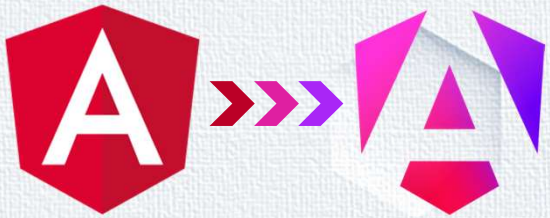
```
C:\ng new_
```



Angulars Renaissance

Hi, I'm...

Kobi Hari

- Freelancer
- Developer, Instructor and Consultant
- Angular, Async development, .NET core

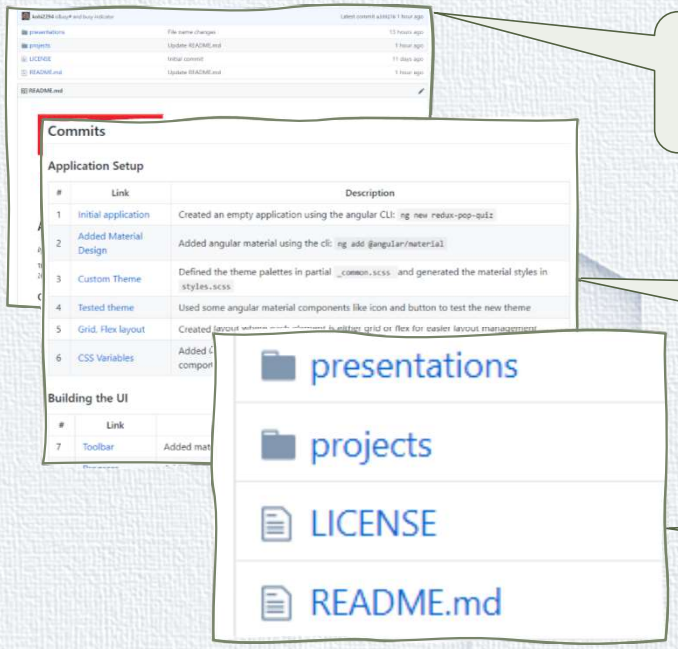
My email: hari@applicolors.com

Courses on Udemy: https://www.udemy.com/user/kobi-hari/

My Angular Channel: https://www.youtube.com/@kobihari

We have a GitHub Repository!

kobi-hari-courses/2511-oracle-ai-days-ngnew



Session Summary

Project Code with Commits Summary

This presentation and other Useful Links

Our Agenda

Signals – Under the hood

Resources – async Signals

Signal Forms (new)

Signals

The new reactivity

Angular

15

Signals

- Normal properties do not maintain their relationships with each other

```
let width = 10;
let height = 20;
let area = width * height; // 200

width = 15;
// area is still 200
```

- With signals, atomic values “signal” when they change, so computed values remain correct

- Yes… It’s kinda like `BehaviorSubject`

```
let width = signal(10);
let height = signal(20);
let area = computed(() => width() * height()); // 200

width.set(15);
console.log(area()); // area is 300
```

The signal primitives

- You create a signal with the `signal()` function
- You create a computed signal with the `computed()` function
  - Do not cause side effects inside it
  - Do not create new signals inside it
  - This should be a pure function that depends on other signals – that's it
- You can read from a signal by simply calling it like a function
  - It is synchronous
  - It will return a value instantly
  - No need to subscribe – key difference from observables

```
let firstName = signal('Kobi');
let lastName = signal('Hari');


let fullName = computed(() =>
     firstName() + ' ' + lastName());

console.log(fullName());
```

Updating signals

- You can only update atomic (writeable) signals
- Use set() when you want to set a new value that does not depend on the previous
- Use update() when you want to set a new value that **does** depend on previous value
- ~~Use mutate() when you want to modify the original value (mutate array or object inside signal).~~
  - Don't…

Do

```
firstName.set('Yakov');
lastName.update(val => val.toLowerCase());
```

Don't

```
firstName.set('Mr ' + firstName());
```

Instead, do

```
let titledName = computed(() => 'Mr ' + firstName());

// or

firstName.update(val => 'Mr ' + val);
```

effect

- Use effect to respond to changes in signals
  - The effect will run once when defined
  - It will run again every time one the signals it depends on change
- You may not change other signals inside effect
  - Ish…
  - This started off as forbidden (in Angular <=18) and then became “not recommended” in Angular 19
  - But you **Can** modify signals inside effect after `await`

```
effect(() => {
    console.log('full name: ', fullName());
});
```

Don’t

```
let salutation = signal('');
effect(() => {
    salutation.set('Hello' + fullName());
});
```

Instead, do

```
let salutation = computed(() => 'Hello' + fullName());
```

This is, actually, allowed

```
effect(async () => {
    if (fullName() === 'Kobi') {
        let res = await (someService.someMethod());
        resultSignal.set(res);
    }
});
```

Limitations

1. Signals are synchronous – so they must have initial value when created

2. effect() must be called in Injection Context
   - Because they rely on `DestroyRef` to complete and unsubscribe

Modern Angular 20 with Signals

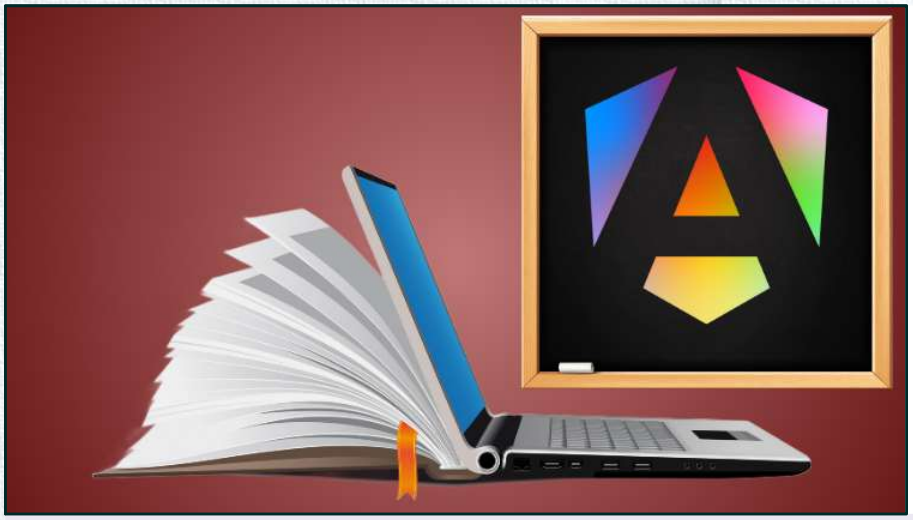
The missing guide

by Kobi Hari

NgRx Signal Store 19

The missing guide

by Kobi Hari



Theming Angular 20 & Material MD3

The missing guide

by Kobi Hari

Angular

16


Angular
17

Signal Components

Zoneless we go!

Signal Inputs

ng old (<18)

```
export class SelectorComponent {
  @Input() options: string[] = [];

  @Input({required: true}) selectedOption!: string;
}
```

ng new (18+)

```
export class SelectorComponent {
  readonly options = input<string[]>([]);

  readonly selectedOption = input.required<string>();
}
```

Inputs are now signals

- You do not need lifecycle hooks to respond to changes
- You can derive computed values from them
- You can respond to their changes using effects
- You can derive observables from them using `toObservable`

```
  readonly selectedIndex = computed(() =>
    this.options().indexOf(this.selectedOption()));
```

```
  constructor() {
    effect(() => {
        localStorage.setItem('selectedOption',
this.selectedOption());
    })
  }
```

Output functions

ng old (<18)

```
export class SelectorComponent {
  @Output() selectedOptionChange = new EventEmitter<string>();
}
```

ng new (18+)

```
export class SelectorComponent {
  selectedOptionChange = output<string>();
}
```

The new `output()`

- It's not a signal – but an event emitter
- Interoperability with RxJS
  - Create output from observable using "`outputFromObservable`"
  - Create observable from output using "`outputToObservable`"

```
import { outputFromObservable } from
'@angular/core/rxjs-interop';

timesUp = outputFromObservable(timer(3000).pipe(
    map(_ => 'Time is up!'))
);
```

models

- Model is a new approach to create two-way binding in angular
- A model input is essentially a writeable signal (while normal inputs are read-only)
- Child component may pass data to parent
  - By calling an event
  - Or, by setting a writeable signal which is passed by reference

Defining and Using model inputs

- Define a model using the model function
  - You can use model.required mandatory inputs
- In the parent component, you can pass value like any other inputs
  - Using expression
  - Using signal values
- In the parent component, you can receive changes
  - Using an event handler
  - Using a writeable signal

```ts
// counter.component.ts
export class CounterComponent {
  readonly value = model.required<number>();

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

```html
// app.component.html
<app-counter [value]="2"/>

<app-counter [value]="mySignal()"/>
```

```html
// app.component.html
<app-counter [value]="2“
    (valueChange)="counterChanged($event)"/>

<app-counter [(value)]="mySignal"/>
```

`[(ngModel)]` is back!

- The “banana-in-a-box” syntax is back
- Two-way-binding
- You can (once again) use ngModel to bind inputs to signals
- Works with any form control.
- Handle with care…

Signal Queries

ng old (<18)

```
// counter.component.html
<div #counterValue>{{value()}}</div>

// counter.component.ts
  @ViewChild(counterValue')
  title: ElementRef | undefined = undefined;
```

ng new (18+)

```
// counter.component.ts
  readonly counterValue = viewChild<ElementRef>(‘counterValue');
```

Signal Queries

ng old (<18)

```
// app.component.ts
  @ViewChildren(CounterComponent)
  counters!: QueryList<CounterComponent>;

// counters is a QueryList<CounterComponent> and
// you can subscribe to it
```

ng new (18+)

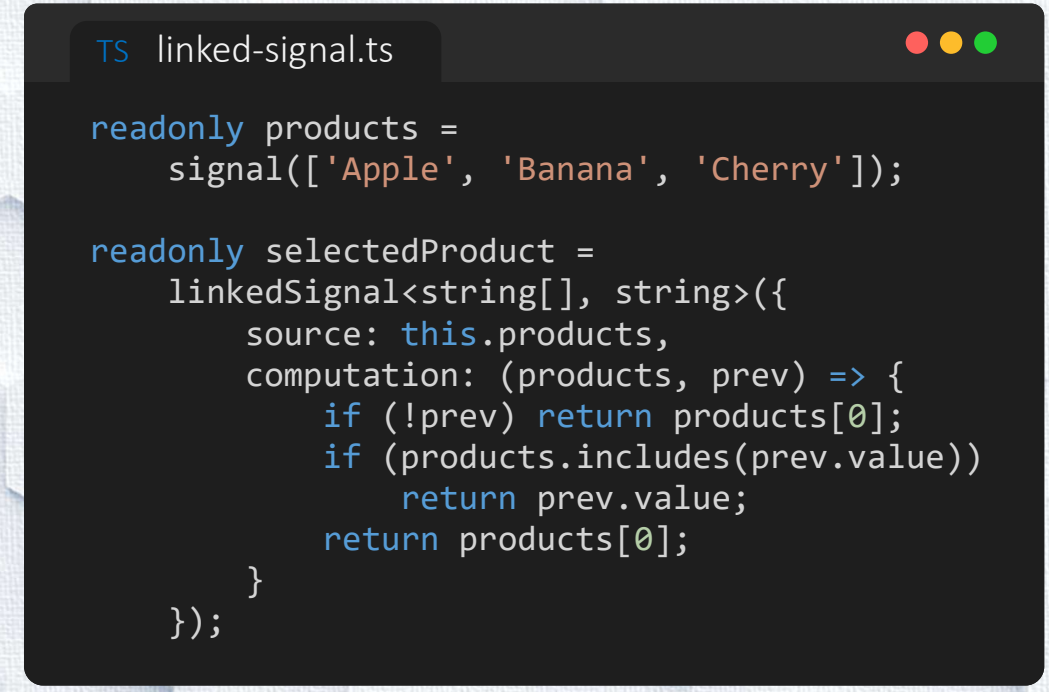
```
// app.component.ts
  readonly counters = viewChildren(CounterComponent);

// counters is Signal<CounterComponent[]>
```


Angular
18

Linked Signal

A combination of computed and writeable signal

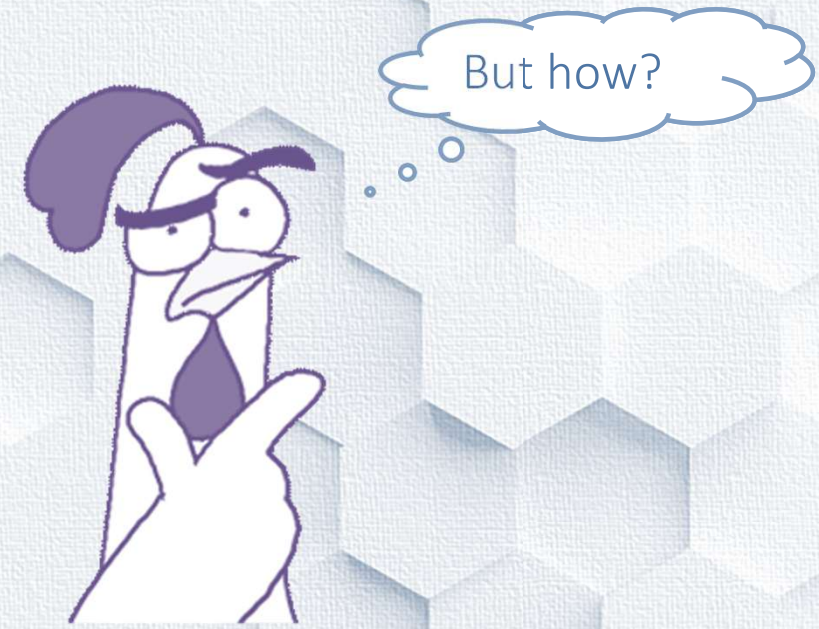
- Like `computed` – it has a computed expression
- But you can override the value
- using `set` and `update`



Signal - Deep Dive

It’s all in the context

No Subscription needed

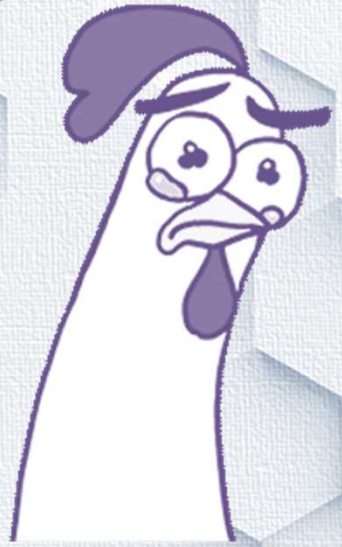


Let’s talk about context

Angular

Injection Context

Reactive Context



What is “context” anyway?

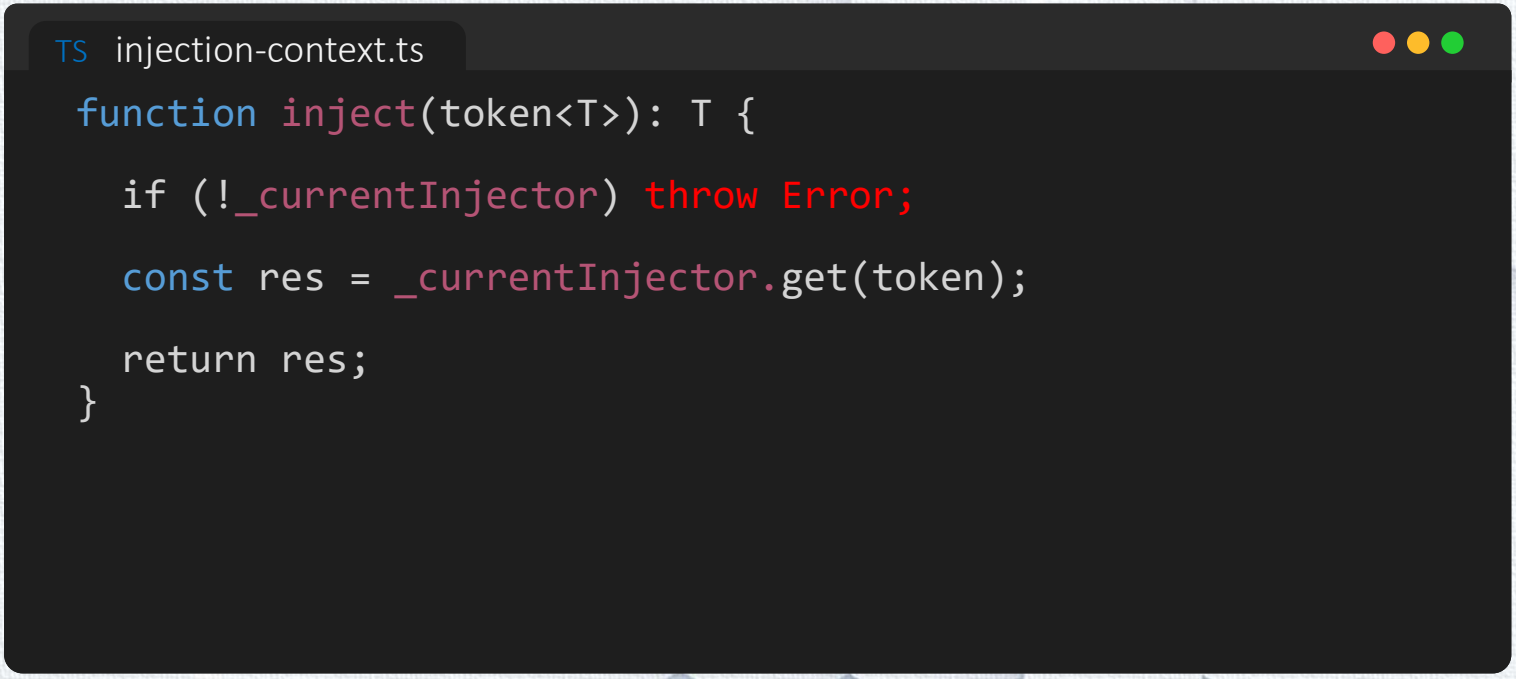
Injection Context

- `inject()` needs to know who called it – to get the correct **`Injector`**
- Angular sets a "`_currentInjector`" and then calls the function
- While the function is running inject can access the `_currentInjector`.
- That means the function is running in "**Injection Context**"

TS injection.ts

```ts
function startTimer() {
  const dr = inject(DestroyRef);
  const timer = setTimeout(…);
  dr.onDestroy(() => {
      clearTimeout(timer);
  });
}
```

Run In Injection context



Run In Injection context

TS injection-context.ts

```ts
function runInInjectionContext(injector: Injector, action: () => T) {
  const prevInjector = _currentInjector;
  _currentInjector = injector;
  const res = action();
  _currentInjector = prevInjector;
  return res;
}
```

Let’s talk about context

Angular

Injection Context

Reactive Context

Injection Context

- `inject()` needs to know who called it – to get the correct **`Injector`**
- Angular sets a "`_currentInjector`" and then calls the function
- While the function is running inject can access the `_currentInjector`.
- That means the function is running in "**Injection Context**"

TS injection.ts

```ts
function startTimer() {
  const dr = inject(DestroyRef);
  const timer = setTimeout(…);
  dr.onDestroy(() => {
      clearTimeout(timer);
  });
}
```

Reactive Context

- `signal()` needs to know who called it – to get the correct **`Reactor`**
- Angular sets a "`_currentReactor`" and then calls the function
- While the function is running the signal can access the `_currentReactor`.
- That means the function is running in "**Reactive Context**"

TS injection.ts

```
effect (() => {
  const val = mySignal();
  // do something with val
})
```

Different Ways to call signals

One time

TS reactive-context.ts

```ts
readonly mySignal = signal(42);
function onClick() {
  const val = mySignal();
  // do something with val
}
```

Different Ways to call signals

One time inside computed

TS reactive-context.ts

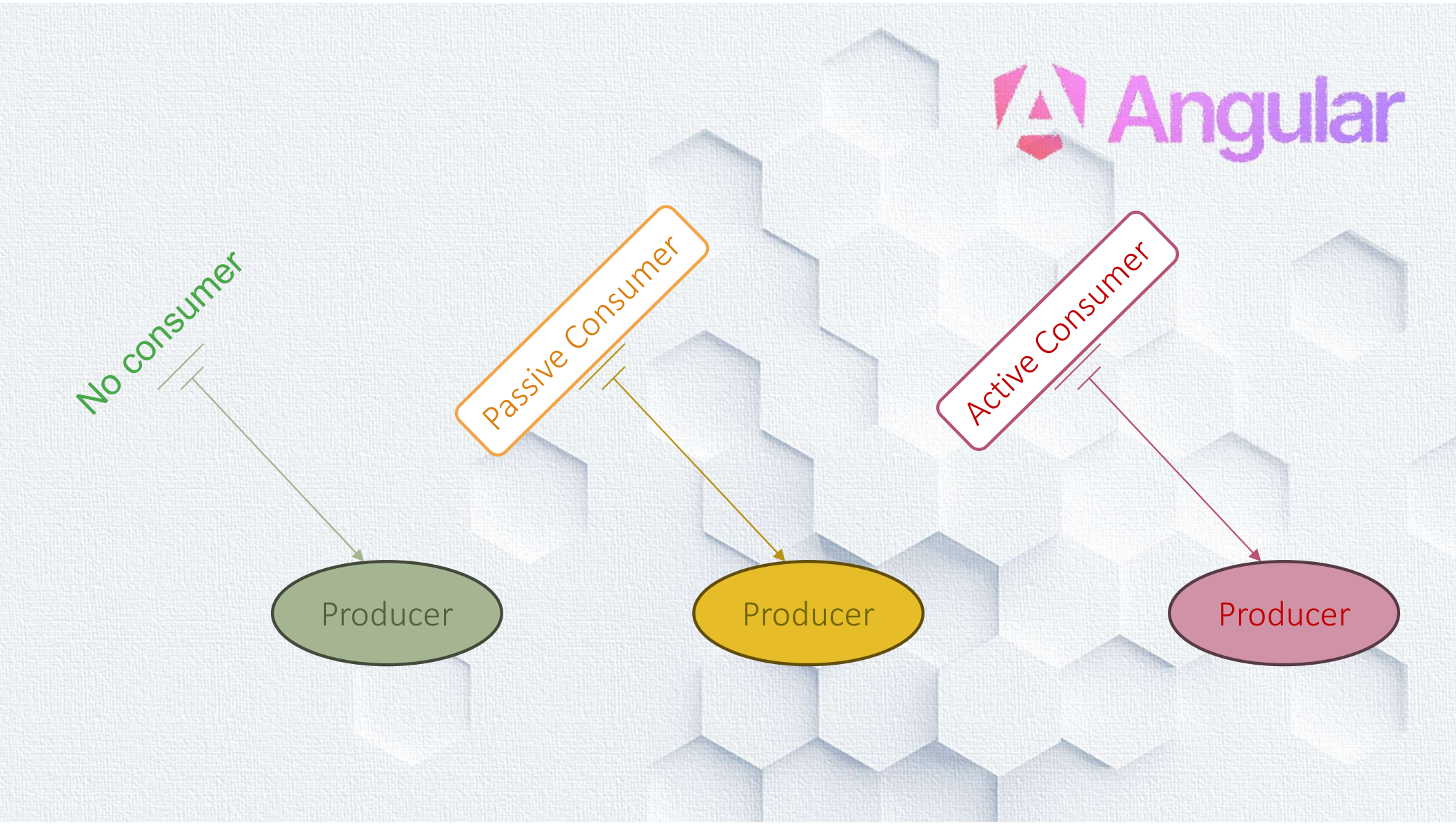
```ts
readonly mySignal = signal(42);
readonly myComputed = computed(() => mySignal() * 2);
function onClick() {
  const val = myComputed();
  // do something with val
}
```

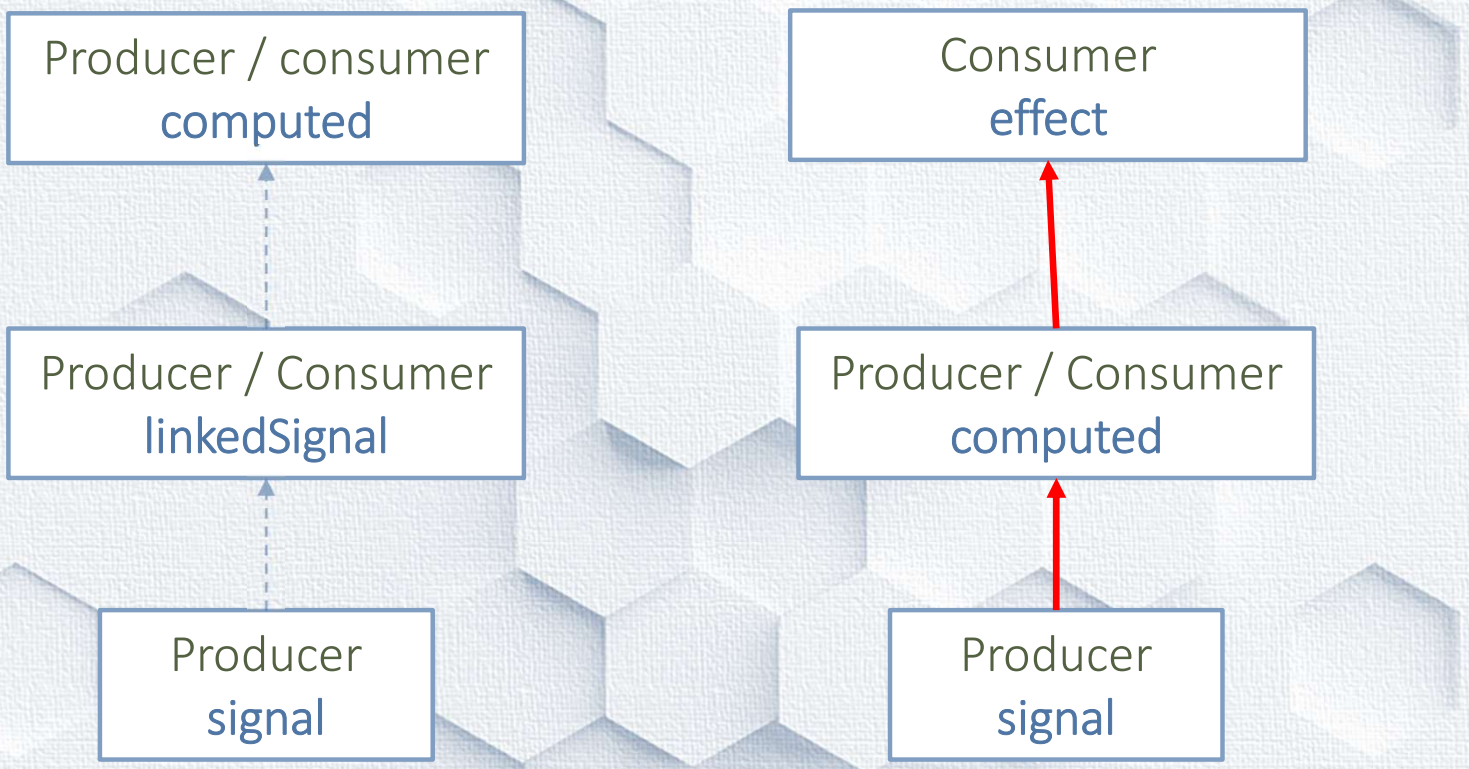
Different Ways to call signals

inside effect (subscription)

TS reactive-context.ts

```ts
readonly mySignal = signal(42);
readonly myComputed = computed(() => mySignal() * 2);
effect (() => {
  const val = myComputed();
  // do something with val
})
```


Angular
No consumer
Passive Consumer
Active Consumer
Producer
Producer
Producer


Producer / consumer
computed
Consumer
effect
Producer / Consumer
linkedSignal
Producer / Consumer
computed
Producer
signal
Producer
signal

Question

Angular

Injection Context

Reactive Context

Which context is used in **effects**?

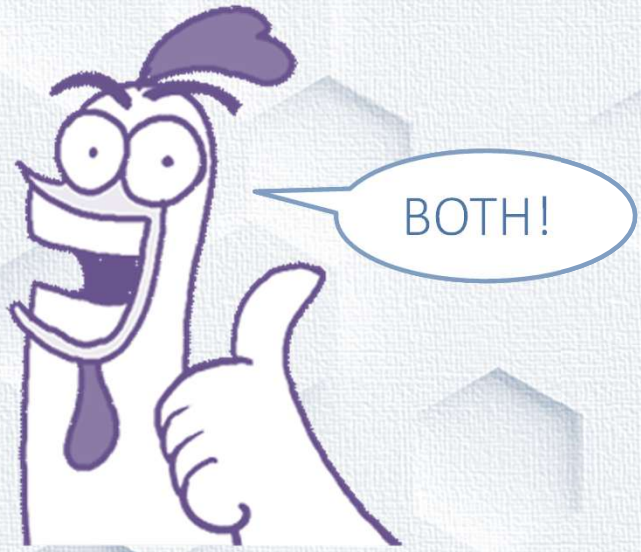
Answer is:

Reactive Context

Signals will know which **reactive consumer** to trigger when they change

Injection Context

Signal subscription is terminated when the **Injector** is destroyed



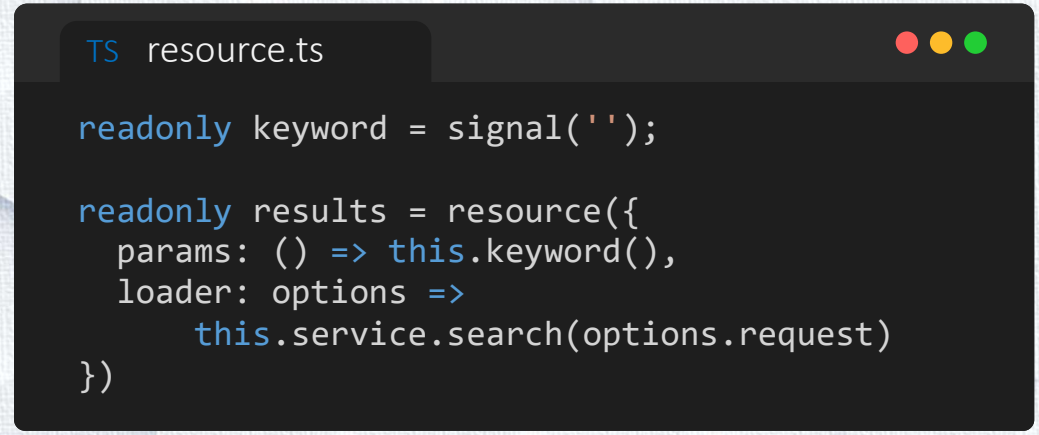
Signal Resources

Asynchronous Signals

Resource signals

Bridges between signals and `async` primitives

- `resource` – for promises
- `rxResource` – for observables
- `httpResource` – Specifically for http



Resource signals

- Contains several signals:
  - `value` – the latest value
  - `error` – the latest error, or undefined
  - `status` – the current async status
  - `isLoading` – guess…
- Methods:
  - `set`, `update` –modify to a **local** value
  - `reload` – reloads with the latest params
  - `asReadonly` – readonly, non writeable version

TS resource.ts

```
constructor() {
  effect(() => {
    console.log('results status’,
        this.results.status());

    console.log('results data’,
        this.results.value());

    console.log('results error’,
        this.results.error());

    console.log('results loading’,
        this.results.isLoading());
  })
}
```

Resource Signals - Statuses

Idle — Not loaded yet

Error — Latest load failed

Loading — Triggered by new request

Reloading — Triggered by reload method

Resolved — Value loaded

Local — Value overridden locally


Angular
19

HttpResource – wraps HttpClient

- Accepts a URL that is wrapped with "effect"
- Can also accept params as object
- Fetches data from the url whenever the url changes
- Supports "reload", and local value
- All Http Client configurations are supported
  - Yes, including Interceptors

TS http-resource.ts

```ts
#searchResult = httpResource<Book[]>(() => ({
  url: `${this.apiBase}/search`,
  params: { q: this.#keyword()},

}), {
  defaultValue: [],
})
```

resource – with streaming

- Resource supports 2 modes
  - Loader
  - Stream
- The “stream” callback is called once per request for setup
- It is required to return a signal
- It is supposed to set a “background” process that updates the signal
- Usually, creates an observable and subscribes to it
- Relatively complex syntax… ☹

TS stream-resource.ts

```
#selectedStock = resource({
    params: () => ({id: this.#selectedBookId()}),
    stream: async (options) => {
      const res = signal<ResourceStreamItem<number>>({value: 0
});
      const o$ = interval(1000);
      o$.subscribe(val => {
           res.set({value: val})
      });
      // need to also take care of cancellation
     }

      return res;
    }
  })
```

`rxResource` – best for streaming

- Works with Observables
- Has **only** stream option
- The stream function needs to calculate a new observable based on the params (and perhaps previous value)
- It will subscribe to the observable and copy its value to the resource
- Automatically supports cancellation by unsubscribing from the observable when needed

TS rx-resource.ts

```ts
#selectedStock = rxResource({
    params: () => ({id: this.#selectedBookId()}),
    stream: (options) => {
      if(!options.params.id) return of(0);
      return interval(1000)
    }
})
```

Demo – Resource Signals

- ✓ Using httpResource
- ✓ Using resource with promises
- ✓ Streaming with rxResource
- ✓ Streaming with web socket


Angular
20

Signal Forms

Ding Dong –
The `ControlValueAcessor` is gone

I Could Tell that…

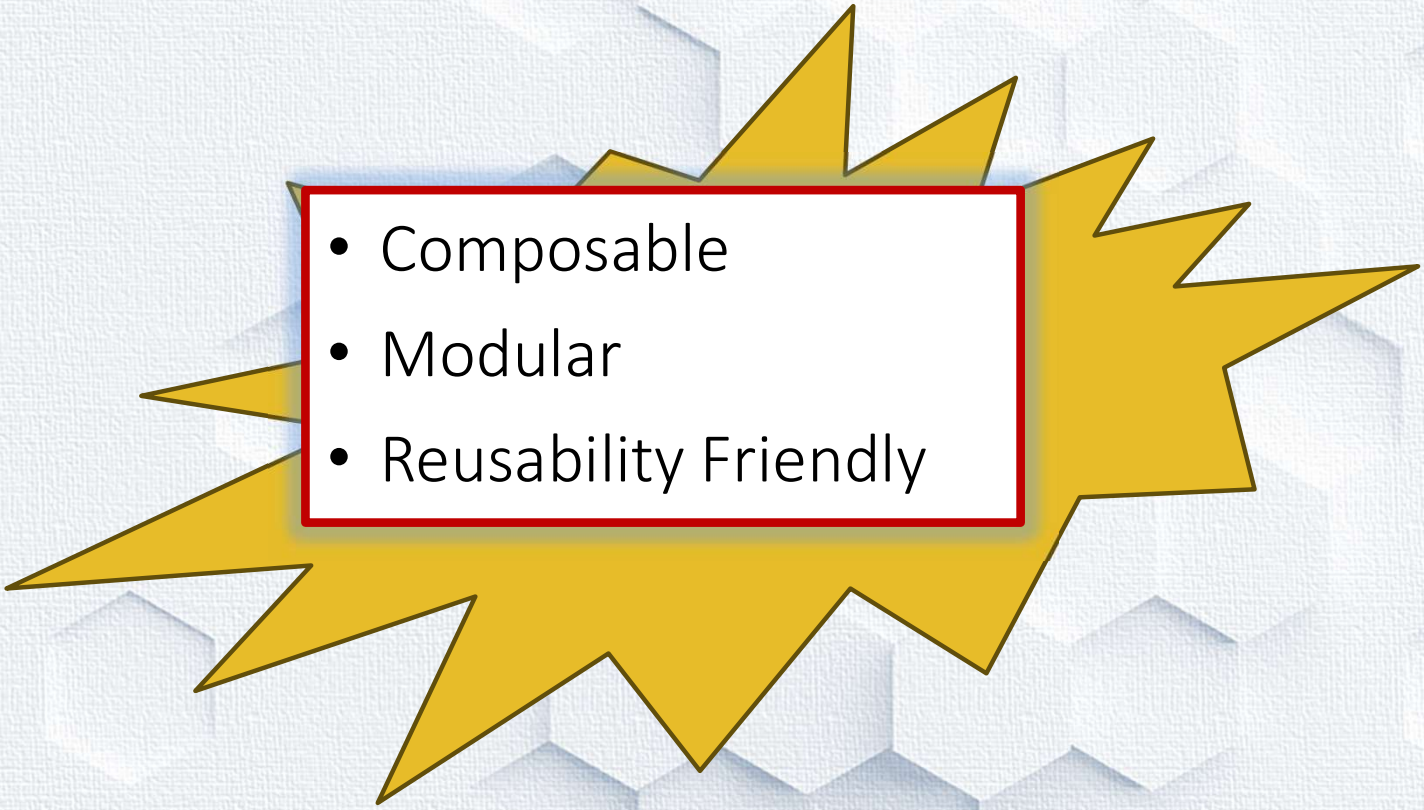
Or that…

Or that…

* Sometimes even less

And of course…

And that its…



Or I can let it just speak for itself

Signal Forms

Demo – Signal Forms

1. The Simplest form ever

Your shortest route to a working form

Demo – Signal Forms

2. Basic Validation

Adding `required` and `minLength` to fields

Demo – Signal Forms

3. Presenting field state

Connecting error state to CSS.

Writing reusable directive to help us

Demo – Signal Forms

4. Presenting Errors

Setting Error messages in the Schema

Demo – Signal Forms

5. Reactive Validation and errors

Setting dynamic thresholds and watching the effect on errors and error messages

Demo – Signal Forms

6. Validators that affect the UI

What happens when you set validators to a “Range” slider

And what happens when the thresholds are dynamic

Demo – Signal Forms

7. Custom Controls

This is where the ControlValueAccessor alternative comes in

Demo – Signal Forms

8. Custom Control with Validation UI

When our control **also** receives validation metadata

“auto-magically”

Demo – Signal Forms

9. Reusability in Schema

How to group schema rules into a reusable function

Demo – Signal Forms

10. Custom Validation - inline

How to create our own custom validation rule

Demo – Signal Forms

11. Reusable Custom Validator

Making out validation rule reusable

Demo – Signal Forms

12. Reusable Field Wrapper

Encapsulating repeating UI features such as labels, validation indicators, hints, and errors – all into a reusable component

Demo – Signal Forms

13. Understanding Metadata in Fields

Using the metadata field to display “required” indicator and “min-max” hints

Demo – Signal Forms

14. Our own custom metadata key

Creating our own metadata key for labels, and a util function that sets it

Demo – Signal Forms

15. Metadata for Custom Validators

Create metadata for our **`minWords`** validator and displaying it in a hint

Demo – Signal Forms

16. Cross Field Validation

How to test more than one field in validation

How to set error on more than one field in validation

Demo – Signal Forms

17. Hidden Fields

How to mark fields as “non relevant” conditionally,

How to remove them from the UI

How to ignore them in validation

Demo – Signal Forms

18. Form Arrays

Adding and removing array items

Array validation

Demo – Signal Forms

19. Form Submit logic and server errors

How to present “in progress” when submitting

And how to present server errors just like client errors

Demo – Signal Forms

20. Async validation using resources

How to take advantage of the resources API to create asynchronous validation.

Demo – Signal Forms

21. Debouncing validation

How to debounce field changes so that validation occurs only after a minimal amount of idle time