EPISODE 855

[INTRODUCTION]

[00:00:00] JM: WebAssembly allows for web-based execution of languages other than JavaScript. Programs written in Rust or C++ can be compiled down to WebAssembly and shipped over the browser for on-the-fly execution in a safe memory controlled environment.

WebAssembly has been in development for more than two years and it's still an immature ecosystem because building the necessary tooling for WebAssembly is hard. That said, it's very interesting and we'll be doing many more WebAssembly shows in the near future.

Much of the web has been built around JavaScript and the V8 JavaScript engine, which has been tuned to optimize an interpreted language, which is JavaScript. WebAssembly modules are often written in C++ or Rust, which are compiled languages. There are engineering challenges at the edges between this interpreted JavaScript runtime and the precompiled WebAssembly modules. That's definitely a reductive presentation of the difficulties in getting WebAssembly's ecosystem off the ground. There are many more interesting facets and we'll go into those today. Till Schneidereit is a senior research engineering manager at Mozilla and he joins the show to discuss the compilation path of WebAssembly and the state of the ecosysotem.

We have a new Software Daily app for iOS. To become an ad-free listener and support the show, you can pay us with $10 a month or $100 a year and listen through the app or listen on softwaredaily.com. The app has been in store for almost two years I think at this point and we've been iterating on it at this point. It's got a lot of polish. So, the new version of the app is pretty polished and it does a lot for you even if you're not paying $10 a month or $100 a year. You can download episodes. You can search through all of our back catalogue. You can look at categories. You can connect with other people in the community and discuss software engineering topics. I'd love to know what you think, what you'd like to get out of this community-based platform for listening to our podcasts and reading our supporting material. You can go softwaredaily.com. You can check out the app in the iTunes App Store or just look in the show notes.

We’re booking sponsorships for 2019. If you’re interested in sponsoring the show and airing some ads or working with us on some content, you can go to softwareengineeringdaily.com/sponsor, and we’re hiring two interns for software engineering and business development. If you’re interested in either of these positions, you can send me an email with your resume to jeff@softwareengineeringdaily.com with internship in the subject line. Let’s get on to today’s show.

[INTERVIEW]

[00:03:05] JM: When I’m building a new product, G2i is the company that I call on to help me find a developer who can build the first version of my product. G2i is a hiring platform run by engineers that matches you with React, React Native, GraphQL and mobile engineers who you can trust. Whether you are a new company building your first product, like me, or an established company that wants additional engineering help, G2i has the talent that you need to accomplish your goals.

Go to softwareengineeringdaily.com/g2i to learn more about what G2i has to offer. We’ve also done several shows with the people who run G2i, Gabe Greenberg, and the rest of his team. These are engineers who know about the React ecosystem, about the mobile ecosystem, about GraphQL, React Native. They know their stuff and they run a great organization.

In my personal experience, G2i has linked me up with experienced engineers that can fit my budget, and the G2i staff are friendly and easy to work with. They know how product development works. They can help you find the perfect engineer for your stack, and you can go to softwareengineeringdaily.com/g2i to learn more about G2i.

Thank you to G2i for being a great supporter of Software Engineering Daily both as listeners and also as people who have contributed code that have helped me out in my projects. So if you want to get some additional help for your engineering projects, go to softwareengineeringdaily.com/g2i.

[INTERVIEW]

[00:04:56] JM: Till Schneidereit, you work at Mozilla. Thanks for coming on Software Engineering Daily.

[00:05:00] TS: Thank you for having me.

[00:05:01] JM: I want to talk to you about WebAssembly in a variety of contexts. The project that eventually became WebAssembly was designed to allow C and C++ applications to run in the browser. Describe the first version of WebAssembly.

[00:05:21] TS: So, you're saying the project that eventually became WebAssembly. So, really, are you asking about what really the first version of WebAssembly or [inaudible 00:05:32].

[00:05:32] JM: So, I believe there were some primordial soup that eventually turned into WebAssembly. It was things like mscript.in, and there were some other things. I just want to understand the early primitives that were developed that were just to run C and C++ in the browser before we get to talking about the modern context.

[00:05:53] TS: Okay. That makes sense. Yeah. So, well before WebAssembly, there was asm.js. I'll explain what that was in a moment or is. But even before asm.js, there was just normal JavaScript, and an engineer at Mozilla identified the possibility of compiling C and C++ and other languages compiling using the LLVM compiler tool chain to JavaScript by essentially – All these languages have in common that they use pointers, and JavaScript doesn't have pointers.

So, instead, what this tool called mscript.in did was to represent pointers as entries in a way, and this was just a normal JavaScript way. Really, just a normal way where you could store everything in it, but the difference here was that instead of storing any arbitrary things like JavaScript objects in it. It stored only numbers. By that, it treated as if it were memory, linear memory that C and C++ applications can understand.

Then there were few other things that we needed to sort of special case to really only use fairly small parts of JavaScript to make this work. Essentially, mostly math operations. Then awhile later, I think 2013 about, another engineer at Mozilla, Luke Wagner, identified that this subset of JavaScript, if the compiler, if the JavaScript engine could identify this subset beforehand, before running code, then it could highly optimize that code.

So, they built this prototype really quickly and then eventually we have an amazingly short amount of time, really full implementation, that made it so that C and C++ code could run with something like 50% or so overhead over native in a browser unmodified, or in an unmodified browser. In Firefox where we had this implementation, it went even faster than in other browser, because we had these specific optimizations, and that's what's called asm.js.

[00:08:18] JM: Can you describe the initial compilation path for a C or C++ module to run in the browser? Describe that initial compilation path in a little more detail.

[00:08:31] TS: Essentially took the LLVM compiler tool chain, which can compile by now a lot of different language through first and internal bytecode, LLVM bit code. Then another part of the compiler tool chain, the backend, takes that code and compiles it for different CPU architectures. Essentially, what mscript.in does is instead of going through the normal backends for CPUs, such as Intel or AMD, x86 CPUs or ARM CPUs, it implemented its own backend for targeting the subset of JavaScript that I described. Really, it uses math operations and indexing into this way the same way that normal backends use the built-in instructions a CPU has that assembly represents.

Then it applied a I lot of different optimizations and small tweaks essentially to the code, because JavaScript turns out isn't actually a CPU and it behaves fairly differently in some ways. For example, it doesn't allow you to just really jump in code. You can't say, "After this instruction, execute this other instruction that is entirely somewhere else in a code," which is a go-to statement allows you to do that. JavaScript doesn't have it. So you can only use normal control flow structures such as while or if and the instructions in the LLVM bit code have to be converted into something that you can express in this. That was one of the main ingredients of mscript.in, the so-called relooper, which identifies control flow that is in go-to statements and changes it into while loops and if statements.

[00:10:34] JM: What were the major lessons that came out of those early experiments?

[00:10:40] TS: One was asm.js, where really the realization was if the engine knows that only the subset is used, we can really optimize it. Then building on that, another realization was JavaScript wasn't the best vehicle for this. To really get the overhead down to get to fastest startup and fastest runtime performance, something else was needed, and that was why WebAssembly eventually happened.

I think one of the biggest achievements there was to get everything on the table – At the table, I guess, to get all the different vendors working on browsers to sit down and work together on identifying what is the best way to represent these applications in a way that integrates really well into JavaScript engines. That's what WebAssembly resulted in.

[00:11:45] JM: How will WebAssembly change the browser experience?

[00:11:50] TS: It will make much more code reuse possible not only of large existing applications, such as AutoCAD, but also of existing libraries, where it used to be that you could use the same code in an application compiled for Windows, Mac OS, Linux, but also Android and iOS, but not on the web. On the web, you had to implement the same functionality in JavaScript. With WebAssembly, you can reuse the same code. So while mainly your application might still be in JavaScript, parts that you want to reuse would be in WebAssembly. That in itself doesn't immediately change the experience, but I think it'll lead to which are applications, because there's just more budget available if you don't have to do a second implementation.

Then there's the speed, of course. There are some things where JavaScript is just not the right language to use to implement some features in particular when it comes to processing large amounts of data and being able to use other languages that are better suited for that, like C++, Rust, will mean that applications on the web can in general become much richer experiences.

[00:13:21] JM: The first thing you mentioned was code reuse. Describe in more detail what you mean by code reuse.

[00:13:29] TS: In the development of native applications, it's fairly normal to have a portable sort of backend almost of your application. Portable core backend I guess is confusing. A portable core of your application that is often written in C++ and that you can compile to all these different desktop and mobile platforms. Then you have frontends, user interfaces that are implemented specifically for each of these platforms and often in other languages. For Android, it would be Java or Kotlin. For iOS, it would be objective C or Swift, and so on. That is all code that you wouldn't reuse. It's also code you often wouldn't want to reuse, because you want to have a UI that really fits well into the target system.

But this core of your application, the business logic if you will, that can be reused across all of these applications. Now, with WebAssembly, we can make that possible for the web also, where the user interface is implemented in a way that is native to the web using JavaScript, HTML and CSS, and the business object can be reused exactly the same way as on these other platforms.

[00:14:56] JM: After the experiment of getting C and C++ code to work in the browser, at least work to some extent. The scope of possibilities for what that early project was, whether you're talking about – I guess asm.js is maybe what you would want to call that early project, that early set of projects. But the potential was massive and it was clear. With all of that potential opportunity for getting this cross-platform different language system for running in the web, how did the people working on WebAssembly decide what to do next? I mean, what happened after that experimental process? Did they start to flesh out a plan or start to flesh out a spec for what this project should encompass?

[00:15:54] TS: Eventually, that is what happened. But before getting to this place where an official plan and then eventually a spec was developed, there were really a lot of informal conversations. WebAssembly really started with conversations between key engineers working on all the major browsers and it wasn't only asm.js that had entered the design. It was also on the Google side. There was the NCL team, native client, which was Google's proposal for how to bring native code into the web in a safe sandboxed way.

The team working on that and our people working on asm.js and key engineers from the other browser vendors worked together on identifying what is the smallest piece we can build to make this really be viable. What is the minimum viable product for bringing significant value to the web by bringing these other languages to it? Why is WebAssembly the MVP? Then they started to formalize that to write up detailed plans and eventually a spec.

[00:17:26] JM: As we start to talk about the spec for WebAssembly, we need to explore some different areas. We need to explore compilation and package management and the interface between a WebAssembly module that you're getting from the internet and your lower-level kernel resources. There's a ton of ground to cover here. Speaking broadly, how should WebAssembly interface with the underlying host operating system?

[00:18:03] TS: That, of course, is one of the key questions that we are encountering now. Really, the important development here is that WebAssembly is now reaching outside the browser. I think a lot of people don't know this, but the people originally designing WebAssembly never meant it to only be used on the web, and it is in fact carefully designed not to rely on anything that only a web browser would provide. It doesn't rely on JavaScript running next to it, for example.

We are now taking very first big steps towards standardizing what this will look like. What WebAssembly use outside of the browser will look like? In fact, just about six week ago, released a blog post. Lin Clark wrote a post announcing the WASI system interface for WebAssembly. So, WASI stands for WebAssembly system interface. That started a standardization process for what exactly this looks like. What does it look like for WebAssembly to not talk to JavaScript in the browser, but talk to the operating system and ultimately the kernel it is running on?

[SPONSOR MESSAGE]

[00:19:43] JM: Software Engineering Daily is a media company, and we run on WordPress just like lots of other media companies, although it's not just media companies that run on WordPress. I know of many organization that manage multiple WordPress sites and it can be hard to manage all of these sites efficiently.

Pantheon is a platform for hosting and managing your WordPress and Drupal sites. Pantheon makes it easier to build, manage and optimize your websites. Go to pantheon.io/sedaily to see how you can use Pantheon. Pantheon makes it easier to manage your WordPress and Drupal websites with scalable infrastructure, a fast CDN and security features, such as disaster recovery.

Pantheon gives you automated workflows for managing dev, test and production deployments, and Pantheon provides easy integrations with GitHub, CircleCI, Jira and more. If you have a WordPress or a Drupal website, check out pantheon.io/sedaily.

Thanks to Pantheon for being a sponsor of Software Engineering Daily.

[INTERVIEW CONTINUED]

[00:21:05] JM: So, I request code over the internet all the time. I am just browsing on the internet and I click to a new page and there's going to be code that loads on to my browser. If we compare the request path that code is usually taking in my browser today from coming over the network to running on my operating system, how does that compare to what we will have with a mature WebAssembly world?

[00:21:39] TS: I'm not sure if there is a single simple answer to this, because WebAssembly will be used in a lot of different places. Often times, it will not actually run on your machine, but it might run on your behalf on other machines. It could run on an edge server like it does in the Fastly CDN who recently started supporting running WebAssembly on their edge servers, where it allows developers of internet services to provide richer experiences for you by making these edge requests more dynamic, or it might run in an IoT device, where it might also provide richer interactions with a device. But it might also run on your machine and be, for example, a command line interface tool, or eventually a full desktop application.

All of these different paths and quite a few more that will eventually emerge all really have different answers to your question. I think for the command line interface, it's perhaps easiest to immediately answer that. So, we believe that one thing is really important that we keep the security guarantees that WebAssembly has alive while bringing it outside the browser. That involves not giving these applications running in WebAssembly just full access to your operating system.

I'm sure you've heard about these issues with, for example, modules on npmjs.com being modified in malicious ways and stealing people's Bitcoin wallets. These kinds of things are very hard to secure against if every application by default just has access to all of your files. So, the WASI system interface is designed in a way where a module or an application written for WASI has to announce beforehand, "I need access to these resources and nothing more." Then the runtime enforces that it only gets access to exactly these things and cannot possibly access anything else.

So, if something says, "I need to be able to do something to file in this specific directory and write into that directory, but nothing else." Then you can safely execute it as long as you know whatever that application will do in that directory. That is fine. But it can't go to somewhere else on your hard drive and still your Bitcoin wallet.

So, the process of downloading things is – There are very interesting questions to answer here, and we believe package management is an important part of this story indeed. But, ultimately, you shouldn't need to worry too much about where something is coming from, because you should be able to just say, "It's fine wherever it might come from, because it cannot harm me in ways that I can know beforehand."

This is really like being able to click on a link in a website. You don't need to know beforehand what exactly that website might be, because you know that it can't go and read arbitrary files on your machine or install viruses.

[00:25:27] JM: WebAssembly code could be compiled before it lands on my machine. It could be compiled after it lands on my machine. You could have some kind of streaming compilation system. You could do half compilation before half after. How should compilation work in the WebAssembly context?

[00:25:52] TS: That actually does tie closely into these trust relationship questions I just described. If you do not trust the source you're getting your code from, it's actually should be the case most of the time, then you really want to have the compilation happen on your machine. So, the compilation here being taking the WebAssembly code and turning it into something that your CPU can understand. Because otherwise, you lose these security guarantees. You have to trust that whoever compiled it actually did it in a way that still makes it safe. That's one part.

The other part is if it's compiled before landing on your machine, then it's not portable anymore. So, whoever sends it to you needs to know exactly what your machine looks like, what CPU it has and, to some extent, also what operating system you're running. In most cases, that won't be the case. But if, for example, you are in control of a fleet of devices, be it IoT devices or you're the administrator for a company internet or you have edge servers, then you can do the compilation in a centralized way and just send the compiled results to all these machines that you control and that implicitly have to trust you anyways.

[00:27:26] JM: How should caching of code work, or how does caching change with the world of WebAssembly?

[00:27:37] TS: Yeah. That's a really interesting question. This is something where in the browser WebAssembly actually has big advantages, but also outside the browser. WebAssembly can be compiled ahead of time as you just pointed out. But that means it can also happen – The compilation can happen while it is downloading, and then as soon as it has finished downloading, the compiled results or shortly thereafter, the compiled result is there and can be stored in the cache. The next time you're using the same content, not only does the file not have to be downloaded, like for a JavaScript file. It also doesn't need to be compiled.

This is not true for JavaScript, because for a JavaScript to reach high-performance, compilation depends on the engine or preserving the runtime behavior, because JavaScript is very dynamic and the compiler needs to know how actually is behaving at one time to be able to compile it to a highly efficient code. For WebAssembly, all of these can happen ahead of time. So the compiled results can be cached. That is also true for outside the browser use cases, where you might compile an application when you're installing it. Then from then on, you just started as if it were normal native application.

[00:29:02] JM: I'd like to go deeper into the security discussion. How does the security model of WebAssembly present a more – Or I guess could you just describe the security properties of the WebAssembly compilation and runtime path.

[00:29:25] TS: Sure. So, by default, code running in WebAssembly cannot interact with the outside world. Literally, the only thing it can do by default is create heat by using your CPU time. Everything, every way for it to interact with the outside world has to be made available by providing a function for the WebAssembly code to call.

In websites, that means when creating the WebAssembly module or when instantiating the WebAssembly module in a website, you parse in JavaScript functions that the WebAssembly module can call. What it can do? That's for you to define by providing exactly a select set of JavaScript functions to call.

Outside the browser, we're essentially pursuing the same approach, where an application has to request. I want to be able to open files and then the environment it is running in has to decide whether it's fine for that application to be able to open files. If so, a function is passed in that is similar to a sys call in an operating system kernel, but in this case, a WebAssembly imported function, and that allows you to open files.

But that is fine for users in the browser. Outside the browser, it's not quite enough. Because at least in a standardized way. In the browser, you can have a JavaScript function that allows accesses to very specific pieces of data just because that's how you wrote the function. But we don't want you to need to write every way to open a file from scratch every time you're using a WebAssembly module. We want to have something standardized that just allows you to open a file and read data from it.

But if we were just to allow you full access to the hard drive, then we are back essentially to the same security issues that other applications have, because opening a file is something you're almost guaranteed to need. So, with WASI, we go a step further. We only allow you access to a specific set of files that you have to request beforehand, like access to a specific directory, or even only specific files. Everything else, if you try to open it, or open other files, it will just fail.

By that, we are really preserving this property that, by default, the application just can't do anything and it wants to do, it needs to that you know about, and you need to be able to say, "Oh, am I actually okay with this application being able to do this?"

[00:32:25] JM: Are web applications have lots of small modules that we are using, importing, creating. Describe how small modules get used in modern web applications and how WebAssembly might change our usage of small modules.

[00:32:49] TS: Small modules are actually a big focus for us at Mozilla working on WebAssembly tools. We are building the Rust to WebAssembly tool chain, and that is very much focused on making it possible to build small dedicated modules that integrate nicely with an application that is otherwise written in other languages, such as JavaScript.

We are working on making that as seamless as possible so that you can really focus on the languages you're using instead of on the configuration and integration issues between those languages. We're making it so that this is as seamless as possible.

One big factor there is WebAssembly functions can only be called parsing in numbers and returning a single number. Soon it will be multiple numbers that can be returned, but ultimately it's still for now all just numbers. That makes it somewhat painful to operate with. So, you can't parse a string for example. What if you want to have a module that allows you to process strings? You wouldn't want to call into and out of these modules very often. So, the small module use case really doesn't work all that fantastically well with just these interfaces.

So, a big part of our tool chain there is called wasm-bindgen, which allows you to in, for now, Rust code say, "I want to export this function that takes complex data types, such as structs," so objects with properties on them and returns, say, a string. All that complexity of serializing these data types into numbers and then also deserializing a string ultimately as to return value from numbers. We cover all of that and make it – So wasm-bindgen generates bindings code, glue code in JavaScript to hide all of these complexities so that you as a developer can focus on writing idiomatic Rust code to implement the functionality and idiomatic JavaScript code to use that module.

[00:35:29] JM: If I'm adding a module to my application today, I'm often using NPM, the node package manager. How does that import process or the package management process change in the WebAssembly world?

[00:35:50] TS: That really depends on the language you're using. For many languages, I think the answer is currently as the author of a module, you have to bundle up in the right format, write a package.json file and upload it to NPM. For the Rust WebAssembly tool chain, we built this tool called wasm-pack, that takes care of all of these steps for you. So, it takes the output that the wasm-bindgen script created.

In fact, it even invokes wasm-bindgen and the Rust compiler in all the right way and create something that also has a package.json file that is ready for uploading to npm. Then it even acts as an NPM client. So, it allows you to authenticate with the npmjs.com server, with your user account there, and upload your package there. So, it's really a seamless process from compiling your code to uploading it. I've seen people do this in tutorial sessions going from having no – Not adjust this tool chain to uploading their first package within 20 minutes or so.

[SPONSOR MESSAGE]

[00:37:20] JM: Deploying to the cloud should be simple. You shouldn't feel locked-in and your cloud provider should offer you customer support 24 hours a day, seven days a week, because might be up in the middle of the night trying to figure out why your application is having errors, and your cloud provider's support team should be there to help you.

Linode is a simple, efficient cloud provider with excellent customer support, and today you can get $20 in free credit by going to linode.com/sedaily and signing up with code SEDaily 2019. Linode has been offering hosting for 16 years, and the roots of the company are in its name. Linode gives you Linux nodes at an affordable price with security, high-availability and customer service.

You can get $20 in free credit by going to linode.com/sedaily, signing up with code SEDaily 2019, and get your application deployed to Linode. Linode makes it easy to deploy and scale those applications with high uptime. You've got features like backups and node balancers to give you additional tooling when you need it. Of course, you can get free credits of $20 by going to linode.com/sedaily and entering code SEDaily 2019.

Thanks for supporting Software Engineering Daily, and thanks to Linode.

[INTERVIEW CONTINUED]

[00:38:57] JM: I'd like to revisit the lower-level part of this conversation. Our WebAssembly applications are interfacing with lower-level host operating system resources. There is this project, the WebAssembly system interface. Tell me more about that interface.

[00:39:21] TS: So this interface uses this security model I described earlier, where an application really needs to be explicit about what kinds of resources it needs access to. That is one of the most important feature of it. But it is also at the same time really quite low-level as you say.

So, it's in many ways modeled on the POSIX interface, the portable operating system interface for Unix, which is an old standard for what system interfaces look like. It allows you to open files, read from them, write to them, open socket and all of these things. We are implementing many of the same features, but always with this security in mind.

One other important aspect here is that we are building all these in a modular way. So there will be the so-called WASI core, which will have a fairly small set of features in it for doing the most crucial operations. But if that were all that WASI brings, then maybe it might not actually take us that much time to define it. Because if you have access to files and, say, to network sockets, you do not need that many more features to say, "Okay, everything else we can now do by layering protocols on top of these features."

Instead, we want to reduce a lot of these complexity by adding more additional modules to WASI that allow you to do different, more complex things. One important aspect of that is asynchronous I/O, where the design space is really quite complicated and doing it in efficient ways is something where an efficient and portable ways across operating systems is something that it will take us a while to figure out.

But it's something where we know that we can't move the complexity of sorting this out into the applications. You can in theory always build an asynchronous interface in terms of a synchronous one, but it will never be as efficient and it will bring a lot of complexity with it. So, that will be an important module to develop. Then we're thinking about a good number of other modules to define overtime. Really, our work will not be concluded for quite some time.

[00:42:21] JM: Another question about the POSIX standard to help clarify my understanding of the analogy between POSIX and WebAssembly system interface. So, correct me where I'm wrong. So, I think the POSIX standard is useful because you have, for example, a browser, a web browser, will run on – My Chrome browser runs on my Mac OS. I also have a Chrome browser on my mobile phone. There's a Chrome browser on a Linux machine that I have. When the browser makes a request to open a file, for example, or open a folder, because a browser has to do that. A browser interfaces with the lower-level file system. Those calls are made in a POSIX standards compliant requests language.

The reason that's useful is because it allows the person who is writing the browser to just think in terms of these abstractions, like I'm opening a file, or I'm opening a folder, and it allows a decoupling of the browser developer and the underlying operating system developer so that if I develop the process of opening a file from my browser, I just call out to opening a file and then the underlying operating system takes care of that. Mac OS will implement opening a file in one way, and Chrome operating system implements it another way. Linux implements that another way. They have this freedom in the implementation of the underlying POSIX implementation. First of all, is there anything you want to add to that description of the POSIX standard?

[00:44:14] TS: I think that actually covers it pretty well.

[00:44:19] JM: Okay. So, help draw the analogy between that POSIX standard and what WebAssembly seeks to do with the WebAssembly system interface.

[00:44:28] TS: On the level you just described, the goals are very similar, with the addition that we really aim for full independence from upgrading systems after having compiled things.

So, POSIX allows you to have portable code, but an application that you compiled for Linux won't run on Mac OS. Even if the source code is fully portable and even if the POSIX's codes are all the same. There are differences in how applications work, how to startup sequence for applications works. A number of other important aspects where there are just incompatibilities.

WASI needs to work the same everywhere, and that includes in the browser, for example. There are by now at least five or six different implementations of the core of WASI that we so far have defined. They all work very differently. So, I think we have proven this out quite well, that we can design this interface in a highly portable way.

[00:45:55] JM: You work at Mozilla. The skills of a WebAssembly engineer are in high-demand. What it is about Mozilla that keeps you there?

[00:46:10] TS: I believe that there's no better place to advance these technologies we're working on than Mozilla and to make that make sense. A lot of what we're doing around WASI is about ultimately protecting users. You shouldn't risk having your personal data be stolen from your hard drive by running some arbitrary application and you should be able to have your experiences be richer and be more tailored to your needs by having dynamic content run for you in all kinds of different places.

To realize that we have to build a significant ecosystem and we need to form partnerships with many different actors in this space. Working for Mozilla makes that possible in a way that I have a hard time imagining working somewhere else. That allows us really to be decoupled from having to provide, for example, value to shareholders or early investors in a very quick manner. We can take some time to build a bit of better foundations and really build something that can last and that can be stable.

[00:47:48] JM: You are a senior research engineering director of developer technologies at Mozilla, and a lot of your focus is on WebAssembly today. I read an article that you wrote with Lin Clark and somebody – I think there was a third person on that.

[00:48:08] TS: Lou Wagner, yes.

[00:48:09] JM: Yes. It was a great article. It was descriptive of what is going on in WebAssembly. How much work has been done and how much work there remains to do. You've got so much that needs to be done. There are things that need to be done in the browser. You've got this compilation path. You've got this runtime environment. You've got applications ranging from small modules to gigantic applications, like Photoshop. There's so many edge cases and hard, honestly for me, hard concepts to understand. These things like streaming compilation, for example, or multi-threading and what multi-threading means for the world of WebAssembly.

With all of these different things to be working on, not to mention by the way all of these different players. You've got established players like Google and Mozilla. Newer players like Fastly. Everybody's got their own incentives, but it seems like it's pretty positive-sum ecosystem at this point, which is fantastic. But with all of these different surface area to cover, how do you set priorities?

[00:49:32] TS: That really is a good question, and we are constantly evaluating our priorities and we are making sure that we are not spreading ourselves too thin. As you say, this is a pretty positive-sum game for pretty much everyone, or almost everyone at this point. That's truly fantastic to see.

For example, we have really excellent relationships with the WebAssembly people at Google who are for the most part focusing on fairly different use cases from us and where we constantly can take integration from each other, but also can move the ecosystem overall forward quicker by focusing on different use cases than we could if we were focusing on – Fiercely competing on exactly the same things.

You're right. We do need to focus on specific use cases. I think we have a fairly nice focus by now at Mozilla in the same way that as far as I understand it, the folks at Google have a really good focus that is somewhat different.

One thing that is just entirely clear in this is all of these is too big to go alone. So, we need good relationships. We need good partnerships with a lot of different players around all of these different concepts and all of these different pieces that need to be moved into place.

[00:51:23] JM: How will WebAssembly change cloud providers?

[00:51:30] TS: Significantly. So, I'm deeply, deeply impressed by what Fastly is doing, who have bought WebAssembly to their CDN to make it possible to bring dynamic code execution of untrusted code their customers provide to their fleet of cloud servers or of edge compute servers without having to significantly increase the power of these machines.

If Fastly were using more heavyweight, more traditional solutions, such as containers, they would need to install much more powerful machines in place of what they have now. With WebAssembly, they can run tens of thousands of instances of client code at the same time on the same machine. That is really unparalleled, and WebAssembly really allows them to significantly increase how powerful their CDN offering is.

It also begins to blur the lines between what is a CDN and what's a cloud provider. I think, ultimately, the differences will shrink to how close are each of the edge. Are you in a data center somewhere where you might have – I don't know, have a dozen data centers across the world, or are you almost in the cellphone tower, perhaps even in the cellphone tower? I think cloud providers and CDNs, it will be more of a spectrum with CDNs being the ones who are closer to your cellphone tower.

[00:53:21] JM: How will WebAssembly change the lives of software developers?

[00:53:26] TS: Well, hopefully for the better, and I believe for the better. So, one aspect that we haven't touched on yet that I think for software developers will be quite significant is a different kind of portability, and that is tooling portability. The ability to use the same standard tool chain for targeting all of these different platforms ideally and unmodified Rust compiler or an unmodified C, C++ compiler just targeting WebAssembly, WASI as the target and then also being able to use your established debugging tools, such as LLDB instead of having to go for all kinds of different tool chains for, say, IoT devices, which are notorious for having sort of esoteric tool chains where you might need to install quite a few of them depending on what devices you're developing for. I think being able to just use off-the-shelf tools for your development instead of having to learn all these quirky different tools will be a significant benefit to developers.

[00:54:44] JM: Till, I want to thank you for coming on Software Engineering Daily. It's been really fun talking to you.

[00:54:47] TS: Thank you for having me. It's really been a pleasure.

[END OF INTERVIEW]

[00:54:53] JM: GoCD is a continuous delivery tool from Thoughworks. If you have heard about continuous delivery but you don't know what it looks like in action, try the GoCD Test Drive at gocd.org/sedaily.

GoCD's Test Drive will set up example pipelines for you to see how GoCD manages your continuous delivery workflows. Visualize your deployment pipelines and understand which tests are passing and which tests are failing.

Continuous delivery helps you release your software faster and more reliably. Check out GoCD by going to gocd.org/sedaily and try out GoCD to get continuous delivery for your next project.

[END]