

## Improving ergonomics of events with Observable #544



...

- Get Observable to web developers more quickly.
- Allow for a more full-featured proposal that will address more developer pain points.
- Address concerns raised in the TC-39 that there has not been sufficient consultation with implementers.

1. First-class objects representing composable repeated events, similar to how promises represent one-time events
2. Ergonomic unsubscription that plays well with `AbortSignal/AbortController`
3. Good integration with promises and `async/await`



```

partial interface EventTarget {
    Observable on(DOMString type, optional AddEventListenerOptions options);
};

[Constructor(/* details elided */)]
interface Observable {
    AbortController subscribe(Function next, optional Function complete, optional Function error);
    AbortController subscribe(Observer observer); // TODO this overload is not quite valid
    Promise<void> forEach(Function callback, optional AbortSignal signal);

    Observable takeUntil(Observable stopNotifier);
    Promise<any> first(optional AbortSignal signal);

    Observable filter(Function callback);
    Observable map(Function callback);
    // rest of Array methods
    // - Observable-returning: filter, map, slice?
    // - Promise-returning: every, find, findIndex?, includes, indexOf?, some, reduce
};

dictionary Observer { Function next; Function error; Function complete; };

```



```
// filtering and mapping:
element.on("click").
  filter(e => e.target.matches(".foo")).
  map(e => ({x: e.clientX, y: e.clientY})).
  subscribe(handleClickAtPoint);

// conversion to promises for one-time events
document.on("DOMContentLoaded").first().then(e =>);

// ergonomic unsubscribe via AbortControllers
const controller = element.on("input").subscribe(e =>);
controller.abort();

// or automatic/declarative unsubscribe via the takeUntil method:
element.on("mousemove").
  takeUntil(document.on("mouseup")).
  subscribe(etc =>);

// since reduce and other terminators return promises, they also play
// well with async functions:
await element.on("mousemove").
  takeUntil(element.on("mouseup")).
  reduce((e, soFar) =>);
```

If there's interest, we're happy to work on fleshing this out into a fuller proposal. What would be the next steps for that?

😊 367 👎 1 😊 18 🎉 95 😞 1 ❤️ 139 🙄 1



Member ...

From what I remember discussing this before one problem with this API is that it does not work with `preventDefault()`. So if you frequently need to override an action (e.g., link clicks), you can't use this API.

👍 5



davidkpiano on Dec 13, 2017

...

From what I remember discussing this before one problem with this API is that it does not work with `preventDefault()`. So if you frequently need to override an action (e.g., link clicks), you can't use this API.

Couldn't this be covered with `EventListenerOptions` options ?

```
element.on('click', { preventDefault: true })
  .filter(/* ... */)
  // etc.
```



👍 10



gsans on Dec 13, 2017

...

Cool! I propose to call them DOMservables

👍 44 🗳️ 1



matthewp on Dec 14, 2017 · edited by matthewp

Edits ...

@annevk Can you explain the `preventDefault()` problem in more detail? From @benlesh's examples, I would think you could call it in any of the `filter()` or `map()` callbacks. Is there a reason why you could not?



domenic on Dec 14, 2017

Member ...

I think @annevk is remembering an async-ified version, perhaps based on async iterators. But observables are much closer to (really, isomorphic to) the `EventTarget` model already, so don't have this problem. In particular, they can call their `next` callback in the same turn as the event was triggered, so `e.preventDefault()` will work fine.

It really is just A Better `addEventListener` (TM). ^\_^

👍 25 ❤️ 9



annevk on Dec 14, 2017

Member ...

(It seems my concern indeed only applies to promise returning methods, such as `first()`.)



appsforartists on Dec 14, 2017

...

@benlesh Can you speak more to why the `subscribe(observer)` signature isn't valid? That's how all the observable implementations I've seen currently work, but they aren't written in C or Rust.



domenic on Dec 14, 2017

Member ...

In web APIs, per the rules of [Web IDL](#), it's not possible to distinguish between a function and a dictionary. (Since functions can have properties too.) So it's just disallowed in all web specs currently. Figuring out how or whether to allow both `o.subscribe(fn)` and `o.subscribe({ next })` is the TODO.

To be clear, the tricky case is

```
function fn() { console.log("1"); }
fn.next = () => { console.log("2") };

o.subscribe(fn);
```



Which does this call? Sure, we could make a decision one way or another, but so far in web APIs the decision has been to just disallow this case from ever occurring by not allowing specs that have such overloads. So whatever we do here will need to be a bit novel.

This is all a relatively minor point though, IMO. Perhaps we should move it to <https://github.com/heyecam/webidl/issues>.

👍 6



benlesh on Dec 14, 2017

Author ...

Another thing that would be interesting to know is what the various frameworks and libraries do here and whether this would make their job easier.

I can't speak for frameworks, directly. Perhaps @IgorMinar or @mhevery can jump in for that, but for RxJS's part, whenever anyone goes to build an app using only RxJS and the DOM, one of the most common things they need from RxJS is `fromEvent`, which this would completely replace. I would also definitely love to see an RxJS that was simply a collection of operators built on top of a native Observable we didn't always have to ship.

👍 12



appsforartists on Dec 14, 2017

...

Thanks for the clarification.

It's interesting to me that TC39 and WHATWG both have the ability to add JS APIs, but with different constraints. The TC39 proposal [decides what to do based on if the first param is callable](#). If the TC39 proposal was stage 4, the browsers would be implementing that behavior, right? (Or maybe the TC39 proposal was supposed to be in WebIDL too and violated this. I [hadn't heard](#) about that, but I'm not a TC39 member either).



keithamus on Dec 14, 2017 · edited by jakearchibald

Edits ...

FWIW in practice the DOM already makes the distinction of function-vs-object in the case of `addEventListener` :

```
const handle = e => console.log('main!', e)
handle.handleEvent = e => console.log('property!', e)
document.body.addEventListener('click', handle)

// Logs with `main!`
```



(Not suggesting WebIDL can't make the distinction, just pointing out there is a precedent here)



2



jhusain on Dec 14, 2017 · edited by jhusain

Edits ▾ ⋮

[@annevk](#) concerns about the ability to call `preventDefault()` when using Promise returning methods are valid. Mutation use cases could be addressed with a `do` method which allows side effects to be interleaved.

```
button.on("click").do(e => e.preventDefault()).first()
```



This method is included in most userland Observable implementations.



7



benlesh on Dec 14, 2017

Author ⋮

[@jhusain](#) it could also be handled with `map`, although it would drive some purists crazy:

```
button.on('click').map(e => (e.preventDefault(), e)).first()
```



appsforartists on Dec 14, 2017

⋮

The biggest thing I see missing from this proposal is an official way to use userland operators. [@benlesh](#), do you think `pipe` is ready to be included?

To the question about frameworks/libraries, as the author of a [userland implementation](#), there are three core types that I'm interested in:

- `Observable`
- `MemorylessSubject` - It's both an `Observer` and an `Observable`. Calls to its `next` method are passed to all the observers who have `subscribe`d.
- `Subject` - In addition to being both an `Observer` and an `Observable`, it remembers its most recent emission and calls new subscribers with it immediately.

If `Observable` is standardized without the `Subject`s, userland libraries will still ship them. Still, any standardization is good for the whole ecosystem - it means operator authors know what interface they must implement to be interoperable with the ecosystem.

Furthermore, if `Observable` is standardized, I expect it will become common knowledge among web authors (just like promises, generators, etc.) That will make it easier for frameworks to depend on it without having to worry about intimidating their users with a steep learning curve - understanding observables becomes part of the job.



2

16 remaining items

Load more



YurySolovyov on Dec 14, 2017

⋮

Btw, any reasoning on why this should be a DOM API and not core language feature?



benlesh on Dec 14, 2017 · edited by benlesh

Edits ▾ Author ⋮

[@TimothyGu](#) Also worth considering: Node and JQuery chose `on` because it happens to be a *really good* name for an event-related method. It's almost an argument for *why* it's the right name. It would be a shame if we didn't consider using it.



9



benlesh on Dec 14, 2017 · edited by benlesh

Edits ▾ Author ⋮

[@YurySolovyov](#) there is a related proposal in the TC39 to add it as a feature to JavaScript: [https://github.com/tc39/proposal-observable/](https://github.com/tc39/proposal-observable) It's basically in the same shape as this, but this particular proposal is more about improving the DOM eventing API with a proven primitive that matches up with `EventTarget` well.



domenico on Dec 14, 2017 · edited by domenico

Edits ▾ Member ⋮

[@YurySolovyov](#) there's a lot of history here. This proposal has repeatedly failed to advance at TC39 (the committee in charge of JS standardization) precisely because it's not a great fit as a language feature. It doesn't expose any fundamentally new capabilities, it doesn't tie in to any syntax, and by staying at the language level it doesn't integrate well with popular platform APIs that would use it. As such there would be no advantage to putting it in the language, as opposed to just continuing to let people use libraries like RxJS. That's why it's been at stage 1 for over a year, unable to advance to stage 2 where "The committee expects the feature to be developed and eventually included in the standard".

The committee's feedback was to do precisely what [@benlesh](#) has done, and propose working with the DOM community to create a feature worth shipping in engines because it ends up with deep integrations with the platform.

Note that being specified in the DOM spec doesn't prevent it from being used across the different JS-using ecosystems, like Node.js. As we've seen so far with URL, `TextEncoder/TextDecoder`, and the performance\* APIs, there are a lot of APIs specified outside the core language which are available across many different JS platforms. I would expect `Observable` to become one of these, and indeed you can see the beginnings of that discussion happening above :).



mgol on Dec 14, 2017


[@domic](#)

This proposal has repeatedly failed to advance at TC39 (the committee in charge of JS standardization) precisely because it's not a great fit as a language feature. It doesn't expose any fundamentally new capabilities, it doesn't tie in to any syntax, and by staying at the language level it doesn't integrate well with popular platform APIs that would use it.

Out of curiosity - what was different in promises vs. observables that the former found their way in the core language then? Was `async-await` already in TC39 minds when promises were getting standardized?



dominic on Dec 14, 2017

Member ...

Yep, that and module loading.



2



rauchg on Dec 14, 2017



It doesn't expose any fundamentally new capabilities, it doesn't tie in to any syntax

Is is out of the question that a native `Observable` type could be `await`-ed? I think one of the core advantages of `Observable` in the first place is that they are an effective superset of `Promise`.



92 remaining items

Load more


[domic](#) mentioned this on Dec 2, 2021

[once\(\) and on\(\) async utilities for EventTarget #1038](#)

[domic](#) mentioned this on Jun 22, 2022

[Small features as a focus area for Interop 2023? web-platform-tests/interop#89](#)


domfarolino on Feb 15, 2023

Member ...

Hey everyone, I know this thread has been quiet for a while, but I've been looking at this proposal from an implementer perspective and I have a few questions, some of which stem from the fact that I think I only recently wrapped my head around the desired semantics here, so bear with me :)

#### 1. Why are Observables needed? Why is synchronous delivery important?

- I understand that the big difference between Observables and things like Streams / async iterators is:
  - [Push vs. Pull](#) (yes I read [@domic's SO answer](#))
  - Inherently multi-consumer
  - Synchronous delivery
- At the risk of sounding dumb, how critical are some of these constraints? I'm trying to figure out why an Observable's magic cannot be recovered by composing together other things like Streams, when you can use `tee()` on Streams to recover the multi-consumer behavior and generally [build](#) "Push" systems out of "Pull" systems. That leaves the synchronous delivery aspect, and I'd love to understand why it's important. When trying to figure this out, I was directed to <https://surma.dev/things/streams-for-reactive-programming/> which described Surma's toy `observables-with-streams` library. Naturally, the biggest difference between his Observables & true Observables is the [synchronous delivery](#) of data. I was surprised to find that it didn't cause any problems with the simple web app he wrote with his library, but I'd love to understand why today's existing, complex use-cases of real Observables could not tolerate that behavior, and what's so important about synchronous delivery?

#### 2. Is "subscriber" the same as "observer"?

- It seems like it, but the argument in the Observable constructor's callback is usually called `subscriber` (when we call `subscriber.next(foo)`), yet the thing passed in to the `Observable#subscribe()` method is called an Observer. I know that these do not reference the same underlying JS objects, but conceptually-speaking are these the same thing? Or is there another actor involved here called `Subscriber` that I'm missing?

#### 3. How big of a concern really, are the Promise-ifying APIs on Observable?

- Much ink has been spilled ([C](#), [C](#), [C](#), [C](#), [C](#), [C](#)) about how the microtask queue is consulted synchronously after browser-initiated event callbacks, but not for JS-initiated ones. The consequence is that JS-initiated events cannot be canceled (among other things) from within Promise callbacks when Promise-ifying an Observable's result (is "result" the right word?).
- [@jakearchibald](#) and others mention that this is kind of a [big deal](#). Obviously by implementing Observables in the browser itself, we'd be opening up their usage (and their Promise-ifying APIs) to people unfamiliar with Observables today who might find themselves a new footgun. But to estimate how much of a footgun it is, I'm interested in hearing if the reactive JS community has ever seen this as a footgun? Is this one of those things where everyone getting started with Observables has that moment where they say "Ah, I too used to get tripped up when trying to cancel my events from Promise-ifying Observable methods, until I figured it out". That's a bit contrived of course, but I'm trying to figure out if this has generally been a gotcha for newcomers or not?
- I ask because we're fortunate to have a robust set of community-curated Observable implementations in userland that already have this potential issue that the browser would not be introducing. Maybe we can estimate how much of a gotcha this might be, by learning how much of a gotcha it already is.

#### 4. How should I think about [whatwg/dom#544](#) vs [tc39/proposal-observable#201](#) (per [#issuecomment-541763071](#))?

- I apologize for not having read the entire [Simplification of Observable API tc39/proposal-observable#201](#) thread yet. Is the ideal proposal/API the old one, or the "simplified" one? What I'm getting at is: was the 2019 simplification a last-ditch effort to try and get it pushed through TC39, that made API compromises in exchange for process support, or is it a strict improvement that we'd pursue if starting over from scratch today with unlimited implementer/standards support?
- There is some discussion in the [Simplification of Observable API tc39/proposal-observable#201](#) simplification about how it's better suited for the "firehose" problem where an observable synchronously (I guess that goes without saying) produces a bunch of values that the Observer can luckily stop, given a token-based cancellation mechanism. How coupled is (a) the simplification proposal with (b) this cancellation mechanism? Are they orthogonal? Can the initial API proposal here just have the Observable constructor's `subscribe` function take in something like an `AbortSignal` that you pass in when calling `subscribe()`? I guess it might look like:

```
const obs = new Observable(subscriber => {
  for (i = 0; i < 99999999; ++i) {
    subscriber.next();

    if (subscriber.signal.aborted)
      break;
  }
})
```



```
});

AbortController controller;
obs.subscribe({
  next: val => {
    if (val > 100)
      controller.abort();
  },
  signal: controller.signal,
});
```

I'm just trying to tease out the cancelation vs the API simplification, and see if they are decoupled at all.



benlesh on Feb 17, 2023 · edited by benlesh

Edits Author ...

[@domfarolino](#) So that's a LOT of questions, and I'll try to answer as many as I can. There are a couple of things worth noting here:

1. This was a proposal made on Google's behalf when I was at Google :) (I still believe in it, though)
2. Note that there's no part of this that is the `Observable` constructor. At the time it was thought to be too contentious to introduce a way to create observables from scratch (although I do think that the platform would benefit from this primitive). However, I'm amenable to adding a way to create observables given new versions of them pop up in the wild in many notable libraries.

## Why are Observables needed? Why is synchronous delivery important?

### Observables as a primitive

Observables, at a basic level, are more primitive than Promises, Streams, or Async Iterables. They are, for lack of a better way to put it: "Specialized functions with some safety added".

Observables could be in *many* shapes, and don't need to be confined to what you've seen above in this proposal. Consider that a straw man.

If we wanted to build the ideal PRIMITIVE `Observable` :

1. Some ability to define a "setup" or "init" function: A way to tie some data producer (Or a "subject" to some "observer", this is the "observer pattern") synchronously when you "subscribe" (call the specialized function, aka "observe", "listen", et al in the wild)
2. A way to notify a "consumer" of each new value as *soon as possible* Such that when your data producing "subject" notifies, you can immediately notify whatever code cares. (In the wild this could be registering a listener, or handler, or observer, et al).
3. A way to notify a consumer that the producer is done producing values successfully
4. A way to notify a consumer than there was an error and the producer will stop sending values.
5. A means of the consumer telling the producer it no longer wants to receive values (cancellation, aborting, unsubscribing, et al)
6. Critically important safety if we're allowing people to *create their own observables*: A means of *tearing down or finalizing* the data producer in any terminal case above (the producer is done, there was an error, or the consumer no longer cares).
7. Critically important safety if we're allowing people to *\_create\_* their own observables: The consumer cannot be notified of anything after finalization (due to the producer being done, erroring, or unsubscribing)
8. A completely new call to the setup or initialization *per subscription*. (Note that you can still multicast with this via closure.)

### Observables in the DOM

In the DOM, we don't necessarily need the whole primitive (although it would be the most useful thing to have). Instead what the DOM needs is the consumer side of the contract. To make this really powerful, the ideal API is to have the whole primitive though.

how critical are some of these constraints?

If you want to be able to accurately model what we can do with `EventTarget` , observables must be able to emit synchronously.

```
console.log('start');
eventTarget.addEventListener('whatever', console.log);
eventTarget.dispatchEvent(new Event('whatever'));
eventTarget.removeEventListener('whatever', console.log);
console.log('stop');

// synchronously logs:
// "start"
// [object Object]
// "stop"
```



I was surprised to find that it didn't cause any problems with the simple web app he wrote with his library

I'm not surprised. I'm amenable to having the observable primitive be asynchronous... however, it's decidedly LESS primitive once you force scheduling:

1. If it's synchronous, you can always compose scheduling in. If it's already scheduling, you can't force it to be synchronous.
2. If it's not synchronous, it doesn't make a good cancellation notifier. (You'll notice that this proposal is trying to use `Observable` to cancel itself).

However, if it IS synchronous, there are definitely some ergonomics gotchas when working with it. For example if you're trying to write an operator in RxJS that may terminate a source (like `takeUntil` etc), you have to be careful to check to see if the cancellation notifier notified synchronously *before* you even subscribe to the source. And what if you wanted to write a `takeOneAfterThisNotifies` sort of operator? It gets harrier. This is why I'm amenable to `Observable` being asynchronous. However, I will say it does make it less useful (and probably more resource intensive) if it schedules (Again, the cancellation example above)

## Is "subscriber" the same as "observer"?

No. `Observer` would be an interface that really amounts to a few named handlers (`{ next, error, complete }`). Where `Subscriber` implements that interface, and generally "wraps" some observer to add the safety in I talked about above. Making sure you don't call `next` after `error` , for example. Or binding the observation to the subscription so it can teardown.

## How big of a concern really, are the Promise-ifying APIs on Observable?

These things have been available for use for quite some time, pretty much since the dawn of Promises, and I haven't seen any real-world issues arise out of it. That said, promisifying observables is more of an ergonomics thing, since JavaScript has gone "all in" on Promise-related APIs (`async/await`, for example). The entire world is used to the lack of cancellation in Promises. That said, I could take the promise features or leave them. But again, I've never seen promise conversion be the issue for observables. In fact, we're going to add `[Symbol.asyncIterator]` handling to RxJS in the next version. The observable *primitive* doesn't need them. It needs to simply *exist* so people stop reinventing it and shipping it out in their libraries.

## How should I think about #544 vs tc39/proposal-observable#201 (per #544 (comment))?

Think of it like this: [@benlesh](#) sees a pressing need to ship a real Observable primitive to the JavaScript community en masse, and I've tweaked my proposals several times over the years to try to reach an audience with the gatekeepers.

Some facts:

1. When the TC39 proposal was made, RxJS (even the older microsoft version) was under a million downloads *a month*. RxJS is now at 47,000,000+ downloads *per week*.
2. Several popular libraries have *completely reinvented* observables in their own code bases in various shapes, some of them as direct ports of RxJS's observable that were then altered, others just organically inventing their own (and maybe not really realizing it was an observable):

- React Router
- XState
- MobX
- Relay
- Recoil
- Apollo GraphQL
- Redux
- Vue
- Svelte
- tRPC
- RxJS

3. The debugging story for a native observable would be SO MUCH BETTER than what we can currently do with any of these homegrown "community driven" implementation.

I'm tired. I don't want to be the sole arbiter of the most popular Observable implementation. It's a primitive type. It should be in our platforms by any means necessary. We're shipping countless bytes over the wire to send these observables out to everyone, where if the *actual primitive existed* I believe more library implementors would use them. Especially if they performed well because they were native, and they had a better debugging story in dev tools (because they're native). I want RxJS to be a cute little library of helper methods over a native type that becomes slowly irrelevant as those features grow in the native platforms. (Thus completing the long-running statement "RxJS is lodash for events").

### What would I find to be acceptable criteria for a good Observable type in the platform?

See the requirements above. No "operators". Doesn't need any promise conversion stuff. That's all just "nice to have". This would be fine:

```
const source = new Observable(subscriber => {
  let n = 0;
  const id = setInterval(() => subscriber.next(n++), 1000);
  return () => clearInterval(id);
});

const ac = new AbortController()
const signal = ac.signal;

source.takeUntil(signal.on('abort')).subscribe(console.log);
```

But for the sake of **this** proposal: The main thing it was trying to do is make events more ergonomic. Like:

```
eventTarget.on('click').takeUntil(signal.on('abort')).subscribe(() => console.log('clicked'))
```

Ironically, [@benjaminrg](#), a LONG time supporter of RxJS may have killed any chance the community had at getting Observable on a platform when he added `{ signal }` to `addEventListener`. 🙄

```
eventTarget.addEventListener('click', () => console.log('clicked'), { signal }); // This works today.
```

The down side? The API we have above with `addEventListener` doesn't compose. It can't communicate when it's "done". It doesn't stop when there was an "error". Use cases that are covered by a real observable like dragging something aren't as straightforward. Imagine this:

```
div.on('mousedown').flatMap(downEvent => {
  const divX = div.getBoundingClientRect().left;
  const startX = downEvent.clientX;
  return document.on('mousemove')
    .map(moveEvent => divX + moveEvent.clientX - startX)
    .takeUntil(document.on('mouseup'))
})
.subscribe(x => {
  div.style.transform = `translate3d(${x}px, 0, 0)`;
})
```

It would be pretty cool to be able to do that *without* RxJS. The imperative version of that is easier to mess up in my opinion. And note that outside of `takeUntil`, the other methods are things that are already found in other JavaScript "iterable" sorts of things. ( `Observable` is the "inverse" of an `Iterable`, BTW... which is another reason it's synchronous, if we wanted to be nerds about it)

This WHATWG proposal was painted as the *best chance* Observable ever had. My discussions with [@domenic](#) years ago signaled that he didn't believe it belonged in the ECMAScript standard (so presumably it's blocked there), and it was his determination that proposing it here was the best idea.

I hope this answers all of your questions, [@domfarolino](#), and I hope you found this helpful. Please feel free to ask more. You know where to find me. (My github notifications are a hot mess because of RxJS, and I'm not paid to work on anything in open source or on GitHub, so if you ping me here I might not see)



domfarolino on Feb 18, 2023

Member ...

Thanks so much for the very thorough response [@benlesh](#). Some replies below:

Note that there's no part of this that is the Observable constructor. At the time it was thought to be too contentious to introduce a way to create observables from scratch (although I do think that the platform would benefit from this primitive). However, I'm amenable to adding a way to create observables given new versions of them pop up in the wild in many notable libraries.

Interesting. We of course could move incrementally, introducing Observable-returning APIs before making Observables themselves fully constructible by JS, however I do see value in providing the full API as a primitive up front. <https://w3ctag.github.io/design-principles/#constructors> recommends something similar, which is good news too.

Critically important safety if we're allowing people to create their own observables: A means of tearing down or finalizing the data producer in any terminal case above (the producer is done, there was an error, or the consumer no longer cares).

I agree with this requirement, but it doesn't seem unique to user-constructed observables, right? For example, if unsubscribing from the platform-constructed Observable returned by `EventTarget::on()` did not remove the subscription from the list of event listeners, that seems like a leak. Are there *additional* concerns here that are unique to user-constructed Observables?

#### Observables in the DOM

If you want to be able to accurately model what we can do with `EventTarget`, observables must be able to emit synchronously.



Thanks, the impact of being synchronous makes sense here, especially for events and cancellation. I agree that the sync semantics here are ideal and give the most flexibility.

Subscriber implements that interface, and generally "wraps" some observer to add the safety in I talked about above.

Exactly, this is what I was asking about. Just wanted to make sure I understood the relationship between the two.

Or binding the observation to the subscription so it can teardown.

Can you clarify this (who exactly is tearing down here)? I understand that if `complete()/error()` is called on the `Subscriber/wrapped Observer`, we'd want to unregister "the subscription" from the event source for clean-up (it doesn't make sense to keep listening for events and having the "safe" `Subscriber` object just *not* call `next()` forever). Is that what you mean?

RE promises: It's good to know that this hasn't been seen as fraught territory so far in the Observable-using community, though I do agree that for something like an Observables MVP, the Promise-ifying APIs could probably be initially left out.



bleistvt on Feb 18, 2023 · edited by bleistvt

Edits ···

For developer ergonomics to improve, Observable has to ship with a complete operator library in my opinion. A standard `Observable` without operators would only benefit library authors and could just as well be a userland standard.

Ironically, benjamngr, a LONG time supporter of RxJS may have killed any chance the community had at getting Observable on a platform when he added `{ signal }` to `addEventListener`.

This might be a case of "worse is better" since it covers the large majority of use cases where no complex combination of event streams etc. is required.

This is how the drag example above looks using `AbortSignal`:

```
div.addEventListener("mousedown", (downEvent) => {
  const divX = div.getBoundingClientRect().left
  const startX = downEvent.clientX
  const abortController = new AbortController()
```



Open

Improving ergonomics of events with Observable #544



```
div.style.transform = `translate3d(${x}px, 0, 0)`
}, {signal: abortController.signal})

document.addEventListener("mouseup", () => abortController.abort(), {once: true})
})
```

Sure, the API is not as nice, but I would argue that it is easier to understand for most developers today and works in all modern browsers. The order of events is also preserved when reading the code (`mousedown`, `mousemove`, `mouseup`).

There is a pitfall however: it is easy to create a memory leak by forgetting to add `{ once }` or `{ signal }`. This might be avoidable with a helper like `abortController.abortOn(target: EventTarget, eventName: string)`.

Once observables are part of the language, developers will expect [common operators](#) to be present. I don't really see how ergonomics are improved if everyone still has to import a library for operators.

After advocating for `addEventListener` over `on*` and `querySelectorAll` over `jQuery` for years, tutorials using event streams and `EventTarget.on` + "import this operator library" as the new best way would be highly confusing for learners.



benjamngr on Feb 21, 2023

Member ···

For what it's worth I disagree with:

Ironically, benjamngr, a LONG time supporter of RxJS may have killed any chance the community had at getting Observable on a platform when he added `{ signal }` to `addEventListener`.

I acknowledge it covers one thing observables did better than `EventTarget` (and we absolutely *should* keep improving `EventTarget`), but it doesn't really compare I think?

You can't `map` events with `EventTarget` (you can however wrap them in an async iterator and `.map` that). When you have synchronous events that matter (like a trusted user event triggering opening a window - where it has to be synchronous) - you can't use async iterators (micro-tick scheduling) and the semantics are clearly push based. Like you said - nothing composes, this has always been the bigger deal for me.

The thing that has been blocking observables to my knowledge for the last few years is that they were not presented to the committee and concerns raised and research did not progress because no one is funding it. I don't expect you to spend another 1000 hours for free on this, but let's remember we can still have observables in JavaScript if someone funds the work to do it or does it. I think WhatWG's and TC39's concerns can be addressed to everyone's happiness.



domfarolino mentioned this on Mar 25, 2023

[The constructor doesn't have a way to register teardown, WICG/observable#3](#)



domfarolino on Jul 28, 2023

Member ···

Hey everyone, just to follow up on this thread: @benlesh and I have been working on this recently over at <https://github.com/domfarolino/observable> where we have a more formal explainer for this proposal, so feel free to engage over there if you'd like! At some point with a few more details ironed out, I'd like to send an Intent-to-Prototype out for Chromium to express formal interest.



ljharb on Jul 29, 2023

···

@domfarolino as i recall, the main signal TC39 was missing was browser interest, so if Chrome is interested, it is *definitely* worth bringing back to TC39.








benlesh mentioned this on Sep 8, 2023

[Observable API WICG/proposals#118](#)

domfarolino mentioned this on Jan 31, 2024

[Spec the forEach\(\) operator WICG/observable#105](#)

-   **domfarolino** mentioned this in 2 issues on Jun 6, 2024
-  [Requesting a position on the Observable API wintercg/proposal-minimum-common-api#72](#)
-  [Observable API nodejs/standards-positions#1](#)
-   **domfarolino** mentioned this on Jun 20, 2024
-  [Should the platform encourage stream-style handling of events? WICG/observable#56](#)




Add a comment


H B I |    |    | @   

Write

Preview

Use Markdown to format your comment

 Paste, drop, or click to add files

 Close issue

Comment

 Remember, contributions to this repository should follow its [contributing guidelines](#).

Assignees

No one assigned

Labels

[addition/proposal](#) [needs implementer interest](#) [topic: events](#)

Type

No type

Projects

No projects

Milestone

No milestone

Relationships

None yet

Development

No branches or pull requests

Notifications

Customize

 Subscribe

You're not receiving notifications from this thread.

Participants

     +29