Kief Morris on Infrastructure as Code

Sven Johann

**INFRASTRUCTURE AUTOMATION**

with domain-specific languages is a key DevOps practice. In episode 268 of Software Engineering Radio, cloud specialist—and author of *Infrastructure as Code*—Kief Morris and I discuss the benefits of infrastructure as code, including security, auditability, testing, documentation, and traceability. Portions of the interview not included here for reasons of space discuss why you should treat your servers as cattle (not pets), as well as how to introduce infrastructure as code to organizations.

The SE Radio podcast now offers full transcripts of new shows thanks to a generous gift from InnoQ. This will make the show available to people who prefer to read rather than listen and will provide an enhanced experience for those who want to do both. Please visit our archives at se-radio.net or via RSS at feeds.feedburner.com/se-radio.

**Sven Johann (SJ):** What's infrastructure as code?

**Kief Morris (KM):** There are a lot of ways to answer that. One is that automation is the “CALM” of DevOps. CALM stands for culture, automation, learning, and measurement. *Infrastructure as Code* is about the automation piece. That’s how people who have been doing DevOps for a while approach it, using tools like Chef, Puppet, Ansible, and SaltStack. The philosophy behind this is that infrastructure has become like data; the physical layer has been abstracted. It’s become software, as opposed to being a physical thing. We can use infrastructure tools the same way we use software. We can bring in best practices from software development, such as continuous integration [CI], test-driven development, and continuous delivery [CD] version control systems, and apply them to managing our infrastructure.

**SJ:** Is there a difference between infrastructure as code, programmable infrastructure, and software-defined infrastructure?

**KM:** Programmable infrastructure, software-defined infrastructure, and software-defined networking are the tools that enable us to program our infrastructure on cloud platforms like AWS [Amazon Web Services] or OpenStack. These platforms all have an API. Infrastructure as code expresses the philosophy of putting your software in definition files and version control, running automated testing against it, and progressing it through environments.

**SJ:** Where did this idea come from? I heard the term “programmable infrastructure”
SJ: Why is infrastructure as code important? What are the benefits? I know a lot of people who like configuration but are afraid of code.

KM: The obvious thing is saving yourself time and work by not doing routine, repetitive things. That’s what machines are for, right? And then there’s the reliability. You want to make sure that every time you install a new webserver or application server it’s done the same way. Before infrastructure as code, we used to have a checklist. But there would always be something a little bit different, and the person doing the checking would have a trick to make it a little bit better. Or maybe a new update to the system comes out, and there’s a new option. So the checklists were never quite up to date. [With infrastructure as code], you don’t have to rely on a particular person doing it the same way every time; you just put it into a file. Anybody can look at that file. It should be easy to understand, and there’s no ambiguity as to what happens.

SJ: What are some other benefits?

KM: There’s testing, which helps people who are concerned about [the risks of automation] and worry that things can be done wrong automatically. Automated tools allow you to do a configuration update or patch a whole bunch of machines all at once. But it can also allow you to damage a whole bunch of machines all at once, to misconfigure many things very quickly.

The idea of testing—particularly automated testing in a test environment—isn’t something we do as rigorously as systems administrators insist our developers do. We don’t give our developers root access to our...
production machine to make code changes directly, but that’s how we tend to do it ourselves on our server configurations. We might have a test environment where we make a configuration change manually and test it to make sure it’s okay, but then we log onto the production server and make the change manually. There, we might make a mistake.

When we have the server configuration in files that are under version control, then whenever someone makes a change to one of those files and commits it to the version control system, we can trigger a CI/CD server (like Jenkins) to apply those changes to a test environment that runs automated tests. This way, we should catch a mistake very quickly. We can also make sure that the environment we’re testing against is exactly the same as production because we’re using those same scripts, and we know that there were no changes made otherwise.

SJ: Does this approach also improve security?

KM: It improves security because everybody can see the code. You can also put in automated tests to check for basic things like open ports and accounts that don’t exist. There’s a pitfall, though. When you have all of your configurations in a version control system, and all your servers are built from a CI or a CD tool, it becomes a very juicy point of attack. If I can get into your version control system or your CI/CD tool, I own you.

I was at a talk a few weeks ago at QCon with a penetration tester who was explaining how she went into different networks. What scared me was that her first line of attack was looking for a CI server. It wasn’t because she wanted to monkey around with the code and insert backdoors; it was because the passwords are in there. It’s so easy to get inside someone’s private network. Then you find their Jenkins server, and there’s the password for the production database. You need to take that issue seriously and focus on not opening yourself up.

SJ: Any other important practices?

KM: Promotion is a good one—moving things from one environment to the next. That’s something that people [often] do naively. I’ve seen people have a separate Terraform for each environment, and they manually edit each one.

SJ: What is Terraform?

KM: Tools like Puppet or Chef can configure a single server. Terraform is a tool from HashiCorp that lets you define an [entire] environment from code in a file. It works across different [cloud] providers like AWS, or you can use it with VMWare. You say, “Here’s what my application server cluster looks like: It uses this server image. It has these networking rules between them.” You put that into the Terraform file, run the [Terraform] tool against it, and it creates or updates an environment to meet those specifications.

People sometimes have a separate Terraform file for each environment. But what you should do is parameterize a single file that says, “Here’s the environment for my application.” [In that file], you’ll have parameters that you set for things that vary from environment to environment, such as cluster size. Obviously my production environment is bigger than my test environments. You push that file through each environment, making sure it’s running and has been tested in the test environment. Then it gets pushed to staging before it gets to production. The same file is used in each environment, so you know they’re consistent.

SJ: What problems do you encounter with infrastructure as code?

KM: Like any application code, over time it becomes a bit unwieldy, bloated, and difficult to change. As with software, thinking about how to make it more modular really helps. That leads to thinking about what you do with a monolithic application. These days, you’re likely to put [a monolithic application] into microservices, which are independent, loosely coupled, and focused.

You can do the same thing with your infrastructure. If you put everything (my Elasticsearch cluster, my webservice cluster, my database server, my Logstash) in one massive file, you end up being afraid to touch that file because so many things might break. The goal is to break that down into separate files. You have to think about dependencies between those files and managing them in a loosely coupled way.

SJ: What other benefits does this approach have?

KM: One of them is that it’s easy to match up dates and changes. I use the example of when one of these high-profile security holes comes out, like Heartbleed, and the CIO [chief information officer] of a company reads about it in the media and asks, “What are we doing about this?” If you’re doing things the old-school way, with servers that are organically grown and managed, the response is usually, “We’re going to
pull people off of other projects, and over the course of a couple of weeks, we’ll have everything all patched up.” But if you have everything under infrastructure as code, the answer will be, “It’s being done right now. We’ve got the patch. We rolled it out to the test servers and verified that it’s all right, and fixed a couple of the incompatibilities with applications. It’s under control, and there’s no impact to other projects.”

SJ: Other benefits are traceability and auditability, right?

KM: Absolutely. This tends to be important for larger organizations that have regulations or need a higher level of control. In healthcare, for instance, sometimes you need to make sure you have rigorous control over changes. It’s funny because we talk to clients who are a bit terrified of changes. It’s funny because we talk to clients who are a bit terrified of changes. It’s funny because we talk to clients who are a bit terrified of changes. It’s funny because we talk to clients who are a bit terrified of changes.

People might take shortcuts. And things change, so people change how they do it without necessarily updating the documentation.

But if you say to the auditor, “This is the script. These are exactly the things that are done every time. And here are the logs. The server shows when this change was made, along with their companies are permitted to post the accepted version of IEEE-copyrighted material on their own web servers without permission, provided that the IEEE copyright notice and a full citation to the original work appear on the first screen of the posted copy. An accepted manuscript is a version which has been revised by the author to incorporate review suggestions, but not the published version with copyediting, proofreading, and formatting added by IEEE. For more information, please go to: http://www.ieee.org/publications_standards/publications/rights/paperversionpolicy.html. Permission to reprint/republish this material for commercial, advertising, or promotional purposes or for creating new collective works for resale or redistribution must be obtained from IEEE by writing to the IEEE Intellectual Property Rights Office, 445 Hoes Lane, Piscataway, NJ 08854-4141 or pubs-permissions@ieee.org. Copyright © 2017 IEEE. All rights reserved.

Abstracting and Library Use: Abstracting is permitted with credit to the source. Libraries are permitted to photocopy for private use of patrons, provided the per-copy fee indicated in the code at the bottom of the first page is paid through the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923.