EPISODE 1015

[INTRODUCTION]

[00:00:00] JM: Slack is a messaging application with millions of users. The desktop application is an electronic app, which is effectively a web browser dedicated to running Slack. This frontend is built with ReactJS and other JavaScript code and the application is incredibly smooth and reliable despite its complexity.

When a user boots up Slack, the application needs to figure out what data to fetch and where to fetch it from. Companies which use Slack heavily have thousands of messages in their history and Slack needs to determine which of those should be pulled into the client. There are profile images, and logos, and custom emojis all of which are used to define the user's custom workspace experience.

Anuj Nair joined Slack in late 2017 and the years since he has been with the company, Anuj helped write the Slack frontend client, including work on the boot up experience, the caching infrastructure and the role of service workers. Anuj joins the show to discuss his work on the Slack frontend architecture and the canonical view layer problems that Slack faces. This was a fantastic episode about frontend infrastructure and I learned a lot of new information about how Slack works and how to build a high-quality desktop web application.

[SPONSOR MESSAGE]

[00:01:25] JM: Logi Analytics believes that any product manager should be successful. Every developer should be proud of their creation and they believe that every application should provide users with value. Applications are the face of your business today and it’s how people interact with you. Logi can help you provide your users the best experience possible. Logi’s embedded analytics platform makes it possible to create, update and brand your analytics so that they seamlessly integrate with your application.

Visit logianalytics.com/sedaily and see what's possible with Logi today. You can use Logi to create dashboards that fit into your application and give your users the analytics that they're looking for. Go to logianalytics.com/sedaily.

[INTERVIEW]

[00:02:23] JM: Anuj Nair, welcome to Software Engineering Daily.

[00:02:25] AN: Thank you. Thank you so much for having me.

[00:02:27] JM: You joined Slack in late 2017. What were the biggest pain points in the frontend performance at that time?

[00:02:34] AN: Oh! That's a really interesting question. When Slack initially came out of beta. I think it came out of beta around February 2014. The entire idea behind engineering and product at the time was just find market fit and make sure that the users who are using it were happy with it. The customers who are paying for it were happy with it. So all of the engineering practices overtime really just developed kind of naturally and what was the fastest to do.

Slack at the time was incredibly fast growing, like incredibly fast growing. So there wasn't always time for best practices. I think the mantra at the time very much was do what you have to do today to make things work and then move on to the next task, just because there was so much to do.

So around 2017 came point as to where the current frontend infrastructure that we had, all of our build infrastructure, things like that, just went scaling to the point at which Slack really needed. So one of the racks that was put out was for one of my roles, which was frontend infrastructure at the time, and it was all based around trying to make infrastructure and architecture for Slack the product and for all of the external sites that you might use to administer Slack and things like that. So I came in and helped to really try and build that up, things like webpack processes, and build processes, and best developer practices, how to monitor our frontend performance, things like that.

[00:03:59] JM: There are clients on mobile, on OSX, on Windows, on desktop web, and I know the clients vary across these different surfaces. Which of the clients did your work touch?

[00:04:15] AN: My work touches all of the desktop clients. That's anything on Windows, on Mac, or on Linux. We use Electron to essentially make our product platform agnostic. It's a great product where you essentially use web technologies, but you can create desktop applications, which is very powerful. We code in JavaScript and CSS, HTML, all of those things as well. We use Hack on the backend. Then we can bundle it up in an Electron executable and ship it off to customers.

[00:04:46] JM: Yes. When I boot up Slack, it loads this Electron desktop app. this is built from React components and JavaScript. When I load Slack, there is a ton of data that could potentially load for my Slack application.

[00:05:01] AN: Absolutely.

[00:05:01] JM: I've got three years' worth of chat messages between me and different people in my and my channels and I have a small company, so there's not much data relative to a bigger company. So there's a ton of data that could potentially load, but you want the boot up time to be low. Obviously, you don't need to load everything. You don't need to go all the way back in history, but there is this essential tradeoff between how much data you want to load and the performance. Can you describe the tradeoffs between loading the Slack application quickly versus loading all the data you could potentially need?

[00:05:37] AN: Yeah. It's really interesting, because like you say, for companies who have been using Slack since the beginning, they have gigabytes and gigabytes, if not terabytes of data all stored on the Slack server. You really have to decide what you want to load and when you want to load it. Very common for web applications is to only load the things which are currently in view. You only load the JavaScript, the CSS, which is in the viewport at the time.

Slack does something similar and that it only loads the JavaScript and CSS it needs for the page. But for the most part, that is the majority of the Slack application, because once you open the Slack app, you've got everything there and you're going to see – There's no scrolling, really, unless it’s in a message pane. The only real difference there becomes how much data do we load?

We have like a prioritization model of how much data we load and when we load it. We call it frecency, which is a mix between how recently it's been used and how frequently it's used. That's all calculated on the backend, using like AI and machine learning and whatnot. Then when you, an individual, loads Slack, we can try and figure out what you might be most interested in seeing really quickly and load that instantly. Then all of the data which you might need eventually comes in the background overtime during idle time and things like that.

[00:06:59] JM: Does the fact that this is an Electron app rather than an actual application in the browser, does that have any impact? Does that matter or does the Electron app basically function the same way that it would function if it was just a browser application?

[00:07:13] AN: Pretty much functions in exactly the same way. Electron allows us closer tie-ins to desktop APIs, so things like if we wanted to interact with Siri or something like that, for example, you can't necessarily do that in a browser, but you can do that using max native bindings and things like that. That's pretty interesting. For the most part, Electron, a node wrapper, essentially allows us to interact with the desktop as if it's a normal desktop application.

[00:07:45] JM: As we get to where Slack is today in terms of its frontend, and we talk about what has happened since you joined the company, there was a rewrite of the client. Can you explain what the motivation was for Slack rewriting its desktop client?

[00:08:03] AN: Yeah, absolutely. This rewrite started actually around when I joined. People were kind of prototyping at the time and a very small subset of frontend engineers, there's like three or four of them at the time, essentially thought to themselves like the frontend infrastructure we have at the moment isn't scaling. Let's completely rethink all of the principles that we initially started Slack with and see if we can make something better.

Two years ago, React very popular. It's a proven model. So many companies are using it and it’s extremely powerful for data-driven applications. Slack is a hugely data-driven application. It made complete sense to move over to a React front-end. Previously we were using templating languages, like think it was Smarty or Handlebars or something like that.

Then React ties really nicely with Redux. So we started exploring Redox as well. One of the very first parts of Slack, which we actually prototyped in this new setup, was the emoji picker. Customers or users are allowed to upload their own emoji and some of our customers have tens of thousands of custom emoji uploaded. It’s like crazy numbers. So we thought this was a really good stress test to see how this would work in our system, but also really quite nice the isolated. We created it. We deployed it. It worked really nicely. We’re like, “Okay. Let's make this a big project. It's a proven thing now. Let's rewrite the client.”

To loop back to your actual question, what were the motivations? The first and foremost was unlocking new features in the Slack app. There’re only specific features which you can enable when all of your components are renderings in exactly the same way or all of your data is accessed in exactly the same way. So things like Dark Mode. It's a really interesting concept because for it to work unanimously across the entire application, every single component which is built has to know how to switch between different themes, right?

Previously when we’re using templating languages, Handlebars, Smarty, things like that, well, if you want to update the DOM, say you open the emoji picker, there are multiple ways to do that. You could interact with the DOM in place and open it. You could create a sub-DOM and then build it and then attach it into your main DOM. You could have hidden parts of the DOM already there and then just unhide them. There’s so many different ways you can interact with the DOM there, but React only allows you to update the DOM in one way. Redux only allows you to access data in one way. So that's a perfect match for features like Dark Mode where you say, “Okay, I want to set my theme as dark mode and then all of the components which are built in exactly the same way know how to render a dark mode version of that component.” Features, that was first one.

The second one was performance, for sure, because Slack had been built in a very kind of like natural way and whatever needed to be done was done. A lot of the principles had just developed over time. At the at the time Slack was built, we kind of sent – Whenever a user booted Slack, we would send them a payload of everything they might need to interact with

Slack, like who are the users in your workspace? What channels are you a member of? What files are there? Things like that. Then we would connect to a WebSocket and then get real-time data, things which came over that WebSocket as you were using Slack.

But you can imagine, like some of our largest customers have hundreds of thousands of uses and hundreds and hundreds of thousands of channels in a single workspace. That becomes really unmanageable. Just think about the database queries which might have to go through to try and get 100,000 users send up or down to a user who's booting Slack. That's a huge amount. This rewrite really allowed us to rethink this principle of sending this large payload down every time someone booted Slack, and instead what we do conditionally fetch everything now.

What we do is we kind of send messages, like messages are the main bulk of Slack. In a message, if we say, "Okay, Jeff is in my message. I mentioned Jeff. I need Jeff's information now. Conditionally go and grab Jeff's information and send it back to me."

That turns our entire principle on its head. Instead of sending everything, we sent down the minimum amounts and then requested the data we needed to fill in all of the different missing parts. So performance, that was a massive win for us.

There were a few other reasons as well, things like developer efficiencies and developer performances. If everyone's developing in the same way, it becomes very easy to move teams, to help debug issues.

[00:12:55] JM: You mean all the different clients.

[00:12:56] AN: All the different components. All of the different – Like iOS, Android are built in those different languages. The Electron app is built in web languages. But we have multiple products at Slack, things like – Most people are familiar with the main chat application, but we also have a calls feature where you can call face-to-face on Slack, or there is a documents feature where you can collaboratively edit a document to put on to Slack. Our latest one actually, a big one at the moment, is the workflow builder, where you can automate tasks in Slack just by clicking or starting actions and things like that.

We have teams who are dedicated to building these products, but at the time, previously, before we rewrote the client, they were all built in different ways or architected in different ways. Their setups, their deployments, everything like that were all very different. So we standardized everything across the board to make sure that every single Slack application, which is built internally, boots in the same way. It uses the same configs, the same ESLint and prettiest settings, all of these kind of things because it allows developers to jump around projects if they want to to help debug other people's issues, and it's actually made off a frontend team extremely close because even though I work on a very specific team on very specific things, if someone has an issue in a part of the developer build pipeline, which I help maintain, I can now jump in and understand all of their code and why it might be breaking or have an issue or something like that.

[00:14:38] JM: Now, you're not talking about the actual client code being the same. You're not saying like the iOS app works the same as the web app in the sense that like it's all React Native or something. You're more talking about the boot up time, the data interchange time, like the networking and the communication between the frontend and the backend.

[00:15:02] AN: Yeah, exactly. I'm talking more about the architecture, the frontend architecture of how Slack clients are built. They all boot in the same way. They all bootstrap themselves in the same way. They all set up Redux in the same way render React in the same way and things like that.

[00:15:21] JM: But are you saying that the mobile applications use Redux?

[00:15:26] AN: No. Sorry. I might be explaining this poorly. When I talk about Slack applications, I'm talking not necessarily about the apps that you install into your workspace. I'm talking more about the products that Slack builds.

[00:15:41] JM: Right. You're saying like there are people at Slack who work on the documents team. There are people at Slack that work on the work flows team. There are people at Slack that work on the calls team. These different teams need to interact with, for example, CDN infrastructure. They need their React components or their frontend components to consume

CDN resources in a similar way. They need to – You have this notion of cold startup versus warm start up where somebody's loading a Slack client on a computer for the first time, that client is going to load differently than if slack has loaded on the computer before across all the different Slack components, the Slack products. You want there to be a similar experience of data interchange.

[00:16:26] AN: Exactly. That's exactly right. It's things like the same webpack config file builds all of those apps and they all use ES modules and things like that. It's more the principles and programming paradigms which are the same between all of these products, when before they were very different.

If the course team had something they wanted to debug, it's very difficult for a docs frontend engineer to jump into that codebase and try and help them out if they needed that extra help.

[00:16:59] JM: Talking about before that rewrite, can you just give us some more context on in the period where everybody had gone down different paths, like the calls team was doing something differently than the core chat team, which was doing something differently than the documents team. When everybody had gone down their own path, what were the consequences to the overall engineering work? How did that actually – That divergence penalize the overall architecture?

[00:17:27] AN: That's a great question. It's not necessarily penalizing the architecture. It was more penalizing the speed at which we could develop new features and help each other to debug different issues.

Now that we are all essentially speaking a common language with our config files and things like that, it's extremely easy to jump around different repos and codebases and just start diving in and developing the things. It's small things, but imagine if you're in one repo and you're coding and your line lengths are set to 100 characters and you have trailing commas on arrays and things like that and then you jump to another codebase and that's not necessarily the case. Well, that's going to slow you down, because your [inaudible 00:18:16] are going to complain, things like that. It's going to slow down the rate of development.

When you remove these like small nuances and just get everyone on a common playing field, things just work and it's just magic and you can just churn out things really fast and develop things in a really efficient way. Those are like the smallest of examples. But the larger examples were things like deployment and how the different products communicated with each other, things like that.

[00:18:46] JM: How has the usage of CDN infrastructure changed since you rewrote the Slack architecture?

[00:18:54] AN: Not too much. Before we rewrote all of the different products, we were still using our CDN in a very similar way. We were bundling all of our assets up. It’s a very, very simple bundler, homegrown, and uploading them to the CDN. Then whenever you boot the app, the app knows which resources to pull down from the CDN and start using them. But when we rewrote the clients, we rewrote it using webpack ES modules, all of those good things, and we still uploaded them in the same way. But as we started introducing new features, especially ones which pulled down from the CDN more often, we eventually found that our CDN class started rising just because we have 12 million – I think it’s like daily active users, which is huge. So if all of them are consistently pulling down items from the CDN, that adds up really quickly.

One of the things that we changed when we rewrote the client was that we tried making things more lazily load so that we didn't pull down as much data from the client. We introduced service workers so that we could start caching data resources locally on the user’s machine and reuse those. So things like that. When you have a huge scale of users, all of these tiny little things start adding up to your bottom line on how effective it is.

[SPONSOR MESSAGE]

[00:20:26] JM: When you need to focus on building software, you don't want to get bogged down by your database. MongoDB is an intuitive, flexible document database that lets you get to building. MongoDB's document model is a natural way to represent data so that you can focus on what matters. MongoDB Atlas is the best way to use MongoDB from the company that creates MongoDB. It's a global cloud database service that gives you all the developer productivity of MongoDB plus the added simplicity of a fully managed database service. You can get started for free with MongoDB Atlas by going to mongodb.com/atlas.

Thank you to MongoDB, and you can get started for free by going to mongodb.com/atlas.

[INTERVIEW CONTINUED]

[00:21:19] JM: I want to talk a little bit more about the boot up of Slack. When Slack starts up on your computer, to boot up time is going to depend on a number of things. If you've loaded a Slack client on your computer before and I guess how much data for your session is stored in the CDN. Also, I guess it’s worth asking, how much you data can you actually store on the user’s computer? When I load my Slack client, is it taking any data from disk or is it going over the network for all of its data?

[00:21:59] AN: We have a couple of concepts when you boot Slack like you mentioned. This concept of a cold boot and a warm boot. We've defined a cold boot to be when you're booting Slack for the very first time and nothing exists. The only thing that we know when you're booting Slack for that very first time is who you are, because you will have to have signed into your account and what team you're trying to boot into.

As soon as we've authenticated that, we then have to go and fetch absolutely everything to boot Slack, from resources, like JavaScript, CSS, fonts, sounds. All the way down to images. All the way down to all of the data, which then fills the Slack client. There are so much stuff that we have to download initially. But for most web applications, that download time and even parsing and executing all of that takes a long time, and download is usually the longest part of that boot up phase.

We introduced service workers, and that introduced the concept of what we call a warm boot. With service workers, we’re able to intercept requests and we can actually store specific responses from the CDN on the user’s machine. When you initially boot Slack in a cold boot, we go and fetch all of that data from the CDN and then we store it via the service worker on the user’s machine in something called a cache storage.

When the user then comes to reboot Slack, if they refresh Slack or they close it down and reopen it, we first check if the assets we need are on the user's machine. If they are, we can just completely skip the network phase and reuse those assets instead. It's extremely powerful, because it cuts out one of the longest phases of booting an application, the network phase.

Part of the rewrite that we did, we actually moved as much resource and data as possible on to edge caches. So not just like a CDN, but also we have a – I guess it's like a homegrown data CDN, which we call flannel. Essentially stores the user's data in a secure way on a network edge instead of it all coming from a server on like the US East Coast or something like that. By serving everything from the edge, we can speed up network responses, but we can speed up boot even more by storing all of that data on the user's machine as well.

[00:24:33] JM: I want to get into a lot of those details that you just mentioned, but one of the reasons we're doing this shows is because I was curious about Slack's usage of service workers. Can you explain what a service worker is?

[00:24:48] AN: Yeah. A service worker is – I guess it's like a proxy between your browser and your server. It works by intercepting as soon as it's been installed. It works by intercepting all network requests and then allows you to do essentially whatever you want to do with them. You can replace them. You can redirect them. You can drop requests if you want. You can do absolutely anything.

But a very common use case for service workers is to intercept the responses and store them on disk and to intercept the requests and then check if you have that request on disk or not. If so, we serve it from disk.

[00:25:35] JM: My understanding is that a service worker is a custom piece of JavaScript that is going to run inside your browser in a separate thread than the main application loading. You can set custom logic within that service worker to respond to, for example, different conditions of how those resources might be loading.

[00:25:59] AN: Yeah. That's exactly right.

[00:26:01] JM: But I guess if your core application needs to load a certain JPEG, if that JPEG is on your desktop, if it's on your disk somewhere, then you would much rather fetch it from disk than go all the way to the server.

[00:26:16] AN: Yeah, that's exactly right.

[00:26:17] JM: So how does a service worker script get on to my browser? How does it get installed?

[00:26:24] AN: Sure. You have to install a service worker from your main application. I think it's called like navigator.serviceworkerinstall or something like that, but you essentially provide your main application a path to where your service worker is located on your CDN or your server. Then browser APIs will go and fetch it and install it in a separate thread.

The interesting thing about service workers though is that they don't take over what's called controlling the page until subsequent requests start occurring. When you first boot Slack, for instance, we tell a browse API to go and fetch the service worker, install it, but you’re technically not using that service worker until you refresh Slack or you close Slack and open it again.

There's a good reason for that. That's because service workers, they want to make sure they are interacting with a compatible version of your code. If you install a service worker, you might have specific logic saying, “Oh! Still go and fetch all of my assets, or my day, or whatever it might be,” which isn't compatible with that specific version of the service worker.” This becomes more apparent when you start making changes to the actual service worker code itself.

If service worker V1 is incompatible with V2, when V2 installs, you don't want it to start controlling the page straightaway because then you’re going to start having errors with the current running code. Service worker technologies work by waiting until all – It’s called active clients. Close down before activating the new service worker which it has downloaded and installed. Once all of those clients have closed down and then you reopen your website, that new version of the service worker can take effect and start controlling it and doing whatever you've told it to do.

[00:28:30] JM: Tell me more about how a service worker interacts with HTML elements.

[00:28:34] AN: It doesn't interact with HTML or the DOM per se. It's a really interesting concept, because as web developers, we usually only have to think of a single thread in JavaScript, single-threaded, and only for a specifi c runtime. But service workers run in a completely independent thread and the scope as to what they can access is much more reduced than what the main thread can access.

Service workers really can only interact with fetches to your APIs or your CDNs or whatever it might be. A service worker has three main events, which you can tell it what to do. The events are called install, activate and fetch. The install event is triggered when the browser detects that a new version of the service worker is available. You've changed the service worker code, and the browser does that automatically for you. Every time you load your site, it does a byte comparison check of the current service worker running and the new one where you've told the service work lives. If they're different, then it would download that new version and run you install event.

At that point, you usually tell it what assets to pre-cache or what you want to store on disk or anything like that. When a service worker gets to activate itself, i.e. all of those running clients were closed and then reopened, the service worker calls the activate event, and that's when it starts taking over and controlling the page which it's been registered on.

At that point, at Slack at least, we usually clean up old assets which have downloaded to disk from previous service workers. They're no longer useful anymore. So we clean all of those up. The new service worker and its install event downloaded all of those new assets it needed. So the activate event for us is all about cleaning up old stuff.

Then that's it. The service worker is essentially running in its own thread now. So every time you write a fetch events or request a resource from the network, the service worker will intercept it and you get to decide what you want to do with that request, like we said, modify it, drop it, reply with something else, respond with something else, whatever you want to do, and it continues to do that for ever.

Every single network request will go through your service worker and your service worker will run indefinitely, and that is really interesting, because it creates a whole new level of thinking for us. We need to now start thinking about our asset lifecycle and our service worker lifecycle and error recovery and things like that. There's a whole new vector of things that you have to start thinking about as soon as you start using service workers.

[00:31:29] JM: Can you summarize a few basic example use cases where a service worker could help me build a web application? Whether you're talking about Slack or something outside of Slack.

[00:31:41] AN: For sure. Yeah. At Slack, we use service workers for two main things. The first one is performance. Like we've said, if you store all of your assets on the user's disk, you can cut out the network phase of trying to fetch those assets. That's usually the majority of trying to download an asset. So that's huge performance.

The second one is actually unlocking new features for us. You can actually use a subset of Slack offline now because everything is available, which is needed to boot Slack on the user's disk. If you turn off your Wi-Fi, if you're in a plane or something like that and you open your laptop and you refresh Slack, Slack will still boot, because it doesn't need anything from the network to boot itself now, which is phenomenal to think about, because websites – Of course, the idea behind all of them is you need to get data from the server.

If we're storing all assets and all data on the user's disk, which they might need, they can use Slack offline. That's great for the people who are commuting or on planes. They can catch up on old messages. They can write drafts. They can start different things. So the idea is that when Slack then detects that it has Wi-Fi connection again, it's on the Internet, it can re-sync itself with the server to catch up and send all of your queued messages over as well.

[00:33:11] JM: The service worker is doing the negotiation around resources, basically switching on is this computer connected to the Internet or not.

[00:33:23] AN: There're browser APIs to help tell you whether a website is connected to the Internet or not. I think it's called navigator.online, and that returns a Boolean, are you connected to Wi-Fi or not? The service worker just acts as a proxy for that if it receives a request where it has that response on disk, it would just cut out the network and just short-circuit it and respond straightaway with that.

[00:33:49] JM: Okay. That's great description of service workers. One other question, if we're just talking about dealing with a desktop application, there is some unpredictability of networking. Your network connection might be flaky. How does service workers help with the unpredictability, the potential of drop packets and so on that might go on in networking?

[00:34:15] AN: I'm going to expand on your question even more, and that Slack is considered a desktop application, but it is written with web technologies as a website. You can access Slack in your browser by going to slack.com/ your team idea or whatever it might be. Certain assumptions come with that from people who are using Slack, which don't usually come from when you're using a website. It's things like what happens with drop packets or what happens if my CDN fails for a specific request and I don't the response I was expecting back? There are lots of assumptions you have about desktop applications that you don't necessarily have about websites.

Service workers have certainly helped us deal with that and that to the end user, dropped packets or failed requests to the network just go unnoticed because sometimes we're able to reserve those assets back from the service worker. Like I mentioned before, we try code splitting large parts of the Slack app where we know that the user isn't going to use that part of Slack until they actually start interacting with it.

For example, when you go into your preferences, we don't actually download the code, the JavaScript and the CSS to show you the preferences pane upfront when you initially boot Slack. You only need it when you actually click on the preferences button to open it, right? But we download all of those assets into the service workers cache, into the cache storage, when the service worker initially loads or installs itself.

When you go to open the preferences, it's almost instantaneous the preferences pane loading, because we don't have to wait for the network to go and download all of those things and then open it up again. So that's great, because for an application, Slack might be 200 MB, 300 MB on a user’s machine if it was installed and coded as an actual desktop application. But it's not. It's in a web languages. We initially download maybe a meg of resources, JavaScript CSS, all of that kind of stuff upfront. We download the rest of it in the background and then start using it via the service worker, requested by the service worker when it's needed.

[00:36:38] JM: Let's talk about more aspects of the frontend. What role does webpack play at Slack?

[00:36:45] AN: So webpack is an asset bundler. Modern JavaScript applications are usually written in ES modules using this import-export syntax and everything is very modulized. It's kind of broken down into individual files and everything is hopefully very pure functions, easily testable, things like that. Easy to navigate when you're a developed in your repo.

But Slack, for instance, we have thousands and thousands of JavaScript modules and we have imports and exports everywhere, but that doesn't necessarily play well with what a user would want when they start using Slack. A user doesn't want to have to download 10,000 JavaScript files to start using Slack. You want to download 5 or however you've code split your application, because the browser just parses it that way before HTTP2 where everything was very synchronously downloaded. Trying to wait for 8,000 files to download would take seconds, if not minutes, which would be crazy.

Webpack is a piece of technology which essentially takes all of these ES modules imports-exports and turns it into a few JavaScript files where all of those things have been concatenated and optimized for that specific file that all of those ES modules end up in. It’s a very powerful, very configurable piece of technology we at Slack use extremely heavily. It powers all of our development workflows. It helps us to deploy code and it helps optimize all of our code for us as well.

[00:38:23] JM: Where does it fit into the build process for Slack?

[00:38:28] AN: Yeah. At Slack we have a continuous integration deployment model. When you open a PR, we have hundreds of CI jobs which just start running to test that your PR isn’t going to break anything or cause any sort of regressions. A part of those things – A part of those tests are end-to-end tests. To do an end-to-end test, you have to build the application with the new changes that you've put into it. Webpack does that for us. It's triggered in a Jenkins box somewhere. It builds the Slack app. It's deployed to a QA environment, which is like a production-like environment for us. Then we run automated tests against it to make sure that nothing has regressed in the changes that your PR has introduced.

If that's all good and done, then we get someone to review the code. Every single PR has to be reviewed as Slack by another developer. I believe that's for security and audit reasons as well. Then if that's all good, then it's merged into master and it's automatically put on to the next deployment train and its staged and pushed out to production. During staging, again, we'll run webpack with all of the PRs, which were recently merged to master to build those production assets and then get them out.

[00:39:46] JM: In what ways has webpack performance been an issue in the past?

[00:39:50] AN: It's interesting. When Slack, the company, initially started, the deployment process was extremely simple, actually scarily simple, and that files were literally just concatenated together and then uploaded to the CDN. That was the asset that you would download. Some of these files ended up growing to like tens of megabytes large because there was no cleverness around how they will optimize or anything like that.

Webpack is very powerful because it can perform those optimizations like tree shaking, i.e. getting rid of code which isn't used anywhere. Compressing and minifying all of your code before it's then uploaded to the CDN. That can be orders of magnitude in the file sizes from just a simple concatenation to doing all the smarts that webpack does. But there is a tradeoff, and that is build time.

A simple file concatenation which is what Slack had previously takes milliseconds. It's just opening files, concatenating them together, writing them somewhere, uploading and you done. But webpack has to go through this entire build process. It has to create an AST of all of you imports and exports. It has to figure out where the optimizations are. It has to run through compression algorithms. All of these kind of things. That takes a lot longer. When we initially introduced webpack, a lot of developers struggled going from waiting seconds for deployment, to waiting minutes for deployment. A good Slack deployment at the moment where everything is build and then put onto staging I think on average takes just over five minutes, which is extremely fast for a webpack deployment.

I've heard of companies who have to build – Use webpack as well and some of their build times are 45 minutes long and take like 50 gigabytes of node memory to try and build it all, which is madness.

[00:41:54] JM: That long period of time, is that due to the fact that webpack has to crawl the source code and build a dependency graph and understand how to reduce that dependency graph and apply like style sheets and stuff to get the resources to what their final shape and size would be?

[00:42:19] AN: Yeah, that's exactly right. Because web applications are large, especially at enterprise companies. They have to traverse all that code. They have to build ASTs. They have to apply optimizations. They have to turn it back into source code, creates source maps for all of your code, things like that. It's an extremely expensive operation. But the benefits to the end user are huge. They have to download a lot less code. They get optimized codes, which loads much faster in their browser or in their application. There are certainly tradeoffs.

When we initially introduced webpack at Slack, we spent a long time trying to optimize our build time and reducing that as much as possible to make sure that our end-users were getting the best results, but our developers and our deployment process was also getting the best result possible.

We've developed a lot of internal plug-ins. We've contributed back to the webpack source code to try and really optimize it as much as possible, and we're in close contact with the what main webpack contributors as well to try and see how we can help them how they can help us optimize things further.

[SPONSOR MESSAGE]

[00:43:38] JM: DigitalOcean makes infrastructure simple. I continue to use DigitalOcean because of the low friction and attention to user experience. DigitalOcean has kept the experience simple and I can spin up a server in less than a minute and get high quality performance for a low price. For an application that needs to scale, DigitalOcean has CPU optimized droplets, memory optimized droplets, managed databases, managed Kubernetes and many more products. DigitalOcean has the flexibility to choose the right instance for the right workload and he could mix-and-match different configurations of CPU and RAM.

If you get stuck, DigitalOcean has thousands of high-quality tutorials, responsive Q&A forums and a customer team who treats customers respectfully. DigitalOcean lets developers focus on what they are building. Visit do.co/sedaily and receive $100 in credit over 60 days. That $100 can be put towards hosting or infrastructure and that includes managed databases, a managed Kubernetes service and more. If you want to get started with Kubernetes, DigitalOcean is a great place to go. You can use your $100 to start building your distributed system and you can get that $100 in credit for free at do.co/sedaily.

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

[INTERVIEW CONTINUED]

[00:45:14] JM: I want to play the humility card here because there was an episode recently where my naïveté when it comes to frontend was revealed where I didn't really understands what webpack was and what it did in practical terms. I know that if you go back 5, 8 years, frontend development, it's like if you want to render frontend application, you send to the user like a CSS file, like an HTML file and some JavaScript and everything gets rendered by the browser, by the client's browser. If your client – Your customer who's downloading your website is having to render all that stuff in their browser, it might render really slowly and they might have a lot of expensive assets that they have to load and there's a lot of network requests.

You fast-forward to today where you can do a lot of server-side rendering and pull in assets into the webpage or pre-cache assets, put them at the CDN or something. If you do a lot of the rendering of the JavaScript frameworks, like React or Vue or whatever and you get the style sheets applied on the server side, then you can push out the resources in a much more fully developed page loading for the user. I guess I'm looking for confirmation that I understand this correctly. That this is actually what webpack does.

[00:46:45] AN: It is. Yeah, that's right. I think what webpack really excels in is helping with the developer experience. Previously like you mentioned, when you were deploying websites, you had to write you HTML page. Then in a separate file, write your JavaScript. If you want it to interact with and element, apply an ID to a specific HTML element and then grab it with JavaScript and start interacting with it in HTML.

There's a disconnect there, because they're not joined together in an explicit way. It's an implicit connection between the two. That's the same for CSS, right? In your HTML, you add a class with whatever class name you want, and then in a CSS file you target that class and apply your different styles to it. Again, an implicit connection between the two.

Where the frontend wowed us today is that all of these things are a lot closer tied together. Instead of writing three separate files now, HTML, CSS, JavaScript, it's actually more combined into one. It's not like a mix of one file doesn't have a section of HTML and a section of JavaScript and a section of CSS, because you'd still have that implicit dependency between them instead of an explicit one.

Instead, the languages which have developed, a language which you may have heard called GSX. It essentially mixes all of those things together to create explicit dependencies between them. I can write HTML code and then in line in that HTML code loop over a loop to create a list of items. Then in that list of items, I can specifically say what styling it should have. All of those things are very explicit. But the browser doesn't understand that. The browser is still I want an HTML file, the JavaScript file, the CSS file. That's where webpack helps us. It essentially takes this conglomerate GSX which we've created and all of the conglomerate GSX files that we've created, it creates this dependency tree between them all to create a single file of GSX. Then it's transforms them into the languages, the native languages which the browser understands. It strips out the JavaScript and puts it in the JavaScript file. It strips out the CSS and puts it in a CSS file. Then the browser knows how to load those things. Just like the old days, the good old days which we had, the browser can download those from the CDN, parse those, execute those, render those.

[00:49:36] JM: Got it, and that happens for every webpage across my application, or is it one big GSX file that satisfies the entire application?

[00:49:45] AN: Webpack produces one big GSX file. But then webpack – Even before it gets to your browser, turns those into individual JavaScript and CSS and HTML files.

[00:49:58] JM: Got it. The term webpack configuration, so that implies that different applications might want webpack to parse their application resources in different ways. What are the things I can tinker with in webpack configuration?

[00:50:18] AN: Oh! Absolutely, everything. It's incredibly powerful, and it's actually really fascinating. Let's think of an example. CSS is really interesting, because you could import a CSS file into a JavaScript file. Then when it goes through webpack, you have an explicit dependency between a CSS file and a React component. So that CSS can then be moved into your production-ready CSS file at the end of the day. That's just a load the file, put it into a new file.

But you might want a more explicit dependency between CSS. You could import the CSS file and then perfectly specify in your GSX where you want that CSS to be used. You say I want to import all of my styles from main.css or whatever it might be. From main.css, I want my list item to have the class list and my bullet points, order bullet point, to have the class lists – – bullet or something like that.

There have been webpack plugins in Lotus which had been written which are smart enough to say, “Okay, I know that even though this developer has imported the entire CSS file, they’ve only specified that they wanted this list and list bulleted class. I'm only going to take those two classes and put them into the final production CSS file,” which a user then downloads. That's extremely powerful, right? You’re only shipping to the end user what is being used at the end of the day, which is great.

There are so many other things which webpack can do. You can write your own custom plugins to move files around, or transform them, or ad things to them, whatever you might want to do.

For example, at Slack, we have webpack plugin which traverses all of our source code and finds all of these strings, which will be shown to a user. Extracts those into a JSON file so that they can be then sent off for translation, because Slack is available in like 15 different languages, right? However many it might be. That's done through webpack. We create an AST, traverse those, grab all the strings, JSON file.

[00:52:40] JM: Got it. It can be customized to create the layout of text files and JavaScript and CSS and stuff that I want my – Basically, my file schema to be all the different files across my application. The complex spaghetti of different precompiled, pre-rendered JavaScript, GSX, I don't know. Maybe the coffee script or typescript and stuff, and the final result is something that is ready to be shipped to the end-user.

[00:53:17] AN: That's exactly right. Ready to be shipped the end-user and something the browser can understand. I think at its most simple form, webpack is essentially – Webpack essentially takes entry points. You tell it where your code starts. Where it can start traversing from. That's step one. Step two is it creates an AST tree of everything it's – All of what it's traversed. Step three is it applies transformations, whatever you tell it to do. Step four is produce the assets from those transformations. That's in its simplest forms; entry, AST, transformations, output. That's it. But you get to define the how the AST is loaded and you get to define those transformations to then produce those final assets.

[00:54:06] JM: Beautiful. We've only got like 10 minutes or so left, but I wanted to get – Or maybe we can go a little bit longer, but I want to get into a little bit of the usage of how Slack uses React and Redux and does state management, because Redux is going to hold by state for the React components on the Slack application. If I close my Slack application, if you were just keeping everything in-memory, then all of that state would just get thrown away. But if you can checkpoint some of that state disk, then the next time I boot up Slack, you can have a quicker load time for the React components.

Can you tell me about how Slack does state management in React?

[00:54:58] AN: We could talk for hours about this. It's a fascinating concept. We're all-in on React and Redux at the moment, but we have come across really interesting issues from not using their technologies, but from implementing their technologies. On large Slack workspaces where there’s tens of thousands of users using Slack, lots of events are coming over your WebSocket all the time. Users sending messages in channels. They’re uploading files. They’re reacting to different messages with emoji. So much is happening, and all of that comes over the WebSocket and all of that is stored in Redux. That the power of Redux. It's a single data store where you can only put in data one way and only get out data one way.

At Slack itself, the company, Slack the company, and individual’s Redux store updates about 10 times a second. 10 times a second we’re updating that Redux store. It’s like a crazy amount. Of course, we’ve optimizing that as much as possible. We throttle and de-bounce and do things like that, but there are some things where you can't do that or you don't want to do that. That's been an interesting edge case for us in adapting these technologies.

We've had to develop very strict like ESLint plugins to try and detect performance regressions. ESLint plugins to detect performance regressions. That’s usually done through end-to-end testing, but there are specific things where if you don't write React components in the most optimized way or you're allowing it to update more frequently than it should do, that could introduce performance issues on the desktop clients that a user is using. It had to be very thoughtful about how that all works and how that's implemented. Slack is so data-driven that Redux being a single source of data, again, it can produce issues like that.

We also use Redux in a slightly different way than what other people might use. We actually – The common use case for Redux is to create a single Redus store and then put all of your data into that store. There’s like a big warning message at the top of the Redux docs which says, “Don't ever create multiple stores, because you’re going to have a bad time.”

At Slack, we have created multiple Redux stores. We’re not having a bad time, but there's a good reason for it. It’s because of security. When you load a Slack application, you might be signed into your workspaces and you might be signed into personal workspaces, and we don't necessarily want that data to cross. We don't want to store all of the channels together or all of the users together, things like that. So we have individual Redux stores for security reasons. Slack is extremely security conscious.

When you actually switch between different workspaces in Slack, what's actually happening is we just switch out the underlying Redux store with the new workspace that you're switching to and then we allow React to re-render everything, because React is data-driven and suddenly it finds itself with a whole load of new data. It says, "Oh! I should be looking at this specific channel and I should be showing this specific message that's re-ended to show those things." That's another interesting thing of how we use Redux as Slack.

There's lots of little things like that where we've tried being clever and there's good business reasons for doing things like that as well. But overall, we've had a great experience with React and Redux. I think the biggest power for us from those two pieces of technology is how standardized it has made our codebase. Like I mentioned before, one of the most powerful things that we have done for all frontend developers at Slack is standardize how they are writing code, because it means everyone understands all of the different repos implicitly and can contribute and can debug anywhere. It's amazing. That is all thanks to React in Redux for sure.

[00:59:15] JM: Let's zoom and give a little bit more context for what you've actually standardized there, because again, there are these clients like Android and iOS that are not using React, right?

[00:59:30] AN: That's correct. Yeah.

[00:59:33] JM: But it sounds like even those client networking flows have been impacted by the fact that you've re-factored the web frontend or the Electron frontend to have a certain data flow. So can you just give a little bit more context? Now that we've talked through a lot of things, just tell me a little bit more about what you mean in terms of this standardization.

[00:59:58] AN: Yes. Standardization, I'm specifically talking about web technologies. That is the Electron app, essentially, on Mac, Windows, Linux, things like that. We have standardized how to access data, how to store data and how that data is then rendered on the screen via React.

[01:00:19] JM: It's like disk versus CDN versus server.

[01:00:23] AN: It's more the developer workfl ows, I guess, or the programming principles. 10 years ago, let’s say, if you wanted to store state about a specifi c component, how would you do that? There are hundreds of ways you could do that. You could – Let's say you’re talking about toggle for a sidebar. You click on it, opens, closes. You could apply a class to that component which says it's open. You could store that states in a global object and read it from that global object. Check what state it’s in. You could store in local storage. You could store in a cookie. There’s like so many different ways that you could have stored data.

If you don't have a forced standardization across your codebase, all [inaudible 01:01:16] different developers are going to store that data in all different locations. But since we have implemented Redux, and Redux has become our source of truth, there is only one way to get data and store data now, and that's through Redux. That standardization has been extremely helpful for us.

Saying that though, we have actually standardized some things across the mobile clients as well. The iOS and the Android and the web infra teams meet on a regular cadence to talk about different optimizations that they've figured out and how to load things better. A lot of the programming principles and the kind of way that things have been designed have been implemented in web and mobile clients. But of course they’ve been done in their own native languages. It's the kind of the text spec which is shared, but the implementation is different.

[01:02:17] JM: What are the outstanding frontend performance issues that you'd still like to make to the Slack frontend?

[01:02:26] AN: Great question. There's always more that we can do. One of our goals when we rewrote the web application was to get Slack to boot in a specifi c amount of time. Previously on large workspaces, Slack took a good 10 seconds to load or something like that. You might remember there were those kind of like cutesy messages, like welcoming you to Slack or things you can confi gure. With the rewrite, we actually had to remove those, because Slack boots so fast now that you don't have time to read those messages, which is nice, right? But we know that we can get it faster. Even faster still. That's kind of an interesting thing to think about, because how often do you really open Slack? You open it once and then it's open for an average of eight hours a day on someone's machine. But you could introduce interesting concepts if you can do really fast boots of Slack. For instance, if a user has Slack open for eight hours a day, while Slack releases code 20 times a day and there might be bug fixes in there. There might be new features in there. How do we get that on to a user's machine if they consistently have Slack open without reloading or refreshing the page?

Well, if you can guarantee that a reload can happen within a second or a couple of seconds, you could do it in the background when you know that Slack isn't being used or is hidden behind different windows. You could really quickly reload the client for them so that they pick up the new code and get all the latest and greatest features and bug fixes and things like that.

So you can almost see it performance as a feature as well when you think about things like that. Other things are always we tend to concentrate a lot on the actual – The typing experience and switching between different messages and how fast you can interact with Slack, because it's insane power. You can connect with your colleagues at an instantaneous rate. If there's any lag typing or channel switching lagging or anything like that, that's going to affect your experience of using Slack.

There's a really interesting concept and performance and that you don't' notice good performance. You only notice bad performance, right? You never notice when a website has gotten faster or more snappier or whatever, but you always notice when it's taking the extra few seconds to load or something like that. That's kind of how we think about performance at slack, is that the work that we do should just make everything more simple, more productive, a more pleasant experience to use. If you haven't notice something performance-wise in Slack, we need to fix that, and we've done a bad job there. That's how we tend to think about it.

[01:05:15] JM: Anuj, thanks for coming on the show. It's been great talking.

[01:05:17] AN: Really appreciate your time, Jeff. Thank you.

[END OF INTERVIEW]

[01:05:28] 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.

[END]