**HashiCorp Terraform**

Hi, my name is Armon Dagar and I want to talk a little bit about HashiCorp Terraform today. Terraform, if you've never heard of it, is our tool for doing provisioning. When we talk about provisioning there are two different problems we're talking about. As we provision, there is our day one challenge, where we haven't started running anything yet. How do we go from running nothing to running something? Then there's our more challenging day two plus, which is, great, day one, we had nothing, and we provisioned our initial set of infrastructures, and then day two, we have an existing set of infrastructures, but we're trying to evolve it. We're changing it over time. We're adding new services, removing services, and generally evolving the way our infrastructure looks.

How does Terraform solve this problem? We fundamentally believe in taking an infrastructure as code approach. What I mean by that is we allow you to declaratively define, in sort of a set of Terraform config files what you want your infrastructure to look like. In something that's very simple, and human readable, you describe your overall typology. What I mean by that is when we talk about infrastructure, there are many moving pieces, I might have a VPC, or my core network, on top of that I might provision my security group rules, the way my network security is set up. Then within that environment, I define my set of virtual machines. I might have some VMs that I want to provision. Then overlaid on top of that, I have a load balancer.

You can imagine how this graph extends over time. I add different pieces, different amounts of complexity to it, and I incrementally evolve my infrastructure day-over-day, week-over-week. While this is graphical in nature, the way we capture it in the Terraform configuration is very lightweight in describing high-level resources. We describe the VPC resource and the fields that are required to create this element of our infrastructure. Then as we define these other things, we can reference the other components of our infrastructure.

What Terraform then does is gives us a workflow for creating and managing this. Ultimately, this is three different main commands that we run. The first command is Terraform refresh. What Terraform refresh does is reconcile what Terraform thinks the world looks like, so Terraform's view of our infrastructure, with the real world. How this works, is Terraform goes out and talks to the real world. It'll query our different infrastructure providers, whether that's VMware, or Amazon, or Azure, and ask them, "What's actually running?".

In this way, Terraform gets an up-to-date view of what our infrastructure is. Then we run a plan. A plan is where Terraform figures out what it needs to do. You can think about a plan as reconciling sort of the real world, what things are running, with what we want to be running, our desired configuration. When we talk about Terraform's configuration, what we're putting into this graph, this is our desired state of the world, this might not be the world as it is. As an example, on day one, when we execute this, nothing is yet running. When Terraform runs the plan, it realizes that there is nothing yet running in the real world. To get to this desired configuration, Terraform needs to boot all four of these resources, and this becomes the plan.

Terraform will give us an output that says, "To bring the world to look like what you want it to be, I'm going to go boot these four things". Then the last part is doing this. This is the apply phase of Terraform's execution. When we apply Terraform's execution, it starts with our plan, of what we need to do, and it goes out and executes it against the real world.

In doing this, Terraform will go out, and figure out what is the right order in which these need to be done. There's a natural dependency here. The network must be defined first before we can define the security groups around it. Once those are defined, we can boot all our VMs, whether we're booting one, or booting hundreds of these in parallel, and Terraform will figure out where has that opportunity to create things in parallel, and where it needs to be sequential. Once these VMs are available, then Terraform can lastly create our load balancer, and say, "All done".

This gives us a way to do our day one infrastructure. Day one, we describe what we want our universe to look like, we ran Terraform refresh, it said there was nothing existing, plans that all these things are going to be created, and then apply goes out and builds it. What now happens on day two, is we start evolving our infrastructure. Where this was our infrastructure day one, day two we might come in and say, "I really want to DNS record that points to my load balancer. I want to have a content distribution network like Fastly setting in front of that. And I want my VMs to be connected to a monitoring system.". I might monitor these different VMs.

Now, we change our Terraform configuration, we enhance it by adding the definition of these other elements to our TF config, and we rerun the exact same workflow. When we run refresh, Terraform will realize these first four resources exist, but these new ones do not. When we run our plan, it'll tell us that nothing needs to be done with these four. They already exist, and they're in the desired state, but that we must add three new elements to get to where we need to be. Now when we run Terraform apply, it goes ahead, and talks to the real world and adds these three new things and brings the world into looking like our desired state.

The advantage of this workflow is a few-fold. One, our day one experience is identical to our day two. This really is important, because day two is sort of forever, this is something we're going to keep doing. Then, the other advantage of this is something we skipped, which is day N, when we decide we might need to decommission this, we don't need this service anymore, we don't want this infrastructure, maybe this was just a staging environment, or a test environment, we then have the option to come in and destroy.

When we do a destroy, it's basically an unwinding of everything that's been done, right? In some sense, it's a specialized version of apply, which really looks at ... creates a special destroy plan, and then goes out, and talks to the real world, to destroy all the elements that exist. Now we can efficiently, at the end of our life cycle go from nothing to something, evolving it day-over-day, and the final day when we need to decommission it, Terraform knows how to clean up all the resources associated with that application.

So far everything we've talked about is really about the workflow, this is not specific to any technology, so how does Terraform make any of this stuff work? This is where Terraform's architecture becomes important. When we talk about the way Terraform works, there's sort of two major components, one is Terraform's core, so the monolithic core is responsible for a few things. It takes the Terraform configuration, which is being provided by the user, and then it takes Terraform's view of the world, or Terraform state. This is management by Terraform itself.

These get fed into the core, and the core is responsible for figuring out what is that graph of our different resources, how did these different pieces relate to each other? What needs to be created? What needs to be updated? What needs to be destroyed? It does all this central lifecycle management. On the backside, Terraform supports many different providers. Providers are how Terraform connects out to the rest of the world. These can be things like cloud providers, things like AWS, or Azure, or GCP. They could be on-premises infrastructure, so things like OpenStack, or VMware, but it's not restricted to just infrastructure as a service. Terraform can also manage higher level things, platforms as a service such as Heroku, or Kubernetes, or things like lambdas. These are higher level than what we would traditionally consider infrastructure as a service, more in the realm of a platform as a service.

The other types of things Terraform can manage are pure software as a service, very high-level things, maybe monitoring, like Datadog, or CDMs, like Fastly, or even things like managing GitHub teams. These are what we'd consider pure software as a service, these are externally managed services, but the view that we take as Terraform authors is that all these pieces are connected. When I talk about a modern infrastructure, I might be composing all these different resources. I might have some set of my infrastructure that is infrastructure as a service, and then I might compose that and say, "I'm going to provision a Kubernetes cluster". On top of my Kubernetes cluster, I'm going to define a namespace, and I'm going to define services.

These now exist on top of infrastructure as a service, and I might compose even these with higher level things like DNS, and CDMs. All of these are important pieces of our infrastructure, we can't deliver our application without the iOS, without the platform as a service, and without the higher-level software as a service, they're all part of our logical end-to-end delivery. What we want to get to with Terraform, is a single unified workflow. We don't want it to be, you use one tool to manage this, and now you must figure out how you get past this disjoint experience to manage the next section, and then another disjoint experience, and you must piece together all these things.

Instead, anything that really has an API, and has a lifecycle associated with it, how can we enable that to be a provider, such that Terraform can help us solve this problem end-to-end? This is really where providers come in. Today we have over 100 providers, and those can individually manage over 1000 different resources, where a resource can be an AWS, EC2VM, or an Azure Blob Store, these different types of resources are anything that a cloud, or even these higher-level things expose.

That list is constantly growing. Terraform is a massive opensource project with thousands of contributors, and so every day there's new providers and new resources. How does this really start getting used at different levels within teams? This becomes an interesting workflow question around how you manage Terraform. Initially, this goes back to that core workflow we talked about. We start with a single individual practitioner, who's using Terraform locally, and their workflow is to start by writing some Terraform configuration, then they run Terraform plan, plan tells them, "What is this configuration going to do? What does Terraform think needs to be done to apply these changes?" If this looks good, the practitioner will apply the changes, and then the cycle continues, they continue to evolve the infrastructure.

This might look very similar to software development, where we're writing some code, we're applying unit tests, we're committing those changes, and then we're continuing our cycle. What happens as we go from a single individual to now; we add multiple team members? We have other people we want to collaborate on managing this same infrastructure. Well, there are a few challenges, much like much with software writing, which is how do we make sure we have a consistent view of what the configuration is, and how do we make sure we don't step on each other's toes, and run multiple changes in parallel? If we both try and add five new VMs, how do we avoid booting 10 new VMs instead?

This problem in some sense ends up being very similar to the problem of using git locally, versus using Git as a team. As we go to a team, we use a system to GitHub, to provide that central collaboration. Our equivalent of that is what we call Terraform Enterprise. Terraform Enterprise workflow augments this a little bit. What we would do, instead of now running it locally, and running these coordination risks, as an individual, we're still writing our Terraform, and planning locally, but now, we're pushing that into a version control system. This could be GitHub, could be Bitbucket, could really be anything, but the idea is how do we move to having a central repository where we're managing this, and coordinating much like we'd do for source control. Then we drive off this, our Terraform Enterprise application.

The goal is here is a few-fold. One, is how do we make sure the state management is done locally? Like I said, Terraform keeps track of all the resources its provisioned so that it knows at destroy time what it needs to tear down, or as we incrementally make changes, what already exists. This state file is an important thing to have centrally managed, so it doesn't diverge or end up with different lineages where we fork off, and then independently run Terraform, and now the state files conflict with another. By managing that centrally, we avoid those sorts of conflicts. The other important advantage here is when we make sure we're only doing one run at a time. We ensure there's the sequential application so if Alice and Bob are both working on the same infrastructure, and they both try and make a change, they don't step on each other's toes.

Lastly, one of the challenges is often, in this environment, developers are using local variables to populate things like AWSI Keys, or other sensitive credentials. We don't want these sensitive credentials to be strewn about on all our developers' machines. Instead, if we want those variables to be kept centrally, and encrypted, so that we don't have to worry about these things getting exposed, or leaking out in plain text, we can put them in a central place, and version control, and access control who can modify, and view these things.

This sort of changes our workflow, and now enables a handful of people to collaborate safely together. Well, what starts happening as we start wanting to add even more people? For example, we might be going from our early operators, who are very familiar with the cloud environment to exposing it more to our development groups. This is where Terraform's notion of a module comes in handy. A Terraform module is sort of like a black box of infrastructure, where what we can define as a set of input variables that let us toggle the behavior of this module and the module might emit a set of output variables.

An example of this might be, I might define a module for doing Java, for deploying my Java application in AWS, and my inputs might be what's the path to my jar, and how many instances of my Java application do I want? Inside this module, we can operate as a black box, and deploy VMs using EC2, we can deploy lambda functions, we could deploy containers with ECS, all of these become a detail hidden within this module that a consumer doesn't need to know about. Their output can just be what's the address of the load balancer that hits my application? In this way we encapsulate that complexity and make it more accessible to a broader audience.

How do we manage and share these modules? In general, we publish these things to a central registry. There's a central registry that exists today for Terraform if you go to [registry.terraform.io](https://registry.terraform.io/), that's our global public registry, and there you'll find hundreds of modules that cover common forms of infrastructure, common architectural patterns, and many of these are published by the cloud providers. You can log in and see Google's official recommendation for how to provision their network or Azure's official recommendation for how to manage a virtual network inside of Azure.

These modules make it much more consumable for us to operate in these environments, 'cause we don't have to know every detail of it. We can just go and say, "I want to network in Amazon, or Google, or Azure," and fill in a few of the inputs that matter to us, and let the module take care of the rest of the complexity, right? This same idea can be extended to different applications, different types of infrastructure.

One way we can apply this same pattern internally to our organization is by running a private registry. Instead of using the public registry that's available globally, we can run a private registry within our organization. This is a capability integrated into Terraform Enterprise. What this really lets us start to do, is make the split between the set of people internally who are publishers, and are familiar with our internal infrastructure, and how it should be managed, and cloud environments, and they can push in to basically a catalog, or a module registry, all the different key patterns that are needed.

We might publish, here's how we do a Java app, here's how we do a C# app, here's how we do a database and expose this out to our internal developer community. Then, our much larger base of consumers can simply go into the catalog and pull out the things they need. These consumers can know much less about how the cloud works, but still get these approved modules that are sort of the best practice internally, right?

There's sort of a few ways of consuming this, for people that are more familiar with Terraform, they can consume it just like a normal Terraform module and do an infrastructure as code approach. For folks that are less familiar, they might want to take more of a whizzy-wig approach, and they can do that with Terraform Enterprise by selecting their modules, filling in the different variables, and basically letting the system automatically generate the equivalent Terraform for them.

They can either use that equivalent Terraform as a starting point, and customize it further from there, or if they have really no interest in learning, and using Terraform directly, they can just use the auto generated Terraform, and self-service their infrastructure. Our goal really becomes how do we enable this large audience of consumers to get to self-service on infrastructure in a way that is still going to be compliant with how the business wants to do it, and not create this immense workflow to publishers to constantly generate custom infrastructure, and custom definitions for the vast body of consumers within our organization.

At a snapshot, this is Terraform, this is how it applies in infrastructure as code approach, to provisioning, really looking at both the day one challenge, and more importantly, and more challengingly, the day two plus challenge. Then the natural journey as we go from a single individual using Terraform, to a small team, to kind of an entire organization who's trying to leverage Terraform for infrastructure management. If you're interested in learning more about Terraform, or Terraform Enterprise, I recommend checking out our online resources. Thank you.