Angular Training

Signal and Use case in form

What are signals?

A **reactivity primitive** introduced in Angular 16

Similar to state management in SolidJS or Svelte

Enables **fine-grained reactivity** without RxJS

Part of Angular’s push towards **simpler, more predictable** state handling

Why signals?

Traditional Approach	Signal-Based Approach
Uses RxJS and Observables	Uses lightweight, built-in APIs
Imperative updates	Reactive updates
Hard to trace changes	Transparent and trackable
Requires async pipe in template	Direct invocation in template

Basic Signal syntax

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


const counter = signal(0);


// Reading

console.log(counter()); // 0


// Updating

counter.set(1);

counter.update(value => value + 1);
```

Computed signal

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


const firstName = signal('Jane');

const lastName = signal('Doe');


const fullName = computed(() => `${firstName()} ${lastName()}`);
```

- Automatically updates when dependencies change
- Useful for derived state

Effects

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


effect(() => {
  console.log('Counter changed:', counter());
});
```

- Side effects triggered when signal values change
- Ideal for logging, API calls, or syncing values

Signals in the template

You can use signals **directly in the template**:

```
<p>{{ counter() }}</p>

<button (click)="counter.set(counter() + 1)">Increment</button>
```

No need for async pipe

Clean and readable syntax

Signals vs Observables

Feature	Signals	Observables (RxJS)
Learning Curve	Easy	Steeper (RxJS)
API Type	Synchronous	Asynchronous
Use in HTML	Direct call ( )	`
Dependency	Built-in	External (RxJS)

When to use Signals

- Component-level state
- Derived UI values
- Forms or inputs
- Replacing `BehaviorSubject` in simple cases

Not suitable for:

- Streams (e.g., WebSocket, long polling)
- Asynchronous workflows (use `RxJS` instead)

Real world example

```
@Component({
  standalone: true,
  selector: 'app-counter',
  template: `
    <h2>Counter: {{ counter() }}</h2>
    <button (click)="increment()">+</button>
  `
})
```

Real world example (2)

```
export class CounterComponent {
  counter = signal(0);

  increment() {
    this.counter.update(v => v + 1);
  }
}
```

Conclusion

Concept	Description
signal()	Creates a reactive state container
computed()	Derives reactive values
effect()	Triggers effect on change
`signal()` in template	Used directly without async pipe

Using signal with Form

Eliminate `valueChanges` subscriptions

Simplify reactivity and state tracking

Ideal for **lightweight, standalone forms**

Native Angular reactivity — no RxJS required

Form setup with signal

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


const name = signal('');
const email = signal('');
const message = signal('');
```

Each input field is tied to a reactive signal

Direct and observable without extra boilerplate

Template binding

```
<form (ngSubmit)="handleSubmit()">
  <input [value]="name()" (input)="name.set($event.target.value)" placeholder="Name" />
  <input [value]="email()" (input)="email.set($event.target.value)" placeholder="Email" />
  <textarea [value]="message()" (input)="message.set($event.target.value)"></textarea>

  <button type="submit">Send</button>
</form>
```

Works with native HTML inputs

Signals are used like local form state

Computed signal for Live preview

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


const preview = computed(() => `${name()} <${email()}> says: ${message()}`);


<div class="preview">
  <h4>Live Preview</h4>
  <p>{{ preview() }}</p>
</div>
```

Validating with signal

```
const isFormValid = computed(() =>
  name().length > 0 &&
  email().includes('@') &&
  message().length >= 10
);
<button [disabled]="!isFormValid()">Submit</button>
```

Simple validation logic, declaratively defined

Updates automatically as input signals change

Handling Submission

```
function handleSubmit() {
  console.log('Form Submitted:', {
    name: name(),
    email: email(),
    message: message(),
  });
}
```

When to Use Signal-Based Forms

- Simple contact forms
- Live previews or instant feedback
- Standalone or embedded components

Not ideal for:

- Dynamic forms (e.g., FormArray)
- Complex validation scenarios (use ReactiveFormsModule instead)

Benefit recap

Feature	Signal-Based Forms
Simplicity	Very low boilerplate
Reactivity	Instant field updates
Setup	No extra modules needed
Template Logic	Clean, signal-based logic
Ideal Use Case	Lightweight forms