge - Âge minimum pour filtrer les utilisateurs.  
 * @returns {Array} Liste d'utilisateurs filtrés et triés.
```

```
filterAndSortUsers(users, minAge) {  
    return users .filter(user => user.age >= minAge) .sort((a, b) => {  
        a.name.localeCompare(b.name);  
    });  
}
```

Exercice

Exercice 1 : Test d'un composant simple

1. Crée un composant **Greeting** qui affiche un message de bienvenue avec le nom passé en input.
2. Écris un test pour ce composant pour vérifier que le message de bienvenue est correctement affiché.

Exercice 2 : Test d'un composant avec gestion d'état

1. Crée un composant **Counter** avec un bouton pour incrémenter, décrémenter et reset un compteur.
2. Écris un test pour vérifier que le compteur augmente correctement lorsque les boutons sont cliqués.

Exercice Bonus : Test d'un formulaire avec validation

1. Crée un composant fonctionnel **Form** avec un champ de texte et un bouton. Le formulaire doit afficher un message d'erreur si le champ est vide lorsque le bouton est cliqué.
2. Écris un test pour vérifier que le message d'erreur est affiché correctement lorsque le formulaire est soumis avec un champ vide.

9. Directives

Type de directives

Composants

Utilisés avec un template. Ce type de directive est le plus courant.

Directives d'attribut

Modifient l'apparence ou le comportement d'un élément, d'un composant ou d'une autre directive.

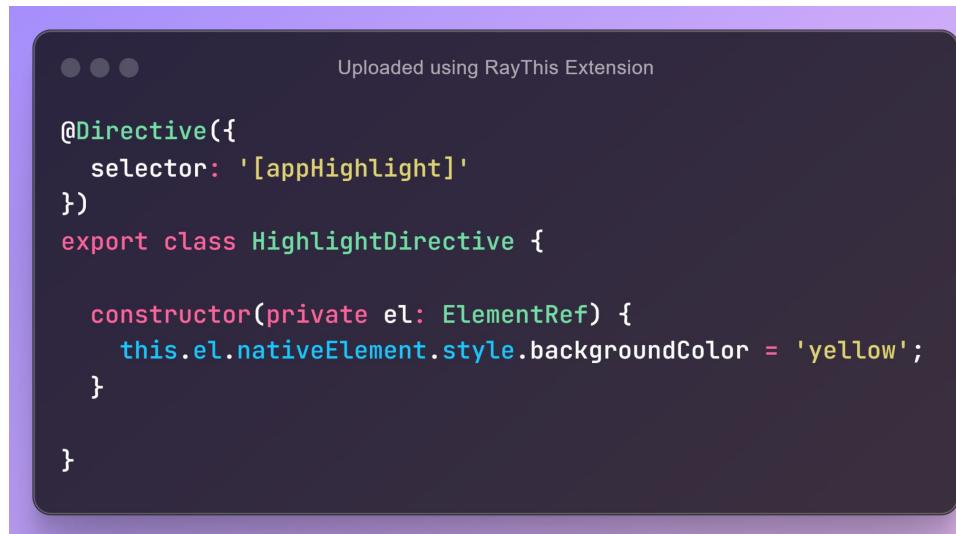
Ex : NgModel

Directives structurelles

Modifient la structure du DOM en ajoutant ou en supprimant des éléments DOM.

Construire une directive d'attribut

ng generate directive highlight



```
Uploaded using RayThis Extension

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {

  constructor(private el: ElementRef) {
    this.el.nativeElement.style.backgroundColor = 'yellow';
  }

}
```

Appliquer une directive d'attribut



Uploaded using RayThis Extension

```
<h1>My First Attribute Directive</h1>
<p appHighlight>Highlight me!</p>
```

Directive avec Input et EventListener

```
... Uploaded using RayThis Extension

export class HighlightDirective {

    constructor(private el: ElementRef) {}

    @Input() colorInput = '';

    @HostListener('mouseenter') onMouseEnter() {
        this.highlight(this.colorInput || 'red');
    }
    @HostListener('mouseleave') onMouseLeave() {
        this.highlight('');
    }
    private highlight(color: string) {
        this.el.nativeElement.style.backgroundColor = color;
    }
}
```

```
... Uploaded using RayThis Extension

<h1>My First Attribute Directive</h1>
<p appHighlight>Highlight me!</p>
<p appHighlight [colorInput]="'yellow'">Highlight me!</p>
```

Exercice

Exercice 1 : Création d'une directive

1. Crée une directive `changeColor` qui change la couleur du texte d'un élément.

Exercice 2 : Directive avec un évènement

2. Crée une directive `hideElement` qui masque l'élément lorsque l'on clique dessus

Exercice 3 : Directive avec Input

1. Crée une directive `border` qui ajoute une bordure avec une couleur et une épaisseur configurables par l'utilisateur.

10. Appels API et Service

Appel API

ng generate service api

```
● ● ● Uploaded using RayThis Extension

import { provideHttpClient } from '@angular/common/http';

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

```
● ● ● Uploaded using RayThis Extension

import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root',
})
export class ApiService {
  private apiUrl = 'https://jsonplaceholder.typicode.com/posts'; // URL de l'API

  constructor(private http: HttpClient) {}

  getPosts(): Observable<any> {
    return this.http.get<any>(this.apiUrl);
  }
}
```

```
● ● ● Uploaded using RayThis Extension

posts: any[] = [];

ngOnInit(): void {
  this.apiService.getPosts().subscribe(
    next: (data) => this.posts = data,
    error: (e) => console.error('Erreur lors de la récupération des posts', e),
    complete: () => console.log("Complete")
  );
}
```

Exercice

Exercice 1 : Afficher une liste de Pokémons

Endpoint à utiliser : <https://pokeapi.co/api/v2/pokemon?limit=20>

1. Crée un **service** `PokeService` qui fait une requête HTTP pour récupérer les 20 premiers Pokémons.
2. Crée un **composant** `PokemonListComponent` qui affiche les noms et images des Pokémons dans une liste.
3. Bonus : Ajoute un bouton "**Charger plus**" pour récupérer 20 Pokémons supplémentaires (gestion de la pagination).

Exercice 2 : Filtrer les Pokémons par type

Objectif : Ajouter un filtre permettant d'afficher uniquement les Pokémons d'un certain type.

Endpoint à utiliser : <https://pokeapi.co/api/v2/type/{type}>

1. Ajoute un **menu déroulant** contenant les types de Pokémons (Fire, Water, Grass...).
2. Lorsqu'un type est sélectionné, affiche uniquement les Pokémons correspondants.

Questions / réponses

- Revenons sur les questions hors plan de cours que vous m'avez posé durant la formation pour y répondre

Votre formateur

Clément Hamon

clement.hamon35@gmail.com

<https://www.linkedin.com/in/clément-hamon-135485209/>

Merci d'avoir suivi cette formation

!

Annexe TP validation des acquis

1. Les principes de base du framework Angular

Consigne :

Créez les composants de base de votre application Jira:

- Un composant **Board** qui contiendra plusieurs **List**.
- Un composant **List** qui contiendra plusieurs **Task**.
- Chaque composant doit utiliser les input pour recevoir et afficher les données de tâches et listes.
- N'oubliez pas d'utiliser le rendu via le **ngFor** pour afficher plusieurs tâches dans une liste.

2. Gérer les événements et rendu conditionnel

Consigne :

Implémentez un système de gestion des événements et des rendus conditionnels :

- Faites un rendu conditionnel pour afficher un message "Aucune tâche" si une liste est vide.
- Les gestions d'événements seront utiles plus tard lors de l'implémentation des formulaires d'ajout de tâches. Vous les utiliserez à ce moment-là.

Annexe TP validation des acquis

3. Signals

Consigne :

Utilisez des signals pour gérer l'état des composants :

- Modifiez la gestion de l'état des composants (`Task`, `List`, ...).

Utilisez un effect pour afficher un message lorsqu'une liste est modifiée.

4. Formulaire

- Ajoutez un formulaire pour ajouter des tâches à une liste, utiliser `FormGroup` ou `ngModel` pour gérer les champs de saisie.

5. Navigation

Consigne :

Ajoutez une navigation entre plusieurs vues dans l'application :

- Créez au moins deux pages : une page d'accueil `Home` avec un aperçu des boards présents avec `Board` et une page `BoardDetails` pour afficher un seul board.
- Implémentez le routage pour naviguer entre les boards.
- Utilisez les paramètres de route pour identifier quel board est affiché !

Annexe TP validation des acquis

6. Le concept de store

Consigne :

Intégrez ngRx pour gérer l'état global de l'application :

- Créez un store pour gérer les données des boards, des listes et des tâches.
- Utilisez des actions et des reducers pour ajouter des tâches à l'état global.
- Configurez **Provider** pour que tous les composants puissent accéder au store.

7. Les tests

Consigne :

Ajoutez des tests unitaires à votre application :.

- Écrivez des tests pour vérifier que les reducers de Redux fonctionnent correctement.
- Testez le fichier **TaskForm** (**formulaire pour ajouter une tâche**).

Annexe TP validation des acquis

Bonus

- Faites du CSS
- Faites en sorte de pouvoir ajouter des board et des listes de façon dynamique
- Mettez en place une `sauvegarde des données dans le localStorage` afin que les boards et tâches persistent après un rafraîchissement.
- Ajoutez un système de `drag-and-drop` (bibliothèque externe ou un événement `drag/drop` pur en JS).
- Implémentez une fonctionnalité d'`édition` des tâches (double clic pour modifier le titre d'une tâche).
- Utilisez des `Promesses` ou `async/await` pour simuler une requête API qui charge les listes initiales.
- Implémentez une redirection automatique après la création d'un nouveau board (par exemple, vers la page du board créé).