

Inlämningsuppgift 2 - Houssam Boughdadi

Förutsättningar

Applikation

Jag kommer använda en simplifierad variant av applikationen som vi jobbat tillsammans med på Kodfredagars lektioner (Med InMemoryMongoDB). Den ligger i ett privat repo på mitt github men skriptet kan enkelt byta repo med bara 1 variabel ändring i huvudskriptet.

Installerade program och extensions

- [VS code](#)
- [Git bash](#)
- [.NET SDK](#)
- [Azure CLI](#) (+ Inloggad på Azure via VS code, förekommer inte i denna Artikel)
- [GitHub CLI](#) (+ Inloggad på git bash "gh auth login" och följ instruktionerna)
- [ARM Template Viewer](#) (Extension för VS Code, för att se ARM template produkter)

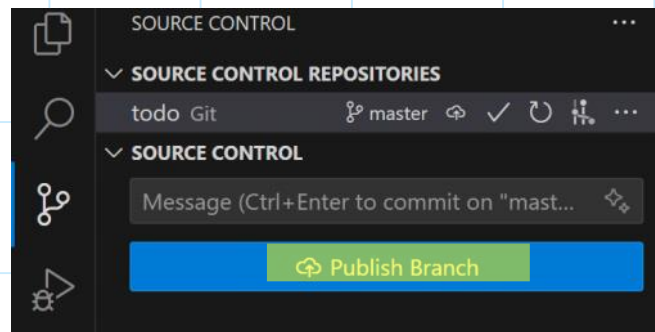
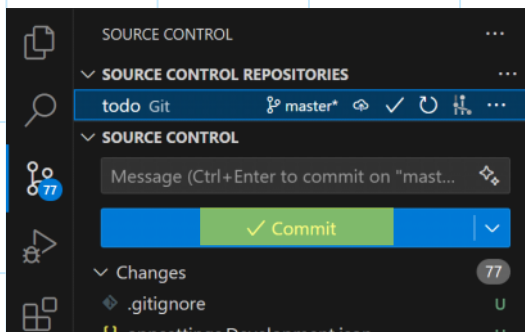
Utveckla applikationen

För att få en applikation till skriptet så behöver den utvecklas. Detta kommer göras i VS code här

1. I en tom mapp (mappens namn bör vara applikationens namn) öppna gitbash terminal

```
dotnet new webapp
dotnet new gitignore
git init
```

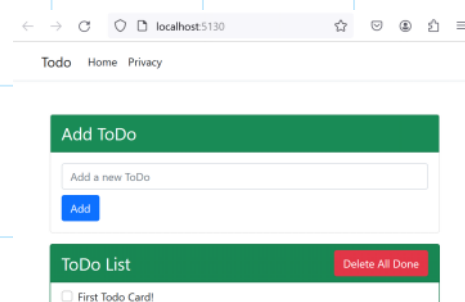
2. Publicera skriptet genom att Commit och Publish till ditt github (privat funkar)



3. Fortsätt utveckla applikationen till nöje, iallafall till första Release. För att kolla applikationen under utvecklingens gång, kör följande kommando.

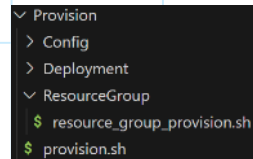
```
dotnet run watch
```

En klar webbapplikation kan se ut såhär



Provisionering

Inom själva provisioneringen så har jag gjort ett shellsript som kallar på andra shellscript med parametrar för att kunna modulärt byta ut olika delar av scriptet som behöver utvecklas eller ändras. Huvudskriptet ligger i "root" av hela provisioneringsmappen och de andra "pusselbitarna" som den kallar på är mer ihopsamlade inom olika mappar. Till exempel:



```
Provision
├── Config
├── Deployment
└── ResourceGroup
    ├── resource_group_provision.sh
    └── provision.sh
```

provision.sh (Huvudskriptet)

Huvudskriptet sätter namnet på repot i github och under vems namn repot ligger på som variabler som kan ändras beroende på vilken applikation man vill ha igång på sitt VM.

Skriptet sen ber ett [annat skript](#) att göra en Resource Group och returnera vilket namn gruppen fått (Finns även implementationen att ta emot Custom namn för Resource Gruppen som går att skicka in som en parameter)

När Resource Group är gjord så körs ett [tredje skript](#) som ska deploya provisionen. Det skriptet tar in argumenten applikationsnamnet och namnet på resource group som blivit hämtad i [tidigare steg](#). Skriptet kommer returnera Publika IP som når appen efter att den blivit deployed.

Huvudskriptet kallar på github cli kommando som bygger artifakterna och när en actions-runner är igång så plockas artifakterna upp till applikationen.

Slutligen skiver skriptet ut i terminalen vilket IP appen nås via (kan ta någon minut för applikationen att dyka upp)

Huvudskriptet finns skriven här:

```
app_name="APPLICATION-NAME"
gh_user="REPOSITORY-OWNER"
resource_group=$(./ResourceGroup/resource_group_provision.sh)
public_ip=$(./Deployment/deployment_provision.sh $resource_group $app_name)
gh workflow run cicd.yaml --repo $gh_user/$app_name --ref master
echo "APP IP: $public_ip"
```

(Klicka på filpatherna för att ta dig till dess kod, funkar för alla länkar i detta PDF)

ResourceGroup/resource_group_provision.sh

Detta skript tar emot ett argument och sätter namnet på Resource gruppen till det eller tar inte emot något argument och därmed generera ett namn för Resource gruppen. Namnet blir en siffra som är lika med antalet existerande Resource grupper + 1 för att inte få överlappande av siffror/namn på Resource grupper om flera grupper existerar med detta system av namngivning.

Efter att den gjort Resource Group så enkapslar det svaret i en response variabel för att förhindra problem av returnering i terminalen vid användning av echo. Sedan returnerar/echo skriptet variabeln för Resource Group namnet.

```
if [ -z "$1" ]; then
    resource_groups=$(az group list --query "[].name" -o tsv)
    new_rg_number=$(( $(echo "$resource_groups" | wc -w) + 1 ))
    resource_group="$new_rg_number"
else
    resource_group="$1"
fi
response=$(az group create --location swedencentral --name $resource_group)
echo $resource_group
```

Deployment/deployment_provision.sh

Detta skript tar emot maximalt 2 argument vid körning. 1a måste vara inkluderat och det är namnet på RG som blev skapat i [tidigare steget](#). 2a är namnet på applikationrepot och måste inte vara inkluderat men då autodefultar den till "MyApp". Namnet var mer relevant vid tidigare implementationer av detta skript och då namnet blev sedan satt utifrån och kopplad till ett github repo så har denna funktionalitet blivit redundant, däremot finns det inte en anledning att ta bort funktionalitet från skriptet om det inte är ivägen.

Nästa steg i skriptet är att samla alla variabler och filvägar som kommer behövas(namn på deployment, filsökvägen till ssh public key etc etc). Filsökvägen för Deployment/parameters.json leder ingenstans däremot så finns det skriven för framtida implementationer

Därefter kallas [ett skript](#) som har uppgiften att skapa CustomData för applikationens VM och den tar emot applikationsnamn och vilken port som applikationen körs på.

Sista stegen använder sig av alla variablerna som blivit samlade för att skapa deployment group med ett ARM template som är länkat och olika parametrar. Sen fångar den upp output till Reverse Proxyns namn från ARM outputs för att hämta dess publika IP och echo:ar resultatet. Anledningen till varför ARM inte outputar IP:n är för att den är inte skapad helt än när outputtar och där för krashar programmet.

```

resource_group=$1
app_name="${2:-MyApp}"
deployment_name="demo"

sshPublicKey=$(cat ~/.ssh/id_rsa.pub)
nginxCustomData=@Deployment/cloud_init_nginx.yaml
appCustomData=@Deployment/cloud_init_app.yaml
template_file="Deployment/BH_RP_APP_Template.json"

$(./Deployment/cloud_init_generator.sh $app_name 5000)

response=$(az deployment group create --resource-group $resource_group --name
$deployment_name --template-file $template_file --parameters nginxCustomData=
$nginxCustomData appCustomData=$appCustomData sshPublicKey="$sshPublicKey")

vm_name=$(az deployment group show --resource-group $resource_group --name
$deployment_name --query properties.outputs.rPName.value --output tsv)

echo $(az vm show --resource-group $resource_group --name $vm_name --show-
details --query [publicIps] --output tsv)

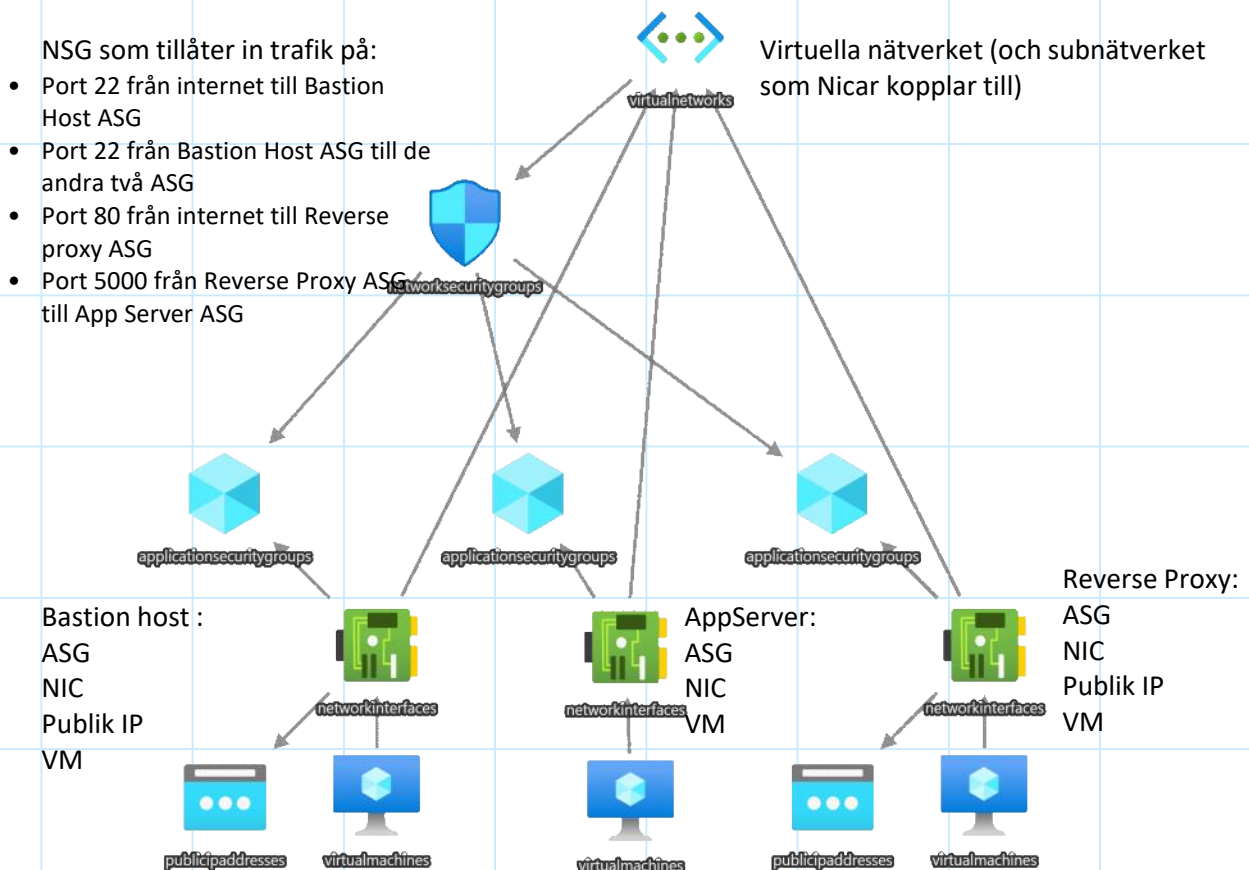
```

ARM

Inom ARM templaterna så börjar den med 3 parametrar den tar emot, min publika ssh nyckel och 2 stycken Custom Data skript, 1 för appservern och 1 för nginx installation och konfigurering för reverse proxy.

Efteråt så sätter den namn på ca 15 variabler för olika namn till komponenter.

Sen så beskrivs alla resurser som skall implementeras och de finns här i en karta.



Sist i ARM templatens så skrivs 2 outputs ut, Namn på Bastion Host VM och Reverse Proxy VM. Detta är för att deras publika IP skall kunnas efterbegäras.

Dependencies

Distributionen av resurser är beroende av varandra i en specifik ordning:

1. Publika IP-adresser är oberoende.
2. ASG är beroende av ingenting.
3. NSG beror på ASG:er.
4. Virtuella nätverket och subnet beror på NSG.
5. NIC:er är beroende av offentliga IP-adresser, virtuellt nätverk och ASG:er.
6. VM:er är beroende av NIC:er.

Connections

- Varje VM kopplar sig till ett NIC.
- Bastion Host och Reverse Proxy VM:ars NIC:ar kopplar sig till 2 publika adresser. Bastion Hosten för att kunna nå VM:erna vid configuration och Reverse Proxy för att nå applikationen.
- Alla NIC:ar kopplar sig till varsitt ASG och till ett enda subnätverk (som finns inom ett VNET)
- NSG sätter upp säkerhetsregler och använder sig av ASG när den beskriver trafikreglerna.
- Virtuella Nätverket får ett subnätverk som kopplar sig till NSG.

VM - AppServer

Appservern VM kommer hosta själva applikationsfilerna och starta den som en service fil vid provisioneringen av VM:et. Däremot kommer det inte finnas en direkt koppling till servern från internet. All trafik till applikationen kommer ske via Reverse Proxy och all trafik till själva servern för konfigurering kommer ske via Bastion Host. Vid provisioneringen kommer den även installera filerna för en self-hosted actions-runner som lyssnar efter uppdateringar i ett github repo som har artefakterna för applikationen som skall köras.

VM - Reverse Proxy

Reverse Proxyns jobb i detta deployment är att styra trafiken från internet på port 80 till appservern på port 5000. Ett skript som laddar ner nginx tjänsten vid provisioneringen finns länkad [här \(Deployment/cloud_init_nginx.yaml\)](#)

VM - Bastion Host

Bastion Hosten fungerar som en säkerhetslager för de andra två VM. Då de inte skall kunnas nå via port 22 av någon främmande ute från internet, så har BastionHost ett eget IP och sina egna säkerhetsregler. En "Tunnel" för trafik sätts upp vid kommunikation med de andra VM på port 22. Då det är en process att kommunicera med appservern och reverse proxy servern vid konfigurering och ändringar så har detta automatiserats i ett [skript \(Config/send_command_bastion.sh\)](#)

Deployment/cloud_init_nginx.yaml

Reverse Proxyns Custom Data. Den laddar ner nginx för trafikstyrning och skriver över default filen och sätter tjänsten att lyssna på port 80 och skicka vidare trafik till samma privata IP som Appservern har samt port 5000. Sedan startar den om nginx för att starta tjänsten med de rätta värden.

Deployment/cloud_init_generator.sh

Detta skript är inte vad som kommer användas som CustomData, den kommer däremot generera filen som kommer användas till Custom Data. Då yaml är kodstandarden för detta provisionering av deployment, men yaml kan inte ha med värden som argument. Så har detta problemet löst sig genom att göra en yaml generator script. Ett skript som skapar Custom Data filen i yaml men skriver in variablernas värde och hur skriptet skall se ut i slutändan när provisioneringen sker. Variabler som behövs skickas med är applikationsnamnet och vilken port applikationen kör på (för skapandet av service fil).

Skriptet kallar även på ett [annat skript](#) som returnerar github token för repot för att kunna skriva in en ny token vid körning då en token varar bara max 1 timme.

I yaml filen som kommer genereras så sker följande steg som appservern gör vid provisionering.

- Skapandet av env fil - även om applikationen inte använder sig av .env fil så finns implementationen för framtida användningar.
- Skapandet av Service fil som beskriver till systemet hur applikationen skall köras.
- Ladda ner dotnet för att kunna köra programmet.
- Starta app som service och hålla igång programmet utan att en användare skall behöva vara kopplad till servern
- Ladda ner self-hosted runner, konfigurera (som azureuser istället för root/sudo) runnern med token och sedan starta den som en tjänst. (Github användaren är hårdkodad och detta bör parametriseras till framtida skript.)

Config/send_command_bastion.sh

Detta Skript är för tillfället bara skrivet för att koppla sig till Appservern då inga ändringar har behövts att göra för Reverse Proxy servern.

Skriptet tar emot den publika IP för Bastion Host servern och vilket kommando som skall köras på appservern som argument. Den kommer sedan starta ett program som håller min privata nyckel för authentication i en session och därmed kunna koppla mig vidare mellan VM inom samma terminal. Sen kopplar den till Bastion Host servern med en flagga som tillåter vidare ssh kopplingar och skickar med vilket kommando som skall köras på appservern.

```
ip=$1
command=$2
ssh_key_path=~/.ssh/id_rsa

eval $(ssh-agent)
ssh-add ~/.ssh/id_rsa

ssh -A -t -o StrictHostKeyChecking=no azureuser@$ip \
"ssh -o StrictHostKeyChecking=no azureuser@10.0.0.12 $command"
```

Config/github_token.sh

Detta skript tar emot 2 argument, Repo namn och Repoägare för att kunna få URL till API call om att få token. Svaret på API call kommer vara JSON object och nästa steg av skriptet kommer filtrera ut allt med SED (search and destroy) så att bara själva token blir returnerad

```
response=$(gh api --method POST -H "Accept: application/vnd.github+json" -H "X-GitHub-API-Version: 2022-11-28" repos/$1/$2/actions/runners/registration-token)
echo "$response" | sed -n 's/.*"token": "[^"]*"$/\1/p'
```

Drifta Applikationen

Om allt är fungerande med applikationskoden så bör applikationen köras direkt vid provisionering. (Med någon minut för byggnad och uppladdning av artifakter). Sålänge Reverse proxyn också är uppe och igång så går applikationen att nås via Reverse proxyns publika IP adress på port 80. Driftningen och deployandet av applikationen görs möjligt med 3 steg.

Systemd service file

En tjänstfil som startar applikationen vid provisionering med information om inställningar för hur tjänsten skall startas, stannas och hanteras av systemet. Den beskriver även miljövariabler såsom vilken IP applikationen skall lyssna på men även för path till fil med framtida .env variabler.

CI/CD

I själva repot ligger en yaml fil som beskriver jobb för GitHub att göra vid körning av workflow. Denna yaml filens första jobb ser till att alla nugets som behövs är installerade, sätter OS och dotnet versioner och sedan bygger artifakterna och publicerar de till github. Det andra jobbet laddar ner de byggda artifakterna från tidigare jobb till en igång action runner. Stoppar applikationen på servern, tar bort de gamla applikationsfiler och skickar in de nya applikationsfilerna. I sista steget så sätter jobbet en restart på själva applikationstjänsten för att köra rätt program.

Self-Hosted Actions Runner

Vid provisionering av Appservern så skapas en action-runner. Runnern körs som en tjänst i bakgrunden på appservern som lyssnar efter nya artifakter att deploya. Runnern registreras på github vid konfigurerings och ifall appservern tas bort så behöver användaren att gå in på github och ta bort runnern manuellt.

Verifiering

1. Kör skriptet

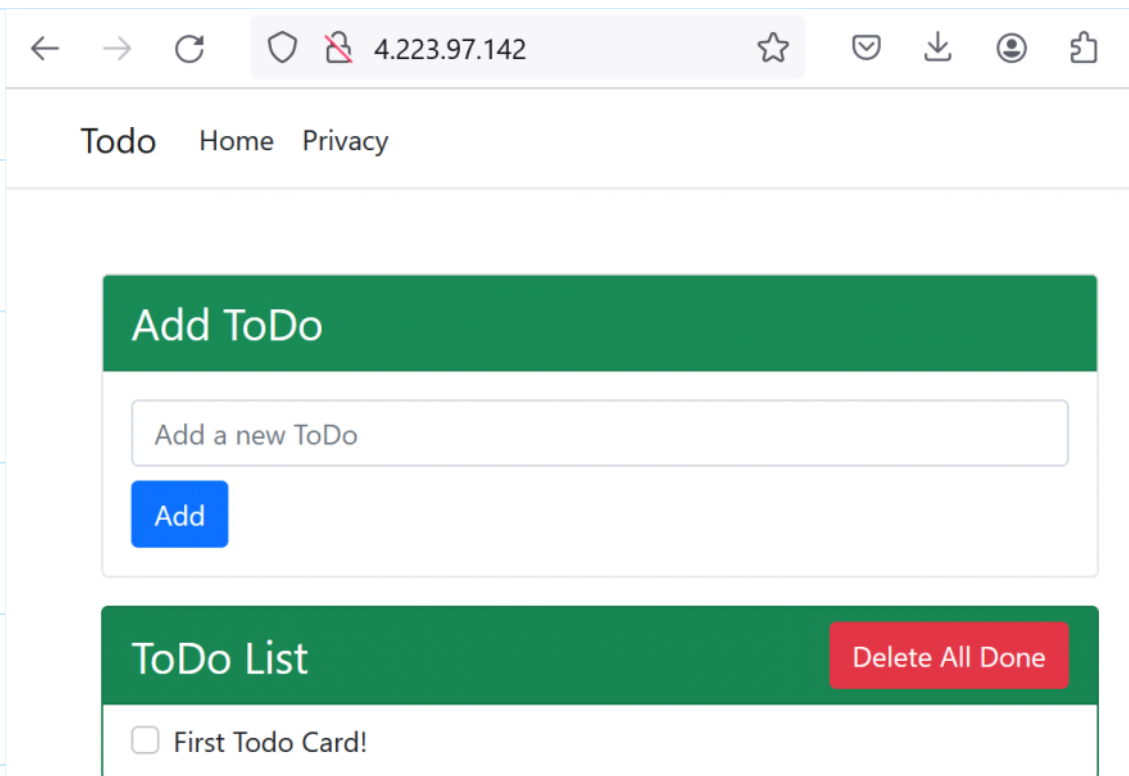
```
$ ./provision.sh
Running ..
```

2. Surfa in på IP adressen som skrevs ut på terminalen. (Kan ta någon minut innan den blir nedladdad från github)

```
$ ./provision.sh
✓ Created workflow_dispatch event for cicd.yaml at master

To see runs for this workflow, try: gh run list --workflow=cicd.yaml
APP IP: 4.223.97.142
```

✓ **Todo**
Todo #15: Manually run by houbou98-19
📅 8 minutes ago ⌚ 1m 19s **master**



Tankar och Framtida Utvecklingsmöjligheter

Dessa skript är fullständiga och fungerande såsom de är just nu. Däremot går det att utöka och komma med fler funktionalitet med mer customization och redundans ifall något misslyckas.

Förbättringar inom skripterna är:

- Failsafe end/remove deployment.
Då många delar av skripten sker imperativt så kan det hända att t ex Resource grupper blir skapade men inga deployment av VM sker ifall något krasha och därför bör skriptet avsluta och ta bort tidigare skapade produkter inom skriptet.
Det borde även finnas ett skript som kan ta bort hela detta specifika deployment även efter att den lyckats att köra igång allt rätt, bara för att snabbt kunna ta ner alla tjänster som är igång och spara på onödiga kostnader som drar igång i bakgrunden.
- Unika selfhosted runner namn samt programatiskt ta bort github runnern som appservern är kopplad till.
Att ha unika self-hosted action-runner namn ger möjligheten att kunna ha 2 eller flera deployment med samma applikatoins repo aktiva samtidigt. Detta kan lösas med datum och tid i själva namnet. Dessutom kan skriptet som tar bort hela deployment även få funktionaliteten att ta bort self-hosted runnern på appservern genom att skicka kommandot via bastionhosten innan den tar ner alla servrar.
- Github Workflow status
Hålla skriptet igång och meddela användaren om när själva deployandet av artefakterna är klart och att applikationen är redo att användas
- Välja server/ip vid Bastion Host tunneling

I dessa skript så har ingen konfigurering behövts göra efter provisioneringen, men ändå så finns det ett skript för att nå appservern och skicka med kommando. Det skriptet kan utökas till att kunna nå alla andra VM inom samma nätverk.

- Parameters.json
ARM templatén tar emot 3 stycken parametrar och dessa skulle kunna finnas i en parameterfil istället för att skickas med som flaggor.

Kod Snuttar

Relevanta kodsnittar som inte kom med i brödtexten.

Hur kommando skickas till appservern via Bastion Hosten. Publika IP skall vara Bastion Hostens IP.

```
#run command through bastionhost
#./Config/send_command_bastion.sh $public_ip "ls"
```

Reverse Proxy Cloud init Yaml fil:

Filen beskrivs [här](#)

```
#cloud-config

package_update: true
packages:
  - nginx

write_files:
  - path: /etc/nginx/sites-available/default
    content: |
      server {
        listen      80 default_server;
        location / {
          proxy_pass      http://10.0.0.12:5000/;
          proxy_http_version 1.1;
          proxy_set_header Upgrade $http_upgrade;
          proxy_set_header Connection keep-alive;
          proxy_set_header Host $host;
          proxy_cache_bypass $http_upgrade;
          proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
          proxy_set_header X-Forwarded-Proto $scheme;
        }
      }
  }
runcmd:
  - |
    systemctl reload nginx
```

Appserverns Cloud init generator:

Beskrivningen av filen görs [tidigare](#) i artikeln

```
#!/bin/bash
# Define variables
app_name="${1:-MyApp}"
app_port="${2:-5000}"
connection_string="empty"
gh_user="${3:-houbou98-19}"
token=$(./Config/github_token.sh $gh_user $app_name)
```

```

# Generate YAML content
app_yaml_content="#cloud-config
write_files:
  - path: /etc/$app_name/$app_name.env
    content: |
      AzureCosmosDBTodoService__ConnectionString=$connection_string
      AzureCosmosDBTodoService__Database="TodoDB"
      AzureCosmosDBTodoService__Collection="TodoList"
  - path: /etc/systemd/system/$app_name.service
    content: |
      [Unit]
      Description=ASP.NET Web App running on Ubuntu
      [Service]
      WorkingDirectory=/opt/$app_name
      ExecStart=/usr/bin/dotnet /opt/$app_name/$app_name.dll
      Restart=always
      RestartSec=10
      KillSignal=SIGINT
      SyslogIdentifier=$app_name
      User=www-data
      Environment=ASPNETCORE_ENVIRONMENT=Production
      Environment=DOTNET_PRINT_TELEMETRY_MESSAGE=false
      Environment="ASPNETCORE_URLS=http://*:$app_port"
      EnvironmentFile=/etc/$app_name/$app_name.env
      [Install]
      WantedBy=multi-user.target
systemd:
  units:
    - name: $app_name.service
      enabled: true

runcmd:
  - |
    declare repo_version=\$(if command -v lsb_release &> /dev/null; then
lsb_release -r -s; else grep -oP '(?<=^VERSION_ID=).+' /etc/os-release | tr -d
'\n'; fi)

    wget https://packages.microsoft.com/config/ubuntu/\\$repo\_version/packages-microsoft-prod.deb -O packages-microsoft-prod.deb
    dpkg -i packages-microsoft-prod.deb
    rm packages-microsoft-prod.deb
    apt update
    apt-get update

    apt-get install -y aspnetcore-runtime-8.0
    mkdir /home/azureuser/actions-runner; cd /home/azureuser/actions-runner
    curl -o actions-runner-linux-x64-2.314.1.tar.gz -L
https://github.com/actions/runner/releases/download/v2.314.1/actions-runner-linux-x64-2.314.1.tar.gz
    tar xzf ./actions-runner-linux-x64-2.314.1.tar.gz
    chown -R azureuser:azureuser /home/azureuser/actions-runner

    sudo -u azureuser ./config.sh --unattended --url https://github.com/
$gh user/$app_name --token $token --name $app_name
    ./svc.sh install azureuser
    ./svc.sh start
    systemctl daemon-reload
    systemctl enable $app_name.service
    systemctl start $app_name.service
"

# Save to a YAML file
echo "$app_yaml_content" > ./Deployment/cloud_init_app.yaml

```