Define k8s services and pods

Overview

1. Create personal git branch
2. Configure etcd entry to store docker proxy settings for k8s
3. Adjust sample apps configuration and add it as etcd secret
4. Define deployments and services
5. Try out services in minikube
6. Deploy to AKS

Prepare the solution

We will create a pod per our docker container.

This will allow us to create replicas per each our our three apps separately.

```
// Set current kubectl config to point to/work with local
minikube cluster
kubectl config set-context minikube

// Create a branch in the afs-demo-infra git repo (replace
$user with your user name)
git checkout -b $user

// Create "provision" directory for the dev environmnet
mkdir dev/provision

// get miniukube IP
minikube ip
```

Configure access to Azure Container registry

We need to configure access to a private container registry from the cluster.

First, login to the registry with your username and password (replace gurba with your name).

```
docker login afsdemogurba.azurecr.io -u afsdemogurba
type %USERPROFILE%\.docker\config.json
```

Output example:

```
{
        "auths": {
                "afsdemogurba.azurecr.io": {}
        },
        "HttpHeaders": {
                "User-Agent": "Docker-Client/18.03.1-ce
(windows)"
        },
        "credsStore": "wincred"
}
```

A Kubernetes cluster uses the Secret of 'docker-registry' type to authenticate with a private container registry.

Create secret named "regcred" in the etcd key vault in your cluster:

```
kubectl create secret docker-registry regcred
--docker-server=afsdemogurba.azurecr.io
--docker-username=afsdemogurba --docker-password=SECRET
--docker-email=aleksei.gurba@gmail.com
```

This secret will be configured in the deployment definition to allow kubectl to pull images from the configured registry.

Note! See https://kubernetes.io/docs/tasks/configure-pod-container/pull-image-private-registry/ for additional details!

Add app config files as secrets

Define our configuration setting files as secrets in etcd.

Copy app_data/ directory from the afs-demo-apps git repo to **dev/provision/config** directory:

```
copy afs-demo-apps\app_data\* dev\provision\config
cd dev\provision\config
```

Your dir should contain following files:

- jwt_1530633205_public.pem
- config.js
- appsettings.json
- app_config.js

Edit dev/provision/config/app_config.js:

```
var config = {};
config.id_url = 'http://10.94.136.113:30088';
config.api_url = 'http://10.94.136.113:30080';
// Note that 10.94.136.113 is the IP of our minikube cluster.
Its value will be changed per each cluster environment. In a
real project, this should be replaced by a host name,
configured in a public DNS
// Ports 30088 and 30080 are going to be exposed by our k8s
services
```

Now create a single secret called **afs-demo-config** from these files:

```
cd dev/provision/config/
kubectl create secret generic afs-demo-config
--from-file=jwt_1530633205_public.pem --from-file=config.js
--from-file=appsettings.json --from-file=app_config.js
```

Note! This secret contains all settings files for our three apps. One secret is created for simplicity in context fo this workshop. Separate secrets should probably be used in production scenarios.

Ensure that secret is properly created:

```
kubectl describe secrets/afs-demo-config

// Optionally, you may try to decode a secret
// This command will output key data in bas64 format. You can
decode it from command line or online at
https://www.base64decode.org/
kubectl get secret mysecret -o yaml
```

It should output something similar to this:

```
C:\Users\gurb011\AppData\Local\Temp>kubectl get secret
afs-demo-config -o yaml
apiVersion: v1
data:
  app_config.js:
dmFyIGNvbmZpZyA9IHt9Ow0KY29uZmlnLmlkX3VybCA9ICdodHRwOi8vaWQ6OD
A4OCc7DQpjb25maWcuYXBpX3VybCA9ICdodHRwOi8vYXBpOjgwODAnOw==
  appsettings.json:
77u/ewogICJKd3RJc3N1ZXIiOiAiZXhhbXBsZS5jb20iLAogICJKd3RBdWRpZW
5jZSI6ICJleGFtcGxlLmNvbSIsCiAgIkp3dEV4cGlyYXRpb25NaW51dGVzIjog
Mjg4MCwKICAiSnd0UmVmcmVzaEV4cGlyYXRpb25NaW51dGVzIjogMTQ0MCwKCi
AgIkp3dEtleXNEaXJlY3RvcnkiOiAiQXBwX0RhdGEvc2VjdXJlIiwKICAiSnd0
V2F0Y2hLZXlDaGFuZ2VzIjogImZhbHNlIiwKICAiSnd0UHVibGljS2V5RmlsZU
5hbWVGb3JtYXQiOiAiand0Xyg/PHNpZ19pZD4uKilfcHVibGljXFwucGVtIiwK
ICAiSnd0UHJpdmF0ZUtleUZpbGVOYW1lRm9ybWF0IjogImp3dF8oPzxzaWdfaW
Q+LiopX3ByaXZhdGVcXC5wZW0iLAoKICAiTW9uZ29Db25uZWN0aW9uU3RyaW5n
IiA6ICJtb25nb2RiOi8vbW9uZ286MjcwMTcvdXNlcnMiCn0=
  config.js:
dmFyIGNvbmZpZyA9IHt9Ow0KY29uZmlnLm1vbmdvX2Nvbm5fc3RyaW5nID0gJ2
1vbmdvZGI6Ly9tb25nbzoyNzAxNy91c2Vycyc7DQptb2R1bGUuZXhwb3J0cyA9
IGNvbmZpZzs=
  jwt_1530633205_public.pem:
LS0tLS1CRUdJTiBQVUJMSUMgS0VZLS0tLS0KTUlJQklqQU5CZ2txaGtpRzl3ME
JBUUVGQUFPQ0FROEFNSUlCQ2dLQ0FRRUFtUWRLUVgrY2J4clIxYlJsRGVFOQpx
L2FiL3BzY285Ky9uRFZxMTVkWGVOUkZPemxUVW9CUUpIb0dkVHFaVDV4N2xpL0
orUjVGazMvMkVOUExCd0VoCjROVTExMFNHNGxzZElvdngxK0N3SzQ3ampVTkQv
a1dsQzFLeWJiZzNkUWVsbnJZY2tRNExZWWhCUVBoaEQ3d3cKZjlvQzJEanl1U2
hoVXdJMWM0c2xvSy9qYUF2SkExVzFlYjZVRUFtRkNxV1hzLzlubVBjcjdYSmp5
Vjd2bGhTSAoyRm9qYmtXNDBtT0c5WUxjL29rY1ppR0pTYTlwcU9rN3FGeFRLT3
kzdm1UeXdvZkRYcHJHcE0vLzJLekJVSjI1Cko2KzFwNFU0aVJuenQ5NXdENmky
QjRDaWhPUzdSQjJFWWVaRW1SZUZPRjBzbGRHcENNwcSt2VmcvR3g5N2VtSzQKM1
FJREFRQUIKLS0tLS1FTkQgUFVCTElDIEtFWS0tLS0tCg==
kind: Secret
```

The afs-demo-config secret will later be mounted as a volume by our pods. The files from this secret will be used as if they are files in a specified pod's directory.

See https://kubernetes.io/docs/concepts/configuration/secret/ for additional details.

Create mongo deployment and service

At first, we will deploy and seed mongo database within our cluster. Next step will be to try out connecting cluster resources to a distributed Cosmos DB account.

We will make a **single replica** of mongo deployment. For the sake of demo, we will also available to the outside.

Create mongodb deployment definition in **dev/provision/mongo-deployment.yml**

```
apiVersion: apps/v1 # for versions before 1.9.0 use
apps/v1beta2
kind: Deployment
metadata:
  name: mongo
  labels:
    app: mongo
spec:
  selector:
    matchLabels:
      app: mongo
      tier: database
  replicas: 1
  template:
    metadata:
      labels:
        app: mongo
        tier: database
    spec:
      imagePullSecrets:
      - name: regcred
      containers:
      - name: app
        image: mongo
        ports:
        - containerPort: 27017
        env:
        - name: GET_HOSTS_FROM
          value: dns
```

A Deployment object defines a Pod creation template and desired replica count.

The Deployment uses a label selector to identify the Pods it manages, and will create or delete Pods as needed to meet the replica count.

Once we have a replicated set of Pods, we need an abstraction that enables connectivity between the layers of the application.
A service provides a way to refer to a set of Pods (selected by labels) with a single static IP address. It may also provide load balancing, if supported by the provider.

Create mongodb service definition in **dev/provision/mongo-service.yml**

```
apiVersion: v1
kind: Service
metadata:
  name: mongo
  labels:
    app: mongo
    tier: database
spec:
  type: NodePort
  ports:
  - port: 27017
    nodePort: 30017
  selector:
    app: mongo
    tier: database
```

Create mongo-seed job

For this demo, we will seed the database manually. Open compass app after service and deployment are started, then connect to your mongo instance. Create 'users' database' and 'users' collection, then import our sample data to it.

> Note! K8s has a concept of just, which allow you to run one-time tasks in the cluster. This would be the perfect type to run seeding or migration jobs later.

Create ID deployment and service

ID service will be accessible externally from the client app.

ID service will have **two replicas**.

Create ID deployment definition in **dev/provision/id-deployment.yml**

```
apiVersion: apps/v1 # for versions before 1.9.0 use
apps/v1beta2
kind: Deployment
metadata:
  name: id
  labels:
    app: id
spec:
  selector:
    matchLabels:
      app: id
      tier: backend
  replicas: 2
  template:
    metadata:
      labels:
        app: id
        tier: backend
    spec:
      imagePullSecrets:
      - name: regcred
      containers:
      - name: id
        image: afsdemogurba.azurecr.io/afs-demo-id:latest
        ports:
        - containerPort: 8088
        env:
        - name: GET_HOSTS_FROM
          value: dns
        volumeMounts: # Using Secret as File from a Pod
        - name: afs-demo-config
          mountPath: "/app/appsettings.json"
          subPath: appsettings.json
          readOnly: true
      volumes:
      - name: afs-demo-config
        secret:
          secretName: afs-demo-config
          items:
          - key: appsettings.json
            path: appsettings.json
```

Create ID service definition in **dev/provision/id-service.yml**

```
apiVersion: v1
kind: Service
metadata:
  name: id
  labels:
    app: id
    tier: backend
spec:
  type: NodePort
  ports:
  - port: 8088
    nodePort: 30088
  selector:
    app: id
    tier: backend
```

Create API deployment and service

API will be accessible externally from the client app.

API will have **two replicas**.

Create API deployment definition in **dev/provision/api-deployment.yml**

```
apiVersion: apps/v1 # for versions before 1.9.0 use
apps/v1beta2
kind: Deployment
metadata:
  name: api
  labels:
    app: api
spec:
  selector:
    matchLabels:
      app: api
      tier: backend
  replicas: 2
  template:
    metadata:
      labels:
        app: api
        tier: backend
    spec:
      imagePullSecrets:
      - name: regcred
      containers:
      - name: api
        image: afsdemogurba.azurecr.io/afs-demo-api:latest
        ports:
        - containerPort: 8080
        env:
        - name: GET_HOSTS_FROM
          value: dns
        volumeMounts: # Using Secret as File from a Pod
        - name: afs-demo-config
          mountPath: "/usr/src/app/config.js"
          subPath: config.js
          readOnly: true
      volumes:
      - name: afs-demo-config
        secret:
          secretName: afs-demo-config
          items:
          - key: config.js
            path: config.js
```

Create API service definition in **dev/provision/api-service.yml**

```
apiVersion: v1
kind: Service
metadata:
  name: api
  labels:
    app: api
    tier: backend
spec:
  type: NodePort
  ports:
  - port: 8080
    nodePort: 30080
  selector:
    app: api
    tier: backend
```

Create App deployment and service

App will have **three replicas**.

Create App deployment definition in **dev/provision/app-deployment.yml**

```
apiVersion: apps/v1 # for versions before 1.9.0 use
apps/v1beta2
kind: Deployment
metadata:
  name: app
  labels:
    app: app
spec:
  selector:
    matchLabels:
      app: app
      tier: frontend
  replicas: 3
  template:
    metadata:
      labels:
        app: app
        tier: frontend
    spec:
      imagePullSecrets:
      - name: regcred
      containers:
      - name: app
        image: afsdemogurba.azurecr.io/afs-demo-app:latest
        ports:
        - containerPort: 80
        env:
        - name: GET_HOSTS_FROM
          value: dns
        volumeMounts: # Using Secret as File from a Pod
        - name: afs-demo-config
          mountPath: "/usr/share/nginx/html/app_config.js"
          subPath: app_config.js
          readOnly: true
      volumes:
      - name: afs-demo-config
        secret:
          secretName: afs-demo-config
          items:
          - key: app_config.js
            path: app_config.js
```

Create Appservice definition in **dev/provision/app-service.yml**

```
apiVersion: v1
kind: Service
metadata:
  name: app
  labels:
    app: app
    tier: frontend
spec:
  type: NodePort
  ports:
  - port: 80
    nodePort: 30089
  selector:
    app: app
    tier: frontend
```

Apply k8s definitions to minikube cluster

Create **dev/provision/build.cmd** file with following contents:

```
SET afs-user=afsdemogurba
SET afs-email=aleksei.gurba@gmail.com
REM set afs-pass=SECRET - provide as environment variable
before running the script. Do not include into source control

echo ON

REM secret
kubectl create secret docker-registry regcred
--docker-server=%afs-user%.azurecr.io
--docker-username=%afs-user% --docker-password=%afs-pass%
--docker-email=%afs-email%

REM config
cd config/
kubectl create secret generic afs-demo-config
--from-file=jwt_1530633205_public.pem --from-file=config.js
--from-file=appsettings.json --from-file=app_config.js
cd ../

REM portainer (to observe and debug deployed containers)
kubectl create -f portainer-deployment.yml
kubectl create -f portainer-service.yml

REM mongo
kubectl create -f mongo-deployment.yml
kubectl create -f mongo-service.yml

REM mongo-seed
REM kubectl create -f mongo-seed-job.yml

REM id service
kubectl create -f id-deployment.yml
kubectl create -f id-service.yml

REM api
kubectl create -f api-deployment.yml
kubectl create -f api-service.yml

REM app
kubectl create -f app-deployment.yml
kubectl create -f app-service.yml

REM get service URLs
minikube service portainer --url
minikube service mongo --url
minikube service id --url
minikube service api --url
minikube service app --url
```

Open terminal and add docker registry secret to the environment variables for the duration of current session.

```
set afs-pass=<YOUR_SECRET_PASS>
```

Note! same operation will be performed by the build server, when environment is created by the CI/CD pipeline.

Execute following commands:

```
./dev/provision/build
```

The command outputs IP addresses of each created service as the last step.

Locate IP address of the App (e.g. http://10.94.136.147:30089) and open it in a browser. Try to play with the services.

Next, open minikube dashboard at http://<minikube_ip>:30000 and observe created artifacts.

Try to scale some of our deployments up or down.

As the final step, commit your changes to git.

Apply k8s definitions to the AKS

Change context to your AKS instance:

```
kubectl config use-context afsdemo-gurba-dev
// Note! replace user name with yours
```

Create new git branch off your existing one:

```
git checkout -b aks
```

Change service type in following files:

- portainer-service.yml
- mongo-service.yml
- id-service.yml
- app-service.yml
- api-service.yml

Set it to this value:

```
spec:
  type: LoadBalancer
```

Note! AKS doesn't support NodeType value, and we must use a LoadBalancer service to acquire a static IP address for our services.

Create the resources in the cluster:

```
cd dev/proivision/
create
```

It will take time, until static IP addresses are created. Use this command to monitor the status:

```
kubectl get rc,service --all-namespaces
```

Wait until all public IPs are assigned.

Connect to the instance of mongo DB using the compass app and import our sample data.

```
// list all pods
kubectl get pods

// locate mongo pod id, the ssh to the container (replace id
from the example)
kubectl exec -it mongo-56978dcb4c-8bdd7 /bin/sh

// run mongoimport
apt update
apt install wget
wget -O init.json
https://raw.githubusercontent.com/alekseigurba/afs-demo-apps/d
ocker/mongo-seed/init.json
mongoimport --host mongo --db users --collection users --type
json --file init.json --jsonArray
```

Then open the app URL in a browser.

Open the dashboard to monitor created artifacts:

```
az aks browse --resource-group afsdemo-gurba-dev --name
afsdemo-gurba-dev
// Note! you can use the dashboard UI to change service type
to LoadBalancer for the kubernetes-dashboard in kube-system
namespace. This way your dashboard will be available from the
public network.
```

Destroy the environment

Here is the additional helper script to destroy all of our artifacts in the cluster.

Create dev/provision/destroy.cmd file:

```
echo ON

REM secrets
kubectl delete secret regcred
kubectl delete secret afs-demo-config

REM portainer (to observe and debug deployed containers)
kubectl delete service portainer
kubectl delete deployment portainer

REM mongo
kubectl delete service mongo
kubectl delete deployment mongo

REM jobs
kubectl delete job mongo-seed

REM id service
kubectl delete service id
kubectl delete deployment id

REM api
kubectl delete service api
kubectl delete deployment api

REM app
kubectl delete service app
kubectl delete deployment app
```

Execute it to delete all previously created resources

Additional links

- Host names - https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/
- 101 - https://kubernetes.io/docs/tutorials/k8s101/
- 201 - https://kubernetes.io/docs/tutorials/k8s201/
- best practices - https://kubernetes.io/docs/concepts/configuration/overview/
- volumes - https://kubernetes.io/docs/tasks/configure-pod-container/configure-volume-storage/
- ingress - https://kubernetes.io/docs/concepts/services-networking/ingress/ (Note! ingress and load balancer are out of scope of this workshop)
- service types - https://kubernetes.io/docs/concepts/services-networking/service/#publishing-services-service-types