Lesson 10: State Management in Angular 19

Lesson 10: State Management in Angular 19

In this tutorial, we'll explore how to manage state in Angular 19, covering both component-level and app-level state management. We'll dive into using services, the new core feature Signals, and briefly touch on NgRx for advanced use cases. By the end, you'll understand how to handle state effectively in Angular 19 applications.

1. Introduction to State Management

What is State?

State is the data that drives your application’s behavior and UI at any given time. Examples include user inputs, UI toggles, or data fetched from a server.

Why is State Management Important?

As your Angular app grows, managing state ensures:
- Consistency: Data stays in sync across components.
- Performance: Updates are efficient.
- Maintainability: Code remains clean and scalable.

In Angular 19, we’ll focus on modern tools like Signals alongside traditional approaches like services.

2. Managing State at the Component Level

What is Component-Level State?

This is data managed within a single component, perfect for isolated features like a counter or toggle.

How to Implement It

Use component properties and methods to track and update state. Angular 19 leverages standalone components by default.

Example: A Simple Counter

Let’s build a counter component.

1. Generate the Component

```
ng generate component counter
```

2. Update the Component

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

@Component({
        selector: 'app-counter',
        standalone: true,
        template: `
        <p>Count: {{ count }}</p>
        <button (click)="increment()">Increment</button>
        `,
})
export class CounterComponent {
        count = 0;

        increment() {
        this.count++;
        }
}
```

3. Add to Your App
   In `app.component.ts`:

```
import { Component } from '@angular/core';
import { CounterComponent } from './counter/counter.component';

@Component({
        selector: 'app-root',
        standalone: true,
        imports: [CounterComponent],
        template: '<app-counter></app-counter>',
})
export class AppComponent {}
```

4. Run the App

ng serve

Click the button to increment the count. Angular’s change detection updates the view automatically.

When to Use

Use component-level state for small, self-contained features.

3. Managing State at the App Level

Why App-Level State?

App-level state is shared across components, such as a shopping cart or user profile data.

Challenges

- Avoid passing data through multiple layers (prop drilling).
- Keep data consistent across the app.

We’ll use services for simple app-level state and explore Signals for reactive updates.

4. Using Services for Simple State Management

What are Services?

Services are injectable singletons in Angular, ideal for sharing state and logic across components.

Example: Managing a Todo List

Let’s create a todo app with a service.

1. Generate the Service

```
ng generate service todo
```

2. Define the Service

typescript

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

@Injectable({
	providedIn: 'root'
})
export class TodoService {
	private todos: string[] = [];

	getTodos(): string[] {
	return this.todos;
	}

	addTodo(todo: string): void {
	this.todos.push(todo);
	}

	removeTodo(index: number): void {
	this.todos.splice(index, 1);
	}
}
```

3. Create Todo List Component

ng generate component todo-list

Update `todo-list.component.ts`:
typescript

```
import { Component, inject } from '@angular/core';
import { TodoService } from '../todo.service';

@Component({
        selector: 'app-todo-list',
        standalone: true,
        template: `
        <h2>Todo List</h2>
        <ul>
        <li *ngFor="let todo of todos; let i = index">
        {{ todo }} <button (click)="removeTodo(i)">Remove</button>
        </li>
        </ul>
        `,
})
export class TodoListComponent {
        private todoService = inject(TodoService);
        todos = this.todoService.getTodos();

        removeTodo(index: number) {
        this.todoService.removeTodo(index);
        }
}
```

4. Create Todo Input Component

ng generate component todo-input

Update `todo-input.component.ts`:
typescript

```
import { Component, inject } from '@angular/core';
import { TodoService } from '../todo.service';

@Component({
        selector: 'app-todo-input',
        standalone: true,
        template: `
        <h2>Add Todo</h2>
        <input #todoInput (keyup.enter)="addTodo(todoInput.value); todoInput.value="">
        <button (click)="addTodo(todoInput.value); todoInput.value="">Add</button>
        `,
})
export class TodoInputComponent {
        private todoService = inject(TodoService);

        addTodo(todo: string) {
        if (todo.trim()) {
        this.todoService.addTodo(todo.trim());
        }
        }
}
```

5. Update App Component

In `app.component.ts`:

typescript

```
import { Component } from '@angular/core';
import { TodoInputComponent } from './todo-input/todo-input.component';
import { TodoListComponent } from './todo-list/todo-list.component';

@Component({
    selector: 'app-root',
    standalone: true,
    imports: [TodoInputComponent, TodoListComponent],
    template: `
    <app-todo-input></app-todo-input>
    <app-todo-list></app-todo-list>
    `,
})
export class AppComponent {}
```

6. Test the App

Run `ng serve`. Add todos and remove them; the list updates instantly.

When to Use

Services are great for small to medium apps with straightforward state sharing.

5. Introduction to Signals in Angular 19

What are Signals?

Introduced as a core feature in Angular 19, Signals provide a reactive way to manage state. They automatically update the UI when state changes, without manual subscriptions like Observables.

Benefits

- Simplified reactivity.
- Fine-grained change detection for better performance.
- Reduces boilerplate compared to older approaches.

Example: Signal-Based Counter

Let’s rewrite the counter using Signals.

1. Update Counter Component

```typescript
import { Component, signal, effect } from '@angular/core';

@Component({
        selector: 'app-counter',
        standalone: true,
        template: `
        <p>Count: {{ count() }}</p>
        <button (click)="increment()">Increment</button>
        `,
})
export class CounterComponent {
        count = signal(0);

        constructor() {
        effect(() => {
        console.log(`The count is now ${this.count()}`);
        });
        }

        increment() {
        this.count.update(value => value + 1);
        }
}
```

2. Run the App
   Click the button; the count updates, and the `effect` logs the change.

 Explanation
- `signal(0)` creates a reactive value.
- `count()` retrieves the value; `count.update()` modifies it.
- `effect` runs automatically when the signal changes.

 When to Use
Use Signals for reactive state in modern Angular 19 apps, especially where performance matters.

Next NGRX