# Angular Components

# Step 1: Understanding Angular Architecture

Angular is built on a **component-based architecture** , which means the application is divided into reusable, self-contained pieces called **components** . These components are organized into **modules** , and they rely on **services** for shared functionality. Let's dive into each part of Angular's architecture.

## 1. Components

A **component** is the most basic building block of an Angular application. It controls a part of the UI and consists of three main parts:

- **Template** : Defines the view (HTML structure).
- **Class** : Contains the logic (TypeScript code).
- **Metadata** : Provides additional information about the component.

**Steps to Create and Understand a Component**

1. **Generate a Component**
    - Use Angular CLI to generate a new component:

## ng generate component my-component

## Shortcut:

## ng g c my-component

**Anatomy of a Component** After generating the component, you'll see the following files in the `src/app/my-component` folder:

- `my-component.component.ts`: The TypeScript file containing the component class.
- `my-component.component.html`: The HTML template file.
- `my-component.component.css`: The CSS file for styling.
- `my-component.component.spec.ts`: The test file.

**Component Class** Open `my-component.component.ts`:

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



@Component({

  selector: 'app-my-component', // Custom HTML tag

  templateUrl: './my-component.component.html', // Template file

  styleUrls: ['./my-component.component.css'] // Styles file

})

export class MyComponent {

  title = 'Hello from MyComponent!';

}
```

- **@Component Decorator** : Defines metadata for the component.
  - `selector`: The custom HTML tag used to include this component in templates.
  - `templateUrl`: Path to the HTML template.
  - `styleUrls`: Array of CSS files for styling.
- **Class** : Contains the logic and data for the component.

**Using the Component**

- Add the component's selector (`<app-my-component>`) to another template (e.g., `app.component.html`):

<h1>Welcome to the App</h1>

<app-my-component></app-my-component>

When you run the app (`ng serve`), you'll see the content of `my-component.component.html`.

## 2. Modules

A **module** is a container for organizing related components, directives, pipes, and services. Every Angular app has at least one module, called the **root module** (`AppModule`).

### Steps to Understand Modules

1. **Root Module (`AppModule`)** Open `src/app/app.module.ts`:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';
import { MyComponent } from './my-component/my-component.component';

@NgModule({
  declarations: [
        AppComponent,
        MyComponent // Declare all components here
  ],
  imports: [
        BrowserModule // Import other modules here
  ],
  providers: [], // Register services here
  bootstrap: [AppComponent] // Root component to bootstrap
})
export class AppModule { }
```

- **@NgModule Decorator** : Defines metadata for the module.
  - `declarations`: List of components, directives, and pipes that belong to this module.
  - `imports`: Other modules required by this module.
  - `providers`: Services available to the entire application.
  - `bootstrap`: The root component to load when the app starts.

**Feature Modules**

- As your app grows, you can create feature modules to organize components logically.
- Generate a feature module:

**ng generate module feature-module**

**Shortcut:**

**ng g m feature-module**

**Shared Modules**

- If you have components, directives, or pipes used across multiple modules, create a shared module:

**ng generate module shared**

**Export reusable components/directives/pipes in the shared module.**

### 3. Templates

A **template** defines the view of a component using HTML. It can include Angular-specific syntax like **data binding** , **directives** , and **pipes** .

**Steps to Work with Templates**

1. **Inline Template** Instead of using `templateUrl`, you can define the template inline:

```
@Component({
  selector: 'app-my-component',
  template: `<h1>{{ title }}</h1>`,
  styleUrls: ['./my-component.component.css']
})
export class MyComponent {
  title = 'Hello from MyComponent!';
}
```

**Data Binding**

- **Interpolation** : Display component properties in the template:

**<h1>{{ title }}</h1>**

**Property Binding** : Bind HTML attributes to component properties:

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

**Event Binding** : Handle user events:

<button (click)="onClick()">Click Me</button>

**Two-Way Binding** : Use `[(ngModel)]` for two-way data binding:

<input [(ngModel)]="name" placeholder="Enter your name">

Hello, {{ name }}!

**Directives**

- **Structural Directives** : Modify the DOM layout (e.g., `*ngIf`, `*ngFor`):

```
<div *ngIf="isVisible">This is visible</div>

<ul>

  <li *ngFor="let item of items">{{ item }}</li>

</ul>
```

**Attribute Directives** : Modify the appearance or behavior of elements (e.g., `ngClass`, `ngStyle`):

```
<div [ngClass]="{ active: isActive }">Active Class</div>
```

## 4. Metadata

**Metadata** provides additional information about a class (e.g., a component or module). It's defined using decorators like `@Component` and `@NgModule`.

#### Examples of Metadata

1. **Component Metadata**

```
@Component({
  selector: 'app-example',
  templateUrl: './example.component.html',
  styleUrls: ['./example.component.css']
})
```

#### Module Metadata

```
@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule],
  providers: [],
  bootstrap: [AppComponent]
})
```

## 5. Services and Dependency Injection

A **service** is a class that provides shared functionality (e.g., fetching data from an API). **Dependency Injection (DI)** is a design pattern used to inject services into components or other services.

#### Steps to Create and Use a Service

1. **Generate a Service**
    - Use Angular CLI to generate a service:

**ng generate service data**

**Shortcut:**

**ng g s data**

**Define the Service** Open `data.service.ts`:

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

@Injectable({
  providedIn: 'root' // Makes the service available app-wide
})
export class DataService {
  getData() {
    return ['Item 1', 'Item 2', 'Item 3'];
  }
}
```

**Inject the Service into a Component** Open `my-component.component.ts`:

```
import { Component } from '@angular/core';
import { DataService } from '../data.service';

@Component({
  selector: 'app-my-component',
  templateUrl: './my-component.component.html',
  styleUrls: ['./my-component.component.css']
})
export class MyComponent {
  items: string[];

  constructor(private dataService: DataService) {
        this.items = this.dataService.getData();
  }
}
```

**Use the Data in the Template** Update `my-component.component.html`:

```
<ul>
  <li *ngFor="let item of items">{{ item }}</li>
</ul>
```

## 6. Angular’s Modular Structure (NgModule)

Angular apps are organized into **modules** , which help manage complexity and enable lazy loading.

### Types of Modules

1. **Root Module (`AppModule`)**
   - The entry point of the application.
   - Defined in `app.module.ts`.
2. **Feature Modules**
   - Organize related components, directives, and pipes.
   - Example: `dashboard.module.ts`.
3. **Shared Modules**
   - Contain reusable components, directives, and pipes.
   - Example: `shared.module.ts`.
4. **Lazy-Loaded Modules**
   - Loaded on demand to improve performance.
   - Example: `admin.module.ts`.

**Steps to Lazy Load a Module**

1. **Create a Feature Module**

```
ng generate module admin --route admin --module app.module
```

**Configure Routing** Open `app-routing.module.ts`:

```
const routes: Routes = [
  { path: 'admin', loadChildren: () => import('./admin/admin.module').then(m => m.AdminModule) }
];
```

**Run the App** Navigate to `/admin` to load the `AdminModule`.

# Step 1: Creating and Registering Components

## What is a Component?

A **component** in Angular is a TypeScript class that controls a part of the UI. It consists of:

- A **template** (HTML structure).
- A **class** (TypeScript logic).
- **Metadata** (decorator `@Component`).

## Steps to Create and Register a Component

1. **Generate a New Component**
    - Use Angular CLI to create a new component:

ng generate component my-component

Shortcut:

ng g c my-component

**Anatomy of a Generated Component** After running the command, Angular generates the following files:

- `my-component.component.ts`: The TypeScript file with the component class.
- `my-component.component.html`: The HTML template.
- `my-component.component.css`: The CSS file for styling.
- `my-component.component.spec.ts`: The test file.

**Register the Component**

- Open `app.module.ts` (or the relevant module file).
- Ensure the component is declared in the `declarations` array:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';
import { MyComponent } from './my-component/my-component.component';
@NgModule({
  declarations: [
        AppComponent,
        MyComponent // Register the component here
  ],
  imports: [BrowserModule],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

**Use the Component in a Template**

- Add the component's selector (defined in `my-component.component.ts`) to another template, such as `app.component.html`:

```
<h1>Welcome to the App</h1>

<app-my-component></app-my-component>
```

## Step 2: Component Lifecycle Hooks

### What are Lifecycle Hooks?

Angular components have a lifecycle managed by Angular itself. Lifecycle hooks allow you to tap into key moments in a component's life, such as initialization, rendering, and destruction.

### Common Lifecycle Hooks

1. **ngOnInit**
    - Called once after the component is initialized.
    - Use it to fetch data or perform setup logic.

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

@Component({
  selector: 'app-my-component',
  template: `<p>{{ message }}</p>`
})
export class MyComponent implements OnInit {
  message: string;
  ngOnInit() {this.message = 'Component Initialized!';}
}
```

**ngOnDestroy**

- Called just before the component is destroyed.
- Use it to clean up resources (e.g., unsubscribe from observables).

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

@Component({
  selector: 'app-my-component',
  template: `<p>Component will be destroyed soon!</p>`
})
export class MyComponent implements OnDestroy {
  ngOnDestroy() {
        console.log('Component Destroyed!');
  }
}
```

**Other Lifecycle Hooks**

- `ngOnChanges`: Called when input properties change.
- `ngAfterViewInit`: Called after the view is fully initialized.
- `ngAfterContentInit`: Called after content projection (`<ng-content>`) is initialized.

# Step 3: Data Binding

## What is Data Binding?

Data binding connects the component class (TypeScript) with its template (HTML). Angular supports four types of data binding:

1. **Interpolation**
   - Binds data from the component class to the template using double curly braces `{{ }}`.

```
export class MyComponent {
  title = 'Hello, Angular!';
}
```

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

**Property Binding**

- Binds a DOM property to a component property using square brackets [ ].

```
export class MyComponent {
  imageUrl = 'https://example.com/image.jpg';
}
```

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

**Event Binding**

- Binds DOM events to component methods using parentheses ( ).

```
export class MyComponent {
  onClick() {
      console.log('Button clicked!');
  }
}
<button (click)="onClick()">Click Me</button>
```

**Two-Way Binding**

- Combines property and event binding using `[(ngModel)]`.
- Requires importing `FormsModule` in your module.

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

@NgModule({
  imports: [BrowserModule, FormsModule], // Import FormsModule
  declarations: [AppComponent, MyComponent],
  bootstrap: [AppComponent]
})
export class AppModule { }

export class MyComponent {
  name = '';
}

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

## Step 4: Input and Output

### What are Input and Output?

- **@Input** : Allows a parent component to pass data to a child component.
- **@Output** : Allows a child component to emit events to a parent component.

### Using @Input

1. **Define an Input Property**

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

@Component({
  selector: 'app-child',
  template: `<p>{{ message }}</p>`
})
export class ChildComponent {
  @Input() message: string;
}
```

**Pass Data from Parent to Child**

**<app-child [message]="'Hello from Parent!'"></app-child>**

**Using @Output**

1. **Define an Output Event**

```
import { Component, Output, EventEmitter } from '@angular/core';
@Component({
  selector: 'app-child',
  template: `<button (click)="sendEvent()">Send Event</button>`
})
export class ChildComponent {
  @Output() notify = new EventEmitter<string>();
  sendEvent() {
        this.notify.emit('Event from Child!');
  }
}
```

**Handle the Event in the Parent**

```
<app-child (notify)="onNotify($event)"></app-child>

export class ParentComponent {
  onNotify(message: string) {
      console.log(message); // Logs: "Event from Child!"
  }
}
```

## Step 5: Content Projection with `<ng-content>`

### What is Content Projection?

Content projection allows you to insert dynamic content into a component's template using `<ng-content>`.

### Steps to Use Content Projection

1. **Define a Component with `<ng-content>`**

```
@Component({
  selector: 'app-card',
  template: `
        <div class="card">
        <ng-content></ng-content>
        </div>
  `,
  styles: ['.card { border: 1px solid black; padding: 10px; }']
})
export class CardComponent {}
```

**Project Content into the Component**

```
<app-card>
  <p>This content is projected into the card!</p>
</app-card>
```

# Step 6: View Encapsulation

## What is View Encapsulation?

View encapsulation ensures that styles defined in a component's CSS file do not leak into other components.

## Types of View Encapsulation

1. **Emulated (Default)** : Styles are scoped to the component but emulated using unique attributes.
2. **None** : Styles are global.
3. **Shadow DOM** : Uses the browser's native Shadow DOM.

## Example

```
import { Component, ViewEncapsulation } from '@angular/core';
@Component({
  selector: 'app-example',
  template: `<p>Styled Paragraph</p>`,
  styles: ['p { color: red; }'],
  encapsulation: ViewEncapsulation.None // Global styles
})
export class ExampleComponent {}
```