**Introduction to Angular and Setting Up the Development Environment**

**What is Angular?**

Angular is a **TypeScript-based** open-source framework developed by Google for building **single-page applications (SPAs)**. It provides a structured way to build scalable web applications using components, services, and dependency injection.

**Key Features of Angular**

* **Component-based architecture** – Applications are broken down into reusable components.
* **Dependency Injection (DI)** – Manages dependencies and improves code reusability.
* **Powerful routing** – Built-in routing module for navigation.
* **Forms and validation** – Supports template-driven and reactive forms
* **State management** – Uses services and NgRx for managing application state.
* **Built-in directives and pipes** – Helps manipulate DOM efficiently.
* **HTTP Client module** – Handles API requests seamlessly.
* **RxJS (Reactive Extensions for JavaScript)** – Supports reactive programming.

**Setting Up the Development Environment**

**1. Install Node.js**

Angular requires **Node.js** because it includes npm (Node Package Manager), which helps install dependencies.

* Download and install **Node.js** from [nodejs.org](https://nodejs.org/)
* Verify installation using:
* node -v
* npm -v

**2. Install Angular CLI**

Angular CLI (Command Line Interface) simplifies Angular development by automating project setup, building, and deployment.

* Install Angular CLI globally:

npm install -g @angular/cli

* Check installation:

ng version

**3. Create a New Angular Project**

Run the following command to create a new project:

ng new my-angular-app

* It will ask for:
  + **Would you like to add Angular routing?** → Choose **Yes** if you want routing.
  + **Which stylesheet format would you like to use?** → Choose **CSS** or **SCSS** based on preference.

Navigate into the project folder:

cd my-angular-app

**4. Run the Angular Application**

Start the development server:

ng serve

* Open http://localhost:4200/ in your browser to see the default Angular welcome page.

**Understanding Angular Project Structure**

Inside the project folder, you'll find:

* **src/** – Main source code.
  + **app/** – Contains all components, modules, and services.
  + **assets/** – Static files like images.
  + **environments/** – Environment-specific configurations.
* **angular.json** – Angular project configuration.
* **package.json** – Dependencies and scripts.
* **node\_modules/** – Installed npm packages.
* **tsconfig.json** – TypeScript configuration.

**Building an Angular Application**

To build a production-ready app:

ng build --prod

This generates optimized files in the dist/ folder, ready for deployment.

That’s the basic setup! Next, I'll explain **TypeScript Essentials for Angular**, covering types, interfaces, and classes. Let me know if you have any questions so far.

**TypeScript Essentials for Angular**

Since Angular is built using **TypeScript**, understanding its core features is essential for writing clean and maintainable Angular code.

**1. Basic Types**

TypeScript introduces strict typing, which helps catch errors at compile time.

let num: number = 10;

let str: string = "Hello, Angular";

let isAvailable: boolean = true;

Other types include null, undefined, bigint, and symbol.

**2. Type Annotations**

Annotations explicitly specify variable types.

function add(a: number, b: number): number {

return a + b;

}

Here, both a and b must be numbers, and the function returns a number.

**3. Interfaces and Type Aliases**

Interfaces define object structures.

interface User {

name: string;

age: number;

}

let user: User = { name: "John", age: 25 };

Type Aliases work similarly but allow more flexibility.

type ID = string | number;

let userId: ID = 123; // Or "ABC123"

**4. Functions**

Functions in TypeScript support optional, default, and rest parameters.

function greet(name: string = "Guest"): string {

return `Hello, ${name}`;

}

Arrow functions:

const multiply = (a: number, b: number): number => a \* b;

**5. Classes**

TypeScript supports object-oriented programming with classes, inheritance, and access modifiers.

class Person {

constructor(public name: string, private age: number) {}

greet() {

console.log(`Hello, my name is ${this.name}.`);

}

}

const john = new Person("John", 30);

john.greet();

* public: Accessible anywhere.
* private: Accessible only within the class.
* protected: Accessible within the class and subclasses.

**6. Modules and Imports/Exports**

TypeScript allows modular programming using import and export.

// file: mathUtils.ts

export function add(a: number, b: number): number {

return a + b;

}

// file: app.ts

import { add } from "./mathUtils";

console.log(add(5, 3));

**7. Generics**

Generics allow reusable functions or classes that work with multiple types.

function identity<T>(arg: T): T {

return arg;

}

console.log(identity<string>("Hello"));

console.log(identity<number>(42));

**8. Union and Intersection Types**

* **Union (|)**: A variable can have multiple types.
* **Intersection (&)**: Combines multiple types into one.

type Admin = { role: string };

type Employee = { id: number };

type AdminEmployee = Admin & Employee;

let adminEmp: AdminEmployee = { role: "Manager", id: 101 };

**9. Type Assertions and Type Guards**

* **Type Assertions**: Tells TypeScript that we know the type better.

let value: any = "Angular";

let length: number = (value as string).length;

* **Type Guards**: Ensures the correct type is used at runtime.

function printID(id: string | number) {

if (typeof id === "string") {

console.log(id.toUpperCase());

} else {

console.log(id);

}

}

**10. Asynchronous Programming**

Angular uses asynchronous programming for API calls and event handling.

**Callbacks**

function fetchData(callback: (data: string) => void) {

setTimeout(() => {

callback("Data received");

}, 2000);

}

fetchData((data) => console.log(data));

**Promises**

function getData(): Promise<string> {

return new Promise((resolve) => {

setTimeout(() => resolve("Promise resolved"), 2000);

});

}

getData().then(console.log);

**Async/Await**

async function fetchDataAsync() {

let data = await getData();

console.log(data);

}

fetchDataAsync();

This completes the **TypeScript Essentials** section. Now, we can move on to **Angular Components** in the next section. Let me know if you have any questions before proceeding!

**Angular Components**

In Angular, components are the **building blocks** of an application. Each component represents a part of the UI and is responsible for displaying data and handling user interactions.

**1. Creating Components**

A component consists of three main files:

1. **HTML Template (.html)** – Defines the UI.
2. **CSS (.css or .scss)** – Styles the component.
3. **TypeScript (.ts)** – Contains the logic.

**Creating a Component Using Angular CLI**

ng generate component my-component

or

ng g c my-component

This command creates:

src/app/my-component/

- my-component.component.ts

- my-component.component.html

- my-component.component.css

- my-component.component.spec.ts

**2. Component Decorator (@Component)**

Every component in Angular is a **TypeScript class** decorated with @Component.

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

@Component({

selector: 'app-my-component', // HTML tag for this component

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

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

})

export class MyComponent {

title: string = 'Hello Angular';

}

To use this component, add its selector (app-my-component) inside a parent component's template:

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

**3. Component Interaction (Data Binding)**

Data binding enables communication between the component and the template.

**a. Property Binding ([property])**

Binds a property from the component to an HTML element.

<h1 [innerText]="title"></h1>

**b. Event Binding ((event))**

Binds an event (e.g., click) to a method in the component.

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

showMessage() {

alert('Button clicked!');

}

**c. Two-Way Binding ([(ngModel)])**

Syncs data between the component and the UI.

<input [(ngModel)]="name" />

<p>Hello, {{ name }}!</p>

Requires **FormsModule** in app.module.ts:

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

**4. Component Lifecycle Hooks**

Angular provides lifecycle hooks to execute code at different stages of a component’s lifecycle.

**Common Lifecycle Hooks**

**Example**

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

@Component({

selector: 'app-lifecycle',

template: `<h1>Lifecycle Hooks Example</h1>`

})

export class LifecycleComponent implements OnInit, OnDestroy {

ngOnInit() {

console.log('Component Initialized');

}

ngOnDestroy() {

console.log('Component Destroyed');

}

}

**5. Parent-Child Component Communication**

**a. @Input() – Passing Data from Parent to Child**

**Parent Component (app.component.html)**

<app-child [message]="parentMessage"></app-child>

**Parent Component (app.component.ts)**

parentMessage = "Hello from Parent";

**Child Component (child.component.ts)**

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

@Component({

selector: 'app-child',

template: `<p>Message from parent: {{ message }}</p>`

})

export class ChildComponent {

@Input() message!: string;

}

**b. @Output() – Emitting Data from Child to Parent**

**Child Component (child.component.ts)**

import { Component, Output, EventEmitter } from '@angular/core';

@Component({

selector: 'app-child',

template: `<button (click)="sendMessage()">Send Message</button>`

})

export class ChildComponent {

@Output() messageEvent = new EventEmitter<string>();

sendMessage() {

this.messageEvent.emit("Hello Parent!");

}

}

**Parent Component (app.component.html)**

<app-child (messageEvent)="receiveMessage($event)"></app-child>

<p>Message from child: {{ childMessage }}</p>

**Parent Component (app.component.ts)**

childMessage!: string;

receiveMessage(event: string) {

this.childMessage = event;

}

This covers **Angular Components** in detail. Next, I’ll explain **Directives and Pipes**. Let me know if you have any doubts before moving ahead!

**Directives and Pipes in Angular**

**1. Directives in Angular**

Directives are used to **manipulate the DOM** and **change the appearance or behavior** of elements in Angular. There are **three types** of directives:

1. **Component Directives** – These are actually Angular components (since they use @Component).
2. **Structural Directives** – Modify the DOM structure (e.g., ngIf, ngFor, ngSwitch).
3. **Attribute Directives** – Change the appearance or behavior of elements (e.g., ngClass, ngStyle, ngModel).

**2. Structural Directives**

These directives **add or remove elements** from the DOM dynamically. They are prefixed with \*.

**a. ngIf – Conditional Rendering**

<p \*ngIf="isVisible">This paragraph is visible</p>

<button (click)="isVisible = !isVisible">Toggle Visibility</button>

**Component:**

isVisible = true;

**b. ngFor – Looping Through Lists**

<ul>

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

</ul>

**Component:**

items = ["Apple", "Banana", "Cherry"];

**c. ngSwitch – Conditional Rendering Based on a Value**

<div [ngSwitch]="color">

<p \*ngSwitchCase="'red'">Red Selected</p>

<p \*ngSwitchCase="'blue'">Blue Selected</p>

<p \*ngSwitchDefault>Other Color Selected</p>

</div>

**Component:**

color = "red";

**3. Attribute Directives**

**a. ngClass – Dynamic Class Binding**

<p [ngClass]="{ 'text-danger': isError, 'text-success': !isError }">

This text changes color based on a condition.

</p>

<button (click)="isError = !isError">Toggle Error</button>

**Component:**

isError = true;

**CSS:**

.text-danger { color: red; }

.text-success { color: green; }

**b. ngStyle – Dynamic Style Binding**

<p [ngStyle]="{ 'color': isError ? 'red' : 'green', 'font-size': '20px' }">

This text changes style dynamically.

</p>

**c. ngModel – Two-Way Data Binding (Requires FormsModule)**

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

<p>Hello, {{ name }}!</p>

**Component:**

name = "";

**4. Creating Custom Directives**

A custom directive extends the functionality of Angular's built-in directives.

**Example: Custom Highlight Directive**

**Generate Directive:**

ng g directive highlight

**Directive Code (highlight.directive.ts):**

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

@Directive({

selector: '[appHighlight]'

})

export class HighlightDirective {

@Input() highlightColor = 'yellow';

constructor(private el: ElementRef) {}

@HostListener('mouseenter') onMouseEnter() {

this.el.nativeElement.style.backgroundColor = this.highlightColor;

}

@HostListener('mouseleave') onMouseLeave() {

this.el.nativeElement.style.backgroundColor = null;

}

}

**Usage in Template (app.component.html):**

<p appHighlight highlightColor="lightblue">Hover over me!</p>

**5. Pipes in Angular**

Pipes transform **data formats** (e.g., date, currency, uppercase).

**a. Built-in Pipes**

Example:

<p>Uppercase: {{ 'angular' | uppercase }}</p>

<p>Date: {{ today | date:'fullDate' }}</p>

<p>Currency: {{ 2500 | currency:'USD' }}</p>

**Component:**

today = new Date();

**6. Creating a Custom Pipe**

A custom pipe allows complex transformations beyond built-in pipes.

**Example: Reverse String Pipe**

**Generate Pipe:**

ng g pipe reverse

**Pipe Code (reverse.pipe.ts):**

import { Pipe, PipeTransform } from '@angular/core';

@Pipe({

name: 'reverse'

})

export class ReversePipe implements PipeTransform {

transform(value: string): string {

return value.split('').reverse().join('');

}

}

**Usage in Template:**

<p>Reversed: {{ 'Angular' | reverse }}</p>

**Summary**

* **Directives** modify the DOM structure (ngIf, ngFor, ngClass, ngStyle).
* **Custom Directives** allow adding reusable behaviors (e.g., a highlight effect).
* **Pipes** transform data (uppercase, date, currency).
* **Custom Pipes** extend built-in pipes (e.g., reversing a string).

That concludes **Directives and Pipes**! Next, I'll explain **Angular Forms (Template-driven and Reactive Forms).** Let me know if you have any questions before we move forward.

**Angular Forms**

Forms in Angular allow users to enter and submit data efficiently. Angular provides two ways to handle forms:

1. **Template-Driven Forms** (Simple, works with ngModel)
2. **Reactive Forms** (More flexible, works with FormControl, FormGroup)

**1. Template-Driven Forms**

Template-driven forms are **declarative** and rely on **Angular directives** like ngModel. They are simpler but offer less control compared to reactive forms.

**Setting Up Template-Driven Forms**

To use ngModel, import **FormsModule** in app.module.ts:

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

@NgModule({

imports: [FormsModule]

})

export class AppModule { }

**Example of a Template-Driven Form**

<form #myForm="ngForm" (ngSubmit)="onSubmit(myForm)">

<label>Name:</label>

<input type="text" name="name" [(ngModel)]="user.name" required />

<label>Email:</label>

<input type="email" name="email" [(ngModel)]="user.email" required />

<label>Age:</label>

<input type="number" name="age" [(ngModel)]="user.age" />

<button type="submit" [disabled]="!myForm.valid">Submit</button>

</form>

**Component Code:**

export class AppComponent {

user = { name: '', email: '', age: null };

onSubmit(form: any) {

console.log('Form Submitted:', form.value);

}

}

**Validation in Template-Driven Forms**

<input type="email" name="email" [(ngModel)]="user.email" required #email="ngModel"/>

<span \*ngIf="email.invalid && email.touched">Invalid Email</span>

**2. Reactive Forms**

Reactive forms are **more powerful**, using FormControl and FormGroup for better validation and dynamic updates.

**Setting Up Reactive Forms**

Import **ReactiveFormsModule** in app.module.ts:

import { ReactiveFormsModule } from '@angular/forms';

@NgModule({

imports: [ReactiveFormsModule]

})

export class AppModule { }

**Example of a Reactive Form**

<form [formGroup]="userForm" (ngSubmit)="onSubmit()">

<label>Name:</label>

<input type="text" formControlName="name" />

<label>Email:</label>

<input type="email" formControlName="email" />

<button type="submit" [disabled]="userForm.invalid">Submit</button>

</form>

**Component Code:**

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

import { FormGroup, FormControl } from '@angular/forms';

export class AppComponent {

userForm = new FormGroup({

name: new FormControl(''),

email: new FormControl('')

});

onSubmit() {

console.log('Form Data:', this.userForm.value);

}

}

**3. Form Validation in Reactive Forms**

**Built-in Validators**

Import Validators from Angular:

import { Validators } from '@angular/forms';

userForm = new FormGroup({

name: new FormControl('', [Validators.required, Validators.minLength(3)]),

email: new FormControl('', [Validators.required, Validators.email])

});

**Custom Validator Example**

import { AbstractControl, ValidatorFn } from '@angular/forms';

export function forbiddenNameValidator(name: string): ValidatorFn {

return (control: AbstractControl) => {

return control.value === name ? { forbiddenName: { value: control.value } } : null;

};

}

Usage in Form:

name: new FormControl('', [forbiddenNameValidator('Admin')])

**4. FormBuilder for Simplified Reactive Forms**

Instead of manually creating form controls, **FormBuilder** makes it easier.

import { FormBuilder, FormGroup } from '@angular/forms';

constructor(private fb: FormBuilder) {

this.userForm = this.fb.group({

name: [''],

email: ['']

});

}

**5. FormArray for Dynamic Forms**

Allows adding/removing form controls dynamically.

**Example:**

import { FormArray } from '@angular/forms';

userForm = new FormGroup({

names: new FormArray([])

});

addName() {

(this.userForm.get('names') as FormArray).push(new FormControl(''));

}

**Summary**

* **Template-Driven Forms**: Simpler, uses ngModel.
* **Reactive Forms**: More flexible, uses FormControl, FormGroup, Validators.
* **FormBuilder** simplifies Reactive Forms.
* **FormArray** allows dynamic form controls.

**Next Topic: Dependency Injection and Services in Angular**

Let me know if you have any doubts before moving ahead!

**Dependency Injection and Services in Angular**

Dependency Injection (DI) is a **design pattern** that makes it easy to **manage dependencies** and improve the **modularity and testability** of an application. Angular provides **a built-in DI system** that allows components to request the services they need.

**1. What is Dependency Injection?**

* DI is a design pattern where **dependencies (services or objects) are injected** into a class instead of creating them manually.
* Angular’s DI system **automatically provides required services** to components, directives, and other services.
* Helps in **decoupling** code, making it more modular and testable.

**2. Creating and Using a Service in Angular**

A **service** is a class with **business logic** that can be shared across multiple components. Services are registered using Angular’s **DI system**.

**Step 1: Generate a Service**

ng g service myService

**Step 2: Implement the Service (my-service.service.ts)**

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

@Injectable({

providedIn: 'root' // Registers this service globally (Singleton)

})

export class MyService {

constructor() {}

getMessage(): string {

return "Hello from MyService!";

}

}

Here, @Injectable({ providedIn: 'root' }) makes the service available **application-wide**.

**3. Injecting a Service into a Component**

Now, we **inject the service** into a component using **constructor injection**.

**Step 3: Use the Service in a Component (app.component.ts)**

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

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

@Component({

selector: 'app-root',

template: `<p>{{ message }}</p>`

})

export class AppComponent {

message: string = '';

constructor(private myService: MyService) {

this.message = this.myService.getMessage();

}

}

**What happens here?**

* Angular **automatically injects** MyService into the component.
* We call getMessage() from the service and store its result in message.
* The component **does not create the service instance**; Angular does it.

**4. Understanding the Provider Scope in Services**

**How to Register Services?**

A service can be registered in different ways:

**Example: Service Registered in a Module**

import { NgModule } from '@angular/core';

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

@NgModule({

providers: [MyService]

})

export class SomeModule {}

**Example: Service Registered in a Component**

@Component({

selector: 'app-some',

template: `<p>{{ message }}</p>`,

providers: [MyService] // Service is only available inside this component

})

export class SomeComponent {

message: string;

constructor(private myService: MyService) {

this.message = myService.getMessage();

}

}

Here, a new instance of MyService is **created only for this component** and **its child components**.

**5. Hierarchical Dependency Injection**

Angular follows a **hierarchical DI system**, meaning:

* Services **registered at a module level** are available **application-wide**.
* Services **registered at a component level** are **isolated** and used only within that component and its children.

**Example of Hierarchical Injection:**

@Component({

selector: 'app-parent',

template: `<app-child></app-child>`,

providers: [MyService] // New instance of MyService is created for this component and its children

})

export class ParentComponent {}

@Component({

selector: 'app-child',

template: `<p>{{ message }}</p>`

})

export class ChildComponent {

constructor(private myService: MyService) {

console.log("ChildComponent received service instance");

}

}

* **ParentComponent and ChildComponent share the same instance of MyService.**
* If MyService was registered in app.module.ts, then **all components** in the app would share a **single instance** of MyService.

**6. Dependency Injection with Parameters (Injecting Dependencies into a Service)**

A service can **depend on another service**.

**Example: Service A Uses Service B**

**Step 1: Create a Second Service (logger.service.ts)**

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

@Injectable({

providedIn: 'root'

})

export class LoggerService {

log(message: string) {

console.log("LoggerService:", message);

}

}

**Step 2: Inject LoggerService into MyService**

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

import { LoggerService } from './logger.service';

@Injectable({

providedIn: 'root'

})

export class MyService {

constructor(private logger: LoggerService) {}

getMessage(): string {

this.logger.log("MyService called getMessage()");

return "Hello from MyService!";

}

}

Now, when MyService.getMessage() is called, it **uses** LoggerService.log().

**7. Injecting External APIs with Dependency Injection**

Angular can **inject third-party services** (like HttpClient).

Example: Injecting HttpClient for API calls:

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

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

@Injectable({

providedIn: 'root'

})

export class ApiService {

constructor(private http: HttpClient) {}

getData() {

return this.http.get('<https://jsonplaceholder.typicode.com/posts>');

}

}

Here, HttpClient is automatically injected into ApiService.

**8. Summary of Dependency Injection and Services**

✔ **Services** store **shared logic** and **business functions**.

✔ @Injectable({ providedIn: 'root' }) registers a service **globally** (Singleton).

✔ Services can be **provided at module or component level** for different scopes.

✔ **Hierarchical DI** ensures **scoped service instances** when needed.

✔ Services can **inject other services**, like HttpClient.

**Next Topic: Angular Routing and Navigation**

Let me know if you have any doubts before moving forward!

**Angular Routing and Navigation**

Routing in Angular allows users to navigate between different pages or views without refreshing the entire application. Angular's **Router Module** helps manage navigation efficiently using URLs.

**1. Setting Up Routing in Angular**

**Step 1: Enable Routing When Creating a New Project**

ng new my-app --routing

This automatically creates app-routing.module.ts.

**Step 2: Define Routes in app-routing.module.ts**

import { NgModule } from '@angular/core';

import { RouterModule, Routes } from '@angular/router';

import { HomeComponent } from './home/home.component';

import { AboutComponent } from './about/about.component';

const routes: Routes = [

{ path: '', component: HomeComponent }, // Default Route

{ path: 'about', component: AboutComponent } // Route for About Page

];

@NgModule({

imports: [RouterModule.forRoot(routes)],

exports: [RouterModule]

})

export class AppRoutingModule {}

Here:

* path: '' → Loads HomeComponent by default.
* path: 'about' → Loads AboutComponent when navigating to /about.

**Step 3: Add Router Outlet in app.component.html**

<nav>

<a routerLink="/">Home</a>

<a routerLink="/about">About</a>

</nav>

<router-outlet></router-outlet>

* <router-outlet> **loads the component** based on the current route.
* routerLink="/" → Navigates to the **Home** page.
* routerLink="/about" → Navigates to the **About** page.

**Step 4: Configure Navigation in Components**

You can navigate using the **Router Service**:

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

import { Router } from '@angular/router';

@Component({

selector: 'app-home',

template: `<button (click)="goToAbout()">Go to About</button>`

})

export class HomeComponent {

constructor(private router: Router) {}

goToAbout() {

this.router.navigate(['/about']);

}

}

* this.router.navigate(['/about']) programmatically navigates to the About page.

**2. Route Parameters and Query Parameters**

**Route Parameters (Dynamic Routing)**

Used when passing values in the URL, like /user/1.

**Step 1: Define a Route with a Parameter**

const routes: Routes = [

{ path: 'user/:id', component: UserComponent }

];

**Step 2: Read Route Parameters in UserComponent**

import { ActivatedRoute } from '@angular/router';

export class UserComponent {

userId: string = '';

constructor(private route: ActivatedRoute) {

this.route.paramMap.subscribe(params => {

this.userId = params.get('id')!;

});

}

}

When navigating to /user/1, userId will be 1.

**Query Parameters**

Used to pass optional data like /products?category=electronics.

**Example: Navigate with Query Parameters**

this.router.navigate(['/products'], { queryParams: { category: 'electronics' } });

**Example: Read Query Parameters**

import { ActivatedRoute } from '@angular/router';

export class ProductsComponent {

category: string = '';

constructor(private route: ActivatedRoute) {

this.route.queryParamMap.subscribe(params => {

this.category = params.get('category')!;

});

}

}

If the URL is /products?category=electronics, category will be electronics.

**3. Nested Routes (Child Routes)**

Used for **layouts** where a page has **multiple sections**.

**Example: Parent-Child Route Setup**

const routes: Routes = [

{

path: 'dashboard',

component: DashboardComponent,

children: [

{ path: 'profile', component: ProfileComponent },

{ path: 'settings', component: SettingsComponent }

]

}

];

Here:

* /dashboard → Loads DashboardComponent.
* /dashboard/profile → Loads ProfileComponent inside DashboardComponent.

**Add <router-outlet> for Child Routes**

<!-- dashboard.component.html -->

<h2>Dashboard</h2>

<a routerLink="profile">Profile</a>

<a routerLink="settings">Settings</a>

<router-outlet></router-outlet> <!-- Child views load here -->

**4. Lazy Loading Modules**

Lazy loading improves performance by **loading modules only when needed**.

**Step 1: Create a Module with Routing**

ng g module feature --routing

**Step 2: Define Routes in feature-routing.module.ts**

const routes: Routes = [

{ path: '', component: FeatureComponent }

];

**Step 3: Configure Lazy Loading in app-routing.module.ts**

const routes: Routes = [

{ path: 'feature', loadChildren: () => import('./feature/feature.module').then(m => m.FeatureModule) }

];

Now, the **Feature Module loads only when /feature is accessed**.

**5. Router Features: Guards and Navigation Events**

**Router Guards (Protecting Routes)**

Guards **control access** to vroutes based on conditions like authentication.

**Types of Router Guards**

1. **CanActivate** → Prevents access to a route.
2. **CanDeactivate** → Prevents leaving a route.
3. **Resolve** → Fetches data before loading a route.

**Example: CanActivate Guard (Auth Protection)**

**Step 1: Create an Auth Guard**

ng g guard auth

**Step 2: Implement canActivate in auth.guard.ts**

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

import { CanActivate, Router } from '@angular/router';

@Injectable({

providedIn: 'root'

})

export class AuthGuard implements CanActivate {

constructor(private router: Router) {}

canActivate(): boolean {

const isLoggedIn = false; // Replace with actual authentication check

if (!isLoggedIn) {

this.router.navigate(['/login']);

return false;

}

return true;

}

}

**Step 3: Apply the Guard to a Route**

const routes: Routes = [

{ path: 'dashboard', component: DashboardComponent, canActivate: [AuthGuard] }

];

Now, users **cannot access /dashboard** unless logged in.

**6. Router Events (Tracking Navigation Changes)**

Angular provides RouterEvents to track **navigation start, end, and errors**.

**Example: Track Navigation Events in AppComponent**

import { Router, Event, NavigationStart, NavigationEnd } from '@angular/router';

export class AppComponent {

constructor(private router: Router) {

this.router.events.subscribe((event: Event) => {

if (event instanceof NavigationStart) {

console.log("Navigation Started");

}

if (event instanceof NavigationEnd) {

console.log("Navigation Ended");

}

});

}

}

* NavigationStart triggers **when navigation begins**.
* NavigationEnd triggers **when navigation is completed**.

**Summary of Angular Routing**

✔ **RouterModule** handles navigation between components.

✔ **routerLink** and **router.navigate()** enable navigation.

✔ **Route Parameters** (/user/:id) and **Query Parameters** (?category=electronics).

✔ **Child Routes** allow nested navigation inside components.

✔ **Lazy Loading** improves performance by loading modules only when required.

✔ **Route Guards** (CanActivate, CanDeactivate) secure routes.

✔ **Router Events** help track navigation changes.

**Next Topic: HTTP Client and APIs in Angular**

Let me know if you have any questions before moving forward!

**HTTP Client and APIs in Angular**

Angular provides the **HttpClient module** to communicate with RESTful APIs. It helps make HTTP requests like **GET, POST, PUT, DELETE**, and handle responses using **Observables (RxJS)**.

**1. Setting Up Angular HTTP Client**

**Step 1: Import HttpClientModule in app.module.ts**

Before using HTTP services, import the HttpClientModule:

import { NgModule } from '@angular/core';

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

@NgModule({

imports: [

HttpClientModule // Enables HTTP services

]

})

export class AppModule {}

**2. Making HTTP Requests**

The HttpClient service is used for **API communication**. Let’s create a service to interact with an API.

**Step 1: Generate an API Service**

ng g service api

**Step 2: Implement api.service.ts**

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

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

import { Observable } from 'rxjs';

@Injectable({

providedIn: 'root'

})

export class ApiService {

private apiUrl = '<https://jsonplaceholder.typicode.com/posts>'; // Sample API

constructor(private http: HttpClient) {}

// GET request

getPosts(): Observable<any> {

return this.http.get(this.apiUrl);

}

// POST request

addPost(post: any): Observable<any> {

return this.http.post(this.apiUrl, post);

}

// PUT request

updatePost(id: number, post: any): Observable<any> {

return this.http.put(`${this.apiUrl}/${id}`, post);

}

// DELETE request

deletePost(id: number): Observable<any> {

return this.http.delete(`${this.apiUrl}/${id}`);

}

}

Here,

* getPosts() → Fetches all posts.
* addPost() → Adds a new post.
* updatePost() → Updates an existing post.
* deletePost() → Deletes a post by ID.

**3. Using HTTP Client in a Component**

Now, let's consume the ApiService in a component.

**Step 1: Inject ApiService in app.component.ts**

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

import { ApiService } from './api.service';

@Component({

selector: 'app-root',

template: `

<h2>Posts</h2>

<ul>

<li \*ngFor="let post of posts">{{ post.title }}</li>

</ul>

<button (click)="addPost()">Add Post</button>

`

})

export class AppComponent implements OnInit {

posts: any[] = [];

constructor(private apiService: ApiService) {}

ngOnInit() {

this.apiService.getPosts().subscribe(data => {

this.posts = data;

});

}

addPost() {

const newPost = { title: 'New Post', body: 'This is a new post' };

this.apiService.addPost(newPost).subscribe(response => {

this.posts.push(response);

});

}

}

**How it works:**

✔ The component **fetches posts** when initialized.

✔ Clicking **"Add Post"** sends a POST request and updates the list.

**4. Handling API Responses with Observables**

**Handling Errors Using catchError**

import { catchError } from 'rxjs/operators';

import { throwError } from 'rxjs';

getPosts(): Observable<any> {

return this.http.get(this.apiUrl).pipe(

catchError(error => {

console.error('Error fetching posts:', error);

return throwError(error);

})

);

}

If an error occurs, it gets logged, and the request **fails gracefully**.

**5. Using Interceptors for Modifying Requests and Responses**

Interceptors modify **HTTP requests and responses globally**.

**Step 1: Create an HTTP Interceptor**

ng g interceptor auth

**Step 2: Implement auth.interceptor.ts**

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

import { HttpInterceptor, HttpRequest, HttpHandler, HttpEvent } from '@angular/common/http';

import { Observable } from 'rxjs';

@Injectable()

export class AuthInterceptor implements HttpInterceptor {

intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

const authReq = req.clone({

setHeaders: { Authorization: `Bearer fake-token` }

});

return next.handle(authReq);

}

}

✔ **Adds Authorization headers** to every request.

**Step 3: Register Interceptor in app.module.ts**

import { HTTP\_INTERCEPTORS } from '@angular/common/http';

import { AuthInterceptor } from './auth.interceptor';

providers: [

{ provide: HTTP\_INTERCEPTORS, useClass: AuthInterceptor, multi: true }

]

**6. Summary of Angular HTTP Client**

✔ HttpClientModule is required for API calls.

✔ HttpClient supports **GET, POST, PUT, DELETE** requests.

✔ Use **Observables** to handle async data.

✔ Use **Interceptors** to modify requests globally.

✔ Handle **errors** using catchError.

**Next Topic: State Management in Angular**

Let me know if you have any questions before moving forward!

**State Management in Angular**

State management is essential for maintaining and synchronizing data across different components in an Angular application. There are different ways to manage state in Angular, ranging from simple service-based state management to using advanced libraries like **NgRx**.

**1. Introduction to State Management**

**Why Do We Need State Management?**

In Angular applications, data often needs to be shared across multiple components. Without proper state management, managing this data can become complex.

✔ Prevents unnecessary API calls.

✔ Maintains consistent data across components.

✔ Improves application performance and maintainability.

**Types of State in Angular**

1. **Component State** – Data stored inside a component (local state).
2. **Service-based State** – Centralized state shared across components.
3. **Global State (NgRx)** – Used for large applications with complex data flow.

**2. Using Services for State Management (Simple Approach)**

**Step 1: Create a Service for State Management**

ng g service state

**Step 2: Implement state.service.ts**

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

import { BehaviorSubject } from 'rxjs';

@Injectable({

providedIn: 'root'

})

export class StateService {

private count = new BehaviorSubject<number>(0);

count$ = this.count.asObservable();

updateCount(value: number) {

this.count.next(value);

}

}

✔ **BehaviorSubject** allows components to subscribe and get the latest state.

✔ updateCount() updates the state and notifies all subscribers.

**Step 3: Use the Service in Components**

**Component 1: counter.component.ts (Updating the State)**

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

import { StateService } from '../state.service';

@Component({

selector: 'app-counter',

template: `

<button (click)="increment()">Increment</button>

`

})

export class CounterComponent {

constructor(private stateService: StateService) {}

increment() {

this.stateService.updateCount(1);

}

}

**Component 2: display.component.ts (Reading the State)**

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

import { StateService } from '../state.service';

@Component({

selector: 'app-display',

template: `

<p>Count: {{ count }}</p>

`

})

export class DisplayComponent implements OnInit {

count = 0;

constructor(private stateService: StateService) {}

ngOnInit() {

this.stateService.count$.subscribe(value => {

this.count = value;

});

}

}

✔ When the button is clicked in CounterComponent, DisplayComponent **automatically updates**.

✔ **No need for Input/Output bindings**—the service manages the state globally.

**3. Using NgRx for Advanced State Management**

**What is NgRx?**

NgRx is a **Redux-based** state management library for Angular applications. It provides a predictable state management system using:

✔ **Store** – Centralized state storage.

✔ **Actions** – Define what changes occur.

✔ **Reducers** – Handle how the state changes.

✔ **Effects** – Perform asynchronous operations like API calls.

**Step 1: Install NgRx**

ng add @ngrx/store

ng add @ngrx/effects

**Step 2: Create a State Model (counter.model.ts)**

export interface CounterState {

count: number;

}

**Step 3: Define Actions (counter.actions.ts)**

import { createAction } from '@ngrx/store';

export const increment = createAction('[Counter] Increment');

export const decrement = createAction('[Counter] Decrement');

export const reset = createAction('[Counter] Reset');

✔ Defines possible state modifications like **increment, decrement, and reset**.

**Step 4: Create a Reducer (counter.reducer.ts)**

import { createReducer, on } from '@ngrx/store';

import { increment, decrement, reset } from './counter.actions';

export const initialState: CounterState = { count: 0 };

export const counterReducer = createReducer(

initialState,

on(increment, (state) => ({ count: state.count + 1 })),

on(decrement, (state) => ({ count: state.count - 1 })),

on(reset, (state) => ({ count: 0 }))

);

✔ The **reducer updates the state** when an action is triggered.

**Step 5: Register the Store in app.module.ts**

import { StoreModule } from '@ngrx/store';

import { counterReducer } from './counter.reducer';

@NgModule({

imports: [

StoreModule.forRoot({ counter: counterReducer }) // Register reducer

]

})

export class AppModule {}

**Step 6: Dispatch Actions in a Component (counter.component.ts)**

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

import { Store } from '@ngrx/store';

import { increment, decrement, reset } from '../counter.actions';

@Component({

selector: 'app-counter',

template: `

<button (click)="increase()">+</button>

<button (click)="decrease()">-</button>

<button (click)="reset()">Reset</button>

`

})

export class CounterComponent {

constructor(private store: Store) {}

increase() {

this.store.dispatch(increment());

}

decrease() {

this.store.dispatch(decrement());

}

reset() {

this.store.dispatch(reset());

}

}

✔ Dispatching an action **updates the global state**.

**Step 7: Read State in Another Component (display.component.ts)**

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

import { Store } from '@ngrx/store';

import { Observable } from 'rxjs';

@Component({

selector: 'app-display',

template: `<p>Count: {{ count$ | async }}</p>`

})

export class DisplayComponent {

count$: Observable<number>;

constructor(private store: Store<{ counter: { count: number } }>) {

this.count$ = this.store.select(state => state.counter.count);

}

}

✔ Uses **Selectors** to read the **counter state** in real-time.

✔ The **async pipe** automatically updates the view when the state changes.

**4. Summary of State Management in Angular**

**Next Topic: Reactive Programming with RxJS**

Let me know if you have any questions before moving forward!

**Reactive Programming with RxJS in Angular**

RxJS (Reactive Extensions for JavaScript) is a powerful library for **reactive programming** in Angular. It allows managing asynchronous operations efficiently using **Observables**, **Operators**, and **Subscriptions**.

**1. What is RxJS?**

RxJS provides a way to handle:

✔ **Asynchronous data streams** (e.g., API calls, user input, WebSockets).

✔ **Event-based programming** with Observables.

✔ **Data transformations** using Operators.

✔ **Cancellation and error handling** for async operations.

**2. RxJS Observables vs Promises**

Example:

// Observable

const observable$ = new Observable(observer => {

observer.next('Hello');

observer.next('RxJS');

observer.complete();

});

// Promise

const promise = new Promise(resolve => {

resolve('Hello RxJS with Promise');

});

✔ Observables **emit multiple values**, while Promises **resolve once**.

**3. Creating Observables in Angular**

**Basic Observable**

import { Observable } from 'rxjs';

const myObservable = new Observable(observer => {

observer.next('First value');

observer.next('Second value');

observer.complete();

});

myObservable.subscribe(value => console.log(value));

✔ The observer.next() emits values.

✔ subscribe() listens for emitted values.

**4. Using Observables with HTTP Client**

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

import { Observable } from 'rxjs';

export class ApiService {

constructor(private http: HttpClient) {}

getUsers(): Observable<any> {

return this.http.get('<https://jsonplaceholder.typicode.com/users>');

}

}

✔ The HTTP request returns an **Observable** instead of a Promise.

**5. RxJS Operators for Data Transformation**

RxJS **operators** modify data streams. Some key operators:

**1. map() - Transform Data**

import { map } from 'rxjs/operators';

this.apiService.getUsers().pipe(

map(users => users.map(user => user.name))

).subscribe(names => console.log(names));

✔ Transforms API response to return only user names.

**2. filter() - Filter Data**

import { filter } from 'rxjs/operators';

this.apiService.getUsers().pipe(

filter(user => user.id < 5)

).subscribe(users => console.log(users));

✔ Returns users with id < 5.

**3. debounceTime() - Handle Rapid Events (e.g., Input Fields)**

import { debounceTime } from 'rxjs/operators';

searchInput.valueChanges.pipe(

debounceTime(300)

).subscribe(value => console.log(value));

✔ Ignores rapid keystrokes in a search field, processing only after **300ms**.

**4. switchMap() - Cancel Previous Requests**

import { switchMap } from 'rxjs/operators';

this.searchInput.valueChanges.pipe(

switchMap(value => this.apiService.searchUsers(value))

).subscribe(result => console.log(result));

✔ Cancels previous API calls if a new request is made (useful in live search).

**6. Error Handling in RxJS**

import { catchError } from 'rxjs/operators';

import { throwError } from 'rxjs';

this.apiService.getUsers().pipe(

catchError(error => {

console.error('Error occurred:', error);

return throwError(error);

})

).subscribe();

✔ Handles API errors without crashing the app.

**7. Unsubscribing from Observables**

Observables can cause **memory leaks** if not unsubscribed properly.

**Using takeUntil()**

import { Subject } from 'rxjs';

import { takeUntil } from 'rxjs/operators';

private destroy$ = new Subject<void>();

this.apiService.getUsers().pipe(

takeUntil(this.destroy$)

).subscribe();

// Unsubscribe when component is destroyed

ngOnDestroy() {

this.destroy$.next();

this.destroy$.complete();

}

✔ Ensures Observables **stop listening** when the component is destroyed.

**8. Summary of RxJS in Angular**

✔ **Observables** are better than Promises for managing async data.

✔ **Operators** like map(), filter(), and debounceTime() modify data streams.

✔ **catchError()** handles API errors.

✔ **takeUntil()** prevents memory leaks.

**Next Topic: Testing Angular Applications**

Let me know if you have any questions before moving forward!

**Testing Angular Applications**

Testing is crucial in Angular applications to ensure reliability and prevent bugs. Angular provides built-in testing tools using **Jasmine**, **Karma**, and **Protractor** (or Cypress for E2E).

**1. Types of Testing in Angular**

✔ **Unit Testing** – Tests individual components, services, and pipes. (Uses **Jasmine + Karma**)

✔ **Integration Testing** – Tests interactions between components or modules.

✔ **End-to-End (E2E) Testing** – Tests the full user flow of an application. (Uses **Cypress / Protractor**)

**2. Unit Testing with Jasmine & Karma**

**What are Jasmine & Karma?**

* **Jasmine** – A testing framework with features like describe(), it(), expect().
* **Karma** – A test runner that runs tests in different browsers.

**Setting Up Tests in Angular**

Angular projects come with preconfigured **Jasmine + Karma**. The test files have a .spec.ts extension.

Run tests:

ng test

**3. Testing Components in Angular**

**Basic Test Setup (app.component.spec.ts)**

import { ComponentFixture, TestBed } from '@angular/core/testing';

import { AppComponent } from './app.component';

describe('AppComponent', () => {

let component: AppComponent;

let fixture: ComponentFixture<AppComponent>;

beforeEach(async () => {

await TestBed.configureTestingModule({

declarations: [AppComponent]

}).compileComponents();

fixture = TestBed.createComponent(AppComponent);

component = fixture.componentInstance;

fixture.detectChanges();

});

it('should create the component', () => {

expect(component).toBeTruthy();

});

});

✔ TestBed.configureTestingModule() sets up the test environment.

✔ fixture.detectChanges() triggers change detection.

✔ expect(component).toBeTruthy(); ensures the component exists.

**4. Testing Services**

**Mocking a Service (data.service.ts)**

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

@Injectable({

providedIn: 'root'

})

export class DataService {

getData() {

return 'Hello Angular!';

}

}

**Testing the Service (data.service.spec.ts)**

import { TestBed } from '@angular/core/testing';

import { DataService } from './data.service';

describe('DataService', () => {

let service: DataService;

beforeEach(() => {

TestBed.configureTestingModule({});

service = TestBed.inject(DataService);

});

it('should return data', () => {

expect(service.getData()).toBe('Hello Angular!');

});

});

✔ Uses TestBed.inject() to get the service instance.

✔ Checks if getData() returns the expected value.

**5. Mocking HTTP Requests with HttpTestingController**

**Service Making API Calls (api.service.ts)**

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

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

import { Observable } from 'rxjs';

@Injectable({

providedIn: 'root'

})

export class ApiService {

constructor(private http: HttpClient) {}

getUsers(): Observable<any> {

return this.http.get('<https://jsonplaceholder.typicode.com/users>');

}

}

**Testing API Calls (api.service.spec.ts)**

import { TestBed } from '@angular/core/testing';

import { HttpClientTestingModule, HttpTestingController } from '@angular/common/http/testing';

import { ApiService } from './api.service';

describe('ApiService', () => {

let service: ApiService;

let httpMock: HttpTestingController;

beforeEach(() => {

TestBed.configureTestingModule({

imports: [HttpClientTestingModule],

providers: [ApiService]

});

service = TestBed.inject(ApiService);

httpMock = TestBed.inject(HttpTestingController);

});

it('should fetch users', () => {

const dummyUsers = [{ id: 1, name: 'John' }, { id: 2, name: 'Jane' }];

service.getUsers().subscribe(users => {

expect(users.length).toBe(2);

expect(users).toEqual(dummyUsers);

});

const request = httpMock.expectOne('<https://jsonplaceholder.typicode.com/users>');

expect(request.request.method).toBe('GET');

request.flush(dummyUsers);

});

afterEach(() => {

httpMock.verify();

});

});

✔ Uses HttpTestingController to mock API responses.

✔ request.flush(dummyUsers) simulates the API returning data.

**6. End-to-End (E2E) Testing with Cypress**

**Installing Cypress**

ng add @cypress/schematic

✔ Cypress provides **faster E2E testing** than Protractor.

**Writing an E2E Test (app.cy.ts)**

describe('App Test', () => {

it('should load homepage', () => {

cy.visit('/');

cy.contains('Welcome to Angular').should('be.visible');

});

});

✔ cy.visit('/') navigates to the homepage.

✔ cy.contains() checks if the page has expected text.

Run E2E tests:

npx cypress open

**7. Summary of Angular Testing**

✔ **Unit tests** ensure components and services work independently.

✔ **Mocking HTTP requests** prevents unnecessary API calls.

✔ **Cypress provides better E2E testing** than Protractor.

**Next Topic: Introduction to Node.js**

Let me know if you have any questions before moving forward!

**Introduction to Node.js**

Node.js is a **runtime environment** that allows JavaScript to run outside the browser. It is commonly used for **server-side applications**, APIs, and backend development.

**1. What is Node.js?**

✔ Built on **Chrome’s V8 JavaScript engine**.

✔ Uses an **event-driven, non-blocking I/O model** (asynchronous operations).

✔ Perfect for **real-time applications** like chat apps, APIs, and streaming services.

**Key Features**

✔ **Single-threaded & non-blocking** – Handles multiple requests efficiently.

✔ **Event-driven** – Uses events instead of traditional request-response models.

✔ **Cross-platform** – Runs on Windows, macOS, and Linux.

**2. Installing Node.js and npm**

1. Download and install Node.js from [nodejs.org](https://nodejs.org/).
2. Verify installation:
3. node -v # Check Node.js version
4. npm -v # Check npm (Node Package Manager) version

✔ npm is used to **install and manage packages** in Node.js projects.

**3. Node.js Architecture: Event Loop & Non-Blocking I/O**

✔ **Traditional servers (PHP, Java, etc.)** use multiple threads to handle requests.

✔ **Node.js uses a single-threaded event loop**, making it lightweight and fast.

**How Non-Blocking Works?**

const fs = require('fs');

fs.readFile('file.txt', 'utf8', (err, data) => {

if (err) throw err;

console.log(data);

});

console.log('Reading file...');

✔ **Non-blocking:** fs.readFile() runs in the background while the next line executes.

✔ **Output Order:**

Reading file...

(file content appears after reading completes)

**4. Node.js Module System**

Node.js follows a **modular approach**, where each file is a module.

**Creating a Module (math.js)**

function add(a, b) {

return a + b;

}

module.exports = add; // Export the function

**Using the Module (app.js)**

const add = require('./math'); // Import module

console.log(add(5, 10)); // Output: 15

✔ **CommonJS syntax** (require and module.exports) is used in Node.js.

**5. Summary of Node.js**

✔ Node.js runs JavaScript **outside the browser**.

✔ Uses **non-blocking, asynchronous I/O** for high performance.

✔ **Built-in modules** like fs, http, and path help in development.

✔ **Modular system** allows code reusability.

**Next Topic: Core Node.js Modules**

Let me know if you have any questions before moving forward!

**Core Node.js Modules**

Node.js comes with several built-in modules that provide essential functionality without needing external packages. These modules include **fs (File System), http (HTTP Server), path, os, events, and more**.

**1. File System Module (fs)**

The **fs (File System) module** allows working with files and directories.

**Reading a File (Asynchronous)**

const fs = require('fs');

fs.readFile('example.txt', 'utf8', (err, data) => {

if (err) throw err;

console.log(data);

});

console.log('Reading file...');

✔ **Non-blocking:** The file is read in the background while the next line executes.

**Writing to a File**

fs.writeFile('output.txt', 'Hello Node.js!', (err) => {

if (err) throw err;

console.log('File written successfully');

});

✔ Creates (or overwrites) output.txt with the given content.

**2. HTTP Module (http)**

The **http module** allows creating web servers.

**Creating a Basic HTTP Server**

const http = require('http');

const server = http.createServer((req, res) => {

res.writeHead(200, { 'Content-Type': 'text/plain' });

res.end('Hello, World!');

});

server.listen(3000, () => {

console.log('Server running at <http://localhost:3000>');

});

✔ Listens on **port 3000** and responds with "Hello, World!".

**3. Path Module (path)**

The **path module** helps handle file and directory paths.

**Working with File Paths**

const path = require('path');

console.log(path.basename('/user/local/file.txt')); // file.txt

console.log(path.dirname('/user/local/file.txt')); // /user/local

console.log(path.extname('/user/local/file.txt')); // .txt

✔ Extracts **filename, directory name, and file extension**.

**4. OS Module (os)**

The **os module** provides system-related information.

**Getting System Info**

const os = require('os');

console.log(os.platform()); // e.g., 'linux', 'win32'

console.log(os.arch()); // e.g., 'x64'

console.log(os.freemem()); // Available RAM

console.log(os.totalmem()); // Total RAM

✔ Useful for checking **system specifications** in a Node.js application.

**5. Events Module (events)**

The **events module** helps handle custom events in Node.js.

**Creating and Handling Events**

const EventEmitter = require('events');

const eventEmitter = new EventEmitter();

eventEmitter.on('greet', () => {

console.log('Hello, Node.js!');

});

eventEmitter.emit('greet'); // Output: Hello, Node.js!

✔ on() registers an event listener, and emit() triggers the event.

**6. Summary of Core Modules**

**Next Topic: Asynchronous Programming Basics in Node.js**

Let me know if you have any questions before moving forward!

**Asynchronous Programming Basics in Node.js**

Node.js uses **asynchronous, non-blocking** programming to handle multiple tasks efficiently. This makes it ideal for handling **I/O operations, APIs, databases, and real-time applications**.

**1. What is Asynchronous Programming?**

✔ **Synchronous Code** – Executes line by line, blocking further execution until the current operation completes.

✔ **Asynchronous Code** – Executes tasks in the background without blocking execution.

**Example: Synchronous vs. Asynchronous**

**Synchronous Code (Blocking)**

console.log('Start');

const result = fs.readFileSync('file.txt', 'utf8'); // Blocks execution

console.log(result);

console.log('End');

✔ The program **waits** until file.txt is read before continuing.

**Asynchronous Code (Non-Blocking)**

console.log('Start');

fs.readFile('file.txt', 'utf8', (err, data) => {

if (err) throw err;

console.log(data);

});

console.log('End');

✔ **Does not wait** for the file to be read, allowing other code to execute.

**2. Callbacks in Node.js**

A **callback** is a function passed as an argument to another function, executed after the task completes.

**Example: Using a Callback**

function fetchData(callback) {

setTimeout(() => {

callback('Data received');

}, 2000);

}

fetchData((message) => {

console.log(message);

});

✔ fetchData() runs in the background and calls the callback after **2 seconds**.

**Callback Hell (Nested Callbacks)**

When multiple asynchronous operations depend on each other, it leads to **callback hell**.

getUser((user) => {

getOrders(user.id, (orders) => {

getOrderDetails(orders[0], (details) => {

console.log(details);

});

});

});

✔ **Hard to read and maintain**.

**3. Promises in Node.js**

A **Promise** is an object that represents the result of an asynchronous operation. It can be:

* **Pending** – Operation in progress.
* **Resolved** – Operation completed successfully (.then()).
* **Rejected** – Operation failed (.catch()).

**Creating and Using a Promise**

function fetchData() {

return new Promise((resolve, reject) => {

setTimeout(() => {

resolve('Data received');

}, 2000);

});

}

fetchData().then((message) => {

console.log(message);

}).catch((error) => {

console.log(error);

});

✔ **Improves readability** compared to nested callbacks.

**4. Async/Await in Node.js**

async/await is a modern way to handle asynchronous operations without .then() chaining.

**Using Async/Await**

function fetchData() {

return new Promise((resolve) => {

setTimeout(() => resolve('Data received'), 2000);

});

}

async function getData() {

console.log('Fetching data...');

const data = await fetchData();

console.log(data);

}

getData();

✔ **More readable** than Promises and Callbacks.

✔ await pauses execution until the Promise resolves.