EPISODE 1140

[INTRODUCTION]

[00:00:00] JM: Deno is a runtime for JavaScript applications. Deno is written in Rust, which changes the security properties of it relative to node. Parts of Deno are also written in TypeScript, which are causing problems in the compilation and organization path of Deno. Elio Rivero is an engineer who has studied Deno and TypeScript, and he joins the show to talk about the newer JavaScript runtime and the issues caused by TypeScript.

[INTERVIEW]

[00:00:32] JM: Elio, welcome to the show.

[00:00:33] ER: Hi, Jeffrey. Thanks for having me.

[00:00:36] JM: We're talking about a variety of topics in JavaScript land today. Let's start with just node and a little bit of back story. Why has Node.js been so successful?

[00:00:47] ER: I think node.js has been very successful because of it was already written in JavaScript, which is a very flexible language, and we have seen JavaScript apply them everywhere. Even the SpaceX consoles had some JavaScript in them. There was no language barrier, so you could immediately access the node and start working with. I think that's part of their success. The other part that might be the V-8 engine that made it superfast and very reliable.

[00:01:21] JM: What are the problems with node?

[00:01:22] ER: Some of the problems with node, maybe I get where you're going. You're going to end up with talking about Deno. So one of the issues of node that wasn't an issue back then, but it is now and I know this is quite outdated now, for example, back when node was created, there were no promises, which we have now for example. So node was successful for its time. It had some other issues like when you import to the module, that decision of leaving out the file

extension made it quieter and troublesome to figure out if you are importing from a module or on a specific file or a directory, I mean, or a specific file. Those things were a bit awkward or confusing maybe.

[00:02:14] JM: What language is node written in?

[00:02:17] ER: Node is written on JavaScript.

[00:02:20] JM: And what are the security issues that emerge from that usage of JavaScript and how it's written?

[00:02:25] ER: Well, there was basically a decision of not saving some security measures to access the local computer, right? That was changed in Deno where they're accessing the network or the file system. It's safe by default. You have to specify some flags, so your script can access the file system network or whatever.

[00:02:51] JM: Well, I suppose to properly start talking about Deno, what is Deno?

[00:02:56] ER: Deno is the new runtime created by Ryan Dahl, the original creator of Node.js, and it builds up on all of his experience on what was successful and what was not successful on nodes. There is some confusion about the pronunciation about – On the TSConf 2019, Ryan Dahl clarify that the official name was pronounced Deno.

Deno, to start with, is built on Rust. It was originally written on Go, but then it was rewritten from scratch on Rust to make it more safe and also to avoid double garbage collection. It's also written in TypeScript. You can run a native TypeScript code in Deno. It's also built on top of the V8 engine, the JavaScript engine. Unlike node that uses – I think it was – I forgot what was node using for the event loop. Deno is using Tokio for the event loop.

Some of the features of Deno is that it's a single runtime and its distributor has a single runtime. The execution is always safe, like we were mentioning. Unlike the nodes, you need to provide a specific access, and it has a certain – A lot of flags to specify read access, write access, network access, whether you wanted to access the environment and the environment variables, if you

wanted to measure execution time, if Deno can run sub processes, and also if you want to execute any Deno plugin. There's also a flag that allows all others, but you should rather use the flags that you were for the permissions that you will be actually using.

The other difference with nodes is that node uses the common JS model import. But Deno uses the ES module import syntax. That's quite a bit of difference because Deno also references a file complete with the file extension, so you always know where you are importing from your modules, either a JS file, a TS file. It's always very certain. Right next to this module importing, you have the fact that unlike node that could only import modules. For example, you could import module locally, but you usually import modules from NPM.

But Deno, you can import modules using a URL and on the Internet, so you just blow a module to GitHub and just import it from there, and it will work. That means that you no longer are attached to NPM or any other package registry system. You can freely import from anywhere. Of course, that also means that you don't have a node modules further anymore in your project. When you import files, Deno catches the file locally. The package, the modules locally and you can reference them by aversion, which is a tag in the repository.

[00:06:22] JM: Have you spent much time using Deno yourself?

[00:06:25] ER: I have been tinkering with it a lot. I build some command line utilities mostly inspired by Embers. The Ember.js framework has a CLI tool, so you can quickly generate components or roots, and we don't have a specific one for react, which is the layer in the library I use the most. So I made a tool on Deno, for example, to quickly generate folders and files for whenever I wanted to create a new component. I could invoke this tool and it will create the JavaScript file for the component. It will create Sass style sheet and a test file for Jest.

That kind of things is very easy to do with Deno, and it's very safe too because you are using the TypeScript, which is another advantage of Deno over node that Deno has a native compiler that understands TypeScript. So you can very straightforwardly write TypeScript and execute it on Deno. You don't need to set up anything. Just feed a TypeScript file and Deno will understand it. That's another advantage. Also, Deno has a native support of WebAssembly. You can read a file using one of the file input-output functions in Deno. You can just write a

WebAssembly file and invoke a couple of constructors in Deno, WebAssembly.Module and WebAssembly.Instance. Your WebAssembly file is ready to work. So that means that you can write, for example, a code in Rust and use a tool to transpire this into WebAssembly.

You also need an interrupt file, so Deno can understand. For example, if you are using U64 integers in Rust, those have to be BigInt on Deno, and you need an intermediate JS file to make this conversion. But it's totally possible to write the code in Rust and compile it to WebAssembly and load it with Deno. One library I use for that is called SSBMUP, M-U-P. It does everything for you. it creates the WebAssembly file and the interrupt file, so you can load it with Deno.

[00:08:52] JM: So are major organizations actually using Deno or is it still like kind of universally experimental at this point?

[00:08:59] ER: I haven't seen any big organization, to be honest. I have yet to see one. I guess the dependency on node is node is too established by now and you can have any kind of stuff like from some server stuff. I mean, the most basic express to more complicated stuff like a library to call for Lego Boost or whatever. We have yet to see those kinds of integrations on Deno. So, no, to be honest, I haven't seen a big organization. I assume that a lot of organizations might be experimenting with Deno but I don't know to be sure.

[00:09:38] JM: Do you see it as a de facto replacement for node eventually? Is this an inevitability or is it an alternative ecosystem?

[00:09:46] ER: I think for a time, they're going to be living together. I think there is a place for Deno. Development is so much faster with Deno. You'll going to need to deal with all the node modules and all that, and you get TypeScript out of the box. So in every way, Deno is very modern approach to writing JavaScript tools, but I don't see Node.js going anytime soon.

[00:10:14] JM: Are you a contributor to Deno yourself?

[00:10:17] ER: No, not really.

[00:10:19] JM: So the security principles we mentioned earlier, the security principles of node,

do you see those as improved in Deno? Are the security principles patched up?

[00:10:29] ER: Yeah. Those are already in the flags that I mentioned earlier. For example, you have this flag, allow-read, that will allow a unrestricted rating to the file system, but you can use the flag and pass a parameter like /temp, and you will only be able to read that directory. You can also - allow-write flag has something similar. If you want to allow read or write two different directors, you can separate them with a comma. There are others like these. For example, they allow net its flags, so your program can access the network and you can restrict that to a single domain on the Internet like google.com. I don't know. But if you don't specify that parameter, you cannot have unrestricted access to the network.

[00:11:25] JM: How does the import process for Deno compare to that of node?

[00:11:29] ER: The import process in Deno is in many ways superior to Node.js because for it to start with, we are using S modules, the modern approach to importing modules in JavaScript. The difference with Node.js, which by the way, it was using the CommonJS, the require approach. But in Deno, when you use the S modules import, you can specify the file extension. So you don't import from /functions.ts and you know specifically where you are importing from because one confusion in Node.js was that since you didn't enforce the file extension, it was confusing whether you were importing from a file or a directory that was referencing the unindexed JS file.

But in Deno, you always know where you are importing from. That's especially true and useful in the case where you import a module from the Internet. Like you have , for example, this Deno length, which is now the repository is more like a forward, but you can refer the entire URL and end with a file extension. For example, there's a package called ensure_dir that checks that directory exist. You can reference it with the .ts extension. So it's always very sure and safe where you are importing from.

[00:13:03] JM: Thanks for that detailed summary. Well, can you describe the event loop of Deno?

[00:13:10] ER: I know it's been on Tokio. But, no, to be honest, I don't know if it's –

[00:13:17] JM: Tokio is like the networking or event stack for Rust.

[00:13:22] ER: Yes, Tokio is the event loop for Rust.

[00:13:26] JM: Okay. You mentioned earlier node was written in JavaScript. Are you sure? I thought node was written in C++?

[00:13:32] ER: Yeah. Node is written on C++ and JavaScript, yeah.

[00:13:37] JM: C++ and JavaScript. Okay, cool. Well, let's get to the point. How does TypeScript fit into Deno?

[00:13:44] ER: Well, TypeScript fit into Deno has some more safe a way to write programs, a way to make things more certain and expected. You know that when you write with TypeScript, you are forced to type everything, so you always know what you're getting. That kind of fail proof typing is extremely useful for large JavaScript software. We recently had to write something that was critical involved with money, and we went with TypeScript approach to make things super stable and predictable. I like that we're predictable in TypeScript. You always know what you are getting into.

The thing is that Deno can work with TypeScript natively. You can fill a TypeScript file, and Deno will parse it just like any other JavaScript file. Deno doesn't care about whether it's TypeScript or JavaScript. The thing is that there are some issues that the Deno team found with TypeScript because they were used in transcript for the Deno core and also for the use of facing API, the code. But they were finding that compiling the TypeScript code was taking a long time, and it was increasingly getting longer and longer.

But another issue they found most important was that they didn't found that TypeScript was helping in
organizing the code because one of the issues they mentioned is that they ended up with two body classes in separate locations. Also, they were maintaining like two TypeScript compiler host, one for the Deno code and the core and another for the external user codes. But both of

them had a similar purpose and goal. In the end, they decided to go back just JavaScript for the core. They will still retain – The user code will still be type checker, will still run through the TypeScript compiler. But they ended up working with JavaScript only for Deno core.

[00:15:52] JM: How does the TypeScript support within Deno compared to that of node?

[00:15:56] ER: Well, as mentioned, you can write native TypeScript and just load it on Deno. You can get all the definition of types in Deno writing, for example, Deno types. You will need to set up anything for Deno to understand TypeScript. You just write your program. It doesn't need any compilation. Deno will just run your TypeScript file.

[00:16:22] JM: Wow, that's interesting. Now, how is that possible? How can you run TypeScript without first compiling it down in JavaScript?

[00:16:30] ER: Well, the compiler is already integrated on Deno.

[00:16:35] JM: Interesting. So it's happening. It's just you don't ever have to have the output of the JavaScript file from TypeScript.

[00:16:44] ER: Exactly.

[00:16:45] JM: Interesting. Why is it even useful? Why does that matter? I guess one way of looking at it is it's kind of like it treats JavaScript code the way that Java treats the JavaScript or the Java bytecode. It's a compilation artifact. You don't even need to expose it to anybody.

[00:17:02] ER: I don't quite understand the question.

[00:17:04] JM: It's not a question. I was just saying like I'm trying to understand how the fact that there is no JavaScript artifact from the Deno compiler when you're compiling TypeScript, like why that is a notable comparison to the node ecosystem.

[00:17:20] ER: Well, just having TypeScript running on Deno has all the benefits of TypeScript itself. Plus you don't need to set up anything to make it work. Also, you can write JavaScript

code and TypeScript code side-by-side on the single Deno project. It's not that the project has to either use TypeScript or JavaScript. You can mix both languages on a Deno project, and Deno will just compile then the TypeScript and use it when you import it in another module. But you're free to use that. I think that lowers the entry barrier to start working with Deno because if Deno had forced or enforced the TypeScript, you will have a much different [inaudible 00:18:08]. Being able to use in JavaScript, and whenever you're ready to make the jump start using TypeScript. In a way, the retaining the low entry barrier of Node.js, except that you can like eventually upgrade let's say to start using TypeScript on your projects.

[00:18:31] JM: Okay. That's a lot of clarification. Thank you. But what are the problems with TypeScript within Deno? Why did you write this article?

[00:18:40] ER: There was a document they published, the Deno core team, mentioning these unfavorable situations they were finding, specifically that the TypeScript compile time when changing files was taking several minutes. We're used to things compile very fast. Even when it takes a few seconds, I start getting anxious. So I can imagine that they specifically mentioned that it took several minutes. I can't imagine how long that will be, but it already makes me more anxious.

Also, that the TypeScript structure that they were using on source files, it was creating some run time performance problems. Another reason was that the internal code and the runtime TypeScript definitions had to be manually kept in sync because the TypeScript compiler, they were unable to use it to generate the definition files for TypeScript. I already mentioned before that they work maintaining two TypeScript compilers, one for the core of Deno and another for the external uses codes.

Yeah, so those are the main issues that they stated as the reason to start using just JavaScript on the Deno core. I think that my might have been very successful because you will no longer have the long compile times of TypeScript. You're just using JavaScript, so that should've solved a bunch of these issues like the long compilation time, maintaining the two compilers. They are still keeping the compiler that compiles the user codes but no longer the one for the core codes, and they no longer had to maintain the definitions and the DTS files. So I think there are a lot of improvements in several areas for performance and coding experience.

[00:20:42] JM: Why is compile time such an issue?

[00:20:46] ER: I don't know. Maybe because time is money. We all want the things to compile as fast as possible.

[00:20:53] JM: Well, how else has TypeScript been counterproductive to Deno?

[00:20:58] ER: Well, supposedly in – TypeScript helps you define. For example, it interfaces your types, and those are meant to be unique throughout your application. But they ended up with different classes in different locations, and the compiler never detected that. The compiler just assume it was okay because everything was type of. That was counterproductive because they were duplicating the work on these two classes.

[00:21:32] JM: What is required to remove TypeScript from Deno?

[00:21:37] ER: I assume that the first step they took was to remove the TypeScript compiler and stop using it for compiling the codes and migrate the files to plain JavaScript. Yeah, those should be all that's needed.

[00:21:56] JM: So the end result is just that Deno no longer compiles TypeScript files.

[00:22:04] ER: Yeah. For the user, it's the same experience. You can continue using your TypeScript files. But the core of Deno is really now in just JavaScript.

[00:22:15] JM: Gotcha. The core of Deno is written in JavaScript or the core of Deno only compiles JavaScript files or only executes JavaScript files.

[00:22:26] ER: Yeah. The Deno runtime is using Rust and JavaScript now. It's not that it's compiled inside Deno. I mean, because JavaScript doesn't need compilation for it.

[00:22:38] JM: Or, sorry, execution.

[00:22:40] ER: Yeah, execution.

[00:22:42] JM: Right, okay. Now, to summarize, basically what we've been talking about is that TypeScript compilation was too slow for Deno, and therefore they remove the ability to compile TypeScript in Deno to compile these TypeScript files or these files that had TypeScript and JavaScript in them. Now, you just have to execute raw JavaScript on Deno.

[00:23:05] ER: No. Let's make it clear. You can continue executing your TypeScript files. The core of Deno is no longer written on TypeScript as it was before.

[00:23:17] JM: Sorry, I misspoke. Right, okay. So the core Deno maintainers are no longer using TypeScript.

[00:23:23] ER: Yeah, that's correct.

[00:23:25] JM: Right. That's what I should've said. So let's talk about Deno in the abstract. How do you predict Deno affecting web development in the long run?

[00:23:34] ER: I think we'll be seeing more like CLI tools that are traditionally written, I don't know, with Python or with Node.js currently. I think we'll see a lot more of these CLI tools. Deno has a lot of tools to help this process like you have. For example, you can write tests in random with Deno with – You don't need any external framework. You don't need Mocha or Jest. There is actually a simple command, deno test, where you can run all the tests. You have this other command, for example, the fmt, that will format all your files to the standard specified by Deno.

You have another command, for example, that's called bundle that will create a bundle with all the files that you have and would create a single one with a result. For example, there's another command called install, and you can reference modules from the Internet and you can install it locally and you can run it like any other shell tool. The amount of tools that – All these tools are in a URL called deno.land/manual/tools. But the amount of tools that Deno has specifically for this gives us a good idea that maybe Deno was built to create with the intention of creating these CLI tools.

[00:25:15] JM: What about the WebAssembly side of things? Do you have any predictions for how Deno supporting WebAssembly will improve the web ecosystem?

[00:25:23] ER: Well, the WebAssembly that runs on almost native speed. So having code read-in, like, for example, in Rust and compile it to WebAssembly that runs at that speed is fantastic. I think that what really benefits from the speed of WebAssembly are video games, for example. I think we'll be seeing more and more amazing experiences regarding video games played right in the browser. With Deno, that gives Deno an incredible execution performance just being able to go through huge amounts of data at native speeds that will be amazing. It will give Deno a niche over Node.js, for example.

[00:26:12] JM: What else excites you about the future of web development?

[00:26:15] ER: Well, what excites me a lot is this WebAssembly for sure. I think that between Rust and WebAssembly on the web and Deno on the backend, those are kind of like the future of the open web development because that's an important ride that these languages and Deno are open. They don't belong to a single company, even though Rust and WebAssembly are heavily backed by Mozilla. But they are open, free for everybody to contribute to and work with. So this Rust, WebAssembly, and Deno are probably going to be the future of the open web.

[00:27:02] JM: Well, that sounds like a great place to close off. Thank you so much for coming on the show, Elio.

[00:27:06] ER: Thank you, Jeffrey. Thank you for having me.

[END]