ANGULAR

13-03-2025

DATA BINDING

Data binding in Angular is the mechanism that connects the data in the component (TypeScript) to the view (HTML) and vice versa. It ensures that the UI and component logic stay in sync.

Data Binding Types

- String Interpolation
- Property Binding
- Event Binding
- Two-Way Data Binding

STRING INTERPOLATION

Syntax: {{ expression }}

Purpose: Display dynamic data from the component in the template.

How It Works?

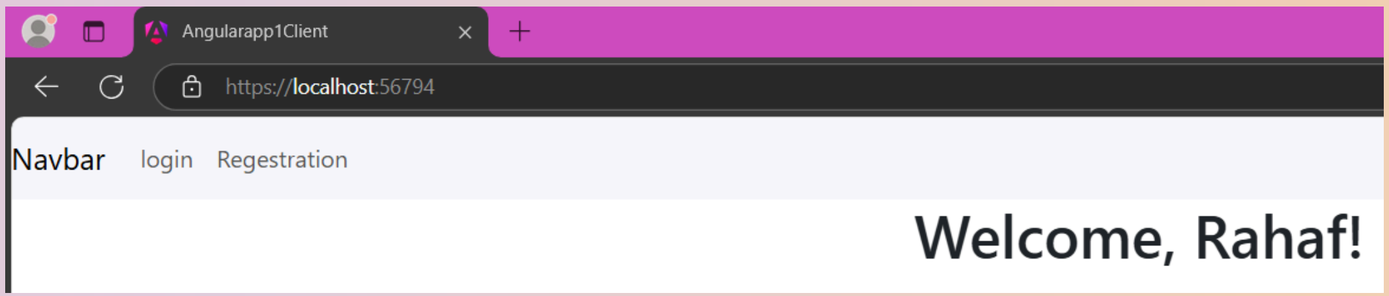
- The expression inside {{ }} is evaluated and converted into a string.
- It's mainly used to display variables or function results in the HTML.

binding.component.ts File

```
export class BindingComponent {
  ///String Interpolation///
  name = 'Rahaf';
}
```

binding.component.html File

```
<h1>Welcome, {{ name }}!</h1>
```



PROPERTY BINDING

Syntax: [property]="expression"

Purpose: Bind a property of an HTML element to a variable in the component.

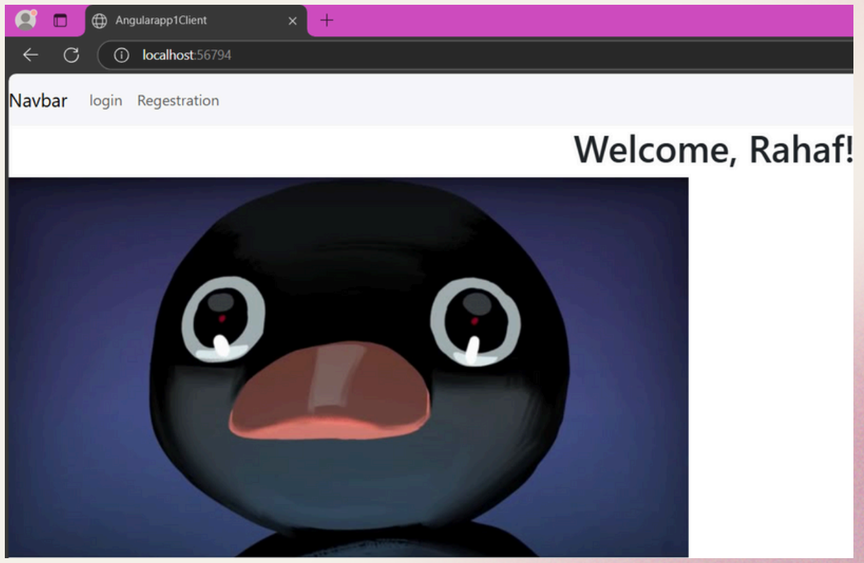
How It Works?

- The value of the expression is assigned to the corresponding property of the element.
- It's often used for setting properties like src, href, disabled, etc.

```
export class BindingComponent {

  ///Property Binding///
  imageUrl = '/assets/Pingu.jpg';
```

```
<img [src]="imageUrl" alt="img">
```



EVENT BINDING

Syntax: (event)="handler"

Purpose: Listen to events (like clicks, input changes, etc.) and trigger methods in the component.

How It Works?

- When the specified event occurs, Angular executes the method defined in the component.

```
export class BindingComponent {

  ///Event Binding///
  message = '';

  showMessage() {
    this.message = 'Button Clicked!';
  }
```

```
<button (click)="showMessage()">Click Me</button>
<p>{{ message }}</p>
```

TWO-WAY DATA BINDING

Syntax: [(ngModel)]="property"

Purpose: Create a two-way binding between the view and the component, meaning changes in the input update the component and vice versa.

How It Works?

- Combines property binding and event binding.
- Requires importing the FormsModule in the app module.

Important: To use [(ngModel)] for two-way data binding in Angular, you need to import FormsModule in your module file.

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

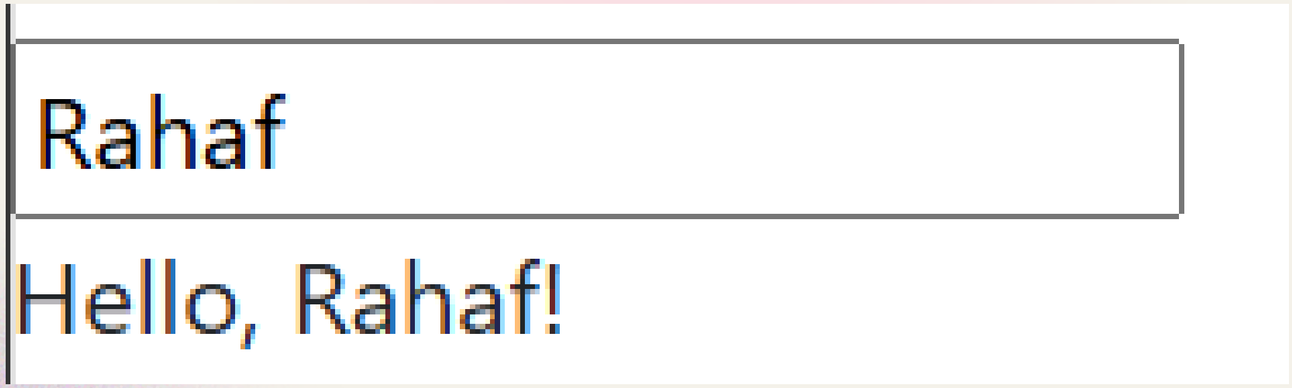
```
@NgModule({
  declarations: [
    AppComponent,
    LoginComponent,
    RegistrationComponent,
    NavbarComponent,
    FooterComponent,
    BindingComponent
  ],
  imports: [
    BrowserModule, HttpClientModule,
    AppRoutingModule,
    FormsModule
  ],
```

TWO-WAY DATA BINDING

```ts
export class BindingComponent {

  ///Two-Way Data Binding///
  username = '';
```

```html
<input [(ngModel)]="username" placeholder="Enter your name">
<p>Hello, {{ username }}!</p>
```



WHAT IS AN ANGULAR SERVICE?

An Angular Service is a reusable, singleton class that is used to organize and share data, logic, and functions across multiple components in an Angular application.

Why Use Services?

- **Reusability:** Share common logic across multiple components.
- **Separation of Concerns:** Keep business logic separate from UI components.
- **Maintainability:** Easier to manage and test.
- **Dependency Injection:** Angular automatically injects services where needed.

How to Create and Use an Angular Service?

1. Creating a Service

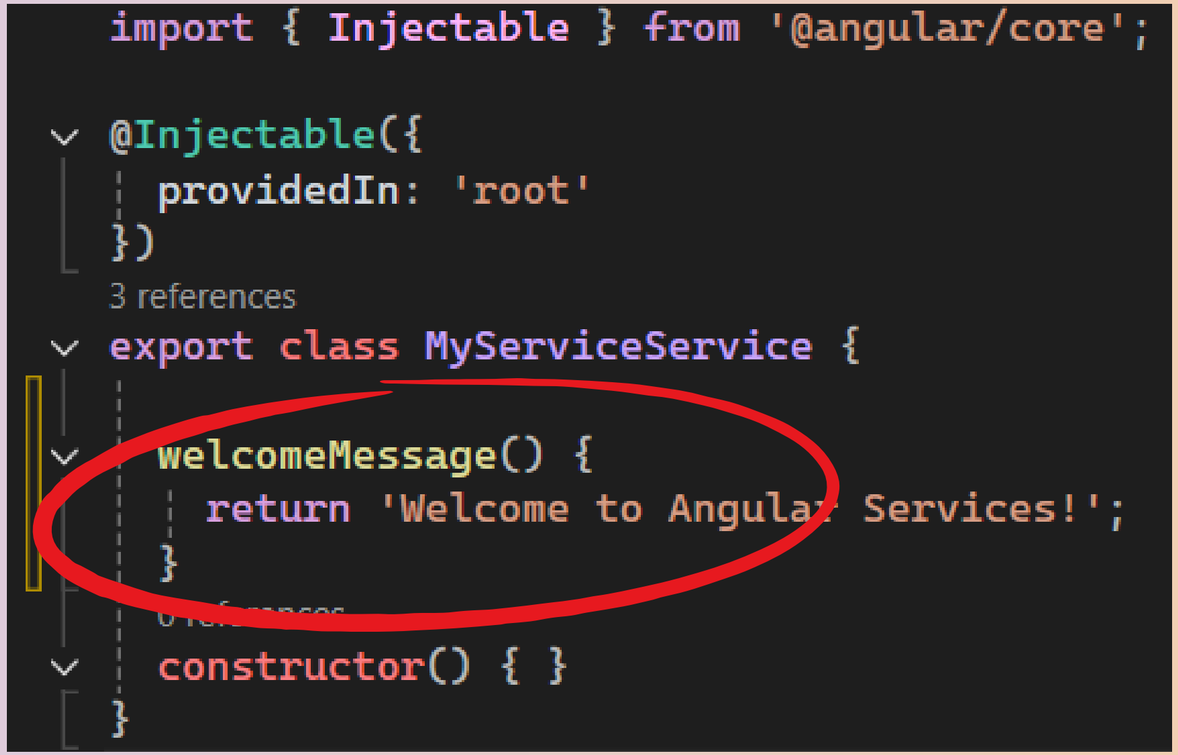
Using the terminal:

ng generate service my-service

This will create two files:

- my-service.service.ts
- my-service.service.spec.ts (for testing)

2. Inside the file “my-service.service.ts”



- The @Injectable decorator marks the class as a service that can be injected.
- The { providedIn: 'root' } ensures the service is singleton and available throughout the app.

3. Injecting and Using the Service in a Component.

binding.component.ts

```
import { MyService } from '../../my-service.service';//
```

```
export class BindingComponent {
  message: string;

  – references
  constructor(private myService: MyService) {
    this.message = this.myService.welcomeMessage();
  }
}
```

binding.component.html

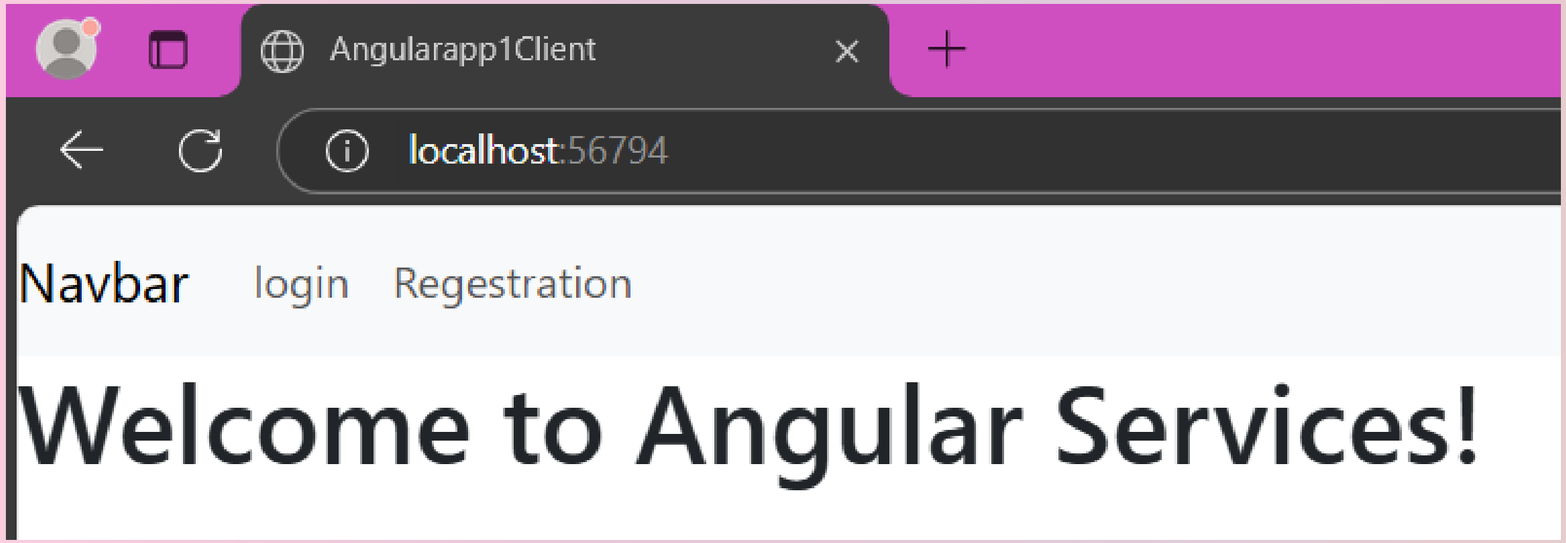
4. Using the Service in HTML

```
<h1>{{ message }}</h1>
```

4. Using the Service in HTML

binding.component.html

```
<h1>{{ message }}</h1>
```



To clarify:

- The service doesn't directly interact with the HTML file. It provides data or logic that can be used in the component.
- The component connects the service to the HTML template.

How It Works?

- Service (my-service.service.ts): Provides the business logic or data.
- Component (app.component.ts): Injects the service and uses it.
- HTML template (app.component.html): Displays data provided by the component.

FETCH DATA FROM API (GET ALL , GET BY ID)

Fetching data from an API in Angular typically involves using HttpClient from Angular's @angular/common/http module. You can use the HttpClient to send GET requests to an API, which will allow you to retrieve data, either as a collection (all records) or based on a specific identifier (single record by ID).

1- Import HttpClientModule (in the app.module.ts file)

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

```
],
imports: [
  BrowserModule,
  HttpClientModule,
  AppRoutingModule,
  FormsModule
],
```

2- Create the Service to Fetch Data (in the my-service.service.ts file)

In my-service.service.ts, you'll use HttpClient to make requests to the API. The service will have methods to:

- Fetch all data (GET request to get all records).
- Fetch data by ID (GET request to get a specific record by ID).

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

```
private apiUrl = 'https://api.example.com/data';

0 references
constructor(private http: HttpClient) { }

// Fetch all data (GET request)
getAllData(): Observable<any[]> {
  return this.http.get<any[]>(this.apiUrl);  // Send GET request to get all data
}

// Fetch data by ID (GET request)
getDataById(id: number): Observable<any> {
  return this.http.get<any>(`${this.apiUrl}/${id}`);  // Send GET request to get data by ID
}
```

Sends a GET request to the base API URL (https://api.example.com/data) and expects an array of data.

Sends a GET request to the API URL with a specific id appended, returning a single data record.

3- Use the Service in the Component (in the binding.component.ts file)

Inject the service and use the methods to fetch the data.

- Fetch all data (GET request to get all records).
- Fetch data by ID (GET request to get a specific record by ID).

```
import { MyService } from '../../my-service.service';
```

```
export class BindingComponent {
  allData: any[] = [];  // Array to hold all data
  singleData: any;       // Object to hold data for a specific ID

  0 references
  constructor(private myService: MyService) { }

  ngOnInit(): void {
    // Fetch all data on component initialization
    this.myService.getAllData().subscribe((data) => {
      this.allData = data;  // Assign fetched data to allData property
    });

    // Fetch data by ID (example: 1)
    this.myService.getDataById(1).subscribe((data) => {
      this.singleData = data;  // Assign fetched single data to singleData property
    });
  }
}
```

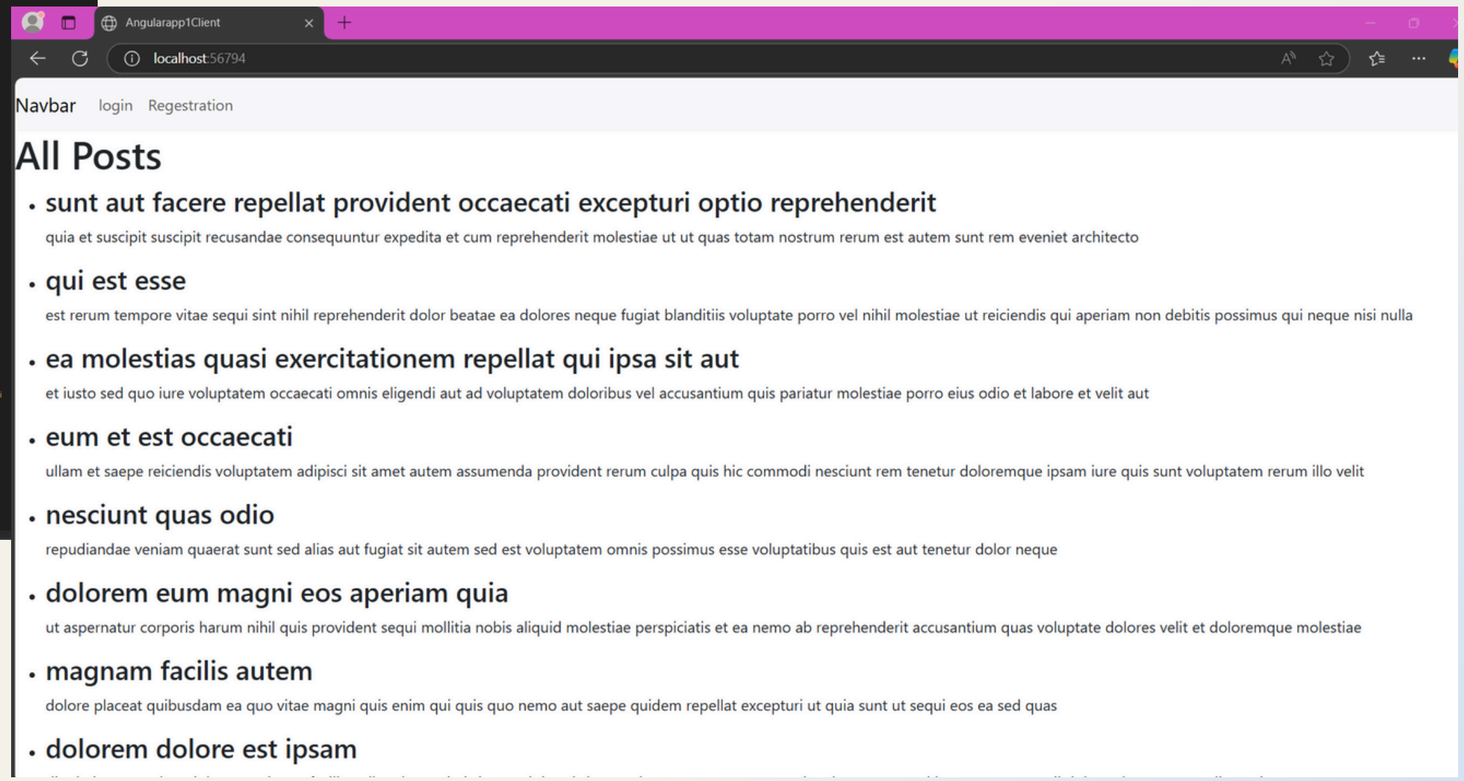
This lifecycle method is called when the component is initialized.

- It calls this.myService.getAllData() to fetch all data and assign it to allData.
- It also calls this.myService.getDataById(1) to fetch a specific record by ID (in this case, ID 1) and assigns it to singleData.

4- Display Data in the HTML Template (in the binding.component.html file)

```html
<h1>All Posts</h1>
<ul>
  <li *ngFor="let item of allData">
    <h3>{{ item.title }}</h3>
    <p>{{ item.body }}</p>
  </li>
</ul>

<h1>Post by ID (ID: 1)</h1>
<div *ngIf="singleData">
  <h3>{{ singleData.title }}</h3>
  <p>{{ singleData.body }}</p>
</div>
```



OBSERVABLES AND THE SUBSCRIBE

In Angular, Observables and the subscribe method are fundamental concepts when working with asynchronous data, especially when making HTTP requests. These concepts are heavily used in Angular's RxJS (Reactive Extensions for JavaScript) library.

1. Observable

An Observable is a stream of data or events that you can observe. It allows you to work with asynchronous data in a very powerful and flexible way.

- Observable is a blueprint for creating streams of data.
- You can think of it like a pipeline that emits values over time.
- Observables can emit three types of events:
    - Next: A new value is emitted by the Observable.
    - Error: Something went wrong, and the Observable emits an error.
    - Complete: The Observable has finished emitting values.

In Angular, Observables are used to handle asynchronous operations like HTTP requests, user input events, WebSockets, etc.

3. Subscribe

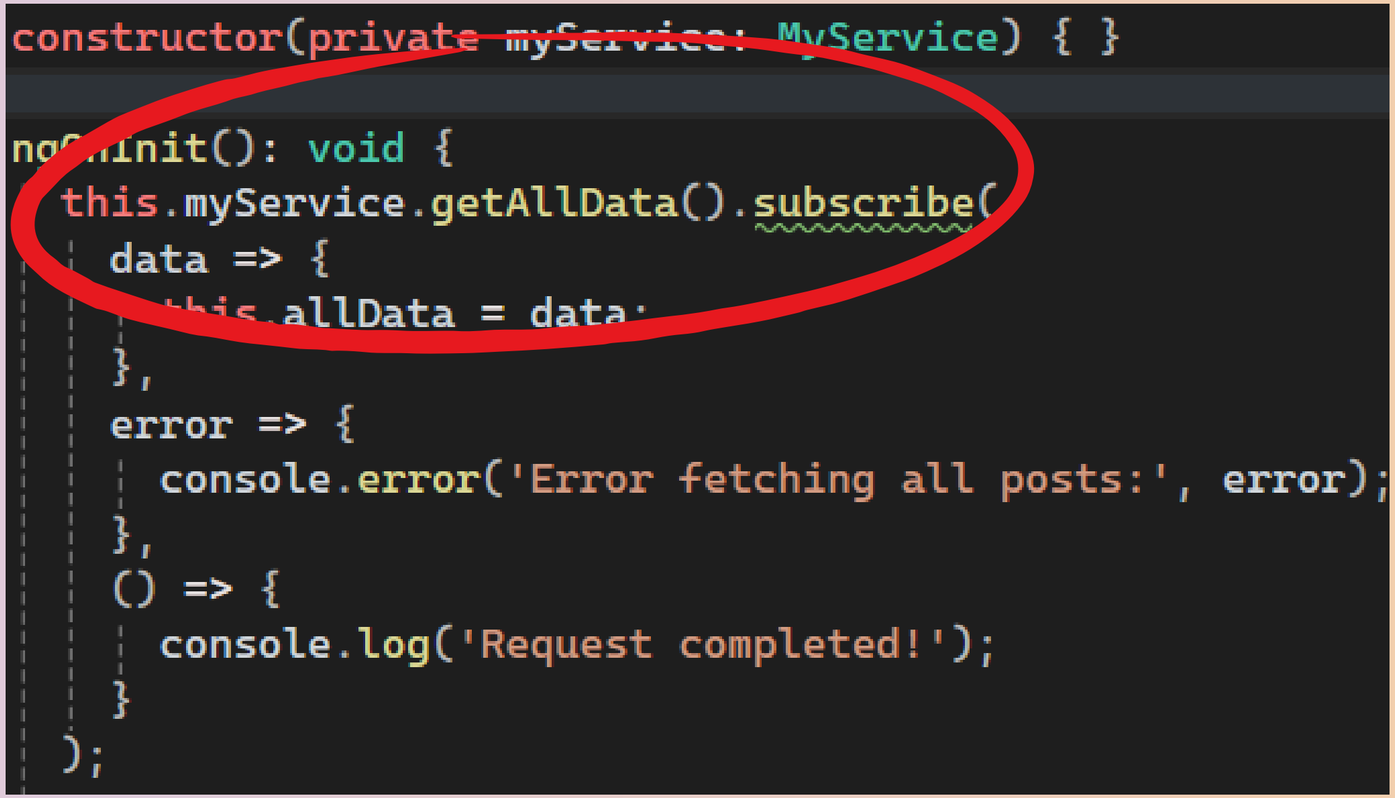
Once you have an Observable, you need to subscribe to it in order to start receiving the data or events that it emits. subscribe() is a method that takes three possible arguments:

1. Next handler: A function that processes each value emitted by the Observable (data received).
2. Error handler: A function that processes errors emitted by the Observable.
3. Complete handler: A function that executes when the Observable finishes emitting values.

In Our Example:

- The Observable is returned by the getAllData() and getDataById() methods from the service. When you call these methods, they return an Observable that represents the data being fetched from the API.

- To start receiving data from the Observable, you need to subscribe to it. The subscribe() method is called on the Observable returned by getAllData() and getDataById(). It listens for the values emitted by the Observable, and executes specific actions (like handling data, errors, or completion) when the values are emitted.



Why Use Observables and Subscribe in Angular?

In Angular, Observables provide a way to handle asynchronous operations and data streams in a consistent, powerful way. Here's why they are useful:

1. Non-blocking Operations: Observables allow Angular to perform HTTP requests and other asynchronous tasks without blocking the rest of your application.
2. Easy Composition: Observables can be combined with other Observables, making it easy to transform, filter, or combine data streams. This is especially useful when dealing with complex data.
3. Cancellation Support: You can unsubscribe from an Observable to cancel HTTP requests or other long-running operations if needed.
4. Reactive Programming: Observables are the foundation for Reactive Programming, where the code reacts to changes in data over time.

THANK YOU FOR LISTENING!