# Objectifs de la séance

## Date: 26/11/2025

À la fin de cette séance, les apprenants seront capables de:

- Comprendre la programmation **asynchrone et réactive** dans une application Angular.
- Mettre en œuvre un **service de données** capable d'émettre et d'actualiser des flux.
- Afficher et actualiser ces données dans la vue à l'aide d'un mécanisme réactif.

# D’une application statique à réactive

Jusqu’ici, *TaskBoard Pro* affiche des pages et des composants.

- Les données sont **fixes**, le contenu ne change pas sans rechargement.
- Objectif : rendre l’application **vivante** — qu’elle s’adapte aux **données qui évoluent**.

« Nous allons apprendre à faire réagir Angular aux changements de données, sans recharger la page. »

# La programmation réactive

**Définition :**

- La programmation réactive est un **paradigme de développement** dans lequel un programme **réagit automatiquement** aux changements de données ou d'événements au lieu de les interroger activement.

- En d'autres termes, le code ne "demande" plus la donnée : il **s'abonne** à un **flux** de valeurs et se met à jour **dès qu'une nouvelle valeur est émise**.

Ex : notification sur nos téléphones. C'est le système réactif qui nous *prévient* dès qu'une nouvelle donnée est disponible.

# Une application réactive Angular

S'appuie sur un modèle basé sur trois rôles :

- **Le service** : il *fournit* les données sous forme de flux (par exemple, des tâches ou des messages).
- **Le composant** : il *s'abonne* à ces données, pour être notifié à chaque changement.
- **Le template (vue)** : il *affiche* automatiquement la dernière valeur reçue grâce au **binding asynchrone** (le | async).

💡 C'est une approche *déclarative* : on ne dit pas "fais ceci quand tu reçois les données", on dit "affiche cette donnée, et Angular s'occupera de la mettre à jour quand elle changera".

# Créons un service

```
import { Injectable } from '@angular/core';
import { of } from 'rxjs';
import { delay } from 'rxjs/operators';

@Injectable({ providedIn: 'root' })
export class TaskService {
  private tasks = [
    { id: 1, title: 'Préparer le cours Angular' },
    { id: 2, title: 'Relire le module RxJS' },
    { id: 3, title: 'Corriger les TPs' },
  ];

  getTasks() {
    // Simule un appel asynchrone avec RxJS
    return of(this.tasks).pipe(delay(1000));
  }
}
```

INFORMATIQUE ET NUMÉRIQUE

GROUPE IGENSIA EDUCATION

ng generate  service core/services/task
ou ng g s core/services/task

# Injectons et utilisons le service

```html
<h2>Liste des tâches</h2>

@if (tasks$ | async; as tasks) {
    <ul>
        @for (t of tasks; track t.id) {
            <li> {{ t.title }}</li>
        }
    </ul>
} @else {
    <p>Chargement des tâches ...</p>

}
```

```typescript
import { Component } from '@angular/core';
import { AsyncPipe } from '@angular/common';
import { Task } from '../core/services/task';

@Component({
  selector: 'app-home',
  imports: [ AsyncPipe],
  templateUrl: './home.html',
  styleUrl: './home.css',
})
export class Home {

  tasks$!: ReturnType<Task['getTasks']>;

  constructor(private taskService: Task) {
    this.tasks$ = this.taskService.getTasks();
  }

  ngOnInit() {
    console.log('ngOnInit executé');
  }

}
```

INFORMATIQUE ET NUMÉRIQUE

GROUPE IGENSIA EDUCATION

“Le service met une seconde avant de renvoyer les tâches,mais pendant ce temps, l’application reste fluide :

elle affiche un message *Chargement…*, elle reste interactive, et Angular continue d’écouter le flux.

C’est ça, l’asynchrone.”

# L’application continue de vivre pendant l’attente

```
count = 0;
ngOnInit() {
 setInterval(() => {
  this.count++;
 }, 500);
}
<p>Temps écoulé : {{ count / 2 }} secondes</p>
```

# Modification avec Zoneless

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

@Component({
  selector: 'app-home',
})
export class Home {
  count = 0;
  private cdr = inject(ChangeDetectorRef);

  ngOnInit() {
    setInterval(() => {
      this.count++;
      this.cdr.markForCheck();   // on prévient Angular de rafraîchir
    }, 1000);
  }
}
```

# Arreter le compter

```
ngOnDestroy() {
    clearInterval(this.intervalId);
    console.log('Compteur stoppé ’);
 }
```

# Programmation réactive complète

Jusqu'ici, notre service émettait une seule fois les données, avec un délai.

C'est asynchrone, mais ce n'est pas encore réactif au sens complet.

Maintenant, on va faire en sorte que les données puissent changer, et qu'Angular mette la vue à jour automatiquement, sans qu'on relance quoi que ce soit.

# Parlons d'RxJS

**RxJS (Reactive Extensions for JavaScript)**est une bibliothèque utilisée par Angular pour gérer les **flux de données** et tout ce qui change dans le temps : les appels HTTP, les formulaires, le routing ou les interactions utilisateur.

Elle introduit une nouvelle structure de données appelée **Observable**, qui permet à une application :

- d'**écouter des événements** ou des données qui changent dans le temps,
- de **réagir automatiquement** quand une nouvelle valeur est émise,
- et de **chaîner des opérations** sur ces données (filtrer, transformer, combiner, etc.).

*En résumé : RxJS est le moteur de la réactivité d'Angular.*

# Comment Angular utilise RxJS

```
getTasks() {
  return of(this.tasks).pipe(delay(2000));
}
```

Explications :

- of(this.tasks) → crée un Observable contenant la liste des tâches
- .pipe(delay(2000)) → simule un appel HTTP (2 secondes)
- Le composant peut écouter ce flux → tasks$ = taskService.getTasks();
- Le template affiche automatiquement → {{ tasks$ | async }}

*Ici, RxJS nous permet de travailler avec des données qui arrivent dans le temps.*

# Vers un flux vivant

Jusqu'ici, nous avons vu qu'un flux asynchrone permettait à Angular de ne pas bloquer l'application. Mais notre service ne renvoie qu'une seule fois les données : il ne "vit" pas dans le temps.

Or, dans une application moderne, les données changent — on ajoute, on supprime, on met à jour. Pour gérer ces changements **sans recharger la page**, Angular s'appuie sur un outil de RxJS : **le BehaviorSubject**.

# Pourquoi un flux vivant

Jusqu'ici, notre service émettait les données **une seule fois** grâce à of(...).pipe(delay()).C'était asynchrone, mais le flux s'arrêtait dès la première émission.

Dans une vraie application, les données **évoluent dans le temps** :

- un utilisateur ajoute une tâche,
- une notification arrive,
- une donnée change côté serveur.

Nous avons donc besoin d'un flux **actif**, qui continue à émettre des valeurs à chaque changement.

C'est le rôle du **BehaviorSubject** dans RxJS :
il garde en mémoire la dernière valeur émise, et notifie **tous les abonnés** dès qu'une nouvelle valeur apparaît.

# Exemple avec BehaviorSubject

- BehaviorSubject garde **en mémoire la dernière version** de la liste des tâches.
- Les composants **écoutent** le flux via tasks$.
- Chaque fois qu'on ajoute une tâche avec addTask(), la méthode **émet une nouvelle valeur** (next()), ce qui met à jour la vue automatiquement.

```
import { Injectable } from '@angular/core';
import { BehaviorSubject } from 'rxjs';

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

  private tasks = [
    { id: 1, title: 'Préparer le cours Angular' },
    { id: 2, title: 'Relire le module RxJS' },
    { id: 3, title: 'Corriger les TPs' },
  ];

  private tasksSubject = new BehaviorSubject(this.tasks);
  tasks$ = this.tasksSubject.asObservable();

  addTask(title: string) {
    const newTask = { id: Date.now(), title };
    this.tasks = [...this.tasks, newTask];
    this.tasksSubject.next(this.tasks);
  }

}
```

INFORMATIQUE ET NUMÉRIQUE

GROUPE IGENSIA EDUCATION

# Modification de home.html

```html
<h2>Liste des tâches</h2>

<!-- Affichage réactif grâce à BehaviorSubject + async -->
@if (tasks$ | async; as tasks) {

  <ul>
    @for (t of tasks; track t.id) {
      <li>{{ t.title }}</li>
    }
  </ul>

} @else {

  <p>Chargement des tâches...</p>

}

<hr />

<h3>Actions</h3>

<!-- Ajouter une tâche -->
<button (click)="addTask('Nouvelle tâche')">
  Ajouter une tâche
</button>
```

INFORMATIQUE ET NUMÉRIQUE

GROUPE IGENSIA EDUCATION

# Modification de home.ts

```
export class Home {

  taskService = inject(Task);
  tasks$ = this.taskService.tasks$;

  addTask(title: string) {
    this.taskService.addTask(title);
  }

}
```

Le BehaviorSubject, c'est comme un carnet partagé :
le service note toujours la dernière version de la liste, et n'importe quel composant qui vient consulter le carnet, même plus tard,
voit tout de suite la dernière version, pas une vieille copie

INFORMATIQUE ET NUMÉRIQUE

GROUPE IGENSIA EDUCATION

# Rappel cycle de vie du flux

- Service → émet une nouvelle valeur (next())
- Composant → reçoit automatiquement la mise à jour
- Template → se ré-affiche via | async

Le composant ne gère plus les données lui-même : il écoute un flux. À chaque mise à jour dans le service, Angular détecte et rafraîchit la vue.

# Synthèse

Dans Angular, le composant ne pilote plus les données. Il les **observe**. Et c'est RxJS, via BehaviorSubject, qui garantit la mise à jour automatique du rendu.

"En entreprise, cette approche permet de mieux séparer les responsabilités : les services gèrent les données, les composants gèrent la présentation."

# TP fil rouge

**1. Crée une nouvelle branche Git**

Sequence-2 Mise en place de la réactivité avec RxJS

**2. Ajoute les notions clés de la séquence dans README.md**

Écris une courte section contenant :

- les concepts que tu as compris
- ce que fait BehaviorSubject
- ce que fait | async
- comment fonctionne le flux service → composant → template

# Exemple README.MD

## Séquence 2 – Logique réactive du flux de données

### 1. Structure du flux
- Le service `TaskService` utilise un **BehaviorSubject** pour stocker et diffuser la liste des tâches.
- Le composant `Home` s'abonne à ce flux via `tasks$` et le **pipe async**.

### 2. Mise à jour des données
- La méthode `addTask()` ajoute une tâche puis appelle `next()` pour émettre la nouvelle liste.
- La méthode `removeTask()` supprime une tâche puis émet à nouveau la liste mise à jour.
- La vue est automatiquement réactualisée sans rechargement.

### 3. Points clés retenus
- Pas besoin d'appeler `getTasks()` à chaque fois : la donnée est **vivante**.
- `| async` gère l'abonnement et le désabonnement automatiquement.
- Le flux reste cohérent entre le service et la vue.