EPISODE 958

[INTRODUCTION]

[00:00:00] JM: HTTP is a protocol that allows browsers and web applications to communicate across the Internet. Everyone knows that HTTP is doing some important work, because HTTP is at the beginning of most URLs that you enter into your browser. You might be familiar with the request response model as well as HTTP request methods such as get, put and post. But unless you've had a reason to learn more about the details of HTTP, you probably don't know much more than that.

Julia Evans is a software engineer and writer who creates Wizard Zines, a series of easy to read online magazines that explain technical software topics. Julia's Zines include Linux debugging tools; Help! I have a manager; and recently, HTTP: Learn your browser's language. Her zines are a creative, innovative format for describing the world of software engineering while also exploring her own artistic pursuits in writing, design and illustration.

Julia was previously on the show to discuss Ruby profiling. That was a great one, and she's returning to the show to discuss HTTP as well as creative process and her goals with Wizard Zines.

Upcoming events that Software Engineering Daily will be at; KubCon San Diego 2019 and AWS Reinvent Las Vegas. We're planning a meet up at Reinvent. There's a link to that event in the show notes, and that will be on Tuesday, December 3rd, I think. We're looking for a venue. If you've got some place that has some space in Las Vegas at the event hopefully, let me know. Send me an email, jeff@softwareengineeringdaily.com.

[SPONSOR MESSAGE]

[00:01:56] JM: Being on-call is hard, but having the right tools for the job can make it easier. When you wake up in the middle of the night to troubleshoot the database, you should be able to have the database monitoring information right in front of you. When you're out to dinner and your phone buzzes because your entire application is down, you should be able to easily find

out who pushed code most recently so that you can contact them and find out how to troubleshoot the issue.

VictorOps is a collaborative incident response tool. VictorOps brings your monitoring data and your collaboration tools into one place so that you can fix issues more quickly and reduce the pain of on-call. Go to victorops.com/sedaily and get a free t-shirt when you try out VictorOps. It's not just any t-shirt. It's an on-call shirt. When you're on-call, your tool should make the experience as good as possible, and these tools include a comfortable t-shirt. If you visit victorops.com/sedaily and try out VictorOps, you can get that comfortable t-shirt.

VictorOps integrates with all of your services; Slack, Splunk, CloudWatch, DataDog, New Relic, and overtime, VictorOps improves and delivers more value to you through machine learning. If you want to hear about VictorOps works, you can listen to our episode with Chris Riley. VictorOps is a collaborative incident response tool, and you could learn more about it as well as get a free t-shirt when you check it out at victorops.com/sedaily.

Thanks for listening and thanks to VictorOps for being a sponsor.

[INTERVIEW]

[00:03:47] JM: Julia Evans, welcome back to Software Engineering Daily.

[00:03:50] JE: I'm so excited to be here again. Thanks for having me.

[00:03:53] JM: We're going to talk about HTTP in gratuitous detail, but before we get to that gratuitousness, let's talk about the browser. I want to start with the browser, because I think this is an easier way for people to understand HTTP. What happens when I enter an address into the address bar and press enter?

[00:04:16] JE: Right. I guess it's a bit of a question of how much detail we want to go into. But one of the things that happens – Well, okay. Let's just talk about the whole thing. The first thing that happens, let's say you type in google.com, it will do a DNS look up for the domain name,

like google.com, and it will get an IP address back from the DNS server. Then once it gets that IP address, it will send an HTTP request for the google.com site to that IP address.

[00:04:45] JM: Of course, saying it's just google.com is a bit reductive. What does the actual address that we are entering in its full form, what does that look like? Explain what it is.

[00:04:59] JE: Right. Because you don't just have google.com. Right now, we're using Zencastr, so like zencastr.com/jeffmayerson/http, right? So you have the domain name, but you also have this path, which is the after sort of like zencastr.com/. When you make a get request, you're not just making a request for google.com. You're making a request for like this thin after the /, like /calendar, or like /jeffmayerson/http.

What gets sent in the get request is you send the path, which is like – Let's use /calendar, because it's less to say if we're talking about google.com/calendar. You say like get/calendar and then you also will say – Because when you make a request to this IP address, it doesn't necessarily actually know that you want a site like google.com. So you also send that header which is like, "I'm interested in the website google.com." That's the domain that I'm working with.

[00:05:53] JM: It's also all prefixed by HTTP or HTTPS to indicate the protocol that you are using to communicate with that IP address that you're targeting.

[00:06:04] JE: That's right. That's a little bit interesting, because HTTPS includes HTTP. If that makes sense? HTTPS isn't a totally separate protocol. HTTPS is like you're sending an HTTP request, but that request encrypted, but you're still sending an HTTP request that looks sort of the same as the HTTP request that you would send if it weren't encrypted. It's just that you encrypt it.

[00:06:28] JM: Most developers have used cURL. They've probably used it from the terminal. How does the process of executing a cURL command compare to what a browser is doing when it's looking up an address?

[00:06:43] JE: In a lot of ways, it's the same. Let's talk about what the same and what's different between what cURL does and what your browser does. Actually, if you do cURL-v, you do

cURL-v google.com, and cURL-v will tell you exactly what cURL is sending to the server. The big difference between what your browser does and what cURL does is the HTTP header it sends.

Every HTTP request in response has these headers, which are sort of like these key value pairs. One example of a header is, for example, like accept language, which says like what language you would like to receive your results in or the user agent header, which says what browser you're using. By default, if you just make a request with cURL for like google.com, it will only send basically like two headers. It will send a user agent header that says this request comes from cURL, and it will send the host header, which is like a required part of the HTTP vertical that every HTTP request has to have.

But your browser will typically send a lot more headers. It will send cookies that it saved from like previous times visiting the website. It will set an accept language header and it will probably set a bunch of others. Yeah, the big difference is that, are the headers.

[00:08:00] JM: At this point, there's a bunch of people who are listening to this and they're like, "This has nothing to do with anything I'm doing at work. I'm working on a big Spring application or like I'm making my microservices. I don't know how HTTP works. Frankly, I don't care. I don't know what op codes my computer is using below the surface to run my assembly language. Why should I care about any of these? Isn't this low-level stuff, like assembly code?"

[00:08:34] JE: Right. Actually, to me, the reason HTTP is exciting is I think that a lot of it is very relevant to what's happening on your computers in real-life. Let's talk about a few of those things. One I think really basic thing that is relevant to many people that HTTP is like the status code that an HTTP response returns. Sometimes you'll see HTTP 200, HTTP 404, 400, 503, 500, and I think if you're developing a website, these are things that you're going to have to see and you're going to have to return, and you're going to have to decide which response code to return sometimes. You need to think about, "Okay. Should this be a 404? Should this be a 500?" Do you want to talk about status quotes?

[00:09:14] JM: If you do. Absolutely.

[00:09:15] JE: Yeah. I think one thing that took me awhile to understand is because you have all these codes, you have 200, 301, 400, 500. At first it seems a little how do you know which one's which, but actually it's pretty easy to classify them, because they all start with either 1, 2, 3, 4, 5. The ones that start with 2 mean that the request was successful. 200 okay is the most common one you see. 2 is good. The ones that start with 3 are typically redirects. You'll say like, "Okay." What you're doing is you're telling the browser, "Hey, this webpage isn't here anymore. Go to this other website instead." There are permanent and temporary read which we will talk about later, because that's a super practical thing and it's very important to get right, and it's one of these things where it's useful to actually understand how the HTTP protocol works. But we'll talk about that later maybe.

The other ones that start with 4 – 4, it supposed to mean like it's like a bad request. For example, if I requested a website that doesn't exist, like Julia.com/cats or something, it's not the server's fault that that page doesn't exist, right? I chose in some sense to like request a page that isn't there.

Another like 4 class – That's like 404 not found, which we've all seen a lot, I think. There's also a 403 forbidden, which means that like you're supposed to authenticate somehow by sending an API key or something, and you didn't send the right one. Again, it's sort of like not the server's fault that you don't know the password, right? You've left out some of the authentication information.

Another example of 400 class code is 429 two-minute request, which is like you're being rate limited, which is one of these like more specific HTTP response code, but that you do see in [inaudible 00:11:03] APIs. You could be using an API and send too many request per second to it and it will send you back a 429 two-minute request. It's like useful to know what these things mean.

The last class is like the 500 class of error or like 5XX, which means that something went wrong with the server. Often, if there's an exception in your application, like it crashed, these will sort of like the idea that it's the server's fault and it's not the requester's fault.

[00:11:30] JM: HTTP is a protocol. What do we need to know about the layers of abstraction below HTTP?

[00:11:40] JE: I would say – What you need to know – The way I think about the layer of abstraction below HTTP kind of at the simplest level, because there is a lot to know about it. But at the simplest level, you're sending – The client sends a stream of bytes and the server sends back some bytes. It's like some information is being sent. You could think of it as like a file or something. But it's like some information is being sent to the server and then some information is being sent back to the client.

[00:12:10] JM: It's worth pointing out that this could be over either TCP or UDP, right?

[00:12:14] JE: Not for HTTP. HTTP only works over at TCP.

[00:12:17] JM: Oh, okay!

[00:12:18] JE: Yeah. Because UDP is actually quite different, because TCP is sort of a stream. Let's say I'm sending you like a megabyte of data. I can send you a megabyte of data over TCP and I know that you'll get that megabyte kind of in order, but UDP packets are not like that. I can't just say like, "Oh! Send this IP address this megabyte of data in order over UDP."

That's not a thing that the protocol supports, because you can only send individual packets and they can't be that big and you don't know if they're going to get there. TCP is sort of like takes care of like assembling all of the network packets that are being sent and it allows you to pretend that you can just send someone a megabyte of data even though that's not something you can do over a computer network in real-life. Does that make sense?

[00:12:59] JM: It does. Maybe we can discuss this later, but I thought that the cutting edge HTTP stuff, like I don't know if it was HTTP 2 or HTTP 3. I thought there was some cool, new UDP kind of –

[00:13:12] JE: There is. Yeah, that's true. HTTP 3 is implemented on top of UDP. But I think sort of how I think about it is almost as if like they re-implemented TCP on top of UDP but a different way and then implemented HTTP on top of that, HTTP 3 on top of that. Does that make sense?

[00:13:30] JM: Absolutely.

[00:13:30] JE: There's something like TCP. I honestly don't know a lot about HTTP 3. I haven't looked into it. I know some things about HTTP2, because I've used it. But HTTP 3 I think is very cutting edge or like – It's quite new. I don't know what that protocol looks like exactly.

[00:13:47] JM: Here's a naïve question. Why is HTTP so important for browsers?

[00:13:52] JE: It's really because every time your browser requests any website, it uses HTTP. It's what powers the entire web. You can't go to a website without using HTTP. That's the only thing you can do. I think the other reason that's important actually is that is a relatively simple protocol. That's sort of like why the web works.

It's like anyone can implement an HTTP server and people have implemented HTTP servers in so many different languages. I just think that that's – Because all you're doing is sort of like you send a request, like – I don't know, get/calendar. Then the response – The rules of an HTTP response are sort of like there needs to be a status code, which is like 200, 300, whatever, 404. There needs to be headers and there's a body, a request is similar. In a request, you have a path, you have headers and you have potentially a body.

I think because the HTTP protocol is actually so simple, you just have the request path or the status code, the headers and the body. It means that you can really understand it pretty easily, or you can understand at least parts of it. You might not understands every single header, but at least you're like, "Okay. There are headers there. This is the one I care about."

To me, that's sort of why it's important. I think it's a big part of probably why the web is successful, right? Because if this HTTP protocol was a lot more complicated, people probably would be able to implement HTTP servers that easily.

[00:15:18] JM: This is one thing that I realized. I read your zine. That's one of the impetuses for doing this episode, is you wrote a zine on HTTP. A zine is this – You're the only person I know who has used that term. I'm sure it's a term that is used elsewhere, but it's a term you use to describe your written, well-illustrated short-form ebooks, or not short-form, not long-form, I guess medium-form ebooks. They are like magazines.

[00:15:45] JE: Whatever 24 pages is.

[00:15:48] JM: Yeah. 24 pages of nicely illustrated easy to read content. But the reason that HTTP is actually quite a good selection of a topic to cover is it this thing that I kind of glossed over. I never really went deep on like what HTTP was basically for the same reasons that I outlined earlier. I don't care about this thing. It's not solving any of my problems in terms of the code that I'm actually writing. Because most of the things that I work with are at abstraction levels that gloss over where I actually need to know about this.

Actually, even that granted, and that's not true for every programmer. There are plenty of programmers who are operating at that level. Learning about it was actually a little bit reassuring. It was this nice clarification on things that were previously opaque to me, and it made me feel like a little bit less of an impostor.

[00:16:47] JE: What something that you learned?

[00:16:49] JM: I mean, I'm going to expose how dumb I am. But like I didn't really know that this was just how browsers were talking – I mean, it sounds naïve, but like I just didn't even think like, "Oh, yeah. My browser is just making all these HTTP requests to servers. It's a simple request response framework, and this browser is the super complicated thing that's just making tons and tons of HTTP requests." I guess I just hadn't thought about it.

[00:17:19] JE: Yeah. No, totally. I think this is why I love to explain this stuff, is because, honestly, I think I learned about HTTP at some point. I don't know when. Definitely when I started out, I didn't know about it and it was something that I had to learn. When I did learn about it, I kind of have this feeling. I was like, "Wait. Why did no one tell me this wasn't that complicated?"

[00:17:37] JM: Absolutely.

[00:17:39] JE: Yeah. I think it has a lot of – I think one important application of HTTP is have you ever worked with a website that was cached?

[00:17:49] JM: I imagine. I mean, I use WordPress, and WordPress has literally a button that you can click to clear your cache.

[00:17:56] JE: Right. Okay. Yeah, that makes sense. Yeah, or like sometimes – I used to work on Stripe.js, which is this like JavaScript file, or I didn't work on it. I worked on serving it, if that makes sense. Making sure that it was being searched correctly from the web servers. This was like this JavaScript file that we had that we were serving to a lot of people and it needed to be like cached by like a content delivery network. But if you have – Maybe a better example of this is just like my personal blog. It's like when I update my blog, the front page cached basically, so I can save money on low-costing.

When I publish a new blog post, then I refresh the page. The new blog post won't be there because it's being – The server that's been served on has like an old version of it. The thing that I think is kind of cool is that if you look at the HTTP headers for response, it will often tell you, "Oh! This was cached by like a content delivery network or something." There's often an H-header that's like I've been caching this for – This is 300 seconds old. If it tells me, "Well, this version of this resource, this HTTML page is like two-hours old and I published my blog post 5 minutes ago, then I know that I need to clear the cache.

[00:19:17] JM: If we think about that from the CDNs perspective, are they literally using the header that – Or like one of the headers that is defining cache behavior, like cache control? Are they using that to route information or to actually do deletions or purge elements from their caches?

[00:19:41] JE: Yeah. The CDN will actually look at the headers that use that and make decisions on what to cache based on it. Exactly. If you say like cache control max age equals

300 seconds. If it gets a request and then look at what it has in its cache, that's one to 5 minutes old, it will delete it and request a new version of the resource.

This is actually – If you're developing a website, which like let's say has like static content, for example, files. I'm trying to say images or JavaScript, especially if you're trying to serve new JavaScript, it's very important to make sure that it gets cached correctly. Because if you're serving old JavaScript for the new version of your website, your website isn't going to work.

[00:20:20] JM: Absolutely. I mean, there was a case where we do a lot of episodes. One of the episodes was with a particular company, and we had used an outdated logo of the company. Obviously, companies change logos overtime, and a 20-year-old company, if you take their logo at 15 years, it looks very outdated relative to their most recent logo.

We used some outdated logo, and actually we had an issue where when we shared our podcast episode to LinkedIn, LinkedIn cached the featured image, which included the logo. Then the company reached out to us and we updated on WordPress. But try as we might, we couldn't figure out how to get LinkedIn to purge its cache. The company was like, "Why is this incorrect image still showing up on LinkedIn?" I'm like, "I'm sorry. I don't know how to fix that. Try as I might."

[00:21:18] JE: Yeah. Right. It's really true.

[00:21:20] JM: But maybe I could have sent some kind of like cache-busting request to the LICDN domain and could have gotten it to work.

[00:21:31] JE: Maybe. I feel like that's a different – They're probably not doing it based on just HTTP. It's probably a different kind of system.

[00:21:39] JM: Probably.

[00:21:40] JE: Yeah. Totally. Caching is a huge pain, but also very important, right? I think a lot of websites would not be on the internet today if – They would not be able to survive if it weren't for caching.

[00:21:54] JM: Absolutely. There are these different kinds of HTTP request methods. Everybody knows that. There's get, post and some other ones.

[00:22:05] JE: I think that's a good description, get, post and some other ones.

[00:22:08] JM: Exactly. Although there's still not like a pay API. That would be cool. HTTP request for pay. Maybe we could talk about that later. But wasn't there like a status code, an HTTP status code for like didn't pay but they never actually implemented it?

[00:22:27] JE: Yeah, I think there is. I don't know about it. But I heard that too.

[SPONSOR MESSAGE]

[00:22:27] JM: I remember the days when I went to an office. Every day, so much of my time was spent in commute. Once I was at the office, I had to spend time going to meeting rooms and walking to lunch and there were so many ways in which office work takes away your ability to be productive. That's why remote work is awesome. Remote work is more productive. It allows you to work anywhere. It allows you to be with your cats. I'm looking at my cats right now. But there's a reason why people still work fulltime in offices. Remote work can be isolating. That's why remote workers join an organization like X-Team.

X-Team is a community for developers. When you join X-Team, you join a community that will support you while allowing you to remain independent, and X-Team will help you find work that you love for some of the top companies in the world. X-Team is trusted by companies like Twitter, Coinbase and Riot Games.

Go to x-team.com/sedaily to find out about X-Team and apply to join the company. If you use that link, X-Team that you came from listening to Software Engineering Daily, and that would mean that you listen to a podcast about software engineering in your spare time, which is a great sign, or maybe you're in office listening to Software Engineering Daily. If that's the case, maybe you should check out x-team.com/sedaily and apply to work remotely for X-Team.

At X-Team, you can work from anywhere and experience a futuristic culture. Actually, I don't even know if I should be saying you work for X-Team. It might be more like you work with X-Team, because you become part of the community rather than working for X-Team, and you work for different companies. You work for Twitter, or Coinbase, or some other top company that has an interesting engineering stack, except that you work remotely.

X-Team is a great option for someone who wants to work anywhere with top companies maintaining your independence, not tying yourself to an extremely long work engagement, which is the norm with these in-person companies, and you can check it out by going to x-team.com/sedaily.

Thanks to X-Team for being a sponsor of Software Engineering Daily.

[INTERVIEW CONTINUED]

[00:25:14] JM: So you have these things, get, post, put, other things. Why does it even matter what kind of HTTP request method we have? Because isn't everything in the body – Doesn't all the processing actually happen at the behest of the body?

[00:25:28] JE: Yeah, that's a great question. Traditionally, you use get request to get resources. I want this image of a cat please, like get/cat.png, and you use post request to send information to the server. I would like to add a new cat to your list of cats, for example. Who knows? Post requests are used, for example, in like a forum submission. When you're logging into a website, you put your username and password and send it to the server, etc. In principle, like we said, get request can have a body. They typically don't, but they can. You could just as easily send your log-in request with a get request. It's like, "Why do you need both?"

One important thing is the browser security model. Browsers do a lot of work to protect you from like malicious websites. For example, let's say you're on some sketchy website – Okay. Let's say –

[00:26:39] JM: I would never –

[00:26:40] JE: Yeah. I'm on a sketchy website. That website is like – Maybe I'm also logged in to my bank at the same time, because why not? Who knows? That website tries to loan an image from my bank. It loads my bank's logo from their website. That's called like a cross-origin request, that's allowed. It can embed – I don't know, my bank's logo and the bank is like, "Who cares?" But if it's like a post or if it sends a post, tries to add a post request to my bank's website, and it's like, "Can you withdraw money from Julia's account?" The browser will be like, "No. Post request to this other website are not allowed." You're not allowed to send that request.

Even though conceptually you can do whatever you want with a get request and a post request, they're not treated the same way by the browser security model. Cross-origin get request or like certain kinds of get requests are allowed, and cross-origin post request with bodies are not allowed

[00:27:37] JM: Just to make sure I understand. You're talking about the fact that you have a page open on two different websites and one website is contacting another?

[00:27:47] JE: Yeah, exactly. Not the website, but the JavaScript on that page. The JavaScript on that page is trying to send a request to the other website.

[00:27:57] JM: I think this might be unintuitive to some people, because you imagine, "Okay. Julia is sitting there. She's got two browser windows open." How do those browser windows know about each other? Isn't each application just saying, "I am talking to Julia."

[00:28:14] JE: That's right. They don't know. It's kind of interesting, because they don't exactly know about each other. But the thing is that like when you make – Let's say you make a get request from the JavaScript. Okay. I have some JavaScript that's on some sketchy website and I try to make a get request to get – I don't know, my bank's – Just like my banking information. The browser will send that request. We'll send this request. It's like get the list of Julia's bank information and it will send my log-in cookies.

A sketchy website can send a request to my bank with my log-in cookies, because the browser is the same browser. The browser has like the cookies for my bank and it will let the sketchy website send a request to my bank with my log-in cookies, which is kind of scary.

[00:29:07] JM: Cookies here basically mean whenever you visit a website and that website decides to leave cookies, those are cookies that – It's information that that website is putting on your browser that is now accessible to any other website that has access to your cookies.

[00:29:30] JE: It's very subtle, because cookie security is really complicated. When I said that like the sketchy website has access to my cookies, it can't read the cookies itself. It can't see what's in them. It can send a request, a get request using those cookies. But the browser won't return the response, if that makes sense. There's this thing called the same origin policy. What the same origin policy kind of says is like if you send a get request, I will send the person's log-in cookies, but I will not – I will send all the cookies for whatever website you want to that server, but I won't tell you what the cookies are and I won't send you the – I won't give you the response.

Of course, this is like – This is called the same origin policy. You could see how if the browser didn't implement the same origin policy, it would be very bad, because then any website could sort of send a request to any other website using my private cookies.

[00:30:24] JM: Let's go a little bit deeper on that acronym. CORS, cross-origin resource sharing.

[00:30:30] JE: That's right.

[00:30:31] JM: Could you define the term origin?

[00:30:34] JE: Yeah, sure. An origin is HTTPS://google.com. It's like the protocol and then possibly a subdomain and a domain.

[00:30:48] JM: A subdomain being the www or the –

[00:30:53] JE: Or like mail.google.com, or whatever.

[00:30:57] JM: All the things under HTTPS://star.google.com are under the same origin.

[00:31:09] JE: No. Calendar.google.com and google.com are different origins.

[00:31:15] JM: Okay.

[00:31:16] JE: Yeah, which is I think something that often tricks people up, because they might have api.mysite.com and like mysite.com. Those are different origins.

[00:31:25] JM: Okay.

[00:31:27] JE: Yeah, according to the browser. They could be completely different domains. The browser sort of doesn't care.

[00:31:32] JM: Okay. Then CORS refers to the cross-origin resource sharing. It's basically the policy around – Can Google.com share with a sketchy website.com? Can docs.google.com share with mail.google.com?

[00:31:49] JE: Yeah, exactly. The way I kind of think about it, because often people talk about, "Oh! I can't do this because of CORS, or like CORS is stopping me from making this request." This is a little nitpicky, but it's not correct. Because actually what's happening is like the browser has this policy called the same origin policy, which is like if the origin is not the same, you can't do stuff. That applies to all origins. By default, if JavaScript from one origin trying to make a request to a different origin, it's not allowed. It is allowed under some like restricted circumstances. In general, it's not allowed.

What CORS is, is a way that allows you to actually do it. CORS is sort of a good thing that makes it possible. The same origin policy is the thing that's stopping it and that's like preventing you from making this cross-origin request. Does that make sense?

[00:32:36] JM: Conceptually. Can you explain how we create bonds between these different origins? How can we allow docs.google.com to share information with mail.google.com?

[00:32:49] JE: Yeah. What you do – Let's say you have some resource that you want to share. For example, like maybe I think in the [inaudible 00:32:57] example, like clothes.com and api.clothes.com. You're like, "Okay. Maybe you want clothes.com to be able to use like API, like api.clothes.com."

What I would do for those API request, because by default that would be not allowed, which it seems like it should be allowed. You're like, "Okay. I have some JavaScript on my website. I want to make request in my own API. Please let me do it."

What you have to do is – The place where you need to put the code to allow the request is on the resource that's being shared. If you're making another request to api.clothes.com, api.clothes.com needs to say that it's allowed, which makes sense from like a security perspective. The resource that's being secured needs to say that it's okay.

What you need to do on api.clothes.com if like that's your thing is you just add this header called access control allow origin, and you'd set that to let's say like clothes.com. If you set that header, then the browser will be like, "Oh! Okay. This request came from this origin like clothes.com." If clothes.com said that this origin is okay, because it's set the access control allow origin header to allow it, so I'm going to allow this request through, and it's fine.

[00:34:06] JM: Oh! Wonderful. Basically, you have api.clothes.com signifies that it is able to handle a request from clothes.com, and clothes.com says, "Well, I want to talk to api.clothes.com," and you basically have – I mean, that's equivalent to having proof from each one of them that they should be talking to one another in this context.

[00:34:35] JE: Yeah, exactly.

[00:34:37] JM: This illustrates an application of HTTP headers. It illustrates that you can use headers to establish relationships between different URLs, essentially, or different URL schemas, different domains.

[00:34:56] JE: Right. Also, that you have to often, right? Because it's very common I think to make cross-origin request in like modern JavaScript. Often you'll need to request something

from another domain. If you can't set those headers, you can't make your website work, right? Because the browser same origin policy won't allow it.

[00:35:18] JM: Right. Let's take this further with another example. Let's say I use a web application and I try to log in and that web application wants to ask me to two-factor authenticate. I try to log in and it says, "Well, sorry. You can log in maybe, but you have to authenticate with another factor. You need your cellphone." Then I feel my pocket buzz and I get a security code to enter, and I enter it and then I get in. Well, that web application, maybe it used Twilio.

Describe the communication between the web application and the Twilio API there, because that's the wen application using an external API. What happened in the communication between the web application and Twilio?

[00:36:06] JE: That's a great question. It's sort of interesting, because you have two choices in that choice. One choice is that the JavaScript owner page makes a request to Twilio. Then the other option is that you make a request in the JavaScript to your own server, and then your server on the server side makes a request to Twilio, because you can sort of choose whether you want to make the request on the frontend or on the backend. Does that make sense?

[00:36:33] JM: It does. Can you go a little bit deeper? Let's put this in terms of HTTP.

[00:36:38] JE: Yeah. If you're making a request on the backend, there needs to be sort of two HTTP request. So you'd make like one HTTP request to your server being like, "Hey, can you make the request to Twilio to check to see if this is okay?" Then the server would make another HTTP request to Twilio and then get a response and then maybe you send a request back being like, "Okay. Twilio said it's okay, or Twilio said it's not okay, whatever." I guess it's really like – Really what's it saying is send a text message. You'd send an HTTP request saying send a text message, and the server would send the request to Twilio to send a text message.

[00:37:10] JM: Absolutely. But let's say you've got your server and you want to communicate with Twilio servers. I presume there needs to be some HTTP headers to get these two origins to trust one another.

[00:37:25] JE: It depends. Not if you do it from your server. That's the thing, right? If you're making a request to Twilio from your server, there's no browser there. If you're just talking from your server to Twilio server. It's only if you make the request to Twilio from the JavaScript that's running in the browser that you need to do this.

[00:37:43] JM: Okay. So let's go through that use case.

[00:37:45] JE: Yeah, let's talk about that.

[00:37:46] JM: The website just talks directly to Twilio.

[00:37:49] JE: Yeah. I just want to say that I don't think that you would want to do this in this case, because if you wanted to send a request to Twilio, you would need to use your API keys, which is a private thing between you and Twilio. I think you would not want to put that in. [inaudible 00:38:08].

[00:38:09] JM: Terrible contrived example.

[00:38:10] JE: No. I think it's a great example, because I think this question of like whether you do the request on the backend or on the frontend is like – It's not an obvious question, right? It's easier to do it in the frontend in a way, because there's like less request. But there are like real reasons why that might not be a good idea.

[00:38:28] JM: You're saying that's because you would have to give your keys to the frontend?

[00:38:33] JE: Yeah, exactly. Then like anyone using your website can potentially look at that request, right? Because if I'm on a website, I can just go into like developer tools and look in the network tab and see all the requests that are being made and all the headers. If you're making a request to Twilio and your API key is in a header, I can just take it and be like, "Sweet! Now I can make a request to Twilio," using your account, which probably you don't want.

[00:38:58] JM: Right. SO then somebody can have untethered access to sending lots of text messages and stuff. I'm sure that is a common security shtick that people pull.

[00:39:08] JE: Yeah. I think it's something that's important to know if you're writing JavaScript, which is that like any request you make from your like frontend JavaScript code, anyone who uses your website can see all of those requests. It doesn't matter if they're over HTTPS. If I'm the one with – If it's my browser, I can see all the requests that it's making.

[00:39:27] JM: By the way, it's worth pointing out, because I want to cater to all levels of education at this point that the JavaScript on your page is making HTTP requests similar to how your browser is making HTTP requests when you just enter an address into the address bar.

[00:39:46] JE: Right. The interesting thing about it, the way your JavaScript makes HTTP requests is that it needs to sort of – Those are all actually run by the browser. The JavaScript sort of needs permission from the browser to make any of those requests, which is where things like the same origin policy come in, right? The JavaScript can't do whatever it wants. It can ask the browser – When you make a request from JavaScript, you're effectively asking the browser, "Hey, can you make this request for me?" Sometimes it will say no, right?

[00:40:11] JM: Beautiful. Okay. In this situation, assuming we wanted to send requests to Twilio from our browser even though we know we kind of shouldn't do this, because we're exposing our API key. What would the request look like? What would we need to put in those headers?

[00:40:29] JE: Right. When we're making the request, it's a post request. We don't have to put anything in particular in the request. The request just looks like it's a normal request. You send whatever the Twilio API says you need to do. You probably send some authentication information. You put an API key. You put a body, which is like maybe a text message you want to send. The request is sort of normal. Then Twilio on the other side will send a response.

If this works, which I guess we're hoping it does, because that's the point of this example that it's worked. Twilio will send a response with the access control allow origin header set to start saying like anyone can make a request to me. I don't care who they are. This is fine. Then the browser will see, "Okay. Twilio says this is okay from any origin." So this request is allowed.

There's actually an additional step here with those requests where you're not allowed – The browser won't even allow you to send a post request right away because sending a post request itself is already sort of a dangerous thing to do. What it will actually do, from your JavaScript, it looks like you just sent a post request, but the browser will first send an options request being like, "Am I allowed to send a post request?" Then Twilio would be like, "Yes, you are allowed to send a post request." Then you'll send the post request and Twilio will be like, "Yes, this is okay again," and then you'll get the response with those requests.

When you make a cross-origin post request, it's actually to request as an options request that's like, "Is this okay?" Then there's the real post request after. I have this whole diagram in the zine which explains this, because it's like quite the – It's quite the dance.

[00:42:04] JM: It is quite the dance. I encourage anybody listening to this that is even mildly intrigued to check out your zine. It is affordably priced at $12 and there is enterprise, I believe, based pricing. Maybe you can get your manager to pay for it for your whole team. I want to talk about the zine creative process a little bit later. But totally different question, because I feel like we addressed the Twilio thing. We can close that door.

Do all browsers treat this situation, this system of HTTP and the way that JavaScript handles HTTP? Do all browsers do it the same or do different browsers have different implementations?

[00:42:49] JE: I would say for the same origin policy, I think it's broadly the same. Definitely not everything. Browsers do, for example, support different headers differently. I really try to focus in the zine on things that are the same between every browser because most of the sort of like core important things, like how are redirects handled? How do the same origin policy work? Kind of all the same between browsers.

I haven't been paying as much attention to this as I want, but I know that Firefox has made some changes to being more stricter at how it enforces the same origin policy. One thing that I noticed is I turn on sort of like I'll look at even stricter same origin policy to disable tracking in Firefox recently. What I noticed is I was making sort of like a fun little Twitter app like just on my

computer that was like hot-linking images from twitter.com to get people's profile pictures. In Firefox I had sort of like more strict, same origin anti-tracking stuff set.

What happened was that it wouldn't get any of the Twitter profile pictures, because it's like I won't make crossover between get requests anymore like this. Because I was being more strict, it disallowed it because it was like, "This is basically the same as like embedding a tracking pixel. So I can't tell that like this image that you're embedding isn't a tracking pixel. So I'm going to block it." I actually like ended up taking that back and making my browser sort of like less secure, because I wanted to be able to embed those pictures from Twitter.com.

[00:44:18] JM: That's a great example and it totally illustrates how the implementation of these things is not some kind of objective science. It comes down to some subjective decisions about what browsers should and should not be able to do.

[00:44:35] JE: Right. I think it's really nice to give people the choice too. I think I'm going to sound like – I really love Firefox. Another thing that Firefox is doing is they have this new feature called containers, or I think it's a plugin, but I think the plugin is developed by Mozilla. Containers allow you to sort of like say, "Okay, separate out my cookies for this site from like everything else so I can have a banking container."

Then that means that there can be like sort of no contact between the cookies for like my banking website and like everything else and browsing on the internet. It's almost as if I like I was using a different browser for banking that had nothing to do with the browser I was using for like browsing sketchy websites. I think that's really nice. I think it's a nice feature.

[00:45:18] JM: Have you looked into WebAssembly very much?

[00:45:20] JE: Not too much yet. No. I want to.

[00:45:24] JM: Yeah, that makes two of us. I mean, you got to imagine, however many questions we have about browser hygiene between different browsers today. I assume WebAssembly will complicate things further. What about Electron? Have you looked at Electron much?

[00:45:40] JE: Also no.

[00:45:41] JM: Okay.

[00:45:42] JE: I fell like when I think about Electron from like I think of it as sort of a different browser. It doesn't have anything to do with like the other browsers that I'm running. I don't think it would share cookies or anything. From that perspective, I don't think.

[SPONSOR MESSAGE]

[00:46:05] JM: As a programmer, you think an object. With MongoDB, so does your database. MongoDB is the most popular document-based database built for modern application developers and the cloud area. Millions of developers use MongoDB to power the world's most innovative products and services, from crypto currency, to online gaming, IoT and more. Try Mongo DB today with Atlas, the global cloud database service that runs on AWS, Azure and Google Cloud. Configure, deploy and connect to your database in just a few minutes. Check it out at mongodb.com/atlas. That's mongodb.com/atlas.

Thank you to MongoDB for being a sponsor of Software Engineering Daily.

[INTERVIEW CONTINUED]

[00:47:01] JM: You know what's interesting about modern browsers is how – Well, I mean you see this with Chrome, Chrome being both a browser and an operating system. That's pretty profound. The fact that you think about it, it's almost like you have – On my Mac, I'm using a browser. That browser is as the functionality of an operating system, which is kind of crazy. It kind of makes you think about like this thing is as complicated as an operating – It must be as complicated as an operating system, because there's basically an operating system that is a browser out there.

[00:47:42] JE: Yeah. I mean, I don't know if that's quite true. ChromeOS, like if you have a Chromebook, my understanding is that, today, Chromebooks run Linux.

[00:47:53] JM: That's correct.

[00:47:55] JE: When I think of an operating system, I think of like device drivers and like keyboard drivers and CPU scheduling and like managing RAM and all of that. The browser isn't doing that. I think there's still like a pretty clear separation between the OS and a browser technically. The browser is definitely doing a lot, and you can have these like very complicated applications running in the browser. At the same time, if you're using a Chromebook, like DoS is still doing all the things that you would expect the OS to do.

[00:48:25] JM: But on the other hand, you open your system monitor and it's like you've got 8 instances of Chrome and Chrome helpers at the top and you're like, "Well, this thing is managing so much memory and it's like –"

[00:48:39] JE: Yeah, definitely.

[00:48:39] JM: Then you think about like, "Well, device drivers, I don't really have very many devices I'm plugging in to my computer." What would be harder? To fully integrate all the features of a browser into an operating system or to fully integrate all the features of an operating system into a browser?

[00:48:56] JE: I mean, I don't know. I think Linux is like 4 million lines of code or something.

[00:49:02] JM: Right. Fair enough.

[00:49:03] JE: It's really a lot. I'm sure Firefox is also like 4 million lines of code, but it's like different lines.

[00:49:10] JM: Yeah. Fair enough. This is a bad deviation. Tell me something about HTTP that is esoteric that you learned while writing this zine.

[00:49:18] JE: I learned about some weird status code, which honestly have evaporated from my brain. I think at some point I looked at all the different status codes, because I gave

examples of like 11, 12 in the zine. Then I looked at the list and I was like, “Okay. Cool. There are like 70.” Some of them I really had not heard of. One I think example of something I learned is that they’re the redirect status codes that are like 301 moved permanently, versus like 302 found.

By the way, the important thing to know about that is that you return at 301 moved permanently, browsers will cache it and you can’t change your mind about doing that redirect. If you aren’t sure that you want to be redirecting for this other site forever the rest of your life, it is better to do a temporary redirect [inaudible 00:50:00]. Anyway, then there’s also a 307 or a 308 which like does something different.

[00:50:06] JM: But you can always just another 301 at whatever the 301 that you’ve – I mean, can’t you just set up another redirect?

[00:50:16] JE: No. You can, but like browsers which already visited your site won’t look at it, because they will just cache the redirect and they’ll do the redirect without asking you again. [inaudible 00:50:25].

[00:50:25] JM: No. You could set a 301 wherever you’re redirecting it to, right?

[00:50:29] JE: Oh! And redirect it back.

[00:50:31] JM: But then I guess that would effectively destroy the link that your 301-ing to initially.

[00:50:35] JE: Yeah. You’d also just end up in a redirect loop forever. If you redirect somewhere and then you redirect back, it would just redirect again.

[00:50:42] JM: I’m not saying redirect back. I’m saying redirect to some third –

[00:50:46] JE: Oh, I see. Yeah. Right. Yeah, you could do that for sure. But you might also want to just go back to not having a redirect at all potentially, right?

[00:50:55] JM: In that case, I see your point. 301 can have dire consequences.

[00:50:59] JE: Yeah. It's just like sort of – You can also set a limit to be like only do this for like two days –

[00:51:07] JM: Why is it permanent? That seems crazy.

[00:51:10] JE: The name of the status code is 301 moved from [inaudible 00:51:12].

[00:51:14] JM: Oh, okay. All right. I mean, that term is used very rarely in software for the permanent term.

[00:51:21] JE: Yeah. I mean, it's definitely faster. If you're sure that you wanted to keep that forever, it means that the browser doesn't need to make that request every time, which is nice for performance. But if you're not sure, I think it's good to start out with a 302. Then maybe if it's been a while, you can upgrade it to 301 and be like, "I think I'll keep this forever."

[00:51:40] JM: What happens if you set a 301 then set a 302?

[00:51:44] JE: Well, because the thing is that the browser will have cached it. It's not going to make the request again to ask it. For new browsers, sure. What I mean by new browsers is like new people who are coming to your website who haven't visited it before. But for someone who's already visited your website, their browser isn't going to make the request again.

If they wanted to accept, they can clear their cache in their browser to say, "Okay. Forget what you remembered at this website," and just like make the request again. Yeah, you have to be careful.

[00:52:15] JM: I could see like malicious attacks where if you could – Let's say like if you hacked a competitor, a competitor's GitHub, and you use their GitHub to push out a bunch of 301s that just destroyed the routing infrastructure for all their customers. I mean, that sounds like it would just be like a permanent disaster.

[00:52:35] JE: Yeah, that'd be like catastrophic actually. That's an amazing idea.

[00:52:37] JM: Not to give anybody ideas.

[00:52:40] JE: Yeah. I've never heard of anyone doing that.

[00:52:43] JM: Please don't do that. Maybe I should remove that.

[00:52:47] JE: Yeah, maybe. I don't know. It seems really bad. Wow!

[00:52:51] JM: Please, nobody ever do that.

[00:52:53] JE: Yeah. It seems like a much worst way [inaudible 00:52:56] someone's website actually.

[00:52:59] JM: Jesus! I honestly wonder, maybe I should just remove that from – That's not like a known attack vector.

[00:53:05] JE: It's interesting.

[00:53:07] JM: All right. Well, maybe I'll leave it then. That 301 permanent thing, that sounds terrible. Can we just lobby to remove that, please? Can we move it to the blockchain somehow? Something like that.

[00:53:20] JE: Yeah. I definitely use it for my zines websites, because I decided like – At least I think I did. Yeah. No. Okay. I can tell you why it's useful to do this move permanently thing. It's because like if you – Let's say you have a URL that has some like Google juice already, something like SEO juice. If you do it [inaudible 00:53:39] move permanently, Google will transfer search juice. I don't know what you say about an SEO person to the new website. It won't do that for a 302, I believe. That's my understanding, my sort of weak understanding of SEO. There are like real reasons to do it, because Google will be like, "Okay, you've promised that this the real new website. So I'll just – Yeah."

[00:54:03] JM: Google. You're incentivizing a dangerous status code.

[00:54:09] JE: Yeah. Sometimes it's okay. I'm pretty sure I set the 301 for one of my sites, then it's fine. I was sure.

[00:54:20] JM: Of all mediums, why zines?

[00:54:24] JE: What happened, the way I learned about zines was I read this book about sort of like punk culture in the 90s called Girls to the Front. That's sort of like about riot girl. They talked about people making zines, sort of like [inaudible 00:54:40] about their personal experiences or about feminism. I thought this was really exiting. I think just the idea of like publishing something on your own that you'd like made yourself with your hands and that had something [inaudible 00:54:52] that you cared about was very exciting to me. There's this sort of like aesthetic whereas like photocopied, handmade. I really love that.

I read this book and then I was giving a conference talk at Python and I was like, "Okay. I'm going to go give this talk and I'm going to write a zine and it's going to be about something I love." I was like, "What do I love? I love strace."

I wrote this 12 or 16 pages zine about straces [inaudible 00:55:20] sharpie. I made 200 copies in it and I handed it out in my conference talk at Python. People loved it. They were like, "Wow! This is cool. Now I can learn about strace from this zine." I think part of like in that context is what I was excited about is like, "I wanted to give people something that they could take home from my talk that was like a physical item.

That wasn't just like –" Because often I [inaudible 00:55:41] people researches or links. But like people are going to forget. You have to weigh all these notes from a conference. I was like, "If I give them a zine about strace, they'll remember because they'll have it in their bag." It won't be like the other things that you forget.

That's how it started. Then I think it just really resonated with people and they were like, "Oh! This is a really good size. It's good amount of information. It's really fun. I learned about strace that I was going to learn about strace otherwise." I kept making them.

[00:56:14] JM: Zine is this very curious term. When I think Zine, I think like I have read Mad Magazine way back in the day. But your work is not really like Mad Magazine.

[00:56:27] JE: No. Not really.

[00:56:28] JM: I think of it more – It's kind of like XKCV. It's kind of like – I don't know. It reminds me a little bit of Calvin and Hobbes for some weird reason.

[00:56:37] JE: I think the thing with zine is like the form factor. It's like it's printed out. Ideally, it's like a small thing and it's something which is like self-produced and like self-published. I think that's what it is. I have like of zines on my bookshelf and they're all about different things. Some of them are about – I have one which is like a guide to safer sex. I have someone that's like about someone's experience being deaf. Some of them are like personal zines about people's experiences. Some of them are informational. Some of them are just like zines that are poems.

The thing that they have in common is that they're all sort of like small and self-published. I think often, they also have sort of a specific voice to them. You can really tell. It sort of feels like it's someone talking to you about something that they think that is important.

[00:57:27] JM: I think it also hints at – It's like a subtle insubordination thing, because computer science education in its most conventional form is like really outdated and it's like it's one of these things that it's just like the conventional university computer science education as far as I can tell. I've been removed it from it for a pretty long, but sense it's still desperately grasping for relevance.

Something like your zine is kind of insubordinate in the sense that it explains material that people would say, "Oh! You need to read the networking textbook and it needs to be this dry, colorless thing." I don't know. I like the idea of things being presented much more appealingly.

[00:58:14] JE: Yeah. I think I have now met a lot of university professors who are really doing incredible work and who are like very relevant. It's definitely all like that. But definitely – I mean, I went to a very theory-focused university. I'm like sort of [inaudible 00:58:27] focused.

[00:58:29] JM: Same.

[00:58:30] JE: Because of that, I think like the practical stuff really kind of wasn't there or like wasn't well done honestly a lot of the times.

[00:58:37] JM: Do you think that's madness? I think it's way easier to learn theory after practice.

[00:58:42] JE: For me, I loved it. I don't know.

[00:58:45] JM: You like the theory.

[00:58:47] JE: I was a pure math major. I was like I just want to do all theory all the time. Then when I graduated from the university and I started to work, I was like, "Oh, cool. This is a lot of stuff that I don't know." For example, I didn't know how HTTP worked. I didn't know a lot about networking. I didn't know about operating system. Then I just picked it up.

I feel like that worked fine for me to sort of do like practice after theory in some ways. I think what I more wanted to communicate to people is that like there's a right or wrong way to learn. You could be like practice first, theory first or whatever. But this practical stuff, how HTTP works is something that you can pick up, whatever. It's like if you don't know it yet, if you don't know like browsers uses HTTP to make requests. That's okay. You could learn today. Maybe today is the day that you learn it, and that's hard.

[00:59:43] JM: I learned it last night.

[00:59:44] JE: Yeah, exactly. That's like fine. Because so many people – I feel like also there're all kinds of things I don't know and I've been successfully doing my work without knowing all kinds of probably basic things. I think we're all kind of out there doing our work without knowing lots of things. I feel like what I want to accomplish with my zines is just like, "Hey, cool. Here's this thing you could know more about." Maybe today is the day you learn about it.

[01:00:10] JM: You outsourced the illustration?

[01:00:12] JE: I outsource the covers. I've been illustrating the covers. I do everything else myself.

[01:00:17] JM: Have you thought about scaling up your creative output by doing more outsourcing? Because I think this is one of the cool kind of nascent opportunities in the kind of freelancing gig economy, knowledge work world, the idea of kind of paying creative people to work with you in kind of a reasonably priced fashion?

[01:00:44] JE: Yeah. I love hiring. The illustrators that I've worked with are all incredible, and every time I work with them, "Oh my God! I can pay this person to do this really cool work for me," and I could never do this. It's like – Because I'm really not that good of an artist. It's like not – It's just a fact. It's so fun. I think that to me the hard thing about writing "content" is the ideas in how to present them.

[01:01:14] JM: Right. That's like your core competency. It almost makes more sense for you to spend as much time as possible doing that and to outsource literally everything else.

[01:01:24] JE: But I don't know. That's like most of what I do – That's what I spend my time on.

[01:01:29] JM: Oh, I believe you.

[01:01:31] JE: I don't feel like there are a lot of other things that I spend my time on other than like figuring out what to write for anything. Because in a way, one thing that I like about the zine format also is that it's quite short. Writing down all the words doesn't actually take that much time, because I focus so much on like how can I explain this in like as concise and like easy to understand way as possible? A lot of these pages have like – It's like a hundred words on them. Yeah.

[01:01:58] JM: Just a few more questions. I noticed that most of your examples use Netlify. What is that your path of choice?

[01:02:04] JE: I like Netlify. Why do I use it? I don't know. I switched to Netlify. Netlify is kind of like GitHub pages. It's better than GitHub pages in the sense that like I use Hugo to generate – All my websites are static sites, and GitHub doesn't support Hugo, and Netlify does support Hugo.

Also, if I'm generating a website with node Netlify – Netlify I guess supports like more complicated build processes for static sites, which is why I use it. Also, in some sense, I could just do the building on my laptop and have like a script that I run that builds it on my laptop and then pushing the HTTML and then run it like literally anywhere else, which is I used to do before I used Netlify, is I just had a script that builds it and then rsync the HTTML over to my webhost. Yeah. I think – They have a bandwidth limit, which is pretty low, which I feel like is my main issue with it. I put a CDN in front of Netlify, because otherwise they charge a lot of download I think.

[01:03:12] JM: Okay. Last question. You've basically gone from – You used to work at Stripe. You're a fairly early employee at Stripe. I know you loved the company. You're passionate about it, but you basically became an "indie hacker", or kind of an indie hacker. Something between indie hacker and I guess just like artist, or a zine author, self-funded individual? What has the psychological shift been like?

[01:03:47] JE: I've been on my own, I guess, for like 2 months. It's very new. I think it's a pretty big transition. When I started working for Stripe, I switched to working remote, which was also a really big transition like from working in an office to working remotely. People who are like across the country. I think of it as sort of like the same scale of transition, if that makes sense.

Yeah, I don't have a lot of like hot takes. I'm like, "This is really different." I am learning about how to think about it. We'll see. But it's been fun so far. The first thing I did was I published this HTTP zine, which is really great and I'm really happy with it.

[01:04:34] JM: I certainly found it useful. Certainly worth $12 to me.

[01:04:38] JE: I'm really happy. I'm always really delighted when people tell me that they learned something from my zines.

[01:04:44] JM: Absolutely. Julia, thanks for coming back on the show. It's been really fun talking.

[01:04:48] JE: Thank you.

[END OF INTERVIEW]

[01:04:58] JM: If you want to extract value from your data, it can be difficult especially for nontechnical, non-analyst users. As software builders, you have this unique opportunity to unlock the value of your data to users through your product or your service.

Jaspersoft offers embeddable reports, dashboards and data visualizations that developers love. Give your users intuitive access to data in the ideal place for them to take action within your application. To check out a sample application with embedded analytics, go to softwareengineeringdaily.com/jaspersoft. You can find out how easy it is to embed reporting and analytics into your application. Jaspersoft is great for admin dashboards or for helping your customers make data-driven decisions within your product, because it's not just your company that wants analytics. It's also your customers.

In an upcoming episode of Software Engineering Daily, we will talk to TIBCO about visualizing data inside apps based on modern frontend libraries like React, Angular, and VueJS. In the meantime, check out Jaspersoft for yourself at softwareengineering.com/jaspersoft.

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

[END]