Ruby Rogues

196 RR Testing Clojure in Ruby with Ashton Kemerling

AVDI:

I'll try not to shred anything while we're on.

CORALINE:

Avdi has a shredding problem.

[This episode is sponsored by Hired.com. Every week on Hired, they run an auction where over a thousand tech companies in San Francisco, New York, and L.A. bid on Ruby developers, providing them with salary and equity upfront. The average Ruby developer gets an average of 5 to 15 introductory offers and an average salary offer of $130,000 a year. Users can either accept an offer and go right into interviewing with the company or deny them without any continuing obligations. It's totally free for users. And when you're hired, they also give you a $2,000 signing bonus as a thank you for using them. But if you use the Ruby Rogues link, you'll get a $4,000 bonus instead. Finally, if you're not looking for a job and know someone who is, you can refer them to Hired and get a $1,337 bonus if they accept a job. Go sign up at Hired.com/RubyRogues.]

[This episode is sponsored by Codeship.com. Don't you wish you could simply deploy your code every time your tests pass? Wouldn't it be nice if it were tied into a nice continuous integration system? That's Codeship. They run your code. If all your tests pass, they deploy your code automatically. For fuss-free continuous delivery, check them out at Codeship.com, continuous delivery made simple.]

[This episode is sponsored by Rackspace. Are you looking for a place to host your latest creation? Want terrific support, high performance all backed by the largest open source cloud? What if you could try it for free? Try out Rackspace at RubyRogues.com/Rackspace and get a $300 credit over six months. That's $50 per month at RubyRogues.com/Rackspace.]

[Snap is a hosted CI and continuous delivery that is simple and intuitive. Snap's deployment pipelines deliver fast feedback and can push healthy builds to multiple environments automatically or on demand. Snap integrates deeply with GitHub and has great support for different languages, data stores, and testing frameworks. Snap deploys your application to cloud services like Heroku, Digital Ocean, AWS, and many more. Try Snap for free. Sign up at SnapCI.com/RubyRogues.]

CHUCK:

Hey everybody and welcome to episode 196 of the Ruby Rogues Podcast. This week on our panel, we have Avdi Grimm.

AVDI:

Hello from Pennsylvania.

CHUCK:

Jessica Kerr.

JESSICA:

Good morning.

CHUCK:

Coraline Ada Ehmke.

CORALINE:

Hi there.

CHUCK:

I’m Charles Max Wood from DevChat.TV. I have a brief announcement. I’m putting together a Kickstarter campaign for a video series on Ruby on Rails, very relevant to this podcast. If you’re interested you can go to DevChat.tv/Kickstarter and check it out. And it’ll be up and out by the time this goes live. So, I would really appreciate your support for the shows, for the video series. So anyway, that’s my brief announcement. We also have a special guest this week and that is Ashton Kemerling.

ASHTON:

Kemerling. Yeah, you got it, got it in one.

CHUCK:

Do you want to introduce yourself really quickly?

ASHTON:

Yeah. I guess I’m a Ruby and Clojure developer of very, very small note based now out of snowy Chicago, Illinois. So, we’ll probably dig into that, the backstory on that a bit.

CHUCK:

So, Ruby and Clojure.

ASHTON:

Ruby and Clojure, yeah.

CHUCK:

Match made in heaven?

ASHTON:

Um, you’d be surprised, yeah, actually. I find that a lot of the things… so, the two are very, very different and the approach the world from very, very different areas. But they both, the things that I think drew people to Ruby originally of this concept of, it should be easy to use, it should be smooth and should have nice tooling so it’s comfortable to interact with, Clojure approaches it from a very, very same angle. It just approaches from a very different language theoretic background. So, while they are very different, if you really enjoy that aspect of Ruby you’ll really enjoy that aspect of Clojure. And I find that the overlap is usually non-trivial. Or sorry, of how many people use both will reach for one or the other depending on what they’re trying to get done and the context and all that jazz.

So, my work. I was at Pivotal Labs until a couple of days ago, probably a couple of weeks when this podcast goes out. And that was a Ruby on Rails shop. I would work on Pivotal Tracker and it was Ruby on Rails top to bottom, bits of Go, and a lot of JavaScript for the reactive frontend. And honestly, mixing Clojure into that, in particular and careful ways, once you get past the fact that it‘s a very, very different language, actually worked out a lot more smoothly than I would have anticipated in advance. If you had told me two years ago, “Hey, you’re going to mix these two in a professional environment,” I would tell you to get future me into some sort of help of some sort before he hurts himself.

[Laughter]

ASHTON:

And be like, start thinking about my life choices well in advance. But no, so it’s strange and the idea of using… I don’t know. How much backstory should I go into this, about how I ended up doing this and what I actually did?

JESSICA:

Tell us what motivated you to bring Clojure into this Ruby app and how you did it.

ASHTON:

Oh, okay.

JESSICA:

Or where you did it.

ASHTON:

So, I'm really interested in making programs correct. Performance is interesting. New features, it can be nice and it has a nice cathartic, "Look. We have a thing that we can do stuff with." But I find correctness is a huge thing for me. That's what gets me going, fixing bugs and being able to say for certain, to point at things and be like, "It's sufficiently fast but it's correct." And that's what I'd like, what I like. So, Pivotal Tracker is a non-trivial application. It's a distributed system, practically. Even though it's a Ruby on Rails backend, you and I are on the same Pivotal Tracker board for example, or project, we're seeing different parts of the same world state. And we're trying to communicate that through a third party, these updates. And [it's] a lot of some tricky edge cases in there.

And so, we were seeing it's pretty stable in general. But as you start to thrash it you might start to see a few weird things here and there. And there were more, but I helped squish quite a few of them. And I was trying to think of, "How can we do this better? How can we do this right?" And in a lot of cases, I would look through cases where we had written something and where Pivotal Tracker's TDD and pairings so the test to code ratio is quite high. I think it's as close to one line of test per two to three lines of code on average. And some of our test suites are taking, closing in on half an hour runtimes if you're not careful about how it's [throwing it] out.

So, we have a lot of tests trying to cover us. But things occasionally would still slip throughout. We would catch the majority of the obvious ones. And I would look at some of these failures. And some of them, I would look at it and be like, "Oh, well the test case covers all the lines of code." But it doesn't, this is [inaudible] is about the combination. Or it's an ordering problem. What's the order of the data that you feed in there? And that's actually a more complicated thing, because I can write a test that covers every single line of code but doesn't actually cover all of the actual branching scenarios or all the possible ways its state builds up on that.

And in my mind, there's two ways around this. One is to get rid of the state. But that's kind of a big sell. And when you have a working product, and don't get me wrong, Pivotal Tracker works. So, I can't go to my higher ups and say, "Hey, it works. But it doesn't fit within my design aesthetic. Let's rewrite it in this language I prefer instead." That's silliness and political suicide. So, the only way around that is like, "Okay, well let's find a better way to test." And I realized that the real weakness in the testing was the human. Humans can sit down and figure out all the possible combinations. They just don't want to because that's terrible. I'm not going to sit down and think of all the possible ways I can shuffle orders in an array. But the computer will happily do that for me.

And I started looking around and I heard a little bit around this generative testing thing, which is like unit testing but you let the computer come up with the cases in advance using a DSL to describe what type of data, or as I like to call it what shape of data. And the most famous one was in Haskell. But I was looking and I was like, "This is going to be a huge pain," because Haskell's weakness is I/O. I don't care that there's monads and all these things that you can do, and that's very nice and very interesting. But if you're going to do nothing but I/O which is what a test that's testing different code is always going to be, that's really, really painful. And you're fighting the language. And whenever you fight the language, your life's going to be more difficult.

So instead, I was like, okay, and I looked around and I found this test.check. And I was like, "Oh, this is great. I can compile it to ClojureScript. I can compile it to Clojure. And I can use this to test integration-style an application." And it started off on a hack week, let's test some of the JavaScript. Compile test.check into ClojureScript, take your finalized, we send this client JavaScript thing, throw them into PhantomJS and see what happens. And that was just a super simple, let's take a bug, I know exactly where it is, I know what line of code I can comment out to put it back into the code. And I can write a test targeting that and see if I would have caught it in advance. Given this tool and this technique, [inaudible] surprised nobody. It did. It just turned out that doing it with ClojureScript and testing JavaScript was the least compelling of all of the use cases, as it turns out. It's like, "Okay, that's cool.

But it's actually kind of a pain, to be completely honest, because the ClojureScript stuff at that point is, it's getting better. But it wasn't, sorry I should be more specific here. ClojureScript is a port of Clojure that compiles through, confusingly enough, the Google Closure Compiler. Good work there, everyone.

[Laughter]

JESSICA:

That's closure with an S in Google's case, right?

ASHTON:

Yeah. Oh, god.

JESSICA:

[Laughs]

CHUCK:

Yeah.

AVDI:

And none of these have anything to do with the computer science concept of closure, just to make things simpler.

ASHTON:

It's like… yes, exactly, exactly. It's like, oh, Clojure itself, the name, is not necessarily the best either. Which is why I pronounce it with a very hard j, because otherwise life gets really difficult really quickly. But Google has that really nasty habit of taking terms that are one, already in use. But more importantly, already in use that's fairly damn similar [chuckles] and then using that to name tools, like the Closure Compiler, or Chrome, where chrome is…

JESSICA:

The word…

CORALINE:

You'd think that Google would be good at SEO, but no such luck.

[Laughter]

ASHTON:

Oh, I hadn't thought about that. Oh, that's wonderful.

JESSICA:

Okay, okay. But Ashton, you used a known bug as a way to test a way of testing on a hack day.

ASHTON:

Oh yeah, yeah. And I think that was actually a moment of brilliance, probably one of three I've had in my life. Because that way, I was trying to use a tool I'm unfamiliar with. I wasn't super familiar with ClojureScript. I wasn't super familiar with generative testing. And interacting with PhantomJS and things from the Google Closure library, and it's a huge pain in the ass. There's no way around that.

You're using PhantomJS which is based on Safari four years ago. And you need a small mountain of polyfills to make the thing play nice. And I actually found out it wouldn't work if you didn't…. if any of your code ever references local storage or the browser storage API or whatever, if it's not pointed at a URL, it won't even load the JavaScript. It just refuses and not even executing it because it's like, "Oh, PhantomJS by default is visiting nowhere." It's like opening up a brand new tab and actually have to say, okay no, we're going to go to this blank page that I'm serving up on localhost because I have to be somewhere apparently. It doesn't even have to have anything on. It just needs to be at a URL. And now, we can go load the code.

And you're talking about, and especially, talk about in the hack week, I've got five days to do this and then I'm being cut off because we've got other stuff to do. So, I need an early win. I need it to be easy and I don't need to make my life more difficult, because I just simply, if I try to get 10 things done at once, I'm never going to get any of them done. So, I was like, okay, let's take something that we know exactly how to reproduce. And we know exactly what a failure condition looks like. And I totally smoothed over something here. Generative testing doesn't work everywhere. It only works in areas where there's a useful assertion that you can make without actually knowing what the data is.

You need something where, so the canonical example for this is sort. It doesn't matter what you're sorting. You know the answer of sorting. You know some of the things that are always true about sorting. Doing it twice gives you the same thing. Reversing it and then sorting, or no matter how you shuffle the things, sorting will give you the same answer. And all of the things inside of it are monotonically increasing. Those are what we would call invariants. There are things that are always true. And when you're looking at generative testing, you need to find areas in your code where that is also possible. Because testing for some little bit of UI thing shows up in only this one case, well that's not… you need to know what the data is before going into it. You can't write a useful invariant.

But in this case I had a bit of code whose responsibility it was, was to take a current list, another of things, a list of the new order that they're going to be in, and then without rebuilding the list completely, move things around to get to that. So, I've got a list of 10,000 things and I've got a new list of 10,000 things with a few differences. It's got to diff the two and then reorder one of the arrays so that the two match. Well, there's a useful invariant there, right? If I create a list of things and then shuffle them somehow and drop them both in, I should get the shuffled order every single time. They should never, ever disagree. And that makes for a super useful invariant. And therefore one, that would be useful for generative testing in general because there's a good useful invariant. It involves a lot of data that requires manually typing it out that no one actually wants to do. And it's just going to suffer from weird ordering cases.

And I knew there was a bug in there in the first place. So, I could undo the fix and know that I've actually succeeded in finding something, which makes a huge difference for the success rate of this kind of thing. I have an immediate, I mean that's also the theory with TDD right, is an immediate and obvious indicator of success or failure. So, you don't…

CHUCK:

So, can I jump in real quick?

ASHTON:

Yeah, sure.

CHUCK:

Because I'm wondering, why couldn't you do this kind of generative testing with Ruby or JavaScript instead of Clojure?

ASHTON:

You absolutely could. One, I wanted an excuse to use the language, because it's fun. And there's a high quality library that was available, and is available, at that point that does all of this things. There is…

CHUCK:

Okay.

JESSICA:

I can chime in on that, having done generative testing in both Ruby and Clojure. The tooling is better in Clojure.

ASHTON:

Mmhmm.

JESSICA:

It's even better in Scala. But the Ruby libraries are not there.

ASHTON:

Yeah.

AVDI:

I'm curious.

JESSICA:

They're getting there.

AVDI:

Is it hard to write? Is this… are these difficult testing libraries to write?

ASHTON:

It's not difficult from a theoretic position. The actual algorithms and structures you're using under the hood are not terribly complicated. It's the DSL I think, is honestly the catch. It's the DSL and the interfacing.

So, Clojure has a couple of huge advantages here. One is it has really, really good data structures. But more importantly than having good data structures, it has a culture of data structures first. There are just not nearly as many objects in Clojure land. Clojure's prefer to communicate with immutable data structures, which means that it's super easy to start generating those and being able to use those things. Because chances are, that's what your code is going… so, if you have a Clojure application chances are it's going to use just simple hash maps, vectors, lists, and sets.

The actual equivalent of Rack in the Clojure world ring just passes a hash map for the request and you return a hash map for the response. And so, you can actually call that without even having a web server because the web server's job is to take HTTP and turn it into that, which means that you could create a generative thing that generates fake requests and feed it in there because it's just a data structure. So, that makes it easier, that kind of culture of that.

And then two, macros mean that you can pull off DSLs that are tricky. I've definitely looked at the Ruby. There is a Ruby generative library. Do you remember the name of it, Jessica?

JESSICA:

There's a couple. Generative is one that's just getting started. And Rantly is one that's older.

ASHTON:

Rantly.

JESSICA:

And yeah, it's a little more mature but it's really, it's not there.

ASHTON:

It's not there, yeah. That was pretty much my opinion. I looked, because I was like, okay well, if I'm going to stand up here and say, "This library is great," I probably should understand what's going on here. And I looked at Rantly. And after seeing a couple of talks of it, and one, there are a couple of patterns I actually use in my own generators that were not actually possible in Rantly at this point in time. And it just looked… the DSL was just much more cumbersome. There's really no way around it. It just looked much more difficult to work with.

In test.check, I keep the generators in global variables and I can reuse them. So for example, one of the generators, the generators are fairly straight… I artificially separate them into two categories. Ones I call basic or simple generators, like there's one called Int. Guess what it does? [Chuckles] There's one called string. One guess what that does. And some of those take some parameters to be like, okay, I want an int between one and ten. Okay, int one space ten space, or close paren.

But there are also compound ones. Like, one of them's vector. And it's basically, give me another generator and I will produce vectors of arbitrary length whose elements are the results of running that generator multiple times. Or there's one called one of, which is give me an arbitrary number of generators and I will pick one of them to execute. Or, there's similar ones for frequency where it's like, give me multiple generator frequency pairs and I will randomly pick one according to the frequency, with the frequencies that you've asked for.

Those kinds of patterns are not currently, to my knowledge, expressed in Rantly. And they are used super heavily in the generators I've written, in particular the frequency one is super useful if you want to provoke date-time collisions. Because you could generate date-time by just saying, give me a number between zero and right now in milliseconds and convert it to a date-time object. But then you'll never have collisions, because that's a huge range. The generator will spread the numbers all over the place. And you're not actually testing things as carefully as you could.

And so, I use the frequency one to say, okay 70% of the time I want a date between 2005 and right now. But 30% of the time I want February 5th 2011, just that one day, noon. Because that way, I can guarantee that some of the stuff I'm feeding into my code has a date-time collision, which can happen in the real world. And that's super useful to mix some of them are the same. Some of them are just spread out, and make sure that it still behaves, is still well-behaved in the face of date-time collisions, which are rare but possible when two users click on the same button at the same time, or at least [inaudible] once you compensate for lag and all that jazz. And that's just not at this point in time, to my knowledge, easily done inside of Rantly, at least not the last time I looked. I'm totally willing to cop to, if it's improved massively while I wasn't looking. But…

JESSICA:

So, you're talking here about one of the other challenges of generative testing. When you're trying to test all the possible scenarios, or in your case when you're trying to test to find the scenarios that trigger a bug in an automatic way…

ASHTON:

Yeah.

JESSICA:

You spend a lot of time adjusting your generators and defining what really is valid input.

ASHTON:

Yeah. So, I actually have opinions on that. I don't think the generators are responsible for defining valid. I actually highly recommend not doing that. And there's going to be some code somewhere whose job it is to go take the data and go do the things, as it were, and determine if it's a success or a failure. That's not automated yet and will never be. And I think that's the place to do that, because you can make more cogent and coherent yay/nay, this data looks like crap and I can't use it decisions.

So for example, stepping forward in time from my ClojureScript one, I'm going to skip another bit of code I wrote and go to the final bit of code I wrote, which was I was using generators to represent user intent. Not individual actions like I click on this button, because no user in the world thinks that way. That user thinks, "I started a story." It doesn't matter how they got around to it. Okay, they opened up the thing and clicked on the dropdown, blah-dy blah-dy blah. But from the user's perspective it's, "I started this story."

So, I wanted to use these generators whose job it is to produce a hash map that represent a user doing a thing. So, one of them is add a task. Give me the story id. Give me the text that you want to add. Or add a comment, or delete a story, or move a story from here to there, and all that stuff, pretty straightforward things at a very high level. And then there was code whose job it was to go do that via Selenium or via API or whatever it might be. And then at the end we do some basic checks of, has it all fallen apart? Is the thing on fire? That kind of stuff. In that scenario, a huge number of the things I'm generating make no sense whatsoever.

So for example, if I ask you to create a story, delete it, [chuckles] and then add a comment, that's a nonsense request. You can't physically do that. But that's not a failure of the test, or that doesn't mean we've caught a bug. That means our generators produced something that's bad. But in order for the generator to do that, the generator produces the data well in advance of us actually doing something. So, the generator, in order to produce valid sequences of actions as I call them, would involve them having to introspect on all of the stuff that's generated up until that point. Or to have weird stateful generators that remember what they've done.

And it just sounded so horrifically error-prone, I was like, "Nope, nope. Not going to do that." Instead, down in the code that's actually running these things against the browser, it will take a look at it and go, "Huh. That's not a story. Okay. Moving on," and pretend like it very happened. Which means that some of the stuff we produce is not nearly as useful as originally anticipated. But then I can, there's a little… I'm sorry, I keep thinking about people who don't understand or haven't used this before and I keep realizing all of the details I've skipped.

With generative tests, there's a number. And it's the minimum number of test cases that that test must run. And it represents a nice little dial that you have. You can say, okay, I've got a good generator. I've got good tests. And currently I want 10 cases. Well, what if I want a thousand? Turn that number up and it'll run longer. But it will catch more things, if there's anything to be caught. And as a result of my 'some of these inputs are useless' I just turn the dial up a little bit higher. It's just like, "Oh, well some of these things aren't useful so I'll turn it up from 10 to 20. That will compensate for the fact that some of these things are nonsense and are therefore not terribly useful or impactful.

CORALINE:

Ashton, I have a question. I see testing as really having four main goals. Correctness is something you focused on a lot.

ASHTON:

Mmhmm.

CORALINE:

And preventing regression error. It just sounds like generative testing would be good for that. But what about test-first design and most importantly in my mind, documentation? I see tests as a form of documentation from one developer to another. So, how does the process of generative testing, what are the artifacts that it generates are like? And are they of use to the developer that comes after you?

ASHTON:

No, they're not. [Chuckles] I don't recommend using generative tests… so, let's say… I understand that testing's a very controversial thing. And a lot of people have opinions on it that start to approach religiousness. And I don't want to offend anyone on that, because there's multiple ways to go about doing things. I do generative tests as a tool for finding test cases and not a replacement in any way, shape, or form for more traditional unit and integration tests. That's my approach on it. It makes a decent regression net because you can… it's possible to write the test that hammers everything. Go test the world, as it were. Although that usually is as slow as you'd anticipate it being. So, I don't recommend that.

What I do, do is I prefer to write my generators such that when a test failure happens, what it does is it will print out the final… and this is unique to this library. It could be done differently. These generators do what's known as shrinking. So, it's going along. It's trying more complicated test cases. And then it hits a failure. And what it tries to do with the failures is it tries to combine and minify all failures to the simplest possible failure it could get away with. And how it does that inside I'm actually not fully aware of. I just had it work for me. So, and each generator has a concept of what simpler is. So, a vector of things, a shorter vector is simpler than a longer one. So, it attempts to find shorter combinations. And when, once it's done all of that and reached its limits it spits out a thing.

And it's like, okay, here's the test and it failed. Here's the number of times I ran it. Here's the first failure that happened, and it's the actual data that it generated. And here's the simplest failure I found, and the data it generated for that. And I'm very, very careful to write my generative tests such that the data that it generates completely describes all of the action of the test. Because if there's any randomness under the hood of that, you're so screwed. Because you don't actually know why it failed. You just know that it did. And then I'd take that thing.

So let's say, in this case, I found a bug that involved icing a story, taking one story from somewhere towards what we called the icebox in Tracker, and starting another one. That was the bug. And it spit out, okay, that's the two actions. I take that and I go write a unit test somewhere else with that. And that somewhere else can literally be I take that data structure, I copy it into another Clojure file, give it a [name], and run it through Clojure, the built-in unit testing framework. Or I go write it in Capybara in Ruby because that's where we keep our documentation type tests. And therefore, maybe that's where that belongs.

And I can totally see arguments for and against in both ways. And it comes down to, what are you comfortable with and where do you look? But for me, the generative test is purely for finding that test case. And once it‘s spat out that, here’s how you break the system, you pick that up and you scurry away and put it somewhere that you’ll remember where you put it. [Chuckles]

CHUCK:

So…

CORALINE:

Does that mean that you’re generative tests are disposable?

ASHTON:

No. in fact actually, they’re significantly less disposable and changed than our unit tests, because with minimal changes to it I can actually increase the complexity and power of it exponentially. Minor changes to generators produce massive downstream effects as far as the power of this. So, in a lot of ways they’re significantly more permanent. It’s just the output is ephemeral and one-time use only.

JESSICA:

And your generative tests, you’re probably not going to run them as often as these unit or Capybara tests that you’ve created based on failures you’ve found.

ASHTON:

Yeah, exactly. And especially since they can also take, in order to be useful they can take arbitrary amounts of time. And CI time’s not free. [Chuckles] So yeah, exactly. I actually tend to run them as a, I’m going to lunch now. But I’ve a four-core 32-gigabyte iMac. So, I guess I’ll just them on and go eat [chuckles] instead of letting it sit there idle for an hour. And I could see the argument for doing it overnight.

And I would never recommend it for an individual check-in just because… and they’re also, more importantly, they’re also slightly more flaky. A failure in it does not necessarily always represent a failure of your product, because you’re doing all this randomization. You’re literally… so, everything we do in the CI world and TDD and stuff like that to make predictable, reliable tests so that they more reliably reflect success and failure of the product given certain metrics or certain design or certain criteria, everything about generative tests is specifically designed to defeat that. [Chuckles] It is trying desperately to make for the most randomized unique environment possible.

And you do, do certain things to make things reproducible after the fact. But you’re trying to throw a random, you’re trying to throw chaos into the tests. That’s not really a pattern for guaranteed success and failure. That’s a pattern for when we find something, it’s seriously interesting. And I probably wouldn’t have found it otherwise. But it might find something that isn’t real. Or it’s like, every once in a while it will cause problems because we’re doing weird things. So, I definitely wouldn’t recommend it on check-in. But I’m sure there’s somebody out there who will tell me how incredibly wrong I am [chuckles] on that. That’s always been my experience.

CHUCK:

It seems to me that this is something that would pay off a lot more on more complicated applications.

ASHTON:

Absolutely, absolutely.

CHUCK:

So, at what point… is there a rule of thumb that you can go with for the level of complexity of the app or the size of the app where this will really, where you should start thinking about this? And on smaller apps, do you just not worry about it?

ASHTON:

Yeah, I really wouldn't worry about it on smaller apps honestly. It's a big tool with a big lever. And that has its pros and its cons. Anyone who tells you that something is an immediate win without any sort of downsides is either not experienced enough to be making such proclamations or is trying to sell you something.

JESSICA:

On the other hand, I use generative tests for some smaller apps. But specifically, Ashton mentioned not wanting to think about all the edge cases.

ASHTON:

Mmhmm.

JESSICA:

So personally, as soon as I find myself getting bored writing unit tests and just not even wanting to think about all the possible vector lengths and orderings that could be sent into my function, I switch immediately to a generative test.

ASHTON:

Yes.

JESSICA:

So, in that case it's not about the size of the app as much the size of possible inputs that I want it to not fall over on.

ASHTON:

that's a super great metric for it that I wish I had thought of, actually. Yeah no, and that's totally it. Certain problem spaces are super amenable to this and some of them are not in the slightest. If you're just testing a simple CRUD app that doesn't have that many possible states, you're killing a molehill with a nuclear device. It's just not… it's too much. But if there's some tricky algorithm work under the hood, it's a possibility. If it's, the codebase is so big that nobody can tell you for certain how all of it works, or if all the combinations are overwhelming as Jessica said, totally, totally reach for that kind of tool.

For me, it was the fact that these small, little areas we're seeing bugs cropping up. And the team's small enough and the product is big enough that nobody really had the whole state of the world in their head and therefore they needed some help tackling that complexity and getting at that. There are certain things inside of Tracker that also fit neatly within that, the possible inputs are of arbitrary size and complexity and therefore humans are just going to get bored trying to nail down all of the test cases. But…

CHUCK:

So, I also want to push the ROI button a little bit more and ask about just the initial setup to do this kind of generative testing. Is it expensive? Or is it relatively cheap?

ASHTON:

It totally depends on your product. You have to sit and think about it a little hard. But that's a really… that's like saying, what's the ROI on TDD? It's like, well there are 10,000 people who will die for it and tell you their experience. But no one actually really knows for certain. You can have positive experiences, but…

CHUCK:

Mmhmm. Right.

ASHTON:

Some people will do great with it. Some people won't, because they're not used to it and all that stuff. So for example, I believe Jessica's team does a lot of Clojure. So for her team, it's going to be a little bit easier because it's already a tool that the team is familiar with. They already know how to use Leiningen. It's just simply pulling in a library and calling it.

For the team I was on, it involves using Selenium and using all of these extra tools which don't make the world simpler. And for the…

JESSICA:

Yet you got it set up in a week. That's pretty good for JavaScript.

ASHTON:

I got that set up in a week. I found a database and API related bug, I'm sorry, mostly database related bug in one day, one pair day of eight hours. Which was actually kind of interesting.

After seeing, that was the bit I said I was going to skip over. Inside Tracker, every once in a while things go wrong. There's a way that, like any software in the world, there's a thing that reorders stories. So, stories fit inside of Tracker in a very specific order. And every once in a while it would get out of order, very, very rarely. But when that happened, a human being, and it wouldn't reorder right, and a human being would have to go in and fix it. Because 99.9% of the time, when it does go out of order which is pretty rare, the reordering thing works just fine. But 0.1% of 0.1% of the time, it would just get wedged. And we'd have to walk in and fix that.

So, all of the previous generative stuff had been ClojureScript. We sat down with Clojure. And a MySQL driver and an API driver, or an HTTP library which was needed to trigger that particular code path, and hashed it out in one day, and found the simplest case, which was four stories long that provoked the bug that had been haunting us for three years. So, it's definitely not like a huge investment. One day for one pair is like basically the minimal unit of investment. Maybe, well maybe one morning. But it would be a little bit smaller. But it's pretty, pretty close to the smallest. So yeah, it's definitely not massive. The trick…

JESSICA:

Now, you're leaving out the real investment there, which is for my team yeah, we can just bring in test.check as a library. But having my team members understand the concepts of generative testing and getting used to thinking in that way…

ASHTON:

Yes.

JESSICA:

That's another matter.

ASHTON:

Yes. That was actually the biggest problem. So, the biggest problem is parentheses.

JESSICA:

[Laughs]

ASHTON:

So, I don’t know… [Sighs] Who here has actually used Clojure aside from myself and Jessica as I’m aware?

CHUCK:

I’ve played with it, but not…

AVDI:

Or Lisp in general, right?

ASHTON:

Yeah, yeah. Lisp in general, Clojure in particular. And what’s the editor choice in the room? It’s Emacs for me.

CHUCK:

Emacs.

AVDI:

Yeah, we’re mostly Emacs dorks here.

ASHTON:

Okay, sure. So, Pivotal Labs is a RubyMine shop. If you’re in Emacs, there’s a plugin called Paredit or Smartparens. There are a couple of different choices now that allow for you to manipulate parentheses-based code structurally. It sounds crazy. And if you’ve never done it, it’s very tricky to explain it. There are videos. But the point here is the editor keeps the parentheses balanced for you so you don’t have to think about them so much. And when you need to wrap and unwrap and split, it gives you short key commands to do that. Okay, great. If you’re not used to that, that is such a mountain to climb. And I have a great Emacs setup that does that. None of my coworkers use Emacs. So, I can’t drop them in that, because now I’m asking to learn a language, an editor, and, and, and it’s going to be a failure. So…

JESSICA:

Yeah, none of your coworkers are your coworkers anymore. So, now what are they doing?

ASHTON:

That is a more interesting question.

JESSICA:

But sorry, I interrupted your…

ASHTON:

Yes.

JESSICA:

Your story. Go on. So, the parentheses are a thing. Emacs is a barrier because Emacs deals well with the parentheses.

ASHTON:

Right. So, there were other editor choices and there are now multiple things that are cropping up. And I don't want to get into a huge editor war thing. But the point here is nobody was mentally prepared or prepared tool-wise to deal with parentheses at that level. And it's totally doable. There are so many people who are used to it and love it and have converted and that's not a barrier. But if you're tossed into it without expecting to, it's kind of a shock. There's not real way around that.

And it'd be like being tossed… if you never saw Java or C ever, and you lived your life in Ruby, I think a lot of people actually do. And you got tossed into Java out of nowhere or you get tossed into Haskell out of nowhere and suddenly you have all of these different things which other people find perfectly acceptable and perfectly usable and whatever. But suddenly you have to deal with this monad thing to talk to the outside world, you're going to freak out. [Chuckles] You know, it's a big, nasty shock. Or you suddenly have to put semicolons at the end of every single line and all that stuff. It's not a huge deal in the end of it. And there are tools and techniques and mental ways of thinking that get around that and actually make that super comfortable and actually productive. But you know, if that's not the way you've been doing things for a long time, that's going to be a bit of a leap.

So, that was the first level. And then the second level of difficulty was generative testing reverses the world. You start approaching things completely wrong from where you were from TDD. With TDD you start thinking, okay what's the input? What's the output? Okay, cool. Generative testing, you start saying, what's the input look like and what should be true after we're done? [Chuckles] You stop being able to think, what is the actual input? Because you just don't know. And you have to start making strong connections between an input and an output without knowing specifically what in the world they are, or what the world should look like after the input and the output is done.

And that is just so different from every single instinct you pick up in unit testing. And very confusing at first, very strange. But like always, I think changing the way you think, it's useful in some ways and painful in other ways. You know, so…

JESSICA:

TDD was painful when we first started doing it, too.

ASHTON:

Yeah.

JESSICA:

Because if forces you to think in different directions. I think generative testing is even more so.

ASHTON:

Oh, absolutely. It's…

JESSICA:

And even more constructive [inaudible].

ASHTON:

Yeah, yeah. So, the benefit of TDD has always been you start thinking about what is the condition for success before you even get going. That is basically like, what metrics am I going to use? And what tests, for lack of a better word, but what flags will I use to say code good or code bad. And to provide a framework around that, I think is the benefit of TDD. And it forces you to have enough tests so that if you need to do refactors later, you're more likely to succeed. Generative testing, similarly, forces you to start thinking about invariants, things that must always be true in advance.

So for example, I start, me thinking about it and getting around… we had a lot of Capybara tests.

We have a very mature suite of Capybara tests. But [inaudible] start this story, drag this story, the order should be this. Okay, I can't do that generatively. So, what are the invariants for that? It shouldn't blow up. It should continue to be available to talk. And all of the browsers and the database should agree about the state of the world. I don't care what the state of the world is. I can't possibly predict that without re-implementing the application. But everyone should agree, right, on what just happened. If the database thinks the order is 1, 2, 3, story numbers, and the client thinks it's 1, 3, 2, stop the world. Something horrible's happened, and we need to figure out what in the hell just happened. I don't care what the actual answer should have been. But if everyone agrees, we're good. There are other tests to do that. Or, it shouldn't throw up a scrim to the user and say, "Sorry. Can't do anything," or, "Reload," or whatever. That's the invariant.

And looking through the code, nobody had thought about that before. Which is not to say that they aren't smart, because they are smart. It's just that that's not the type of behavior and the type of thinking that TDD makes you do. Whereas generative testing, it forces you down that. And that constraint is useful. It helps you think about the world in a useful way and be creative within the constraints.

Talk to a designer about how much constraints help them. You tell them to make a thing and they're just like, "Well, but that's an infinite set." Okay, you need a website to do this and you need the users to do that, and blah-dy blah. And those constraints guide them into doing something for you and with you. And similarly, the constraints of, you can't know what the data is forces you to think about what should always be true. And that is what guides you forward on that.

CORALINE:

So Ashton, I learned about generative testing from Jessica and I got really excited about it and did a little bit of it myself. And when I went to talk to people about my experience with generative testing, I could see something, a cloud pass over their face.

ASHTON:

[Chuckles]

CORALINE:

They were not really comfortable with the idea. Why do you think generative testing has a bad reputation among some developers?

ASHTON:

I was going to say the cloud goes over people's eyes when I talk a little bit too much no matter what I'm talking about. So, I don't know if I'm [chuckles] the right person to bug on this one. But all jokes aside, that's actually a really difficult and subtle one. I think a lot of it is the familiarity problem. But that's also a little bit of a tricky answer in some ways, because TDD was super unfamiliar and yet it's taken over the world. And Ruby shares very, very little syntactic heritage with a lot of super popular languages. It's not [inaudible], it's not a Lisp. And yet, it's taken over. And same deal with Python. Those two are like, those two were weird and different when they came out.

So, I think probably the best way to think of it, best answer I could give was it takes some agency away from the developer. I think a lot of developers tend to be, I'm trying to think of a polite way of saying this and not being mean to my fellow colleagues. But I think a lot of developers what to drive the computer themselves. They want to feel like they are the master of the machine and of the code. They understand it completely and they can figure it out themselves. And I think the idea that it's too complicated for a human being to tackle in every single scenario takes some of that agency and mastery away from the human, and forces us to face facts that we have built things that are actually too complicated for somebody to know. Which I think is… there are a lot of techniques you can do to try to control that. And we should.

But on the whole, any useful application is just simply, there's too much going on for you to possibly know 100% of it. When you're writing Ruby on Rails stuff, you don't understand how Active Record works under the hood. You just know what your interface to it does. And you don't understand how MySQL works and all that stuff either, per se. Well, some people do. But most people don't. So, I think that's what it is to some degree. Plus the unfamiliarity. And then you also add in that usually it's in a language that people are not familiar with as well. And it's just too much of a molehill to climb, or a mountain, as it were.

My way around that usually, because I'm just assuming that's probably going to be the next question, is I frame it, rather than framing it, our code is too complicated for a human being to attack and understand, to simply say, you don't want to write all of these cases. Okay, you've got this algorithm or this function that takes a vector of things of arbitrary length and ordering counts in the results. You don't want to write all the test cases. You just simply don't want to do it because humans are lazy, to a large degree. And engineers doubly so.

JESSICA:

And I don't want to maintain all those test cases even if you do write them.

ASHTON:

Absolutely, absolutely. So, that's the way I couch it to most people, is like, this is an easier way for it. You don't want to write those. You don't want to maintain those. And they're ugly. So, let the machine do it for you. This isn't about letting the machine take over some of the power. This is about the machine doing the work for you, which is more comfortable, because that's one of the reasons some of us got into engineering, is the ability to make machines do things. And the idea that the machine has to take over is, well for some people maybe not necessarily comfortable.

CORALINE:

So, that covers two of the three Larry Wall virtues of a programmer, laziness and impatience.

ASHTON:

[Chuckles] Yeah.

[Chuckles]

ASHTON:

Yeah, and it takes away the arrogance thing pretty handily, unfortunately. But that's…

CHUCK:

So, one other question I have is, so you use these generative tests. You find some edge case and it's a valid edge case.

ASHTON:

Mmhmm

CHUCK:

Something's broken. How do you get, take those results and turn them into an actual test that then verifies on a regular basis through CI or whatever that the thing's actually fixed?

ASHTON:

Hmm, okay. So, that's a great one. I think that depends on what you're using. So, if you're using generative testing with Clojure, that's going to be… generative testing and your application is written in Clojure, you can probably copy that sucker over into a unit test case. And for me, what it spits out is actually data structures. And the actual, the test itself, if you look at my, the generative testing stuff I wrote, which you can't because it's at my previous employers and they're not going to give it up. But there's like, there generators are fairly big. And there's a whole bunch of code whose job it is to actually execute the results of that.

But each individual test case, because you write an actual thing that looks fairly similar to a regular test case, plus the generator declaration and the number of times that you're going to run it, it calls a function and only one function. There's one entrance point. And then that goes and does all of the, do all of the things, as it were, runs and does the assertions and all that jazz. Which, the reason why I did that is I wanted to make it very easy to be like, "Oh, well I want to add this other generator and run it separately." Okay. It's the generator and any extra code you might need to execute, plus four lines. And I wanted to make it easy to recombine these things cheaply and find out if there's anything useful there.

So, as a result of that, I can take what is spit out in the console, copy paste that into another file or the same file, it doesn't matter, write a non-generative test case, and simply call that exact same function. Good to go. No problems. And if I want to run the same thing again and again and again, and really look at that underneath the microscope. So, if you were in a Clojure shop or all of your testing is in Clojure, which I would highly recommend because the Selenium wrapper for Clojure actually works, but you would just simply copy and paste that around.

For us, because Pivotal Labs is not a Clojure shop and Tracker was written in Ruby, I would rewrite in Capybara just because that's what the team is familiar with. And that's where the team instinct is to go look for it, which is kind of a shame because I don't like Capybara. But you know, sometimes you've got to use what you have to use. And that was totally my process. And I think that was probably the correct thing to do. Despite the fact that it's not the way I would prefer to do things, it was still the right way to do it because it was the best for the product and the team.

JESSICA:

So, if I'm working at a Ruby shop and I want to consider doing testing in this way, one of the things my team leads and upward are going to ask is, what happens if you leave? How would I ever maintain these? So, how are the people at Pivotal going to be able to maintain these tests now that you're gone?

ASHTON:

That was a question that was not fully answered before I left. And now, [chuckles] the question will probably be answered on their own now that I'm gone. I'm not sure, to be honest. I think that's actually… so, the technical side of things, I'm super comfortable with. I can go on and on, clearly, about how that works out. When we hit this bit, that's where I hit the wall and I'm actually not sure at this point.

JESSICA:

Well, if nothing else, they have the test cases that you copied into unit tests

ASHTON:

Yes.

JESSICA:

And that you got value out of those immediately by finding those.

ASHTON:

Yes. And I did as good of a job as I am capable of as an engineer of separating those things out into very clear… excuse me, cleanly delineated units of abstraction and power inside of that. So for example, I use multimethods to actually dispatch to the individual, “Oh, you want to add a task.” Okay, here’s the multimethod dispatch, and I separated those out very nicely which makes it easy to add. And I separated the generators out, which makes it easier to add, and all that stuff. And I do all of this thing to try to get it as precise as possible. But it is entirely possible that it will not survive very much long beyond, after me leaving. And that’s a tricky thing to deal with.

So, I guess part of the answer is you need to actually, if a Ruby team was considering this kind of stuff, you really need to set it up so that everyone is committing to it, to a small degree. Because this isn’t, it can’t be somebody’s curiosity. And those kinds of things are fun and great and we all do it every once in a while. It’ll be like sometimes with managerial blessing, sometimes not. But it’s like, okay, we’ll go have a hack week or something and we’ll go play with this stuff. And that’s all good and well. But if you want it to be part of a serious attempt from the team to be a long-lived thing despite the fact that nobody stays at a company forever, you need to get buy-in from multiple people that they will put effort in it.

And you’ll also need the ability to not add all of the things that you think could be added into that. Because if one engineer always knows exactly how to do it, that’s that engineer’s thing. And it stops being a team project and starts becoming that person’s individual project. And it will never survive beyond that person’s career. So, you have to make it a team effort, as it were. It’s probably what I would recommend and it’s probably what I could have done better, if I had more time and if I could try it all again. I kind of blundered into all of this stuff on accident. And you know, with the joys of hindsight, I would have done that.

And the various techniques like pairing and stuff like that can make it easier. I know that’s also another controversial thing. But that’s one of the things that Pivotal Labs does in general. It’s very well-known for being a 100% pairing organization. Take that as you will. But that actually proved to be super useful in some cases, and also in making people more comfortable with it. So, the one day, finding the three-year-old bug thing, that was totally pairing. That was 100% pairing. And when we finished, my pair who didn’t know Clojure going in and didn’t know Clojure going out, still felt he understood the general high level of what the thing was trying to do. He was like, “Oh, I get it. You generate random stories in random order, dump them into the database, run the reorder routine, and assert that the stories are in business logic order, that they conform to those rules. This makes sense to me. Perfect.”

Even though he doesn’t know Clojure, he’s an engineer. He can learn the language. But he understands what we’re trying to get done. And therefore, he could pick up Clojure and go work on that, should he see something else that fits that need. And then also adding it to CI means that if it ever goes red, you have to fix it or you have to delete it. And that forces the problem to be resolved as well. But I think that’s the interesting question, unfortunately. And it’s the one that’s interesting because I don’t have an answer for it. [Chuckles]

AVDI:

A minute ago, you were talking about what turns people off to generative testing.

ASHTON:

Mmhmm.

AVDI:

And I’ll tell you what has turned me off to it a little bit in the past. I don’t know if turning me off is the right phrase. Made me think that I didn’t need it. And it’s really the functional programming problem at large.

ASHTON:

[Chuckles]

AVDI:

Which is the problem that every functional programming language, you go to the homepage and it shows you how concise the implementation of factorial is.

ASHTON:

Yeah. But no one ever actually implements that. [Inaudible]

AVDI:

[Laughs] And you know, but factorial is right there in the prelude. So, why do I care?

ASHTON:

Mmhmm.

AVDI:

And there was a similar issue for me in a lot of the tutorials that I saw about generative programming, which is that it made it seem very much like it was all about algorithms, it was all about testing algorithms. It's like, here's the generative test that I ran over my quicksort algorithm and I found a bug.

ASHTON:

Yeah.

AVDI:

Well, that's great for you. I'll use your quicksort implementation.

ASHTON:

Yeah. Oh my god.

AVDI:

[Laughs]

ASHTON:

Yes, that is such a huge problem.

AVDI:

But what has fascinated me in listening to you talk today is that it really sounds like by testing the application from the frontend…

ASHTON:

Mmhmm.

AVDI:

Your inputs are actions. Your inputs are high level human actions.

ASHTON:

Yeah.

AVDI:

And it's almost like the algorithm that you're testing is the application moving over the state space, which is cool. And which is also something that I don't see as much talked about, at least in the introductory material for generative testing. That's actually not a question.

ASHTON:

[Chuckles]

AVDI:

As you can tell by the lack of question mark on the end. What I'm curious about, does this change your perception of your application architecture at all? Or do you think differently about architecting your application because you're thinking about actions, human actions, as these composable parts of input?

ASHTON:

So, I think it makes me, I don't actually know how designers think. But I think that makes me think more like a designer, or at least what I [inaudible]…

AVDI:

When I say design, I am talking about application architecture.

ASHTON:

Oh, I am fully aware of that as well.

AVDI:

Yeah.

ASHTON:

I'm thinking, what it's caused my mental picture of things is what I think a UX designer actually thinks. I have no idea how they actually think because I've never done UX professionally. But I think it leads me more in that direction, from a viewing users' actions first and then figuring out the details underneath instead of saying, well I've got Mongo and therefore it has to fit inside of Mongo's model. That's completely uninteresting to me now, because that's a detail. Who cares?

AVDI:

Mmhmm.

ASHTON:

And I think that's, so that's true on one hand. One the other hand, I'm probably also the worst person to ask about this because I'm super opinionated about immutability, simplicity, all of these things. And I won't shut up about them half the time.

AVDI:

[Laughs]

ASHTON:

Ask my ex-coworkers. And so, I've always been changing the way I think about things. And I've always viewed the application in a very, very different way than a lot of other people.

And I think, for example, the biggest split between me and my previous coworkers was they would think about ease. I have no interest in ease. Ease is uninteresting to me, because ease is how easy is it to do a thing, is a matter of familiarity and a matter of experience and a matter of age and a matter of background and what have you. Emacs is easy for me. But that's not, that's a completely un-compelling statement because if I try to put somebody non-technical in front of it they're going to start swearing at me.

I'm interested in simple. And that's something that, and I think that's one of the reasons why I did so well with this generative testing stuff, is it matches my world view. I want a simple application that the user can understand intuitively what it's trying to do. Not an application that they can master once they're familiar with it, which makes it complicated, or sorry, difficult. Or that is intuitively, or is immediately easy but then therefore they end up outgrowing or what have you.

To bring that back a little bit, I always say that Ruby is easy but not simple. It's easy to get going. The syntax is straightforward. Tooling's good, or decent at the minimum, blah-dy blah. And look, you can create your hello world thing in so long. But call a function and tell me where that code's going to go. Tell me where the path of execution is going to go. Tell me how module imports work and all that stuff. All of that stuff's actually shockingly complicated. I want to go the other way. And I want to go the other way with my application.

And thinking at a high level about user actions and generative stuff really changes that. It forces you into that way of thinking. Well, maybe forces, I was already headed that way in the first place, so I can't really assume that that would have led me there. But so, that heads in the direction of, "Oh, I want something to be simple so that the user can intuitively fit it all inside of their head and they don't have to try to feel like they have to become masters of it." It just becomes intuitive.

AVDI:

Right.

ASHTON:

What it's trying to get done. I think the other side of that question though is, does this change how generative testing is actually presented to the user? And I think absolutely. And I think that's actually probably the most exciting thing, is like as you said, generative testing, it's a little bit like, it is a lot like how functional programming likes to present themselves. It's like, "Oh, we'll show you this awesome theoretical thing." But frankly, no one cares, or very few people care. We're all here to get something done. I don't care how accurate your sorting algorithm is, because that's such a basic thing. And it's a thing that you can pop out and fix. Like if the sorting algorithm in Ruby was broken, it could be fixed at some point in the future under the hood, just no problems whatsoever. You'll never notice unless if it's provoking a specific bug. You want to know, does my application work? [Chuckles]

AVDI:

Right.

ASHTON:

I don't care if sorting works. I care if my application works. And that's something that I think, and this is something that I think Clojure does very well, much better than a lot of other functional programming languages, and I think this is something that I'm trying to do with generative testing, is to watch my fellow engineers and say, "Look. You're a professional. You're experienced. We both know you're just trying to get something done. Yes, there are fun side-projects and we've done those before. But we're here to produce a product that does a thing for your end user. I have a tool which will help you." It's not about being smart. It's not about being clever. It's about fixing code and working your way forward.

And if you want to go play with the experimental stuff that's on the forefront and the very edge, if you want to put that into prod, god bless you. Because we need somebody who's trying that stuff out. I sure as hell am not going to do that with my products [chuckles]. But bringing it back to…

ruthlessly practical I think is a good thing to be doing in general. That's something that we all need to do from time to time.

AVDI:

Yeah. I will say that I feel like if I were doing this from the frontend, I would immediately start thinking along the lines of, I'm testing my application as an algorithm over a series of inputs here. Can I move that back from the frontend? Can I start looking at these human actions on the backend side as discreet actions that I can also string together and test as, when you run the application over this series of events, what is the outcome?

ASHTON:

Without diving too deep, and I do have a little bit of a cheat on that, which is that Tracker uses commands under the hood.

AVDI:

Mmhmm.

ASHTON:

It's an event sourcing kind of setup. While the commands are not touched directly by any of my generative tests, it does make that an easy thing to start thinking of a sequence of events that drives state and all that stuff. So, that helps me head in that direction mentally in the first place. So, there's a little bit of a cheat there, if that makes sense.

AVDI:

Nice.

JESSICA:

And hey, if generative testing is something that academics have been using to test their algorithms, then a lot of the fun part of being a programmer is taking all of that academic computer science stuff and using it to make people's lives easier.

ASHTON:

Exactly. Optimizing compilers were a research problem that would never, or a research toy that would never ever be used by practical day-to-day salt of the earth engineers. And now the JVM has a freaking JIT built into it. So, you know, big things move forward. It's a process. It's a thing. And it's a hell of a lot of fun, because I'm driving a Selenium web browser and it's blinking and doing things and moving crap all over the place. I can look at that and go, "Huh. That's really entertaining. Look at it go." But maybe that's just my ADD speaking.

So, I think, it was funny. I was hanging out with a guy at the Cogn who turned out to be the guy who wrote the clj-webdriver, the Selenium wrapper for Clojure. And I didn't realize until after my talk that he actually had made that. And I was just you know, getting coffee with him and chatting and stuff like that. And afterwards, he was like, "Hey, I'm the one who made that," which was cool that he didn't have to toot his own horn upfront. And it was kind of interesting, running into people who've done all that stuff. But he and I think another person were like, I had a thought of possibly doing that. And then I couldn't figure out how to do it. The actual, nitty-gritty, to the bolts, how you wire this kind of contraption together, because you're making something brand new out of a whole cloth here when you're talking about strapping these things together and using them in such a strange way.

So, to my knowledge, one, assuming that there's not any sort of private project somewhere that didn't announce that they'd done this or anything, I think I might have been the first one to do this.

It’s also possible that’s not true. But more importantly, two, this means this is something I think a lot of people have thought about possibly doing. And I’m hoping that having this discussion and talking about it will one, possibly convince more people to try it because I think it’s a fun tool and I really like the Clojure ecosystem. I like the people who use Clojure I’ve met. I’ve had a lot of fun and all that stuff. And it’s always been an incredibly positive experience, one. And I’d be happy if this kind of stuff became more popular. But two, I think there’s people who are thinking about doing this kind of thing, or they’ve already internalized that they need this kind of thing. They just don’t know how to go do it.

And so, I’m hoping this discussion can really funnel people into that and get that going. So, I can stand up here and say, “Look, it can be done.” And it can be done for fairly trivial amounts of time and an engineering effort. I’m not talking about you need to stop working on your product and just make tests for four months. I’m talking about, “Hey, it’s a week.” It’s a week of one engineer or a pair, depending on your style. It’s time which is not a huge ask. It’s totally possible to get it done. I’m hoping that will kick people on the pants and be like, “Alright. Let’s do it.” Or they can stand before their managers and say, “This guy is talking about the bugs, finding a bug that had been haunting them for three years, or finding a user report from someone and finding the minimal reproduction case in 36 minutes instead of a whole day.”

That kind of stuff is I think huge and useful and how we advance forward and do newer more interesting things as a community and as a profession. And I’m hoping that this might trigger some of that, one would hope, at a minimum. I guess at the minimum, I got kind of a cool trip to DC out of it. So, I’ve already gotten at least something out of it.

CHUCK:

[Chuckles] Awesome. Alright, well we should probably get to the picks. But this has been really, really interesting to get to explore mentally. And if somebody wants to explore it in more practice, are there good resources that people can go and pick up?

ASHTON:

I think I’ve written a couple of blogposts. I know that Reid Draper who wrote the library in the first place actually had a couple of good talks. There’s my Cogn talk. And I’ve been mulling over, and I think Jessica and I talked about this earlier, finding someplace that we can go write examples for this for. Because the problem as we’ve covered earlier, even if you are not an academic researcher there’s not really a ton of practical examples. Because by the time you actually reach the complexity that you need, it’s a huge investment. If you want to write tests in the first place, you would need an actual application [chuckles] first. So, and that takes a lot to actually write and display.

So, we’ve been mulling over finding an open source application, like a chat server or something, that we could write tests for that. And possibly, if anyone has any ideas on that. Because by mulling over, I mean I’ve thought about it twice. [Chuckles] So, if anyone has any ideas about that, that would be I think massively useful. And it would put, because I can show you bits and snippets of the stuff I wrote at Tracker. But it’s their code. I wrote it as an employee and therefore it is their code. And I can’t share it. And you can’t run it without the source code of the entire application, which they sure as hell are not going to give up. So, having an example in the public domain or in the open source world will be massively useful. And I’m hoping to do that in the near future.

But there are talks and blogposts and whatnot that will start to point you in the right direction. And I think I have a blogpost in the works, which instead of going high level, here’s where you’re going with it. Details with a little bit more like, here are the small things that wouldn’t fit into the talk that you wouldn’t expect. But don’t do this, do this, what have you.

CHUCK:

Alright. Sounds great. Let's go ahead and do the picks. Coraline, do you want to start us off?

CORALINE:

Sure. I was at RubyConf Australia a couple of weeks back. And there were a couple of talks that I wanted to share that I think are very worthwhile for people to look at. And the thing they have in common is they both talk about how to teach programming to kids. The first talk was by a guy named Joseph Wilk. His talk was called 'Code as Performance' and he talked about using music and generating music with Ruby and Clojure I believe, as a way to teach kids fundamentals of programming, specifically things like concurrency. Because you want music to happen on beat. So, he had a really interesting talk followed by an amazing performance that was both visual and audio. And I think everyone should check that out.

Second talk was by Linda Liukas. Her name might be familiar to you. She kickstarted a project called Hello Ruby, which is books suited for kids between five and eight as a way to teach them about computers and technology and programming. So, she has a talk, talking about her experience with writing the book and what her intended audience is. And it's really, really a fascinating topic. So, we'll post links to both of these talks in the show notes.

CHUCK:

Alright. Jessica, do you have some picks for us?

JESSICA:

One thing that I thought about mentioning during the show that I think was really cool that I heard about at CodeMesh last year in London. There's a generative testing continuous integration tool called QuickCheck CI, which I think is really cool. And it does automatically exactly what Ashton was talking about with the taking tests that have failed and adding them to the regression suite. Now, that's only going to help you if you happen to be writing your tests in Erlang, which you could be testing C code with that. But still, I think it's really neat. And I think we should write something like that for Ruby. Yeah. That's it.

ASHTON:

I was going to say, that's the coolest thing I've heard in quite some time actually.

JESSICA:

Well, there you go.

ASHTON:

It's like that moment of, "Oh, goddammit, why didn't I think of that? Alright, cool."

JESSICA:

Oh, there's a John Hughes talk. I'll post the link if it's online, about this that you can watch.

ASHTON:

Yeah, I'm going to have to watch that.

CHUCK:

Alright. Avdi, what are your picks?

AVDI:

My first pick is going to be a talk that I watched recently. It was a talk by Kevlin Henney back in 2012. And it's called something like, 'Can you even do object oriented programming in Java?' And this is going to sound like a slam on Java that I'm picking this. But it's really not. What I found fascinating about the talk was stuff that was probably almost equally applicable to Ruby or any supposedly OO language. He got into a lot of the, some of the theory of objects and of what object orientation was supposed to be, which is a topic that I've been putting a lot of study into lately. And he brings up some very, very though-provoking points about some of the things that we think of as object-oriented and the way that OO is often taught. So, regardless of the Java in the title, I totally recommend watching this talk.

And I'll also pick a piece of hardware. I've been making an effort to go relatively paperless in my office lately. And one of the things that I bought to facilitate that process was a ScanSnap iX500 scanner. I already had a scanner but this one's optimized for the paperless office thing. It is not cheap and I'm not completely happy with it because for as expensive as it is I would expect it to sort of act autonomously. And in fact, it really works best when paired with a Mac or a PC to run the software that it pairs with. But once you do have that set up, it is crazy fast. What it has in it is a high-end document feeder for a consumer-level device. So, it's able to feed stacks of paper really, really well even if some of them are a little crinkled and some of them are different sizes and stuff like that. It's very good at feeding stuff through without getting jammed.

And it's also, paired with the software, it's very clever about figuring out okay, which way is the text going on this page? Let's generate a PDF with the text going in the right direction. And how big is this piece of paper? Okay, let's not create a giant scan with a tiny little thing in the middle that was actually the receipt that you scanned, and stuff like that. And also again, paired with the software on the PC, it generates OCR PDFs. So, the PDFs that it creates, if you choose to create a PDF actually have the text, as much text as it could recognize embedded in it. So, they become searchable and copy and paste-able and stuff like that. So, not an unreserved pick. But it does seem to be about the best thing in its class. And again, it's crazy fast. So, I've been pretty impressed with that. I think that's it for me.

CHUCK:

Very cool. I have a ScanSnap printer as well. And the thing I like the best about it is the software. I think I had all of you guys do picks, so I'll go ahead and throw some picks out there. First off, I've been working on getting all of the podcast episodes on YouTube. So basically, what I'm doing is I'm pairing up the audio with the album art and then copying it up. And it's been a lot of work. And anyway, the tools I've been using are ffmpeg. And I just wrote a little script that hits our RSS feed, pulls down the audio, and then makes a video out of it. And then I just copy it over to YouTube.

The other thing is I'm really getting into YouTube. And I'm really enjoying some of the features that are there. So, I'm going to pick YouTube as well. Not YouTube for wasting time or watching cat videos, but YouTube as a tool to share content.

And then I've got a few projects going that I want to share with everybody. The first one is something that I've announced. I think I announced it on this show before. But it's Dev Box Club. And basically what it is, is it's a box that you'll get every month. It's a subscription. It has a bunch of stuff in it. You can go get on the mailing list and then I'll announce when I'm actually going to open that up. But I think I'm just going to open it up to 50 spots, and then see what the logistics are as far as putting those together and shipping them, so that I have an idea of what, if I need to change my pricing or if I need to strong-arm some of the vendors to give me better deals or things like that. So, if you're interested in that, go check it out. DevBoxClub.com. That's just for any developer community. But I'm looking at ways that I can provide stuff for people in specific communities like Ruby or JavaScript or whatever.

The next one is last week we did JS Remote Conf which was a JavaScript online conference. And I had a great time. I got a lot of good feedback. And so, I'm going to be putting together a Ruby Remote Conf. And so, I'm going to have that website up where you can actually join. Again, it's going to put you on a mailing list so that when I have the details together as far as when and who, when it's going to be and who's going to be speaking, I can announce that.

And finally, I'm going to remind you about the Kickstarter thing. So, DevChat.tv/Kickstarter. If you're interested in a RailsCasts or Ruby Tapas style video series, then you can get all the details over at DevChat.tv/Kickstarter. Ashton, what are your picks?

ASHTON:

Probably in case no one's ever heard of it, there's a neat little database called Datomic that I find actually pretty fascinating. It's a for pay database. But it's accumulate-only. So, with a small API tweak you can go look at what the database looked like at an arbitrary point in time. So, it makes time as a first level thing. And that opens up a whole bunch of interesting caching and performance things that you could do with it. But I think the most interesting idea of that is it has an API. And I've been toying around the idea of a lot of people need historical stuff or trying out duplicating all application state between that and a traditional Postgres or whatnot, and trying that out. And I've been playing with that on my own. And I've actually found that a real joy to work with. So, that will be my pick.

JESSICA:

And there is a free version, right? For play.

ASHTON:

There's a free version that can do testing. And the biggest selling point is it's written in Clojure and Java so it'd work with JRuby and whatnot. But when you partition one, you can say, okay, here's the storage mechanism that I want. Because it actually uses an external storage for all of its historical stuff. So, you can back it by DynamoDB. You can back it with Postgres. You can back it by Cassandra [inaudible] Cassandra, Riak or Infinispan or whatnot. But there's an in-memory one for testing. And the experience of testing something that is in-memory but is the exact same code-wise and therefor you don't need to guard against, "Well, we're using this in testing and not in prod," is the most sublime experience in the world. Because it's just fast, but it's the same thing. So, I would recommend people check that out and play around with that a little bit.

CHUCK:

alright, cool. Well, thanks for coming.

ASHTON:

Oh yeah, thanks for having me.

AVDI:

Yeah, thanks a lot.

CHUCK:

Bent my brain in a way that it hadn't been bent for a while.

ASHTON:

[Chuckles] I'm going to take that as a compliment. I'm not sure if it was. But I'm going to take it as such.

CHUCK:

It is, definitely. Those are the ones that I enjoy the most, are the ones that make me think. Alright, well we'll wrap up the show and we'll catch you all next week.

[This episode is sponsored by WatchMeCode. Ruby and JavaScript go together like peanut butter and jelly.

Have you been looking for regular high-quality video screencasts on building JavaScript done by someone who really understands JavaScript? Derick Bailey's videos cover many of the topics we talk about on JavaScript Jabber and Ruby Rogues and are up on the latest tools and tricks you'll need to write great JavaScript. He covers language fundamentals so there's plenty for everyone. Looking over the catalogue, I got really excited and can't wait to watch them all. Go check them out at RubyRogues.com/WatchMeCode.]

[This episode is sponsored by MadGlory. You've been building software for a long time and sometimes it's get a little overwhelming. Work piles up, hiring sucks, and it's hard to get projects out the door. Check out MadGlory. They're a small shop with experience shipping big products. They're smart, dedicated, will augment your team and work as hard as you do. Find them online at MadGlory.com or on Twitter at MadGlory.]

[Hosting and bandwidth provided by the Blue Box Group. Check them out at Blubox.net.]

[Bandwidth for this segment is provided by CacheFly, the world's fastest CDN. Deliver your content fast with CacheFly. Visit CacheFly.com to learn more.]

[Would you like to join a conversation with the Rogues and their guests? Want to support the show? We have a forum that allows you to join the conversation and support the show at the same time. You can sign up at RubyRogues.com/Parley.]