Independent Skill Development Mission

Independent Skill Development Mission NXT
EMPOWERING FUTURES, SHAPING TOMORROW

ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION)

INTRODUCTION TO ANGULAR & TYPESCRIPT (WEEKS 1-2)

INTRODUCTION TO ANGULAR AND SINGLE PAGE APPLICATIONS (SPAS)

ISDM-NXT

CHAPTER 1: UNDERSTANDING ANGULAR

1.1 What is Angular?

Angular is a **TypeScript-based, open-source web application framework** developed and maintained by **Google**. It is used for building **dynamic, scalable, and high-performance** single-page web applications (SPAs).

✓ **Component-based architecture** – Applications are built using **reusable components**.
✓ **Two-way data binding** – Synchronizes data between the model and the view automatically.
✓ **Dependency injection (DI)** – Improves modularity and code organization.
✓ **Powerful tools and CLI** – Simplifies development, testing, and deployment.

Angular provides a **structured and scalable** approach to web development, making it a preferred choice for **enterprise applications, dashboards, and complex UI systems**.

1.2 Evolution of Angular

Angular has evolved significantly since its launch:

Version	Key Features
AngularJS (2010)	First version, based on JavaScript and MVC pattern.
Angular 2 (2016)	Complete rewrite, component-based architecture, TypeScript support.
Angular 4-6	Improved performance, animation enhancements, CLI advancements.
Angular 7-9	Ivy renderer, dynamic imports, improved build times.
Angular 10-13	New lifecycle hooks, default strict mode, performance optimizations.
Angular 14+	Standalone components, better debugging, server-side rendering improvements.

Each version introduced **enhancements in speed, scalability, and development experience**.

CHAPTER 2: WHAT IS A SINGLE PAGE APPLICATION (SPA)?

2.1 Understanding SPAs vs. Traditional Web Applications

ISDM-NXT

A **Single Page Application (SPA)** is a web application that loads **a single HTML page** and dynamically updates the content without reloading the page.

✓ **Faster user experience** – Only necessary content is updated, reducing load times.
✓ **Efficient server communication** – Uses **AJAX and APIs** to fetch data asynchronously.
✓ **Better user interaction** – Smooth navigation without page reloads.

2.2 SPA vs. Multi-Page Applications (MPAs)

Feature	Single Page Application (SPA)	Multi-Page Application (MPA)
Page Reloads	No full-page reloads, only content updates dynamically.	Each interaction loads a new page from the server.
Performance	Faster and more responsive after initial load.	Slower due to multiple full-page reloads.
Data Handling	Uses APIs and AJAX for dynamic data fetching.	Refreshes entire page for new data.
Examples	Gmail, Facebook, Twitter, Google Drive.	E-commerce sites, blogs, government portals.

SPAs are **ideal for interactive web applications** like dashboards, real-time applications, and progressive web apps (PWAs).

CHAPTER 3: HOW ANGULAR POWERS SPAS

ISDM-NXT

3.1 Angular Features for Building SPAs

Angular provides **built-in tools and techniques** to efficiently build SPAs:

✓ **Routing Module** – Handles navigation **without reloading** the page.
✓ **Lazy Loading** – Loads only required components, improving performance.
✓ **Data Binding** – Synchronizes UI with data, reducing manual DOM manipulation.
✓ **State Management with RxJS** – Efficiently handles **real-time data streams**.

3.2 Example: Basic SPA Structure in Angular

An Angular SPA typically consists of:

1. **App Module (app.module.ts)** – Defines components and imports.
2. **App Component (app.component.ts)** – The root component for the application.
3. **Routing Module (app-routing.module.ts)** – Manages page navigation.

Example: Creating a Simple Angular SPA

Step 1: **Create a New Angular Project**

```
ng new my-angular-spa
```

```
cd my-angular-spa
```

```
ng serve
```

✓ Runs the Angular app on http://localhost:4200/.

Step 2: **Add a New Component**

ISDM-NXT

ng generate component home

✓ Creates a new **HomeComponent** for the SPA.

Step 3: **Set Up Routing for SPA Navigation**

Modify app-routing.module.ts to define navigation routes:

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { HomeComponent } from './home/home.component';

const routes: Routes = [
  { path: '', component: HomeComponent },
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

✓ Enables **client-side navigation** in the SPA.

CHAPTER 4: BENEFITS AND CHALLENGES OF SPAS

4.1 Advantages of Using SPAs

ISDM-NXT

✓ **Fast Performance** – No unnecessary page reloads.
✓ **Smooth User Experience** – Feels like a desktop application.
✓ **Efficient API Calls** – Uses RESTful APIs for dynamic data.
✓ **Better Caching** – Reduces server load and speeds up content delivery.

4.2 Challenges of SPAs

🚧 **Initial Load Time** – Needs to download JavaScript before displaying content.
🚧 **SEO Issues** – Search engines may struggle to index content.
🚧 **Security Concerns** – Requires proper **CORS and authentication** handling.

🔹 **Solution:** Use **Angular Universal** for **server-side rendering (SSR)** to improve SEO and performance.

Case Study: How a FinTech Company Built an Angular SPA for Real-Time Transactions

Background

A **FinTech company** needed a **real-time, secure** SPA for **managing transactions**.

Challenges

✓ **Slow page reloads** in the old system.
✓ **Poor API handling,** causing data delays.
✓ **Security vulnerabilities** in client-server communication.

Solution: Implementing an Angular SPA

ISDM-NXT

✓ **Built an Angular-based dashboard** for seamless navigation.
✓ **Used RxJS for real-time data updates** on transactions.
✓ **Enabled lazy loading** to improve initial load time.

Results

🚀 **50% faster load times,** improving user experience.
🔒 **Secured API communication,** reducing data breaches.
⚡ **Efficient state management,** handling 1000+ transactions per second.

By using **Angular's SPA capabilities,** the company **transformed its financial dashboard** into a **high-performance web application**.

Exercise

1. Install **Angular CLI** and create a new Angular project.
2. Create a **HomeComponent** and display "Welcome to Angular SPA".
3. Implement **Routing** to navigate between multiple pages.
4. Analyze **network requests** using the browser's Developer Tools.

Conclusion

In this section, we explored:
✓ **What Angular is and how it powers SPAs**.
✓ **The benefits of Single Page Applications vs. Traditional Web Apps**.
✓ **How to create a basic Angular SPA using components and routing**.

ISDM-NXT

✓ **How Angular improves performance, API integration, and user experience**.

ISDM-NXT

SETTING UP THE ANGULAR DEVELOPMENT ENVIRONMENT

CHAPTER 1: INTRODUCTION TO ANGULAR DEVELOPMENT

1.1 Understanding Angular as a Frontend Framework

Angular is a **TypeScript-based frontend framework** developed by Google for building **single-page applications (SPAs)**. It provides:

✓ **Modular architecture** – Breaks applications into reusable components.
✓ **Two-way data binding** – Synchronizes the model and the view automatically.
✓ **Dependency injection** – Helps manage application services efficiently.
✓ **Built-in CLI (Command Line Interface)** – Automates project setup and management.

Setting up the Angular development environment is the **first step** to building robust applications efficiently.

CHAPTER 2: INSTALLING PREREQUISITES FOR ANGULAR

2.1 Required Software for Angular Development

Before setting up Angular, install the following dependencies:

Software	Purpose	Download Link
Node.js	Runs the JavaScript	https://nodejs.org/

ISDM-NXT

	runtime needed for Angular CLI.	
npm (Node Package Manager)	Manages Angular and other dependencies.	Included with Node.js
Angular CLI	Automates project setup, builds, and testing.	Installed via npm
Code Editor (VS Code recommended)	Used to write and manage Angular code.	https://code.visualstudio.com/
Git (Optional)	Version control for managing code changes.	https://git-scm.com/

CHAPTER 3: INSTALLING NODE.JS AND NPM

3.1 Checking for Existing Installation

Before installing, check if **Node.js** and **npm** are already installed:

node -v

npm -v

✓ If versions appear, Node.js and npm are installed.
✓ If not, proceed with installation.

3.2 Installing Node.js

ISDM-NXT

1. Download the **LTS version** from Node.js official website.
2. Run the installer and follow the **default setup**.
3. Verify installation:

node -v

npm -v

✓ Confirms successful installation.

CHAPTER 4: INSTALLING ANGULAR CLI

4.1 What is Angular CLI?

The **Angular CLI (Command Line Interface)** is a tool that:

✓ **Creates and configures Angular projects**.
✓ **Generates components, services, and modules** automatically.
✓ **Builds and deploys Angular applications**.

4.2 Installing Angular CLI

Run the following command to install Angular CLI globally:

npm install -g @angular/cli

✓ The **-g flag** installs Angular CLI globally on the system.

4.3 Verifying Angular CLI Installation

To check if Angular CLI is installed, run:

ng version

📌 Sample Output:

Angular CLI: 16.1.4

Node: 18.16.1

ISDM-NXT

OS: Windows 10

✓ Confirms that Angular CLI is ready for use.

Chapter 5: Creating a New Angular Project

5.1 Generating a New Angular Application

To create a new Angular project, use:

ng new my-angular-app

✓ my-angular-app is the **project name**.
✓ The CLI will prompt you to:

- Choose **routing** (Yes or No).
- Select **CSS preprocessor** (CSS, SCSS, LESS).

5.2 Navigating into the Project

cd my-angular-app

✓ Changes the directory to the new Angular project.

Chapter 6: Running an Angular Application

6.1 Starting the Angular Development Server

To launch the Angular application, run:

ng serve

✓ Compiles the project and starts a **development server**.
✓ The default port is **http://localhost:4200/**.

6.2 Changing the Default Port

Run the command with a specific port:

ISDM-NXT

ng serve --port 8080

✓ Starts the Angular app on http://localhost:8080/.

CHAPTER 7: UNDERSTANDING THE ANGULAR PROJECT STRUCTURE

After creating a new Angular project, the directory structure looks like this:

📂 **my-angular-app/**
├── 📂 **src/** *(Main application files)*
│ ├── 📂 **app/** *(Contains components, modules, services)*
│ ├── 📂 **assets/** *(Static assets like images, fonts, and styles)*
│ ├── **index.html** *(Main HTML entry point)*
│ ├── **styles.css** *(Global styles for the application)*
│ ├── **main.ts** *(Application bootstrap file)*
├── **angular.json** *(Angular project configuration file)*
├── **package.json** *(Project dependencies and scripts)*
├── **README.md** *(Project documentation file)*

✓ This structure keeps Angular projects well-organized and modular.

Case Study: How a Startup Set Up Angular for Rapid Development

Background

A **tech startup** needed to develop a **real-time dashboard** for financial data analytics. The project required:

✓ **Scalability** – A framework that could grow with the application.
✓ **Fast development setup** – To meet **tight deadlines**.

ISDM-NXT

✓ **Component-based architecture** – To **reuse UI components** across multiple dashboards.

Challenges

- Initial **setup delays** due to unfamiliarity with Angular CLI.
- **Inconsistent development environments** across teams.
- **Slow testing and debugging process**.

Solution: Using Angular CLI for Rapid Setup

The team:

✓ Used **Angular CLI** to quickly scaffold components and services.
✓ Set up **a shared component library** for reusable UI elements.
✓ Used **ng serve with hot reloading** for faster development.

Results

🚀 **Setup time reduced from 2 days to 3 hours**.
⚡ **Real-time updates were implemented seamlessly**.
🔍 **Code consistency improved,** reducing debugging efforts.

By using **Angular CLI and best practices,** the startup successfully **launched its product ahead of schedule**.

Exercise

1. Install **Node.js and npm** on your machine.
2. Install **Angular CLI** globally using npm install -g @angular/cli.
3. Create a new Angular project named my-angular-app.
4. Start the development server using ng serve and view the app in a browser.

ISDM-NXT

5. Modify app.component.html to display "Welcome to Angular Development!".

Conclusion

In this section, we explored:

✓ **How to set up the Angular development environment.**

✓ **How to install Node.js, npm, and Angular CLI.**

✓ **How to create, run, and structure an Angular project.**

ISDM-NXT

CREATING A BASIC ANGULAR PROJECT

CHAPTER 1: INTRODUCTION TO ANGULAR AND PROJECT SETUP

1.1 Understanding Angular and Its Benefits

Angular is a **TypeScript-based front-end framework** developed by Google for building **dynamic, single-page applications (SPAs)**. It offers:

✓ **Component-based architecture** – Reusable and modular code structure.
✓ **Two-way data binding** – Synchronizes data between UI and logic.
✓ **Dependency injection** – Efficient service management.
✓ **Built-in directives and pipes** – Enhances UI development.

To start developing an Angular application, we must set up the **Angular development environment**.

CHAPTER 2: SETTING UP THE ANGULAR DEVELOPMENT ENVIRONMENT

2.1 Installing Node.js and Angular CLI

Before creating an Angular project, install the required dependencies:

Step 1: Install Node.js

Angular requires **Node.js** and **npm (Node Package Manager)**.

1. Download and install Node.js from nodejs.org.
2. Verify the installation by running the following commands:

node -v

ISDM-NXT

npm -v

✓ This confirms Node.js and npm are installed.

Step 2: Install Angular CLI (Command Line Interface)

The **Angular CLI** simplifies project creation and management.

Run the following command to install it globally:

npm install -g @angular/cli

✓ This installs the **latest version of Angular CLI**.
✓ Verify the installation:

ng version

✓ Displays the installed Angular CLI version.

CHAPTER 3: CREATING A NEW ANGULAR PROJECT

3.1 Generating an Angular Project Using CLI

To create a new Angular project:

ng new my-angular-app

✓ **The CLI prompts for configurations:**

- **Would you like to add Angular routing?** → Type Y (Yes) or N (No).
- **Which CSS preprocessor to use?** → Choose CSS, SCSS, or LESS.

The CLI **automatically creates the project structure** with essential files.

3.2 Navigating into the Project Folder

ISDM-NXT

Move into the newly created project directory:

cd my-angular-app

✓ The project contains essential files like:

- **src/app** – Contains application components.
- **angular.json** – Project configuration file.
- **package.json** – Manages dependencies.

CHAPTER 4: RUNNING THE ANGULAR APPLICATION

4.1 Serving the Angular Project

To start the development server:

ng serve

✓ By default, the Angular application runs on **http://localhost:4200/**.

4.2 Running on a Custom Port

To change the port:

ng serve --port 4500

✓ Now, the application runs on **http://localhost:4500/**.

CHAPTER 5: UNDERSTANDING ANGULAR PROJECT STRUCTURE

5.1 Key Files and Directories

File/Directory	Description

ISDM-NXT

src/app/	Contains the core application logic and components.
src/index.html	The main HTML file that loads the Angular app.
src/main.ts	The entry point of the Angular application.
src/app/app.component.ts	The root component of the application.
src/app/app.module.ts	Declares all components, modules, and dependencies.

5.2 Modifying the Default Component

The default component is AppComponent, located in src/app/app.component.html.

Change the default content:

<h1>Welcome to My First Angular App</h1>

This is a basic Angular project setup.

✓ Refreshing the browser **updates the UI instantly**.

CHAPTER 6: CREATING AND USING COMPONENTS

6.1 Generating a New Component

Angular applications consist of **components** that manage UI sections.

To generate a new component:

ng generate component my-component

or

ng g c my-component

✓ The CLI creates:

- **my-component.component.ts** – Component logic.
- **my-component.component.html** – Component template.
- **my-component.component.css** – Component styles.
- **my-component.component.spec.ts** – Unit tests.

6.2 Using the Component in the App

To display the component, use its **selector** in app.component.html:

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

✓ The new component **renders inside the main app component**.

Case Study: How a Startup Built Their First Angular App

Background

A **startup** developing an **e-commerce platform** wanted to:

✓ Build a **modern, dynamic UI** using Angular.
✓ Create a **scalable, modular application**.
✓ Deploy the project **quickly with minimal setup**.

Challenges

- **No frontend framework** – Initial development was slow.
- **Manual DOM manipulation** caused performance issues.
- **Complex data handling** required **real-time updates**.

Solution: Using Angular for a Scalable UI

The team followed these steps:

ISDM-NXT

✓ **Created an Angular project** using ng new.
✓ **Generated modular components** for **products, cart, and checkout**.
✓ **Implemented routing** for a seamless user experience.

Results

🚀 **Reduced development time by 50%** with Angular's component-based architecture.
🔍 **Improved UI performance,** eliminating manual DOM handling.
⚡ **Easier feature updates,** ensuring scalability.

By adopting Angular, the startup **delivered a fast, responsive, and scalable application**.

Exercise

1. **Set up a new Angular project** using Angular CLI.
2. **Modify the app.component.html file** to display a welcome message.
3. **Create a new component** and add it to app.component.html.
4. **Serve the application** and verify it runs successfully.

Conclusion

In this section, we explored:
✓ **How to install and set up Angular CLI**.
✓ **How to create and run a basic Angular project**.
✓ **How to generate and use components**.

ISDM-NXT

UNDERSTANDING ANGULAR CLI

CHAPTER 1: INTRODUCTION TO ANGULAR CLI

1.1 What is Angular CLI?

Angular CLI (Command Line Interface) is a powerful tool that helps developers create, manage, and build Angular applications efficiently. It automates common development tasks such as:

✓ **Project setup** – Quickly scaffolding a new Angular application.
✓ **Code generation** – Creating components, services, modules, and directives.
✓ **Development server** – Running applications locally with live reload.
✓ **Testing & debugging** – Automating unit and end-to-end testing.
✓ **Building applications** – Optimizing Angular projects for production deployment.

1.2 Why Use Angular CLI?

✓ **Speeds up development** – Automates repetitive tasks.
✓ **Enforces best practices** – Generates code following Angular's architecture.
✓ **Manages dependencies** – Handles third-party libraries and configurations.
✓ **Improves consistency** – Ensures uniform project structure for teams.

To install Angular CLI globally, use:

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

To check the installed version:

ISDM-NXT

ng version

✓ Ensures Angular CLI is properly installed.

CHAPTER 2: CREATING AN ANGULAR PROJECT USING ANGULAR CLI

2.1 Setting Up a New Angular Project

To create a new Angular application, run:

ng new my-angular-app

✓ This command:

- Prompts for configurations (e.g., routing, CSS framework).
- Installs all necessary dependencies.
- Creates a structured Angular project.

2.2 Project Structure of an Angular App

A newly created Angular project includes:

Folder/File	Description
src/	Main source code of the application.
app/	Contains components, modules, and services.
assets/	Stores images, stylesheets, and other static files.
environments/	Configurations for development and production.
angular.json	Configuration file for Angular CLI.
package.json	Defines project dependencies.

✓ Maintains an **organized project structure** for scalability.

ISDM-NXT

CHAPTER 3: RUNNING AND SERVING AN ANGULAR APPLICATION

3.1 Starting the Angular Development Server

To launch the application, use:

cd my-angular-app

ng serve

✓ Runs the app at http://localhost:4200/ with **live reloading**.

3.2 Customizing the Development Server

To change the port:

ng serve --port 4500

✓ Serves the application on **port 4500** instead of the default **4200**.

To enable **automatic opening in the browser**:

ng serve --open

✓ Opens the browser automatically upon running the app.

CHAPTER 4: GENERATING COMPONENTS, SERVICES, AND MODULES USING CLI

4.1 Creating Components

To generate a new component, run:

ng generate component components/header

OR

ng g c components/header

✓ Creates a **header component** with:

- header.component.ts – Component logic.

ISDM-NXT

- header.component.html – Template file.
- header.component.css – Stylesheet.
- header.component.spec.ts – Unit test file.

4.2 Creating Services

To create a service, use:

ng generate service services/auth

OR

ng g s services/auth

✓ Creates **auth.service.ts**, which can handle API calls and authentication logic.

4.3 Creating Modules

To generate a module, run:

ng generate module modules/user

OR

ng g m modules/user

✓ Creates a **user.module.ts** file for feature-based modularization.

CHAPTER 5: MANAGING DEPENDENCIES AND THIRD-PARTY PACKAGES

5.1 Installing Dependencies Using Angular CLI

To install third-party libraries, use:

ISDM-NXT

ng add @angular/material

✓ Installs Angular Material with **preconfigured settings**.

To manually install a package:

npm install lodash

✓ Adds **Lodash** as a dependency in package.json.

To remove a package:

npm uninstall lodash

✓ Removes **Lodash** from the project.

CHAPTER 6: BUILDING AND DEPLOYING ANGULAR APPLICATIONS

6.1 Building an Angular Project for Production

To generate an optimized build, use:

ng build --prod

✓ Produces **minified, optimized, and compressed files** in the dist/ directory.

6.2 Deploying an Angular App Using Angular CLI

To deploy to **GitHub Pages,** install the Angular deployment package:

ng add angular-cli-ghpages

Build and deploy the project:

ng deploy --base-href=/my-angular-app/

✓ **Deploys the app to GitHub Pages** for free hosting.

ISDM-NXT

Case Study: How a Tech Startup Optimized Development Using Angular CLI

Background

A **tech startup** building a **real-time dashboard** struggled with:

✓ **Manual project setup delays**.
✓ **Slow development workflow** due to inefficient coding practices.
✓ **Difficulties in managing components and services**.

Challenges

- Setting up **project structure manually** was **time-consuming**.
- Developers **created components and services manually,** slowing progress.
- Manually linking files **led to errors and inconsistencies**.

Solution: Implementing Angular CLI for Development

The team adopted Angular CLI to:

✓ **Automate project setup** using ng new.
✓ **Generate components and services** using ng g c and ng g s.
✓ **Improve development speed** with ng serve for live reload.

Results

🚀 **Project setup time reduced from 2 hours to 5 minutes**.
⚡ **Faster coding workflow,** reducing **manual errors**.
🔧 **Improved scalability** with well-structured code organization.

By using **Angular CLI,** the startup **accelerated development and improved efficiency**.

Exercise

ISDM-NXT

1. Install Angular CLI and create a new project called my-angular-app.
2. Generate a new component called navbar.
3. Create a service called authService inside a services folder.
4. Run the application on port 4300.
5. Build the project for production and deploy it to GitHub Pages.

Conclusion

In this section, we explored:

✓ **How Angular CLI simplifies Angular project setup and development**.

✓ **How to create components, services, and modules using CLI**.

✓ **How to manage dependencies and deploy Angular applications**.

ISDM-NXT

INTRODUCTION TO TYPESCRIPT: VARIABLES, FUNCTIONS, AND CLASSES

CHAPTER 1: UNDERSTANDING TYPESCRIPT

1.1 What is TypeScript?

TypeScript is a **strongly typed, object-oriented, and compiled superset of JavaScript** developed by **Microsoft**. It adds **static typing** and advanced features to JavaScript, making it more scalable and maintainable.

✓ **Improves code quality** – Catches errors **at compile time**, not runtime.
✓ **Supports modern JavaScript features** – Compiles to JavaScript for browser compatibility.
✓ **Enhances development with strong typing** – Reduces debugging time.
✓ **Used in frameworks like Angular** – Ensures better project structure.

1.2 Why Use TypeScript in Angular?

✓ **Stronger Type Safety** – Prevents runtime errors by catching them during development.
✓ **Improved Code Readability** – Type definitions make large projects easier to understand.
✓ **Better Development Tools** – Works well with **VS Code, ESLint, and debugging tools**.

To install TypeScript globally:

```
npm install -g typescript
```

ISDM-NXT

To check the installed version:

tsc --version

CHAPTER 2: DECLARING VARIABLES IN TYPESCRIPT

2.1 Using let, const, and var

In TypeScript, we use **let and const** instead of var for variable declarations.

Example: Variable Declarations

let username: string = "John";

const age: number = 25;

var isAdmin: boolean = true;

✓ **let** – Preferred for variables that change.
✓ **const** – Used for constants that should not be reassigned.
✓ **var (avoid using)** – Has function scope, leading to unexpected behaviors.

2.2 Type Annotations

TypeScript allows specifying **explicit types** to ensure correctness.

Example: Declaring Variables with Types

let count: number = 10;

let message: string = "Hello, TypeScript!";

let isCompleted: boolean = false;

let user: any = "John"; // 'any' allows any type

ISDM-NXT

✓ Using **explicit types** helps catch errors early.

✓ **The any type** allows dynamic typing but reduces TypeScript's benefits.

2.3 Arrays and Tuples

TypeScript provides **typed arrays** and **tuples (fixed-length arrays)**.

Example: Defining an Array

```
let numbers: number[] = [1, 2, 3, 4, 5];
```

Example: Defining a Tuple

```
let userInfo: [string, number] = ["John", 30];
```

✓ Ensures **correct data types and sequence** in arrays.

CHAPTER 3: FUNCTIONS IN TYPESCRIPT

3.1 Declaring Functions with Type Annotations

TypeScript allows defining **return types** and **parameter types** for functions.

Example: Function with Parameter and Return Type

```
function addNumbers(a: number, b: number): number {
  return a + b;
}

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

✓ Ensures a and b are numbers.

✓ Ensures the function **returns a number**.

ISDM-NXT

3.2 Optional and Default Parameters

You can mark function parameters as **optional** using ? or assign **default values**.

Example: Function with Optional Parameter

```
function greet(name: string, greeting?: string): string {
  return `${greeting || "Hello"}, ${name}!`;
}


console.log(greet("Alice")); // Output: Hello, Alice!
console.log(greet("Bob", "Good Morning")); // Output: Good Morning, Bob!
```

✓ The greeting parameter is **optional** (? indicates it may be undefined).

3.3 Function Overloading

TypeScript allows **multiple function signatures** using **overloading**.

Example: Function Overloading

```
function display(value: string): void;
function display(value: number): void;
function display(value: any): void {
  console.log(value);
}


display("Hello");  // Output: Hello
```

ISDM-NXT

```
display(100);     // Output: 100
```

✓ The **same function name** works with **different input types**.

CHAPTER 4: OBJECT-ORIENTED PROGRAMMING (OOP) IN TYPESCRIPT

4.1 Defining Classes in TypeScript

TypeScript supports **classes and OOP concepts** like inheritance, encapsulation, and interfaces.

Example: Creating a Class

```
class Person {
  name: string;
  age: number;

  constructor(name: string, age: number) {
    this.name = name;
    this.age = age;
  }

  greet(): string {
    return `Hello, my name is ${this.name} and I am ${this.age} years old.`;
  }
}
```

ISDM-NXT

```
const person1 = new Person("Alice", 28);

console.log(person1.greet());
```

✓ The constructor initializes **name and age** properties.
✓ The greet() method returns a **formatted string**.

4.2 Access Modifiers (public, private, protected)

TypeScript provides **access control** using public, private, and protected.

Example: Using Access Modifiers

```
class Employee {
  public name: string;    // Accessible everywhere
  private salary: number;  // Accessible only inside this class

  constructor(name: string, salary: number) {
    this.name = name;
    this.salary = salary;
  }

  public getDetails(): string {
    return `Employee: ${this.name}, Salary: ${this.salary}`;
  }
}
```

ISDM-NXT

```
const emp = new Employee("John", 50000);

console.log(emp.getDetails()); // ✅ Works fine

// console.log(emp.salary); ❌ Error: Property 'salary' is private
```

✓ public – Accessible everywhere.
✓ private – Accessible **only within the class**.
✓ protected – Accessible **within the class and its subclasses**.

4.3 Inheritance in TypeScript

Classes can **inherit** from other classes using extends.

Example: Inheritance

```
class Employee {
  name: string;

  constructor(name: string) {
    this.name = name;
  }

  display(): void {
    console.log(`Employee Name: ${this.name}`);
  }
}
```

ISDM-NXT

```
class Manager extends Employee {
  department: string;

  constructor(name: string, department: string) {
    super(name);
    this.department = department;
  }

  display(): void {
    console.log(`Manager: ${this.name}, Department: ${this.department}`);
  }
}

const mgr = new Manager("Alice", "HR");
mgr.display(); // Output: Manager: Alice, Department: HR
```

✓ Manager extends Employee, reusing its properties and methods.
✓ Uses super() to call the **parent constructor**.

Case Study: How TypeScript Helped a Banking App Improve Security & Performance

Background

A banking company built a **loan processing system** using JavaScript, but faced:

ISDM-NXT

✓ **Frequent runtime errors** due to missing type checks.
✓ **Security risks** with incorrect user inputs.
✓ **Slow debugging process** due to loosely typed data.

Solution: Migrating to TypeScript

The team:

✓ Implemented **strict type checking** to prevent data inconsistencies.
✓ Used **classes with private fields** for security.
✓ Optimized function signatures to **ensure correct data flow**.

Results

🚀 **50% reduction in debugging time**.
🔒 **Improved security with strict typing**.
⚡ **Faster execution** due to optimized TypeScript compilation.

By switching to **TypeScript**, the banking app became **safer, faster, and more reliable**.

Exercise

1. Define a variable username as a string and assign a value.
2. Create a function that takes two numbers and returns their sum.
3. Implement a class Car with properties brand and year, and a method getInfo().
4. Extend Car to create a subclass ElectricCar with an additional property batteryLife.

ISDM-NXT

Conclusion

In this section, we explored:

✓ **How TypeScript improves JavaScript with strong typing**.

✓ **How to declare variables, functions, and classes with TypeScript**.

✓ **How OOP concepts like access modifiers and inheritance work in TypeScript**.

ISDM-NXT

UNDERSTANDING INTERFACES AND TYPESCRIPT DECORATORS

CHAPTER 1: INTRODUCTION TO TYPESCRIPT INTERFACES AND DECORATORS

1.1 Why Use TypeScript in Angular?

TypeScript is the **primary language** for Angular development because it provides:

✓ **Static typing** – Detects errors during development.
✓ **Object-oriented programming (OOP) support** – Works well with classes and interfaces.
✓ **Decorator-based architecture** – Essential for defining Angular components.

Two fundamental TypeScript features used in **Angular development** are:

1. **Interfaces** – Define object structures and ensure consistency.
2. **Decorators** – Attach metadata to classes, properties, and methods.

CHAPTER 2: UNDERSTANDING TYPESCRIPT INTERFACES

2.1 What is an Interface?

An **interface** in TypeScript defines the **structure of an object** but does not provide an implementation. It ensures **type safety** by enforcing properties and methods in objects.

ISDM-NXT

✓ **Prevents missing or incorrect properties** in objects.
✓ **Improves code readability** and **reduces runtime errors**.
✓ **Supports inheritance** for reusable data structures.

2.2 Defining and Using Interfaces

Example: Creating a User Interface

```
interface User {
  id: number;
  name: string;
  email: string;
  isAdmin?: boolean; // Optional property
}
```

✓ Defines a User structure with required and optional properties.

Example: Using the Interface in a Function

```
function displayUser(user: User): void {
  console.log(`User: ${user.name}, Email: ${user.email}`);
}

const user1: User = { id: 1, name: "John Doe", email:
"john@example.com" };
displayUser(user1);
```

✓ Ensures user1 follows the User interface structure.

2.3 Extending Interfaces

ISDM-NXT

Interfaces can **extend other interfaces,** allowing reusability.

Example: Extending an Interface

```
interface Employee extends User {
  department: string;
}

const employee: Employee = {
  id: 2,
  name: "Alice",
  email: "alice@example.com",
  department: "Engineering"
};
```

✓ Employee inherits all properties from User **and adds department**.

2.4 Using Interfaces with Angular Services

Interfaces are **commonly used** in Angular to define data models.

Example: Defining an Interface for API Response

```
export interface Product {
  id: number;
  name: string;
  price: number;
}
```

ISDM-NXT

Example: Using the Interface in an Angular Service

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';
import { Product } from './product.interface';

@Injectable({
  providedIn: 'root'
})
export class ProductService {
  private apiUrl = 'https://api.example.com/products';

  constructor(private http: HttpClient) {}

  getProducts(): Observable<Product[]> {
    return this.http.get<Product[]>(this.apiUrl);
  }
}
```

✓ Ensures that **API responses match the expected data structure**.

CHAPTER 3: UNDERSTANDING TYPESCRIPT DECORATORS

3.1 What is a Decorator in TypeScript?

ISDM-NXT

A **decorator** is a special function in TypeScript that **modifies a class, method, or property** at runtime.

✓ Used to **add metadata** to Angular components, services, and modules.
✓ Helps define **Angular components and services**.
✓ Follows **the @ syntax,** such as @Component or @Injectable.

3.2 Types of Decorators in TypeScript

Decorator Type	Usage
Class Decorators	Used on **classes** (e.g., @Component, @Injectable).
Property Decorators	Used on **class properties** (e.g., @Input, @Output).
Method Decorators	Used on **methods** (e.g., @HostListener).
Parameter Decorators	Used on **constructor parameters** (e.g., @Inject).

3.3 Class Decorators in Angular

Example: @Component Decorator for Angular Components

The @Component decorator marks a class as an **Angular component**.

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


@Component({
 selector: 'app-hello',
 template: `<h1>Hello, Angular!</h1>`,
```

ISDM-NXT

```
  styleUrls: ['./hello.component.css']
})
export class HelloComponent {
  constructor() {
    console.log("Component Initialized");
  }
}
```

✓ **Defines an Angular component** with an HTML template.
✓ **Adds metadata** (e.g., selector, template, styles).

3.4 Property Decorators in Angular

Property decorators **modify class properties** in Angular components.

Example: @Input Decorator for Parent-Child Communication

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

@Component({
  selector: 'app-user-card',
  template: `<h2>{{ user.name }}</h2>`
})
export class UserCardComponent {
  @Input() user: { name: string };
}
```

ISDM-NXT

✓ The @Input() decorator **receives data** from the parent component.

3.5 Method Decorators in Angular

Method decorators **modify class methods** to handle events or lifecycle hooks.

Example: @HostListener for Event Handling

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


@Directive({
  selector: '[appClickLogger]'
})
export class ClickLoggerDirective {
  @HostListener('click', ['$event'])
  onClick(event: MouseEvent) {
    console.log("Element clicked:", event);
  }
}
```

✓ The @HostListener() decorator **listens for DOM events**.

3.6 Parameter Decorators in Angular

Parameter decorators **inject dependencies** into class constructors.

Example: @Injectable for Dependency Injection

ISDM-NXT

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


@Injectable({
 providedIn: 'root'
})
export class AuthService {
 login(username: string, password: string) {
  console.log(`Logging in user: ${username}`);
 }
}
```

✓ The @Injectable() decorator **registers the service** for dependency injection.

Case Study: How a Banking App Used TypeScript Interfaces & Decorators in Angular

Background

A **banking web application** needed:

✓ **Strong type safety** for user authentication and transactions.
✓ **Reusable components** for account summaries and transaction history.
✓ **Dependency injection** for efficient service management.

Challenges

- **Data inconsistencies** due to missing or incorrect fields.
- **Code duplication** in UI components.

ISDM-NXT

- **Slow API integration** due to unstructured responses.

Solution: Implementing Interfaces & Decorators

✓ **Used interfaces for API models,** ensuring structured data.
✓ **Implemented @Input decorators** for reusable components.
✓ **Used @Injectable decorators** for services, improving efficiency.

Results

🚀 **Faster development** with reusable components.
🔍 **Improved data accuracy,** reducing runtime errors.
⚡ **Optimized service management,** enhancing application speed.

By leveraging **TypeScript interfaces and decorators,** the app became **scalable, secure, and efficient**.

Exercise

1. Create an interface for a Product with id, name, and price.
2. Define a **service** that fetches products using @Injectable.
3. Create an Angular **component** and use the @Input decorator to pass data.
4. Use the @HostListener decorator to log **button clicks** in the UI.

Conclusion

In this section, we explored:
✓ **How TypeScript interfaces improve type safety in Angular**.
✓ **How decorators add metadata and behavior to Angular components**.

ISDM-NXT

✓ **How to use @Component, @Input, @Injectable, and @HostListener effectively**.

ISDM-NXT

WORKING WITH MODULES AND NAMESPACES IN ANGULAR & TYPESCRIPT

CHAPTER 1: INTRODUCTION TO MODULES AND NAMESPACES

1.1 Understanding Modules and Namespaces in TypeScript

In **TypeScript and Angular, modules** and **namespaces** help organize code, making it **reusable, maintainable, and scalable**.

✓ **Modules** – Allow developers to break applications into **smaller, manageable units**.
✓ **Namespaces** – Group related functionalities to **avoid naming conflicts**.

1.2 Why Use Modules and Namespaces?

✓ **Code Organization** – Divides the app into separate **feature modules**.
✓ **Encapsulation** – Prevents naming conflicts across different files.
✓ **Reusability** – Modules allow **sharing of components and services**.
✓ **Lazy Loading** – Optimizes performance by loading modules **only when needed**.

ISDM-NXT

CHAPTER 2: UNDERSTANDING MODULES IN ANGULAR

2.1 What are Angular Modules?

An **Angular Module** (NgModule) is a container for:

✓ **Components**
✓ **Directives**

✓ **Pipes**
✓ **Services**

Angular applications **must have at least one module,** known as the **root module (AppModule)**.

2.2 Structure of an Angular Module

An **Angular module** is defined in a TypeScript file using the @NgModule decorator.

Example: Basic Module (app.module.ts)

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

@NgModule({
  declarations: [AppComponent], // Components, Directives, Pipes
  imports: [BrowserModule], // Other Modules
  providers: [], // Services
  bootstrap: [AppComponent] // Root Component
})
export class AppModule { }
```

✓ declarations – Lists components, directives, and pipes.
✓ imports – Imports other Angular modules (e.g., BrowserModule).
✓ providers – Registers services globally.
✓ bootstrap – Specifies the root component.

ISDM-NXT

2.3 Creating a Feature Module

For large applications, use **Feature Modules** to separate functionalities.

Step 1: Generate a New Feature Module

ng generate module user

✓ Creates user.module.ts inside src/app/user/.

Step 2: Define the Feature Module

Modify user.module.ts:

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { UserComponent } from './user.component';

@NgModule({
  declarations: [UserComponent],
  imports: [CommonModule],
  exports: [UserComponent]
})
export class UserModule { }
```

✓ exports – Allows the component to be used in other modules.

Step 3: Import the Module in app.module.ts

```
import { UserModule } from './user/user.module';

@NgModule({
```

ISDM-NXT

```
  imports: [UserModule]
})
export class AppModule { }
```

✓ The UserModule is now part of the main application.

2.4 Lazy Loading Modules

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

Step 1: Modify app-routing.module.ts to Lazy Load the User Module

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

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

✓ Loads the UserModule **only when the user navigates to /user**.

CHAPTER 3: UNDERSTANDING NAMESPACES IN TYPESCRIPT

ISDM-NXT

3.1 What are Namespaces?

A **namespace** is a way to **group related functions, interfaces, and classes** to **prevent naming conflicts**.

- Useful in **large applications** to **logically separate functionalities**.

3.2 Creating a Namespace in TypeScript

```
namespace MathOperations {
 export function add(x: number, y: number): number {
  return x + y;
 }

 export function multiply(x: number, y: number): number {
  return x * y;
 }
}
```

✓ **export keyword** makes functions available outside the namespace.

3.3 Using a Namespace in TypeScript

```
let sum = MathOperations.add(10, 5);
console.log(sum); // Output: 15
```

✓ **Accesses functions using NamespaceName.functionName**.

3.4 Using Namespaces Across Multiple Files

ISDM-NXT

Step 1: Define a Namespace in math.ts

```
namespace MathOperations {
  export function subtract(x: number, y: number): number {
    return x - y;
  }
}
```

Step 2: Import and Use the Namespace in app.ts

```
/// <reference path="math.ts" />

let difference = MathOperations.subtract(20, 5);
console.log(difference); // Output: 15
```

✓ /// <reference path="file.ts" /> is used to reference external files.

CHAPTER 4: DIFFERENCES BETWEEN MODULES AND NAMESPACES

Feature	Angular Modules	TypeScript Namespaces
Purpose	Organizes Angular components and services	Groups related functions and classes
Scope	Works within Angular applications	Works in TypeScript globally
Usage	Used for lazy loading and component sharing	Used to prevent naming conflicts
Best For	Large Angular applications	Utility libraries and global functionalities

ISDM-NXT

- **Angular Modules** are recommended for Angular apps, while **Namespaces** are better suited for **TypeScript utility functions**.

Case Study: How a SaaS Company Optimized Code with Angular Modules & Namespaces

Background

A SaaS company developing a **multi-tenant dashboard** faced:

✓ **Code duplication** in different modules.
✓ **Slow loading times** due to large scripts.
✓ **Naming conflicts** between different developers' functions.

Challenges

- **Difficult to maintain shared components**.
- **Manual script imports caused errors**.
- **Large app size slowed down user experience**.

Solution: Using Angular Modules & Namespaces

✓ **Created feature modules** for each section (dashboard, reports, users).
✓ **Lazy loaded modules** to improve performance.
✓ **Used namespaces** to organize shared utilities.

```
namespace Utils {
 export function formatCurrency(amount: number): string {
  return `$${amount.toFixed(2)}`;
 }
}
```

ISDM-NXT

Results

🚀 **Reduced code duplication,** making maintenance easier.
⚡ **Improved app performance** with lazy loading.
🔍 **Faster debugging** due to structured namespaces.

By leveraging **Angular Modules and TypeScript Namespaces,** the company **built a scalable, maintainable application**.

Exercise

1. Create an Angular module named OrdersModule.
2. Generate a component inside the module called OrderListComponent.
3. Set up **lazy loading** for OrdersModule in app-routing.module.ts.
4. Create a TypeScript **namespace Utils** with a function capitalize().
5. Use the capitalize() function in a component to modify a string.

Conclusion

In this section, we explored:
✓ **How Angular Modules help organize applications**.
✓ **How to create and use Angular Feature Modules**.
✓ **How lazy loading optimizes Angular performance**.
✓ **How TypeScript Namespaces prevent naming conflicts**.

ISDM-NXT

ERROR HANDLING AND DEBUGGING IN TYPESCRIPT

CHAPTER 1: INTRODUCTION TO ERROR HANDLING IN TYPESCRIPT

1.1 Understanding Error Handling in TypeScript

Error handling is an essential part of any application to ensure smooth execution and proper debugging when something goes wrong. **TypeScript,** being a superset of JavaScript**,** provides **static typing** and **better error detection** at compile time**,** reducing runtime errors.

✓ **Detects errors at compile time** – Prevents runtime crashes.
✓ **Improves code quality** – Ensures type safety and correctness.
✓ **Enhances debugging** – Helps developers catch mistakes early.

1.2 Common Types of Errors in TypeScript

Errors in TypeScript can be categorized into three main types:

Error Type	Description	Example
Syntax Errors	Occur when TypeScript code does not follow the correct syntax.	`console.log("Hello World" // Missing closing parenthesis
Type Errors	Occur when a variable is assigned a value of an incorrect type.	let age: number = "twenty-five";
Runtime Errors	Errors that happen when the code is	let obj = null; console.log(obj.value);

ISDM-NXT

	executed, even if compiled correctly.	

✓ **TypeScript reduces runtime errors** by detecting issues at compile time.

CHAPTER 2: USING TRY-CATCH FOR ERROR HANDLING

2.1 Handling Errors with try-catch Blocks

The try-catch block is used to handle **unexpected errors** and prevent application crashes.

Example: Using try-catch to Handle Runtime Errors

```
try {
  let data = JSON.parse('{invalidJson}');
  console.log(data);
} catch (error) {
  console.error("An error occurred:", error);
}
```

✓ Prevents the program from crashing due to **invalid JSON parsing**.

2.2 Throwing Custom Errors

Developers can **manually throw errors** using throw.

Example: Throwing a Custom Error

```
function divide(a: number, b: number): number {
  if (b === 0) {
    throw new Error("Division by zero is not allowed");
```

ISDM-NXT

```
  }
  return a / b;
}

try {
  console.log(divide(10, 0));
} catch (error) {
  console.error(error.message);
}
```

✓ **Prevents illegal operations** like division by zero.

CHAPTER 3: TYPESCRIPT-SPECIFIC ERROR HANDLING TECHNIQUES

3.1 Using Type Guards to Prevent Type Errors

Type guards **validate variable types before use** to prevent unexpected errors.

Example: Using Type Guards

```
function printLength(value: string | number) {
  if (typeof value === "string") {
    console.log(value.length);
  } else {
    console.log("Not a string");
  }
}
```

ISDM-NXT

printLength("Hello"); // Outputs: 5

printLength(123); // Outputs: Not a string

✓ **Prevents type mismatches** and ensures correct function behavior.

3.2 Using Optional Chaining and Nullish Coalescing

✓ **Optional chaining (?.)** prevents errors when accessing properties of null or undefined.
✓ **Nullish coalescing (??)** provides a default value when encountering null or undefined.

Example: Using Optional Chaining (?.)

let user: { name?: string } = {};

console.log(user.name?.toUpperCase()); // No error, returns undefined instead of crashing.

Example: Using Nullish Coalescing (??)

let userInput: string | null = null;

let output = userInput ?? "Default Value";

console.log(output); // Outputs: Default Value

✓ **Prevents crashes when handling null or undefined values**.

CHAPTER 4: DEBUGGING IN TYPESCRIPT

4.1 Debugging with Console Logging

ISDM-NXT

console.log() is the most common way to debug issues in TypeScript.

Example: Using Console Logging for Debugging

```
let userName = "Alice";

console.log("User name is:", userName);
```

✓ Helps track variable values at different execution points.

4.2 Using Breakpoints in VS Code

Visual Studio Code (VS Code) provides powerful debugging features for TypeScript:

1. Open the TypeScript file in VS Code.
2. Add a **breakpoint** by clicking on the left margin of the line number.
3. Press F5 to **start debugging**.
4. Step through the code using **Step Over (F10)** and **Step Into (F11)**.

✓ Breakpoints allow developers to inspect variables and flow control in real-time.

4.3 Debugging with Source Maps

TypeScript compiles into JavaScript, making debugging difficult. **Source maps** allow developers to debug TypeScript **as if it were JavaScript**.

Enable Source Maps in tsconfig.json

```
{
  "compilerOptions": {
    "sourceMap": true
```

ISDM-NXT

```
 }
}
```

✓ This enables debugging in the **browser's Developer Tools**.

Chapter 5: Handling Errors in Promises and Async/Await

5.1 Handling Errors in Promises

If a **Promise rejects,** it should be handled using .catch().

Example: Handling Promise Errors

```
fetch("https://invalid-url.com")
  .then(response => response.json())
  .catch(error => console.error("Fetch error:", error));
```

✓ Prevents **uncaught promise rejection errors**.

5.2 Handling Errors in Async/Await

To handle errors in **async/await,** use try-catch.

Example: Handling Errors in Async Functions

```
async function fetchData() {
  try {
    let response = await fetch("https://invalid-url.com");
    let data = await response.json();
    console.log(data);
  } catch (error) {
```

ISDM-NXT

```
    console.error("Error fetching data:", error);
  }
}

fetchData();
```

✓ Prevents **application crashes** due to failed API calls.

Case Study: How an E-Commerce Platform Improved Stability with TypeScript Error Handling

Background

An e-commerce platform faced frequent **runtime errors** in production, causing:

✓ **Crashes during checkout,** frustrating customers.
✓ **Slow debugging process,** increasing development time.
✓ **Uncaught API errors,** leading to incomplete transactions.

Challenges

- **Type mismatches** caused runtime failures.
- **Unhandled errors in async operations** led to **cart losses**.
- **Lack of logging and debugging tools** made issue tracking difficult.

Solution: Implementing TypeScript Error Handling Techniques

The development team:

ISDM-NXT

✓ Used **strict typing** to prevent type mismatches.
✓ Implemented **try-catch blocks** in async operations.
✓ Enabled **source maps for easier debugging**.

Results

🚀 **Reduced production errors by 80%**.
⚡ **Faster bug resolution,** improving development speed.
🔍 **Improved customer experience,** increasing conversion rates.

By using **TypeScript error handling best practices,** the platform ensured **stable and reliable operations**.

Exercise

1. Write a function that **divides two numbers** but throws an error if the denominator is 0.
2. Create an **async function** that fetches data from a non-existing API and handles errors.
3. Implement **optional chaining** in an object and check if a property exists.
4. Enable **source maps** in tsconfig.json and debug a TypeScript file in VS Code.

Conclusion

In this section, we explored:
✓ **How TypeScript improves error handling through compile-time checks**.
✓ **How to handle errors using try-catch, type guards, and optional chaining**.

ISDM-NXT

✓ **How to debug TypeScript applications using breakpoints and source maps.**

✓ **How to manage errors in async operations using Promises and async/await.**

ISDM-NXT

ASSIGNMENT:

BUILD A BASIC ANGULAR APPLICATION USING TYPESCRIPT AND CLI

ISDM-NXT

SOLUTION: BUILDING A BASIC ANGULAR APPLICATION USING TYPESCRIPT AND CLI

Step 1: Install Angular CLI and Create a New Angular Project

1.1 Installing Angular CLI

First, ensure that **Node.js** is installed on your system. Then, install Angular CLI globally using the following command:

npm install -g @angular/cli

To verify that Angular CLI is installed correctly, run:

ng version

1.2 Creating a New Angular Project

Now, create a new Angular project using:

ng new my-angular-app

✓ When prompted, select **CSS** for styling.

Move into the project directory:

cd my-angular-app

Start the development server to see the default Angular app:

ng serve

✓ Open http://localhost:4200/ in a browser to view the app.

Step 2: Understanding the Project Structure

When the project is created, the following structure appears:

ISDM-NXT

```
my-angular-app/
├── src/
│   ├── app/
│   │   ├── app.component.ts
│   │   ├── app.component.html
│   │   ├── app.component.css
│   │   ├── app.module.ts
│   │   ├── home/ (to be created)
│   │   │   ├── home.component.ts
│   │   │   ├── home.component.html
│   │   │   ├── home.component.css
│   ├── assets/
│   ├── environments/
│   ├── main.ts
├── angular.json
├── package.json
```

Step 3: Creating a New Component

Angular applications are built using **components**. We will create a new **HomeComponent** to display a simple welcome message.

3.1 Generate a New Component

Run the following command:

ISDM-NXT

ng generate component home

✓ This creates a new component inside src/app/home/.

Step 4: Updating the Home Component

4.1 Editing home.component.ts

Modify home.component.ts to define a message variable using **TypeScript**:

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


@Component({
 selector: 'app-home',
 templateUrl: './home.component.html',
 styleUrls: ['./home.component.css']
})
export class HomeComponent {
 message: string = "Welcome to My Angular Application!";
}
```

✓ Defines a message variable and binds it to the template.

4.2 Editing home.component.html

Modify home.component.html to display the message dynamically:

```
<h2>{{ message }}</h2>
```

✓ **{{ message }}** uses Angular **interpolation** to display the variable content.

ISDM-NXT

Step 5: Integrating the Home Component in the Main App

5.1 Editing app.component.html

Replace the default content of app.component.html with:

<h1>My Angular Application</h1>

<app-home></app-home>

✓ **<app-home>** is the **selector** of the HomeComponent, embedding it inside AppComponent.

Step 6: Running the Application

Now, restart the development server:

ng serve

Open http://localhost:4200/, and you should see:

My Angular Application

Welcome to My Angular Application!

Step 7: Adding Basic Styling

Modify home.component.css to style the text:

```
h2 {
  color: blue;
  text-align: center;
}
```

ISDM-NXT

✓ Refresh the browser, and the **message should now appear in blue**.

Step 8: Adding a Button for Interaction

Modify home.component.html to include a button:

```
<h2>{{ message }}</h2>

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

Modify home.component.ts to add a method that updates the message:

```
changeMessage(): void {
  this.message = "You clicked the button!";
}
```

✓ Clicking the button **changes the message dynamically**.

Step 9: Final Testing and Deployment

9.1 Testing the Application

Run:

```
ng test
```

✓ Ensures all components are working correctly.

9.2 Building for Production

To create a **production-ready build**, run:

```
ng build --prod
```

ISDM-NXT

✓ Generates optimized files inside the dist/ folder, ready for deployment.

Final Code Summary

home.component.ts

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


@Component({
 selector: 'app-home',
 templateUrl: './home.component.html',
 styleUrls: ['./home.component.css']
})
export class HomeComponent {
 message: string = "Welcome to My Angular Application!";


 changeMessage(): void {
  this.message = "You clicked the button!";
 }
}
```

home.component.html

```
<h2>{{ message }}</h2>
<button (click)="changeMessage()">Click Me</button>
```

ISDM-NXT

Case Study: How a Tech Startup Built an Angular SPA for Client Management

Background

A **tech startup** wanted a **lightweight, fast** client management dashboard.

Challenges

✓ **Slow page reloads** in traditional multi-page apps.
✓ **No interactive UI elements**.
✓ **Difficult to scale** with growing client data.

Solution: Using Angular SPA

✓ **Implemented Angular components** for modular design.
✓ **Used TypeScript for type safety and scalability**.
✓ **Integrated button click functionality** to update the UI dynamically.

Results

🚀 **50% faster UI interactions**.
⚡ **Seamless page transitions without reloads**.
🔍 **Easier debugging and maintainability with TypeScript**.

Exercise

1. Modify the home.component.ts to display the **current date and time**.
2. Add a button that changes the **background color** of the message dynamically.
3. Create a new component called about and display it below home.

ISDM-NXT

4. Deploy the Angular app to a cloud platform like **Firebase or Netlify**.

Conclusion

In this guide, we learned:

✓ **How to create an Angular application using TypeScript and CLI.**

✓ **How to define components and bind data dynamically.**

✓ **How to handle user interactions using event binding.**

✓ **How to deploy an Angular application for production.**

ISDM-NXT