Angular Signals

ANJU MUNOTH

Reactivity

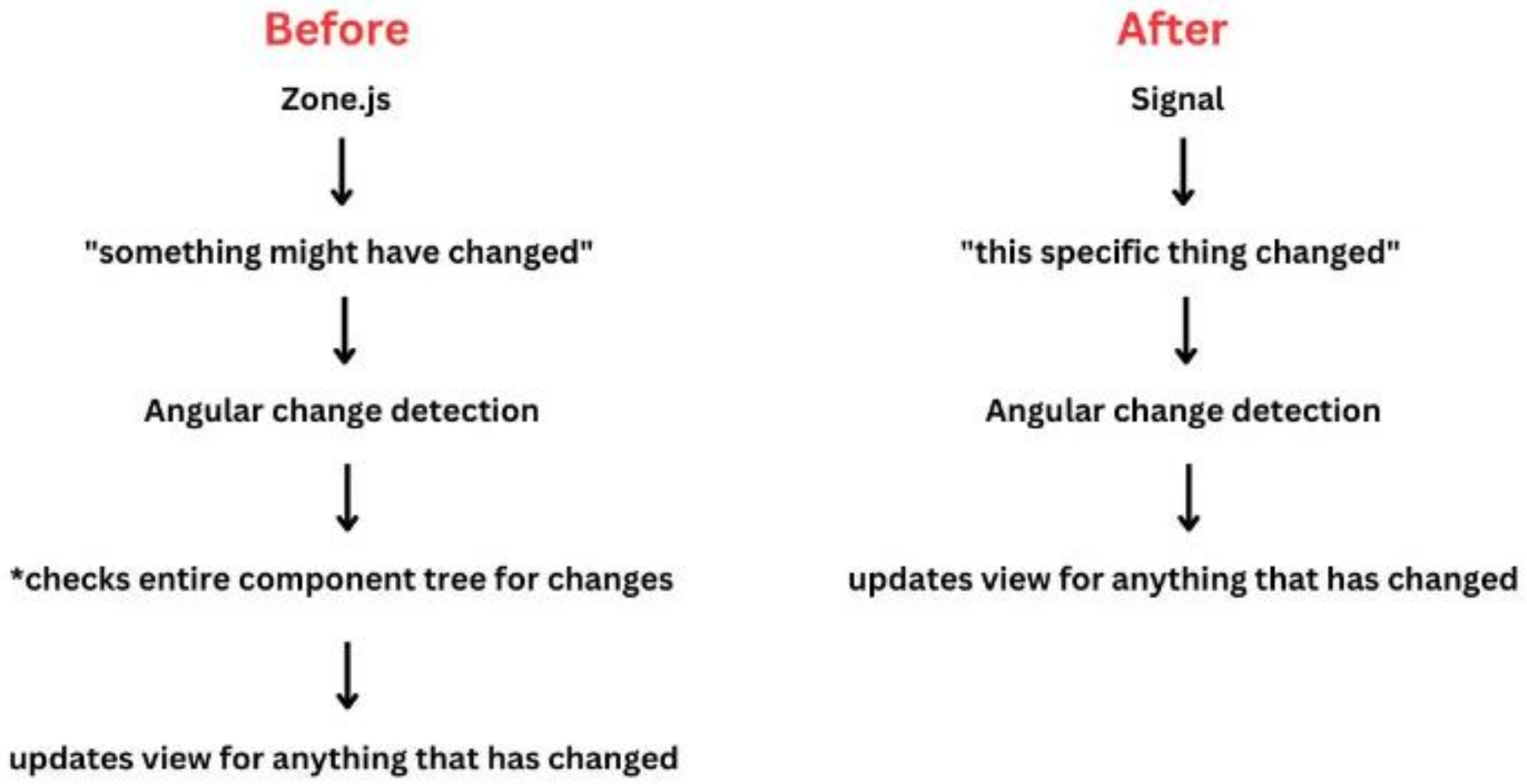
- Mechanism of the framework to keep the application's data model and the UI in sync.
- Ability to update a component's displayed template with information and changes from the component's class or services.
- Updates can happen either synchronously or asynchronously, which can make it challenging to know when a re-render is needed.
- Require a highly efficient mechanism that promptly synchronizes the UI with the application state, guaranteeing a seamless user experience by providing instant updates.

Zone

- Angular's reactivity relies on zone.js, which can only notify us when something might have happened in the application.
- Lacks the ability to provide detailed information about **what exactly happened, what has changed or what is the impact of the change.**
- Does not provide "fine-grained" information about changes in the application state.
- Angular knows only that something happened, doesn't know what, doesn't know where and most certainly doesn't know what parts of the view need to be updated (if any).

What happens when Angular is notified about a potential change?

- Checks all bindings in all views (components) by running Change Detection on every view.
- Process is not performant at all.
- Slightly improved when using components registered with the OnPush strategy, as some parts of the view tree won't be checked.
- Most components will still undergo the Change Detection process.
- In development mode, Change Detection is performed twice to check for any potential side-effects and report on them.
- Not the most performant solution, and quite heavy in terms of the mental model behind it.
- **Possible solution to that could be to invent a new reactive primitive.**


Before
Zone.js
"something might have changed"
Angular change detection
*checks entire component tree for changes
updates view for anything that has changed
After
Signal
"this specific thing changed"
Angular change detection
updates view for anything that has changed

New primitive

To grow and build better functioning, lighter framework we would need such abilities:

- Synchronizing only the parts of the UI that actually need to be updated, even at or below the granularity of individual components
- Writing fully zoneless applications, eliminating the overhead, pitfalls, and quirks of zone.js
- Reducing common pitfalls that often lead to poor change detection performance
- Avoiding common pain points such as ExpressionChangedAfterItHasBeenChecked errors
- Simplifying lifecycle hooks and component queries, making them easier to work with

By removing zone.js

- Clear the overhead of 100kB from the main bundle size (This is quite significant—the complete JavaScript bundle for the lessons module is 74 kB.)
- Enable Angular engine to support async and await natively, eliminating the need to convert them to Promises under the hood to support the poor zone.js monkey-patching mechanism

Angular Signals

- By providing fine-grained knowledge about dependencies and change impact, signals empowers Angular to confidently update data and refresh specific views, ensuring optimal amount of recalculations.
- Angular Signals is a system that granularly tracks how and where your state is used throughout an application, allowing the framework to optimize rendering updates.

Signals

- A signal is a wrapper around a value that can notify interested consumers when that value changes.
- Signals can contain any value, from simple primitives to complex data structures.
- A signal's value is always read through a getter function, which allows Angular to track where the signal is used.
- Signals may be either *writable* or *read-only*.

Signal

- Here is how the base SignalNode type is defined:

```
export interface SignalNode<T> extends ReactiveNode {
value: T;
equal: ValueEqualityFn<T>;
readonly[SIGNAL]: SignalNode<T>;
}
```

- signal always have a value!
- there is an equality function
- every signal has a unique identifier based on their symbol called SIGNAL, which Angular uses to recognize it
- ReactiveNode --core of dependency tracking mechanism. A node can work as a producer and/or as consumer which participates in the reactive graph of signals.

Types of Signals

Writable Signals

- Signals allow you to directly update their values using the mutation API.
- To update application state stored in signals, you can modify one or more writable signals.

Computed Signals

- With computed signals, you can derive their values from the values of other signals.
- The computation function associated with a computed signal should be side-effect free, meaning it only accesses the values of dependent signals and avoids any mutation operations.
- Computed signals provide a powerful way to derive values without directly changing them.

Writable signals

- Writable signals provide an API for updating their values directly.
- Create writable signals by calling the signal function with the signal's initial value:

```
const count = signal(0);
console.log('The count is: ' + count());
```

- // Signals are getter functions - calling them reads their value.

Writable signals

- To change the value of a writable signal, you can either .set() it directly:

count.set(3);

- or use the .update() operation to compute a new value from the previous one:
- // Increment the count by 1.

count.update(value => value + 1);

- Writable signals have the type WritableSignal.

Computed signals

- A computed signal derives its value from other signals. Define one using computed and specifying a derivation function:

```
const count: WritableSignal<number> = signal(0);
const doubleCount: Signal<number> = computed(() => count() * 2);
```

- The doubleCount signal depends on count.
- Whenever count updates, Angular knows that anything which depends on either count or doubleCount needs to update as well.

Computed signals

- Computed signals are both lazily evaluated and memoized
- doubleCount's derivation function does not run to calculate its value until the first time doubleCount is read.
- Once calculated, this value is cached, and future reads of doubleCount will return the cached value without recalculating.
- When count changes, it tells doubleCount that its cached value is no longer valid, and the value is only recalculated on the next read of doubleCount.
- As a result, it's safe to perform computationally expensive derivations in computed signals, such as filtering arrays.

Computed signals

```
// Create two signals: price and quantity
const price = signal(10);

const quantity = signal(5);

// Create a computed signal for total cost based on price and quantity
const totalCost = computed(() => price() * quantity());

console.log(totalCost()); // Output: 50
```

Computed signals

- **Computed signals are great for handling complicated calculations**.
- For instance, if we have a signal representing an array of products, we can easily use them to calculate the total value of all the products.

```
// Create a signal for the array of products
products = signal([
{ name: 'Product A', price: 10 },
{ name: 'Product B', price: 15 },
{ name: 'Product C', price: 20 },
]);
// Create a computed signal for total value based on products
totalValue = computed(() =>
this.products().reduce((sum, product) => sum + product.price, 0)
);
//reading the signal
console.log("computed2",this.totalValue()); // Output: 45
```

Computed signals

- Computed signals are not writable signals
- You cannot directly assign values to a computed signal. That is,

doubleCount.set(3);

- produces a compilation error, because doubleCount is not a WritableSignal.

Computed signals

- Computed signal dependencies are dynamic
- Only the signals actually read during the derivation are tracked. For example, in this computed the count signal is only read conditionally
- When reading conditionalCount, if showCount is false the "Nothing to see here!" message is returned without reading the count signal. This means that updates to count will not result in a recomputation.
- If showCount is later set to true and conditionalCount is read again, the derivation will re-execute and take the branch where showCount is true, returning the message which shows the value of count. Changes to count will then invalidate conditionalCount's cached value.
- Note that dependencies can be removed as well as added. If showCount is later set to false again, then count will no longer be considered a dependency of conditionalCount.

Computed signals

```
const showCount = signal(false);
const count = signal(0);
const conditionalCount = computed(() => {
  if (showCount()) {
    return `The count is ${count()}.`;
  } else {
    return 'Nothing to see here!';
  }
});
```

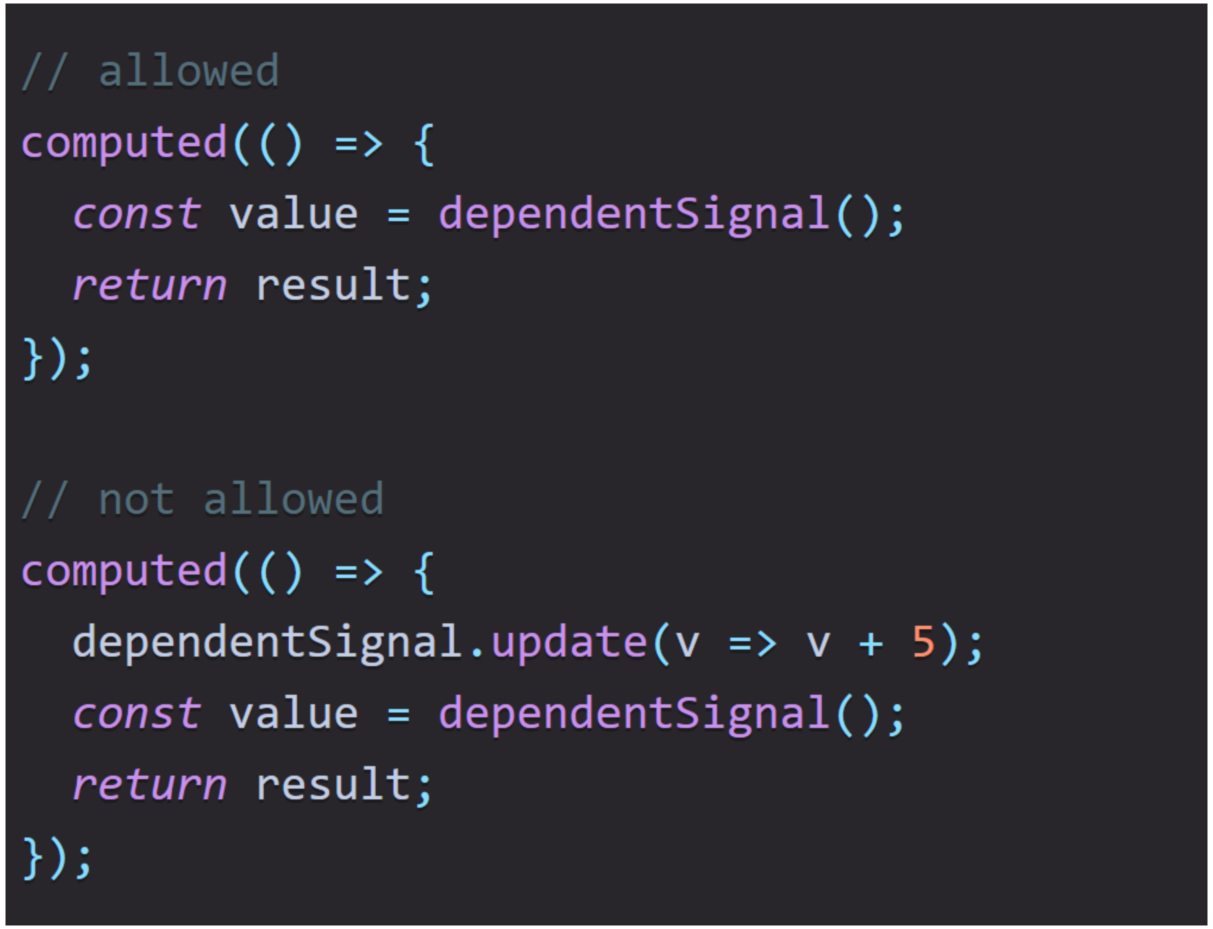
Why signals

- Synchronous access to signal values - This ensures real-time and reliable data source.
- Simplified components lifecycle hooks - Although this feature is not available yet, with signal-based components, fewer lifecycle hooks will be necessary. This simplifies the handling of state changes and updates since we have precise knowledge of when the UI will be updated, allowing us to directly react to those changes.
- Reading signals cannot trigger side effects - This separation between data access and side effect execution ensures a clear, and readable implementation.
- Glitch-free states and no inconsistent states - New primitive provides a reliable and consistent user experience without any desynchronization between state and UI, which was happening before. The famous ExpressionChangedAfterItHasBeenChecked error will be a thing of the past.
- Signals can be used not only in components but also in other parts of your application like services - This allows for a flexible and consistent reactivity architecture.
- Signals provides seamless integration with other reactivity systems like RxJS - This ensures the best developer experience without forcing any breaking changes with the current architecture design and team practices.

Computed Details

- Some important details to keep in mind when working with computed signals.
- **Side-effect free computation**
- The computation function should be side-effect free, meaning it should only access values of dependent signals (or other values involved in the computation) and avoid any updates.
- Attempting to write to other signals from within a computed function will result in an error.
- Computed signals are a handy way to deal with calculated values.
- Work efficiently by skipping unnecessary calculations and remembering results for speed.
- Using computed signals keeps our app responsive and efficient, especially for complex tasks like computations, data filtering, or generating values from other signals.

Side-effect free computation



```
// allowed
computed(() => {
  const value = dependentSignal();
  return result;
});

// not allowed
computed(() => {
  dependentSignal.update(v => v + 5);
  const value = dependentSignal();
  return result;
});
```

Lazy computations

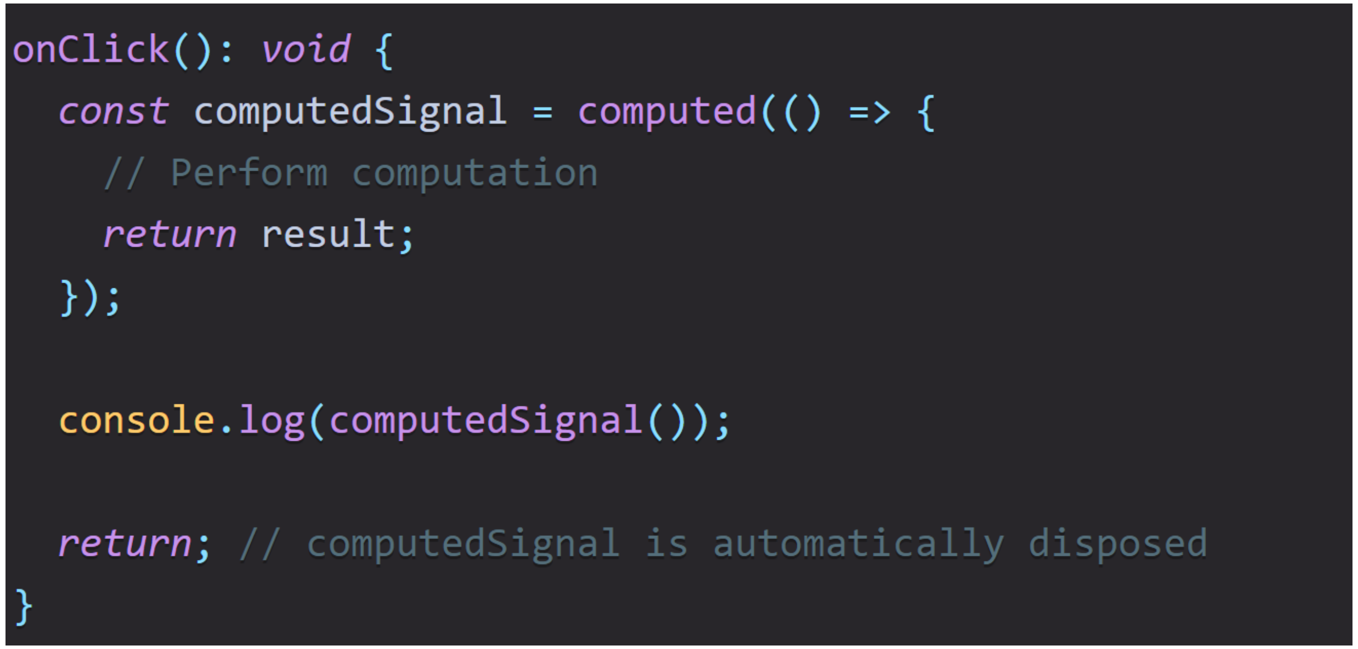
- Computations are lazy, which means that the computation function is only invoked when someone is interested in (reads) its value.
- Helps optimize performance by avoiding unnecessary computations.

```
const computedSignal = computed(() => {
  return dependentSignal();
});

dependentSignal.set(0); // computedSignal won't trigger/read anything
dependentSignal.set(1); // computedSignal won't trigger/read anything
dependentSignal.set(2); // computedSignal won't trigger/read anything
const value = computedSignal(); // value: 2
```

Automatic Disposal

- Computations are automatically disposed of when the computed signal reference goes out of scope.
- Ensures that resources are freed up and no explicit cleanup operations are required.



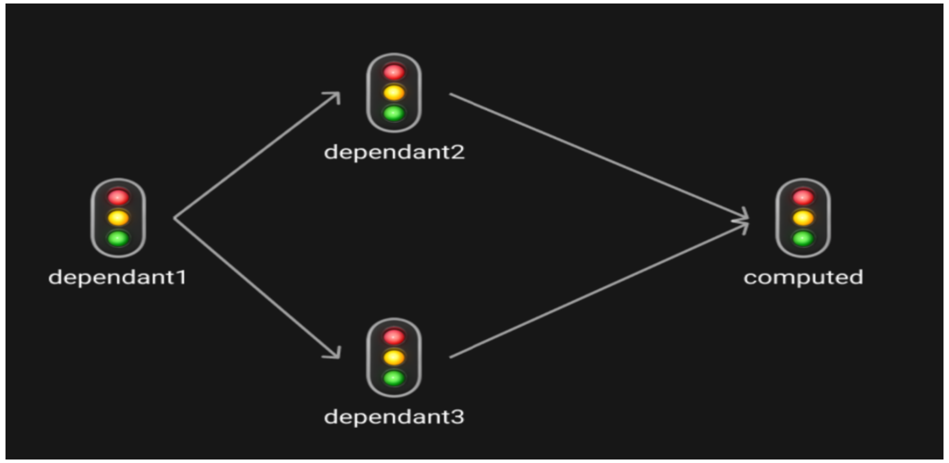
```
onClick(): void {
  const computedSignal = computed(() => {
    // Perform computation
    return result;
  });

  console.log(computedSignal());

  return; // computedSignal is automatically disposed
}
```

Glitch-Free computations

- Computations are glitch-free --Executed a minimal number of times in response to changes in dependencies.
- Computation never executes with stale or intermediate dependency values.
- Computed signals are immune to the famous "diamond dependency problem" further ensuring reliable and accurate computations



Glitch-Free computations

```
const dependentSignal1 = signal(0);

const dependentSignal2 = computed(() => dependentSignal1() + 10);
const dependentSignal3 = computed(() => dependentSignal1() * 10);

const computedSignal = computed(() => {
  const value1 = dependentSignal2();
  const value2 = dependentSignal3();
  return value1 + value2;
});

// changes to dependent signals trigger the computation only once
dependentSignal1.set(5);
computedSignal(); // 15 + 50 = 65
```

Effects

- An effect is used when we want to consume signals and react to them by invoking some side effect, rather than producing a new signal like in computed signals.
- Effect function registers itself as a consumer of the dependent signals used inside its function.
- Whenever any of those signals change, the effect is automatically scheduled to be re-run.
- Not allowed to write anything to other signals.
- Effects are always active and listen for changes in their signal dependencies. Effects are destroyed (cleaned up in memory) when its Destroy context is triggered.
- **Effects should be used for running side effects in response to something happening in signals. Therefore, they act as a consumer node rather than a producer.**

Effects

- Effects are operations that are triggered when the signals they depend on are changed.
- Allow us to perform side effects in a controlled manner, ensuring that they run only when necessary

Effects

- An effect is an operation that runs whenever one or more signal values change.
- Can create an effect with the effect function:

```
effect(() => {
  console.log(`The current count is: ${count()}`);
});
```

- Effects always run at least once.
- When an effect runs, it tracks any signal value reads.
- Whenever any of these signal values change, the effect runs again.
- Similar to computed signals, effects keep track of their dependencies dynamically, and only track signals which were read in the most recent execution.
- **Effects always execute asynchronously, during the change detection process.**

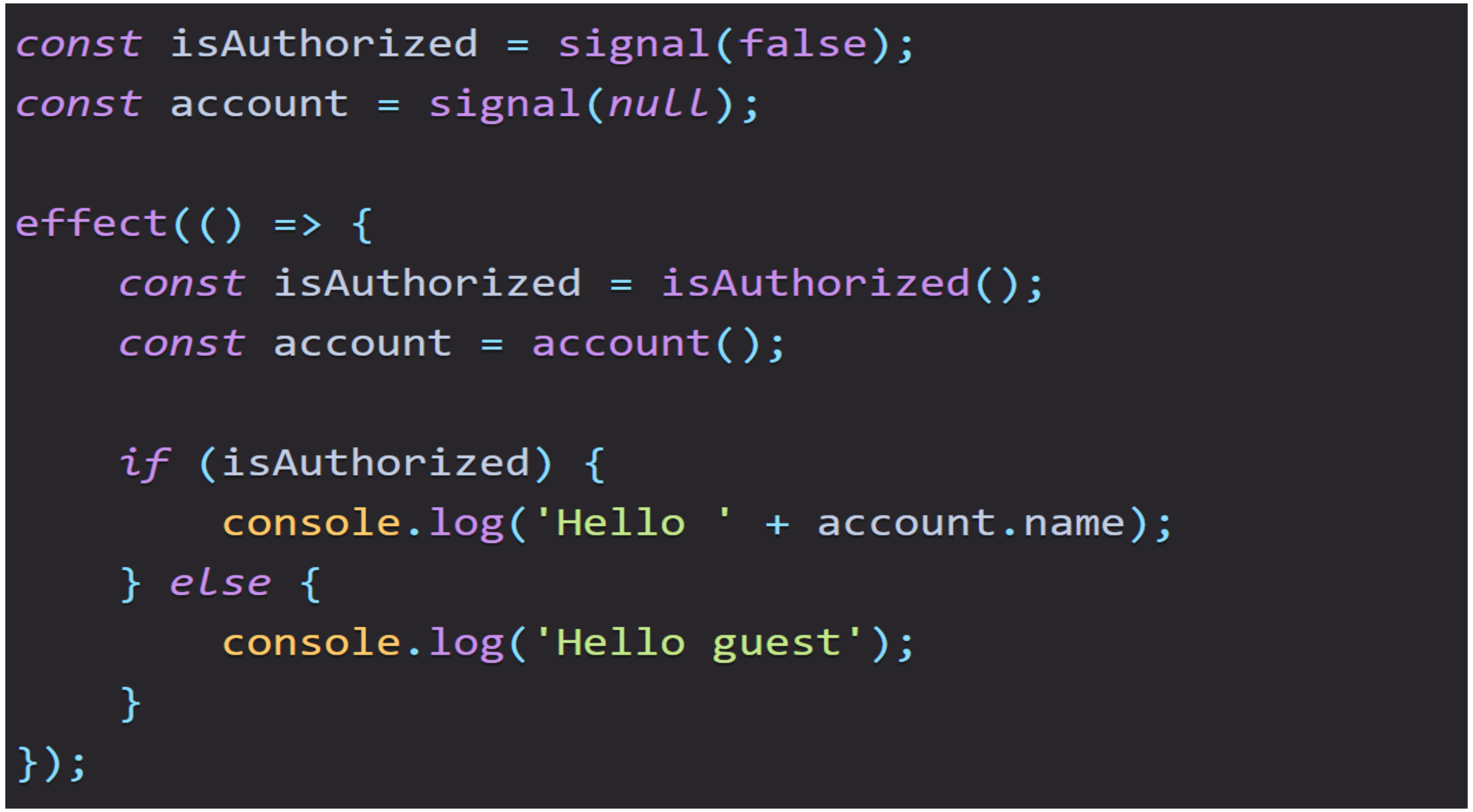
Timing

- Effects in Angular Signals must always be executed after the operation of changing a signal has completed.
- Please note that the exact timing of when effects are executed can vary depending on different scenarios.
- As a result, it is not recommended for application developers to rely on specific execution timings.

However, the following assurances can be provided:

- Effects will execute at least once
- Effects will execute in response to changes in their dependencies at some point in the future
- Effects will execute the minimal number of times. If an effect depends on multiple signals and several of them change simultaneously, only one execution of the effect will be scheduled

Effects



```
const isAuthorized = signal(false);
const account = signal(null);

effect(() => {
    const isAuthorized = isAuthorized();
    const account = account();

    if (isAuthorized) {
        console.log('Hello ' + account.name);
    } else {
        console.log('Hello guest');
    }
});
```

Effects

```
// Create a signal for user authentication status
isAuthenticated = signal(false);

// Create an effect to perform actions based on authentication status
effect(() => {
if (this.isAuthenticated()) {
console.log('User is authenticated. Redirecting to dashboard...');
// Code to redirect to the dashboard can be added here
} else {
console.log('User is not authenticated. Redirecting to login page...');
// Code to redirect to the login page can be added here
}
});

// Simulate authentication status change
this.isAuthenticated.set(true);
```

Use cases for effects

- Logging data being displayed and when it changes, either for analytics or as a debugging tool
- Keeping data in sync with window.localStorage
- Adding custom DOM behavior that can't be expressed with template syntax
- Performing custom rendering to a <canvas>, charting library, or other third party UI library

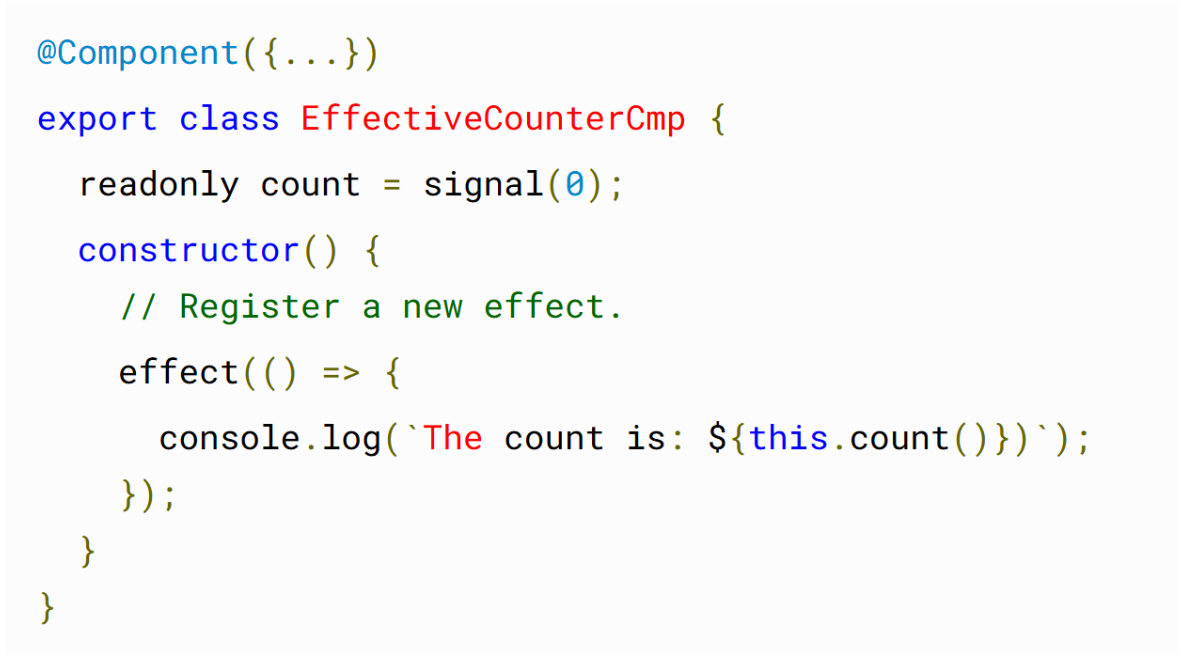
When not to use effects

- Avoid using effects for propagation of state changes. This can result in ExpressionChangedAfterItHasBeenChecked errors, infinite circular updates, or unnecessary change detection cycles.

- Because of these risks, setting signals is disallowed by default in effects, but can be enabled if absolutely necessary.

Injection context

- By default, registering a new effect with the effect() function requires an injection context (access to the inject function).
- Easiest way to provide this is to call effect within a component, directive, or service **constructor**



```
@Component({...})
export class EffectiveCounterCmp {
  readonly count = signal(0);
  constructor() {
    // Register a new effect.
    effect(() => {
      console.log(`The count is: ${this.count()})`);
    });
  }
}
```

Injection context

- To create an effect outside of the constructor, you can pass an Injector to effect via its options:

```
@Component({...})
export class EffectiveCounterCmp {
  readonly count = signal(0);
  constructor(private injector: Injector) {}

  initializeLogging(): void {
    effect(() => {
      console.log(`The count is: ${this.count()})`);
    }, {injector: this.injector});
  }
}
```

Destroying effects

- When you create an effect, it is automatically destroyed when its enclosing context is destroyed. T
- Effects created within components are destroyed when the component is destroyed.
- Same goes for effects within directives, services, etc.
- Effects return an EffectRef that can be used to destroy them manually, via the .destroy() operation.
- Can also be combined with the manualCleanup option to create an effect that lasts until it is manually destroyed.
- Be careful to actually clean up such effects when they're no longer required.

Effect cleanup functions

- Effects might start long-running operations, which should be cancelled if the effect is destroyed or runs again before the first operation finished.
- When you create an effect, your function can optionally accept an onCleanup function as its first parameter.
- This onCleanup function lets you register a callback that is invoked before the next run of the effect begins, or when the effect is destroyed.

```
effect((onCleanup) => {
  const user = currentUser();

  const timer = setTimeout(() => {
    console.log(`1 second ago, the user became ${user}`);
  }, 1000);

  onCleanup(() => {
    clearTimeout(timer);
  });
});
```

To summarize : what is a signal?

- A signal is a variable + change notification
- A signal is reactive, and is called a "reactive primitive"
- A signal always has a value
- A signal is synchronous
- A signal is not a replacement for RxJS and Observables for asynchronous operations, such as http.get

Where can we use signals?

- Use them in **components** to track local component state
- Use them in **directives**
- Use them in a **service** to share state across components
- Read them in a **template** to display signal values
- Or use them **anywhere else** in your code

Signals and Observables: unique strengths

- Signals and Observables each have their own strengths and use cases.
- Signals will be particularly useful for enhancing (or even rebuilding in the future) the Change Detection process.
- Allow for lazily calculated computed signals and seamless synchronization of the UI with the application state.
- Observables excel at handling asynchronous events and reacting to streams of values.

Power of using both

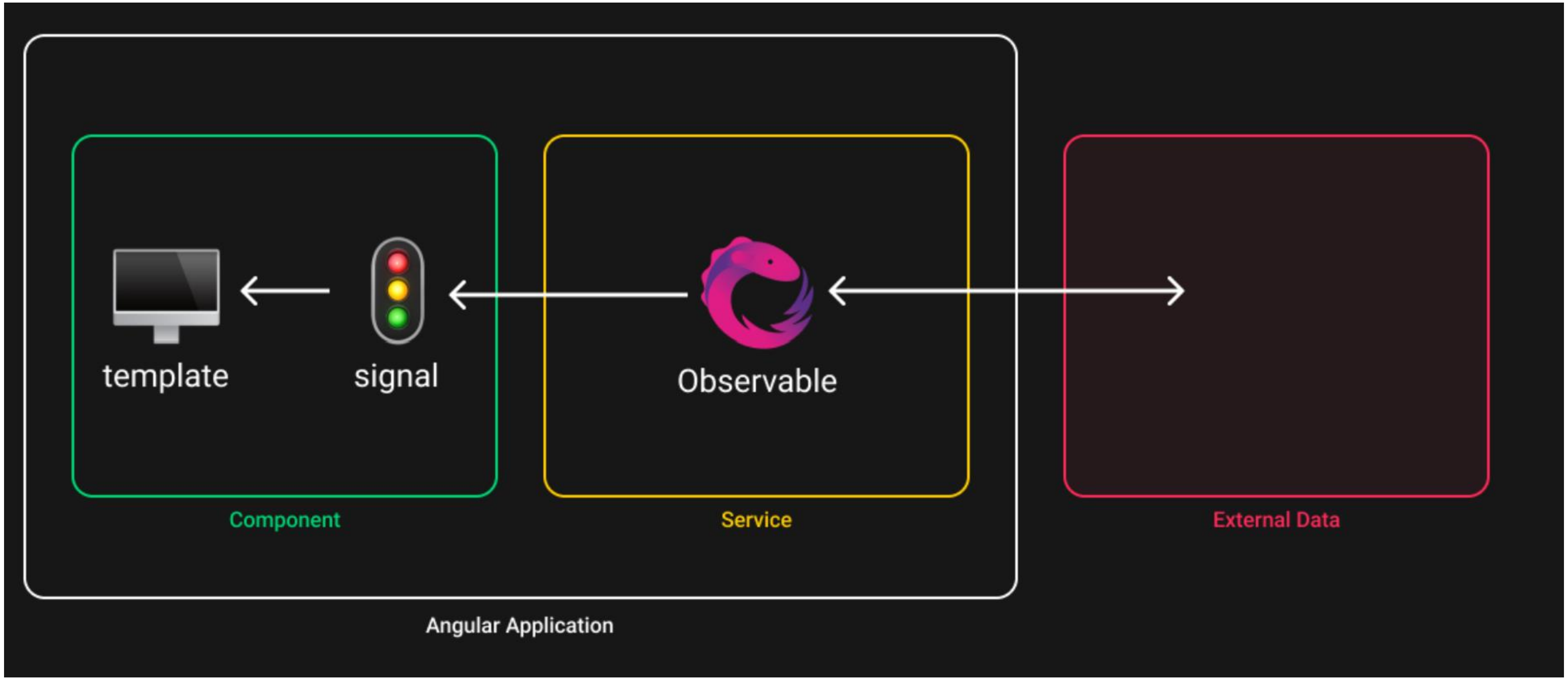
- The best approach is often to use signals and Observables together to create the most effective Angular application.
- Signals are ideal for synchronizing the application state with the UI, ensuring that data in components or services is accurately reflected in the rendered template.
- Significantly improves Change Detection efficiency, as views are refreshed only when necessary.
- Observables, are still essential for example for retrieving the application state from external services.

- HttpClient will continue to emit Observables as it currently does, as they are more efficient at handling asynchronous events compared to signals.

Simplifying data flow

- Using both will likely reduce the need for manual subscriptions.
- Can use Observables and convert them into signals that components can use to render the state.
- Simplifies the communication between components and the UI, making the entire data flow more manageable.

Simplifying data flow

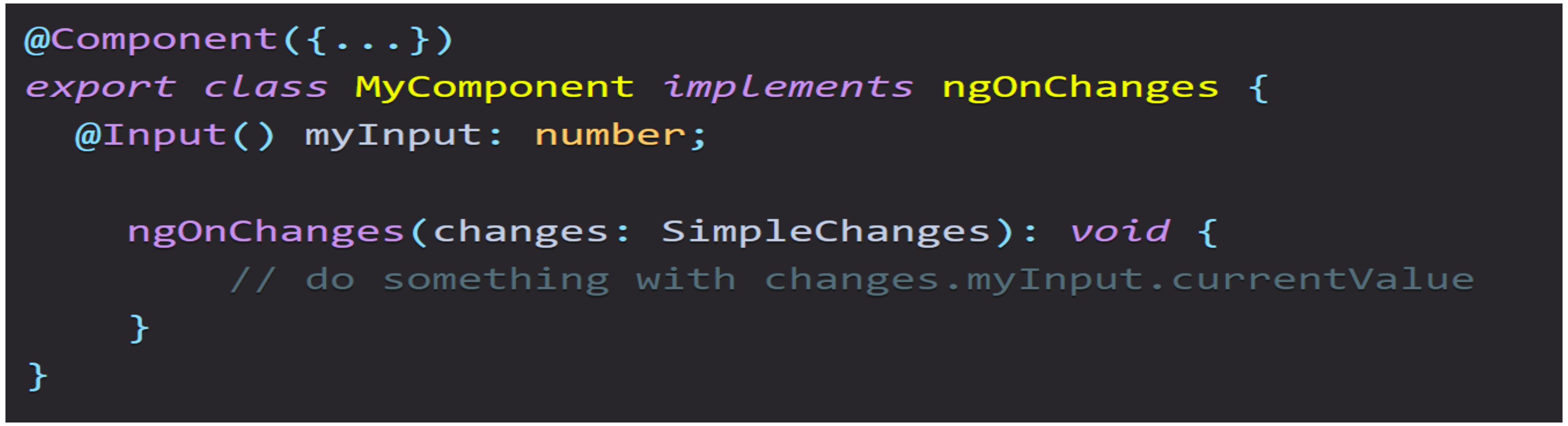


Signal Input

- One of the most popular ways to communicate with the child component - its inputs.
- Values bound to the input often change during the component's lifetime.
- Crucial to be able to react to those changes to process the data or run some side effects.

Before Angular v17.1

- In the past, we could create inputs by using the @Input decorator to highlight any component's property as its input.
- Angular v16 even brought us some cool enhancements, like setting input as required, and the ability to pass the transformation function.
- Reacting to the component's input changes was still a bit clunky.
- Could use ngOnChanges (recommended by Angular)



```
@Component({...})
export class MyComponent implements ngOnChanges {
  @Input() myInput: number;

    ngOnChanges(changes: SimpleChanges): void {
        // do something with changes.myInput.currentValue
    }
}
```

Signals time!

- Introduced in Angular v17.1.0.
- Signals already support reacting on their changes, so by simply making input properties a signal, instead of a plain value, we can solve all of our problems and improve DX.
- Instead of using decorators, we have a neat, concise function that generates a signal input:

```
myInput = input<number>();
```

- Implementation is type-safe, and it returns a non-writable signal.
- In this example, we created an optional input, so at the beginning, it will have a value of undefined.

Required Signal Input

- Need to supply them, while with optional inputs we are never sure if they are truly optional, or if we need to pass them for our component to work properly, but it won't throw any error indicating that.
- Here is how we can declare a required signal input, which will throw an error if not provided:

```
myInput = input.required<number>();
```

- Note that while the read type of the previous input was number | undefined, this one is just number, even though there is no initial value passed when declaring the input.
- The Framework guarantees the input will have a value.
- **input.required cannot have a default value.**

With default value

- Whenever we want to supply the initial value, we can do this by passing a value as an argument to the creation function, like this:

```
myInput = input<number>(123);
```

Aliasing

- Can still alias inputs, so their template attribute names differ from the component property names.
- Useful when we want the consumer to use different naming than our internal implementation within the component class.

```
myInput = input<number>(0, { alias: 'value' });
```

- Then from the consumer perspective we can use it like this:

```
<child [value]="123" />
```

Transformation

- Can use transform functions to automatically process incoming value to the input, and store it in the desired format.
- For example, let's say we have an input like this:

myInput = input<string | number>();

- Right now its full type would be:

InputSignal<string | number, string | number>

- Meaning it receives a string or a number, and it also returns a string or a number.

Transformation

- However, we might want to always operate on a string and convert numbers to string. We can do this by creating a transform function:

const toString = (value: string | number) => `${value}`

- Then, we can apply that to our input:

myInput = input<string, string | number>(0, {transform: toString});

- The full type will be then:

InputSignal<string, string | number>

- Meaning that however we accept a string or a number, at the end we will always have a string stored in this input property.

Reactive inputs

- Because Inputs are now implemented as signals, we can benefit from that by reacting to their changes declaratively.
- No longer need input setters, nor the ngOnChanges. The input’s nature is now reactive and we can use that!

- Derive values - we can easily take advantage of computed signals to transform our input data.
- Side effects - accordingly we can also use effects to react on changes and run some side effects.

Why should we use signal inputs and not @Input()?

- Signal inputs are a reactive alternative to decorator-based @Input().

In comparison to decorator-based @Input, signal inputs provide numerous benefits:

- Signal inputs are more type safe:
- Required inputs do not require initial values, or tricks to tell TypeScript that an input always has a value.
- Transforms are automatically checked to match the accepted input values.
- Signal inputs, when used in templates, will automatically mark OnPush components as dirty.
- Values can be easily derived whenever an input changes using computed.
- Easier and more local monitoring of inputs using effect instead of ngOnChanges or setters.

Angular component outputs with output()

- The output() API is a direct replacement for the traditional @Output() decorator.
- Angular has added output() as a new way to define component outputs in Angular, in a way that is more type-safe and better integrated with RxJs than the traditional @Output and EventEmitter approach.
- output function returns an OutputEmitterRef.

```
import { Component, output } from "@angular/core";

@Component({
  selector: "book",
  standalone: true,
  template: `<div class="book-card">
    <b>{{ book()?.title }}</b>
    <div>{{ book()?.synopsis }}</div>
    <button (click)="onDelete()">Delete Book</button>
  </div>`,
})
class BookComponent {
  deleteBook = output<Book>();

  onDelete() {
    this.deleteBook.emit({
      title: "Angular Deep Dive",
      synopsis: "A deep dive into Angular core concepts",
    });
  }
}
```

```
<book (deleteBook)="deleteBookEvent($event)" />
```

```
deleteBookEvent(book: Book) {
 console.log(book);
}
```

Signal Model

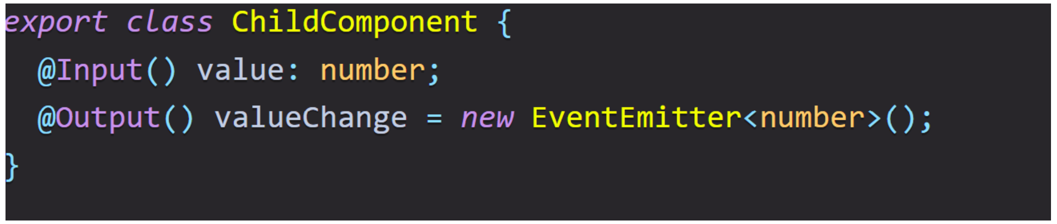
- Not only do we need to receive data from the parent, we need to be able to update it. This way, everything stays in sync between the parent and child components.

Pre-Angular v17.2 Era

- Once upon a time, before signals came into our lives, achieving two-way data binding involved combining input and output in the child component.
- This way, we could maintain the current value while emitting any updates to it. The parent component could then use this to keep everything in sync.
- Thanks to the "banana in the box" syntax, parents could bind both input and output to a single variable like this:

<child [(value)]="someProperty" />

- Angular does all the heavy lifting, keeping the binding up to date.
- So, how did this magic happen? Well, the child component needed to implement input and output following a certain convention: the output property name had to be inputChange, where input is the name of the input property.



```
export class ChildComponent {
  @Input() value: number;
  @Output() valueChange = new EventEmitter<number>();
}
```

Signal Model

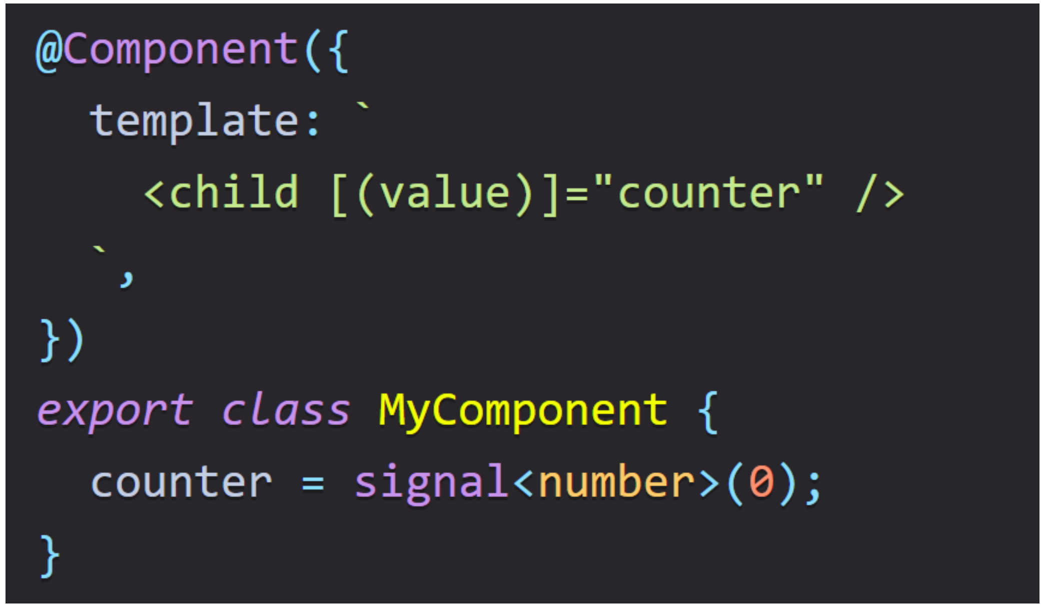
- Introduced in Angular v17.2.0.

- With the introduction of the signal model, Angular allowed us to merge pairs of input and output into a single model.
- This makes a lot of sense when you think about it. If our main goal is to sync everything up, it's much easier with just one property to work with.

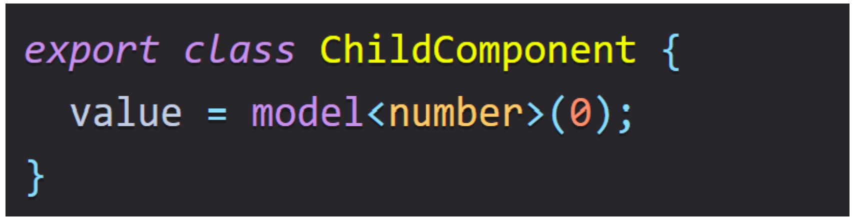
- Syntax is really similar to that of the input. Angular gives us a new function, model.
- Model signals need to be writable signals, so while we can pass signals created by the signal function, we can't bind readonly signals like computed ones using two-way data binding.

Model

- The child component receives the initial value of 0 (as specified in the parent's property).
- Whenever the child component updates its model property, Angular will automatically update the parent's counter property.
- And here's the golden part: if the parent decides to update its counter value, it will also be automatically updated in the child model.



```
@Component({
  template: `
    <child [(value)]="counter" />
  `,
})
export class MyComponent {
  counter = signal<number>(0);
}
```



```
export class ChildComponent {
  value = model<number>(0);
}
```

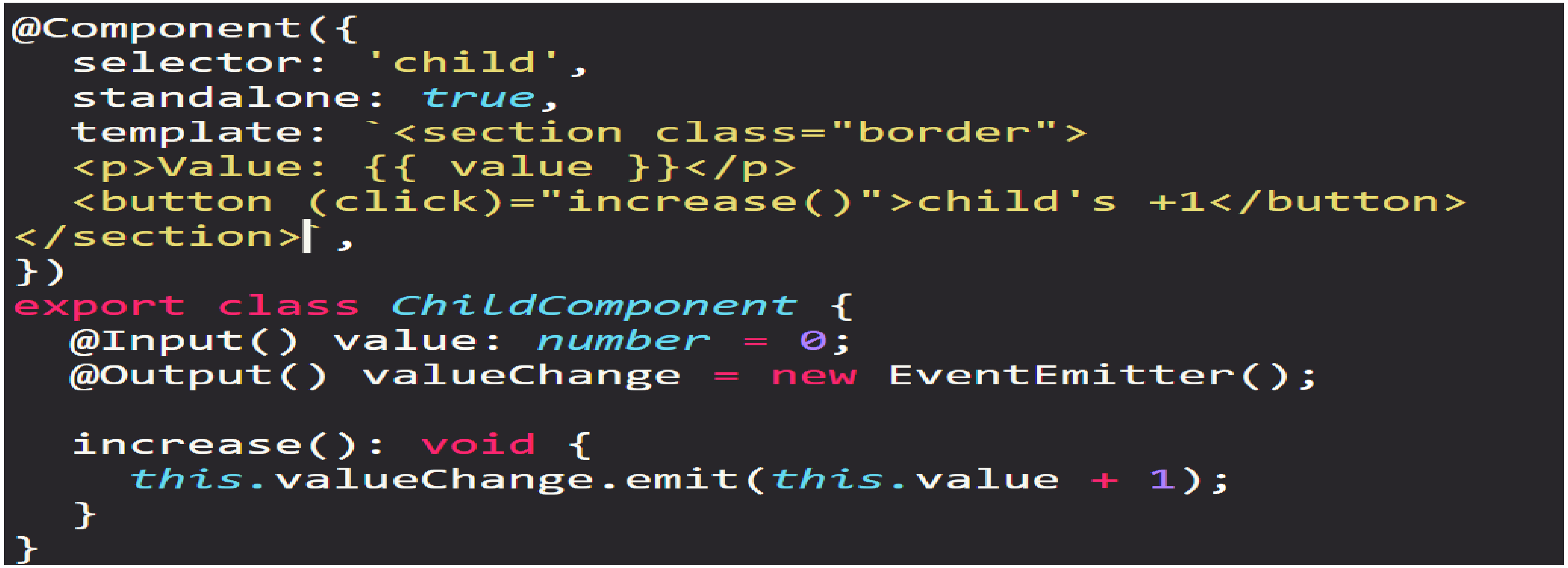
Binding to Non-Signal Values

- In larger applications, it might not be straightforward to switch everything over to signals.
- So, it's important to know how to introduce new components with signals, while still using them in existing non-signal components.
- Even though our ChildComponent uses the signal model, we can still bind it to the parent's non-signal property
- Angular will keep track of this and automatically update bindings, keeping the experience consistent and not breaking existing conventions in your application.

```
@Component({
  template: `
    <child [(value)]="counter" />
  `,
})
export class MyComponent {
  counter = 0;
}
```

Non-signal child

- Possible to implement a new signal-based parent component while still using an old non-signal child.



```
@Component({
  selector: 'child',
  standalone: true,
  template: `<section class="border">
  <p>Value: {{ value }}</p>
  <button (click)="increase()">child's +1</button>
</section>`,
})
export class ChildComponent {
  @Input() value: number = 0;
  @Output() valueChange = new EventEmitter();

  increase(): void {
    this.valueChange.emit(this.value + 1);
  }
}
```

Aliases and Required

- Just like with inputs, we can alias models, so their attribute names are different from property names.

```
myModel = model<number>(0, { alias: 'value' });
```

- Then from the user's perspective, we can use it like this, even though the model name is myModel:

```
<child [(value)]="property" />
```

- Required -- Is also available with model:

```
value = model.required<number>();
```

- In this case, we can no longer provide an initial value - the value will definitely come from the parent.

Transform

- The last input option is the ability to transform its value, but that's **not possible** with models.
- The only other difference to standard inputs is that the signal exposed via the model function is a writable signal. This is quite obvious, as we need to update the model within the child.

When to use model inputs

- Use model inputs in components that exist to modify a value based on user interaction.
- Custom form controls, such as a date picker or combobox, should use model inputs for their primary value.

- Avoid using model inputs as a convenience to avoid introducing an additional class property for containing local state.

Differences between model() and input()

Both input() and model() functions are ways to define signal-based inputs in Angular, but they differ in a few ways:

- model() defines both an input and an output. The output's name is always the name of the input suffixed with Change to support two-way bindings.
  - It will be up to the consumer of your directive to decide if they want to use just the input, just the output, or both.
- ModelSignal is a WritableSignal which means that its value can be changed from anywhere using the set and update methods.
  - When a new value is assigned, the ModelSignal will emit to its output.
  - This is different from InputSignal which is read-only and can only be changed through the template.
- Model inputs do not support input transforms while signal inputs do.

Benefits of Using Signals

1. **Reactivity:** Signals provide a more reactive way to work with data in Angular. Can easily track changes and react to them without relying solely on Angular's change detection.
2. **Performance:** Signals can lead to better performance in your Angular applications. By tracking changes at a granular level, can minimize unnecessary re-renders and improve the overall efficiency of your app.
3. **Simplified Code:** Signals can simplify your code by reducing the need for complex change detection strategies and event handling. Make your codebase cleaner and easier to maintain.
4. **Independence from Zone.JS:** Signals work independently of Zone.JS, which can be advantageous in scenarios where you want more control over change detection or when working with libraries that don't play well with Zone.JS.
5. **Efficient State Management:** Signals can be used for state management within your Angular components. You can create signals for different pieces of state and easily compose them to derive new state values.
6. **Batching Changes:** Signals provide a built-in mechanism for batching changes, which can be useful when you need to make multiple updates to signals in a single operation, reducing unnecessary change detection cycles.