Live, Laugh, Signal

An in-depth look into Angular Signals

Dan Klingmann, Staff Software Engineer, Celonis

3rd December, 2025

About me

Daniel “Dan” Klingmann

Obsessed about **Frontend Architecture & Performance**

Working in frontend since the double float margin bug was a thing (that’s 2001).

Hobbies include composing music in my home studio, gaming or working on an MMO quiz game show.

Where we're coming from > Reactivity before Signals

- Promises
- Proxies (Vue for example)
- EventEmitters
- RxJS Observables

One problem though…

They’re not well suited for state

Workarounds for state

State management libraries centered around redux

- ngrx
- ngxs

Other state management libraries

Or DIY solutions with services

Signals:
A new* reactive primitive

* Calling Angular v17 new is a stretch but chances are you’re working on enterprise software so you’re probably stuck with ancient versions of Angular anyways so it might indeed be new for you. Surely you’ve already convinced management you need to migrate away from AngularJS by now, right? What do you mean you’re still using JQuery?

What’s the hype about signals?

Signals are lazily evaluated (unless they’re live)

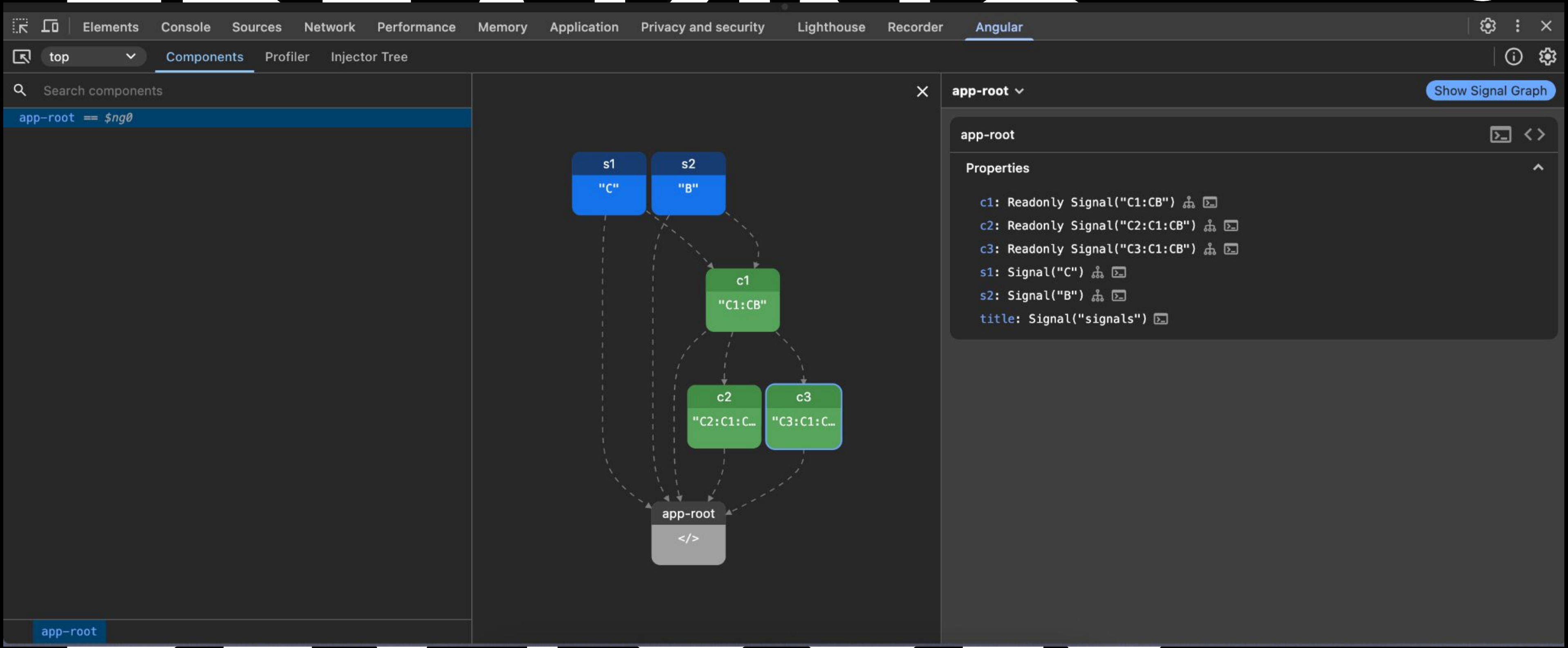
Computed signals are memoized

They’re highly performant

They come included with Angular

They use the Angular Injection Context and thus can be garbage collected automatically (because we all sometimes forget to unsubscribe, right?)

Oh and…



```
c1: Readonly Signal("C1:CB")
c2: Readonly Signal("C2:C1:CB")
c3: Readonly Signal("C3:C1:CB")
s1: Signal("C")
s2: Signal("B")
title: Signal("signals")
```

Types of Signals › signal / WritableSignal

```
function signal<T>(
  initialValue: T
): WritableSignal<T>
```

- Initial value
- Can be written to hence returning a WritableSignal
- Can be changed via set(newValue)
- Can be derived via update(oldValue => oldValue + myChange)

Similar to the RxJS BehaviorSubject

Types of Signals > signal / Signal

```
signalFn.asReadonly = signalAsReadonlyFn.bind(signalFn)
function signalAsReadonlyFn<T>(
 this: SignalGetter<T>
): Signal<T>
```

- Can be called as signal.asReadonly
- Depends on the existing signal
- Can **NOT** be written to hence only returning a Signal

Similar to the RxJS Subject.asObservable()

Types of Signals › computed / Signal

```
function computed<T>(
  computation: () => T
): Signal<T>
```

- Runs a given computation lazily when accessed
- Can not be written to

Similar to RxJS combineLatest with a mapping function

Types of Signals > linkedSignal / WritableSignal

```
function linkedSignal<T>({
  source: () => S;
  computation: ComputationFn<S, D>;
}): WritableSignal<D>
```

- Runs a given computation every time the source signal changes
- **Can actually be written to** as it’s a WritableSignal

Similar to RxJS switchMap with a mapping function

Omissions

HttpResource - Allows you to call an HTTP endpoint on change of a signal with complete control over the request (methods, payload, headers)

Resource - Similar, but works with any arbitrary async function

Effect - Kind of the basis for the other above, run any code when something changes.

But how does that work in Angular?

Disclaimer: This is all taken from source code and interpreted by the author. The intention is to give an overview of the things Angular does to provide you with some basic understanding of how it works. The signal implementation of Angular is very well documented and I hope to pique your interest enough to take a look at the implementation yourself.

It’s a reactive graph.
That’s it.

Thanks for joining my talk. Enjoy the food and drinks.

Reactive Nodes

ALL reactive nodes are created via Object.create and inherit from the REACTIVE_NODE[1] object implementing the ReactiveNode interface[2].

They then selectively override some of the properties.

```
const REACTIVE_NODE: ReactiveNode = {
 version: 0 as Version,
 lastCleanEpoch: 0 as Version,
 dirty: false,
 producers: undefined,
 producersTail: undefined,
 consumers: undefined,
 consumersTail: undefined,
 recomputing: false,
 consumerAllowSignalWrites: false,
 consumerIsAlwaysLive: false,
 kind: 'unknown',
 producerMustRecompute: () => false,
 producerRecomputeValue: () => {},
 consumerMarkedDirty: () => {},
 consumerOnSignalRead: () => {},
};
```

[1] https://github.com/angular/angular/blob/f35b2ef47cef778e35c3f62ba51212b3585a33f2/packages/core/primitives/signals/src/graph.ts#L64
[2] https://github.com/angular/angular/blob/f35b2ef47cef778e35c3f62ba51212b3585a33f2/packages/core/primitives/signals/src/graph.ts#L112

Angular splits signals into **two roles**

Producers	**Consumers**
Hold values that can be updated at any time.	Read the values that are stored in producers.
signal	effects
computed	computed
linkedSignal	linkedSignal
resource	resource
	template

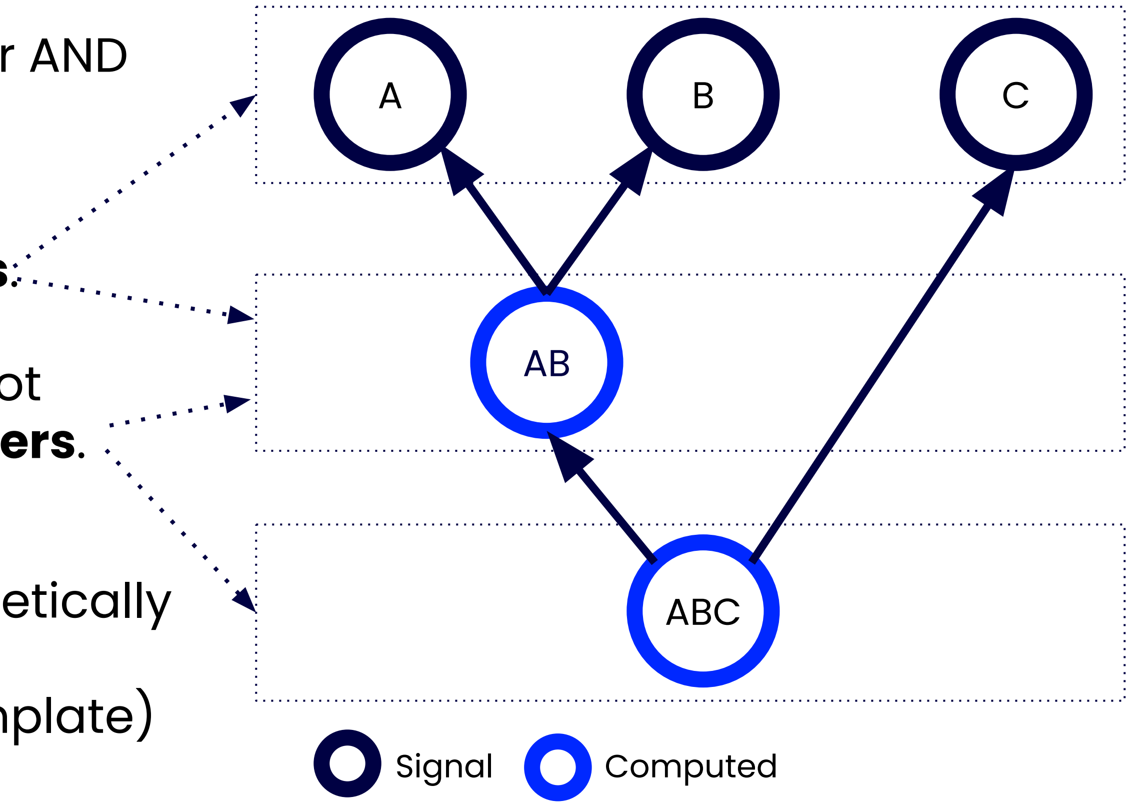
Graph structure

Signals can be both, producer AND consumer.

Signals on a graph that are not an end are **producers**.

Signals on a graph that are not at the beginning are **consumers**.

(Computeds at the end theoretically also produce values that are probably consumed by a template)



How does Angular know what needs to be updated?

Dependency tracking

Signals track dependencies in a linked list of producers using the ReactiveLink[1] interface

On accessing a signal, it is checked if the access was coming from a reactive context[2].

Reactive context means, a consumer has set itself as active consumer.

```
interface ReactiveLink {
 producer: ReactiveNode;
 consumer: ReactiveNode;
 lastReadVersion: number;
 prevConsumer?: ReactiveLink;
 nextConsumer?: ReactiveLink;
 nextProducer?: ReactiveLink;
}
```

[1] https://github.com/angular/angular/blob/f35b2ef47cef778e35c3f62ba51212b3585a33f2/packages/core/primitives/signals/src/graph.ts#L82
[2] https://github.com/angular/angular/blob/f35b2ef47cef778e35c3f62ba51212b3585a33f2/packages/core/primitives/signals/src/graph.ts#L213

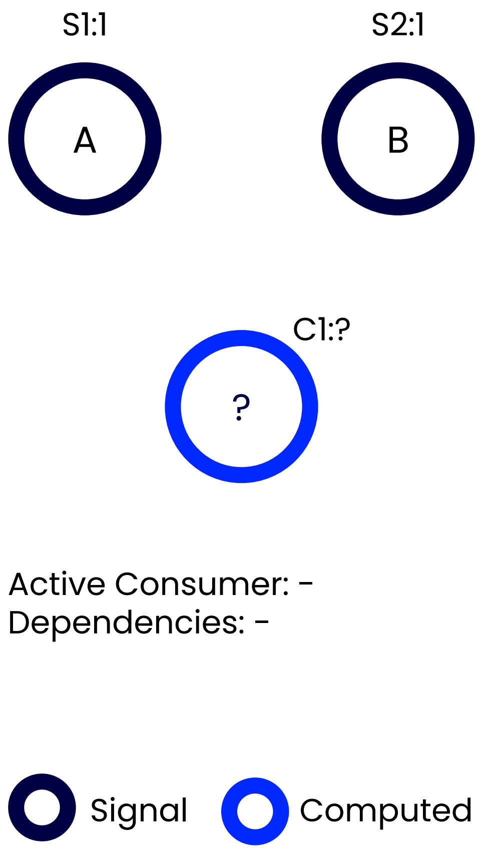
Dependency tracking - Example

S1 and **S2** are simple signals holding a string character

C1 is a computed signal:

() => s1() + s2()

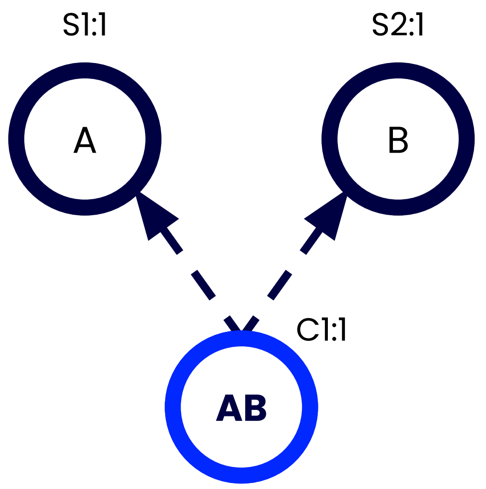
As **C1** was not read yet, it doesn't have a value and no recorded dependencies.



Dependency tracking - Example

Calling **C1** now does the following (although quite a bit simplified):

- Set itself as active consumer
- Run the computation fn
- First call **S1** adding the first dependency
- Then call **S2** adding the second dependency
- Store and return the value



Active Consumer: **C1**
Dependencies:

S1 (lastReadVersion: 1)
S2 (lastReadVersion: 1)

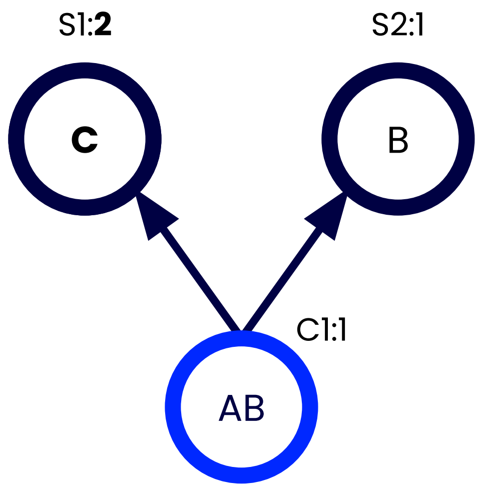
○ Signal ○ Computed

Dependency tracking - Example

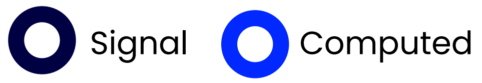
Changing the value of **S1** now leads to the following:

- Increase it's own version
- Increase the epoch* counter
- Update it's value

S1 does not know of **C1** so it can't force an update.



Active Consumer:
Dependencies:

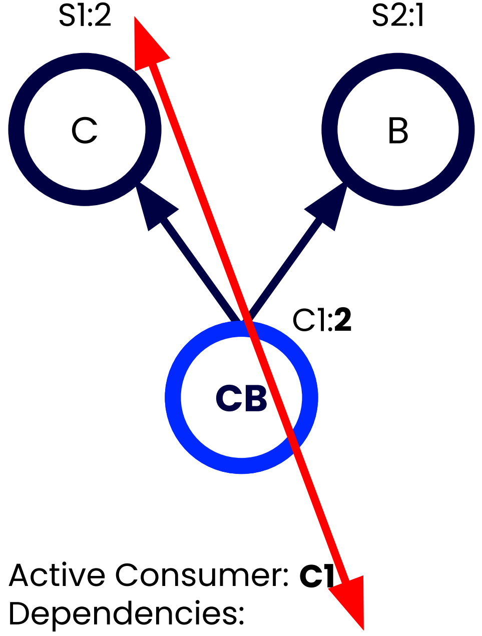


*Well come back to epochs later

Dependency tracking - Example

Reading **C1** now does the following

- Check lastRead versions against the dependencies
- See that **S1:1** and **S1:2** don't match so get new value and run computation again.
- Store new lastReadVersion
- Increase own version
- Set new value



Active Consumer: **C1**
Dependencies:

S1 (lastReadVersion: **2**)

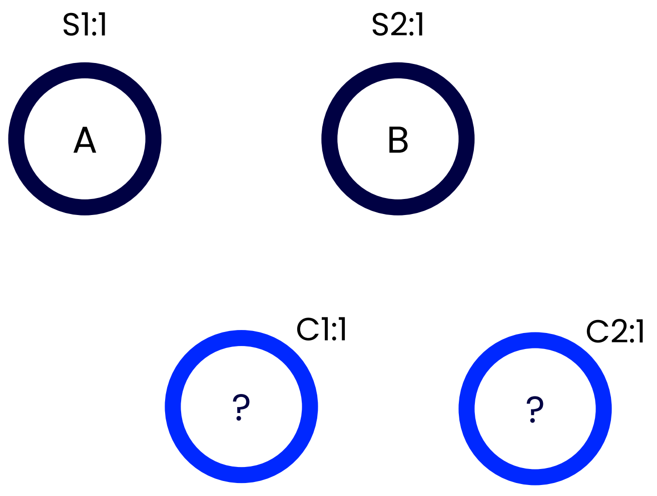
S2 (lastReadVersion: 1)

● Signal ● Computed

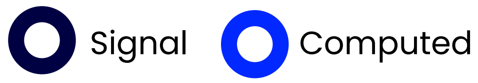
Dependency tracking - Example

What if we have another computed **C2** depending on **C1**?

Let’s access **C2** and see what happens.



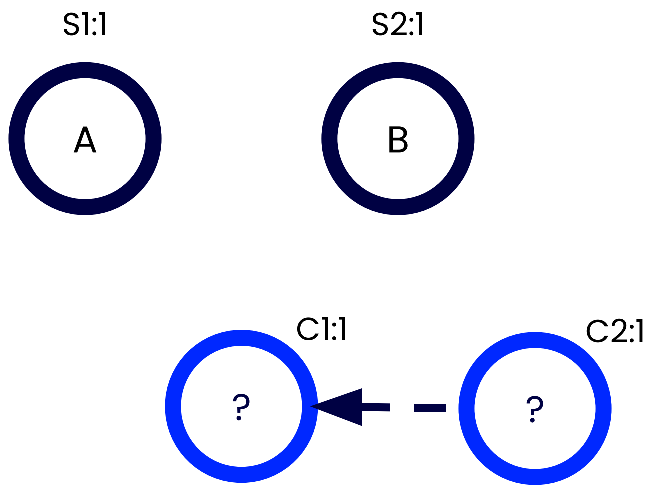
Active Consumer:
Dependencies:



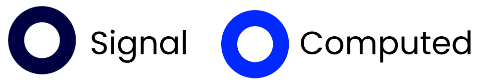
Dependency tracking - Example

First off, **C2** sets itself as active consumer and calls **C1**.

However, **C1** itself does not have a value yet.



Active Consumer: **C2**
Dependencies:

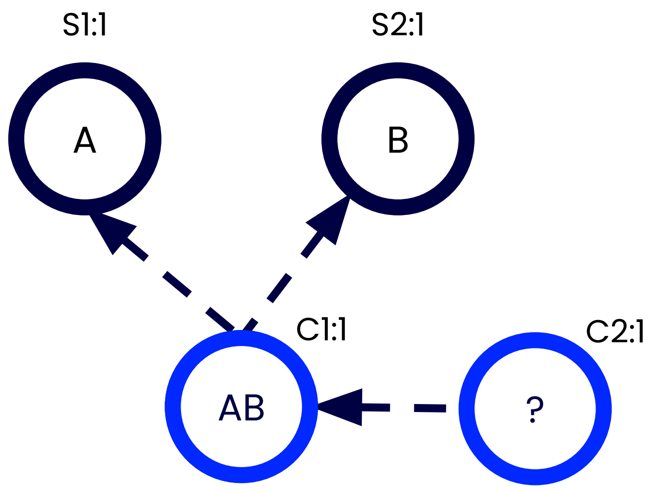


Dependency tracking - Example

C1 thus sets itself as active consumer

It then runs its computation first and accesses **S1** and **S2** storing it as dependencies and setting its value.

After it completes, it sets the previous consumer back as active consumer



Active Consumer: **C2**
Dependencies:

S1 (lastReadVersion: 1)

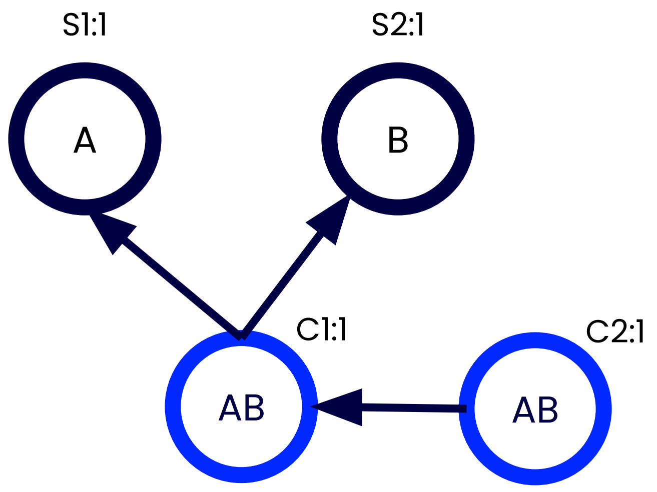
S2 (lastReadVersion: 1)

○ Signal ○ Computed

Dependency tracking - Example

C2 can now get the value from **C1** and sets it as dependency.

Let’s change a signal’s value now again.



Active Consumer: **C2**
Dependencies:

C1 (lastReadVersion: 1)

● Signal ● Computed

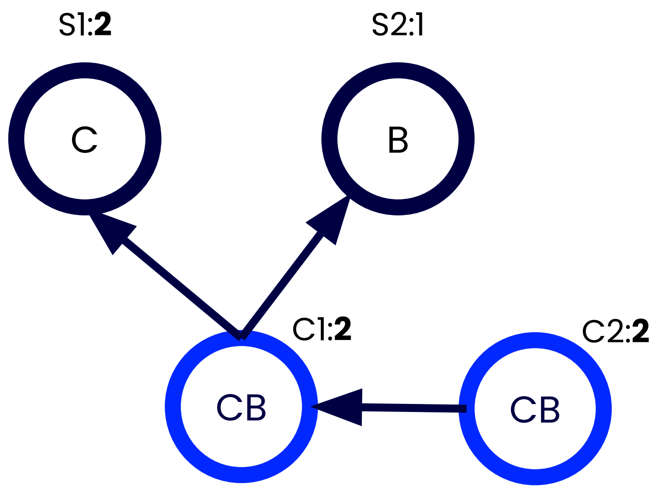
Dependency tracking - Example

We change **S1** to have a new version and now access **C2** again.

However, just checking for the version now doesn't work.

Angular first checks the version, then runs the update on the producer and then runs the check again.[1]

But what if two computed Signals depend on a large calculation?



Active Consumer: C2
Dependencies:
- C1 (lastSeen 1)

Signal
Computed

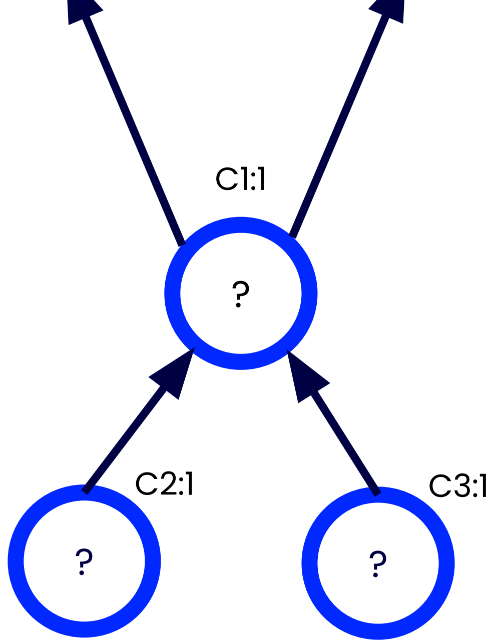
[1] https://github.com/angular/angular/blob/e26aa86aa54a1fcd58869296660633c66898fc94/packages/core/primitives/signals/src/graph.ts#L456

Dependency tracking - Example

In this example, the computed **C1** is a particularly heavy calculation which has **C2** and **C3** as dependents.

Based on the previous example, you would expect **C1** to run multiple times as it might be stale leading to expensive recalculations.

That is, unless we track that "somehow".



Introducing: Epochs

Big fan of the gothic epoch but I digress.

It’s awesome though, JUST LOOK AT THIS ->

Epochs

When a signal’s value is changed[1], it

1. Increases its own version
2. Increases the global epoch
3. Notifies any live consumers of the change

producerIncrementEpoch is literally just epoch++

Let’s see how the epoch counter helps against recalculations.

```
function signalValueChanged (node)
{
 node.version++;
 producerIncrementEpoch ();
 producerNotifyConsumers (node);
 postSignalSetFn ?. (node);
}
```

[1] https://github.com/angular/angular/blob/f35b2ef47cef778e35c3f62ba51212b3585a33f2/packages/core/primitives/signals/src/signal.ts#L121

Epochs

Producers will always call **producerUpdateValueVersion** when it is requested to update its own value.

One of the checks is for the **lastCleanEpoch**. If it's not equal, it will run the computation again.

The last call simply sets the node's dirty flag to false and the **lastCleanEpoch** to the current so calculations in the same epoch are guaranteed to have the up-to-date value.

```
…
if (
  !node.dirty &&
  node.lastCleanEpoch === epoch
) {
  return;
}
…
node.producerRecomputeValue(node)
producerMarkClean(node);
```

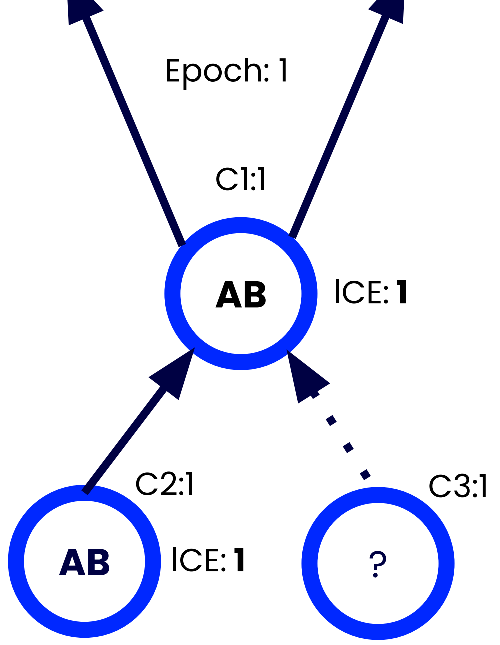
Epochs - Example

Let’s assume we are at epoch 1.

C2 calls **C1** and requests an update.

->**C1** doesn’t have a value yet

C1 calculates a value and sets its last clean epoch to 1 and returns the value to **C2** which also sets its last clean epoch.



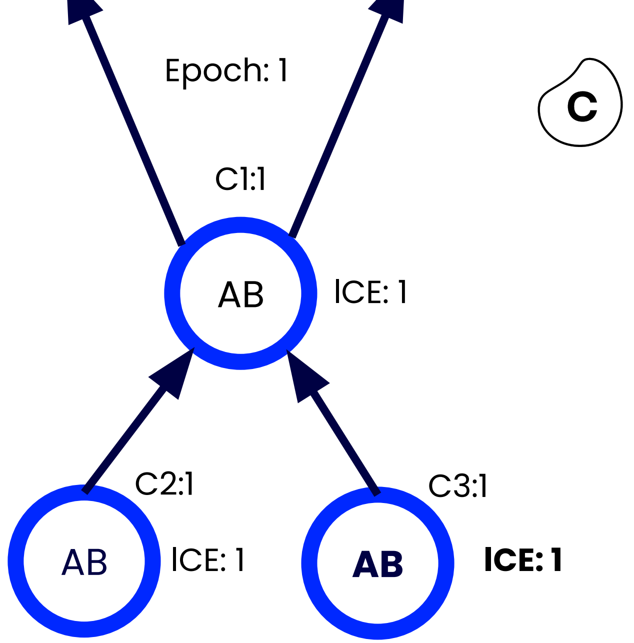
Epochs - Example

Now **C3** also is called and requests the value from **C1**.

C1 now checks if the epoch changed.

As it didn’t, its value is guaranteed to be up-to-date and no computations need to run.

C3 is now also at epoch 1.



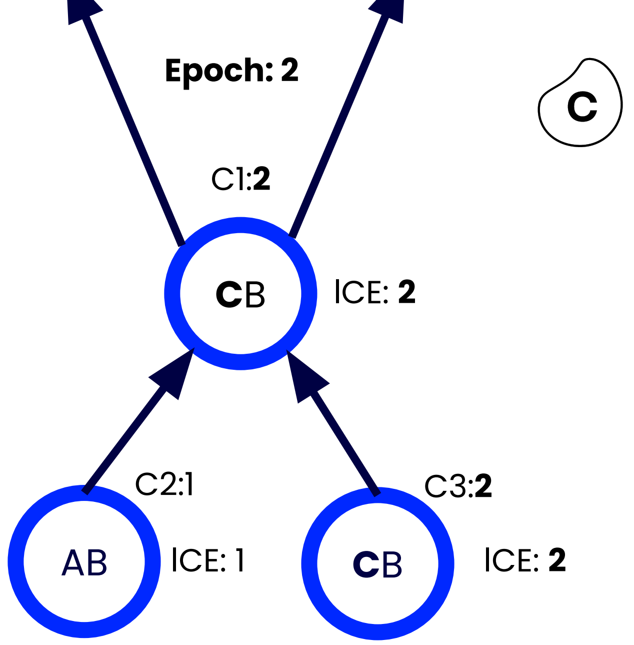
Epochs - Example

Let’s change an upstream Signal. This will set the **epoch to 2**.

Accessing **C3** will now see that the epoch changed and ask it’s dependency to update.

C1 checks the lastSeenVersions of its own dependencies and then update accordingly.

No matter if it had to update or not, the last clean epoch is set again and the new value returned to **C3**.



How does this actually work in templates?

Live Signals

Signals like template, effect or resource are live.

Not really lazily evaluated as they're not actively called by the user

Update via Push/Pull principle to prevent glitches[1]

- **Push** a dirty value in a notification phase to all tracked consumers
- Consumers then **pull** the new values evaluating all dependencies

Wait a second…

tracked consumers?

[1] https://github.com/angular/angular/blob/main/packages/core/primitives/signals/README.md#pushpull-algorithm

Live Signals - Tracking consumers

A signal that's live will ALWAYS result in all its dependencies being treated as live signals so it can receive updates.

For this, producers NEED to know who depends on them thus tracking them in the consumers linked list.

If a producers with such a dependents value changes, it will mark all live dependents dirty recursively and calls **consumerMarkedDirty**[1] on them.

[1] https://github.com/angular/angular/blob/a0ad5d4b2b2a99fb8eee2003393f85c7824c7b72/packages/core/primitives/signals/src/graph.ts#L359

Signals in templates

Special **ReactiveLViewConsumer** type (basically a big computed)

Attached to the LView of a component

Is **always live**

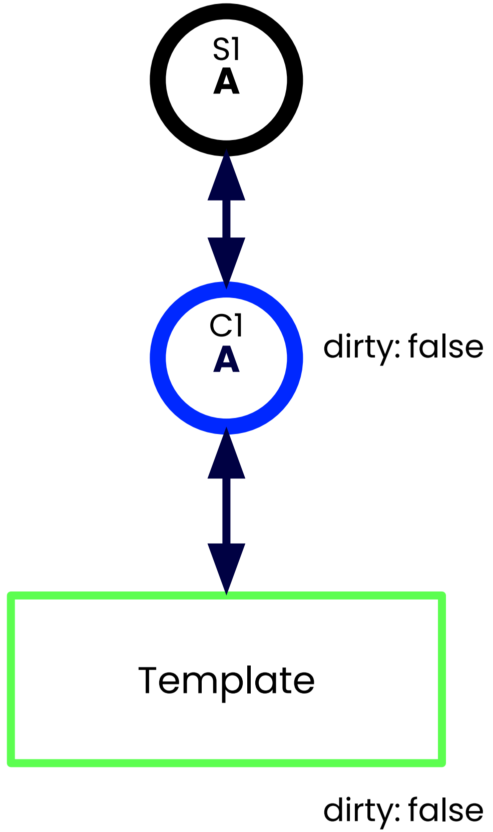
Consumes all signals called in the template

Schedules a change detection cycle on **consumerMarkedDirty**

Change detection cycle checks if the consumer is marked dirty and if it is, rerenders the template

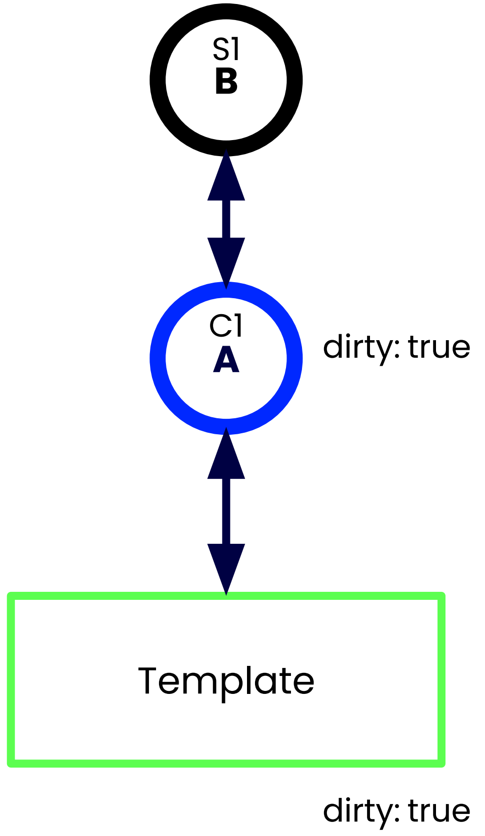
Template - Initial value example

1. Template sets itself as active consumer and executes (as before)
2. Calls **C1** which realizes the consumer is live and adds it as dependency to consumer.
3. **C1** recalculates and thus calls **S1**
4. **S1** realizes **C1** is transitively live and stores it as consumer.



Template - Update example

1. We now update the value **S1**
2. **S1** sees live consumers and enters a notification phase, iterating through consumers and marking them dirty.
3. **C1** gets the notification and recursively marks the **template** dirty
4. The template's **consumerMarkedDirty** function schedules a new change detection cycle.
5. Normal update happens

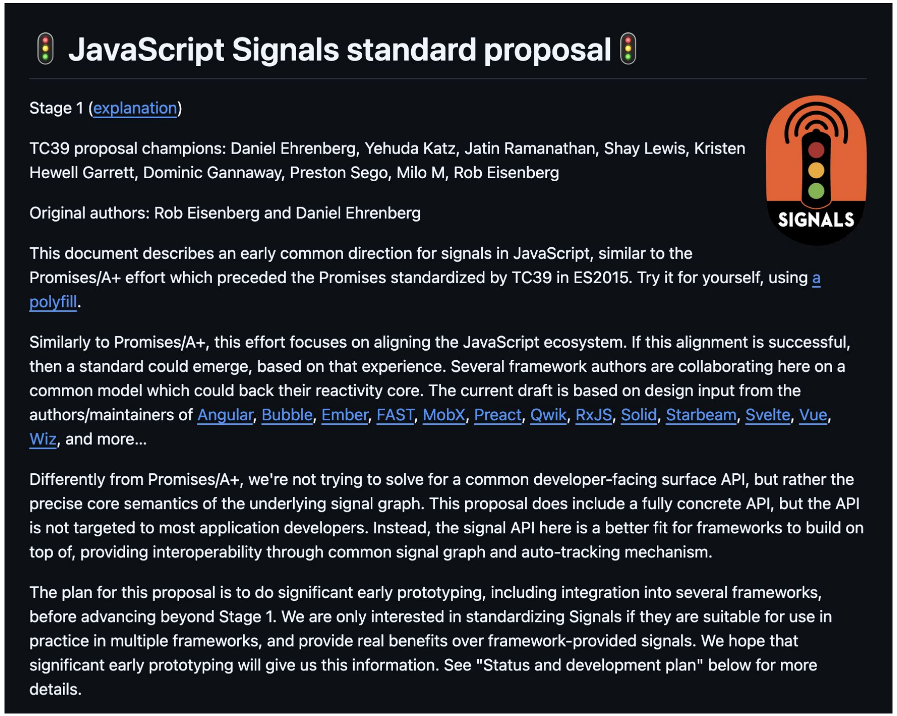


Where we're going to (hopefully)

Signals are already a proposed standard[1] albeit currently in draft

"The current draft is based on design input from the authors/maintainers of Angular, Bubble, Ember, FAST, MobX, Preact, Qwik, RxJS, Solid, Starbeam, Svelte, Vue, Wiz, and more…"

One can hope.



🚦 JavaScript Signals standard proposal 🚦

Stage 1 (explanation)

TC39 proposal champions: Daniel Ehrenberg, Yehuda Katz, Jatin Ramanathan, Shay Lewis, Kristen Hewell Garrett, Dominic Gannaway, Preston Sego, Milo M, Rob Eisenberg

Original authors: Rob Eisenberg and Daniel Ehrenberg

This document describes an early common direction for signals in JavaScript, similar to the Promises/A+ effort which preceded the Promises standardized by TC39 in ES2015. Try it for yourself, using a polyfill.

Similarly to Promises/A+, this effort focuses on aligning the JavaScript ecosystem. If this alignment is successful, then a standard could emerge, based on that experience. Several framework authors are collaborating here on a common model which could back their reactivity core. The current draft is based on design input from the authors/maintainers of Angular, Bubble, Ember, FAST, MobX, Preact, Qwik, RxJS, Solid, Starbeam, Svelte, Vue, Wiz, and more…

Differently from Promises/A+, we're not trying to solve for a common developer-facing surface API, but rather the precise core semantics of the underlying signal graph. This proposal does include a fully concrete API, but the API is not targeted to most application developers. Instead, the signal API here is a better fit for frameworks to build on top of, providing interoperability through common signal graph and auto-tracking mechanism.

The plan for this proposal is to do significant early prototyping, including integration into several frameworks, before advancing beyond Stage 1. We are only interested in standardizing Signals if they are suitable for use in practice in multiple frameworks, and provide real benefits over framework-provided signals. We hope that significant early prototyping will give us this information. See "Status and development plan" below for more details.

[1] https://github.com/tc39/proposal-signals

Outlook on Signal forms[1]

A new way to create reactive forms introduced recently.

Allows you to define your form state in a signal in an easy way including validation.

Available with Angular v21[2]

```
// TS
interface LoginData {
 email: string;
 password: string;
}
const loginModel = signal<LoginData>({
 email: '',
 password: '',
});
const loginForm = form(loginModel);

// HTML
<input type="text" [field]="loginForm.name" />
<input type="email" [field]="loginForm.email" />
```

[1] https://angular.dev/essentials/signal-forms
[2] https://blog.angular.dev/announcing-angular-v21-57946c34f14b

If you want to dive in yourself…

Most of the reactive code can be found in
/packages/core/primitives/signals

Wrappers for common usage of signals are in
/packages/core/src/render3/reactivity

If you're brave enough and see a use-case, you could even implement your own custom signals on top of this.

For example, why not try something with WebSockets?

Q&A

c

Thank you.