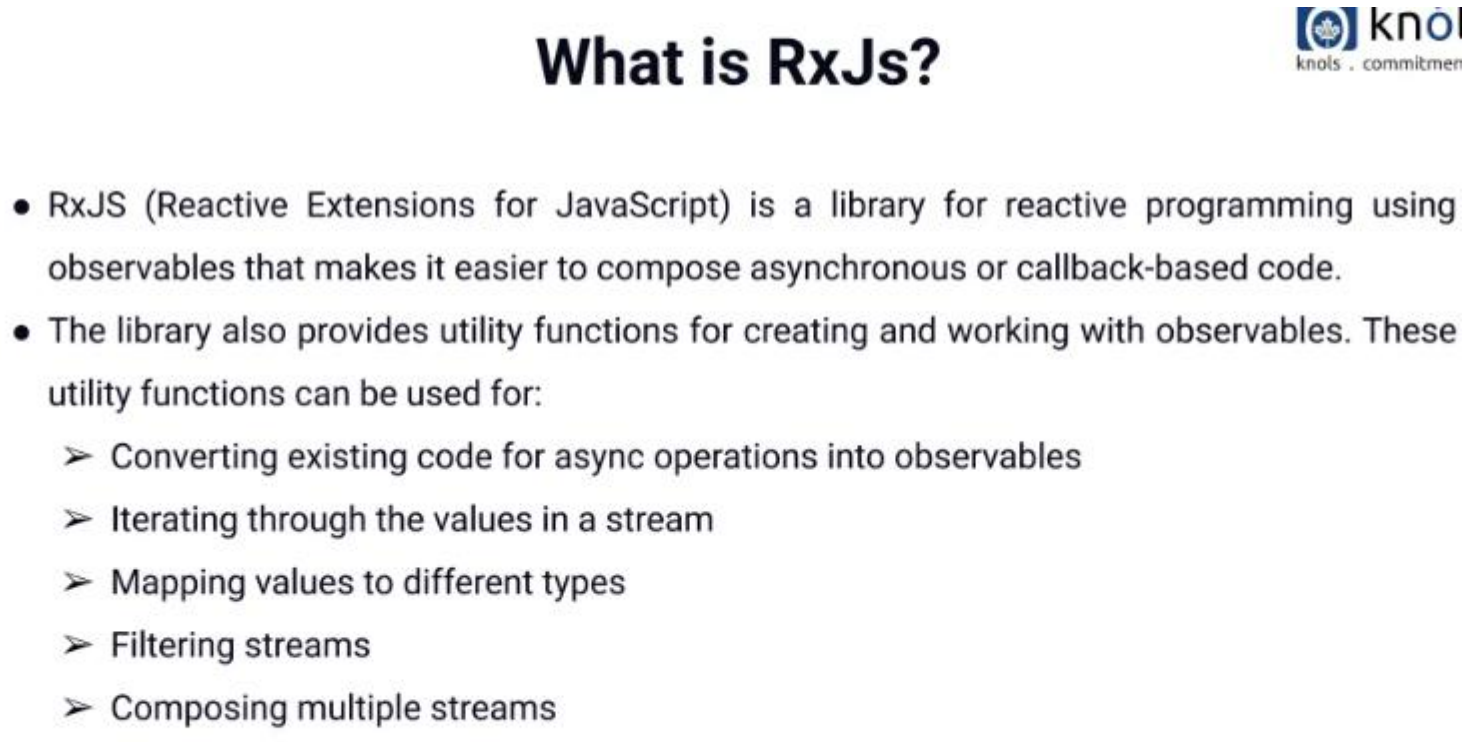
Angular v20

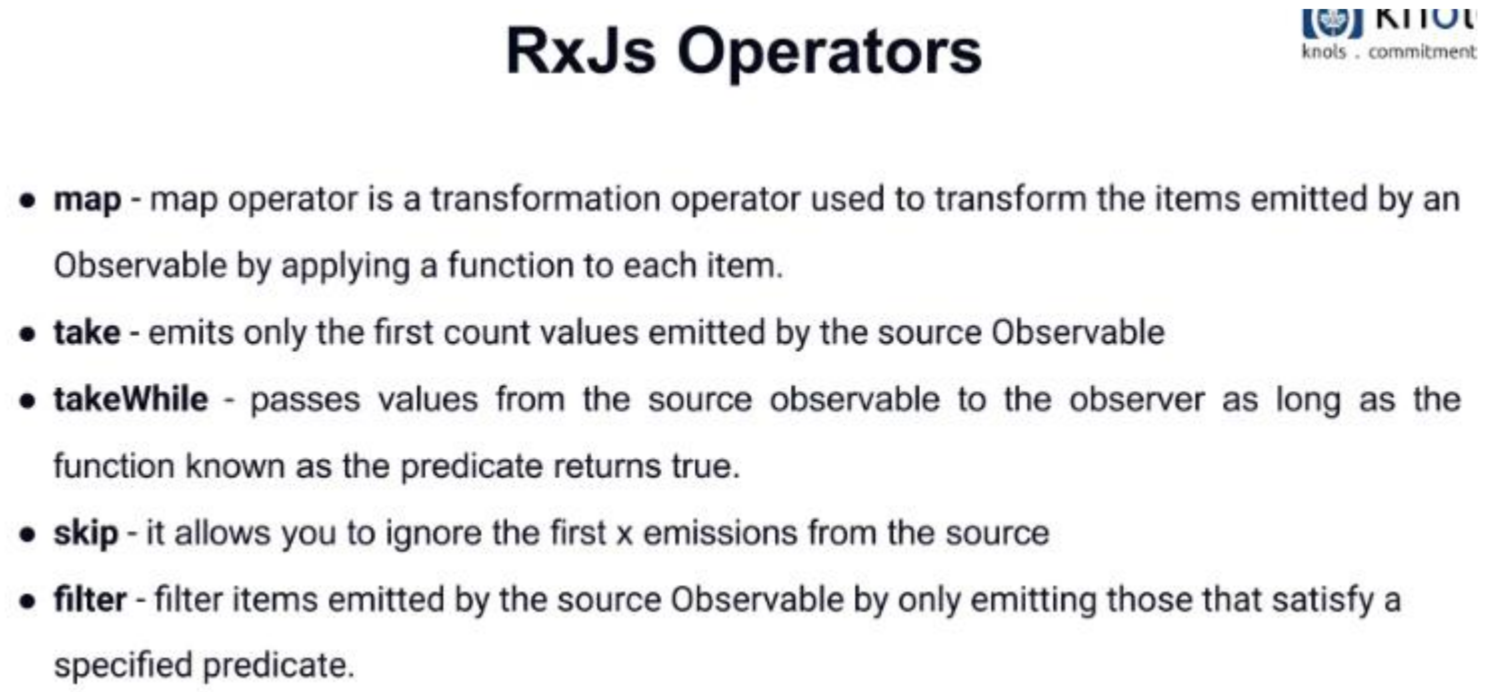
Observables



What is RxJs?

- RxJS (Reactive Extensions for JavaScript) is a library for reactive programming using observables that makes it easier to compose asynchronous or callback-based code.
- The library also provides utility functions for creating and working with observables. These utility functions can be used for:
  - ➢ Converting existing code for async operations into observables
  - ➢ Iterating through the values in a stream
  - ➢ Mapping values to different types
  - ➢ Filtering streams
  - ➢ Composing multiple streams

RxJS



RxJs Operators

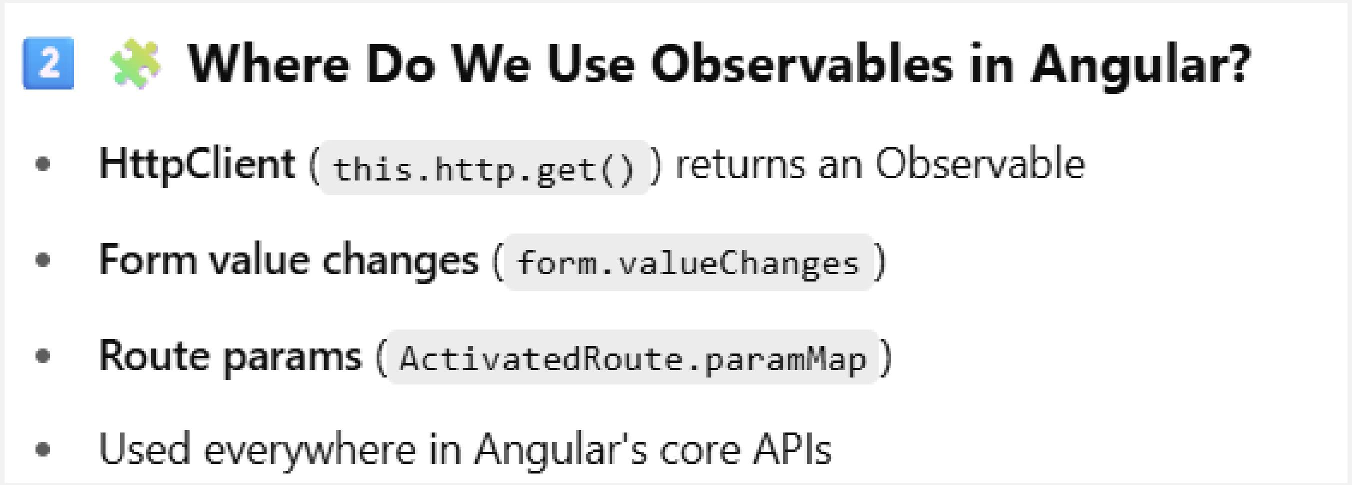
- **map** - map operator is a transformation operator used to transform the items emitted by an Observable by applying a function to each item.
- **take** - emits only the first count values emitted by the source Observable
- **takeWhile** - passes values from the source observable to the observer as long as the function known as the predicate returns true.
- **skip** - it allows you to ignore the first x emissions from the source
- **filter** - filter items emitted by the source Observable by only emitting those that satisfy a specified predicate.

Observables in Angular

Subs
1- Ahmed
2- ~~Mohamed~~

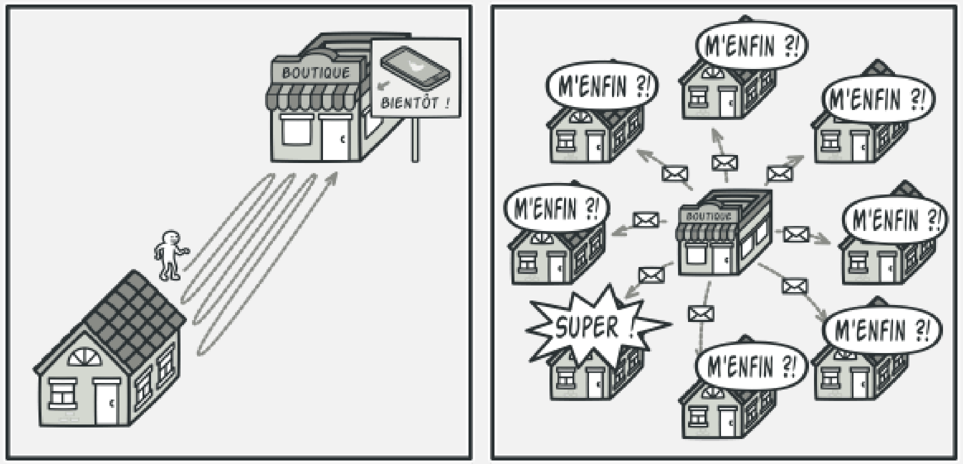
A way to **handle asynchronous data** (similar to **Promises**, but more powerful).

"An Observable is like a **YouTube channel** — you *subscribe* to it, and it pushes videos (data) to you over time."

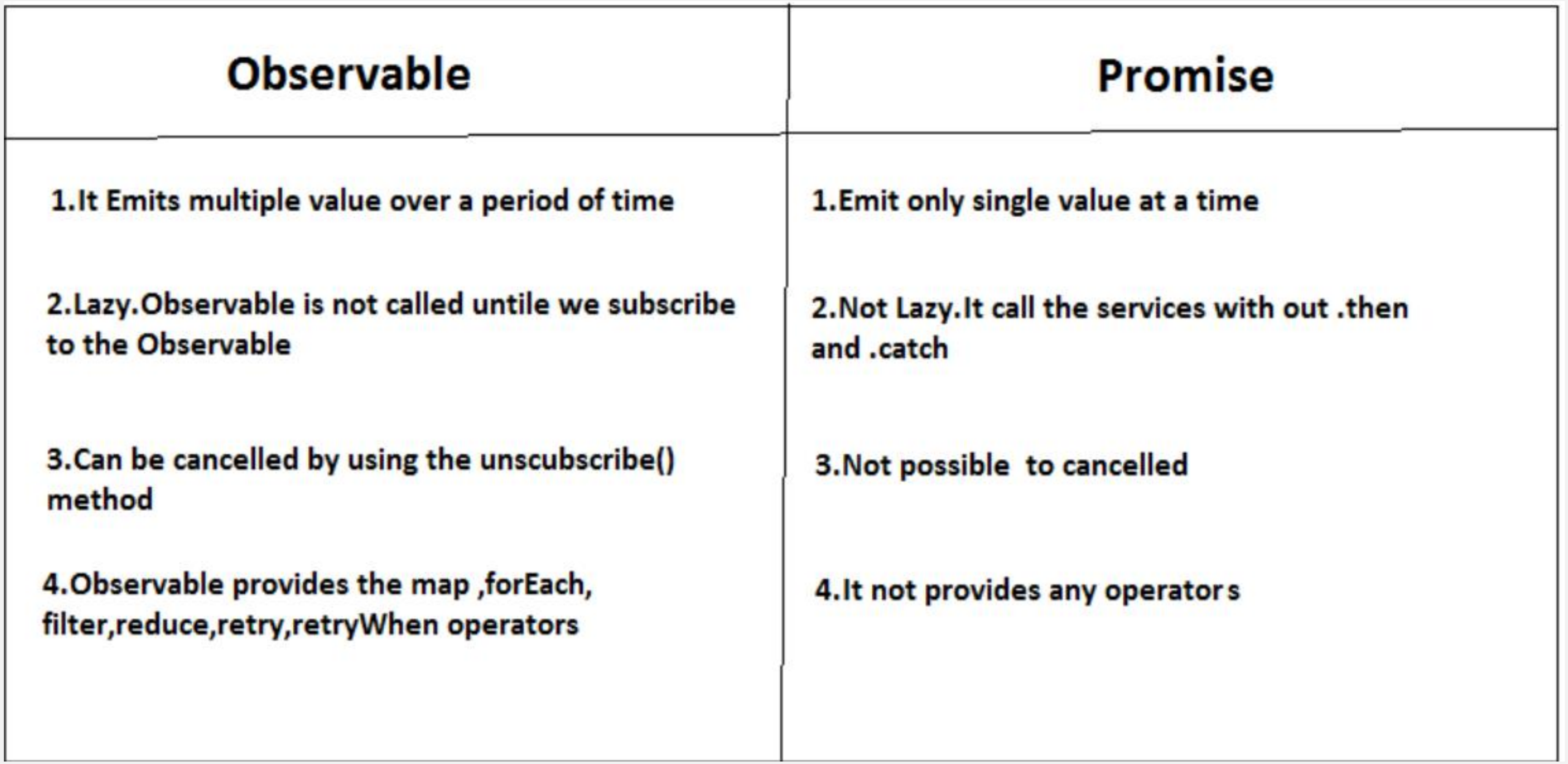


2️⃣ 🧩 Where Do We Use Observables in Angular?

- **HttpClient** (`this.http.get()`) returns an Observable
- **Form value changes** (`form.valueChanges`)
- **Route params** (`ActivatedRoute.paramMap`)
- Used everywhere in Angular's core APIs

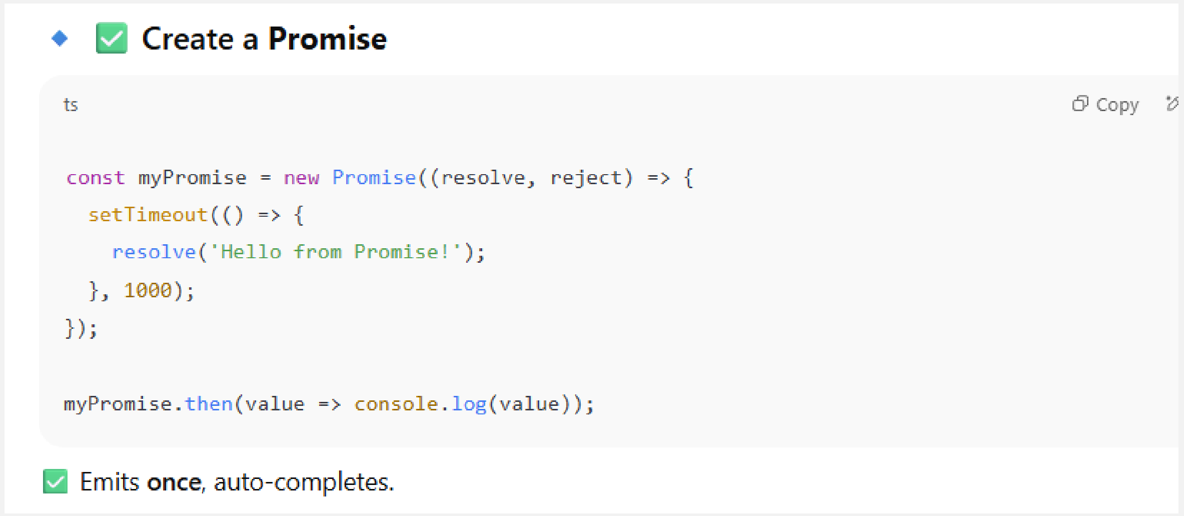


Observable vs Promise



Observable	Promise
1.It Emits multiple value over a period of time	1.Emit only single value at a time
2.Lazy.Observable is not called untile we subscribe to the Observable	2.Not Lazy.It call the services with out .then and .catch
3.Can be cancelled by using the unscubscribe() method	3.Not possible to cancelled
4.Observable provides the map ,forEach, filter,reduce,retry,retryWhen operators	4.It not provides any operators

Observable vs Promise

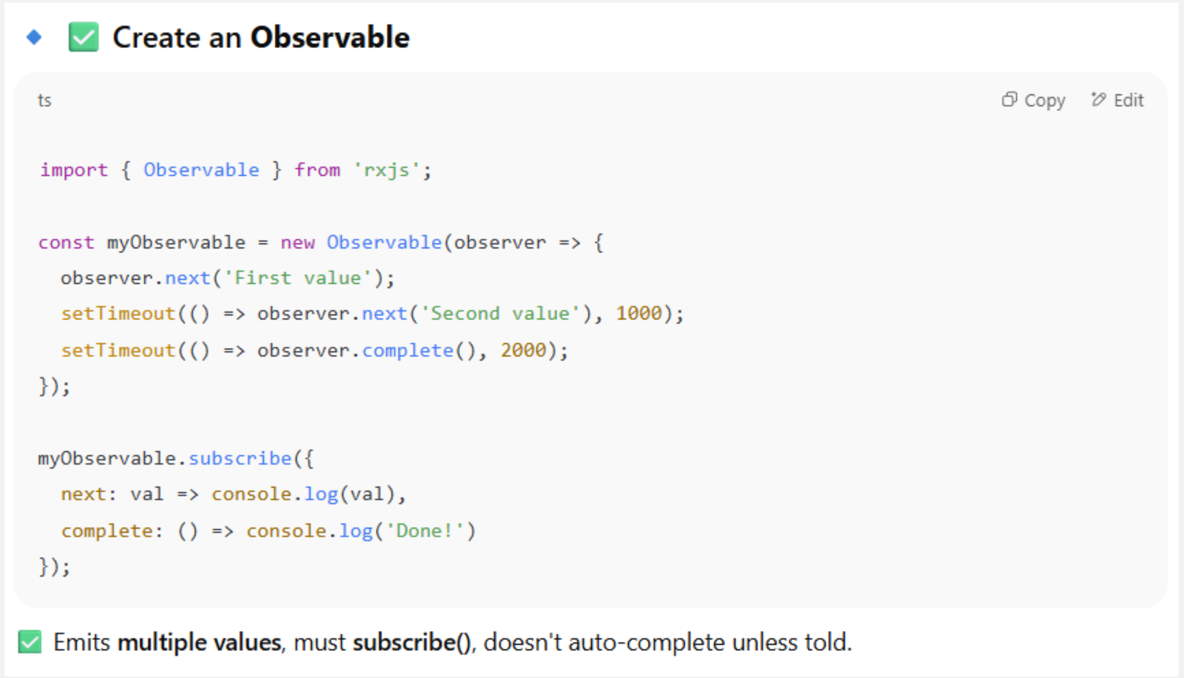


🔹 ✅ Create a **Promise**

ts

```ts
const myPromise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('Hello from Promise!');
  }, 1000);
});

myPromise.then(value => console.log(value));
```

✅ Emits **once**, auto-completes.

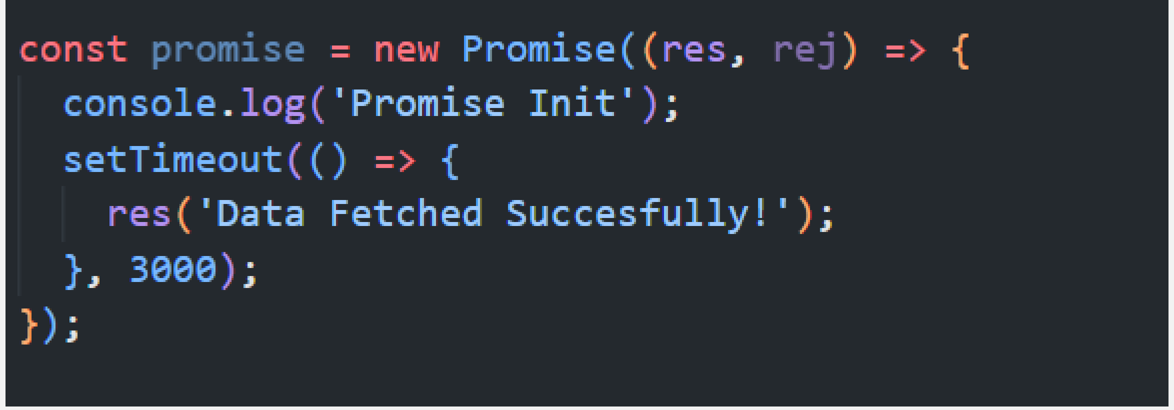


🔹 ✅ Create an **Observable**

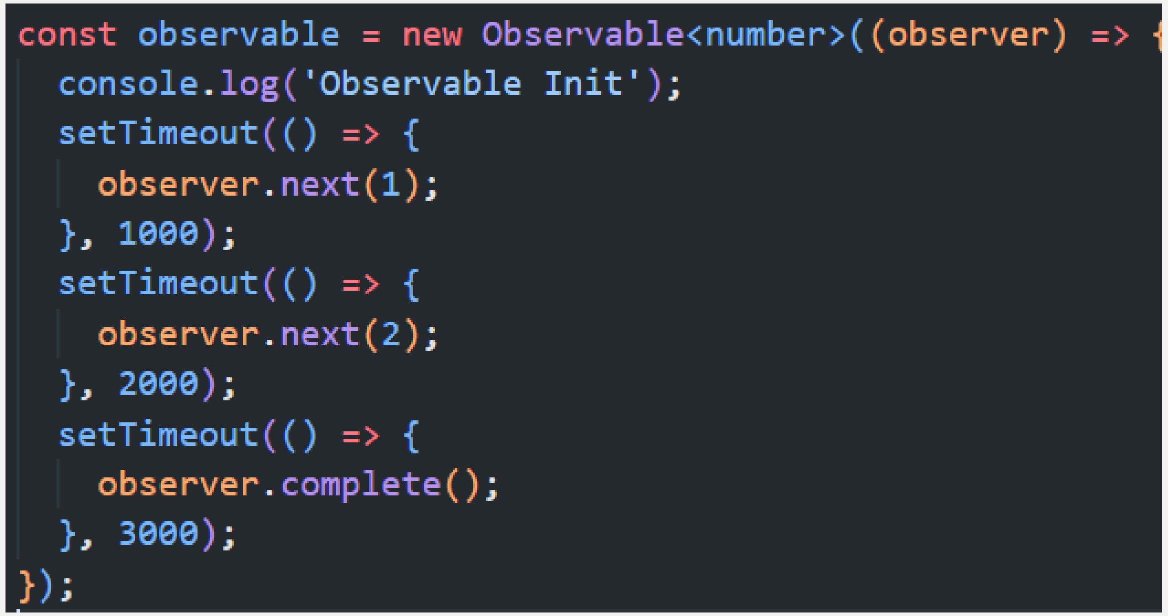
ts

```ts
import { Observable } from 'rxjs';

const myObservable = new Observable(observer => {
  observer.next('First value');
  setTimeout(() => observer.next('Second value'), 1000);
  setTimeout(() => observer.complete(), 2000);
});

myObservable.subscribe({
  next: val => console.log(val),
  complete: () => console.log('Done!')
});
```

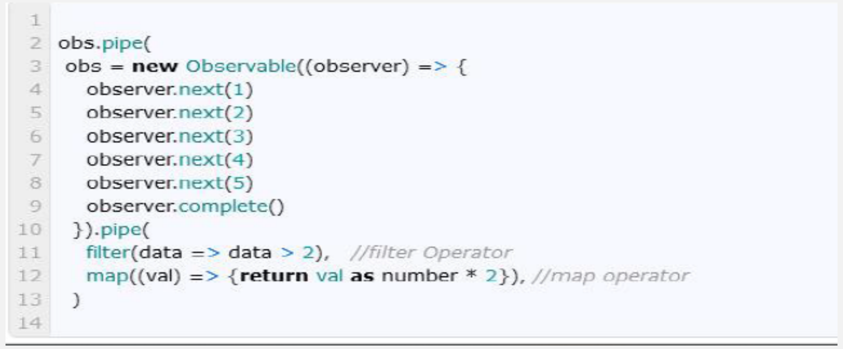
✅ Emits **multiple values**, must **subscribe()**, doesn't auto-complete unless told.



```ts
const promise = new Promise((res, rej) => {
  console.log('Promise Init');
  setTimeout(() => {
    res('Data Fetched Succesfully!');
  }, 3000);
});
```

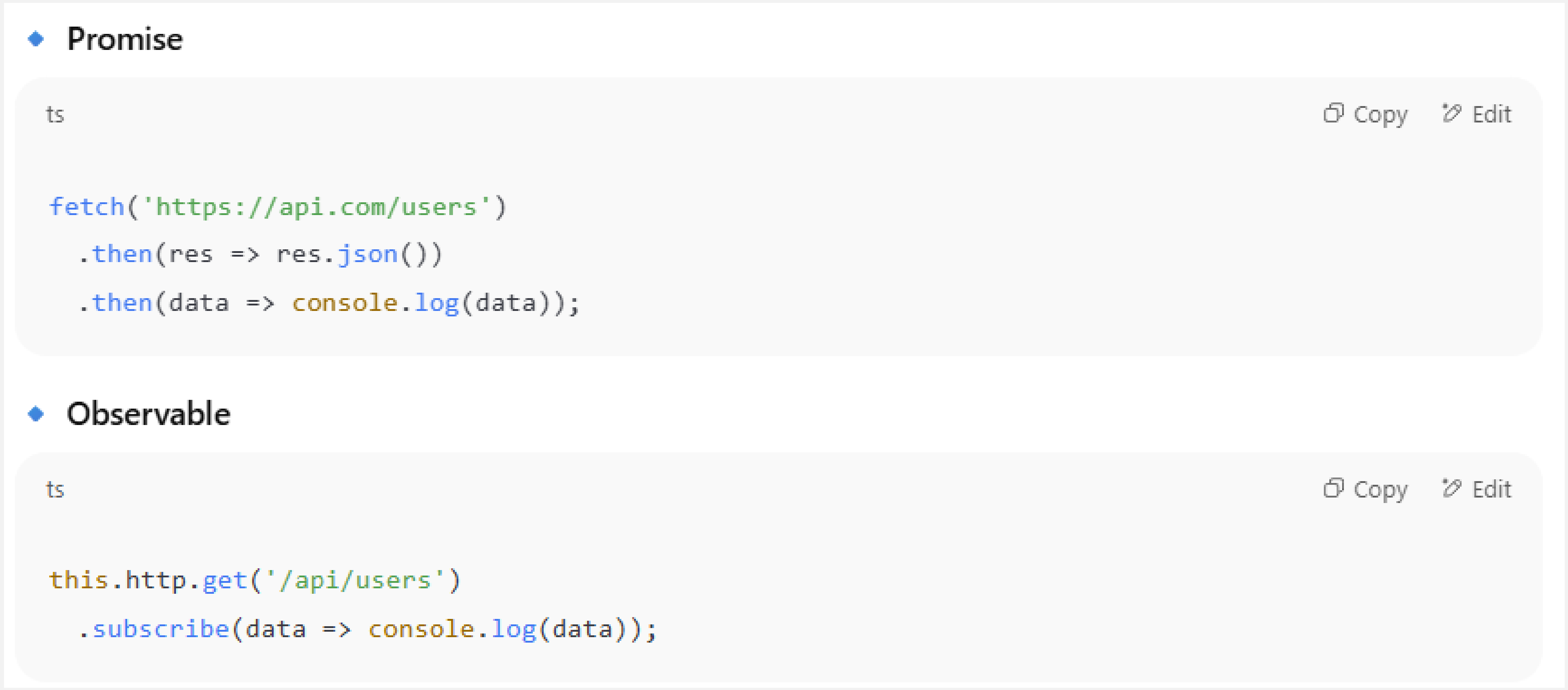


```ts
const observable = new Observable<number>((observer) => {
  console.log('Observable Init');
  setTimeout(() => {
    observer.next(1);
  }, 1000);
  setTimeout(() => {
    observer.next(2);
  }, 2000);
  setTimeout(() => {
    observer.complete();
  }, 3000);
});
```



```ts

obs.pipe(
 obs = new Observable((observer) => {
   observer.next(1)
   observer.next(2)
   observer.next(3)
   observer.next(4)
   observer.next(5)
   observer.complete()
  }).pipe(
   filter(data => data > 2),   //filter Operator
   map((val) => {return val as number * 2}), //map operator
  )

```

Observable vs Promise



🔹 Promise

```ts
fetch('https://api.com/users')
  .then(res => res.json())
  .then(data => console.log(data));
```

🔹 Observable

```ts
this.http.get('/api/users')
  .subscribe(data => console.log(data));
```

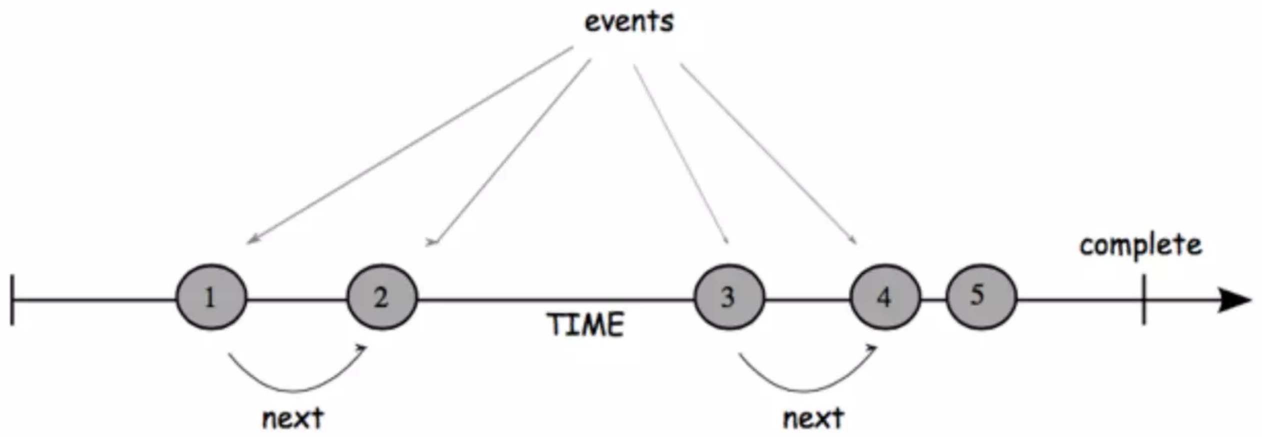
Streams

Stream is simply - **sequence of events over a given time.**

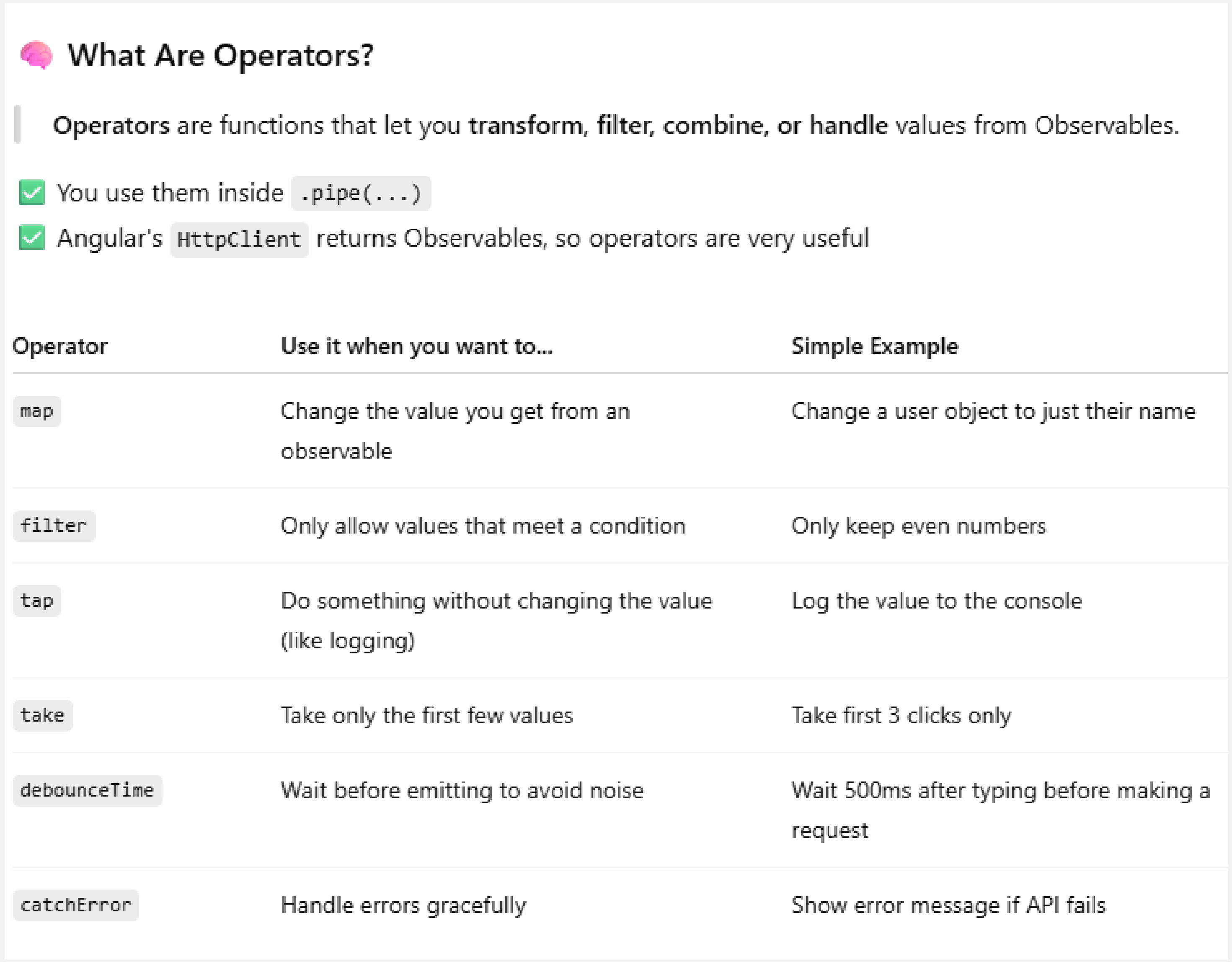
Streams can be used to process any of type of event such as

- mouse clicks,
- key presses,
- bits of network data, etc.

You can think of streams as variables that with the ability to react to changes emitted from the data they point to.



Observable Operators



🧠 What Are Operators?

> **Operators** are functions that let you **transform, filter, combine, or handle** values from Observables.

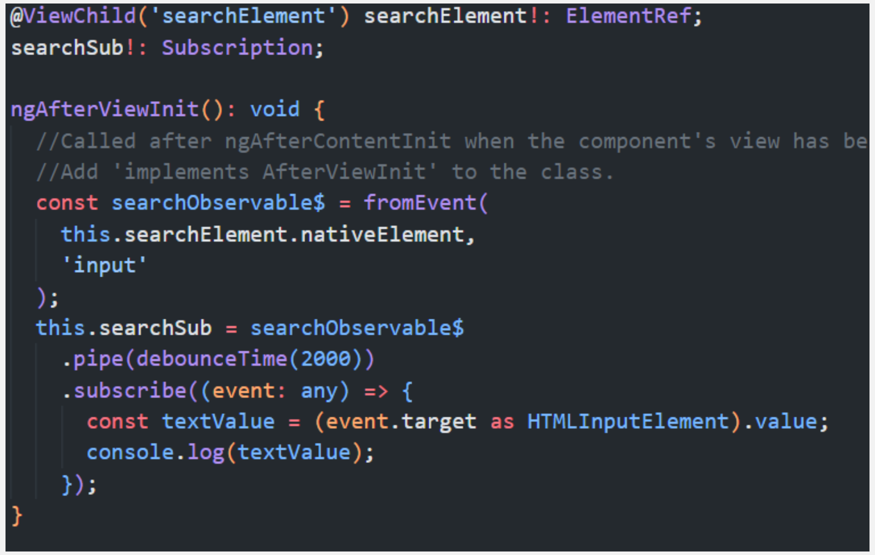
✅ You use them inside `.pipe(...)`
✅ Angular's `HttpClient` returns Observables, so operators are very useful

Operator	Use it when you want to...	Simple Example
`map`	Change the value you get from an observable	Change a user object to just their name
`filter`	Only allow values that meet a condition	Only keep even numbers
`tap`	Do something without changing the value (like logging)	Log the value to the console
`take`	Take only the first few values	Take first 3 clicks only
`debounceTime`	Wait before emitting to avoid noise	Wait 500ms after typing before making a request
`catchError`	Handle errors gracefully	Show error message if API fails

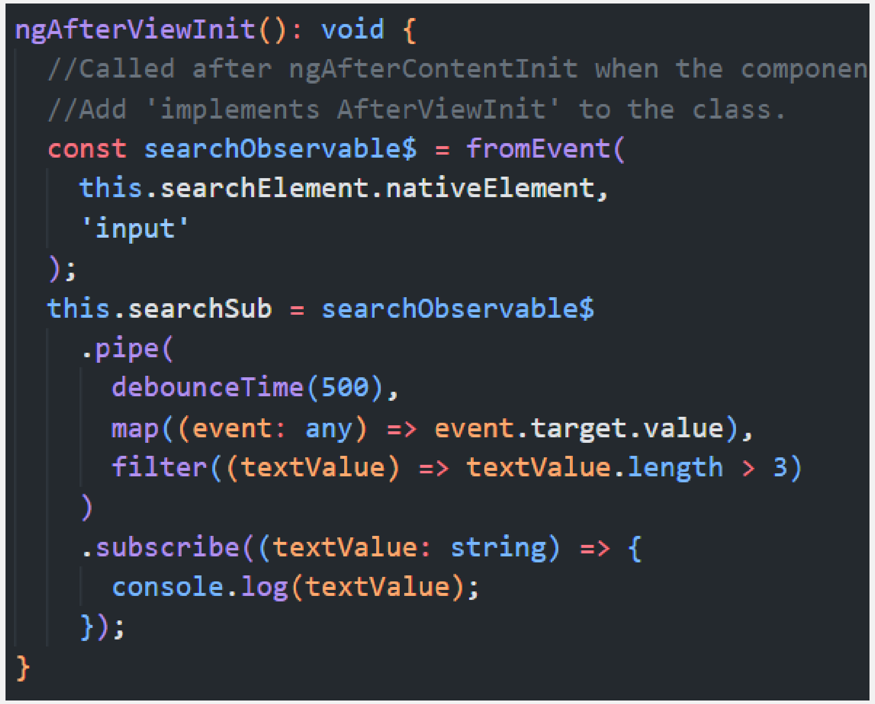
📘 Slide: Example – Basic Pipeline

ts

```ts
this.http.get<User[]>('/api/users').pipe(
  tap(() => this.loading = true),
  filter(users => users.length > 0),
  map(users => users.filter(user => user.active)),
  catchError(err => {
    console.error(err);
    return of([]); // fallback to empty list
  })
).subscribe(filteredUsers => {
  this.users = filteredUsers;
  this.loading = false;
});
```

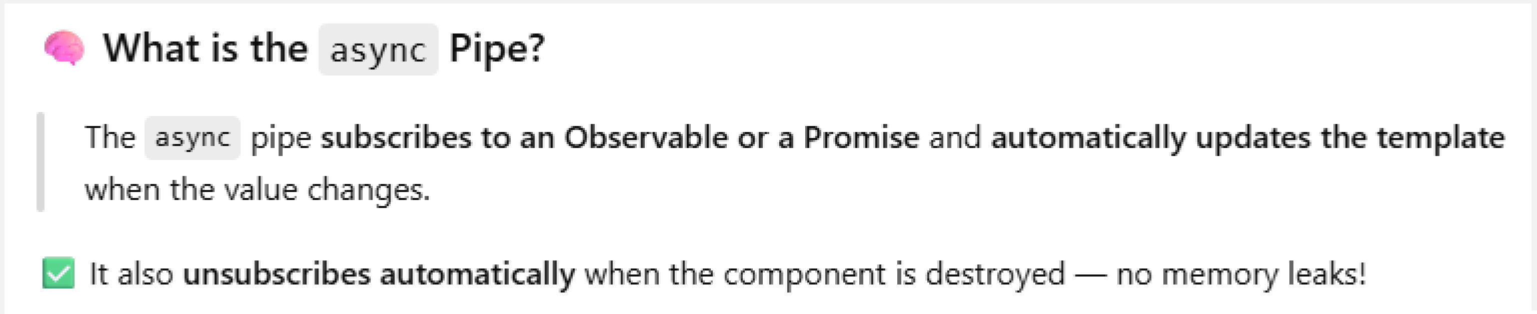


```ts
@ViewChild('searchElement') searchElement!: ElementRef;
searchSub!: Subscription;

ngAfterViewInit(): void {
  //Called after ngAfterContentInit when the component's view has be
  //Add 'implements AfterViewInit' to the class.
  const searchObservable$ = fromEvent(
    this.searchElement.nativeElement,
    'input'
  );
  this.searchSub = searchObservable$
    .pipe(debounceTime(2000))
    .subscribe((event: any) => {
      const textValue = (event.target as HTMLInputElement).value;
      console.log(textValue);
    });
}
```



```ts
ngAfterViewInit(): void {
  //Called after ngAfterContentInit when the componen
  //Add 'implements AfterViewInit' to the class.
  const searchObservable$ = fromEvent(
    this.searchElement.nativeElement,
    'input'
  );
  this.searchSub = searchObservable$
    .pipe(
      debounceTime(500),
      map((event: any) => event.target.value),
      filter((textValue) => textValue.length > 3)
    )
    .subscribe((textValue: string) => {
      console.log(textValue);
    });
}
```

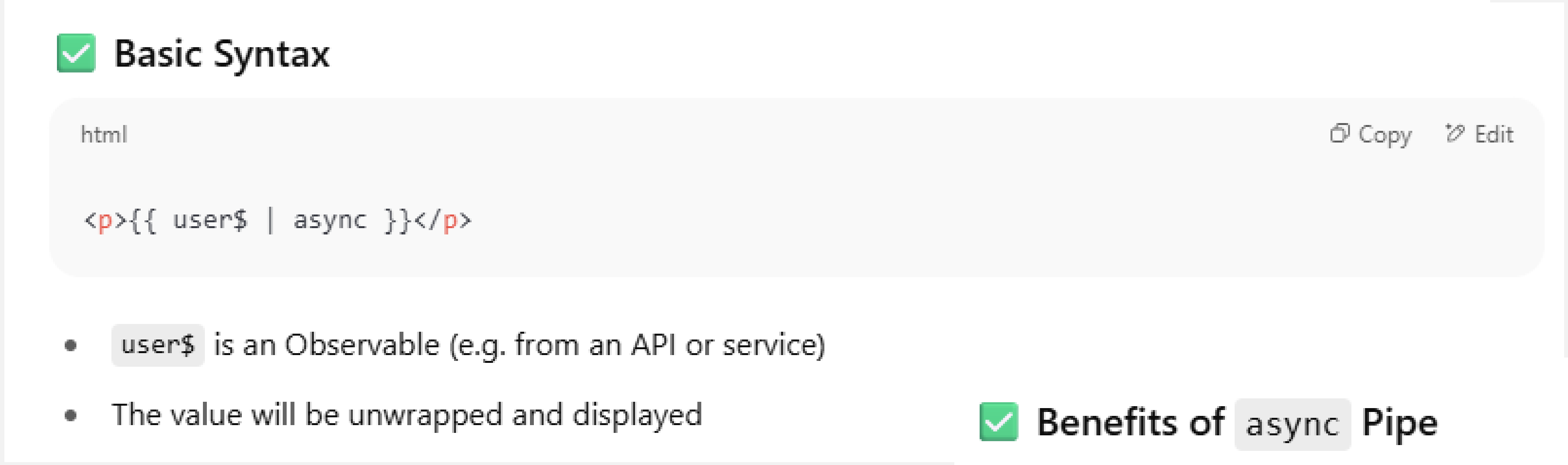
Async Pipe



🧠 What is the `async` Pipe?

> The `async` pipe **subscribes to an Observable or a Promise** and **automatically updates the template** when the value changes.

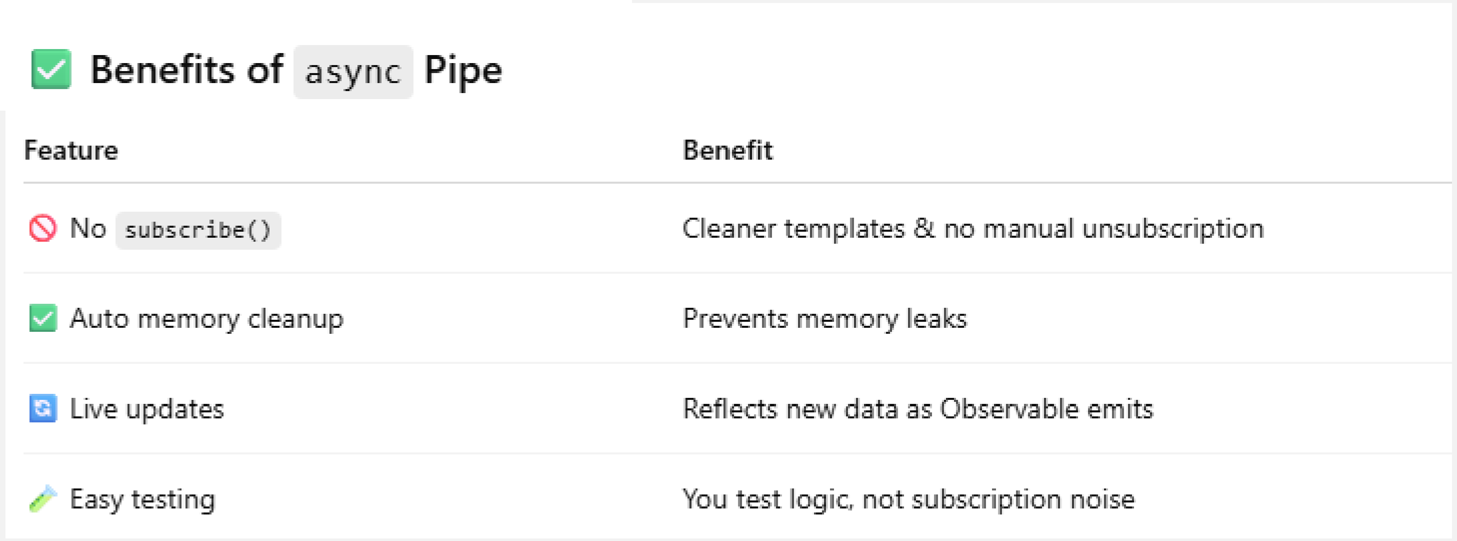
✅ It also **unsubscribes automatically** when the component is destroyed — no memory leaks!



✅ Basic Syntax

```html
<p>{{ user$ | async }}</p>
```

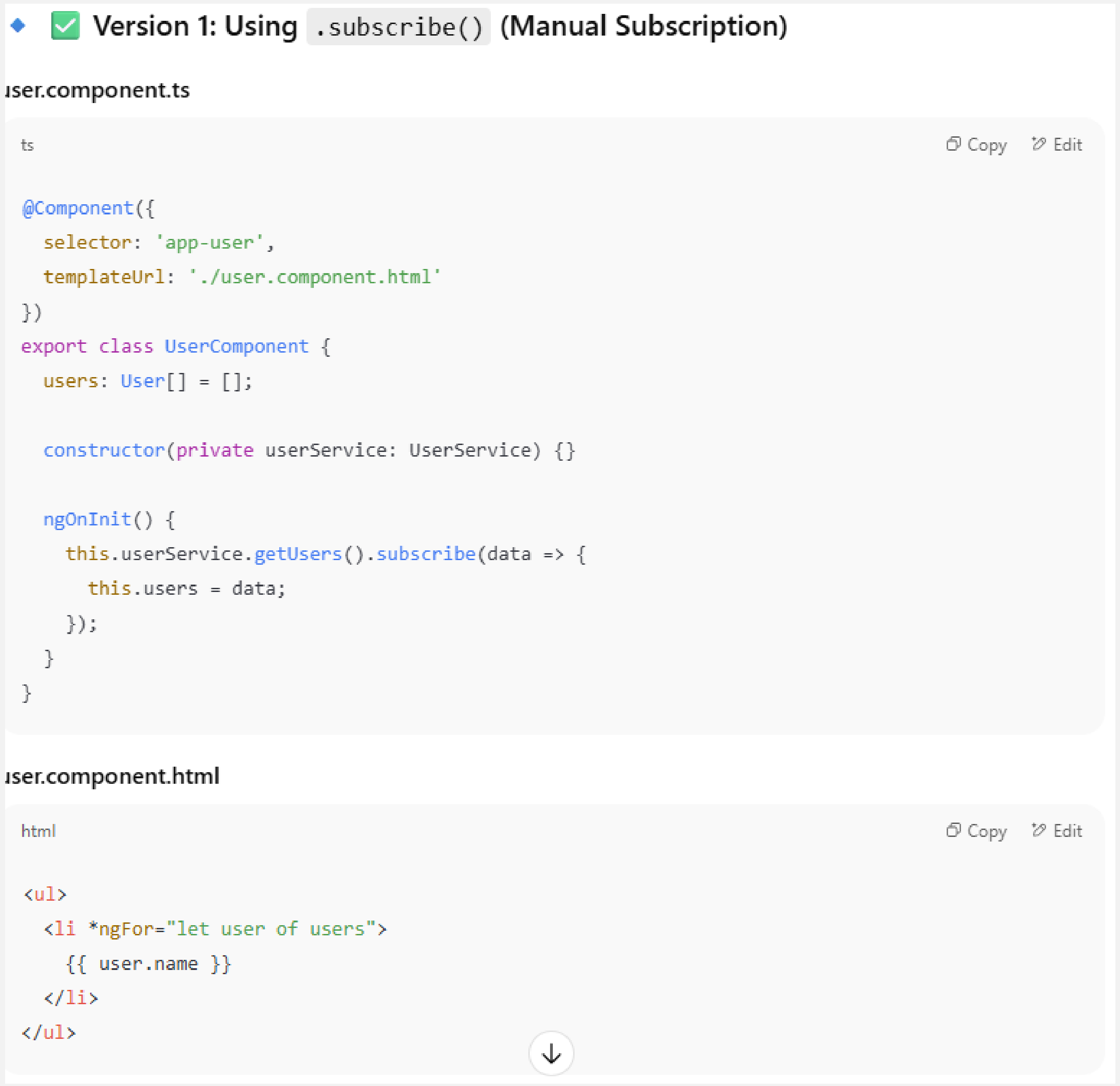
- `user$` is an Observable (e.g. from an API or service)
- The value will be unwrapped and displayed



✅ Benefits of `async` Pipe

Feature	Benefit
🚫 No `subscribe()`	Cleaner templates & no manual unsubscription
✅ Auto memory cleanup	Prevents memory leaks
🔄 Live updates	Reflects new data as Observable emits
🧪 Easy testing	You test logic, not subscription noise

📘 Manual .subscribe() VS async Pipe



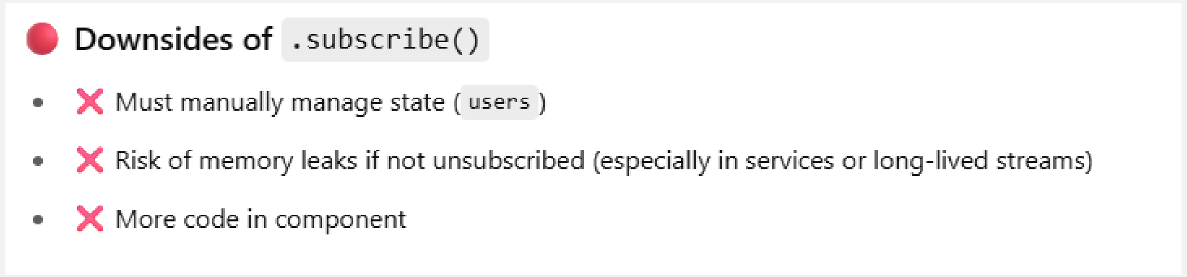
◆ ✅ Version 1: Using `.subscribe()` (Manual Subscription)

ıser.component.ts

```ts
@Component({
  selector: 'app-user',
  templateUrl: './user.component.html'
})
export class UserComponent {
  users: User[] = [];

  constructor(private userService: UserService) {}

  ngOnInit() {
    this.userService.getUsers().subscribe(data => {
      this.users = data;
    });
  }
}
```

ıser.component.html

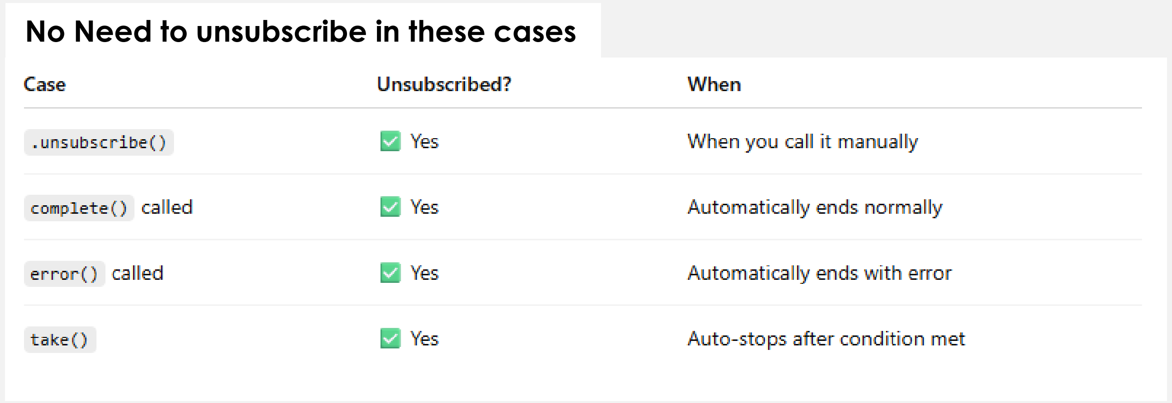
```html
<ul>
  <li *ngFor="let user of users">
    {{ user.name }}
  </li>
</ul>
```



🔴 Downsides of `.subscribe()`

- ❌ Must manually manage state (`users`)
- ❌ Risk of memory leaks if not unsubscribed (especially in services or long-lived streams)
- ❌ More code in component

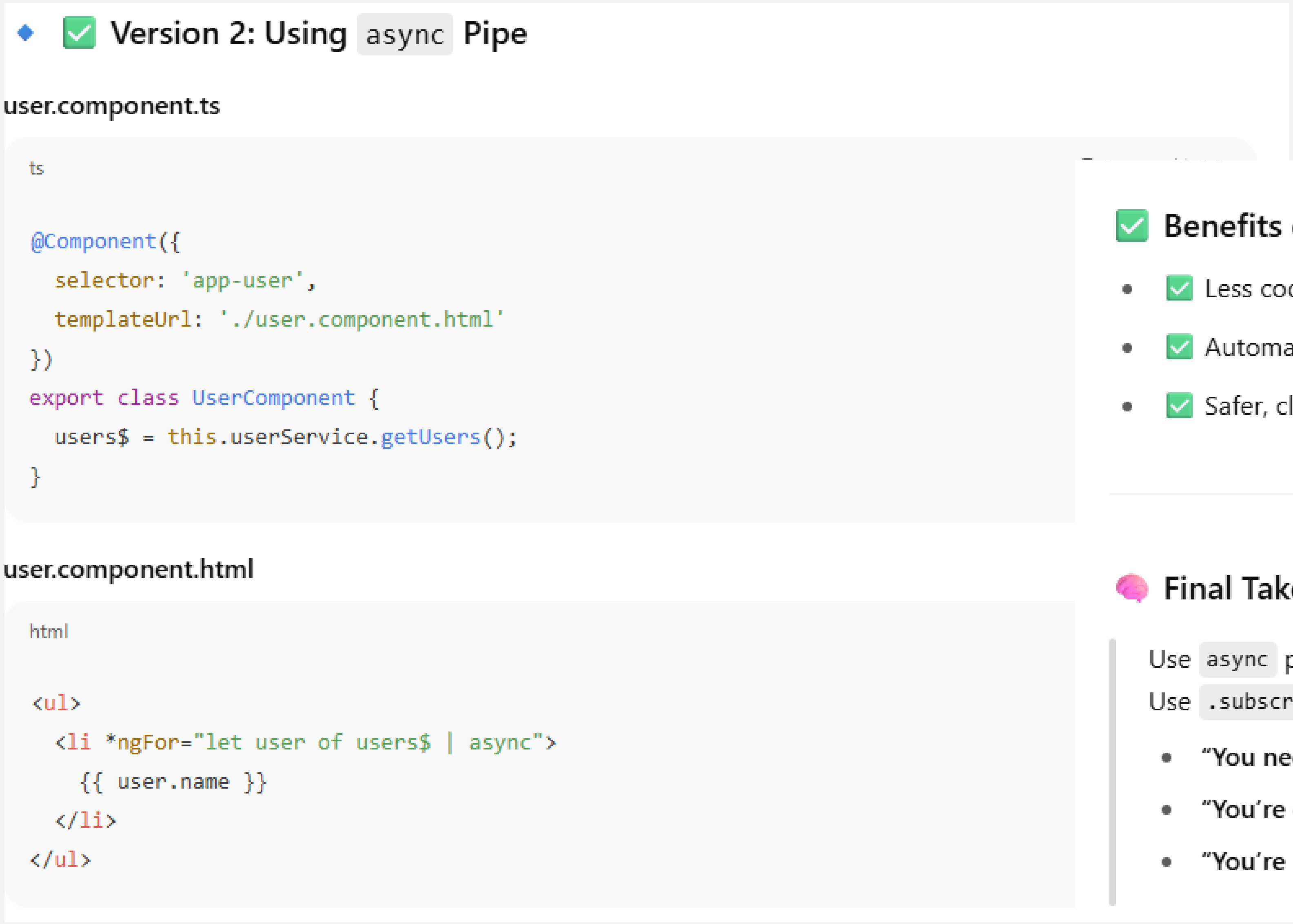
DON'T FORGET TO UNSUBSCRIBE



No Need to unsubscribe in these cases

Case	Unsubscribed?	When
`.unsubscribe()`	✅ Yes	When you call it manually
`complete()` called	✅ Yes	Automatically ends normally
`error()` called	✅ Yes	Automatically ends with error
`take()`	✅ Yes	Auto-stops after condition met

📘 Manual .subscribe() vs async Pipe



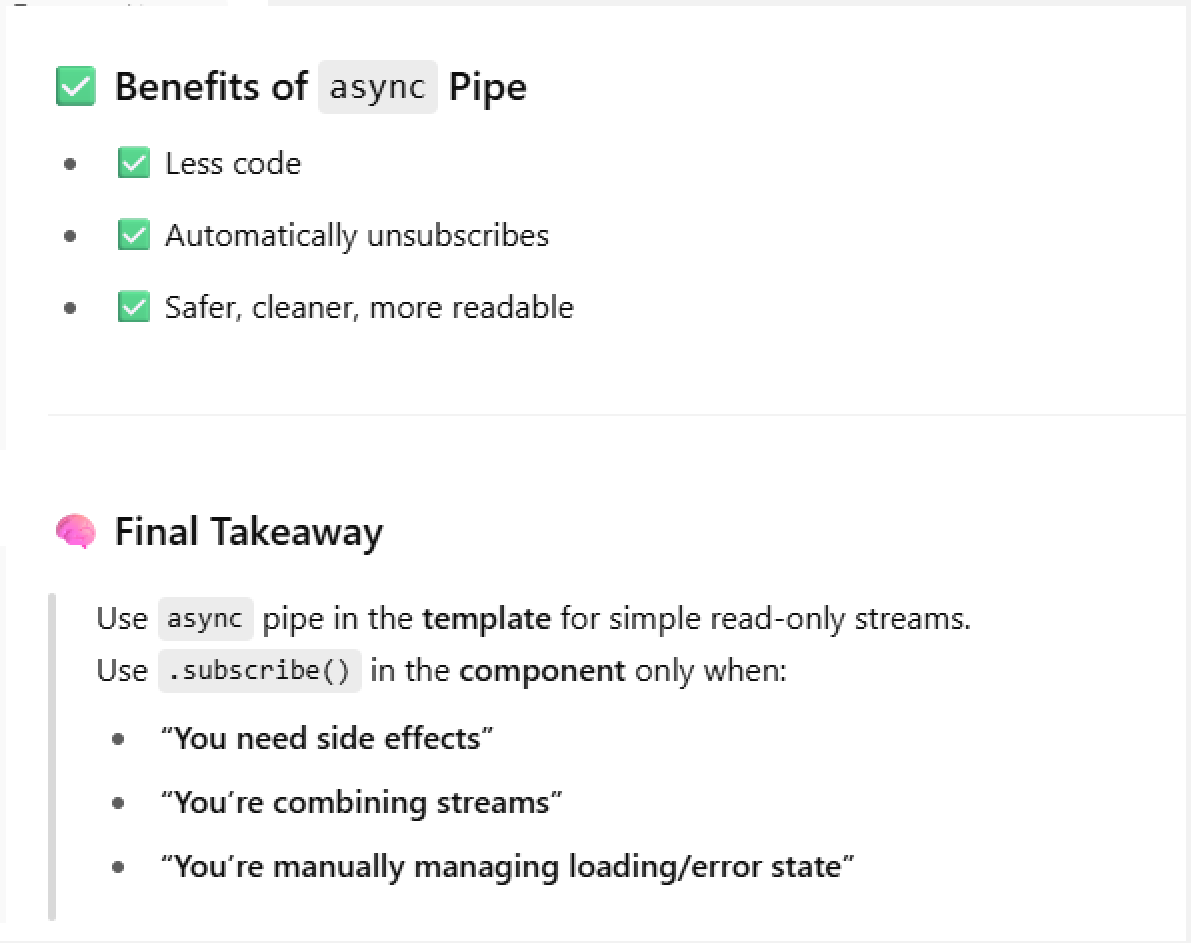
🔹 ✅ Version 2: Using `async` Pipe

user.component.ts

```ts
@Component({
  selector: 'app-user',
  templateUrl: './user.component.html'
})
export class UserComponent {
  users$ = this.userService.getUsers();
}
```

user.component.html

```html
<ul>
  <li *ngFor="let user of users$ | async">
    {{ user.name }}
  </li>
</ul>
```



✅ Benefits of `async` Pipe

- ✅ Less code
- ✅ Automatically unsubscribes
- ✅ Safer, cleaner, more readable

🧠 Final Takeaway

> Use `async` pipe in the **template** for simple read-only streams.
> Use `.subscribe()` in the **component** only when:
>
> - **"You need side effects"**
> - **"You're combining streams"**
> - **"You're manually managing loading/error state"**

HTTP Client Module

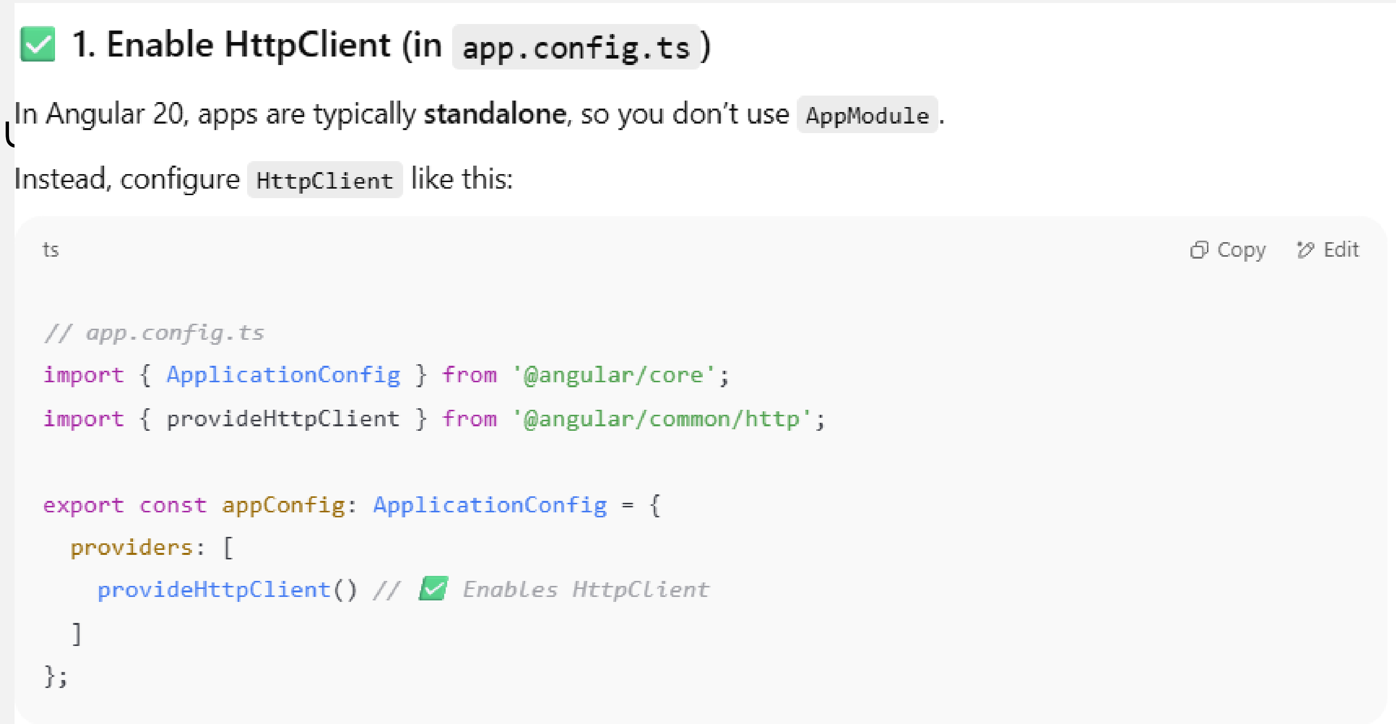
HTTP Client Module

A module provided by Angular that allows your app to **communicate with backend APIs** using HTTP requests like GET, POST, PUT, DELETE, etc.

✅ It's the **official Angular way** to perform HTTP calls.

🛠️ Requirements

- ~~Import the Module. (Before angular 20)~~
- Provide HttpClient in AppConfig.
- Inject HttpClient Service (Provided by Angu



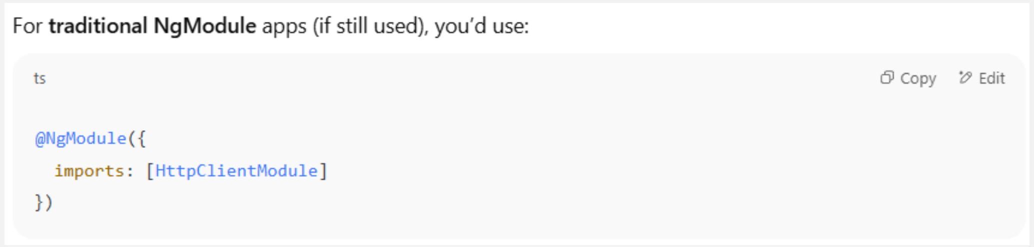
✅ 1. Enable HttpClient (in `app.config.ts`)

In Angular 20, apps are typically **standalone**, so you don't use `AppModule`.

Instead, configure `HttpClient` like this:

ts

```ts
// app.config.ts
import { ApplicationConfig } from '@angular/core';
import { provideHttpClient } from '@angular/common/http';

export const appConfig: ApplicationConfig = {
  providers: [
    provideHttpClient() // ✅ Enables HttpClient
  ]
};
```



For **traditional NgModule** apps (if still used), you'd use:

ts

```ts
@NgModule({
  imports: [HttpClientModule]
})
```

✅ Example: GET Request

user.service.ts

```ts
@Injectable({ providedIn: 'root' })
export class UserService {
  constructor(private http: HttpClient) {}

  getUsers() {
    return this.http.get<User[]>('https://jsonplaceholder.typicode.com/users');
  }
}
```

In your component:

```ts
users$: Observable<User[]> = this.userService.getUsers();
```

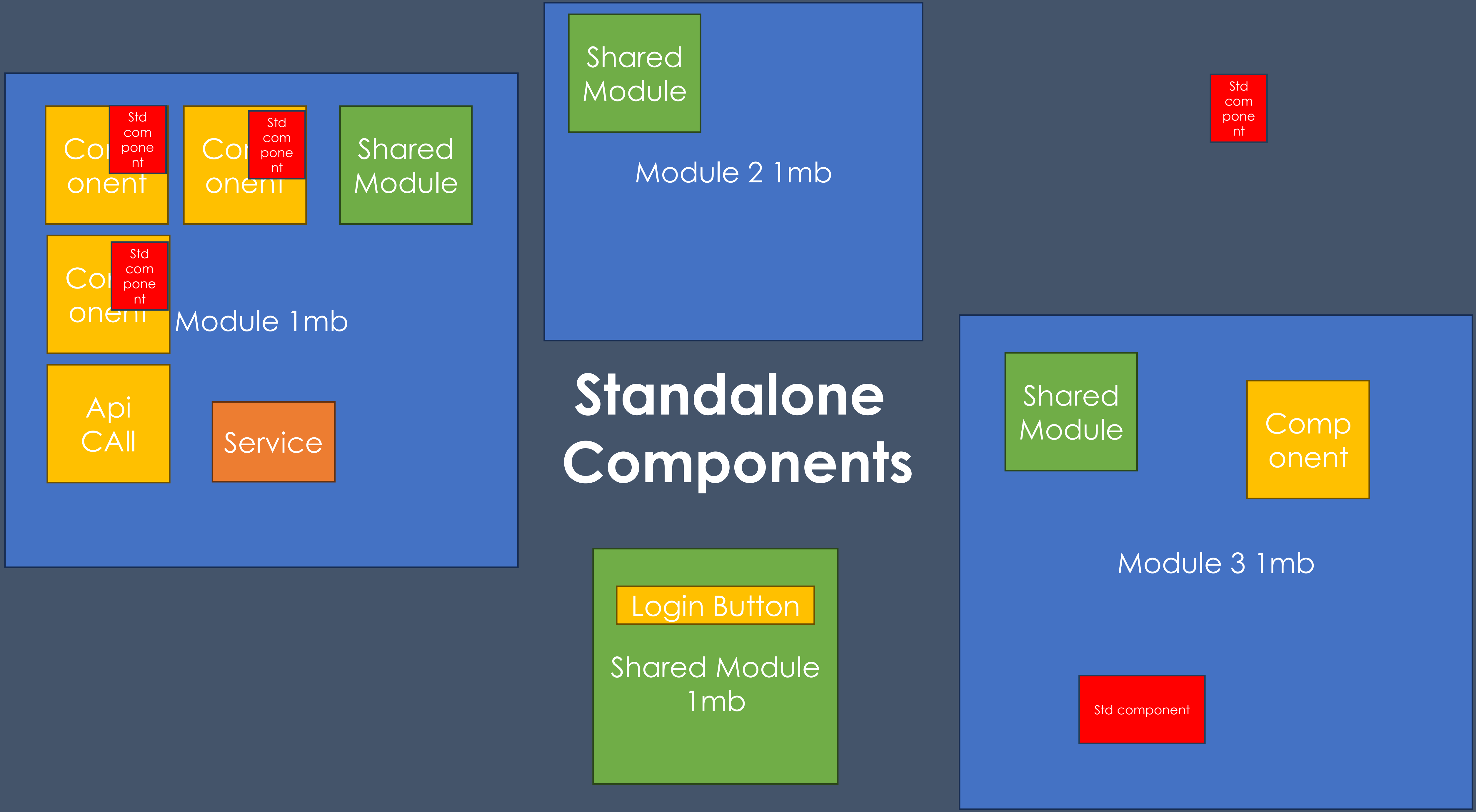
In the template (with async pipe):

```html
<ul>
  <li *ngFor="let user of users$ | async">
    {{ user.name }}
  </li>
</ul>
```

✅ Example: POST Request

```ts
addUser(user: User) {
  return this.http.post('https://api.com/users', user);
}
```

Standalone Components



🔹 What Are Standalone Components?

> A **Standalone Component** is a component that **doesn't require to be declared in an NgModule**.

✅ It can **import other components, directives, and pipes directly**.

🎯 Why Use Standalone Components?

Traditional Angular	Standalone Component
Needs `declarations` in module	✅ Self-contained component
NgModule is required	❌ NgModule is optional
Harder for lazy loading	✅ Easier for lazy & dynamic loading
More boilerplate	✅ Cleaner, modular code

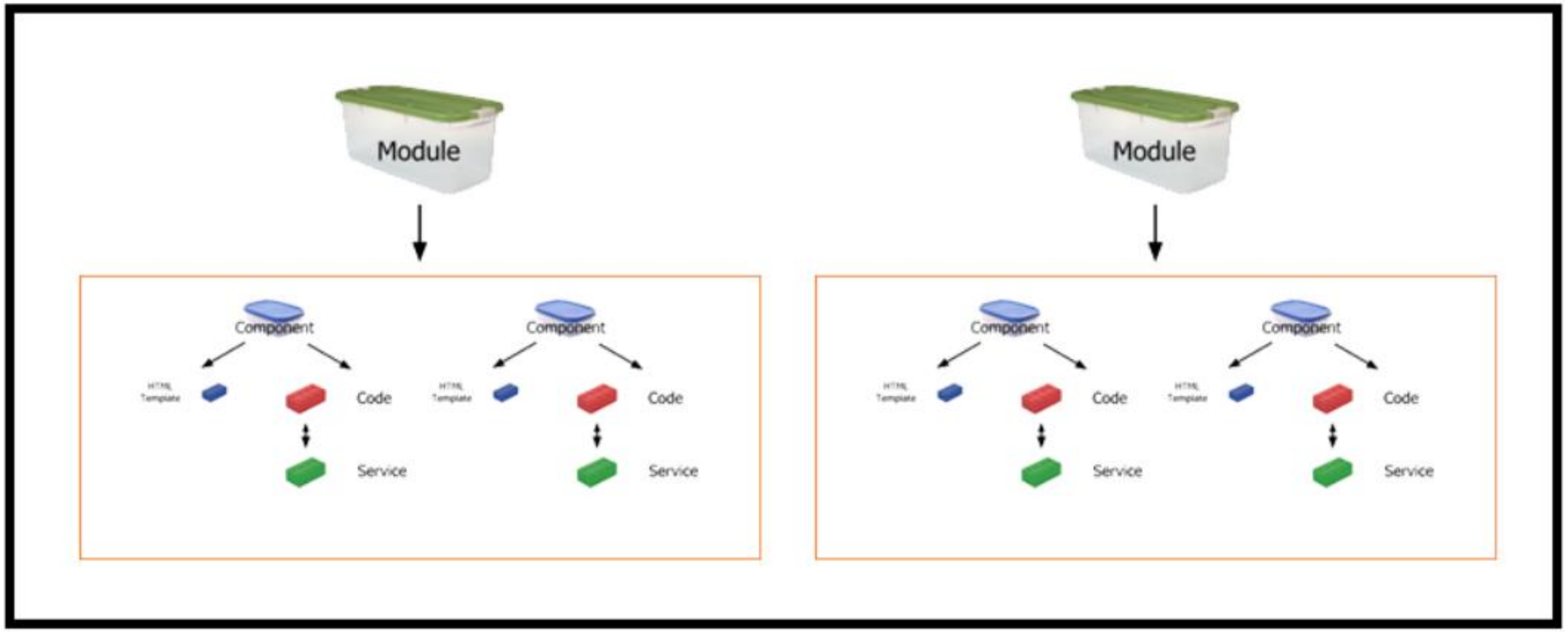
📦 Example

ts

```ts
@Component({
  standalone: true,
  selector: 'app-hello',
  template: `<h1>Hello, Angular 20!</h1>`,
  imports: [CommonModule] // You import what you use
})
export class HelloComponent {}
```

NgModules

App


Module
Module
Component
Component
Component
Component
HTML Template
Code
Service
HTML Template
Code
Service
HTML Template
Code
Service
HTML Template
Code
Service