ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION

ADVANCED ANGULAR CONCEPTS & PERFORMANCE OPTIMIZATION (WEEKS 7-8)

PARENT-CHILD COMPONENT INTERACTION (INPUT & OUTPUT DECORATORS) IN ANGULAR

CHAPTER 1: INTRODUCTION TO PARENT-CHILD COMPONENT COMMUNICATION

1.1 Understanding Parent-Child Component Interaction

In Angular, **Parent-Child Component Communication** allows components to **share data and events** between a **parent component** and a **child component**. This is essential for **building modular and scalable applications**.

✓ **Parent-to-Child Communication** → Uses @Input() to pass data **from parent to child**.
✓ **Child-to-Parent Communication** → Uses @Output() and EventEmitter to send data **from child to parent**.

- **Example Use Cases:**
  - Passing **user details** from the parent to the child component.

- Sending **form submission events** from the child to the parent.
- **Dynamic updates** where a child component modifies data in the parent.

CHAPTER 2: PASSING DATA FROM PARENT TO CHILD USING @INPUT()

2.1 What is @Input()?

The @Input() decorator allows a **parent component** to pass data **into a child component** via **property binding** ([ ]).

✓ **Useful for dynamic components** that display **data passed from the parent**.
✓ **Prevents redundant API calls** by sending data **directly to the child**.

2.2 Implementing @Input() in Angular

Step 1: Generate a Child Component

ng generate component child

Step 2: Define @Input() Property in Child Component (child.component.ts)

Modify child.component.ts to receive data from the parent:

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


@Component({
  selector: 'app-child',
```

```
  templateUrl: './child.component.html',
  styleUrls: ['./child.component.css']
})
export class ChildComponent {
  @Input() userName: string = '';  // Receives data from parent
}
```

✓ @Input() makes userName **accessible inside the child component**.

Step 3: Display Parent Data in child.component.html

```
<p>Welcome, {{ userName }}!</p>
```

✓ This displays the userName passed from the **parent component**.

Step 4: Pass Data from Parent to Child (app.component.html)

Modify app.component.html:

```
<app-child [userName]="'Alice'"></app-child>
```

✓ The [userName] binds **static data ('Alice')** to the child component.

- **Example of Passing Dynamic Data from Parent**

Modify app.component.ts:

```
export class AppComponent {
  parentUserName = 'Bob';
}
```

Modify app.component.html:

<app-child [userName]="parentUserName"></app-child>

✓ The value updates dynamically **if parentUserName changes**.

2.3 Passing Objects Using @Input()

Modify child.component.ts to accept an **object**:

@Input() user: { name: string, age: number } = { name: '', age: 0 };

Modify child.component.html to display object properties:

Name: {{ user.name }}

Age: {{ user.age }}

Modify app.component.ts:

export class AppComponent {

userInfo = { name: 'Charlie', age: 25 };

}

Modify app.component.html:

<app-child [user]="userInfo"></app-child>

✓ **Passes complex data structures** instead of single variables.

CHAPTER 3: SENDING DATA FROM CHILD TO PARENT USING @OUTPUT()

3.1 What is @Output()?

The @Output() decorator **emits events from the child** to the **parent component,** enabling **reverse communication**.

✓ Uses EventEmitter to **send custom events** from child to parent.
✓ Allows parent components to **listen and react** to child component events.

3.2 Implementing @Output() in Angular

Step 1: Modify Child Component to Emit Data (child.component.ts)

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


@Component({
 selector: 'app-child',
 templateUrl: './child.component.html',
 styleUrls: ['./child.component.css']
})
export class ChildComponent {
 @Output() messageEvent = new EventEmitter<string>();


 sendMessage() {
  this.messageEvent.emit('Hello from Child Component!');
 }
}
```

✓ The sendMessage() function **emits an event** with a message.

Step 2: Trigger Event from Child Component (child.component.html)

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

✓ Clicking the button **triggers sendMessage()** and notifies the parent.

Step 3: Handle the Event in Parent Component (app.component.html)

Modify app.component.html to listen for the event:

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

<p>Message from Child: {{ childMessage }}</p>
```

✓ (messageEvent) listens for the **event emitted by the child**.

Step 4: Process the Event in Parent Component (app.component.ts)

```
export class AppComponent {
 childMessage: string = '';

 receiveMessage(message: string) {
  this.childMessage = message;
 }
}
```

✓ The receiveMessage() function **stores the child's message** in childMessage.

3.3 Passing Objects Using @Output()

Modify child.component.ts to emit an **object**:

```
@Output() userUpdated = new EventEmitter<{ name: string, age: number }>();


updateUser() {
 this.userUpdated.emit({ name: 'Dave', age: 30 });
}
```

Modify child.component.html:

```
<button (click)="updateUser()">Update User</button>
```

Modify app.component.ts to receive the object:

```
export class AppComponent {
 userInfo: { name: string, age: number } = { name: '', age: 0 };


 updateUserInfo(updatedUser: { name: string, age: number }) {
  this.userInfo = updatedUser;
 }
}
```

Modify app.component.html:

```
<app-child (userUpdated)="updateUserInfo($event)"></app-child>

<p>Updated User: {{ userInfo.name }} ({{ userInfo.age }} years old)</p>
```

✓ Passes **complex objects** instead of just strings.

Case Study: How a Dashboard Uses Parent-Child Communication

Background

A **company dashboard** needed:
✓ **User profile updates** from child to parent.
✓ **Live data updates** between components.
✓ **Efficient state management** with services.

Challenges

- **Redundant API calls** every time user data was updated.
- **Data inconsistency** between the sidebar and main dashboard.
- **Complex event handling** across multiple components.

Solution: Using @Input() and @Output() for Efficient Data Flow

✓ **Passed user data from the parent to multiple components using @Input()**.
✓ **Used @Output() to notify parent when user data changed**.
✓ **Reduced API calls by 60%** by **caching data in services**.

<app-user-profile (userUpdated)="refreshUserData($event)"></app-user-profile>

<app-dashboard [user]="userData"></app-dashboard>

Results

🚀 **Faster updates** without excessive API requests.
🔍 **Consistent data state** across components.
⚡ **Scalable communication** for future component additions.

Exercise

1. Create a **parent component** (app.component) and a **child component** (user-card).
2. Use @Input() to pass a **user object** from parent to child.
3. Use @Output() to **notify the parent** when the user clicks "Update Profile".
4. Display the **updated user details** in the parent component.

Conclusion

✓ **Used @Input() to pass data from parent to child**.

✓ **Used @Output() and EventEmitter for child-to-parent communication**.

✓ **Built a dynamic, interactive Angular application**.

USING VIEWCHILD AND CONTENTCHILD IN ANGULAR

CHAPTER 1: INTRODUCTION TO @VIEWCHILD AND @CONTENTCHILD

1.1 What are @ViewChild and @ContentChild in Angular?

Angular provides decorators **@ViewChild** and **@ContentChild** to access and manipulate **child elements or components**.

✓ @ViewChild – Accesses **direct child elements** inside a component's own template.
✓ @ContentChild – Accesses **projected content** inside a component (via <ng-content>).

These decorators allow:
✓ **DOM element and component manipulation**.
✓ **Efficient inter-component communication**.
✓ **Accessing child components' properties and methods**.

CHAPTER 2: UNDERSTANDING @VIEWCHILD

2.1 What is @ViewChild?

@ViewChild is used to **access elements or components** inside the component's own template (.html file).

✓ **Works with elements, components, and directives**.
✓ **Available after ngAfterViewInit lifecycle hook**.
✓ **Useful for DOM manipulation and accessing child component methods**.

2.2 Example: Accessing a DOM Element with @ViewChild

Step 1: Accessing an Element in the Template

```
import { Component, ViewChild, ElementRef, AfterViewInit } from '@angular/core';


@Component({
 selector: 'app-example',
 template: `<p #message>Hello, ViewChild!</p>
        <button (click)="changeText()">Change Text</button>`
})
export class ExampleComponent implements AfterViewInit {
 @ViewChild('message') messageElement!: ElementRef;


 ngAfterViewInit() {
  console.log("Text:", this.messageElement.nativeElement.innerText);
 }


 changeText() {
  this.messageElement.nativeElement.innerText = "Text Updated!";
 }
}
```

✓ @ViewChild('message') selects the <p> element.

✓ messageElement.nativeElement.innerText **modifies text dynamically**.

2.3 Example: Accessing a Child Component with @ViewChild

Step 1: Create a Child Component

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


@Component({
 selector: 'app-child',
 template: `<p>Child Component</p>`
})
export class ChildComponent {
 greet() {
  return "Hello from Child Component!";
 }
}
```

Step 2: Access the Child Component in the Parent

```
import { Component, ViewChild, AfterViewInit } from
'@angular/core';
import { ChildComponent } from './child.component';


@Component({
```

```
 selector: 'app-parent',
 template: `<app-child></app-child>
        <button (click)="callChildMethod()">Call Child
Method</button>`
})
export class ParentComponent implements AfterViewInit {
 @ViewChild(ChildComponent) childComponent!:
ChildComponent;

 ngAfterViewInit() {
  console.log(this.childComponent.greet());
 }

 callChildMethod() {
  alert(this.childComponent.greet());
 }
}
```

✓ **Calls the child component's method** from the parent.

CHAPTER 3: UNDERSTANDING @CONTENTCHILD

3.1 What is @ContentChild?

@ContentChild is used to **access projected content** passed via <ng-content> in the parent template.

✓ Works when a **child component receives external content**.
✓ Available after ngAfterContentInit lifecycle hook.
✓ Helps **modify or interact with projected content dynamically**.

3.2 Example: Accessing Projected Content with @ContentChild

Step 1: Create a Parent Component with <ng-content>

```
import { Component, ContentChild, ElementRef, AfterContentInit } from '@angular/core';


@Component({
 selector: 'app-card',
 template: `<div class="card">
          <h2>Card Title</h2>
          <ng-content></ng-content>
        </div>`
})
export class CardComponent implements AfterContentInit {
 @ContentChild('cardContent') cardContent!: ElementRef;


 ngAfterContentInit() {
  console.log("Projected Content:", this.cardContent.nativeElement.innerText);
 }
```

```
}
```

✓ ng-content allows **external content projection**.
✓ @ContentChild('cardContent') selects the **projected content**.

Step 2: Use the Card Component in a Parent Template

```
<app-card>
  <p #cardContent>This is projected content!</p>
</app-card>
```

✓ The @ContentChild reference accesses <p #cardContent> inside the <app-card>.

3.3 Example: Accessing a Child Component in Content Projection

Step 1: Create a Child Component

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

@Component({
  selector: 'app-message',
  template: `<p>Message Component</p>`
})
export class MessageComponent {
  getMessage() {
    return "Hello from Message Component!";
  }
}
```

Step 2: Access the Child Component in Another Component Using @ContentChild

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

import { MessageComponent } from './message.component';


@Component({

 selector: 'app-container',

 template: `<div class="container">

          <ng-content></ng-content>

        </div>`

})

export class ContainerComponent implements AfterContentInit {

 @ContentChild(MessageComponent) messageComp!: MessageComponent;


 ngAfterContentInit() {

  console.log(this.messageComp.getMessage());

 }

}
```

✓ Calls getMessage() **from the projected MessageComponent** inside <ng-content>.

Step 3: Use the Components in the Parent Template

```
<app-container>
```

<app-message></app-message>

</app-container>

✓ The parent accesses app-message **inside ng-content**.

CHAPTER 4: DIFFERENCES BETWEEN @VIEWCHILD AND @CONTENTCHILD

Feature	@ViewChild	@ContentChild
Works Inside	Component's **own template**.	**Projected content** using <ng-content>.
Accessible After	ngAfterViewInit.	ngAfterContentInit.
Use Case	Accessing **elements or child components** directly inside the component.	Accessing **content passed from a parent** via <ng-content>.
Scope	Only works inside the component's own **view (template)**.	Works with **external content injected** into <ng-content>.

Case Study: How a Dashboard Application Used @ViewChild and @ContentChild for Dynamic UI Components

Background

A **dashboard application** needed:

✓ **Dynamically loaded components** (charts, tables).
✓ **Custom card components** that accept projected content.
✓ **Efficient communication between parent-child components**.

Challenges

- **Complex UI interactions** required **accessing child elements dynamically**.
- **Projected content needed customization** while keeping components reusable.
- **Performance issues** due to direct DOM manipulations.

Solution: Implementing @ViewChild and @ContentChild

✓ Used @ViewChild to access **chart instances** dynamically.
✓ Used @ContentChild to manage **custom card components**.
✓ Optimized rendering using **lifecycle hooks (ngAfterViewInit, ngAfterContentInit)**.

Results

🚀 **50% faster component rendering** using Angular DI.
🔍 **Reusable and flexible UI components,** reducing development effort.
⚡ **Smooth UI interactions,** improving user experience.

By using **@ViewChild and @ContentChild,** the dashboard became **highly dynamic and maintainable**.

Exercise

1. Use @ViewChild to modify a **button's text** inside a component.
2. Create a **child component** and access its method from a parent using @ViewChild.
3. Create a **wrapper component with <ng-content>** and access the projected content using @ContentChild.

4. Use @ContentChild to interact with a **child component passed via <ng-content>**.

Conclusion

In this section, we explored:

✓ **How @ViewChild accesses elements and child components dynamically**.

✓ **How @ContentChild interacts with projected content inside <ng-content>**.

✓ **How to use lifecycle hooks (ngAfterViewInit, ngAfterContentInit) for better performance**.

EVENT EMITTERS AND SUBJECT OBSERVABLES IN ANGULAR

CHAPTER 1: UNDERSTANDING EVENT EMITTERS AND OBSERVABLES

1.1 What Are Event Emitters and Observables in Angular?

Angular applications require communication between components, especially **parent-child** and **sibling components**. Two common ways to achieve this are:

✓ **Event Emitters (EventEmitter)** – Used for **child-to-parent communication** via component outputs.
✓ **Subject Observables (Subject)** – Used for **cross-component communication,** especially **sibling components** or global event handling.

These techniques help **keep applications modular, maintainable, and scalable** by ensuring **efficient data flow**.

CHAPTER 2: USING EVENT EMITTERS FOR PARENT-CHILD COMMUNICATION

2.1 What Are Event Emitters?

An EventEmitter is a **special Angular class** that allows child components to **emit events** to their parent components.

✓ **Used inside child components** to send data to the parent.
✓ **Requires @Output() decorator** to make the event accessible in the parent.
✓ **Triggers actions in response to user events**.

2.2 Implementing Event Emitters

Step 1: Create a Child Component

Run the following command to generate a child component:

ng generate component child

Step 2: Define the Event in child.component.ts

Modify child.component.ts to create an **EventEmitter**:

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


@Component({
 selector: 'app-child',
 templateUrl: './child.component.html',
 styleUrls: ['./child.component.css']
})
export class ChildComponent {
 @Output() notifyParent: EventEmitter<string> = new
EventEmitter<string>();


 sendMessage() {
  this.notifyParent.emit('Hello from Child Component!');
 }
}
```

✓ @Output() marks notifyParent as an event that can be **listened to in the parent**.
✓ The sendMessage() method **emits a message to the parent** when called.

Step 3: Handle the Event in parent.component.ts

Modify parent.component.ts to listen for the event:

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


@Component({
 selector: 'app-parent',
 templateUrl: './parent.component.html',
 styleUrls: ['./parent.component.css']
})
export class ParentComponent {
 receivedMessage: string = '';


 handleChildEvent(message: string) {
  this.receivedMessage = message;
 }
}
```

✓ The handleChildEvent() method **receives data from the child**.

Step 4: Bind the Event in parent.component.html

Modify parent.component.html to include the child component and bind the event:

```
<h2>Parent Component</h2>

<p>Message from Child: {{ receivedMessage }}</p>


<app-child (notifyParent)="handleChildEvent($event)"></app-child>
```

✓ The (notifyParent) binding **listens for events from the child**.
✓ $event contains the emitted **message from the child component**.

2.3 Summary of Event Emitters

Feature	Event Emitters (EventEmitter)
Used For	Child-to-parent communication
Decorator	@Output()
How It Works	Child emits an event → Parent listens and responds
Best For	Passing small amounts of data like **user input or button clicks**

CHAPTER 3: USING SUBJECT OBSERVABLES FOR CROSS-COMPONENT COMMUNICATION

3.1 What Are Subject Observables?

A **Subject** is a **special type of Observable** that allows:

✓ **Multicast event streams** – Multiple components can listen to the same event.
✓ **Component-to-component communication** – Works between **siblings, unrelated components, or global event handling**.
✓ **Emits new values dynamically** – Unlike traditional Observables, Subject **can push new data dynamically**.

3.2 Implementing Subject Observables for Component Communication

Step 1: Create a Shared Service

Run the following command to generate a service:

```
ng generate service event-bus
```

Modify event-bus.service.ts to include a Subject:

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

@Injectable({
  providedIn: 'root'
})
export class EventBusService {
  private messageSubject = new Subject<string>();

  message$ = this.messageSubject.asObservable();
```

```
  sendMessage(message: string) {
    this.messageSubject.next(message);
  }
}
```

✓ **messageSubject** is a Subject that emits messages.
✓ **messages** is an Observable that components can subscribe to.
✓ **sendMessage()** sends data to all subscribed components.

Step 2: Publish Events in sender.component.ts

Modify sender.component.ts to send data:

```
import { Component } from '@angular/core';
import { EventBusService } from '../event-bus.service';

@Component({
  selector: 'app-sender',
  templateUrl: './sender.component.html',
  styleUrls: ['./sender.component.css']
})
export class SenderComponent {
  constructor(private eventBus: EventBusService) {}

  sendMessage() {
    this.eventBus.sendMessage('Message from Sender Component');
```

```
  }
}
```

✓ Calls sendMessage() in EventBusService to **emit data to all listeners**.

Step 3: Subscribe to the Event in receiver.component.ts

Modify receiver.component.ts to listen for messages:

```
import { Component, OnInit } from '@angular/core';
import { EventBusService } from '../event-bus.service';

@Component({
  selector: 'app-receiver',
  templateUrl: './receiver.component.html',
  styleUrls: ['./receiver.component.css']
})
export class ReceiverComponent implements OnInit {
  receivedMessage: string = '';

  constructor(private eventBus: EventBusService) {}

  ngOnInit() {
    this.eventBus.message$.subscribe(message => {
      this.receivedMessage = message;
```

```
  });
 }
}
```

✓ **Subscribes to message$** and updates receivedMessage dynamically.

Step 4: Display Data in receiver.component.html

Modify receiver.component.html to display received messages:

```
<h2>Receiver Component</h2>
<p>Message: {{ receivedMessage }}</p>
```

✓ Displays messages sent from the SenderComponent.

3.3 Summary of Subject Observables

Feature	Subject Observables (Subject)
Used For	Sibling-to-sibling or global communication
How It Works	One component publishes data → Multiple components subscribe and react
Best For	**Cross-component event sharing** (e.g., notifications, chat messages)

Case Study: How a Chat Application Used Event Emitters & Subjects

Background

A **chat application** needed a **real-time message system** where:

✓ Users could **send messages instantly**.
✓ **Different components** handled **message input and display separately**.
✓ **Multiple chat rooms** needed **separate communication channels**.

Challenges

- **Event Emitters only worked for parent-child communication.**
- **Messages needed to be broadcasted to multiple components.**
- **Users required real-time updates without page refresh.**

Solution: Implementing Subjects for Real-Time Messaging

✓ **Used Subject in an event bus service** for real-time updates.
✓ **Messages were dynamically pushed** to all subscribed components.
✓ **Provided seamless chat experience** without reloading pages.

Results

🚀 **Instant message delivery,** improving user experience.
⚡ **Reduced API calls,** as messages were handled within the frontend.
📈 **Scalable system,** allowing multiple users in different chat rooms.

By combining **Event Emitters and Subject Observables,** the chat app achieved **real-time communication** effectively.

Exercise

1. Implement **Event Emitters** to send data from a child to a parent component.
2. Create an **Event Bus Service** using Subject for cross-component communication.
3. Use Subject.next() to emit data dynamically and update multiple components.
4. Display received messages in a subscriber component.

Conclusion

In this section, we explored:

✓ **How Event Emitters enable child-to-parent communication**.

✓ **How Subject Observables facilitate cross-component event sharing**.

✓ **When to use Event Emitters vs. Subject Observables**.

Optimizing Angular Applications for Speed

Chapter 1: Introduction to Angular Performance Optimization

1.1 Why Optimize Angular Applications?

Performance optimization ensures **faster load times, smooth user experience, and reduced resource usage** in Angular applications. Key reasons to optimize include:

✓ **Faster page loads** – Reduces the initial bundle size.
✓ **Better user experience** – Ensures smooth interactions.
✓ **Lower server load** – Reduces API calls and memory usage.
✓ **SEO improvements** – Faster websites rank higher on search engines.

1.2 Key Areas of Performance Optimization

To improve Angular app performance, focus on:

1. **Reducing initial load time** – Optimize bundle size and lazy load resources.
2. **Efficient change detection** – Reduce unnecessary DOM updates.
3. **Optimizing API calls** – Implement caching and request optimizations.
4. **Enhancing rendering speed** – Minimize reflows and repaints.
5. **Memory management** – Prevent memory leaks and reduce resource consumption.

CHAPTER 2: REDUCING BUNDLE SIZE WITH ANGULAR OPTIMIZATION TECHNIQUES

2.1 Using Angular Production Build

A production build optimizes code by **removing unnecessary files, minifying assets, and tree shaking unused code**.

Command to Generate a Production Build

ng build --configuration=production

✓ **Reduces bundle size** by minifying and optimizing assets.
✓ **Removes development-only features** for faster execution.

2.2 Lazy Loading Modules

Lazy loading **loads modules only when needed,** reducing initial load time.

Modify app-routing.module.ts to Enable Lazy Loading

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

✓ **Loads DashboardModule only when the user visits /dashboard**.
✓ **Improves application startup time** by loading critical components first.

2.3 Removing Unused Code with Tree Shaking

Tree shaking **eliminates unused JavaScript** during build time. It's automatically enabled in **Angular production builds**.

How to Check Bundle Size After Tree Shaking

ng build --stats-json

Then analyze the bundle using:

npx source-map-explorer dist/my-app/main.js

✓ Identifies **which modules contribute to bundle size**.
✓ Helps **remove unnecessary dependencies**.

CHAPTER 3: OPTIMIZING CHANGE DETECTION FOR FASTER RENDERING

3.1 Understanding Change Detection in Angular

Angular **automatically updates the UI** when data changes. However, excessive checks **slow down performance**.

✓ **By default, Angular checks the entire component tree** when a change occurs.
✓ **Unoptimized change detection can cause unnecessary re-renders.**

3.2 Using OnPush Change Detection Strategy

The OnPush strategy updates the UI **only when input properties change**, reducing unnecessary re-renders.

Example: Implementing OnPush Strategy

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

```
@Component({
 selector: 'app-user',
 template: `<p>{{ user.name }}</p>`,
 changeDetection: ChangeDetectionStrategy.OnPush
})
export class UserComponent {
 @Input() user: any;
}
```

✓ **Angular skips change detection** unless @Input properties change.
✓ **Speeds up UI updates** by reducing re-render cycles.

3.3 Detaching and Reattaching Change Detection

For **highly dynamic applications,** use ChangeDetectorRef to **manually control** when updates occur.

Example: Detaching Change Detection

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

@Component({
 selector: 'app-performance',
 template: `<p>Performance Optimized Component</p>`
})
```

```
export class PerformanceComponent {
 constructor(private cdr: ChangeDetectorRef) {}

 pauseUpdates() {
  this.cdr.detach(); // Stops change detection
 }

 resumeUpdates() {
  this.cdr.reattach(); // Resumes change detection
 }
}
```

✓ **Useful for performance-heavy components** (e.g., real-time dashboards).

CHAPTER 4: OPTIMIZING API CALLS AND NETWORK REQUESTS

4.1 Using RxJS Operators to Reduce API Calls

✓ Prevents **unnecessary API requests** with operators like debounceTime() and switchMap().

Example: Optimizing Search API Calls

```
import { Component } from '@angular/core';
import { debounceTime, switchMap } from 'rxjs/operators';
import { FormControl } from '@angular/forms';
import { ApiService } from './api.service';
```

```
@Component({
 selector: 'app-search',
 template: `<input [formControl]="searchControl" placeholder="Search">`
})
export class SearchComponent {
 searchControl = new FormControl();

 constructor(private apiService: ApiService) {
  this.searchControl.valueChanges.pipe(
   debounceTime(300),
   switchMap(query => this.apiService.search(query))
  ).subscribe(results => console.log(results));
 }
}
```

✓ **Waits 300ms** before sending the request, preventing multiple API calls.
✓ **Cancels previous requests** when a new search is entered.

4.2 Caching API Responses

Caching reduces API calls by **storing results** in memory.

Example: Implementing API Caching

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

@Injectable({
  providedIn: 'root'
})
export class ApiService {
  private cache = new Map();

  constructor(private http: HttpClient) {}

  getData(url: string) {
    if (this.cache.has(url)) {
      return of(this.cache.get(url)); // Return cached data
    }
    return this.http.get(url).pipe(
      tap(data => this.cache.set(url, data)) // Store response in cache
    );
  }
}
```

✓ Prevents redundant API calls, **improving response time**.

Chapter 5: Optimizing Rendering and Load Performance

5.1 Using Virtual Scrolling for Large Lists

Virtual scrolling **renders only visible items,** improving performance.

Example: Implementing Virtual Scrolling

Install Angular CDK:

```
npm install @angular/cdk
```

Modify component template:

```
<cdk-virtual-scroll-viewport itemSize="50" class="list">
  <div *cdkVirtualFor="let item of items">{{ item }}</div>
</cdk-virtual-scroll-viewport>
```

✓ **Reduces DOM updates,** improving performance for large lists.

5.2 Lazy Loading Images with loading="lazy"

Lazy loading **delays image loading** until they appear on screen.

Example: Implementing Lazy Loading in HTML

```
<img src="large-image.jpg" loading="lazy" alt="Lazy Loaded Image">
```

✓ Reduces initial page load time.

Case Study: How an E-Commerce Platform Improved Speed Using Angular Optimization Techniques

Background

An **e-commerce platform** faced:

✓ **Slow page loads** due to large product images.
✓ **Excessive API calls** slowing performance.
✓ **Laggy user interactions** with large product lists.

Challenges

- Users **waited 5-8 seconds** for product pages to load.
- Multiple redundant **API calls for the same data**.
- Scrolling through products **caused slow rendering**.

Solution: Applying Angular Performance Optimization

The team implemented:

✓ **Lazy loading images** to reduce initial page load.
✓ **API caching** to minimize redundant network requests.
✓ **Virtual scrolling** for smooth product list rendering.

Results

🚀 **Page load time reduced from 8s to 2s**.
⚡ **API request volume decreased by 50%,** improving server response time.
🔄 **Smooth user experience,** increasing conversions.

By **optimizing performance,** the platform improved **speed, efficiency, and user engagement**.

Exercise

1. **Enable Lazy Loading** for Angular modules.

2. **Implement OnPush Change Detection** in a component.
3. **Use Virtual Scrolling** for a list with 1000+ items.
4. **Optimize API calls** using caching and debounceTime().

Conclusion

In this section, we explored:

✓ **How to reduce Angular bundle size using lazy loading and tree shaking**.

✓ **How to optimize change detection for faster rendering**.

✓ **How to improve API efficiency with RxJS operators and caching**.

✓ **How to use virtual scrolling and lazy loading for better UI performance**.

LAZY LOADING AND CODE SPLITTING IN ANGULAR

CHAPTER 1: INTRODUCTION TO LAZY LOADING AND CODE SPLITTING

1.1 What is Lazy Loading?

Lazy loading is an **optimization technique** in Angular that **loads feature modules only when needed,** instead of loading them at the application's startup.

✓ **Improves initial load time** – Loads only essential components first.
✓ **Enhances performance** – Reduces the size of the initial bundle.
✓ **Optimizes network usage** – Fetches additional modules dynamically.

- **Example Use Case:**
  - Loading the **dashboard module** only when the user logs in.
  - Deferring **admin panel components** until accessed.

1.2 What is Code Splitting?

Code splitting is the process of **breaking large JavaScript bundles** into smaller chunks.

✓ Reduces **page load time**.
✓ Fetches **only the required JavaScript files**.
✓ Improves **mobile performance** by reducing initial load size.

- **How it Works:**

- **Eager Loading:** Loads everything at once (default in Angular).
- **Lazy Loading:** Loads modules **on demand** using Angular's Router.

CHAPTER 2: SETTING UP LAZY LOADING IN ANGULAR

2.1 Creating Feature Modules

Feature modules allow **separating different parts of an application**.

Step 1: Generate a Feature Module

ng generate module dashboard --route dashboard --module app.module

✓ Creates a dashboard module with **lazy loading enabled**.

2.2 Configuring Lazy Loading in Routing

Modify app-routing.module.ts to **enable lazy loading**:

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

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


const routes: Routes = [

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

];


@NgModule({
```

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

✓ loadChildren loads DashboardModule **only when the user navigates to /dashboard**.

2.3 Creating Components in the Lazy Loaded Module

Navigate to the module directory and create a new component:

```
cd src/app/dashboard
ng generate component dashboard-home
```

Modify dashboard.module.ts:

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { DashboardHomeComponent } from './dashboard-home/dashboard-home.component';
import { RouterModule, Routes } from '@angular/router';

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

@NgModule({
```

```
  declarations: [DashboardHomeComponent],
  imports: [CommonModule, RouterModule.forChild(routes)]
})
export class DashboardModule { }
```

✓ Defines **routes inside the feature module** instead of the root module.

CHAPTER 3: CODE SPLITTING IN ANGULAR

3.1 How Angular Handles Code Splitting

When using **lazy loading**, Angular automatically:

✓ **Splits JavaScript bundles** into smaller files.
✓ Fetches only **necessary modules** on navigation.
✓ Loads **JS files asynchronously**, reducing initial load time.

3.2 Checking Code Splitting Using Webpack

Run the following command:

```
ng build --prod --stats-json
```

Then analyze the **generated bundles**:

```
npx webpack-bundle-analyzer dist/angular-app/stats.json
```

✓ Displays **JavaScript chunk sizes** to identify performance improvements.

3.3 Preloading Strategy for Faster Navigation

Preloading loads modules **in the background** to optimize **subsequent navigation**.

Modify app-routing.module.ts:

```
import { PreloadAllModules } from '@angular/router';


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

✓ This preloads **all lazy-loaded modules after the initial load**.

CHAPTER 4: OPTIMIZING LAZY LOADING PERFORMANCE

4.1 Best Practices for Lazy Loading

✓ **Keep the root module lightweight** – Avoid importing unnecessary modules globally.
✓ **Use feature modules** for independent functionalities.
✓ **Optimize images and assets** to prevent large bundles.
✓ **Implement preloading** for frequently accessed modules.

4.2 Debugging Lazy Loading Issues

Common Errors & Fixes:

Issue	Cause	Solution

Module not found	Incorrect path in loadChildren	Ensure correct module import path.
Component not loading	Component not declared in module	Add the component to declarations in *.module.ts.
Slow loading	Large bundle size	Optimize assets, enable gzip compression.

Case Study: How an E-Commerce Platform Improved Load Time with Lazy Loading

Background

An **e-commerce startup** faced **slow loading times** due to **large JavaScript bundles**.

Challenges

✓ **Homepage load time exceeded 5 seconds**.
✓ **Unused admin panel components** loaded on every page.
✓ **Mobile users experienced high data consumption**.

Solution: Implementing Lazy Loading

✓ Moved **Admin Panel** and **User Dashboard** into feature modules.
✓ Used **PreloadAllModules** to **reduce perceived delay**.
✓ Implemented **code splitting** for optimized script loading.

Results

🚀 **Load time reduced from 5s to 1.8s**.
📉 **Mobile data usage decreased by 50%**.
🔍 **Improved SEO rankings** due to faster page speed.

By **implementing lazy loading**, the startup **enhanced performance, reduced costs, and improved user experience**.

Exercise

1. **Create a feature module** (orders.module.ts) with a component (order-list).
2. **Enable lazy loading** for the OrdersModule in app-routing.module.ts.
3. **Analyze bundle size** before and after lazy loading.
4. **Enable preloading strategy** for optimized performance.

Conclusion

✓ **Lazy loading optimizes performance by loading modules on demand**.
✓ **Code splitting reduces JavaScript bundle size,** improving user experience.
✓ **Preloading strategies enhance navigation speed**.
✓ **Modular architecture ensures scalable applications**.

CHANGE DETECTION STRATEGIES IN ANGULAR

CHAPTER 1: INTRODUCTION TO CHANGE DETECTION IN ANGULAR

1.1 What is Change Detection?

Change detection is the mechanism that **updates the DOM (Document Object Model)** when the application state changes. Angular automatically **checks for changes in component data** and updates the UI accordingly.

✓ **Detects changes in component state and updates the view**.
✓ **Optimizes performance by reducing unnecessary updates**.
✓ **Improves responsiveness in real-time applications**.

1.2 How Does Change Detection Work in Angular?

Angular's change detection **monitors component state** and updates the view when:

✓ **Component properties change** (@Input() values).
✓ **Events occur** (click, keydown, keyup, etc.).
✓ **Asynchronous data arrives** (setTimeout(), HTTP requests, Observables).

CHAPTER 2: UNDERSTANDING CHANGE DETECTION MECHANISM

2.1 How Angular Detects Changes

Angular uses a **tree structure** to detect changes, checking:

✓ **Component properties and bindings**.
✓ **Events and async operations**.
✓ **Parent-to-child property changes** (@Input()).

2.2 Zones and Change Detection

Angular uses **NgZone** to track **asynchronous tasks** like:

✓ API requests (HttpClient).
✓ Timers (setTimeout, setInterval).
✓ Event listeners (click, keyup).

Whenever an **asynchronous operation completes**, Angular runs **change detection** to check if the UI needs updates.

CHAPTER 3: CHANGE DETECTION STRATEGIES IN ANGULAR

3.1 Default Change Detection Strategy

Angular's default strategy is **"Check Always"**, meaning:

✓ **Every component is checked** when a change occurs anywhere.
✓ **Works well for small applications** but **impacts performance in large apps**.

Example: Default Change Detection

```
@Component({
 selector: 'app-default-change',
 template: `<p>{{ counter }}</p>
        <button (click)="increase()">Increase</button>`
})
export class DefaultChangeComponent {
```

```
 counter = 0;


 increase() {
  this.counter++;
 }
}
```

✓ **Angular automatically detects changes** and updates the UI.

3.2 OnPush Change Detection Strategy

The **OnPush strategy** improves performance by **only detecting changes when**:

✓ **@Input() properties change.**
✓ **A new reference is assigned to an object.**

Example: Using ChangeDetectionStrategy.OnPush

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


@Component({
 selector: 'app-onpush-change',
 template: `<p>{{ user.name }}</p>`,
 changeDetection: ChangeDetectionStrategy.OnPush
})
export class OnPushChangeComponent {
```

```
  @Input() user: { name: string };
}
```

✓ **Angular skips checking this component** unless @Input() changes.
✓ **Improves performance in large applications**.

CHAPTER 4: MANUALLY TRIGGERING CHANGE DETECTION

4.1 Using ChangeDetectorRef to Trigger Change Detection

Sometimes, changes happen **outside Angular's detection mechanism** (e.g., when modifying objects directly).

Example: Using ChangeDetectorRef.detectChanges()

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

@Component({
  selector: 'app-manual-detect',
  template: `<p>{{ message }}</p>
        <button (click)="updateMessage()">Update</button>`
})
export class ManualDetectComponent {
  message = "Initial Message";

  constructor(private cdr: ChangeDetectorRef) {}
```

```
 updateMessage() {
  setTimeout(() => {
   this.message = "Updated Message!";
   this.cdr.detectChanges(); // Manually trigger change detection
  }, 2000);
 }
}
```

✓ detectChanges() **forces Angular to check for changes manually**.

4.2 Using markForCheck() to Detect Changes in OnPush Components

If a component uses ChangeDetectionStrategy.OnPush, updates might not be detected.
Use markForCheck() to **manually mark the component for checking**.

Example: Using markForCheck()

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


@Component({
 selector: 'app-mark-for-check',
 template: `<p>{{ counter }}</p>
        <button (click)="increase()">Increase</button>`,
 changeDetection: ChangeDetectionStrategy.OnPush
```

```
})
export class MarkForCheckComponent {
 counter = 0;

 constructor(private cdr: ChangeDetectorRef) {}

 increase() {
  this.counter++;
  this.cdr.markForCheck(); // Marks the component for checking
 }
}
```

✓ **Ensures OnPush components detect changes when needed**.

CHAPTER 5: OPTIMIZING PERFORMANCE WITH CHANGE DETECTION

5.1 When to Use Default vs. OnPush

Strategy	**When to Use**
Default (Check Always)	Small applications with fewer components.
OnPush (Check When Needed)	Large apps where performance is critical.

5.2 Best Practices for Efficient Change Detection

✓ **Use OnPush when components receive @Input() properties**.
✓ **Avoid mutating objects directly** – Use Object.assign() or spread

operator ({ ...obj }).

✓ **Use trackBy in *ngFor to optimize list rendering**.

✓ **Minimize use of detectChanges()** – Use only when necessary.

Example: Using trackBy in *ngFor

```
@Component({
 selector: 'app-list',
 template: `<div *ngFor="let user of users; trackBy: trackById">{{ user.name }}</div>`
})
export class ListComponent {
 users = [{ id: 1, name: "Alice" }, { id: 2, name: "Bob" }];

 trackById(index: number, item: any) {
  return item.id; // Returns unique ID
 }
}
```

✓ Prevents Angular from **re-rendering unchanged list items**.

Case Study: How an E-Commerce App Optimized Performance with OnPush

Background

An **e-commerce platform** faced **performance issues** due to:

✓ **Too many components triggering change detection**.
✓ **Slow UI updates** affecting the shopping experience.
✓ **Unnecessary re-renders of product listings**.

Challenges

- Every small UI change **triggered full component tree updates**.
- Large product lists **were being reloaded completely**.
- The checkout page **lagged due to excessive event updates**.

Solution: Implementing OnPush & TrackBy

✓ Used ChangeDetectionStrategy.OnPush to **reduce unnecessary checks**.
✓ Used trackBy in *ngFor to **optimize product listing updates**.
✓ Used markForCheck() in cart updates **to detect only relevant changes**.

Results

🚀 **UI updates were 60% faster**.
🔍 **Fewer re-renders,** improving customer experience.
⚡ **Optimized shopping cart interactions**.

By using **efficient change detection strategies,** the platform **improved speed and responsiveness**.

Exercise

1. Create a **simple counter component** using default change detection.
2. Convert it to **OnPush strategy** and observe changes.

3. Modify an object property and use markForCheck() to detect updates.
4. Use trackBy in *ngFor to optimize a list update.

Conclusion

In this section, we explored:

✓ **How Angular change detection works**.

✓ **When to use Default vs. OnPush change detection strategies**.

✓ **How to optimize performance with trackBy, markForCheck(), and detectChanges()**.

DEBUGGING & PROFILING ANGULAR APPLICATIONS

CHAPTER 1: INTRODUCTION TO DEBUGGING & PROFILING IN ANGULAR

1.1 Why Debugging and Profiling Matter?

Debugging and profiling are **essential** for maintaining a **high-performance Angular application**.

✓ **Debugging** helps **identify and fix errors** in the application.
✓ **Profiling** helps **analyze performance issues** like **slow rendering, high memory usage, or inefficient API calls**.

- **Common Angular Issues That Require Debugging & Profiling**:
  - **Template binding errors** (Cannot read property of undefined).
  - **Slow rendering due to unnecessary component re-renders**.
  - **Memory leaks due to unoptimized Observables and event listeners**.
  - **High CPU usage from inefficient change detection**.

CHAPTER 2: DEBUGGING ANGULAR APPLICATIONS

2.1 Using Browser Developer Tools

Modern browsers provide **built-in debugging tools** that allow developers to inspect, debug, and profile Angular applications.

Step 1: Open DevTools in Chrome or Edge

1. **Right-click on the application → Click Inspect**
2. **Go to the Console tab** to check for **JavaScript errors**.
3. **Go to the Sources tab** to **set breakpoints and debug code execution**.

2.2 Debugging Component Issues with console.log()

One of the simplest ways to debug is using console.log() to inspect data.

Example: Debugging a Component Property

```
ngOnInit() {
  console.log('User Data:', this.user);
}
```

✓ Displays the **value of user when the component initializes**.

2.3 Using debugger for Breakpoints

A **manual breakpoint** can be added using debugger in TypeScript code.

Example: Pausing Execution in a Function

```
getUserData() {
  debugger; // Execution stops here in DevTools
  return this.userService.getUser();
}
```

✓ The browser **pauses execution,** allowing inspection of **variables and stack traces**.

2.4 Inspecting Elements & Angular Components

Use the **Elements tab** in DevTools to inspect Angular components and styles.

1. **Select an element** in the DOM.
2. **Check applied styles** to debug CSS issues.
3. **Modify values in real-time** to test fixes before applying them in code.

2.5 Debugging Angular Templates (ExpressionChangedAfterItHasBeenCheckedError)

A common Angular error occurs when data updates **after change detection** has run.

✓ Fix: Wrap the code inside setTimeout() to schedule changes in the next cycle.

Example: Fixing Change Detection Error

```
ngOnInit() {
  setTimeout(() => {
    this.userName = 'John Doe';
  });
}
```

✓ Allows **Angular's change detection cycle** to complete before updating the UI.

CHAPTER 3: PROFILING ANGULAR APPLICATIONS

3.1 Measuring Performance with Chrome DevTools

1. Open **DevTools** → Click on the **Performance tab**.
2. Click **Record** and interact with the application.
3. Click **Stop** to analyze the results.

- **What to Look For?**
  - **Long task execution (>50ms)** – Indicates slow functions.
  - **Unnecessary re-renders** – Identify excessive component updates.
  - **Heavy API calls** – Optimize redundant HTTP requests.

3.2 Using console.time() for Manual Profiling

The console.time() method measures the execution time of functions.

Example: Measuring Execution Time of a Function

```
console.time('getUsers');
this.apiService.getUsers().subscribe(() => {
  console.timeEnd('getUsers'); // Logs the execution time
});
```

✓ Helps **identify slow API responses** and optimize performance.

CHAPTER 4: DEBUGGING HTTP REQUESTS WITH ANGULAR HTTPCLIENT

4.1 Using tap() to Log API Calls

The tap() operator in RxJS logs API responses **before they reach the component**.

Example: Debugging API Response in Service

```
import { tap } from 'rxjs/operators';


getUsers() {
 return this.http.get('/api/users').pipe(
  tap(response => console.log('API Response:', response))
 );
}
```

✓ Ensures that the **correct data is received** from the backend.

4.2 Monitoring Network Requests in DevTools

1. Open **DevTools** → Click on the **Network tab**.
2. Select **XHR/Fetch** to filter API requests.
3. Check **status codes (200, 404, 500)** for errors.
4. Analyze **response time** to optimize slow requests.

Chapter 5: Debugging Memory Leaks & Unsubscribed Observables

5.1 Identifying Memory Leaks

A **memory leak** occurs when **unused resources are not released,** leading to high RAM usage.

✓ **Common Causes in Angular**:

- Subscriptions that are **not unsubscribed**.
- Event listeners that are **not removed**.

5.2 Using ngOnDestroy() to Unsubscribe from Observables

Example: Preventing Memory Leaks in a Component

```
import { Subscription } from 'rxjs';


export class UserComponent implements OnDestroy {
 private userSubscription: Subscription;


 ngOnInit() {
  this.userSubscription = this.apiService.getUsers().subscribe();
 }


 ngOnDestroy() {
  this.userSubscription.unsubscribe(); // Prevents memory leaks
 }
}
```

✓ Ensures the **subscription is properly disposed of** when the component is destroyed.

5.3 Using takeUntil to Auto-Unsubscribe

The takeUntil operator **automatically unsubscribes** when a condition is met.

```
import { Subject } from 'rxjs';
import { takeUntil } from 'rxjs/operators';

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

ngOnInit() {
 this.apiService.getUsers().pipe(
  takeUntil(this.destroy$)
 ).subscribe();
}

ngOnDestroy() {
 this.destroy$.next(); // Cleanup
 this.destroy$.complete();
}
```

✓ **Avoids manual unsubscription,** making the code **cleaner**.

Case Study: How an E-Commerce Website Improved Performance with Debugging & Profiling

Background

An **e-commerce platform** noticed:

✓ **Slow product page loading**.

✓ **High API response times**.
✓ **Inconsistent UI updates** after adding items to the cart.

Challenges

- **Excessive component re-renders,** causing UI lag.
- **Memory leaks from unoptimized subscriptions**.
- **Unnecessary API requests,** increasing server load.

Solution: Using Debugging & Profiling

✓ **Enabled Performance Profiling** to identify slow functions.
✓ **Fixed excessive re-renders** using ChangeDetectionStrategy.OnPush.
✓ **Optimized API calls** by caching frequently accessed data.

```
@Component({
 selector: 'app-product',
 templateUrl: './product.component.html',
 changeDetection: ChangeDetectionStrategy.OnPush // Prevents unnecessary re-renders
})
export class ProductComponent { }
```

Results

🚀 **50% faster product page loads,** reducing customer drop-off.
⚡ **Reduced API requests by 30%,** optimizing server usage.
🔍 **Smoother UI experience,** improving user engagement.

By leveraging **debugging and profiling tools,** the company significantly **improved its application's performance**.

Exercise

1. **Use console.log()** to inspect a component's data in ngOnInit().
2. **Set a breakpoint in DevTools** and step through an API request.
3. **Use console.time()** to measure execution time for a function.
4. **Enable ChangeDetectionStrategy.OnPush** and observe the performance improvement.
5. **Find and fix a memory leak** by unsubscribing from an Observable in ngOnDestroy().

Conclusion

In this section, we explored:

✓ **How to debug Angular applications using DevTools, console logs, and breakpoints**.

✓ **How to profile performance and optimize API calls**.

✓ **How to prevent memory leaks using unsubscribe() and takeUntil**.

ASSIGNMENT:

DEVELOP A FEATURE-RICH ANGULAR APPLICATION WITH COMPONENT COMMUNICATION AND LAZY LOADING

Assignment Solution: Developing a Feature-Rich Angular Application with Component Communication and Lazy Loading

---

Application Overview

We will build a **Task Management App,** where users can:

✅ Add new tasks

✅ View a list of tasks

✅ Mark tasks as complete

✅ Navigate between different modules efficiently using **lazy loading**

Step 1: Set Up the Angular Project

1.1 Install Angular CLI (If Not Installed)

Ensure Angular CLI is installed:

npm install -g @angular/cli

1.2 Create a New Angular Project

Run the following command:

ng new task-manager

cd task-manager

✅ Select **Routing** and choose **CSS** for styling.

Step 2: Create Components for Task Management

2.1 Generate Components

ng g c components/task-list

ng g c components/task-item

ng g c components/task-form

✓ TaskListComponent – Displays all tasks.
✓ TaskItemComponent – Represents a single task.
✓ TaskFormComponent – Handles adding new tasks.

Step 3: Implement Component Communication

3.1 Define Task Model

Create models/task.model.ts:

```
export interface Task {
  id: number;
  title: string;
  completed: boolean;
}
```

3.2 Implement Task Service for Shared Data

Create services/task.service.ts:

```
import { Injectable } from '@angular/core';
import { BehaviorSubject } from 'rxjs';
import { Task } from '../models/task.model';
```

```
@Injectable({
 providedIn: 'root'
})
export class TaskService {
 private tasks: Task[] = [
  { id: 1, title: "Learn Angular", completed: false },
  { id: 2, title: "Build a project", completed: false }
 ];

 private tasksSubject = new BehaviorSubject<Task[]>(this.tasks);
 tasks$ = this.tasksSubject.asObservable();

 addTask(title: string) {
  const newTask: Task = { id: Date.now(), title, completed: false };
  this.tasks.push(newTask);
  this.tasksSubject.next([...this.tasks]);
 }

 toggleTaskCompletion(task: Task) {
  task.completed = !task.completed;
  this.tasksSubject.next([...this.tasks]);
```

```
  }
}
```

✓ **Manages tasks** and **updates state using RxJS BehaviorSubject**.
✓ Enables **real-time updates** across components.

3.3 Implement Task Form Component (@Output() for Event Communication)

Modify task-form.component.ts:

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


@Component({
  selector: 'app-task-form',
  templateUrl: './task-form.component.html'
})
export class TaskFormComponent {
  @Output() taskAdded = new EventEmitter<string>();
  taskTitle = '';

  addTask() {
    if (this.taskTitle.trim()) {
      this.taskAdded.emit(this.taskTitle);
      this.taskTitle = '';
    }
```

```
  }
}
```

Modify task-form.component.html:

```
<input type="text" [(ngModel)]="taskTitle" placeholder="Enter a task">
<button (click)="addTask()">Add Task</button>
```

✓ **Uses @Output() to send task data to the parent component.**

3.4 Implement Task Item Component (@Input() for Data Passing)

Modify task-item.component.ts:

```
import { Component, Input, Output, EventEmitter } from '@angular/core';
import { Task } from 'src/app/models/task.model';

@Component({
  selector: 'app-task-item',
  templateUrl: './task-item.component.html'
})
export class TaskItemComponent {
  @Input() task!: Task;
  @Output() taskToggled = new EventEmitter<Task>();

  toggleTask() {
```

```
  this.taskToggled.emit(this.task);
 }
}
```

Modify task-item.component.html:

```
<div [class.completed]="task.completed">
 <input type="checkbox" [checked]="task.completed"
(change)="toggleTask()">
 {{ task.title }}
</div>
```

✓ **Displays task details and notifies the parent on status change.**

3.5 Implement Task List Component (@Input() and @Output())

Modify task-list.component.ts:

```
import { Component } from '@angular/core';
import { TaskService } from 'src/app/services/task.service';
import { Task } from 'src/app/models/task.model';

@Component({
 selector: 'app-task-list',
 templateUrl: './task-list.component.html'
})
export class TaskListComponent {
 tasks$ = this.taskService.tasks$;
```

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

 addTask(taskTitle: string) {
  this.taskService.addTask(taskTitle);
 }

 toggleTask(task: Task) {
  this.taskService.toggleTaskCompletion(task);
 }
}
```

Modify task-list.component.html:

```
<app-task-form (taskAdded)="addTask($event)"></app-task-form>
<div *ngFor="let task of tasks$ | async">
 <app-task-item [task]="task"
(taskToggled)="toggleTask($event)"></app-task-item>
</div>
```

✓ **Subscribes to tasks using async pipe** for real-time updates.
✓ **Implements two-way communication** between parent (task-list) and children (task-item, task-form).

Step 4: Implement Lazy Loading for Modules

4.1 Generate Modules

ng g module modules/tasks --routing

4.2 Move Components into tasks Module

Modify tasks-routing.module.ts:

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { TaskListComponent } from '../components/task-list/task-list.component';

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

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

Modify app-routing.module.ts:

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

✓ **Loads TasksModule only when the user navigates to /tasks,** improving performance.

Step 5: Style and Enhance the UI

Modify styles.css to enhance UI:

```
.completed {
  text-decoration: line-through;
  color: gray;
}
input, button {
  padding: 8px;
  margin: 5px;
}
```

✓ **Improves visual clarity and user experience**.

Step 6: Run and Test the Application

To start the Angular development server:

```
ng serve
```

Navigate to http://localhost:4200/tasks to see the app in action. 🚀

Summary of Key Features

✓ **Component Communication:**

- @Input() to pass task data to child components.
- @Output() to notify parent components of events.
- Shared TaskService for state management.

✓ **Lazy Loading:**

- TasksModule is loaded **only when needed,** reducing the initial load time.

✓ **Reactive UI Updates:**

- Uses RxJS BehaviorSubject to **stream live updates**.

✓ **User-Friendly Design:**

- Simple yet effective **task list with completion toggles**.

Conclusion

🎯 **We have successfully built a feature-rich Angular application** that demonstrates:

✓ **Component communication using @Input() and @Output()**.
✓ **State management with an Angular service**.
✓ **Lazy loading to improve application performance**.
✓ **Event-driven updates using RxJS**.

💡 **Next Steps:**

- Extend the app by **adding authentication and API integration**.
- Implement **pagination and search** for better usability.
- Improve styling with **Angular Material or Bootstrap**.