ngworx.ag

From Dev to Ops

August 2019
written by Guest Author

When we became a Google Cloud Platform Partner earlier this year we thought about which services we should move to GCP.

We had to consider a lot of factors for each service, such as availability, reachability or delay but also non-technical aspects such as protection of data privacy. Of course, we also wanted to gain more than we pay for. Therefore, we also considered reducing our amount of work while maintaining the same level of functionality or even improving it.

The first thing we decided to move to GCP was our DNS infrastructure. Google provides a nameserver infrastructure with a 100% SLA called Cloud DNS. More than we could ever provide on our own, at a good price point.

Slow Ops from back then

We currently own quite a few domains, as is the case for most companies. Our current setup does not provide a lot of automation. So, every time a new subdomain or host is added, a record needs to be manually added to every domain. With a growing number of domains and records, the required work will grow with a complexity of $\Theta (n * k)$. Being a developer by training and heart I see a linear problem, with a suboptimal implemented solution.

While we are required to define every domain, there is no need to redefine the records. Furthermore, we would like to have a traceable history of changes. And the documentation should also be mapped to the real infrastructure whenever it changes.

Coding for a brighter future

For the first problem, git is the go-to tool in a developer's world. It allows to trace changes and when commit messages are properly used it also allows to understand why changes were made. If needed rollbacks to a certain state can be done. Tags allow us to easily identify certain states. Furthermore, the git workflow using branches and merge requests for changes forces the team into shared ownership. No single person is furthermore able or needed to implement changes and knowledge is implicitly shared among the team.

The second problem is also well known to a developer and there is a wide range of tools to solve it. We use Jenkins to automate our build chains. Jenkins, with its plugins, provides a feature-rich ecosystem built to enable continuous integration and delivery for software developers. By integrating with the version control system in place Jenkins can pick any change made and act on it in a timely manner. In the current situation, we will use a pipeline build job to automatically deploy our changes whenever there is a change in the git repository.

To close the gap between Jenkins and GCP we use Terraform. Terraform is a tool that provides the foundation of infrastructure as code. It has a backend provider for GCP which is developed by Google. GCP beta features are available by the GCP beta provider. With Terraform we can define our infrastructure in plain text files. Terraform will take care of calculating and executing the needed changes in our current infrastructure to transform into the desired state.

Reuse rather than repeat

First, we built a module that creates our DNS entries as needed in a reusable manner. We have several domains which all should allow reaching the same subdomains, as stated before. Therefore, the module requires two input variables. The domain name, which is used to create the corresponding zone within Cloud DNS and a list of records that are added to the zone. Thereby we only need to define the list records once within a local variable in our Terraform files. Afterwards,

we can apply the same list to all domains by referencing the list. If a zone needs an additional record, we can just add them with Terraform built-in functionality to join lists.

Next, we can write down our DNS configuration, reducing the work as much as possible by using our module. Within our build system, we use immutable infrastructure. This means that we spawn build agents as needed and destroy them as soon as their job is done. Terraform depends on knowing the current state when computing the needed changes. To solve this issue Terraform allows storing state outside the current working directory, in our case a GCP bucket. Thereby the state can be shared among multiple persons or systems, allowing no single point of failure. Furthermore, we can encrypt the stored information with 256-bit AES and thereby ensuring that no information is leaked. While that is not a big issue with public DNS entries, it is a security concern when you provision other parts of your infrastructure with Terraform. For example, when using Cloud SQL, the state would include the password to your database. Also, encrypting your shared state is best practice and there is no reason not to follow best practices.

Once everything is written down in Terraforms language as needed, we can build our pipeline. Following Terraforms workflow we need three stages. First, we need to initialize our environment. Terraform needs to download the needed providers. This needs to be done in every run as the environment is destroyed after each run. Furthermore, we validate the given Terraform files. Should the validation fail the committer of the changes is notified and the job will abort. In the second stage, we plan the changes needed. To have an additional layer of security we then send a notification to our infrastructure team via e-mail and wait for their approval before any changes are applied. One engineer needs to follow the send link and continue the job by a click on the Jenkins web interface. For convenience, we also send an e-mail to the infrastructure team once the changes have been successfully applied.

Conclusion

Within this article, we outlined the simple process of moving DNS infrastructure to the cloud and how to apply development tools automation in operations. With less than 200 lines of code, we were able to turn our historic DNS operations into a highly automated, safe and secure process. Using Cloud DNS we furthermore reduced our costs while increasing our availability, performance and ability to scale.

Changes to the existing infrastructure are now visible and traceable. Following a git workflow,

changes will only be applied after a second pair of eyes has had a look at them. Thereby, shared ownership is implemented in our infrastructure team, increasing the truck factor as a side effect.

Terraform and Jenkins allow us to reduce the complexity of the process while ensuring that the documentation and current state always match up. Rather than filling a knowledge grave, we are now certain that our infrastructure matches the documentation. Changes to either one do not get unnoticed.