Create one RedHat Linux Management node

Login to the node install kops

kops helps you create, destroy, upgrade and maintain production-grade, highly available, Kubernetes clusters from the command line. AWS (Amazon Web Services) is currently officially supported, with GCE and VMware vSphere in alpha and other platforms planned.

Install Kops

wget -O kops https://github.com/kubernetes/kops/releases/download/1.7.0/kops-linux-amd64

chmod +x ./kops

sudo mv ./kops /usr/local/bin/

Install kubectl

wget -O kubectl https://storage.googleapis.com/kubernetes-release/release/`curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt`/bin/linux/amd64/kubectl

chmod +x ./kubectl

sudo mv ./kubectl /usr/local/bin/kubectl

Install AWS CLI

sudo yum -y install wget

wget <https://dl.fedoraproject.org/pub/epel/epel-release-latest-7.noarch.rpm>

sudo yum -y install epel-release-latest-7.noarch.rpm

sudo yum -y install python-pip

sudo pip install awscli

Configure Aws

aws configure

ACCESS KEY:

SECRET KEY:

REGION:

OUTTYPE:

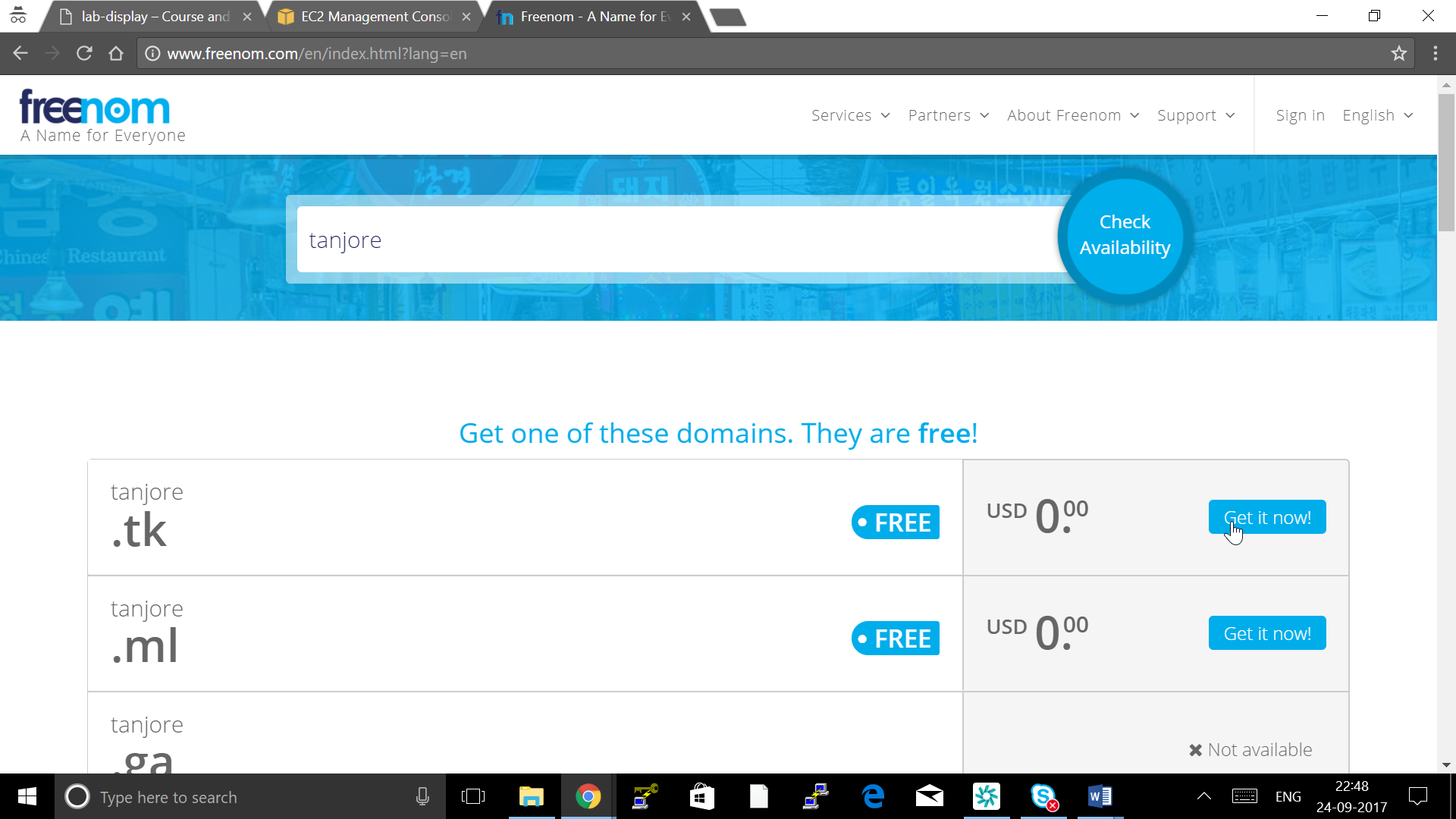
Create a domain name

Go to <http://www.freenom.com/en/index.html?lang=en>

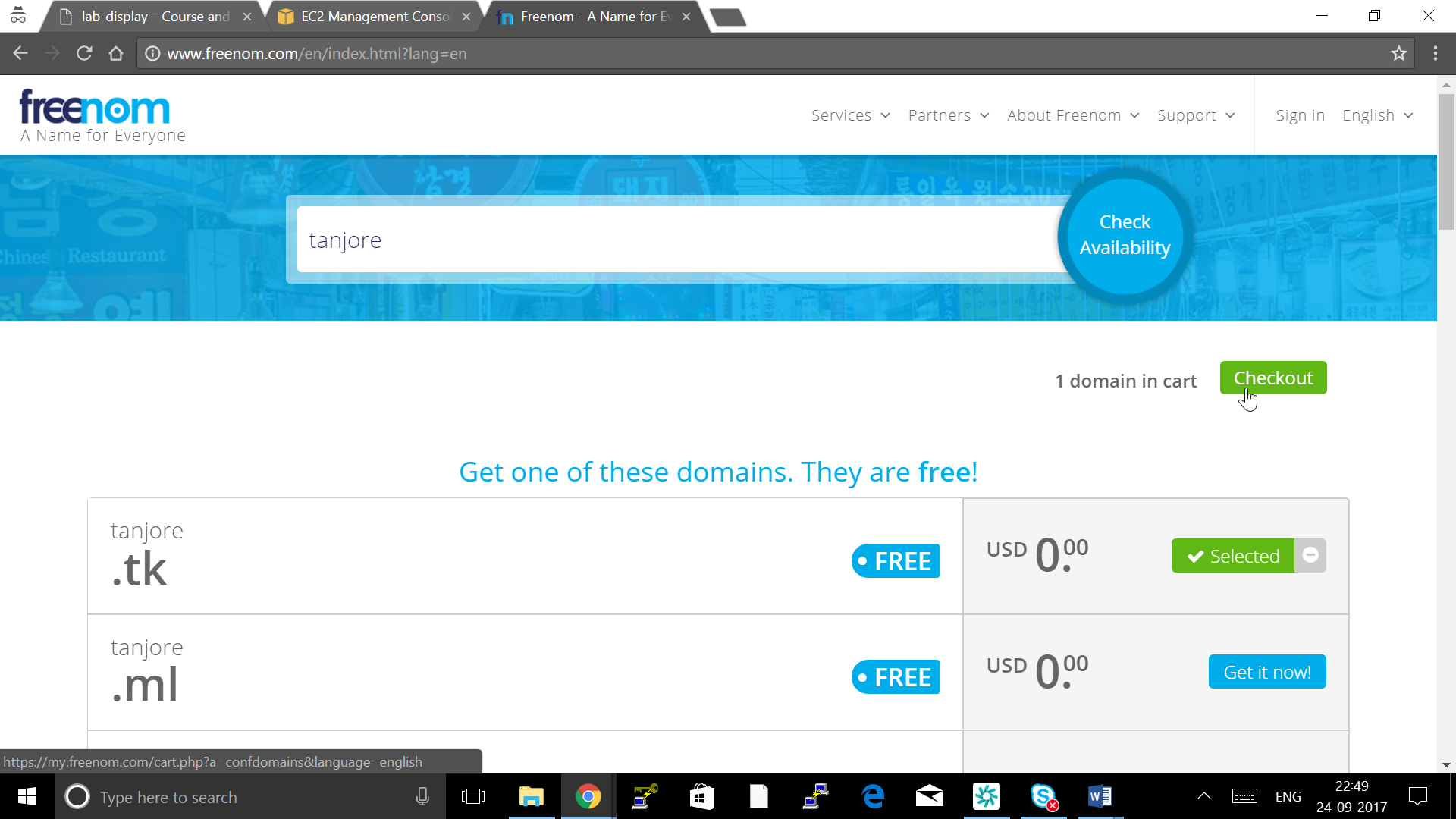
Type name of your domain and check for availability

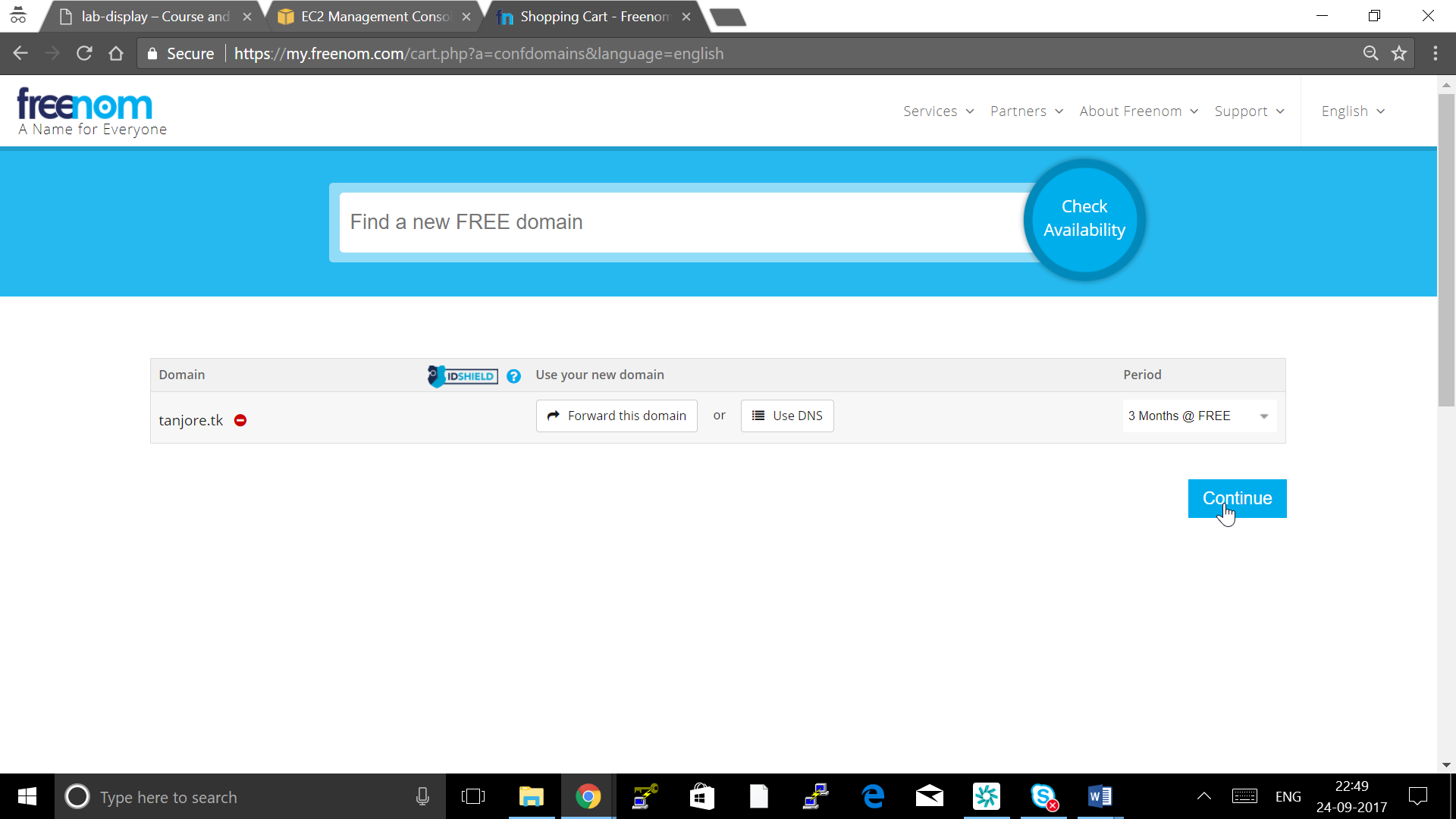


Click on Get it now

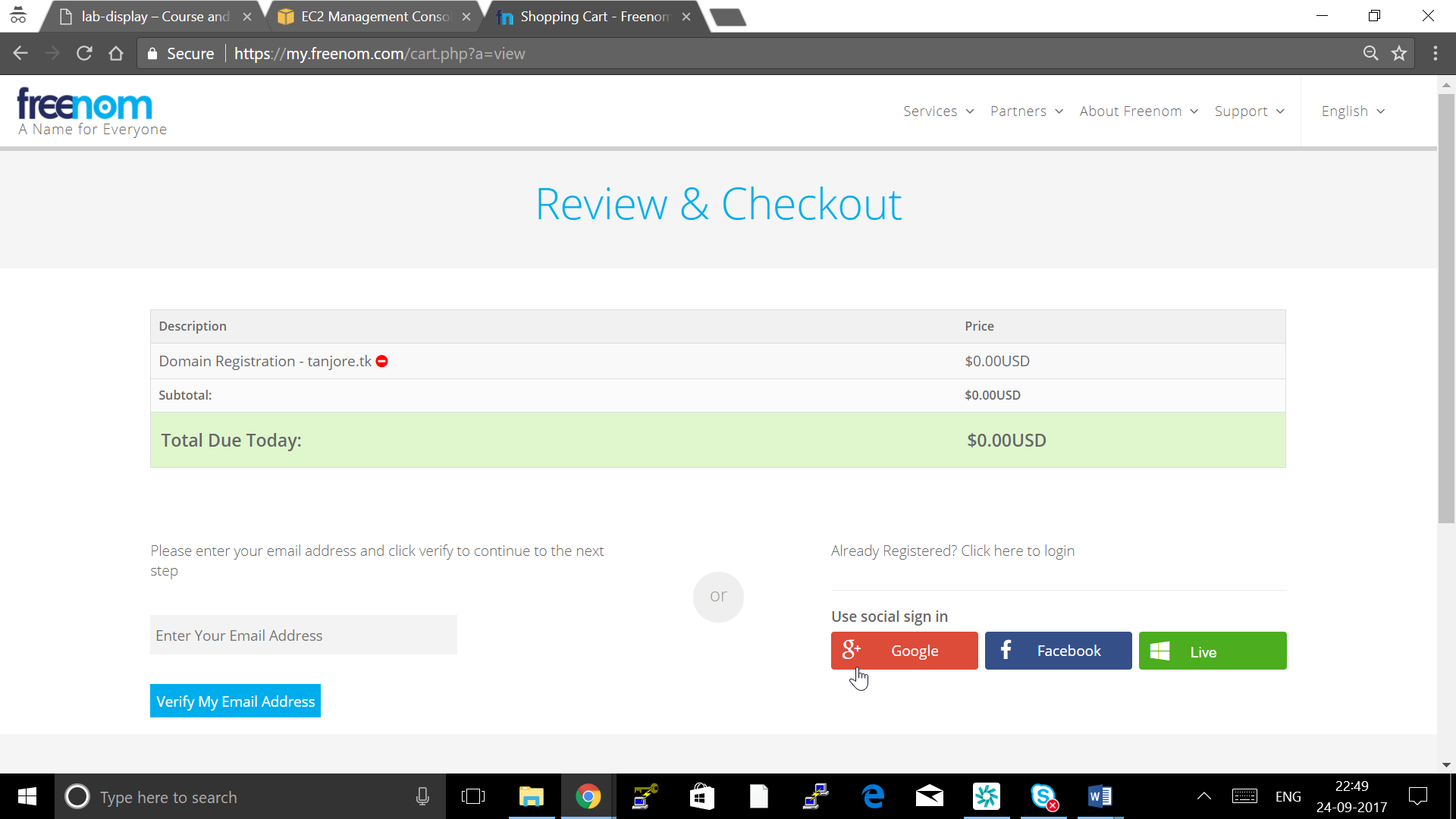


Click on Checkout

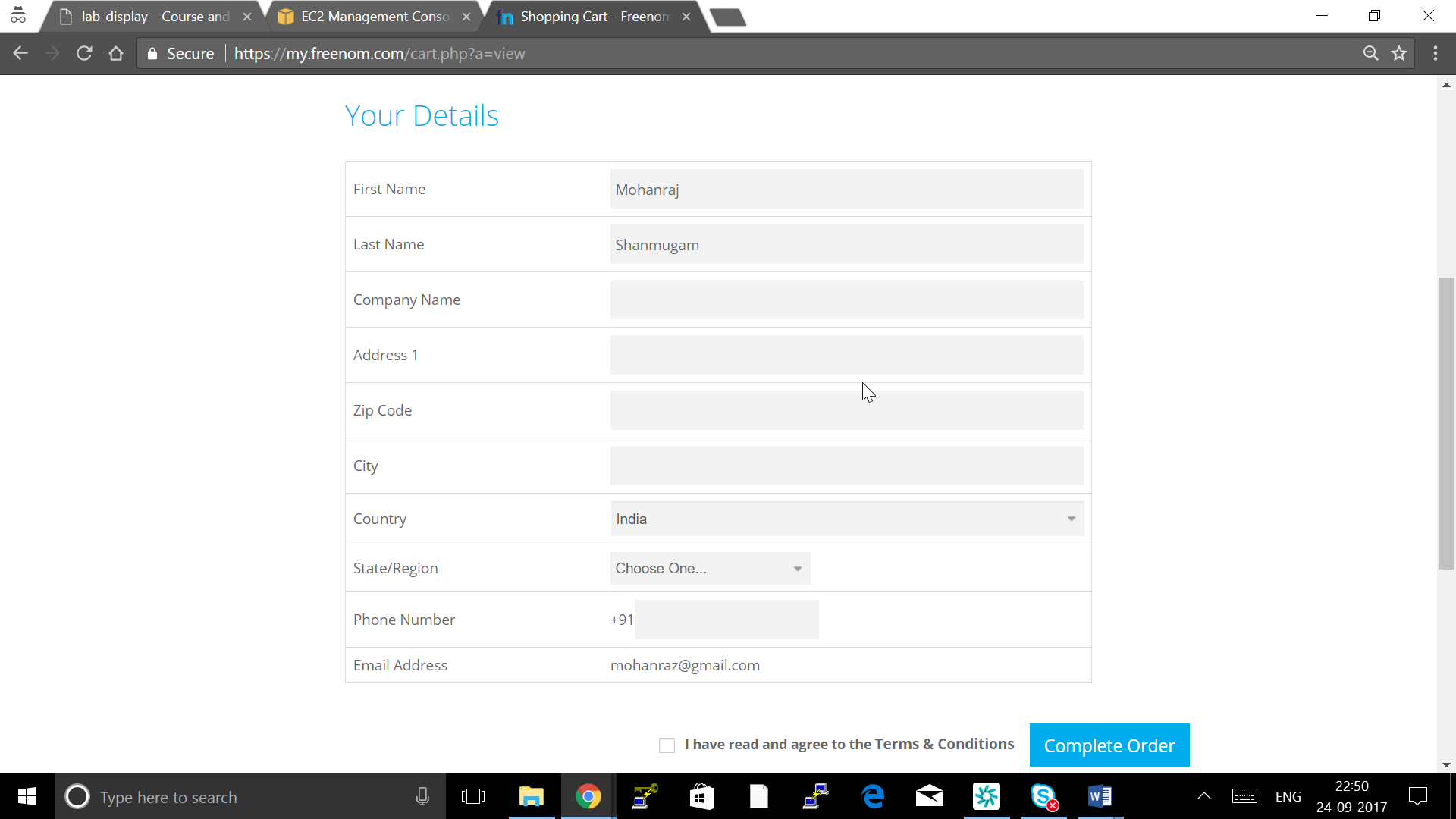
Click Continue

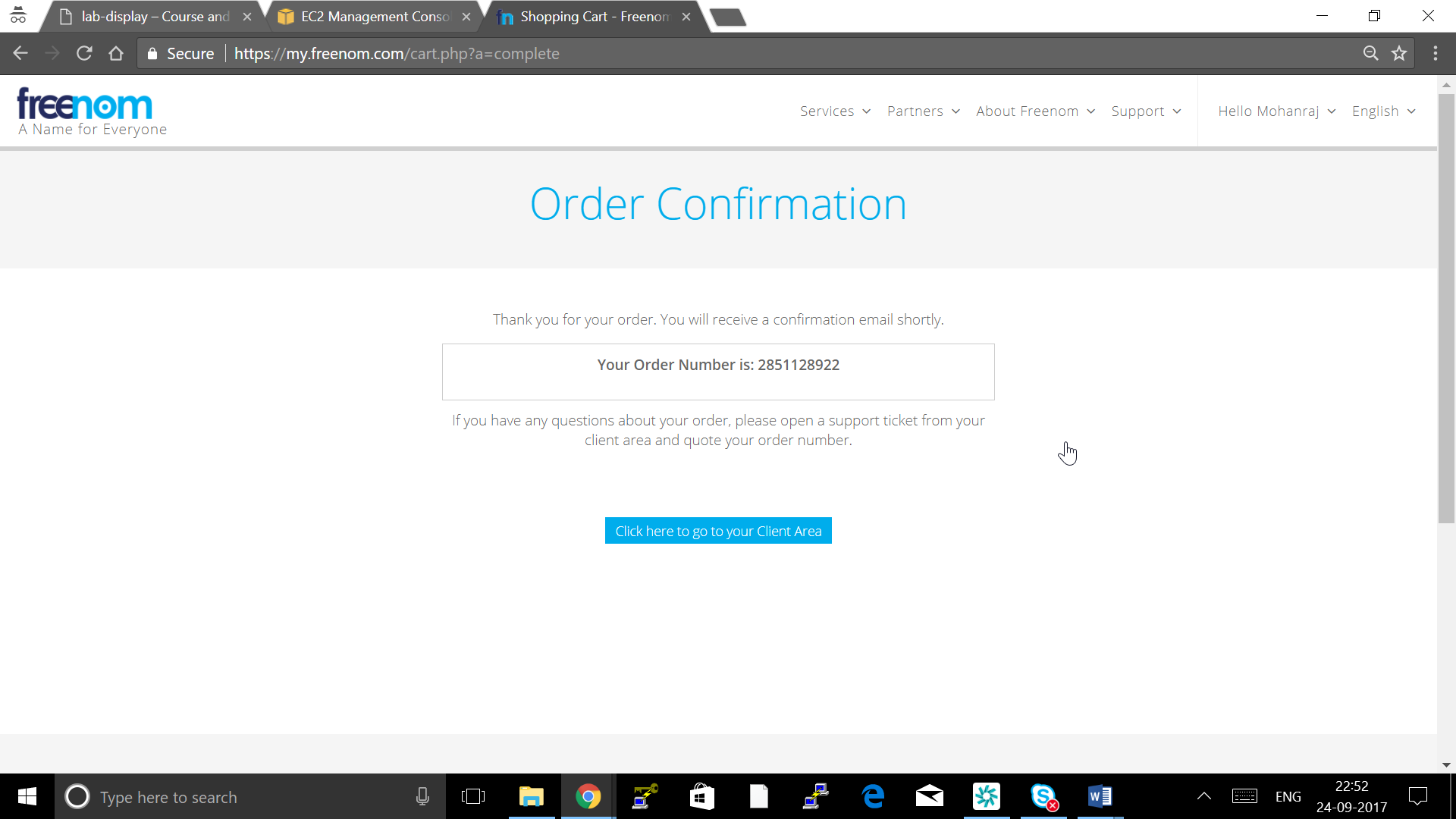


Click google login with your gmail id

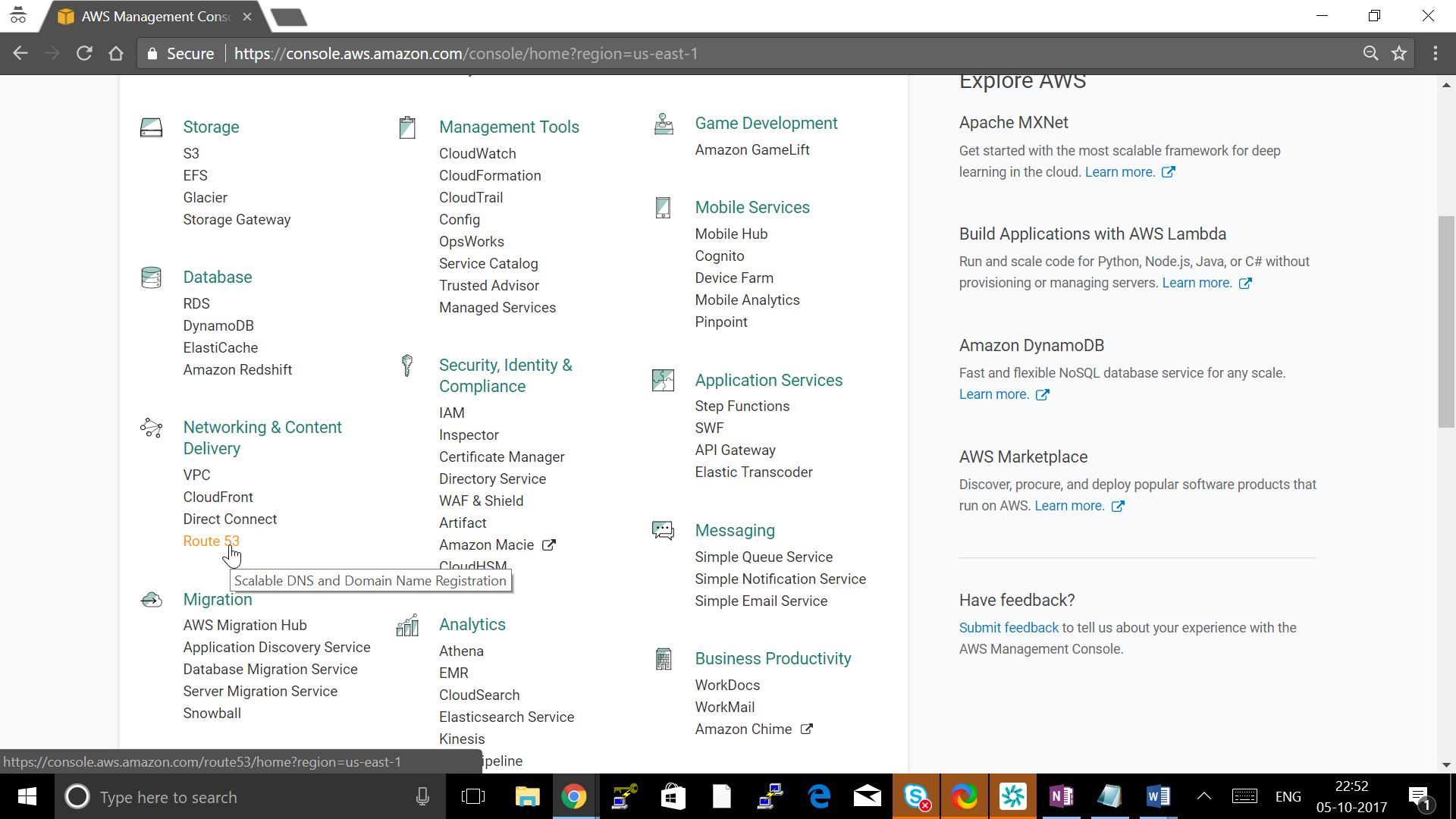


Fill all your details ( This can be dummy details ) and click Complete Order





Go to AWS Route-53



Click Hosted zone



Create Hosted Zone

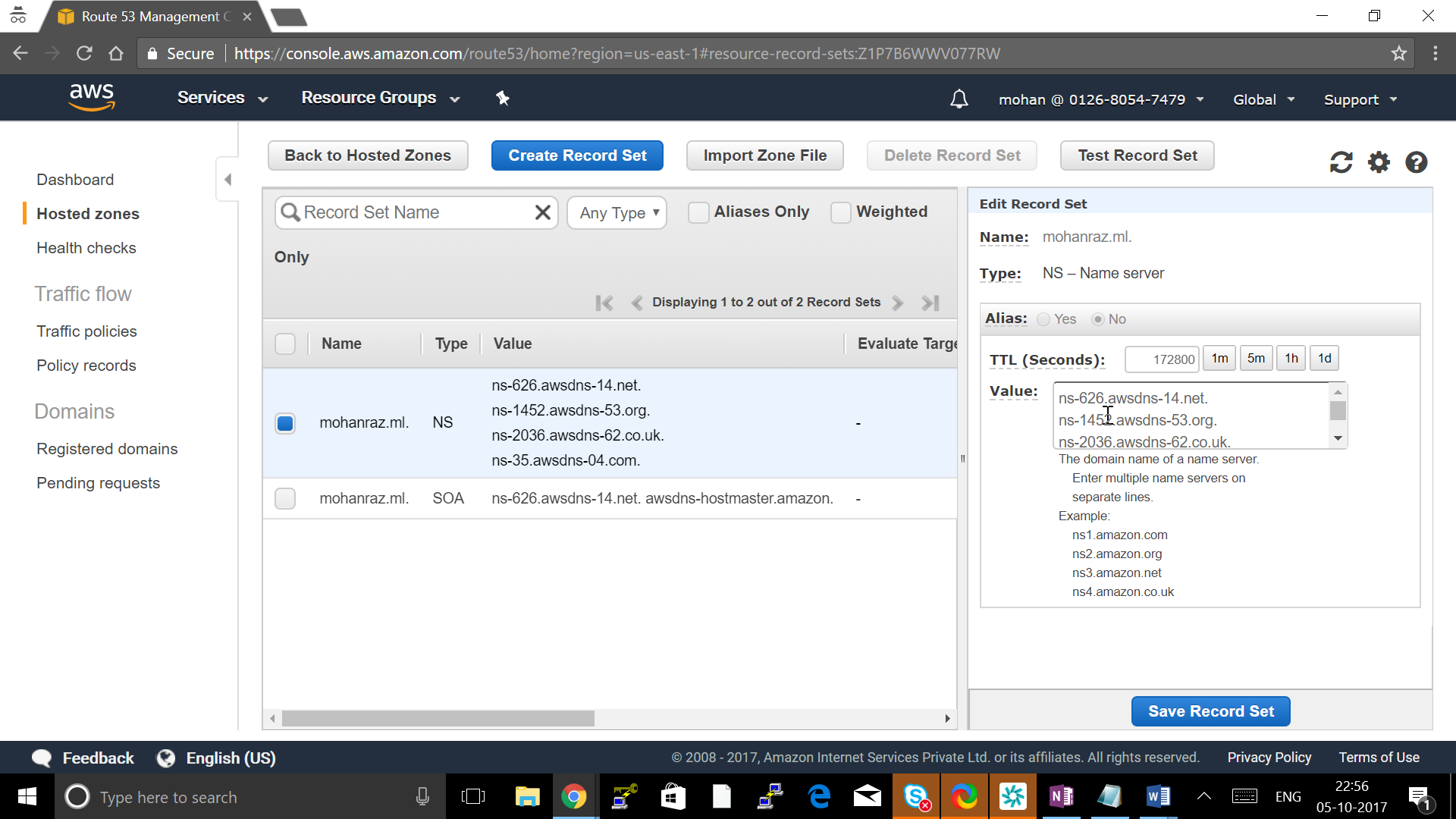
Give : Domain Name created

Type: Public hosted zone

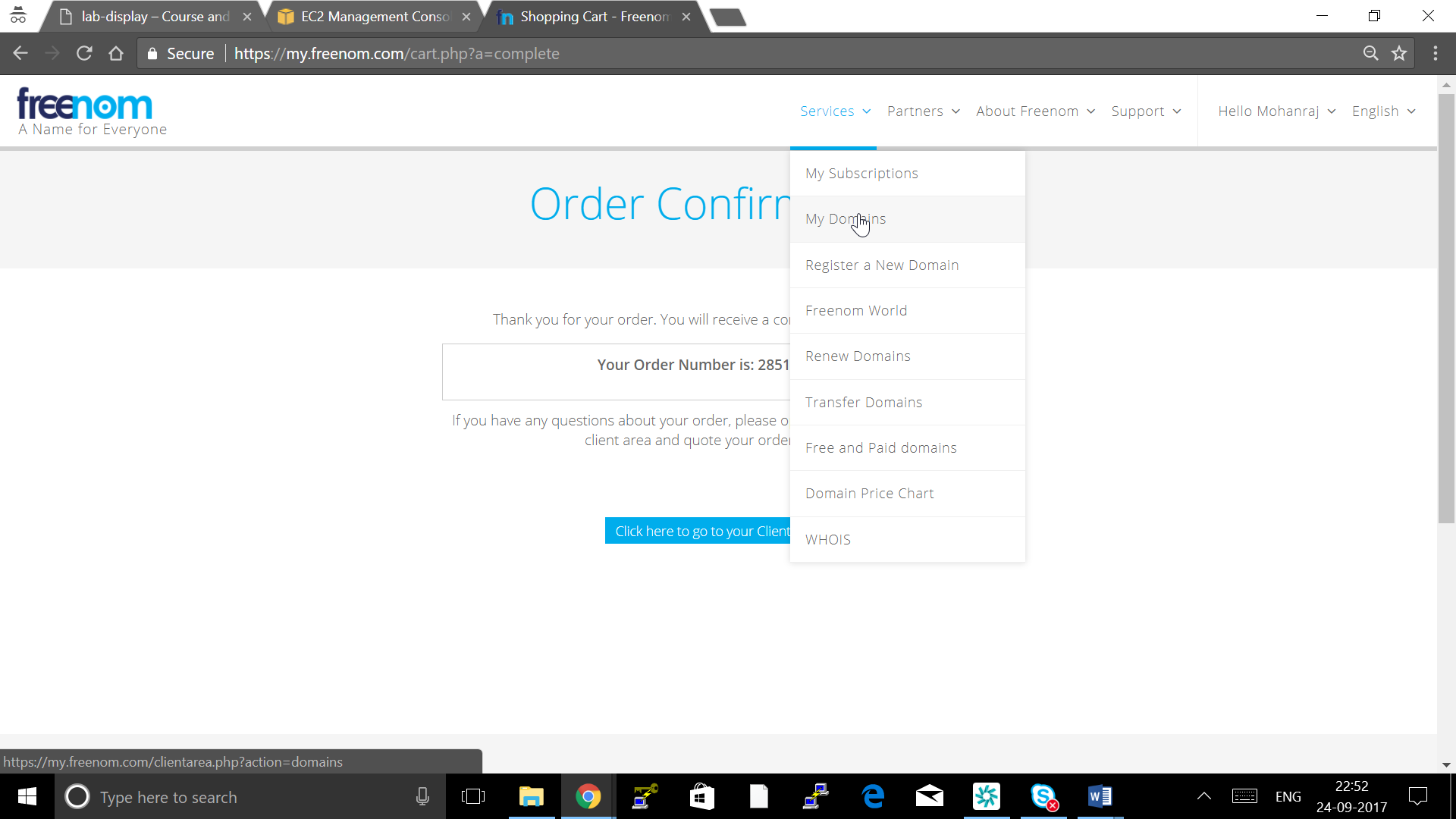
Click create



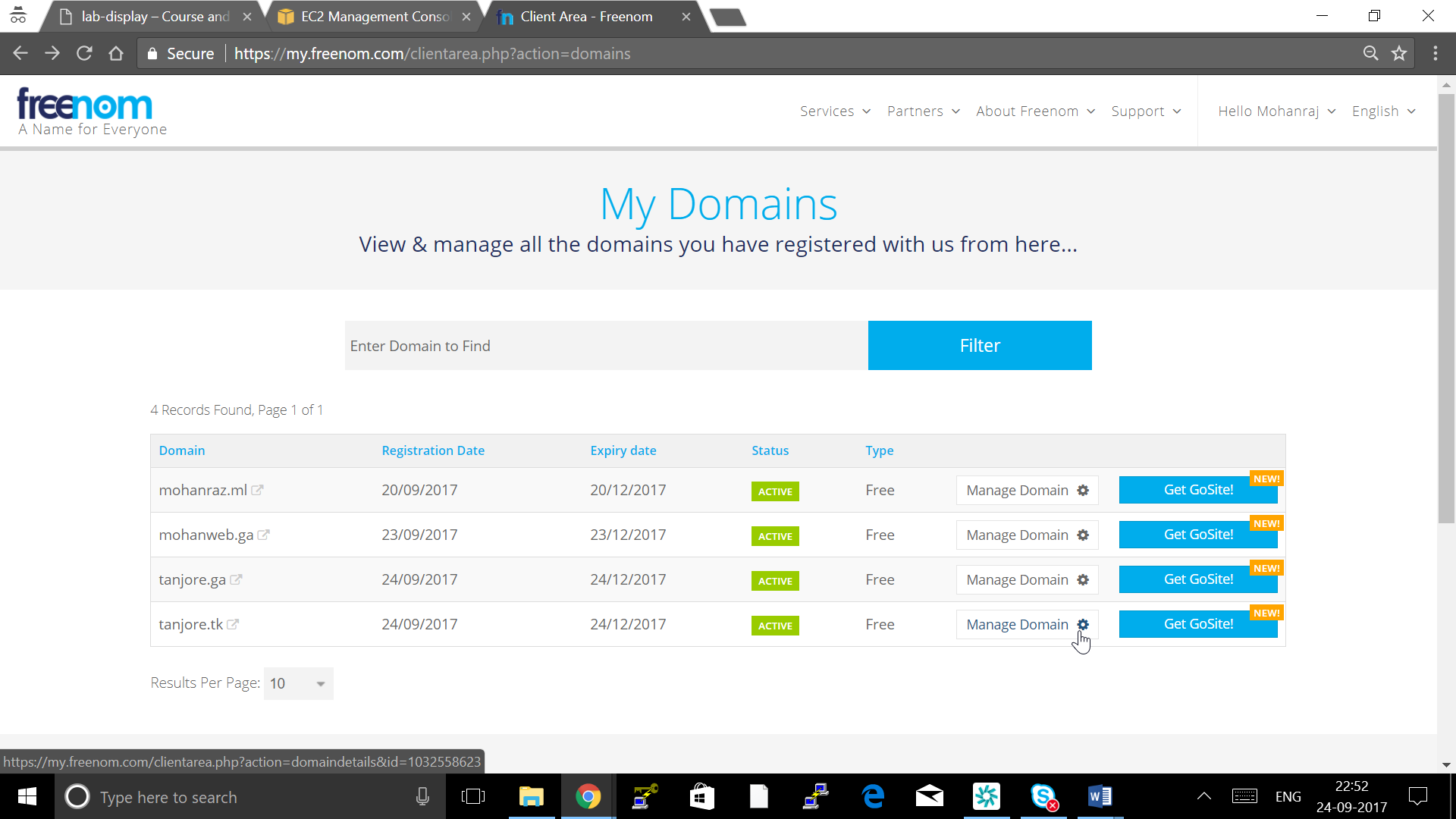
Click on NS Record and note down all 4 NS DNS



Go to Freenom-> services -> My Domain

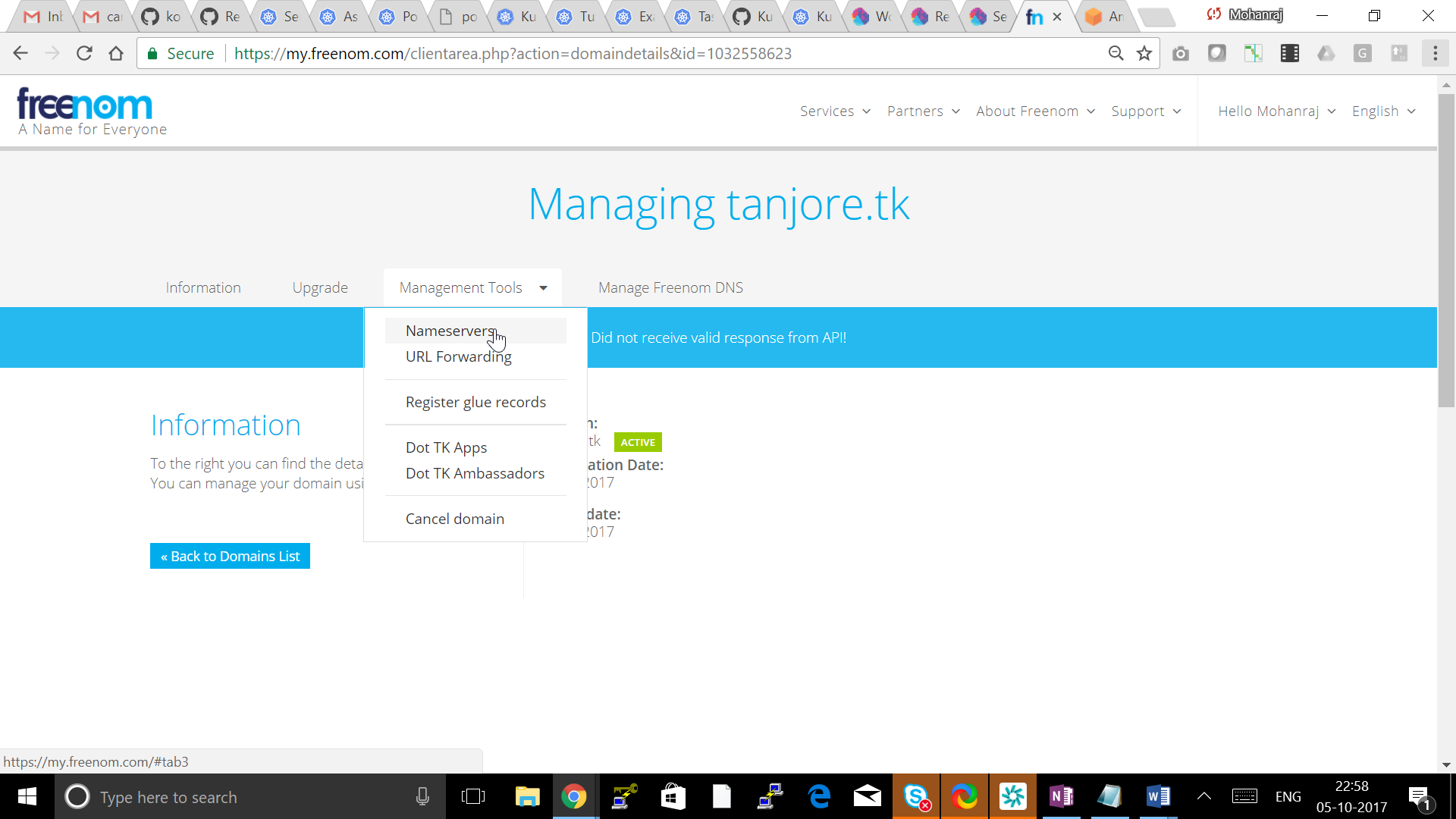


Click on Manage Domain

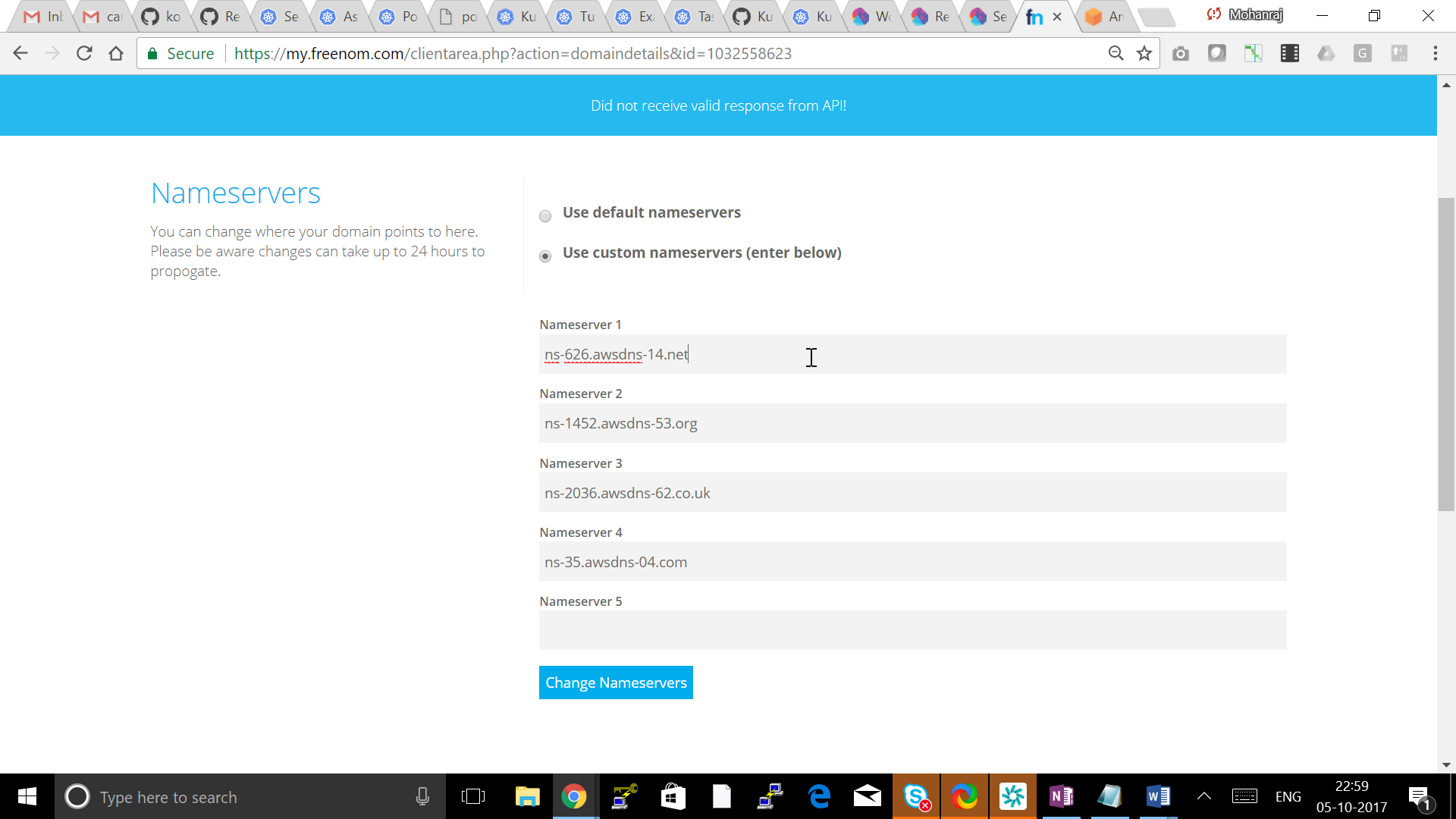




Go to Manage Tools -> Nameserver



Put all 4 Name server shown in Route53 NS record and Click Change name Server



**Testing your DNS setup**

This section is not be required if a gossip-based cluster is created.

You should now able to dig your domain (or subdomain) and see the AWS Name Servers on the other end.

## [ec2-user@ip-10-53-3-250 ~]$ dig ns mohanraz.ml

## ; <<>> DiG 9.8.2rc1-RedHat-9.8.2-0.62.rc1.56.amzn1 <<>> ns mohanraz.ml

## ;; global options: +cmd

## ;; Got answer:

## ;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 16447

## ;; flags: qr rd ra; QUERY: 1, ANSWER: 4, AUTHORITY: 0, ADDITIONAL: 0

## ;; QUESTION SECTION:

## ;mohanraz.ml. IN NS

## ;; ANSWER SECTION:

## mohanraz.ml. 57 IN NS ns-1452.awsdns-53.org.

## mohanraz.ml. 57 IN NS ns-2036.awsdns-62.co.uk.

## mohanraz.ml. 57 IN NS ns-35.awsdns-04.com.

## mohanraz.ml. 57 IN NS ns-626.awsdns-14.net.

## ;; Query time: 0 msec

## ;; SERVER: 10.53.0.2#53(10.53.0.2)

## ;; WHEN: Fri Oct 6 01:23:37 2017

## ;; MSG SIZE rcvd: 168

## Cluster State storage

In order to store the state of your cluster, and the representation of your cluster, we need to create a dedicated S3 bucket for kops to use. This bucket will become the source of truth for our cluster configuration. In this guide we'll call this bucket example-com-state-store, but you should add a custom prefix as bucket names need to be unique.

[ec2-user@ip-10-53-3-250 ~]$ aws s3api create-bucket --bucket kube-mohanraz-ml-state-store --region us-east-1

{

"Location": "/kube-mohanraz-ml-state-store"

}

# [ec2-user@ip-10-53-3-250 ~]$ aws s3api put-bucket-versioning --bucket kube-mohanraz-ml-state-store --versioning-configuration Status=Enabled

# Creating your first cluster

## [ec2-user@ip-10-53-3-250 ~]$ export NAME=testcluster.mohanraz.ml

## [ec2-user@ip-10-53-3-250 ~]$ export KOPS\_STATE\_STORE=s3://kube-mohanraz-ml-state-store

## Create cluster configuration

kops create cluster \

--zones=us-east-1a,us-east-1b,us-east-1c \

--master-zones=us-east-1a,us-east-1b,us-east-1c \

--node-count=4 \

--node-size=t2.medium \

--master-size=t2.medium \

--name ${NAME}

## Customize Cluster Configuration

Now we have a cluster configuration, we can look at every aspect that defines our cluster by editing the description.

kops edit cluster ${NAME}

This opens your editor (as defined by $EDITOR) and allows you to edit the configuration. The configuration is loaded from the S3 bucket we created earlier, and automatically updated when we save and exit the editor.

We'll leave everything set to the defaults for now, but the rest of the kops documentation covers additional settings and configuration you can enable.

## Build the Cluster

Now we take the final step of actually building the cluster. This'll take a while. Once it finishes you'll have to wait longer while the booted instances finish downloading Kubernetes components and reach a "ready" state.

kops update cluster ${NAME} --yes

## Validate and Wait until cluster is running

## [ec2-user@ip-10-53-3-250 ~]$ kops validate cluster

## Using cluster from kubectl context: testcluster.mohanraz.ml

## Validating cluster testcluster.mohanraz.ml

## INSTANCE GROUPS

## NAME ROLE MACHINETYPE MIN MAX SUBNETS

## master-us-east-1a Master t2.medium 1 1 us-east-1a

## master-us-east-1b Master t2.medium 1 1 us-east-1b

## master-us-east-1c Master t2.medium 1 1 us-east-1c

## nodes Node t2.medium 4 4 us-east-1a,us-east-1b,us-east-1c

## NODE STATUS

## NAME ROLE READY

## ip-172-20-107-129.ec2.internal master True

## ip-172-20-125-104.ec2.internal node True

## ip-172-20-54-42.ec2.internal master True

## ip-172-20-59-193.ec2.internal node True

## ip-172-20-82-35.ec2.internal node True

## ip-172-20-89-128.ec2.internal master True

## ip-172-20-90-6.ec2.internal node True

## Your cluster testcluster.mohanraz.ml is ready

## Use the Cluster

Remember when you installed kubectl earlier? The configuration for your cluster was automatically generated and written to ~/.kube/config for you!

A simple Kubernetes API call can be used to check if the API is online and listening.

Let's use kubectl to check the nodes.

kubectl get nodes

You will see a list of nodes that should match the --zones flag defined earlier. This is a great sign that your Kubernetes cluster is online and working.

Also kops ships with a handy validation tool that can be ran to ensure your cluster is working as expected.

kops validate cluster

You can look at all the system components with the following command.

kubectl -n kube-system get pods

## Run a Web Application using Deployment

## Create namespace

**kubectl create namespace test**

## YAML

apiVersion: apps/v1beta1

kind: Deployment

metadata:

name: nginx-deployment

spec:

selector:

matchLabels:

app: nginx

replicas: 2 # tells deployment to run 2 pods matching the template

template: # create pods using pod definition in this template

metadata:

# unlike pod-nginx.yaml, the name is not included in the meta data as a unique name is

# generated from the deployment name

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:1.7.9

ports:

- containerPort: 80

## Create a Deployment based on the YAML file:

**kubectl apply -f nginx-deploy.yaml -–namespace=test**

## Display information about the Deployment:

**kubectl describe deployment nginx-deployment -–namespace=test**

## The output is similar to this:

**user@computer:~/kubernetes.github.io$ kubectl describe deployment nginx-deployment -–namespace=test**

**Name: nginx-deployment**

**Namespace: default**

**CreationTimestamp: Tue, 30 Aug 2016 18:11:37 -0700**

**Labels: app=nginx**

**Annotations: deployment.kubernetes.io/revision=1**

**Selector: app=nginx**

**Replicas: 2 desired | 2 updated | 2 total | 2 available | 0 unavailable**

**StrategyType: RollingUpdate**

**MinReadySeconds: 0**

**RollingUpdateStrategy: 1 max unavailable, 1 max surge**

**Pod Template:**

**Labels: app=nginx**

**Containers:**

**nginx:**

**Image: nginx:1.7.9**

**Port: 80/TCP**

**Environment: <none>**

**Mounts: <none>**

**Volumes: <none>**

**Conditions:**

**Type Status Reason**

**---- ------ ------**

**Available True MinimumReplicasAvailable**

**Progressing True NewReplicaSetAvailable**

**OldReplicaSets: <none>**

**NewReplicaSet: nginx-deployment-1771418926 (2/2 replicas created)**

**No events.**

1. List the pods created by the deployment:

**kubectl get pods -l app=nginx -–namespace=test**

The output is similar to this:

**NAME READY STATUS RESTARTS AGE**

**nginx-deployment-1771418926-7o5ns 1/1 Running 0 16h**

**nginx-deployment-1771418926-r18az 1/1 Running 0 16h**

1. Display information about a pod:

**kubectl describe pod <pod-name>**

where **<pod-name>** is the name of one of your pods

## Updating the deployment

You can update the deployment by applying a new YAML file. This YAML file specifies that the deployment should be updated to use nginx 1.8.

| [**deployment-update.yaml**](https://raw.githubusercontent.com/kubernetes/kubernetes.github.io/master/docs/tutorials/stateless-application/deployment-update.yaml) |
| --- |
| **apiVersion: apps/v1beta1**  **kind: Deployment**  **metadata:**  **name: nginx-deployment**  **spec:**  **selector:**  **matchLabels:**  **app: nginx**  **replicas: 2**  **template:**  **metadata:**  **labels:**  **app: nginx**  **spec:**  **containers:**  **- name: nginx**  **image: nginx:1.8 *# Update the version of nginx from 1.7.9 to 1.8***  **ports:**  **- containerPort: 80** |

1. Apply the new YAML file:

**kubectl apply -f nginx-deploy.yaml -–namespace=test**

1. Watch the deployment create pods with new names and delete the old pods:

**kubectl get pods -l app=nginx -–namespace=test**

## Scaling the application by increasing the replica count

You can increase the number of pods in your Deployment by applying a new YAML file. This YAML file sets repicas to 3 which specifies that the Deployment should have 3

**apiVersion: apps/v1beta2**

**kind: Deployment**

**metadata:**

**name: nginx-deployment**

**spec:**

**selector:**

**matchLabels:**

**app: nginx**

**replicas: 4 *# Update the replicas from 2 to 4***

**template:**

**metadata:**

**labels:**

**app: nginx**

**spec:**

**containers:**

**- name: nginx**

**image: nginx:1.8**

**ports:**

**- containerPort: 80**

1. Apply the new YAML file:

**kubectl apply -f nginx-deploy.yaml**

1. Verify that the Deployment has four pods:

**kubectl get pods -l app=nginx**

The output is similar to this:

**NAME READY STATUS RESTARTS AGE**

**nginx-deployment-148880595-4zdqq 1/1 Running 0 25s**

**nginx-deployment-148880595-6zgi1 1/1 Running 0 25s**

**nginx-deployment-148880595-fxcez 1/1 Running 0 2m**

**nginx-deployment-148880595-rwovn 1/1 Running 0 2m**

## Creating a Service for an application

1.List the replica set for the two Hello World pods:

**kubectl get replicasets --selector="app=nginx"**

The output is similar to this:

**NAME DESIRED CURRENT AGE**

**hello-world-2189936611 2 2 12m**

2. Create a Service object that exposes the replica set:

**kubectl expose rs <your-replica-set-name> --type="LoadBalancer" --name="example-service"**

where **<your-replica-set-name>** is the name of your replica set.

1. Display the IP addresses for your service:

**kubectl get services example-service**

The output shows the internal IP address and the external IP address of your service. If the external IP address shows as **<pending>**, repeat the command.

## This will create a Load balancer automatically , you can view the Load balancer in AWS console -> EC2 -> Load Balancer Click on the LB name and get description.

## Create a Two tier Application using Kubernetes Deployment and Services

### **Creating the backend using a Deployment**

The backend is a simple hello greeter microservice. Here is the configuration file for the backend Deployment:

| [**hello.yaml**](https://raw.githubusercontent.com/kubernetes/kubernetes.github.io/master/docs/tasks/access-application-cluster/hello.yaml) |
| --- |
| **apiVersion: apps/v1beta1**  **kind: Deployment**  **metadata:**  **name: hello**  **spec:**  **replicas: 7**  **template:**  **metadata:**  **labels:**  **app: hello**  **tier: backend**  **track: stable**  **spec:**  **containers:**  **- name: hello**  **image: "gcr.io/google-samples/hello-go-gke:1.0"**  **ports:**  **- name: http**  **containerPort: 80** |

Create the backend Deployment:

**kubectl create -f https://k8s.io/docs/tasks/access-application-cluster/hello.yaml**

View information about the backend Deployment:

**kubectl describe deployment hello**

The output is similar to this:

**Name: hello**

**Namespace: default**

**CreationTimestamp: Mon, 24 Oct 2016 14:21:02 -0700**

**Labels: app=hello**

**tier=backend**

**track=stable**

**Annotations: deployment.kubernetes.io/revision=1**

**Selector: app=hello,tier=backend,track=stable**

**Replicas: 7 desired | 7 updated | 7 total | 7 available | 0 unavailable**

**StrategyType: RollingUpdate**

**MinReadySeconds: 0**

**RollingUpdateStrategy: 1 max unavailable, 1 max surge**

**Pod Template:**

**Labels: app=hello**

**tier=backend**

**track=stable**

**Containers:**

**hello:**

**Image: "gcr.io/google-samples/hello-go-gke:1.0"**

**Port: 80/TCP**

**Environment: <none>**

**Mounts: <none>**

**Volumes: <none>**

**Conditions:**

**Type Status Reason**

**---- ------ ------**

**Available True MinimumReplicasAvailable**

**Progressing True NewReplicaSetAvailable**

**OldReplicaSets: <none>**

**NewReplicaSet: hello-3621623197 (7/7 replicas created)**

**Events:**

**...**

### **Creating the backend Service object**

The key to connecting a frontend to a backend is the backend Service. A Service creates a persistent IP address and DNS name entry so that the backend microservice can always be reached. A Service uses selector labels to find the Pods that it routes traffic to.

First, explore the Service configuration file:

| [**hello-service.yaml**](https://raw.githubusercontent.com/kubernetes/kubernetes.github.io/master/docs/tasks/access-application-cluster/hello-service.yaml) |
| --- |
| **kind: Service**  **apiVersion: v1**  **metadata:**  **name: hello**  **spec:**  **selector:**  **app: hello**  **tier: backend**  **ports:**  **- protocol: TCP**  **port: 80**  **targetPort: http** |

In the configuration file, you can see that the Service routes traffic to Pods that have the labels **app: hello** and **tier: backend**.

Create the **hello** Service:

**kubectl create -f https://k8s.io/docs/tasks/access-application-cluster/hello-service.yaml**

At this point, you have a backend Deployment running, and you have a Service that can route traffic to it.

### **Creating the frontend**

Now that you have your backend, you can create a frontend that connects to the backend. The frontend connects to the backend worker Pods by using the DNS name given to the backend Service. The DNS name is “hello”, which is the value of the **name** field in the preceding Service configuration file.

The Pods in the frontend Deployment run an nginx image that is configured to find the hello backend Service. Here is the nginx configuration file:

| [**frontend/frontend.conf**](https://raw.githubusercontent.com/kubernetes/kubernetes.github.io/master/docs/tasks/access-application-cluster/frontend/frontend.conf) |
| --- |
| **upstream hello {**  **server hello;**  **}**  **server {**  **listen 80;**  **location / {**  **proxy\_pass http://hello;**  **}**  **}** |

Similar to the backend, the frontend has a Deployment and a Service. The configuration for the Service has **type: LoadBalancer**, which means that the Service uses the default load balancer of your cloud provider.

| [**frontend.yaml**](https://raw.githubusercontent.com/kubernetes/kubernetes.github.io/master/docs/tasks/access-application-cluster/frontend.yaml) |
| --- |
| **kind: Service**  **apiVersion: v1**  **metadata:**  **name: frontend**  **spec:**  **selector:**  **app: hello**  **tier: frontend**  **ports:**  **- protocol: "TCP"**  **port: 80**  **targetPort: 80**  **type: LoadBalancer**  **---**  **apiVersion: apps/v1beta1**  **kind: Deployment**  **metadata:**  **name: frontend**  **spec:**  **replicas: 1**  **template:**  **metadata:**  **labels:**  **app: hello**  **tier: frontend**  **track: stable**  **spec:**  **containers:**  **- name: nginx**  **image: "gcr.io/google-samples/hello-frontend:1.0"**  **lifecycle:**  **preStop:**  **exec:**  **command: ["/usr/sbin/nginx","-s","quit"]** |

Create the frontend Deployment and Service:

**kubectl create -f https://k8s.io/docs/tasks/access-application-cluster/frontend.yaml**

The output verifies that both resources were created:

**deployment "frontend" created**

**service "frontend" created**

**Note**: The nginx configuration is baked into the [container image](https://kubernetes.io/docs/tasks/access-application-cluster/frontend/Dockerfile). A better way to do this would be to use a [ConfigMap](https://kubernetes.io/docs/tasks/configure-pod-container/configmap/), so that you can change the configuration more easily.

### **Interact with the frontend Service**

Once you’ve created a Service of type LoadBalancer, you can use this command to find the external IP:

**kubectl get service frontend**

The external IP field may take some time to populate. If this is the case, the external IP is listed as **<pending>**.

**NAME CLUSTER-IP EXTERNAL-IP PORT(S) AGE**

**frontend 10.51.252.116 <pending> 80/TCP 10s**

Repeat the same command again until it shows an external IP address:

**NAME CLUSTER-IP EXTERNAL-IP PORT(S) AGE**

**frontend 10.51.252.116 XXX.XXX.XXX.XXX 80/TCP 1m**

### **Send traffic through the frontend**

The frontend and backends are now connected. You can hit the endpoint by using the curl command on the external IP of your frontend Service.

**curl http://<EXTERNAL-IP>**

The output shows the message generated by the backend:

**{"message":"Hello"}**

## Autoscaling in Kubernetes

## Step One: Run & expose php-apache server

To demonstrate Horizontal Pod Autoscaler we will use a custom docker image based on the php-apache image. The Dockerfile can be found [here](https://kubernetes.io/docs/user-guide/horizontal-pod-autoscaling/image/Dockerfile). It defines an [index.php](https://kubernetes.io/docs/user-guide/horizontal-pod-autoscaling/image/index.php) page which performs some CPU intensive computations.

First, we will start a deployment running the image and expose it as a service:

**$ kubectl run php-apache --image=gcr.io/google\_containers/hpa-example --requests=cpu=200m --expose --port=80**

**service "php-apache" created**

**deployment "php-apache" created**

## Step Two: Create Horizontal Pod Autoscaler

Now that the server is running, we will create the autoscaler using [kubectl autoscale](https://github.com/kubernetes/kubernetes/blob/master/docs/user-guide/kubectl/kubectl_autoscale.md). The following command will create a Horizontal Pod Autoscaler that maintains between 1 and 10 replicas of the Pods controlled by the php-apache deployment we created in the first step of these instructions. Roughly speaking, HPA will increase and decrease the number of replicas (via the deployment) to maintain an average CPU utilization across all Pods of 50% (since each pod requests 200 milli-cores by [kubectl run](https://github.com/kubernetes/kubernetes/blob/master/docs/user-guide/kubectl/kubectl_run.md), this means average CPU usage of 100 milli-cores). See [here](https://git.k8s.io/community/contributors/design-proposals/autoscaling/horizontal-pod-autoscaler.md#autoscaling-algorithm) for more details on the algorithm.

**$ kubectl autoscale deployment php-apache --cpu-percent=50 --min=1 --max=10**

**deployment "php-apache" autoscaled**

We may check the current status of autoscaler by running:

**$ kubectl get hpa**

**NAME REFERENCE TARGET MINPODS MAXPODS REPLICAS AGE**

**php-apache Deployment/php-apache/scale 0% / 50% 1 10 1 18s**

Please note that the current CPU consumption is 0% as we are not sending any requests to the server (the **CURRENT** column shows the average across all the pods controlled by the corresponding deployment).

## Step Three: Increase load

Now, we will see how the autoscaler reacts to increased load. We will start a container, and send an infinite loop of queries to the php-apache service (please run it in a different terminal):

**$ kubectl run -i --tty load-generator --image=busybox /bin/sh**

**Hit enter for command prompt**

**$ while true; do wget -q -O- http://php-apache.default.svc.cluster.local; done**

Within a minute or so, we should see the higher CPU load by executing:

**$ kubectl get hpa**

**NAME REFERENCE TARGET CURRENT MINPODS MAXPODS REPLICAS AGE**

**php-apache Deployment/php-apache/scale 305% / 50% 305% 1 10 1 3m**

Here, CPU consumption has increased to 305% of the request. As a result, the deployment was resized to 7 replicas:

**$ kubectl get deployment php-apache**

**NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE**

**php-apache 7 7 7 7 19m**

**Note** Sometimes it may take a few minutes to stabilize the number of replicas. Since the amount of load is not controlled in any way it may happen that the final number of replicas will differ from this example.

## Step Four: Stop load

We will finish our example by stopping the user load.

In the terminal where we created the container with **busybox** image, terminate the load generation by typing **<Ctrl> + C**.

Then we will verify the result state (after a minute or so):

**$ kubectl get hpa**

**NAME REFERENCE TARGET MINPODS MAXPODS REPLICAS AGE**

**php-apache Deployment/php-apache/scale 0% / 50% 1 10 1 11m**

**$ kubectl get deployment php-apache**

**NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE**

**php-apache 1 1 1 1 27m**

Here CPU utilization dropped to 0, and so HPA autoscaled the number of replicas back down to 1.

**Note** autoscaling the replicas may take a few minutes.

## Deleting a deployment

Delete the deployment by name:

**kubectl delete deployment nginx-deployment**

## Delete the Cluster

Running a Kubernetes cluster within AWS obviously costs money, and so you may want to delete your cluster if you are finished running experiments.

You can preview all of the AWS resources that will be destroyed when the cluster is deleted by issuing the following command.

kops delete cluster --name ${NAME}

When you are sure you want to delete your cluster, issue the delete command with the --yes flag. Note that this command is very destructive, and will delete your cluster and everything contained within it!

kops delete cluster --name ${NAME} --yes