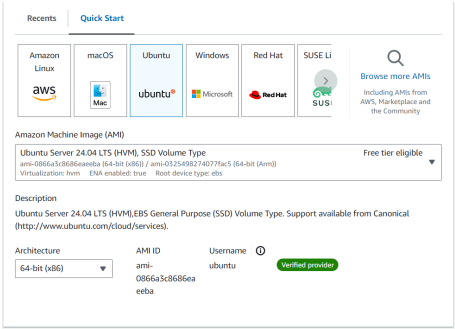
Kubernetes Basics

Felicián Németh, Balázs Fodor, István Pelle, Balázs Sonkoly

2024-10-17

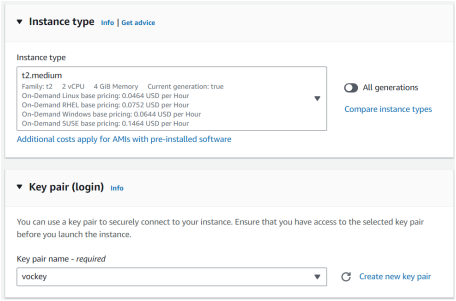
Start an AWS EC2 instance 1

- make sure you select the Ubuntu image



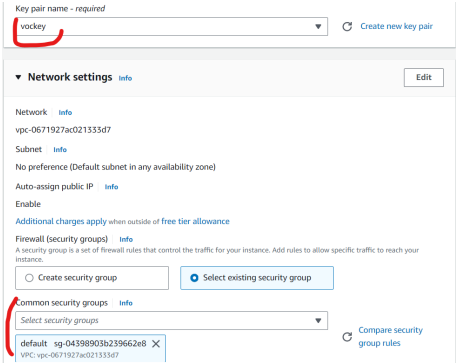
Start an AWS EC2 instance 2

- Instance type: t2.medium
- Key pair: vockey
  - you do not need to create a new key!



Start an AWS EC2 instance 3

- Common security groups: select the default security group

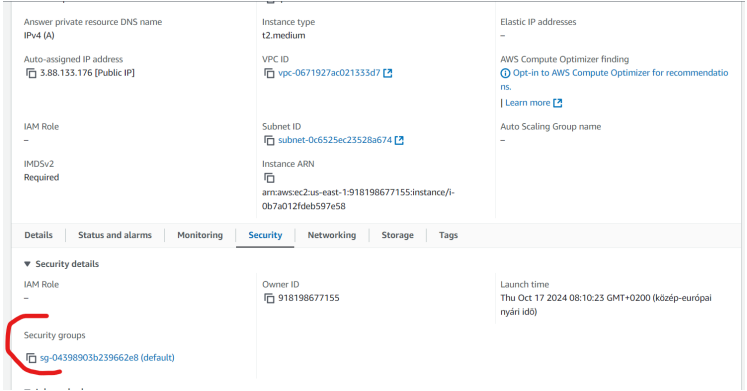


Security Group Update

- Configure the following inbound rules in your EC2's security group:
- Watch the source parameter, e.g. allow traffic from 0.0.0.0/0 to access from anywhere
- Allow SSH port 22
- Allow HTTP traffic on port 80
  - the built-in ingress-controller of k3s will use it
- Allow HTTP traffic on port 88
  - we will use it later to expose a service
- Allow TCP traffic on port 6443
  - we only need this rule, if we want to use kubectl on our local machine to access the cluster
- Allow TCP traffic on port range 30000-32767
  - when using NodePort for a kubernetes Service, it will automatically get a port in this range

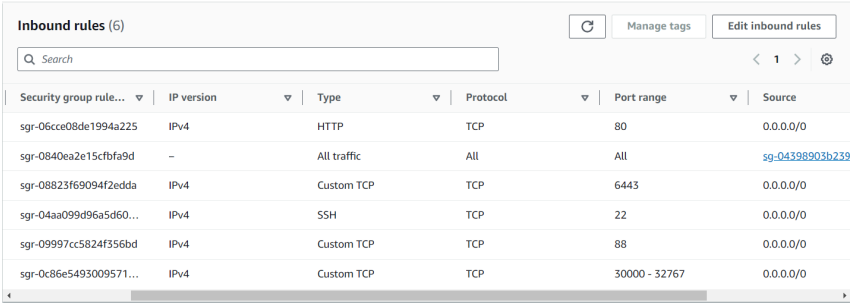
Security Group Update

- make sure you update the right Security Group
- you can find that information in the Security tab of your EC2's details page



Security Group Update

- You should have something like this:



Inbound rules (6)

Manage tags | Edit inbound rules

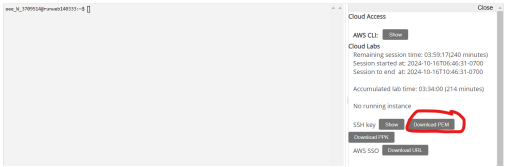
Search

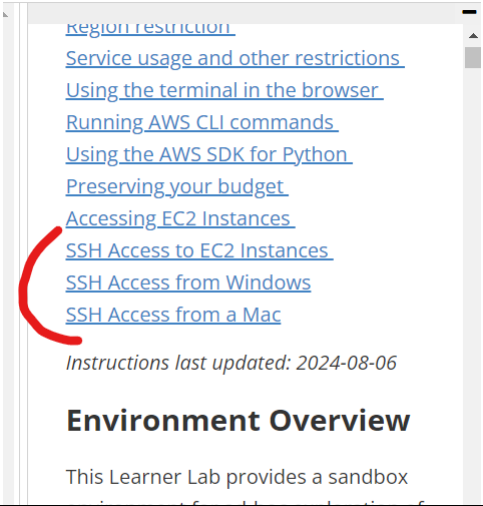
1

Security group rule...	IP version	Type	Protocol	Port range	Source
sgr-06cce08de1994a225	IPv4	HTTP	TCP	80	0.0.0.0/0
sgr-0840ea2e15cfbfa9d	–	All traffic	All	All	sg-04398903b239
sgr-08823f69094f2edda	IPv4	Custom TCP	TCP	6443	0.0.0.0/0
sgr-04aa099d96a5d60...	IPv4	SSH	TCP	22	0.0.0.0/0
sgr-09997cc5824f356bd	IPv4	Custom TCP	TCP	88	0.0.0.0/0
sgr-0c86e5493009571...	IPv4	Custom TCP	TCP	30000 - 32767	0.0.0.0/0

Access the instance

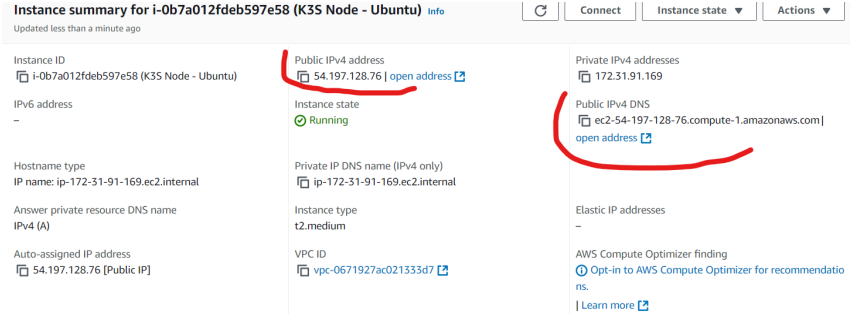
- From the Learner Lab portal, you can download the .pem key which can be used to access the instance (using SSH)
- If you don't have ssh, you can use PuTTY
  - in this case you have to download the .ppk key from the Learner Lab
- Additional information on how to access the instance can be found in the Learner Lab





Access the instance (using SSH client)

- In the AWS Console under EC2 > Instances > <your instance> you can find the Public IP and Public Host name (Public IPv4 DNS)
  - WE WILL USE THESE VALUES LATER, SO REMEMBER WHERE TO FIND THEM




```
ssh ubuntu@[Public IPv4 DNS of your instance] -i <path to the downloaded .pem file>
```

Install k3s on an amazon node

Tools we are going to use

- k3s: lightweight kubernetes distribution
  - we are going to create a one node cluster
- kubectl: command line interface to interact with the kubernetes cluster

Install k3s on an amazon node

- https://docs.k3s.io/quick-start
- https://docs.k3s.io/installation

```
curl -sfL https://get.k3s.io | sh -s - --write-kubeconfig-mode 644
```

Verify the install

```
# list the nodes currently available in your cluster
kubectl get nodes
```

Very Optional: Access the cluster from outside

- THIS MAY TAKE SOME TIME
- what you can achieve: You can run kubectl commands on your local machine
- start k3s the following way:

```
curl -sfL https://get.k3s.io | sh -s - --write-kubeconfig-mode 644 --tls-san <Public IP of your EC2 instance>
```

- `--tls-san` option is required if we want to access the cluster from outside using kubectl
- download kubectl to your local machine
- https://docs.k3s.io/cluster-access#accessing-the-cluster-from-outside-with-kubectl
- Copy `/etc/rancher/k3s/k3s.yaml` on your machine located outside the cluster as `~/.kube/config`
- Then replace the value of the server field with the IP or name of your K3s server (EC2 instance)
- kubectl on your local machine can now manage your K3s cluster.
- or you can use a configuration file as follows
  - if you copied the config to ./config.yaml

```
kubectl --kubeconfig ./config.yaml get nodes
```

Optional (but recommended) bash completion with TAB

Command line completion can be extremely useful:

```
## Load the kubectl completion code for bash into the current shell
source <(kubectl completion bash)
```

The documentation details how it can be enabled permanently and for other shells.

Files for the current lab

- Every configuration we show on these slides are uploaded to the following repository
- Please clone the repository and use the files in it

```
# you probably have git installed in your EC2 instance, if not, you can install it with this command
sudo apt install git

# clone and enter the repository
git clone https://github.com/hsnlab/edu-cloud-native.git
cd edu-cloud-native/k8s-intro
```

Install docker

```
./docker-install.sh
```

Building Docker images

```
FROM golang:alpine AS builder

WORKDIR /
COPY *.go /
RUN CGO_ENABLED=0 GOOS=linux\
    go build -ldflags="-s -w" \
             -trimpath \
             -o hello_server \
             hello_server.go

#############
FROM alpine

RUN apk add --no-cache \
    bash \
    tar \
    curl \
    tcpdump
COPY --from=builder /hello_server /usr/bin/hello_server
EXPOSE 8080
ENTRYPOINT ["hello_server"]
```

- build command:

```
sudo docker build -t practice/06 .
```

- import into the k3s node:

```
sudo docker save practice/06 |\
  sudo k3s ctr images import -
```

Deployment / Pod

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-server
spec:
  replicas: 1
  selector:
    matchLabels:
      app: my-server
  template:
    metadata:
      labels:
        app: my-server
    spec:
      containers:
      - name: server
        image: docker.io/practice/06
        imagePullPolicy: Never
        ports:
        - containerPort: 8080
```

```
# list the pods in your default namespace
# we can replace "pods" with "deployments" or "services"
# and check other resource types (kubernetes objects)
kubectl get pods
# list all pods in your cluster
kubectl get pods -A
# create a deployment containing the container we've just built
kubectl apply -f deployment.yaml
# deploy a pod which we use to send test requests
kubectl apply -f net-debug.yaml
# we can get more detailed information about our resources with these commands
kubectl get pods -o wide
# using the -o wide option we can see the IP address of the Pods
# Make note of this IP address, we will use it later

kubectl get pods -o yaml
kubectl describe pod <pod name>
kubectl describe deployment <deployment name>
# Test our newly created Pod!
# 1. enter our debug pod, from where we can send requests
kubectl exec -it <name of your net-debug pod> -- bash
# 2. send a request to our Pod
curl http://<ip of my-server Pod>:8080

# if you want to delete something
kubectl delete -f <config file to delete>
```

Resiliency with replica count

- open another terminal with a shell in it
- shell #1: `kubectl get pods --watch` , continuously show changes in the state of pods
- shell #2: `kubectl delete pod my-server-xxx`
- shell #1: observe the changes
- what do you see? (hopefully you will see another Pod created instead of the one you deleted)
- shell #2: `EDITOR=nano kubectl edit deployment my-server`
  - (change replicas to 2)
- shell #1: observe the changes
- what do you see? (hopefully you will see the creation of an additional pod)

Service within the cluster

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: my-server
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
```

```
kubectl apply -f service.yaml

# You can see the CLUSTER-IP column of the services
# This IP can be used to access the Pods "behind" the Service
kubectl get services
kubectl get svc
# use the describe to find the namespace the Service is deployed into
kubectl describe svc my-service
```

- from the net-debug Pod we can try to access our Pods different ways

```
# we can still use the IP of our Pod
# but we have to use the specific IP of the Pod
curl http://<IP of one of the my-service Pod>:8080/

# We can use the Cluster-IP of our service
curl http://<cluster-ip of my-service>:80/
# We can also use the domain name (and aliases) generated for the service
# If the service is deployed to the default namespace
curl http://my-service:80/
curl http://my-service.default:80/
curl http://my-service.default.svc.cluster.local
```

Service within the cluster

- Note that we used the 8080 container port when accessing the Pod directly, and 80 when accessing via Service
  - why is that? Check the Service configuration yaml
- If you have two Pods, and you send multiple requests to the service, you can see the load balancing in action
  - the response contains the name of the responding Pod

External service

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-external-service
spec:
  selector:
    app: my-server
  ports:
    - name: http
      protocol: TCP
      port: 88
      targetPort: 8080
  type: LoadBalancer # We added this
  ↪  option
```

```
# apply the service
kubectl apply -f service-external.yaml
kubectl get svc
# we can have our result in a formatted JSON as well
kubectl get svc my-external-service -o json | jq .
# we can see an External IP here
kubectl get svc my-external-service -o jsonpath={.status.loadBalancer}
# the last two commands also demonstrated
# different output formats useful for automation
```

- access my-external-service from your own laptop (i.e., externally)
  - with `curl` or with a normal browser
  - Use the External IP of the service and the 88 port we defined as Service port
  - make sure to access the webserver using http, not https!

Recap: some definitions

- *Cluster IP*: address accessible from the whole cluster
- *Node port*: port is allocated on every node of the cluster
- *External IP*: the service is accessible from outside of the cluster via this address

Configuring Ingress

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-service-http
  annotations:
    traefik.ingress.kubernetes.io/router.entrypoints:
    ↪ web
spec:
  rules:
    - host: <Host name of EC2> # WE NEED TO UPDATE
    ↪ THIS
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: my-service
                port:
                  number: 80
```

- k3s comes with a built-in ingress controller
- ingress controller
  - gives externally reachable URL to our service
  - load balance traffic
  - name-based virtual hosting
  - handles http, https traffic
  - we can use this to expose our webserver
- edit the ingress.yaml file to use the host name of your EC2 instance
- then execute:

```
# create an ingress rule
kubectl apply -f ingress.yaml
# verify it
kubectl get ingress
```

- let's see if we can access the webserver from our browser
- Use the host name of the EC2 instance and the port 80

Labels

"Labels are key/value pairs that are attached to objects such as Pods."

```yaml
# This is an excerpt demonstrating label keys used in practice
apiVersion: apps/v1
kind: StatefulSet
metadata:
  labels:
    app.kubernetes.io/name: mysql
    app.kubernetes.io/instance: mysql-abcxyz
    app.kubernetes.io/version: "5.7.21"
    app.kubernetes.io/component: database
    app.kubernetes.io/part-of: wordpress
    app.kubernetes.io/managed-by: Helm
```

```
kubectl get pods -A --show-labels
kubectl get pods -l app=my-server
kubectl get pods -l 'app in (my-server, my-client)'  # demonstartes an OR expression
kubectl get pods -L app
```

- the documentation has more
- tasks:
  - write a cli command that lists name of the services in every namespace which have a label key named "component"
  - how many pods have a "k8s-app" label key?

Container command line arguments

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-server
spec:
  replicas: 1
  selector:
    matchLabels:
      app: my-server
  template:
    metadata:
      labels:
        app: my-server
    spec:
      containers:
      - name: server
        image: docker.io/practice/06
        imagePullPolicy: Never
        ports:
        - containerPort: 8080
        env:
          - name: FOO
            value: BAR
        command: ["hello_server"]
        args: ["-root-env-var", "FOO"]
```

- set environment variables with: `spec.template.spec.containers.env`
  - here `FOO` will be set to `BAR`
- change container's entry point with `.command` and `.args`

`hello_server` command line arguments:

- `-root-env-var VarName`: server returns the value of the environment variable "VarName" to the request of the root document ("/")
- `-root-file AbsFileName`: server returns the content of AbsFileName to the request of the root document.

- `kubectl apply -f deployment-2.yaml`
- `curl http://<my-service>/`
- `kubectl apply -f deployment-3.yaml`
- `curl http://<my-service>/`

Environment Variables and Self-Awareness

- previous slide showed how a pod specification can define environment variables for its containers
- TASK: let's modify `deployment-2.yaml` to pass the IP address of the pod (*pod id*) to its container
- this is harder because pod-ip is not known beforehand
- help: read the Self Awareness chapter of the "Kubernetes Patterns"
  - available from Files tab of the General channel in Teams

(`deployment-4.yaml` contains the solution)

ConfigMap

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: game-demo
data:
  # property-like keys; each key maps to a simple value
  player_initial_lives: "3"
  ui_properties_file_name: "user-interface.properties"

  # file-like keys
  game.properties: |
    enemy.types=aliens,monsters
    player.maximum-lives=5
  user-interface.properties: |
    color.good=purple
    color.bad=yellow
    allow.textmode=true
```

- example is from the official docs
- TASK: using the docs, update `deployment-3.yaml`, so the server return the value of `game.properties`

```
$ kubectl apply -f configmap.yaml
$ kubectl apply -f deployment-5.yaml
$ curl http://<my-service>/

enemy.types=aliens,monsters
player.maximum-lives=5
$
```

iMSc: ngnix with password access

1. create a ngnix deployment/service from docker image "ngnix"
2. modify the server deployment to provide a custom config file to the server that protects all of its contents with a password

- First solve the problem with a ConfigMap
- For extra points, use a Secret object for the password