Angular Development Training

Duration: 10 days, 6 hours per day.

Objectives:

- Enable participants to build modern web applications using Angular.
- Teach core concepts like components, forms, routing, services, and state management.

Expected Outcomes:

- Participants will create a fully functional Angular application.
- They will understand how to optimize and test Angular apps.

Hardware:

- Laptops/desktops with at least:
    1. Processor: Intel i5 or equivalent.
    2. RAM: 8GB (16GB Preferred)
    3. Storage: 256GB SSD or higher.
- Internet connection

- Node.js (Latest LTS version).
- Angular CLI (npm install -g @angular/cli).
- Visual Studio Code with extensions:
    1. Angular Essentials
    2. Prettier
    3. TypeScript

Training Methodology

- **Lectures**: Theory explained with slides and diagrams.
- **Live Coding**: Demonstrations of Angular concepts in real-time.
- **Hands-On Exercises**: Daily exercises to reinforce learning.
- **Capstone Project**: Building a task management system over the 10 days.
- **Assessments**: Final project submission.

Table of Content:

Introduction to Angular and TypeScript

Learning Objectives:

- Understand the purpose and key architectural features of Angular.
- Gain familiarity with TypeScript fundamentals and its importance in Angular development.
- Set up the Angular development environment on the local machine.

Topics Covered:

What is Angular?

- Overview of Angular as a platform and framework for building modern web applications.
- Key features of Angular:
  - Component-based architecture.
  - Dependency Injection.
  - Powerful data binding.
  - Built-in routing and form management.
  - Support for modular development.
- Angular's popularity in the industry and its advantages over other frameworks.

Angular vs. Other Frameworks:

- Differences between Angular and React, Vue.js:
  - Opinionated framework vs. library approach.
  - Full-featured ecosystem (Angular) vs. lightweight with optional libraries (React, Vue.js).
- Use cases where Angular is a better fit (e.g., enterprise applications).

Introduction to Single Page Applications (SPA):

- Definition and benefits of SPAs.
- How Angular enables SPAs with efficient routing and state management.
- The role of Angular in building dynamic, seamless user experiences.

Installing Development Tools:

- **Node.js**: Installing the latest LTS version.
- **Angular CLI**: Using `npm` to install Angular's Command Line Interface.

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

- Install Angular CLI using `npm`:

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

   - Download and install Visual Studio Code from code.visualstudio.com.
   - Install recommended VS Code extensions.

Create a New Angular Project Using Angular CLI:

- Create a new project:

```
ng new my-first-app
```

- Choose routing support and CSS/SCSS as the styling option.
- Navigate to the project directory:

```
cd my-first-app
```

- Run the app:

```
ng serve
```

- Access the app in the browser at `http://localhost:4200`.

Modify the Default Angular App Template:

- Open the project in VS Code.
- Navigate to `src/app/app.component.html`.
- Replace the default content with a custom message:

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

<p>This app was created during the Angular training course.</p>
```

Write a Basic TypeScript Class:

- Create a new TypeScript file `src/app/person.ts`.
- Define a simple `Person` class with properties, a constructor, and a method:

```
export 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.`;
  }
}
```

- Use the class in the `AppComponent`:

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  person: Person = new Person('John Doe', 30);

  getGreeting(): string {
        return this.person.greet();
  }
}
```

- Update the `app.component.html` file to display the greeting:

```
<h1>{{ getGreeting() }}</h1>
```

Expected Outputs

1. Successfully installed Node.js, Angular CLI, and VS Code.
2. A new Angular app running locally with a custom welcome message displayed.
3. A working TypeScript class integrated into the Angular app to display a dynamic message.

Angular Components and Templates

Learning Objectives

- Understand the role and structure of Angular components.
- Learn how templates and metadata define component behavior and presentation.
- Work with data binding to create dynamic, interactive UIs.

Topics Covered

Anatomy of an Angular Component

- **Component Decorators:** Metadata to configure components (`@Component`).
- **Component Class:** Defines logic, properties, and methods.
- **Template and Styles:** HTML and CSS for the component's view.

Creating Components

- Using Angular CLI to generate components:

```
ng generate component component-name
```

- What happens during component generation.
- Component structure: `.ts`, `.html`, `.css`, and `.spec.ts` files.

Data Binding

- **Property Binding:** Bind DOM element properties to component properties:

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

- **Event Binding:** Capture and respond to user actions:

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

- **Two-Way Binding:** Sync component data with form elements using `[(ngModel)]`:

```
<input [(ngModel)]="userName">
```

Component Communication

- **@Input Decorator:** Pass data from parent to child components.
- **@Output Decorator:** Emit events from child to parent components.
- Creating a reusable component and passing data to it.

Templates and Metadata

- Using inline templates and styles vs. separate files.
- Template expressions and directives (`*ngIf`, `*ngFor`).

Exercises

Create a Header and Footer Component:

- Generate components using Angular CLI:

```
ng generate component header

ng generate component footer
```

- Update the `app.component.html` to include these components:

```
<app-header></app-header>

<div>

  <h1>Welcome to Task Manager</h1>

  <p>Manage your tasks efficiently.</p>

</div>

<app-footer></app-footer>
```

- Add basic styles to make the header and footer visually distinct.

Dynamic Header Title Using Property Binding

- Add a `title` property to `HeaderComponent`

```
export class HeaderComponent {
  title = 'Task Manager';
}
```

- Update the `header.component.html`:

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

Add Interactivity with Event Binding

- Add a button to `HeaderComponent` and handle click events:

```
<button (click)="changeTitle()">Change Title</button>
```

- Update the component class to handle the event:

```
export class HeaderComponent {
  title = 'Task Manager';

  changeTitle() {
        this.title = 'Updated Task Manager';
  }
}
```

Two-Way Data Binding in the Footer

- Add an input field to `FooterComponent` for user feedback:

```html
<input [(ngModel)]="feedback" placeholder="Enter your feedback">
<p>Your feedback: {{ feedback }}</p>
```

- Add a `feedback` property to `FooterComponent`:

```typescript
export class FooterComponent {
  feedback = '';
}
```

Parent-Child Communication Using @Input and @Output

- Create a `TaskItemComponent`:

```
ng generate component task-item
```

- Add an `@Input` property to accept task data:

```typescript
@Input() task: string;
```

- In `app.component.html`, pass a task to the `TaskItemComponent`:

```html
<app-task-item [task]="'Complete Angular exercise'"></app-task-item>
```

- Add a button to `TaskItemComponent` to emit an event using `@Output`:

```typescript
@Output() taskCompleted = new EventEmitter<string>();

completeTask() {
  this.taskCompleted.emit(this.task);
}
```

- Handle the event in `AppComponent`:

```html
<app-task-item [task]="'Complete Angular exercise'"
(taskCompleted)="onTaskCompleted($event)"></app-task-item>
```

Expected Outputs

1. A `HeaderComponent` displaying a dynamic title, with a button to change it.
2. A `FooterComponent` with an interactive input field for user feedback using two-way data binding.
3. A `TaskItemComponent` that receives a task via `@Input` and emits an event when the task is marked complete.
4. Updated `AppComponent` integrating the header, footer, and task item components.

Integration to Project

The components created will form the foundation of the **Task Manager Application**:

- Header for navigation or branding.
- Footer for user feedback.
- Task items for displaying task details dynamically.

Directives and Pipes

Learning Objectives

- Understand how to manipulate the DOM using Angular directives.
- Use structural and attribute directives to create dynamic, conditional, and styled UI elements.
- Transform and format data using built-in and custom pipes.

Topics Covered

Introduction to Directives

- **Structural Directives**: Change the DOM layout by adding or removing elements.
  - Examples: `*ngIf`, `*ngFor`, `*ngSwitch`.
- **Attribute Directives**: Change the appearance or behavior of DOM elements.
  - Examples: `ngClass`, `ngStyle`.

Using Built-in Directives

- **`*ngIf`**: Conditionally render an element.

```
<p *ngIf="isTaskCompleted">Task Completed!</p>
```

- **`*ngFor`**: Loop over a collection and display each item.

```
<ul>
  <li *ngFor="let task of tasks">{{ task }}</li>
</ul>
```

- **`ngClass`**: Dynamically apply CSS classes.

```
<p [ngClass]="{ 'completed': isCompleted, 'pending': !isCompleted }">Task</p>
```

- **`ngStyle`**: Dynamically apply inline styles.

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

Introduction to Pipes

- **Built-in Pipes**: Format data like dates, numbers, and text.
  - Examples: `date`, `currency`, `uppercase`, `lowercase`, `percent`.
- **Chaining Pipes**:

```
{{ taskDueDate | date:'short' | uppercase }}
```

- **Parameterizing Pipes**:

```
{{ amount | currency:'USD':'symbol':'1.2-2' }}
```

Creating Custom Pipes

- Transform data according to specific requirements.
- Example: Capitalize the first letter of a string.

Combining Directives and Pipes

- Use directives to display lists of tasks and pipes to format task data dynamically.

Exercises

Using Structural Directives with Tasks

- In `AppComponent`, define a list of tasks:
```
tasks = [
  { title: 'Complete Angular exercise', completed: false },
  { title: 'Read Angular documentation', completed: true },
];
```
- Display tasks conditionally using `*ngIf` and `*ngFor`:
```
<ul>
  <li *ngFor="let task of tasks">
        <span *ngIf="task.completed" style="color: green;">✔</span>
        <span *ngIf="!task.completed" style="color: red;">✘</span>
        {{ task.title }}
  </li>
</ul>
```

Style Tasks Dynamically with Attribute Directives

- Use `ngClass` to style completed and pending tasks:
```
<li *ngFor="let task of tasks" [ngClass]="{ 'completed': task.completed, 'pending':
!task.completed }">
  {{ task.title }}
</li>
```
- Add styles in `styles.css`:
```
.completed {
  text-decoration: line-through;
  color: green;
}
.pending {
  color: red;
}
```

Format Data Using Pipes

- Add a due date to each task and display it using the `date` pipe:
```
tasks = [
  { title: 'Complete Angular exercise', completed: false, dueDate: new Date() },
  { title: 'Read Angular documentation', completed: true, dueDate: new
Date('2024-12-25') },
];
```
- Format the due date in the template:
```
<li *ngFor="let task of tasks">
```

```
  {{ task.title }} - Due: {{ task.dueDate | date:'shortDate' }}
</li>
```

Create a Custom Pipe:

- Generate a custom pipe using Angular CLI:

```
ng generate pipe capitalize
```

- Implement the `CapitalizePipe` to capitalize the first letter of each word:

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

@Pipe({
  name: 'capitalize',
})
export class CapitalizePipe implements PipeTransform {
  transform(value: string): string {
          return value
          .split(' ')
          .map(word => word.charAt(0).toUpperCase() + word.slice(1))
          .join(' ');
  }
}
```

- Use the pipe in the template:

```
<li *ngFor="let task of tasks">
  {{ task.title | capitalize }}
</li>
```

Combine Pipes and Directives

- Use `*ngIf` to show a message if no tasks are available:

```
<p *ngIf="tasks.length === 0">No tasks available!</p>
```

- Format task details with pipes and style them dynamically.

Expected Outputs

1. A styled list of tasks where completed tasks are visually distinct using `ngClass` and `ngStyle`.
2. Task due dates formatted using the `date` pipe.
3. Dynamic messages displayed conditionally using `*ngIf`.
4. A custom pipe applied to task titles to capitalize the first letter of each word.

Integration to Project

- Enhance the **Task Manager Application**:
    - Create a dynamic and styled task list using `*ngFor` and `*ngIf`.
    - Apply custom formatting with pipes.
    - Add visual indicators for completed and pending tasks using `ngClass`.

Forms in Angular

Learning Objectives

- Build and validate forms using Angular's template-driven and reactive approaches.
- Understand how to use Angular's `FormsModule` and `ReactiveFormsModule`.
- Implement dynamic form controls and real-time validation.

Topics Covered

Overview of Angular Forms

- Forms in Angular: Template-driven vs. Reactive.
- When to use each approach:
  - Template-driven: Simple, less complex forms.
  - Reactive: Complex, dynamic, or programmatically controlled forms.
- Adding `FormsModule` and `ReactiveFormsModule` to the app:

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

@NgModule({
  imports: [
        FormsModule,
        ReactiveFormsModule
  ],
})
```

Template-driven Forms

- Binding HTML forms to Angular components using `ngModel`.
- Creating and using form controls.
- Implementing validation in template-driven forms:
  - Built-in validators (e.g., `required`, `minlength`, `pattern`).
  - Displaying validation messages.

Reactive Forms

- Using the `FormGroup` and `FormControl` classes to define forms programmatically.
- Adding validators and reacting to form state changes.
- Dynamically adding and removing form controls.

Custom Validation

- Writing custom validators for advanced validation rules.

Form Submission and Handling Data

- Handling form submission in Angular.
- Binding form data to the component logic.

Exercises

Create a Template-driven Form for Adding Tasks

- Add a form to `app.component.html`:

```
<form #taskForm="ngForm" (ngSubmit)="onSubmit(taskForm)">
  <div>
        <label for="title">Title:</label>
        <input type="text" id="title" name="title" [(ngModel)]="task.title" required>
        <div *ngIf="taskForm.controls.title?.invalid &&
taskForm.controls.title?.touched">
        Title is required.
        </div>
  </div>
  <div>
        <label for="description">Description:</label>
        <textarea id="description" name="description" [(ngModel)]="task.description"
required></textarea>
        <div *ngIf="taskForm.controls.description?.invalid &&
taskForm.controls.description?.touched">
        Description is required.
        </div>
  </div>
  <div>
        <label for="dueDate">Due Date:</label>
        <input type="date" id="dueDate" name="dueDate"
[(ngModel)]="task.dueDate" required>
  </div>
  <button type="submit" [disabled]="taskForm.invalid">Add Task</button>
</form>
```

- Update `AppComponent` to handle the form submission:

```
export class AppComponent {
  task = {
        title: '',
        description: '',
        dueDate: ''
  };

  onSubmit(taskForm: any) {
        console.log('Task added:', this.task);
        taskForm.reset();
  }
}
```

Convert the Template-driven Form to a Reactive Form

- Update `app.component.ts`:

```
import { Component } from '@angular/core';
import { FormGroup, FormControl, Validators } from '@angular/forms';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  taskForm = new FormGroup({
        title: new FormControl('', [Validators.required]),
        description: new FormControl('', [Validators.required]),
        dueDate: new FormControl('', [Validators.required]),
  });

  onSubmit() {
        console.log('Task added:', this.taskForm.value);
        this.taskForm.reset();
  }
}
```

- Update `app.component.html`:

```
<form [formGroup]="taskForm" (ngSubmit)="onSubmit()">
  <div>
        <label for="title">Title:</label>
        <input type="text" id="title" formControlName="title">
        <div *ngIf="taskForm.controls.title.invalid && taskForm.controls.title.touched">
        Title is required.
        </div>
  </div>
  <div>
        <label for="description">Description:</label>
        <textarea id="description" formControlName="description"></textarea>
        <div *ngIf="taskForm.controls.description.invalid &&
taskForm.controls.description.touched">
        Description is required.
        </div>
  </div>
  <div>
        <label for="dueDate">Due Date:</label>
        <input type="date" id="dueDate" formControlName="dueDate">
  </div>
  <button type="submit" [disabled]="taskForm.invalid">Add Task</button>
</form>
```

Add Custom Validation

- Add a custom validator to check for a minimum title length:
```
import { AbstractControl, ValidationErrors } from '@angular/forms';

export function minLengthValidator(control: AbstractControl): ValidationErrors | null {
  const value = control.value as string;
  if (value && value.length < 5) {
        return { minLength: true };
  }
  return null;
}
```
- Apply the custom validator to the title field:
```
title: new FormControl('', [Validators.required, minLengthValidator]),
```
- Update the template to show an error message for the custom validation:
```
<div *ngIf="taskForm.controls.title.errors?.minLength">
  Title must be at least 5 characters long.
</div>
```

Dynamically Add or Remove Form Controls

- Add a dynamic tags input for tasks:
```
import { FormArray } from '@angular/forms';

taskForm = new FormGroup({
  title: new FormControl('', [Validators.required]),
  description: new FormControl('', [Validators.required]),
  dueDate: new FormControl('', [Validators.required]),
  tags: new FormArray([]),
});

get tags() {
  return this.taskForm.get('tags') as FormArray;
}

addTag(tag: string) {
  this.tags.push(new FormControl(tag));
}

removeTag(index: number) {
  this.tags.removeAt(index);
}
```
- Update the template to include dynamic tags:
```
<div>
  <label for="tags">Tags:</label>
  <div *ngFor="let tag of tags.controls; let i = index">
        <input [formControlName]="i">
        <button (click)="removeTag(i)">Remove</button>
  </div>
  <button (click)="addTag('New Tag')">Add Tag</button>
</div>
```

Expected Outputs

1. A template-driven form for adding tasks with validation.
2. A reactive form with validators and dynamic behavior.
3. A task creation form with custom validation for title length.
4. A dynamic tag management system integrated into the form.

Integration to Project

- The task creation form will become part of the **Task Manager Application**, allowing users to add and validate tasks.
- Custom validators and dynamic form controls will enhance usability.

Routing and Navigation

Learning Objectives

- Learn how to set up and configure routes in an Angular application.
- Pass data between routes using parameters and query strings.
- Use navigation guards to control access to specific routes.

Topics Covered

Introduction to Angular Router

- Overview of routing in Single Page Applications (SPAs).
- Benefits of using Angular Router.
- Configuring routes in `AppModule`:

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

const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'tasks', component: TaskListComponent },
  { path: 'task/:id', component: TaskDetailsComponent },
];

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

Setting Up Routes

- Adding route definitions in the `AppRoutingModule`.
- Linking to routes using `<a routerLink="/path">` or `routerLinkActive` for active states.
- Lazy loading feature modules to optimize app performance.

Passing Data with Routes

- Using **route parameters** to pass dynamic data (e.g., task ID).
- Accessing parameters in a component using `ActivatedRoute`:

```
constructor(private route: ActivatedRoute) {}
ngOnInit() {
  const id = this.route.snapshot.paramMap.get('id');
  console.log(id);
}
```

- Using **query strings** to pass additional data.

Navigation Guards

- Protecting routes with `canActivate` and `canDeactivate` guards.
- Example: Prevent navigation if a task form is incomplete.

```
export class CanDeactivateGuard implements
CanDeactivate<TaskFormComponent> {
  canDeactivate(
        component: TaskFormComponent
  ): boolean {
        return component.isFormSaved() || confirm('Unsaved changes! Are you sure
you want to leave?');
  }
}
```

Child Routes

- Nested routing for modular components.
- Example: Creating a dashboard with sub-routes for statistics and settings.

Programmatic Navigation

- Using `Router` service to navigate dynamically:

```
this.router.navigate(['/task', taskId]);
```

Exercises

Configure Basic Routes

- Define routes for `Home`, `Tasks`, and `Task Details` pages in `AppRoutingModule`.
- Update `app.component.html` with a navigation menu:

```
<nav>
  <a routerLink="/" routerLinkActive="active">Home</a>
  <a routerLink="/tasks" routerLinkActive="active">Tasks</a>
</nav>
<router-outlet></router-outlet>
```

- Verify navigation between pages.

Pass Parameters to Task Details

- Update the `TaskListComponent` template to include a link to task details:

```
<ul>
  <li *ngFor="let task of tasks">
        <a [routerLink]="['/task', task.id]">{{ task.title }}</a>
  </li>
</ul>
```

- Retrieve the task ID in `TaskDetailsComponent`:

```
constructor(private route: ActivatedRoute) {}
ngOnInit() {
  const taskId = this.route.snapshot.paramMap.get('id');
  console.log('Task ID:', taskId);
}
```

Use Query Strings for Filtering Tasks

- Update `TaskListComponent` to add a filter link with query parameters:
  ```
  <button [routerLink]="['/tasks']" [queryParams]="{ status: 'completed' }">Show Completed</button>
  ```
- Access query parameters in `TaskListComponent`:
  ```
  this.route.queryParams.subscribe(params => {
    const status = params['status'];
    console.log('Filter Status:', status);
  });
  ```

Add a Route Guard

- Create a `canDeactivate` guard for `TaskFormComponent`:
  ```
  @Injectable({ providedIn: 'root' })
  export class CanDeactivateGuard implements CanDeactivate<TaskFormComponent> {
    canDeactivate(component: TaskFormComponent): boolean {
          return component.isSaved || confirm('You have unsaved changes. Leave anyway?');
    }
  }
  ```
- Apply the guard to the `task/add` route:
  ```
  { path: 'task/add', component: TaskFormComponent, canDeactivate: [CanDeactivateGuard] }
  ```

Programmatic Navigation

- Add a button to redirect to the home page programmatically:
  ```
  <button (click)="goHome()">Go to Home</button>
  ```
- Implement the navigation in `TaskDetailsComponent`:
  ```
  constructor(private router: Router) {}
  goHome() {
    this.router.navigate(['/']);
  }
  ```

Expected Outputs

1. A functional navigation system allowing users to switch between `Home`, `Tasks`, and `Task Details`.
2. Task details page dynamically showing information based on the route parameter.
3. Filter tasks using query parameters in the URL.
4. Route guard preventing users from leaving a form with unsaved changes.
5. Programmatic navigation integrated into the app.

Integration to Project

- **Task Manager Application** will now have:
    - A navigation bar for switching between pages.
    - A task list linked to task details pages using route parameters.
    - Route guards to ensure data integrity.

Services and Dependency Injection

Learning Objectives

- Understand the role of services in Angular for sharing data and logic between components.
- Learn how Angular's Dependency Injection system works and its benefits.
- Use Angular's `HttpClient` to interact with APIs for fetching and manipulating data.

Topics Covered

What are Angular Services?

- Definition and purpose of services.
- Reusable logic and centralized data management.
- How services promote modularity and clean code practices.

Creating and Using Services

- Creating a service using Angular CLI:
  `ng generate service task`
- Injecting a service into a component:
  `constructor(private taskService: TaskService) {}`

Dependency Injection (DI) in Angular

- How DI works and its advantages.
- Angular's hierarchical injector and provider configuration.
- Using `@Injectable` decorator to register services.

HttpClient for API Interaction

- Setting up `HttpClientModule` in `AppModule`:
  ```
  import { HttpClientModule } from '@angular/common/http';

  @NgModule({
    imports: [HttpClientModule],
  })
  ```
- Performing HTTP operations:
  - GET: Fetching data from an API.
  - POST: Adding new data.
  - PUT: Updating existing data.
  - DELETE: Removing data.
- Handling HTTP errors with RxJS operators like `catchError`.

Observable-based Services

- Using Observables to return and manage asynchronous data.
- Subscribing to Observables in components to fetch data.

Exercises

Create a `TaskService` for Managing Tasks

- Generate the service:
  ```
  ng generate service task
  ```
- Add methods to manage tasks in `task.service.ts`:
  ```
  import { Injectable } from '@angular/core';

  @Injectable({
    providedIn: 'root',
  })
  export class TaskService {
    private tasks = [
          { id: 1, title: 'Learn Angular', completed: false },
          { id: 2, title: 'Build an app', completed: true },
    ];

    getTasks() {
          return this.tasks;
    }

    addTask(task: any) {
          this.tasks.push(task);
    }
  }
  ```

Inject the `TaskService` into a Component

- Use the service in `TaskListComponent` to fetch and display tasks:
  ```
  import { Component, OnInit } from '@angular/core';
  import { TaskService } from '../task.service';

  @Component({
    selector: 'app-task-list',
    templateUrl: './task-list.component.html',
    styleUrls: ['./task-list.component.css'],
  })
  export class TaskListComponent implements OnInit {
    tasks: any[] = [];

    constructor(private taskService: TaskService) {}
  ```

```
  ngOnInit(): void {
        this.tasks = this.taskService.getTasks();
  }
}
```

- Display the tasks in the `task-list.component.html`:

```
<ul>
  <li *ngFor="let task of tasks">
        {{ task.title }}
  </li>
</ul>
```

Add HTTP Integration Using `HttpClient`

- Replace the static task list in `TaskService` with data fetched from an API:

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

@Injectable({
  providedIn: 'root',
})
export class TaskService {
  private apiUrl = 'https://jsonplaceholder.typicode.com/todos';

  constructor(private http: HttpClient) {}

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

- Update `TaskListComponent` to subscribe to the Observable and fetch tasks:

```
ngOnInit(): void {
  this.taskService.getTasks().subscribe((data) => {
        this.tasks = data.slice(0, 10); // Limit to 10 tasks for display
  });
}
```

Handle Errors in HTTP Requests

- Modify `TaskService` to handle errors using `catchError`:

```
import { catchError } from 'rxjs/operators';
import { of } from 'rxjs';

getTasks(): Observable<any> {
  return this.http.get<any[]>(this.apiUrl).pipe(
        catchError((error) => {
        console.error('Error fetching tasks', error);
```

```
            return of([]); // Return an empty array on error
            })
    );
  }
```

Add Task Functionality

- Add a method in `TaskService` to POST a new task to the API:

  ```
  addTask(task: any): Observable<any> {
    return this.http.post(this.apiUrl, task);
  }
  ```
- Create a form in `TaskFormComponent` to add tasks:

  ```
  <form (ngSubmit)="onSubmit()">
    <label for="title">Title:</label>
    <input type="text" id="title" [(ngModel)]="task.title" name="title" required>
    <button type="submit">Add Task</button>
  </form>
  ```
- Update `TaskFormComponent` to call `addTask` on form submission:

  ```
  onSubmit() {
    this.taskService.addTask(this.task).subscribe((newTask) => {
            console.log('Task added:', newTask);
    });
  }
  ```

Expected Outputs

1. A `TaskService` managing all task-related operations.
2. A `TaskListComponent` fetching tasks from an API and displaying them in a list.
3. A fully functional `TaskFormComponent` allowing users to add tasks to the API.
4. Error handling implemented for HTTP requests.

Integration to Project

- **Task Manager Application**:
  - Centralized task management using `TaskService`.
  - Dynamic task fetching from an external API.
  - Task creation integrated with a live API.

Observables and RxJS

Learning Objectives

- Understand what Observables are and their importance in Angular.
- Use RxJS to handle asynchronous data streams in Angular applications.
- Learn about common RxJS operators and how to use them for transforming data streams.
- Implement error handling and dynamic search using Observables.

Topics Covered

Introduction to Observables

- What are Observables?
  - Asynchronous streams of data.
  - A core concept in Angular for handling async operations.
- Observables vs Promises:
  - Observables are lazy and can handle multiple values over time.
  - Promises are eager and resolve a single value.
- Creating, subscribing to, and unsubscribing from Observables.

Using Observables with Angular

- Observables in Angular's `HttpClient`.
- Subscribing to Observables in components.
- Managing subscriptions to prevent memory leaks with `OnDestroy`.

Introduction to RxJS

- Overview of the Reactive Extensions for JavaScript (RxJS).
- Common RxJS operators:
  - `map`: Transform data.
  - `filter`: Filter data based on conditions.
  - `debounceTime`: Add delay to avoid frequent updates (useful in search inputs).
  - `switchMap`: Cancel previous Observable and subscribe to the latest one.

Error Handling in Observables

- Using `catchError` to handle errors.
- Providing fallback values with `of`.

Dynamic Search with Observables

- Using `Subject` and `BehaviorSubject` for user input handling.

- Building a dynamic search bar using Observables.

Exercises

Creating and Subscribing to Observables

- Create a simple Observable in a service:
  ```
  import { Injectable } from '@angular/core';
  import { Observable } from 'rxjs';

  @Injectable({
    providedIn: 'root',
  })
  export class ExampleService {
    getObservable(): Observable<string> {
          return new Observable((observer) => {
          observer.next('Hello');
          observer.next('World');
          observer.complete();
          });
    }
  }
  ```
- Subscribe to the Observable in a component:
  ```
  constructor(private exampleService: ExampleService) {}

  ngOnInit() {
    this.exampleService.getObservable().subscribe({
          next: (value) => console.log(value),
          complete: () => console.log('Observable completed'),
    });
  }
  ```

Fetch and Transform Data Using RxJS Operators

- Modify `TaskService` to use `map` to transform API data:
  ```
  getTasks(): Observable<any[]> {
    return this.http.get<any[]>(this.apiUrl).pipe(
          map((tasks) => tasks.map((task) => ({ ...task, completed: !!task.completed })))
    );
  }
  ```
- Display transformed tasks in `TaskListComponent`.

Implement a Dynamic Search Bar

- Update `TaskListComponent` to include a search bar:
  ```
  <input type="text" (input)="onSearch($event.target.value)" placeholder="Search
  tasks...">
  <ul>
  ```

```
  <li *ngFor="let task of filteredTasks">{{ task.title }}</li>
</ul>
```

- Update the component logic to handle dynamic searching:

```
import { Subject } from 'rxjs';
import { debounceTime, distinctUntilChanged, switchMap } from 'rxjs/operators';

export class TaskListComponent implements OnInit {
  tasks: any[] = [];
  filteredTasks: any[] = [];
  searchTerms = new Subject<string>();

  constructor(private taskService: TaskService) {}

  ngOnInit() {
        this.taskService.getTasks().subscribe((tasks) => {
        this.tasks = tasks;
        this.filteredTasks = tasks;
        });

        this.searchTerms.pipe(
        debounceTime(300),
        distinctUntilChanged(),
        switchMap((term) =>
        this.taskService.getTasks().pipe(
        map((tasks) =>
        tasks.filter((task) => task.title.toLowerCase().includes(term.toLowerCase()))
        )
        )
        )
        ).subscribe((filteredTasks) => {
        this.filteredTasks = filteredTasks;
        });
  }

  onSearch(term: string): void {
        this.searchTerms.next(term);
  }
}
```

Handle Errors Gracefully

- Update `TaskService` to handle errors with `catchError`:

```
import { of } from 'rxjs';
import { catchError } from 'rxjs/operators';

getTasks(): Observable<any[]> {
  return this.http.get<any[]>(this.apiUrl).pipe(
        catchError((error) => {
        console.error('Error fetching tasks', error);
        return of([]);
        })
  );
}
```

- Show a fallback message in `TaskListComponent` if an error occurs.

Expected Outputs

1. Tasks are fetched and displayed dynamically using Observables.
2. Search functionality filters tasks dynamically based on user input with debouncing.
3. Errors during data fetching are gracefully handled and logged.

Integration to Project

- **Task Manager Application**:
  - Fetch tasks asynchronously using Observables.
  - Implement a dynamic, real-time search bar with debouncing for filtering tasks.
  - Ensure robust error handling for better user experience.

State Management

Learning Objectives

- Understand the importance of state management in Angular applications.
- Learn to manage state using Angular services and local storage.
- Gain an introduction to state management libraries like NgRx for more complex applications.
- Use actions, reducers, and stores to manage state effectively.

Topics Covered

What is State Management?

- Definition and importance of state in Angular applications.
- Challenges of managing state in large applications.
- Examples of state: UI state, application state, server state.

Managing State with Angular Services

- Using services to centralize state logic.
- Sharing state between components via dependency injection.

Local Storage for State Persistence

- How to save and retrieve state from local storage for persistence across sessions.

Introduction to NgRx (Optional for Advanced Users)

What is NgRx, and when should it be used?

- **Core concepts of NgRx:**
    - **Actions: Dispatching events.**
    - **Reducers: Managing state updates.**
    - **Store: A central location for state.**
    - **Effects: Handling side effects (e.g., API calls).**

Comparing Service-based and NgRx-based State Management

- Benefits and trade-offs of each approach.

Exercises

Implement Service-based State Management

- Update `TaskService` to manage state for tasks:
  import { Injectable } from '@angular/core';

```
@Injectable({
  providedIn: 'root',
})
export class TaskService {
  private tasks: any[] = [];

  getTasks() {
          return this.tasks;
  }

  addTask(task: any) {
          this.tasks.push(task);
  }

  updateTask(id: number, updatedTask: any) {
          const index = this.tasks.findIndex((task) => task.id === id);
          if (index !== -1) {
          this.tasks[index] = { ...this.tasks[index], ...updatedTask };
          }
  }

  deleteTask(id: number) {
          this.tasks = this.tasks.filter((task) => task.id !== id);
  }
}
```

- Update the `TaskListComponent` to display tasks from the service:

```
tasks: any[] = [];

ngOnInit() {
  this.tasks = this.taskService.getTasks();
}
```

Add Local Storage Persistence

- Enhance `TaskService` to use local storage:

```
getTasks() {
  const storedTasks = localStorage.getItem('tasks');
  this.tasks = storedTasks ? JSON.parse(storedTasks) : [];
  return this.tasks;
}

addTask(task: any) {
  this.tasks.push(task);
  this.saveToLocalStorage();
}

private saveToLocalStorage() {
  localStorage.setItem('tasks', JSON.stringify(this.tasks));
}
```

Introduction to NgRx (Optional for Advanced Users)

- Install NgRx dependencies:
  ```
  npm install @ngrx/store @ngrx/effects
  ```
- Create an `actions` file:
  ```
  import { createAction, props } from '@ngrx/store';

  export const addTask = createAction('[Task] Add Task', props<{ task: any }>());
  export const removeTask = createAction('[Task] Remove Task', props<{ id: number
  }>());
  ```
- Create a `reducer` file:
  ```
  import { createReducer, on } from '@ngrx/store';
  import { addTask, removeTask } from './task.actions';

  export const initialState: any[] = [];

  const _taskReducer = createReducer(
    initialState,
    on(addTask, (state, { task }) => [...state, task]),
    on(removeTask, (state, { id }) => state.filter((task) => task.id !== id))
  );

  export function taskReducer(state: any, action: any) {
    return _taskReducer(state, action);
  }
  ```
- Configure the `StoreModule` in `AppModule`:
  ```
  import { StoreModule } from '@ngrx/store';
  import { taskReducer } from './task.reducer';

  @NgModule({
    imports: [StoreModule.forRoot({ tasks: taskReducer })],
  })
  export class AppModule {}
  ```
- Dispatch actions in components:
  ```
  this.store.dispatch(addTask({ task: newTask }));
  ```

Expected Outputs

1. Tasks managed centrally using a service with state accessible across components.
2. State persisted in local storage, retaining data even after a page refresh.
3. (Optional) NgRx integration with actions and reducers for scalable state management.

Integration to Project

- **Task Manager Application**:
  - Centralized task management using `TaskService`.
  - Tasks persisted in local storage for session persistence.
  - Optionally, introduce NgRx for scalable and advanced state management.

Testing Angular Applications

Learning Objectives:

- Learn the fundamentals of testing in Angular using Jasmine and Karma.
- Write and execute unit tests for components and services.
- Perform end-to-end testing with Protractor or Cypress (optional).
- Understand best practices for Angular testing.

Topics Covered

Introduction to Testing in Angular

- Importance of testing in software development.
- Types of testing:
    - Unit Testing: Testing individual components and services.
    - Integration Testing: Testing interactions between units.
    - End-to-End (E2E) Testing: Testing the application flow.

Angular Testing Tools

- **Jasmine**: A behavior-driven testing framework.
- **Karma**: A test runner for executing tests in various browsers.
- Protractor (optional): An E2E testing tool for Angular applications.

Setting Up Testing in Angular

- Angular CLI automatically configures Jasmine and Karma.
- Running tests:
  ng test

Unit Testing Components

- Writing tests for component methods and templates.
- Using the `TestBed` utility for creating testing modules.

Unit Testing Services

- Mocking dependencies and HTTP requests using `HttpTestingController`.
- Verifying service behavior with `spyOn`.

End-to-End Testing

- Setting up Protractor for testing Angular apps (optional: Cypress for advanced users).
- Writing E2E tests to validate user flows.

Exercises

Write Unit Tests for a Component

- Write tests for `TaskListComponent` to verify its functionality:

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { TaskListComponent } from './task-list.component';
import { TaskService } from '../task.service';

describe('TaskListComponent', () => {
  let component: TaskListComponent;
  let fixture: ComponentFixture<TaskListComponent>;
  let mockTaskService: jasmine.SpyObj<TaskService>;

  beforeEach(() => {
          mockTaskService = jasmine.createSpyObj('TaskService', ['getTasks']);
          mockTaskService.getTasks.and.returnValue([{ id: 1, title: 'Test Task',
completed: false }]);

          TestBed.configureTestingModule({
          declarations: [TaskListComponent],
          providers: [{ provide: TaskService, useValue: mockTaskService }],
          });

          fixture = TestBed.createComponent(TaskListComponent);
          component = fixture.componentInstance;
  });

  it('should create the component', () => {
          expect(component).toBeTruthy();
  });

  it('should fetch tasks on initialization', () => {
          component.ngOnInit();
          expect(component.tasks.length).toBe(1);
          expect(component.tasks[0].title).toBe('Test Task');
  });
});
```

Unit Test a Service with HTTP Requests

- Test the `TaskService` for fetching tasks using `HttpClientTestingModule`:

```
import { TestBed } from '@angular/core/testing';
import { HttpClientTestingModule, HttpTestingController } from
'@angular/common/http/testing';
import { TaskService } from './task.service';

describe('TaskService', () => {
  let service: TaskService;
  let httpMock: HttpTestingController;

  beforeEach(() => {
        TestBed.configureTestingModule({
        imports: [HttpClientTestingModule],
        providers: [TaskService],
        });

        service = TestBed.inject(TaskService);
        httpMock = TestBed.inject(HttpTestingController);
  });

  it('should fetch tasks from API', () => {
        const mockTasks = [{ id: 1, title: 'Test Task', completed: false }];

        service.getTasks().subscribe((tasks) => {
        expect(tasks.length).toBe(1);
        expect(tasks[0].title).toBe('Test Task');
        });

        const req =
httpMock.expectOne('https://jsonplaceholder.typicode.com/todos');
        expect(req.request.method).toBe('GET');
        req.flush(mockTasks);
  });
});
```

Write End-to-End (E2E) Tests

- Test the task creation flow:

```
import { browser, by, element } from 'protractor';

describe('Task Manager App', () => {
  it('should add a new task', () => {
        browser.get('/tasks');
        element(by.css('input[name="title"]')).sendKeys('New Task');
        element(by.css('button[type="submit"]')).click();

        const taskList = element.all(by.css('ul li'));
        expect(taskList.count()).toBeGreaterThan(0);
  });
});
```

Add a Test for Form Validation

- Verify that the task creation form displays validation errors:

```
it('should display validation error if title is missing', () => {
  const titleInput = fixture.nativeElement.querySelector('input[name="title"]');
  titleInput.value = '';
  titleInput.dispatchEvent(new Event('input'));
  fixture.detectChanges();

  const errorMessage = fixture.nativeElement.querySelector('.error-message');
  expect(errorMessage).toBeTruthy();
});
```

Expected Outputs

1. Unit tests ensure the functionality of components and services.
2. HTTP requests in services are tested and verified for expected behavior.
3. End-to-end tests validate the complete user flow, ensuring tasks can be created and managed.
4. Form validation tests confirm that user input constraints are enforced.

Integration to Project

- **Task Manager Application:**
    - **Fully tested components, services, and forms.**
    - **Reliable and verified data-fetching logic.**
    - **End-to-end tests to validate core functionalities.**

Optimizing Angular Applications and Capstone Project

Learning Objectives

- Learn optimization techniques for Angular applications to enhance performance and scalability.
- Use Angular CLI tools like AOT (Ahead of Time) compilation and lazy loading.
- Complete the capstone project by integrating all the concepts covered during the course.
- Deploy the application to a live server.

Topics Covered

Optimizing Angular Applications

- **AOT Compilation:**
  - **Reduces the size of the application and speeds up its startup time.**
  - **Enable AOT during the build process:**
    ng build --prod
- **Lazy Loading Modules:**
  - **Load feature modules only when needed to improve performance.**
  - **Example: Configure lazy loading for the `TasksModule`:**
    const routes: Routes = [
    { path: 'tasks', loadChildren: () => import('./tasks/tasks.module').then(m => m.TasksModule) }
    ];
- **Tree Shaking:**
  - **Removes unused code to reduce the final bundle size.**
  - **Automatically enabled during `ng build --prod`.**
- **Minification and Compression:**
  - **Enable minification and gzip compression to reduce file sizes.**
  - **Use tools like Nginx or Webpack for server-side compression.**

Preparing the Application for Deployment

- Building the app for production:
  ng build --prod
- Configuring the output:
  - Generated files in the `dist/` folder.
- Hosting options:
  - Deploy to a local server (e.g., using `http-server`):
    npm install -g http-server
    http-server dist/my-app
- Deploy to platforms like Firebase, Netlify, or AWS.

Capstone Project

- Build the **Task Manager Application** by integrating all concepts:
  - Components, forms, routing, services, state management, and optimization.
- Core Features:
  - **Task List:** Display tasks using `*ngFor`, filtering dynamically.
  - **Task Creation Form:** Add new tasks with validation.
  - **Task Details:** Use routing to show task-specific details.
  - **Search and Filter:** Implement dynamic search with Observables and RxJS.
  - **Persistent State:** Use services and local storage to save and retrieve tasks.
  - **API Integration:** Fetch and manage tasks using an external API.
  - **Error Handling:** Display user-friendly error messages for failed requests.
  - **Optimized App:** Use lazy loading, AOT, and minification.

Deployment

- Deploy the final application on a live platform for users to access.

Exercises

Optimize Application Modules with Lazy Loading

- Move task-related components into a `TasksModule`:
  ng generate module tasks --route tasks --module app.module
- Verify that the tasks are loaded only when navigating to `/tasks`.

Build and Deploy the Application

- Build the app with AOT enabled:
  ng build --prod
- Deploy the app locally using `http-server`:
  npx http-server dist/task-manager
- Deploy to Firebase (example):
  npm install -g firebase-tools
  firebase init
  firebase deploy

Complete the Capstone Project

- Integrate all features developed during the course into a cohesive application.
- Test the app thoroughly to ensure all functionalities work seamlessly.

Expected Outputs

1. A production-ready Angular application optimized for performance.
2. Lazy-loaded modules reducing initial load time.
3. Fully functional **Task Manager Application**:
   a. Task creation, viewing, editing, filtering, and deletion.

    b. Persistent state and API integration.
4. Deployed application accessible via a live URL.

Conclusion

Summary of the Course

- **Core Angular Concepts:**
    - **Component-based architecture, directives, forms, and routing.**
- **Application State Management:**
    - **Using services, Observables, and (optionally) NgRx.**
- **Backend Integration:**
    - **Consuming REST APIs with `HttpClient` and error handling.**
- **Performance Optimization:**
    - **AOT, lazy loading, and tree shaking for a faster and smaller app.**
- **Testing:**
    - **Unit testing for components and services, and E2E testing for user flows.**
- **Deployment:**
    - **Steps to deploy Angular apps on live platforms.**

Key Takeaways

- Participants can build scalable, maintainable, and high-performance Angular applications.
- The capstone project showcases their ability to integrate Angular concepts into a real-world app.
- Knowledge of testing and optimization ensures they can deliver reliable and efficient software.

Resources for Further Learning

1. Angular Official Documentation
2. RxJS Documentation
3. NgRx Documentation
4. Angular CLI Reference
5. Books:
    - *Pro Angular 9* by Adam Freeman
    - *Learning Angular* by Aristeidis Bampakos and Pablo Deeleman