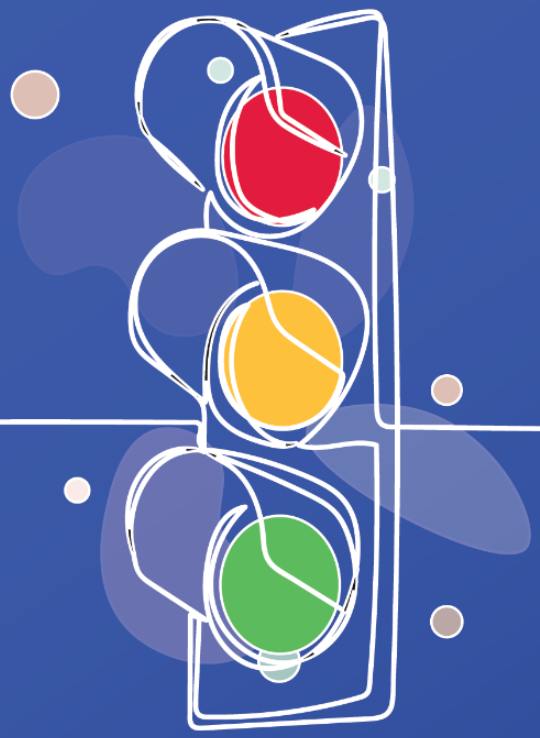


MASTERING ANGULAR SIGNALS

get ready for Angular's future

ANGULAR EXPERTS

KEVIN KREUZER

Table of contents

Acknowledgments **4**
About the author **5**
Introduction **6**
Why Signals? **7**
Change detection - a quick recap 8
Reduce checks with OnPush 11
Limitations of todays change detection 16
Zone.js 17
Zone.js & monkey patching 17
Zone.js drawbacks 19
The need for a new reactive primitive 21
Signals - the Chosen Reactive Primitive **23**
Angular Signals API **25**
Creating a signal 25
Writable Signals 28
Set 28
Update 28
Mutate (Removed) 28
Why was mutate removed? 29
Equality Comparisons 31
Signals are lazy **32**
Lazy execution 32
Angular templates and Signals 33
Signals and OnPush **37**
Computed Signals **41**
Creating a computed Signal 41
Equality function 42
Computed Signals inside an Angular template 42
Computed Signals characteristics 45
Laziness 45
Automatic Disposal 47
Glitch-Free Execution 47
Effects **53**

Creating effects 53
Effects and the Injection context 55
runInInjectionContext 57
Stopping Effects 60
Cleanup functions 61
Scheduling and Timing of Effects 62
Writing to Signals from Effects 62
Preventing dependencies with untracked 67
Signals Push -> Poll -> Pull nature 71
Dynamic dependency graph 71
An Algorithm in two phases 73
Polling versions 75
Push -> Poll -> Pull 77
Interaction between writable Signals and computed signal 77
Interaction between writable signal, computed signal and effect 80
Applied Push -> Poll -> Pull algorithm 86
Initial pull 86
Lazy computation 90
Dynamic dependency graph 93
Effect execution 102
Signals and RxJS 105
RxJs drawbacks 105
RxJS interoperability 106
toSignal - Converting Observables to Angular Signals 106
Synchronous Emit 106
Asynchronous Emit 107
Managing the Subscription 107
Error and Completion States 108
toObservable - Converting Angular Signals to Observables 109
Lifecycle and Cleanup 109
Asynchronously delivered values 110
One Effect for All Observers (via shareReplay) 113
Will Signals replace RxJS? 114
Event handling complexity 114
Network Requests and Asynchronicity 114
RxJS investment 115
Conclusion: Signals and RxJS Coexistence 115
Best of both worlds 116

The current state of Angular Signals **121**
Impact on Angular's future **122**
The Rise of Signal-based Components 122
Lifecycle Evolution 122
Local Change Detection 123
Zoneless Applications on the Horizon 123
Coexistence and Interoperability 123
Conclusion 124
Get in touch & feedback **125**
More **126**
Workshops 126
Skol 127
Angular Enterprise Ebook 128
Angular UI component library starter 129
Omniboard 130
Project review 130
Blog & Videos 131
Other services 131

Acknowledgments

This book is a product of collective efforts and would not have been possible without the incredible support and contributions from the Angular community. I am deeply grateful for their dedication and passion for pushing the boundaries of web development.

Special thanks goes to Tomas Trajan for his invaluable long-term collaboration and his extensive research and article on the Poll -> Push -> Pull-based Signals concepts. Tomas's expertise and insights have been instrumental in shaping the ideas presented in this book.

I would also like to express my gratitude to the Angular team for their exceptional work and for introducing Angular Signals. Their innovation and commitment to enhancing the Angular ecosystem have been a constant source of inspiration.

Furthermore, I extend my appreciation to all the Google Developer Experts (GDE) and the wider community for their invaluable feedback on the Signals RFC. Your input and suggestions have helped refine the concepts discussed here.

I am humbled and honored to be a part of this vibrant and supportive community, and we hope this book serves as a valuable resource to everyone on their journey of Angular exploration.

Thank you all for your support and encouragement.

Kevin

About the author

Hey there! I'm Kevin, and I'm a passionate frontend engineer. You might call me a trainer, consultant, and senior front-end engineer, but really, I'm just a regular guy who's super into the modern web. I've got this Google Developer Expert title for all things Angular & Web, which is pretty cool.

My story in tech has been a wild ride. I've seen web technologies go through some crazy changes, and I've been right in the thick of it. I've had the chance to work with big companies, helping them build, maintain, and upgrade their web apps and core tools. It's been a blast!

But you know what gets me going even more than code? Sharing what I know with others. I love teaching and getting folks excited about modern web stuff. I do it on stages, in workshops, through podcasts, videos, and even in good old articles. Oh, and in 2019, I was the busiest Angular In-Depth writer – that was a cool highlight.

Thanks a bunch for grabbing a copy of this book. I put a lot of effort into creating it, and your support means the world to me. But let's no longer dwell on me – who's up for learning Angular Signals?

“If you'd like to stay connected with me and explore more Angular-related content, please consider visiting one of the following links. Your support and engagement are greatly appreciated!”





Introduction

In the ever-evolving landscape of web development, keeping up with the latest updates and releases is crucial for developers striving to build robust and cutting-edge applications.

Angular is no exception to this rule. With its strong community and commitment to constant improvement, Angular ensures regular updates and enhancements. Therefore in the past Angular has introduced a series of groundbreaking features that have propelled web development to new heights. These features have not only boosted developer productivity but have also revolutionized the way applications are built.

One of those remarkable features that will impact and shape the future of Angular applications is Signals.

In this book, we will delve into all the essential aspects of Signals, providing you with a thorough understanding of their significance in Angular. We will explore the Signal type in Angular and examine its API. Additionally, we will take a closer look at the Push -> Poll -> Pull-based architecture of Signals.

Throughout this book, we aim to present the material clearly and engagingly, ensuring that you grasp the intricacies of Signals effectively. By the end of our journey together, you will possess a solid foundation in understanding Signals and their vital role in Angular development.

Why Signals?

Angular is a very mature framework and has been around for quite some time. In the early days of Angular, many design choices had to be made. The design choices were influenced by the available technologies and the state of knowledge at the time.

Now, let's be real—there's no such thing as a perfect solution in software development. It's a world of trade-offs, where decisions are made with a mix of intuition and best practices.

Over time, some of the design decisions made in the early days have started to show their age and new technological possibilities arised. Angular is aware of them and tries to move forward by embracing the new possibilities.

But integrating the latest technologies and improving design decisions made in the past sounds easier than it actually is. Especially if you want to be backwards compatible.

Being backwards compatible often makes it harder to move forward.

Today, Angular powers thousands of applications across the globe. Of course, businesses can't afford to rewrite their entire codebase every time a new version of a framework comes out. Therefore the Angular team puts a lot of effort into backwards compatibility.

But, the framework didn't just sit in its comfort zone; it grew, expanded, and evolved. We got so many revolutionary features such as Ivy, lazy loading, standalone components, to just name a few. The Angular team is all about progress, innovation, and finding ways to improve while keeping things smooth for the businesses that rely on them.

One of Angular's latest innovations to improve a design decision from the past are Signals. Signals serve multiple purposes but mainly aim to improve change detection and simplify state management.

Change detection - a quick recap

To understand why we need Signals, we first have to understand today's problems with Change detection. Let’s take a look.

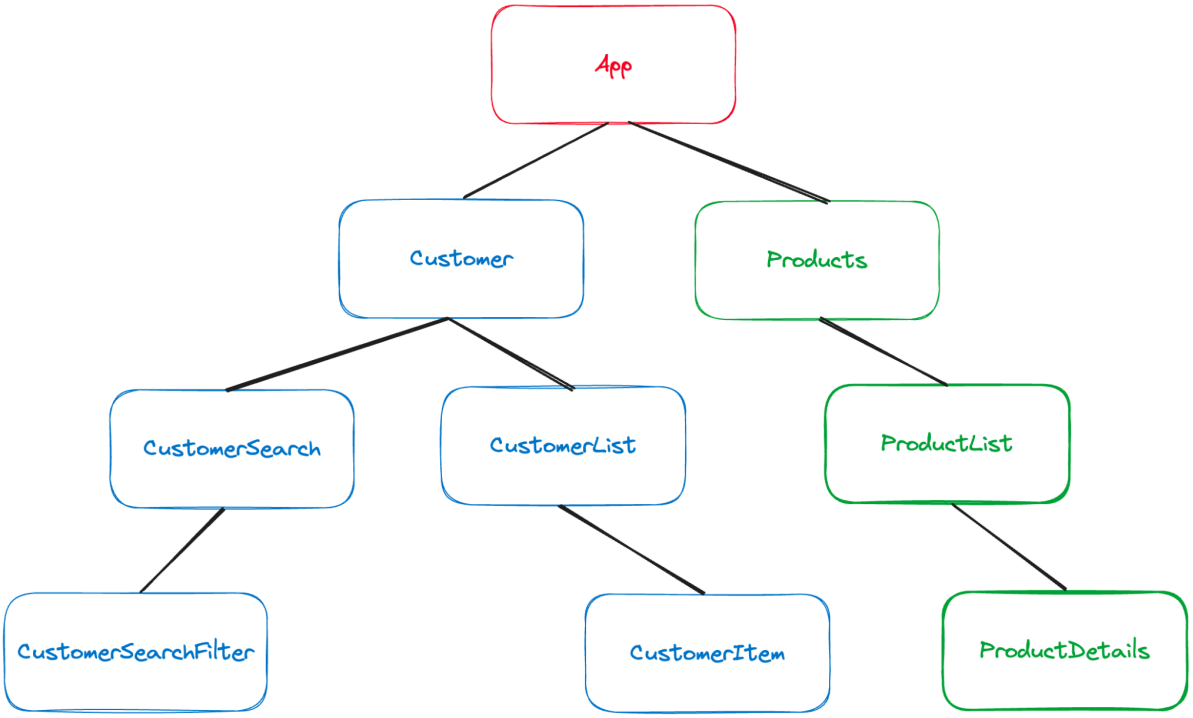
Change detection is a complex subject with many facets that could fill an entire book on its own. However, this book primarily focuses on Signals.

Our aim in this section is to provide you with a fundamental understanding of change detection, without delving into exhaustive detail.

This foundational knowledge will help us identify both the strengths and limitations of change detection, allowing us to explore how Signals can enhance it in the future.

Let's begin by examining the component tree within Angular. Internally, Angular maintains a representation of the application's structure. Although this tree resembles the DOM tree, it’s a different abstraction with a different purpose.

The component tree is a conceptual representation of our application's structure. To illustrate, let's visualize a basic component tree for an application featuring two modules: "customer" and "products."



This tree is an essential part of change detection. Change detection is the process through which Angular monitors and updates the user interface to reflect any alterations in the application's state. It ensures that the displayed content remains consistent with the underlying data.

When an event triggers a change in the application, Angular starts its journey from the root component and traverses down the component tree, checking the previous state against the current state.

What's interesting in this scenario is that Angular lacks information about where a change occurred, necessitating a comprehensive traversal of the entire component tree to pinpoint the source of the change.

Change detection is always performed from top to bottom starting at the root component.

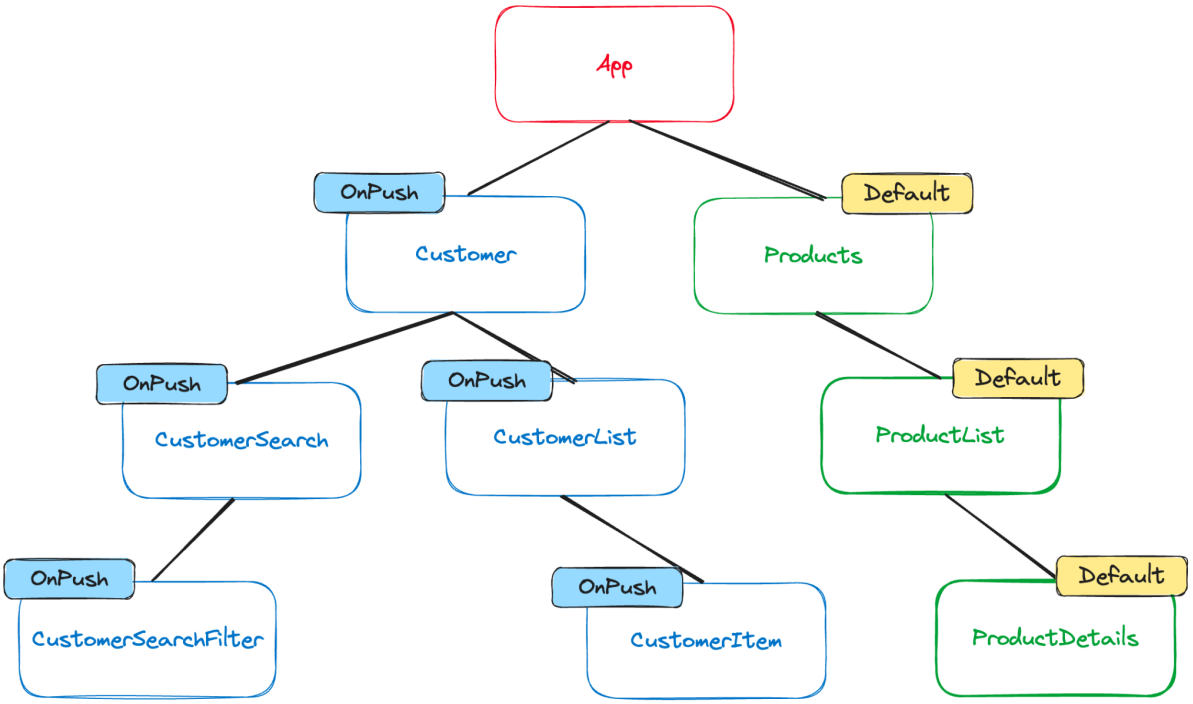
To detect a change, Angular uses a mechanism that is often referred to as “dirty-checking”. Dirty checking is a mechanism by which Angular monitors changes in the application's data and components. Angular compares values from before a change with the values after the change.

By default Angular uses a deep equality check for dirty checking.

This approach works great, but as you might guess there are a lot of checks that Angular has to perform. In bigger applications with huge component trees this can be problematic since it can lead to performance issues.

Therefore additional to the the ***`Default`*** change detection strategy which internally is also referred to as the “check always strategy” the Angular team introduced a new change detection strategy named ***OnPush***.

The change detection strategy can be set per component. Means, we can mix the ***Default*** and the ***OnPush*** strategy in our application.



In the application above all Customer features have been configured with the ***OnPush*** change detection strategy, while the Product features use the

Default change detection strategy. So what is the benefit of using the *OnPush* change detection strategy?

Reduce checks with OnPush

As previously mentioned, during each change detection run, Angular begins its process at the root component, then systematically traverses the entire component tree. For each component encountered along the way, it compares the previous model values with the current model values.

During change detection, Angular performs a lot of checks. This can lead to performance issues in bigger applications.

To improve performance we need a way to reduce the amount of checks needed. And that's where the *OnPush* change detection strategy comes in. With the *OnPush* strategy, components and their subtree are only checked in the following scenarios:

1. **If a component input (*@Inputs*) changes**

If a parent component updates the input properties of a child component, the child component with *OnPush* change detection will be checked for updates.

```
<my-child [value]="value"/>
```

By default Angular uses a deep equality check for dirty checking. When using ***OnPush*** Angular compares values by reference.

Imagine a click handler that alters an object in the following way:

```
value.myProp = 'foo';
```

This would be detected as a change in ***Default*** change detection strategy but not in ***OnPush***.

2. If an event was emitted from within the view

If an event is emitted from within the component, change detection will run. The following code inside the child component would trigger change detection.

```
<button (click)="updateValue()">Update value</button>
```

3. If our component is set to *`markForCheck`*

You can also manually trigger change detection for a component using the ***`ChangeDetectorRef`*** service. For example, you can call ***`detectChanges()`*** or ***`markForCheck()`*** methods on the component's change detector to run change detection manually.

markForCheck schedules a future change detection run.

detectChanges triggers an immediate change detection for the current component and its children.

In the following example we have a component that uses the ***`OnPush`*** change detection strategy. In this example we update the ***`myValue`*** property inside a ***`setTimeout`*** call. Without explicitly calling ***`markForCheck`*** no change to ***`myValue`*** would be detected.

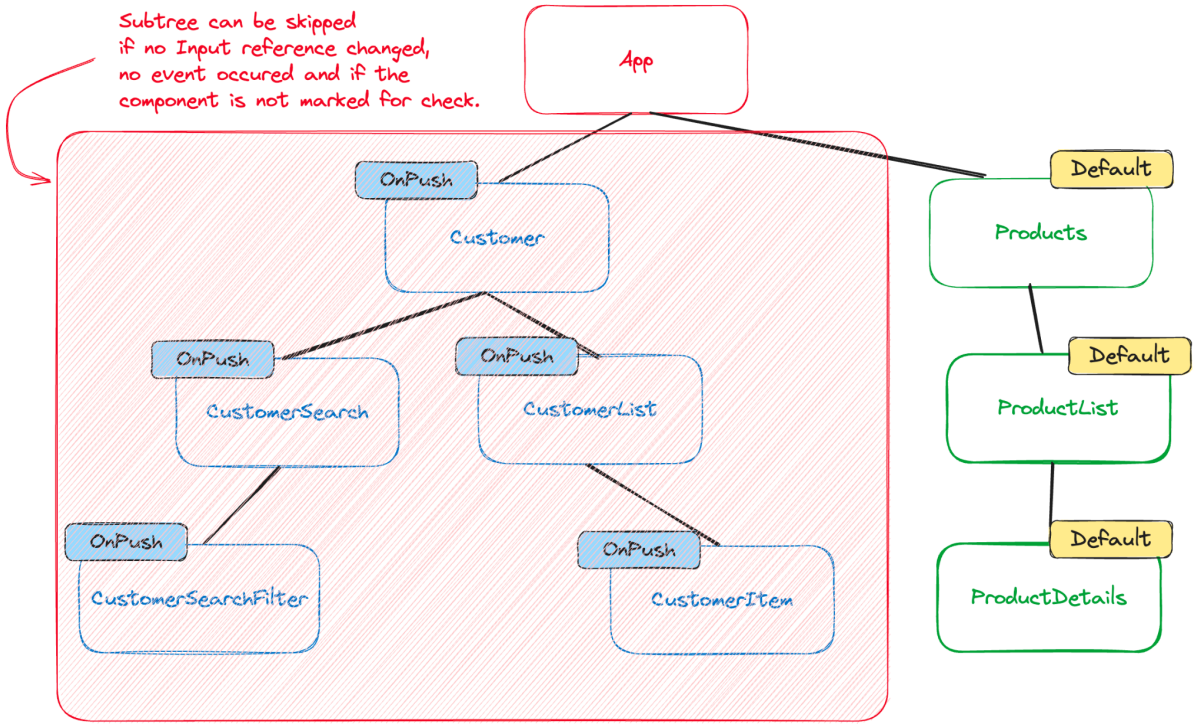
```
@Component({
  //...
  template: `{{ myValue }}`,
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class FooComponent {
  myValue = 'initial value';
  private cdr = inject(ChangeDetectorRef);

  constructor(){
    setTimeout(() => {
     this.myValue = 'updated value';
     /* Without this line the change detection will skip this
       component because it was not marked for check.*/
     this.cdr.markForCheck();
    }, 2000);
  }
}
```

The async pipe calls ***markForCheck*** under the hood whenever an Observable emits a value. Therefore ***OnPush*** components also run change detection if an Observable that the ***async*** pipe is subscribed to emits.

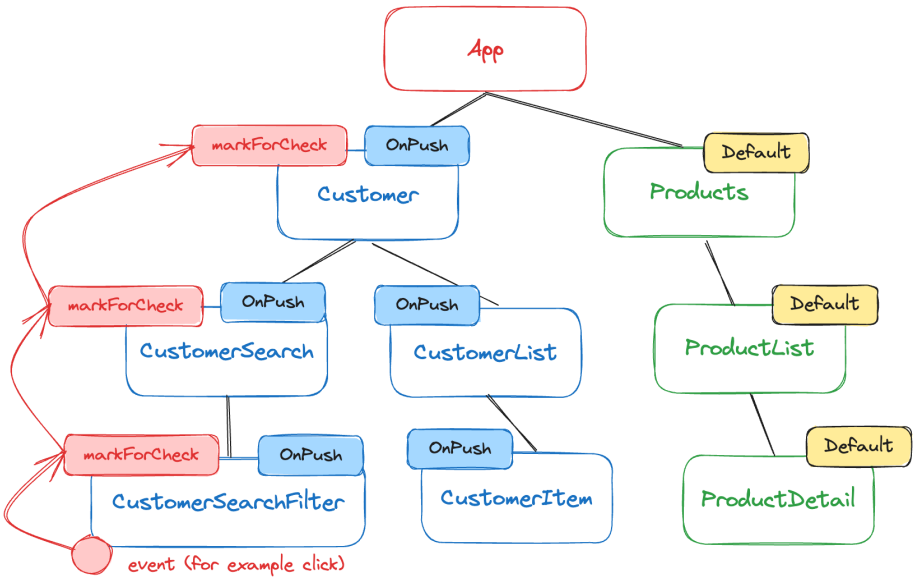
```
@Component({
  //...
  template: `{{ myValue$ | async }}`,
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class FooComponent {
  myValue$ = of('test').pipe(delay(2000));
}
```

The ***OnPush*** strategy enables us to reduce checks by bypassing subtree checks when none of the specified criteria on top occurs.

Let's explore the graphic below. Imagine a change occurs within the Products feature, the **OnPush** strategy significantly reduces the number of required checks since a subtree can be skipped.



Here it is also worth noticing that if one of the ***OnPush*** child components fires an event, for example the ***CustomerSearchFilter*** component. Not only the ***CustomerSearchFilter*** is marked for check but the whole subtree will be marked for check.



It's a good practice to use ***OnPush*** change detection for all of your components inside your application.

Use the ***angular.json*** file to configure schematics to generate new components with the ***OnPush*** change detection by default.

```
"schematics": {
  "@schematics/angular:component": {
   "standalone": true,
   "changeDetection": "OnPush",
   "style": "scss"
}
```

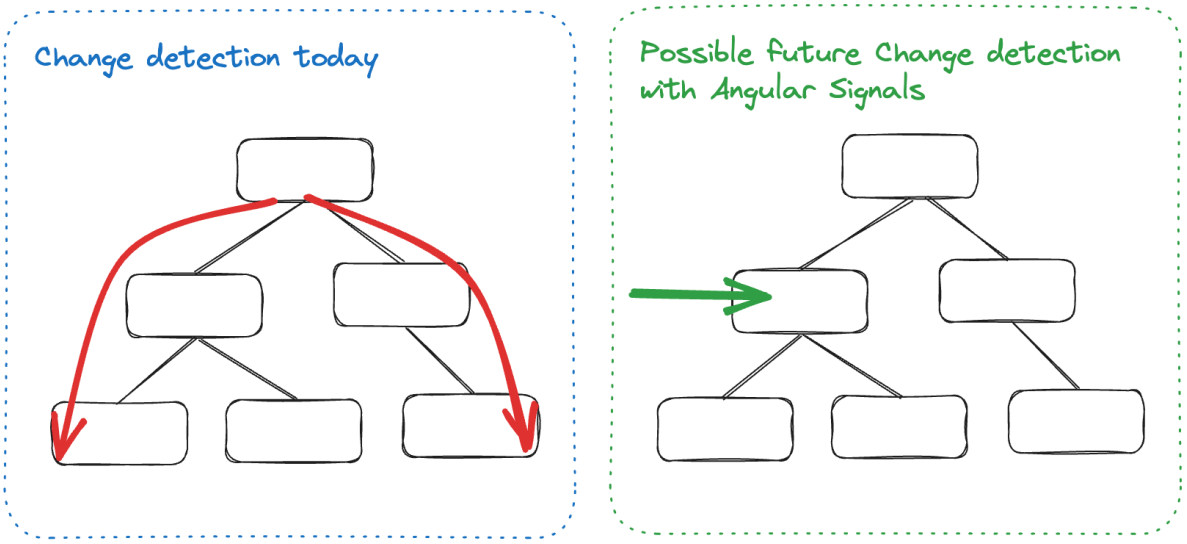
The ***OnPush*** strategy introduces a level of complexity, but its power lies in its ability to substantially decrease the number of checks, resulting in a significant performance boost.

Limitations of todays change detection

Despite the ***OnPush*** strategy, the situation is not yet ideal because change detection is still initiated from the root component, and Angular lacks awareness of where a specific change actually occurs. Due to this lack of information, Angular has to perform a lot of checks which can lead to bad performance.

Indeed, Signals represent the initial step towards enabling local change detection in the future within the Angular framework. This development is aimed at improving the efficiency and precision of change detection, making Angular applications even more powerful and performant.

Angular could build the notification system that informs Angular about changes right into Signals. This would mean that Angular no longer needs to walk the graph from top to bottom which would reduce the amount of unnecessary comparisons. This change could dramatically improve performance.



Not knowing where a change has happened is not the only major issue with todays change detection. Let's delve into another significant drawback of the current change detection mechanism: ***Zone.js***.

Zone.js

So far we've discussed how change detection works but haven't touched on how it's invoked. Completing the picture, it's essential to mention the actual invocation of change detection execution.

Zone.js & monkey patching

In Angular, change detection is invoked by a library called ***zone.js***. ***Zone.js*** is a library that monkey patches most of the browser APIs.

Monkey patching in JavaScript means modifying or adding code to existing objects or functions during runtime. Let's take a look at the following example:

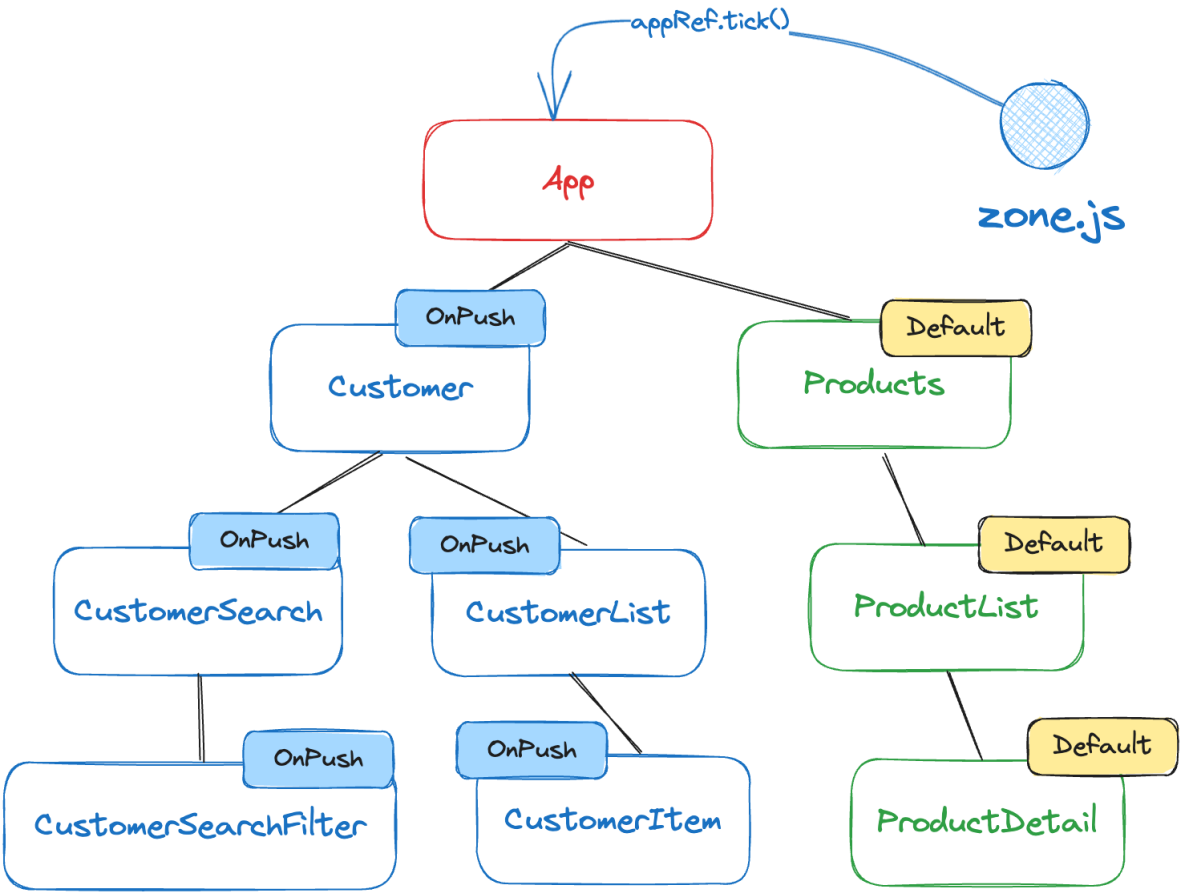
```
// MathOperations object with an 'add' method
const MathOperations = {
  add: function(a, b) {
    return a + b;
  }
};

// Save a reference to the original 'add' method
MathOperations.addOriginal = MathOperations.add;

/* Monkey patch the 'add' method to double the
result */
MathOperations.add = function(a, b) {
  return this.addOriginal(a, b) * 2;
};

// Use the patched 'add' method
const result = MathOperations.add(3, 4);
// This will return 14 (7 * 2)
```

Zone.js monkey patches more than 250 events.

abort animationcancel animationend animationiteration auxclick beforeinput blur cancel canplay canplaythrough change compositionstart compositionupdate compositionend cuechange click close contextmenu curechange dblclick drag dragend dragenter dragexit dragleave dragover drop durationchange emptied ended error focus focusin focusout gotpointercapture input invalid keydown keypress keyup load loadstart loadeddata loadedmetadata lostpointercapture mousedown mouseenter mouseleave mousemove mouseout mouseover mouseup mousewheel ...

When any of these browser events or asynchronous operations occur, ***`Zone.js`*** gets notified and informs Angular that an event has taken place.

Internally zone.js calls ***ApplicationRef.tick()***. This function forces Angular to perform change detection on the entire application.



While ***Zone.js*** offers several unique advantages, it's also important to acknowledge that it comes with some notable limitations and drawbacks.

Zone.js drawbacks

Zone.js generally functions effectively, but it does come with certain challenges and issues. Notably, ***Zone.js*** operates as a standalone library that performs monkey patching on browser events. As new browser APIs are introduced, ***Zone.js*** must be continuously updated to accommodate these APIs for seamless Angular change detection.

This task can be quite demanding, and it's not always feasible. Consequently, there are certain APIs that ***Zone.js*** does not fully support.

> ⚠ ***Zone.js*** doesn't work with native ***async / await!***

Even though *zone.js* does not patch native *async/await* we can still use *async/await* in our Angular code. The reason behind this is that the Angular CLI implicitly transforms *async/await* statements into *promises*.

Due to the extensive event patching by *Zone.js*, developers often worry about the potential for multiple redundant change detection runs. Each of these runs assesses the entire application tree unless optimized with *OnPush*. Consequently, the objective is to minimize the frequency of change detection runs as much as possible.

Zone.js provides a *eventCoalescing* option that helps avoiding multiple change detection runs in nested elements where events bubble up.

Imagine the following scenario:

```
<div (click)="parentHandler()">
  <button (click)="childHandler()"></button>
</div>
```

When a button is clicked, event bubbling causes both event handlers to be called, triggering two change detections by default.

If *eventCoalescing* is enabled change detection will only run once in the scenario above.

Zone.js can be configured to not patch certain events. However, this should be used with caution since you might enter scenarios where devs would expect updates but nothing happens since *Zone.js* did not patch an event.

Zone.js operates in a somewhat "magical" manner, which can make debugging challenging when it doesn't behave as expected. Additionally, since it's a standalone library, it has its own weight that needs to be loaded initially, impacting the performance of the initial page load.

One significant drawback of ***Zone.js*** is its inability to provide granular information about changes in the model. ***Zone.js*** primarily notifies Angular at its root about events, but it lacks the capability to provide detailed information regarding what specific changes occurred or where they occurred within the application.

Due to all those issues Angular developers have long wished for zoneless applications, and projects like RxAngular reflect the community's efforts to tackle this challenge.

One of the main goals of Signals is to enhance change detection by eliminating the reliance on ***Zone.js***.

The need for a new reactive primitive

Let's quickly summarize the issues with todays change detection and why we therefore need a new reactive primitive.

1. Angular lacks the ability to pinpoint where a change occurs in the application, leading to top-down checks starting from the root component.

2. As an application scales, Angular's need to check numerous components can negatively impact performance. While ***OnPush*** can alleviate this, it isn't always the ideal solution.

3. ***Zone.js***, which triggers change detection, can potentially introduce excessive change detection cycles due to its event patching, hampering performance.

4. ***Zone.js*** does not provide detailed information about what changed or the specific location of a change.

5. ***Zone.js*** needs to be loaded initially which slows down performance and initial load time of an application.

To address these challenges and enhance reactivity in Angular, a new reactivity model is needed — one that goes beyond the capabilities of ***Zone.js*** and the current dirty checking mechanism. This necessitates introducing a new foundational element into the framework, capable of offering fine-grained insights into model changes.

The goal is to seamlessly integrate this new reactivity model into Angular, making it intuitive for developers to use. Moreover, the new approach should maintain backward compatibility and be easily adoptable within existing applications.

After extensive exploration, the Angular team has introduced a new reactive primitive known as "Signals" to address these challenges and improve the reactivity model in Angular.

Signals - the Chosen Reactive Primitive

During research and experimentation, Signals stood out as a reactivity model that meets the requirements while offering a natural developer experience. Signals are not entirely new; other frameworks like Preact, Solid, and Vue have used similar concepts effectively.

Signals are wrappers around values that can notify interested consumers when a value might have changed.

Imagine a Signal as more than just a typical variable; it's like a dynamic entity that encapsulates both a value and a change notification.

Essentially, a Signal acts as a container, holding a certain value, but it also acts as a Producer and therefore possesses the ability to notify other Signals when its value undergoes any modifications.

You can think of a Signal as a box, holding a value within. It's like a treasure chest, guarding a secret that can change at any moment. The magic of a Signal lies in its ability to alert you whenever the value it holds transforms as if whispering, "Change has come!".

Just like a wrapped gift, the value inside the Signal box remains concealed until you unwrap it. Opening the box grants you access to the content within, allowing you to explore and utilize the updated information that awaits.

In essence, a Signal is a versatile construct that not only stores data but also acts as a messenger, broadcasting that something changed whenever its content is altered. This unique characteristics sets it apart from conventional variables.

Signals fulfill all the requirements for the new reactivity solution in Angular:

- They provide fine-grained information about model changes affecting individual components. In the future those fine-grained information can tell Angular what changed and enable Angular to run change detection on the affected views only.

In the future change detection will most likely be performed on the View level. Theoretically Angular could also perform it on an expression level but this would be to fine granular since the dependency graph (which we will cover later in this book) also has a cost.

- Signals allow synchronous access to their values, ensuring template bindings always have current values. This is an advantage over Observables. Observables can either be sync or async and are not guaranteed to provide an initial value. Therefore the ***`async`*** pipe initially delivers ***`null`*** as a value to the template.
- Reading Signals is side-effect-free and therefore prevents unintended side effects.
- Modern signal implementations are glitch-free and therefore avoid inconsistent states. Signals are immune to the so-called diamond dependency problem which we will cover later in this book.
- Automatic dependency tracking makes the usage convenient for developers. Signals internally construct and use a graph to know about dependent Signals. Again, we will cover this in a later section in more detail.

Additionally, Signals offer other advantages, such as lazily computed Signals and their easy composition with other reactivity systems like RxJS and Angular's current zone-based reactivity ensuring backwards compatibility.

Angular Signals API

With our theoretical foundation in place, let's now explore the practical aspects of working with Signals in Angular applications by diving into the API and understanding how to effectively utilize them.

Shall we begin?

Creating a signal

Creating a basic Signal in Angular is straightforward. You can do this by importing the Signal type and using its constructor, which requires an initial value. Here's how you can create a simple Signal:

A Signal always has an initial value!

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

@Component({
 //...
})
export class MyComponent {
  private myValue = signal('hello world');
}
```

We created a box and put the value “hello world” into it.

When we create a signal the type is automatically inferred.

```
private myValue = signal('hello world');
```

results in the following type:

```
(property) MyComponent.myValue:
WritableSignal<string>
```

Once you've instantiated a Signal, you can access its current value by invoking a zero-argument function.

```
constructor() {
  console.log(this.myValue()); //logs 'hello world'
}
```

By calling a signal we open the box to take the value out and log it to the console.

Angular introduces two interfaces, a ***Signal*** and a ***WritableSignal***.

Signals and writable Signals serve different purposes. A Signal is readonly while a writable Signal allows data modifications through three built-in methods: ***Set***, ***Update*** and ***mutate***. We will cover each of those methods in the next section.

Creating a new Signal in Angular via constructor call returns a ***WritableSignal.***

In most cases, we'll primarily work with ***WritableSignal***. However, there are situations where we need to use the read-only counterpart, ***Signal***.

The ***Signal*** interface becomes particularly valuable when we're defining functions and want to guarantee that we don't inadvertently modify the Signal within the function's body.

```
private myValue = signal('hello world');

constructor(){
  this.readValue(this.myValue);
  this.updateValue(this.myValue);
}

readValue(value: Signal<string>){
  // no set, update, mutate available
}

updateValue(value: WritableSignal<string>){
  // set, update, mutate available
}
```

Let's now take a close look at the ***set***, ***update*** and ***mutate*** functions.

Writable Signals

Writable Signal allow us to modify the signal value. They provide two built-in methods to do so: ***Set*** and ***Update***.

Set

The ***set*** method allows developers to directly set the Signal to a new value. This is useful for changing primitive values or replacing data structures when the new value is independent of the old one.

```
const myValue = signal('hello world');
console.log(myValue()); // logs: hello world

myValue.set('new value');
console.log(myValue()); // logs: new value
```

Update

The ***update*** method, on the other hand, enables updating the Signal's value based on its current value.

```
const myValue = signal(5);
console.log(myValue()); // logs: 5

myValue.update(v => v + 5);
console.log(myValue()); // logs: 10
```

Mutate (Removed)

Some of you may have already experimented with Signals when they were initially introduced. At that time, Signals came with a ***set***, ***update***, and a ***mutate*** function.

The ***mutate*** function was designed for changing a Signal's value by mutating it in place, making it suitable for Signals that hold non-primitive JavaScript values like arrays or objects.

```
const myValue = signal(['foo', 'bar']);
console.log(myValue()); // logs: ['foo', 'bar'];

myValue.mutate(v => v.push('baz'));
console.log(myValue()); // logs: ['foo', 'bar', 'baz']
```

On October 6th 2023 the Angular team removed the ***mutate*** function form the ***WritableSignal*** interface.

The removal of the ***mutate*** function isn't a significant loss, as everything achievable with ***mutate*** can also be accomplished using the ***update*** function. For instance, the code above can be rewritten more elegantly using the ***update*** function as follows:

```
myValue.update(v => [...v, 'baz']);
```

Why was mutate removed?

Mutate was dropped mainly for two reasons. With ***mutate*** the object reference always stays the same, therefore any code outside of the Angular Signals library can't know that an object is modified. Means, any change notification is lost as soon as Signals are read outside of a reactive context. Let's take a look at an example.

```
@Component({
  standalone: true,
  selector: 'child',
})
export class ChildComponent implements OnChanges {
  @Input() items: any;

  ngOnChanges(changes: any) {
    console.log('Changes', changes);
  }
}

@Component({
  //...
  template: `
    <button (click)="update()">Update</button>
    <button (click)="mutate()">Mutate</button>
    <child [items]="myValue()"/>
  `
})
export class App {
  myValue = signal(['foo']);

  update() {
    this.myValue.update((v) => [...v, 'baz']);
  }

  mutate() {
    this.myValue.mutate((v) => v.push('bar'));
  }
}
```

The application displays two buttons. The update button updates the values of our ***myValue*** Signal using the ***update*** method. The mutate button updates the value of our ***myValue*** signal using the ***mutate*** method.

The Signal is then passed to a child component via input.

Would you expect a log statement coming from ***ngOnChanges*** if we click on the update button?

Would you expect a log statement coming from ***ngOnChanges*** if we click on the mutate button?

We get a log statement if we click on update but we dont get one if we click on mutate. ***ngOnChanges*** does not get notified about the change from the ***mutate*** function since its not part of the reactive context.

The second reason why mutate was dropped is that in order to make the ***mutate*** function work, Angular defaulted the Signal value equality function to consider non-primitive values as always different. Which of course, is unfortunate for people working with immutable data structures since it makes the application less performant.

Equality Comparisons

Signals offer optional support for equality comparators, allowing developers to specify custom comparison functions for Signal values. Imagine we have an application that displays some products. We know that products all have a unique identifier. Therefore we can safely assume that if the ***id*** of a product is the same, the product is the same. In such cases it makes sense to pass in a custom equality comparison.

```
const myValue = signal(myProduct, {
  equal: (productA, productB) => productA.id === productB.id
})
```

Even without explicit equality comparators, Signals are designed to work efficiently with both mutable and immutable data, catering to a wide range of use cases. By default Angular uses ***Object.is()*** to compare the two values and to make sure that they are strictly equal.

Signals are lazy

Lazy execution

In the previous section we learned how to create a Signal and how to read a Signal. Let's quickly recap.

```
constructor() {
  // creating a Signal
  const mySignal = signal('hello world');

  // reading a Signal
  console.log(mySignal());
}
```

Signals are lazy. Means nothing happens until we call the Signal. This behavior is the same as with functions. Let's take a look at the following code snippet.

```
constructor() {

  // creating a Signal
  const mySignal = signal('hello world');

  // reading the Signal
  console.log(mySignal());

  mySignal.set('It is a beautiful day');
  // nothing happens since the Signal is not read
}
```

In this example, we initially create a Signal, read its value, and later update it. It's essential to recognize that no automatic updates occur; the Signal's change doesn't trigger any actions.

Angular templates and Signals

Now, consider the same scenario where we use the ***mySignal*** variable inside an Angular template. To update the ***mySignal*** value we incorporate a button that, upon being clicked, invokes a function which then uses the ***set*** method to update the value.

```
@Component({
  //...
  template: `
    <div>
      <p>Signal Value: {{ signalValue() }}</p>
      <button (click)="updateSignal()">Update Signal</button>
    </div>
  `,
})
export class SignalDemoComponent {
  signalValue = signal('hello world');

  updateSignal() {
    this.signalValue.set('It is a beautiful day');
  }
}
```

Initially our application renders “Hello world”. And once we click on the button the application renders “It is a beautiful day”. But why is that? What is the difference between the usage inside an Angular template and the constructor usage?

Well, there's a significant difference between the two snippets. Inside the Angular template the ***signalValue*** is called more times than in the constructor snippet. Called more times?

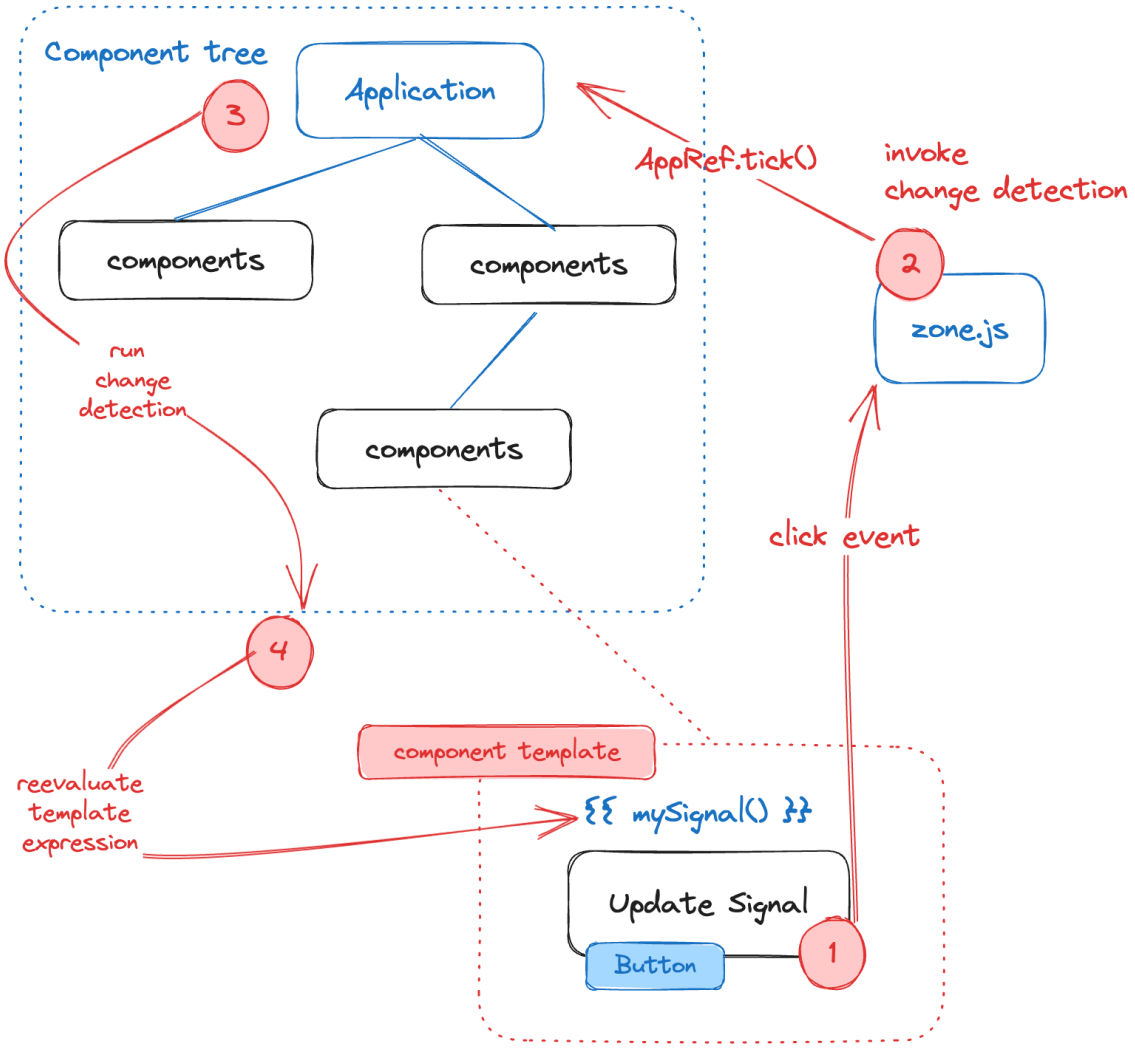
Yes, because the template is rerun and therefore our Signal is called again. We can even mimic this in our constructor example.

```
constructor() {
  // creating a Signal
  const mySignal = signal('hello world');

  // reading the Signal
  console.log(mySignal()); // logs: 'hello world'

  // updating the Signal
  mySignal.set(`It's a beautiful day`);

  // mimic the template call
  console.log(mySignal()); // logs: 'It’s a beautiful day'
}
```

This example here is pretty straight forward, but it still doesn’t explain why the value inside an Angular template just magically updates.

Well, the reason for that is our good old friend ***Zone.js***.

When a user clicks on the "Update Signal" button a click event is fired. ***Zone.js*** has monkey-patched the click event and listens to it. Once the click happens, ***Zone.js*** jumps in and notifies Angular that it should run change detection because something has changed.



Once Angular runs change detection and reruns the components template, ***mySingal*** is called again.

> **In general, calling functions in Angular template is considered bad practice.**
>
> **Signals do minimal computational work and therefore its not an issue to call them from templates.**

Currently, the Signals value inside the template is automatically updated because the Signal getter is reinvoked by the template evaluation during change detection. This magic works because of ***Zone.js*** and the way current change detection works.

Zone.js operates on a global level. This means as long as you have ***Zone.js*** in your app, ***Zone.js*** will run.

In the future, one of the primary goals of Signals is to eliminate the need for ***Zone.js***. With upcoming signal-based components, the Angular team aims to provide optimized change detection. A change detection mechanism that doesn't wait for specific events like clicks or timeouts but constantly monitors Signals and instantly re-renders when a change is detected.

The API for signal-based components is still in the RFC stage, and the exact form it will take in the future remains uncertain. However, a possible API could look something like this:

```
@Component({
  signal: true,
  standalone: true,
    //...
})
```

Signal-based components are still in RFC and it looks like we might not get them stable before Angular 18. But rest assured that the Angular team is actively working on delivering this enhancement. For the moment we still rely on ***Zone.js***.

The Angular team already mentioned that they want to support the mix of Signal based components and zone based components. This coexistence simplifies migration towards zoneless applications.

Up to this point, we've gained a solid understanding of Signals and how they play together with ***Zone.js***. Let's now explore how Signals behave once we start using them inside a component that uses the ***OnPush*** change detection strategy.

Signals and OnPush

As discussed in the OnPush chapter, a component that is using the change detection strategy ***OnPush*** only rechecks the View if an input reference changed, an event within the component was emitted or if the component was marked as dirty.

Let's take a look at the following example:

```
export function getProducts(): Observable<any[]> {
  return of(['shoes', 'socks', 'shirt']).pipe(
   delay(2000)
  );
}

@Component({
  //...
  template: `<div>{{ products }}</div>`,
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class ChildComponent {
  products = [];

  constructor() {
    // returns an Observable
    getProducts().subscribe(products => {
      this.products = products;
    })
  }
}
```

What would you expect to be rendered if we run this code?

Since the component is ***OnPush*** and none of the tree criterias happened, nothing actually happens - nothing gets rendered. So we could easily fix this by directly assigning the returned Observable to a ***products$*** field which we then use inside our template with the ***async*** pipe.

```
export function getProducts(): Observable<any[]> {
  return of(['shoes', 'socks', 'shirt']).pipe(
   delay(2000)
  );
}

@Component({
  //...
  template: `<div>{{ products$ | async }}</div>`,
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class ChildComponent {
  products$ = getProducts();
}
```

Here change detection is triggered because the ***async*** pipe under the hood calls ***markForCheck()***.

Great, so what about Signals? What happens if we use a Signal inside the template?

```
export function getProducts(): Observable<any[]> {
  return of(['shoes', 'socks', 'shirt']).pipe(
   delay(2000)
  );
}

@Component({
  //...
  template: `<div>{{ products() }}</div>`,
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class ChildComponent {
  products = toSignal(getProducts(), {initialValue: []});
}
```

The ***toSignal()*** function is a function that converts an Observable to a Signal. We will cover it later in more detail in this book.

Alternatively we could also have written the following code, which is just more verbose and involves dealing with the subscription ourselves.

```
constructor() {
  getProducts().subscribe(p =>
    this.products.set(p));
}
```

Always use ***toSignal*** over such an approach 😉

What would you expect to be rendered if we run this code? Would you expect the Signal to be invoked? Remember that the component is ***OnPush***.

Well, if the updated Signal value is updated or not completely depends if the template is invoked or not. And the only way for the template to be rerun in this case is if the view is marked for check.

And indeed, it turns out that the template gets updated once we use Signals. Behind the scenes Angular uses ***`ReactiveLViewConsumer`*** which serves as a consumer of our Signal. Whenever a Signal emits a value the ***`ReactiveLViewConsumer`*** gets notified and marks the view for check. Therefore the component is rechecked, template is rerun, our Signal gets called and the updated value gets displayed.

Now, let's shift our focus towards exploring the concepts of computed Signals and effects.

Computed Signals

In this chapter, we will explore the concept of computed Signals, a powerful feature that allows developers to create derived values based on one or more Signals.

Computed Signals are designed to create derived values that depend on other Signal values.

Creating a computed Signal

Let's start by creating a simple computed Signal.

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

// creating a signal
const counter = signal(0);

// creating a computed signal
const isEven = computed(() => this.counter() % 2);
```

To craft a computed Signal, we begin by importing the ***computed*** function from ***@angular/core***. Next, we call this function, supplying it with a callback function, often referred to as a computation function.

Within this computation function, we possess the freedom to leverage other Signals or other computed Signals.

The computation function is expected to be side-effect-free, meaning it should only access values of dependent Signals and avoid any mutation operations.

Attempting to write to other Signals from within the computed Signals computation function will raise an error.

```
Error: NG0600: Writing to Signals is not allowed in
a computed or an effect by default. Use
allowSignalWrites in the CreateEffectOptions to
enable this inside effects.
```

Equality function

Creating a computed Signal involves defining the computation function and, optionally, providing an equality function to optimize recomputations.

If the equality function determines that two values are equal, the computation for the computed Signal is skipped, reducing unnecessary recalculations.

```
cheapestProduct = computed(
  () => this.findCheapestProduct(this.selectedProducts()), {
  equal: (a, b) => {
    return a.id === b.id;
  }
});
```

Computed Signals inside an Angular template

Let's take a look at a concrete example of how we can use a Signal together with two buttons to increment and decrement a counter. Let's then create a computed Signal ***doubleCount*** that displays the double count in the Angular template.

```
@Component({
  selector: 'app-root',
  standalone: true,
  template: `
    <h1>Count: {{ counter() }}</h1>
    <h2>Double count: {{ doubleCount() }}</h2>
    <button (click)="incrementCounter()">
      Increment counter
    </button>
    <button (click)="decrementCounter()">
      Decrement counter
    </button>
`
})
export class AppComponent {

  counter = signal(1);
  doubleCount = computed(() => this.counter() * 2);

  incrementCounter() {
    this.counter.update((count) => count + 1);
  }

  decrementCounter() {
    this.counter.update((count) => count - 1);
  }
}
```

Initially the Signal and the computed Signal is pulled by the template and the following output renders.

Once we click on the “Increment counter” button the count would go up to *2* and the double count would go up to *4*.

This only works because *Zone.js* patches the click event, runs change detection and then the values are pulled from the signals in the template again.

Maybe you wonder how computed Signals know when to recompute. Angular internally uses a clever Push -> Poll -> Pull algorithm which we will cover in detail later in this book.

Let’s take a look at some of the benefits and characteristics of computed Signals.

Computed Signals characteristics

The implementation of computed Signals in Angular guarantees certain features that ensure efficiency and accuracy.

Laziness

Computed Signals computations are lazy, meaning the computation function is only invoked when someone reads its value and only if one of the Signals used in the computation body has produced a new value. This lazy evaluation minimizes unnecessary calculations.

Don't worry if this doesn't make much sense yet. We will explore a concrete example once we have explained the Push -> Poll -> Pull algorithm.

Internally Angular uses a graph data structure with dynamic dependency tracking. So a computed Signal internally always knows about the Signals it depends on.

WritableSignals **are internally abstracted as** ***Producers*****. Once a computed Signal uses a Producer inside the computation function, the computed Signal becomes a Consumer of that Producer.**

Let's take a look at the following code snippet:

```
const counter = signal(5);
const doubleCount = computed(() => {
  console.log('Executing the computation function')
  return counter() * 2
});

console.log(doubleCount());

counter.set(6);
console.log(doubleCount());

counter.set(6);
console.log(doubleCount());
```

What log output would you expect? How many times do you think the computation function is invoked?

```
Executing the computation function
10
Executing the computation function
12
12
```

1. The first time we log ***doubleCount*** the computation function is invoked and returns the value of ***10***. Prior to this step nothing happened from the point of view of our computed Signal, no recomputation, nothing.
2. Next, we update our ***counter*** signal and set its value to ***6***.
3. We call ***doubleCount*** which again invokes the getter and therefore the value of ***12*** is returned.
4. We then again set the value of our counter to ***6***, which is equal to the latest value of the ***counter*** Signal. Means, no semantic new value is produced.

5. Even though we log out the ***doubleCount*** in the last line of our code snippet, no computation function is invoked since there is no recomputation needed.

Automatic Disposal

Internally Angular uses a concept of "liveness" for producer-to-consumer dependency tracking. This concept is important since it helps to avoid memory leaks.

```
const counter = signal(1);

let doubleCount = computed(() => counter() * 2);
console.log(doubleCount());

doubleCount = null;
```

In this example the ***doubleCount*** Signal is automatically eligible for garbage collection as soon as its out of scope / its value is set to ***null***.

If the dependency graph had a strong reference from ***counter*** to ***doubleCount***, then ***doubleCount*** would not be eligible for garbage collection even if the user no longer holds a reference to the actual Signal. However, since ***doubleCount*** is no longer needed, the graph doesn't maintain a reference from ***counter*** to ***doubleCount***, allowing ***doubleCount*** to be freed when the user drops it.

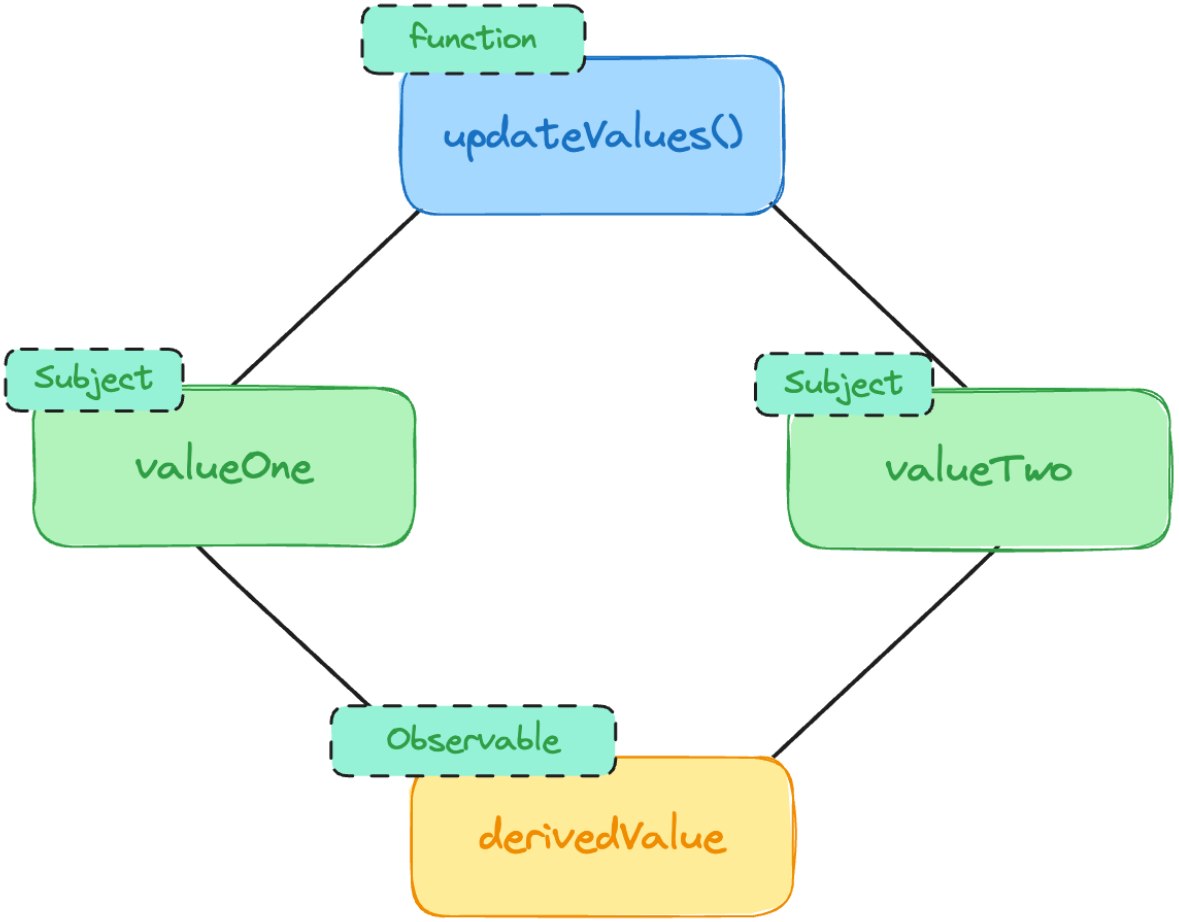
Glitch-Free Execution

Let's take a look at a very known issue with many reactive libraries. The diamond dependency problem. This is a known problem in the world of RxJs. Let's explain it on a real example.

```
@Component({
  //...
  imports: [AsyncPipe],
  template: `
    <h2>{{ combinedStatement$ | async }}</h2>
    <button (click)="updateValues()">Update values</button>
`
})
export class MyComponent {
  private valueOne$ = new Subject('Value one: initial');
  private valueTwo$ = new Subject('Value two: initial');

  public derivedValue$ = combineLatest(
    [this.foo$, this.bar$]
  ).pipe(
    tap(([valueOne, valueTwo]) =>
      console.log(`LOG: ${valueOne} / ${valueTwo}`)
    ),
    map(([valueOne, valueTwo]) => `${valueOne} / ${valueTwo}`)
  );

  updateValues(){
    this.valueOne$.next('Value one: updated');
    this.valueTwo$.next('Value two: updated');
 }
}
```

We have two Subjects, ***valueOne$*** and ***valueTwo$***. We then use ***combineLatest*** to create a ***derivedValue$*** which we display inside our template using the ***async*** pipe. Additionally, we also have a button that on click updates the values of the ***valueOne$*** and the ***valueTwo$*** at the same time.

The setup on top can be illustrated in the following way.



I hope the illustration gives you an idea why this problem is referred to as the diamond dependency problem. Alright, but what is actually the issue? In the code we use the `tap` operator so that we can inspect our log statements. The log statements will help us inspect the glitch behavior.

Let's start our app, click on our button, and inspect the console output.

What log output would you expect?

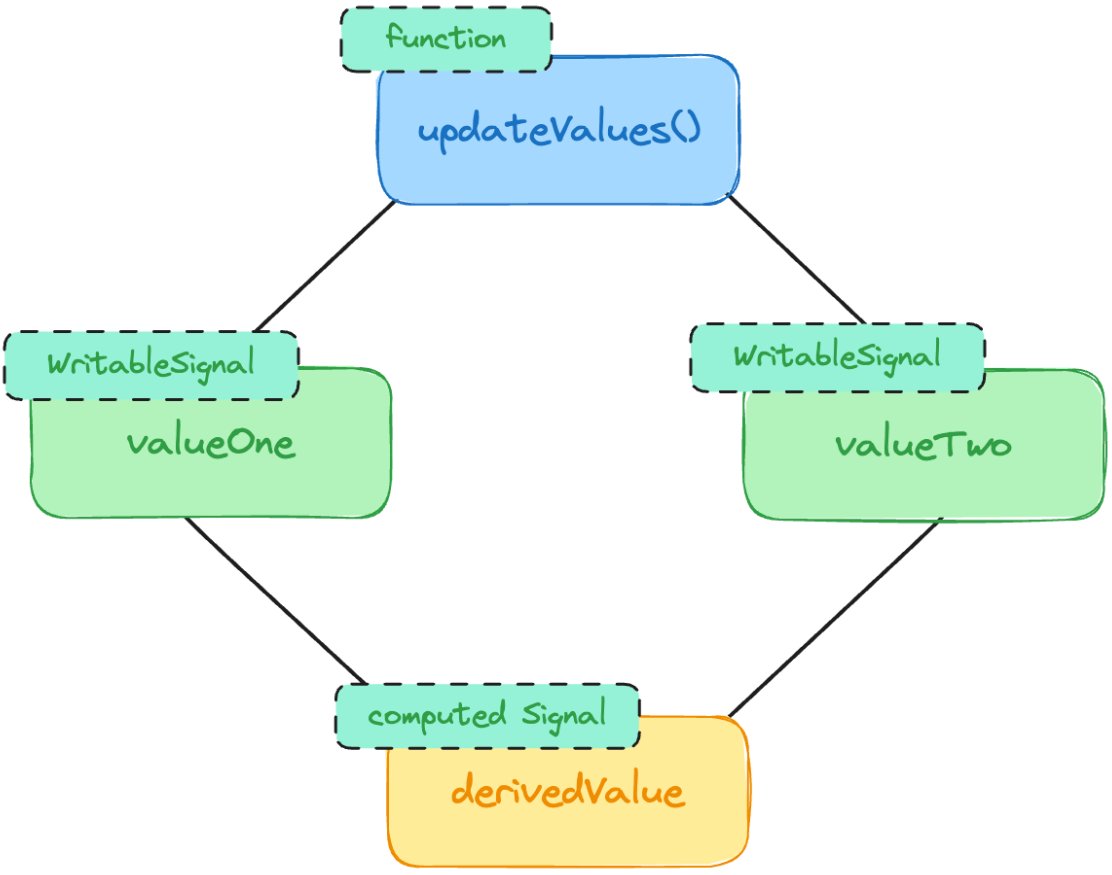
```
LOG: Value one: initial / Value two: initial
LOG: Value one: updated / Value two: initial
LOG: Value one: updated / Value two: updated
```

We start with the initial values of the ***valueOne$*** and ***valueTwo$*** subject. Once we click we get two log statements. Why is that? It's because we are using ***combineLatest***.

When we use ***combineLatest*** we end up with an intermediate state.

```
LOG: Value one: updated / Value two: initial
```

Getting this intermediate state is referred to as glitch. Let's refactor this code into Signals and check out the log statements. Again, the graph looks exactly the same, just that we use Signals instead of Subjects, and a computed Signal instead of an Observable.



```
@Component({
  //...
  template: `
    <h2>{{ derivedValue() }}</h2>
    <button (click)="updateValues()">Update values</button>
`
})
export class MyComponent {
  counter = 0;
  private valueOne = signal('Value one: initial');
  private valueTwo = signal('Value two: initial');

  public derivedValue = computed(() => {
    console.log(`LOG: ${this.valueOne()} / ${this.valueTwo()}`)
    return `${this.valueOne()} / ${this.valueTwo()}`
  })

  updateValues(){
    this.valueOne.set('Value one: updated');
    this.valueTwo.set('Value two: updated');
  }
}
```

Again, we click on the button and inspect the log output:

Do you expect the log output to differ? If yes, how?

```
LOG: foo: initial / bar: initial
LOG: foo: updated / bar: updated
```

In the Signal version we only get two log statements. In other words, we never enter the intermediate state and only end up with the final result.

Computed Signals never execute with stale or intermediate dependency values, and they are immune to the "diamond dependency problem".

The reason behind that is again Angular's clever Push -> Poll -> Pull algorithm in combination with *`Zone.js`* and Angulars template execution.

We often mentioned the Push -> Poll -> Pull Algorithm and I bet you can't wait to find out more about it. And we will get there, promised. But before we go there we first have to cover the effects.

Effects

In this chapter, we will explore the concept of effects in Angular Signals, a powerful tool for managing side-effectful operations within reactive applications.

Creating effects

An effect is a side-effectful operation that reads the value of one or more Signals. When any of the dependency Signals change, the effect is automatically scheduled to re-run.

The execution timing of effects is not guaranteed and may vary based on different strategies chosen by Angular.

Let's imagine we have a simple app that displays a todo list. We want to add a simple feature that shows confetties on the page once we have done all of our todos. Let's implement this feature with the help of an ***`effect`***.

Let's import the effect from ***`@angular/core`*** and then pass a callback to it. Inside the callbacks body we can then access other writable Signals or computed Signals.

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

constructor() {
  effect(() => {
    if(openTodos() === 0){
      showConfetti();
    }
  })
}

showConfetti() {
  // use confetti-js to handle display of confetti
}
```

As Signals and computed Signals effects are lazy as well. Means somebody has to call it in order for something to happen.

We as developers decide if a Signal or a computed Signal is called by using them either in our TypeScript code or inside template expressions.

Effects on the other side are called by Angular internally.

In the example above we register an effect that Angular then runs. If we have no more open todos our Signal contains a new value which will then be pulled by the effect to show some confetti on our page.

Again, behind the scenes Angular uses the Push -> Poll -> Pull algorithm for this.

Effects are a great way to perform side effects such as triggering network requests, perform rendering actions, and more. Effects have a variety of use cases, including:

- Logging information into the console
- Triggering network requests
- Opening modals
- Programmatic navigation
- Scrolling
- etc...

Effects and the Injection context

Internally effects inject Angulars ***`DestroyRef`*** and can therefore only be called inside an injection context.

In Angular version 16, a new provider called ***DestroyRef*** has been introduced. It enables you to register callbacks that trigger when a specific component or directive is destroyed or when a corresponding injector is destroyed.

Injecting the ***`DestroyRef`*** service behind the scenes enables self-cleanup of effects out of the box. This feature makes the code more robust and easier to manage.

An injection context allows you to inject values from the Angular Dependency Injection System.

There are currently default three injections contexts:

- a constructor
- a factory injector
- a field initializer

✅ Injection context
❌ Not an injection context

```
@Component({
  //...
  providers: [
    {
      provide: SOME_VALUE_FROM_FOO,
      useFactory: () => inject(FooService).value ✅
    }
  ]
})
export class MyComponent implements OnInit {
  foo = inject(FooService); ✅

  constructor(){
    const fooTwo = inject(FooService); ✅
  }

  ngOnInit() {
   const fooThree = inject(FooService); ❌
  }

  clickHandler() {
    const fooFour = inject(FooService); ❌
  }
}
```

Even though a field declaration is an injection context, it is not really suitable for effects since you will get a TypeScript error unless you assign the returned ***effectRef*** to a field.

```
export class MyComponent {

 // ✅
 private effectRef = effect(() => {
   // do something
 });

 // ❌
 effect(() => {
   // do something
 });

 constructor() {

   // ✅
   effect(() => {
     // do something
   });
 }
}
```

Most of the time we will define effects inside our ***`constructor`***.

runInInjectionContext

Usually it's enough to define an ***`effect`*** inside a ***`constructor`*** but this might not always be the case. Imagine the following scenario.

```
export class MyComponent {
  @Input({required: true}) myNumber!: number;
  myMultiplicator = signal(2);

  constructor() {
    effect(() => {
      console.log(
        `Result: ${this.myMultiplicator() * this.myNumber}`
      )
  });
}
```

If we run this example we would get the following log output:

Angular schedules and executes the ***effect*** which results in an initial execution of our callback function.

```
Result: NaN
```

Inputs are only available in ***ngOnChagnes*** and ***ngOnInit*** but not in a ***constructor***.

To get the value of the ***myNumber*** input we have to move our effect call into the ***ngOnInit*** lifecycle hook. But then we get the error that an effect can only be used within an injection context. So how do we solve this?

We can solve this by getting a hold of the ***Injector*** and passing it to a ***runInInjectionContext*** function.

```
export class MyComponent implements OnInit {
  private injector = inject(Injector);

  @Input({required: true}) myNumber!: number;
  myMultiplicator = signal(2);

  ngOnInit() {
    runInInjectionContext(this.injector, () =>
      effect(() => {
        console.log(
          `Result: ${this.myMultiplicator() * this.myNumber}`)
        }
      )
    );
  }
}
```

By moving the code into the ***ngOnInit*** lifecycle hook our effect now correctly accesses the ***myNumber*** variable and therefore logs the correct output:

```
Result: 8
```

This only works for the first change of ***myNumber***. If ***myNumber*** changes at a later point and we want to react to it we would need to use the ***ngOnChanges*** lifecycle hook, signalify our input or hopefully in the future just use a signal based input.

Stopping Effects

By default, Angular effects are tied to the underlying ***DestroyRef*** in the framework, meaning they will be automatically tear down when the component or directive that created them is destroyed.

Signals are automatically cleaned up while Observables on the other side often have to be cleaned up manually if they can not be subscribed in a template with the *async* pipe.

```
export class MyComponent implements OnDestroy {

  destroy$ = new Subject();
  private activatedRoute = inject(ActivatedRoute)

  constructor() {
    this.activatedRoute.params.pipe(
      takeUntil(this.destroy$)
    )
    .subscribe(() => {
      // do something with params
    });
  }

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

When possible you should always try to use the *async* pipe over manual subscriptions inside the TS code of your component.

If more control over the lifespan scope is needed, developers can use the ***manualCleanup*** option during effect creation to prevent automatic destruction. ***manualCleanup*** can be used in combination with the ***EffectRef*** instance returned from the effect to explicitly destroy effects.

```
private effectRef = effect(
  () => {
    console.log('Something');
  },
  {
     manualCleanup: true,
  }
);

stopEffect() {
  this.effectRef?.destroy();
}
```

In this example, we set up an effect with the ***manualCleanup*** option enabled and store its reference in a ***effectRef*** member. Once ***stopEffect*** is called we shut down the effect and remove it from any upcoming scheduled executions.

Cleanup functions

Effect functions can optionally register a cleanup function. The cleanup function is executed before the next effect run and provides a way to cancel any work started by the previous effect run.

Imagine we start an interval inside an effect. In that case we want to clean up that interval before the next effect run.

```
effect(onCleanup => {
  const id = setInterval(() => {
    console.log(`${getFood()} delivered`);
  }, 1000);
  onCleanup(() => {
    console.log('Clearing and re-scheduling effect');
    clearInterval(id);
  });
});
```

Scheduling and Timing of Effects

The execution timing of effects is not guaranteed and may vary based on different strategies chosen by Angular.

Developers should not depend on any specific observed execution timing. However, the following guarantees apply:

1. Effects will execute at least once.

2. Effects will execute in response to changes in their dependencies at some point in the future.

3. Effects will execute a minimal number of times; if multiple dependencies change simultaneously, only one effect execution will be scheduled.

Writing to Signals from Effects

Angular Signals generally discourage writing to Signals from effects, as it can lead to unexpected behavior, such as infinite loops, and make data flow challenging to follow.

Writing to Signals from an effect will be reported as an error and blocked by default.

However, developers can override this behavior by passing the ***`allowSignalWrites`*** option during effect creation if they have a specific use case that requires writing to Signals.

A complex form is a great sample use case where you want to set ***`allowSignalWrites`*** to ***`true`***. Imagine we have a form with two selects. The values of the second select change based on the selection of the first select.

```
@Component({
  //...
  template: `
    <form [formGroup]="favoriteFood">
      <select formControlName="category">
        <option>Food</option>
        <option>Beverages</option>
      </select>
      <select
        *ngIf="products().length" formControlName="products">
        <option *ngFor="let product of products()">
         {{ product }}</option>
      </select>
    </form>
`,
})
export class MyComponent {
  products = signal([]);
  favoriteFood = inject(FormBuilder).group({
    category: [''], products: [''],
  });

  categoryChange = toSignal(
    this.favoriteFood.get('category')!.valueChanges, {
      initialValue: ''
  });

  constructor() {
    effect(() => {
      if(this.sportChange() === 'Food'){
        this.products.set(FOOD_PRODUCTS);
      }

      if(this.sportChange() === 'Beverage'){
        this.products.set(FOOTBALL_PLAYERS);
      }
   }, { allowSignalWrites: true})
  }
}
```

Whenever we change the category we want to update the ***products*** Signal. If the user chooses the category to be “Food” we will display a selection with different food. If he chooses “Beverage” then we will show a list of beverages in the second box. To achieve this we have signalified the ***valueChanges*** of our ***category*** select. Whenever such a change occurs our effect runs and then updates the ***products*** values.

Therefore the example below would result in the following error:

Writing to Signals is not allowed in a ***computed*** or an ***effect*** by default. Use ***allowSignalWrites*** in the ***createEffectOptions*** to enable this inside effects.

Let’s take a look at a concrete example where we try to change the `doubleCount` variable inside an `effect`.

```
@Component({
  //...
  template: `
    <h1>Count: {{ counter() }}</h1>
    <h2>Doublecount: {{ doubleCount() }}</h2>
    <button (click)="incrementCounter()">
      Increment counter
    </button>
    <button (click)="decrementCounter()">
      Decrement counter
    </button>
`
})
export class AppComponent {
  counter = signal(1);
  doubleCount = signal(2);

  constructor() {
    effect(() => {
      this.doubleCount.set(this.counter() * 2);
    })
  }

  incrementCounter() {
    this.counter.update((count) => count + 1);
  }

  decrementCounter() {
    this.counter.update((count) => count - 1);
  }
}
```

In the example above we should have used ***computed*** instead of ***effect***. We should never use an ***effect*** to calculate derived data.

To make this example work we have to refactor our ***effect*** function inside the ***constructor***.

```
constructor() {
  effect(() => {
    this.doubleCount.set(this.counter() * 2);
  },
  {
     allowSignalWrites: true
  });
}
```

By enabling the ***allowSignalsWrite*** flag, our effect gains the capability to write to Signals.

This powerful feature comes with great potential, but it's crucial to exercise caution while using it.

Effects in Angular Signals provide a powerful mechanism for managing side-effectful operations within reactive applications. By automatically responding to changes in dependent Signals and offering optional cleanup functions, effects ensure efficient and robust handling of side effects.

Preventing dependencies with untracked

As we have seen an effect or a computed Signal recomputes if one of the Signals accessed inside the callback produces a new value.

In the following application we display a `summarizedCount` that goes up wheter we click on the “Increment counter one” button or the “Increment counter two” button. Furthermore the callback function provided to our `effect` also runs and logs the sum to the console.

```
@Component({
  //...
  template: `
    {{ summarizedCount() }}
    <button (click)="incrementOne()">
      Increment counter one
    </button>
    <button (click)="incrementTwo()">
      Increment counter two
    </button>
`,
})
export class App {
  counterOne = signal(1);
  counterTwo = signal(2);
  summarizedCount = computed(
    () => this.counterOne() + this.counterTwo()
  );

  constructor() {
    effect(() => {
      console.log(
       `Sum: ${this.counterOne() + this.counterTwo())}`
      );
    })
  }

  incrementOne() {
    this.counterOne.update(v => v + 1);
  }

  incrementTwo() {
    this.counterTwo.update(v => v + 1);
  }
}
```

Now let's say that we have a use case where we only want to update the ***summarizedCount*** and log the sum to the console if ***counterOne*** changed. How would we do that?

Well, Angular provides us a ***untracked*** function.

The *untracked* function enables us to access a Signal's value without initiating reactivity. In other words, it allows us to read the value of a WritableSignal (Producer) within its callback without causing any computations to be triggered in connected effects or computed Signals.

The ***untracked*** function itself accepts a callback wherein we use a Signal. Let's take a look.

```
effect(() => {
  console.log(foo(), untracked(() => bar()));
})
```

The ***effect*** here would only react on changes made to ***foo***, but not on changes made to ***bar***. Let's go ahead and update the example above and take advantage of the ***untracked*** function so that our summarized counter and the log statements only update / get logged if the first counter gets incremented.

```
@Component({
  //...
  template: `
    {{ summarizedCount() }}
    <button (click)="incrementOne()">
      Increment counter one
    </button>
    <button (click)="incrementTwo()">
      Increment counter two
    </button>
`,
})
export class App {
  counterOne = signal(1);
  counterTwo = signal(2);
  summarizedCount = computed(
    () => this.counterOne() +
         untracked(() => this.counterTwo())
  );

  constructor() {
    effect(() => {
      console.log(
       `Sum: ${this.counterOne() +
        untracked(() => this.counterTwo()))}`
      );
    })
  }

  incrementOne() {
    this.counterOne.update(v => v + 1);
  }

  incrementTwo() {
    this.counterTwo.update(v => v + 1);
  }
}
```

Signals Push -> Poll -> Pull nature

In this book, we have frequently touched upon the fundamental Push, Poll, and Pull-based dynamics that underlie Signals in Angular. Now, the moment has arrived for us to dive deeper into the algorithm that drives Angular Signals.

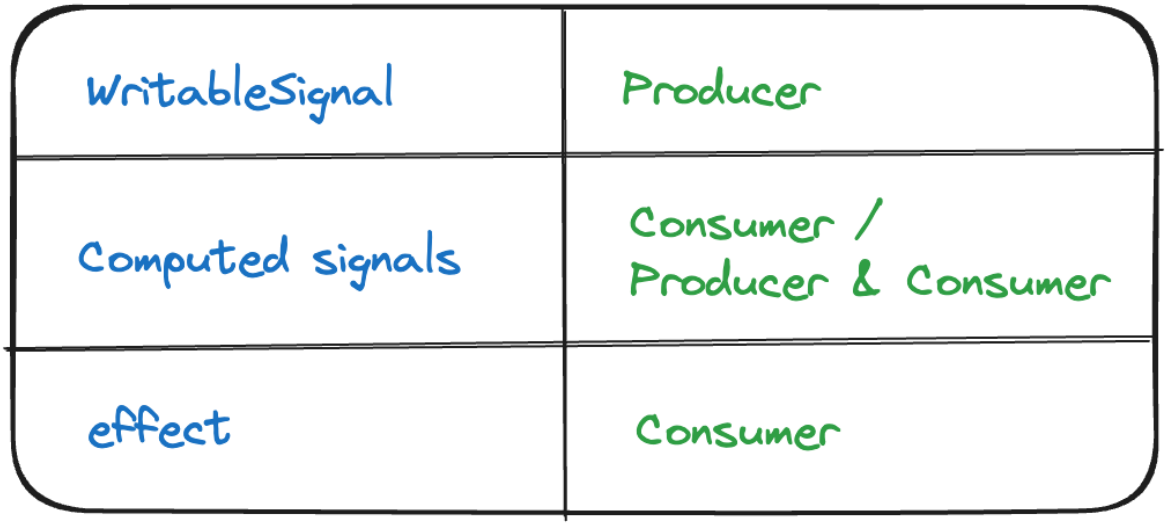
Dynamic dependency graph

Before delving into the intricate details of this algorithm, it's essential to familiarize ourselves with two foundational abstractions: Consumers and Producers. These fundamental components will serve as the building blocks for our journey into the inner workings of Angular Signals.

- **Producer:** Delivers change notification to its consumers.
- **Consumers:** Represents a reactive context that depends on producers.

Producers produce reactivity and consumers consume it.

Depending on the specific context or flavor, Signals can serve either as a Consumer, Producer, or even as both simultaneously.



Both Producers and Consumers keep track of the dependencies to each other. Each Consumer knows about its Producers while each Producer knows about its Consumers. All the references are always bidirectional.

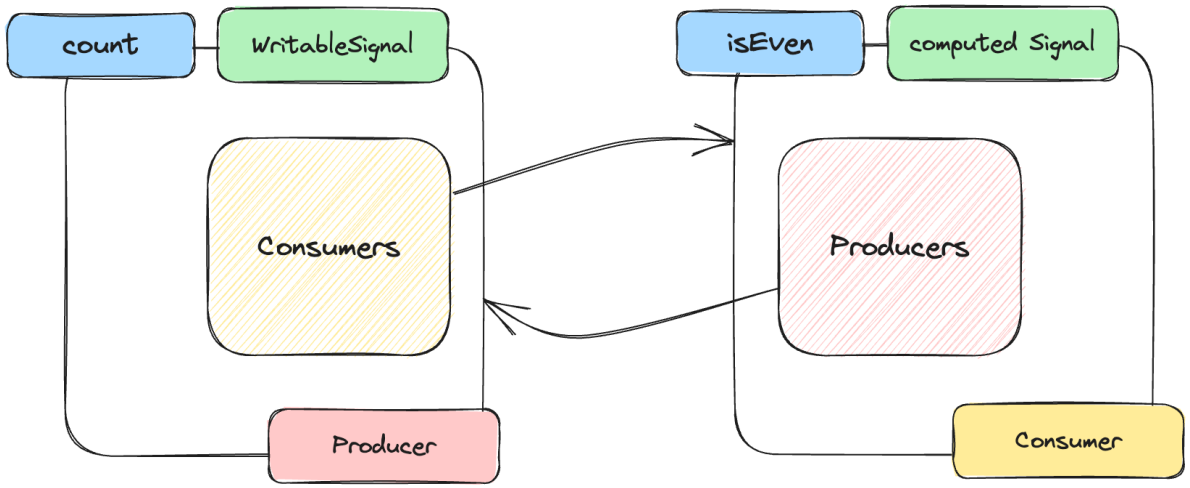
Internally all Consumers and Producers are represented as a ***ReactiveNode***. The reactive nodes track the dependencies between each other. Collectively, these connections form what we commonly refer to as the "dependency graph".

This dependency graph is dynamic, which means it is updated every time we invoke a Consumer. Let's take a look at the following code.

```
const count = signal(2);
const isEven = computed(() => this.count() % 2);
console.log(isEven());
```

In this scenario, we have a ***count*** Signal that fulfills the role of a Producer. This ***count*** Signal is subsequently consumed within a computed Signal named ***isEven***. The pivotal moment occurs when we invoke the ***isEven*** Signal, triggering an update to our dependency graph. This real-time graph adaptation is a fundamental aspect of how Angular Signals manage their dynamic relationships.

Let's visualize the graph and the connections between the ***count*** Signal and the ***isEven*** computed Signal.



Angular establishes a bidirectional connection between the ***count*** Signal (Producer) and the ***isEven*** computed Signal (Consumer) forming a graph.

Indeed, the dependency graph takes on a pivotal role in the functioning of Angular Signals. As the connections between Consumers and Producers are established within this graph, it becomes the cornerstone for the recursive traversal necessary for the application of the Push -> Poll -> Pull algorithm.

An Algorithm in two phases

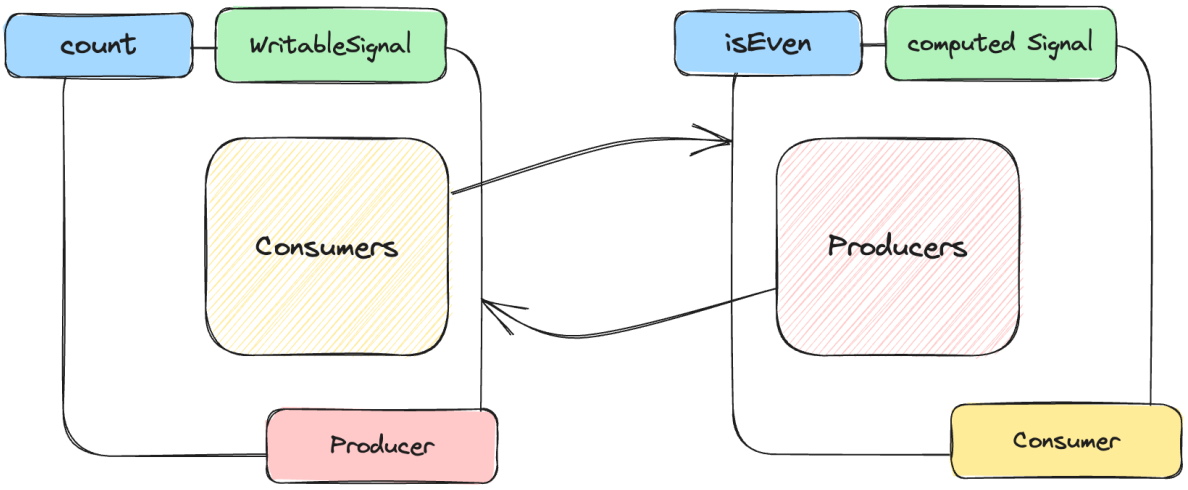
The Push -> Poll -> Pull algorithm is split into two phases.

- **First phase:** Push
- **Second phase:** Poll / Pull

The split into two phases is the secret behind the laziness. Let's illustrate the two phases a bit further by looking at the example above.

```
const count = signal(2);
const isEven = computed(() => this.count() % 2);
console.log(isEven());
```

The example here would result in the following graph. count is a producer of ***isEven*** and ***isEven*** is a consumer of ***count***.



In the first run every Signal starts dirty and the value of the computed Signal is eagerly computed because Angular internally uses a symbol named ***UNSET*** - which indicates that no value has yet been produces and therefore a value has to be computed.

After the initial computation, nothing happens. The first phase starts once we update a writable Signals using either the ***set***, or ***update*** function.

```
count.set(3);
```

At this point a change notification (dirtiness) is synchronously propagated through the graph. Resulting in a situation where the Producer and every Consumer of the Producer becomes dirty.

Dirtiness indicates that something might have changed. In this phase it is never communicated if something changed and what actually changed.

During this phase no side effects are run and no computation function is invoked.

That's everything that happens in first phase. The second phase then kicks off once we start reading our computed ***isEven*** signal.

```
console.log(isEven());
```

The second phase is only executed if we or a template calls our computed Signal again.

In the second phase Producer versions are polled and if necessary Producer values are pulled. This may or may not trigger a recomputation.

In the first phase "dirtiness" is eagerly pushed. In the second phase versions are polled and eventually values are pulled and recomputed.

Okay, so we know that the Push -> Poll -> Pull algorithm is split into two phases. A first one that does the Push and a second one that does the Poll and Pull. Let's now take a closer look at the Polling.

First phase: Push
Second phase: Poll -> Pull

Polling versions

Each Producer is equipped with a ***version***. The ***version*** is incremented whenever a producer produces a semantically new value. The current ***version*** is then stored on the ***ReactiveNode*** whenever a Consumer reads from a Producer.

ReactiveNode is the base class for nodes inside the graph. ***Consumers*** and ***Producers*** are both instances of ***ReactiveNode***.

When a Consumer is invoked, it initiates a process in which it internally begins to pull values from its associated Producers. However, before the actual pulling of values occurs, a preliminary step known as "polling" takes place.

Polling serves as a preparatory action in consumers that helps determine whether it is necessary pull the latest value from a producer.

So before the Consumers trigger their reactive operations, for example rerunning the compute function of a computed Signal, they start polling the ***version*** of a Producer and refresh it if the ***version*** differs from the one they have stored in their current Producer.

If the ***version*** is still the same, no recomputation is needed. If the ***version*** changed, a recomputation and the equality check are executed. If the recomputation produces a new value and it also happens that the Consumer is also a Producer the ***version*** of that Consumer/Producer is incremented as well.

Polling and pulling is a recursive process that happens throughout the graph.

Why not simply poll the ***value*** directly and compare it? Why polling a ***version***?

Well that's a good question. The reason for that is performance. It is very cheap to compare two integers (***version***). But it may be very expensive to compare two objects on equality.

The theory is very abstract and may be hard to grasp. Let's visualize it so that things become clearer.

Push -> Poll -> Pull

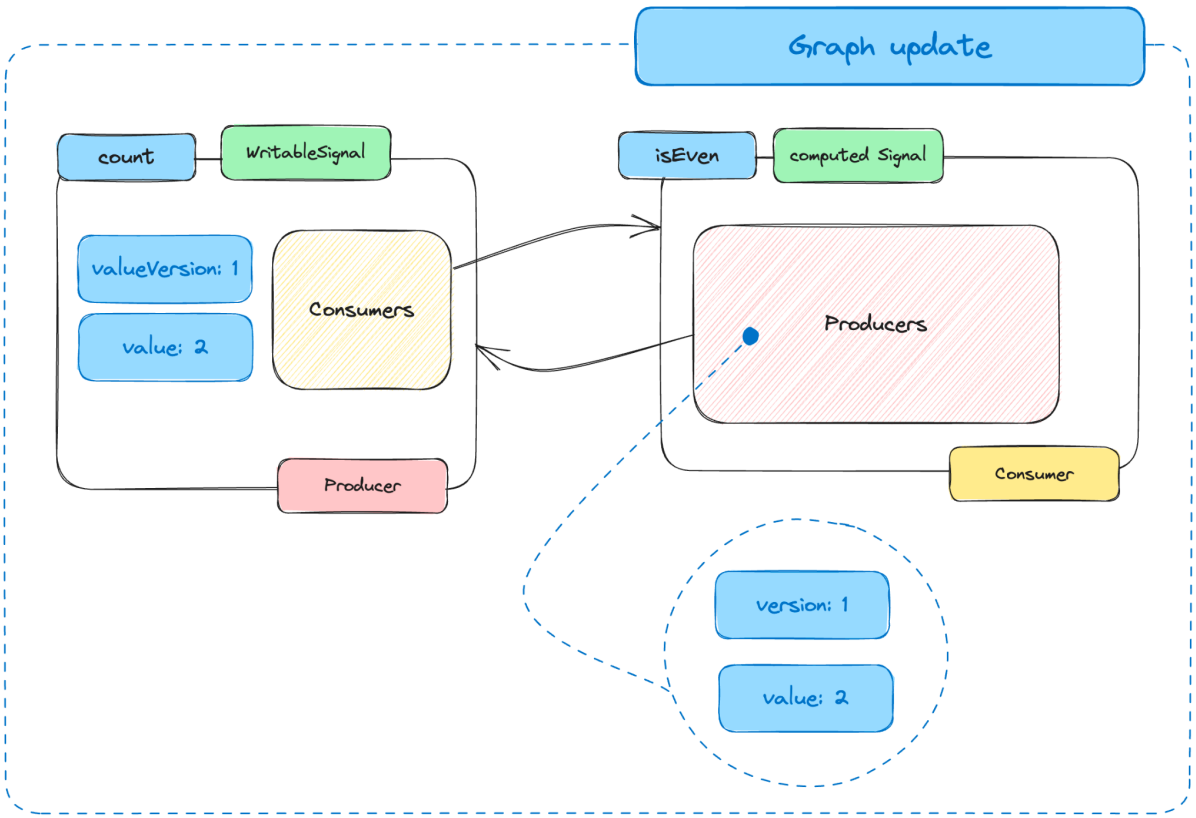
Now that we've established the foundational terminology, it's time to assemble the puzzle pieces into a coherent whole.

Interaction between writable Signals and computed signal

Let's illustrate the concepts from above in more detail by again looking at the following example.

```
const count = signal(2);
const isEven = computed(() => this.count() % 2);

console.log(isEven());
```

After invoking the ***isEven*** function, the initial pull and computation happens. Furthermore, the graph undergoes an update, leading to the following visual representation.

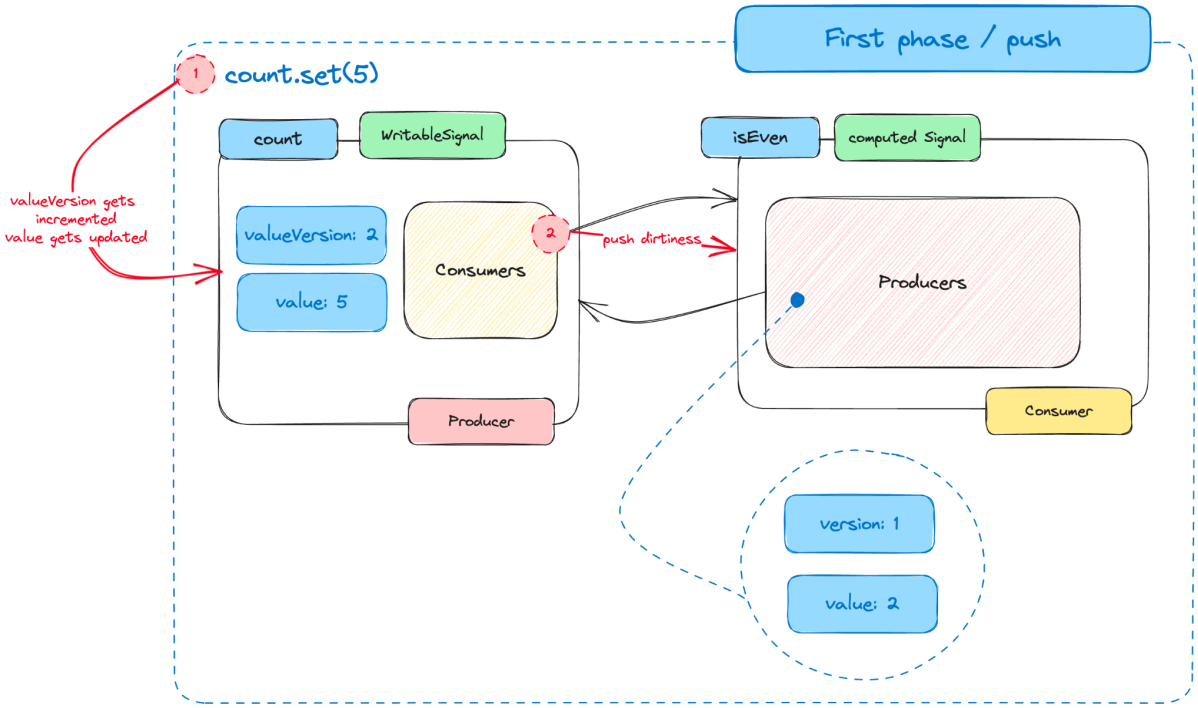


The picture resembles the picture we have already seen in the introduction. But the image contains some important additions. Our count Signal now contains a ***version*** and a ***value*** property. The ***version*** is a counter and therefore starts at ***1***, while the ***value*** is set to the initial value of the Signal.

At this point, we are all set to kick off the Push -> Poll -> Pull algorithm.

```
count.set(5);
```

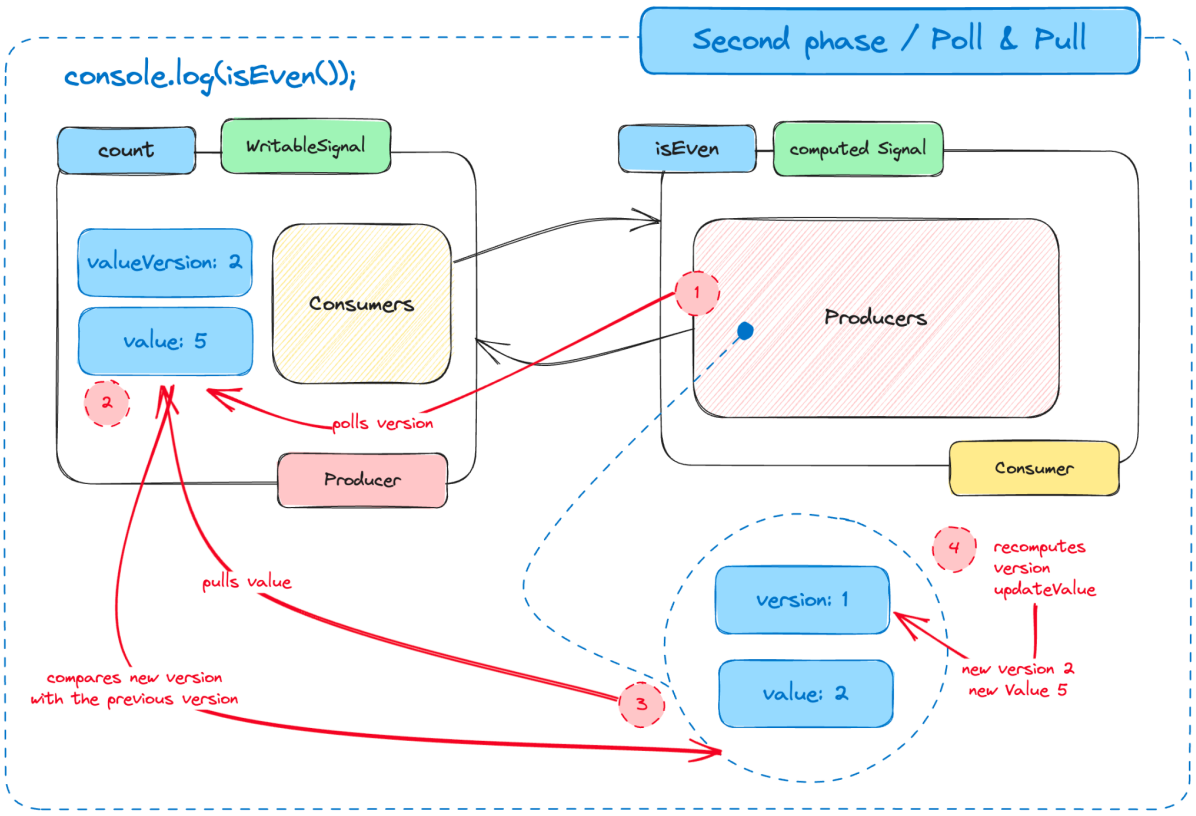
Updating a writable Signal kicks off the push phase. Means the Producer now synchronously starts to push his dirtiness along the graph and updates it's internal value and increments the ***version***.



The second phase then starts once the ***isEven*** computed Signal is read.

```
console.log(isEven());
```

Here we explicitly call the ***isEven*** Signal to retrieve its value. However, in practice Signals are often used in templates and are called once the template is rerun.



During the second phase, the Consumer follows a specific sequence of actions. Initially, it polls the ***version*** of its consumers. Subsequently, it performs a comparison between this ***version*** and the one stored in its dependency edge. In the event that they diverge, the Consumer retrieves the value from the Producer and proceeds to recalculate its own value. Following this computation, it proceeds to update both the ***version*** and the ***value*** attributes on the dependency edge accordingly.

Great. Let's take this concept one step further by adding an effect to the mix.

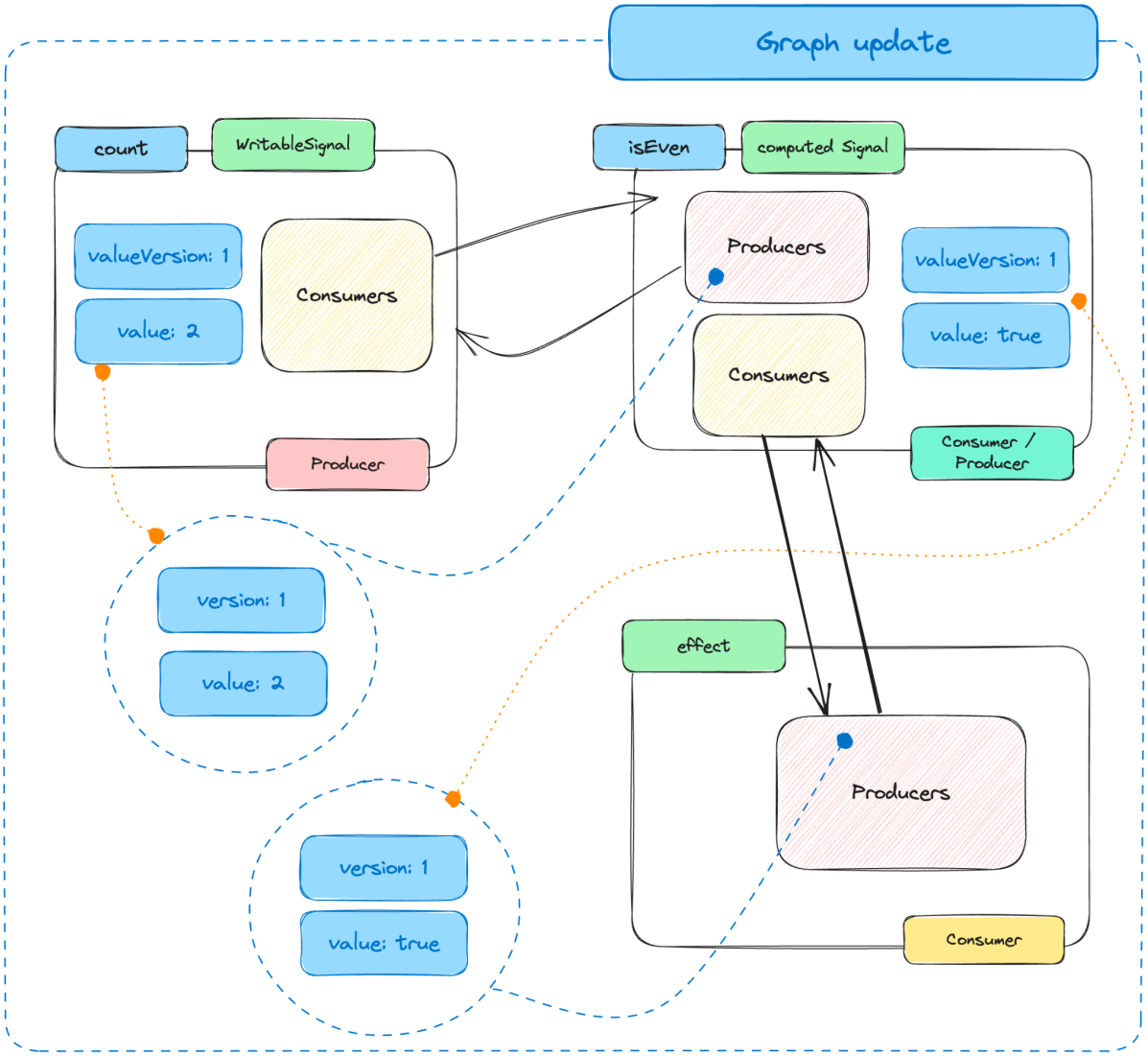
Interaction between writable signal, computed signal and effect

Up to this point, we've explored the interaction between a writable Signal and a computed signal, with the computed Signal primarily functioning as a consumer.

However, as mentioned earlier, there are situations where a computed Signal assumes a dual role—not only as a Consumer but also as a Producer.

```
const count = signal(2);
const isEven = computed(() => this.count() % 2);
effect(() => {
  console.log(isEven());
});
```

In this example, the ***isEven*** computed Signal is utilized within an effect. Consequently, our initial graph configuration appears as follows:

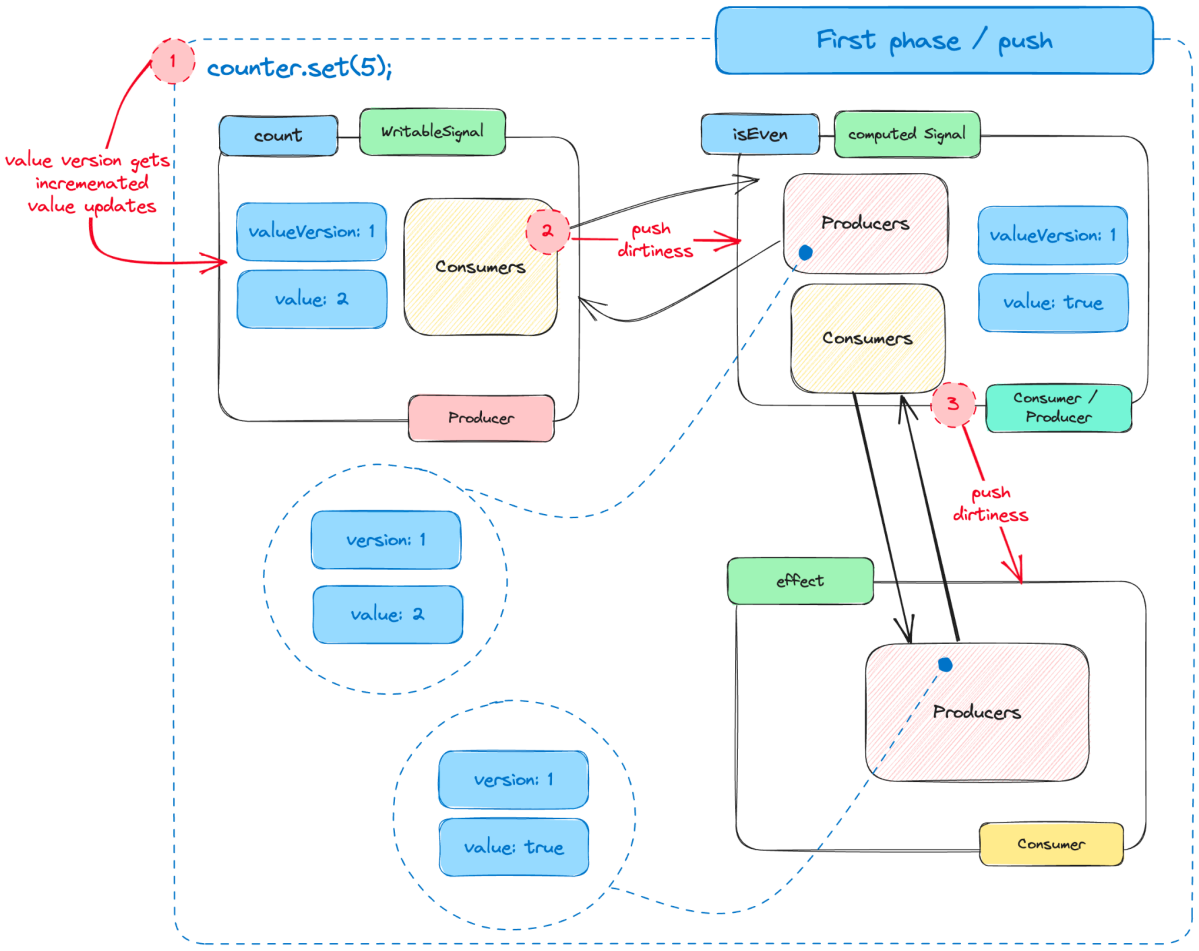


As observed, a new participant has entered the scene—the effect. This effect, serving as a Consumer, similarly references its Producers and maintains the ***version*** and ***value*** on its dependency edge.

A noteworthy aspect in the graphic above is that the ***isEven*** computed Signal has assumed a dual role, functioning both as a Consumer and a Producer. It meticulously monitors references to both Producers and Consumers.

Now that the graph has been established, and all elements are in place, we can commence with phase one. Once again, initiating the initial phase requires us to make a mutation to the writable Signal.

```
counter.set(5);
```



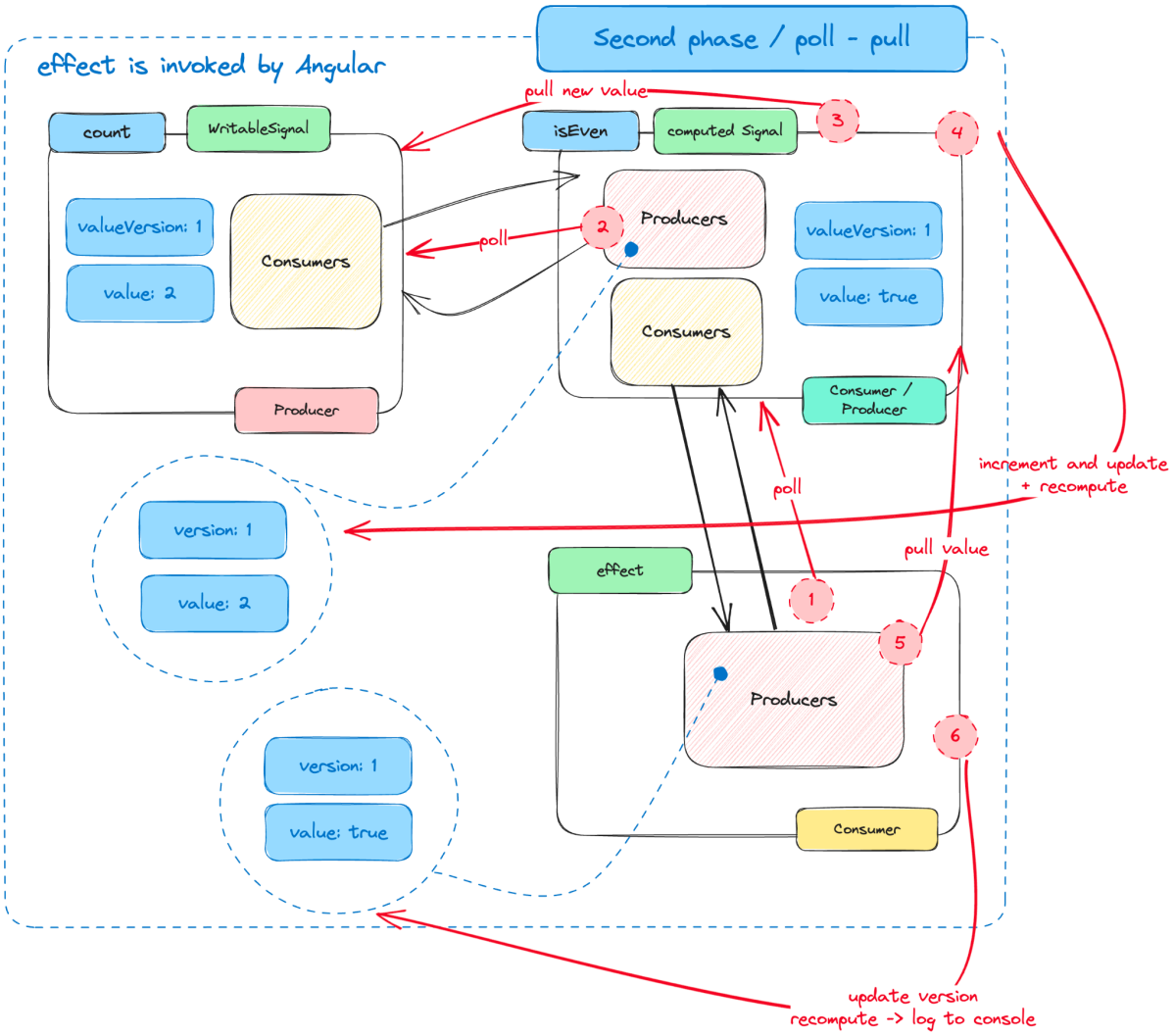
In the push phase, the writable Signal first updates its ***value*** and the ***version***. It then pushes its dirtiness down the graph.

The writable Signal pushes the dirtiness to the computed Signal which then pushes it further to the effect.

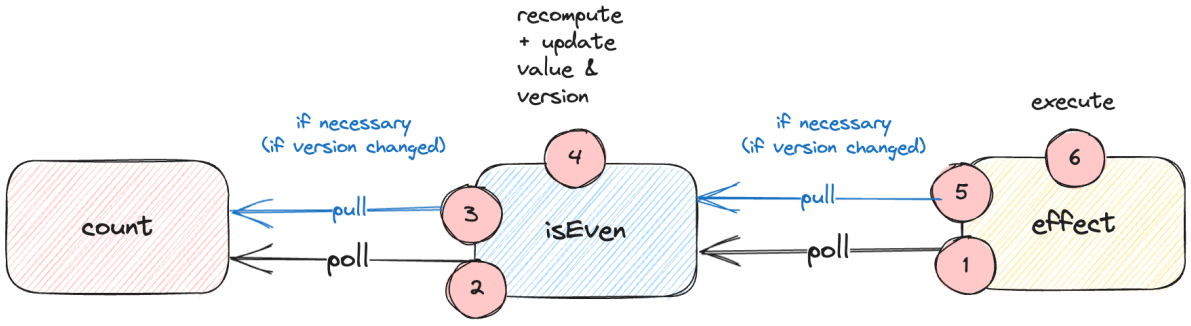
Up to this point, there have been no further actions or recomputations. Progressing to the next phase necessitates the invocation of our effect, which, as we learned, is done by Angular.

Angular might adjust when the effect is called, but the important thing to note here is that somebody actually has to call the effect to kick off the Poll / Pull phase.

Okay, so Angular calls our effect, which kicks off phase two.



There's a lot going on in this picture. Let's simplify this picture to demonstrate the basic flow in a simplified way.



Upon examining this graphic, it becomes evident that the algorithm at play here exhibits recursive behavior. Regardless of the size our chain may reach, the underlying algorithm remains the same.

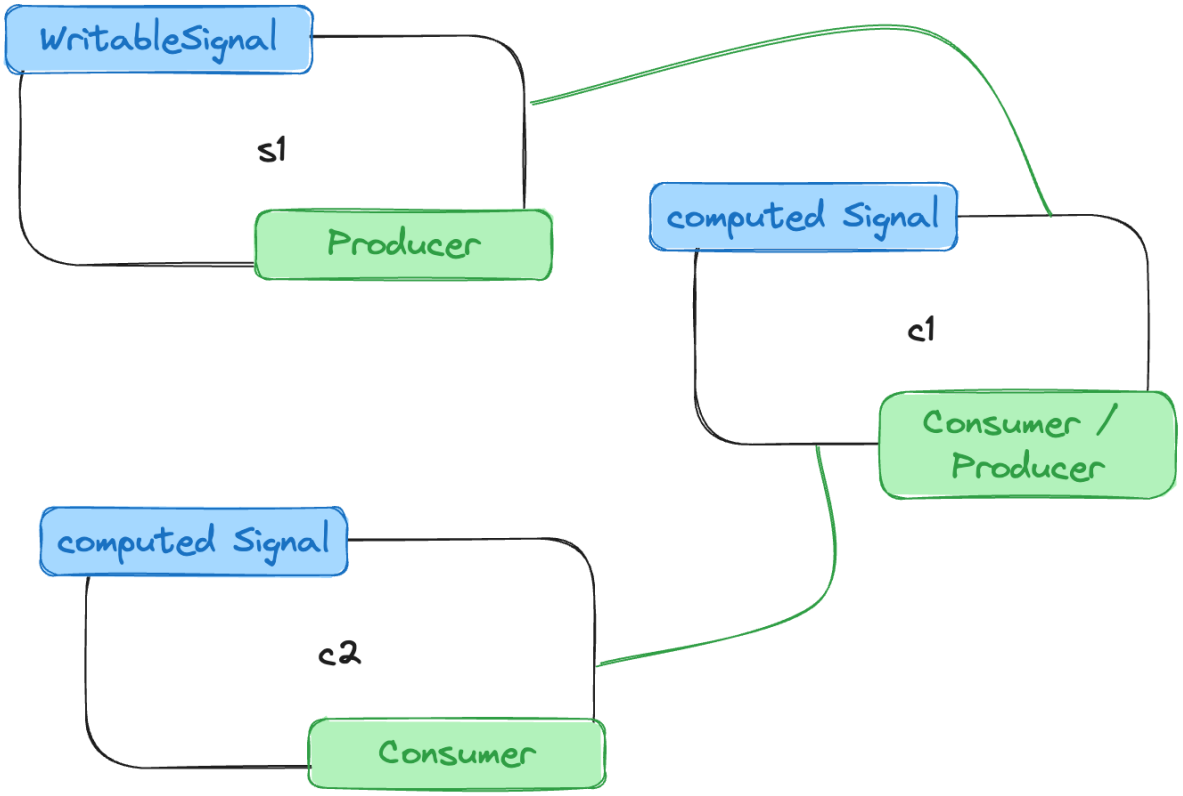
Now that we've gained a more profound insight into the workings of Signals, let's apply this newfound knowledge and examine some interesting Signal code examples.

Applied Push -> Poll -> Pull algorithm

To enhance our understanding of the Push -> Poll -> Pull algorithm, let's fortify our knowledge by exploring some interesting Signal examples.

Initial pull

To begin, let's visualize the setup using a simplified graphic. This visual aid will provide a clear overview of our scenario.



We have a signal ***s1***, which is used inside a computed Signal ***c1***. That computed Signal then again is used inside another computed Signal ***c2***. Pretty straight forward. Let's transform the graphic on top into code.

```
@Component({
  standalone: true,
  selector: 'example-five',
  template: `
    <h1>Example five</h1>
    <button (click)="incrementS1()">Increment s1</button>
    <p>{{ c2() }}</p>
  `,
})
export class MyComponent {
  s1 = signal(0);
  c1 = computed(() => {
    console.log('Calling the computation function of c1');
    return this.s1() + 1;
  });
  c2 = computed(() => {
    console.log('Calling the computation function of c2');
    return `Value of c1 + 2: ${this.c1() + 1}`;
  });

  incrementS1() {
    this.s1.update((v) => v + 1);
  }
}
```

Our component implements the setup drawn above. The components template has a button to increment ***s1*** and furthermore, it displays the value of ***c2*** in the template.

Let's begin by launching our application and examining the log output.

What should we anticipate observing within the log?

Well, if you thought of the Push -> Poll -> Pull algorithm you might have thought that the computation function of ***c1*** is called before the computation function of ***c2***. But actually, that's not what is happening. ***c2*** is called before ***c1***.

```
Calling the computation function of c2
Calling the computation function of c1
```

The initial run holds a unique significance. During this run, eager pulls happen because all the computed Signal values are ***UNSET***. This results in execution of the computation functions down the graph.

UNSET is a dedicated Symbol that is used before a computed value has been calculated for the frist time. It tells Angular that no computation happened so far and that a value has to be eagerly calculated.

Now, let's take a moment to review the log outputs when we click on the "increment" button.

Which log would you expect? Which computation function is first called? ***c1*** or ***c2***?

```
Calling the computation function of c1
Calling the computation function of c2
```

The ***c1*** is now called before ***c2***. This now refers to the Push -> Poll -> Pull algorithm. Here's what's happening:

1. **User Interaction:** When a user clicks on the increment button, the application triggers the ***update*** method, which in turn modifies the value of ***s1***. This modification initiates the process of pushing "dirtiness propagation" throughout the graph.

2. **Zone.js and Change Detection:** ***Zone.js*** detects the click event and Signals Angular to perform change detection. Angular responds by reruning our template.

3. **Rerun of the template:** Within the template, there's a reference to ***c2***, which prompts its invocation as part of the change detection process.

4. **Polling Phase (c2):** By reruning the template, ***c2*** enters a polling state. It checks the status of ***c1***, which in turn polls ***s1***.

5. **Pulling Data (c1 and s1):** ***c1*** knows that the ***version*** of ***s1*** changed and therefore pulls the current value of ***s1***. Subsequently, ***c1*** recomputes (invokes the computation function) its value and updates its ***value*** and increments the ***version***.

6. **Data Propagation (c2):** After ***c1*** has completed its update, ***c2*** knows that the ***version*** changed and therefore pulls the newly computed value of ***c1***.

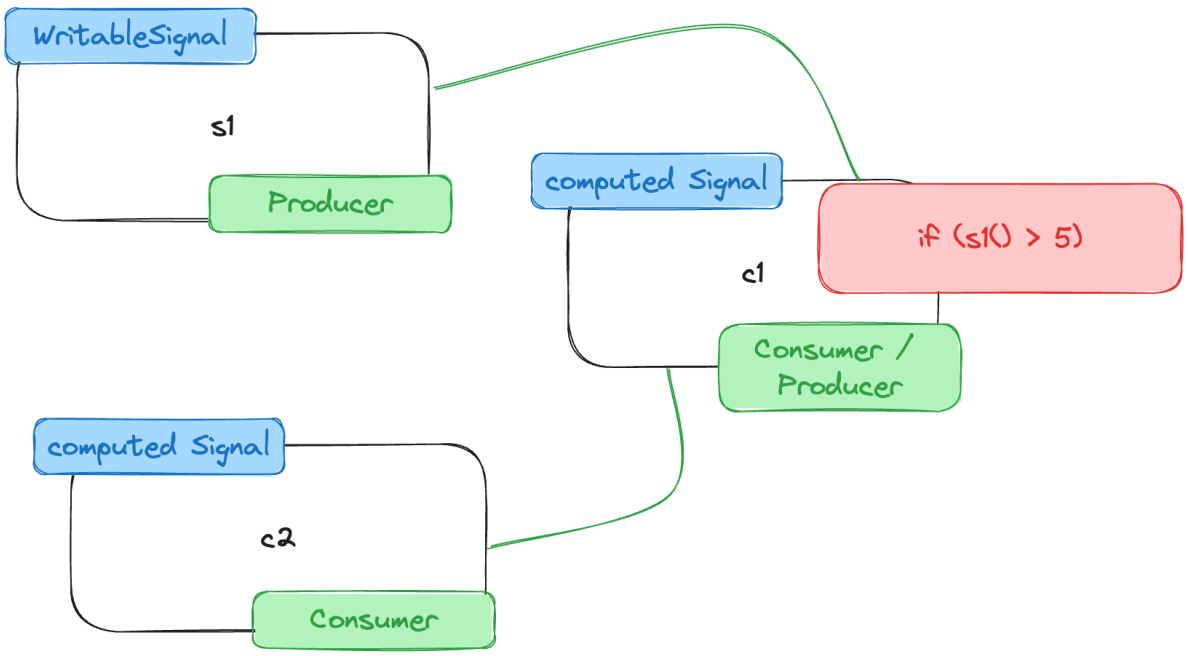
7. **Update:** The updated value is displayed to the user.

This ordered sequence of operations reflects why the computation function of ***c1*** is executed before the computation function of ***c2***.

Lazy computation

Let’s reuse the example above but let's change one small thing. Let’s add a condition to the computation function of ***c1***.

```
const c1 = computed(() => return this.s1() > 5);
```



With this code ***c1*** emits ***false*** as long as the value of ***s1*** is smaller than ***5***. Let’s take a look at the complete component code.

```
@Component({
  standalone: true,
  selector: 'my-component',
  template: `
    <h1>Example one</h1>
    <button (click)="incrementS1()">Increment s1</button>
    <p>{{ c2() }}</p>
  `,
})
export class MyComponent {
  s1 = signal(0);
  c1 = computed(() => {
    console.log('Calling the computation function of c1');
    return this.s1() > 5;
  });
  c2 = computed(() => {
    console.log('Calling the computation function of c2');
    return `Is bigger than 5: ${this.c1()}`;
  });

  incrementS1() {
    this.s1.update((v) => v + 1);
  }
}
```

Let's bypass the initial log statements and focus on the log generated after clicking the Increment button.

Which log would you expect? Which computation function do you think runs first this time? *c1* or *c2*?

```
Calling the computation function of c1
```

This time only the computation function of *c1* runs. The reason *c2's* computation function doesn't execute at all is because *c1* never produced a new value since the value of *s1* remains less than *5*. Consequently, when

c2 checks for changes during its polling, it recognizes that nothing has changed, and as a result, it doesn't need to perform a recomputation.

Which log output would you expect if we restart our application and click the button 6 times?

Let's go through the log step by step. Initially all computed Signals start dirty and values are eagerly pulled. This results in the following log output:

```
Calling the computation function of c2
Calling the computation function of c1
```

After that we start clicking 5 times on our button.

```
Calling the computation function of c1
Calling the computation function of c1
Calling the computation function of c1
Calling the computation function of c1
Calling the computation function of c1
```

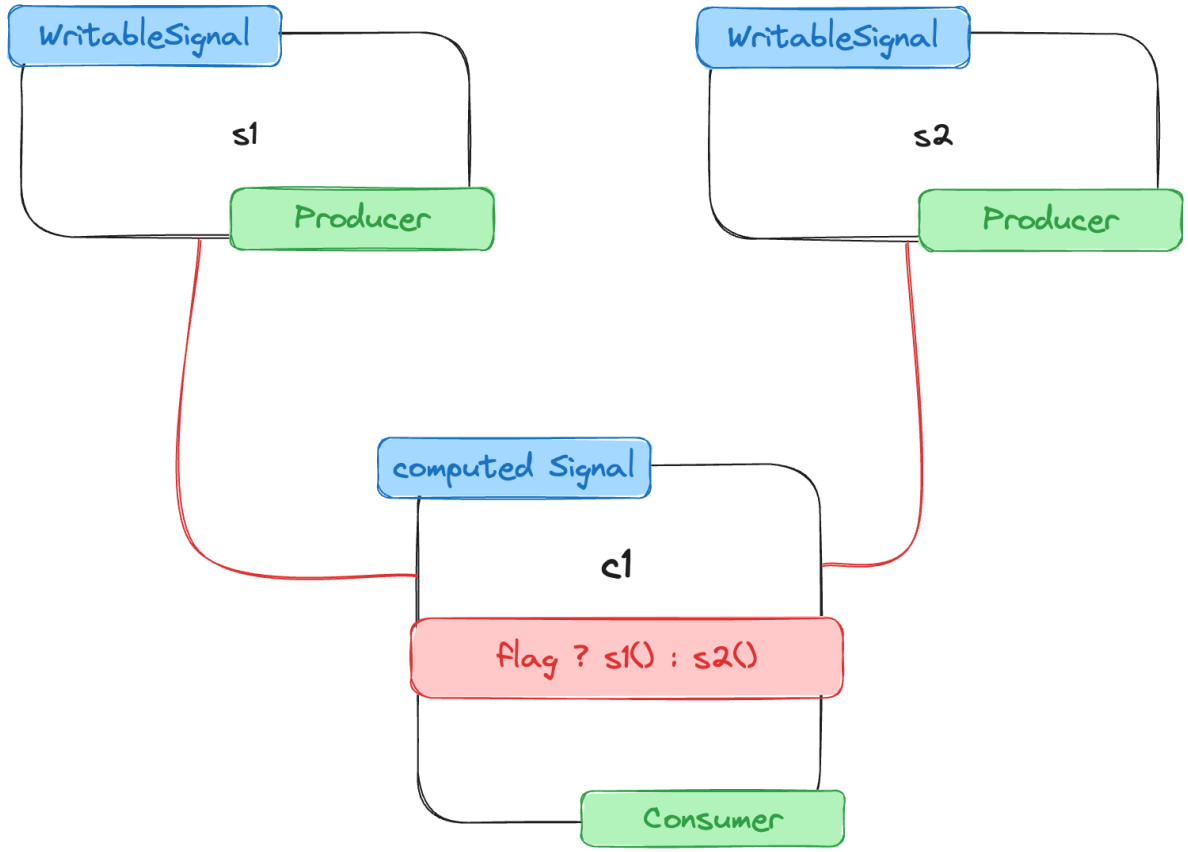
For each button the computation function of *c1* is executed. But since *c1* doesn't result in a new value, *c2* doesn't need to be executed. The value of *c1* is still false since the value of *s1* is *5* at this point.

But the next click is interesting, since *s1* will be incremented to *6*. Therefore *c1* recomputes and this time *c1* also produces a new value which causes *c2* to recompute as well.

```
Calling the computation function of c1
Calling the computation function of c2
```

Dynamic dependency graph

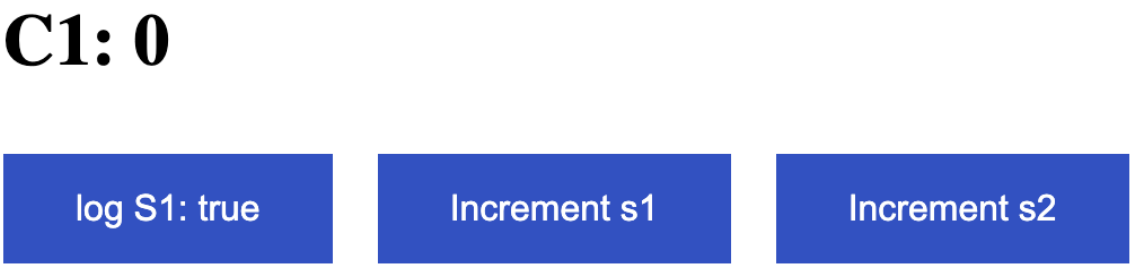
Let's take a look at another interesting example.



In this example we have two Signals, ***s1*** and ***s2***. Both are consumed by a computed signal ***c1***. The interesting thing is that ***c1*** contains a condition that uses a flag to determine whether we log the value of ***s1*** or ***s2***.

```
@Component({
  standalone: true,
  selector: 'my-component',
  template: `
    <h1>{{ c1() }}</h1>
    <button (click)="logS1 = !logS1">
       Toggle log S1: {{logS1}}
    </button>
    <button (click)="incrementS1()">Increment s1</button>
    <button (click)="incrementS2()">Increment s2</button>
  `
})
export class MyComponent {
  logS1 = true;
  s1 = signal(0);
  s2 = signal(0);

  c1 = computed(() =>
    this.logS1 ? this.s1() : this.s2()
  );

  incrementS1() {
    this.s1.update((v) => v + 1);
  }

  incrementS2() {
    this.s2.update((v) => v + 1);
  }
}
```

In the template we render ***c1***. We also have buttons to increment ***s1***, ***s2*** and to toggle the ***logS1*** flag. The initial state of our application looks like this.

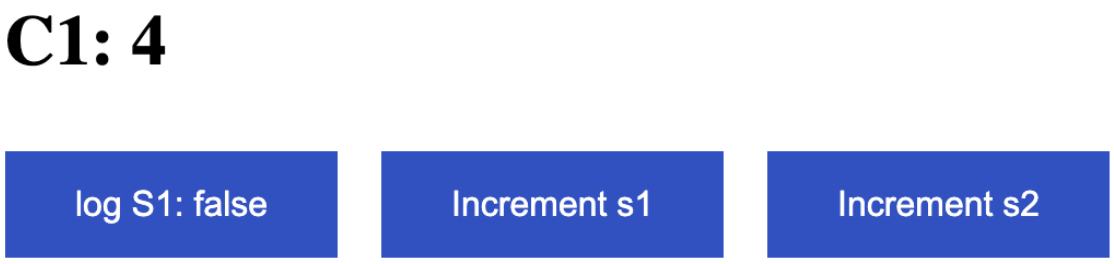


Let’s go ahead and click a bunch of times on the “Increment s1” button to bump up our ***s1*** signal. And, everything works as expected, the displayed counter in our template goes up. Wonderful.

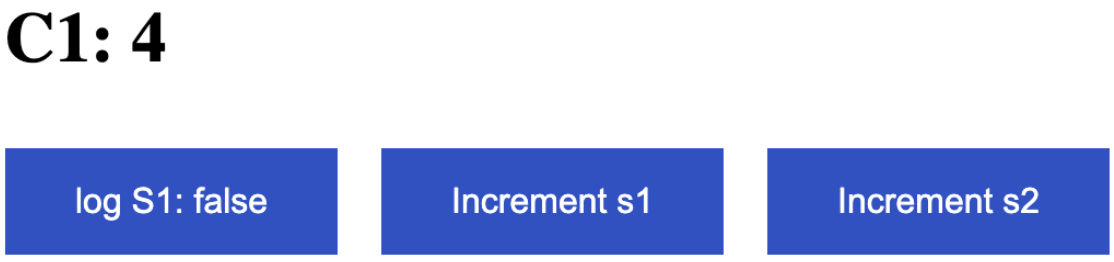
Now, let's explore what happens when we click the "Toggle log S1" button followed by the "Increment S2" button.

> **Take a moment to think about what you would expect to happen at this point?**

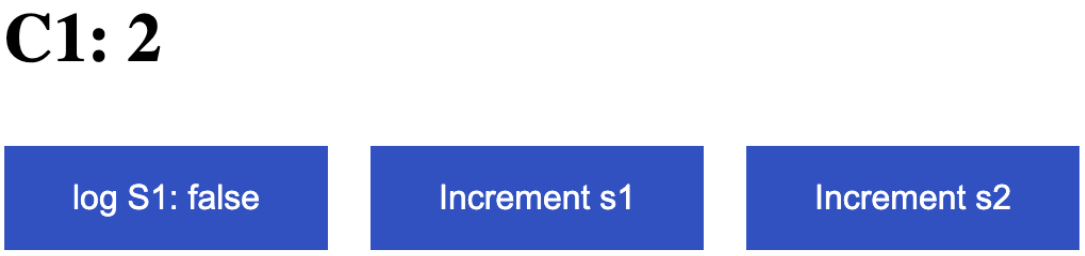
In most cases, developers would expect to see a value of ***C1: 1*** displayed. However, let's observe what actually occurs to gain a better understanding of the behavior.



After clicking the first button, we notice that the flag has indeed changed. However, it's noteworthy that the counter remains at four, despite clicking the "Increment s2" button. This might prompt us to wonder whether it was a misclick. To clarify, let's go ahead and click "Increment s2" once more.

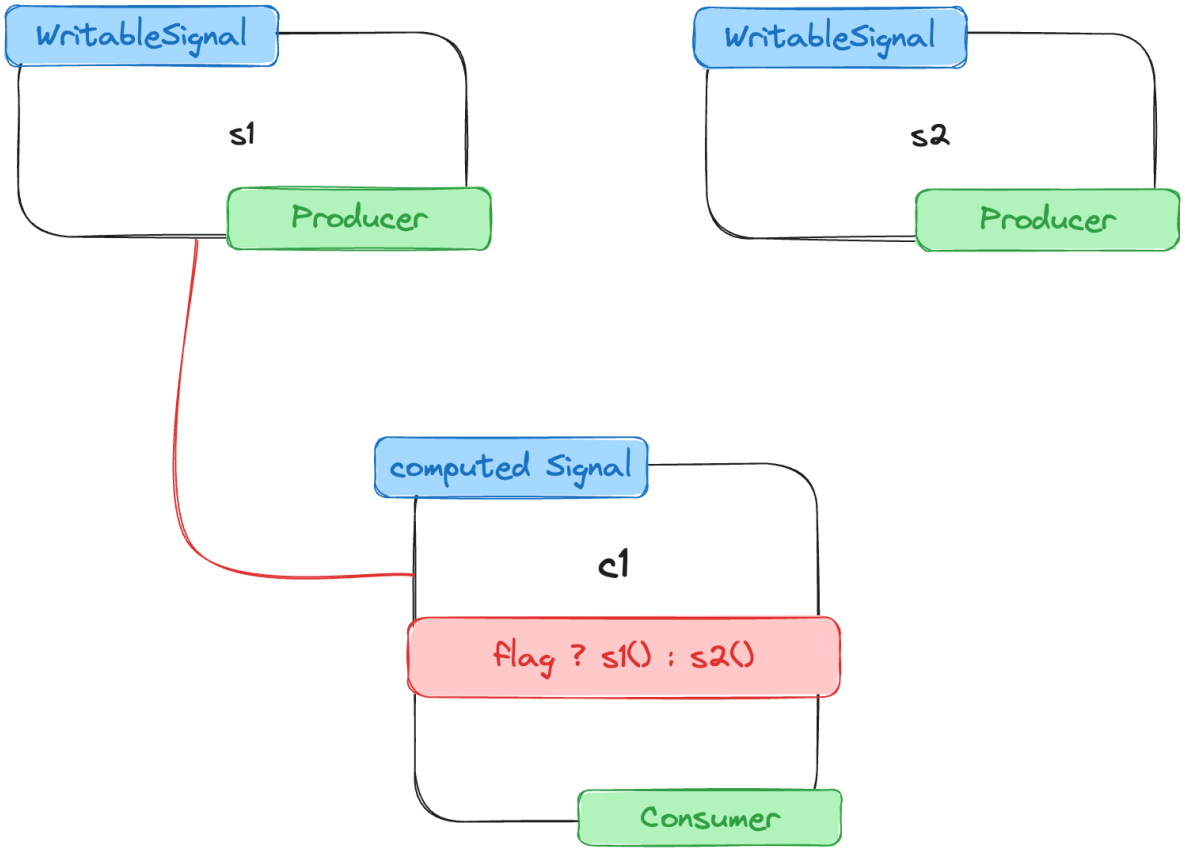


Despite the previous actions, it seems the situation hasn't changed. Now, let's consider what happens when we click on the "Increment s1" button.



What?! Now it logs, `C1: 2`, which is the value of `s2`. What?! It may seem surprising at first, but there's a rationale behind this behavior. Let's walk through the logic to understand what's happening here.

When we first call our computed Signal `c1` the dependency graph is constructed. At this point the `logS1` flag is set to `true`, which means that `s1` is part of the graph but `s2` not. So our initial graph looks like this.



c1 only knows about one producer, ***s1***. You would expect ***s2*** to become a Consumer when toggling the ***logS1*** flag, but it doesn't. Why might this be the case? Let's delve into the reasons behind this unexpected behavior.

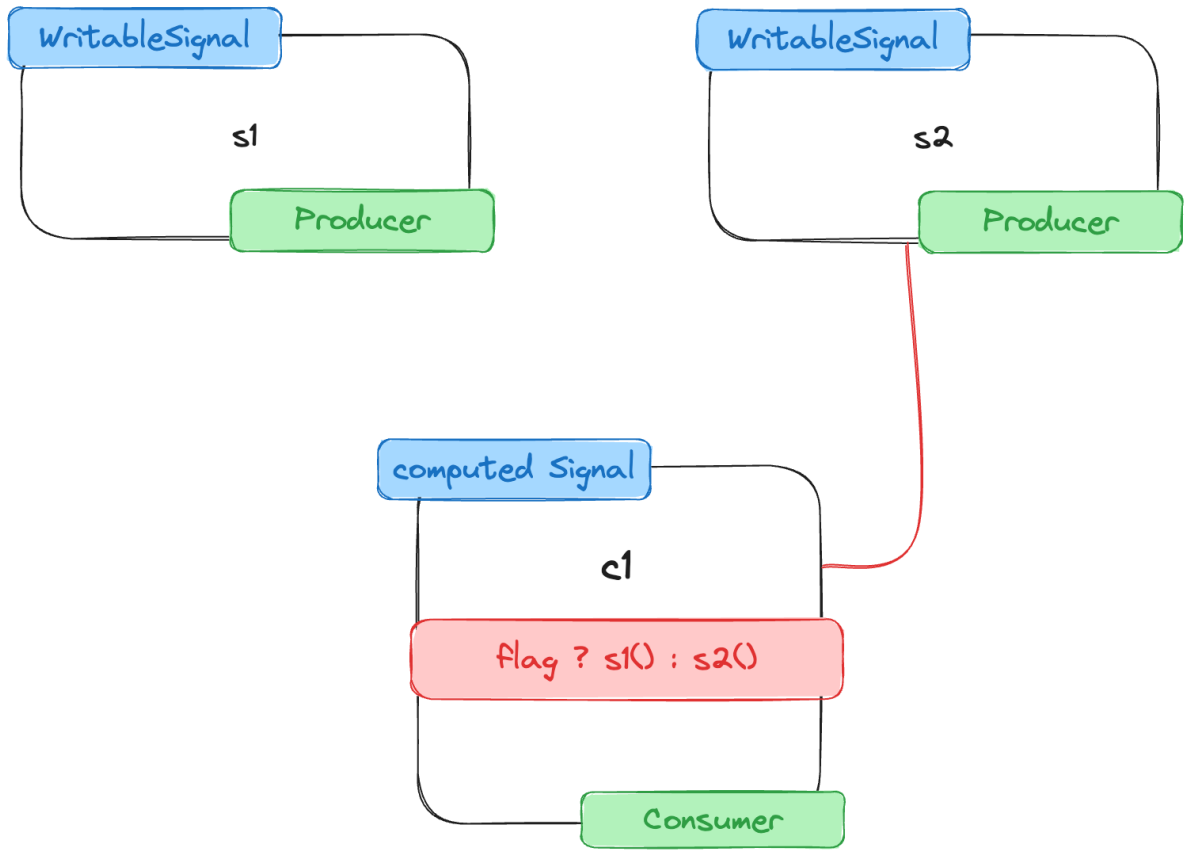
The key reason is that at this point, there is still only one Consumer, which is ***s1***. And this Consumer hasn't produced a new ***version*** during its polling process. This lack of a new ***version*** from the Consumer ***s1*** is why ***s2*** doesn't become a Consumer despite toggling the ***logS1*** flag.

> **Computations are lazy and only executed initially or if the *version* of a Producer changed during a poll.**

In this scenario, even though ***s2*** is capable of producing new values, ***c1*** doesn't recognize it as a Producer because it hasn't been explicitly registered as one. To include ***s2*** as a Producer and trigger computations, you need to invoke a computation by incrementing the value of ***s1***. This

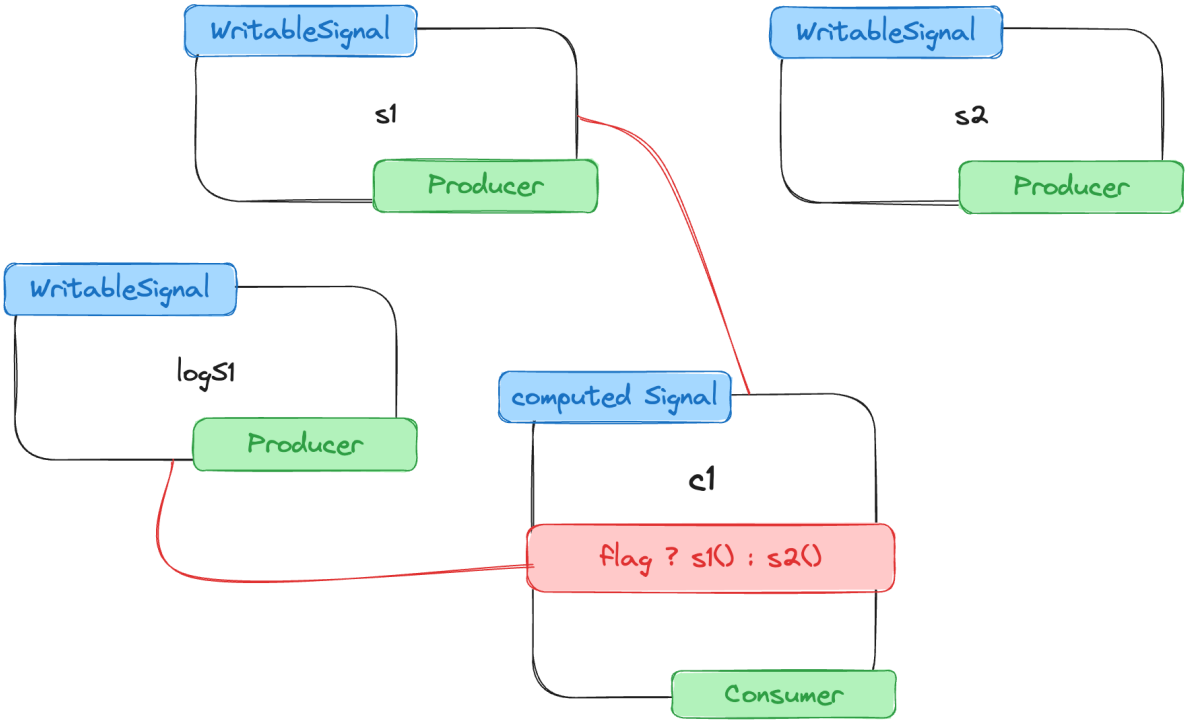
will establish the necessary connection and enable ***c1*** to consider ***s2*** as a Producer.

When we increment ***s1***, the ***version*** is incremented, and a new value is produced. Subsequently, this new value is polled and pulled by ***c1***, leading to an actual recomputation. This recomputation also updates the dependency graph, resulting in the following state:



After this series of events, we end up with the inverse situation where ***s2*** is now a Consumer of ***c1***, but ***s1*** is no longer a Consumer.

How can we improve this behavior? Well what about we change our primitive boolean flag to a Signal. How would that impact the situation? Let's go ahead and update our graphic and refactor our code.

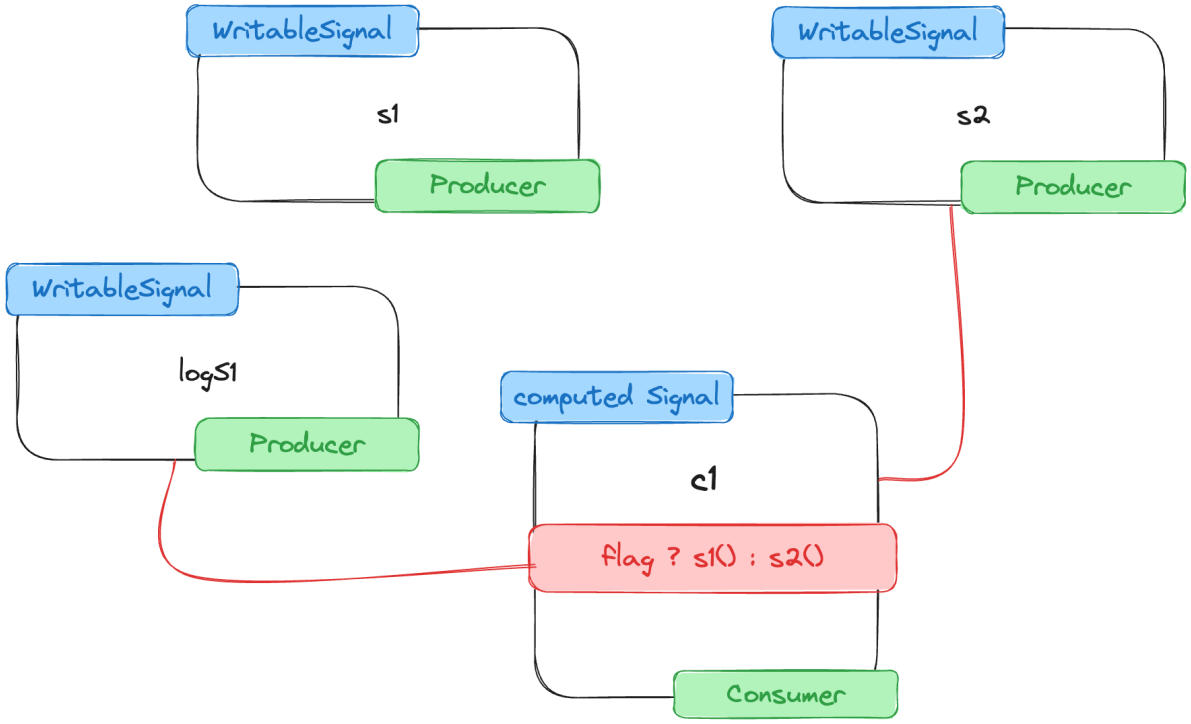


As we can see our computed signal ***s1*** now has a new additional Producer which is ***logS1***. Now let's focus on the situation where we click on the toggle button to update the value of ***logS1***.

Once we click, the push phase happens where ***logS1*** produces a new ***value,*** increments its ***version*** and eagerly pushes dirtiness.

Again, ***Zone.js*** kicks in, notifies Angular that something changed, template reruns, our computed Signal gets called and we enter stage two.

We now poll and pull the latest value which now results in a recomputation. This recomputation updates the graph to look like this.



If you access a value that changes inside a computed Signal it should be a Signal as well.

If you are curious on how the code looks like with a Signal flag, here's the final component code.

```
@Component({
  standalone: true,
  selector: 'example-three',
  template: `
    <h1>{{ c1() }}</h1>
    <button (click)="toggleLogS1()">
       Log S1: {{logS1()}}
    </button>
    <button (click)="incrementS1()">Increment s1</button>
    <button (click)="incrementS2()">Increment s2</button>
  `
})
export class MyComponent {
  logS1 = signal(true);
  s1 = signal(0);
  s2 = signal(0);

  c1 = computed(() =>
    this.logS1 ? this.s1() : this.s2()
  );

  toggleLogS1() {
    this.logS1.update(v => !v);
  }

  incrementS1() {
    this.s1.update((v) => v + 1);
  }

  incrementS2() {
    this.s2.update((v) => v + 1);
  }
}
```

Now that we have seen many examples that use computed Signals let's take a look at an example that uses effects.

Effect execution

Let's look at a simple component that uses an ***effect*** inside the ***constructor***.

```
@Component({
  /* ... */
})
export class EffectExampleComponent {
  constructor() {
    const counter = signal(0);

    effect(() => {
      console.log('Effect runs with: ', counter());
    });
  }
}
```

Would you expect a log output if we go ahead and run our application?

If we go ahead and start up this Angular application we indeed get a console output:

```
Effect runs with: 0
```

This output is a result of the effect being executed for the first time. When the effect is created, its state is value is set to ***UNSET*** which causes it to pull the initial values from the producer Signals referenced in the effect implementation. In this case, the effect runs its computation and logs "Effect runs with: 0" to the console.

Effects are scheduled and called by Angular. This is a significant difference to computed Signals which are completely lazy and will not execute until we explicitly call them.

In summary, when the Angular application starts, the given component will trigger the effect to run with the initial value of ***0*** and produce the console output "Effect runs with: 0".

Let's take a look at a more sophisticated example where we not only print out the Signal value inside the ***constructor*** but also change and update the Signal value.

```
@Component({
  /* ... */
})
export class EffectExampleComponent {
  constructor() {
    const counter = signal(0);

    effect(() => {
      console.log('Effect runs with: ', counter());
    });

    counter.set(1);
    counter.set(2);
    counter.update((current) => current + 1);
    counter.update((current) => current + 1);
  }
}
```

In the current implementation, the constructor of the component is updating the value of the counter Signal using both the ***set*** and ***update*** methods synchronously.

As discussed earlier, this will cause the producer ***counter*** Signal to push multiple notifications to the effect of acting as a consumer, indicating that its value might have changed, but not the actual values themselves.

Which log output do you expect if we run the example above?

```
Effect runs with: 4
```

The reason for this is that all the Producer updates will occur before the first and only evaluation of the effect which is scheduled by Angular in microtask queue which happens after sync execution of the ***constructor***.

To summarize, even though there are multiple updates to the ***counter*** Signal, the effect is triggered only once with the microtask queue which happens after all updates have been processed.

To make the explanation more concise:

The component's constructor updates the counter Signal using ***set*** and ***update*** methods synchronously. However, as the effect is marked as dirty and runs after the ***constructor*** is executed, it will be triggered only once with the final value, resulting in a single line of output in the console:

```
Effect runs with: 4
```

Signals and RxJS

We talked a lot about Signals as a new reactive primitive. But, actually Angular already provides reactivity with the help of RxJS. So why not use RxJS as the reactive primitive?

RxJs drawbacks

While Observables are a foundational and powerful primitive for expressing computation functionally and reactively, this chapter delves into why they are not the ideal fit for template rendering in Angular.

We will explore the challenges and limitations of using Observables in this context and why Angular's reactivity system has evolved to embrace a different approach.

- **The Asynchronous Nature of Observables:** Observables, in their nature, can model both synchronous and asynchronous data flow. While this flexibility is advantageous for many use cases, it creates challenges when dealing with template rendering, which requires immediate and consistent data access. Observables in template bindings often lead to "pending" states.

- **Side-Effects and Cold Observables:** Observables often involve side-effectful behavior, enabling them to handle complex data flows and asynchronous tasks such as HTTP requests. However, this introduces challenges when using Observables for template rendering, as multiple subscriptions may inadvertently trigger redundant operations. Often we use ***`share`*** or ***`shareReplay`*** operators to mitigate such problems.

- **Glitches in Reactive Systems:** Glitches occur when a calculation or reaction observes an inconsistent intermediate state. RxJS Observables may exhibit glitches in synchronous operations, leading to undesirable consequences in the UI.

While Observables are a powerful and essential primitive for reactive programming, their application in Angular's template rendering poses specific challenges and limitations.

Due to the reasons listed above Angular decided to introduce a new reactive primitive called Signals. Observables are still used and will remain to be used in many use cases and even many of Angular's core APIs will still return an Observable.

Therefore we need a way to seamlessly integrate Signals and Observables. Let's take a look at what the interoperability between Signals and Observables will look like.

RxJS interoperability

In this chapter, we will explore the new APIs introduced in Angular to facilitate the interoperability between Angular Signals and RxJS - ***`toSignal`*** and ***`toObservable`***.

These functions act as bridges that allow developers to convert Observables to Angular Signals and vice versa, enabling seamless integration between the two paradigms. We will delve into the usage of these functions, understand their signatures, and explore how they handle asynchronous and synchronous data flows.

toSignal - Converting Observables to Angular Signals

The ***`toSignal`*** function is used to convert an Observable to an Angular Signal. It offers two different signatures, each supporting a different way of dealing with the initial value:

- Synchronous emit
- Asynchronous emit

Let's start by looking at the synchronous emit.

Synchronous Emit

```
const mySignal = toSignal(myStream, {
  requireSync: true
})
```

When an Observable is known to emit synchronously, the ***toSignal*** function can be configured to require synchronous emits. This means that the Observable must produce a value immediately, and an initial value doesn't need to be provided.

If the Observable later becomes asynchronous, *toSignal* will throw an error.

Asynchronous Emit

```
const mySignal = toSignal(myStream, {
  initialValue: 'loading'
})
```

A Signal always has an initial value. Observables that emit asynchronously do not provide an initial value. Therefore the ***toSignal*** function requires us to provide an initial value for the returned Signal. If not explicitly provided, the initial value is set to ***undefined***, similar to how the ***async*** pipe returns ***null*** initially.

Managing the Subscription

The ***toSignal*** function internally subscribes to the given Observable and updates the returned Signal whenever the Observable emits a value. Unlike lazy subscriptions, this subscription is created immediately to avoid side effects when reading the signal for the first time.

When converting Observables to Signals, Angular automatically handles the cleanup of the Observable subscription.

Under the hood it uses the ***DestroyRef***. Therefore the same restrictions that we encountered for the usage of an ***effect*** also apply here. ***toSignal*** can only be used inside an injection context.

If ***toSignal*** is used in a component or directive, the subscription is cleaned up when the component or directive is destroyed. For services, the subscription is cleaned up when the service's injector is destroyed.

Error and Completion States

An Observable has three types of notifications when an Observer subscribes to it: ***next***, ***error***, and ***complete***. The value of a Signal is directly linked to the ***next*** notification of the Observable.

When the Observer created by ***toSignal*** is notified of an error, it will throw this error the next time the Signal is read. This error can be handled the same way any other error coming from a Signal would be.

Signals do not have a concept of being complete as they are just wrappers around a value.

However, if the completion state is meaningful for an Observable, it can be represented in a different Signal using operators like ***finalize***.

```
@Component({
  template: `{{ completed() }}`,
})
export class App {
  myObservable = of('hello world').pipe(
    delay(2000),
    finalize(() => this.completed.set(true))
  );
  completed = signal(false);
  mySignal = toSignal(this.myObservable);
}
```

toObservable - Converting Angular Signals to Observables

The ***toObservable*** function serves the opposite purpose, converting an Angular signal to an Observable:

```
const myStream = toObservable(mySignal);
```

This function allows us to interoperate with existing RxJS code that consumes Observables. The resulting Observable will emit values whenever the signal changes its value.

Lifecycle and Cleanup

Unlike the ***toSignal*** function, which automatically cleans up its subscription when the enclosing component or service is destroyed, ***toObservable*** takes a different approach to manage the lifecycle of the effect.

When an Observable created by ***toObservable*** is subscribed to, it creates an effect to monitor the Signal, and this effect remains active until the subscriber unsubscribes.

This design decision was made to accommodate cases where the incoming Signal may have a different lifecycle than the component using it. Signals remain valid as long as someone holds a reference to them, and they might be passed in externally with lifecycles independent of the component.

Additionally, Observables created by ***toObservable*** might be used in external RxJS computations, and services, or persisted for longer periods. By not tying the resulting Observable to the component's lifecycle, developers have more flexibility and control over managing their subscriptions and side effects.

If developers desire to tie the resulting Observable to the component's lifecycle manually, they can do so using RxJS operators like ***takeUntil***, as demonstrated in the following example:

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

// Inside the component class
const myValue$  = toObservable(myValue)
  .pipe(
    takeUntil(this.destroy$)
  );
```

Asynchronously delivered values

One important aspect to consider when using Observables created by ***toObservable*** is that all emissions will be delivered asynchronously. The reason for this behavior lies in the underlying implementation of the effect, which inherently introduces asynchronicity.

Let's consider the following example where we have a ***updateSubjectValue*** method and a ***updateSignalValue*** method. We also have a ***mySubject*** which is a ***BehaviourSubject***. Furthermore we have a ***mySignal*** Signal which we then use in combination with the ***toObservable*** operator to create a ***myObservable*** Observable. Inside the ***constructor*** we then subscribe to the ***mySubject*** as well as the ***myObservable*** and log out their values.

```
mySubject = new BehaviorSubject(0);
mySignal = signal(0);
myObservable = toObservable(this.mySignal);

constructor(){
  this.mySubject.subscribe(
    e => console.log(`Subject emits new value: ${e}`)
  );
  this.myObservable.subscribe(
    e => console.log(`MyObservable emits new value: ${e}`)
  );
}

updateSubjectValue() {
  this.mySubject.next(this.mySubject.value + 1);
  this.mySubject.next(this.mySubject.value + 1);
  this.mySubject.next(this.mySubject.value + 1);
}

updateSignalValue() {
  this.mySignal.update(v => v + 1);
  this.mySignal.update(v => v + 1);
  this.mySignal.update(v => v + 1);
}
```

If we run this application we would initially log the value of ***mySubject*** as well as the value of ***myObservable***.

```
Subject emits new value: 0
MyObservable emits new value: 0
```

Which log output would you expect if we invoke the ***udpateSubjectValue*** method.

We would get three values emitted.

```
Subject emits new value: 1
Subject emits new value: 2
Subject emits new value: 3
```

Which log output would you expect if we invoke the *updateSignalValue* method.

In this case we would only get one log with the final value.

```
MyObservable emits new value: 3
```

If a Signal is set multiple times in quick succession, the Observable created from it will only emit the final value due to the asynchronous nature of the effect. In contrast, a ***BehaviorSubject*** emits synchronously and produces all intermediate values. This is an important distinction to keep in mind when working with Signals and Observables interchangeably.

To obtain the first value synchronously, developers can use the ***startWith*** operator to set an initial value for the Observable, as shown in the example below:

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

const obs$ = toObservable(mySignal)
  .pipe(
    startWith(mySignal())
  )
);
```

One Effect for All Observers (via shareReplay)

When an Observable produced from a Signal is expected to have many subscribers, using the ***shareReplay*** operator is recommended. This operator avoids creating a new effect for each subscriber, which can improve performance.

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

const obs = toObservable(mySignal).pipe(
  shareReplay({ refCount: true, bufferSize: 1 })
);
```

Using ***shareReplay*** ensures that the underlying effect is cleaned up after all users unsubscribe, and the ***bufferSize*** of ***1*** ensures that only the latest value is cached.

The ***toObservable*** function in Angular provides a powerful tool for bridging the gap between Angular Signals and RxJS Observables. By converting Signals to Observables, developers can seamlessly integrate reactive programming paradigms in their applications, leveraging the strengths of both approaches.

Understanding the differences in lifecycle management, synchronous/asynchronous emissions, and techniques for optimization will empower developers to make informed decisions when working with Observables created from Angular Signals. With this knowledge, you can design robust and efficient data flow management in your Angular applications.

Will Signals replace RxJS?

When Signals were first introduced, many developers were wondering if Signals will replace RxJs. Well, not replace, but for certain use cases Signals might be enough and therefore enable us to write Angular apps without RxJs.

The Angular team recently mentioned that Observables might become optional at some point in the future.

While Signals bring some benefits to the table, there are still some use cases where RxJs offers tons of benefits.

Event handling complexity

One of the fundamental strengths of RxJS lies in its ability to handle events from various sources, effortlessly mapping and filtering them as needed. While Signals can handle similar tasks, they often result in less readable and more complex code compared to RxJS. This complexity arises from the absence of a comprehensive set of operators that RxJS offers.

RxJS boasts a rich library of operators that empower developers to manipulate and transform data streams with ease. These operators make code more expressive and maintainable, reducing the cognitive load when dealing with intricate event handling scenarios.

Network Requests and Asynchronicity

Handling network requests is a common requirement in modern web applications. RxJS shines in this domain due to its ability to manage asynchronous operations effectively. XHR (XMLHttpRequest) requests, a common method for making HTTP requests, are inherently asynchronous. Observables in RxJS allow developers to elegantly manage the entire lifecycle of these requests, including success, error, and completion events.

Moreover, RxJS provides critical operators like ***catchError*** and ***switchMap***, which are indispensable for managing HTTP requests. For instance, ***catchError*** is used to gracefully handle errors during network requests, while ***switchMap*** enables the cancellation of ongoing requests when a new one with updated data arrives. Signals, in contrast, lack these essential operators, making it challenging to achieve the same level of control and robustness when dealing with network requests.

RxJS investment

Any significant shift, such as completely removing RxJS, would have far-reaching implications for Angular developers. Such a transition would necessitate a substantial investment of time and effort into refactoring existing codebases, a daunting task, especially for larger projects.

In many cases, rewriting an entire project to accommodate such a change may be the only viable option. This kind of upheaval can lead to a cascade of challenges, including disrupted workflows, increased development time, and potential compatibility issues with third-party libraries.

Conclusion: Signals and RxJS Coexistence

Signals are a promising addition to Angular's toolkit and they may make RxJS optional for some applications in the future. However, RxJs still is important, especially in more complex applications.

The strengths of RxJS in event handling, network requests make RxJs still an outstanding library for most frontends.

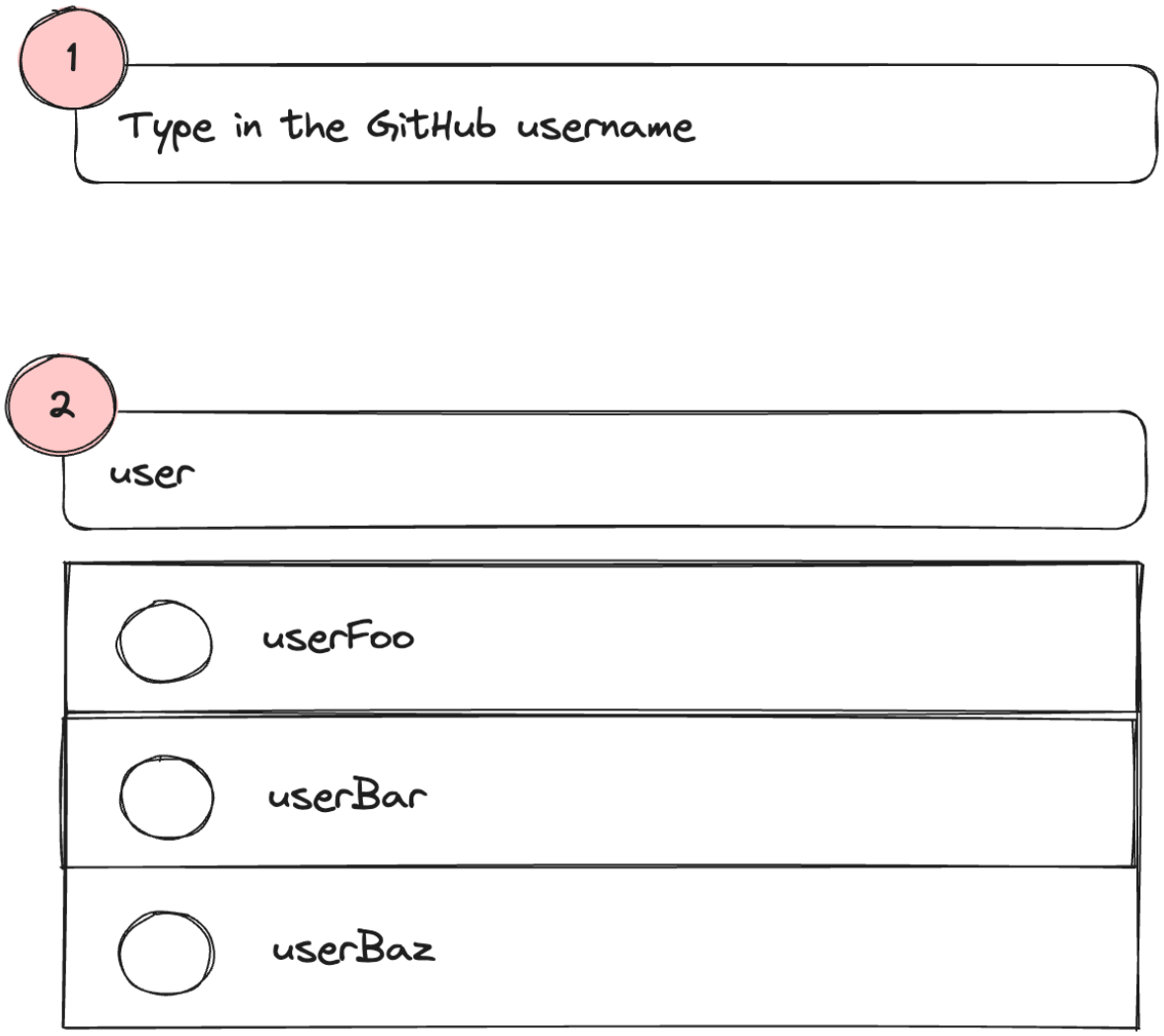
Instead of viewing Signals as a replacement for RxJS, it may be more prudent to see them as complementary tools. Developers can leverage Signals where simplicity and straightforward event handling are paramount, while relying on RxJS for complex scenarios that require the full range of operators and features it offers.

The coexistence of these two paradigms provides developers with the best of both worlds. Let’s take a look at a concrete example that illustrates exactly how we can use Signals and RxJs together to create even more powerful applications.

Best of both worlds

Let's dive into a concrete example that demonstrates how we can harness the power of Signals and RxJS to implement a GitHub user search.

The GitHub user search contains a simple input field that once we start typing requests an API to retrieve and display Users in our UI. A simple mockup for such an application could look something like this:



We have a simple application that contains a simple input field. Once we start typing into the input field the app searches for GitHub users and displays them in a dropdown.

This scenario is an ideal fit for RxJS, because it deals with UI interactions, requests and actually contains more complexity that you might initially think. To implement this feature in a clean and profound way we need to make sure the following requirements are fulfilled.

- We should only send a request if the input has more than two characters. If we would fire a request initially we would get a very big list. Two characters allow us to narrow down the users list.
- We want to implement a performant application. Therefore we need to make sure that we do not fire unnecessary requests. If a user searches for a user name we do not want to fire a request on every keystroke, we only want to fire a request if the user didn't type for some amount of time - for example 200ms.
- We need to make sure that we always display the correct data for a request. Therefore if the search changes and a request is still in process we need to cancel ongoing requests and send a new request.
- Whenever we load data we want to display a loading spinner while the request is being made.

Even though its a simple use case, there are many requirements and a lot of asynchronous processes going on.

Let's break down the requirements into a general idea of how we are going to implement the required features.

1. We start by signalifying our input changes.

> **Theoretically we could also use a Subject instead of signalifying.**

2. If we read through the requirements we can see that there are a lot of asynchronous processes going on. Handling those processes is complex but becomes easy with the power of RxJS and its built-in operators. To take advantage of RxJs we need to enter the Observable land which is exactly what we aim to do by using the ***`toObservable`*** function.

3. Once we are in RxJs land we can apply various RxJS operators to implement the requirements.

4. To finally display the processed data in the template, we can use the ***`toSignal`*** operator to convert our stream back into a Signal.

> **We could also use the *`async`* pipe to directly subscribe but using a Signal is better since it will enable fine-grained change detection in the future.**

5. Now, we have our data in the Signal format again, and we can seamlessly call the Signal in the template.

Let's transform those ideas into code.

```
@Component({
  //...
  template: `
    <input (input)="searchTerm.set($any($event.target).value)"/>

    <ng-template #loadingTemplate>
      <div>Loading...</div>
    </ng-template>

    <ul *ngIf=”!loading; else loadingTemplate”>
      <li *ngFor="let user of users()">{{ user }}</li>
    </ul>
`
})
export class App {
  private ghUsers = inject(GithubUserService);
  searchTerm = signal<string>('');
  loading = false;

  users = toSignal(
    toObservable(this.searchTerm).pipe(
      debounceTime(500),
      distinctUntilChanged(),
      filter((value) => value.length > 2),
      tap((e) => this.loading = true),
      switchMap((value) => this.ghUsers.getGithubUsers(value)),
      tap((e) => this.loading = false)
    ), { initialValue: [] }
  );
}
```

The important thing to note here is that we can work with Signals and as soon as we need the RXJS extra power we simply use the ***toObservable*** operator to convert our ***Signal*** into an Observable and right before we display our Observable in the template we convert back to a ***Signal*** using the ***toSignal*** function.

This example nicely illustrates the power of RxJs. Imagine you would need to implement this feature without the power of RxJs. The code would be very complex, hard to read and maintain.

It’s really great that we can use Signals for most cases and RxJs when it makes sense. Having the best of both worlds is truly amazing.

The current state of Angular Signals

Signals, computed Signals, effects, and the RxJS interoperability functions have been introduced in Angular 16, and they are now available for us to try out in the developer preview. The whole API will most likely be production-ready in Angular 17.

In Angular, a "developer preview" is an early-release version of a software update that provides developers with access to new features and changes for testing and feedback, but it is not intended for production use, helping developers prepare for upcoming stable releases.

This means that while you can start experimenting with them, it's important to keep in mind that the API may undergo some changes in upcoming versions.

Signals are already very exciting but what's even more exciting than Signals itself is the possibilities and features it enables for the future.

Impact on Angular's future

In the future, Signals will have a significant impact on various areas of the Angular framework. There's a very exciting RFC open that describes the vision of Angular Signals. Let's take a look at the most important points of the RFC.

The Rise of Signal-based Components

One of the most significant changes heralded by Angular Signals is the introduction of signal-based components. These components are designed to leverage Signals for their reactivity and rendering.

The API for signal-based components is still in the RFC stage, and the exact form it will take in the future remains uncertain. However, a possible API could look something like this:

```
@Component({
  standalone: true,
  signal: true,
  //...
})
```

Signal based components would introduce many changes. Let's check out changes Signal based components might introduce as well as possibilities they might offer.

Lifecycle Evolution

Signal-based components come with an entirely new set of lifecycle hooks. These hooks are tailored to the Signals paradigm, enabling developers to orchestrate the behavior of their components with fine-grained control. This promises to streamline the development process opens up exciting possibilities for optimizing application performance.

Local Change Detection

One standout feature of Signal-based components is their local change detection. Unlike the current zone-based change detection, Signal-based components only trigger change detection when a Signal they rely on in their template notifies Angular of a change. This targeted approach to change detection minimizes unnecessary computations, making applications more efficient and responsive.

Most likely changes will be detected on the level of Views and not expressions. The Angular team already mentioned that they could also detect changes on the expression level but the graph also has a cost. Therefore detecting changes on the View level seems to be the right tradeoff between fine granular change detection and performance.

Zoneless Applications on the Horizon

Perhaps one of the most intriguing promises of Signal-based components is the potential to develop zoneless applications. This means shedding the traditional reliance on NgZone, a critical part of Angular's change detection mechanism. The move toward zoneless applications offers the prospect of cleaner, more predictable code and improved performance.

Coexistence and Interoperability

For developers concerned about transitioning to Signal-based components, rest assured that Angular's current zone-based change detection will continue to be supported for the foreseeable future. This commitment to compatibility ensures that existing applications can seamlessly coexist with and leverage the advantages of the new signal-based reactivity.

Conclusion

The future of Angular is bright, and Angular Signals are at the forefront of this exciting journey. As we embrace Signal-based components, new opportunities for fine-tuning reactivity and optimizing performance emerge. The advent of local change detection and the potential for zoneless applications mark a significant step forward in Angular's evolution.

Get in touch & feedback

Thank you so much for picking up a copy of my book. Writing this book has been an incredible adventure, and your support means a lot to me.

I'm genuinely looking forward to hearing your thoughts and insights as you read through the pages.

Simply drop me a message at get-in-touch@angularexperts.io or reach out to me on X via direct message.

Thanks again for purchasing this book and for supporting my work. Your support fuels my passion as a creator and I am thankful for every page you've turned. Always bet on Angular!

Kevin

More

If you enjoyed this book, you'll find our workshops and products to be the perfect companions for your learning journey. Explore our offerings today and continue your journey of discovery with us.

Workshops

Getting Reactive with RxJs

RxJs is the most complex element of Angular development.

Elevate your team's RxJs skills to deliver maintainable features confidently and on time!

Angular State Management with NgRx

Still managing your Angular application state with buggy ad hoc services?

Master your Angular application state management with NgRx's robust, repeatable patterns and best practices!

Angular Mastery

This workshop will teach you all necessary concepts to become proficient Angular developer by building a real world single page application!

Lots of actionable tips and pros & cons of specific decisions based on the extensive experience!

Frontend Heroes

This workshop will teach you all the necessary concepts to kickstart your enterprise frontend development expertise.

We are going to explore and get hands on experience with frontend platform fundamentals like HTML, CSS and JavaScript together with most essential developer tools and environments!

Find out more

Skol

“My favourite IDE theme”

"As programmers, we spend countless hours looking at a screen. So why not look at some pleasant UI that is nice to look at and fun to use."

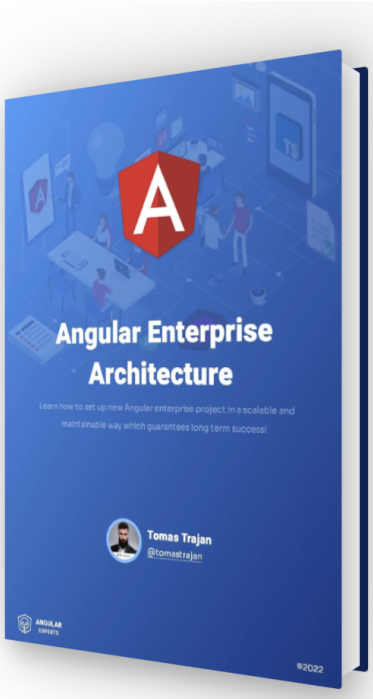
In a decade of developing software, I've gained a lot of experience and learned how to become effective and productive with development tools. The most essential tool for us software developer is an Integrated Development Environment or IDE.

In my long career, I have tried many IDEs, from plain Vim and Eclipse to Atom and VSCode. After evaluating these variations, I concluded that the Jetbrains IDEs are the most powerful and complete IDE for programming. There was just one thing missing, a beautiful aesthetic theme!

[Find out more]
[Get it now]

Angular Enterprise Ebook

Learn how to architect and scaffold a new enterprise grade Angular application with clean, maintainable and extendable architecture in almost no time! Lots of actionable tips and pros & cons of specific decisions based on the extensive experience!



[Find out more]

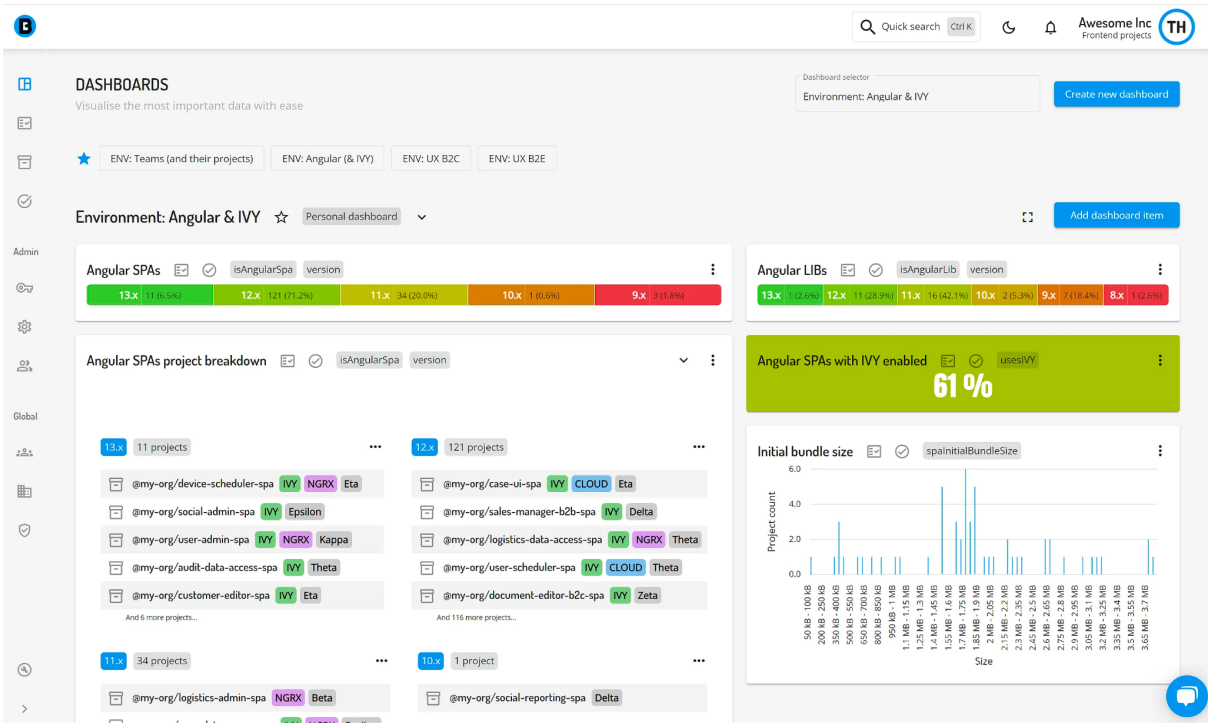
Angular UI component library starter

This product is for you and your team if you want to build a high-quality UI component library with an outstanding showcase app in no time. With this starter, you can focus on building your components and avoid losing countless hours on tooling, project setup, and exploring best practices.

Find out more
Live demo

Omniboard

The best tool for lead software engineers and architects to understand and drive change in the code bases of their enterprise organization!



Find out more
Live demo

Project review

With our Angular project review, we will take a deep dive into your codebase, ensuring it's not just robust, but also highly maintainable and extendable. You can expect that we're going to identify any areas of potential improvements and make sure you're on the right track by implementing best practices, allowing your team to focus on business-specific features, not code headaches.

Find out more

Blog & Videos

Looking to advance your web development skills? Our blog and video content cover Angular, NgRx, RxJS, and NX in-depth, offering valuable insights, tips, and tricks.

Blog
Videos

Other services

Looking for assistance or have a specific request in mind? Look no further! Our website offers many services and resources waiting just for you. Whether you need expert guidance, useful tools, or insightful information, we've got you covered.

Explore our website, discover what we can do for you, and don't hesitate to reach out for any request or feedback you might have.

angularexperts.io