Caltech Post Graduate Program Devops

Capstone project

Bent Vader

cloudctrlschagen@gmail.com

0031657614504  
  
<https://gitlab.com/lab259/kube>https://github.com/linuxjefeprive/capstone-eks-project/

Pictures included in “screenshots” folder. (Folder is tarred, Gitlab files are also included in kube-main.tar.

This will be my final project for completing the DevOps program offered by CalTech and Simplilearn. The project I choose to work on is the following;

**Implementation requirements:**

1. Create the cluster (EC2 instances with load balancer and elastic IP in case of AWS)
2. Automate the provisioning of an EC2 instance using Ansible or Chef Puppet
3. Install Docker and Kubernetes on the cluster
4. Implement the network policies at the database pod to allow ingress traffic from the front-end application pod
5. Create a new user with permissions to create, list, get, update, and delete pods
6. Configure application on the pod
7. Take snapshot of ETCD database
8. Set criteria such that if the memory of CPU goes beyond 50%, environments automatically get scaled up and configured

**The following tools must be used:**

1. EC2
2. Kubernetes
3. Docker
4. Ansible or Chef or Puppet

**The following things to be kept in check:**

1. You need to document the steps and write the algorithms in them.
2. The submission of your GitHub repository link is mandatory. In order to track your tasks, you need to share the link of the repository.
3. Document the step-by-step process starting from creating test cases, then executing them, and recording the results.
4. You need to submit the final specification document, which includes:

* Project and tester details
* Concepts used in the project
* Links to the GitHub repository to verify the project completion
* Your conclusion on enhancing the application and defining the USPs (Unique Selling Points)

I want this project to be my best one, so I discussed with my teacher about how free we were in choosing tools to use with this project. The teacher told me it was okay to use EKS, and “personalize” the way we do things, as long as the same principles as listed in the requirements are met.

This lead me to decide to work on this project like it was code that would actually be used in production. Because of the high requirements for a production environment, I sometimes divert a little from the “simple” requirements listed in the assignment, because there are better and more elegant ways to do things. (Requirements are still met ofcourse.)

My goals are as follows;

Register a domain, set up DNS, and create a bucket to save state used with Terraform.

Use Terraform to create an EKS cluster, networking, load balancing, security groups, IAM policies, 2 master instances, certificates, and recipe’s for spot instances.

Set up a gitlab pipeline connected to the EKS cluster, this gives us major benefits over simply backing up the ECTD database, since all and every change to the app is saved in gitlab, gitlab automatically tests our code, it automatically tests deployment, after which we can simple click “deploy” to actually deploy the app to our cluster.

Note; Everything is automatic with this code, all we do is set up a Route53 DNS and bucket to save state. After running the terraform code we connect our gitlab pipeline and we end up with a working, load balanced app exposed on our domain name. (capstone.unixinfradevelopment0.click).

In terms of app I decided to go with NGINX because it is easy to work with in terms of testing the load balancing and metrics. This was allowed.

After our app is exposed and fully working we do the following;

Use kubectl to connect to the EKS cluster, install metrics-server, set up kubernetes autoscaling, open port 4443 from nodes to cluster and the other way around (using AWS, I forgot to put this in the terraform code, and at the time of writing this I didn’t have the time to readjust this yet, will do asap. But for the project it doesn’t really matter to do it manually by adjusting the security policy. )

After opening port 4443 and deploying metrics we can set up the autoscaling, i’ve added photos that show the pods scaling out under load. After several minutes of low load the pods scale down again.

This completes all the requirements for the project. Because of lack of time I have not yet finished the AWS autoscale deployment, but the basis for this is already in the code. I will be fine tuning the code in the future so not only the pods scale out, but spot instances get deployed when the nodes themselves get under a lot of stress. This is the last part I will need to add to the code (and add port 4443 automatically via Terraform) to make this a true one click deployment of a complete Kubernetes cluster with HPA and CA applied. Everything exposes on the DNS address automatically.

You will find my code added in this document, I’ve also cloned my Gitlab files, for easy verification. Also included is my github project. The code provided has been commented. All variables have been seperated from infrastructure code for easy management.

Steps to apply my code;

1 Go to **AWS route53** webpage, andset up a domain. Also set up a subdomain you want the service exposed at. In my case the main domain I chose is unixinfradevelopment0.click, with the subdomain being **capstone.unixinfradevelopment0.click**. Please take extra care of making sure the main domain registration (not the hosted zone) contains the nameservers of both hosted zones. (main and sub ).

2 Download and set up AWS CLI and terraform. Use aws configure to set up the desired account credentials.

3 Set up a S3 bucket to save Terraform state. In my case the bucket is called capstone-eks-bucket.  
  
**aws s3 mb s3://capstone-eks-bucket --region us-west-2**

4 Adjust the backend.tfvars file to contain the correct values for your environment. (bucket, zone, etc.) Then initiate the backend, and create a new terraform workspace called development.

**terraform init -backend~~-~~config=backend.tfvars  
terraform workspace new development**

4 After setting up the terraform environment, it is time to adjust all the variables to match the configuration you desire. All variables are contained in .tfvars files. Please edit and adjust the following;

core-network.tfvars  
core-eks.tfvars  
setup-eks.tfvars  
setup-external-dns.tfvars  
setup-iam.tfvars  
setup-ingress.tfvars  
setup-namespace.tfvars   
  
5 After all environment variables are set correctly, we can run our terraform code by running these commands:   
  
terraform plan -out=development.tfplan -var~~-~~file=core-network.tfvars -var~~-~~file=core-eks.tfvars -var~~-~~file=setup-eks.tfvars -var~~-~~file=setup-external-dns.tfvars -var~~-~~file=setup-iam.tfvars.tfvars -var~~-~~file=setup-ingress.tfvars -var~~-~~file=setup-namespace.tfvars

terraform apply development.tfplan

As you can see by inspecting the code, running these commands will make us end up with a fully configured EKS cluster, running behind a loadbalancer, with an ingress controller connected to AWS, spot instances ready for future node scale out, connectivity with route53, and a HA network setup. Also the security policies have been set up correctly.

Inspect the AWS VPC, EKS and EC2 pages to see if our deployment has been configured correctly.   
  
NOTE: if we want metrics to work, we should adjust the eks node-group security policy to allow 4443 traffic from the cluster security group.   
  
6 Now that we have our cluster, we have 2 environments we want to connect to it. Our local node; we do this by running   
  
aws eks update-kubeconfig --region region-code --name cluster-name

And our gitlab environment, where the code for the NGINX application is stored, versioned, and run through our test pipeline to make sure we do not deploy with any errors. We set up gitlab by pressing “Infrastructure” in the repository, and clicking “Kubernetes”, and then “add cluster”. Simply run the commands, and everything will be set up.   
  
Last part of our configuration is to set the following variables in the CI/CD pipeline settings;   
  
AWS\_DEFAULT\_REGION   
AWS\_ACCESS\_KEY\_ID  
AWS\_SECRET\_ACCESS\_KEY  
KUBERNETES\_CLUSTER  
CLUSTER  
REGION  
  
Use the variables that you have set up earlier obviously. CLUSTER and REGION are simply copies of the AWS\_DEFAULT\_REGION and KUBERNETES\_CLUSTER variable. (for easier usage, )  
  
7 Now that everything is connected, we can install our metrics server, by running   
  
kubectl apply -f components.yaml from our working directory. (This deploys metrics server, all set up correctly )   
  
8 The last part of this project is invoking Gitlab to first test and then deploy our app. Before we do this, please adjust the   
  
values.yaml and the config/values.yaml files in the helm folder to contain your environment variables.

After adjusting the variables, Gitlab will run the pipeline, and if everything checks out (it will) your pipeline will be able to (after manually invoking) deploy the application. Simply click deploy from the pipeline menu, and the app will be deployed into the kubernetes cluster, exposed behind load-balancer on the DNS address we set up for it. (browse to the DNS address in your browser to see the deployment working, and being load-balanced over different pods each refresh ).

9 The last thing we need to do is set up autoscaling for the app. We do this by running   
  
**kubectl autoscale deployment sample-app -n sample-apps –cpu-percent=50 –min=1 –max=20**check to see if metrics are available for deployment

**Kubectl get hpa -n sample-apps**Now we simply produce load, there are multiple ways to do this, simple way is to open a terminal and run

**while sleep 0.01; do wget -q -O- capstone.unixinfradevelopment0.click; done   
  
(use your own DNS name)**

After running this for a few minutes, check

**Kubectl get hpa -n sample-apps**You will see the cluster has scaled up the amount of pods. After stopping the load the cluster will scale down again.

**Thats it for now! A automatically autoscaling cluster running a web-application exposed to the outside world via DNS. Everything is load balanced and set up automatically.**

**POINTS FOR IMPROVEMENT:**In the future I would like to add the AWS CA scaling options as well, and configure this better. As of now the spot instances are set up, but I have not fine-tuned the configuration yet. This is something I will definitely do in the future.  **SELLING POINTS:**

Setting up an environment in this manner has major benefits. First of all the app can automatically scale under load, saving costs and operations, second of all the code can be re-used to create similar environments. The usage of gitlab with included tests for the code makes sure we have a history of code changes, and are able to automatically deploy code to our cluster. The setup of the EKS cluster is done in such a way that adding more apps simply means setting up another subdomain, adjusting variables and deploying the code. This gives us major flexibility. The gitlab way is in my opinion way better than manually backing up ECTD to save state; we can roll back and forth between deployments easily, and have good oversight about what is being changed.

The last benefit is the spot instances AWS offers; we can deploy apps and fill our servers until the total CPU count gets too high, then we will automatically scale out spot instances to maintain good latency during peak hours. When load decreases so do the instances and as a result the costs.

All in all this is a very flexible way of working. I really liked building this project, and I’ve learned a lot.