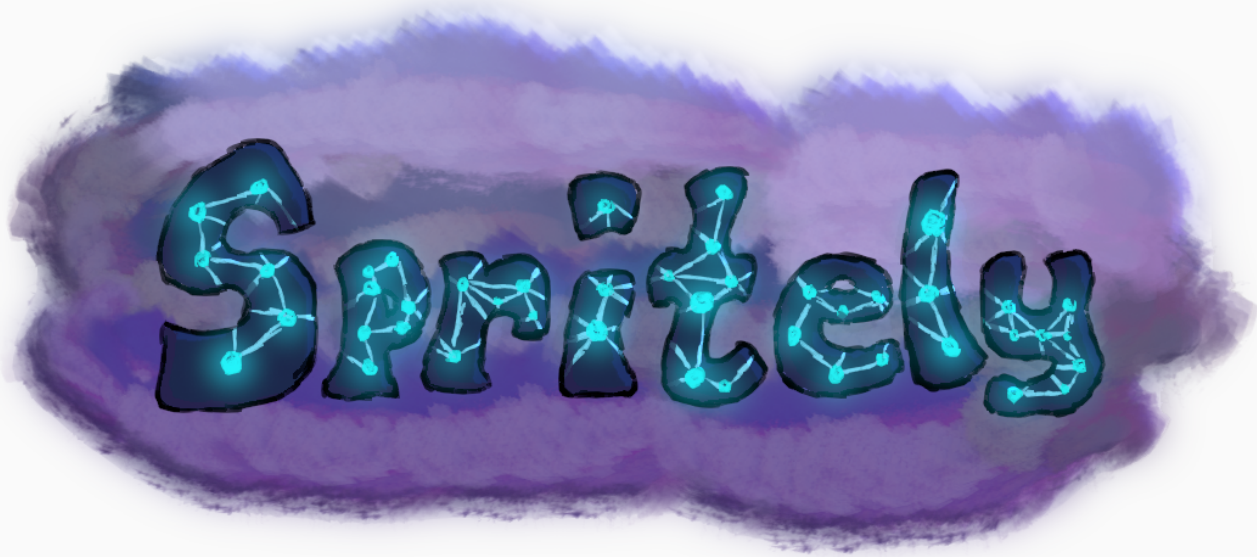
Re-decentralizing the web platform with Wasm GC

David Thompson

Sunday, February 1st, 2026

Hi, I'm Dave



https://spritely.institute

WebAssembly and I

- I'm a user and implementer of the Wasm spec
- Hoot is a Scheme → Wasm compiler
- Hoot is a full Wasm toolchain (including interpreter)

`https://spritely.institute/hoot`

Monolithic web browsers and monopoly control

- The Web has grown too big to implement unless you are Google (or funded by Google)
- Some have tried to make an alternative browser but it's an uphill battle
- We need to subdivide the problem

WebAssembly?

- What might a more modular browser look like?
- What if a browser was little more than a Wasm runtime?

From here to there

- Today: high-trust, monolithic C++ core with JavaScript userspace
- The vision: minimal core for OS access with Wasm userspace
- Polyglot environment: Use different languages for different tasks

Vision of a decentralized web platform

joey/ blog/ entry/ WASM Wayland Web (WWW)

search

Edit RecentChanges History Preferences Branchable 3 comments

So there are only 2 web browser engines, and it seems likely there will soon only be 1, and making a whole new web browser from the ground up is effectively impossible because the browsers vendors have weaponized web standards complexity against any newcomers. Maybe eventually someone will succeed and there will be 2 again. Best case. What a situation.

So throw out all the web standards. Make a browser that just runs WASM blobs, and gives them a surface to use, sorta like Wayland does. It has tabs, and a throbber, and urls, but no HTML, no javascript, no CSS. Just HTTP of WASM blobs.

This is where the web browser is going eventually anyway, except in the current line of evolution it will be WASM with all the web standards complexity baked in and reinforcing the current situation.

Would this be a mass of proprietary software? Have you looked at any corporate website's "source" lately? But what's important is that this would make it easy enough to build new browsers that they would stop being a point of control.

Want a browser that natively supports RSS? Poll the feeds, make a UI, download the WASM enclosures to view the posts. Want a browser that supports IPFS or gopher? Fork any browser and add it, the mantenance load will be minimal. Want to provide access to GPIO pins or something? Add an extension that can be accessed via the WASI component model. This would allow for so many things like that which won't and can't happen with the current market duopoly browser situation.

And as for your WASM web pages, well you can still use HTML if you like. Use the WASI component model to pull in a HTML engine. It doesn't need to support *everything*, just the parts of web standards that you want to use. Or you can do something entitely different in your WASM that is not HTML based at all but a better paradigm (oh hi Spritely or display postscript or gemini capsules or whatever).

Dual innovation sources or duopoly? I know which I'd prefer. This is not my project to build though.

`https://joeyh.name/blog/entry/WASM_Wayland_Web_WWW/`

WebAssembly!

- Portable, stack-based virtual machine
  - Lower level than JS
  - Higher level than your average assembly language
  - Statically typed
  - Structured control flow (no `goto`)
  - Import/export system for composition
- Designed to be a good compilation target, faster than JS, with quick boot time
  - Successor to asm.js

Simple Wasm example

```
(module
 (func (export "factorial") (param $n i32) (result i32)
       (local $result i32)
       (local.set $result (i32.const 1))
       (loop $loop (result i32)
        (if (result i32)
            (i32.eq (local.get $n) (i32.const 1))
            (then (local.get $result))
            (else (local.set $result
                             (i32.mul (local.get $n) (local.get $result)))
                  (local.set $n (i32.sub (local.get $n) (i32.const 1)))
                  (br $loop))))))
```

Sketch of a modular browser

- Nearly everything is an extension
  - The Emacs of Browsers™
- Implementation of Web standards as Wasm modules that can be linked together
- Modules could be swapped out or not installed at all

Sketch of a modular browser (continued)

- Minimal browsers with support for a subset of the Web become feasible
  - e.g. a small Electron distro tailored to a specific application
- Major browsers could look more like Linux distros
  - A curated collection of modules

Problem: Decoupling means coordination

- How can modules share references?
- How can modules know when another module is done using a reference?

How about the component model?

- WebAssembly System Interface (WASI) introduces the component model
- Takes a borrow checking approach to coordination
  - Assertion: this is insufficient for the web
  - Built for non-web use cases, anyway

> When a `resource` type name is wrapped with `borrow<...>`, it stands for a "borrowed" resource. A borrowed resource represents a temporary loan of a resource from the caller to the callee for the duration of the call. In contrast, when the owner of an owned resource drops that resource, the resource is destroyed. (Dropping the resource means either explicitly dropping it if the underlying programming language supports that, or returning without transferring ownership to another function.)

`https://component-model.bytecodealliance.org/design/wit.html`

Solution: Garbage collection

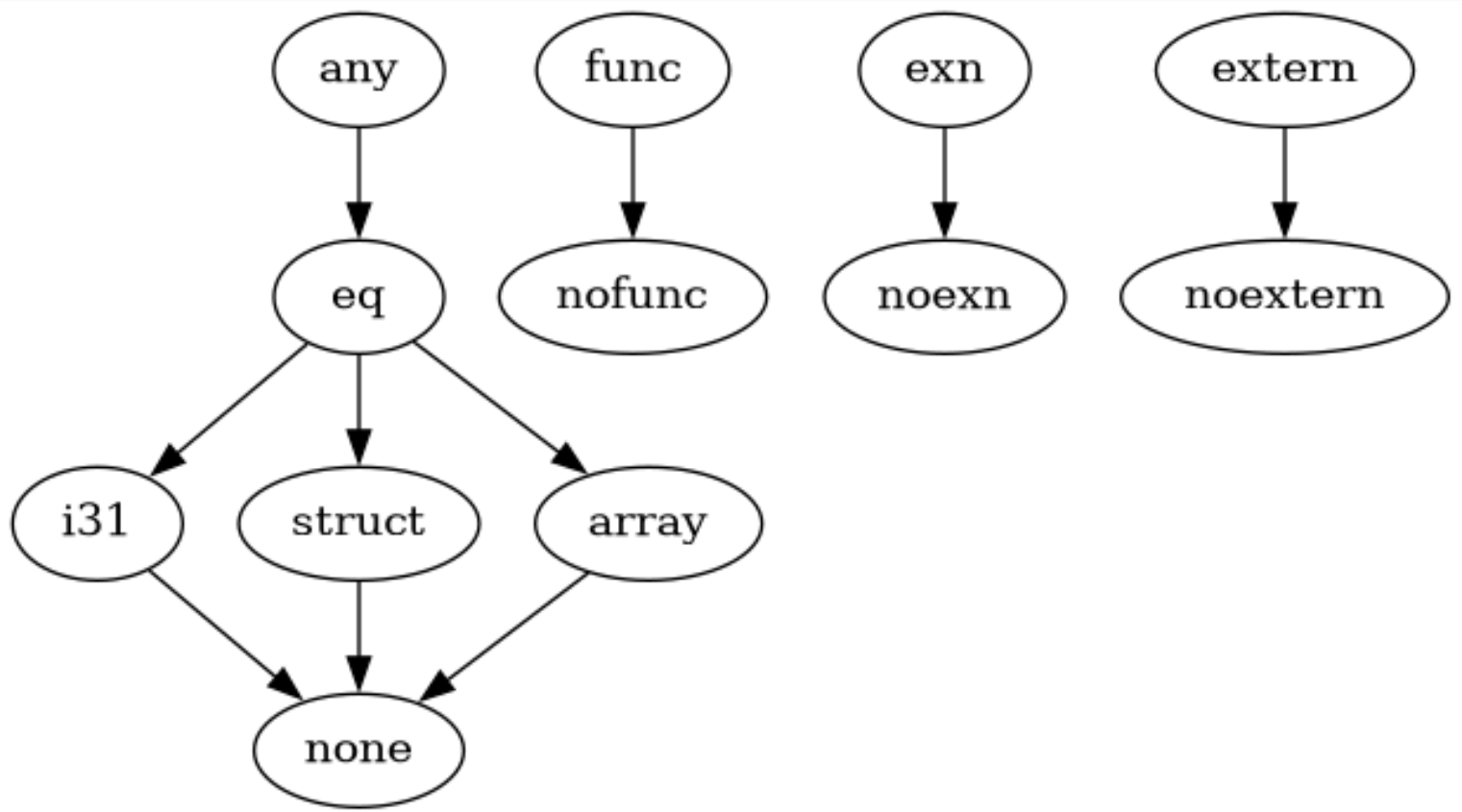
- Wasm 1.0 has linear memory only

```
(i32.store $memory offset=0
           (i32.const 64) (i32.const 42))
```

- Wasm 3.0 has GC-managed reference types

```
(struct.new $pair
            (i32.const 0)
            (ref.i31 (i32.const 17))
            (ref.i31 (i32.const 1)))
```

Reference type hiearchy



For the GC skeptics

- Rust *could* use Wasm GC, too!
- Borrowing within module; GC between modules

Problem: Decoupling means distributing trust

- Browser is a secure environment
- Reduced coupling requires *even more security*
- How can modules collaborate safely?

Solution: Capabilities

- If you don't *have* it, you can't *use* it
- It's just like passing arguments to functions
- Lexical scope as a security model
- No ambient authority

Contrived example

Ambient global state:

```
function setValueAndFocus(id, val) {
  const elem = document.findElementById(id);
  elem.value(val);
  elem.focus();
}
```

Explicit reference passing:

```
function setValueAndFocus(elem) {
  elem.value(val);
  elem.focus();
}
```

Not a new idea

A.I. Memo No. 1564

MASSACHUSETTS INSTITUTE OF TECHNOLOGY ARTIFICIAL INTELLIGENCE LABORATORY

March 1996

A Security Kernel Based on the Lambda Calculus

Jonathan A. Rees

This publication can be retrieved by anonymous ftp to publications.ai.mit.edu.

Abstract

Cooperation between independent agents depends upon establishing a degree of security. Each of the cooperating agents needs assurance that the cooperation will not endanger resources of value to that agent. In a computer system, a computational mechanism can assure safe cooperation among the system's users by mediating resource access according to desired security policy. Such a mechanism, which is called a *security kernel*, lies at the heart of many operating systems and programming environments.

Capability security has good UX

Not One Click for Security

Alan H. Karp
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304
alan.karp@hp.com

Marc Stiegler
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304
marc.d.stiegler@hp.com

Tyler Close
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304
tyler.close@hp.com

ABSTRACT

Conventional wisdom holds that security must negatively affect usability. We have developed SCoopFS (Simple Cooperative File Sharing) as a demonstration that need not be so. SCoopFS addresses the problem of sharing files, both with others and with ourselves across machines. Although SCoopFS provides server authentication, client authorization, and end-to-end encryption, the user never sees any of that. The user interface and underlying infrastructure are designed so that normal user acts of designation provide all the information needed to make the desired security decisions. While SCoopFS is a useful tool, it may be more important as a demonstration of the usability that comes from designing the infrastructure and user interaction together.

Categories and Subject Descriptors

H.1.2 [**User/Machine Systems**] Human factors; H.5.2 [**User Interfaces**]: User-centered design; H.5.3 [**Group and Organization Interfaces**]: Computer-supported cooperative work; K.4.3 [**Organizational Impacts**]: Computer-supported cooperative work

for SCoopFS[1], a system for Simple Cooperative File Sharing.

Even the people who never use anything but email for sharing work on documents voiced the same complaint. "You've got to remember to send the latest version and apply the updates when they come in." In addition, most of them reported resorting to convoluted conventions to avoid losing work due to edit conflicts. A few even mentioned the lack of security, since almost nobody bothers to encrypt email. Several people wanted to share files between Windows and Linux machines. We designed SCoopFS to address these requirements. This paper discusses the interaction design of SCoopFS. Details of its implementation will be reported separately.

SCoopFS demonstrates two points. First, it shows that managing rights at a fine granularity is easier than dealing with them in large chunks. Second, SCoopFS shows that, at the very least, security need not impede usability and can be largely invisible to the user. The degree to which this view is counter to people's intuitions forced us to change the name of our project. The first "S" in SCoopFS originally stood for "Secure", but several prospective

Wasm is a capability system

- Wasm binaries have no privileges by default
- Imports passed in at module instantiation time
- Imports may be host functions or exports from another Wasm module
- Imports are compile-time capabilities
- References are runtime capabilities

Example

Wasm:

```
(func $bignum-from-i32
      (import "rt" "bignum_from_i32")
      (param i32)
      (result (ref extern)))
```

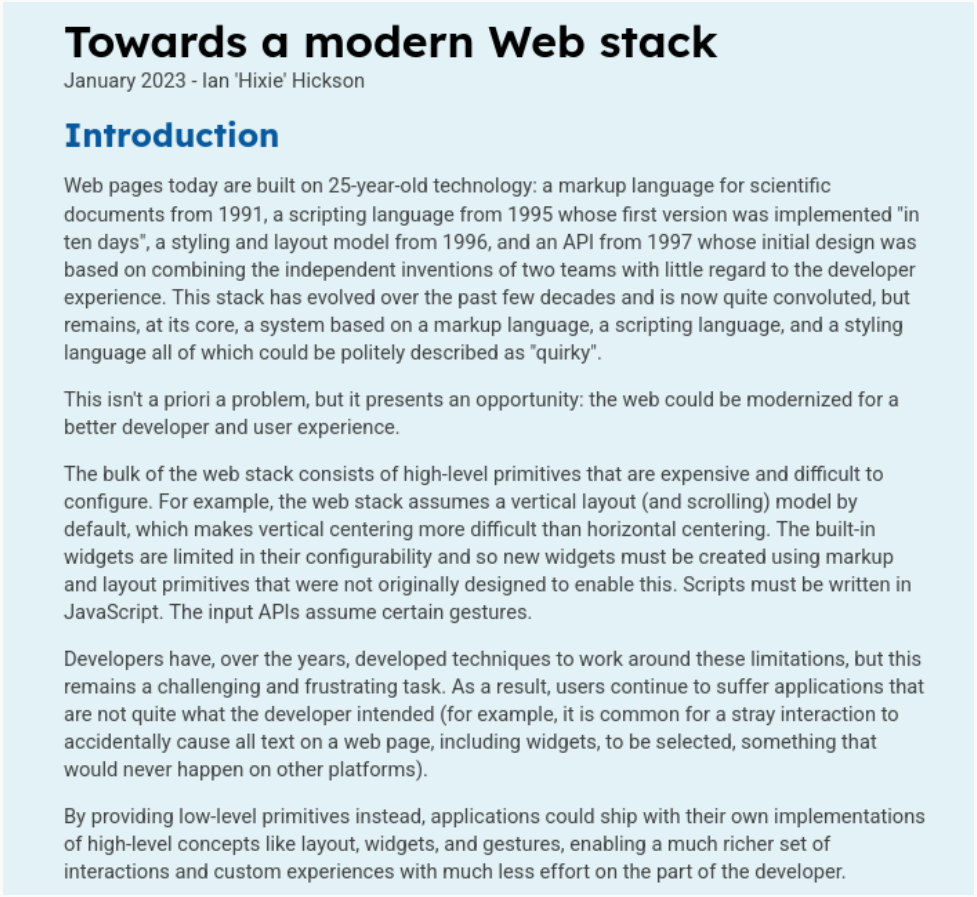
JS:

```
WebAssembly.instantiate(blob, {
  rt: {
    bignum_from_i32(n) { return BigInt(n); }
  }
});
```

The instantiator is in control

```
WebAssembly.instantiate(blob, {
  rt: {
    bignum_from_i32(n) { return "nope"; }
  }
});
```

A rift in the web: applications vs. documents



Towards a modern Web stack

January 2023 - Ian 'Hixie' Hickson

Introduction

Web pages today are built on 25-year-old technology: a markup language for scientific documents from 1991, a scripting language from 1995 whose first version was implemented "in ten days", a styling and layout model from 1996, and an API from 1997 whose initial design was based on combining the independent inventions of two teams with little regard to the developer experience. This stack has evolved over the past few decades and is now quite convoluted, but remains, at its core, a system based on a markup language, a scripting language, and a styling language all of which could be politely described as "quirky".

This isn't a priori a problem, but it presents an opportunity: the web could be modernized for a better developer and user experience.

The bulk of the web stack consists of high-level primitives that are expensive and difficult to configure. For example, the web stack assumes a vertical layout (and scrolling) model by default, which makes vertical centering more difficult than horizontal centering. The built-in widgets are limited in their configurability and so new widgets must be created using markup and layout primitives that were not originally designed to enable this. Scripts must be written in JavaScript. The input APIs assume certain gestures.

Developers have, over the years, developed techniques to work around these limitations, but this remains a challenging and frustrating task. As a result, users continue to suffer applications that are not quite what the developer intended (for example, it is common for a stray interaction to accidentally cause all text on a web page, including widgets, to be selected, something that would never happen on other platforms).

By providing low-level primitives instead, applications could ship with their own implementations of high-level concepts like layout, widgets, and gestures, enabling a much richer set of interactions and custom experiences with much less effort on the part of the developer.

`https://docs.google.com/document/d/1peUSMsvFGvqD5yKh3GprskLC3KVdAlLGOsK6gFoEOD0/preview`

Wasm has room to grow

- The GC support in Wasm today is MVP
- Promising proposals
  - Stack switching
  - Multibyte array access
  - Reference-typed strings

Summary

- The near-monopoly for browser engines is unhealthy
- We need to break up the monolith
- Wasm is a suitable low-level substrate for safe collaboration
- We should push Wasm to its limit
- Let's take back the web!

Thanks! Questions?

https://spritely.institute/donate