HACKS

Search Mozilla Hacks

WebAssembly's post-MVP future: A cartoon skill tree

By Lin Clark, Till Schneidereit, Luke Wagner

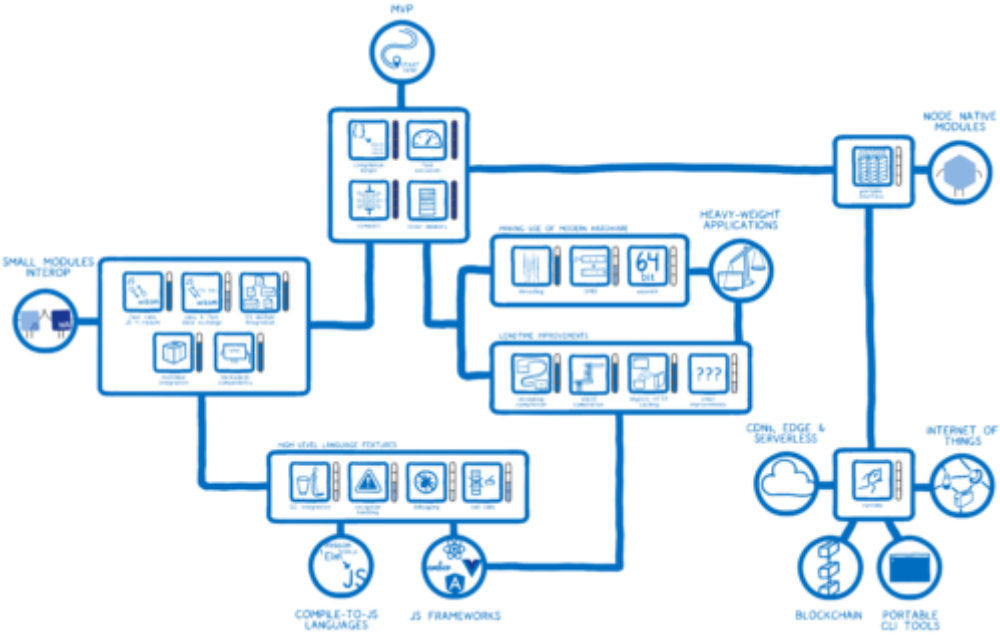
Posted on October 22, 2018 in Code Cartoons and Featured Article

People have a misconception about WebAssembly. They think that the WebAssembly that landed in browsers back in 2017—which we called the minimum viable product (or MVP) of WebAssembly—is the final version of WebAssembly.

I can understand where that misconception comes from. The WebAssembly community group is really committed to backwards compatibility. This means that the WebAssembly that you create today **will** continue working on browsers into the future.

But that doesn't mean that WebAssembly is feature complete. In fact, that's far from the case. There are many features that are coming to WebAssembly which will fundamentally alter what you can do with WebAssembly.

I think of these future features kind of like the skill tree in a videogame. We've fully filled in the top few of these skills, but there is still this whole skill tree below that we need to fill-in to unlock all of the applications.



So let's look at what's been filled in already, and then we can see what's yet to come.

Minimum Viable Product (MVP)

The very beginning of WebAssembly's story starts with Emscripten, which made it possible to run C++ code on the web by transpiling it to JavaScript. This made it possible to bring large existing C++ code bases, for things like games and desktop applications, to the web.

The JS it automatically generated was still significantly slower than the comparable native code, though. But Mozilla engineers found a type system hiding inside the generated JavaScript, and figured out how to make this JavaScript run really fast. This subset of JavaScript was named asm.js.

Once the other browser vendors saw how fast asm.js was, they started adding the same optimizations to their engines, too.

But that wasn't the end of the story. It was just the beginning. There were still things that engines could do to make this faster.

But they couldn't do it in JavaScript itself. Instead, they needed a new language—one that was designed specifically to be compiled to. And that was WebAssembly.

So what skills were needed for the first version of WebAssembly? What did we need to get to a minimum viable product that could actually run C and C++ efficiently on the web?

Skill: Compile target

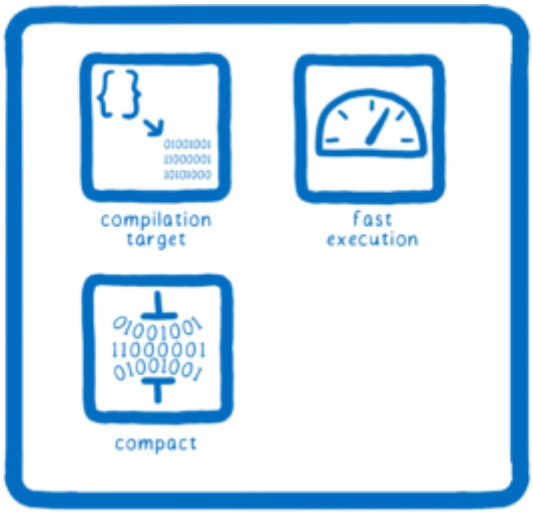
The folks working on WebAssembly knew they didn't want to just support C and C++. They wanted many different languages to be able to compile to WebAssembly. So they needed a language-agnostic compile target.

They needed something like the assembly language that things like desktop applications are compiled to —like x86. But this assembly language wouldn't be for an actual, physical machine. It would be for a conceptual machine.

Skill: Fast execution

That compiler target had to be designed so that it could run very fast. Otherwise, WebAssembly applications running on the web wouldn't keep up with users' expectations for smooth interactions and game play.

Skill: Compact

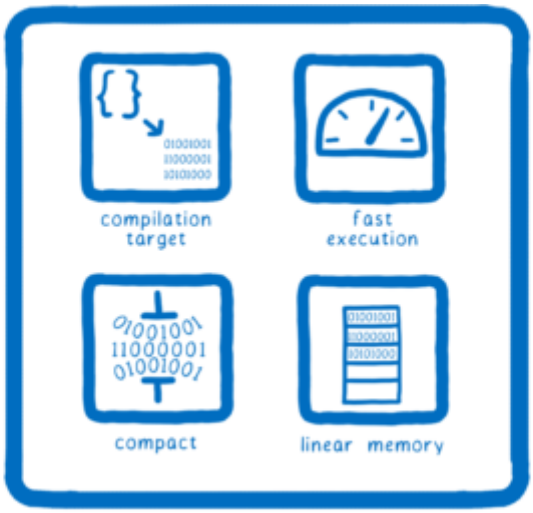


In addition to execution time, load time needed to be fast, too. Users have certain expectations about how quickly something will load. For desktop applications, that expectation is that they will load quickly because the application is already installed on your computer. For web apps, the expectation is also that load times will be fast, because web apps usually don't have to load nearly as much code as desktop apps.

When you combine these two things, though, it gets tricky. Desktop applications are usually pretty large code bases. So if they are on the web, there's a lot to download and compile when the user first goes to the URL.

To meet these expectations, we needed our compiler target to be compact. That way, it could go over the web quickly.

Skill: Linear memory



These languages also needed to be able to use memory differently from how JavaScript uses memory. They needed to be able to directly manage their memory—to say which bytes go together.

This is because languages like C and C++ have a low-level feature called pointers. You can have a variable that doesn't have a value in it, but instead has the memory address of the value. So if you're going to support pointers, the program needs to be able to write and read from particular addresses.

But you can't have a program you downloaded from the web just accessing bytes in memory willy-nilly, using whatever addresses they want. So in order to create a secure way of giving access to memory, like a

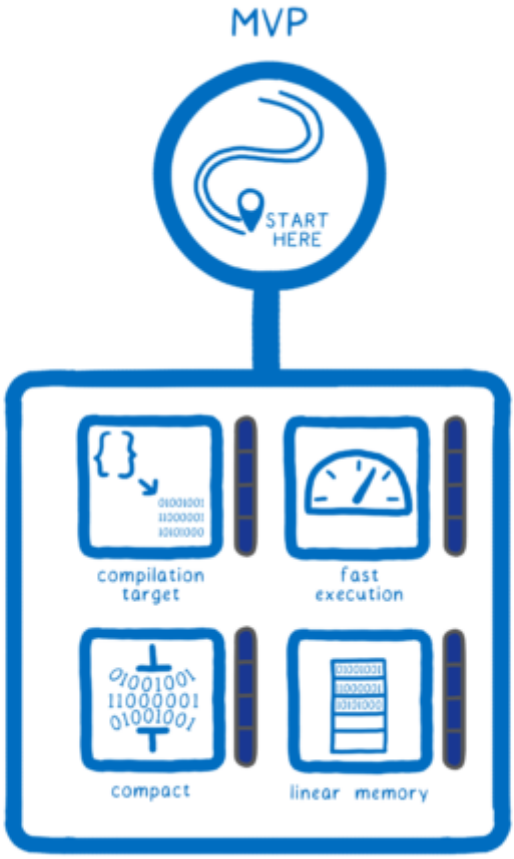
native program is used to, we had to create something that could give access to a very specific part of memory and nothing else.

To do this, WebAssembly uses a linear memory model. This is implemented using TypedArrays. It's basically just like a JavaScript array, except this array only contains bytes of memory. When you access data in it, you just use array indexes, which you can treat as though they were memory addresses. This means you can pretend this array is C++ memory.

Achievement unlocked

So with all of these skills in place, people could run desktop applications and games in your browser as if they were running natively on their computer.

And that was pretty much the skill set that WebAssembly had when it was released as an MVP. It was truly an MVP—a minimum viable product.



This allowed certain kinds of applications to work, but there were still a whole host of others to unlock.

Heavy-weight Desktop Applications

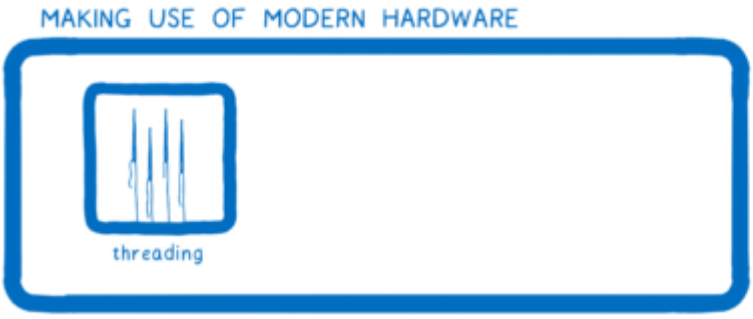
The next achievement to unlock is heavier weight desktop applications.

Can you imagine if something like Photoshop were running in your browser? If you could instantaneously load it on any device like you do with Gmail?

We've already started seeing things like this. For example, Autodesk's AutoCAD team has made their CAD software available in the browser. And Adobe has made Lightroom available through the browser using WebAssembly.

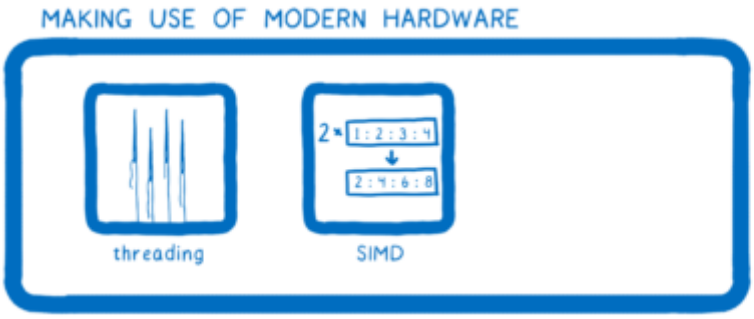
But there are still a few features that we need to put in place to make sure that all of these applications—even the heaviest of heavy weight—can run well in the browser.

Skill: Threading



First, we need support for multithreading. Modern-day computers have multiple cores. These are basically multiple brains that can all be working at the same time on your problem. That can make things go much faster, but to make use of these cores, you need support for threading.

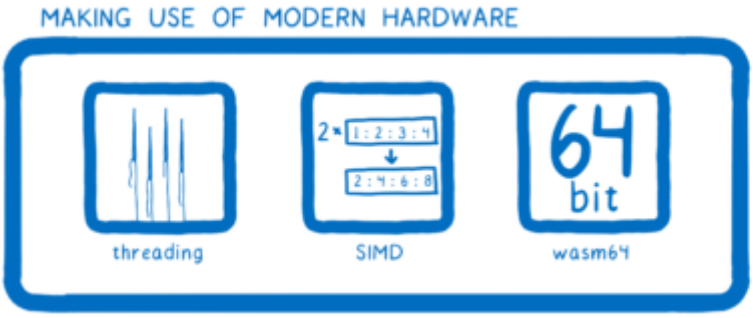
Skill: SIMD



Alongside threading, there's another technique that utilizes modern hardware, and which enables you to process things in parallel.

That is SIMD: single instruction multiple data. With SIMD, it's possible to take a chunk of memory and split up across different execution units, which are kind of like cores. Then you have the same bit of code—the same instruction—run across all of those execution units, but they each apply that instruction to their own bit of the data.

Skill: 64-bit addressing

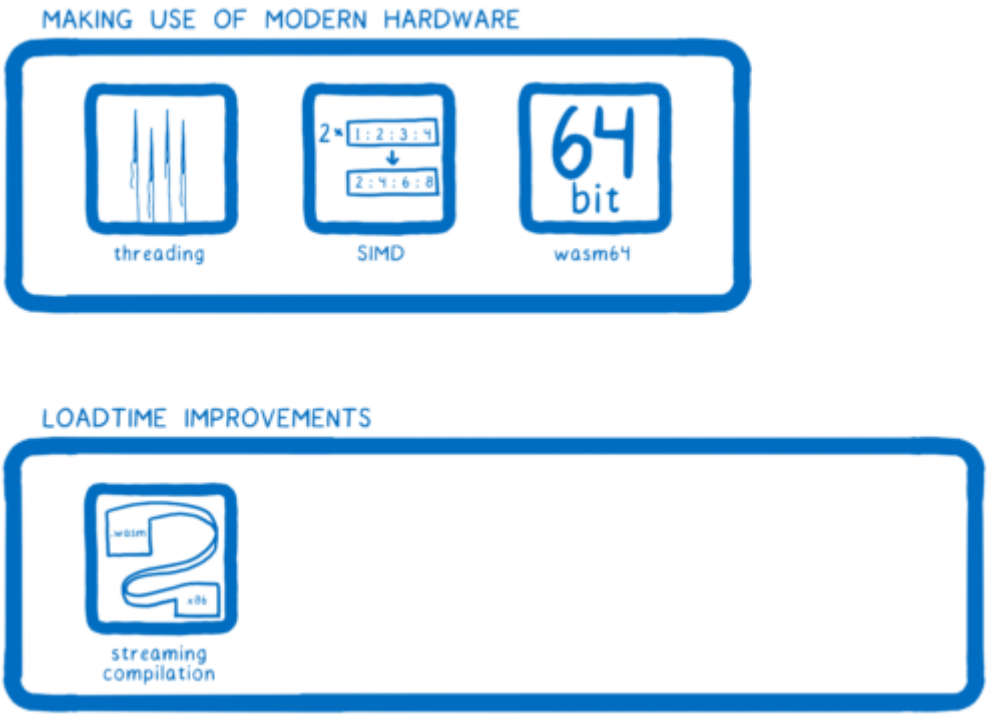


Another hardware capability that WebAssembly needs to take full advantage of is 64-bit addressing.

Memory addresses are just numbers, so if your memory addresses are only 32 bits long, you can only have so many memory addresses—enough for 4 gigabytes of linear memory.

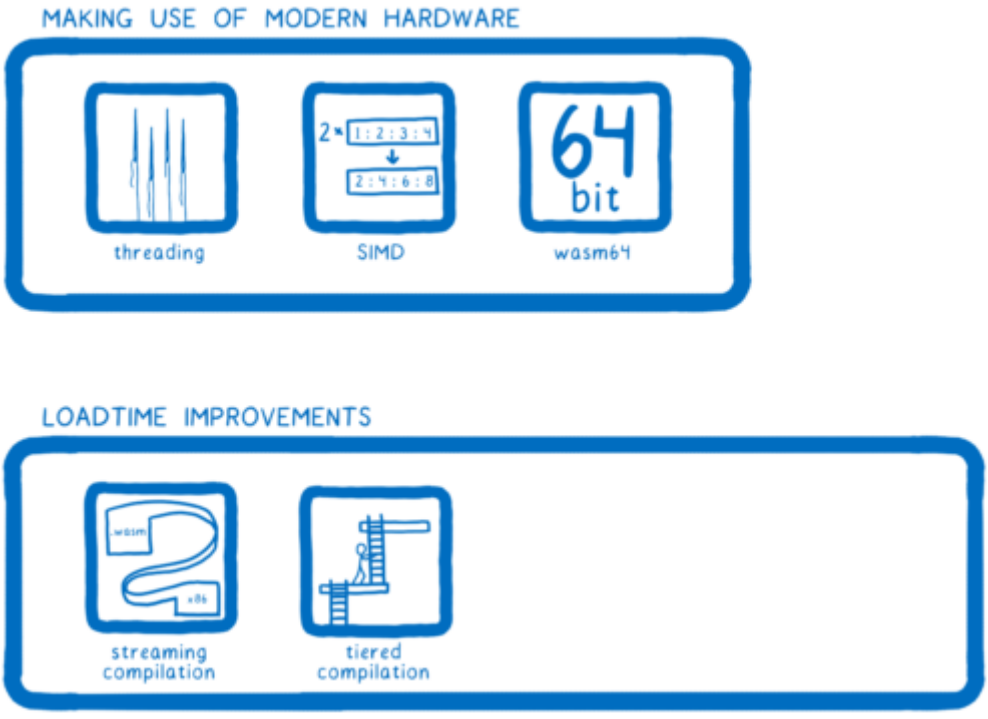
But with 64-bit addressing, you have 16 exabytes. Of course, you don't have 16 exabytes of actual memory in your computer. So the maximum is subject to however much memory the system can actually give you. But this will take the artificial limitation on address space out of WebAssembly.

Skill: Streaming compilation



For these applications, we don't just need them to run fast. We needed load times to be even faster than they already were. There are a few skills that we need specifically to improve load times.

One big step is to do streaming compilation—to compile a WebAssembly file while it's still being downloaded. WebAssembly was designed specifically to enable easy streaming compilation. In Firefox, we actually compile it so fast—faster than it is coming in over the network— that it's pretty much done compiling by the time you download the file. And other browsers are adding streaming, too.



Another thing that helps is having a tiered compiler.

For us in Firefox, that means having two compilers. The first one—the baseline compiler—kicks in as soon as the file starts downloading. It compiles the code really quickly so that it starts up quickly.

The code it generates is fast, but not 100% as fast as it could be. To get that extra bit of performance, we run another compiler—the optimizing compiler—on several threads in the background. This one takes longer to compile, but generates extremely fast code. Once it's done, we swap out the baseline version with the fully optimized version.

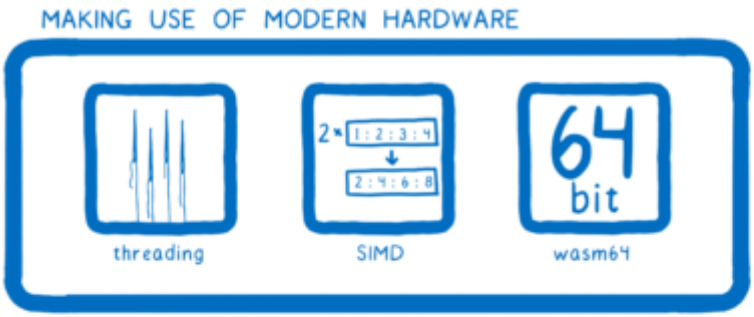
This way, we get quick start up times with the baseline compiler, and fast execution times with the optimizing compiler.

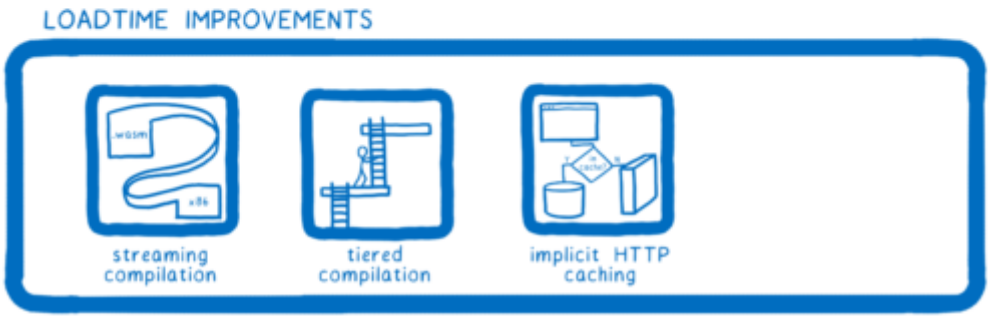
In addition, we're working on a new optimizing compiler called Cranelift. Cranelift is designed to compile code quickly, in parallel at a function by function level. At the same time, the code it generates gets even better performance than our current optimizing compiler.

Cranelift is in the development version of Firefox right now, but disabled by default. Once we enable it, we'll get to the fully optimized code even quicker, and that code will run even faster.

But there's an even better trick we can use to make it so we don't have to compile at all most of the time...

Skill: Implicit HTTP caching

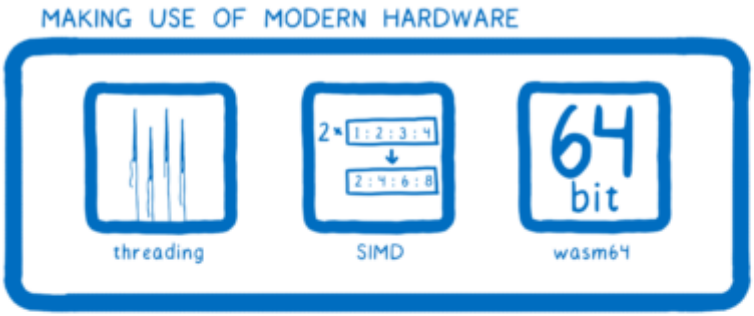


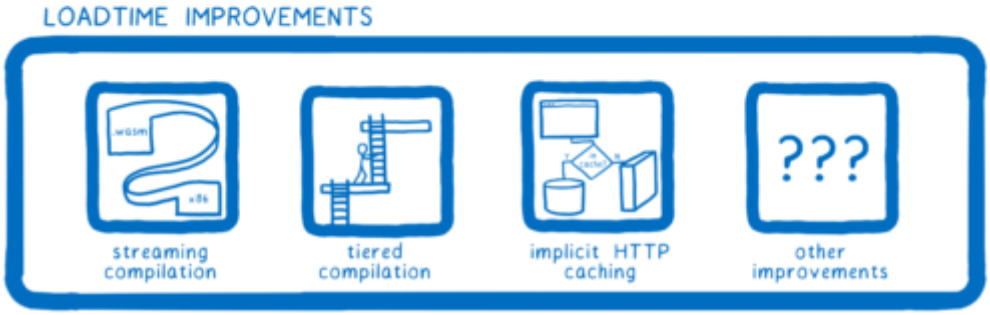


With WebAssembly, if you load the same code on two page loads, it will compile to the same machine code. It doesn't need to change based on what data is flowing through it, like the JS JIT compiler needs to.

This means that we can store the compiled code in the HTTP cache. Then when the page is loading and goes to fetch the .wasm file, it will instead just pull out the precompiled machine code from the cache. This skips compiling completely for any page that you've already visited that's in cache.

Skill: Other improvements

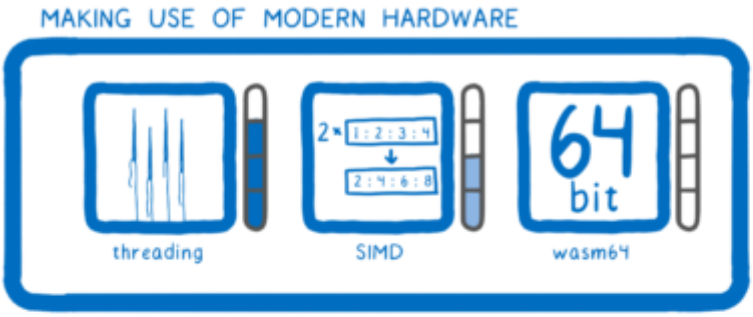


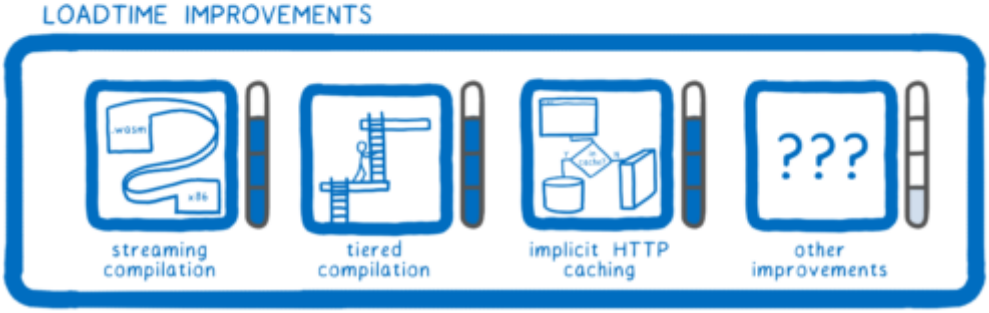


Many discussions are currently percolating around other ways to improve this, skipping even more work, so stay tuned for other load-time improvements.

Where are we with this?

Where are we with supporting these heavyweight applications right now?





Threading

For the threading, we have a proposal that's pretty much done, but a key piece of that—SharedArrayBuffers—had to be turned off in browsers earlier this year.
They will be turned on again. Turning them off was just a temporary measure to reduce the impact

of the Spectre security issue that was discovered in CPUs and disclosed earlier this year, but progress is being made, so stay tuned.

SIMD

SIMD is under very active development at the moment.

64-bit addressing

For wasm-64, we have a good picture of how we will add this, and it is pretty similar to how x86 or ARM got support for 64 bit addressing.

Streaming compilation

We added streaming compilation in late 2017, and other browsers are working on it too.

Tiered compilation

We added our baseline compiler in late 2017 as well, and other browsers have been adding the same kind of architecture over the past year.

Implicit HTTP caching

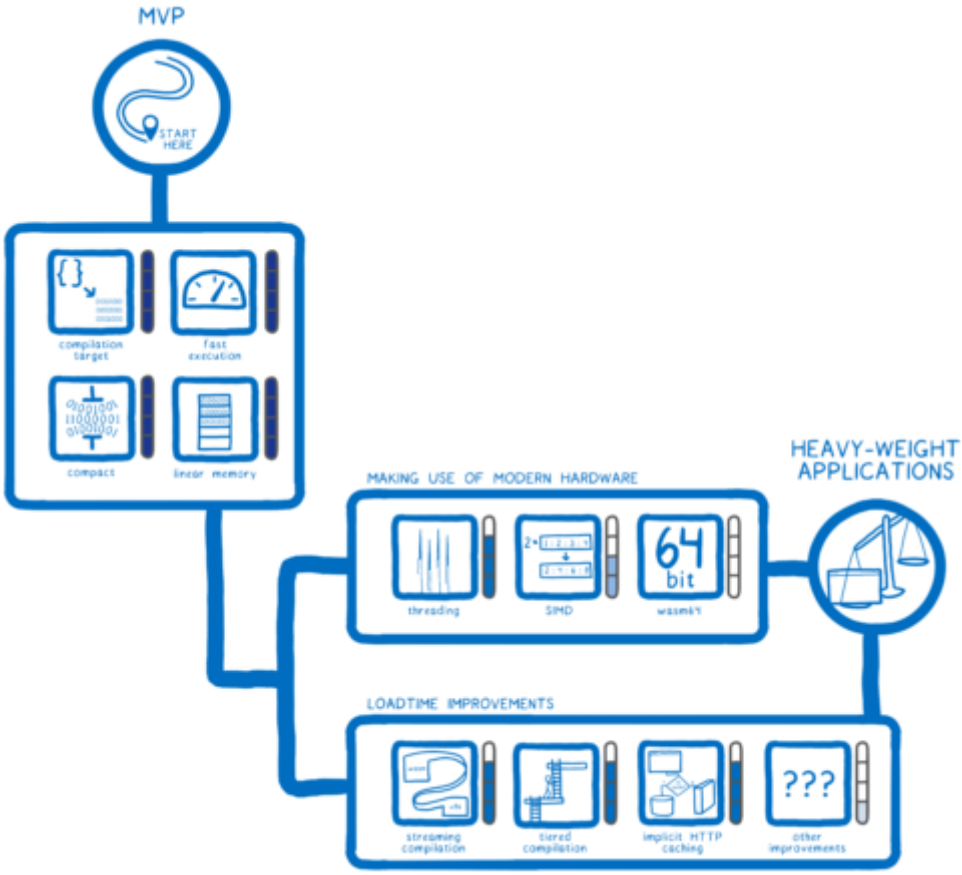
In Firefox, we're getting close to landing support for implicit HTTP caching.

Other improvements

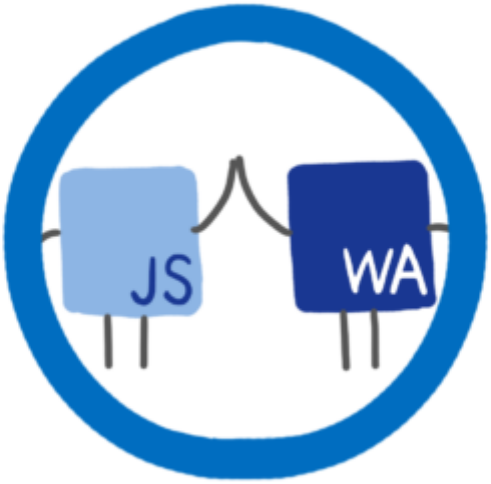
Other improvements are currently in discussion.

Even though this is all still in progress, you already see some of these heavyweight applications coming out today, because WebAssembly already gives these apps the performance that they need.

But once these features are all in place, that's going to be another achievement unlocked, and more of these heavyweight applications will be able to come to the browser.



Small modules interoperating with JavaScript



But WebAssembly isn't just for games and for heavyweight applications. It's also meant for regular web development... for the kind of web development folks are used to: the small modules kind of web development.

Sometimes you have little corners of your app that do a lot of heavy processing, and in some cases, this processing can be faster with WebAssembly. We want to make it easy to port these bits to WebAssembly.

Again, this is a case where some of it's already happening. Developers are already incorporating WebAssembly modules in places where there are tiny modules doing lots of heavy lifting.

One example is the parser in the source map library that's used in Firefox's DevTools and webpack. It was rewritten in Rust, compiled to WebAssembly, which made it 11x faster. And WordPress's Gutenberg parser is on average 86x faster after doing the same kind of rewrite.

But for this kind of use to really be widespread—for people to be really comfortable doing it—we need to have a few more things in place.

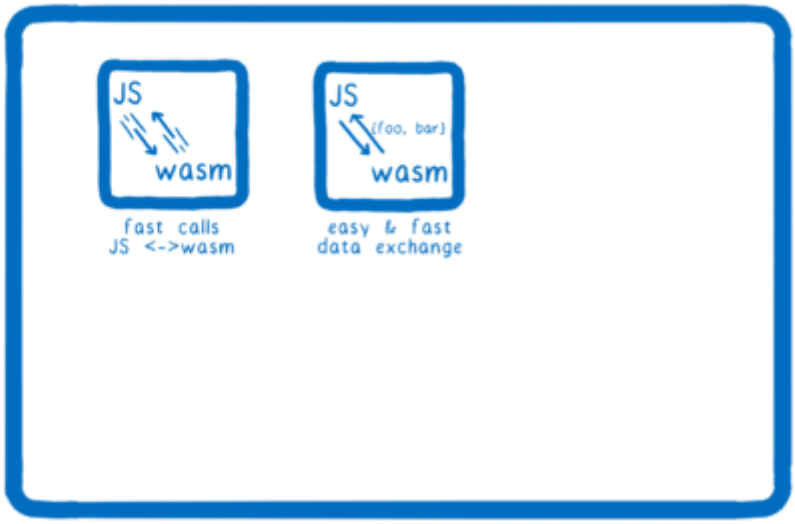
Skill: Fast calls between JS and WebAssembly

First, we need fast calls between JS and WebAssembly, because if you're integrating a small module into an existing JS system, there's a good chance you'll need to call between the two a lot. So you'll need those calls to be fast.

But when WebAssembly first came out, these calls weren't fast. This is where we get back to that whole MVP thing—the engines had the minimum support for calls between the two. They just made the calls work, they didn't make them fast. So engines need to optimize these.

We've recently finished our work on this in Firefox. Now some of these calls are actually faster than non-inlined JavaScript to JavaScript calls. And others engines are also working on this.

Skill: Fast and easy data exchange



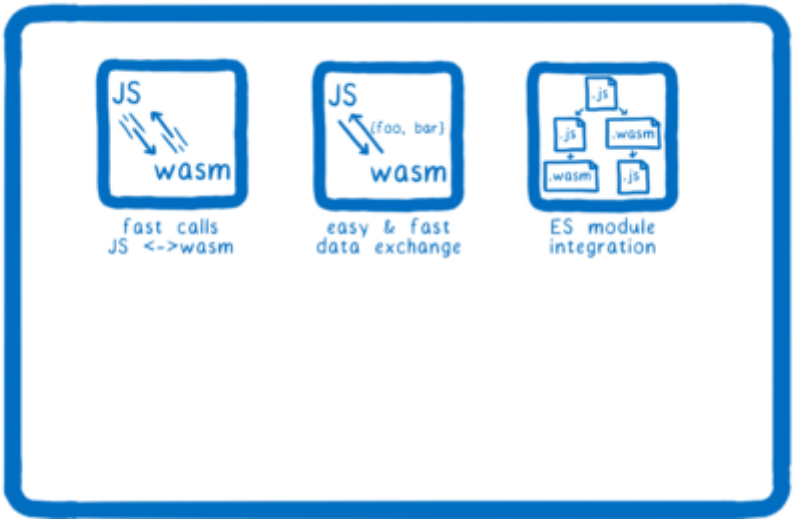
That brings us to another thing, though. When you're calling between JavaScript and WebAssembly, you often need to pass data between them.

You need to pass values into the WebAssembly function or return a value from it. This can also be slow, and it can be difficult too.

There are a couple of reasons it's hard. One is because, at the moment, WebAssembly only understands numbers. This means that you can't pass more complex values, like objects, in as parameters. You need to convert that object into numbers and put it in the linear memory. Then you pass WebAssembly the location in the linear memory.

That's kind of complicated. And it takes some time to convert the data into linear memory. So we need this to be easier and faster.

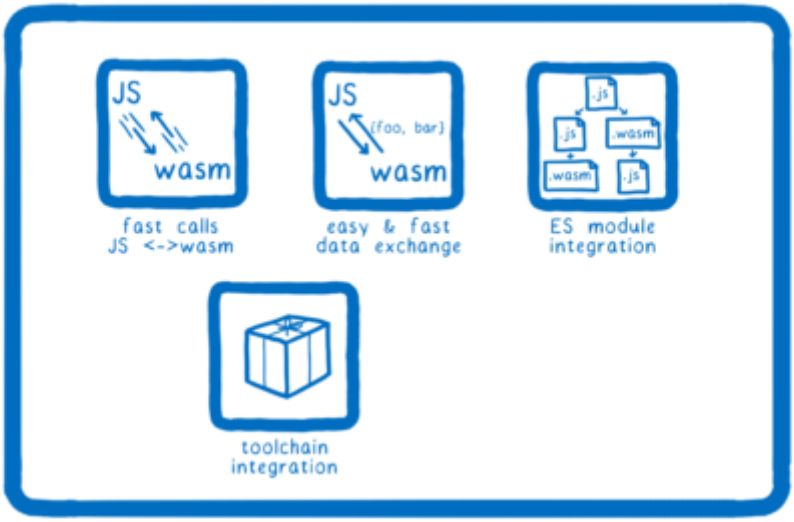
Skill: ES module integration



Another thing we need is integration with the browser's built in ES module support. Right now, you instantiate a WebAssembly module using an imperative API. You call a function and it gives you back a module.

But that means that the WebAssembly module isn't really part of the JS module graph. In order to use import and export like you do with JS modules, you need to have ES module integration.

Skill: Toolchain integration



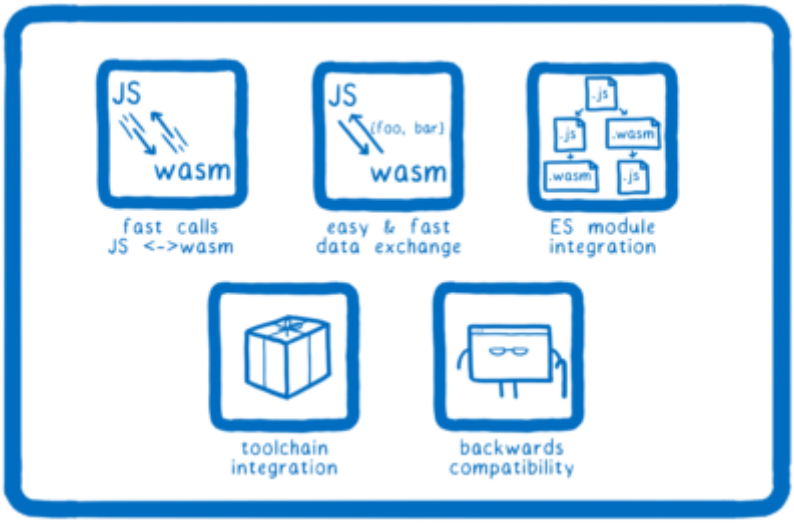
Just being able to import and export doesn't get us all the way there, though. We need a place to distribute these modules, and download them from, and tools to bundle them up.

What's the npm for WebAssembly? Well, what about npm?

And what's the webpack or Parcel for WebAssembly? Well, what about webpack and Parcel?

These modules shouldn't look any different to the people who are using them, so no reason to create a separate ecosystem. We just need tools to integrate with them.

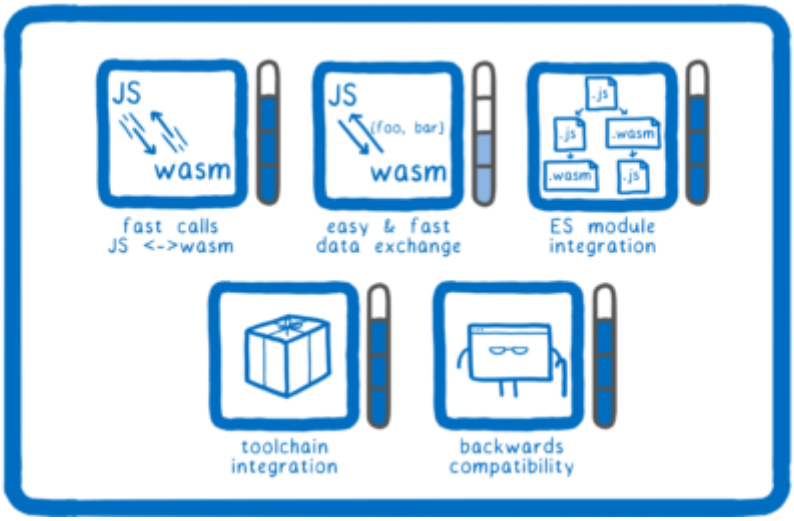
Skill: Backwards compatibility



There's one more thing that we need to really do well in existing JS applications—support older versions of browsers, even those browsers that don't know what WebAssembly is. We need to make sure that you don't have to write a whole second implementation of your module in JavaScript just so that you can support IE11.

Where are we on this?

So where are we on this?



Fast calls between JS and WebAssembly

Calls between JS and WebAssembly are fast in Firefox now, and other browsers are also working on this.

Easy and fast data exchange

For easy and fast data exchange, there are a few proposals that will help with this.

As I mentioned before, one reason you have to use linear memory for more complex kinds of data is because WebAssembly only understands numbers. The only types it has are ints and floats.

With the reference types proposal, this will change. This proposal adds a new type that WebAssembly functions can take as arguments and return. And this type is a reference to an object from outside WebAssembly—for example, a JavaScript object.

But WebAssembly can't operate directly on this object. To actually do things like call a method on it, it will still need to use some JavaScript glue. This means it works, but it's slower than it needs to be.

To speed things up, there's a proposal that we've been calling the Web IDL bindings proposal. It let's a wasm module declare what glue must be applied to its imports and exports, so that the glue doesn't need to be written in JS. By pulling glue from JS into wasm, the glue can be optimized away completely when calling builtin Web APIs.

There's one more part of the interaction that we can make easier. And that has to do with keeping track of how long data needs to stay in memory. If you have some data in linear memory that JS needs access to, then you have to leave it there until the JS reads the data. But if you leave it in there forever, you have what's called a memory leak. How do you know when you can delete the data? How do you know when JS is done with it? Currently, you have to manage this yourself.

Once the JS is done with the data, the JS code has to call something like a free function to free the memory. But this is tedious and error prone. To make this process easier, we're adding WeakRefs to JavaScript. With this, you will be able to observe objects on the JS side. Then you can do cleanup on the WebAssembly side when that object is garbage collected.

So these proposals are all in flight. In the meantime, the Rust ecosystem has created tools that automate this all for you, and that polyfill the proposals that are in flight.

One tool in particular is worth mentioning, because other languages can use it too. It's called wasm-bindgen. When it sees that your Rust code should do something like receive or return certain kinds

of JS values or DOM objects, it will automatically create JavaScript glue code that does this for you, so you don't need to think about it. And because it's written in a language independent way, other language toolchains can adopt it.

ES module integration

For ES module integration, the proposal is pretty far along. We are starting work with the browser vendors to implement it.

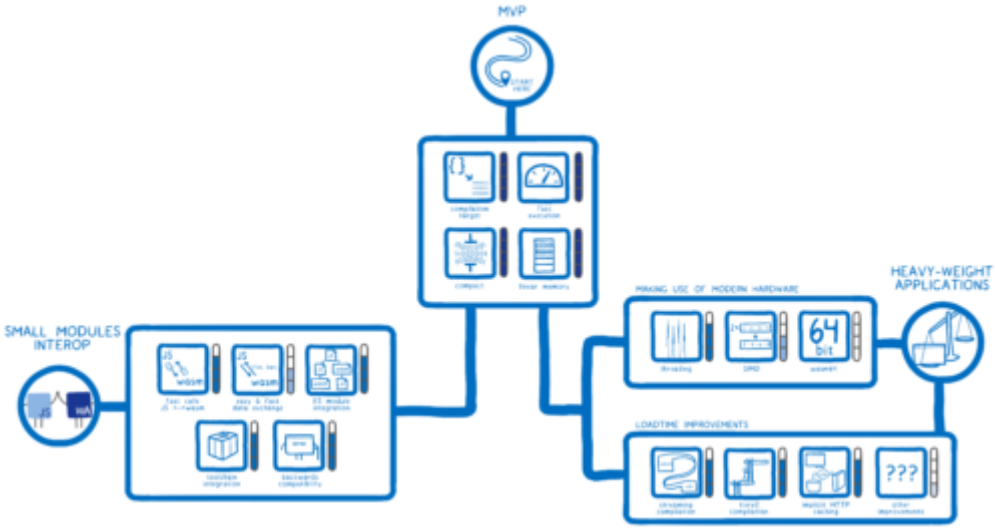
Toolchain support

For toolchain support, there are tools like `wasm-pack` in the Rust ecosystem which automatically runs everything you need to package your code for npm. And the bundlers are also actively working on support.

Backwards compatibility

Finally, for backwards compatibility, there's the `wasm2js` tool. That takes a wasm file and spits out the equivalent JS. That JS isn't going to be fast, but at least that means it will work in older versions of browsers that don't understand WebAssembly.

So we're getting close to unlocking this achievement. And once we unlock it, we open the path to another two.



JS frameworks and compile-to-JS languages

One is rewriting large parts of things like JavaScript frameworks in WebAssembly.

The other is making it possible for statically-typed compile-to-js languages to compile to WebAssembly instead—for example, having languages like Scala.js, or Reason, or Elm compile to WebAssembly.

For both of these use cases, WebAssembly needs to support high-level language features.

Skill: GC

We need integration with the browser's garbage collector for a couple of reasons.

First, let's look at rewriting parts of JS frameworks. This could be good for a couple of reasons. For example, in React, one thing you could do is rewrite the DOM diffing algorithm in Rust, which has very

ergonomic multithreading support, and parallelize that algorithm.

You could also speed things up by allocating memory differently. In the virtual DOM, instead of creating a bunch of objects that need to be garbage collected, you could used a special memory allocation scheme. For example, you could use a bump allocator scheme which has extremely cheap allocation and all-at-once deallocation. That could potentially help speed things up and reduce memory usage.

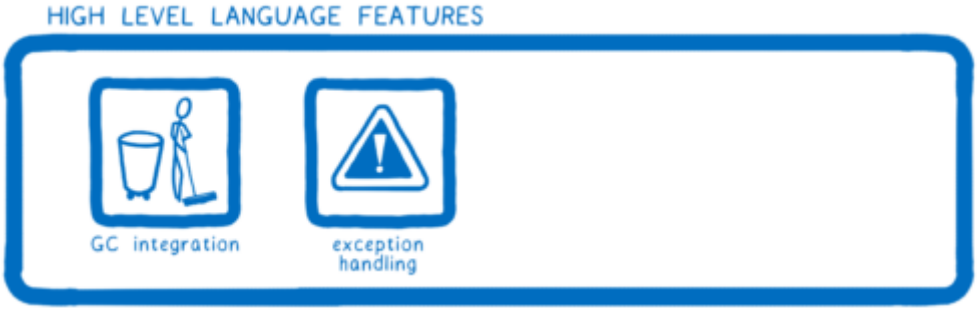
But you'd still need to interact with JS objects—things like components—from that code. You can't just continually copy everything in and out of linear memory, because that would be difficult and inefficient.

So you need to be able to integrate with the browser's GC so you can work with components that are managed by the JavaScript VM. Some of these JS objects need to point to data in linear memory, and sometimes the data in linear memory will need to point out to JS objects.

If this ends up creating cycles, it can mean trouble for the garbage collector. It means the garbage collector won't be able to tell if the objects are used anymore, so they will never be collected. WebAssembly needs integration with the GC to make sure these kinds of cross-language data dependencies work.

This will also help statically-typed languages that compile to JS, like Scala.js, Reason, Kotlin or Elm. These language use JavaScript's garbage collector when they compile to JS. Because WebAssembly can use that same GC—the one that's built into the engine—these languages will be able to compile to WebAssembly instead and use that same garbage collector. They won't need to change how GC works for them.

Skill: Exception handling



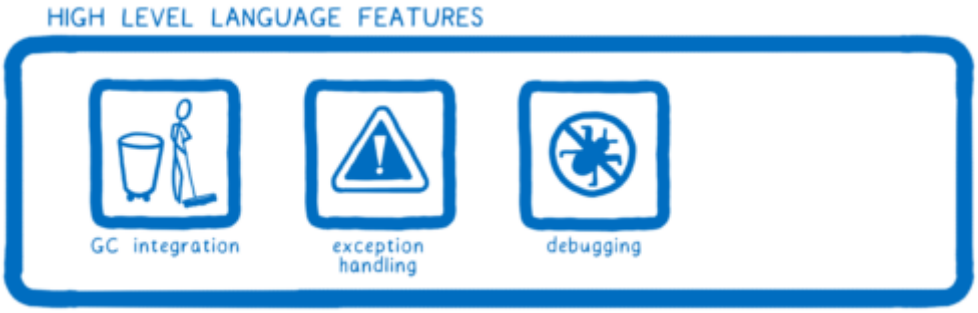
We also need better support for handling exceptions.

Some languages, like Rust, do without exceptions. But in other languages, like C++, JS or C#, exception handling is sometimes used extensively.

You can polyfill exception handling currently, but the polyfill makes the code run really slowly. So the default when compiling to WebAssembly is currently to compile without exception handling.

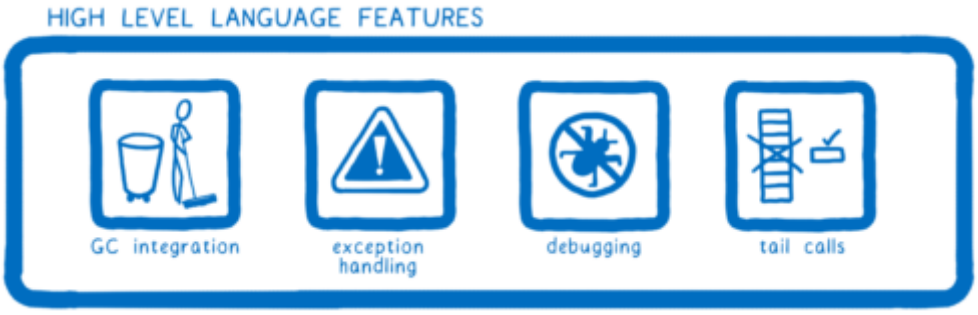
However, since JavaScript has exceptions, even if you've compiled your code to not use them, JS may throw one into the works. If your WebAssembly function calls a JS function that throws, then the WebAssembly module won't be able to correctly handle the exception. So languages like Rust choose to abort in this case. We need to make this work better.

Skill: Debugging



Another thing that people working with JS and compile-to-JS languages are used to having is good debugging support. Devtools in all of the major browsers make it easy to step through JS. We need this same level of support for debugging WebAssembly in browsers.

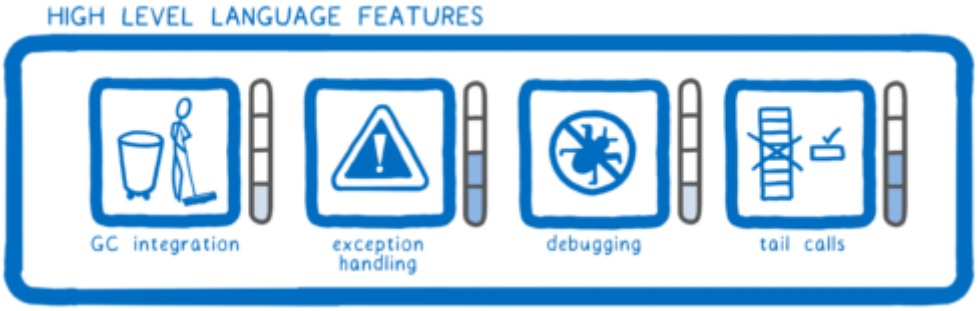
Skill: Tail calls



And finally, for many functional languages, you need to have support for something called tail calls. I'm not going to get too into the details on this, but basically it lets you call a new function without adding a new stack frame to the stack. So for functional languages that support this, we want WebAssembly to support it too.

Where are we on this?

So where are we on this?



Garbage collection

For garbage collection, there are two proposals currently underway:

The Typed Objects proposal for JS, and the GC proposal for WebAssembly. Typed Objects will make it possible to describe an object's fixed structure. There is an explainer for this, and the proposal will be discussed at an upcoming TC39 meeting.

The WebAssembly GC proposal will make it possible to directly access that structure. This proposal is under active development.

With both of these in place, both JS and WebAssembly know what an object looks like and can share that object and efficiently access the data stored on it. Our team actually already has a prototype of this working. However, it still will take some time for these to go through standardization so we're probably looking at sometime next year.

Exception handling

Exception handling is still in the research and development phase, and there's work now to see if it can take advantage of other proposals like the reference types proposal I mentioned before.

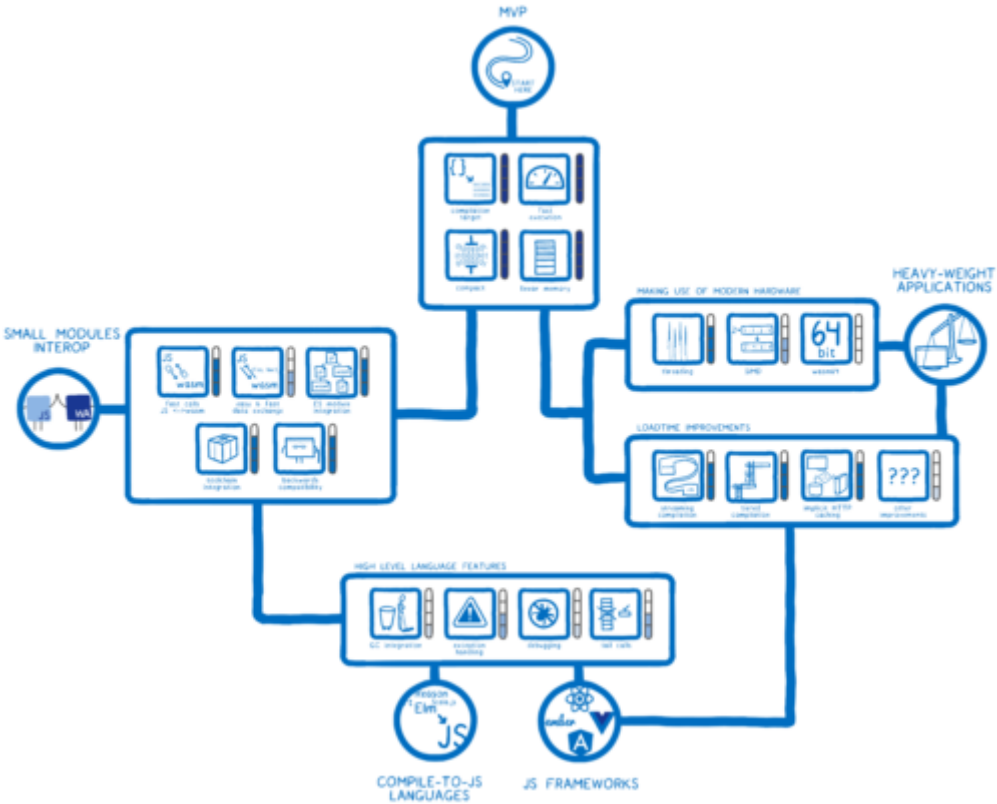
Debugging

For debugging, there is currently some support in browser devtools. For example, you can step through the text format of WebAssembly in Firefox debugger.But it's still not ideal. We want to be able to show you where you are in your actual source code, not in the assembly. The thing that we need to do for that is figure out how source maps—or a source maps type thing—work for WebAssembly. So there's a subgroup of the WebAssembly CG working on specifying that.

Tail calls

The tail calls proposal is also underway.

Once those are all in place, we'll have unlocked JS frameworks and many compile-to-JS languages.



So, those are all achievements we can unlock inside the browser. But what about outside the browser?

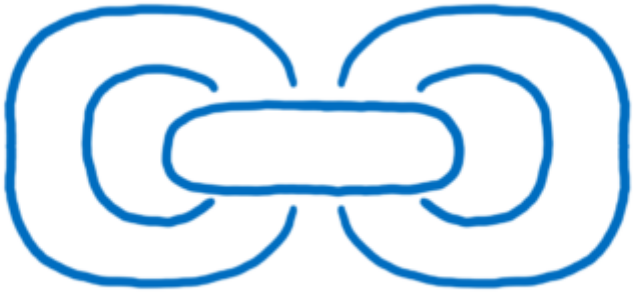
Outside the Browser

Now, you may be confused when I talk about "outside the browser". Because isn't the browser what you use to view the web? And isn't that right in the name—WebAssembly.

But the truth is the things you see in the browser—the HTML and CSS and JavaScript—are only part of what makes the web. They are the visible part—they are what you use to create a user interface—so they are the most obvious.

But there's another really important part of the web which has properties that aren't as visible.

That is the link. And it is a very special kind of link.



The innovation of this link is that I can link to your page without having to put it in a central registry, and without having to ask you or even know who you are. I can just put that link there.

It's this ease of linking, without any oversight or approval bottlenecks, that enabled our web. That's what enabled us to form these global communities with people we didn't know.

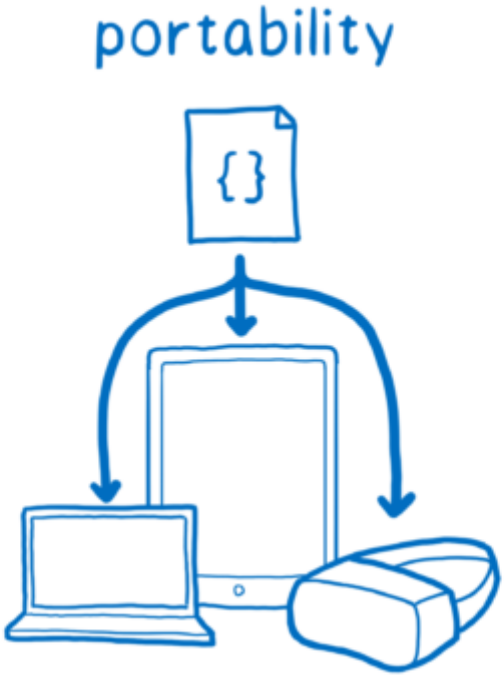
But if all we have is the link, there are two problems here that we haven't addressed.

The first one is... you go visit this site and it delivers some code to you. How does it know what kind of code it should deliver to you? Because if you're running on a Mac, then you need different machine code than you do on Windows. That's why you have different versions of programs for different operating systems.

Then should a web site have a different version of the code for every possible device? No.

Instead, the site has one version of the code—the source code. This is what's delivered to the user. Then it gets translated to machine code on the user's device.

The name for this concept is portability.



So that's great, you can load code from people who don't know you and don't know what kind of device you're running.

But that brings us to a second problem. If you don't know these people who's web pages you're loading, how do you know what kind of code they're giving you? It could be malicious code. It could be trying to take over your system.

Doesn't this vision of the web—running code from anybody who's link you follow—mean that you have to blindly trust anyone who's on the web?

This is where the other key concept from the web comes in.

That's the security model. I'm going to call it the sandbox.

Basically, the browser takes the page—that other person's code—and instead of letting it run around willy-nilly in your system, it puts it in a sandbox. It puts a couple of toys that aren't dangerous into that sandbox so that the code can do some things, but it leaves the dangerous things outside of the sandbox.

So the utility of the link is based on these two things:

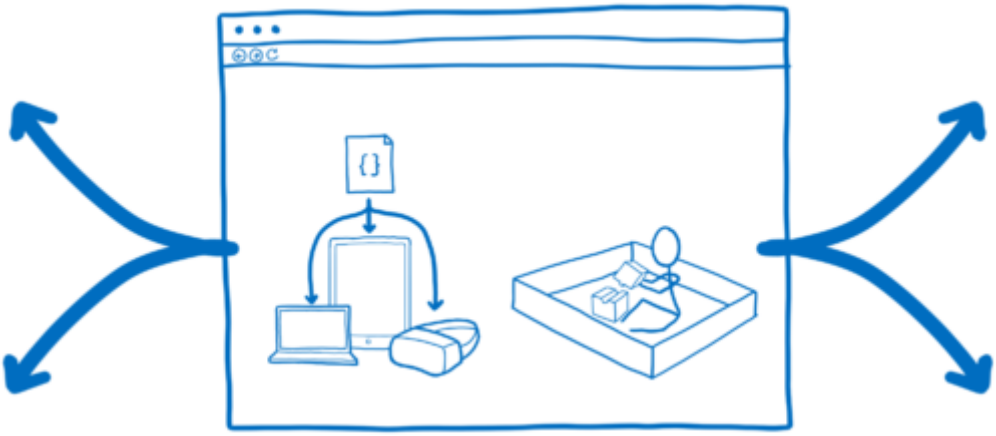
- Portability—the ability to deliver code to users and have it run on any type of device that can run a browser.
- And the sandbox—the security model that lets you run that code without risking the integrity of your machine.

So why does this distinction matter? Why does it make a difference if we think of the web as something that the browser shows us using HTML, CSS, and JS, or if we think of the web in terms of portability and the sandbox?

Because it changes how you think about WebAssembly.

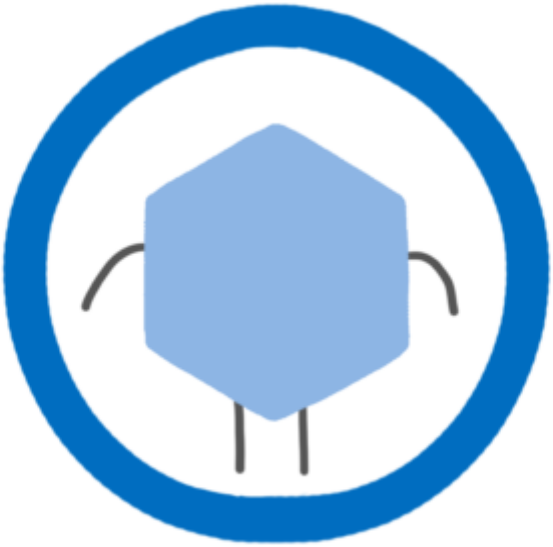
You can think about WebAssembly as just another tool in the browser's toolbox... which it is.

It is another tool in the browser's toolbox. But it's not just that. It also gives us a way to take these other two capabilities of the web—the portability and the security model—and take them to other use cases that need them too.



We can expand the web past the boundaries of the browser. Now let's look at where these attributes of the web would be useful.

Node.js



How could WebAssembly help Node? It could bring full portability to Node.

Node gives you most of the portability of JavaScript on the web. But there are lots of cases where Node's JS modules aren't quite enough—where you need to improve performance or reuse existing code that's not written in JS.

In these cases, you need Node's native modules. These modules are written in languages like C, and they need to be compiled for the specific kind of machine that the user is running on.

Native modules are either compiled when the user installs, or precompiled into binaries for a wide matrix of different systems. One of these approaches is a pain for the user, the other is a pain for the package maintainer.

Now, if these native modules were written in WebAssembly instead, then they wouldn't need to be compiled specifically for the target architecture. Instead, they'd just run like the JavaScript in Node runs. But they'd do it at nearly native performance.

So we get to full portability for the code running in Node. You could take the exact same Node app and run it across all different kinds of devices without having to compile anything.

But WebAssembly doesn't have direct access to the system's resources. Native modules in Node aren't sandboxed—they have full access to all of the dangerous toys that the browser keeps out of the sandbox. In Node, JS modules also have access to these dangerous toys because Node makes them available. For example, Node provides methods for reading from and writing files to the system.

For Node's use case, it makes a certain amount of sense for modules to have this kind access to dangerous system APIs. So if WebAssembly modules don't have that kind of access by default (like Node's current modules do), how could we give WebAssembly modules the access they need? We'd need to pass in functions so that the WebAssembly module can work with the operating system, just as Node does with JS.

For Node, this will probably include a lot of the functionality that's in things like the C standard library. It would also likely include things that are part of POSIX—the Portable Operating System Interface—which is an older standard that helps with compatibility. It provides one API for interacting with the system across a bunch of different Unix-like OSs. Modules would definitely need a bunch of POSIX-like functions.

Skill: Portable interface

What the Node core folks would need to do is figure out the set of functions to expose and the API to use.

But wouldn't it be nice if that were actually something standard? Not something that was specific to just Node, but could be used across other runtimes and use cases too?

A POSIX for WebAssembly if you will. A PWSIX? A portable WebAssembly system interface.

And if that were done in the right way, you could even implement the same API for the web. These standard APIs could be polyfilled onto existing Web APIs.

These functions wouldn't be part of the WebAssembly spec. And there would be WebAssembly hosts that wouldn't have them available. But for those platforms that could make use of them, there would be a unified API for calling these functions, no matter which platform the code was running on. And this would make universal modules—ones that run across both the web and Node—so much easier.

Where are we with this?

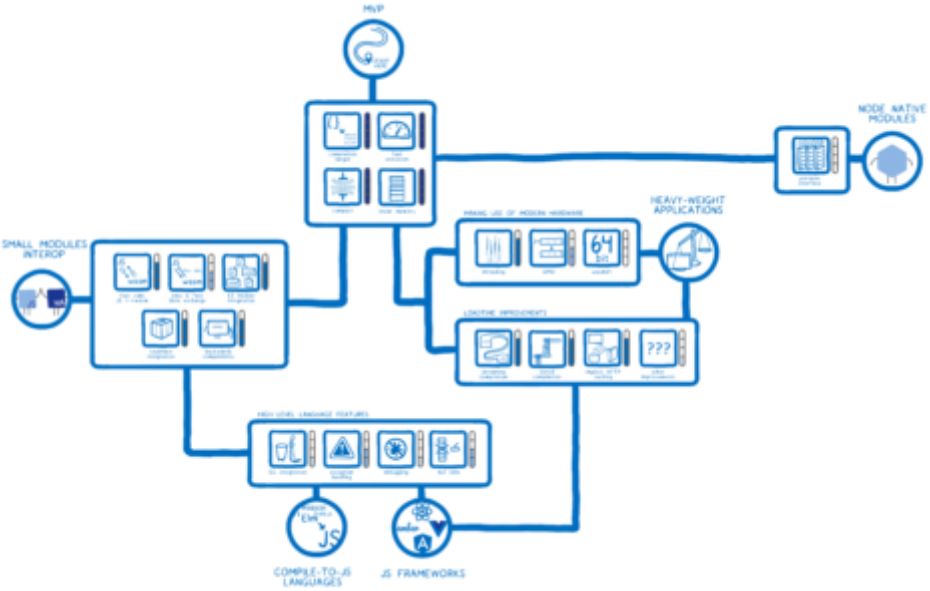
So, is this something that could actually happen?

A few things are working in this idea's favor. There's a proposal called package name maps that will provide a mechanism for mapping a module name to a path to load the module from. And that will likely be supported by both browsers and Node, which can use it to provide different paths, and thus load entirely different modules, but with the same API. This way, the .wasm module itself can specify a single (module-name, function-name) import pair that Just Works on different environments, even the web.

With that mechanism in place, what's left to do is actually figure out what functions make sense and what their interface should be.

There's no active work on this at the moment. But a lot of discussions are heading in this direction right now. And it looks likely to happen, in one form or another.

Which is good, because unlocking this gets us halfway to unlocking some other use cases outside the browser. And with this in place, we can accelerate the pace.



So, what are some examples of these other use cases?

CDNs, Serverless, and Edge Computing

One example is things like CDNs, and Serverless, and Edge Computing. These are cases where you're putting your code on someone else's server, and they make sure that the server is maintained and that the code is close to all of your users.

Why would you want to use WebAssembly in these cases? There was a great talk explaining exactly this at a conference recently.

Fastly is a company that provides CDNs and edge computing. And their CTO, Tyler McMullen, explained it this way (and I'm paraphrasing here):

If you look at how a process works, code in that process doesn't have boundaries. Functions have access to whatever memory in that process they want, and they can call whatever other functions they want.

When you're running a bunch of different people's services in the same process, this is an issue. Sandboxing could be a way to get around this. But then you get to a scale problem.

For example, if you use a JavaScript VM like Firefox's SpiderMonkey or Chrome's V8, you get a sandbox and you can put hundreds of instances into a process. But with the numbers of requests that Fastly is servicing, you don't just need hundreds per process—you need tens of thousands.

Tyler does a better job of explaining all of it in his talk, so you should go watch that. But the point is that WebAssembly gives Fastly the safety, speed, and scale needed for this use case.

So what did they need to make this work?

Skill: Runtime

They needed to create their own runtime. That means taking a WebAssembly compiler—something that can compile WebAssembly down to machine code—and combining it with the functions for interacting with the system that I mentioned before.

For the WebAssembly compiler, Fastly used Cranelift, the compiler that we're also building into Firefox. It's designed to be very fast and doesn't use much memory.

Now, for the functions that interact with the rest of the system, they had to create their own, because we don't have that portable interface available yet.

So it's possible to create your own runtime today, but it takes some effort. And it's effort that will have to be duplicated across different companies.

What if we didn't just have the portable interface, but we also had a common runtime that could be used across all of these companies and other use cases? That would definitely speed up development.

Then other companies could just use that runtime—like they do Node today—instead of creating their own from scratch.

Where are we on this?

So what's the status of this?

Even though there's no standard runtime yet, there are a few runtime projects in flight right now. These include WAVM, which is built on top of LLVM, and wasmjit.

In addition, we're planning a runtime that's built on top of Cranelift, called wasmtime.

And once we have a common runtime, that speeds up development for a bunch of different use cases. For example...

Portable CLI tools

WebAssembly can also be used in more traditional operating systems. Now to be clear, I'm not talking about in the kernel (although brave souls are trying that, too) but WebAssembly running in Ring 3—in user mode.

Then you could do things like have portable CLI tools that could be used across all different kinds of operating systems.

And this is pretty close to another use case...

Internet of Things

The internet of things includes devices like wearable technology, and smart home appliances.

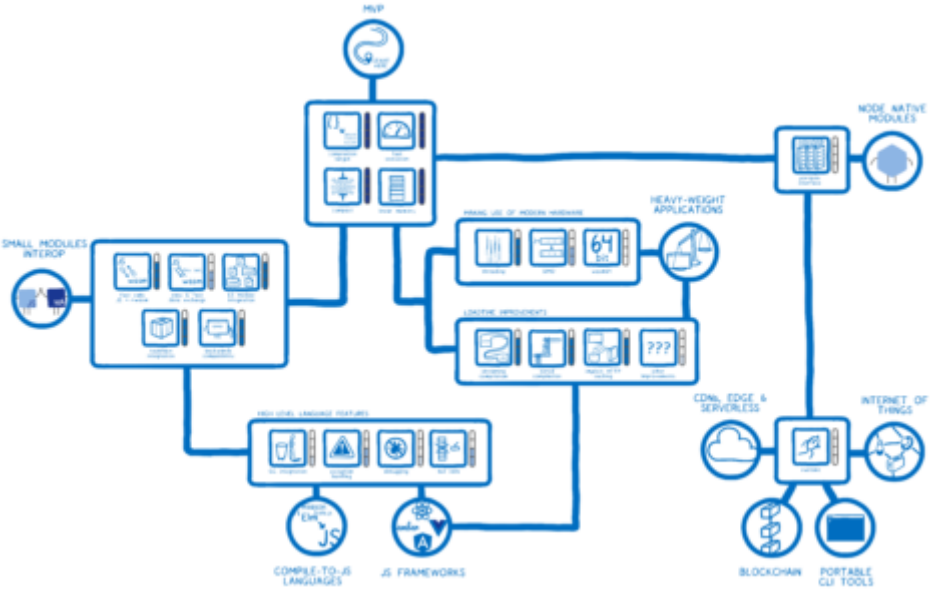
These devices are usually resource constrained—they don't pack much computing power and they don't have much memory. And this is exactly the kind of situation where a compiler like Cranelift and a runtime like wasmtime would shine, because they would be efficient and low-memory. And in the extremely-resource-constrained case, WebAssembly makes it possible to fully compile to machine code before loading the application on the device.

There's also the fact that there are so many of these different devices, and they are all slightly different. WebAssembly's portability would really help with that.

So that's one more place where WebAssembly has a future.

Conclusion

Now let's zoom back out and look at this skill tree.



I said at the beginning of this post that people have a misconception about WebAssembly—this idea that the WebAssembly that landed in the MVP was the final version of WebAssembly.

I think you can see now why this is a misconception.

Yes, the MVP opened up a lot of opportunities. It made it possible to bring a lot of desktop applications to the web. But we still have many use cases to unlock, from heavy-weight desktop applications, to small modules, to JS frameworks, to all the things outside the browser… Node.js, and serverless, and the blockchain, and portable CLI tools, and the internet of things.

So the WebAssembly that we have today is not the end of this story—it's just the beginning.

About Lin Clark

Lin works in Advanced Development at Mozilla, with a focus on Rust and WebAssembly.

https://twitter.com/linclark
@linclark

More articles by Lin Clark…

About Till Schneidereit

Till works on developer technologies in Mozilla Research, and on JavaScript and WebAssembly standardization in TC39 and the WebAssembly CG.

More articles by Till Schneidereit…

About Luke Wagner

Luke Wagner is a Mozilla software engineer and hacks on JavaScript and WebAssembly in Firefox.

More articles by Luke Wagner…

29 comments

Ralf

Thanks for this article, and as usual great job in particular on the cartoons. :)

The link to WAVM seems wrong though, it goes to a radio station. (Heh, talking about being able to link to anything… ;)