JARED: Should people write tests first?

JOEL: No.

JARED: Hello. Welcome to Dead Code. I'm Jared. Today, we're talking about a bunch of things. Our guest is Joel Drapper. He works on lots of things, and we're going to talk about a whole bunch of them. We're going to talk about his view framework. We're going to talk about Literal, whatever we want to call that. We're going to talk about his testing framework.

Despite what you heard in the cold open, he does write tests. He actually has an interesting perspective on writing tests and when to write tests first that we talked about after the interview, so maybe we'll have him on at some point to expand upon that. And finally, we'll be talking about the projects that he's putting together that combines all of these other open-source projects that he's been doing, and some other people, like Stephen Margheim, have been working on. Let's get into it.

Joel, welcome to the podcast.

JOEL: Hey, thanks.

JARED: Can you tell our listeners a little bit about who you are?

JOEL: Sure. Yeah. So, I am Joel Drapper. I am a software engineer based in the UK, in Oxfordshire, and I work on a bunch of different open-source projects in the Ruby ecosystem. I'm working on my kind of grand vision for a new full-stack web framework, eventually, that probably will never happen, but hopefully, is going to produce some interesting side projects along the way.

JARED: Yeah. And those various projects are why I wanted to have you on because I've used some of them, and I'm really curious about some of the other ones. I thought maybe we'd start by talking about Phlex. So, what's Phlex?

JOEL: Yeah. So, Phlex is, I think...I don't know if it's the first...or one of the first projects I started doing open source. It is basically a way to write your view layer just by using Ruby objects. So, you can create...if you think about anything that you could sort of draw a circle around on a webpage and give a name, you can think about that as being an object. It's an instance of an object.

And that object can have a method that describes how it's going to be rendered. And in that method, you just use Ruby methods to create the HTML. So, every HTML element is a method that you call. You pass a block for the content, and you pass keyword arguments for the attributes. And so, you can kind of use it to build HTML views in a purely object-oriented way.

JARED: How does it differ from something like ViewComponent?

JOEL: Yeah. So, ViewComponent is very similar, except that ViewComponent, when it comes to the template, it goes out to a different language. So, it uses ERB, I think, by default. And I think you can use various other templating languages like Slim and that kind of thing. Whereas because Phlex is just using Ruby, it means that you have a bit more flexibility. You can, for example, without extracting a whole component, you can extract just a small piece that you might use again and again within your component.

So, like, you might have a nav component, and you want to have a snippet for your nav items. You don't need to have a nav item component. You can just have a method called item that takes the parameters of each item and then outputs, you know, just the part of HTML required for that, and that can actually be public. When you make a method public like that, you could expose it so that when you call to render the nav, you can pass the block and pick up the yield, and then you can call that item method multiple times on it. It's really difficult to try to explain that in audio form there.

JOEL: Yeah, yeah, no, it took me a second to wrap my head around that the first time I tried to do it, but that pattern is really cool. I've used Phlex on a number of projects, and my personal website is all built in Phlex and some other stuff we've done is in Phlex. And I found that it was really nice to have that sort of pure I just make a file, that's my component, and being able to sort of fit that into the sort of broader web development patterns on a project.

So, if a project, for example, uses, like, BEM for its CSS conventions, for example, you can build helpers into your setup to automatically generate your BEM classes based on the names of your components. And that sort of reduces sort of quite a bit sort of duplication and boilerplate in the projects I was working on, certainly.

JOEL: Yeah, it's really cool because it's just Ruby. There's a lot you can do in terms of, like, you can just inherit behavior. So, you can either use class inheritance or mixin modules or something like that to layer on behavior. And you can even do things like you could, for a specific component, you could override an HTML element. So, you could say, if I render a div in this component, I'm actually going to override that method, and it's going to call super with these specific overrides, or something like that.

I used this for a markdown component that I made that would take some markdown. It would render the markdown using, you know, H1, H2, H3, et cetera. But then I wanted to be able to just say, “I'm going to override H1 and say that it's actually going to call super with these Tailwind classes.” And so, you can have this component where you just say, I'm going to inherit from this kind of abstract markdown component and override what do I want each of these different pieces of the syntax to do? So, that's pretty cool.

JARED: Yeah, that's really interesting because I also had to deal with rendering markdown in the context of Phlex components. So, what I ended up doing was not that, though I should have considered that. What I ended up doing is implementing my own Redcarpet::Renderer and

having that redefining the various components that I wanted to attach. I think it was Tailwind. No, it wouldn't have been Tailwind. It would have just been BEM classes, too, having that call out to my Phlex components, which sort of does illustrate, on the other side, how having these reusable components is really useful because you can just sort of plug them in where you need them, regardless of what other technologies you're using.

JOEL: Yeah, exactly. And also, if you're good at writing Ruby and, you know, your team is made up predominantly of Rubyists, I think it can just be nice to keep everything in one language and in one place. A lot of the code, like, component code that I work on it's constantly either reading an instance variable or iterating over some kind of enumerable object. And there's no switching between Ruby and then ERB, HTML, that kind of thing. So, that can be really nice. And then, it also just means you can use your existing formatters. Like, you can use RuboCop. You can use Ruby LSP.

We've done a lot of work recently to make Phlex kind of expose all of its methods to the Ruby LSP. So, all of the HTML elements they're actually defined dynamically. There's, like, a loop that goes over this big list and defines them. Well, that's how it originally started. And we updated it to actually just spell out every single one and even put in the...we couldn't do it for all of the HTML attributes, but for, like, kind of a good example, but, like, input, for example, input, you're typically going to have a name, a type, various other attributes.

We actually updated the code that generates the HTML elements to make it create basically, like, this empty method first just so the Ruby LSP can pick it up and see that, ah, I know that there's a method there called input. And then, we would run the macro that would generate the fancy dynamic.

JARED: The real method.

JOEL: The real method, yeah, which is just a lot of, like, class_eval.

JARED: Cool. I mean, that's awesome because that's something that's been a pain in a lot of view templating languages and other options in the spaces that, you know, whether it's SLIM, or ERB, or HAML, or any of these, you don't typically get linting and formatting as good as you get with, you know, the tools we have for just pure Ruby. And it's always trying to keep projects consistent within your views ends up just really annoying.

JOEL: Yeah. Yeah. Another really interesting thing about it that I think is probably less well-known is how rich the attribute serialization is. For example, if you say, “My class is,” and then you pass it an array, within that array...so in Ruby, it's really easy to use inline if conditionals to basically provide conditional items in an array, where you can say, it's active if, and then, since this variable is true, or something like that.

And so, it makes it really easy to do things like passing in conditional classes or conditional styles. And then, for attributes like style, you can pass a hash. So, you can use, essentially,

Ruby hashes as your style tags, and it automatically converts font-weight as a symbol to the font dash weight CSS rule.

JARED: That's really cool. I did like the style one. I did not know about the class one, which is kind of funny, in retrospect, because I basically just reimplemented that for myself.

JOEL: [laughs]

JARED: My project literally has a class names helper that sort of emulates what you're describing, which is also similar...if anyone's used the classnames NPM package in React, basically the same idea, but that's funny. Mine might do a little more than what you get out of the box because mine, you could do things like hashes and stuff in there. So, if you did, like, key-value, then it would treat the keys as classes and the values as a Boolean, whether you're supposed to apply that class or not.

JOEL: I think ours supports that, but I can't remember.

JARED: I have to try it out [inaudible 10:45].

JOEL: I'm pretty sure it does because I think that was originally the way that we were recommending people do it. But I liked the syntax, like, the way that it looks in Ruby when you do the array version, it's just really nice. Because if you use a hash, you have to sort of read the rocket. You have to read that rocket as if. Like, when you see it, you say the word if, but you can actually replace it with the word if, if you use an array.

JARED: Yeah, cool. Well, I love Phlex. We work on a bunch of projects that use ViewComponents and some that are just, you know, regular ERB. And I've had a really great time using Phlex. I know that Phlex Rails ships with Rails generators for generating components. And after my conversation with Garrett Dimon, I actually ended up going and implementing my own custom generators that tweaked a few things and set up the style sheets and everything for the components.

JOEL: Awesome.

JARED: So, that was really cool sort of leveraging both sides and being able to use the generators that were already there as sort of a reference, so...

JOEL: Yeah. One thing that's amazing that we could do, and we are not doing it yet, but we so could do this and should do this, when you know what the framework is and you have a direct connection to the framework, which we could do with Phlex Rails, right? We've got two different projects. Phlex is just a Ruby project with no dependencies at all. And then, Phlex Rails can actually lean on Rails a bit. So, one thing that I really want to build into Phlex is source maps. So, we can actually tell not only where every one of your components is defined, but also where every single element is defined, like, down to the Ruby source.

And so, if we can come up with a format where we can put this information into comments when you're in development mode, then we could have some JavaScript that we inject into the page that gives you this dev interface that lets you select an element. And then, it will pop up and show you how you can jump to that file. And you could click it, and that file could open up automatically for you because it could send a message to the Rails server that's running in the background. And the Rails server can, obviously, execute shell commands, and one of those shell commands could just be, like, Z, and then this path, or whatever editor you're using. And it should launch your editor for you straight from the browser, which I think would be so cool.

JARED: I'm pretty sure there used to be a thing like that for just regular Rails views.

JOEL: Yeah, and for controllers, you could do that.

JARED: At the partial level was able to do that. So yeah, that would be really cool and definitely totally doable.

JOEL: Yeah.

JARED: So, let's talk about some of your other projects. Tell me about Literal.

JOEL: Yeah. So, Literal, I think, it began its life in Phlex in a really, really early version of Phlex. And, basically, like, creating all these components, I ended up making a lot of very small objects, Ruby objects. And it just felt like there was a lot of boilerplate involved in doing that. Like, for every one of the properties that your object has, you have to create an initializer.

Wait, so you create an initializer, and it has to take each one of those properties as a parameter. Then it has to assign it to an instance variable, and then you have to specify separately whether you want at a reader, at a writer, at a recessor. If you want predicates, so, like, methods with question marks after them that just check the truthy-falsy value of one of these instance variables, you have to do that separately as well.

And so, I wanted a way to sort of generate these from a single definition of each property. And, additionally, I wanted a way to sort of provide some validation at runtime that if I'm expecting a string here and you send me an array, like, raise an error, complain. If I have a thousand objects being initialized to render a page and every single one of those objects is being very cautious about what it receives, and it's going to raise an error if it receives the wrong thing, that means that just running a test that renders the page, even if you don't make any assertions about it, you just render it.

Now, I know, obviously, that's not enough. You need to test your stuff properly. But you get a significant amount of signal for just doing that. I don't know if you could put a figure on it, but in terms of every time that something's wrong, it tells you immediately very quickly that something is wrong. Like, that just seems to be the case if you build stuff like this.

So, Literal is basically...the main thing is Literal properties, and it's a module that you can extend into a class. And, typically, you would just do this into some abstract class, like your application component or something, your base component that everything else inherits from. And then, you can use the keyword prop and then define these rules for your properties.

So, you can define the name of the property, whether there's a reader or a writer or a predicate method, and you can specify, you know, private, public, protected, or false for each of those, as well as coercion rules. So, like, your question would just be the block that you give to this prop method, which means you can say, like, ampersand two S, or something like that to say, I want this to be, you know, attempt to stringify this thing before you check that it's a string or whatever. And you also specify a type. And the way that the type interface works is it basically uses Ruby's triple-equals interface. So, this is the interface that is used when you use a case statement or a passer matching.

JARED: Yeah. People call it case equality sometimes.

JOEL: Yeah. And case equality is really interesting. It takes a bit to kind of wrap your head around it because, essentially, it is like the object saying, I'm going to assert whether the thing that you have passed me is my kind, which means that it's a bit different when you're talking about a class of thing versus an actual thing. So, any string, if you send triple equals to the string class, it will be true. But if you send triple equals to a string, it will be the equivalent of double equals, right? It would check that the string is exactly that string. Anyway, you can think of a type or a validation rule as just being any object that responds to triple equals with truthy or falsy.

JARED: And Ruby has lots of them. Like, the regular expressions test whether the string you've passed matches the regular expression. Ranges test whether the value you're given is within that range. We have a sort of rich sort of ecosystem of things that respond to triple equals to test whether this is that kind of thing.

JOEL: Right. Yeah. There's loads of them. And there's also a rich ecosystem of things that take triple equalsable things like array dot any, array dot all. You can pass a block, or you can pass anything that responds to triple equals.

JARED: I didn't know that.

JOEL: Yeah. And it's much faster because it doesn't allocate anything. So, there's loads of places that already use this in Ruby. It's like this already well-recognized pattern, so you just need to create these objects. So, Literal also has these types, these kinds of generic types that you can use to create more constrained types of things.

So, you can use string as a type, but you can use underscore string, and now you have the ability to pass in constraints. So, you could say, length 10. And what you're doing there is you're

saying, create me a string type that has this length constraint. And so, Literal is going to call the length method on the object. And it's going to use triple equals again to compare what you set that length to with the value that comes back from calling that method.

So, again, you can use a type. So, you could use a range. So, I could say length 10 dot dot 20. And now it's going to check that that string is within, you know, the length is within 10 to 20. And so, there's a bunch of these. There's, like, union, which means basically or. It's going to be this or this or this. That's very useful for components if you have position and you want it to be a symbol, but you want it to be one of four symbols: top, right, bottom, left. You can say position equals union top right, bottom left. And then you can say, define the attribute position, and then its type is this position, capital P.

JARED: Sort of like an enum in some type systems.

JOEL: Yes. Yeah. I mean, Literal also has enums as well as unions, but unions are kind of like a more lightweight version of it. So, there's a bunch of these types, for want of a better word, like, generic types. They're really just functions that return objects, but they're just special objects that are designed to essentially just respond to triple equals, and then that's how validations work. So, you can make loads of these objects that work like that.

And then, Literal also has, essentially, equivalence to struct and data objects. So, the main difference is they're automatically serializable. So, they kind of make some assumptions that you are basically using these just to store data, and it's basically frozen or not frozen, depending on whether you use Literal struct or Literal data. And then, so they are marshallable really efficiently. You can convert them to hashes. You can use double equals to compare them to each other, and they have a default implementation of that. They support dot hash. And they basically support a bunch of things that you'd want from kind of a basic object or an AST node or something like that. And so, they also are very useful.

The third thing is enums, which are essentially object-oriented constant enumerations. So, Rails has an enums feature, where, in Rails, an enum is, like, an attribute of the parent rather than being an object on its own. So, if you have a post status enum, in Rails, the default is, like, the post has status as a property of itself rather than it has a status object. And I like to think about a post as having a status object, which itself can have behavior, can be compared to things, can be enumerated on the class. So, Literal has a way to define these and to define methods on them and things like that.

JARED: Cool. I mean, that sounds very useful. I've seen a few sort of similar attempts to this. I've seen some of them, at least one was in a project that we inherited but was abandoned, and that's...

JOEL: Oh no. Was it my one [laughs]?

JARED: I don't think --

JOEL: Because I made Literal enums as a separate thing originally, and then I've abandoned it and merged it into Literal, the gem.

JARED: No, this was much, much older, and this implementation looks a lot more serious, let's say. We've inherited a number of projects over the years where people were the, you know, the CTO had decided to invent his own dependency injection framework and promptly abandoned it when he changed jobs and left the company to inherit this weird piece of code that he wrote. And we had a similar case with some similar project to this. But both feature-wise and, like, feature and functionality-wise, this seems more like what I would want out of this.

JOEL: Oh, it has been very carefully designed. And I've been through, I think, about four or five different iterations, like, complete rewrites of Literal enums. Where we're at now, I really like because it's designed to integrate with Ruby LSP again. So, when you define your Literal enum, you say, like, class status inherits from Literal enum integer, and then you define a constant. So, you'd say, published equals new, and then you pass in the value zero or whatever, or the string published, however, you want to serialize it.

And because Ruby LSP can see that you are defining the constant published and then it equals new, so, it's going to be an instance of this enum, it knows. And so, you can put comments in and things like that. And the auto-complete is amazing. It shows all your comments, and Ruby LSP supports markdown. So, in Yippee, we have this enum for HTTP status codes. And we have documentation from MDN, as well as a link to the MDN documentation on each status code that just pops up in your editor as you start typing out the enum. So, it's really cool.

JARED: Yeah, that's awesome. I guess one question I would ask is, how's performance? Should I be thinking about whether or not to use this in, you know, really performance-critical parts of my application?

JOEL: Yeah, performance is pretty good. There are some types that do kind of like O(N) checks, so, like, the array type that we have. If you say I have an array of strings, that is going to do O(N) checks when you pass it an array. If you give it an array of a thousand things, it's going to check a thousand items to make sure that they're all strings.

But all of the checks that happen at runtime are zero allocations. So, every single one of the types that we generate, when you actually test the type, so you've created the type already, maybe you've assigned it to a constant, or you've just used it in a class, that happens once when the application boots. And then, every time you call triple equals on it, it's doing no allocations. So, it is very fast, but yeah, some of the types can be a bit slower.

One of the things that we're working on at the moment to kind of get around some of this is to have Literal objects that can stand in for these collection types. So, if we have a Literal array object that can stand in for a real array and kind of has a very similar interface, then we can

check the types as you put them into the array, and then we don't have to check them ever again.

So, if you pass that array around to different things that are expecting that specific generic Literal array, like a Literal array of strings, we know that it can't contain integers. And also, we can do stuff like...if you have a Literal array of strings and you map them with the Proc ampersand length, we can actually look up that Proc on string because we have a map that says that that must return an integer.

So, actually, we can map from a Literal array of strings to a Literal array of integers with a specific Proc without actually checking anything because we can kind of see how these types progress through the mapping. So, that's a bit more involved, a bit more, like, you have to actually use Literal's types for that. But yeah, it should mean that it can be a bit faster in some cases.

JARED: Yeah, that's cool, providing that sort of next step if you run into those kinds of performance issues. Speaking of performance and speed, I think it's good to have different voices on the podcast, to have different opinions on certain things. And who better to come on and talk about the other side of testing, someone who doesn't write tests first, than somebody who's writing a test framework anyway? So, tell us about Quickdraw.

JOEL: Yeah. Okay. So, first of all, Quickdraw is the most experimental project that I have published, I think, so don't use it right now. Probably you'll run into a world of pain if you try to use it, probably give it a few more months.

But Quickdraw is basically an alternative take on testing. It kind of started out quite different and explored some different ideas. And now it's landed somewhere much closer to feeling like Minitest, but it is fundamentally built on the capability of using multiple CPU cores. So, it is multi-process and multi-threaded, and that comes with a lot of challenges. It basically makes everything that you want to try to do more difficult. So, it doesn't have a lot of the basics that you might expect to have from a testing library.

For a long time, the test output was terrible. It's getting better now. Like, for example, you can't say, "Fail fast," because, actually, fail fast is really tricky to do when you have 12 different processes running multiple tests.

JARED: Yeah, when you're orchestrating all of these different things, and it's like, well, what does first even, you know.

JOEL: [laughs] Right.

JARED: Now we got to shut down all these tests, yeah.

JOEL: Yeah. So, we're going to have to take an interpretation of what fail fast means, and probably it will be the first process to get a message back to the parent process that says, I have a failing test. Like, that's the one that we'll show you, and we'll just ignore the rest of the results or something.

But for a long time, it didn't even have two-way communication, really. It was using pipes. We recently re-architected it, and I think it's much better now. So, originally, it was, like, basically, just splitting up all your tests and kind of just handing them out evenly, but, of course, not all tests are even. Some tests take much longer than others.

And so, what we're doing now is pretty interesting. We'll basically load all of your tests into a big bucket. And then, we will fork one process for each of your performance cores if you're on a Mac, or all of your CPU cores minus one if you're on anything else. And then, they will have this thing called a sized queue. It's like this Ruby object where you can say, basically, I want you to have a maximum size of 100. And that means when you try to put something into the queue, if it's reached its maximum size, it is going to block. It's not going to let you do that, so that thread is going to block. If you're trying to take stuff out and you run out of stuff, that's also just going to block.

So, they have a sized queue, and the size is kind of figured out based on how many tests you have and how many CPU cores you have and stuff. They will send a message to the host and say, "Hey, I need my next batch." The host will respond. It can do this in, I think, four or five bytes. It will respond with the next number that you can take a batch from. And then, it's going to basically grab that batch from its copy on right copy of the tests, shove them into the queue, or try to shove them into the queue, but probably it's going to fail and have to just wait and keep working on it.

Eventually, it then has this thread pool, and they are basically just working, pulling stuff out of the queue. So, you can have maybe eight threads per process or something like that. And, eventually, it's going to get below this size, this maximum size, which means it can then go out and fetch the next batch from the parent. And, hopefully, there's enough in the queue to keep the thread pool working while it does that round trip through the Unix socket up to the parent and then back down again with the next batch.

So, that's basically how it works, but it took quite a while to figure that out. And the result is we can essentially run it at however many performance cores you have times as fast as something like RSpec, which is awesome.

JARED: Yeah. Was that the goal, just basically leveraging concurrency to get a faster test runner?

JOEL: Yes. Yes. Because my Mac has, I think, 16 CPU cores, or something like that. And it ran RSpec, I think, about 10% faster than my previous one, which had, I think, eight. I was like, that's just not right. First of all, it's like three generations, or something like that, two generations

faster CPUs, but, also, like, it should be faster. I started digging into why this was. And, obviously, Ruby is single-threaded, or Ruby has this thing called threads that aren't really threads. It runs on a single CPU core and, yeah, tests don't need to be run like that. So, the idea was to try to use that capability and kind of build it in from the beginning.

JARED: That's really cool. I mean, I would gladly accept a faster test runner. So, I think we'll have to check back in with you once it's a little more mature and you're recommending people go try it out.

JOEL: Yeah. The big thing that I need to get done, I think, before I could recommend people try it out, is compatibility. There's a compatibility mode now for Minitest and RSpec. So, the idea isn't to be able to run your RSpec suite perfectly, but that maybe you'd have to change less than 10% of your tests. You can, for the most part, just copy-paste your RSpec tests, and they will run because we can implement all of the different matches and things.

JARED: Cool. So, the final project that I really wanted to hear some more about is Yippee. What is it?

JOEL: Yeah. So, Yippee is a full...well, it's nothing right now. It's a private repo that I've been working on for a while with Stephen Margheim. And the idea is a very opinionated, lightweight, full-stack framework that is built on top of SQLite and Ruby. So, yeah, the idea is if you are willing to accept that your app is going to run on one server and you're just going to scale that server to be really big and have, like, I don't know, I think you can go up to 200-plus CPU cores now, then you get a bunch of benefits in terms of just simplicity of deployment, of everything that you do.

If everything is based on SQLite and this very simple Ruby application, you can scale to, you know, millions of dollars in revenue, I think. I think you can scale to hundreds and hundreds of thousands of requests a minute easily with SQLite and a single server. And so, it's basically...it's bringing together all this stuff we've been working on in terms of Phlex, Literal, Quickdraw with a new router, a new job system, and a new ORM. And that's, like, the big part that we've been building out over the last year or so.

So, we started working on this ORM, and we got to migrations, and we were like, okay, wouldn't it be cool if you could just define your schema and then run a command, and it would generate the migration for you? Because it's so much nicer just to say what I want and then have it generate the thing, and then I can read it to make sure it's okay before I deploy it. But I don't have to think about how I'm going to change it.

JARED: Kind of spiritually similar to Django's automatic migration stuff.

JOEL: Probably, yeah, yeah. And there’s some other things that we were trying to do. We realized, basically, you can only get so far with regular expressions. We need a SQL parser.

Like, we need to be able to actually pass this schema dump, at least. But probably we'll only be able to pass every piece of SQL, like, that would be an amazing capability for our framework.

So, we kind of stopped working on Yippee and started working on a SQLite parser, which is, it's, like, specifically designed to parse SQLite's flavor of SQL perfectly, and it's been a lot of work. We've got to the point now where it's parsing create table statements, I think, a hundred percent, which is what we need for the ability to do these migrations. So, there's that. But then we also built a router, which it uses a similar router to Roda.

JARED: Cool.

JOEL: I don't know if you know, Roda uses this thing, what do they call it, a routing tree?

JARED: Yeah. And it's built so that, like, the...forgive me for pronouncing the word differently than you, but --

JOEL: [inaudible 35:08]

JARED: Yeah, no, it's fine. I just...you say it your way. I'll say it my way.

JOEL: No, it's always confusing because we've got this root, route [chuckles]. But I would call a root [inaudible 35:17].

JARED: [laughs] Oh yeah. Yeah. But the routing tree ends up being, essentially, just part of the syntax of the AST of your router, which I thought was a really nice pattern and has, you know, it has advantages and disadvantages. The Rails router we use now has some really neat optimizations it can do because it, you know, builds up sort of in-memory understanding of your routes and then can optimize itself. But if you're not going to go down that path, then something that's syntax-based routing tree is really cool.

JOEL: Yeah, you would think that...I don't know how the Rails router works, like, I don't really understand how it fundamentally works. But I understand that it is somewhat compiled at boot time. And you would think that that would make it significantly faster than something like a routing tree, but I don't think that it does.

Routing trees can get slow if there's lots of different places that you could go at any given level. Like, if you have, I don't know, 2,000 places you could branch off to at a single level, then routing trees can be quite slow, but otherwise, they're incredibly fast. Like, our router in micro-benchmarks is about 15 times faster than the Rails router from what we can see running locally in production mode because Rails is massive. It's doing thousands of method calls per request just going through the Rails framework that are pretty unnecessary for a lot of applications. And I'm sure there's good reasons for them being there, but, at the end of the day, if you're doing 50,000 method calls to load a single hello world, it's not going to be the fastest router.

JARED: Yeah. And one of the things I do like about the routing tree-based approach is you're able to sort of make performance decisions about, oh, even if you do have a lot of places to branch off of at a certain point, like, potentially at the root level of your routing tree, you can put the most critical ones first and do those checks and get those so that your worst case scenario at that particular branch you are going to only hit that for a very small number of actual requests.

JOEL: Yep. For larger applications, we have this alternative way of defining your kind of branching logic for a specific branch in the tree. Instead of it having to walk through all of your code and check each condition one by one, there's a short circuit that you can define as a hash. So, it can be like, if the next segment of the URL that we're trying to pass, so segment being between each forward slash, actually just matches one of these items in this hash, then we're going to immediately forward you to that branch and let you continue. So, that means that the routing is, like, a one when you have something like that.

And so, you don't need to use this all the time, but a lot of the time, it makes sense. Like, I think probably for your root root or your admin root or something like that, you can just have these shortcuts where you can say, if it's slash admin slash products, I don't have to check, you know, 50 different things. I can just look up this hash and then jump straight to the product's controller.

And in Yippee, we call these things, like, controllers. So, you have a route controller that starts...and its job basically is just to redirect you to different controllers, and each controller in Roda would be a block. But, for us, it's usually a class though it can be a block if you want to define everything in line. But it's usually a class, and you can kind of just handle each segment of your routing in its own controller.

JARED: Cool. That's a really neat approach. What's next for Yippee? You mentioned it's just a private repo now. When, are we going to see the first steps of it in the wild?

JOEL: I think we really need to finish the ORM for that to make sense because it's not going to be an amazing experience for anyone as just a router. I guess you could sort of use it now as a router plus Phlex plus Literal. But we'd really like to get the database support baked in there, especially as the whole thing about it is it's meant to be SQLite, everything native SQLite from the beginning.

And I think once we've done that, it probably won't be much of a stretch to get jobs and that kind of thing built-in as well. We've got some ideas about how we're going to do that. We might have some kind of hybrid approach where you can either just run a job on a background thread or even send it to another Puma worker that can run it on a background thread that's less busy, or you put it in a queue with a SQLite write. I think it depends on what kind of thing you're doing.

But there's lots of things we can explore when we have this constraint of we know that there's just going to be one server, so we don't need to worry about distributing things between different servers, going out to Redis, or anything like that.

JARED: Yeah, that's an interesting approach. I really like it, deciding, like, the...you mentioned the framework is opinionated, and I generally appreciate that. I think that you get into a lot of trouble building, especially frameworks and tools and things, and trying to make them for everyone. Whereas if you can say, you know, "This is the use case. If you don't fit inside the use case, sorry. Maybe you should use something else," and then optimizing really well for that specific use case. It's good for especially, you know, you call it a micro framework. It's good for staying micro. And it's good for knowing whether or not you want any given feature in the framework.

JOEL: Yeah. Yeah, and it's going to challenge the way that we think. Like, I think to use it, you have to really kind of adopt a different mindset as well. For example, and, honestly, like, I know this, and when I started building it, I got it wrong immediately. I started thinking about, like, okay, how can we avoid N+1s? Like, how can we make the ORM so that it's really difficult to get yourself in the situation where you have these really slow database queries? Because that's, honestly, the biggest performance issue for web apps like Rails apps is they keep running into these kinds of either unindexed queries or N+1s.

And we can deal with the unindexed query problem because if we can analyze your schema, we can actually find these immediately and suggest indexes. But N+1s in SQLite are actually great. Like, it is so much better to do N+1s than to have really big complex queries because that takes a lot more memory, and it holds up...any kind of locking that you might have, it's going to hold that lock for much longer.

JARED: And it turns out that the round trip to your SQLite database is not the same as the round trip to your, you know, [inaudible 42:19] database that's on another server.

JOEL: Yeah, we're talking probably a few nanoseconds or something to do this round trip or at least microseconds. And so, yeah, it just doesn't matter. And then, there's other things like, okay, so now does it mean that it maybe makes sense to put certain queries in views? Maybe. I don't know. Like, previously, I would say we should definitely try to build up ahead of time all of the database queries, and then send that down to the views as one thing, because, otherwise, you can have all of these N+1s and yadda, yadda, yadda.

But maybe you've got some logic in the views that means that if this condition is such and such, you're not even going to run those queries. So, I don't know, maybe it means that actually we can forget that pattern and start doing stuff a bit later and only doing it when it's necessary. You can have a condition in your view that decides whether or not it's going to query the database in a certain way because it's just so fast it doesn't matter.

JARED: Yeah, it's always interesting to sort of step into a set of tools or a different environment where the performance characteristics are just totally different like that. And there's definitely a lot of benefits, you know, we've seen a lot of work going into making SQLite a, you know, a

production-ready database, and that's been really cool. And it's cool to see projects like yours trying to leverage that in new ways.

JOEL: Yeah, I hope it all works out. I think the biggest challenge is going to be convincing people that it's going to be fine. Like, by the time that this becomes a problem, you're going to be making so much money from this application that you can solve any problem that you run into. Like, you can go multi-server eventually. Like, you can rebuild the, you know, build sharding into Yippee, whatever [laughs]. You can solve it.

The latency issue is, I think, the biggest issue, but if you can deal with the latency of just having one server that is, you know, as local to your users as it can be, then I think it can be pretty awesome.

One thing that we're really leaning into with Yippee, which is kind of ironic given that I worked on Phlex, is, like, single-page applications, believe it or not [chuckles]. I think that they're actually kind of getting really good now, like, really, really good. Certainly, I've been very impressed with Svelte. And I think that that goes a long way towards reducing the impact of the latency of having a single server serve your requests. Because if you have, you know, your entire interface is kind of local first and running locally, and then you're just going out to the server to fetch new data, sometimes ahead of time, you can still build really compelling experiences like that.

JARED: Yeah. The user is not necessarily going to notice the latency if the UI is done really well.

JOEL: Yeah.

JARED: Cool. It's really cool. Excited to see what comes out of all that experimentation with all these tools. Where can people go to follow all of this online?

JOEL: Mostly just on Bluesky. I don't really post anywhere else anymore. So, I'm at joel.drapper.me on Bluesky. Also, follow Stephen Margheim @fractaledmind on Bluesky, because he's posting a lot about Plume, our SQLite parser, and Yippee.

JARED: I've seen a bunch of the SQL parsing stuff on there.

JOEL: Yeah, yeah, he's been working with me. Like, he really, like, did 99% of the SQLite parser. I've just jumped in at the end. "I'm like, I didn't think you'd actually do this [laughs]. But you've gotten so far, so I'm just going to help you now," yeah.

And also, come join the Naming Things Discord. There's, I think, something like 450 people on there. We're talking about these kinds of projects and just generally fun, exciting things happening in the Ruby world, SQLite, that kind of thing.

JARED: Yeah, no, I don't check Discord super often, but I have been popping in there and seeing some really good conversations. So, that's definitely...I endorse the Naming Things Discord. There's some cool conversations happening in there. Cool. Well, thanks for coming on the podcast, Joel.

JOEL: You're welcome. It's been great.

JARED: I've been following Joel's work for quite a while. Very cool, really interesting to see all the different approaches and reworking he's done to make, you know, Phlex super performant, seeing all of the different cases that Literal can handle. I'm really excited to try Quickdraw when it's ready, and Yippee is really also really intriguing.

It's cool to see all of that work that's gone into SQLite over the last couple of years, allowing for these different approaches to making applications. Because there was, you know, when I was initially getting into Rails, SQLite would have not been even close to an appropriate database to use for any kind of production application whatsoever. And lots of cool stuff has happened since then, and it is now a viable option for certain cases. Sadly, not usually the case that I work in. In e-commerce, I think it would be a little tricky.

We do want to be able to scale in certain ways and get our servers closer to customers, but that's not the situation a lot of application developers are in. And that means that there's a big use case for SQLite and, you know, Rails has started adopting that. And it's cool to see new offerings coming to this space.

This episode has been produced and edited by Mandy Moore.

Now go delete some…