UNIT-IV

Express and Angular

Getting Started with Express

Step 1: Set Up Your Project

Open your terminal and create a new project folder:

```
mkdir my-express-app
cd my-express-app
```

Initialize a new Node.js project:

```
npm init -y
```

Install Express:

```
npm install express
```

Step 2: Create a Basic Server

Create a file named app.js (or index.js):

```
// app.js
const express = require('express');
const app = express();
const PORT = 3000;

// Define a simple route
app.get('/', (req, res) => {
  res.send('Hello, Express!');
});

// Start the server
app.listen(PORT, () => {
  console.log(`Server is running on http://localhost:${PORT}`);
});
```

Run the server:

```
node app.js
```

Visit http://localhost:3000 in your browser — you should see **"Hello, Express!"**

Step 3: Add Middleware (Optional, but Common)

For example, to parse JSON data from req.body:

```
app.use(express.json());
```

Step 4: Add More Routes

```
app.get('/about', (req, res) => {
  res.send('This is the About page.');
});

app.post('/data', (req, res) => {
  res.json({ received: req.body });
});
```

Configuring Routes in Express

Routing in **Express.js** defines how your application responds to client requests at different **URL paths** and **HTTP methods** (GET, POST, PUT, DELETE, etc.).

1. Basic Route Setup

```
const express = require('express');
const app = express();
const PORT = 3000;

app.get('/', (req, res) => {
  res.send('Home Page');
});

app.post('/submit', (req, res) => {
  res.send('Form submitted');
});

app.listen(PORT, () => {
  console.log(`Server is running on http://localhost:${PORT}`);
});
```

2. Route Parameters

Capture values from the URL:

```
app.get('/users/:userId', (req, res) => {
  const id = req.params.userId;
  res.send(`User ID is ${id}`);
});
```

3. Query Parameters

Capture values from the query string (?name=John):

```
app.get('/search', (req, res) => {
```

```
  const { name } = req.query;
  res.send(`Searching for: ${name}`);
});
```

4. Modular Routes (Best Practice)

Create a Route File (e.g., routes/userRoutes.js):

```
const express = require('express');
const router = express.Router();

router.get('/', (req, res) => {
  res.send('All Users');
});

router.get('/:id', (req, res) => {
  res.send(`User ID: ${req.params.id}`);
});

module.exports = router;
```

Import and Use the Router in app.js:

```
const express = require('express');
const app = express();
const userRoutes = require('./routes/userRoutes');

app.use('/users', userRoutes); // Mounts all user routes under /users

app.listen(3000, () => {
  console.log('Server running on port 3000');
});
```

5. Handling 404 Routes

```
app.use((req, res) => {
  res.status(404).send('Page not found');
});
```

Using the Request Object (req) in Express

In **Express.js**, the req (request) object represents the **HTTP request** and contains information about the request made by the client, such as parameters, body data, query strings, headers, and more.

Common Properties of req:

Property	Description
req.params	Route parameters (e.g., /users/:id)
req.query	URL query string parameters (e.g., ?search=apple)
req.body	Body data (used in POST/PUT requests – needs middleware

Property	Description
req.header	Request headers
req.metho	HTTP method used (GET, POST, etc.)
req.url	Full URL path of the request
req.cookie	Cookies (requires middleware like cookie-parser)

1. Accessing Route Parameters

```
app.get('/users/:id', (req, res) => {
  const userId = req.params.id;
  res.send(`User ID: ${userId}`);
});
```

2. Accessing Query Strings

URL: /search?term=express

```
app.get('/search', (req, res) => {
  const term = req.query.term;
  res.send(`You searched for: ${term}`);
});
```

3. Accessing Request Body (POST/PUT)

You must use middleware to parse JSON or form data:

```
app.use(express.json()); // For JSON
app.use(express.urlencoded({ extended: true })); // For form data
app.post('/login', (req, res) => {
  const { username, password } = req.body;
  res.send(`Logged in as ${username}`);
});
```

4. Reading Headers

```
app.get('/headers', (req, res) => {
  const userAgent = req.headers['user-agent'];
  res.send(`Your User-Agent is: ${userAgent}`);
});
```

5. Checking HTTP Method or URL

```
app.use((req, res, next) => {
  console.log(`Request Method: ${req.method}, URL: ${req.url}`);
  next();
});
```

Using the Response Object (res) in Express

In **Express.js**, the res (response) object is used to **send data back to the client**. It provides several methods to set status codes, return HTML, JSON, files, redirects, and more.

Common res Methods

Method	Description
res.send()	Sends a response of various types (string, object, etc.
res.json()	Sends a JSON response
res.status()	Sets the HTTP status code
res.redirect()	Redirects the client to a different URL
res.render()	Renders a view (if using a templating engine)
res.sendFile(	Sends a file as a response

1. res.send() – Send Text, HTML, or Simple Data

```
app.get('/', (req, res) => {
  res.send('Welcome to Express!');
});
```

2. res.json() – Send JSON Data

```
app.get('/api/user', (req, res) => {
  res.json({ name: 'Alice', age: 25 });
});
```

3. res.status() – Set Status Code

```
app.get('/not-found', (req, res) => {
  res.status(404).send('Page not found');
});
```

Chain with other methods:

```
res.status(201).json({ message: 'Created successfully' });
```

4. res.redirect() – Redirect to Another URL

```
app.get('/google', (req, res) => {
  res.redirect('https://www.google.com');
});
```

5. res.sendFile() – Serve a File

```
const path = require('path');

app.get('/file', (req, res) => {
  res.sendFile(path.join(__dirname, 'example.pdf'));
});
```

6. res.render() – Render a Template (if using a view engine)

You must set up a view engine like EJS, Pug, or Handlebars:

```
app.set('view engine', 'ejs');

app.get('/profile', (req, res) => {
  res.render('profile', { name: 'John Doe' });
});
```

Angular

Angular is a popular **front-end web application framework** developed and maintained by **Google**. It is widely used for building **dynamic, single-page web applications (SPAs)**. Here's why Angular is important:

1. Component-Based Architecture

- Angular uses a modular design where UI is divided into reusable, encapsulated components. This promotes **maintainability**, **testability**, and **scalability**.

2. Two-Way Data Binding

- Angular enables automatic synchronization between the **model** (data) and the **view** (UI), reducing boilerplate code and making UI updates easier to manage.

3. Dependency Injection (DI)

- It includes a powerful DI system that improves **modularity** and **code reuse**, making it easier to manage and test components.

4. TypeScript Support

- Angular is built with **TypeScript**, which adds static typing to JavaScript, helping developers catch errors early and write more robust code.

5. Built-In Tools

- Angular comes with a complete ecosystem: **Angular CLI** for project scaffolding and development, **RxJS** for reactive programming, and tools for **routing**, **HTTP**, **form handling**, and **testing**.

6. Cross-Platform Development

- Angular supports **progressive web apps (PWAs)** and **native mobile apps** (with frameworks like Ionic), allowing for a unified codebase across platforms.

7. Strong Community and Support

- Maintained by Google and supported by a large community, Angular receives regular updates and improvements, making it a reliable long-term choice.

Key Concepts in Angular

1. Modules

- Angular apps are divided into **NgModules**, which are containers for components, services, and other code.
- The main module is usually AppModule.

2. Components

- Components are the **building blocks** of Angular applications.
- Each component consists of:
  - HTML Template (View)
  - TypeScript Class (Logic/Controller)
  - CSS Styles (Styling)
- Example:

```
@Component({
  selector: 'app-hello',
  template: `<h1>Hello {{name}}</h1>`
})
export class HelloComponent {
  name = 'World';
}
```

3. Templates and Data Binding

- Templates define the **UI** using Angular syntax (directives, expressions).
- Angular supports:
  - **Interpolation**: {{ value }}
  - **Property binding**: [property]="value"
  - **Event binding**: (event)="handler()"
  - **Two-way binding**: [(ngModel)]="value"

4. Directives

- Directives are special markers in templates that tell Angular to **do something with DOM elements**.
  - Examples: *ngIf, *ngFor, ngClass

5. Services and Dependency Injection

- Services are classes for **business logic**, **data handling**, etc.
- Angular uses **Dependency Injection (DI)** to manage how components access services.

6. Routing

- The Angular Router enables navigation between views or pages in a single-page application.

- Configured using the RouterModule.

7. Observables & RxJS

- Angular uses **Observables** for handling asynchronous data streams, particularly in HTTP requests and reactive forms.

Example: Angular in Action

```
@Component({
  selector: 'app-greeter',
  template: `<p>Hello {{name}}!</p>`
})
export class GreeterComponent {
  name = 'Alice';
}
```

This will display: Hello Alice!

Creating a Basic Angular Application

Step 1: Prerequisites

Make sure you have **Node.js** and **npm** installed.

Check using:

```
node -v
npm -v
```

If not installed, download from: https://nodejs.org

Step 2: Install Angular CLI

Angular CLI is a command-line tool for creating and managing Angular apps.

```
npm install -g @angular/cli
```

Step 3: Create a New Angular App

```
ng new my-angular-app
```

CLI will ask:

- Add Angular routing? (choose **Yes** or **No**)
- Which stylesheet format? (choose **CSS** for simplicity)

Then it creates the app in a folder called my-angular-app.

Step 4: Run the Development Server

Navigate into your app folder and run:

```
cd my-angular-app
ng serve
```

Open your browser and visit:
 http://localhost:4200

You should see the default Angular welcome page.

Step 5: Modify the App Component

Let's make a simple change to the homepage.

Open:

src/app/app.component.ts

Change the title:

```
export class AppComponent {
  title = 'My First Angular App';
}
```

Then update the HTML:

src/app/app.component.html

```
<h1>Welcome to {{ title }}!</h1>
```

Now save and refresh the browser — you should see your new message.

Step 6: Add a New Component

Generate a new component called hello:

```
ng generate component hello
```

This creates a new folder src/app/hello with component files.

Update hello.component.ts:

```
export class HelloComponent {
  message = 'Hello from Angular!';
}
```

Update hello.component.html:

{{ message }}

Then include it in your app by adding this tag to app.component.html

<app-hello></app-hello>

Angular Components

A component in Angular is made up of:

1. **Class** – defines the logic (TypeScript)
2. **Template** – defines the HTML view
3. **Styles** – defines the CSS/SCSS for the view
4. **Metadata** – links class, template, and styles together using the @Component decorator

Example of a Simple Component

```
// hello.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-hello',
  templateUrl: './hello.component.html',
  styleUrls: ['./hello.component.css']
})
export class HelloComponent {
  name: string = 'Angular User';
}
<!-- hello.component.html -->
<h2>Hello, {{ name }}!</h2>
/* hello.component.css */
h2 {
  color: blue;
}
```

You can then **use this component** in another component's template by placing:

<app-hello></app-hello>

Component File Structure

When you generate a component using Angular CLI:

ng generate component hello

You get:

hello/

```
├── hello.component.ts       (logic)
├── hello.component.html     (template)
├── hello.component.css      (styles)
├── hello.component.spec.ts  (unit test)
```

Key Features of Angular Components

- **Selector**: Acts like a custom HTML tag to use the component
- **Template**: Defines what's rendered on the screen
- **Data Binding**: Displays dynamic data ({{ }} for interpolation)
- **Inputs & Outputs**: Enables data flow between components
- **Lifecycle Hooks**: Special methods like ngOnInit() let you run code at certain points

Component Communication

1. **Parent to Child** – using @Input()
2. **Child to Parent** – using @Output() and EventEmitter
3. **Service-based Communication** – via shared services and observables

1. Angular Expressions

Angular expressions are code snippets you write in templates (HTML) to **display dynamic data** from the component class.

They look like JavaScript, but they are **evaluated in the Angular context**, meaning they can only access:

- Component properties and methods
- Global Angular-safe objects (like Date)
- No access to the window, document, or console objects (for security)

Basic Syntax

{{ expression }}

Example:

```
<p>{{ 10 + 5 }}</p>            <!-- 15 -->
<p>{{ username }}</p>          <!-- Displays the value of username -->
<p>{{ user.age >= 18 ? 'Adult' : 'Minor' }}</p>
```

Features of Angular Expressions

1. **One-way binding** from component → view
2. Can include:
    - Arithmetic: {{ price * quantity }}
    - String operations: {{ 'Hello ' + name }}
    - Function calls: {{ getStatus() }}
    - Conditional logic: {{ isActive ? 'Yes' : 'No' }}
    - Object access: {{ user.name }}

3. Can be used **only in templates** – not in the component class.

Limitations of Angular Expressions

- No access to:
  - Global objects (window, document, etc.)
  - Loops (e.g. for, while)
  - Assignment (=)
- Should not have side effects (no changing variables, only reading)

Example in a Component

app.component.ts

```
export class AppComponent {
  username = 'Alice';
  price = 100;
  quantity = 2;

  getTotal() {
    return this.price * this.quantity;
  }
}
```

app.component.html

```
<p>Hello, {{ username }}!</p>
<p>Total: {{ price * quantity }}</p>
<p>Total using method: {{ getTotal() }}</p>
```

Data binding

Data binding in Angular connects your component's **logic (TypeScript)** with the **view (HTML template)**. It ensures that data can flow **from the component to the UI** and **from the UI back to the component**.

Angular provides **four types of data binding**:

1. Interpolation (One-way binding: Component → View)

Displays dynamic values in the HTML using {{ }}.

Syntax:

```
{{ propertyName }}
```

Example:

```
// app.component.ts
export class AppComponent {
  name = 'Angular';
```

```
}
<!-- app.component.html -->
<h1>Welcome, {{ name }}!</h1>
```

2. Property Binding (One-way binding: Component → DOM property)

Binds a DOM element property to a component property.

Syntax:

```
[property]="componentProperty"
```

Example:

```
<img [src]="imageUrl" alt="Logo">
<button [disabled]="isDisabled">Submit</button>
```

3. Event Binding (One-way binding: View → Component)

Handles events from the DOM and calls component methods.

Syntax:

```
(event)="method()"
```

Example:

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

onClick() {
  alert('Button clicked!');
}
```

4. Two-Way Binding (View ↔ Component)

Keeps form inputs and component properties in sync using [(ngModel)].

Important: Requires importing FormsModule in app.module.ts.

Syntax:

```
[(ngModel)]="property"
```

Example:

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

Summary Table

Binding Type	Direction	Syntax	Use Case
Interpolation	Component → View	{{ value }}	Display text/data
Property Binding	Component → DOM	[property]="value"	Set image src, button disable
Event Binding	View → Component	(event)="handler()"	Respond to user input
Two-Way Bindin	View ↔ Component	[(ngModel)]="value"	Form controls

Built-in Directives in Angular

In Angular, **directives** are special instructions in the DOM that tell Angular how to manipulate elements. There are **three types** of directives:

Types of Directives

Type	Purpose	Example
Componen	A directive with a template	@Component
Structural	Changes the **DOM layout** (adds/removes elements	*ngIf, *ngFor
Attribute	Changes the **appearance/behavior** of elements	ngClass, ngStyl

1. Structural Directives

They use * prefix and **modify the DOM structure**.

***ngIf – conditionally include elements**

```
<p *ngIf="isLoggedIn">Welcome back!</p>
```

***ngFor – loop over a list**

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

***ngSwitch – switch-case logic**

```
<div [ngSwitch]="status">
  <p *ngSwitchCase="'active'">Active</p>
  <p *ngSwitchCase="'inactive'">Inactive</p>
  <p *ngSwitchDefault>Unknown</p>
</div>
```

2. Attribute Directives

They **modify the appearance or behavior** of an element.

ngClass – set CSS classes dynamically

```
<p [ngClass]="{ 'highlight': isImportant }">Message</p>
```

ngStyle – set inline styles dynamically

```
<p [ngStyle]="{ color: isError ? 'red' : 'green' }">Status</p>
```

3. Component Directives

These are custom directives defined using @Component. Any Angular component is technically a directive with a template.

Example:

```
@Component({
  selector: 'app-user',
  template: `<p>User: {{ name }}</p>`
})
export class UserComponent {
  name = 'Alice';
}
```

Used in HTML as:

```
<app-user></app-user>
```

Custom Directives in Angular

Custom directives in Angular allow you to extend HTML by creating **your own behaviors** that can be applied to DOM elements. These are typically **attribute directives** (used like ngClass or ngStyle) or **structural directives** (like *ngIf).

Why Use Custom Directives?

- Encapsulate and reuse behavior
- Add dynamic styling or logic to elements
- Manipulate the DOM in a clean, reusable way

Example: Create a Custom Attribute Directive

Let's create a simple directive that **highlights text** when hovered.

Step 1: Generate the Directive

ng generate directive highlight

This creates:

src/app/highlight.directive.ts

Step 2: Edit highlight.directive.ts

```
import { Directive, ElementRef, HostListener, Renderer2 } from '@angular/core';

@Directive({
  selector: '[appHighlight]'  // Use this as an attribute
})
export class HighlightDirective {

  constructor(private el: ElementRef, private renderer: Renderer2) {}

  @HostListener('mouseenter') onMouseEnter() {
    this.renderer.setStyle(this.el.nativeElement, 'backgroundColor', 'yellow');
  }

  @HostListener('mouseleave') onMouseLeave() {
    this.renderer.removeStyle(this.el.nativeElement, 'backgroundColor');
  }
}
```

Step 3: Use the Directive in HTML

In your component template:

```
<p appHighlight>This text will be highlighted on hover!</p>
```

Key Concepts Used

Concept	Description
@Directive	Marks the class as a directive
selector	How you apply the directive in templates
ElementRef	Gives direct access to the DOM element
Renderer2	Safely modifies DOM styles
@HostListene	Listens to DOM events (like mouseenter

Optional: Pass Input to the Directive

You can make your directive dynamic:

```
@Input('appHighlight') highlightColor: string;

@HostListener('mouseenter') onEnter() {
  this.renderer.setStyle(this.el.nativeElement, 'backgroundColor', this.highlightColor || 'yellow');
}
<p [appHighlight]='"lightblue"'>Hover me for blue highlight</p>
```

Implementing Angular Services in Web Applications

In Angular, **services** are used to organize and share business logic, data, and functions across components. They help you **separate concerns**, promote **code reuse**, and support **dependency injection**.

What Is an Angular Service?

An Angular **service** is a class with a specific purpose, like:

- Fetching data (e.g., via HTTP)
- Logging
- Authentication
- Utility functions
- State management

Step-by-Step: Creating and Using a Service

Step 1: Generate a Service

ng generate service data

This creates:

src/app/data.service.ts

Step 2: Define the Service Logic

```
// data.service.ts
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'  // Automatically provides it at the app level
})
export class DataService {
  private data: string[] = ['Angular', 'React', 'Vue'];

  getData() {
    return this.data;
  }

  addData(item: string) {
    this.data.push(item);
  }
}
```

Step 3: Inject and Use the Service in a Component

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

@Component({
  selector: 'app-root',
  template: `
    <h2>Frameworks</h2>
    <ul>
```

```
    <li *ngFor="let item of frameworks">{{ item }}</li>
   </ul>
   <button (click)="addFramework()">Add Svelte</button>
  `
})
export class AppComponent {
  frameworks: string[] = [];

  constructor(private dataService: DataService) {}

  ngOnInit() {
    this.frameworks = this.dataService.getData();
  }

  addFramework() {
    this.dataService.addData('Svelte');
    this.frameworks = this.dataService.getData();  // Refresh view
  }
}
```

What @Injectable({ providedIn: 'root' }) Does

This tells Angular to **register the service with the root injector**, making it **a singleton** and available application-wide without needing to declare it in providers manually.

Common Use Cases for Services

Use Case	How Service Helps
HTTP requests	Centralize API calls using HttpClien
Shared state	Store and access shared data
Business logic	Move logic out of components
Utility function	Reuse common helper methods